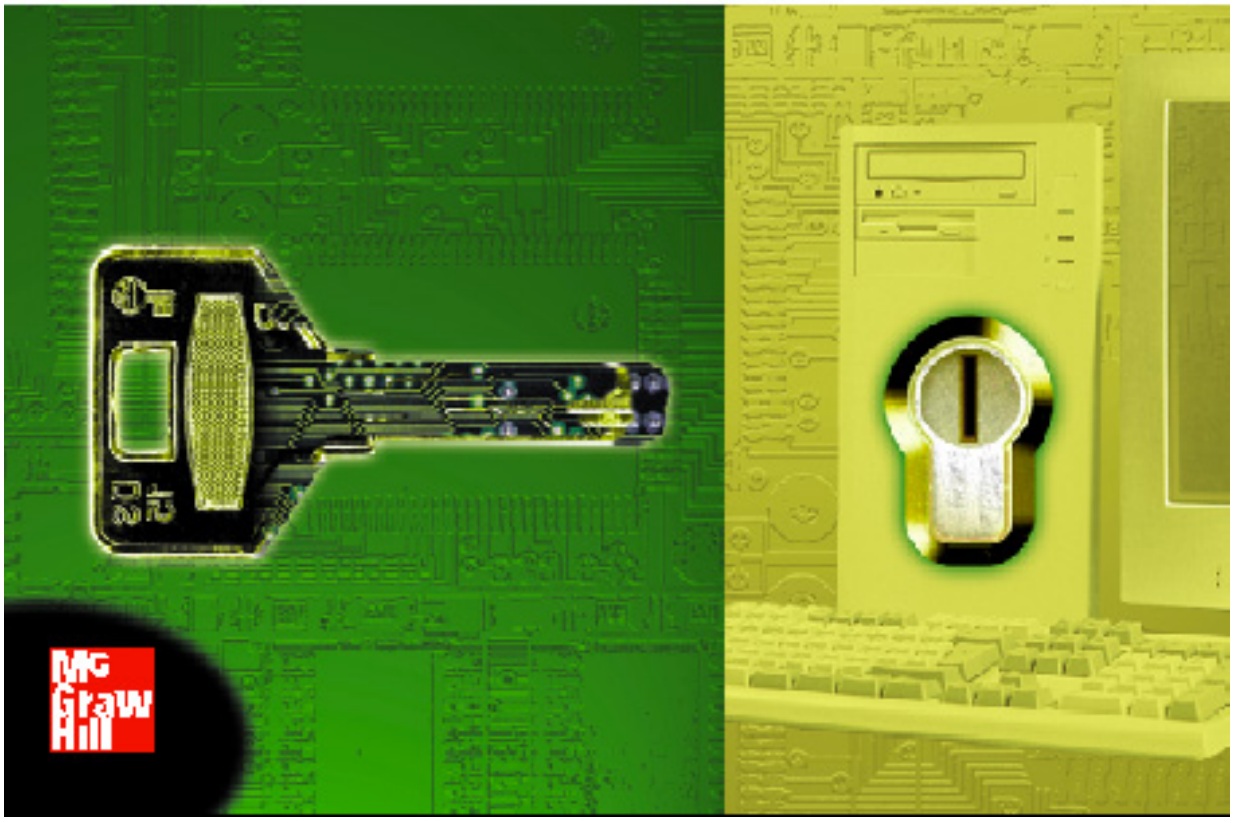
SISTEMAS OPERATIVOS

Una visión aplicada

Jesús CARRETERO PÉREZ
Félix GARCÍA CARBALLEIRA
Pedro DE MIGUEL ANASAGASTI
Fernando PÉREZ COSTOYA



SISTEMAS OPERATIVOS
Una visión aplicada

Digitalización realizada con propósito académico

CONSULTOR EDITORIAL
ÁREA DE INFORMÁTICA Y COMPUTACIÓN

Gerardo Quiroz Vieyra
Ingeniero de Comunicaciones y Electrónica
Por la ESIME del Instituto Politécnico Nacional
Profesor de la Universidad Autónoma Metropolitana
Unidad Xochimilco
MÉXICO

Digitalización realizada con propósito académico

SISTEMAS OPERATIVOS
Una visión aplicada

Jesús Carretero Pérez
Félix García Carballeira
Universidad Carlos II de Madrid

Pedro Miguel Anasagasti
Fernando Pérez Costoya
Universidad Politécnica de Madrid

MADRID * BUENOS AIRES * CARACAS * GUATEMALA * LISBOA * MÉXICO
NUEVA YORK * PANAMÁ * SAN JUAN * SANTAFÉ DE BOGOTÁ * SANTIAGO * SAO PAULO
AUCLAND * HAMBURGO * LONDRES * MILAN * MONTREAL * NUEVA DELHI
PARÍS * SAN FRANCISCO * SIDNEY * SINGAPUR * ST. LUIS * TOKIO * TORONTO

Digitalización realizada con propósito académico

SISTEMAS OPERATIVOS. Una visi{on aplicada

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.

DERECHOS RESERVADOS © 2001, respecto a la primera edición en español, por McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S.A.U.

Edificio Valrealty, 1. planta
Basauri, 17
28023 Aravaca (Madrid)

ISBN: 84-481-3001-4
Depósito legal: M. 13.413-2001

Editora: Concepción Fernández Madrid
Diseño de cubierta: Dima
Preimpresión: MonoComp, S.A.
Impreso en Impresos y revistas, S.A. (IMPRESA)

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Digitalización realizada con propósito académico

Contenido

Prólogo XV

1. CONCEPTOS ARQUITECTÓNICOS DE LA COMPUTADORA 1

1.1. Estructura y funcionamiento de la computadora 2
1.2. Modelo de programación de la computadora 3
1.2.1. Niveles de ejecución 4
1.2.2. Secuencia de funcionamiento de la computadora 5
1.2.3. Registros de control y estado 6
1.3. Interrupciones 7
1.4. El reloj 9
1.5. Jerarquía de memoria 10
1.5.1. Migración de la información 11
1.5.2. Parámetros característicos de la jerarquía de memoria 12
1.5.3. Coherencia 12
1.5.4. Direccionamiento 12
1 .5.5. La proximidad referencial 13
1.6. La memoria virtual 15
1.6.1. Concepto de memoria virtual 16
1.6.2. La tabla de páginas 18
1.6.3. Caso de varios programas activos 22
1.6.4. Asignación de memoria principal y memoria virtual 22
1.7. Entrada/salida 23
1.7.1. Periféricos 23
1.7.2. E/S y concurrencia 25
1.7.3. E/S y memoria virtual 27
1.8. Protección 27
1.8.1. Mecanismos de protección del procesador 27
1.8.2. Mecanismos de protección de memoria 28
1.9. Multiprocesador y multicomputadora 30
1.10. Puntos a recordar 31
1.11. Lecturas recomendadas 31
1.12. Ejercicios 32

2. INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS 33

2.1. ¿Qué es un sistema operativo 34
2.1.1. Máquina desnuda 34
2.1.2. Funciones del sistema operativo 34
2.1.3. Concepto de usuario y de grupo de usuarios 37
2.2. Arranque de la computadora 38
2.3. Componentes y estructura del sistema operativo 41
2.3.1. Componentes del sistema operativo 41
2.3.2. Estructura del sistema operativo 42
2.4. Gestión de procesos 44
2.4.1. Servicios de procesos 45
2.5. Gestión de memoria 46
2.5.1. Servicios 47
2.6. Comunicación y sincronización entre procesos 47
2.6.1. Servicios de comunicación y sincronización 48
2.7. Gestión de la E/S 49
2.7.1. Servicios 50
2.8. Gestión de archivos y directorios 50
2.8.1. Servicio de archivos 50
2.8.2. Servicio de directorios 53
2.8.3. Sistema de archivos 55
2.9. Seguridad y protección 55
2.10. Activación del sistema operativo 56
2.11. Interfaz del programador 59
2.11.1. POSIX 59
2.11.2. Win32 60
2.12. Interfaz de usuario del sistema operativo 61
2.12.1. Funciones de la interfaz de usuario 62
2.12.2. Interfaces alfanuméricas 63
2.12.3. Interfaces gráficas 65
2.13. Historia de los sistemas operativos 67
2.14. Puntos a recordar 72
2.15. Lecturas recomendadas 74
2.16. Ejercicios 74

3. PROCESOS 77

3.1. Concepto de proceso 78
3.2. Multitarea 79
3.2.1. Base de la multitarea 80
3.2.2. Ventajas de la multitarea 82
3.2.3. Grado de multiprogramación y necesidades de memoria principal 82
3.3. Información del proceso 84
3.3.1. Estado del procesador 84
3.3.2. Imagen de memoria del proceso 85
3.3.3. Información del BCP 90
3.3.4. Tablas del sistema operativo 91
3.4. Formación de un proceso 93

Digitalización realizada con propósito académico

3.5. Estados del proceso 93
3.5.1. Cambio de contexto 95
3.6. Procesos ligeros 98
3.6.1. Estados del proceso ligero 99
3.6.2. Paralelismo 100
3.6.3. Diseño con procesos ligeros 101
3.7. Planificación 102
3.7.1. Algoritmos de planificación 105
3.7.2. Planificación en POSIX 107
3.7.3. Planificación en Windows NT/2000 108
3.8. Señales y excepciones 110
3.8.1. Señales 110
3.8.2. Excepciones 111
3.9. Temporizadores 112
3.10. Servidores y demonios 112
3.11. Servicios POSIX 114
3.11.1. Servicios POSIX para la gestión de procesos 114
3.11.2. Servicios POSIX de gestión de procesos ligeros 131
3.11.3. Servicios POSIX para la planificación de procesos 136
3.11.4. Servicios POSIX para gestión de señales y temporizadores 139
3.12. Servicios de W1N32 146
3.12.1. Servicios de Win32 para la gestión de procesos 146
3.12.2. Servicios de Win32 para la gestión de procesos ligeros 152
3.12.3. Servicios de planificación en Win32 154
3.12.4. Servicios de Win32 para el manejo de excepciones 155
3.12.5. Servicios de temporizadores 157
3.13. Puntos a recordar 159
3.14. Lecturas recomendadas 160
3.15. Ejercicios 160

4. GESTIÓN DE MEMORIA 163
4.1. Objetivos del sistema de gestión de memoria 164
4.2. Modelo de memoria de un proceso 172
4.2.1. Fases en la generación de un ejecutable 172
4.2.2. Mapa de memoria de un proceso 178
4.2.3. Operaciones sobre regiones 182
4.3. Esquemas de memoria basados en asignación contigua 183
4.4. Intercambio 186
4.5. Memoria virtual 187
4.5.1. Paginación 188
4.5.2. Segmentación 197
4.5.3. Segmentación paginada 198
4.5.4. Paginación por demanda 199
4.5.5. Políticas de reemplazo 201
4.5.6. Política de asignación de marcos de página 204
4.5.7. Hiperpaginación 205
4.5.8. Gestión del espacio de swap 207
4.5.9. Operaciones sobre las regiones de un proceso 208

Digitalización realizada con propósito académico

4.6. Archivos proyectados en memoria 210
4.7. Servicios de gestión de memoria 212
4.7.1. Servicios genéricos de memoria 212
4.7.2. Servicios de memoria de POSIX 212
4.7.3. Servicios de memoria de Win32 216
4.8. Puntos a recordar 219
4.9. Lecturas recomendadas 220
4.10. Ejercicios 221

5. COMUNICACIÓN Y SINCRONIZACIÓN DE PROCESOS 223
5.1. Procesos concurrentes 224
5.1.1. Tipos de procesos concurrentes 225
5.2. Problemas clásicos de comunicación y sincronización 226
5.2.1. El problema de la sección crítica 226
5.2.2. Problema del productor-consumidor 230
5.2.3. El problema de los lectores-escritores 230
5.2.4. Comunicación cliente-servidor 231
5.3. Mecanismos de comunicación y sincronización 232
5.3.1. Comunicación mediante archivos 232
5.3.2. Tuberías 233
5.3.3. Sincronización mediante señales 237
5.3.4. Semáforos 237
5.3.5. Memoria compartida 242
5.3.6. Mutex y variables condicionales 243
5.4. Paso de mensajes 248
5.5. Aspectos de implementación de los mecanismos de sincronización 253
5.5.1. Implementación de la espera pasiva 254
5.6. Interbloqueos 257
5.7. Servicios POSIX 258
5.7.1. Tuberías 258
5.7.2. Semáforos POSIX 265
5.7.3. Mutex y variables condicionales en POSIX 270
5.7.4. Colas de mensajes POSIX 274
5.8. Servicios Wjn32 285
5.8.1. Tuberías 286
5.8.2. Secciones críticas 294
5.8.3. Semáforos 295
5.8.4. Mutex y eventos 299
5.8.5. Mailslots 303
5.9. Puntos a recordar 305
5.10. Lecturas recomendadas 306
5.11. Ejercicios 306

6. INTERBLOQUEOS 309
6.1. Los interbloqueos: una historia basada en hechos reales 310
6.2. Los interbloqueos en un sistema informático 311
6.2.1. Tipos de recursos 311

Digitalización realizada con propósito académico

6.3. Un modelo del sistema 317
6.3.1. Representación mediante un grafo de asignación de recursos 318
6.3.2. Representación matricial 322
6.4. Definición y caracterización del interbloqueo 324
6.4.1. Condición necesaria y suficiente para el interbloqueo 325
6.5. Tratamiento del interbloqueo 326
6.6. Detección y recuperación del interbloqueo 327
6.6.1. Detección del interbloqueo 328
6.6.2. Recuperación del interbloqueo 334
6.7. Prevención del interbloqueo 334
6.7.1. Exclusión mutua 335
6.7.2. Retención y espera 336
6.7.3. Sin expropiación 336
6.7.4. Espera circular 337
6.8. Predicción del interbloqueo 337
6.8.1. Concepto de estado seguro 338
6.8.2. Algoritmos de predicción 339
6.9. Tratamiento del interbloqueo en los sistemas operativos 345
6.10. Puntos a recordar 347
6.11. Lecturas recomendadas 349
6.12. Ejercicios 349

7. ENTRADA/SALIDA 351
7.1. Introducción 352
7.2. Caracterización de los dispositivos de E/S 354
7.2.1. Conexión de un dispositivo de E/S a una computadora 354
7.2.2. Dispositivos conectados por puertos o proyectados en memoria 355
7.2.3. Dispositivos de bloques y de caracteres 356
7.2.4. E/S programada o por interrupciones 357
7.2.5. Mecanismos de incremento de prestaciones 361
7.3. Arquitectura del sistema de entrada/salida 363
7.3.1. Estructura y componentes del sistema de E/S 363
7.3.2. Software de E/S 364
7.4. Interfaz de aplicaciones 369
7.5. Almacenamiento secundario 373
7.5.1. Discos 374
7.5.2. El manejador de disco 379
7.5.3. Discos en memoria 384
7.5.4. Fiabilidad y tolerancia a fallos 385
7.6. Almacenamiento terciario 387
7.6.1. Tecnología para el almacenamiento terciario 388
7.6.2. Estructura y componentes de un sistema de almacenamiento terciario 389
7.6.3. Estudio de caso: Sistema de almacenamiento de altas prestaciones (HPSS) 391
7.7. El reloj 393
7.7.1. El hardware del reloj 393
7.7.2. El software del reloj 394

Digitalización realizada con propósito académico

7.8. El terminal 397
7.8.1. Modo de operación del terminal 397
7.8.2. El hardware del terminal 398
7.8.3. El software del terminal 400
7.9. La red 404
7.10. Servicios de entrada/salida 405
7.10.1. Servicios genéricos de entrada/salida 405
7.10.2. Servicios de entrada/salida en POSIX 406
7.10.3. Servicios de entrada/salida en Win32 410
7.11. Puntos a recordar 414
7.12. Lecturas recomendadas 416
7.13. Ejercicios 417

8. GESTIÓN DE ARCHIVOS Y DIRECTORIOS 419
8.1. Visión de usuario del sistema de archivos 420
8.2. Archivos 420
8.2.1. Concepto de archivo 421
8.2.2. Nombres de archivos 423
8.2.3. Estructura de un archivo 424
8.2.4. Métodos de acceso 427
8.2.5. Semánticas de coutilización 428
8.3. Directorios 429
8.3.1. Concepto de directorio 429
8.3.2. Estructuras de directorio 432
8.3.3. Nombres jerárquicos 435
8.3.4. Construcción de la jerarquía de directorios 437
8.4. Servicios de archivos y directorios 438
8.4.1. Servicios genéricos para archivos 439
8.4.2. Servicios POSIX para archivos 440
8.4.3. Ejemplo de uso de servicios POSIX para archivos 443
8.4.4. Servicios Win32 para archivos 445
8.4.5. Ejemplo de uso de servicios Win32 para archivos 449
8.4.6. Servicios genéricos de directorios 451
8.4.7. Servicios POSIX de directorios 451
8.4.8. Ejemplo de uso de servicios POSIX para directorios 454
8.4.9. Servicios Win32 para directorios 456
8.4.10. Ejemplo de uso de servicios Win32 para directorios 458
8.5. Sistemas de archivos 459
8.5.1. Estructura del sistema de archivos 461
8.5.2. Otros tipos de sistemas de archivos 465
8.6. El servidor de archivos 468
8.6.1. Estructura del servidor de archivos 469
8.6.2. Estructuras de datos asociadas con la gestión de archivos 472
8.6.3. Mecanismos de asignación y correspondencia de bloques a archivos 474
8.6.4. Mecanismos de gestión de espacio libre 477
8.6.5. Mecanismos de incremento de prestaciones 479
8.6.6. Montado de sistemas de archivos e interpretación de nombres 483

Digitalización realizada con propósito académico

8.6.7. Fiabilidad y recuperación 485
8.6.8. Otros servicios 489
8.7. Puntos a recordar 491
8.8. Lecturas recomendadas 493
8.9. Ejercicios 493

9. SEGURIDAD Y PROTECCIÓN 497
9.1. Conceptos de seguridad y protección 498
9.2. Problemas de seguridad 499
9.2.1. Uso indebido o malicioso de programas 500
9.2.2. Usuarios inexpertos o descuidados 501
9.2.3. Usuarios no autorizados 501
9.2.4. Virus 502
9.2.5. Gusanos 503
9.2.6. Rompedores de sistemas de protección 504
9.2.7. Bombardeo 504
9.3. Políticas de seguridad 505
9.3.1. Política militar 505
9.3.2. Políticas comerciales 507
9.3.3. Modelos de seguridad 508
9.4. Diseño de sistemas operativos seguros 509
9.4.1. Principios de diseño y aspectos de seguridad 509
9.4.2. Técnicas de diseño de sistemas seguros 512
9.4.3. Controles de seguridad externos al sistema operativo 515
9.4.4. Controles de seguridad del sistema operativo 518
9.5. Criptografía 519
9.5.1. Conceptos básicos 519
9.5.2. Sistemas de clave privada y sistemas de clave pública 522
9.6. Clasificaciones de seguridad 524
9.6.1. Clasificación del Departamento de Defensa (D0D) de Estados Unidos 524
9.7. Seguridad y protección en sistemas operativos de propósito general 526
9.7.1. Autenticación de usuarios 526
9.7.2. Palabras clave o contraseñas 528
9.7.3. Dominios de protección 531
9.7.4. Matrices de protección 534
9.7.5. Listas de control de accesos 535
9.7.6. Capacidades 538
9.8. Servicios de protección y seguridad 540
9.8.1. Servicios genéricos 540
9.8.2. Servicios POSIX 541
9.8.3. Ejemplo de uso de los servicios de protección de POSIX 543
9.8.4. Servicios de Win32 545
9.8.5. Ejemplo de uso de los servicios de protección de Win32 548
9.9. El sistema de seguridad de Windows NT 550
9.10. Kerberos 552
9.11. Puntos a recordar 556

Digitalización realizada con propósito académico

9.12. Lecturas recomendadas 557
9.13. Ejercicios 557

10. INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS 561
10.1. Sistemas distribuidos 562
10.1.1. Características de un sistema distribuido 562
10.1.2. Redes e interconexión 563
10.1.3. Protocolos de comunicación 564
10.2. Sistemas operativos distribuidos 566
10.3. Comunicación de procesos en sistemas distribuidos 570
10.3.1. Sockets 570
10.3.2. Llamadas a procedimientos remotos 582
10.3.3. Comunicación de grupos 592
10.4. Sincronización de procesos en sistemas distribuidos 593
10.4.1. Ordenación de eventos en sistemas distribuidos 593
10.4.2. Exclusión mutua en sistemas distribuidos 596
10.5. Gestión de procesos 598
10.5.1. Asignación de procesos a procesadores 598
10.5.2. Algoritmos de distribución de la carga 599
10.5.3. Planificación de procesos en sistemas distribuidos 601
10.6. Sistemas de archivos distribuidos 601
10.6.1. Nombrado 602
10.6.2. Métodos de acceso remotos 603
10.6.3. Utilización de cache en sistemas de archivos distribuidos 604
10.7. Gestión de memoria en sistemas distribuidos 606
10.8. Puntos a recordar 607
10.9. Lecturas recomendadas 609
10.10. Ejercicios 609

11. ESTUDIO DE CASOS: LINUX 611
11.1. Historia de LINUX 612
11.2. Características y estructura de LINUX 613
11.3. Gestión de procesos 614
11.4. Gestión de memoria 615
11.5. Entrada/salida 616
11.6. Sistema de archivos 616
11.7. Puntos a recordar 617
11.8. Lecturas recomendadas 617

12. ESTUDIO DE CASOS: WINDOWS NT 619
12.1. Introducción 620
12.2. Principios de diseño de Windows NT 620
12.3. Arquitectura de Windows NT 621

Digitalización realizada con propósito académico

12.4. El núcleo de Windows NT 623
12.5. El ejecutivo de Windows NT 624
12.5.1. Gestor de objetos 624
12.5.2. Gestor de procesos 625
12.5.3. Gestor de memoria virtual 627
12.5.4. Llamada a procedimiento local 630
12.5.5. Gestor de entrada/salida 631
12.6. Subsistemas de entorno de ejecución 635
12.7. Sistemas de archivos de Windows NT 636
12.7.1. Sistemas de archivos tipo FAT 637
12.7.2. Sistemas de archivos de alto rendimiento (HPFS) 638
12.7.3. NTFS 639
12.7.4. Comparación de los sistemas de archivos PAT, HPFS y NTFS 642
12.8. El subsistema de seguridad 642
12.8.1. Autenticación de usuarios 643
12.8.2. Listas de control de acceso en Windows NT 645
12.9. Mecanismos para tolerancia a fallos en Windows NT 646
12.10. Puntos a recordar 648
12.11. Lecturas recomendadas 649

A. Comparación de los servicios POSIX y Win32 651

B. Entorno de programación de sistemas operativos 657

C. Trabajos prácticos de sistemas operativos 669

Bibliografía 709

Índice 721

Digitalización realizada con propósito académico

o

Digitalización realizada con propósito académico

Prólogo

Los sistemas operativos son una parte esencial de cualquier sistema de computación, por lo que todos los planes de estudio de informática incluyen uno o más cursos sobre sistemas operativos. La mayoría de libros de sistemas operativos usados en estos cursos incluyen gran cantidad de teoría general y aspectos de diseño, pero no muestran claramente cómo se usan.

Este libro está pensado como un texto general de sistemas operativos, pudiendo cubrir tanto la parte introductoria como los aspectos de diseño de los mismos. En él se tratan todos los aspectos fundamentales de los sistemas operativos, tales como procesos, gestión de memoria, comunicación y sincronización de procesos, entrada/salida, sistemas de archivos y seguridad y protección. Además, en cada tema, se muestra la interfaz de programación de los sistemas operativos POSIX y Win32, con ejemplos de uso de las mismas. Esta solución permite que el lector no sólo conozca los principios teóricos, sino cómo se aplican en sistemas operativos reales.

CONTEXTO DE DESARROLLO DEL LIBRO

A principios de los noventa, los profesores del Departamento de Arquitectura y Tecnología de la Facultad de Informática de la Universidad de Politécnica de Madrid comenzaron a elaborar apuntes que incluían teoría y problemas de Sistemas Operativos, con vistas a desarrollar un nuevo plan de estudios de informática. Se revisaron cuidadosamente los planes de estudio existentes en dicha universidad, así como los de varias otras escuelas similares. Además, se llevó a cabo una revisión exhaustiva de bibliografía relacionada con los sistemas operativos.

La motivación para llevar a cabo este trabajo surgió de la insatisfacción con los libros de texto existentes en su momento, que, en líneas generales, se caracterizaban por enfatizar en los siguientes aspectos:

- Teoría general sobre sistemas operativos.
- Aspectos de diseño detallado, generalmente específicos de un sistema operativo.
- Desarrollo en un ambiente de sistemas operativos clásicos.

Comparando esta situación con la del mundo real se observaban considerables diferencias:

- Demanda de los estudiantes para tener apoyo en las cuestiones teóricas con ejemplos prácticos.
- Necesidad de conocer los sistemas operativos desde el punto de vista de programación de sistemas.
- Visión generalista del diseño de los sistemas operativos, estudiando distintos sistemas.

Digitalización realizada con propósito académico

Esta situación obligaba a los autores a mezclar textos generales sobre sistemas operativos con otros libros que estudiaban sistemas operativos concretos y la forma de programarlos. Por esta razón, entre- otras, el cuerpo de los apuntes, mencionado anteriormente, fue creciendo y modernizándose hasta llegar a este libro.

ORGANIZACIÓN DEL LIBRO

El libro está organizado en doce temas, cuyo índice se muestra a continuación. Su contenido cubre todos los aspectos de gestión de una computadora, desde la plataforma hardware hasta los sistemas distribuidos. Además, se incluyen tres apéndices.

Los temas son los siguientes:

Conceptos arquitectónicos de la computadora

En este tema se hace una somera descripción de la estructura y funcionamiento de una computadora, desde el punto de vista de la plataforma hardware. La motivación para incluir este capítulo es evitar la necesidad de que el lector posea conocimientos previos de estructura de computadoras. En él se tratan aspectos tales como el modelo de programación de la computadora, tratamiento de interrupciones, jerarquía de memoria, entrada/salida y concurrencia. Además, se comentan breve mente los mecanismos de protección hardware.

Introducción a los sistemas operativos

En este tema se explica qué es un sistema operativo, cuáles son sus funciones principales, los tipos de sistemas operativos existentes actualmente y cómo se activa un sistema operativo. También se introduce brevemente la estructura del sistema operativo y de sus componentes principales (procesos, memoria, archivos, comunicación, etc.), que se describen en detalle en capítulos posteriores. Además, se ponen dos ejemplos concretos, como son LINUX y Windows NT. Para terminar, se muestra la interfaz de usuario y de programador del sistema operativo.

Procesos

El proceso es la entidad más importante de un sistema operativo moderno. En este tema se estudia en detalle el concepto de proceso, la información asociada al mismo, sus posibles estados y las señales y temporizadores que pueden ser asociadas a un proceso. Un sistema operativo gestiona una colección de procesos que se ejecutan de forma concurrente. La planificación de dichos procesos es crucial para la gestión de una computadora. Es esencial explotar los recursos de forma eficiente, equitativa y evitar bloqueos entre procesos. Además, se estudia en este capítulo el concepto de proceso ligero (thread) y su influencia sobre los aspectos anteriores del sistema. Todo ello se complementa con ejemplos de uso en POSIX y Windows NT

Gestión de memoria

Un proceso en ejecución reside siempre en la memoria de la computadora. Por tanto, gestionar dicha memoria de forma eficiente es un aspecto fundamental de cualquier sistema operativo. En este tema se estudian los requisitos de la gestión de memoria, el modelo de memoria de un proceso,

Digitalización realizada con propósito académico

cómo se genera dicho modelo y diversos esquemas de gestión de memoria, incluyendo la memoria virtual. Este tema está relacionado con el Capítulo 7, debido a que la gestión de la memoria virtual se apoya en los discos como medio auxiliar de almacenamiento de la imagen de los procesos que no cabe en memoria principal. Al final del tema se muestran los servicios de gestión de memoria existentes en POSIX y Win32 y algunos ejemplos de uso de los mismos.

Comunicación y sincronización de procesos

Los procesos no son entidades aisladas, sino que en muchos casos cooperan entre sí y compiten por los recursos. El sistema operativo debe ofrecer mecanismos de comunicación y sincronización de procesos concurrentes. En este tema se muestran los principales mecanismos usados en sistemas operativos, tales como tuberías, semáforos o el paso de mensajes, así como algunos aspectos de implementación de los mismos. Al final del tema se muestran los servicios de comunicación y sincronización existentes en POSIX y Win32 y algunos ejemplos de uso de los mismos.

Interbloqueos

Las comunicaciones, el uso de recursos compartidos y las sincronizaciones son causas de bloqueos mutuos entre procesos, o interbloqueos. En este capítulo se presenta el concepto de interbloqueo, así como los principales métodos de modelado de interbloqueos. Además, se describen los principales algoritmos existentes para gestión de interbloqueos, incluyendo los de prevención, detección y predicción de interbloqueos.

Entrada/salida

El procesador de una computadora necesita relacionarse con el mundo exterior. Esta relación se lleva a cabo mediante los dispositivos de entrada/salida (E/S) conectados a la computadora. El sistema operativo debe ofrecer una interfaz de acceso a dichos dispositivos y gestionar los detalles de bajo nivel de los mismos. En este tema se muestran aspectos del hardware y el software de E/S, estudiando una amplia gama de dispositivos, tales como los de almacenamiento secundario y terciario, los relojes o el terminal. Al final del tema se muestran los servicios de entrada/salida existentes en POSIX y Win32 y algunos ejemplos de uso de los mismos.

Gestión de archivos y directorios

El sistema operativo debe proporcionar al usuario mecanismos de alto nivel para acceder a la información existente en los dispositivos de almacenamiento. Para ello, todos los sistemas operativos incluyen un sistema de gestión de archivos y directorios. El archivo es la unidad fundamental de almacenamiento que maneja el usuario. El directorio es la unidad de estructuración del conjunto de archivos. En este tema se muestran los conceptos fundamentales de archivos y directorios, la estructura de sus gestores y los algoritmos internos usados en los mismos. Al igual que en otros temas, se muestran los servicios de archivos y directorios existentes en POSIX y Win32 y algunos ejemplos de uso de los mismos.

Seguridad y protección

Un sistema de computación debe ser seguro. El usuario debe tener la confianza de que las acciones internas o externas del sistema no van a ser un peligro para sus datos, aplicaciones o para las actividades de otros usuarios. El sistema operativo debe proporcionar mecanismos de protección

Digitalización realizada con propósito académico

entre los distintos procesos que ejecutan en un sistema y entre los distintos sistemas que estén conectados entre sí. En este tema se exponen los conceptos de seguridad y protección, posibles problemas de seguridad, mecanismos de diseño de sistemas seguros, los niveles de seguridad que puede ofrecer un sistema y los controles existentes para verificar si el estado del sistema es seguro. Además, se estudian los mecanismos de protección que se pueden usar para controlar el acceso a los distintos recursos del sistema, Al final del tema se muestran los servicios de protección existen en POSIX y Win32 y algunos ejemplos de uso de los mismos.

Introducción a los sistemas distribuidos

Los sistemas de computación actuales raramente están aislados. Es habitual que estén conectados formando conjuntos de máquinas que no comparten la memoria ni el reloj, es decir, sistemas distribuidos. Este tema presenta una breve introducción a dichos sistemas, estudiando las características de los sistemas distribuidos, sus problemas de diseño, su estructura y sus distintos elementos (redes, comunicación, memoria distribuida, sistemas de archivo distribuido, etc.). También se muestran distintas técnicas de diseño de aplicaciones cliente-servidor en sistemas distribuidos.

Estudio de casos: LINUX

Este capítulo muestra en detalle los aspectos de LINUX desarrollados a lo largo del libro. Para ello se describe, tema por tema, cómo es la arquitectura del sistema operativo LINUX, cómo son los procesos de LINUX, sus mecanismos de comunicación y seguridad, etc.

Estudio de casos: Windows NT

Este capítulo muestra en detalle los aspectos de Windows NT desarrollados a lo largo del libro. Para ello se describe, tema por tema, cómo es la arquitectura del sistema operativo Windows NT, cómo son los procesos de Windows NT, su sistema de E/S, sus mecanismos de comunicación y seguridad, etcétera.

Apéndice A. Comparación de los servicios POSIX y Win32

Tabla de llamadas al sistema de POSIX y Win32. Para cada función del sistema se muestra la llamada POSIX y la de Win32 que lleva a cabo dicha función, junto a un breve comentario de la misma.

Apéndice B. Entorno de programación de sistemas operativos

En este apéndice se describe cómo editar, compilar y ejecutar un programa C en UNIX/LINUX y Windows NT. Para el caso de LINUX se usa el compilador gcc. Para el caso de Windows NT, el Visual C++.

Apéndice C. Trabajos prácticos de sistemas operativos

En este apéndice se describen varios proyectos de prácticas de sistemas operativos desarrollados por los autores durante varios años. Todos ellos se han llevado a efecto, por lo que se incluyen también comentarios acerca de la realización de los mismos por los alumnos.

Digitalización realizada con propósito académico

Materiales suplementarios

Existe una página Web con materiales suplementarios para el libro, situada en la dirección:
http: //arcos.inf.uc3m.es/' La misma información se encuentra duplicada en:
http://datsi.fi. upm. es/ ssoo-va.

En esta página Web se puede encontrar el siguiente material:

- **Información sobre el libro**, como el prólogo, tabla de contenidos, capítulos de ejemplo en PDF, erratas, etc.
- **Información de los autores** y dirección de contacto.
- **Material para el profesor**, como figuras del libro, transparencias, soluciones de ejercicios y problemas propuestos y material de prácticas. Las prácticas que se presentan han sido diseñadas como trabajos de laboratorio para estudiantes de las asignaturas de Sistemas Operativos de la Universidad Politécnica de Madrid y de la Universidad Carlos III de Madrid. Se ha hecho un importante esfuerzo para generalizar sus enunciados, de forma que puedan desarrollarse fácilmente sobre sistemas operativos de amplia difusión como Linux, UNIX o Windows. En casi todos los trabajos prácticos expuestos se hace referencia al material de apoyo existente para las prácticas, que también se puede conseguir en las páginas Web anteriores.
- **Material para el estudiante**, como código fuente de los programas, figuras en PowerPoint, problemas propuestos de sistemas operativos, etc.

Comentario de los autores

Es un placer para nosotros poder presentar este texto a las personas interesadas en los sistemas operativos, su diseño y su programación. La elaboración de este texto ha supuesto un arduo trabajo para nosotros, tanto por la extensión de la obra como por los ejemplos prácticos incluidos en la misma. Además, se ha hecho un esfuerzo importante para tratar de unificar la terminología usada en distintos países de habla hispana. Con todo, creemos que el resultado final hace que el esfuerzo realizado haya merecido la pena.

El esfuerzo realizado por mostrar los dos sistemas operativos más actuales, LINUX y Windows NT, ha dado como resultado final un texto didáctico y aplicado, que puede ser usado tanto en cursos de introducción como de diseño de sistemas operativos. En el libro se incluyen ejemplos que muestran el uso de las llamadas al sistema de POSIX y Win32. Dichos ejemplos han sido cuidadosamente compilados y enlazados en los dos entornos en que estaban disponibles: Visual C y gcc.

Nos gustaría mostrar nuestro agradecimiento a todas las personas que han colaborado en este texto con su ayuda y sus comentarios. Este agradecimiento se dirige especialmente a Francisco Rosales García, Alejandro Calderón Mateos y José María Pérez Menor por su ayuda en la compilación de los programas de ejemplo y en algunos proyectos de sistemas operativos.

Jesús Carretero Pérez
Félix García Caballeira
Departamento de Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid
Madrid, España

Pedro de Miguel Anasagasti
Fernando Pérez Costoya
Departamento de Arquitectura y Tecnología
de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
Madrid, España

Digitalización realizada con propósito académico

Digitalización realizada con propósito académico

1

Conceptos arquitectónicos de la computadora

En este capítulo se presentan los conceptos de arquitectura de computadoras más relevantes desde el punto de vista de los sistemas operativos. El capítulo no pretende convertirse en un tratado de arquitectura, puesto que su objetivo es el de recordar y destacar los aspectos arquitectónicos que afectan de forma directa al sistema operativo.
Para alcanzar este objetivo, el capítulo se estructura en los siguientes grandes temas:

- *Funcionamiento básico de las computadoras y estructura de las mismas.*
- *Modelo de programación, con énfasis en su secuencia de ejecución.*
- *Concepto de interrupción.*
- *Diversas acepciones de reloj.*
- *Aspectos más relevantes de la jerarquía de memoria y, en especial, de la memoria virtual .*
- *Concurrencia de la LIS con el procesador.*
- *Mecanismos de protección.*

1.1. ESTRUCTURA Y FUNCIONAMIENTO DE LA COMPUTADORA

La computadora es una máquina destinada a procesar datos. En una visión esquemática, como la que muestra la Figura 1.1, este procesamiento involucra dos flujos de información: el de datos y el de instrucciones. Se parte del flujo de datos que han de ser procesados. Este flujo de datos es tratado mediante un flujo de instrucciones de maquina, generado por la ejecución de un programa, y produce el flujo de datos resultado.

Figura 1.1. *Esquema de funcionamiento de la computadora.*

Para llevar a cabo la función de procesamiento, una computadora con arquitectura von Neuman está compuesta por los cuatro componentes básicos representados en la Figura 1.2.

La memoria principal se construye con memoria RAM y memoria ROM. En ella han de residir los datos a procesar, el programa máquina (Aclaración 1.1) a ejecutar y los resultados.

La memoria está formada por un conjunto de celdas idénticas. Mediante la información de dirección se selecciona de forma única la celda sobre la que se quiere realizar el acceso, pudiendo ser éste de lectura o de escritura. En las computadoras actuales es muy frecuente que el direccionamiento se realice a nivel de byte, es decir, que las direcciones 0, 1, 2,... identifiquen los bytes 0, 1, 2,... Sin embargo, el acceso se realiza sobre una palabra de varios bytes (típi ente de 4 o de 8 bytes) cuyo primer byte se sitúa en la dirección utilizada.

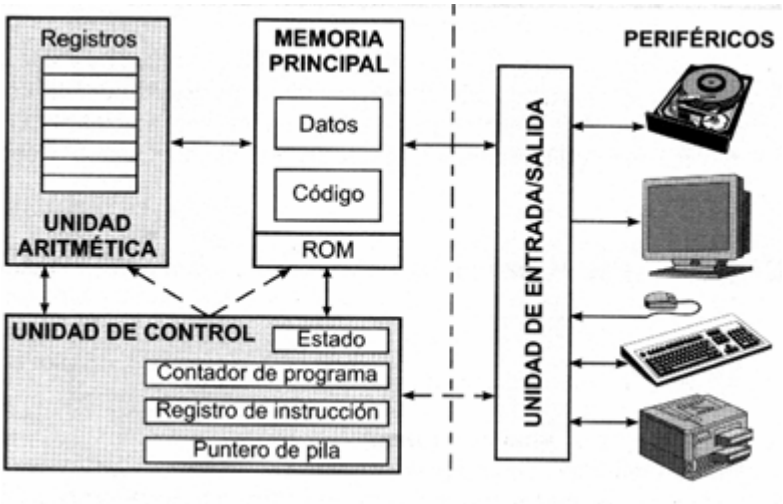


Figura1.2. Componentes básicos de la computadora

Digitalización realizada con propósito académico

ACLARACIÓN 1.1

Se denomina programa máquina (o código) al conjunto de instrucciones máquina que tiene por objeto que la computadora realice una determinada función. Los programas escritos en cualesquiera de los lenguajes de programación han de convertirse en programas máquina para poder ser ejecutados por la computadora

La **unidad aritmética** permite realizar una serie de operaciones aritméticas y lógicas sobre uno o dos operandos. Los datos sobre los que opera esta unidad están almacenados en un conjunto de registros, o bien provienen directamente de memoria principal. Por su lado, los resultados también se almacenan en registros o en memoria principal.
La **unidad de control** es la que se encarga de hacer funcionar al conjunto, para lo cual realiza las siguientes funciones:

- Lee de memoria las instrucciones máquina que forman el programa.
- Interpreta cada instrucción leída.
- Lee los datos de memoria referenciados por cada instrucción.
- Ejecuta cada instrucción.
- Almacena el resultado de cada instrucción.

La unidad de control tiene asociados una serie de registros, entre los que cabe destacar: el **contador de programa** (PC, *program counter),* que indica la dirección de la siguiente instrucción de máquina a ejecutar, el puntero de pila (**SP**, *snack pointer),* que sirve para manejar cómodamente una pila en memoria principal, el **registro de instrucción (RL)**, que permite almacenar la instrucción de maquina a ejecutar, y **el registro de estado (RE)**, que almacena diversa información producida por la ejecución de alguna de las últimas instrucciones del programa (bits de estado aritméticos) e información sobre la forma en que ha de comportarse la computadora (bits de interrupción, nivel de ejecución, etc.).

Finalmente, la unidad de **entrada/salida *(E/S)*** se encarga de hacer la transferencia de información entre la memoria principal (o los registros) y los periféricos. La entrad salida se puede hacer bajo el gobierno de la unidad de control ***(E/S*** programada) o de forma independiente (**DMA**), como se verá en la Sección 1.7.

Se denomina **procesador**, o unidad central de proceso (**UCP**), al conjunto de la unidad aritmética y de control. Actualmente, el procesador suele construirse en un único circuito integrado.
Desde el punto de vista de los sistemas operativos, nos interesa más profundizar en el funcionamiento interno de la computadora que en los componentes físicos que la constituyen.

1.2. MODELO DE PROGRAMACIÓN DE LA COMPUTADORA

El modelo de programación a bajo nivel de una computadora se caracteriza por los siguientes aspectos, que se muestran gráficamente en la Figura 1.3:

- **Elementos de almacenamiento**. En esta sección se consideran aquellos elementos de almacenamiento de la computadora que son visibles a las instrucciones maquina. En esta categoría están incluidos los registros generales, el contador de programa, el puntero de pila, el registro de estado, la memoria principal y el mapa de **E/S** (Aclaración 1.2).

Digitalización realizada con propósito académico

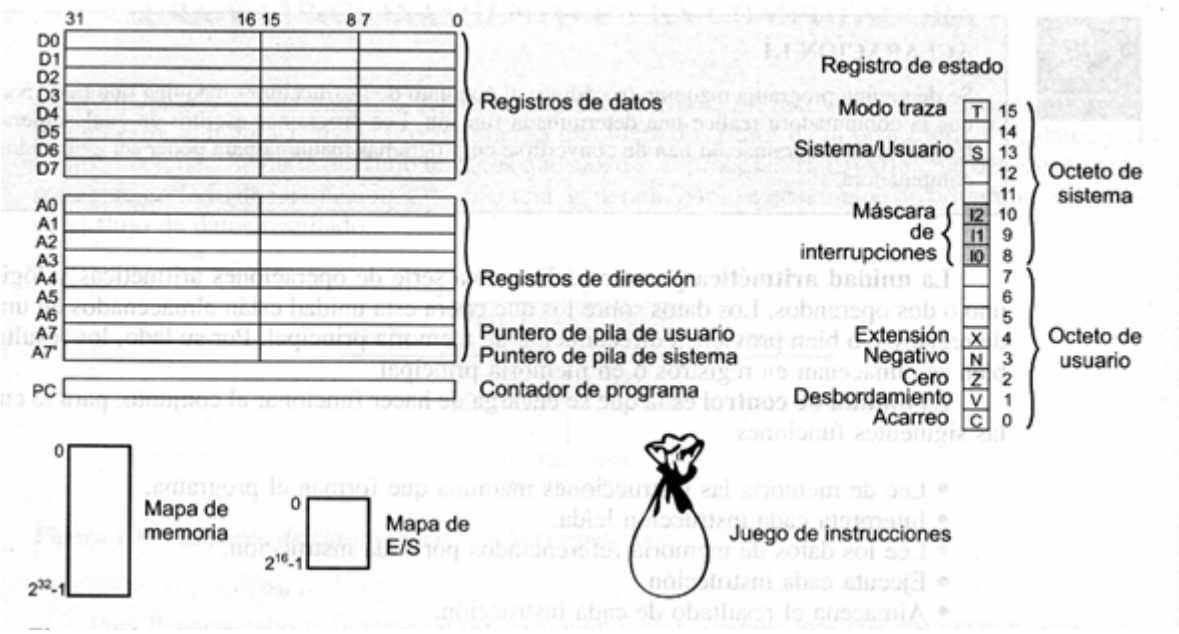


Figura 1.3. *Modelo de programación de una computadora.*

- **Juego de instrucciones** con sus correspondientes **modos de direccionamiento**. El juego de instrucciones máquina define las operaciones que es capaz de hacer la computadora. Los modos de direccionamiento determinan la forma en que se especifica la identidad de los elementos de almacenamiento que intervienen en las instrucciones máquina.
- **Secuencia de funcionamiento**, Define el modo en que se van ejecutando las instrucciones máquina.
- Un aspecto crucial de las computadoras, que está presente en todas ellas menos en los modelos más simples, e. que disponen de más de un nivel de ejecución, concepto que se analiza en la sección siguiente.

ACLARACIÓN 1.2

Es muy frecuente que las computadoras incluyan el mapa de E/S dentro del mapa de memoria. En este caso, se reserva una parte del mapa de memoria para realizar la E/S.

1.2.1. Niveles de ejecución

La mayoría de las computadoras actuales presentan dos o más niveles de ejecución. En el nivel menos permisivo, generalmente llamado **nivel de usuario**, la computadora ejecuta solamente un subconjunto de las instrucciones máquina, quedando prohibidas las demás. Además, el acceso a determinados registros, o a partes de esos registros, y a determinadas zonas del mapa de memoria y de E/S t bien queda prohibido. En el nivel más permisivo, denominado **nivel de núcleo**, la computadora ejecuta todas sus instrucciones sin ninguna restricción y permite el acceso a todos los registros y mapas de direcciones.

Digitalización realizada con propósito académico

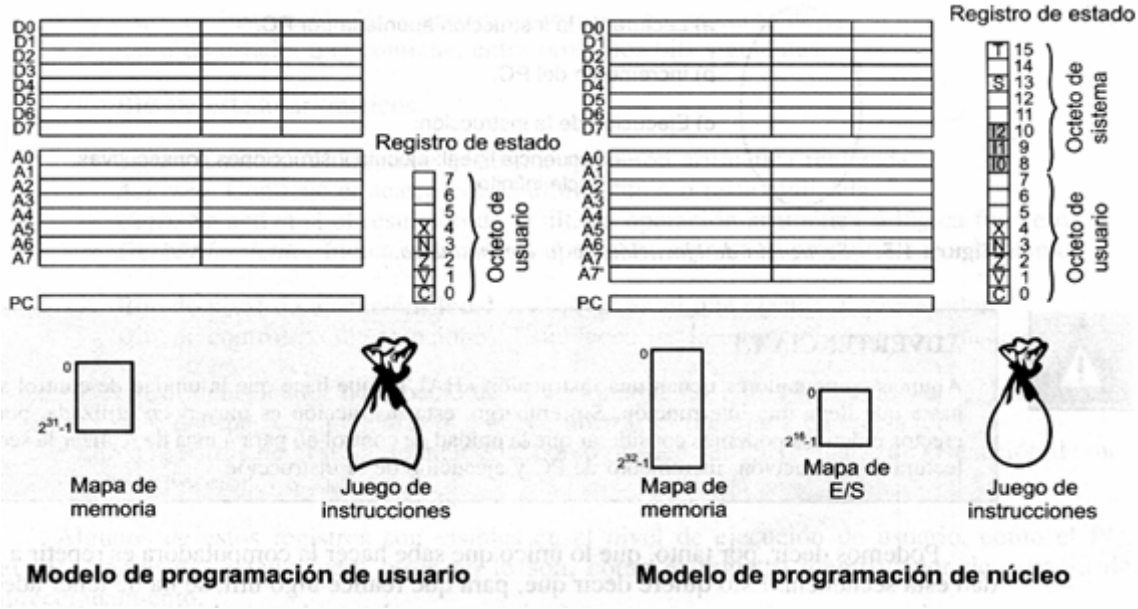


Figura 1.4. *Modelos de programación de usuario y de núcleo.*

Se puede decir que la computadora presenta mas de un modelo de programación. Uno más restrictivo, que permite realizar un conjunto limitado de acciones, y otros más permisivos que permiten realizar un mayor conjunto de acciones. Uno o varios bits del registro de estado establecen el nivel en el que está ejecutando la máquina. Modificando esto. bits se cambia de nivel de ejecución.

Como veremos más adelante, los niveles de ejecución se incluyen en las computadoras para dar soporte al sistema operativo. Los programas de usuario, por razones de seguridad, no podrán realizar determinadas acciones al ejecutar en nivel de usuario. Por su lado, el sistema operativo, que ejecuta en nivel de núcleo, puede ejecutar todo tipo de acciones.

Típicamente, en el nivel de usuario la computadora no permite operaciones de E/S, ni modificar una gran parte del registro de estado, ni modificar los registros de soporte de gestión de memoria. La Figura 1.4 muestra un ejemplo de dos modelos de programación de una computadora.

1.2.2. Secuencia de funcionamiento de la computadora

La unidad de control de la computadora es la que establece el funcionamiento del mismo. Este funcionamiento está basado en una secuencia sencilla, que se repite a alta velocidad (cientos de millones de veces por segundo). Como muestra la Figura *1.5,* esta secuencia consiste en tres pasos:

a) lectura de memoria principal de la instrucción máquina apuntada por el contador de programa, *b)* incremento del contador de programa —para que apunte a la siguiente instrucción máquina— y *c)* ejecución de la instrucción.

Esta secuencia tiene dos propiedades fundamentales: es lineal, es decir, ejecuta de forma consecutiva las instrucciones que están en direcciones consecutivas, y forma un bucle infinito, Esto significa que la unidad de control de la computadora está continua e ininterrumpidamente realizando esta secuencia (Advertencia 1.1).

Digitalización realizada con propósito académico

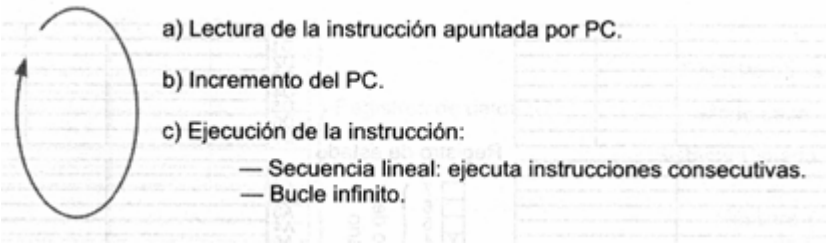


Figura 1.5. *Secuencia de ejecución de la computadora.*

ADVERTENCIA 1.1

Algunas computadoras tienen una instrucción «HALT» que hace que la unidad de control se detenga hasta que llega una interrupción. Sin embargo, esta instrucción es muy poco utilizada, por lo que a efectos prácticos podemos considerar que la unidad de control no para nunca de realizar la secuencia de lectura de instrucción, incremento de PC y ejecución de la instrucción.

Podemos decir, por tanto, que lo único que sabe hacer la computadora es repetir a gran veloci- dad esta secuencia. Esto quiere decir que, para que realice algo útil, se ha de tener adecuadamente cargados en memoria un programa máquina con sus datos y hemos de conseguir que el contador de programa apunte a la instrucción máquina inicial del programa.
El esquema de ejecución lineal es muy limitado, por lo que se añaden unos mecanismos que permiten alterar esta ejecución lineal. En esencia todos ellos se basan en algo muy simple: modifi- can el contenido del contador de programa, con lo que se consigue que se salte o bifurque a otro segmento del programa o a otro programa (que, lógicamente, también ha de residir en memoria).

Los tres mecanismos básicos de ruptura de secuencia son los siguientes:

- Las instrucciones máquina de salto o bifurcación, que permiten que el programa rompa su secuencia lineal de ejecución pasando a otro segmento de si mismo.
- Las interrupciones externas o internas, que hacen que la unidad de control modifique valor el del contador de programa saltando a otro programa.
- La instrucción de máquina «**TRAP**», que produce un efecto similar a la interrupción, haciendo que se salte a otro programa.

Si desde el punto de vista de la programación son especialmente interesantes las instrucciones de salto, desde el punto de vista de los sistemas operativos son mucho más importantes las interrupciones y las instrucciones de **TRAP**. Por tanto, centraremos nuestro interés en resaltar los aspectos fundamentales de estos dos mecanismos.

1.2.3. Registros de control y estado

Como se ha indicado anteriormente, la unidad de control tiene asociada una serie de registros denominamos de control y estado. Estos registros dependen de la arquitectura de la computadora muchos de ellos se refieren a aspectos que se analizarán a lo largo del texto, por lo que no se intentará explicar aquí su función. Entre los más importantes se pueden encontrar los siguientes:

- Contador de programa **PC.** Contiene la dirección de la siguiente instrucción de máquina.
- Puntero de pila **SP.** Contiene la dirección de la cabecera de la pila.

Digitalización realizada con propósito académico

- Registro de instrucción **RI.** Contiene la instrucción en curso de ejecución.
- Registro de estado, que contiene, entre otros, los bits siguientes:

  - Bits de estado aritméticos:

  Signo. Contiene el signo de la ultima operación aritmética realizada.
  Acarreo. Contiene el acarreo de la ultima suma o resta realizada,
  Cero. Se activa si el resultado de la ultima operación aritmética o lógica fue cero.
  Desbordamiento. Indica si la última operación aritmética produjo desbordamiento.

  - Bits de nivel de ejecución. Indican el nivel en el que ejecuta el procesador.
  - Bits de control de interrupciones. Establecen las interrupciones que se pueden aceptar.

- Registro identificador de espacio de direccionamiento **RIED** (Sección 1.8.2). Identifica el espacio del mapa de memoria que puede utilizar el programa en ejecución.
- Otros registros de gestión de memoria, como pueden ser los registros de protección de memoria (Sección 1.8.2).

Algunos de estos registros son visibles en el nivel de ejecución de usuario, como el **PC**, el **SP** y parte del estado, pero otros no lo son, como el registro identificador de espacio de direccionamiento.

1.3. INTERRUPCIONES

A nivel físico, una interrupción se solicita activando una señal que llega a la unidad de control. El agente generador o solicitante de la interrupción ha de activar la mencionada señal cuando necesite que se le atienda, es decir, que se ejecute un programa que le atienda.
Ante la solicitud de una interrupción, siempre y cuando esté habilitado ese tipo de interrupción, la unidad de control realiza un **ciclo de aceptación de interrupción**. Este ciclo se lleva a cabo en cuanto termina la ejecución de la instrucción maquina que se esté ejecutando y consiste en las siguiente. operaciones:

- Salva algunos registros del procesador, como son el de estado y el contador de programa.
- Eleva el nivel de ejecución del procesador, pasándolo a núcleo.
- Carga un nuevo valor en el contador de programa, por lo que pasa a ejecutar otro programa.

La Figura 1.6 muestra la solución más usualmente utilizada para determinar la dirección de salto. Se puede observar que el agente que interrumpe ha de suministrar un vector, que especifica la dirección de comienzo del programa que desea que le atienda (programa que se suele denominar de tratamiento de interrupción). La unidad de control, utilizando un direccionamiento indirecto, toma la mencionada dirección de una tabla de interrupciones y la carga en el contador de programa. El resultado de esta carga es que la siguiente instrucción maquina ejecutada es la primera del mencionado programa de tratamiento de interrupción.

Obsérvese que tanto la tabla de interrupciones como la rutina de tratamiento de la interrupción se han considerado parte del sistema operativo. Esto suele ser así por razones de seguridad; en concreto, para evitar que los programas que ejecuta un usuario puedan perjudicar a los datos o programas de otros usuarios. Como se verá en el Capítulo 2, la seguridad es una de las funciones primordiales del sistema operativo.

Digitalización realizada con propósito académico

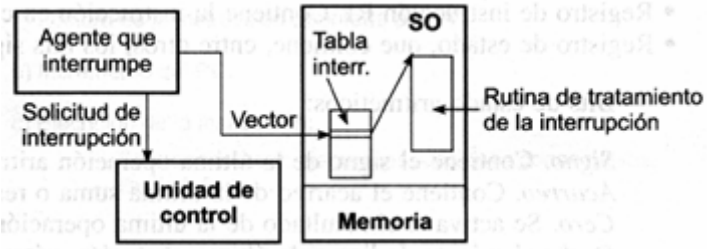


Figura 1.6. *Acceso a la rutina de tratamiento de la interrupción*

Las interrupciones se pueden generar por diversas causas, que se pueden clasificar de la siguiente forma:

- Excepciones de programa. Hay determinadas causas que hacen que un programa presente un problema en su ejecución, por lo que deberá generarse una interrupción, de forma que el sistema operativo trate dicha causa. Ejemplos son el desbordamiento en las operaciones aritméticas, la división por cero, el intento de ejecutar una instrucción con código operación incorrecto o de direccionar una posición de memoria prohibida (Advertencia 1.2).
- Interrupciones de reloj, que se analizarán en la sección siguiente.
- Interrupciones de **E/S**. Los controladores de los dispositivos de E/S necesitan interrumpir para indicar que han terminado una operación o conjunto de ellas.
- Excepciones del hardware. La detección de un error de paridad en la memoria o un corriente se avisan mediante la correspondiente interrupción.
- Instrucciones de **TRAP.** Estas instrucciones permiten que un programa genere una interrupción. Como veremos más adelante, estas instrucciones se emplean fundamentalmente solicitar los servicios del sistema operativo.

ADVERTENCIA 1.2

En este caso no existe un agente externo que suministre el vector necesario para entrar en la tabla de interrupciones. Será la propia unidad de control del procesador la que genere este vector.

Como complemento al mecanismo de aceptación de interrupción, las computadoras incluyen una instrucción máquina para retorno de interrupción, llamada **RETI**. El efecto de esta instrucción es restituir los registros de estado y **PC**, desde el lugar en que fueron salvados al aceptarse interrupción (p. ej.: desde la pila).

Las computadoras incluyen varias señales de solicitud de interrupción, cada una de las les tiene una determinada prioridad. En caso de activarse al tiempo varias de estas seña tratará la de mayor prioridad, quedando las demás a la espera de ser atendidas, Además, la computadora incluye un mecanismo de **inhibición** selectiva que permite detener todas o determinas señales de interrupción. Las señales inhibidas no son atendidas hasta que pasen a estar desinhibidas. La información de inhibición de las interrupciones suele incluirse en la parte del registro estado que solamente es modificable en nivel de núcleo, por lo que su modificación queda restringida al sistema operativo.

Digitalización realizada con propósito académico

1.4. EL RELOJ

El término *reloj* se aplica a las computadoras con tres acepciones diferentes, si bien relacionadas, como se muestra en la Figura 1.7. Estas tres acepciones son las siguientes:

- Señal que gobierna el ritmo de ejecución de las instrucciones máquina.
- Generador de interrupciones periódicas.
- Contador de fecha y hora,

El oscilador que gobierna las fases de ejecución de las instrucciones máquina se denomina *reloj*. Cuando se dice que un microprocesador es de 600 MHz, lo que se está especificando es que el oscilador que gobierna el ritmo de su funcionamiento interno produce una onda cuadrada con una frecuencia de 600 MHz.

La señal producida por el oscilador anterior, o por otro oscilador, se divide mediante un divisor de frecuencia para generar una interrupción cada cierto intervalo de tiempo. Estas interrupciones, que se están produciendo constantemente, se denominan **interrupciones de reloj o ticks**, dando lugar al segundo concepto de reloj. El objetivo de estas interrupciones es, como veremos más adelante, hacer que el sistema operativo entre a ejecutar de forma sistemática cada cierto intervalo de tiempo. De esta manera, el sistema operativo puede evitar que un programa monopolice el uso de la computadora y puede hacer que entren a ejecutarse programas en determinados instantes de tiempo. Estas interrupciones se producen cada varios milisegundos, por ejemplo cada 20 milisegundos.

La tercera acepción de reloj se aplica a un contador que permite conocer la **fecha y la hora**. Este contador se va incrementando con cada interrupción de reloj de forma que, tomando como referencia un determinado instante (p. ej: O horas del 1 de enero de 1990 (Advertencia 1.3)], se puede calcular la hora y fecha en que estamos. Observe que este concepto de reloj es similar al del reloj electrónico de pulsera. En las computadoras actuales esta cuenta se hace mediante un circuito dedicado que, además, está permanentemente alimentado, de forma que, aunque se apague la computadora, se siga manteniendo el reloj. En sistemas más antiguos, el sistema operativo se encargaba de hacer esta cuenta, por lo que había que introducir la fecha y la hora al arrancar la computadora.

ADVERTENCIA 1.3

En el caso de UNIX se cuentan segundos y se toma como referencia las 0 horas del 1 de enero de 1970. si se utiliza una palabra de 32 bits, el mayor número que se puede almacenar es el 2.147.483.647, que se corresponde a las 3h 14m y 7s de enero de 2038. esto significa que, a partir de ese instante, el contador tomará el valor de 0 y la fecha volverá a ser el 1 de enero de 1970.

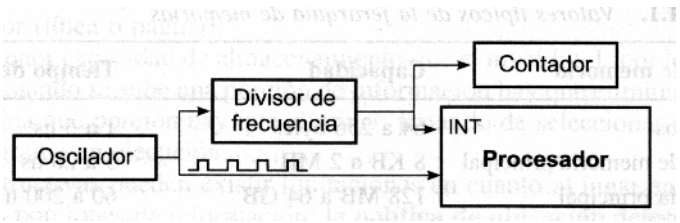


Figura 1.7. *Reloj de la computadora.*

Digitalización realizada con propósito académico

1.5. JERARQUÍA DE MEMORIA

Dado que la memoria de alta velocidad tiene un precio elevado y un tamaño reducido, la memoria de la computadora se organiza en forma de una jerarquía como la mostrada en la Figura 1.8. En esta jerarquía se utilizan memorias permanentes de alta capacidad y baja velocidad, como son los cos, para almacenamiento permanente de la información. Mientras que se emplean memorias semiconductores de un tamaño relativamente reducido, pero de alta velocidad, para almacenar la información que se está utilizando en un momento determinado.

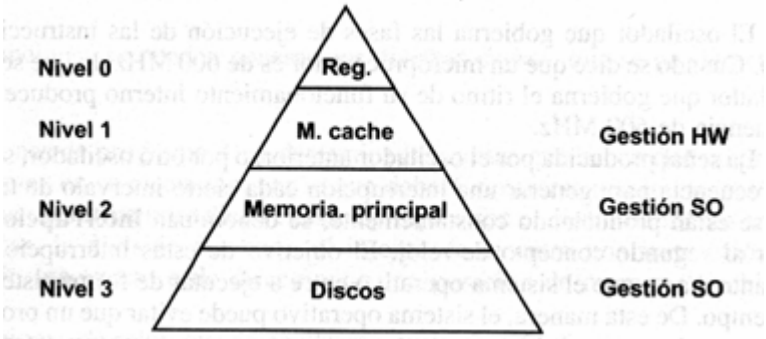


Figura 1.8. *Jerarquía de memoria.*

El funcionamiento de la jerarquía de memoria exige hacer adecuadas copias de información de los niveles más lentos a los niveles más rápidos, en los cuales son utilizadas (p. ej.: cuando se ejecutar un programa hay que leer el fichero ejecutable y almacenarlo en memoria principal). Inversamente, cuando se modifica o crea la información en un nivel rápido, y se desea su permanencia, hay que enviarla al nivel de disco o cinta.

Para entender bien el objetivo y funcionamiento de la jerarquía de memoria, es muy importante tener presente siempre tanto el orden de magnitud de los tiempos de acceso de cada tecnología de memoria como los tamaños típicos empleados en cada nivel de la jerarquía. La Tabla 1.1 presenta algunos valor típicos.

La gestión de la jerarquía de memoria, puesto que a de tener en cuenta las copias de información que están en cada nivel y a de realizar las trasferencias de información a niveles mas rápidos, así como las actualizaciones hacia los niveles permanentes. Una parte muy importante de esta gestión corre a cargo del sistema operativo, aunque, para hacerla correctamente, requiere de la ayuda de hardware. Por ello, se revisan en esta sección los conceptos mas importantes de la

Tabla 1.1. *Valores típicos de la jerarquía de memorias*

Nivel de memoria	Capacidad	Tiempo de acceso	Tipo de acceso
Registros	64 a 256 bytes	1 a 5 ns	Palabra
Cache de memoria principal	8 KB a 2 MB	5 a 20 ns	Palabra
Memoria principal	128 MB a 64 GB	60 a 200 ns	Palabra
Disco	50 MB a 40 GB	10 a 30 ms	Sector

Digitalización realizada con propósito académico

jerarquía de memoria, para analizar más adelante su aplicación a la memoria virtual, de especial interés para nosotros dado que su gestión la realiza el sistema operativo.

1.5.1. Migración de la información

La explotación correcta de la jerarquía de memoria exige tener, en cada momento, la información adecuada en el nivel adecuado. Para ello, la información ha de moverse de nivel, esto es, ha de migrar de un nivel a otro. Esta migración puede ser bajo demanda explícita o puede ser automática. La primera alternativa exige que el programa solicite explícitamente el movimiento de la información, como ocurre, por ejemplo, con un pro rama editor, que va solicitando la parte del archivo que está editando en cada momento el usuario. La segunda alternativa consiste en hacer la migración transparente al programa, es decir, sin que este tenga que ser consciente de que se produce. La migración automática se utiliza en las memorias cache y en la memoria virtual, mientras que la migración bajo demanda se utiliza en los otros niveles.

Sean *k* y *k* + 1 dos niveles consecutivos de la jerarquía, siendo *k* el nivel mas rápido. La existencia de una migración automática de información permite que el programa referencia la información en el nivel *k* y que, en el caso de que no exista una copia adecuada de esa información en dicho nivel *k,* se traiga esta desde el nivel *k* + 1 sin que el programa tenga que hacer nada para ello.

El funcionamiento correcto de la migración automática exige un mecanismo que consiga tener en el nivel *k* aquella información que necesita el programa en ejecución en cada instante. Idealmente, el mecanismo debería predecir la información que éste necesite para tenerla disponible en el nivel rápido *k.* El mecanismo se basa en los siguientes aspectos:

- Tamaño de los bloques transferidos.
- Política de extracción.
- Política de reemplazo.
- Política de ubicación.

Por razones de direccionamiento (Sección 1.5.4), y para aprovechar la proximidad espacial (Sección 1.5.5), la migración automática se hace en porciones de información de un t año determinado. En concreto, para la memoria cache se transfieren **líneas** de unas pocas palabras, mientras que p a la memoria virtual se transfieren paginas de uno o varios KB. El tamaño de estas porciones e una característica muy importante de la jerarquía de memoria.

La **política de extracción** define qué información se sube del nivel *k* + 1 al *k* y cuándo se sube. La solución más corriente es la denominada **por demanda** y consiste en subir aquella información que referencia el programa, justo cu do la referencia. El éxito de esta política se basa en la proximidad espacial (Sección *1.5.5),* por lo que no se sube exclusivamente la información referenciada sino una porción mayor (línea o página).

El nivel *k* tiene menor capacidad de almacenamiento que el nivel *k* + 1, por lo que normalmente está lleno. Por ello, cuando se sube una porción de información hay que eliminar otra. La **política de reemplazo** determina qué porción hay que eliminar, atando de seleccionar una que ya no sea de interés para el programa en ejecución.

Por razones constructivas pueden existir limitaciones en cuanto al lugar en el que se pueden almacenar las diversas porciones de información; la **política de ubicación** determina dónde almacenar cada porción.

Digitalización realizada con propósito académico

1.5.2. Parámetros característicos de la jerarquía de memoria

La eficiencia de la jerarquía de memoria se mide mediante los dos parámetros siguientes:

- Tasa de aciertos o *hit ratio (Hr).*
- Tiempo medio de acceso efectivo *(Tef).*

La **tasa dé aciertos** *(Hrk)* del nivel *k* de la jerarquía se define como la probabilidad de encontrar en ese nivel la información referenciada. La tasa de fallos *Frk* es 1 - *Hrk.* La tasa de aciertos ha de ser alta para que sea rentable el uso del nivel *k* de la jerarquía. Los factores más importantes que determinan *Hrk* son los siguientes:

- Tamaño de la porción de información que se transfiere al nivel *k.*
- Capacidad de almacenamiento del nivel *k.*
- Política de reemplazo.
- Política de ubicación.
- Programa específico que se esté ejecutando (cada programa tiene un comportamiento propio).

El tiempo de acceso a una información depende de que se produzca o no un fallo en el nivel *k*. Denominaremos tiempo de acierto al tiempo de acceso cuando la información se encuentra en nivel *k.* mientras que denominaremos penalización de fallo al tiempo que se tarda en realizar migración de la porción cuando se produce fallo. El **tiempo medio de acceso efectivo** *(Tef)* de un programa se obtiene promediando los tiempos de todos los accesos que realiza el programa a largo de su ejecución. *Tef* depende básicamente de los factores siguientes:

- Tiempo de acierto.
- Penalización de fallo.
- Tasa de aciertos *(Hrk)* del nivel *k.*

1.5.3. Coherencia

Un efecto colateral de la jerarquía de memoria es que existen varias copias de determinadas porcio- nes de información en distintos niveles. Al escribir sobre la copia del nivel *k,* se produce una discrepancia con la copia del nivel *k* + 1; esta situación se denomina falta de coherencia. Se dice que una porción de información está sucia si ha sido escrita.

La coherencia de la jerarquía de memoria exige medidas para eliminar la falta de coherencia. En concreto, una porción sucia en el nivel *k* ha de ser escrita en algún momento al nivel *k* + 1 para eliminar la falta de coherencia. Con esta operación de escritura se limpia la porción del nivel *k.*

Existen diversas políticas de actualización de la información creada o modificada, que se caracterizan por el instante en el que se copia la información al nivel permanente.

1.5.4. Direccionamiento

La jerarquía de memoria presenta un problema de direccionamiento. Supóngase que el programa en ejecución genera la dirección X del dato *A* al que quiere acceder. Esta dirección X está referida al nivel *k* + 1, pero se desea acceder al dato *A* en el nivel *k,* que es más rápido. Para ello se necesitara conocer la dirección *Y* que ocupa *A* en el nivel *k,* por lo que será necesario establecer un mecanismo de traducción de direcciones X en sus correspondientes *Y.* La Figura 1.9 presenta este problema de direccionamiento.

Digitalización realizada con propósito académico

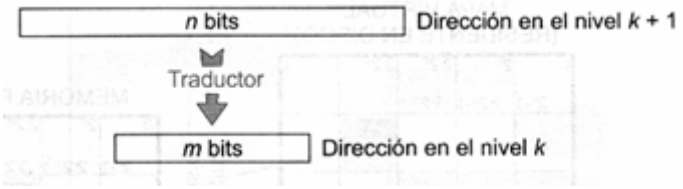


Figura 1.9. *Traducción de direcciones.*

El problema de traducción no es trivial, supóngase que el espacio de nivel *k* + 1 es de 2 GB, lo que equivale a suponer que *n* = 31, y que el espacio de nivel *k* es de 8 MB, lo que supone que *m* = 23. El traductor tiene aproximadamente dos mil millones de valores de entrada distintos y ocho millones de direcciones finales.

Para simplificar la traducción, y aprovechar la proximidad espacial, se dividen los mapas de direcciones de los espacios *k* + 1 y *k* en porciones de tamaño Y. Estas porciones constituyen la unidad de información mínima que se transfiere de un nivel al otro. El que la porción tenga tamaño 2^p permite dividir la dirección en dos partes: los *m* —~ *p* bits mas significativos sirven para identificar la porción, mientras que los p bits menos significativos sirven para especificar el byte dentro de la porción (Fig. 1.10). Por su parte, la Figura 1.11 muestra el caso de la memoria virtual que se divide en páginas.

Suponiendo, para el ejemplo anterior, que las páginas son de 1 KB (p = 10), el problema de direccionamiento queda dividido por 1.024, pero si e siendo inviable plantear la traducción mediante una tabla directa completa, pues sería una tabla de unos dos millones de entradas y con sólo 8.192 salidas no nulas.

1.5.5. La proximidad referencial

La proximidad referencial es la característica que hace viable la jerarquía de memoria, de ahí su importancia. En términos globales, la proximidad referencial establece que un programa en ejecución utiliza en cada momento una pequeña p te de toda la información que usa.

Para exponer el concepto de proximidad referencial de forma más específica, partimos del concepto de traza. La **traza** de un programa en ejecución es la lista ordenada en el tiempo de las direcciones de memoria que referencia para llevar a cabo su ejecución. Esta traza *R* estará compuesta por las direcciones de las instrucciones que se van ejecutando y por las direcciones de los datos empleados, es decir:

$$Re = re(l), re(2), re(3), \ldots re(j)$$

donde *re(i)* es la i-ésima dirección generada por la ejecución del programa *e.*

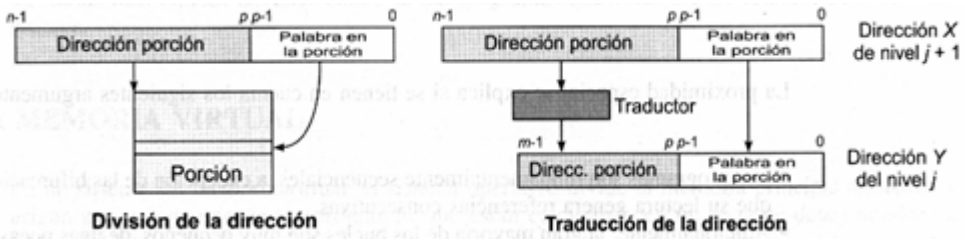


Figura 1.10. *El uso de porciones de 2" facilita la traducción.*

Digitalización realizada con propósito académico

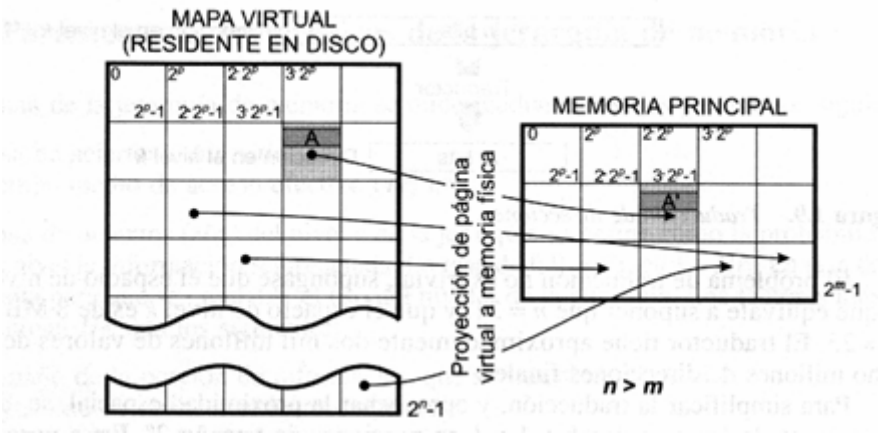


Figura 1.11. *División en páginas de los espacios de memoria.*

Adicionalmente, se define el concepto de distancia *d(u, y)* entre dos direcciones *u* y *y* como diferencia en valor absoluto | u — *v*|. La distancia entre dos elementos *j* y *k* de una traza *re* es, por tanto, *d(re(j), re(k))* = |*re(j)* — re(k)|.

También se puede hablar de traza de E/S, refiriéndonos, en este caso, a la secuencia de la direcciones empleadas en operaciones de E/S.

La proximidad referencial presenta dos facetas: la proximidad espacial y la proximidad temporal.

La **proximidad espacial** de una traza postula que, dadas dos referencias *re(j)* y *re(i)* próximas en el tiempo (es decir, que i — *j* sea pequeño), existe una alta probabilidad de que su distancia *d(re(j), re(i))* sea muy pequeña. Además, como muchos trozos de programa y muchas es estructuras datos se recorren secuencialmenté, existe una gran probabilidad de que la referencia siguiente *re(j)* coincida con la dirección de memoria siguiente (Recordatorio 1.1). Este tipo especial de proximidad espacial recibe el nombre de **proximidad secuencial**.

RECORDATORIO 1.1

Aquí conviene incluir una aclaración. Dado que las memorias principales se direccional a nivel de byte pero se acceden a nivel de palabra, la dirección siguiente no es la dirección actual mas 1. Para palabras de 4 bytes la dirección siguiente es la actual mas 4.

La proximidad espacial se explica si se tienen en cuenta los siguientes argumentos:

- Los programas son fundamentalmente secuénciales, a excepción de las bifurcaciones, por que su lectura genera referencias consecutivas.
- Adicionalmente, la gran mayoría de los bucles son muy pequeños, de unas pocas instrucciones máquina, por lo que su ejecución genera referencias con distancias pequeñas.

Digitalización realizada con propósito académico

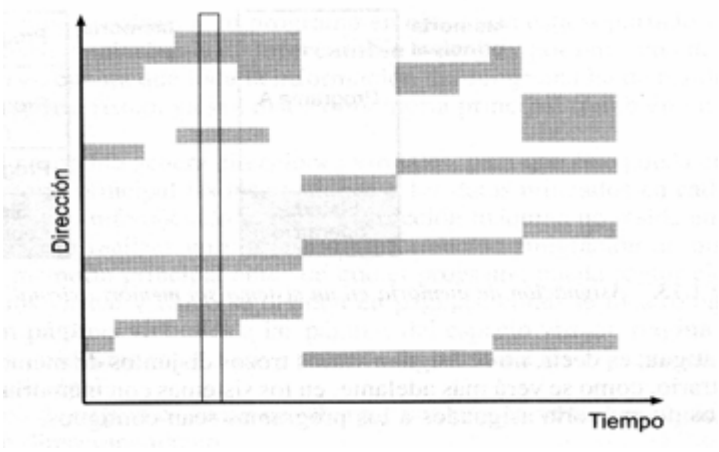


Figura 1.12. *Proximidad referencial.*

- Las estructuras de datos que se recorren de forma secuencial o con referencias muy próximas son muy frecuentes. Ejemplos son los vectores, las listas, las pilas, las matrices, etc. Además, las zonas de dato suelen estar agrupadas, de manera que las referencias que se generan suelen estar próximas.

La **proximidad temporal** postula que un programa en ejecución tiende a referenciar direcciones empleadas en un pasado próximo. Esto es, existe una probabilidad muy alta de que la próxima referencia *re(j* + 1) este entre las *n* referencias anteriores *re(j — n* + 1), *re(j — n* + 2),....., *re(j* -2), *re(j —* 1), *re (j*). La proximidad temporal se explica si se tienen en cuenta los siguientes argumentos:

- Los bucles producen proximidad temporal, además de proximidad espacial.
- El uso de datos o parámetros de forma repetitiva produce proximidad temporal.
- Las llamadas repetidas a subrutinas también son muy frecuentes y producen proximidad temporal. Esto es muy tipico con las funciones o subrutinas aritméticas, de conversión de códigos, etc.

En la práctica, esto significa que las referencias producidas por la ejecución de un programa están agrupadas en unas pocas zonas, tal y como muestra la Figura 1,12. Puede observarse también que, a medida que avanza la ejecución del programa, van cambiando las zonas referenciadas.
El objetivo principal de la gestión de la jerarquía de memoria será conseguir que residan en las memorias más rápidas aquellas zonas de los programas que están siendo referenciadas en cada instante.

1.6. MEMORIA VIRTUAL

En un sistema sin memoria virtual, el sistema operativo divide la memoria principal en trozos y asigna uno a cada uno de los programas que están ejecutando en un instante determinado. La Figura 1.13 muestra el reparto típico de la memoria para el caso de un solo programa o de varios programas. Observe que el espacio asignado a un programa consiste en una zona de memoria principal

Digitalización realizada con propósito académico

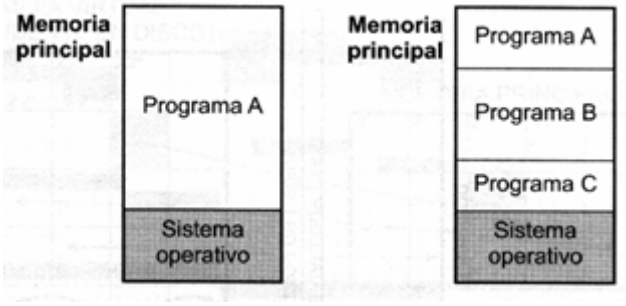


Figura 1.13. *Asignación de memoria en un sistema sin memoria virtual.*

contigua, es decir, no se asignan varios trozos disjuntos de memoria a un mismo programa. Por el contrario, como se verá más adelante, en los sistemas con memoria virtual no es necesario que los
espacios de memoria asignados a los programas sean contiguos.

1.6.1. Concepto de memoria virtual

La memoria virtual utiliza dos niveles de la jerarquía de memoria: la memoria principal y una memoria de respaldo (que suele ser el disco, aunque puede ser una memoria expandida). Sobre memoria de respaldo se establece un mapa uniforme de memoria virtual. Las direcciones generadas por el procesador se refieren a este mapa virtual, pero, sin embargo, los accesos reales se realiza sobre la memoria principal.

Para su funcionamiento, la memoria virtual exige una gestión automática de la parte de la jerarquía de memoria formada por los niveles de memoria principal y de disco.

Insistimos en que la gestión de la memoria virtual es automática y la realiza el sistema operativo con ayuda del hardware de la máquina. Como muestra la Figura 1.14, esta gestión incluye toda la memoria principal y una parte del disco, que sirve de respaldo a la memoria virtual.

Los aspectos principales en los que se basa la memoria virtual son los siguientes:

- Las direcciones generadas por las instrucciones máquina, tanto para referirse a datos como a otras instrucciones, están referidas al espacio virtual, es decir, forman parte del mapa de memoria virtual. En este sentido se suele decir que el procesador genera direcciones virtuales

Figura 1.14. *Fundamento de la memoria virtual.*

- El mapa virtual asociado a un programa en ejecución esta soportado físicamente por una zona del disco, denominada **de intercambio o *swap*,** y por una zona de la memoria principal. Tenga en cuenta que toda la información del programa ha de residir obligatoriamente en algún soporte físico, ya sea disco o memoria principal (también puede estar duplicada en ambos).
- Aunque el programa genera direcciones virtuales, para que éste pueda ejecutarse, han de residir en memoria principal las instrucciones y los datos utilizados en cada momento. Si, por ejemplo, un dato referenciado por una instrucción máquina no reside en la memoria principal es necesario realizar un trasvase de información (migración de información) entre el disco y la memoria principal antes de que el programa pueda seguir ejecutando.
- Los espacios virtual y físico se dividen en páginas, como se mostró en la Figura 1.11. Se denominan **páginas virtuales** a las páginas del espacio virtual, **paginas de intercambio** a las páginas residentes en el disco y **marcos de página** a los espacios en los que se divide la memoria principal.
- Cada marco de página es capaz de albergar una página virtual cualquiera, sin ninguna restricción de direccionamiento.
- Existe una unidad hardware, denominada **MMU** *(Memo Management Unit),* que traduce las direcciones virtuales a direcciones de memoria principal. Aplicando lo visto anteriormente, se puede decir que esta traducción se restringe a traducir el número de página virtual en el correspondiente número de marco de página. Insistimos en que esta traducción hay que hacerla por hardware dada la alta velocidad a la que debe hacerse (una fracción del tiempo de acceso de la memoria principal).
- Dado que en cada instante determinado solamente reside en memoria principal una fracción de las páginas del programa, la traducción no siempre es posible. Por tanto, la MMU producirá una **excepción de fallo de página** cuando ésta no esté en memoria principal.

Los fallos de página son atendidos por el sistema operativo (Prestaciones 1.1) que se encarga de realizar la adecuada migración de páginas, para traer la página requerida por el programa a un marco de página. Se denomina **paginación** al proceso de migración necesario para atender los fallos de pagina.

PRESTACIONES 1.1

Esto significa que los fallos de página son atendidos por software. Dado que el tiempo que se tarda en traer una página desde disco está en el entorno de centenas de milisegundos, en comparación, el coste de procesar el fallo de página por software es despreciable.

Finalmente, conviene resaltar que el tamaño del espacio virtual suele ser muy grande. En la actualidad se emplean direcciones de 32, 48 o hasta 64 bits, lo que significa espacios virtuales de 232, 248 y 264 bytes. Dado que los programas requieren en general mucho menos espacio, una de las funciones que realiza el sistema operativo es la **asignación de espacio virtual** a los programas para su ejecución. El programa no podrá utilizar todo el espacio virtual sino que ha de restringirse a la zona o zonas que le asigne el sistema operativo. La Figura 1.15 muestra que el espacio virtual reservado al programa A puede estar en una única zona o puede estar dividido en varias zonas, que se denominan **segmentos**.

.

Digitalización realizada con propósito académico

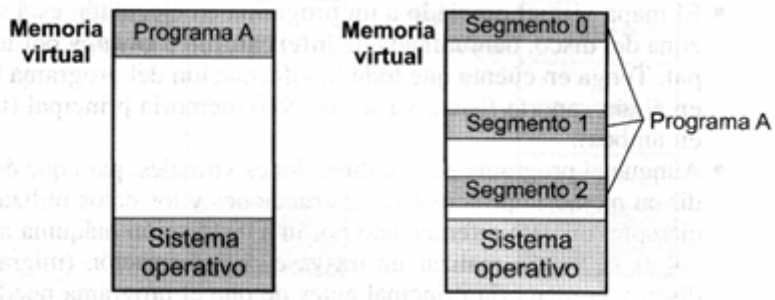


Figura *1.15.* *Asignación de memoria en un sistema con memoria virtual.*

Una de las características importantes de los lenguajes de programación actuales es que permiten la asignación dinámica de memoria. Esto significa que no se conoce a *priori* la cantidad de memoria que necesitará el programa para su ejecución. Por tanto, el sistema operativo ha de ser capa de aumentar o reducir el espacio asignado a un programa de acuerdo a las necesidades que vayan surgiendo en su ejecución.

1.6.2. La tabla de paginas

La tabla de páginas es una estructura de información que contiene la información de dónde residen las páginas de un programa en ejecución. Esta tabla permite, por tanto, saber si una página esta en memoria principal y, en su caso, en que marco específico reside.

Según se ha visto anteriormente, dado que el tamaño del espacio virtual suele ser muy grande, el tamaño de la correspondiente tabla de páginas puede ser muy grande (de millones de elementos). Sin embargo, como hemos visto, el sistema operativo se encarga de asignar a cada programa en ejecución un espacio virtual de tamaño ajustado a su: necesidades. De esta forma, la tabla de páginas queda reducida al valor necesario para que ejecute el programa.

La Figura 1.16 muestra la solución más sencilla de tabla de páginas de un nivel. En este caso se supone que toda la memoria asignada al programa es contigua. El número de la página virtual se utiliza como índice para entrar en la tabla. Cada elemento de la tabla tiene un bit para indicar si la página está en memoria principal y el número de marco en el que se encuentra la mencionada página o un valor nulo.

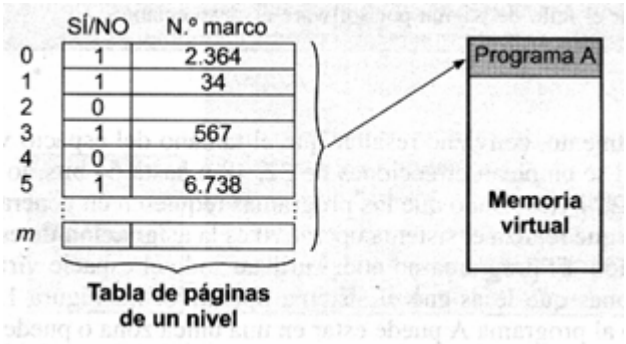


Figura 1.16. *Tabla de páginas de un nivel y espacio virtual asignado*

Digitalización realizada con propósito académico

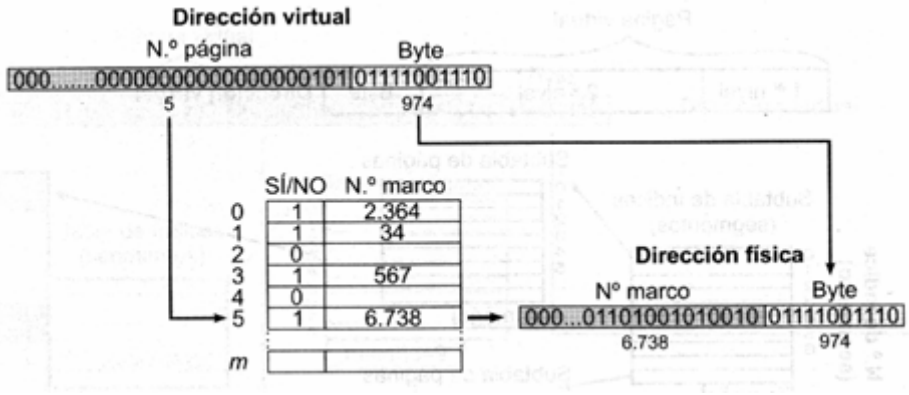


***Figura 1.17**.Ejemplo de traducción mediante tabla de páginas de un nivel*

La Figura 1.17 muestra un ejemplo de traducción para el caso de tabla de paginas de un nivel. Se supone que las página son de 2 KB, por lo que los 11 bits inferiores de la dirección virtual sirven para especificar el byte dentro de la pagina, mientras que el resto especifican la página virtual, que en este caso es la *5.* Entrando en la posición N^0 *5* de la tabla observamos que la página está en memoria principal y que esta en el marco numero 6.738. Concatenando el número de marco con los 11 bits inferiores de la dirección virtual se obtiene la dirección de memoria principal donde reside la información buscada.
El mayor inconveniente de la tabla de un nivel es su falta de flexibilidad. Toda la memoria virtual asignada ha de ser contigua (Advertencia 1.4) y la ampliación de memoria asignada solamente puede hacerse final de la existente.

ADVERTENCIA 1.4

El espacio virtual asignado es contiguo, pero no lo son los marcos de página que le pertenezcan en un momento dado, puesto que estarán dispersos por toda la memoria principal.

Sin embargo, los programas están compuestos por varios elementos, como son el propio programa objeto, la pila y los bloques de datos. Además, tanto la pila como los bloques de datos han de poder crecer. Por ello, un esquema de tabla de un nivel obliga a dejar grandes huecos de memoria virtual sin utilizar , pero que están presentes en la tabla con el consiguiente desperdicio de espacio.

Por ello se emplean esquemas de tablas de páginas de más de un nivel. La Figura 1.18 muestra el caso de tabla de páginas de dos niveles. Con este tipo de tabla, la memoria asignada esta compuesta por una serie de bloques de memoria virtual, es decir, por unos segmentos. Cada segmento está formado por una serie contigua de byte. que puede variar su tamaño, siempre y cuando no choque con otro segmento. La dirección se divide en tres pres. La primera identifica el segmento de memoria donde esta la información que se desea acceder. Con este valor se entra en una subtabla de segmentos, que contiene un puntero por segmento, puntero que indica el comienzo de la subtabla de paginas del segmento. Con la segunda parte de la dirección se entra en la subtabla de páginas seleccionada. Esta subtabla es similar a la tabla mostrada en la Figura 1.18, lo que permite obtener el marco en el que está la información deseada.

Digitalización realizada con propósito académico

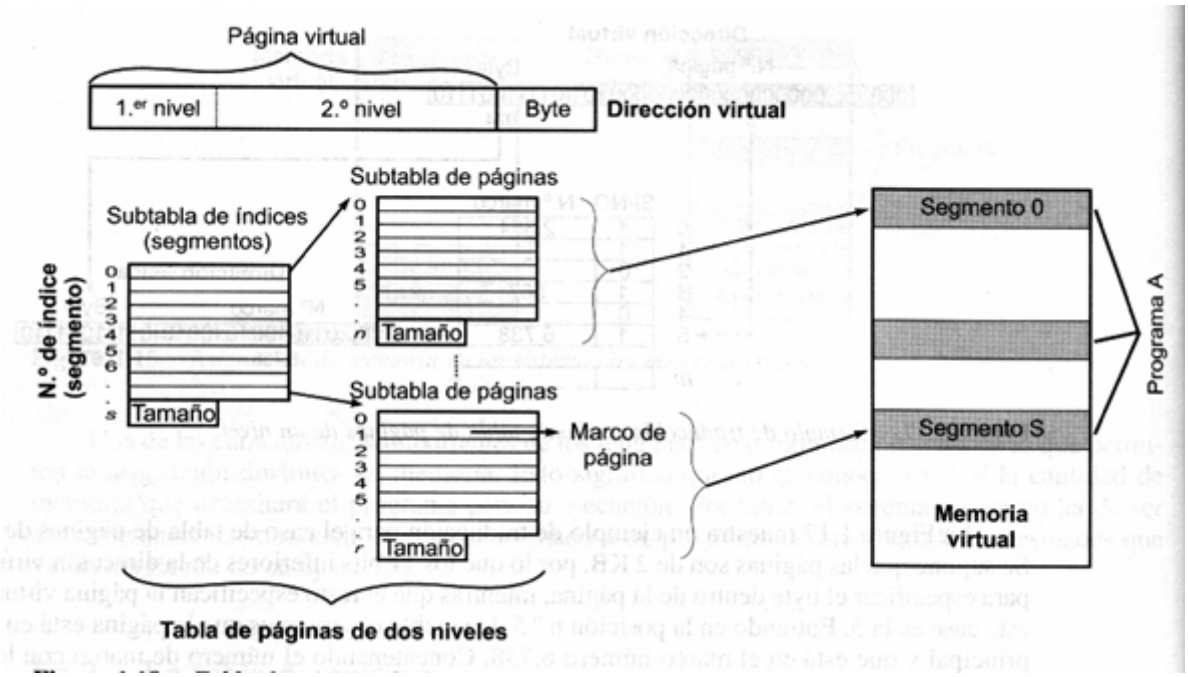


Figura 1.18. *Tabla de páginas de dos niveles.*

Obsérvese que se ha añadido a cada subtabla su tamaño. De esta forma se detectan las llamadas **violaciones de memoria**, producidas cuando el programa en ejecución intenta acceder una dirección que no pertenezca a los espacios asignados por el sistema operativo.
La ventaja del diseño con varios niveles es que permite una asignación de memoria más flexible que con un solo nivel, puesto que se pueden asignar bloques de memoria virtual disjuntos, por lo que pueden crecer de forma a independiente. Además, la tabla de páginas no tiene espacios vacíos, por lo que ocupa solamente el espacio imprescindible.
Las computadoras actuales suelen proporcionar tablas de varios niveles, algunos llegan asta cuatro, con lo que se consigue una mayor flexibilidad en la asignación de espacio de memoria.
La Figura 1.19 muestra un ejemplo de traducción mediante tabla de páginas de dos niveles. El segmento direccionado es el *5,* por lo que hay que leer la entrada *5* de la tabla de segmentos. Con ello se obtiene la dirección donde comienza la tabla de páginas de este segmento. La página direccionada es la 3, por lo que entramos en el elemento 3 de la tabla anterior. En esta tabla encontramos que el marco es el Hex4A24 (Advertencia 1.5), por lo que se puede formar la dirección física en la que se encuentra la información buscada.

ADVERTENCIA 1.5

La Figura 1.19 muestra que el contenido de la tabla de índices es rw-476AC2 y que el de la tabla de páginas es r-4A24. Con ello se muestra que cada elemento de la tabla, además de la dirección correspondiente, contiene otras informaciones que se analizarán más adelante en otros capítulos. En concreto, los bytes rwx (*read, write* y *execution*) se emplean para determinar los tipos de accesos que se permiten en el segmento y en la página.

Digitalización realizada con propósito académico

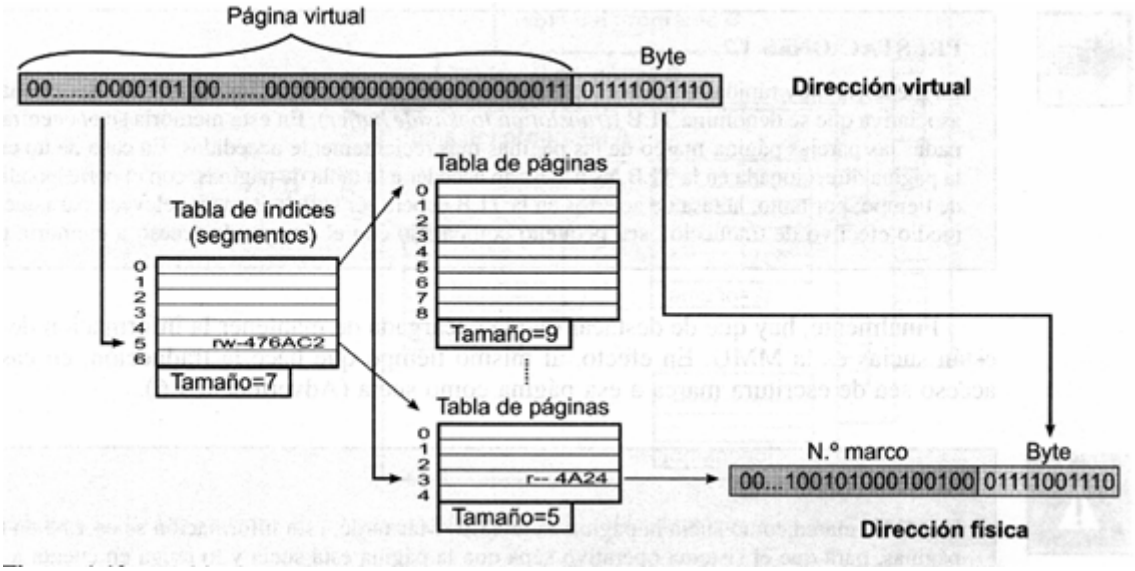


Figura 1.19, *Tabla de páginas de dos niveles.*

Por el contrario, las direcciones de la Figura 1.20 son incorrectas, puesto que el segmento 5 (101) no tiene la página 7 (111) y no existe el segmento 21(10101).

Traducción de direcciones

La asignación de memoria y, por tanto, la construcción de la tabla de páginas es misión del sistema operativo. Sin embargo, es la MMU la que se encarga de realizar la traducción de las direcciones. Esta división de trabajo es necesaria puesto que la traducción de direcciones hay que hacerla de forma muy rápida para que no afecte negativamente al tiempo de acceso a la memoria.

Para que una computadora con memoria virtual pueda competir con una sin memoria virtual, la traducción ha de tardar una fracción del tiempo de acceso a memoria. En caso contrario, sería mucho más rápido y por ende mas económico el sistema sin memoria virtual. Suponiendo una memoria principal de 100 ns y un traductor de *5* ns, el tiempo de acceso para el caso de memoria virtual es de 105 ns, es decir, un *5* por 100 más lento que en el caso de no tener memoria virtual. Sin embargo, si la traducción tardase 100 ns, la computadora con memoria virtual sería la mitad de rápida, algo que la haría imposible de competir.

La tabla de páginas es una estructura que mantiene el sistema operativo y que reside en memoria principal (a veces, hay una parte en la propia MMU y otra en memoria principal). Observe que esto parece un contrasentido, puesto que para acceder a memoria hay que traducir la dirección virtual, lo que supone realizar un acceso a memoria por cada nivel que tenga la tabla de páginas. Según se ha visto, esto suponía un retardo inadmisible en los accesos a memoria. Para resolver este problema se dota a la MMU de una memoria muy rápida que permite hacer la traducción para la mayoría de los casos en una fracción del tiempo que se tarda en acceder a la memoria principal (Prestaciones 1.2).

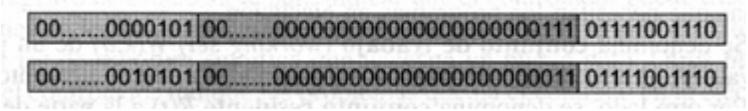


Figura 1.20. *Ejemplo de direcciones incorrectas.*

Digitalización realizada con propósito académico

PRESTACIONES 1.2

La memoria muy rápida incluida en la MMU para realizar la traducción de direcciones es una memoria asociativa que se denomina TLB (*translation lookaside buffer*). En esta memoria se encuentran almacenadas las parejas página-marco de las páginas más recientemente accedidas. En caso de no encontrarse la página direccionada en la TLB, es necesario acceder a la tabla de páginas, con el correspondiente gasto de tiempo; por tanto, la tasa de aciertos en la TLB deberá ser suficientemente elevada para que el tiempo medio efectivo de traducción sea pequeño comparado con el tiempo de acceso a memoria principal.

Finalmente, hay que destacar que la encargada de mantener la información de que página están sucias es la MMU. En efecto, al mismo tiempo que hace la traducción, en caso de que acceso sea de escritura marca a esa pagina como sucia (Advertencia 1.6).

ADVERTENCIA 1.6

La MMU marca como sucia la página en la TLB. Más tarde, esta información se escribe en la tabla de páginas, para que el sistema operativo sepa que la página está sucia y lo tenga en cuenta a la hora de hacer la migración de páginas.

1.6.3. Caso de varios programas activos

Como se verá en el Capítulo 2, los sistemas operativos permiten que existan varios programa activos al tiempo. De estos programas solamente puede haber uno en ejecución en cada instante, encargándose el sistema operativo de ir poniendo en ejecución uno detrás de otro de forma ordenada. Sin embargo, cada uno de los programas ha de tener asignado un espacio de memoria, por lo que ha de tener su propia tabla de páginas.

La MMU ha de utilizar la tabla de paginas correspondiente al programa que está en ejecución. Para ello, como muestra la Figura 1.21, el procesador tiene un **registro identificador de espacio de direccionamiento (RIED).** Este registro contiene la dirección en la cual está almacenada la tabla de índices o segmentos del programa. Cuando el sistema operativo pone en ejecución un programa ha de actualizar el valor del RIED para que apunte a la tabla de páginas adecuada.

1.6.4. Asignación de memoria principal y memoria virtual

En un sistema con memoria virtual, un programa en ejecución tiene asignado un espacio virtual, parte del cual reside en unos marcos de pagina de la memoria principal.

El objetivo de la políticas de extracción y de reemplazo que utilice el sistema operativo para hacer la migración de información entre el *intercambio* y la memoria principal tiene como objetivo conseguir, con el mínimo trabajo posible, que e estén en cada momento en memoria principal las páginas que necesitan los programas en ejecución.

Se denomina **conjunto de trabajo** *(working set) W(k,q)* de un programa en ejecución en el intervalo [k;q] al conjunto de páginas referenciadas entre el elemento *k* y el q de su traza.

Por otro lado, se denomina **conjunto residente** *R(t)* a la parte del proceso que está realmente almacenada u memoria principal en el instante *t.*

Digitalización realizada con propósito académico

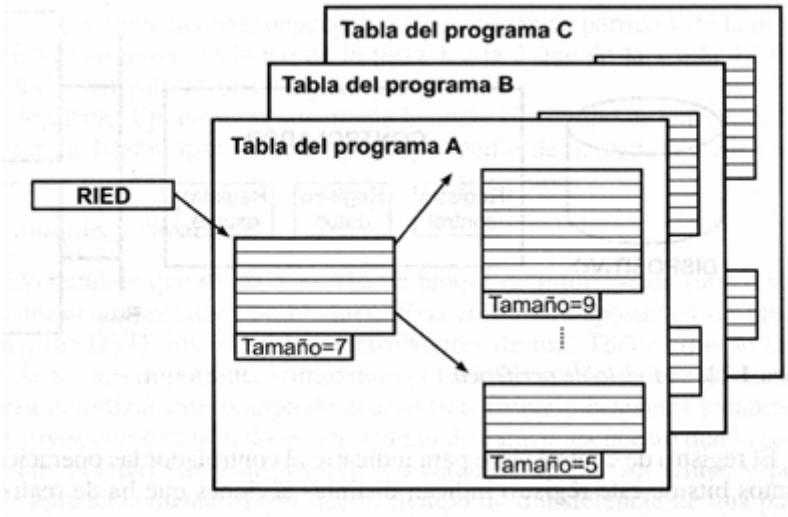


Figura 1.21 *El registro RIED permite determinar la tabla de páginas en uso.*

Supóngase que en el instante t el programa está por su referencia *k* y que el conjunto residente *R(t)* coincide con el conjunto de trabajo *W(k,q).* Esto significaría que ese programa tiene la garantía de que sus próximas q — *k* referencias se refieren a pagina: que están en memoria principal, por lo que no se generaría ningún fallo de página.
Dado que el sistema operativo no conoce de antemano cuales van a ser las referencias que generará un programa, ha de basan e en la trayectoria pasada de la ejecución del mismo para mantener un conjunto reciente que sea lo más parecido posible a su futuro conjunto de trabajo, para así minimizar la paginación.

1.7. ENTRADA-SALIDA

Los mecanismo: de E/S de la computadora tienen por objetivo el intercambio de información entre los periféricos y la memoria o los registros del procesador. En este capítulo se presentan los dos aspectos de la E/S que revisten mayor relevancia de cara al sistema operativo: la concurrencia de la E/S con el procesador y el impacto de la memoria virtual.

1.7.1. Periféricos

La Figura 1.22 muestra el esquema general de un periférico, compuesto por el dispositivo y su controlador. Este último tiene una serie de registros incluidos en el mapa de E/S de la computadora, por lo que ‘e pueden acceder mediante instrucciones de maquina de entrad salida.

El registro de datos sirve para el intercambio de datos. En él ira cargando el controlador los datos leídos y de él ira extrayendo lo datos para su escritura en el periférico.
Un bit del registro de estado sirve para indicar que el controlador puede transferir una palabra. En las operaciones de lectura esto significa que ha cargado en el registro de datos un nuevo valor, mientras que en las de escritura significa que necesita un nuevo dato. Otro: bits de este registro sirven para que el controlador indique los problemas que ha encontrado en la ejecución de la última operación de entrada/salida.

Digitalización realizada con propósito académico

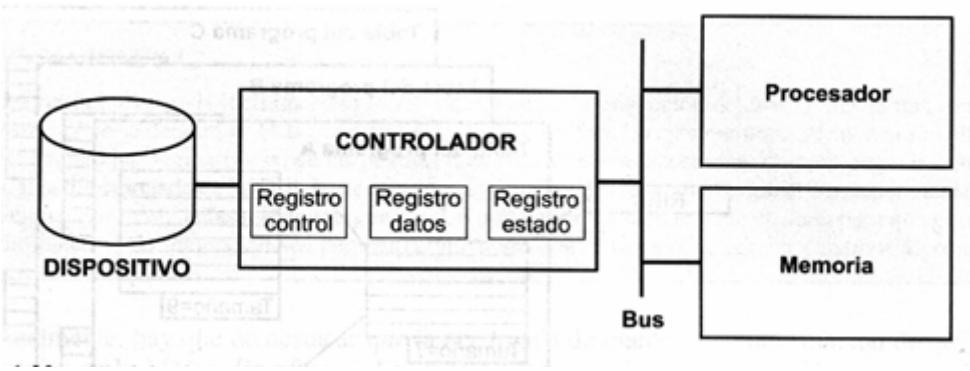


Figura 1.22 Modelo de Periférico

El registro de control sirve para indicarle al controlador las operaciones que ha de realizar. Los distintos bits de este registro indican distintas accione que ha de realizar el periférico.

El disco magnético

El disco magnético es, para el sistema operativo, el periférico más importante, puesto que sirve de espacio de intercambio a la memoria virtual y sirve de almacenamiento permanente para los programa y los datos, encargándose el sistema operativo de la gestión de este tipo de dispositivo.

Para entender la forma en que el sistema operativo trata a los discos magnéticos es necesario conocer las características de los mismos, entre las que destacaremos tres: organización de la información, tiempo de acceso y velocidad de transferencia.

La **organización de la información** del disco se realiza en contenedores de tamaño fijos denominados **sectores** (tamaños típicos del sector son *256,* 512 o 1.024 bytes). Como muestra Figura 1.23, el disco se divide en pistas que, a su vez, se dividen en sectores.

Las operaciones se realizan a nivel de sector, es decir, no se puede escribir o leer una palabra o byte individual: hay que escribir o leer de golpe uno o varios sectores.

El **tiempo de acceso** de estos dispositivos viene dado por el tiempo que tardan en Posicionar el brazo en la pista deseada, esto es, por el **tiempo de búsqueda**, más el tiempo que tarda la información de la pista en pasar delante de la cabeza por efecto de la rotación del disco, esto es, mas la

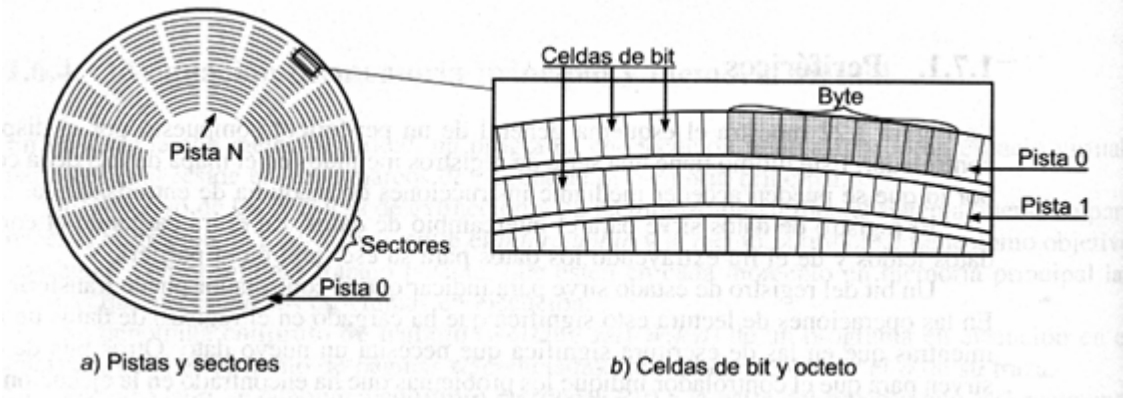


Figura 1.23. *Organización del disco.*

latencia. Obsérvese que estos tiempos dependen de la posición de partida y de la posición deseada. *No* se tarda lo mismo en mover el brazo de la pista 1 a la 2 que de la 1 a la 1.385. Por ello, los fabricantes suelen dar los valores medios y los peores.

Un sistema de cabeza fija presenta solamente latencia sin tiempo de sincronización. por tanto, suponiendo que gira a 10.000 rpm, tendrá un tiempo medio de acceso de 3 ms (1/2 revolución).

Dispositivos de bloques y caracteres

El disco magnético requiere que se lea o escriba un bloque de información (uno o varios sectores), por lo que se denomina dispositivo de bloques. Existen otros dispositivos de bloques como las cintas magnéticas, los DVD, los CD y los controladores de red. Todos ellos se caracterizan por tener un tiempo de acceso importante comparado con el tiempo de transferencia de una palabra, por lo que interesa amortizar este tiempo de acceso transfiriendo bastantes palabras.
Otros dispositivos como el teclado se denominan de caracteres. puesto que la operación básica de acceso es de un carácter. Estos dispositivos se cauterizan por ser lentos y por no tener un tiempo de acceso apreciablemente mayor que el tiempo de transferencia de una palabra.

1.7.2. E/S y concurrencia

Los periféricos son sensiblemente más lentos que el procesador, por ejemplo, durante el tiempo que se tarda en acceder a una información almacenada en un disco, un procesador moderno es capaz de ejecutar varios millones de instrucciones de máquina (Prestaciones 1.3). Es, por tanto, muy conveniente que mientras se está esperando a que se complete una operación de E/S el procesador esté ejecutando un programa útil y no un bucle de espera.

PRESTACIONES 1.3

Una computadora actual de 300 MHz puede ejecutar 600 MIPS (600 millones de instrucciones por segundo). Si se compara esto con el tiempo de acceso de un disco, que es del orden de los 15 ms, se puede calcular que en este tiempo la computadora ha ejecutado unos nueve millones de instrucciones de máquina.

Las computadoras presentan tres modos básicos de realizar operaciones de E/S: E/S programada, E/S por interrupciones y E/S por DMA *(direct memory access).* La E/S programada exige que el procesador esté ejecutando un programa de E/S, por lo que no existe ninguna concurrencia entre el procesador y la E/S. Sin embargo, en los otros dos modos de E/S el procesador no tiene que estar tendiendo directamente a la E/S, por lo que puede estar ejecutando otro programa. Se dice, entonces, que existe concurrencia entre la E/S y el procesador. Esta concurrencia permite optimizar el uso del procesador, pero exige que las unidades de control de los periféricos sean más inteligentes, lo que supone que sean más complejas y más caras.

En términos generales, una operación de E/S se compone de tres fases, envío de la orden al periférico, lectura o escritura de los datos y fin de la operación.

La fase de envío de la orden consiste en escribir la orden en los registros del

Digitalización realizada con propósito académico

controlador del periférico, operación que puede hacerse mediante unas cuantas instrucciones de salida, Dado que el controlador es un dispositivo electrónico, estas escrituras se pueden hacer a la velocidad de procesador, sin esperas intermedias.

En la fase de transferencia de los datos interviene el periférico, típicamente mucho más lento que el procesador. Imaginemos una lectura a disco. En caso de realizar esta operación con E/S programada debemos ejecutar un bucle que lea el registro de estado del controlador, observe si esta activo el bit de dato disponible y, en caso positivo, lo lea. El bucle podría tener la estructura que se muestra en el Programa 1.1.

Programa 1.1. Bucle de E/S programada.

```
n= O
while n < m
read registro_control
                          if (registro_control = datoAisponible)
                        read registro_datos
                            store en memoria principal
                        n=n+ 1
                          endi f
endwhile
```

Observe que, hasta que se disponga del primer dato, el bucle puede ejecutarse cerca del millón de veces y que, entre dato y dato, se repetirá varias decenas de veces. Al llegar a completar número *m* de datos a leer, se termina la operación de *E/S.*

Se denomina **espera activa** cuando un programa queda en un bucle hasta que ocurra un evento. La espera activa consume tiempo del procesador, por lo que es muy poco recomendable cuando el tiempo de espera es grande en comparación con el tiempo de ejecución de una instrucción

En caso de utilizar E/S con interrupciones, el procesador, tras enviar la orden al controlador del periférico, puede dedicarse a ejecutar otro programa. Cuando el controlador disponga de un dato generará una interrupción. La rutina de interrupción deberá hacer la lectura del dato y su almacenamiento en memoria principal, lo cual conlleva un cierto tiempo del procesador. En este caso se dice que se hace **espera pasiva**, puesto que el programa que espera el evento no esta ejecutándose la interrupción se encarga de <<despertar >> al programa cuando ocurre el evento.

Finalmente, en caso de utilizar E/S por DMA, el controlador se encarga directa de ir transfiriendo los datos del periférico a memoria sin molestar al procesador. Una vez terminada la transferencia de todos los datos, genera una interrupción de forma que se sepa que ha terminado (Recordatorio 1.2).

RECORDATORIO 1.2

Un controlador que trabaje por DMA dialoga directamente con la memoria principal de la computadora. La fase de envío de una orden a este tipo de controlador exige incluir la dirección de memoria donde está el buffer de la transferencia, el número de palabras a transmitir y la dirección de la zona del periférico afectada.

La Tabla 1.2 presenta la ocupación del procesador en la realización de las distintas actividades de una operación de E/S según el modelo de FIS utilizado.

Digitalización realizada con propósito académico

Tabla 1.2. *Ocupación del procesador en operaciones de entrada/salida*

	Envío orden	Espera dato	Transferencia dato	Fin operación
E/S programada	Procesador	Procesador	Procesador	Procesador
E/S por interrupciones	Procesador	Controlador	Procesador	Procesador
E/S por DMA	Procesador	Controlador	Controlador	Procesador

Tabla 1.2. *Ocupación del procesador en operaciones de entrada/salida*

Puede observarse que la solución que presenta la máxima concurrencia y que descarga al máximo al procesador es la de E/S por DMA. Por otro lado, es la que exige una mayor inteligencia por parte del controlador.

Un aspecto fundamental de esta concurrencia es su explotación. En efecto, de nada sirve descargar al procesador del trabajo de E/S si durante ese tiempo no tiene nada útil que hacer. Será una función importante del sistema operativo el explotar esta concurrencia la E/S y el procesador, haciendo que este ultimo tenga trabajo útil el mayor tiempo posible.

1.7.3. E/S y memoria virtual

La memoria virtual presenta un problema importante frente a la entrada/salida. En efecto, el programa que solicita una operación de E/S especifica una variable que determina un buffer de memoria sobre el que se hace la operación. Para que el controlador del periférico que realiza la operación pueda operar por DMA, el buffer ha de residir en memoria principal. En caso contrario, se intentaría hacer, por ejemplo, una lectura de unos datos de disco para escribir en una página, que también está en disco. El hardware no es capaz de hacer este tipo de operación.

El sistema operativo ha de garantizar que los buffers de usuario sobre los que se hacen operaciones de entrada/salida estén en memoria principal durante toda la duración de la operación. En especial los marcos afectados no podrán ser objeto de paginación.

1.8. PROTECCIÓN

Como veremos mas adelante, una de las funciones del sistema operativo es la protección de unos usuarios contra otros: ni por malicia ni por descuido un usuario deberá acceder a la información de otro.

La protección hay que comprobarla en tiempo de ejecución, por lo que se ha de basar en mecanismos hardware. En esta sección se analizaran estos mecanismos para estudiar más adelante cómo los aplica el sistema operativo. Se analizara en primer lugar los mecanismos de protección que ofrece el procesador para pasar seguidamente a los mecanismos de protección de memoria.

1,8.1. Mecanismos de protección del procesador

Los mecanismos de protección del procesador se basan en los niveles de ejecución del mismo. En nivel de ejecución de núcleo se pueden ejecutar todas las instrucciones de maquina y se pueden acceder a todos los registros y a la totalidad de los mapas de memoria y de *E/S.* Sin embargo, en modo usuario se ejecuta un subconjunto de las instrucciones y se limitan los registros y los mapas accesibles.

Uno de los objetivos prioritarios de protección son los periféricos. En efecto, prohibiendo el acceso directo de los usuarios a los periféricos se impide que puedan acceder a la información

Digitalización realizada con propósito académico

almacenada por otros usuarios en esos periféricos. Por esta razón, toda la E/S es accesible solamente en nivel de núcleo, nivel que debe estar reservado al sistema operativo.

Para evitar que un programa de usuario pueda poner el procesador en nivel de núcleo, no existe ninguna instrucción máquina que realice este cambio (Advertencia 1.7). Sin embargo, existe la instrucción máquina inversa, que cambia de nivel de núcleo a nivel de usuario. Esta instrucción es utilizada por el sistema operativo antes de dejar que ejecute un programa de usuario.

ADVERTENCIA 1.7

Se puede argumentar que la instrucción TRAP realiza el cambio de nivel de ejecución de usuario a núcleo. Sin embargo, el objetivo de esta instrucción es producir una interrupción. Como efecto indirecto de la interrupción producida, se hace un cambio de nivel de ejecución y se deja de ejecutar el programa que incluye la instrucción de TRAP, pasándose a ejecutar el sistema operativo.

El mecanismo suministrado por el hardware para que el procesador pase a nivel de núcleo es la interrupción. En efecto, toda interrupción, ya sea interna o externa, tiene como efecto poner al procesador en nivel de núcleo. Siempre y cuando el sistema operativo se encargue de atender todas las interrupciones y de poner en nivel de usuario procesador antes de lanzar la ejecución de los programas de usuario, se garantiza que solamente sea el sistema operativo el que ejecute en nivel de núcleo.

1.8.2. Mecanismos de protección de memoria

Los mecanismos de protección de memoria deben evitar que un programa en ejecución direccione posiciones de memoria que no le hayan sido asignadas por el sistema operativo.

Una solución empleada en algunas máquinas que no tienen memoria virtual consiste en incluir una pareja de registros valla (limite y base), como los mostrados en la Figura 1.24. En esta solución se le asigna al programa una zona de memoria contigua. Todos los direccionamientos se calculan sumando el contenido del registro base, de forma que para el programa es como si estuviese cargado a partir de la posición «0» de memoria. Además, en cada acceso un circuito comparador comparara la dirección generada con el valor del registro limite. En caso de desbordamiento, el comparador genera una *excepción* de violación de memoria, para que el sistema operativo trate el tema, lo en la práctica, supone que parará el programa produciendo un volcado de memoria *(core dump* (Consejo de programación 1.1).

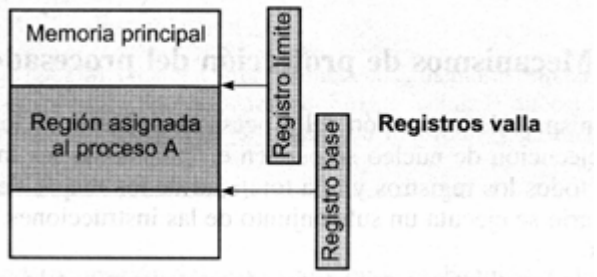


Figura 1.24. *Uso de registros vallo.*

Digitalización realizada con propósito académico

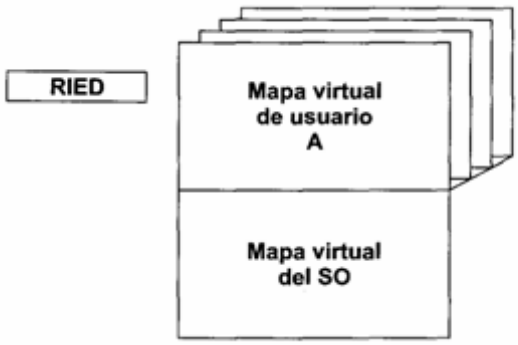


Figura 1,25. *División del mapa de memoria.*

CONSEJO DE PROGRAMACIÓN 1.1

Analizando el volcado de memoria mediante una adecuada herramienta de depuración el usuario puede determinar la razón del fallo y proceder a corregir el programa.

En los sistemas con memoria virtual existen dos mecanismos de protección de memoria. Por un lado, en nivel de usuario el procesador no permite acceder más que a una parte del mapa de memoria. Como muestra la Figura *1.25,* una solución es que en nivel de núcleo se pueda direccional todo el mapa de memoria virtual, mientras que en nivel de usuario solamente se pueda direccional una parte del mapa (por. ej.: las direcciones que empiezan por «0»). La MMU generará una *excepción* de violación de memoria en caso de que en nivel de usuario se genere una dirección no permitida (por. ej.: que empiece por «1»).

Además, una solución bastante frecuente consiste en dotar al procesador de un ***registro* RIED** (registro de identificador de espacio de direccionamiento). De esta forma se consigue que cada programa ejecución disponga de su propio espacio virtual, por lo que no puede acceder a tos clon espacios de memoria de los otros procesos.

El otro mecanismo se basa en la tabla de páginas y viene representado en la Figura 1.26. La tabla de páginas de cada programa en ejecución se selecciona mediante el RIED y especifica los

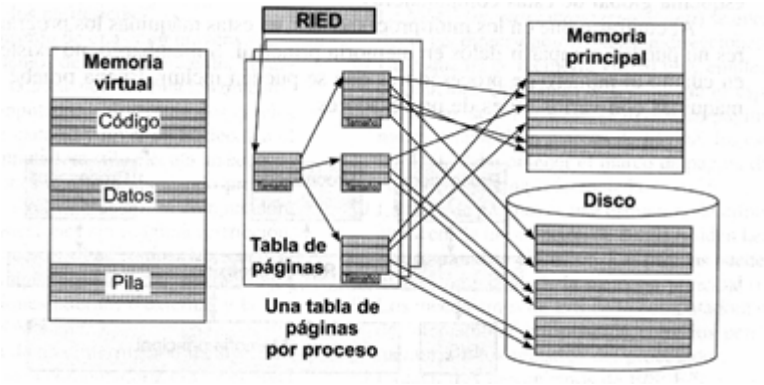


Figura 1.26. *La tabla de páginas como mecanismo de protección de memoria.*

Digitalización realizada con propósito académico

espacio. de memoria asignados por el sistema operativo a ese programa. La MMU, al mismo tiempo que realiza la traducción de cada dirección, comprueba que no se sobrepase el límite de ninguna de las tablas involucradas. En caso de sobrepasarse, generara una excepción de violación de memoria, para que el sistema operativo realice la acción correctora oportuna.

1.9. MULTJPROCESADOR Y MULTICOMPUTADORA

Dada la insaciable apetencia por máquinas de mayor potencia de proceso, cada vez es mas corriente encontrarse con computadoras que incluyen más de un procesador. Esta situación tiene una repercusión inmediata en el sistema operativo, que ha de ser capaz de explotar adecuadamente todos estos procesadores.
Las dos arquitecturas para estas computadoras son la de multiprocesador y la de multicomputadora, que se analizan seguidamente.

Multiprocesador

Como muestra la Figura 1.27, un multiprocesador es una máquina formada por un conjunto procesadores que comparten el acceso a una memoria principal común.
Cada procesador ejecuta su propio programa, debiendo todos ellos compartir la memoria principal común.
Una ventaja importante de esta solución es que el acceso a datos comunes por parte de varios programas es muy sencillo, puesto que utilizan la misma memoria principal. El mayor inconveniente es el limitado número de procesadores que se pueden incluir sin incurrir en el problema de saturar el ancho de banda de la memoria común (un límite típico es el de 16 procesadores).

Multicomputadora

La multicomputadora es una máquina compuesta por varios nodos, estando cada nodo formado un procesador, su memoria principal y, en su caso, elementos de E/S. La Figura 1.28 muestra esquema global de estas computadoras.
Al contrario que en los multiprocesadores, en estas máquinas los programas de do: procesadores no pueden compartir datos en memoria principal. Sin embargo, no existe la limitación anterior en cuanto al número de procesadores que e pueden incluir, Buena prueba de ello es que existen máquinas con varios miles de procesadores.

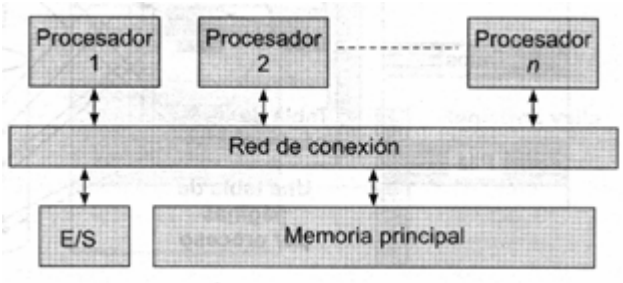


Figura 1.27. *Estructura de un multiprocesador.*

Digitalización realizada con propósito académico

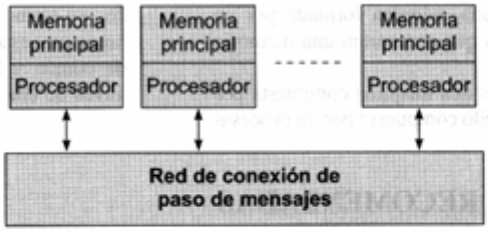


Figura 1.28. *Estructura de una multicomputadora.*

1.10. PUNTOS A RECORDAR

- La computadora es una máquina destinada a procesar datos mediante una serie de instrucciones máquina.
- Los cuatro componentes básicos de una computadora son: la memoria principal, la unidad aritmética, la unidad de control y la unidad de entrada/salida.
- La memoria principal almacena los datos a procesar, el programa máquina a ejecutar y los resultados.
- Se denomina programa máquina al conjunto de instrucciones máquina que tiene por objeto que la computadora realice una determinada función.
- La unidad aritmética se encarga de realizar una serie de operaciones aritméticas y lógicas sobre uno o dos operandos.
- La unidad de control se encarga de leer de memoria las instrucciones máquina que forman el programa a ejecutar, interpretar cada instrucción, leer los datos de memoria referenciados por cada instrucción, ejecutar cada instrucción y almacenar el resultado de cada instrucción.
- La unidad de entrada/salida (E/S) se encarga de realizar la transferencia de información entre la memoria principal (o los registros) y los periféricos.
- El modelo de programación de una computadora se caracteriza por: los elementos de almacenamiento, el juego de instrucciones y la secuencia de funcionamiento.
- La mayoría de las computadoras presentan dos niveles de ejecución: nivel de usuario y nivel de núcleo. En el nivel de usuario la computadora sólo ejecuta un subconjunto de las instrucciones máquina, quedando prohibidas las demás. En el nivel de núcleo, la computadora ejecuta todas sus instrucciones sin ninguna restricción.
- Los tres mecanismos que permiten romper la secuencia de ejecución de la computadora son: las instrucciones de salto, las interrupciones internas o externas y la instrucción máquina «TRAP».
- El ciclo de aceptación de una interrupción incluye: salvar algunos registros de la computadora, elevar el nivel de ejecución del procesador a modo núcleo y cargar un nuevo valor en el contador de programa para pasar a ejecutar otro programa.
- La memoria de la computadora se estructura como una jerarquía que utiliza memorias permanentes de alta capacidad y baja velocidad (como por ejemplo los discos) para almacenamiento permanente de la información, y memorias de semiconductores de tamaño reducido y alta velocidad para almacenar la información que se está utilizando en un momento determinado.
- Los parámetros característicos de una jerarquía de memoria son: la tasa de aciertos y el tiempo medio de acceso efectivo. La primera define la probabilidad de encontrar en un nivel de la jerarquía la información referenciada. La segunda mide el tiempo medio de acceso a la información.
- Utilizando memoria virtual, las direcciones generadas por el procesador se refieren a un espacio de direcciones virtual que puede residir en memoria principal o en disco. La memoria virtual exige la gestión automática de la parte de la jerarquía de memoria formada por la memoria principal y el disco.
- La MMU (*memory management unit*) se encarga de traducir las direcciones virtuales a direcciones físicas.
- Un esquema de memoria virtual muy empleado es el de paginación. Utilizando este esquema el espacio de direcciones de un proceso se divide en páginas y la memoria principal en marcos de página. En este esquema la MMU debe obtener el marco de página donde se encuentra la página referenciada.
- La tabla de páginas es una estructura de información que almacena la información de donde residen las páginas de un programa en ejecución. Las páginas pueden residir en marcos de página de la memoria principal o en disco.
- Los mecanismos de E/S de la computadora se encargan de intercambiar información entre los periféricos y la memoria o los registros del procesador.
- Existen tres mecanismos de E/S: E/S programada, E/S por interrupciones y DMA (acceso directo a memoria).

Digitalización realizada con propósito académico

- Un multiprocesador es una máquina formada por un conjunto de procesadores que comparten una memoria principal común.
- Una multicomputadora es una máquina compuesta por varios nodos, con cada nodo compuesto por un procesador, su memoria principal y elementos de E/S. Los distintos procesadores no comparten acceso a una memoria común y la comunicación entre ellos se realiza a través de una red de interconexión.

1.11. LECTURAS RECOMENDADAS

Para completar el contenido de este tema puede consultarse cualquiera de los siguientes libros:

[De Miguel, 1998] P. De Miguel. *Fundamentos de los Computadores.* Ed. Paraninfo, 1998.
[Patterson, 1994] D. Patterson, y D. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 1994.
[Stallings, 1996] W. Stallings. *Computer Organization and Architecture*, 4.ª edición, Prentice-Hall, 1996
[Tanenbaum, 1999] A. S. Tanenbaum. *Structured Computer Organization.* 4.ª edición, Prentice-Hall, 1999

1.12. EJERCICIOS

1.1. ¿Qué contiene una entrada de la tabla de vectores de interrupción?

a) El nombre de la rutina de tratamiento.
b) La dirección de la rutina de tratamiento.
c) El número de la interrupción.
d) El nombre de la tarea del SO que trata la interrupción.

1.2. ¿Cuál de las siguientes instrucciones máquina no debería ejecutarse en modo protegido? Razone su respuesta:

a) Inhibir interrupciones.
b) Instrucción TRAP.
c) Modificar el reloj del sistema.
d) Cambiar el mapa de memoria.

1.3. Considere un sistema con un espacio lógico de memoria de 128 K páginas con 8 KB cada una, una memoria física de 64 MB y direccionamiento a nivel de byte. ¿Cuántos bits hay en la dirección lógica?

1.4. Sea un sistema de memoria virtual paginada con direcciones lógicas de 32 bits que proporcionan un espacio virtual de 2^{20} páginas y con una memoria física de 32 MB. ¿Cuánto ocupará la tabla de marcos de página si cada entrada de la misma ocupa 32 bits?

1.5. Sea una computadora con memoria virtual y un tiempo de acceso a memoria de 70 ns. El tiempo necesario para tratar un fallo de página es de 9 ms. Si la tasa de aciertos a memoria principal es del 98 por 100, ¿cuál será el tiempo medio de acceso a una palabra en esta computadora?

Digitalización realizada con propósito académico

2

Introducción a los sistemas operativos

Un sistema operativo es un programa que tiene encomendadas una serie de funciones diferentes y cuyo objetivo es simplificar el manejo y la utilización de la computadora, haciéndolo seguro y eficiente. En este capítulo, prescindiendo de muchos detalles en aras a la claridad, se introducen varios conceptos sobre los sistemas operativos. Muchos de los temas cubiertos se plantearán con más detalle en los capítulos posteriores, dado el carácter introductorio empleado aquí. De forma más concreta, el objetivo del capítulo es presentar una visión globalizadora de los siguientes aspectos:

- *Definición y funciones del sistema operativo.*
- *Arranque de la computadora y del sistema operativo.*
- *Componentes y estructura del sistema operativo.*
- *Gestión de procesos.*
- *Gestión de memoria.*
- *Gestión de la E/S.*
- *Gestión de archivos y directorios.*
- *Comunicación y sincronización entre procesos.*
- *Seguridad y protección.*
- *Activación del sistema operativo y llamadas al sistema.*
- *Interfaz de usuario del sistema operativo e interfaz del programador.*
- *Historia y evolución de los sistemas operativos.*

2.1. ¿QUÉ ES UN SISTEMA OPERATIVO?

En esta sección se plantea en primer lugar el concepto de maquina desnuda para pasar acto seguido a introducir el concepto de sistema operativo y sus principales funciones.

2.1.1. Maquina desnuda

El término de maquina desnuda se aplica a una computadora carente de sistema operativo. El término es interesante porque resalta el hecho de que una computadora en sí misma no hace nada. Como se vio en el capítulo anterior, una computadora solamente es capaz de repetir a alta velocidad la secuencia de: lectura de instrucción máquina, incremento del PC y ejecución de la instrucción leída.

Para que la computadora realice un función determinada ha de tener en memoria principal un programa maquina específico p a realizar dicha función y ha de conseguirse que el registro PC contenga la dirección de comienzo del programa.

La misión del sistema operativo, como se verá en la sección siguiente, es completar (vestir) a la máquina mediante una serie de programas que permitan su cómodo manejo y utilización.

2.1.2. Funciones del sistema operativo

Un *sistema operativo* (SO) es un programa que tiene encomendadas una serie de funciones diferentes cuyo objetivo es simplificar el manejo y la utilización de la computadora, haciéndolo seguro y eficiente. Históricamente se han ido completando las misiones encomendadas al sistema operativo, por lo que lo. productos comerciales actuales incluyen una gran cantidad de funciones, como son interfaces gráficas, protocolos de comunicación, etc. (Recordatorio 2.1).

RECORDATORIO 2.1

Buena muestra de esta tendencia es la inclusión por parte de Microsoft de su navegador Web dentro de sus sistemas operativos.

Las funciones clásica, del sistema operativo se pueden agrupar en las tres categorías siguientes:

- Gestión de os recursos de la computadora.
- Ejecución de servicios para lo: programas.
- Ejecución de los mandato: de los usuarios.

Como muestra la Figura 2.1, el sistema operativo esta formado conceptualmente por tres capas principales. La capa m : cercana al hardware se denomina **núcleo *(kernel)*** y es la que gestiona los recursos hardware del sistema y la que suministra otra la funcionalidad básica del sistema operativo. Esta capa ha de ejecutar en nivel núcleo, mientras que las otras pueden ejecutar en niveles menos permisivos.

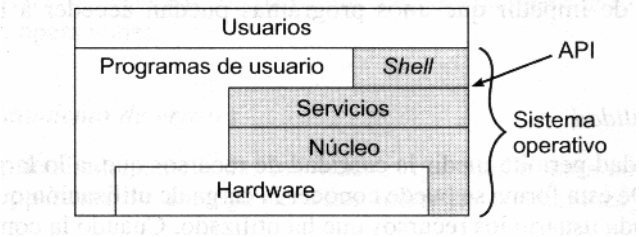


Figura 2.1. *Niveles del sistema operativo.*

La capa de servicios o **llamadas al sistema** ofrece a los programas unos servicios en forma de una interfaz de programación o API *(application programming interface).* Desde el punto de vista de los programas, esta capa extiende la funcionalidad de la computadora, por lo que se suele decir que el sistema operativo ofrece una máquina virtual extendida a los programas. De esta forma se facilita la elaboración de éstos, puesto que se apoyan en las funciones que le suministra el sistema operativo.

La capa de **intérprete de mandatos o *shell*** suministra una interfaz a través de la cual el usuario puede dialogar de forma interactiva con la computadora. El *shell* recibe los mandatos u órdenes del usuario, los interpreta y, si puede, los ejecuta. Dado que el *shell* suele ejecutar en nivel de usuario, algunos autores consideran que no forma p te del sistema operativo. En opinión de los autores es uno mas de los elementos del mismo.

Seguidamente, se analizarán cada una de estas tres facetas del sistema operativo.

El sistema operativo como gestor de recursos

En una computadora actual suelen coexistir varios programas, del mismo o de varios usuarios, ejecutándose simultáneamente. Estos programas compiten por los recursos de la computadora, siendo el sistema operativo el encargado de arbitrar su asignación y uso. Como complemento a la gestión de recursos, el sistema operativo ha de garantizar la protección de unos programas frente a otros y ha de suministrar información sobre el uso que se hace de los recursos.

a) *Asignación de recursos*

El sistema operativo se encarga de asignar los recursos a los programas en ejecución. Para ello, ha de mantener unas estructuras que le permitan saber que recursos están libres y cuáles están asignados a cada programa. La asignación de recursos se realiza según la disponibilidad de lo. mismos y la prioridad de los programas, debiéndose resolver los conflictos que aparecen por las peticiones simultáneas.

Especial mención reviste la recuperación de los recursos cuando los programas ya no los necesitan. Una mala recuperación de recursos puede hacer que el sistema operativo considere, por ejemplo, que ya no le queda memoria disponible cuando, en realidad, sí la tiene. La recuperación se puede hacer porque el programa que tiene asignado el recurso le comunica al sistema operativo que ya no lo necesita o bien porque el programa haya terminado.

Los recursos manejados por el sistema operativo son físicos y lógicos. Entre los físicos se encuentra el procesador, la memoria principal y los periféricos. Entre los lógicos se pueden citar los archivos y los puertos de comunicación.

b) *Protección*

El sistema operativo ha de garantizar la protección entre los usuarios del sistema. Ha de asegurar la confidencialidad de la información y que unos trabajos no interfieran con otros. Para conseguir este

objetivo ha de impedir que unos programas puedan acceder a los recursos asignados a otros programas.

c) *Contabilidad*

La contabilidad permite medir la cantidad de recursos que, a lo largo de su ejecución, utiliza cada programa. De esta f6rma se puede conocer la carga de utilización que tiene cada recurso y se puede imputar a cada usuario los recursos que ha utilizado. Cuando la contabilidad se emplea meramente para conocer la carga de los componentes del sistema se suele denominar monitorización.

El sistema operativo como máquina extendida

El sistema operativo ofrece a los programas un conjunto de servicios, o **llamadas al sistema**, que pueden solicitar cuando lo necesiten, proporcionando a los programas una visión de máquina extendida. El modelo de programación que ofrece el hardware e se complementa con estos servicios software, que permiten ejecutar de forma cómoda y protegida ciertas operaciones. La alternativa consistiría en complicar los programas de usuario y en no tener protección frente a o os usuarios.

Los servicios se pueden agrupar en las cuatro clases siguientes: ejecución de programas, operaciones de E/S, operaciones sobre archivos y detección y tratamiento de errores. Gran parte e de este texto se dedicará a explicar los servicios ofrecidos por los sistemas operativos, por lo que aquí nos limitaremos a hacer unos simples comentarios sobre cada una de sus cuatro clases.

d) *Ejecución de programas*

El sistema operativo incluye servicios para lanzar la ejecución de un programa, así como para pararla o abortarla. También existen servicios p a conocer y modificar las condiciones de ejecución de los programa, para comunicar y sincronizar unos programas con otros.

La ejecución de programas da lugar al concepto de **proceso**. Un proceso se puede definir como un programa en ejecución. El proceso es un concepto fundamental en los sistemas operativos, puesto que el objetivo ultimo de éstos es crear, ejecutar y destruir procesos, de acuerdo a las órdenes de los usuarios.

Para que un programa pueda convertirse en un proceso ha de estar traducido a código máquina y almacenado en un dispositivo de almacenamiento como el disco. Bajo la petición de un usuario, el sistema operativo creará un proceso para ejecutar el programa. Observe que varios procesos pueden estar ejecutando el mismo programa. Por ejemplo, varios usuarios pueden haber pedido al operativo la ejecución del mismo programa editor.

b) *Órdenes de E/S*

Los servicios de E/S ofrecen una gran comodidad y protección al proveer a los programas de operaciones de lectura, escritura y modificación del estado de los periféricos. En efecto, la programación de las operaciones de E/S es muy compleja y dependiente del hardware específico de cada periférico. Los servicios del si tema operativo ofrecen un alto nivel de abstracción de forma que el programador de aplicaciones no tenga que preocuparse de esos detalles.

e) *Operaciones sobre archivos*

Los archivos ofrecen un nivel de abstracción mayor que el de las órdenes de E/S, permitiendo operaciones tales como creación, borrado, renombrado, apertura, escritura y lectura de chivos.

Observe que muchos de los servicios son parecidos a las operaciones de E/S y terminan concretándose en este tipo de operaciones.

d) *Detección y tratamiento de errores*

Además de analizar detalladamente todas las ordenes que recibe, para comprobar que se pueden realizar , el sistema operativo se encarga de tratar todas las condiciones de error que detecte el hardware.

Entre las condiciones de error que pueden aparecer destacaremos las siguientes: errores en las operaciones de E/S, errores de paridad en los accesos a memoria o en los buses y errores de ejecución en los programas, como desbordamientos, violaciones de memoria, códigos de instrucción prohibidos, etc.

El sistema operativo como interfaz de usuario

El módulo del sistema operativo que permite que los usuarios dialoguen de forma interactiva con el sistema es el intérprete de mandatos o *shell.*

El *shell* se comporta como un bucle infinito que está repitiendo constantemente la siguiente secuencia:

- Espera una orden del usuario. En el caso de interfaz textual, el *shell* está pendiente de lo que escribe el usuario en la línea de mandatos. En las interfaces gráficas está pendiente de los eventos del apuntador (ratón) que manipula el usuario, además, de los del teclado.
- Analiza la orden y, en caso de ser correcta, la ejecuta, para lo cual emplea los servicios del sistema operativo.
- Concluida la orden vuelve a la espera.

El dialogo mediante interfaz textual exige que el usuario memorice la sintaxis de los mandatos, con la agravante de que son distintos para cada sistema operativo (p. ej.: para listar el contenido de un chivo en MS-DOS e emplea el mandato **type,** pero en UNIX se usa el mandato **cat**). Por esta razón cada vez son más populares los intérpretes de mandatos con interfaz gráfica, como el de Windows NT.

Archivos de mandatos

Casi todos los intérpretes de mandatos pueden ejecutar archivos de mandatos, llamados *shell scripts.* Estos chivos incluyen varios mandatos totalmente equivalentes a los mandatos que se introducen en el terminal. Además, para realizar funciones complejas, pueden incluir mandatos especiales de control del flujo de ejecución, como puede ser el **goto**, el **for** o el **if**, así como etiquetas para identificar líneas de mandatos,

Para ejecutar un archivo de mandatos basta con invocarlo de igual forma que un mandato estándar del intérprete de mandatos.

2.1.3. Concepto de usuario y de grupo de usuarios

Un usuario es una persona autorizada para utilizar un sistema informático. El usuario se autentica mediante su nombre de cuenta y su contraseña o *password.*

En realidad, el sistema operativo no asocia el concepto de usuario con el de persona física sino con un nombre de cuenta. Una persona puede tener más de una cuenta y una cuenta puede ser utilizada por más de una persona. Internamente, el sistema operativo asigna a cada usuario (cuenta)un identificador «uid» *(user identifier)* y un perfil.

El sistema de seguridad de los sistemas operativos está basado en la entidad usuario. Cada usuario tiene asociados unos derechos, que definen las operaciones que le son permitidas.

Existe un usuario privilegiado, denominado **súperusuario** o **administrador**, que no tiene ninguna restricción, es decir, que puede hacer todas las operaciones sin ninguna traba. La figura súperusuario es necesaria para poder administrar el sistema (Recomendación 2.1),

RECOMENDACIÓN 2.1

La figura del superusuario entraña no pocos riesgos, por su capacidad de acción. Es por tanto muy importante que la persona o personas que estén autorizadas para utilizar una cuenta de superusuario sean de toda confianza y que las claves utilizadas sean difíciles de adivinar. Además, como una buena norma de administración de sistemas, siempre se deberá utilizar la cuenta con los menores derechos posibles que permiten realizar la función deseada. De esta forma se minimiza la posibilidad de cometer errores irreparables.

grupo.

2.2. ARRANQUE DE LA COMPUTADORA

El arranque de una computadora actual tiene dos fases: la fase de arranque hardware y la fase arranque del sistema operativo. La Figura 2.2 resume las actividades más importantes que se realizan en el arranque de la computadora.

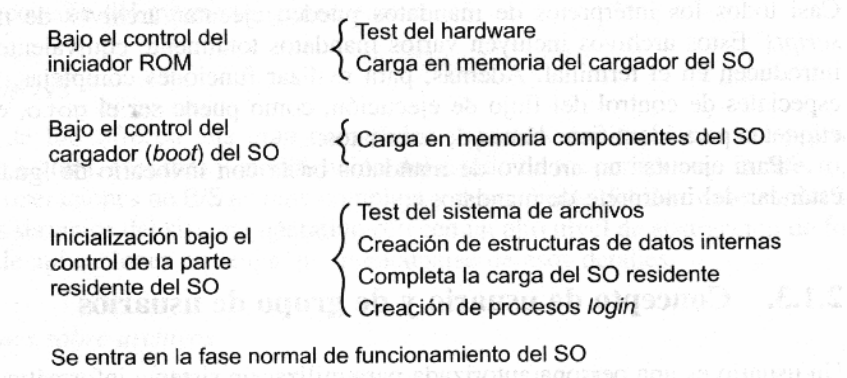


Figura 2.2. *Operaciones realizadas en el arranque de la computadora.*

Arranque hardware

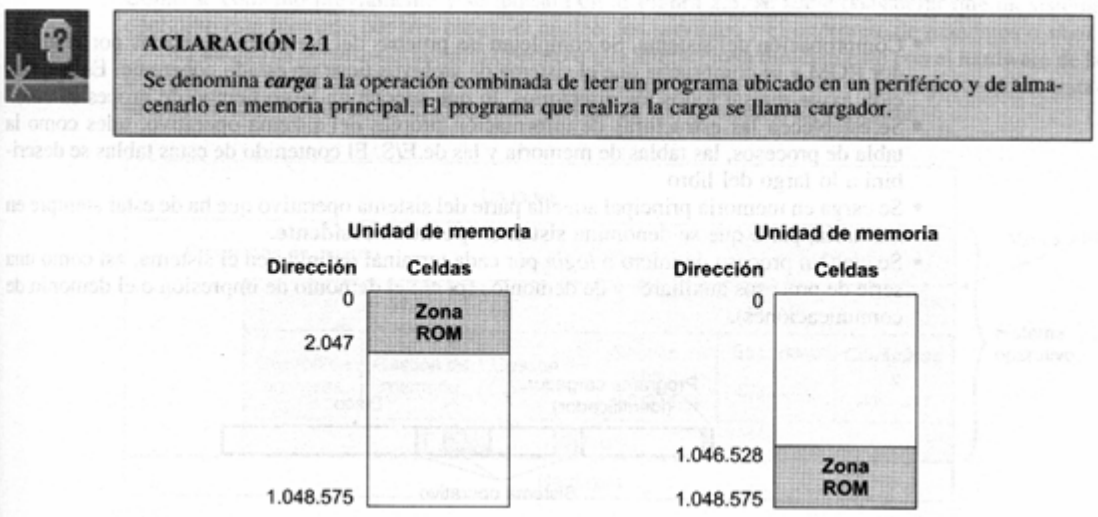
Como se ha indicado con anterioridad, la computadora solamente es capaz de realizar actividades útiles si cuenta con el correspondiente programa cargado en memoria principal. Ahora bien, la memoria principal de las computadoras es volátil, lo que significa que, cuando se enciende la máquina, no contiene ninguna información válida, Por tanto, al arrancar la computadora no es capaz de realizar nada,

Para resolver esta situación, las computadoras antiguas tenían una serie de conmutadores que permitían introducir una a una palabras en la memoria principal y en los registros. El usuario debía introducir a mano, y en binario, un primer programa que permitiese cargar otros programas almacenados en algún soporte, como la cinta de papel.

En la actualidad, la solución empleada es mucho más cómoda puesto que se basa en un programa permanente grabado en una memoria ROM. En efecto, como muestra la Figura 2.3, una parte del mapa de memoria está construido con memoria ROM no volátil. En esta memoria ROM se encuentra a un programa de arranque, que esta siempre disponible, puesto que la ROM no pierde su contenido. Llamemos iniciador ROM a este programa.

Cuando se arranca la computadora, o cuando se pulsa el botón de RESET, se genera una señal eléctrica que carga uno. valores predefinidos en los registros. En especial, esta señal carga en el contador de programa la dirección de comienzo del iniciador ROM. De esta forma se cumplen todas las condiciones para que la computadora ejecute un programa y realice funciones útiles.

El iniciador ROM realiza tres funciones, Primero hace una comprobación del sistema, que para sirve para detectar sus características (p. ej.; la cantidad de memoria principal disponible o los periféricos instalados) y comprobar si el conjunto funciona correctamente, Una vez pasada la como son probación, entra en la fase de lectura y almacenamiento en memoria del programa cargador del sistema operativo (Ablación 2.1). Finalmente da control a este programa, bifurcando a la dirección de memoria en la que l ha almacenado, Para tener una mayor flexibilidad se hace que el programa iniciador ROM sea independiente del sistema operativo.



ACLARACIÓN 2.1

Se denomina ***carga*** a la operación combinada de leer un programa ubicado en un periférico y de almacenarlo en memoria principal. El programa que realiza la carga se llama cargador.

Figura 2.3. *Una parte del mapa de memoria está construido con memoria*

En el caso de una computadora de tipo PC, la memoria ROM contiene, además del programa iniciador, software de E/S denominado BIOS *(basic input-output system).* La BIOS de una computadora la proporciona el fabricante y suele contener procedimientos para leer y escribir de leer caracteres del teclado y escribir en la pantalla.

Ubicación del sistema operativo

El sistema operativo se encuentra almacenado en una unidad de disco, tal y como muestra Figura 2.4. Hay una parte del mismo en la que estamos especialmente interesados ahora: se trata del programa cargador del sistema operativo o *boot* del sistema operativo. Este programa está almacenado en una zona predefinida del disco (p. ej.: los cuatro primeros sectores del disco) y tamaño prefijado.

Como se indicó anteriormente, el iniciador ROM trae a memoria principal el programa cargado del sistema operativo. El programa iniciador ROM y el sistema operativo tienen un convenio sobre la ubicación, dirección de arranque y tamaño del cargador del sistema operativo. Obsérvese que el iniciador ROM es independiente del sistema operativo siempre que este cumpla con el convenio anterior, por lo que la máquina podrá soportar diversos sistemas operativos.

Para una mayor seguridad, el programa cargador del sistema operativo incluye, en una posición prefijada por el iniciador ROM, una contraseña (palabra mágica). De esta forma, el iniciador ROM puede verificar que la información contenida en la zona prefijada contiene efectivamente programa cargador de un sistema operativo.

Arranque del sistema operativo

El programa cargador del sistema operativo tiene por misión traer a memoria principal algunos los componentes del sistema operativo. Una vez cargados estos componentes, se pasa a la fase iniciación, que incluye las siguientes operaciones:

- Comprobación del sistema. Se completan las pruebas del hardware realizadas por dar ROM y se comprueba que el sistema de archivos tiene un estado coherente. Esta operación exige revisar todos los directorios, lo que supone un largo tiempo de procesamiento.
- Se establecen las estructuras de información propias del sistema operativo, tales como tabla de procesos, las tablas de memoria y las de E/S. El contenido de estas tablas se describirá a lo largo del libro,
- Se carga en memoria principal aquella parte del sistema operativo que ha de estar siempre memoria, parte que se denomina **sistema operativo residente**.
- Se crea un proceso de inicio o *login* por cada terminal definido en el sistema, así como una serie de procesos auxiliares y de demonios (p. ej.; el demonio de impresión o el demonio comunicaciones).

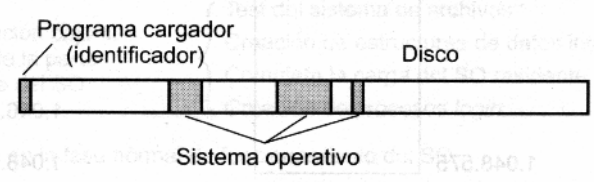


Figura 2.4. *El sistema operativo se encuentra almacenado en una unidad de disco.*

Los procesos de inicio presentan en su terminal el mensaje de bienvenida y se quedan a la espera de que un usuario a arranque una sesión, para lo cual ha de teclear el nombre de su cuenta y su contraseña o *password.* El proceso de inicio **autentica** al usuario, comprobando que los datos introducidos son correctos y lanza un proceso *shell* (Aclaración 2.2). El proceso *shell* primero ejecuta uno o varios archivos de mandatos, como es el «autoexec.bat» en MS-DOS o los «.login» y **«**.cshrc» en UNIX. A continuación, el *shell* se queda esperando órdenes de los usuarios, ya sean textuales o como acciones sobre un menú o un icono. Para llevar a cabo las operaciones solicitadas por el usuario, el *shell* genera uno o varios procesos.

ACLARACIÓN 2.2

En algunos de los sistemas operativos monousuario utilizados en las computadoras personales no hay fase de login, creándose directamente el proceso shell para atender al usuario.

2.3. COMPONENTES Y ESTRUCTURA DEL SISTEMA OPERATIVO

El sistema operativo está formado por una serie de componentes especializados en determinadas funciones. Cada sistema operativo estructura estos componentes de forma distinta. En esta sección se describen en primer lugar los distintos componentes que conforman un sistema operativo, para pasar a continuación a ver las distintas formas que tienen los sistemas operativos de estructurar estos componentes.

2.3.1. Componentes del sistema operativo

Como se comentó previamente y se muestra en la Figura *2.5,* se suele considerar que un sistema operativo está formado por tres capas: el núcleo, los servicios y el intérprete de mandatos o *shell.*

El núcleo es la parte del sistema operativo que interacciona directamente con el hardware de la máquina. Las funciones del núcleo se centran en la gestión de recursos, como el procesador, tratamiento de interrupciones y las funciones básicas de manipulación de memoria,

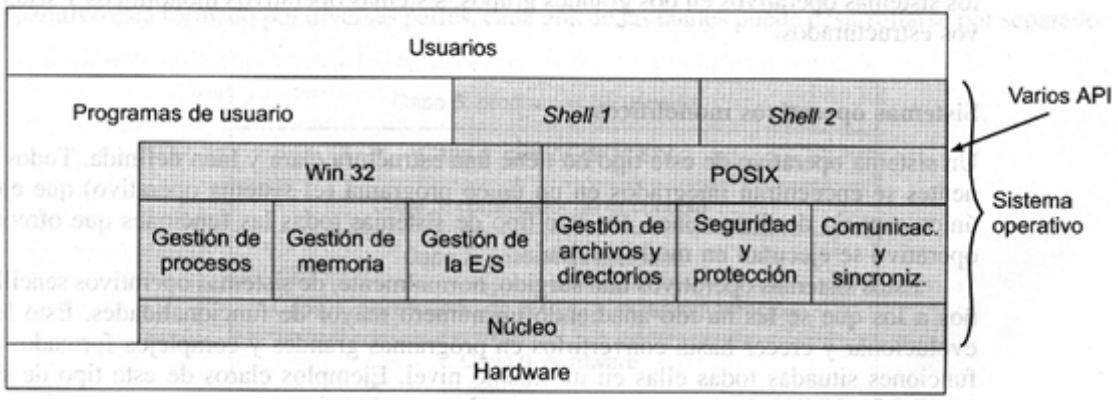


Figura 2.5. *Componentes del sistema operativo.*

Los servicios se suelen agruparse según su funcionalidad en varios componentes, cada uno de cuales se ocupa de las siguientes funciones:

- Gestión de procesos. Encargada de la creación, planificación y destrucción de procesos.
- Gestión de memoria. Componente encargada de saber qué partes de memoria están libres y cuáles ocupadas, así como de la asignación y liberación de memoria según la necesiten los procesos.
- Gestión de la *EIS.* Se ocupa de facilitar el manejo de los dispositivos periféricos.
- Gestión de archivos y directorios. Se encarga del manejo de archivos y directorios y de 1a administración del almacenamiento secundario,
- Comunicación y sincronización en e procesos. Encargada de ofrecer mecanismos los procesos puedan comunicase y sincronizarse.
- Seguridad y protección. Este componente debe encargarse de garantizar la usuarios y de definir lo que pueden hacer cada uno de ellos con los recursos del sistema.

Todos estos componentes ofrecen una serie de servicios a través de una interfaz de llamadas sistema. Como se muestra en la Figura *2.5,* un sistema operativo puede incluir más de una interfaz de servicios (en la figura se han considerado las interfaces Win32 y POSIX, interfaces que ser
descritas a lo largo del presente libro). En este caso, los programas podrán elegir sobre qué interfaz quieren ejecutar, pero no podrán mezclar servicios de varias interfaces. Se dice, en este caso, que sistema operativo presenta al usuario varias máquinas virtuales.

De igual forma, el sistema operativo puede incluir varios intérpretes de mandatos, unos textuales y otros gráficos, pudiendo el usuario elegir el que mas le interese. Sin embargo, hay que observar que no se podrán mezclar mandatos de varios intérpretes.

En las secciones siguientes de este capitulo se van a describir, de forma muy breve, cada uno de los componentes anteriores, pero antes se van a describir las distintas formas que tienen los sistemas operativos de estructurar dichos componente.

2.3.2. Estructura del sistema operativo

Un sistema operativo es un programa grande y complejo que está compuesto, como se ha visto en la sección anterior, por una serie de componentes con funciones bien definidas. Cada sistema operativo estructura estos componentes de distinta forma. En función de esta estructura se pueden agrupar los sistemas operativos en dos grandes grupos: sistemas operativos monolíticos y sistemas operativos estructurados.

Sistemas operativos monolíticos

Un sistema operativo de este tipo no tiene una estructura clara y bien definida. Todos sus componentes se encuentran integrados en un único programa (el sistema operativo) que ejecuta en un único espacio de direcciones. En este tipo de sistemas todas las funciones que ofrece el sistema operativo se ejecuta en un modo núcleo.

Estos sistemas operativos han surgido, normalmente, de sistemas operativos sencillos y pequeños a los que se les ha ido añadiendo un número mayor de funcionalidades. Esto les ha hecho evolucionar y crece hasta convertirlos en programas grandes y complejos formados por muchas funciones situadas todas ellas en un mismo nivel. Ejemplos claros de este tipo de sistemas son MS-DOS y UNIX. Ambos comenzaron siendo pequeños sistemas operativos, que fueron haciéndose cada vez mas grandes debido a la gran popularidad que adquirieron.

El problema que plantean este tipo de sistemas radica en lo complicado que es modificar el sistema operativo para añadir nuevas funcionalidades y servicios. En efecto, añadir una nueva característica al sistema operativo implica la modificación de un gran programa, compuesto por miles de líneas de código fuente y funciones, cada una de las cuales puede invocar a otras cuando así lo requiera. Además, en este tipo de sistemas no se sigue el principio de ocultación de la información. Para solucionar este problema es necesario dotar de cierta estructura al sistema operativo.

Sistemas operativos estructurados

Cuando se quiere dotar de estructura a un sistema operativo, normalmente se recurre a dos tipos de soluciones: sistemas por capas y sistemas cliente-servidor.

a) *Sistemas por capa.*

En un sistema por capas, el sistema operativo se organiza como una jerarquía de capas, donde cada capa ofrece una interfaz clara y bien definida a la capa superior y solamente utiliza los servicios que le ofrece la capa inferior.

La principal ventaja que ofrece este tipo de estructuras es la modularidad y la ocultación de la información. Una capa no necesita conocer como se ha implementado la capa sobre la que se construye, únicamente necesita conocer la interfaz que ofrece. Esto facilita enormemente la depuración y verificación del sistema, puesto que las capas se pueden ir construyendo y depurando por separado.

Este enfoque lo utilizo por primera vez el sistema operativo THE [Dijkstra, 1968], un sistema operativo sencillo que estaba formado por seis capas, como se muestra en la Figura 2.6. Otro ejemplo de sistema operativo diseñado por capas es el OS/2 [Deitel, 1994], descendiente de MS-DOS.

b) *Modelo cliente-servidor*

En este tipo de modelo, el enfoque consiste en implementar la mayor parte de los servicios y funciones del sistema operativo en procesos de usuario, dejando solo una pequeña parte del sistema operativo ejecutando en modo núcleo. A esta parte se le denomina *micronúcleo* y a los procesos que ejecutan el resto de funciones se les denomina *servidores.* La Figura 2.7 presenta la estructura de un sistema operativo con estructura cliente-servidor, Como puede apreciarse en la figura, el sistema operativo está formado por diversas partes, cada una de las cuales puede desarrollarse por separado.

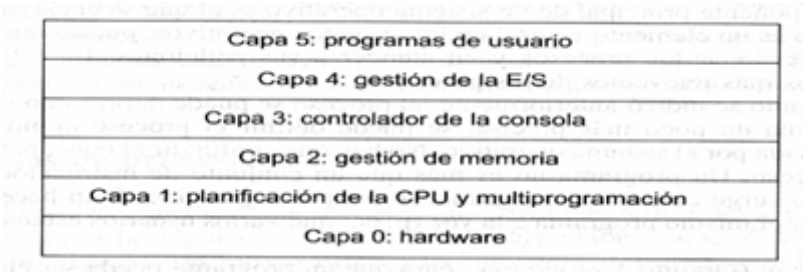


Figura 2.6. *Estructura por capa. del sistema operativo THE,*

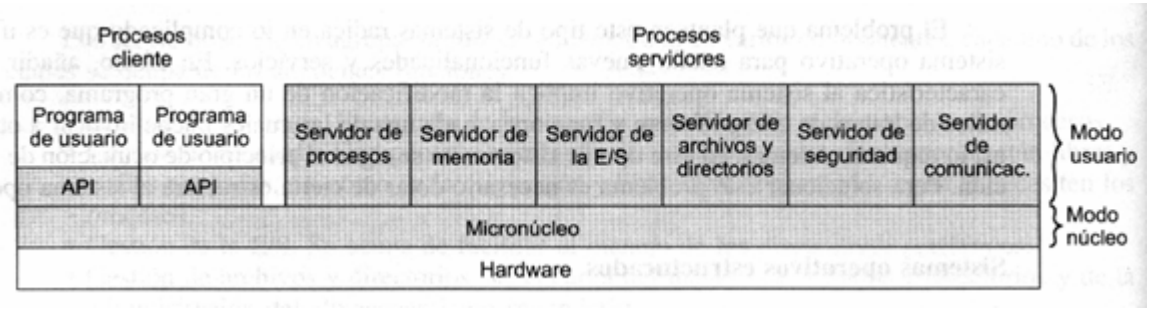


Figura 2.7. *Estructura cliente-servidor en un sistema operativo.*

No hay una definición clara de las funciones que debe llevar a cabo un micronúcleo. La mayoría incluyen la gestión de interrupciones, gestión básica de procesos y de memoria y servicios básicos de comunicación entre procesos. Para solicitar un servicio en este tipo de sistemas, como por ejemplo crear un proceso, el proceso de usuario (proceso denominado cliente) solicita el servicio al servidor del sistema operativo correspondiente, en este caso al servidor de procesos. A su vez, el proceso servidor puede requerir los servicios de otros servidores, como es el caso del servidor de memoria. En este caso, el servidor de procesos se convierte en cliente del servidor de memoria.

La ventaja de este modelo es la gran flexibilidad que presenta. Cada proceso servidor sólo ocupa de una funcionalidad concreta, lo que hace que cada parte pueda ser pequeña y
Esto a su vez facilita el desarrollo y depuración de cada uno de los procesos servidores.

En cuanto a las desventajas, citar que estos sistemas presentan una mayor sobrecarga en el tratamiento de los servicios que los sistemas monolíticos. Esto se debe a que los distintos componentes de un sistema operativo de este tipo ejecutan en espacios de direcciones distintos, lo que hace que su activación requiera más tiempo.

Minix [Tanenbaum, 1998], Mach [Accetta, 1986] y Amoeba [Mullender, 1990] son ejemplos de sistemas operativos que siguen este modelo. Windows NT también sigue esta filosofía de diseño, aunque muchos de los servidores (el gestor de procesos, gestor de E/S, gestor de memoria, etc.) se ejecutan en modo núcleo por razones de eficiencia.

2.4. GESTIÓN DE PROCESOS

El componente principal de un sistema operativo es el que se encarga de la gestión de procesos. El proceso es un elemento central en los sistemas operativos, puesto que su función consiste en generar y gestionar los procesos y en atender a sus peticiones. En esta sección se introducirán los aspectos más relevantes de los procesos.

Como se indicó anteriormente, el proceso se puede definir como un **programa en ejecución.** De forma un poco más precisa, se puede definir el proceso como la unidad de procesamiento gestionada por el sistema operativo. No hay que confundir el concepto de programa con el concepto de proceso. Un programa no es más que un conjunto de instrucciones máquina, mientras que el proceso surge cuando un programa se pone en ejecución. Esto hace que varios procesos puedan ejecutar el mismo programa a la vez (p. Ej.: que varios usuarios estén ejecutando el mismo editor textos).

En el Capítulo 1 se vio que, para que un programa pueda ser ejecutado, ha de residir con sus datos en memoria principal, tal y como muestra la Figura 2.8. Al contenido de los segmentos de

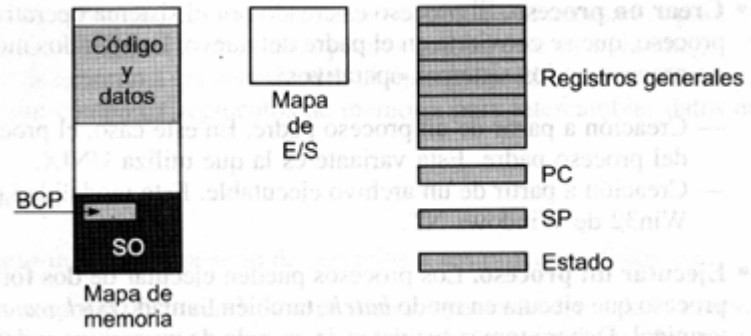


Figura 2.8. *Elementos que constituyen un proceso.*

memoria en los que reside el código y los datos del proceso se le denomina **imagen de memoria.** Observe que, durante su ejecución, el proceso va modificando los registros del modelo de programación de la computadora, de acuerdo a las instrucciones de maquinas involucradas. El contenido de los registros del modelo de programación es lo que se conoce como **estado del procesador.**

El sistema operativo mantiene por cada proceso una serie de estructuras de información que permite identificar las características de éste así como los recursos que tiene asignados. Una parte muy importante de esta estructura es el **bloque de control del proceso** (BCP) que, como se verá en el Capítulo 3, incluye, entre otra información, el estado de los registros del proceso, cuando éste no está ejecutando. El sistema operativo debe encargarse también de ofrecer una serie de servicios para la gestión de procesos y de gestionar los posibles interbloqueos que surgen cuando los procesos acceden a diferentes recursos.

Dependiendo del número de procesos y de usuarios que puedan ejecutar simultáneamente, un sistema operativo puede ser:

- Monotarea, también llamado monoproceso. Este tipo de sistemas operativos sólo permite que exista un proceso en cada instante.
- Multitarea o multiproceso. Permite que coexistan varios procesos activos a la vez. El sistema operativo se encarga de ir repartiendo el tiempo del procesador entre estos procesos.
- Monousuario. Está previsto para soportar a un solo usuario,
- Multiusuario. Soporta varios usuarios trabajando simultáneamente desde varios terminales. A su ve cada usuario puede tener activo más de un proceso, por lo que el sistema, obligatoriamente, ha de ser multitarea. Los sistemas multiusuario reciben también el nombre de tiempo compartido, porque el sistema operativo ha de repartir el tiempo de la computadora entre los usuarios para que las tareas de todos ellos avancen de forma razonable.

En el Capítulo 3 se estudiarán con detalle la gestión de procesos, los servicios ofrecidos para la gestión de procesos y las técnicas básicas de implementación de procesos. En el Capítulo 6 se estudiará el concepto de ínterbloqueo y los mecanismos para manejarlo.

2.4.1. Servicios de procesos

El sistema operativo ofrece una serie de servicios que permiten definir la vida de un proceso. Esta vida está constituida por las siguientes fases; creación, ejecución y muerte del proceso.

En general, los sistemas operativos ofrecen los siguientes servicios para la gestión de procesos:

- **Crear un proceso.** El proceso es creado por el sistema operativo cuando así lo solicita otro proceso, que se convierte en el padre del nuevo. Existen dos modalidades básicas para crear un proceso en los sistemas operativos;

  — Creación a partir de un proceso padre. En este caso, el proceso hijo es una copia exacta del proceso padre. Esta variante es la que utiliza UNIX.
  — Creación a partir de un archivo ejecutable. Esta modalidad es la que se define en el API Win32 de Windows NT.

- **Ejecutar un proceso.** Los procesos pueden ejecutar de dos formas; *batch* e interactiva. Un proceso que ejecuta en modo *batch,* también llamado *background,* no está asociado a ningún terminal. Deberá tomar sus datos de entrada de un archivo y deberá depositar sus resultados en otro archivo. Un ejemplo típico de un proceso *batch* es un proceso de nóminas, que parte del archivo de empleados y del chivo de los partes de trabajo y genera un archivo de órdenes básicas para pagar las nóminas.

  Por el contrario, un proceso que ejecuta en modo interactivo está asociado a un terminal, por el que recibe la información del usuario y por el q e contesta con los resultados. Un ejemplo típico de un proceso interactivo es un proceso de edición.
- **Terminar la ejecución de un proceso**. Un proceso puede finalizar su ejecución por varias causas, entre las que se encuentran las siguientes:

  — Ha terminado de ejecutar el programa.
  — Se produce una condición de error en su ejecución (p. ej.; división por O o violación de memoria).
  — Otro proceso o el usuario deciden que ha de terminar.

- **Cambiar el programa de un proceso**. Algunos sistemas operativos incluyen, además de los anteriores, un servicio que cambia el programa que está ejecutando un proceso por otro programa almacenado en disco, Observe que esta operación no consiste en crear un nuevo proceso que ejecuta ese nuevo programa. Se trata de eliminar el programa que está ejecutando el proceso y d incluir un nuevo programa que se trae del disco.

2.5. GESTIÓN DE MEMORIA

El gestor de memoria es uno de los componentes principales del sistema operativo. Su actividad se centra fundamentalmente en la categoría de gestión de recursos, puesto que tiene por objetivo casi exclusivo la gestión del recurso memoria. En este sentido se encarga de:

- Asignar memoria a los procesos para crear su imagen de memoria.
- Proporcionar memoria a los procesos cuando la soliciten y liberarla cuando así lo requieran.
- Tratar los posibles e ores de acceso a memoria, evitando que unos procesos interfieran en la memoria de otros.
- Permitir que los procesos puedan compartir memoria entre ellos. De esta forma los procesos podrán comunicarse entre ellos.
- Gestiona la jerarquía de memoria y tratar los fallos de página en los sistemas con memoria virtual,

Además de estas funciones, el gestor de memoria, en la categoría de servicios a los programas, suministra los siguientes servicios: el de solicitar memoria, el de liberarla y el de permitir que los

procesos compartan memoria. Los dos primeros servicios son necesarios para los programas que requieren asignación dinámica de memoria, puesto que, en este caso, la imagen de memoria ha de crecer o decrecer de acuerdo a las necesidades de ejecución. El tercer servicio es necesario cuando los procesos desean comp tir segmentos de memoria p a intercambi datos entre sí.

2.5.1. Servicios

El gestor de memoria ofrece una serie de servicios a los procesos. Estos son;

- **Solicitar memoria.** Este servicio aumenta el espacio de datos de la imagen de memoria del proceso. El sistema operativo satisf á la petición siempre y cuando cuente con los recursos necesarios para ello. En general, el sistema operativo devuelve un apuntador con la dirección de la nueva memoria. El programa utilizara este nuevo espacio a través del mencionado apuntador, mediante direccionamientos relativos al mismo.
- **Liberar memoria.** Este servicio sirve para devolver trozos de la memoria del proceso. El sistema operativo recupera el recurso liberado y lo añade a sus listas de recursos libres, para su posterior reutilización (Advertencia 2.1),
- **Compartir memoria.** Dentro de esta categoría, el gestor de memoria se encarga de ofrecer servicios que permiten que los procesos puedan comunicarse utilizando un segmento de memoria compartida. Para ello se permite que los procesos *creen* y *liberen* este tipo de segmentos.

ADVERTENCIA 2.1

La utilización de estos dos servicios suele plantear dos problemas de programación que pueden ser difíciles de detectar. Se trata del problema de los apuntadores inválidos y del problema de la pérdida de memoria. Veamos seguidamente estos dos problemas.

Imaginemos que un programa almacena en la variable A el valor del apuntador suministrado por el sistema operativo como respuesta a una solicitud de memoria. Después de utilizar esta zona de memoria a través del apuntador almacenado en A, el programa libera la memoria. El sistema operativo recuperará la memoria liberada, por lo que ya no formará parte de la imagen de memoria del proceso, pero no tocará la variable A, puesto que es interna al proceso. Si, a continuación, el programa utiliza dicha variable para acceder a un dato, nos encontraremos con una situación de error. El apuntador ya no apunta a una dirección de memoria válida.

La pérdida de memoria se produce cuando no se libera la memoria después de haber terminado su uso. El programa puede ir pidiendo más memoria, pero es posible que llegue un momento en que ya no exista memoria libre, por lo que el sistema operativo no podrá atender la solicitud. La pérdida de memoria no es un error fatal, pero es acumulativa, por lo que puede llegar a bloquear el sistema si se produce de forma repetitiva.

En el Capítulo 4 se estudiar los conceptos relativos a la gestión de memoria, los servicios ofrecidos por el gestor de memoria y las técnicas de gestión de memoria.

2.6. COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Los procesos son entes independientes y aislados, puesto que, por razones de seguridad, no deben interferir unos con o os. Sin embargo, cuando se divide un trabajo complejo e varios procesos que

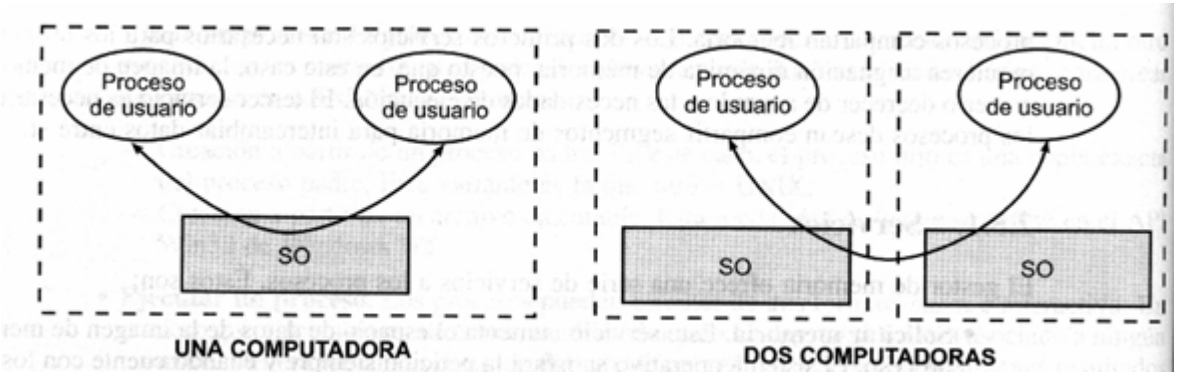


Figura 2.9. *Comunicación entre procesos locales y remoto,*

cooperan entre sí para realizar ese trabajo, es necesario que se comuniquen para transmitirse datos y órdenes y se sincronicen en la ejecución de sus acciones. Por tanto, el sistema operativo debe incluir servicios de comunicación y sincronización entre procesos que, sin romper los esquemas de seguridad, han de permitir la cooperación entre ellos.

El sistema operativo ofrece una serie de mecanismos básicos de comunicación que permiten transferir cadenas de bytes, pero han de ser los procesos que se comunican los que han de interpretar la cadena de bytes transferida. En este sentido, se han de poner de acuerdo en la longitud de la información y en los tipos de datos utilizados. Dependiendo del servicio utilizado, la comunicación se limita a los procesos de una máquina (procesos locales) o puede involucrar a procesos de máquinas distintas (proceso. remotos). La Figura 2.9 muestra ambas situaciones.

El sistema operativo ofrece también mecanismos que permiten que los procesos esperen (se bloqueen) y se despierten (continúen su ejecución) dependiendo de determinados eventos.

2.6.1. Servicios de comunicación y sincronización

Como se ha visto anteriormente, existen distintos mecanismos de comunicación y sincronización, cada uno de los cuales se puede utilizar a través de un conjunto de servicios propios. Estos mecanismos son entidades vivas, cuya vida presenta las siguientes fases;

- Creación del mecanismo.
- Utilización del mecanismo.
- Destrucción del mecanismo,

De acuerdo con esto, los servicios básicos de comunicación, que incluyen todos los mecanismos de comunicación, son los siguientes;

- **Crear.** Permite que el proceso solicite la creación del mecanismo.
- **Enviar o escribir.** Permite que el proceso emisor envíe información a otro.
- **Recibir o leer.** Permite que el proceso receptor reciba información de otro,
- **Destruir**. Permite que el proceso solicite la creación o destrucción del mecanismo.

Por otro lado, la comunicación puede ser síncrona o asíncrona.

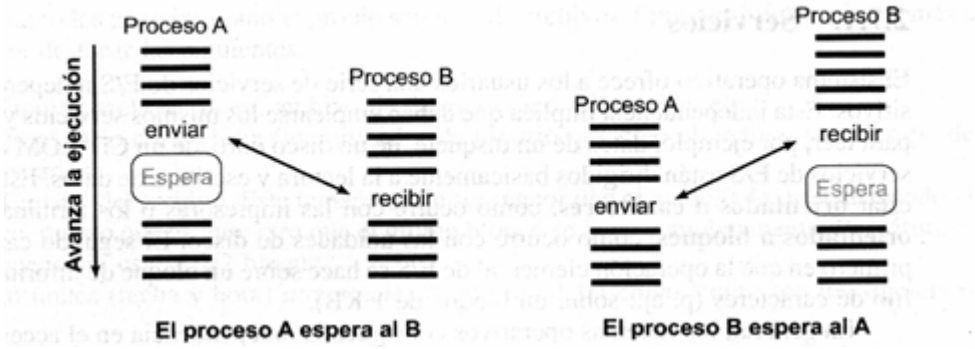


Figura 2.10. *Comunicación síncrona entre procesos.*

En la comunicación **síncrona** los dos procesos han de ejecutar los servicios de comunicación al mismo tiempo, es decir, el emisor ha de estar en el servicio de enviar y el receptor ha de estar en el servicio de recibir, Normalmente, para que esto ocurra, uno de ellos ha de esperar a que el otro llegue a la ejecución del correspondiente servicio (Fig. 2.10).

En la comunicación **asíncrona** el emisor no tiene que esperar a que el receptor solicite el servicio recibir, hace el envío y sigue con la ejecución. Esto obliga a que el sistema operativo establezca un almacenamiento intermedio para guardar la información enviada hasta que el receptor la solicite,

En cuanto a los mecanismos de sincronización, los mecanismos suelen incluir los siguientes servicios:

- **Crear.** Permite que el proceso solicite la creación del mecanismo.
- **Bloquear.** Permite que el proceso se bloquee hasta que ocurra un determinado evento.
- **Despertar.** Permite despertar a un proceso bloqueado.
- **Destruir.** Permite que el proceso solicite la destrucción del mecanismo.

En el Capítulo *5* se estudiarán en detalle los principales mecanismos de comunicación y sincronización que ofrecen los sistemas operativos así como sus servicios.

2.7. ESTIÓN DE LA E/S

Una de las principales funciones de un sistema operativo es la gestión de los recursos de la computadora y, en concreto, de los dispositivos periféricos. El gestor de *EIS* debe controlar el funcionamiento de todos los dispositivos de *E/S* para alcanzar los siguientes objetivos:

- Facilitar el manejo de los dispositivos periféricos. Para ello debe ofrecer una interfaz sencilla, uniforme y fácil de utilizar entre los dispositivos, y gestionar los errores que se pueden producir en el acceso a los mismos,
- Ofrecer mecanismos de protección que impidan a los usuarios acceder sin control a los dispositivos periféricos.

Dentro de la gestión de *E/S,* el sistema operativo debe encargarse de gestionar los distintos dispositivos de E/S; relojes, terminales, dispositivos de almacenamiento secundario y terciario, teclado, etc.

2.7.1. Servicios

El sistema operativo ofrece a los usuarios una serie de servicio, de *E/S* independiente de los dispositivos, Esta independencia implica que deben emplearse los mismos servicios y operaciones de *E/S* para leer, por ejemplo, datos de un disquete, de un disco duro, de un CD-ROM o de un teclado, Los servicios de *E/S* están dirigidos básicamente a la lectura y escritura de datos. Estos servicios pueden **estar orientados a caracteres**, como ocurre con las impresoras o los terminales, o pueden **estar orientados a bloques**, como ocurre con las unidades de disco. El segundo caso se diferencia del primero en que la operación elemental de *E/S* se hace sobre un bloque de información de un número fijo de caracteres (p. ej.: sobre un bloque de 1 KB).

En general, los sistemas operativos consiguen la independencia en el acceso a los dispositivos modelándolos como archivos especiales. La gestión de chivos y sus servicios se describen en la siguiente sección.

En el Capítulo 7 se estudiará el software de E/S del sistema operativo, la gestión del almacenamiento secundario y terciario, de los relojes y terminales. También se presentarán los servicios de *EIS* que ofrecen los sistemas operativos.

2.8. GESTIÓN DE ARCHIVOS Y DIRECTORIOS

El *servidor de archivos* es la parte del sistema operativo que cubre una de las cuatro clases de funciones que tiene éste en su faceta de máquina extendida. Los objetivos fundamentales del servidor de archivos son los dos siguientes:

- Facilitar el manejo de los dispositivos periféricos. Para ello ofrece una visión lógica simplificada de los mismos en forma de chivos.
- Proteger a los usuarios, poniendo limitaciones a los chivos que es capaz de manipular cada usuario.

Los servicios que se engloban en el servidor de archivos son de dos tipos: los servicios dirigidos al manejo de datos, o *archivos,* y los dirigidos al manejo de los nombres, o *directorios.*

El servidor de archivos ofrece al usuario (Fig. 2.11) una **visión lógica** compuesta por una serie de objetos (chivos y directorios) identificables por un nombre lógico sobre los que puede realiza una serie de operaciones. La **visión física** ha de incluir los detalles de cómo están almacenados estos objetos en los periféricos correspondientes (p. ej.: en los discos).

2.8.1. Servicio de archivos

Un archivo es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de información relacionada entre sí bajo un mismo nombre, Cada archivo tiene una información asociada que

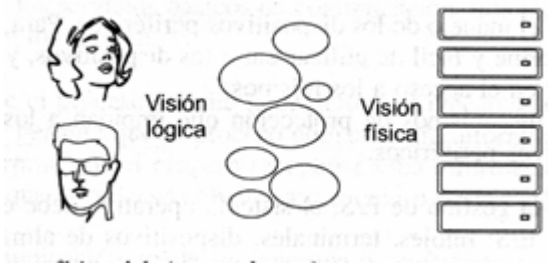


Figura 2.11. *Visión lógica y física del sistema de archivos.*

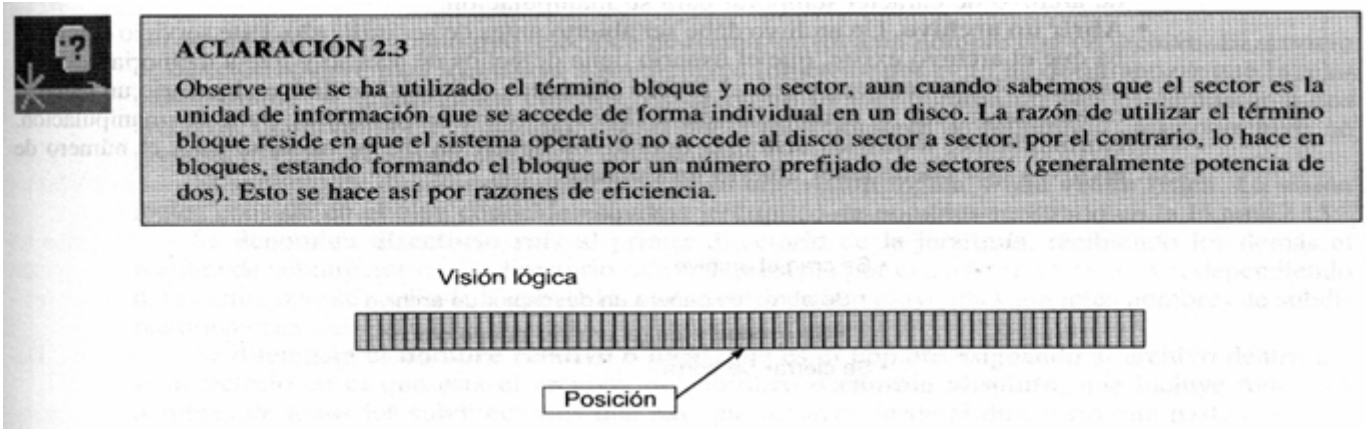
utilizan tanto los usuarios como e! propio servidor de archivos. Entre las informaciones más usuales se pueden destacar las siguientes.

- Tipo de archivo (p, ej., archivo de datos, ejecutable, elc.).
- Propietario del archivo (identificador de usuario que creó el archivo y del grupo de dicho usuario).
- Tamaño del archivo. Este tamaño suele ser menor que el espacio de disco asignado al archivo, puesto que es muy raro que el último bloque se llene completamente. Por término medio queda sin usarse 1/2 bloque.
- Instantes (fecha y hora) importantes de la v.ida del archivo, como son los siguientes:
  - Instante en que se creó.
  - Instante de la última modificación.
  - Instante del último acceso.
- Derechos de acceso al archivo (sólo lectura, lectura-escritura, sólo escritura, ejecución,...).

Las operaciones sobre archivos que ofrece el servidor de chivos están referidas a la visión lógica de los archivos, La solución más común es que el archivo se visualice como un vector de bytes o caracteres, tal y como indica la Figura 2.12. Algunos sistemas de archivos ofrecen visiones lógicas más elaboradas, orientadas a re que pueden ser de longitud fija o variable. La ventaja de la sencilla visión de vector de caracteres es su flexibilidad, puesto que no presupone ninguna estructura específica interna en el archivo.

La visión lógica del archivo incluye normalmente un **puntero de posición** Este puntero permite hacer operaciones de lectura y escritura consecutivas sin tener que indicar la posición de la operación. inicialmente el puntero indica la posición 0, pero después de hacer, por ejemplo, una operación de lectura de 7.845 bytes señalará a la posición 7.845. Otra lectura posterior se referirá a los bytes 7.845, 7.846, etc,

La visión física está formada por los elementos físicos del periférico que soportan al archivo. En el caso más usual de tratarse de discos, la visión física consiste en la enumeración ordenada de los bloques de disco en los que reside el archivo (Aclaración 2.3). La Figura 2.13 muestra un ejemplo en el que el archivo A está formado, en ese orden, por los bloques 12, 20, 1, 8, 3, 16 y 19.



ACLARACIÓN 2.3

Observe que se ha utilizado el término bloque y no sector, aun cuando sabemos que el sector es la unidad de información que se accede de forma individual en un disco. La razón de utilizar el término bloque reside en que el sistema operativo no accede al disco sector a sector, por el contrario, lo hace en bloques, estando formando el bloque por un número prefijado de sectores (generalmente potencia de dos). Esto se hace así por razones de eficiencia.

Figura 2.12. Visión lógica del archivo.

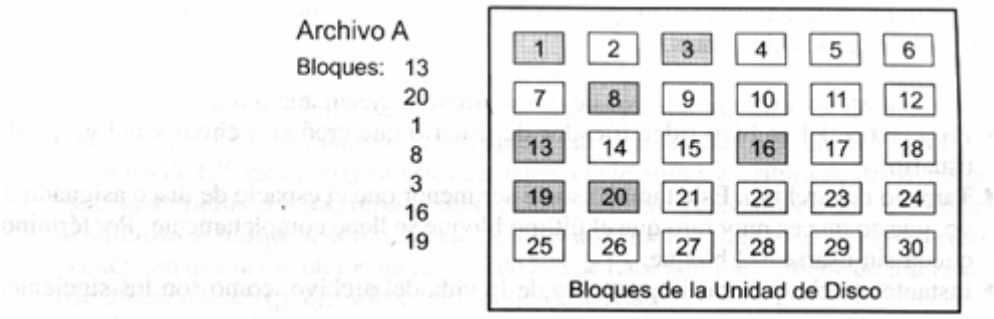


Figura 213. Visión física del archivo.

Observe que debe existir una estructura de información que recoja la composición física cada archivo, que se denominará de forma genérica descripción física del archivo. Esta estructura es la FAT en MS-DOS y el nodo-i en UNIX Finalmente, es de destacar que estas estructuras de información han de residir en e propio periférico (p. ej.: disco), para que éste sea autocontenido y se pueda transportar de un sistema a otro.

El servidor de archivos es capaz de encontrar e interpretar estas estructuras de información liberando a los programas de usuario de esta tediosa labor

Servicios de archivos

Un archivo es una entidad viva, que va evolucionando de acuerdo a los servicios que se solicitan d sistema operativo. La Figura 2.14 resume las fases de esta vida.

Los servicios que ofrece el servidor de archivos son los siguientes:

- **Crear un archivo.** Este servicio cerca un archivo vacío. La creación de un archivo exige una interpretación dcl nombre, puesto que el servidor de archivos ha de comprobar que el nombre es correcto y que el usuario puede hacer la operación solicitada. La creación de un archivo deja abierto a éste devolviendo al usuario un identificador, descriptor o manejador de archivo de carácter temporal para su manipulación.
- **Abrir un archivo.** Un archivo debe ser abierto antes de ser utilizado. Este servicio comprueba que el archivo existe, que el usuario tiene derechos de acceso y trae a memoria información del objeto para optimizar el acceso al mismo Además devuelve al usuario un identificador, descriptor o manejador de archivo de carácter temporal para su manipulación Normalmente, todos los sistemas operativos tienen un límite máximo para el número archivos que puede tener abierto un usuario.

- Se crea el archivo
  - Se abre: se genera un descriptor de archivo
    - Se escribe y lee (el archivo puede crecer)

• Se cierra Se borra

Figura 2.14. Servicios y vida del archivo.

- **Escribir y leer.** Estos servicios se realizan utilizando el identificador, descriptor o manejador de archivo (devuelto en las operaciones de creación y apertura), en vez del nombre lógico del mismo.

  Una operación de lectura permite traer datos del archivo a memoria. Para ello se especifica el identificador, descriptor o manejador obtenido en la apertura, la posición de memoria y la cantidad de información a leer. Normalmente, se lee a partir de la posición que indica el puntero de posición del archivo, Las operaciones de escritura permiten llevar datos situados en memoria al archivo. Para ello, y al igual que en las operaciones de lectura, se debe especificar el identificador obtenido en la creación o apertura, la posición de memoria y la cantidad de información a escribir, Normalmente se escribe a partir de La posición que indica el puntero de posición del archivo. Si está en medio, se sobrescribirán los datos, Si está al final del archivo, su tamaño crece. En este caso, el sistema operativo se encarga de hacer crecer el espacio físico del archivo añadiendo bloques libres.

  En algunos sistemas operativos se puede especificar la posición del archivo en la que se realizará la siguiente lectura o escritura.

  • **Cerrar un archivo.** Terminada la utilización del archivo se debe cerrar, con lo que se elimina el identificador temporal obtenido en la apertura o creación y se liberan los recursos de memoria que ocupa el archivo,

  • **Borrar un archivo.** El archivo se puede borrar, lo que supone que se borra su nombre del correspondiente directorio y que el sistema de archivos ha de recuperar los bloques de datos y el espacio de metainformación que tenía asignado (Aclaración 2.4).

ACLARACIÓN 2.4

La metainformación de un archivo se refiere a toda la información auxiliar que es necesario mantener para ofrecer la visión lógica de un archivo. Esta información incluye entre otros los bloques que ocupa el archivo en disco.

2.8.2. Servicio de directorios

Un directorio es un objeto que relaciona de forma unívoca un nombre con un archivo. El servicio de directorios sirve para identificar a los archivos (objetos), por tanto ha de garantizar que la relación [nombre — > archivo] sea unívoca. Es decir, un mismo nombre no puede identificar a dos archivos. Observe, por el contrario, que el que vados nombres se refieran al mismo archivo no presenta ningún problema, son simples sinónimos.

El servicio de directorios también presenta una **visión lógica** y una **visión física**. La visión lógica consiste en el bien conocido esquema jerárquico de nombres mostrado en la Figura 2.15.

Se denomina **directorio raíz** al primer directorio de la jerarquía, recibiendo los demás el nombre de subdirectorios. El directorio raíz se representa por el carácter (“/ ” o “ ”,dependiendo del sistema operativo. En la Figura 2.15, el directorio raíz incluye los siguientes nombres de subdirectorios: Textos, Div11 y Div2,

Se diferencia el **nombre relativo o local**, que es el nombre asignando al archivo dentro del subdirectorio en el que está el archivo, del nombre o camino absoluto, que incluye todos los nombres de todos los subdirectorios que hay que recorrer desde el directorio raíz hasta el objeto considerado, concatenados por el símbolo “/” o “”.Un ejemplo del nombre relativo es “Ap11 “, mientras que su nombre absoluto es “/Textos/Tipo/Sec1/Ap11”.

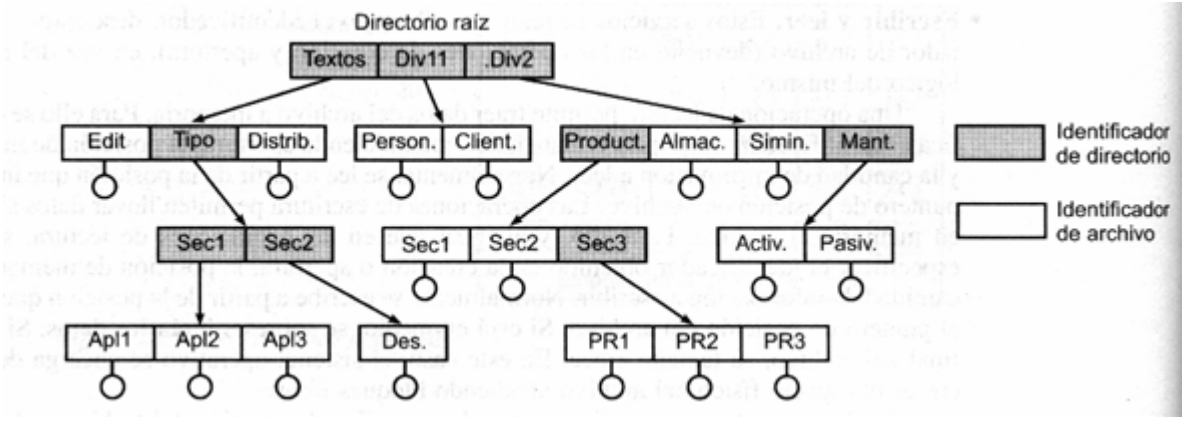


Figura2.15. *Esquema jerárquico de directorios*

La ventaja del esquema jerárquico es que permite una gestión distribuida de los nombres, garantizar de forma sencilla que no exista nombres repetidos. En efecto, hasta con que los nombres relativos de cada. subdirectorio sean distintos, aunque los nombres relativos de subdirectorios distintos sean iguales, para que no exista duplicación de nombres, puesto que quedarán diferencia dos por el camino hasta llegar al correspondiente subdirectorio.

La **visión física** del sistema de directorios consiste en unas estructuras de información que permiten relacionar cada nombre lógico con la descripción física del correspondiente archivo. En esencia, se trata de una tabla NOMBRE- IDENTIFICADOR por cada subdirectorio. El NOMBRE no es mas que el nombre relativo del archivo, mientras que el.DIENTIFICADOR es una información que permite localizar la descripción física del archivo.

Servicios de directorios

Un objeto directorio es básicamente un conjunto de entradas que relacionan nombres y archivos El servidor de archivos incluye una serie de servicios que permiten manipular directorios. Estos son.:

- **Crear un directorio.** Crea un objeto directorio y lo sitúa en el árbol de directorios donde se especifique en el nombre, absoluto o relativo, del nuevo directorio.
- **Borrar un directorio.** Elimina un objeto directorio de forma que nunca más pueda accesible y borra su entrada del árbol de directorios. Normalmente, sólo se puede borrar directorio vacío, es decir, un directorio sin entradas.
- **Abrir un directorio.** Abre un directorio para leer los datos del mismo. Al igual que un chivo, un directorio debe ser abierto para poder acceder a. su contenido. Esta operación vuelve al usuario un identificador, descriptor o manejador de directorio de carácter temporal que permite su manipulación.
- **Leer un directorio.** Extrae la siguiente entrada de un directorio, abierto previamente. Devuelve una estructura de datos como la que define la entrada de directorios
- **Cerrar un directorio.** Cierra un directorio, liberando el identificador devuelto en la operación de apertura, así como los recursos de memoria y del sistema operativo relativos al mismo.

2.8.3. Sistema de archivos

Se denomina sistema de archivos al conjunto de archivos incluidos en una unidad de disco. El sistema de archivos está compuesto por los datos de los archivos, así como por toda la información auxiliar que se requiere.

Se denomina **rnetainformación** a toda la información auxiliar que es necesario mantener en un volumen. Resumiendo y completando lo visto en las secciones anteriores, la metainformación está compuesta por los siguientes elementos:

- Estructura física de los archivos (nodos-i de UNIX o FAT de MS-DOS)
- Directorios (archivos que contienen las tablas nombre-puntero).
- Estructura física del sistema de archivos (superbloque en UNIX).
- Estructura de información de bloques y nodos-i libres (mapas de bits).

Cada sistema operativo organiza las particiones de disco de una determinada forma, repartiendo el espacio disponible entre: el programa de carga (*boot*) del sistema operativo, la metainforinación y los datos. Normalmente, las tablas de los subdirectorios se almacenan como archivos, por lo que compiten por los bloques de datos con los archivos de datos.

En el Capítulo 8 se estudiará en detalle la gestión de archivos y directorios, presentando los conceptos, los servicios y los principales aspectos de implementación.

2.9. SEGURIDAD Y PROTECCIÓN

.

La seguridad, reviste dos aspectos, uno es garantizar la identidad de los usuarios y otro es definir lo que puede hacer cada uno de ellos, El primer aspecto se trata bajo el término de autenticación, mientras que el segundo se hace mediante los privilegios. La seguridad es una de las funciones del sistema operativo que, para llevarla a cabo, se ha de basar en los mecanismos de protección que le proporciona e hardware.

Autenticación

El objetivo de la autenticación es determinar que un usuario (persona, servicio o computadora) es quien dice ser. El sistema operativo dispone de un módulo de autenticación que se encarga de decidir la identidad de los usuarios. En la actualidad, las contraseñas (passwords) son el método más utilizado como mecanismo de autenticación. En el Capítulo 9 se verán otras formas de autenticación.

Privilegios

Los privilegios especifican los recursos que puede acceder cada usuario. Para simplificar la información de privilegios es corriente organizar a los usuarios en grupos, asignando determinados privilegios a cada grupo.

La información de los privilegios se puede asociar a los recursos o a los usuarios.

- **Información por recurso.** En este caso se asocia la denominada lista de control de acceso (ACL, access control list) a cada recurso. Esta lista especifica los grupos y usuarios que pueden acceder al recurso.
- **Información por usuario.** Se asocia a cada usuario o grupo la lisia de recursos que puede acceder, lista que se llama de capacidades (capabilities).

Dado que hay muchas formas de utilizar un recurso, la lista de control de acceso, o la de capacidades, han de incluir el **modo** en que se puede utilizar el recurso Ejemplos de modos de utilización son los siguientes: leer, escribir, ejecutar, eliminar, test, control y administrar.

Los servicios relacionados con la seguridad y la protección se centran en la capacidad para asignar atributos de seguridad a los usuarios y a los recursos.

En el Capítulo 9 se describirán todos los aspectos relacionados con la seguridad y la protección y se presentarán los principales servicios.

2.10. ACTIVACIÓN DEL SISTEMA OPERATIVO

Una vez presentadas las funciones y principales componentes del sistema operativo, es importante describir cuáles son las acciones que activan la ejecución del mismo,

El sistema operativo es un servidor que está a la espera de que se le encargue trabajo. La secuencia normal es la mostrada en la Figura 2,16. Está ejecutando un proceso Ay, en un instante determinado, se solicita la atención del sistema operativo. Éste entra en ejecución y salva el estado en el bloque de control del proceso A. Seguidamente realiza la tarea solicitada. Una vez finalizada esta tarea, entra en acción el planificador, módulo del sistema operativo que selecciona un proceso B para ejecutar. La actuación del sistema operativo finaliza con el activador, módulo que se encarga de restituir los registros con los valores almacenados en el bloque de control dc proceso 13. instante en el que se restituye el contador de programa marca la transición del sistema operativo. ejecución del proceso B.

El trabajo del sistema operativo puede provenir de las siguientes fuentes:

• Llamadas al sistema emitidas por los programas.

▪ Interrupción producida por los periféricos.

• Condiciones de excepción o error del hardware.

En todos estos casos se deja de ejecutar el proceso en ejecución y se entra a ejecutar el sistema operativo se vio en el Capítulo 1, los mecanismos para romper la sentencia lineal de ejecución son dos: las instrucciones de bifurcación y las interrupciones.

El primero de estos dos mecanismos no sirve para invocar al sistema operativo, puesto que el proceso ejecuta en nivel de usuario y el sistema operativo ha de ejecutar en nivel de núcleo y espacios de direcciones distintos. Esto significa que los servicios del sistema operativo no se pueden solicitar mediante una instrucción máquina CALL, o lo que es lo mismo, mediante una llamada a procedimiento o función.

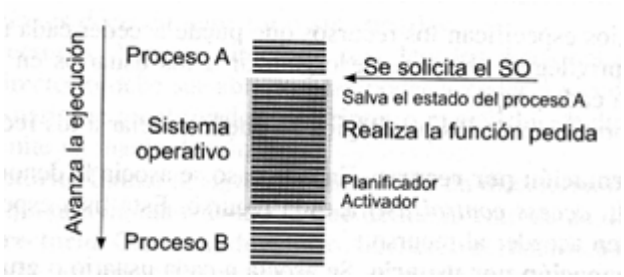


Figura 2.16. Fases en la activación del sistema operativo.

Por tanto, la activación del sistema operativo solamente se realiza mediante el mecanismo de las interrupciones. Cuando es un proceso en ejecución el que desea un servicio del sistema operativo ha de utilizar una instrucción TRAP, que genera la interrupción pertinente. En los demás casos será una interrupción, interna o externa, la que reclame la atención del sistema operativo.

Cuando se programa en un lenguaje de alto nivel, como C, la solicitud de un servicio del sistema operativo se hace mediante una llamada a una función (p. ej.: u = fork ()) (Aclaración 2.5). No hay que confundir esta llamada con el servicio del sistema operativo. La función fork del lenguaje C no realiza el servicio fork, simplemente se limita a solicitar este servicio del sistema operativo. En general estas función solicitan los servicios del sistema operativo se componen de:

- Una parte inicial que prepara los parámetros del servicio de acuerdo con la forma en que los espera el sistema operativo.
- La instrucción TRAP que realiza el paso al sistema operativo.
- Una parte final que recupera los parámetros de contestación del sistema operativo, para devolverlos al programa que le llamó.

ACLARACIÓN 2.5

La llamada `fork()` es el servicio que ofrece el API de POSIX para la creación de un nuevo proceso.

Todo este conjunto de funciones se encuentran en una biblioteca del sistema y se incluyen en el código en el momento de su carga en memoria.

Para completar la imagen de que se está llamando a una función, el sistema operativo devuelve un valor, como una función real. Al programador le parece, por tanto, que invoca al sistema operativo como a una función. Sin embargo, esto no es así, puesto que lo que hace es invocar una función que realiza la solicitud al sistema operativo. El siguiente código muestra una hipotética implementación de la llamada al sistema fork.

```
int fork() {
    int r;
    LOAD R8, FORK_SYSTEM_CALL
    TRAP
    LOAD r, R9
    return(r);
}
```

El código anterior carga en uno de los registros de la computadora (el registro RS por ejemplo) el número que identifica la llamada al sistema (en este caso FORK_SYSTEM_CALL). En el caso de que la llamada incluyera parámetros, éstos se pasarían en otros registros o en la. pila. A continuación, la función de biblioteca ejecuta la instrucción TPAP, con lo que se transfiere el control al sistema operativo, que accede al contenido del registro R8 para identificar la llamada a ejecutar y realizar el trabajo. Cuando el control se transfiere de nuevo al proceso que invocó la llamada fork, se accede al registro R9 para obtener el valor devuelto por la llamada y éste se retorna, finalizando así la ejecución de la función de biblioteca.

La Figura 2.17 muestra todos los pasos involucrados en una llamada fork() al sistema operativo, indicando el código que interviene en cada uno de ellos.

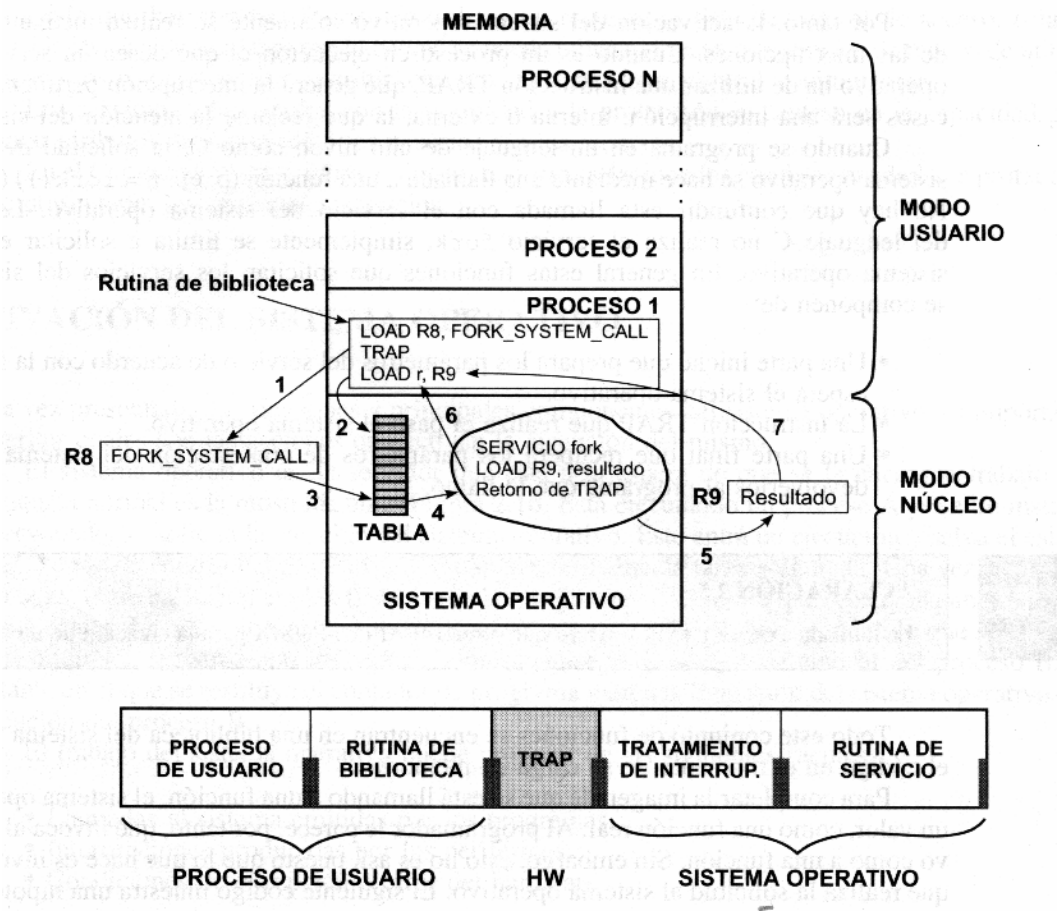


Figura 2.17. Pasos de la llamada al sistema operativo.

Por ejemplo, si el programa quiere escribir datos en un archivo, el código del programa usuario hace un CALL a la rutina en código máquina write, con código similar al mostrado anteriormente para la llamada. fork, Esta rutina prepara los parámetros de la operación de escritura C ejecuta la instrucción TRAP. El sistema operativo trata la interrupción, identifica que se trata de una solicitud de servicio y que el servicio solicitado es la escritura en un archivo. Seguidamente ejecutan, la rutina que lanza la pertinente operación de E/S. Finalmente, ejecuta e planificador y el activador para dar paso a la ejecución de otro proceso.

Cuando el periférico termina la operación, su controlador genera una solicitud de interrupción que es tratada por el sistema operativo. Como resultado de la misma, el proceso anterior pasará a estar listo para ejecución. En su momento, se seleccionará este proceso para que ejecute En su momento la rutina en código máquina write recibe los parámetros que ha devuelto el sistema operativo y ejecuta un RET para volver al programa de usuario que la llamó.

En este libro emplearemos el lenguaje C para ilustrar los ejemplos de invocación de servicios del sistema operativo. Aunque desde el C se estará llamando a funciones, no hay que olvidar que lo único que hace la función invocada es llamar al sistema operativo, que... es quien hace el trabajo.

2.11. INTERFAZ DEL PROGRAMADOR

La interfaz del sistema operativo con el programador es la que recupera los servicios y llamadas al sistema que los usuarios pueden utilizar directamente desde sus programas. Esta es, quizá, una de las partes más importantes de un sistema operativo, ya que recupera la visión que corno máquina extendida tiene el usuario de sistema operativo. En este libro se van a estudiar dos de las interfaces más utilizadas en la actualidad: POSIX y los servicios de Win32.

2.11. POSIX

POSIX [IEEE96] es el estándar de **interfaz de sistemas operativos potables** de 1EEE basado en el sistema operativo UNIX. Aunque UNIX era prácticamente un estándar industrial, había bastantes, diferencias entre las distintas implementaciones de UNIX, lo que provocaba que las aplicaciones no se pudieran transportar fácilmente entre distintas plataformas UNIX. Este problema motivó a los implementadores y usuarios a desarrollar un estándar internacional con el propósito de conseguir la portabilidad de las aplicaciones en cuanto a código fuente.

POSIX se ha desarrollado dentro de IEEE con la. referencia 1003 y también está siendo desarrollado corno estándar internacional con la referencia 1SO/ 9945.

POSIX es una familia de estándares (Aclaración 2.6) en evolución, cada uno de tos cuales cubre diferentes aspectos de los sistemas operativos. Algunos de estos estándares ya han sido aprobados, mientras que otros están todavía en 6ise de desarrollo. POSIX incluye servicios de sistema operativo para muchos entornas de aplicación. La Tabla 2.1 lista algunos de los estándares básicos de POSIX.

POSIX es una interfaz ampliamente utilizada. Se encuentra disponible en todas las versiones de Unix y Linux. También Windows NT ofrece un subsistema que permite programar aplicaciones POSIX

ACLARACIÓN 2.6

POSIX es una especificación estándar, no define una implementación. Los distintos sistemas operativos pueden ofrecer los servicios POSIX con diferentes implementaciones.

Tabla 2.1. *Algunos de los estándares base de POSIX*

1003.1	Servicios básicos del sistema operativo.
1003.1a	Extensiones a los servicios básicos.
1003.1b	Extensiones de tiempo real.
1003.1c	Extensiones de procesos ligeros.
1003.1d	Extensiones adicionales de tiempo real.
1003.1e	Seguridad.
1003.2	Shell y utilidades.
1003.2b	Utilidades adicionales.

Algunas de las características de POSIX son:

- Algunos tipos de datos utilizados por las funciones no se definen como parle del estándar, pero se definen como parte de la implementación. Estos tipos se encuentran definidos en el archivo de cabecera. <sys/tipes-i>. Estos tipos acaban con el sufijo_t. Por ejemplo uid_t es el tipo que se emplea para almacenar un identificador de usuario.
- Los nombres de las funciones en POSIX son en general cortos y con todos sus letras en minúsculas Ejemplos de funciones en POSIX son:
- Las funciones, normalmente, devuelven cero si se ejecutaron con éxito 0-l en caso de error; Cuando una función devuelve -1, se almacena en una variable global, denominada errno,. el código de error. Este código de error es un valor entero. La variable errno se encuentra definida en el archivo de cabecera <errno . h>.
- La mayoría de los recursos gestionados por el sistema operativo se reverencian mediante descriptores. Un descriptor es un número entero mayor o igual que cero.

2.1l . Win32

Win32 define los servicios ofrecidos por los sistemas Windows 95/98, Windows NT y Windows 2000. En este caso no se trata de un estándar genérico, sino de los servicios establecido por una casa comercial determinada (Microsoft).

El API de Win32 es totalmente diferente al estándar POSIX. A continuación, se citan ah de las principales características de Win32 [Hart, 1998]:

- Prácticamente todos los recursos gestionados por el sistema operativo se tratan como objetos, que se reverencian por medio de manejadores. Estos manejadores son similares a los descriptores de archivos de POSIX. Aunque sigue los principios de la programación orienta da a objetos, Win32 no es orientado a objetos.
- Los nombres de las funciones en Win32 son largos y descriptivos, a diferencia de ocurre en POSIX Ejemplos de funciones en Win32 son

  —- GetFileAttributes, para obtener los atributos de un archivo.

  — CreateNarnedpipe, para crear una tubería con nombre.

- Win32 tiene una, se de tipos de datos predefinidos, por ejemplo:

  — Bool, objeto de 32 bits que almacena un valor lógico.

  — DWCRD, entejo sin signo de 32 bits.

  — TCHAP, tipo carácter de dos bytes. LPSTR, puntero a una cadena de caracteres.

- Los tipos predefinidos en Win32 evitan el uso del operador de indirección de C (*), Así, ejemplo, LPSTR está definido corno *TCHAR
- Los nombres de las variables, al menos en los prototipos de las funciones, también siguen una serie de convenciones. Por ejemplo, lpszFileName representa un puntero largo a cadena de caracteres terminada por el carácter nulo.

- En Win32, las funciones devuelven, en general, true si la llamada se ejecutó con éxito o false en caso contrario.

Aunque Win32 incluye muchas funciones, las cuales tienen en muchos casos numerosos parámetros (muchos de los cuales normalmente no se utilizan), este libro se va a centrar en las funciones más importantes del API. Además, Win32 define funciones y servicios gráficos que no serán trata dos en este libro,

2.12. INTERFAZ DE USUARIO DEL SISTEMA OPERATIVO

Cuando un usuario trabaja con una computadora necesita poder interactuar con el sistema operativo para poder llevar a cabo operaciones tales como ejecutar un programa o borrar un archivo, sin necesidad de escribir un programa que realice dicha operación utilizando los servicios del sistema operativo.

El sistema operativo, por tanto, además de dotar de servicios (llamadas al sistema) a las aplicaciones, debe proporcionar una interfaz de usuario que permita dar instrucciones al sistema para realizar diversas operaciones. Sin esta interfaz, aunque el sistema operativo estuviese listo para dar servicio a las aplicaciones, el usuario no podría. arrancar ninguna.

Hay que tener en cuenta que la mayoría de la gente que trabaja con un sistema informático no pretende realizar ninguna tarea de programación, sino que, simplemente, quiere trabajar en modo interactivo con el mismo. Por tanto, no tendría sentido que los usuarios tuvieran que realizar un programa para poder sacar partido de los servicios del sistema operativo a la hora de realizar una determinada labor (como, por ejemplo, borrar un archivo). El sistema operativo ofrece, a través de su interfaz de usuario, un conjunto de operaciones típicas, que son las que necesitan llevar a cabo los usuarios con más frecuencia, Así, para borrar un archivo, en vez de tener que realizar un programa, el usuario sólo tendrá que teclear un mandato (rm en UNIX o del en MS-DOS) o, en el caso de una interfaz gráfica, manipular un icono que representa al archivo.

La interfaz de usuario de tos sistemas operativos, al igual que la de cualquier otro tipo de aplicación, ha sufrido una gran. evolución. Esta evolución ha venido condicionada, en gran parte, por la enorme difusión del uso de las computadoras, que ha tenido como consecuencia que un gran número de usuarios sin conocimientos informáticos trabajen cotidianamente con ellas. Se ha pasado de interfaces alfanuméricas, que requerían un conocimiento bastante profundo del funcionamiento de la computadora a interfaces gráficas, que ocultan al usuario la complejidad del sistema. proporcionándole una visión intuitiva del mismo.

Ha existido también una evolución en la integración de la interfaz de usuario con el resto del sistema operativo. Se ha. pasado de sistemas en los que el módulo que maneja la interfaz de usuario estaba dentro del núcleo del sistema operativo (la parte del mismo que ejecuta en modo privilegiado) a sistemas en los que esta función es realizada por un conjunto de programas externos al núcleo que ejecutan en modo no privilegiado y usan los servicios del sistema operativo como cualquier otro programa. Esta estrategia de diseño proporciona una gran flexibilidad, pudiendo permitir que cada usuario utilice un programa de interfaz que se ajuste a sus preferencias o que, incluso, cree uno propio. EL sistema operativo, por tanto, se caracteriza principalmente por los servicios que proporciona y no por su interfaz que, al fin y al cabo, puede ser diferente para los distintos usuarios,

Dado que no se va a estudiar este aspecto en el resto del libro, a continuación se comenta las principales funciones de la interfaz de usuario de un sistema operativo.

2.12.1 Funciones de la interfaz de usuario

La principal misión de la interfaz, sea del tipo que sea, es permitir al usuario acceder y manipular Y los objetos y recursos del sistema En esta sección se presentaran de forma genérica cuáles son las operaciones que típicamente ofrece el sistema operativo a sus usuarios, con independencia de cómo Y lleven éstos a c'abo el diálogo con el mismo.

A la hora de realizar esta enumeración, surge una cuestión sobre la que no hay un acuerdo Y. general: ¿cuáles de los programas que hay en un determinado sistema se consideran parte de la interfaz del sistema y cuáles no? ¿Un compilador es parte de la interfaz de usuario del sistema operativo? ¿Y un navegador Web?

Una alternativa sería considerar que forman parte de la interfaz del sistema todos los programas que se incluyen durante la instalación del sistema operativo y dejar fuera de dicha categoría a los programas que se instalan posteriormente. Sin embargo, no hay un criterio único ya que diferentes fabricantes siguen políticas distintas. Por ejemplo, algunos sistemas incluyen uno o más compiladores de lenguajes de alto nivel, mientras que otros no lo hacen. Prueba de esta confusión es el Y contencioso legal al que se enfrenta Microsoft sobre si el navegador Web debe formar parte de su Y sistema operativo o no.

En. la clasificación que se plantea a continuación se han seleccionado aquellas funciones sobre las que parece que hay un consenso general en cuanto a que forman parte de la interfaz del sistema. Se han distinguido las siguientes categorías:

- Manipulación de archivos y directorios. La interfaz debe proporcionar operaciones para crear, borrar, renombrar y, en general, procesar archivos y directorios.
- Ejecución de programas. El usuario tiene que poder ejecutar programas y controlar la ejecución de Los mismos (p, ej.: parar temporalmente su ejecución o terminarla incondicional mente).
- Herramientas para el desarrollo de las aplicaciones. El usuario debe disponer de utilidades, tales como ensambladores, enlazadores y depuradores, para construir sus propias aplicaciones. Observe que se han dejado fuera de esta categoría a los compiladores, por los motivos antes expuestos.
- Comunicación con otros sistemas. Existirán herramientas para acceder a recursos localizados en. otros sistemas accesibles a través de una red. de conexión. En esta categoría se consideran herramientas básicas, tales como ftp y telnet (Aclaración 2.7), dejando fuera aplicaciones de más alto nivel como un navegador Web.
- Información de estado del sistema. El usuario dispondrá de utilidades para obtener informaciones tales como la fecha, la hora, el numero de usuarios que están trabajando en el sistema o la cantidad de memoria disponible.
- Configuración de la propia interfaz y del entorno. Cada usuario tiene que poder configurar el modo de operación de la interfaz (le acuerdo a sus preferencias. Un ejemplo sería la configuración de los aspectos relacionados con las características específicas del entorno geográfico Y del usuario (lenguaje, formato de fechas y de cantidades de dinero, etc.). La flexibilidad de Y configuración de la interfaz será una de las medidas que exprese su calidad.
- Intercambio de datos entre aplicaciones. El usuario va a disponer de mecanismos que le permitan especificar que, por ejemplo, una aplicación utilice los datos que genera otra.
- Control de acceso. En sistemas multiusuario, la interfaz debe encargarse de controlar el acceso de los usuarios al sistema para mantener la seguridad del. mismo.

- Normalmente, el mecanismo de control estará basado en que cada usuario autorizado tenga una contraseña que deba introducir para acceder al sistema.
- Otras utilidades y herramientas. Para no extender innecesariamente la clasificación, se han agrupado en esta sección utilidades misceláneas tales corno calculadoras o agendas.
- Sistema de ayuda interactivo, La interfaz debe incluir un completo entono de ayuda que ponga a disposición de usuario toda la documentación del sistema.

ACLARACIÓN 2.7

La aplicación ftp (*file transfer protocol*) permite transferir archivos entre computadoras conectadas por una red de conexión. La aplicación telnet permite a los usuarios acceder a computadoras remotas, de tal manera que la computadora en la que se ejecuta la aplicación telnet se convierte en un terminal de la computadora remota.

Para concluir esta sección, es importante resaltar que en un sistema, además de las interfaces disponibles para los usuarios normales, pueden existir otras específicas destinadas a los administradores del sistema. Más aún, el propio programa (residente normalmente en ROM) que se encarga de la carga del sistema operativo proporciona generalmente una interfaz de usuario muy simplificada y rígida que permite al administrador realizar operaciones tales como pruebas y diagnósticos del hardware o la modificación de los parámetros almacenados en la memoria RAM no volátil de la máquina que controlan características de bajo nivel del sistema.

2.12.2. interfaces alfanuméricas

La característica principal de este tipo de interfaces es su modo de trabajo basado en líneas de texto. El usuario, par dar instrucciones al sistema, escribe en su terminal un mandato terminado con un carácter de final de línea. Cada mandato está normalmente estructurado corno un nombre de mandato (p. ej.: borrar) y unos argumentos (p. ej.: el. nombre del archivo que se quiere borrar). Observe que en algunos sistemas se permite que se introduzcan varios mandatos en una línea. El **intérprete de mandatos,** que es como se denomina típicamente al módulo encargado de la interfaz, lee la línea escrita por el usuario y lleva a cabo las acciones especificadas por la misma. Una vez realizadas, el intérprete escribe una indicación (prompt) en el terminal para notificar al usuario que está listo para recibir otro mandato. Este ciclo repetitivo define el modo de operación de este tipo de interfaces, El usuario tendrá disponibles un conjunto de mandatos que le permitirán realizar actividades tales como manipular archivos y directorios, controlar la ejecución de los programas, desarrollar aplicaciones, comunicarse con otros sistemas, obtener información del estado del sistema o acceder al sistema. de ayuda interactivo.

Esta forma de operar, basada en líneas de texto, viene condicionada en parte por el tipo de dispositivo que se usaba como terminal en los primeros sistemas de tiempo compartido. Se trataba de teletipos que imprimían la salida en papel y que, por tanto, tenían intrínsecamente UI) funciona miento basado en líneas. La disponibilidad posterior de terminales más sofisticados que, aunque seguían siendo de carácter alfanumérico, usaban una pantalla para mostrar la información y ofrecían, por tanto, la posibilidad de trabajar con toda la. pantalla no cambió, sin embargo, la forma de trabajo de la interfaz que continuó siendo en modo línea, Como reflejo de esta herencia, obsérvese que en el mundo UNIX se usa el término tty (abreviatura de teletype) para referirse a un terminal, aunque no tenga nada que ver con los primitivos teletipos. Observe que, sin embargo, muchas aplicaciones sí que se aprovecharon del. modo de trabajo en modo pantalla. Como ejemplo, se puede

observar la evolución de los editores en UNIX: se pasó de editores en modo línea como el ed a editores orientados a pantalla como el vi y el emars.

A pesar de que el modo de operación básico apenas ha cambiado, Su estructura e implementación han evolucionado notablemente desde la aparición de los primeros sistemas de tiempo compartido hasta la actualidad. Como ya se ha comentado anteriormente, se pasó de tener el intérprete incluido en el sistema operativo a su un módulo externo que usa los servicios del mismo, lo proporciona una mayor flexibilidad facilitando su modificación o incluso su reemplazo. Dentro esta opción existen básicamente dos formas de esturar el módulo que maneja la interfaz de usuario intérprete con mandatos internos e interprete con mandatos externos

Intérprete con mandatos internos

El intérprete de mandatos es un único programa que contiene el código para e todos los mandatos. El intérprete, después de leer la línea tecleada por el usuario, determina de qué mandato se trata y salta a la parle (le su código que lleva a cabo la acción especificada por el mandato. Si no se trata de ningún mandato, se interpreta que el usuario quiere arrancar una determinada aplicación, en cuyo caso el intérprete iniciará la ejecución del programa correspondiente en el contexto de un nuevo proceso y esperará hasta que termine. Con esta estrategia, mostrada en el Programa 21, los mandatos son internos al intérprete. Obsérvese que en esta sección se está suponiendo que hay un único mandato en cada línea.

Programa 2.1. Esquema de un intérprete con mandatos internos.

```
Repetir Bucle
    Escribir indicación de preparado
    Leer e interpretar línea -> Obtiene operación y argumentos
    Caso operación
        Si  " fin"
            Terminar ejecución de intérprete /
        Si  "renombrar"
            Renombrar archivos según especifican argumentos
        Si "borrar"
            Borrar archivos especificados por argumentos
        Si no (No se trata de un mandato)
            Arrancar programa "operación" pasándole "argumentos"
            Esperar a que termine el programa
Fin Bucle
```

Intérprete con mandatos externos

Existe un programa por cada mandato. El intérprete de mandatos no analiza la línea tecleada por el usuario, sino que directamente inicia la ejecución del programa correspondiente en el contexto de un nuevo proceso y espera que éste termine. Se realiza un mismo tratamiento ya se trate de un mandato o de cualquier otra aplicación. Con esta estrategia, mostrada en el Programa 2.2, los mandatos son externos al intérprete y la interfaz de usuario está compuesta por un conjunto de programas del sistema: un programa por cada mandato más el propio intérprete.

Programa 22, Esquema de un intérprete con mandatos externos.

```
Repetir Bucle
    Escribir indicación de preparado
    Leer e interpretar línea —> Obtiene operación y argumentos
        Si operación = “fin”
            Terminar ejecución de intérprete
        Si no
            Arrancar programa “operación” pasándole“argumentos”
            Esperar a que termine el programa
Fin Bucle
```

La principal ventaja de la primera estrategia es la eficiencia, ya que los mandatos los lleva a cabo el propio intérprete sin necesidad de ejecutar programas adicionales. Sin. embargo, el intérprete puede llegar a ser muy grande y la inclusión de un nuevo mandato, o la modificación de uno existente, exige cambiar el código del intérprete y recompilarlo. La segunda solución es más recomendable ya que proporciona un tratamiento y visión uniforme de los mandatos del sistema y las restantes aplicaciones. El intérprete no se ve afectado por la inclusión o la modificación de un mandato.

En los sistemas reales puede existir tina mezcla de las dos estrategias. El intérprete de manda tos de MS-DOS (COMMAND. COM)se enmarca dentro de la primera categoría, esto es, intérprete con mandatos internos. El motivo de esta estrategia se debe a que este sistema operativo se diseñó para poder usarse en computadoras sin disco duro y, en este tipo de sistemas, el uso de un intérprete con mandatos externos exigiría que el disquete correspondiente estuviese insertado para ejecutar un determinado mandato. Sin embargo, dadas las limitaciones de memoria de MS-DOS, para mantener el tamaño del intérprete dentro de un valor razonable, algunos mandatos de uso poco frecuente, como por ejemplo DISKCOPY, están implementados como externos.

Los intérpretes de mandatos de UNIX, denominados *shells*, se engloban en la categoría de intérpretes con mandatos externos. Sin embargo, algunos mandatos se tienen que implementar como internos debido a que su efecto sólo puede lograrse si es el propio intérprete el que ejecuta el mandato. Así, por ejemplo, el mandato cd, que cambia el directorio actual de trabajo del usuario usando la llamada chdir, requiere cambiar a su vez el directorio actual de trabajo del proceso que ejecuta el intérprete, lo cual sólo puede conseguirse si el mandato lo ejecuta directamente el intérprete.

2.12.3. Interfaces gráficas

El auge de las interfaces gráficas de usuario (GUI, Graphical UserInterface) se debe principalmente a la necesidad de proporcionar a los usuarios no especializados una visión sencilla e intuitiva del sistema que oculte toda su complejidad. Esta necesidad ha surgido por la enorme difusión de las computadoras en todos los ámbitos de la vida cotidiana Sin embargo, el desarrollo de este tipo de interfaces más amigables ha requerido un avance considerable en la potencia y capacidad gráfica de las computadoras dada la gran cantidad de recursos que consumen durante su operación.

Las primer experiencias con este tipo de interfaces se remontan a Los primeros años de la década de los setenta. En Xerox PARC (un centro de investigación de Xerox) se desarrolló lo que actualmente se considera la primera estación de trabajo a. la que se denominó Alto. Además de otros muchos avances, esta investigación estableció los primeros pasos en el campo de los GUI.

Con la aparición, a principios de los ochenta, de las computadoras personales dirigidas a usuarios no especializados se acentuó la necesidad de proporcionar este tipo de interfaces. Así la compañía Apple adopto muchas de las ideas de la investigación de Xerox PARC para lanzar su computadora personal [Macintosh, 1984] con una interfaz gráfica que simplificaba enormemente el manejo de la computadora. El otro gran competidor en este campo, el sistema operativo MS tardó bastante más en dar este paso. En sus primeras versiones proporcionaba una. interfaz alfanumérica similar a la de UNIX pero muy simplificada. Como paso intermedio, hacia 1988, incluyo una interfaz denominada. DOS-SHELL que, aunque seguía siendo alfanumérica, no estaba basada en líneas sino que estaba orientada al uso de toda la. pantalla y permitía realizar operaciones mediante menús. Por fin, ya en los noventa, lanzó una interfaz. gráfica, denominada. Windows, que tomaba prestadas muchas de las ideas del Macintosh.

En el mundo UNIX. se produjo una evolución similar. Cada fabricante incluía en su sistema. una interfaz gráfica además de la convencional. La. aparición del sistema de ventanas X a mediados de los ochenta y su aceptación generalizada, que le ha convertido en un estándar de facto, ha permitido que la mayoría de los sistemas UNIX incluyan una interfaz gráfica común. Como resultado de este proceso, prácticamente todas las computadoras de propósito general existentes actualmente poseen una interfaz de usuario gráfica

Hoy en día, este tipo de interfaces tiene su mayor representante en los sistemas operativo Windows de Microsoft. En la Figura 2.18 se muestra uno de los elementos clave de la interfaz gráfica de este tipo de sistemas, el explorador de Windows.

A continuación, se revisan las características comunes de este tipo de interfaces. En primer lugar, todos ellos están basados en ventanas que permiten al usuario trabajar simultáneamente en distintas actividades. Asimismo, se utilizan iconos y menús para representar los recursos del sistema y poder realizar operaciones sobre los mismos, respectivamente. El usuario utiliza. un ratón (o dispositivo equivalente) para interaccionar con estos elementos. Así, por ejemplo, para arrancar una:

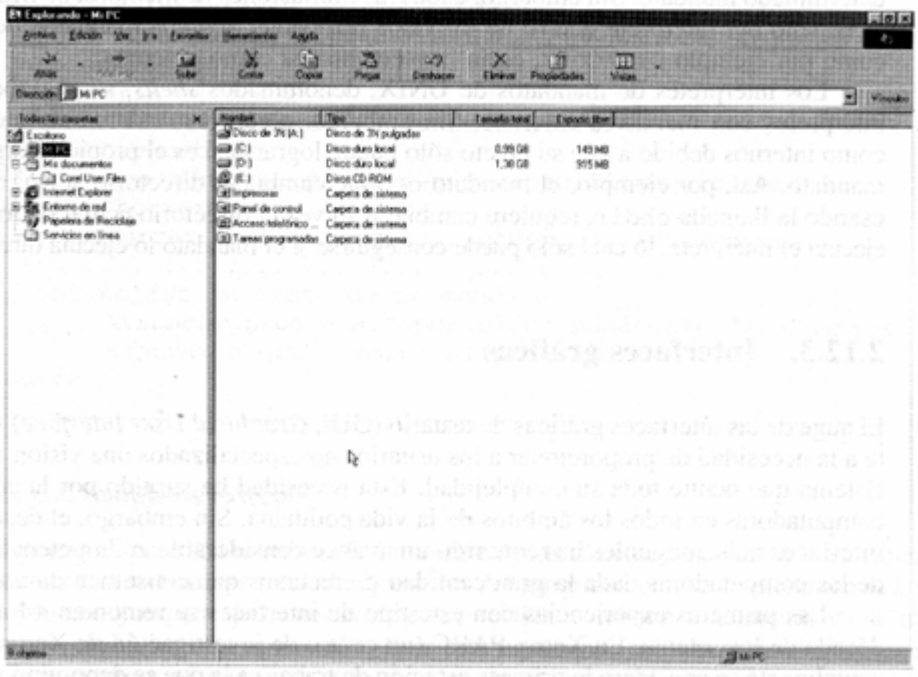


Figura 2.18. Explorador de Windows

aplicación el usuario podría tener que apuntar a un icono con el ratón y apretar un botón del mismo, para copiar un archivo señalar al icono que lo representa y, manteniendo el botón del ratón apretado, moverlo hasta ponerlo encima de un icono que representa el directorio destino. Generalmente, para agilizar el trabajo de los usuarios más avanzados, estas interfaces proporcionan la posibilidad de realizar estas mismas operaciones utilizando ciertas combinaciones de teclas. Dado el carácter intuitivo de estas interfaces, y el amplio conocimiento que posee de ellas todo el inundo, no parece necesario entrar en más detalles sobre su forma de trabajo,

En cuanto a su estructura interna, las interfaces gráficas normalmente están formadas por un conjunto de programas que, usando los servicios del sistema, trabajan conjuntamente para llevar a cabo las peticiones del usuario. Así, por ejemplo, existirá un gestor de ventanas para mantener el. estado de las mismas y permitir su manipulación, un administrador de programas que permita al usuario arrancar aplicaciones, un gestor de archivos que permita manipular archivos y directorios, o una herramienta de configuración de la propia interfaz y del entorno. Observe la diferencia con las interfaces alfanuméricas, en las que existía un programa por cada mandato, Además de la funcionalidad comentada, otros aspectos que conviene resaltar son los siguientes:

- Intercambio de datos entre aplicaciones. Generalmente se le proporciona al usuario un mecanismo del tipo copiar y pegar (copy-and-paste) para poder transferir información entre dos aplicaciones.
- Sistema de ayuda interactivo, Los sistemas de ayuda suelen ser muy sofisticados basándose muchos de ellos en hipertexto.
- Oferta de servicios a las aplicaciones (API gráfico). Además de encargarse de atender al usuario, estos entornos gráficos proporcionan a las aplicaciones una biblioteca de primitivas gráficas que permiten que los programas creen y manipulen objetos gráficos.
- Posibilidad de acceso a la interfaz alfanumérica. Muchos usuarios se sienten encorsetados dentro de la interfaz gráfica y prefieren usar una interfaz alfanumérica para realizar ciertas operaciones. La posibilidad de acceso a dicha interfaz desde el entorno gráfico ofrece al usuario un sistema con lo mejor de los dos mundos.

2.13. HISTORIA DE LOS SISTEMAS OPERATIVOS

Como se decía al comienzo del capítulo, el sistema operativo lo forman un conjunto de programas que ayudan a los usuarios en la explotación de una computadora, simplificando, por un lado, su uso, y permitiendo, por otro, obtener un buen rendimiento de la máquina. Es difícil tratar de dar una definición precisa de sistema operativo, puesto que existen muchos tipos, según sea la aplicación deseada, el tamaño de la computadora usada y el énfasis que se dé a su explotación. Por ello se va a realizar un bosquejo de la evolución histórica de los sistemas operativos, ya que así quedará plasmada la finalidad que se les ha ido atribuyendo.

Se pueden encontrar las siguientes etapas en el desarrollo de los sistemas operativos, que coinciden con las cuatro generaciones de las computadoras.

Prehistoria

Durante esta etapa, que cubre los años cuarenta, se construyen las primeras computadoras. Como ejemplo de computadoras de esta época se pueden citar el ENJAC (*Electronic Numerical Integrator Analyzer and computer),* financiado por el. Laboratorio de Investigación Balística de los Estados Unidos. El ENIAC era una máquina enorme con un peso de 30 toneladas, que era capaz de realizar 5,000 sumas por segundo, 457

multiplicaciones por segundo y 38 divisiones por segundo. Otra computadora de esta época fue el EDVAC (Electronic Discrete Variable Automatic Computer).

En esta etapa no existían sistemas operativos. El usuario debía codificar su programa a mano y en instrucciones máquina, y debía introducirlo personalmente en la computadora, mediante conmutadores o tarjetas perforadas, Las salidas se imprimían o se perforaban en cinta de papel para su posterior impresión. En caso de errores en la ejecución de Los programas, el usuario tenía que depurarlos examinando el contenido de la memoria y los registros de la computadora.

En esta primera etapa todos los trabajos se realizaban en serie Se introducía un programa en la computadora, se ejecutaba y se imprimían los resultados y se repetía este proceso con otros programas. Otro aspecto importante de esta época es que se requería mucho tiempo para preparar y ejecutar un programa, ya que el programador debía encargarse de codificar todo el programa e introducirlo en la computadora de forma manual.

Primera generación (años cincuenta)

Con la aparición de la primera generación de computadoras (años cincuenta) se hace necesario racionalizar su explotación, puesto que ya empieza a haber una base mayor de usuarios. El tipo de operación seguía siendo serie como en el caso anterior, esto es, se trataba un trabajo detrás de otro teniendo cada trabajo las fases siguientes:

- Instalación de cintas o fichas perforadas en los dispositivos periférico En su caso, instalación del papel en la impresora.
- Lectura mediante un programa cargador del programa a ejecutar y de sus datos.
- Ejecución del programa.
- Impresión o grabación de los resultados.
- Retirada de cintas, fichas y papel.

La realización de la primera fase se denominaba montar el trabajo.

El problema básico que abordaban estos sistemas operativos primitivos era optimizar el flujo de trabajos, minimizando el tiempo empleado en retirar un trabajo y montar el siguiente. También empezaron a abordar el problema de la. E/S, facilitando al usuario paquetes de rutinas de BIS, para simplificar la programación de estas operaciones, apareciendo así los primeros manejadores de dispositivos. Se introdujo también el concepto de *system file name*, que empleaba un nombre o número simbólico para referirse a los periféricos, haciendo que su manipulación fuera mucho mas flexible que mediante las direcciones físicas.

Para minimizar el tiempo de montaje de los trabajos, éstos se agrupaban en lotes (*batch*) del mismo tipo (p. ej programas Fortran, programas Cobol, etc), lo que evitaba tener que montar y desmontar las cintas de los compiladores y montadores, aumentando el rendimiento.

En las grandes instalaciones se utilizaban computadoras auxiliares, o satélites, para realizar las funciones de montar y retirar los trabajos. Así se mejoraba el rendimiento de la computadora principal, puesto que se le suministraban los trabajos montados en cinta magnética y éste se limitaba a procesarlos y a grabar los resultados también en cinta magnética. En este caso se decía que la E/S se hacía **fuera de línea** (off-line).

Los sistemas operativos de las grandes instalaciones tenían las siguientes características:

- Procesaban un único flujo de trabajos en lotes.
- Disponían de un conjunto de rutinas de E/S.
- Usaban mecanismos rápidos para pasar de un trabajo al siguiente.
- Permitían la recuperación del sistema si un trabajo acababa en error.
- Tenían un lenguaje de control (le trabajos que permitía especificar los recursos a utilizar las operaciones a realizar por cada trabajo.

Como ejemplos (le sistemas operativos de esta época se pueden citar el FMS (*Fortran System*) e IBYSS, el sistema opera de la IBM 7094.

Segunda generación (años sesenta)

Con la aparición de La segunda generación de computadoras (principios de los sesenta) se hizo más necesario, dada la mayor competencia entre los fabricantes, mejorar la explotación de estas maquinas de altos precios. La **multiprogramación** se impuso en sistemas de lotes como una forma de aprovechar el tiempo empleado en las operaciones de E/S. La base de estos sistemas reside en la gran diferencia. que existe, como se vio en el Capitulo 1, entre las velocidades de los periféricos y de la UCP, por lo que esta última, en las operaciones de E/S, se pasa mucho tiempo esperando a los periféricos. Una forma de aprovechar ese tiempo consiste en mantener varios trabajos simultáneamente en memoria principal (técnica llamada de multiprogramación), y en realizar las operaciones de E/S por acceso directo a. memoria. (Cuando un trabajo necesita una operación de E/S la solicita al sistema operativo que se encarga de:

- Congelar el trabajo solicitante.
- Iniciar la mencionada operación de E/S por DMA.
- Pasar a realizar otro trabajo residente en memoria. Estas operaciones las realiza el. sistema operativo multiprogramado de forma transparente al usuario.

También en esta época aparecen otros modos de funcionamiento muy importantes:

- Se construyen los primeros multiprocesadores, en los que varios procesadores forman una sola máquina de mayores prestaciones
- Se introduce el concepto de **independencia de dispositivo**. El usuario ya no tiene que referirse en sus programas a una unidad de cinta magnética o a una impresora en concreto. Se limita a especificar que quiere grabar un archivo determinado o imprimir unos resultados. El sistema operativo se encarga de asignarle, de forma dinámica, una unidad disponible, y de indicar al operador del sistema la unidad seleccionada, para que éste monte la cinta o el papel correspondiente.
- Comienzan los sistemas de **tiempo compartido** o **timesharing**. Estos sistemas, a los que 4 estamos muy acostumbrados en la actualidad, permiten que varios usuarios trabajen de forma interactiva o conversacional con la computadora desde terminales, que en aquellos días eran teletipos electromecánicos. El sistema operativo se encarga de repartir el tiempo de la UCP entre los distintos usuarios, asignando de forma rotativa pequeños intervalos de tiempo de UCP denominadas **rodajas** (time slice). En sistemas bien dimensionados, cada usuario tiene la impresión. de que la computadora le atiende exclusivamente a él, respondiendo rápidamente a sus órdenes. Aparecen así los primeros planificadores.

- Aparecen, en esta época, los primeros sistemas de tiempo real. Se trata de aplicaciones militares, en concreto para detección de ataques aéreos. En este caso, la computadora están conectadas a un sistema externo y debe responder velozmente a las necesidades de ese sistema externo. En este tipo de sistemas las respuestas deben producirse en períodos de tiempo previamente especificados, que en la mayoría de los casos son pequeños. Los primeros sistemas de este tipo se construían en ensamblador y ejecutaban sobre maquina desnuda (Sección 2.1.1), lo que hacía de estas aplicaciones sistemas muy complejos

Finalmente, cabe mencionar que Burroughs introduce, en 1963, el «Master Control Program», que además de ser multiprograma y multiprocesador incluía memoria virtual y ayudas para depuración en lenguaje fuente.

Durante esta época se desarrollaron, entre otros, los siguientes sistemas operativos: el. CTSS [CORBATO 1962], desarrollado en el MIT y que fue el primer sistema de tiempo compartido. Este sistema operativo se utilizó en un IBM 7090 y llegó a manejar hasta 32 usuarios interactivos. El OS/360 [Mealy, 1966], sistema operativo utilizado en las máquinas de la línea 360 de IBM y el sistema MULTICS [Organick, 1972], desarrollado en el MIT con participación de los laboratorios Bell y que evolucionó posteriormente para convertirse en el sistema operativo UNIX MULTICS fue diseñado para dar soporte a cientos de usuarios; sin embargo, aunque una versión primitiva de este sistema operativo ejecutó en 1969 una computadora GE 645, no proporcionó los servicios para los que fue diseñada y los laboratorios Bell finalizaron su participación en el proyecto.

Tercera generación (años setenta)

La tercera generación es la época de los sistemas de propósito general y se Caracteriza por los sistemas operativos multimodo de operación, esto es, capaces de operar en lotes, en multiprogramación, en tiempo real, en tiempo compartido y en modo multiprocesador. Estos sistemas operativos fueron costosísimos de realizar e interpusieron entre el usuario y el hardware una gruesa capa de software, de forma que éste sólo veía esta capa, sin tenerse que preocupar de los detalles de la circuitería.

Uno de los inconvenientes de estos sistemas operativos era su complejo lenguaje de control, que debían aprenderse los usuarios para preparar sus trabajos, puesto que era necesario especificar multitud de detalles y opciones. Otro de los inconvenientes era el gran consumo de recursos que ocasionaban, esto es, los grandes espacios de memoria principal y secundaria ocupados, así corno el tiempo de UCP consumido, que en algunos casos superaba el 50 por 100 del tiempo total.

Esta década fue importante por la aparición de dos sistemas que tuvieron una gran difusión, UNIX [Bach, 1986] y MVS [Samson,1990] de IBM De especial importancia fue UNIX, desarrollado en los laboratorios Bell para una PDP-7 en 1970. Pronto se transportó a una PDP- 11, para lo cual se rescribió utilizando el lenguaje de programación C. Esto fue algo muy importante en la historia de los sistemas operativos, ya que hasta la fecha ninguno se había escrito utilizando un lenguaje de alto nivel, recurriendo para ello a los lenguajes ensambladores propios de cada arquitectura. Sólo una pequeña parte de UNIX, aquella que accedía de forma directa al hardware, siguió escribiéndose en ensamblador. La programación de un sistema operativo utilizando un lenguaje de alto nivel como es C hace que un sistema operativo sea fácilmente transportable a una. Amplia gama de computadoras. En la actualidad, prácticamente todos los sistemas operativos se escriben en lenguajes de alto nivel, fundamentalmente en C.

La primera versión ampliamente disponible de UNIX fue la versión 6 de los laboratorios Be1l,. que apareció en 1976. A ésta le siguió la versión 7 distribuida en 1978, antecesora de prácticamente de todas las versiones modernas de UNIX. En 1982 aparece

una versión de UNIX desarrollada por la Universidad de California en Berkeley, la cual se distribuyó como la versión BSD (Berkeley Software .Distribution) de UNIX Esta versión de UNIX introdujo mejoras importantes como la inclusión de memoria virtual y la interfaz de *sockets* para la programación de aplicaciones sobre protocolos TCP/IP.

Más tarde, AT&T (propietario de los laboratorios Beill) distribuyó la versión de UNIX conocida como Systern V o RVS4. Desde entonces muchos han sido los fabricantes de computadoras que has adoptado a UNIX como sistema operativo de sus máquinas. Ejemplos de estas versiones son: Solaris de SUN, HP UNIX, IRIX de SGI y AIX de IBM.

Cuarta generación (años ochenta hasta la actualidad)

La cuarta generación se caracteriza por una evolución de los sistemas operativos de propósito general de la tercera generación, tendente a su especialización, a su simplificación y a dar más importancia a la productividad de usuario que al rendimiento de la máquina.

Adquiere cada vez más importancia el tema de las redes de computadoras, tanto redes de largo alcance como locales. En concreto, la disminución del coste del hardware hace que se difunda el **proceso distribuido**, en contra, de la tendencia centralizadora anterior. El proceso distribuido consiste en disponer de varias computadoras, cada una situada en el lugar de trabajo de las personas que la emplean, en lugar de una única central. Estas computadoras suelen estar unidas mediante una red, de forma que puedan compartir información y periféricos.

Se difunde el concepto d **máquina virtual**, consistente en que una computadora X, incluyen do su sistema operativo, sea simulada por otra computadora Y. Su ventaja es que permite ejecutar. en la computadora Y, programas preparados para la computadora X, lo que posibilita el empleo de .software elaborado para la computadora X, sin necesidad de disponer de dicha. computadora.

Durante esta época, los sistemas de bases de datos sustituyen a los archivos en multitud de aplicaciones. Estos sistemas se diferencian de un conjunto de archivos en que sus datos están estructurados de tal forma que permiten acceder a la información de diversas maneras, evitar datos redundantes y mantener su integridad y coherencia.

La difusión de las computadoras personales ha traído una humanización en los sistemas informáticos. Aparecen los sistemas «amistosos» o ergonómicos, en los que se evita que el usuario tenga que aprenderse complejos lenguajes de control, sustituyéndose éstos por los sistemas dirigidos por menú, en los que la selección puede incluso hacerse mediante un manejador de cursor. En estos sistemas, de orientación monousuario, el objetivo primario del sistema operativo ya no es aumentar el rendimiento del sistema, sino la productividad del usuario En este sentido, la tendencia actual consiste en utilizar sistemas operativos multiprogramados sobre los que se añade un gestor de ventanas, lo que permite que el usuario tenga activas, en cada momento, tantas tareas como desee y que las distribuya a su antojo sobre la superficie del terminal

Los sistemas operativos que dominaron el campo de las computadoras personales fueron UNIX, MS-DOS y los sucesores de Microsoft para este sistema: Windows 95/98, Windows NT y Windows 2000. La primera, versión de Windows NT (versión 3.1) apareció en 1993 e incluía la misma interfaz de usuario que Windows 3.1, En 1996 aparece la versión 4.0, que se caracterizó por la inclusión dentro del ejecutivo de Windows NT de diversos componentes gráficos que ejecutaban anteriormente modo usuario. Durante el año 2000, Microsoft distribuye la versión denominada Windows 2000.

También ha tenido importancia durante esta época eL desarrollo de Linux. Linux es un sistema operativo similar a UNIX, desarrollado de forma desinteresada durante la década de los noventa por miles de voluntarios conectados a Internet. Linux está creciendo fuertemente debido sobre todo a su bajo coste y a su gran estabilidad, comparable a cualquier otro sistema UNIX. Una de las principales características de Linux es que su código fuente esta disponible, lo que le hace especialmente atractivo para el estudio de la estructura interna de sistema operativo. Su aparición ha tenido también mucha importancia en el mercado del software ya que ha hecho que se difunda el concepto de software libre.

Durante esta etapa se desarrollan también los **sistemas operativos de tiempo real**, encargados de ofrecer servicios especializados para el desarrollo de aplicaciones de tiempo real. Algunos ejemplos son: QNX [QNX, 1997], RTEMS y VRTX [Ready, 1986].

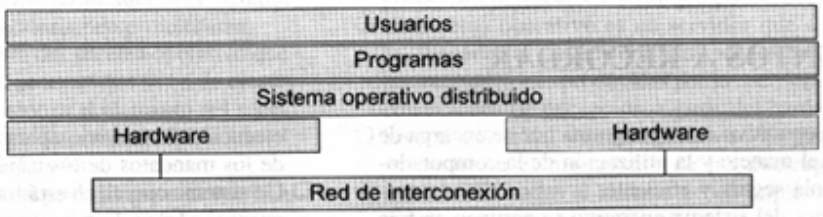


Figura 2.19 *Estructura de un sistema distribuido que utiliza un sistema operativo distribuido*

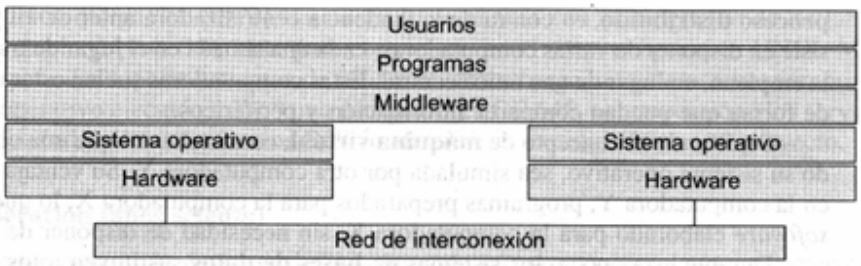


Figura 2.20. *Estructura de un sistema distribuido que emplea un middleware.*

A mediados de los ochenta aparecen los **sistemas operativos distribuidos**. Un sistema operativo distribuido es un sistema operativo (Fig. 2.19) común utilizado en una serie de computadoras conectadas por una red, Este tipo de sistemas aparece al usuario como un único sistema operativo centralizado, haciendo por tanto más fácil el uso de una red de computadoras. Un sistema operativo de este tipo sigue teniendo las mismas características que un sistema operativo convencional pero aplicadas a un sistema distribuido. Como ejemplo de sistemas operativos distribuidos se puede citar:
Mach [Accetta, 1986], Chorus [Roizer, 1988] y Amoeba [Mullender, 1990],

Los sistemas operativos distribuidos han dejado de tener importancia y han evolucionado durante la década de los noventa a lo que se conoce como *middleware.* Un *middleware* (Fig. 2.20) es una capa de software que se ejecuta sobre un sistema operativo ya existente y que se encarga de gestión un sistema distribuido. En este sentido, presenta las mismas funciones que un sistema operativo distribuido. La diferencia es que ejecuta sobre sistemas operativos ya existentes que pueden ser además distintos, lo que hace más atractiva su utilización. Dos de los *middleware* más importantes de esta década han sido

DCE [Rosenberry, 1992] y CORBA [Otte, 1996]. Microsoft también ofrece su propio *middleware* conocido como DCOM [Rubin, 1999].

En cuanto a las interfaces de programación, durante esta etapa tiene importancia el desarrollo del estándar POSIX. Este estándar persigue que las distintas aplicaciones que hagan uso de los servicios de un sistema operativo sean potables sin ninguna dificultad a distintas plataformas con sistemas operativos diferentes. Cada vez es mayor el número de sistemas operativos que ofrecen esta interfaz. Otra de las interfaces de programación mas utilizada es la interfaz Win32, interfaz de los sistemas operativos Windows *95/98,* Windows NT y Windows 2000.

En el futuro próximo, la evolución de los sistemas operativos se va a orientar hacia las plataformas distribuidas y la computación móvil. Gran importancia tendrá la construcción de sistemas operativos y entornos que permitan utilizar estaciones de trabajo heterogéneas (computadoras de diferentes fabricantes con sistemas operativos distintos) conectadas por redes de interconexión, como una gran máquina centralizada, lo que permitirá disponer de una mayor capacidad de cómputo y facilitará el trabajo cooperativo.

2.14. PUNTOS A RECORDAR

- Un sistema operativo es un programa que se encarga de simplificar el manejo y la utilización de la computadora, haciéndola segura y eficiente.
- Las funciones del sistema operativo se agrupan en tres categorías: gestión de los recursos de la computadora, ejecución de servicios para los programas y ejecución de los mandatos de los usuarios.
- Un sistema operativo está formado por tres grandes bloques: el núcleo, la capa de servicios o llamadas al sistema y el intérprete de mandatos o *shell.*

- Los servicios se suelen agrupar según su funcionalidad en varios componentes, cada uno de los cuales se ocupa de las siguientes funciones: gestión de procesos, gestión de memoria, gestión de E/S, gestión de archivos y directorios, comunicación y sincronización entre procesos, y seguridad y protección.
- El iniciador ROM es el programa de arranque que se encuentra en memoria ROM y cuyas funciones son: comprobar el sistema, leer y almacenar en la memoria de la computadora el programa cargador del sistema operativo, y por último ceder el control a este programa.
- El cargador del sistema operativo tiene por misión traer a memoria principal algunos de los componentes del sistema operativo.
- En un sistema operativo monolítico, todos sus componentes se encuentran integrados en un único programa, que ejecuta en un único espacio de direcciones. Todas las funciones que ofrece el sistema operativo se ejecutan en modo núcleo.
- En un sistema por capas, el sistema operativo se organiza como una jerarquía de capas, donde cada capa ofrece una interfaz clara y bien definida a la superior y solamente utiliza los servicios ofrecidos por la capa inmediatamente inferior.
- En un sistema operativo con modelo cliente-servidor, la mayor parte de los servicios y funciones del sistema operativo se implementan en procesos de usuario, dejando sólo una pequeña parte del sistema operativo ejecutando en modo núcleo.
- Un proceso es un programa en ejecución. La información que compone un proceso es la siguiente: imagen de memoria, estado del procesador y contenido del bloque de control de proceso (BCP). El BCP es una estructura de datos que almacena diversa información sobre el proceso.
- Los sistemas operativos ofrecen servicios para crear procesos, ejecutarlos y matarlos.
- Un sistema operativo monotarea o monoproceso sólo permite que exista un proceso en cada instante. Por el contrario, un sistema operativo multitarea o multiproceso permite que coexistan varios procesos activos a la vez.
- Un sistema monousuario sólo permite dar soporte a un usuario. En un sistema operativo multiusuario, el sistema operativo da soporte a varios usuarios trabajando simultáneamente desde varios terminales. Un sistema multiusuario ha de ser obligatoriamente multitarea.
- El planificador es el elemento del sistema operativo que se encarga de seleccionar el proceso que se ha de ejecutar a continuación. El activador es el elemento del sistema operativo que se encarga de poner en ejecución el proceso elegido por el planificador.
- El gestor de memoria se encarga de asignar memoria a los procesos para crear su imagen de memoria, proporcionar memoria a los procesos cuando la solicitan y liberarla cuando lo requieran. También se ocupa de tratar los posibles errores en el acceso a memoria, permitir que los procesos puedan compartir memoria, y gestionar la jerarquía de memoria y los fallos de página en sistemas con memoria virtual.
- El gestor de E/S controla el funcionamiento de todos los dispositivos de E/S, proporcionando una interfaz que permite utilizar los periféricos de forma sencilla y uniforme, y ofreciendo mecanismos de protección que impidan a los usuarios acceder sin control a los dispositivos periféricos.
- El servidor de archivos ofrece al usuario una visión lógica compuesta por una serie de objetos (archivos y directorios) identificables por un nombre lógico sobre los que puede realizar una serie de operaciones. La visión física incluye los detalles de cómo están almacenados estos objetos en los periféricos.
- El sistema de archivos es el conjunto de archivos incluidos en una unidad de disco. Está compuesto por los datos de los archivos y por metainformación. La metainformación es toda la información auxiliar que necesita el sistema de archivos para representar los archivos y directorios.
- El sistema operativo ofrece mecanismos de comunicación que permiten transferir cadenas de bytes entre procesos. También ofrece mecanismos de sincronización que permiten que los procesos se bloqueen y se despierten.
- La seguridad en un sistema operativo se encarga de garantizar la identidad de los usuarios y de definir lo que puede hacer cada uno de ellos. El primer aspecto se conoce como autenticación y el segundo se consigue mediante privilegios.
- El objetivo de la autenticación es determinar que un usuario es quien dice ser. El modo tradicional de autenticación se basa en el uso de contraseñas (*passwords*).
- Los privilegios en un sistema operativo especifican los recursos que puede acceder cada usuario. La información de los privilegios se puede asociar a los recursos o a los usuarios.
- Una lista de control de acceso especifica los usuarios y grupos que pueden acceder a un recurso.
- Una capacidad o *capability* asocia a cada usuario o grupo la lista de recursos a los que puede acceder.
- El sistema operativo es un servidor que se activa con las llamadas al sistema emitidas por los programas, con las interrupciones producidas por los periféricos y con las condiciones de excepción o error del hardware.
- La activación del sistema operativo sólo puede realizarse mediante el mecanismo de las interrupciones, por tanto las llamadas al sistema deben implementarse mediante una instrucción TRAP.

2.15. ¿Es un proceso un archivo ejecutable? Razone su respuesta.

2.16. ¿Debe ser un sistema operativo multitarea de tiempo compartido? ¿Y viceversa? Razone su respuesta.

2.17. ¿Qué diferencias existen entre un archivo y un directorio?

2.18. ¿Qué ventajas considera que tiene escribir un sistema operativo utilizando un lenguaje de alto nivel?

2.19. ¿En qué época se introdujeron los primeros manejadores de dispositivos? ¿Y los sistemas operativos de tiempo compartido?

3
Procesos

Este capítulo se dedica a estudiar todos los aspectos relacionados con los procesos, conceptos fundamentales para comprender el funcionamiento de un sistema operativo. Los temas que se cubren en este capítulo son:

- *Procesos.*
- *Multitarea.*
- *Estructuras de información de los procesos.*
- *Estados del proceso.*
- *Procesos ligeros.*
- *Planificación.*
- *Señales y excepciones.*
- *Temporizadores.*
- *Servidores y demonios.*
- *Servicios POSIX y Win32 relacionados con la gestión de procesos.*

3.1CONCEPTO DE PROCESO

Todos los programas, cuya ejecución solicitan los usuarios, se ejecutan en forma de procesos, de ahí la importancia para el informático de conocerlos en detalle. Como ya se vio en el Capítulo 2, el proceso se puede definir como un **programa en ejecución y,** de una forma un poco más precisa, como la unidad de procesamiento gestionada por el sistema operativo.

En el Capítulo 1 se vio que para que un programa pueda ser ejecutado ha de residir con sus datos en memoria principal. Observe que durante su ejecución el proceso va modificando los registros del modelo de programación de la computadora, de acuerdo a las instrucciones de máquina involucradas (Fig. 1.3).

El sistema operativo mantiene por cada proceso una serie de estructuras de información que permiten identificar las características de éste, así como los recursos que tiene asignados. En esta última categoría entran los descriptores de los segmentos de memoria asignados, los descriptores de los archivos abiertos, los descriptores de los puertos de comunicaciones, etc.

Una parte muy importante de estas informaciones se encuentra en el llamado **bloque de control del proceso (BCP).** El sistema operativo mantiene una tabla de procesos con todos los BCP de los procesos. Por razones de eficiencia, la tabla de procesos se construye normalmente como una estructura estática, que tiene un determinado número de BCP, todos ellos del mismo tamaño. El contenido del BCP se analizará con más detalle en secciones posteriores; sin embargo, de manera introductoria se puede decir que la información que compone un proceso es la siguiente:

- Contenido de los segmentos de memoria en los que residen el código y los datos del proceso. A esta información se le denomina imagen de memoria o *core image.*
- Contenido de los registros del modelo de programación.
- Contenido del BCP.

Es de destacar que el proceso no incluye información de E/S, puesto que ésta suele estar reservada al sistema operativo.

Jerarquía de procesos

La secuencia de creación de procesos vista en la Sección 2.2 genera un árbol de procesos como el incluido en la Figura 3.1.

Para referirse a las relaciones entre los procesos de la jerarquía se emplean los términos de padre, hijo. hermano o abuelo. Cuando el proceso A solicita al sistema operativo que cree el proceso B. se dice que A es padre de B y que B es hijo de A. Bajo esta óptica, la jerarquía de procesos puede considerarse como un árbol genealógico.

Algunos sistemas operativos, como UNIX, mantienen de forma explícita esta estructura jerárquica de procesos —un proceso sabe quién es su padre—, mientras que otros sistemas operativos como el Windows NT no la mantienen.

Entorno del proceso

El entorno del proceso consiste en un conjunto de variables que se le pasan al proceso en el momento de su creación.

El entorno está formado por una tabla NOMBRE-VALOR que se incluye en la pila del proceso. El NOMBRE especifica el nombre de la variable y el VALOR su valor. Un ejemplo de entorno en UNIX es el siguiente:

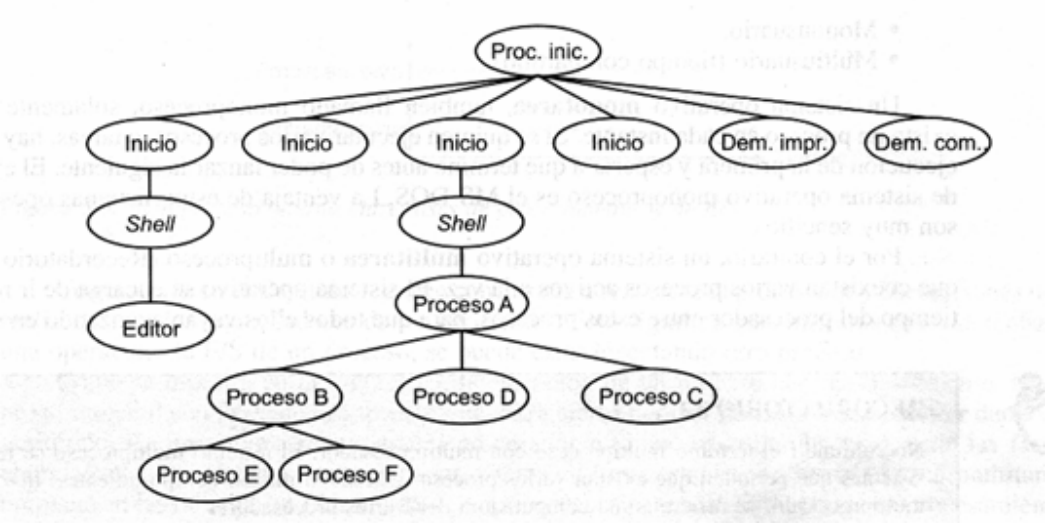


Figura 3.1 Jerarquía de proceso

```
PATH=/ USR/ bin: / home / pepe / bin
TERM= vt 100
HOME= /home/pepe
PWD= / home /pepe /libros / primero
```

En este ejemplo, PATH indica la lista de directorios en los que el sistema operativo busca los programas ejecutables, TERM el tipo de terminal, HOME el directorio inicial asociado al usuario y PWD cl directorio de trabajo actual.

Los procesos pueden utilizar las variables del entorno para definir su comportamiento. Por ejemplo, un programa de edición responderá a las teclas de acuerdo al tipo de terminal que esté utilizando el usuario, que viene definido en la variable TERM.

Grupos de procesos

Los procesos forman grupos que tienen diversas propiedades. El conjunto de procesos creados a partir de un *shell* puede formar un grupo de procesos. También pueden formar un grupo los procesos dependientes de un terminal.

El interés del concepto de grupo de procesos es que hay determinadas operaciones que se pueden hacer sobre todos los procesos de un determinado grupo, como se verá al estudiar algunos de los servicios. Un ejemplo es la posibilidad de matar a todos los procesos pertenecientes a un mismo grupo.

3.2. MULTITAREA

Como ya se vio en el capítulo anterior, dependiendo del número de procesos y de usuarios que puedan ejecutar simultáneamente, un sistema operativo puede ser:

- Monotarea o monoproceso.
- Multitarea o multiproceso.

- Monousuario.
- Multiusuario (tiempo compartido).

Un sistema operativo **monotarea,** también llamado monoproceso, solamente permite que exista un proceso en cada instante. Si se quieren ejecutar varios procesos, o tareas, hay que lanzar la ejecución de la primera y esperar a que termine antes de poder lanzar la siguiente. El ejemplo típico de sistema operativo monoproceso es el MS-DOS. La ventaja de estos sistemas operativos es que son muy sencillos.

Por el contrario, un sistema operativo **multitarea** o multiproceso (Recordatorio 3. 1) permite que coexistan varios procesos activos a la vez. El sistema operativo se encarga de ir repartiendo el tiempo del procesador entre estos procesos, para que todos ellos vayan avanzando en su ejecución.

RECORDATORIO 3.1

No confundir el término multiproceso con multiprocesador. El término multiproceso se refiere a los sistemas que permiten que existan varios procesos activos al mismo tiempo, mientras que el término multiprocesador se refiere a una computadora con varios procesadores.

Un sistema **monousuario** está previsto para dar soporte a un solo usuario. Estos sistemas pueden ser monoproceso o multiproceso. En este último caso el usuario puede solicitar varias tareas al mismo tiempo, por ejemplo, puede estar editando un archivo y, simultáneamente, puede estar accediendo a una página Web de la red.

El sistema operativo **multiusuario** da soporte a varios usuarios que trabajan simultáneamente desde varios terminales. A su vez, cada usuario puede tener activos más de un proceso, por lo que el sistema, obligatoriamente, ha de ser multitarea. Los sistemas multiusuario reciben también el nombre de **tiempo compartido,** puesto que el sistema operativo ha de repartir el tiempo de la computa-. dora entre los usuarios, para que las tareas de todos ellos avancen de forma razonable.

La Figura 3.2 recoge estas alternativas.

3.2.1. Base de la multitarea

La multitarea se basa en las tres características siguientes:

• Paralelismo real entre E/S y procesador.
• Alternancia en los procesos de fases de E/S y de procesamiento.
• Memoria principal capaz de almacenar varios procesos.

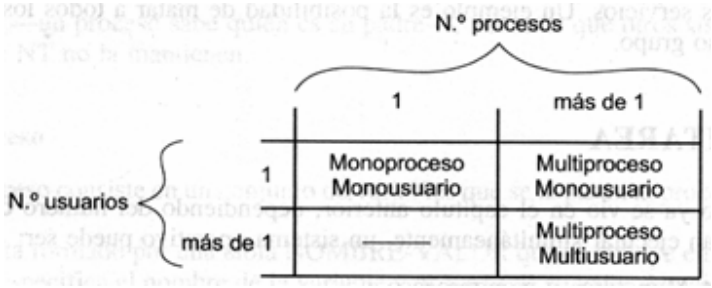


Figura 3.2. Tipos de sistemas operativos en función del numero de procesos y usuarios

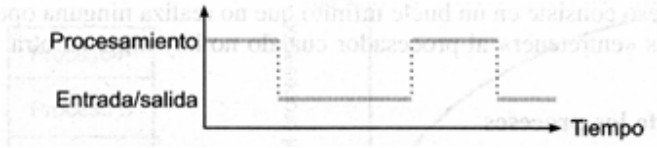


Figura **3.3.** *Un proceso Alterna entre fases de procesamiento y de E/S.*

Como se vio en el Capítulo 1, existe concurrencia real entre el procesador y las funciones de F/S realizadas por los controladores de los periféricos. Esto significa que, mientras se están realizando una operación de FIS de un proceso, se puede estar ejecutando otro proceso.

Como se muestra en la Figura 3.3, la ejecución de un proceso alterna, típicamente, fases de procesamiento con fases de *FIS,* puesto que, cada cierto tiempo, necesita leer o escribir datos en un periférico. En un sistema monotarea el procesador no tiene nada que hacer durante las fases de entrada/salida, por lo que desperdicia su potencia de procesamiento. En un sistema multitarea se aprovechan las fases de entrada/salida de unos procesos para realizar las fases de procesamiento de otros.

La Figura 3.4 presenta un ejemplo de ejecución multitarea con tres procesos activos. Observe que, al finalizar la primera fase de procesamiento del proceso A, hay un intervalo de tiempo en el que no hay trabajo para el procesador.

Como muestra la figura, el sistema operativo entra a ejecutar al final de las fases de procesamiento y al final de las fases de entrada/salida. Esto es así puesto que las operaciones de F/S no las gobiernan directamente los procesos, sino que se limitan a pedirle al sistema operativo que las realice. Del mismo modo, el sistema operativo trata las interrupciones que generan los controladores para avisar que han completado una operación.

Finalmente es importante destacar que la multitarea exige tener más de un proceso activo y cargado en memoria principal. Por tanto, hay que disponer de suficiente memoria principal para albergar a estos procesos.

Proceso nulo

Como se indicó en el Capítulo 1, el procesador no para de ejecutar nunca. Esto parece que contradice a la Figura 3.4, puesto que muestra un intervalo en el que el procesador no tiene nada que hacer. Para evitar esta contradicción, los sistemas operativos incluyen el denominado proceso nulo.

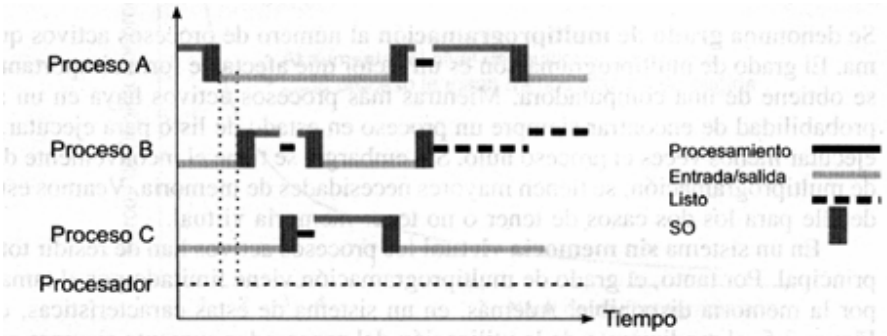


Figura 3.4 Ejemplo de ejecución en un sistema multitarea

Este proceso consiste en un bucle infinito que no realiza ninguna operación útil. El objetivo de este proceso es «entretener» al procesador cuando no hay ninguna otra tarea.

Estados de los procesos

De acuerdo con la Figura 3.4, un proceso puede estar en varias situaciones (procesamiento, listo para ejecutar y espera), que denominaremos estados. A lo largo de su vida, el proceso va cambiando de estado según evolucionan sus necesidades. En la Sección 3.5 se describirán con mayor detalle los estados de un proceso.

Planificador y activador

El planificador *(scheduler)* forma parte del núcleo del sistema operativo. Entra en ejecución cada vez que se activa el sistema operativo y su misión es seleccionar el proceso que se ha de ejecutar a continuación.
El activador *(dispatcher)* también forma parte del sistema operativo y su función es poner en ejecución el proceso seleccionado por el planificador.

3.2.2. Ventajas de la multitarea

La multiprogramación presenta varias ventajas, entre las que se pueden resaltar las siguientes:

- Facilita la programación. Permite dividir las aplicaciones en varios procesos, lo que beneficia a su modularidad.
- Permite prestar un buen servicio, puesto que se puede atender a varios usuarios dc forma eficiente, interactiva y simultánea.
- Aprovecha los tiempos muertos que los procesos pasan esperando a que se completen sus operaciones de E/S.

- Aumenta el uso de la UCP, al aprovechar los espacios de tiempo que los procesos están bloqueados.

Todas estas ventajas hacen que, salvo para situaciones muy especiales, no se conciba actualmente un sistema operativo que no soporte multitarea.

3.2.3. Grado de multiprogramación y necesidades de memoria principal

Se denomina **grado de multiprogramación** al número de procesos activos que mantiene un sistema. El grado de multiprogramación es un factor que afecta de forma importante el rendimiento que se obtiene de una computadora. Mientras más procesos activos haya en un sistema, mayor es la probabilidad de encontrar siempre un proceso en estado de listo para ejecutar, por lo que entrará a ejecutar menos veces el proceso nulo. Sin embargo, se tiene el inconveniente de que, a mayor grado de multiprogramación, se tienen mayores necesidades de memoria. Veamos este fenómeno con mas detalle para los dos casos de tener o no tener memoria virtual.

En un sistema **sin memoria virtual** los procesos activos han de residir totalmente en memoria principal. Por tanto, el grado de multiprogramación viene limitado por el tamaño de los procesos y por la memoria disponible. Además, en un sistema de estas características, como se indica en la Figura 3.5, el rendimiento de la utilización del procesador aumenta siempre con el grado de multiprogramación. Esto es así ya que los procesos siempre residen en memoria principal.

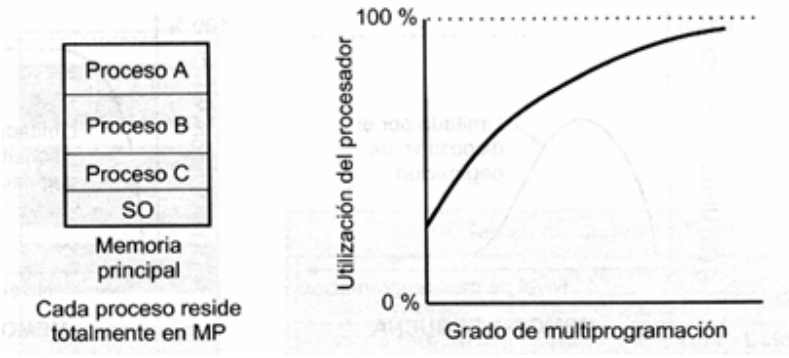


Figura *3.5. Grado de multiprogramación y utilización del procesador.*

En los sistemas **con memoria virtual la** situación es más compleja, puesto que los procesos sólo tienen en memoria principal su conjunto residente (Recordatorio 32), lo que hace que quepan mas procesos. Sin embargo, al aumentar el número de procesos disminuye el conjunto residente de cada uno, situación que se muestra en la Figura 3.6.

Cuando el conjunto residente de un proceso se hace menor de un determinado valor ya no representa adecuadamente al futuro conjunto de trabajo (Recordatorio 3.2) del proceso, lo que tiene como consecuencia que se produzcan muchos fallos de página.

Cada fallo de página consume tiempo de procesador, porque el sistema operativo ha de tratar el fallo, y tiempo de *FIS,* puesto que hay que hacer una migración de páginas. Todo ello hace que, al crecer los fallos de páginas, el sistema dedique cada vez más tiempo al improductivo trabajo de resolver estos fallos de página.

La Figura 3.7 muestra que, en un sistema con memoria virtual, el aumento del grado de multiprogramación conlleva primero un aumento del rendimiento del procesador. Sin embargo, superado un determinado valor de grado de multiprogramación los conjuntos residentes de los procesos empiezan a ser demasiado pequeños, por lo que el sistema baja su rendimiento al perder el tiempo paginando.

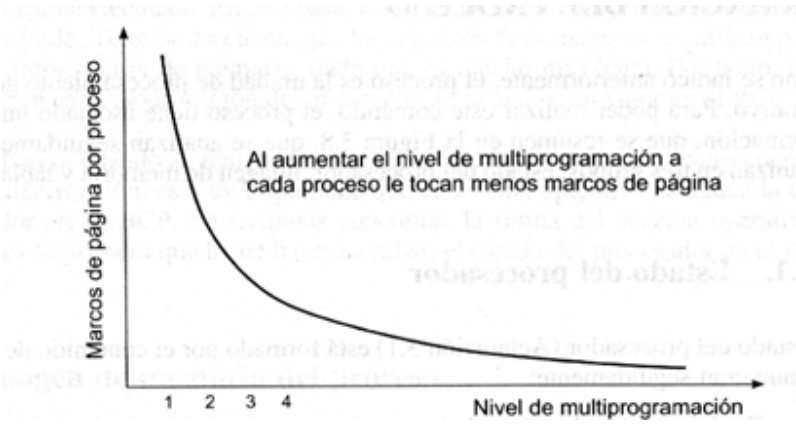


Figura 3.6. *El conjunto residente medio decrece con el grado de multiprogramación*

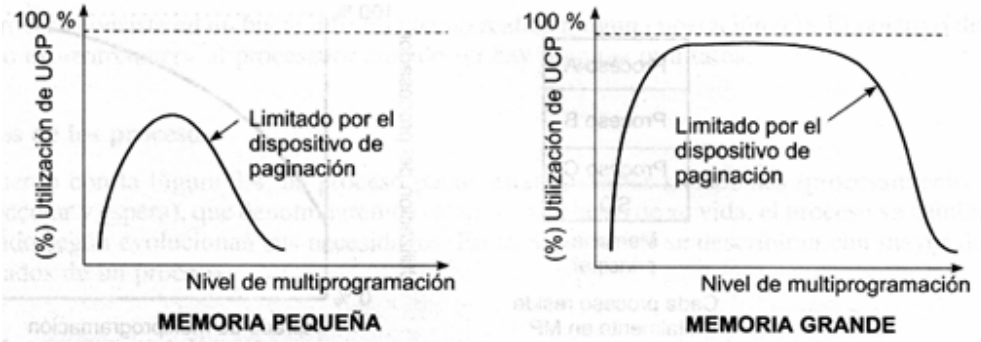


Figura 3.7. *Rendimiento del procesador* y *grado de multiprogramación.*

Se denomina **hiperpaginación** *(trashing)* a la situación de alta paginación producida cuando los conjuntos residentes de los procesos son demasiado pequeños.

RECORDATORIO 3.2

Se denomina conjunto residente a las páginas que un proceso tiene en memoria principal. El conjunto de trabajo de un proceso está formado por el conjunto de páginas que un proceso está actualmente utilizando.

Cuando la memoria principal disponible es pequeña, se llega a la situación de hiperpaginación antes de alcanzar una cota alta de utilización del procesador. Para aumentar el rendimiento de un sistema que esté en esta situación es necesario añadir más memoria principal. Cuando la memoria es grande se llega a saturar el procesador con menos procesos de los que caben en memoria. En este caso se puede aumentar el rendimiento del sistema manteniendo la memoria pero aumentando la potencia del procesador o añadiendo otro procesador.

3.3. INFORMACIÓN DEL PROCESO

Como se indicó anteriormente, el proceso es la unidad de procesamiento gestionada por el sistema operativo. Para poder realizar este cometido, el proceso tiene asociado una serie de elementos de información, que se resumen en la Figura 3.8, que se analizan seguidamente. Estos elementos se organizan en tres grupos: estado del procesador, imagen de memoria y tablas del sistema operativo.

3.3.1. Estado del procesador

El estado del procesador (Aclaración 3.1) está formado por el contenido de todos sus registros, que se enumeran seguidamente:

- Registros generales. De existir registros específicos de coma flotante también se incluyen aquí.
- Contador de programa.

Figura 3.8. *Información de un proceso.*

- Puntero de pila.

• Registro o registros de estado.
• Registros especiales. Como puede ser el RIED (registro identificador de espacio de direccionamiento).

ACLARACIÓN 3.1

No confundir el estado del procesador con el estado del proceso.

El estado del procesador de un proceso reside en los registros del procesador, cuando el proceso está en ejecución, o en el bloque de control de proceso (BCP), cuando el proceso no está en ejecución.

Cuando el proceso está ejecutando, el estado del procesador varía de acuerdo al flujo de instrucciones maquina ejecutado. En este caso, la copia del estado del procesador que reside en el BCP no está actualizada. Téngase en cuenta que los registros de la máquina se utilizan para no tener que acceder a la información de memoria, dado que es mucho más lenta. Por tanto, no tiene sentido plantear que, en cada modificación de un registro, se actualice su valor en el BCP, puesto que está en memoria.

Sin embargo, cuando se detiene la ejecución de un proceso, como consecuencia de la ejecución dc una interrupción, es muy importante que el sistema operativo actualice la copia del estado del procesador en su BCP. En términos concretos, la rutina del sistema operativo que trata las interrupciones lo primero que ha de hacer es salvar el estado del procesador en el BCP del proceso interrumpido.

3.3.2. Imagen de memoria del proceso

La imagen de memoria del proceso está formada por los espacios de memoria que está autorizado a utilizar. Las principales características de la imagen de memoria son las siguientes:

- El proceso solamente puede tener información en su imagen de memoria y no fuera de ella. Si genera una dirección que esté fuera de ese espacio, el hardware de protección deberá detectarlo y levantar una excepción. Esta excepción activará la ejecución del sistema operativo que se encargará de tomar las acciones oportunas, que por lo general consistirán en abortar ¡a ejecución del proceso.
- Dependiendo de la computadora, la imagen de memoria estará referida a memoria virtual o a memoria física. Observe que esto es transparente (irrelevante) para el proceso, puesto que él genera direcciones que serán virtuales o físicas según el caso.
- Los procesos suelen necesitar asignación dinámica de memoria. Por tanto, la imagen de memoria de los mismos se deberá adaptar a estas necesidades, creciendo o decreciendo adecuadamente.

- No hay que confundir la asignación de memoria con la asignación de marcos de memoria. El primer término contempla la modificación de la imagen de memoria y se refiere a espacio virtual en los sistemas con este tipo de espacio. El segundo sólo es de aplicación en los ¡sistemas con memoria virtual y se refiere a la modificación del conjunto residente del proceso.

El sistema operativo asigna la memoria al proceso, para lo cual puede emplear distintos modelos de imagen de memoria, que se analizan seguidamente.

Imagen de memoria con un único segmento de tamaño fijo

Este es el modelo más sencillo de imagen de memoria y su uso se suele restringir a los sistemas sin memoria virtual. El proceso recibe un único espacio de memoria que, además, no puede variar de tamaño.

Proceso con un único segmento de tamaño variable

Se puede decir que esta solución no se emplea. En sistemas sin memoria virtual los segmentos no pueden crecer a menos que se deje espacio de memoria principal de reserva; se chocaría con otro proceso. Ahora bien, la memoria principal es muy cara como para dejarla de reserva. En sistemas Con memoria virtual sí se podría emplear, pero es más conveniente usar un modelo de varios segmentos, pues es mucho más flexible y se adapta mejor a las necesidades reales de los procesos.

Proceso con un número fijo de segmentos de tamaño variable

Un proceso contiene varios tipos de información, cuyas características se analizan seguidamente:

- **Texto o código.** Bajo este nombre se considera el programa máquina que ha dc ejecutar el proceso. Aunque el programa podría automodificarse. no es esta una práctica recomendada, por lo cual se considerará que esta información es fija y que solamente se harán operaciones de lectura sobre ella (Aclaración 3.2).
- **Datos.** Este bloque de información depende mucho de cada proceso. Los lenguajes de programación actuales permiten asignación dinámica de memoria, lo que hace que varíe el tamaño del bloque de datos al avanzar la ejecución del proceso. Cada programa estructura sus datos de acuerdo a sus necesidades, pudiendo existir los siguientes tipos:
  -Datos con valor inicial. Estos datos son estáticos y su valor se fija al cargar el proceso desde el archivo ejecutable. Estos valores se asignan en tiempo dc compilación.

  -Datos sin valor inicial. Estos datos son estáticos, pero no tienen valor asignado. por lo que no están presentes en el archivo ejecutable. Será el sistema operativo el que, al cargar el proceso, rellene o no rellene esta zona de datos con valores predefinidos.
  -Datos dinámicos. Estos datos se crean y se destruyen de acuerdo a las directrices del programa.
  Los datos podrán ser de lectura-escritura o solamente de lectura.

- **Pila.** A través del puntero de pila, los programas utilizan una estructura dc pila residente en memoria. En ella se almacenan, por ejemplo, los bloques de activación de los procedimientos llamados. La pila es una estructura dinámica, puesto que crece y decrece según avanza la ejecución del proceso.

RECORDATORIO 3.2

Se denomina conjunto residente a las páginas que un proceso tiene en memoria principal. El conjunto de trabajo de un proceso está formado por el conjunto de páginas que un proceso está actualmente utilizando.

Para adaptarse a estas informaciones, se puede utilizar un modelo de imagen de memoria con un número fijo de segmentos de tamaño variable.

El modelo tradicional utilizado en UNIX contempla tres segmentos: texto, pila y datos. La Figura 3.9 presenta esta solución. Observe que el segmento de texto es de tamaño lijo (el programa habitualmente no se modifica) y que los segmentos de pila y de datos crecen en direcciones contrarias.

Este modelo se adapta bien a los sistemas con memoria virtual, puesto que el espacio virtual reservado para que puedan crecer los segmentos de datos y pila no existe. Solamente se crea. gastando recursos de disco y memoria, cuando se asigna. No ocurrirá lo mismo en un sistema sin memoria virtual, en el que el espacio reservado para el crecimiento ha de existir como memoria principal, dando como resultado que hay un recurso costoso que no se está utilizando.

Si bien este modelo es más flexible que los dos anteriores, tiene el inconveniente de no prestar ayuda para la estructuración de los datos. El sistema operativo ofrece un espacio de datos que puede crecer y decrecer, pero deja al programa la gestión de este espacio.

Proceso con un número variable de segmentos de tamaño variable

Esta solución es más avanzada que la anterior, al permitir que existan los segmentos que desee el proceso. La Figura 3.10 presenta un caso de siete segmentos, que podrán ser de texto. de pila o de datos.

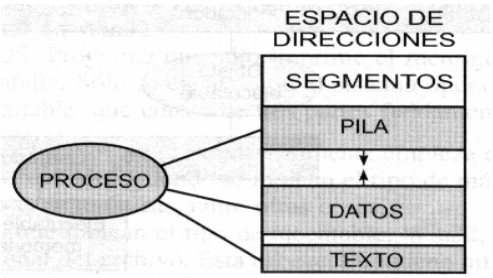
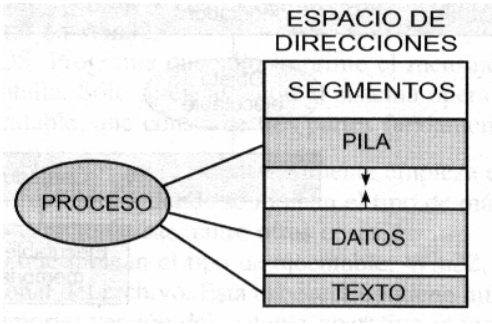


Figura 3.9. *Modelo de imagen de memoria con estructura de segmentos fija.*


ESPACIO DE
DIRECCIONES
SEGMENTOS
PILA
PROCESO
DATOS
TEXTO

Los primeros pasos se refieren a la fase de programación de la aplicación, que desembocan en un objeto ejecutable. Sin embargo, este objeto no suele ser un programa máquina totalmente completo, puesto que no incluye las rutinas del sistema. Esta solución tiene la ventaja de no repetir en cada archivo ejecutable estas rutinas, con el consiguiente ahorro de espacio de disco. El sistema operativo, en el proceso de carga, incluye las rutinas necesarias formando un ejecutable completo en memoria.

EI objeto ejecutable es un archivo que incluye la siguiente información:

- Cabecera que contiene entre otras informaciones las siguientes:
  - Estado inicial de los registros.
  - Tamaño del código y de los datos.
  - Palabra «mágica» que identifica al archivo como un ejecutable.
- Código.
- Datos con valor inicial.

Los datos sin valor inicial no necesitan residir en el archivo ejecutable puesto que el sistema operativo se encargará de asignarles valor (normalmente O) cuando cree el proceso encargado de ejecutar dicho programa.

Ejecutable de Win32

El ejecutable de Win32 tiene e! siguiente formato:

- Cabecera MZ de MS-DOS.
- Programa MS-DOS.
- Cabecera del ejecutable.
- Cabecera de Sección 1.
- Cabecera de Sección 2.
- Cabecera de Sección 3.
- ...
- Cuerpo de la Sección 1.
- Cuerpo de la Sección 2.
- Cuerpo de la Sección 3.
- ...

1. Cabecera MZ de MS-DOS. Esta cabecera está presente en todos los ejecutables de Win32 para preservar la «compatibilidad hacia atrás». La cabecera del programa imprime en pantalla el conocido mensaje «This programs runs under Windows». Esta cabecera es ignorada al ejecutar en Win32.

2. Programa MS-DOS. Programa que sólo imprime el mensaje <<This program runs under Windows» en pantalla. Sólo se ejecuta con el sistema operativo MS-DOS.
3. Cabecera del ejecutable, que consta de tres partes fundamentales:

— Cabecera del archivo. Aquí es donde realmente empieza el ejecutable de Win32. Esta cabecera es una estructura donde se indican el tipo de máquina, la hora y la fecha del ejecutable, sus características, entre otras cosas.
— Firma. 4 bytes que indican el tipo de ejecutable: Win32, OS/2...
— Cabecera opcional del archivo. Esta cabecera contiene información de como cargar el archivo en memoria, versión del sistema operativo requerido, versión del enlazador, etcétera.

4. Cabeceras de las secciones. Los datos del ejecutable en sí están contenidos en las secciones. Las secciones se componen de una cabecera y un cuerpo. En el cuerpo de la sección están los datos y en la cabecera de la sección información de cómo están organiiados los datos y de qué tipo son (de lectura, de escritura, de lectura/escritura...).
5. Cuerpos de las secciones. Después de las cabeceras de las secciones están los cuerpos de las secciones, donde están contenidos todos los datos del ejecutable. Las secciones mas comunes son: código, datos con valor inicial, datos con valor inicial de sólo lectura, datos sin valor inicial, funciones importadas, funciones exportadas, depuracíon...

3.3.3. Información del BCP

El BCP contiene la información básica del proceso, entre la que cabe destacar la siguiente:

Información de identificación

Esta información identifica al usuario y al proceso**.** Como ejemplo, se incluyen los siguientes datos:

- Identificador del proceso.
- Identificador del proceso padre, en caso de existir relaciones padre-hijo como en UNIX.
- Información sobre el usuario (identificador de usuario. identificador de grupo).

Estado del procesador

Contiene los valores iniciales del estado del procesador o su valor en el instante en que fue interrumpido el proceso.

Información de control del proceso

En esta sección se incluye diversa información que permite gestionar al proceso. Destacaremos los siguientes datos:

- **Información** de planificación y estado.

—Estado del proceso.
—Evento por el que espera el proceso cuando está bloqueado.
—Prioridad del proceso.
—Información de planificación.

• Descripción de los segmentos de memoria asignados al proceso.
• Recursos asignados, tales como:
—Archivos abiertos (tabla de descriptores o manejadores de archivo).
—Puertos de comunicación asignados.
• Punteros para estructurar los procesos en colas o anillos. Por ejemplo, los procesos que están en estado de listo pueden estar organizados en una cola, de forma que se facilite la labor del planificador.
• Comunicación entre procesos. El BCP puede contener espacio para almacenar las señales y para algún mensaje enviado al proceso.

3.3.4. Tablas del sistema operativo

Como se muestra en la Figura 3.8, el sistema operativo mantiene una serie de tablas que describen a los procesos y a los recursos del sistema. Algunos de los aspectos de estas tablas ya se han ido comentando, pero aquí se profundizará y detallará el contenido de las mismas.

La información asociada a cada proceso se encuentra parcialmente en el BCP y parcialmente fuera de él. La decisión de incluir o no una información en el BCP se toma según dos argumentos: eficiencia y necesidad de compartir información.

Eficiencia

Por razones de eficiencia, es decir, para acelerar los accesos, la tabla de procesos se construye normalmente como una estructura estática, formada por un número determinado de BCP del mismo tamaño. En este sentido, aquellas informaciones que pueden tener un tamaño variable no deben incluirse en el BCP. De incluirlas habría que reservar en cada BCP el espacio necesario para almacenar el mayor tamaño que puedan tener estas informaciones. Este espacio estaría presente en todos los BCP, pero estaría muy desaprovechado en la mayoría de ellos.

Un ejemplo de este tipo de información es la tabla de páginas, puesto que su tamaño depende dc las necesidades de memoria de los procesos, valor que es muy variable de unos a otros. En este sentido, el BCP incluirá el RIED (registro identificador de espacio de direccionamiento) e incluso una descripción de cada segmento (p. ej.: incluirá la dirección virtual donde comienza el segmento, su tamaño, la zona reservada para su crecimiento y el puntero a la subtabla de páginas, pero no la subtabla en sí).

Compartir información

Cuando la información ha de ser compartida por varios procesos, no ha de residir en el BCP, que es dc acceso restringido al proceso que lo ocupa. A lo sumo, el BCP contendrá un apuntador que permita alcanzar esa información.

Un ejemplo de esta situación lo presentan los procesos en relación con los punteros de posición de los archivos que tienen abiertos. Dos procesos pueden tener simultáneamente abierto el mismo archivo por dos razones: el archivo se heredó abierto de un proceso progenitor o el archivo se abrió de forma independiente por los dos procesos.

En el primer caso se trata de procesos emparentados que, lógicamente, se han diseñado para compartir el archivo. Téngase en cuenta que un proceso hijo que no desea

utilizar un archivo abierto heredado del padre deberá, de acuerdo a las prácticas correctas de programación, cerrarlo. Al tratarse de procesos que están diseñados para compartir el archivo, lo más conveniente es que compartan el puntero de posición (PP). De esta forma, si ambos escriben en el archivo, lo escrito por uno se pondrá a continuación de lo escrito por el otro, pero no encima.

El compartir los punteros de posición exige que no estén ubicados en el BCP. La Figura 3. 12 nuestra una posible solución, basada en una tabla de archivos externa a los BCP y compartida por todos los procesos. En dicha tabla se encuentra el puntero de posición PP, además del identificador fisico del archivo IDFF. Este es el modelo que sigue UNIX.

En la Figura 3.12, el proceso con BCP 7 es un hijo del proceso con BCP 4. El descriptor de archivo (fd) 3 de ambos procesos utiliza la entrada 4 de la tabla de archivos, lo que significa que comparten cl archivo y su puntero de posición. En dicha tabla observamos que el IDFF del archivo es cl 34512 y que cl puntero PP tiene un valor de 10000.

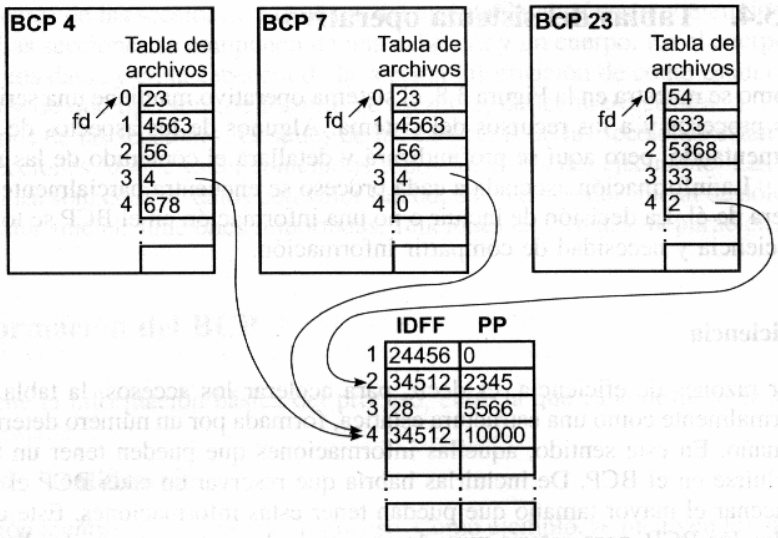


Figura 3.12. *Tabla de archivos con los punteros de posición y los descriptores físicos de archivo.*

Ahora bien, se puede dar el caso de que dos procesos independientes A y B abran el mismo archivo, Imaginemos que el proceso A lee 1.000 bytes del archivo y que seguidamente el proceso B lee 500 bytes. Está claro que el proceso B, que no tiene nada que ver con el proceso A, espera leer los primeros 500 bytes del archivo y no los bytes 1 .000 a 1 .499. Para evitar esta situación, el sistema operativo asignará una nueva entrada en la tabla externa de archivos cada vez que se realice una operación de apertura de archivo. De esta forma, cada apertura obtiene su propio PP.

La Figura 3.12 presenta esta situación, puesto que analizando el proceso con BCP 23 se observa que su fd 4 utiliza la entrada 2 de la tabla externa de archivos. Esta entrada se refiere al archivo de IDFF = 345 12, que coincide con el de la entrada 4. Los tres procesos de la figura comparten el archivo 34512, pero de forma distinta, puesto que los de BCP 4 y 7 comparten el PP, mientas que el de BCP 23 utiliza otro puntero de posición (Aclaración 3.3).

ACLARACIÓN 3.3

La coutilización de archivos puede plantear problemas. En efecto, cuando dos procesos independientes abren el mismo archivo para escribir, el contenido final del archivo es impredecible, pues dependerá del orden en que el planificador los ponga a ejecutar.

Finalmente, diremos que otra razón que obliga a que las tablas de páginas sean externas al BCP es para permitir que se pueda compartir memoria. En este caso, como se analizará en detalle en el Capítulo 4, dos o más procesos comparten parte de sus tablas de páginas.

Tablas de E/S

En las tablas de entrada/salida el sistema operativo mantiene la información asociada a los periféricos y a las operaciones de entrada/salida. En general, el sistema operativo mantendrá una cola por cada dispositivo, en la que se almacenarán las operaciones pendientes de ejecución, así como la operación en curso de ejecución.

3.4. FORMACIÓN DE UN PROCESO

La formación de un proceso consiste en completar todas las informaciones que lo constituyen, como se muestra en la Figura 3.13.
De forma más específica, las operaciones que debe hacer el sistema operativo son las siguientes:

- Asignar un espacio de memoria para albergar la imagen de memoria. En general, este espacio será virtual y estará compuesto por varios segmentos.
- Seleccionar un BCP libre de la tabla de procesos.
- Rellenar el BCP con la información de identificación del proceso, con la descripción de la memoria asignada, con los valores iniciales de los registros indicados en el archivo objeto, etc.
- Cargar en el segmento de texto el código más las rutinas de sistema y en el segmento de datos los datos iniciales contenidos en el archivo objeto.
- Crear en el segmento de pila la pila inicial del proceso. La pila incluye inicialmente el entorno del proceso y los parámetros que se pasan en la invocación del programa correspondiente.

Una vez completada toda la información del proceso, se puede marcar como listo para ejecutar, dc forma que el planificador, cuando lo considere oportuno, lo seleccione para su ejecución.

3.5.ESTADOS DEL PROCESO

Como se puede observar en la Figura 3.4, no todos los procesos activos de un sistema multitarea están en la misma situación. Se diferencian, por tanto, tres estados básicos en los que puede estar un proceso, estados que detallamos seguidamente:

• **Ejecución.** En este estado está el proceso que está siendo ejecutado por el procesador, es decir, que está en fase de procesamiento. En esta fase el estado del proceso reside en los registros del procesador.

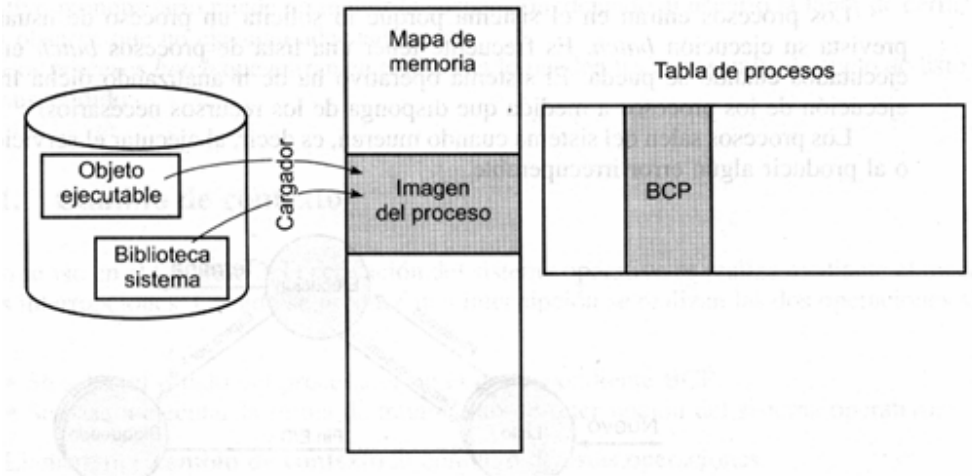


Figura 3.13. Formación de un proceso.

• **Bloqueado.** Un proceso bloqueado está esperando a que ocurra un evento y no puede seguir ejecutando hasta que suceda el evento. Una situación típica de proceso bloqueado se produce cuando el proceso solicita una operación de E/S. Hasta que no termina esta operación, el proceso queda bloqueado. En esta fase, el estado del proceso reside en el BCP.
• **Listo.** Un proceso está listo para ejecutar cuando puede entrar en fase de procesamiento. Dado que puede haber varios procesos en este estado, una de las tareas del sistema operativo será seleccionar aquel que debe pasar a ejecución. El módulo del sistema operativo que toma esta decisión se denomina **planificador.** En esta fase, el estado del proceso reside en el BCP.

La Figura 3. 14 presenta estos tres estados, indicando algunas de las posibles transiciones entre ellos. Puede observarse que sólo hay un proceso en estado de ejecución, puesto que el procesador solamente ejecuta un programa en cada instante (Prestaciones 3. 1). Del estado de ejecución se pasa al estado de bloqueado al solicitar, por ejemplo, una operación de *EIS.* También se puede pasar del estado de ejecución al de listo cuando el sistema operativo decide que ese proceso lleva mucho tiempo en ejecución. Del estado de bloqueado se pasa al estado de listo cuando se produce el evento por el que estaba esperando el proceso (p. ej.: cuando se completa la operación de E/S solicitada). Todas las transiciones anteriores están gobernadas por el sistema operativo, lo que implica la ejecución del mismo en dichas transiciones.

PRESTACIONES 3.1

En una máquina multiprocesador se puede tener simultáneamente en estado de ejecución tantos procesos como procesadores.

Estados suspendidos

Además de los tres estados básicos de ejecución, listo y bloqueado, los procesos pueden estar en los estados de espera y de suspendido. El diagrama de estados completo de un proceso se representa en la Figura 3.15.

Los procesos entran en el sistema porque lo solicita un proceso de usuario o porque está prevista su ejecución *hatch.* Es frecuente tener una lista de procesos *batch* **en espera** para ser ejecutados cuando se pueda. El sistema operativo ha de ir analizando dicha lista para lanzar la ejecución de los procesos a medida que disponga de los recursos necesarios.

Los procesos salen del sistema cuando mueren, es decir, al ejecutar el servicio correspondiente o al producir algún error irrecuperable.

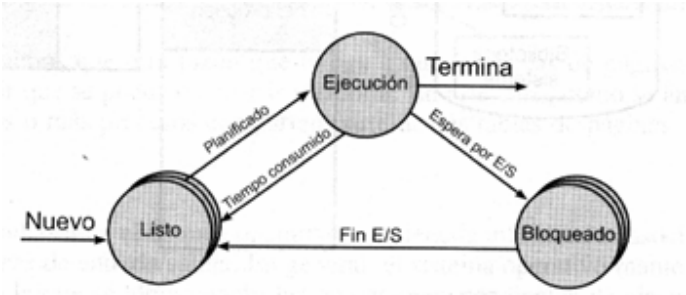


Figura 3.14. E*stados básicos de un proceso.*

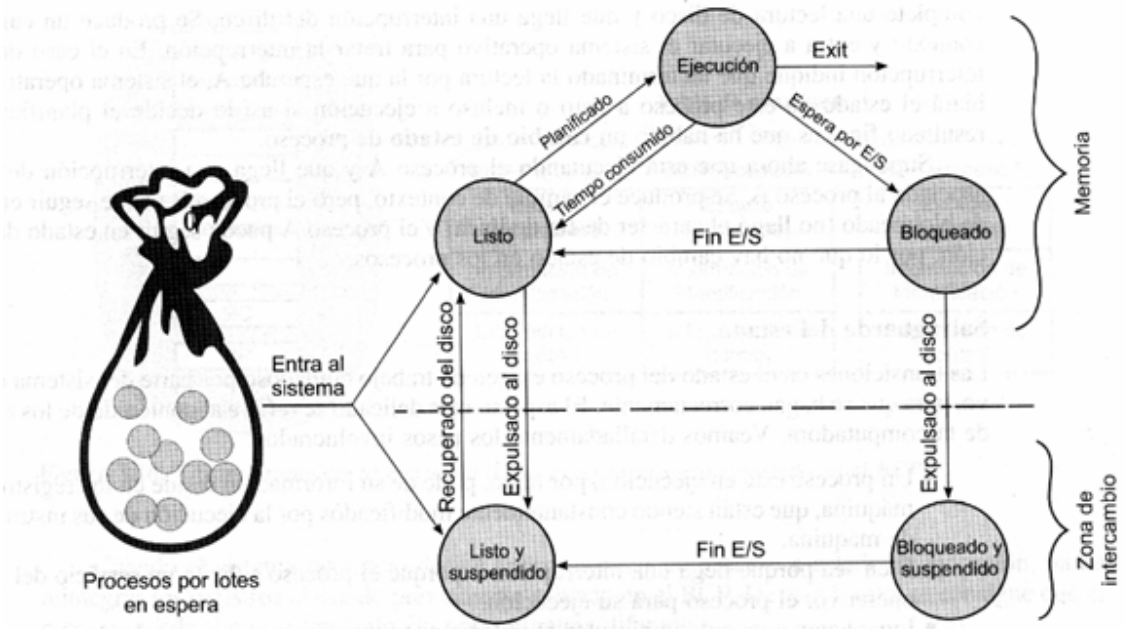


Figura 3.15. *Diagrama completo con los estados de un proceso.*

Para disminuir el grado de multiprogramación efectivo, el sistema operativo puede **suspender** algunos procesos, lo que implica que les retira todos sus marcos de páginas, dejándolos enteramente en la zona de intercambio. En la Figura 3. ¡5 se muestra cómo los procesos listos o bloqueados pueden suspenderse. El objetivo de la suspensión estriba en dejar suficiente memoria a los procesos no suspendidos para que su conjunto residente tenga un tamaño adecuado que evite la hiperpaginacion.
No todos los sistemas operativos tienen la opción de suspensión. Por ejemplo, un sistema operativo monousuario puede no incluir la suspensión, dejando al usuario la labor de cerrar procesos si observa que no ejecutan adecuadamente.
Los procesos *hatch* que entran en el sistema lo pueden hacer pasando al estado de listo o al de listo suspendido.

3.5.1. Cambio de contexto

Como se vio en el Capítulo 2, la activación del sistema operativo se realiza mediante el mecanismo de las interrupciones. Cuando se produce una interrupción se realizan las dos operaciones siguientes:

- Se salva el estado del procesador en el correspondiente BCP.
- Se pasa a ejecutar la rutina de tratamiento de interrupción del sistema operativo.

Llamaremos **cambio de contexto** al conjunto de estas operaciones.

Como resultado del cambio de contexto se puede producir un cambio en el estado de algunos procesos, pero no necesariamente. Supóngase que el proceso A está bloqueado esperando a que se

complete una lectura de disco y que llega una interrupción del disco. Se produce un cambio de contexto y entra a ejecutar el sistema operativo para tratar la interrupción. En el caso de que la interrupción indique que ha terminado la lectura por la que esperaba A, el sistema operativo cambiará el estado de este proceso a listo o incluso a ejecución si así lo decide el planificador. El resultado final es que ha habido un **cambio de estado** de proceso.

Supóngase ahora que está ejecutando el proceso A y que llega una interrupción de teclado asociado al proceso B. Se produce el cambio de contexto, pero el proceso B puede seguir en estado de bloqueado (no llegó el carácter de fin de línea) y el proceso A puede seguir en estado de ejecución, por lo que no hay cambio de estado en]os procesos.

Salvaguarda del estado

Las transiciones en el estado del proceso exigen un trabajo cuidadoso por parte del sistema operativo, para que se hagan correctamente. El aspecto más delicado se refiere al contenido de los registros de la computadora. Veamos detalladamente los pasos involucrados:

- Un proceso está en ejecución, por tanto, parte de su información reside en los registros de la máquina, que están siendo constantemente modificados por la ejecución de sus instrucciones de máquina.
- Bien sea porque llega una interrupción o porque el proceso solicita un servicio del sistema operativo, el proceso para su ejecución.
- Inmediatamente entra a ejecutar el sistema operativo, ya sea para atender la interrupción o para atender el servicio demandado.
- La ejecución del sistema operativo, como la de todo programa, modifica los contenidos de los registros de la máquina, destruyendo sus valores anteriores.

Según la secuencia anterior, si se desea más adelante continuar con la ejecución del proceso, se presenta un grave problema: los registros ya no contienen los valores que deberían. Supongamos que el proceso está ejecutando la secuencia siguiente:

```
LD      .5,# CANT
                          <= En este punto llega una interrupción y se pasa al SO
LD      .1, [.5]
```

Supongamos que el valor de CANT es HexA4E78, pero que el sistema operativo al ejecutar modifica cl registro . 5 dándole el valor HexEB7A4. Al intentar, más tarde, que siga la ejecución del mencionado proceso, la instrucción “LD . 1, [. 5] “ cargará en el registro . 1 el contenido de la dirección HexEB7A4 en vez del contenido de la dirección HexA4E78.

Para evitar esta situación, lo primero que hace el sistema operativo al entrar a ejecutar es salvar el contenido de todos los registros, teniendo cuidado de no haber modificado el valor de ninguno de ellos. Como muestra la Figura 3.16, al interrumpirse la ejecución de un proceso el sistema operativo almacena los contenidos de los registros en el BCP de ese proceso (Recordatorio 3.3).

RECORDATORIO 3.3

Como sabemos, la propia interrupción modifica el contador de programa y el registro de estado (cambia el bit que especifica el nivel de ejecución para pasar a nivel de núcleo). Sin embargo, esto no presenta ningún problema puesto que el propio hardware se encarga de salvar estos registros en la pila antes de modificarlos.

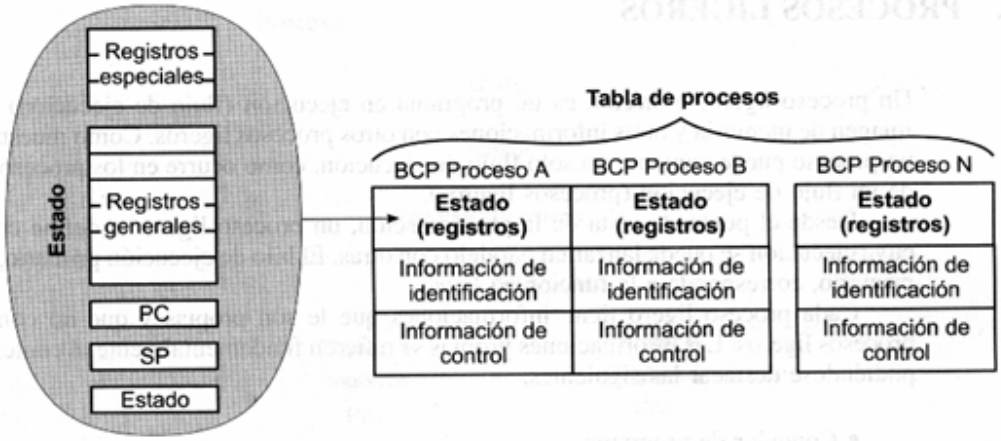


Figura 3.16. *Al interrumpirse la ejecución de un proceso se salva su estado en el BCP.*

Cuando el sistema operativo decide volver a pasar un proceso a estado de ejecución, ha dc reintegrar los registros al estado previamente salvado en el BCP. De esta forma, se consigue que el proceso pueda seguir ejecutando sin notar ninguna diferencia.

Activación de un proceso

El módulo del sistema operativo que pone a ejecutar un proceso se denomina **activador o *dispatcher.*** La activación de un proceso consiste en copiar en los registros del procesador el estado del procesador, que está almacenado en su BCP. De esta forma, el proceso continuará su ejecución en las mismas condiciones en las que fue parado. El activador termina con una instrucción RETI (Recordatorio 3.4) de retorno de interrupción. El efecto de esta instrucción es restituir el registro de estado y el contador de programa, lo cual tiene los importantes efectos siguientes:

- Al restituir el registro de estado, se restituye el bit que especifica el nivel de ejecución. Dado que cuando fue salvado este registro el bit indicaba nivel de usuario, puesto que estaba ejecutando un proceso, su restitución garantiza que el proceso seguirá ejecutando en nivel de usuario.
- Al restituir el contador de programa, se consigue que la siguiente instrucción máquina que ejecute el procesador sea justo la instrucción en la que fue interrumpido el proceso. En este momento es cuando se ha dejado de ejecutar el sistema operativo y se pasa a ejecutar el proceso.

RECORDATORIO 3.4

La instrucción RETI es una instrucción máquina que restituye los registros salvados por el hardware al aceptar la interrupción. En general, los registros son el de estado y el contador de programa, que se suelen almacenar en la pila.

3.6. PROCESOS LIGEROS

Un proceso ligero, o *thread,* es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria y otras informaciones con otros procesos ligeros. Como muestra la Figura 3.17, un proceso puede contener un solo flujo de ejecución, como ocurre en los procesos clásicos, o más de un flujo de ejecución (procesos ligeros).

Desde el punto de vista de la programación, un proceso ligero se define corno una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario, o proceso ligero primario, corresponde a la función main.

Cada proceso ligero tiene informaciones que le son propias y que no comparte con otros
procesos ligeros. Las informaciones propias se refieren fundamentalmente al contexto de ejecución, pudiéndose destacar las siguientes:

- Contador de programa.
- Pila.
- Registros.
- Estado del proceso ligero (ejecutando, listo o bloqueado).

Todos los procesos ligeros de un mismo proceso comparten la información del mismo. En concreto, comparten:

- Espacio de memoria.
- Variables globales.
- Archivos abiertos.
- Procesos hijos.
- Temporizadores.
- Señales y semáforos.
- Contabilidad.

Es importante destacar que todos los procesos ligeros de un mismo proceso comparten el **mismo** espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes procesos ligeros. Esto hace que no exista protección de memoria entre los procesos ligeros de un mismo proceso, algo que sí ocurre con los procesos convencionales.

El proceso ligero constituye la unidad ejecutable en Windows NT. La Figura 3. IX representa de forma esquemática la estructura de un proceso de Windows NT con sus procesos ligeros.

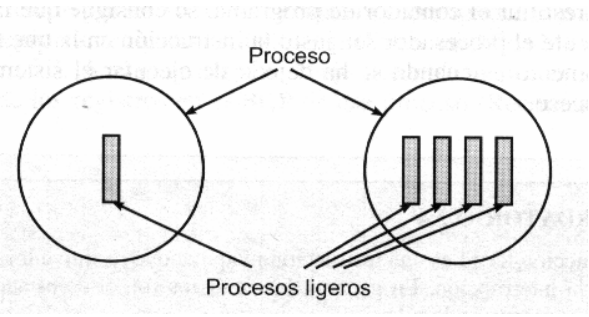


Figura 3.17. *Proceso ligero.*

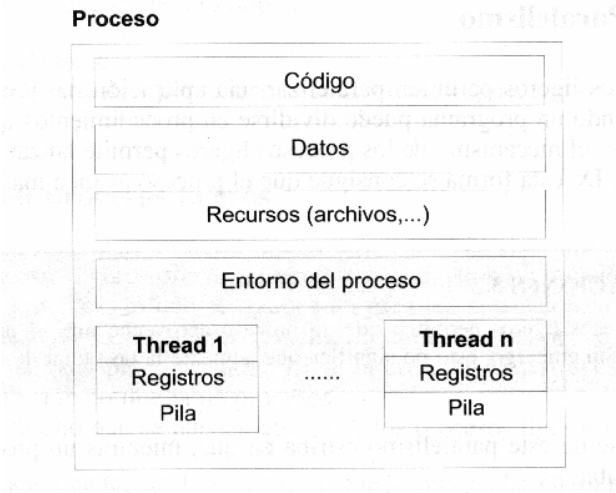


Figura 3.18. *Estructura de un proceso en Windows NT*

3.6.1. Estados del proceso ligero

El proceso ligero puede estar en uno de los tres estados siguientes: ejecutando, listo para ejecutar y bloqueado. Como muestra la Figura 3.19, cada proceso ligero de un proceso tiene su propio estado, pudiendo estar unos bloqueados, otros listos y otros en ejecución (Aclaración 3.4).

ACLARACIÓN 3.4

En realidad podría haber en ejecución tantos procesos ligeros de un mismo proceso como procesadores tenga la máquina.

El estado del proceso será ¡a combinación de los estados de sus procesos ligeros. Por ejemplo, si tiene un proceso ligero en ejecución, el proceso está en ejecución. Si no tiene proceso ligeros en ejecución, pero tiene alguno listo para ejecutar, el proceso está en estado de listo. Finalmente, si todos sus procesos ligeros están bloqueados, el proceso está bloqueado. No tiene sentido el estado dc suspendido, puesto que si el proceso está suspendido, todos sus procesos ligeros lo estarán.

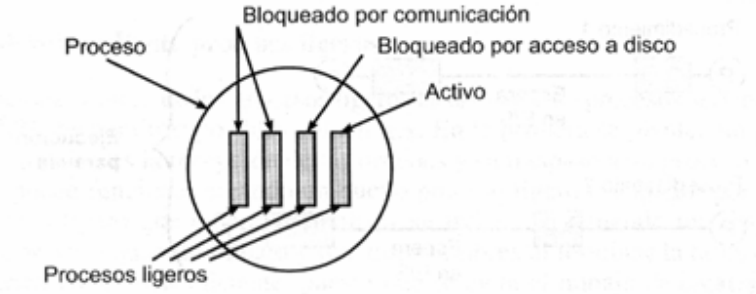


Figura 3.19. *Estado dc un proceso ligero.*

3.6.2.Paralelismo

Los procesos ligeros permiten paralelizar una aplicación, tal y como muestra la Figura 3.20. En efecto, cuando un programa puede dividirse en procedimientos que pueden ejecutar de forma independiente, el mecanismo de los procesos ligeros permite lanzar simultáneamente la ejecución de todos ellos. De esta forma se consigue que el proceso avance más rápidamente (Prestaciones 3.2).

PRESTACIONES 3.2

Los procesos ligeros permiten que un proceso aproveche más el procesador, es decir, ejecute más deprisa. Sin embargo, esto no significa que aumente la tasa total de uso del procesador

La base de este paralelismo estriba en que, mientras un proceso ligero está bloqueado, otro puede ejecutar.
Comparando el uso de los procesos ligeros con otras soluciones se puede decir que:

- Los procesos ligeros:

— Permiten paralelismo y variables compartidas.
— Utilizan llamadas al sistema bloqueantes por proceso ligero. Esta es la solución más sencilla de conseguir paralelismo desde el punto de vista de la programación.

- Un proceso convencional con un solo proceso ligero:

— No hay paralelismo.

— Utiliza llamadas al sistema bloqueantes.

• Un proceso con un solo proceso ligero, que usa llamadas no bloqueantes y estructurado en máquina de estados finitos:

— Permite paralelismo.
— Utiliza llamadas al sistema no bloqueantes, lo que lleva a un diseño muy complejo y difícil de mantener.

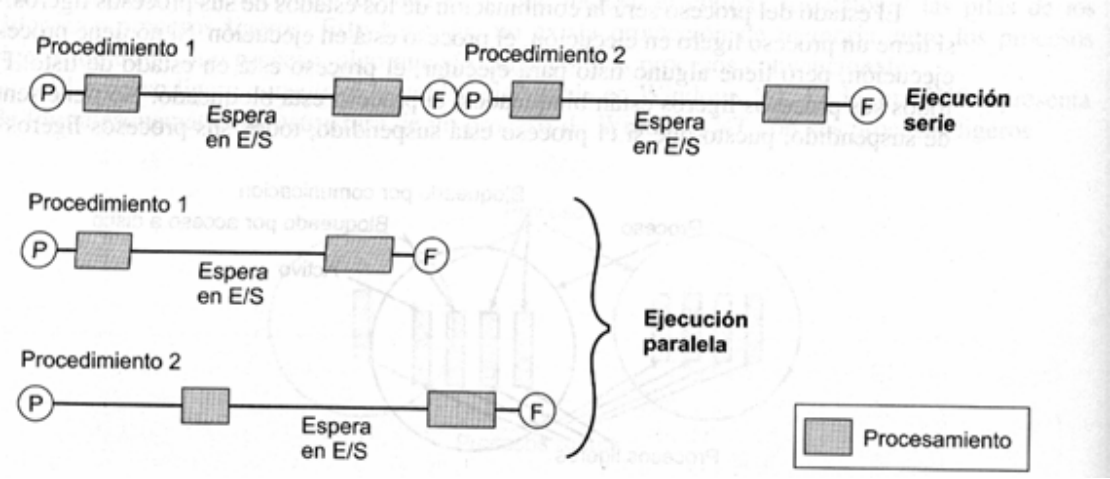


Figura 3.20. *Los procesos ligeros permiten paralelizar la ejecución de una aplicación.*

• Varios procesos convencionales cooperando:

— Permite paralelismo.
— No comparte variables, por lo que la comunicación puede consumir mucho tiempo.

3.6.3. Diseño con procesos ligeros

La utilización de procesos ligeros ofrece las ventajas de división de trabajo que dan los procesos, pero con una mayor sencillez, lo que se traduce en mejores prestaciones. En este sentido, es de destacar que los procesos ligeros comparten memoria directamente, por lo que no hay que añadir ningún mecanismo adicional para utilizarla, y que la creación y destrucción de procesos ligeros requiere mucho menos trabajo que la de procesos.
Las ventajas de diseño que se pueden atribuir a los procesos ligeros son las siguientes:

- Permite separación de tareas. Cada tarea se puede encapsular en un proceso ligero independiente.
- Facilita la modularidad, al dividir trabajos complejos en tareas.
- Aumenta la velocidad de ejecución del trabajo, puesto que aprovecha los tiempos de bloqueo de unos procesos ligeros para ejecutar otros.

El paralelismo que permiten los procesos ligeros, unido a que comparten memoria (utilizan variables globales que ven todos ellos), permite la programación concurrente. Este tipo de programación tiene un alto nivel de dificultad, puesto que hay que garantizar que el acceso a los datos compartidos se haga de forma correcta. Los principios básicos que hay que aplicar son los siguientes:

- Hay variables globales que se comparten entre varios procesos ligeros. Dado que cada proceso ligero ejecuta de forma independiente a los demás, es fácil que ocurran accesos incorrectos a estas variables.
- Para ordenar la forma en que los procesos ligeros acceden a los datos se emplean mecanismos de sincronización, como el mutex, que se describirán en el Capítulo 5. El objetivo de estos mecanismos es impedir que un proceso ligero acceda a unos datos mientras los esté utilizando otro.
- Para escribir código correcto hay que imaginar que los códigos de los otros procesos ligeros que pueden existir están ejecutando cualquier sentencia al mismo tiempo que la sentencia que se está escribiendo.

Diseño de servidores mediante procesos ligeros

Una de las aplicaciones típicas de los procesos ligeros es el diseño de procesos servidores paralelos.
La Figura 3.21 muestra tres posibles soluciones. En la primera se plantea un proceso ligero distribuidor cuya función es la recepción de las órdenes y su traspaso a un proceso ligero trabajador. El esquema puede funcionar creando un nuevo proceso ligero trabajador por cada solicitud dc servicio, proceso ligero que muere al finalizar su trabajo, o teniendo un conjunto de ellos creados a los que se va asignando trabajo y que quedan libres al terminar la tarea encomendada. Esta segunda alternativa es más eficiente, puesto que se evita el trabajo de crear y destruir a los procesos ligeros.

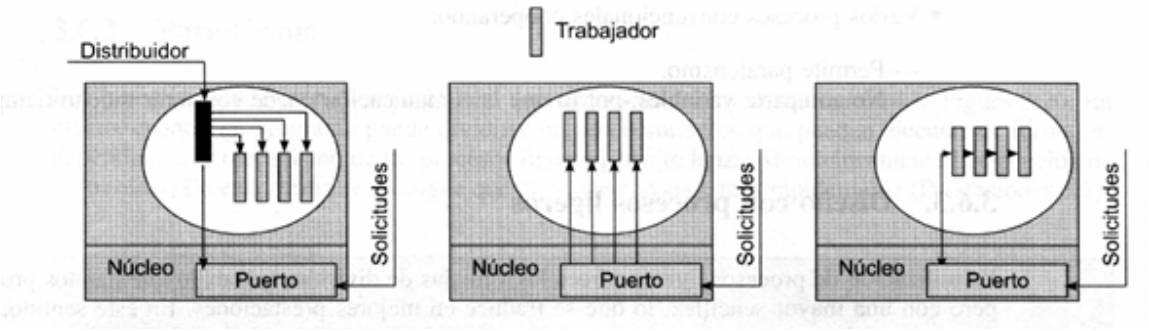


Figura 3.21. *Ejemplos de diseño de servidores paralelos mediante procesos ligeros.*

La segunda solución consiste en disponer de un conjunto de procesos ligeros iguales, todos los cuales pueden aceptar una orden (leer de un puerto en el caso de la figura). Cuando llega una solicitud, la recibe uno de los procesos ligeros que están leyendo del puerto. Este proceso ligero tratará la petición y, una vez finalizado el trabajo solicitado, volverá a la fase de leer una nueva petición del puerto.
La tercera solución aplica la técnica denominada de segmentación *(pipe-line).* Cada trabajo se divide en una serie de fases, encargándose de cada una de ellas un proceso ligero especializado. El esquema permite tratar al mismo tiempo tantas solicitudes como fases tenga la segmentación, puesto que se puede tener una en cada proceso ligero.

3.7. PLANIFICACIÓN

El objetivo de la planificación de procesos y procesos ligeros es el reparto del tiempo de procesador entre los procesos que pueden ejecutar. El **planificador** es el módulo del sistema operativo que realiza la función de seleccionar el proceso en estado de listo que pasa a estado de ejecución, mientras que el **activador** es el módulo que pone en ejecución el proceso planificado.
Los sistemas pueden incluir varios niveles de planificación de procesos. La Figura 3.22 muestra el caso de tres niveles: corto, medio y largo plazo.
La planificación a **largo plazo** tiene por objetivo añadir nuevos procesos al sistema, tomándolos de la lista de espera. Estos procesos son procesos de tipo *batch,* en los que no importa el instante preciso en el que se ejecuten (siempre que se cumplan ciertos límites de espera).
La planificación a medio **plazo** trata la suspensión de procesos. Es la que decide qué procesos pasan a suspendido y cuáles dejan de estar suspendidos. Añade o elimina procesos de memoria principal modificando, por tanto, el grado de multiprogramación.
La planificación a **corto plazo** se encarga de seleccionar el proceso en estado de listo que pasa a estado de ejecución. Es, por tanto, la que asigna el procesador.
También es importante la planificación de entrada/salida. Esta planificación decide el orden en que se ejecutan las operaciones de entrada/salida que están encoladas para cada periférico.

Expulsión

La planificación puede ser con expulsión o sin ella. En un sistema sin expulsión un proceso conserva el procesador mientras lo desee, es decir, mientras no solicite del sistema operativo un servicio que lo bloquee. Esta solución minimiza el tiempo que gasta el sistema operativo en planificar y

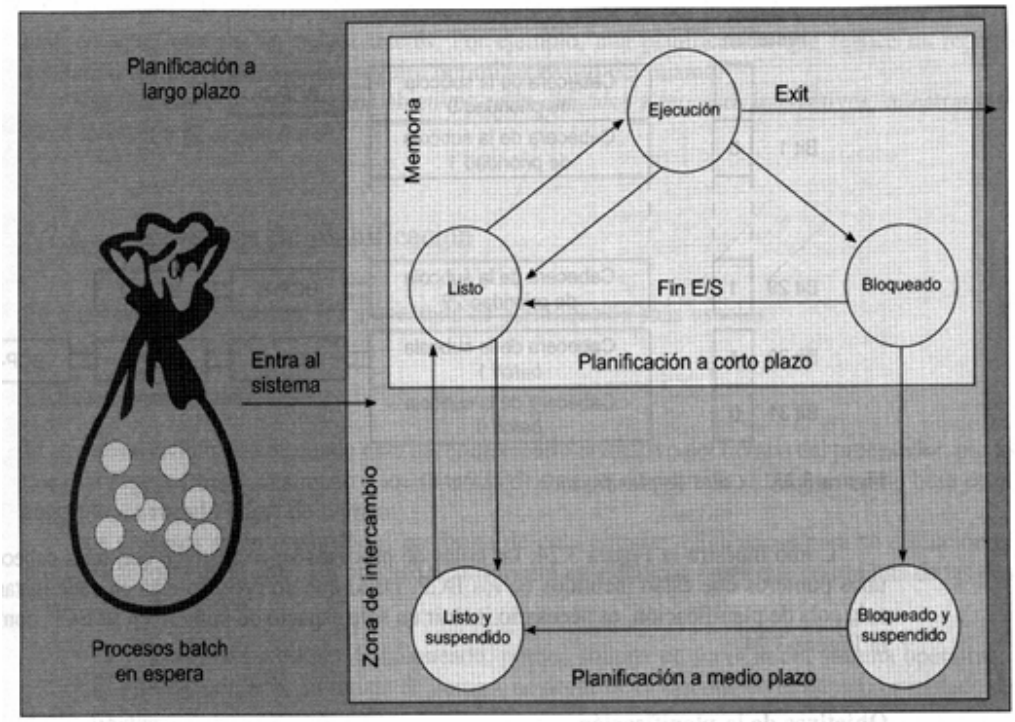


Figura 3.22. *Tipos de planificación.*

activar procesos, pero tiene como inconveniente que un proceso puede monopolizar el procesador (imagínese lo que ocurre si el proceso, por error, entra en un bucle infinito).

En los sistemas con expulsión, el sistema operativo puede quitar a un proceso del estado de ejecución aunque éste no lo solicite. Esta solución permite controlar el tiempo que está en ejecución un proceso, pero requiere que el sistema operativo entre de forma sistemática a ejecutar para así poder comprobar si el proceso ha superado su límite de tiempo de ejecución. Como sabemos, las interrupciones sistemáticas del re]oj garantizan que el sistema operativo entre a ejecutar cada pocos milisegundos, pudiendo determinar en estos Instantes si ha de producirse un cambio dc proceso o no.

Colas de procesos

Para realizar las funciones de planificación, el sistema operativo organiza los procesos listos en una serie de estructuras de información que faciliten la búsqueda del proceso a planificar. Es muy frecuente organizar los procesos en colas de prioridad y de tipo.

La Figura 3.23 muestra un ejemplo con 30 colas para procesos interactivos y 2 colas para procesos *batuh.* Las 30 colas interactivas permiten ordenar los procesos listos interactivos según 30 niveles dc prioridad, siendo, por ejemplo, el nivel 0 el más prioritario. Por su lado, las dos colas *batch* permiten organizar los procesos listos *batch* en dos niveles de prioridad.

Se puede observar en la mencionada figura que se ha incluido una palabra resumen. Esta palabra contiene un 1 si la correspondiente cola tiene procesos y un 0 si

está vacía. De esta forma, se acelera cl planificador, puesto que puede saber rápidamente dónde encontrará procesos listos.

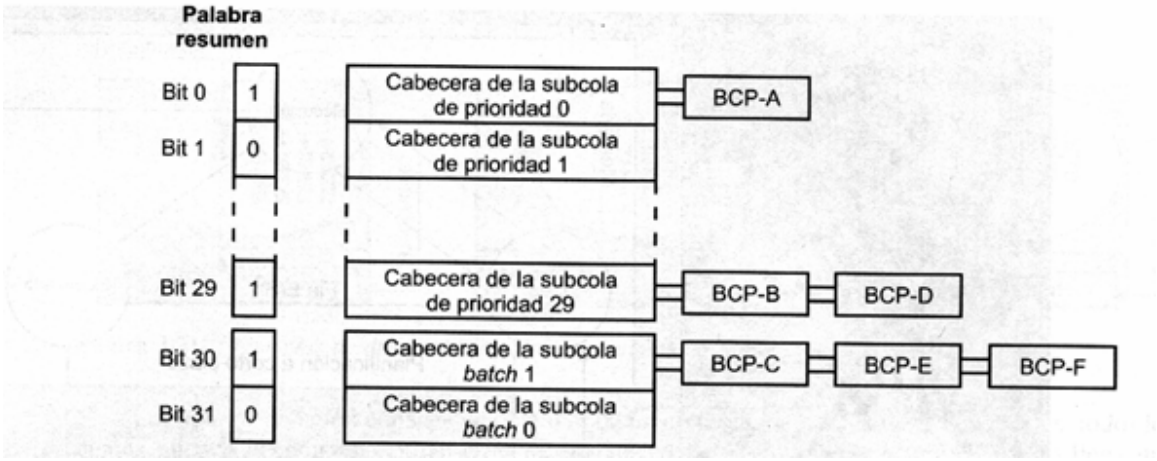


Figura 3.23. *Colas de planificación.*

Corno muestra la Figura 3.24, las colas de procesos se construyen con unas cabeceras y con unos punteros que están incluidos en los BCP. Dado que un proceso está en cada instante en una sola cola de planificación, es necesario incluir un solo espacio de puntero en su BCP, como muestra la Figura 3.24.

Objetivos de la planificación

El objetivo de la planificación es optimizar el comportamiento del sistema. Ahora bien, el comportamiento de un sistema informático es muy complejo, por tanto, el objetivo de la planificación se deberá centrar en la faceta del comportamiento en el que se esté interesado. Entre los objetivos que se suelen perseguir están los siguientes:

- Reparto equitativo del procesador.
- Eficiencia (optimizar el uso del procesador).
- Menor tiempo de respuesta en uso interactivo.
- Menor tiempo de espera en lotes *(batch).*
- Mayor número de trabajos por unidad de tiempo *(batch).*
- Cumplir los plazos de ejecución de un sistema de tiempo real.

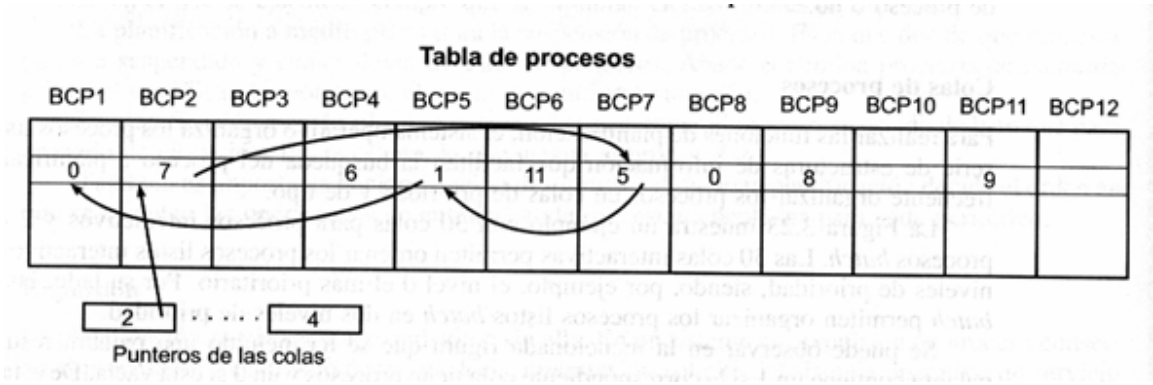


Figura 3.24. *Implementación de las colas de planificación.*

La mayoría de estos objetivos son incompatibles entre sí, por lo que hay que centrar la atención en aquel que sea de mayor interés. Por ejemplo, una planificación que realice un reparto equitativo del procesador no conseguirá optimizar el uso del mismo.

Hay que observar que algunos objetivos están dirigidos a procesos interactivos, mientras que otros lo están a procesos *batch.*

3.7.1. Algoritmos de planificación

En esta sección se presentan los algoritmos de planificación más usuales.

Cíclica o *Round-robin*

El algoritmo cíclico está diseñado para hacer un reparto equitativo del tiempo del procesador, por lo que está especialmente destinado a los sistemas de tiempo compartido. El algoritmo se basa en el concepto de **rodaja *(slot)*** de tiempo.

Los procesos están organizados en forma de cola circular, eligiéndose para su ejecución el proceso cabecera de la cola. Un proceso permanecerá en ejecución hasta que ocurra una de las dos condiciones siguientes:

- El proceso pasa a estado de bloqueado, porque solicita un servicio del sistema operativo.
- El proceso consume su rodaja de tiempo, es decir, lleva ejecutando el tiempo estipulado de rodaja.

Un proceso que ha consumido su rodaja de tiempo es expulsado y pasa a ocupar el último lugar en la cola. De esta forma, se consigue que todos los procesos pasen a ejecutar, repartiendo el tiempo del procesador de forma homogénea entre ellos. La Figura *3.25* muestra cómo el proceso *5,* al consumir su rodaja de tiempo, pasa al final de la cola.

Según se construya el planificador, los procesos que pasan de bloqueados a listos se pueden incluir como primero o como último de la cola.

FIFO

En este caso, la cola de procesos en estado de listo está ordenada de acuerdo al instante en que los procesos pasan al estado de listo. Los que llevan más tiempo esperando están más cerca de la cabecera.

El algoritmo es sencillo, puesto que consiste en tomar para ejecutar al proceso de la cabecera de la cola. No se plantea expulsión, por lo que el proceso ejecuta hasta que realiza una llamada bloqueante al sistema operativo.

Es aplicable a los sistemas *batch,* pero no a los interactivos.

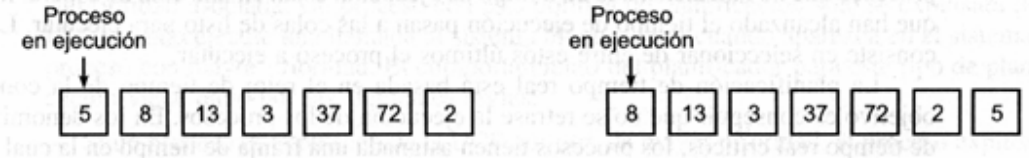


Figura *3.25. Planificación cíclica.*

Prioridades

En el algoritmo de prioridades se selecciona para ejecutar el proceso en estado de listo que tenga la máxima prioridad.

Cuando las prioridades son fijas puede surgir el problema de la **inanición,** que implica que un proceso puede estar esperando indefinidamente sin llegar a ejecutar. Esto ocurrirá si van apareciendo siempre procesos de mayor prioridad que estén en estado de listo.

Para evitar este problema, se puede añadir un mecanismo de envejecimiento, que se encargue dc aumentar la prioridad a los procesos que lleven un determinado tiempo esperando a ser ejecutados. Por ejemplo, suponiendo que la mayor prioridad es la 0, un proceso de prioridad 22 que lleve más de X ms esperando en estado de listo pasaría a prioridad 21, si pasados otros **X** ms siguiese sin ejecutar pasaría a prioridad 20 y así sucesivamente. Una vez que haya ejecutado volverá a tomar su prioridad normal de 22.

Dado que puede haber varios procesos listos con el mismo nivel de prioridad, es necesario utilizar otro algoritmo para decidir entre ellos. Se podría utilizar, por ejemplo, un cíclico, si el sistema es interactivo o un FIFO si es *hatch.* Los algoritmos basados en prioridades suelen ser con expulsión.

Primero el trabajo más corto

Este algoritmo exige conocer a *priori* el tiempo de ejecución de los procesos, por lo que es aplicable a trabajos *batch* repetitivos cuyo comportamiento se tenga analizado.

El algoritmo consiste en seleccionar para ejecución al proceso listo con menor tiempo de ejecución. No se plantea expulsión, por lo que el proceso sigue ejecutándose mientras lo desee.

La ventaja de este algoritmo es que produce el menor tiempo de respuesta, pero a costa de penalizar los trabajos de mayor tiempo de ejecución. También puede sufrir de

inanición, puesto que, en el caso de que estén continuamente apareciendo procesos con tiempo de ejecución pequeño, un proceso largo puede no llegar a ejecutar.

Aleatorio o lotería

Este algoritmo consiste en elegir al azar el proceso a ejecutar. Se puede basar en un generador de números pseudoaleatorios.

Planificación de sistemas de tiempo real

Los sistemas de tiempo real se caracterizan porque los procesos tienen que ejecutar en instantes predeterminados. Se pueden diferenciar dos tipos de procesos de tiempo real: a plazo fijo y periódico. La diferencia estriba en que los de plazo fijo tienen que ejecutar una vez, en un instante determinado, mientras que los periódicos deben ejecutar de forma repetitiva cada cierto tiempo.
Como muestra la Figura 3.26, se asocia a cada proceso el instante en el que debe ejecutar. Los procesos que no han alcanzado su tiempo de ejecución están en una cola de espera, mientras que los que han alcanzado el tiempo de ejecución pasan a las colas de listo para ejecutar. La planificación consiste en seleccionar de entre estos últimos el proceso a ejecutar.
La planificación de tiempo real está basada en el reloj de tiempo de la computadora y su objetivo es conseguir que no se retrase la ejecución de los procesos. En los denominados sistemas de tiempo real críticos, los procesos tienen asignada una franja de tiempo en la cual deben ejecutar y. en ningún caso, se ha de rebasar el tiempo máximo sin que el proceso complete su ejecución.

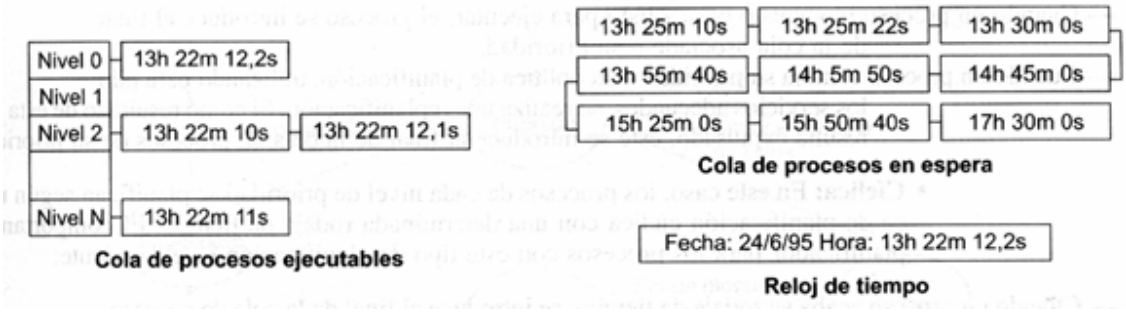


Figura 3.26. *Planificación de sistemas de tiempo real.*

Los sistemas de tiempo real se suelen diseñar de forma que estén bastante descargados, es decir, que tengan pocos procesos en estado de listo. De esta forma se consigue que no se retrase la ejecución de los mismos. Además, se evitan los mecanismos que introducen retardos en la ejecución, como puede ser la memoria virtual, puesto que la paginación puede introducir retardos inadmisibles en la ejecución.

Valoración de los algoritmos de planificación

En la valoración de los algoritmos de planificación hay que tener en cuenta los beneficios que presenta cada una de las alternativas a analizar. Además, no hay que

olvidar el tiempo de procesador que se consume debido al propio algoritmo (tiempo gastado en añadir elementos a las colas, ordenar éstas y analizarlas). En este sentido, un algoritmo menos bueno pero muy simple, como el aleatorio, que consume muy poco tiempo de procesador, puede ser globalmente mejor que otro más sofisticado, puesto que la ganancia de rendimiento conseguida puede no compensar el mayor tiempo consumido.

3.7.2. Planificación en POSIX

POSIX especifica una serie de políticas de planificación, aplicables a procesos y procesos ligeros, que debe implementar cualquier sistema operativo que ofrezca esta interfaz. En POSIX cada unidad ejecutable (proceso o proceso ligero) lleva asociada una política de planificación y una prioridad. Estos parámetros, como se verá en la Sección 3.11, pueden ser modificados mediante los servicios correspondientes.

Cada política de planificación lleva asociada un rango de prioridades. POSIX especit5ca que cada implementación debería ofrecer un rango de, al menos, 32 niveles de prioridad. El planificador elegirá siempre para ejecutar el proceso o proceso ligero con la prioridad más alta. Las políticas de planificación disponibles en POSIX son las siguientes:

- **FIFO:** Los procesos que se planifican según esta política se introducen al final de la cola de su prioridad asociada. Un proceso con esta política de planificación sólo se expulsará de la UCP cuando ejecute una llamada al sistema bloqueante o cuando aparezca en el sistema un proceso con mayor prioridad. El comportamiento del planificador para este tipo de planilicación viene dado por las siguientes reglas:
  __ Si un proceso es expulsado de la UCP por otro de mayor prioridad, el proceso expulsado pasa a ser el primero de la cola asociada a su prioridad

— Cuando un proceso bloqueado pasa a listo para ejecutar, el proceso se introduce al final de la cola asociada a su prioridad.
— Cuando un proceso cambia su prioridad o su política de planificación, utilizando para ello los servicios adecuados, se realiza una replanificación. Si como resultado de ésta el proceso resulta expulsado, éste se introduce al final de la cola de procesos de su prioridad.

- **Cíclica:** En este caso, los procesos de cada nivel de prioridad se planifican según una política de planificación cíclica con una determinada rodaja de tiempo. El comportamiento del planificador para los procesos con este tipo de planificación es el siguiente:

— Cuando un proceso acaba su rodaja de tiempo, se introduce al final de la cola de procesos de su prioridad.
— Cuando un proceso es expulsado por otro de mayor prioridad, se introduce al principio de su cola sin restaurar su rodaja de tiempo. De esta forma, cuando el proceso continúe la ejecución, lo hará hasta que consuma el resto de la rodaja de tiempo.

• **Otra:** Hace referencia a una política de planificación cuyo comportamiento depende de cada implementación. De acuerdo a esto, todo sistema operativo que siga la norma POSIX debe ofrecer al menos dos políticas de planificación: FIFO y cíclica, pudiendo, además, ofrecer otra política de planificación.

Observe que la política de planificación se realiza por proceso y, por tanto, las tres políticas conviven en el planificador. Observe, además, que la planificación en POSIX es con expulsión.

3.7.3. Planificación en Windows NT/2000

En Windows NT la unidad básica de ejecución es el proceso ligero y, por tanto, la planificación se realiza sobre este tipo de procesos. La Figura 3.27 (tomada de (Solomon, 1998)) describe los distintos estados por los que puede pasar un proceso ligero durante su ejecución. Estos estados son:

- **Listo.** Los procesos ligeros en este estado están listos para ejecutar.
- **Reserva.** Un proceso ligero en este estado será el siguiente proceso ligero a ejecutar en un procesador determinado. Sólo puede haber un proceso ligero en este estado por procesador.
- **Ejecución.** El proceso ligero permanece ejecutando hasta que se cumpla alguna de estas condiciones: el sistema operativo lo expulsa para ejecutar un proceso ligero de mayor prioridad, la rodaja de tiempo del proceso termina o bien el proceso ligero finaliza su ejecución.
- **Bloqueado.** Cuando el proceso ligero deja de estar bloqueado puede, dependiendo de su prioridad, comenzar su ejecución inmediatamente o pasar al estado de listo para ejecutar.
- **Transición.** Un proceso ligero entra en este estado cuando está listo para ejecutar, pero la pila que utiliza el sistema operativo para ese proceso no reside en memoria principal. Cuando la página vuelva a memoria, el proceso ligero pasará al estado de listo para ejecutar.
- **Finalizado.** Cuando un proceso ligero finaliza su ejecución, pasa a este estado. Una vez terminado, el proceso ligero puede o no ser eliminado del sistema. En caso de no ser eliminado, podría ser reutilizado de nuevo.

Windows NT implementa una planificación cíclica con prioridades y con expulsión. En Windows NT existen 32 niveles de prioridad, de 0 a 31, siendo 31 el nivel de prioridad máximo. Estos niveles se dividen en tres categorías:

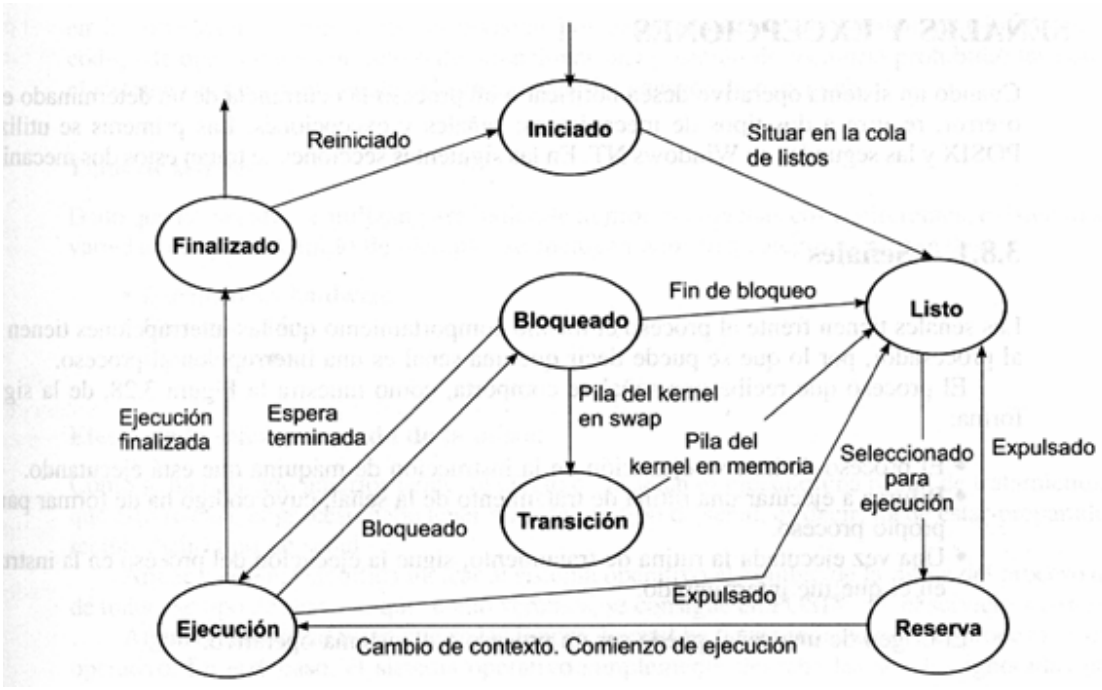


Figura **3.27.** *Estados de los procesos ligeros en Windows NT*

- Dieciséis niveles con prioridades de tiempo real (niveles 16 al 31).
- Quince niveles con prioridades variables (niveles 1 al *15).*
- Un nivel de sistema (0).

Todos los procesos ligeros en el mismo nivel se ejecutan según una política de planificación cíclica con una determinada rodaja de tiempo (Prestaciones 3.3). En la primera categoría, todos los procesos ligeros tienen una prioridad fija. En la segunda, los procesos comienzan su ejecución con una determinada prioridad y ésta va cambiando durante la vida del proceso, pero sin llegar al nivel 16. Esta prioridad se modifica según el comportamiento que tiene el proceso durante su ejecución. Así, un proceso (situado en el nivel de prioridades variable) ve decrementada su prioridad si acaba la rodaja de tiempo. En cambio, si el proceso se bloquea, por ejemplo, por una petición de E/S bloqueante, su prioridad aumentará. Con esto se persigue mejorar el tiempo de respuesta de los procesos interactivos que realizan *E/S.*

PRESTACIONES 3.3

Las versión Workstation de Windows NT (Professional en Windows 2000) tiene una rodaja de tiempo menor que la que se utiliza en la versión Windows NT Server (Advanced Server en Windows 2000). Esto se debe a que Windows NT Server está pensado para ejecutar aplicaciones servidoras que deberían disponer del suficiente tiempo de UCP para completar las peticiones de los clientes. En la versión Workstation lo que se desea, con una menor rodaja de tiempo, es tener un mejor tiempo de respuesta al usuario.

3.8. SEÑALES Y EXCEPCIONES

Cuando un sistema operativo desea notificar a un proceso la ocurrencia de un determinado evento, o error, recurre a dos tipos de mecanismos: señales y excepciones. Las primeras se utilizan en POS IX y las segundas en Windows NT. En las siguientes secciones se tratan estos dos mecanismos.

3.8.1. Señales

Las señales tienen frente al proceso el mismo comportamiento que las interrupciones tienen frente al procesador, por lo que se puede decir que una señal es una interrupción al proceso.

El proceso que recibe una señal se comporta, como muestra la Figura 3.28, de la siguiente forma:

- El proceso detiene su ejecución en la instrucción de máquina que está ejecutando.
- Bifurca a ejecutar una rutina de tratamiento de la señal, cuyo código ha de formar parte del propio proceso.
- Una vez ejecutada la rutina de tratamiento, sigue la ejecución del proceso en la instrucción en el que fue interrumpido.

El origen de una señal puede ser un proceso o el sistema operativo.

Señal proceso → proceso

Un proceso puede enviar una señal a otro proceso que tenga el mismo identificador de usuario (uid), a no los que lo tengan distinto (Aclaración *3.5).* Un proceso también puede mandar una señal a un grupo de procesos, que han de tener su mismo uid.

ACLARACIÓN 3.5

Un proceso del superusuario puede mandar una señal a cualquier proceso, con independencia de su uid.

Señal sistema operativo → proceso

El sistema operativo también toma la decisión de enviar señales a los procesos cuando ocurren determinadas condiciones. Por ejemplo, las excepciones de ejecución programa (el desbordamiento

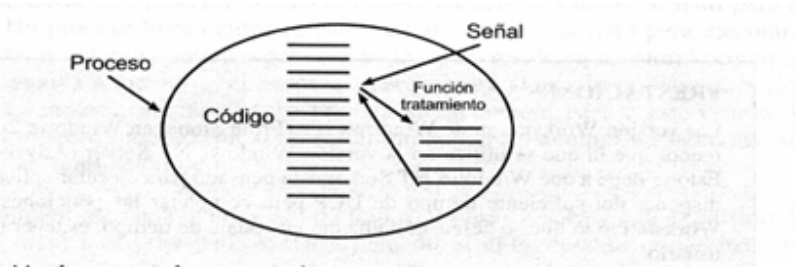


Figura 3.28. Recepción de una señal por parte de un proceso.

en las Operaciones aritméticas, la división por cero, el intento de ejecutar una instrucción con código de operación incorrecto o de direccionar una posición de memoria prohibida) las convierte el sistema operativo en señales al proceso que ha causado la excepción.

Tipos de señales

Dado que las señales se utilizan para indicarle al proceso muchas cosas diferentes, existen una gran variedad de ellas. A título de ejemplo, se incluyen aquí tres categorías de señales:

- Excepciones hardware.
- Comunicación.
- *FIS* asíncrona.

Efecto de la señal y armado de la misma

Como se ha indicado anteriormente, el efecto de la señal es ejecutar una rutina de tratamiento. Para que esto sea así, cl proceso debe tener armada ese tipo de señal, es decir, ha de estar preparado para recibir dicho tipo de señal.

Armar una señal significa indicar al sistema operativo el nombre de la rutina del proceso que ha de tratar ese tipo de señal, lo que, como veremos, se consigue en **POSIX** con el servicio sigaction.

Algunas señales admiten que se las ignore, lo cual ha de ser indicado por el proceso al sistema operativo. En este caso, el sistema operativo simplemente desecha las señales ignoradas por ese proceso. Un proceso también puede enmascarar diversos tipos de señales. El efecto es que las señales enmascaradas quedan bloqueadas (no se desechan), a la espera de que el proceso las desenmascare.

Cuando un proceso recibe una señal sin haberla armado o enmascarado previamente, se ejecuta la acción por defecto, que en la mayoría de los casos consiste en matar al proceso (Aclaración 3.6).

ACLARACIÓN 3.6

Conviene anotar en este punto el nombre del servicio que sirve para enviar una señal a un proceso. El servicio se llama `kill`, porque su uso más frecuente es para matar procesos.

3.8.2. Excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa y que requiere la ejecución dc un fragmento de código situado fuera del flujo normal de ejecución.
Las excepciones son generadas por el hardware o el software. Ejemplos de excepciones hardware incluyen la división por cero o la ejecución de instrucciones ilegales. Las excepciones software incluyen aquellas detectadas y notificadas por el sistema operativo o el propio proceso. Cuando ocurre una excepción, tanto hardware como software, el control es transferido al Sistema operativo, que ejecuta la rutina de tratamiento de excepción correspondiente. Esta rutina crea un registro de excepción que contiene información sobre la excepción generada. Si existe un manejador para la excepción generada, el sistema operativo transfiere el control a dicho manejador, en caso contrario aborta la ejecución del proceso.

El manejo de excepciones necesita el soporte del lenguaje de programación para que el programador pueda especificar el manejador a ejecutar cuando se produzca una excepción. Un esquema habitual es el que se presenta a continuación:

```
try{
    Bloque donde puede producirse una excepción
}
except{
    Bloque que se ejecutará si se produce una excepción
    en el bloque anterior
}
```

En el esquema anterior, el programador encierra dentro del bloque try el fragmento de código que quiere proteger de la generación de excepciones. En el bloque except sitúa el manejador de excepciones. En caso de generarse una excepción en el bloque try, el sistema operativo transfiere el control al bloque except que se encargará de manejar la correspondiente excepción.

Windows NT utiliza el mecanismo de excepciones, similar al anterior, para notificar a los procesos los eventos o errores que surgen como consecuencia del programa que están ejecutando. Win32 hace uso del concepto de **manejo de excepciones estructurado** que permite a las aplicaciones tomar el control cuando ocurre una determinada excepción. Este mecanismo será tratado en la Sección 3.12.4.

3.9. TEMPORIZADORES

El sistema operativo mantiene en cada BCP un temporizador que suele estar expresado en segundos. Cada vez que la rutina del sistema operativo que trata las interrupciones de reloj comprueba que ha transcurrido un segundo, decrementa todos los temporizadores que no estén a «0» y comprueba si han llegado a «0». Para aquellos procesos cuyo temporizador acaba de llegar a «0», el sistema operativo notifica al proceso que el temporizador ha vencido. En POSIX se genera una señal SIGALRN. En Win32 se ejecuta una función definida por el usuario y que se asocia al temporizador.

El proceso activa el temporizador mediante un servicio en el que especifica el número de segundos o milisegundos que quiere temporizar. Cuando vence la

temporización, recibirá la correspondiente señal o se ejecutará la función asociada al mismo.

3.10. SERVIDORES Y DEMONIOS

Los servidores y los demonios son dos tipos de procesos muy frecuentes y que tienen unas características propias que se analizan seguidamente.
Un servidor es un proceso que está pendiente de recibir órdenes de trabajo que provienen de otros procesos, que se denominan clientes. Una vez recibida la orden, la ejecuta y responde al peticionario con el resultado. La Figura 3.29 muestra cómo el proceso servidor atiende a los procesos clientes.

El proceso servidor tiene la siguiente estructura de bucle infinito:

- Lectura de orden. El proceso está bloqueado esperando a que llegue una orden.
- Recibida la orden, el servidor la ejecuta.

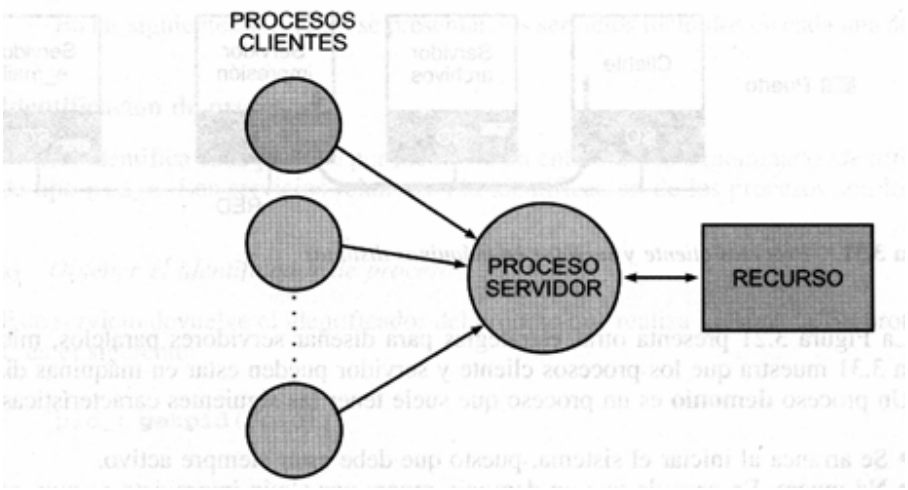


Figura 3.29. *Proceso servidor.*

- Finalizada la ejecución, el servidor responde con el resultado al proceso cliente y vuelve al punto de lectura de orden.

Una forma muy difundida de realizar la comunicación entre el proceso cliente y el servidor es mediante puertos. El proceso servidor tiene abierto un puerto, del que lee las peticiones. En la solicitud, el proceso cliente envía el identificador del puerto en el que el servidor debe contestar.

Un servidor será secuencial cuando siga estrictamente el esquema anterior. Esto implica que hasta que no ha terminado el trabajo de una solicitud no admite otra. En muchos casos interesa que el servidor sea paralelo, es decir, que admita varias peticiones y las atienda simultáneamente. Para conseguir este paralelismo se puede proceder de la siguiente manera:

• Lectura de la orden. El proceso está bloqueado esperando a que llegue una orden.
• Asignación de un nuevo puerto para el nuevo cliente.
• Generación de un proceso hijo que realiza el trabajo solicitado por el cliente.
• Vuelta al punto de lectura de orden.

De esta forma, el proceso servidor dedica muy poco tiempo a cada cliente, puesto que el trabajo lo realiza un nuevo proceso, y puede atender rápidamente nuevas peticiones. La Figura 3.30 muestra esta secuencia.

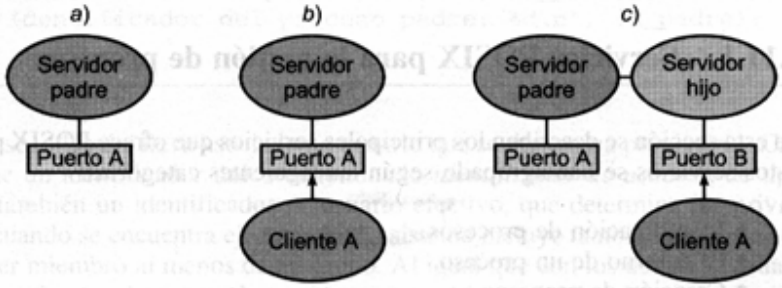


Figura 3.30. *Funcionamiento de un proceso servidor.*

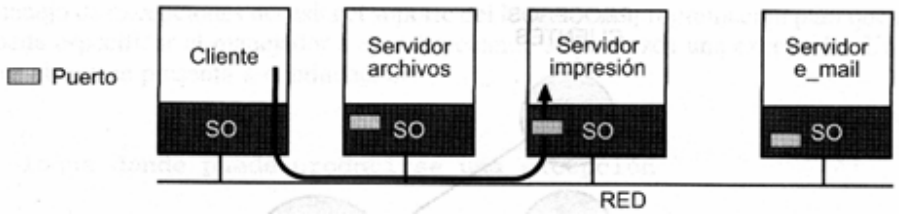


Figura 3.31. *proceso cliente servidor en maquinas distintas*

La Figura 3.21 presenta otras estrategias para diseñar servidores paralelos, mientras que la figura 3.3 muestra que los procesos cliente y servidor pueden estar en máquinas distintas.

Un proceso **demonio** es un proceso que suele tener las siguientes características:

- Se arranca al iniciar el sistema, puesto que debe estar siempre activo.
- No muere. En caso de que un demonio muera por algún imprevisto es muy recomendable que exista un mecanismo que detecte la muerte y lo rearranque.
- En muchos casos está a la espera de un evento. En el caso frecuente de que el demonio sea un servidor, el evento por el que espera es que llegue una petición al correspondiente puerto.
- En otros casos, el demonio tiene encomendada una labor que hay que hacer de forma periódica. ya sea cada cierto tiempo o cada vez que una variable alcanza un determinado valor.

- Es muy frecuente que no haga directamente el trabajo: lanza otros procesos para que lo realicen.
- Los procesos servidores suelen tener el carácter de demonios.
- Los demonios ejecutan en *haekground* y no están asociados a un terminal o procesos *login.*

Como ejemplos de procesos servidores se pueden citar los siguientes: el servidor de FTP el servidor de TELNET, el servidor de WEB y el *spooler* de impresora.

3.11. SERVICIOS POSIX

Esta sección describe los principales servicios que ofrece POSIX para la gestión de procesos, procesos ligeros y planificación. También se presentan los servicios que permiten trabajar con señales y temporizadores

3.11.1. Servicios POSIX para la gestión de procesos

En esta sección se describen los principales servicios que ofrece POSIX para la gestión de procesos. Estos servicios se han agrupado según las siguientes categorías:

- Identificación de procesos.
- El entorno de un proceso.
- Creación de procesos.
- Terminación de procesos.

En las siguientes secciones se presentan los servicios incluidos en cada una de estas categorías.

Identificación de procesos

POSIX identifica cada proceso por medio **de un entero** único denominado *identificador de proceso* de tipo pid_ti. Los servicios relativos a la identificación de los procesos son los siguientes:

a) *Obtener el identificador de proceso*

Este servicio devuelve el identificador del proceso que realiza la llamada. Su prototipo en C lenguaje es el siguiente:

Pid_t **getpid(void);**

b) *Obtener el identificador del proceso padre*

Devuelve el identificador del proceso padre. Su prototipo es el que se muestra a continuación.

pid ti **getppid(void);**

El Programa 3. 1 muestra un ejemplo de utilización de ambas llamadas.

Programa 3.1. Programa que imprime el identificador del proceso y el identificador de su proceso padre.

```
# include <sys/types.h>
#include <stdio.h>
main  ( )
{
        pid_id_proceso;
        pid_id_padre;

        id proceso = getpid ( );
        id_padre = getppid ( );

        printf(”identificador de proceso: %d\n”, íd_proceso>;
        printlf(”identificador del proceso padre: %d\n”, Id_padre>;
}
```

Cada proceso, además, lleva asociado un usuario que se denomina propietario. Cada usuario en el sistema tiene un identificador único denominado *identificador de usuario,* de tipo uid_ti. El proceso tiene también un identificador de usuario efectivo, que determina los privilegios que un proceso tiene cuando se encuentra ejecutando. El sistema incluye también grupos de usuarios, cada usuario debe ser miembro al menos de un grupo. Al igual que con los usuarios, cada proceso lleva asociado cl identificador de grupo al que pertenece y el identificador de grupo efectivo. Los servicios que permiten obtener estos identificadores son las siguientes:

c)*Obtener el identificador de usuario real*

Este servicio devuelve el identificador de usuario real del proceso que realiza la llamada. Su prototipo es:

```
uid_t getuid(void);
```

d) *Obtener el identificador de usuario efectivo*
Devuelve el identificador de usuario efectivo. Su prototipo es:

```
uid_t geteuid(void);
```

e) *Obtener el identificador de grupo real*

Este servicio permite obtener el identificador de grupo real. El prototipo que se utiliza para invocar este servicio es el siguiente:

```
gid_t getgid (void);
```

Obtener el identificador de grupo efectivo

Devuelve el identificador de grupo efectivo. Su prototipo es:

```
gid_t getegid (void);
```

El siguiente programa muestra un ejemplo de utilización de estos cuatro servicios.

Programa 3.2. Programa que imprime la información de identificación de un proceso.

```
#include <sys/types.h>
#include <stdio.h>
main ( )

printf("identificador de usuario: %d\n", getuid ());
printf("identificador de usuario efectivo: %d\n" ,geteuid ());
printf ("identificador de grupo: %d\n", getgid ());
printf("identificador de grupo efectivo: %d\n", getegid ());
}
```

El entorno de un proceso

El entorno de un proceso viene definido por una lista de variables que se pasan al mismo en el momento de comenzar su ejecución. Estas variables se denominan variables de entorno y son accesibles a un proceso a través de la variable externa environ, declarada de la siguiente forma:

```
extern char **environ;
```


Esta variable apunta a una lista de variables de entorno. Esta lista no es más que un vector de punteros a cadenas de caracteres de la forma nombre = valor, donde nombre hace referencia al nombre de una variable de entorno y valor al contenido de la misma.

El Programa 3.3 imprime la lista de variables de entorno de un proceso.

Programa 3.3. Programa que imprime el entorno del proceso.

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void main(int argc, char *argv)
{
      int i;
     printf(" de variables de entorno de %s\n", argv[0]);
```

```
    for(i=O; environ[i] NULL; i++)
    printf("environ[%d] = %s\n", 1, environ[i]);
}
```

Cada aplicación interpreta la lista de variables de entorno de forma específica. POSIX establece el significado de determinadas variables de entorno. Las más comunes son:

- HOME: directorio de trabajo inicial del usuario.
- LOGNAME: nombre del usuario asociado a un proceso.
- PATH: prefijo de directorios para encontrar ejecutables.
- TERM: tipo de terminal.
- TZ: información de la zona horaria.

a) Obtener el valor de una variable de entorno

El servicio getenv permite buscar una determinada variable de entorno dentro de la lista de variables de entorno de un proceso. La sintaxis de esta función es:

```
char *getenv(const char *name)
```

Esta función devuelve un puntero al valor asociado a la variable de entorno de nombre name. Si la variable de entorno no se encuentra definida, la función devuelve NIJLL.

El Programa 3.4 utiliza la función getenv para imprimir el valor de la variable de entorno
HOME.

Programa 3.4 Programa que imprime el valor de la variable HOME.

```
#include <stdio.h>
#include <stdlib.h>

main ()

char *home = NULL;
```



```
  home = getenv ("HOME");
  if (home = = NULL)
      printf("HOME no se encuentra definida\n">;
else
      printf("E1 valor de HOME es %s\n", home);
```

Creación de procesos

En esta sección se describen los principales servicios POSIX relativos a la creación de procesos.

a) Crear un proceso

La forma de crear un proceso en un sistema operativo que ofrezca la interfaz POSIX es invocando el servicio fork. El sistema operativo trata este servicio realizando una donación del proceso que lo solicite. El proceso que solicita el servicio se convierte en el proceso padre del nuevo proceso, que es, a su vez, el proceso hijo.

El prototipo de esta función es el siguiente:

Pid_t **fork** ();

La Figura 3.32 muestra que la donación del proceso padre se realiza copiando la imagen de memoria y el BCP. Observe que el proceso hijo es una copia del proceso padre en el instante en que éste solicita el servicio fork. Esto significa que los datos y la pila del proceso hijo son los que tiene el padre en ese instante de ejecución. Es más, dado que, al entrar el sistema operativo a tratar el servicio, lo primero que hace es salvar los registros en el BCP del padre, al copiarse el BCP se copian los valores salvados de los registros, por lo que el hijo tiene los mismos valores que el padre.

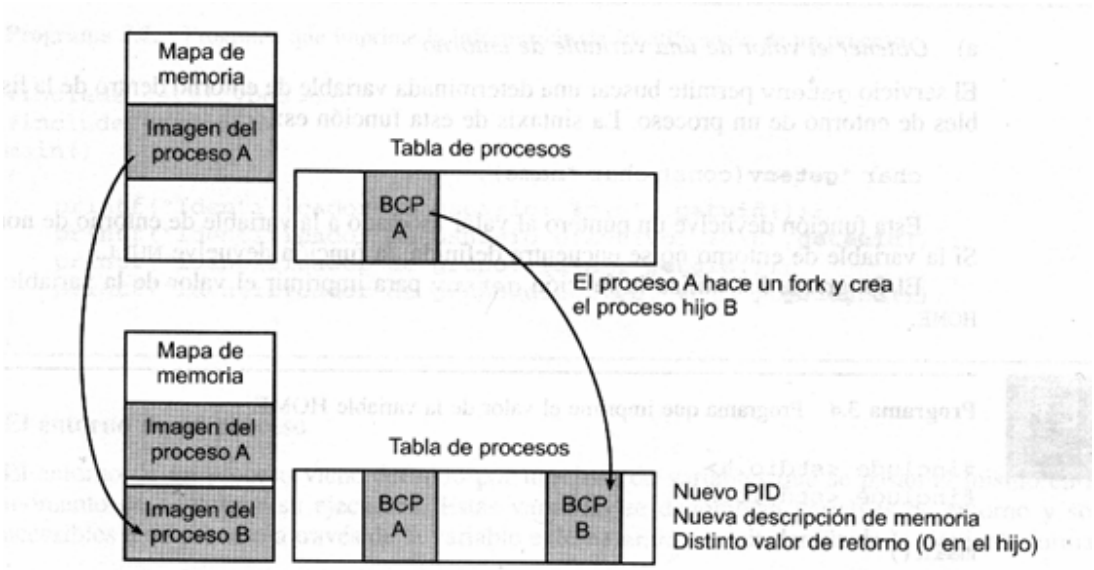


Figura 3.32. *Creación de un proceso mediante el servicio* fork.

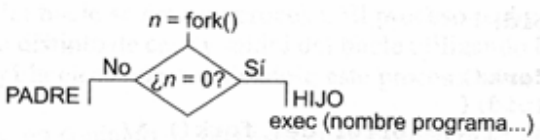


Figura 3.33. *Uso de la llamada* fork.

Esto significa, en especial, que el contador de programa de los dos procesos tiene el mismo valor, por lo que van a ejecutar la misma instrucción máquina. No hay que caer en el error de pensar que el proceso hijo empieza la ejecución del código en su punto de inicio. Repetimos que el hijo empieza a ejecutar, al igual que el padre, en la sentencia que esté después del fork.

En realidad, el proceso hijo no es totalmente idéntico al padre, puesto que algunos de los
valores del BCP han de ser distintos. Las diferencias más importantes son las siguientes:

- El proceso hijo tiene su propio identificador de proceso, distinto al del padre.
- El proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga los mismos segmentos con el mismo contenido, no tienen por qué estar en la misma zona de memoria (esto es especialmente cierto en el caso de sistemas sin memoria virtual).
- El tiempo de ejecución del proceso hijo se iguala a cero.
- Todas las alarmas pendientes se desactivan en el proceso hijo.
- El conjunto de señales pendientes se pone a vacío.
- El valor que retorna el sistema operativo como resultado del fork es distinto en el hijo y el padre.

— El hijo recibe un «0».
— El padre recibe el identificador de proceso del hijo.

Este valor de retorno se puede utilizar mediante una cláusula de condición para que el padre y el hijo sigan flujos de ejecución distintos, como se muestra en la Figura 3.33.

Observe que las modificaciones que realice el proceso padre sobre sus registros e imagen de memoria después del FORK no afectan al hijo y, viceversa, las del hijo no afectan al padre. Sin embargo, el proceso hijo tiene su propia copia de los descriptores del proceso padre. Esto hace que el hijo tenga acceso a los archivos abiertos por el proceso padre. El padre y el hijo comparten el puntero de posición de los archivos abiertos en el padre.

El Programa 3.5 muestra un ejemplo de utilización de la llamada fork. Este programa hace uso dc la función de biblioteca perror que imprime un mensaje describiendo el error de la última llamada ejecutada. Después de la llamada fork, los procesos padre e hijo imprimirán sus identificadores de proceso utilizando la llamada getpid, y los identificadores de sus procesos padre por medio de la llamada getppid. Observe que los identificadores del proceso padre son distintos en cada uno de los dos procesos.

Programa 3.5. Programa que crea un proceso.

```
#include <sys/ypes.h>
#ineclude <stdio.h>

main ()
{
```



```
pid_t pid;
pid = fork ();
switch(pid) {
```

```
case -1: /* error del fork<) *1
     perror(”fork”)
     break;
case 0: /* proceso hilo */
     printf(”Proceso %d; padre = %d \n”, getpidü, getppid();
     break;
default: /* padre */
     printf(”Proceso %d; padre = %d \n”, getpidú, getppid();
```

El código del Programa 3.6 crea una cadena de *n* procesos como se muestra en la Figura 3.34.

Programa 3.6. Programa que crea la cadena de procesos de la Figura 3.34.

```
#include <sys/types.h>
#include <stdio.h>

main ()
{
          piclit pid;
          int i;
          int n = 10;

          for (1 = 0; i < n; i++){
               pid = fork ();
               if (pid!= 0)
                    break;
}
printf(”E1 padre del proceso %d es %d\n”, getpid(), getppid());

}
```

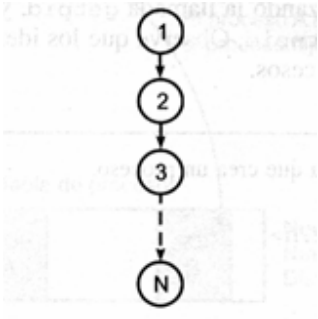


Figura 3.34. *Cadena de procesos.*

En cada ejecución del bucle se crea un proceso. El proceso padre obtiene el identificador del proceso hijo, que será distinto de cero y saldrá del bucle utilizando la sentencia break de C. El proceso hijo continuará la ejecución, repitiéndose este proceso hasta que se llegue al final del bucle.

El Programa 3.7 crea un conjunto de procesos cuya estructura se muestra en la Figura *3.35*

¡ **Programa** 3.7. Programa que crea la estructura de procesos de la Figura 3.35.

```
1 ¡  #include <stdio.h>
     #include <sys/types.h>

mamo

pidt pid;
mt i;
mt n = 10;

for (1 = 0; i < n; i++){
pid = fork<);
II                  <pid        0)
break;

printf("E1 padre del proceso %d es %d\n", getpidü, getppid<));
```

En este programa, a diferencia del anterior, es el proceso hijo el que finaliza la ejecución del bucle ejecutando la sentencia break, siendo el padre el encargado de crear todos los procesos.

h) *Ejecutar un programa*

El servicio exec de POSIX tiene por objetivo cambiar el programa que está ejecutando un proceso. Se puede considerar que el servicio tiene dos fases. En la primera se vacía el proceso de casi todo su contenido, mientras que en la segunda se carga en nuevo programa. La Figura 3.36 muestra estas dos fases.

2 N

Figura 3.35. *Creación de* n *procesos hL~os.*

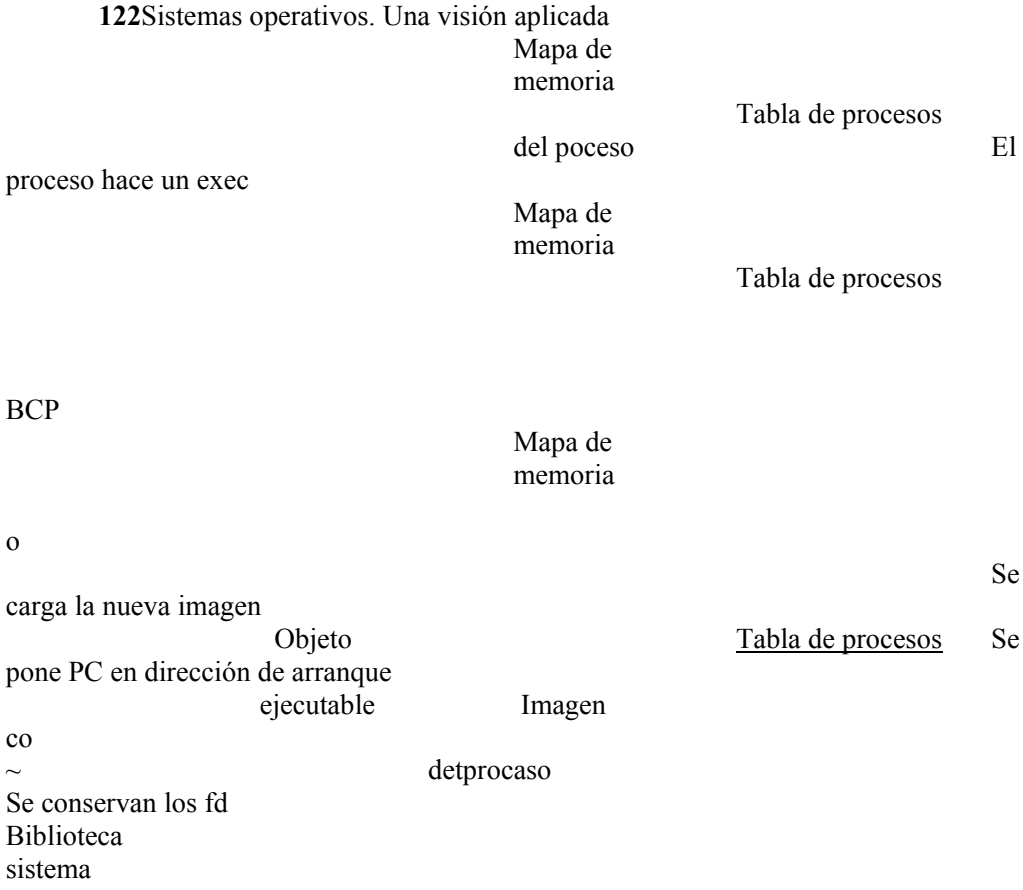


Figura 3.36. *Funcionamiento de la llamada* exec.

En la fase de vaciado del proceso se conservan algunas informaciones, como:

- Entorno del proceso. Que el sistema operativo incluye en la nueva pila del proceso.
- Algunas informaciones del BCP como: identificador de proceso, identificador del proceso padre, identificador de usuario y descriptores de archivos abiertos.

En la fase de carga hay que realizar las siguientes operaciones:

- Asignar al proceso un nuevo espacio de memoria.
- Cargar el texto y los datos iniciales en los segmentos correspondientes.
- Crear la pila inicial del proceso con el entorno y los parámetros que se pasan al programa.

• Rellenar el BCP con los valores iniciales de los registros y la descripción de los nuevos segmentos de memoria.

Recuerde que el servicio fork crea un nuevo proceso que ejecuta el mismo programa que el
proceso padre y que el servicio exec no crea un nuevo proceso, sino que permite que un proceso pase a ejecutar un programa distinto.
En POSIX existe una familia de funciones exec, cuyos prototipos **se** muestran a continuación:

in~ **excl(char '~path, char ~arg, ...);** icÉ execv(char *path char *argv[]); ints execle(char *path char *arg, **...);**

mt **execve(char** *path, char *argv[] char *envp[1); mt **execlp(char** *file const char *arg, ...);
mt **execvp<char** *file, char *argv[1);

La familia de funciones exec reemplaza la imagen del proceso actual por una nueva imagen. Esta nueva imagen se construye a partir de un archivo ejecutable. Si la llamada se ejecuta con éxito, ésta no devolverá ningún valor puesto que la imagen del proceso habrá sido reemplazada, en caso contrario devuelve —1.
La función main del nuevo programa llamado tendrá la forma:

mt main(int argc, char **argv)

donde argc representa el número de argumentos que se pasan al programa, incluido el propio nombre del programa, y argv es un vector de cadenas de caracteres, conteniendo cada elemento de este vector un argumento pasado al programa. El primer componente de este vector (argv [0 1) representa el nombre del programa.
El argumento path apunta al nombre del archivo ejecutable donde reside la nueva imagen del proceso. El argumento file se utiliza para construir el nombre del archivo ejecutable. Si el argunlcnt() file contiene el carácter /, entonces el argumento file constituye el nombre del archivo ejecutable. En caso contrario, el prefijo del nombre para el archivo se construye por medio de la búsqueda en los directorios pasados en la variable de entorno PATH.
El argumento argv contiene los argumentos pasados al programa y debería acabar con un puntero NULL. El argumento envp apunta al entorno que se pasará al nuevo proceso y se obtiene de la variable externa environ.
Los descriptores de los archivos abiertos previamente por el proceso que realiza la llamada exec permanecen abiertos en la nueva imagen del proceso, excepto aquellos abiertos con el valor Fi)_CLOEXEC. Los directorios abiertos en el proceso que realiza la llamada serán cerrados en la nueva imagen del proceso.
Las señales con la acción por defecto seguirán por defecto. Las señales ignoradas seguirán ignoradas por el nuevo proceso y las señales con manejadores activados tomarán la acción por defecto en la nueva imagen del proceso.

Si el archivo ejecutable tiene activo el bit de modo set—user—ID (Aclaración 3.7), el identiticador efectivo del nuevo proceso será el identificador del propietario del archivo ejecutable.

ACLARACIóN 3.7

Cuando un usuario ejecuta un archivo ejecutable con el bit de modo set -user- ID, el nuevo proceso creado tendrá como identificador de usuario efectivo el identificador de usuario del propietario del archivo ejecutable. Como se verá en el Capítulo 9, este identificador es el que se utiliza para comprobar los permisos en los accesos a determinados recursos como son los archivos. Pør tanto, cuando se ejecuta un programa con este bit, el nuevo proceso ejecuta con los permisos que tiene el propietario del archivo ejecutable.

El nuevo proceso hereda del proceso que realiza la llamada los siguientes atributos:

- ldcntiticador de proceso.
- ldentificador del proceso padre.
- ldentificador del grupo del proceso.
- ldentificador de usuario real.

- ldentificador de grupo real.
- Directorio actual de trabajo.
- Directorio raíz.
- Máscara de creación de archivos.
- Máscara de señales del proceso.
- Señales pendientes.

En el Programa 3.8 se muestra un ejemplo de utilización de la llamada execlp. Este programa crea un proceso hijo, que ejecuta el mandato ls —1, para listar el contenido del directorio actual de trabajo.

Programa 3.8. Programa que ejecuta el mandato 1s —1.

```
#include <sys/types.h>
#include <stdio.h>

mamo

pidt pid;
mt status;
```

```
pid = forkü;
switch(pid)
case —1: /* error del fork(> */
exit(-1);
case O: /* proceso hijo */
execlp("ls", "ls", "-1" ,NULL);
perror("exec")
break;
default: /* padre */

printf ("Proceso padre\n");
```

Igualmente, el Programa 3.9 ejecuta el mandato ls —1 haciendo uso de la llamada execvp.

Programa 3.9. Programa que **ejecuta el mandato ls** —1 mediante la llamada execvp.

```
#include <sys/types .
#include <stdio.h>

main(int argc, char **argv)

pidt pid;
char *argumentos[3];

argumentos[O] = argumentos[11 = "—1"; argumentos[2] = NULL;
```



```
pid = fork<);
switch(pid)
case ~-1: /~ error del fork() */
•Xit(-l)
case O: /~ proceso hijo */
execvp(argumentos[O] argumentos);
perror ( "exec");
break;
```

```
default: /* padre */
printf ( “Proceso padre\n”);
```

El siguiente programa (Programa 3.10) crea un proceso que ejecuta un mandato recibido en la línea de argumentos.

Programa 3.10. Programa que ejecuta el mandato recibido en la línea de mandatos.

```
#include <sys/types.h>
#include <stdio.h>

main(int argc, char **argv)



pid = forkü;
switch<pid)
case -1: /* error del fork() */
•xit(-1);
case O: /* proceso hijo */
if (execvp(argv[1], &argv[1])< O)
perror(”exec”)
default: /* padre */
printf (“Proceso padre\n”);
}
```

Terminación de procesos

En esta sección se presentan los servicios relacionados con la terminación de procesos.

a) Terminar la ejecucion de un proceso

Un proceso puede terminar su ejecución de forma normal o anormal. Un proceso puede terminar su ejecución de forma normal usando cualquiera de las tres formas siguientes:

- Ejecutando una sentencia return en la función main.
- Ejecutando la llamada exit.

```
Ejecutando la llamada _exit.
pidt pid;
```

Cuando un programa ejecuta dentro de la función main la sentencia return (valor), ésta es similar a exit (valor). El prototipo de la función exit es:

voidexit(int status);

Estos servicios tienen por función finalizar la ejecución de un proceso. Ambos reciben como parámetro un valor que sirve para que el proceso dé una indicación de cómo ha terminado. Como se verá más adelante, esta información la puede recuperar el proceso padre que, de esta forma, puede conocer cómo ha terminado el hijo.
La finalización de un proceso tiene las siguientes consecuencias:

- Se cierran todos los descriptores de archivos.
- La terminación del proceso no finaliza de forma directa la ejecución de sus procesos hijos.
- Si el proceso padre del proceso que realiza la llamada se encuentra ejecutando una llamada walt o wai tpid (su descripción se realizará a continuación), se notitica la terminación del proceso.
- Si el proceso padre no se encuentra ejecutando una llamada walt o waitpid, el código de finalización (status) se salva hasta que el proceso padre ejecute la llamada wait o waitpid.

- Si la implementación soporta la señal SIGCHLD, ésta se envía al proceso padre.
- El sistema operativo libera todos los recursos utilizados por el proceso.

Un proceso también puede terminar su ejecución de forma anormal, llamando a la función abort, por la recepción de una señal que provoca su finalización. La señal puede ser causada por un evento externo (p. ej.: se pulsa CTRL-C), por una señal enviada por otro proceso, o por un error dc ejecución, como, por ejemplo, la ejecución de una instrucción ilegal, un acceso ilegal a una posicion de memoria, etc.
Cuando un proceso termina de forma anormal, generalmente, se produce un archivo denominado *core* que incluye la imagen de memoria del proceso en el momento de producirse su terminación y que puede ser utilizado para depurar el programa.
En realidad, exit es una función de biblioteca que llama al servicio _exit después de preparar la terminación ordenada del proceso. El prototipo de _exit es:

```
void _exit(int status>;
```

En general se recomienda utilizar la llamada exit en vez de _exit, puesto que es más portahlc y permite volcar a disco los datos no actualizados. Una llamada a exit realiza las siguientes lu nc iones:

- Todas las funciones registradas con la función estándar de C atexit, con llamadas en orden inverso a su registro. Si cualquiera de estas funciones llama a exit, los resultados no serán
portahles. La sintaxis de esta función es:

```
intatexit(void (*func) (void));
```

- Todos los archivos abiertos son cerrados y todos los datos en el almacenamiento intermedio son copiados al sistema operativo.
- Sc llama a la función _exit.

El Programa 3.11 finaliza su ejecución utilizando la llamada exit. Cuando se llama a esta tuncion se ejecuta la función fin, registrada previamente por medio de la función atexit.

Programa 3.11. Ejemplo de utilización de exit y atexit.

```
#include <stdio.h>
#include <stdlib.h>

void íin(void)

printf('Fin de la ejecución del proceso %d\n", getpid());
retiu_n;


void main(void)
        if (atexit(fin)        O)
perror("atexit")
exit (1)


exit(O); /~ provoca la ejecución de la función fin *7
```

h> *Esperar por la jtnalizacion de un proceso hijo*

Permite a un ~~OCCS() padre esperar hasta que termine la ejecución de un proceso hijo (el proceso padre se queda bloqueado hasta que termina un proceso hijo). Existen dos formas de invocar este servicio:

```
pidt wait(int *status)
pidt waitpid(pid t pid, mt *status, mt options)
```

Ambas llamadas esperan la finalización de un proceso hijo y permiten obtener información
sobre ci estado de terminación del mismo.
La llamada wait suspende la ejecución del proceso hasta que finaliza la ejecución de uno de
sus procesos hijos. Esta función devuelve el identificador del proceso hijo cuya ejecución ha finali¡ado. Si status es distinto de NULL, entonces se almacena en esta variable información relativa al
proceso que ha terminado. Si el hijo retomó un valor desde la función main, utilizando la sentencia return, o pasó un valor como argumento a exit, éste valor puede ser obtenido por medio de la variable utilizando las siguientes macros definidas en el archivo de cabecera sys/wait:

- WTFEXITED(status): devuelve un valor verdadero (distinto de cero) si el hijo terminó normal me n te.
- WE:X1TSTATUs (status): permite obtener el valor devuelto por el proceso hijo en la llamada exit o el valor devuelto en la función main, utilizando la sentencia return. Esta macro

solo puede ser utilizada cuando WIFEXITED devuelve un valor verdadero (distinto de cero).

• WIFSIGNALED(status): devuelve un valor verdadero (distinto de cero) si el proceso hijo finalizó su cjecución como consecuencia de la recepción de una señal para la cual no se
había programado manejador.

• WTERI4SIG (status): devuelve el número de la señal que provocó la finalización del proceso hijo. Esta macro sólo puede utilizarse si WIFSIGNALED devuelve un valor verdadero
(distinto de cero).

• WIFSTOPPED (status): devuelve un valor verdadero si el estado fue devuelto por un proceso hijo actualmente suspendido. Este valor sólo puede obtenerse en la función waitpid
con la opción WUNTRACED, como se verá a continuación.

• WSTOPSIG (status): devuelve el número de la señal que provocó al proceso hijo suspenderse. Esta macro sólo puede emplearse si WIFSTOPPED devuelve un valor distinto de cero.

El servicio waitpid tiene el mismo funcionamiento si el argumento pid es —1 y el argumento
options es cero. Su funcionamiento general es el siguiente:

• Si pid es —1, se espera la finalización de cualquier proceso.

• Si pid es mayor que cero, se espera la finalización del proceso hijo con identificador pid.

• Si pid es cero, se espera la finalización de cualquier proceso hijo con el mismo identificador de grupo de proceso que el del proceso que realiza la llamada.

• Si pid es menor que —1, se espera la finalización de cualquier proceso hijo cuyo identificador de grupo de proceso sea igual al valor absoluto del valor de pid.

• El argumento options se construye mediante el OR binario de cero o más valores definidos en el archivo de cabecera sys/wait .h. De especial interés son los siguientes:

— WNOHANG. La función waitpid no suspenderá al proceso que realiza la llamada si el estado del proceso hijo especificado por pid no se encuentra disponible. WUNTRACED. Permite que el estado de cualquier proceso hijo especificado por pid que esté suspendido sea devuelto al programa que realiza la llamada waitpid.

E] siguiente programa (Programa 3.12) ejecuta un mandato recibido en la línea de argumentos por la funcion main. En este programa, el proceso padre realiza una llamada a la función wait para esperar la finalización del mandato. Una

vez concluida la ejecución, el proceso padre imprime información sobre el estado de terminación del proceso hijo.

Programa 3.12. Programa que imprime información sobre el estado de terminación de un proceso hijo.

```
#include <sys/types.h> #include <sys/wait.h> #include <stdio.h>

main(int argc, char **argv)
{
pidt pid;
mt valor;

pid = fork~
switch(pid)
```

```
case —1: /* error del fork() */
exit(-l)
case O: /~ proceso hijo */
mf (exeevp(argv[11, &argv[l]) <O)
perror(”exec”);
```

```
default: /* padre ~/ while (wait(&valor) 1= pid);
```



```
if (valor == O)
printfV’El mandato se ejecuto de formal normal\n”);
else
if (WIFEXITED(valor))
printf(”El hijo termino normalmente y su valor
devuelto fue %d\n”, WEXITEDSTATUS(valor));
}
if (WIFSIGNALZD(valor))
printfV’El hijo termino al recibir la señal
                                    %d\n”, WTERMSIG(valor));
```

La utilización de los servicios fork, exec, wait y exit hace variar la jerarquía de procesos (Fig. 3.37). Con el servicio fork aparecen nuevos procesos y con el exit desaparecen. Sin embargo, existen algunas situaciones particulares que conviene analizar.

La primera situación se refleja en la Figura 3.38. Cuando un proceso termina y se encuentra con que el padre no está bloqueado en un servicio wai t, se presenta el problema de dónde almacenar el estado de terminación que el hijo retorna al padre. Esta información no se puede almacenar en el BCP del padre, puesto que puede haber un número indeterminado de hijos que hayan terminado. Por ello, la información se deja en el BCP del hijo, hasta que el padre la adquiera mediante el oportuno wa it. Un proceso muerto que se encuentra esperando el wai t del padre se dice que está en estado zombie. El proceso ha devuelto todos sus recursos con excepción del BCP, que contiene el pid del padre y la palabra de estado de terminación.

pid P

padre

pidP pidP $pIdH_1$ pidP DidH EJ ~DD~D

$EZZ1$ ‘EZZ —EZ ‘$EZZIEZZ_1$

Figura 3.37. *Uso de las llamadas* fork, exec, wait y exit.

pid P

EJ

Texto

Datos
Pila
Archivos, tuberías,...
}
y
1

¡

¡ nit

Proceso A
forko

Figura 3.38. *Proceso zomhie.*

lnit

mit
¡ nit
)

) 1

mit
)

)

La segunda situación se presenta en la Figura 3.39 y se refiere al caso de que un proceso con hijos termine antes que éstos. De no tomar alguna acción correctora, estos procesos contendrían su BCP una información obsoleta del pid del padre, puesto que ese proceso ya no existe. La solución adoptada en UNIX es que estos procesos «huérfanos» los toma a su cargo el proceso *mit.* Er concreto, el proceso B de la mencionada figura pasará a tener como padre al **P~OCCSO** *ini,*.

Para que los procesos heredados por *mit* acaben correctamente, y no se conviertan en zombies,

el proceso *mit* está en un bucle infinito de walt.

La mayoría de los intérpretes de mandatos *(shells)* permiten ejecutar mandatos en *huckgroi*

finalizando la línea de mandatos con el carácter **&.** Así, cuando se ejecuta el siguiente mandato;

ls —1 &

El shell ejecutará el mandato ls —l sin esperar la terminación del proceso que lo ejecutó. Esto
permite al intérprete de mandatos estar listo para ejecutar otro mandato, el cual puede ejecutarse d

lnit
Proceso A
forkO
¡ nit

¡ nit
)
)
(

(
l'~igura 3.39. *En UNIX el proceso* ini *~ heredo los proce.~os hilos que se quedan sin padre.*
)

)
((
II
(

¡(irma concurrente a ls —1. Los procesos en *hackground,* además, se caracterizan porque no se pueden interrumpir con CTRL —C.
El Programa 3.13 crea un proceso que ejecuta un mandato pasado en la línea de argumentos en
I>a(k~4rOund.

Programa 3.13. Ejecución de un proceso en *hackground.*

```
#inciude <sys/types.h>
#include <stdio.h>
```

```
main(inL argc, char **argv)

picit pid;
                pué      fork(>;
SW]LCh(pid>

case -1: /~ error del fork() */
exit(-1>
case O: /* proceso hijo */

if (execvp(argv[l], &argv[1]) < O)
perrorV'exec")
defauit: /~ padre *7
exit (O);
```

1

3.11.2. Servicios POSIX de gestión de procesos ligeros

En esta sección se describen los principales servicios POSIX relativos a la gestión de procesos ligeros. Estos servicios se han agrupado de acuerdo a las siguientes categorías:

- Atributos de un proceso ligero.
- Creación e identificación de procesos ligeros.
- Terminación de procesos ligeros.

Atributos de un proceso ligero

Cada proceso ligero en POSIX tiene asociado una serie de atributos que representan sus propiedades. Los valores de los diferentes atributos se almacenan en un objeto atributo de tipo pth~eadattr_t. Existen una serie de servicios que se aplican sobre el tipo anterior y que permiten modificar los valores asociados a un objeto de tipo atributo. A continuación, se describen las principales funciones relacionadas con los atributos de los procesos ligeros.

a) *Crear atribulas de un I)r~(es() ligero*

Este servicio permite iniciar un objeto atributo que se puede utilizar para crear nuevos procesos ligeros. El prototipo de esta función es:

```
mt pthread_attr_init(pthread_attr_t *attr)
```


b) *Destruir atributos*

Destruye el objeto de tipo atributo pasado como argumento a la misma. Su prototipo es:

intpthread_attr_destroy(pthread_attr_t*attr>;

c) *Asignar el tamaño de la pila*

Cada proceso ligero tiene una pila cuyo tamaño se puede establecer mediante esta función, cuyo prototipo es el siguiente:

ini **pthread_attr_setstacksize<pthreadattr_t** ~ ini stacksize>;

d) *Obtener el tamaño de la pila*

El prototipo del servicio que permite obtener el tamaño de la pila de un proceso es:

ini **pthread_attr getntacksize(pthread** attr_t *attr, mt. *stacksize);

e) *Establecer el estado de terminación* El prototipo de este servicio es:

ini **pthread_attr_setdetachstate(pthread_attr_t** *attr,
ini detachstate)

Si el valor del argumento detachstate es PTHREAD_CREATE_DETACHED, el proceso ligero que sc cree con esos atributos se considerará como independiente y liberará sus recursos cuando finalice su ejecución. Si el valor del argumento detachs tate es PTHREAD_CREATE_JOINABLE, el proceso ligero se crea como no independiente y no liberará sus recursos cuando finalice su ejecución. En este caso es necesario que otro proceso ligero espere por su finalización. Esta espera se consigue mediante el servicio pthread bm, que se describirá más adelante.

f) *Obtener el estado de terminación*

Permite conocer el estado de terminación que se especifica en un objeto de tipo atributo. Su prototies:

mt **pthread attr.getdetachstate(pthread** attr_t *attr,
ini *detachstate>;

Creación e identificación de procesos ligeros

Los servicios relacionados con la creación **e identificación de procesos ligeros son los siguientes:**

a) *Crear un proceso ligero*

Este servicio permite crear un nuevo proceso ligero que ejecuta una determinada función. Su prototipo es:

mt **pthx-ead_create(pthread_t** ~ pthread_attr_r *attr,
void * (*start_routine) (void*h void *arg);

El primer argumento de la función apunta al identificador del proceso ligero que se crea, este identificador viene determinado por el tipo pthread_t. El segundo argumento especifica los atributos de ejecución asociados al nuevo proceso ligero. Si el valor de este segundo argumento es NULL, se utilizarán los atributos por defecto, que incluyen la creación del proceso como no independiente. El tercer argumento indica el nombre de la función a ejecutar cuando el proceso ligero comienza su ejecución. Esta función requiere un solo parámetro que se especifica con el cuarto argumento, arg.

h) *Obtener el identWcador de un proceso ligero*

Un proceso ligero puede averiguar su identificador invocando este servicio, cuyo prototipo es el siguiente:

pthread~t **pthread self** (void)

Terminación de procesos ligeros

Los servicios relacionados con la terminación de procesos ligeros son los siguientes:

a) *Esperar la terminación de un proceso ligero*

Este servicio es similar al walt, pero a diferencia de éste, es necesario especificar el proceso ligero por el que se quiere esperar, que no tiene por qué ser un proceso ligero hijo. El prototipo de la función es:

mt **pthread** join(pthread thid, void **value)

La función suspende la ejecución del proceso ligero llamante hasta que el proceso ligero con identificador *thid* finalice su ejecución. La función devuelve en el segundo argumento el valor que *pasa el proceso ligero que finaliza su ejecución en el servicio pthread exi* t, *que se verá a continuación. Únicamente se puede solicitar el servicio pthreadlioín sobre procesos ligeros creados*

1
como no independientes.
b) Finalizar la ejecución de un proceso ligero
Es análogo al servicio exlt sobre procesos. Su prototipo es:

ini **pthread exit** (void *value)

Incluye un puntero a una estructura que es devuelta al proceso ligero que ha ejecutado la correspondiente llamada a pthread joln, lo que es mucho más genérico que el parámetro que permite el servicio walt.
La Figura 3.40 muestra una jerarquía de procesos ligeros. Se supone que el proceso ligero A es el primario, por lo que corresponde a la ejecución del main. Los procesos B, C y D se han creado mediante pthread_create y ejecutan respectivamente los procedimientos b o, c O) y d O). El proceso ligero D se ha creado como «no independiente», por lo que otro proceso puede hacer una

)

pcreate Nc

5~so
ligero B

Figura 3.40. *E/emplo de jerarquía de procesos ligeros.*

operación hin sobre **~1. La** figura muestra que el proceso ligero **C hace una** operacion */0w* sobre el D, por lo que se queda bloqueado hasta que termine.
El Programa 3.14 crea dos procesos ligeros que ejecutan la función func. Una vez creados se espera su finalización con la función pthreadjoin.

Programa 3.14. Programa que crea dos procesos ligeros no independientes.

```
#include <pthread.h>
#include <st.dio.h>

void func(void)

printf('Thread %d \n", pthreadself(>);
pthreadexit (O);

mo i n

pthread_t thl, th2;

M se crean dos procesos ligeros con atribucos por defecto Y
pthread_create(&thl, NULL, func, NULL); pthreadcreate(&th2, NULL, func, NIJLL);

printf('E1 proceso ligero principal continua ejecutando\n");
p_create
```

Proceso
ligero O
1'

(

/~ se espera su terminación ~/
pthreadjoin(thl, NULL); **pthreadjoin(th2,** NULL);

exit (O)

Lii ~ que se muestra a **continuación (Programa** 3.15) **crea diez procesos** ligeros indepcndientcs. que liberan sus recursos cuando finalizan (se han creado con el atributo PJHHEAD CREATIi_DETACHED). En este caso, no se puede esperar la terminación de los procesos ligeros. p~r lo que el proceso ligero principal que ejecuta el código de la función main debe continuar su e~ccucion en paralelo con ellos. Para evitar que el proceso ligero principal finalice la ejecución dc la lbnción main. lo que supone la ejecución del servicio exit y, por tanto, la finalización de todo el proceso (Aclaración 3.8), junto con todos los procesos ligeros, el proceso ligero principal suspendc su ejecución durante cinco segundos para dar tiempo a la creación y destrucción dc los procesos ligeros que se han creado.

Prograria 3.15. Programa que crea diez procesos ligeros independientes.

```
# nciode <pthread. h>
jI 1 tic') wde tetcii o.

II de) irte tIAX _ '111 REA1)3 10

V() i Cl i UflC ( vn 1(11>

printt ('Thread lid \n", pthreadself()
pthread_exit (O);

VC)ii] naln))

lot 3;

pt~1iread dtt r — t attr;

pthCeád t LhidIMAXTHREADS]

7~ Oc inicia los atributos y se marcan como independientes *7
pthread attr mit (&attr);
pthread_attr_setdetachstate(&attr, PTHREADCREATEDETACHED)
    for(j        O; j e MAX_THREADS; j ++)
pthreadcreate(&thid[i1, &attr, func, NULL>;
```



```
/* El proceso ligero principal no puede esperar la finalización */
/~ de los procesos ligeros que ha creado y se suspende durante un */
/~ cierto tiempo esperando su finalización */
sleep(5>
```

El siguiente programa (Programa 3.16) crea un proceso ligero por cada número que se introduce. Cada proceso ligero ejecuta el código de la función imprimir.

¡ **Programa 3.16.** Programa que crea un proceso ligero por cada número introducido.

```
#include <pthread. h>
#include <stdio.h>

#define MAX_THREADS 10

void imprimir(int ~‘n)
{
printf(”Thread %d %d \n”, pthread_self(>, *fl);
pthread_exit (0);


main ()

pthread_attrt attr;
pthread_t thid;
mt num;

pthread_attrinit(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

while (1)
scanf(”%d”, &num); /~ espera */
pthreadcreate (&thid, &attr, imprimir, &num);
```

Este programa, tal y como se ha presentado, presenta un problema ya que falla si el número que se pasa a un proceso ligero es sobrescrito por el proceso ligero principal en la siguiente iteración del bucle while, antes de que el proceso ligero que se ha creado lo haya utilizado. Este problema requiere que los procesos ligeros se sincronicen en el acceso a este número. La forma de sincronizar procesos y procesos ligeros se tratará en el Capítulo *5*.

3.11.3. Servicios POSIX para la planificación de procesos

Como se describió en la Sección 3.7.2, **POSIX** proporciona planificación basada en prioridades. Estas prioridades pueden ser modificadas dinámicamente mediante servicios específicos.

POSIX incluye servicios para gestionar la política de planificación de los procesos y procesos ligeros. Como se vio anteriormente, POSIX especifica tres tipos de políticas de planificación. Estas
políticas se encuentran definidas por los siguientes símbolos declarados en el archivo de cabecera <sched. h>:

- SCHED_FIFO: política de planificación FIFO. Los procesos ejecutan según una planificación FIFO, pero pueden ser expulsados por procesos de mayor prioridad.
- SCHEDRR: política de planificación cíclica *(round-robin).* Los procesos dentro del mismo nivel de prioridad ejecutan según una política de planificación *round-robin.* Los procesos en este caso son expulsados de la UCP cuando acaba su rodaja de tiempo o cuando son expulsados por un proceso de mayor prioridad.
- SCHED_QTHER: otra política de planificación. Esta política se deja libre a la implenientación, la que debe estar perfectamente documentada.

Para cada política de planificación hay un rango de prioridades mínimo que toda implementación debe soportar. Para políticas FIFO y cíclicas, debe haber al menos 32 niveles de prioridad.
La prioridad de ejecución de los procesos y procesos ligeros se puede especificar mediante la
estructura schedparam, que debe incluir al menos el siguiente campo, que especifica la prioridad de planificación:

```
mt schedpriority;
/~ prioridad de ejecución del proceso */
```

A continuación, se describen los servicios que ofrece POSIX para la planificación de procesos y procesos ligeros.

Servicios de planificación de procesos

A continuación, se describen los principales servicios de planificación de procesos.

a) Modificar los parámetros de planificación de un proceso

Un proceso puede modificar su prioridad mediante el servicio:

mt **sched_setparam(piclit** pid, const struct schedparam *param)
Permite modificar la prioridad (especificada en el argumento param) del proceso pid. La función devuelve O en caso de éxito o --1 en caso de error.
Para modificar tanto la prioridad como la política de planificación de un proceso se utiliza el
servicio sched_setscheduler, cuyo prototipo es el siguiente:

mt **sched_scheduler(pidt** pid, mt policy,
const struct schedparam *param)

Cambia la política de planificación (SCHED_FIFQ SCHED_RR o SCHEDOTHER) y la prioridad
del proceso pid.

b) *Obtener los parámetros de planificación de un proceso*

Hay diversas funciones que permiten obtener información sobre los parámetros de planificación de un proceso. Estas son las siguientes:

mt **sched getparain(pidt** pid, struct schedpararn *param);

La luncion devuelve la prioridad del proceso pid en el campo schedprioritY de la estructura param.

ini **schedgetscheduler(Pid** t pid);

Devuelve la política de **planificación** del proceso (SCHED~FIFO, SCHEWRR o SCHE'D OTHER).

ini **schedgetpriOritYmifl(iflt** policy);
ini **schedgetpriOritYiftaX(iflt** poiicy);

Estas dos funciones devuelven los niveles de prioridad mínimo *y* máximo dc la **política** de
planificación poiicy.

Servicios de planificación de procesos ligeros

Los parámetros de planificación de los procesos ligeros se pueden asignar en su creación mediante los atributos correspondientes o mediante servicios adecuados.

a) *Atributo.~ (le* ***¡)ldllLJi(Uciofl***

Los procesos ligeros en un sistema operativo conforme a la norma POSIX pueden crearse con dos **arnhitos** dc planificación: ámbito de planificación global y ámbito de planificación local. En el *ámbito dc planificación global,* los procesos ligeros de un proceso compiten con el resto de procesos ligeros del sistema por la UCP. En el *amhito de planilicaciofl local,* los procesos ligeros de un proceso sólo compiten entre ellos por

la UCP. El ámbito de planificación de los procesos ligeros, así corno la prioridad y la política de planificación, se pueden añadir a los atributos de creación de un pro~cso ligero mediante los siguientes servicios:

Inc **pthreadattr~SetSCOPe(Pihread—ait_** E ~ ini conieni3nflScOpe>;
ini **pthread_attr getscope(pthread_attr_i** *attr,
ini *conteniiOnSCOpe)

La primera permite fijar en el atributo attr el **ámbito** de planificación y la segunda permite
obtenerlo. El ámbito de planificación se fija mediante los siguientes símbolos:

- PTHREAD_SCOPE SYSTEM especifica un ámbito de planificación global.
- PTHREAD_SCOPE_PROCESS especifica un ámbito de planificación local.

ini **pthreadattrsetSChedPOliCY(Pthread** attr t *aitr, ini poiicy>;
inc **pthrea~attrgetSChedPO1iCY(Pthread** attr_i *aiir, ini ~po1iCy);

La primera fija la política de planificación en el atributo atÉr y la segunda obtiene la política
de planificación almacenada en el atributo attr.

mt **pthreadattrfietSChedParalU(Pthread** actr t *aitr,
coflsi siruci sched~param *param)
ini **pthreadattrgetsChedParafll(Pthread** attr_t *atir,
struct schedparam *param)

La primera función fija la prioridad de planificación y la segunda la obtiene.

h) *Modificar los parámetros de plantficacion de un proceso ligero*

Un P~OCCS() ligero puede modificar su prioridad y política de planificación utilizando cl servicio:

in~ **Pthreadsetschedparam(pthread_t** thread, mt policy
const struct schedparam *param)

c) *Obtener los /)arametros de /)lantflcación de un proceso ligero*

Un ~~OCCS() ligero puede consultar su prioridad y su política de planificación mediante el siguiente servicio:

inti **pthreadgetschedparam(pthreaa_t** thread, mt *pOliCy
struct sched~param *param)

3.11.4. Servicios POSIX para gestión de señales y temporizadores

Lii esta scccion se presentan los principales servicios que ofrece POSIX para la gestión de señales y teinporizadores.
El archivo de cabecera signal . h declara la lista de señales posibles que se pueden enviar a los procesos en un sistema. A continuación se presenta una lista de las señales que deben incluir todas ¡as implementaciones. La acción por defecto para todas estas señales es la terminación anormal de proceso.

- SIGABRT: terminación anormal.
- SIGALRM: señal de fin de temporización.
- SIGFPE: operación aritmética errónea.
- SJGHUP: desconexión del terminal de control.
- £IGTLL: instrucción hardware inválida.
- SIGTN'F: señal de atención interactiva.
- SIGKILL: señal de terminación (no se puede ignorar ni armar).
- SICJPIPE: escritura en una tubería sin lectores.
- STGQUIT: señal de terminación interactiva.
- SIGSEV: referencia a memoria inválida.
- SIGTERN: señal de terminación.
- SIGUSR1: señal definida por la aplicación.
- STGUSR2: señal definida por la aplicación.

Si una iniplernentación soporta control de trabajos, entonces también debe dar soporte, entre otras, a las siguientes señales:

- SIGCHLD: indica la terminación del proceso hijo.
- SICONT: continuar si está detenido el proceso.
- £IGsToP: señal de bloqueo (no se puede armar ni ignorar).

La acción por defecto para la señal SIGCHLD, que indica la terminación de un proceso hijo. es ignorar la señal.
A continuación, sc describen los principales servicios POSIX relativos a las señales. Estos servicios se han agrupado de acuerdo a las siguientes categorías:

- Conjuntos de señales.
- Envío dc señales.

- Armado de una señal.

- Bloqueo de señales.
- Espera de señales.
- Servicios de temporización.

Conjuntos de señales

Como se ha indicado anteriormente, existe una serie de señales que un proceso puede recibir durante su ejecución. Un proceso puede realizar operaciones sobre grupos o conjuntos de señales. Estas operaciones sobre grupos de señales utilizan conjuntos de señal de tipo sigset~t y son las sigu lentes:

a) *Iniciar un conjunto de señales*

Existen dos servicios que permiten iniciar un conjunto de señales. El servicio

mt sigeiuptyset(sigset **t** *~~11)

inicia un conjunto de señales de modo que no contenga ninguna señal. El servicio

inicia un conjunto de señales con todas las señales disponibles en el sistema.

b) *Añadir una señal a un conjunto de señales*

Añade una señal a un conjunto de señales previamente iniciado. El prototipo de este servicio es:

mt **sigaddset(sigset** t *set, mt signo);

La función añade al conjunto set la señal con número signo.

c) *Eliminar una señal de un conjunto de señales*

Elimina una señal determinada de un conjunto de señales. El prototipo de este servicio es:

mt **sigdelset(sigset** t *set, mt signo);

Elimina del conjunto set la señal con número signo.

d) *Comprobar si una señal pertenece a un conjunto*

Permite determinar si una señal pertenece a un conjunto de señales. Su prototipo es:

mt sigisme~ber(sigset_t *set mt signo)

Esta función devuelve 1 si la señal signo se encuentra en el conjunto de señales set. En caso
contrario devuelve O.
El resto de las funciones devuelven o si tienen éxito o —1 si hay un error.

Procdsos

141

Envío de señales

Algunas señales como SIGSEGV o SIGBUS las genera el sistema operativo cuando ocurren ciertos errores. Otras señales se envían de unos procesos a otros utilizando el siguiente servicio:

una señal

Permite enviar una señal a un proceso. El prototipo de este servicio en lenguaje C es cl siguiente:

£nL **kill** (pidt pid, mt sig)

Envía la señal sig al proceso o grupo de procesos especificado por pid. Para que un ~fOCC5() pueda enviar una señal a otro proceso designado por pid, el identificador de usuario efectivo o real del proceso que envía la señal debe coincidir con el identificador real o efectivo del proceso que la recibe, a no ser que el proceso que envía la señal tenga los privilegios adecuados, por ejemplo es un
~fOCC5() ejecutado por cl superusuario.

Si pid es mayor que cero, la señal se enviará al proceso con identificador de proceso igual a
p~ ci. Si pid es cero, la señal se enviará a todos los procesos cuyo identificador de grupo sea igual al dentificador de grupo del proceso que envía la señal. Si pid es negativo pero distinto de —1, la

señal será enviada a todos los procesos cuyo identificador de grupo sea igual al valor absoluto de

1
p Ld. Para pid igual a —1, POSIX no especifica funcionamiento alguno.
Armado de tina **señal**

El servicio que permite armar una señal es:

mt sigaction(int sig, struct sigaction *act,
struct sigaction *oact);

Esta llamada tiene tres parámetros: el número de señal para la que se quiere establecer el inane jador. un puntero a una estructura de tipo struct sigaction para establecer el nuevo manejador y un puntero a una estructura también del mismo tipo que almacena información sobre el manejador establecido anteriormente.
La estructura sigact ion, definida en el archivo de cabecera signal .h, está formada por los siguientes campos:

```
    struct sigaction
                                        void (*sahandler) ~        /*
Manejador para la señal */
slgsets samask;                          /* Señales bloqueadas durante la
                                ejecución del manejador */
                                        mt sa_ filags;     /* opciones
especiales */
```

El primer argumento indica la acción a ejecutar cuando se reciba la señal. Su valor puede ser:

- SIG_DFL: indica que se lleve a cabo la acción por defecto que, en la mayoría de los casos, consiste en matar al proceso.
- SIG_IGN: especifica que la señal deberá ser ignorada cuando se reciba.
- Una función que devuelve un valor de tipo void y que acepta como parámetro un número entero. Cuando envía una señal al proceso el sistema operativo, éste coloca como parámetro

del manejador el número de la señal que se ha enviado. Esto permite asociar un mismo manejador para diferentes señales, de tal manera que el manejador realizará una u otra acción en función del valor pasado al mismo y que identifica a la señal que ha sido generada.

Si el valor del tercer argumento es distinto de NULL, la acción previamente asociada con la
señal será almacenada en la posición apuntada por oact. La función devuelve o en caso de éxito o
-1 si hubo algún error.
El siguiente fragmento de código hace que el proceso que lo ejecute ignore la señal SIGINT
que se genera cuando se pulsa CTLR-C.

```
struct sigaction act;
                act.sa_handler = SIG_IGN;          /* ignorar la señal */
act.saflags = O;        /* ninguna acción especial */ /* Se inicia el conjunto de señales a bloquear cuando
se reciba la señal */
sigemptyset<&act.sa_mask);
sigaction(SIGINT, &act, NULL)
```

Máscara de señales

La máscara de señales de un proceso proporciona un conjunto de señales que serán bloqueadas. Bloquear una señal es distinto a ignorarla. Cuando un proceso bloquea una señal, ésta no será enviada al proceso hasta que se desbloquee. Si el proceso ignora la señal, ésta simplemente sc ignora. Los servicios asociados con la máscara de señales de un proceso se describen a continua**ClOn.**

a) *Modificar la máscara de señales*

Permite modificar o examinar la máscara de señales de un proceso. El prototipo de la función es:

```
mt sigprocmask<int how, sigsett *set, sigset t *oset>;
```

Con esta función se permite bloquear un conjunto de señales de tal manera que su envío será
congelado hasta que se desbloqueen.
El valor del argumento how indica la manera en la cual el conjunto de señales set será camNado. Los posibles valores de este argumento se encuentran definidos en el archivo de cabecera signal .h y son los siguientes:

- SIG_BLK: añade un conjunto de señales a la actual máscara de señales del proceso.
- SIG_UNBLOCK: elimina de la máscara de señales de un proceso las señales que se encuentran en el conjunto set.

• SIG_SETMASK: crea la nueva máscara de señales de un proceso.

Si el segundo argumento de esta función es NTJLL, el tercero proporcionará la máscara de
señales del proceso que se está utilizando sin ninguna modificación. Si el valor del tercer argumento es distinto de NULL, la máscara anterior se almacenará en oset.
El siguiente fragmento de código bloquea la recepción de todas las señales.

sigset_t mask;

sigfillset(&mask)
sigprocmask(SIG_SETMASK, &mask, NULL);

Si sc quiere, a continuación, desbloquear la señal SIGSEGV, se deberá ejecutar el siguiente
fragmento de código:

sigsett mask;

sigemptyset (&mask)
sigaddset(&mask, SIGSEGV>;
sigprocmask(SIG_UNBLOCK, &mask, NULL);

h) *Obtener las señales bloqueadas pendientes de entrega*

Esta función devuelve el conjunto de señales bloqueadas que se encuentran pendientes de entrega al proceso. Su prototipo es:

mt sigpending(sigset t *~~>

La lbnción almacena en set el conjunto de señales bloqueadas pendientes de entrega.

Espera de señales

Cuando se quiere esperar la recepción de una señal, se utiliza el pause. Este servicio bloquea al P~OCCS() que la invoca hasta que llegue una señal. Su prototipo es:

mt pause(void);

Este servicio no permite especificar el tipo de señal por la que se espera, por tanto, cualquier
señal no ignorada sacará al proceso del estado de bloqueo.

Servicios de temporización

En esta sección se describen los servicios relativos a los temporizadores.

a) *Activar un temporizador*

Para activar un temporizador se debe utilizar el servicio:

unsigned mt alarm(unsigned mt seconds>

Esta función envía al proceso la señal SIGALRM después de pasados el número de segundos
especificados en el parámetro seconds. Si seconds es igual a cero, se cancelará cualquier petición realizada anteriormente.

El siguiente programa (Programa 3.17) ejecuta la función tratar_alarma cada tres segundos. Para ello, arma un manejador para la señal SIGALRN mediante la función sigaction. A continuación, se entra en un bucle infinito en el que el programa activa un temporizador especificando como argumento de la llamada alarm, 3 segundos. Seguidamente, el proceso suspende su ejecución, mediante el servicio pause, hasta que se reciba una señal, en concreto la señal
SIGALRM.

Durante la ejecución de la función tratar_alarma, se bloquea la recepción de la señal S1GINT, que se genera cuando se teclea CTRL-C.

Programa 3.17. Programa que imprime un mensaje cada 3 segundos.

```
#include <stdio.h>
#include <signal.h>

void tratar_alarma (void)

printf (“Activada \n”);


void main(void)

struct sigaction act;
sigset_s mask;

/* establece el manejador */
act.sa_handler = tratar_alarma; /“ función a ejecutar */ act.saflags = O; /* ninguna acción específica*/

/* Se bloquea la señal SIGINT cuando se ejecute la función tratar_alarma */
sigemptyset (&act .mask);
sigaddset(&act.mask, SIGINT);


sigaction(SIGALRM, &act, NULL>;


for ( ; ;
alarm(3)
pause ()
/* se arma el temporizador */
/* se suspende el proceso hasta que se
reciba una señal */
```

El Programa 3. 1 8 muestra un ejemplo en el que un proceso temporiza la ejecución de un proceso hijo. El programa crea un proceso hijo que ejecuta un mandato recibido en la línea de mandatos y espera su finalización. Si el proceso hijo no termina antes de que haya transcurrido una determinada cantidad de tiempo, el padre mata al proceso enviándole una señal mediante el servicio kill. La señal que se envía es SIGKILL, señal que no se puede ignorar ni armar.

Programa 3.18. Programa que temporiza ¡a ejecución de un proceso hijo.

```
#include <sys/types.h> #include <signal.h>
#include <stdio.h>
```



```
pidt pid;

void matar~proceso (void>

kill(pid, SIGKILL); 7* se envía la señal al hijo *7


main(inc argc, char **argv)

mt status;
char **argumentos;
scruct sigaction act;
argumentos = &argv[l];
/k Se crea el proceso hijo */
pid                forkü;
switch <pid)

case —1: 7* error del fork() */
exitL-l>
case O: /* proceso hijo *7
7* El proceso hijo ejecuta el mandato recibido *7 execvp(argumentos[O] argumentos>;
perrorV'exec")
exit(—l>
default: 7* padre *7
7* establece el manejador *7
act.sa_handler matarproceso; /~ función a ejecutar ~/
act.sa_flags = Q~ /~ ninguna acción específica *7
sigemptyset(&act.sa_mask);
sigaction(SIGALRM, &act, MiLL);
alarm(5)
7* Espera al proceso hijo *7
    wait(&status)
1

exit (O);
```

En el programa anterior, una vez que el proceso padre ha armado el temporizador, se bloquea esperando la *finalización* del proceso hijo, mediante una llamada a walt. Si la señal SIGALRM se recibe antes de que el proceso haya finalizado, el padre ejecutará la acción asociada a la recepción de esta señal. Esta acción se corresponde con la función matarproceso, que es la que se encarga de enviar al proceso hijo la señal SIGKILL. Cuando el proceso hijo recibe esta señal, se finaliza su ejecución.

h) *Suspender la ejecución de un proceso*

Suspende al proceso durante un número determinado de segundos. Su prototipo es:

mt sleep(unsigned mt seconds)

El proceso se suspende durante un número de segundos pasado como parámetro. El proceso
despierta cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.

3.12. SERVICIOS DE W1N32

Esta sección describe los principales servicios que ofrece Win32 para la gestión de procesos, procesos ligeros y planificación. También se presentan los servicios que permiten trabajar con excepciones y temporizadores.

3.12.1. Servicios de Win32 para la gestión de procesos

En Win32 cada proceso contiene uno o más procesos ligeros. Como se indicó anteriormente, en Windows el proceso ligero o *thread* es la unidad básica de ejecución. Los procesos en Win32 se diferencian de los de POSIX, en que Win32 no mantiene ninguna relación padre-hijo. Por conveniencia, sin embargo, en el texto se asumirá que un proceso padre crea a un proceso hijo. Los servicios que ofrece Win32 se han agrupado, al igual que en POSIX, en las siguientes categorías:

- Identificación de procesos.
- El entorno de un proceso.
- Creación de procesos.
- Terminación de procesos.

A continuación, se presentan los servicios incluidos en las categorías anteriores.

Identificación de procesos

En Win32, los procesos se designan mediante identificadores de procesos y manejadores. Un identiticador de proceso es un objeto de tipo entero que identifica de forma única a un proceso en el sistema. El manejador se utiliza para identificar al proceso en todas las funciones que realizan operaciones sobre el proceso.

Existen dos funciones que permiten obtener la identificación del proceso que realiza la llamada.

HANDLE **GetCurrentProcess** (VOID);
DWORD **GetCurrentProcessld(VOID);**

La primera devuelve el manejador del proceso que realiza la llamada y la segunda su identificador de proceso.
Conocido el identificador de un proceso, se puede obtener su manejador mediante el siguiente
servicio:

HANDLE **OpenProcess(DWORD** fdwAccess, BOOL flnherit, DWORD IdProcess);

El primer argumento especifica el modo de acceso al objeto que identifica al proceso con idcntificador Idprocess. Algunos de los posibles valores para este argumento son:

- PROCESS_ALL_ACCESS: especifica todos los modos de acceso al objeto.
- SYNCHRONIZE: permite que el proceso que obtiene el manejador pueda esperar la terminación del proceso con identificador IdProcess.

- PROCESS_TER~INATE: permite al proceso que obtiene el manejador finalizar la ejecución del proceso con identificador IdProcess.
- PROCESS~QUERY_INFORMATION: el manejador se puede utilizar para obtener información sobre el proceso.

El argumento flnherit especifica si el manejador devuelto por la función puede ser heredado por los **flUC VOS** procesos creados por el que realiza la llamada. Si su valor es TRUE, el manejador se puede heredar.
La función devuelve el manejador del proceso en caso de éxito o NULL si se produjo algún
error.

El entorno de un proceso

Un proceso recibe su entorno en su creación (mediante el servicio createProcess descrito en la siguiente sección). Cualquier proceso puede obtener el valor de una variable de entorno o modificar su valor mediante los dos servicios siguientes:

DWORD **GetEnvironmentVariable(LPCTSTR** lpszName, LPTSTR lpszValue,
DWORD cchValue);
BOOL **SetEnvironmentVariable(LPCTSTR** lpszName, LPTSTR lpzsValue>;

La primera función obtiene en lpszvalue el valor de la variable de entorno con nombre lpszName. El argumento cchValue especifica la longitud del buffer en memoria al que

apunta ¡lpszValue. **La** función devuelve la longitud de la cadena en la que se almacena el valor de la variable (lpszvalue) o O si hubo algún error.
La función SetEnvironmentVariable permite modificar el valor de una variable de entorno. Devuelve TRTJE si se ejecutó con éxito.
Un proceso, además, puede obtener un puntero al comienzo del bloque en el que se almacenan
las variables de entorno mediante la siguiente función:

LPVOID **GetEnvironmentStrings** (VOID);

El Programa 3.19 ilustra el uso de esta función para imprimir la lista de variables de entorno de
un proceso.

Programa 3.19. Programa que lista las variables de entorno de un proceso en Windows.

```
~    ¡
#include <windows.h>
#include cstdio.h>

void rnain(void)

char *lpszVar
void *lpvEnv;
lpvEnv                    GetEnvironinentStrizigs ~
if (lpvEnv == NULL)
printrf(”Error al acceder al entorno\n”)
exit<O~
```



```
/* las variables de entorno se encuentran separadas por un NULL ‘~/
/~ el bloque de variables de entorno termina también en NULL ~/
for (lpszVar = <char *) lpvEnv; lpszVar NULL; lpszVar++)
while (*lpszVar)

putchar<*lpszVar++);

putchar ( “\n”i;
```

Creación de procesos

En Win32, los procesos se crean mediante la llamada CreateProcess. **Este** servicio es similar ala combinación fork-exec de POSIX. Win32 no permite a un proceso cambiar su imagen de memoria y ejecutar otro programa distinto. El prototipo de la función CreateProcess es el siguiente:

EOOL CreateProcess

LPCTSTR lpszlmageName,
LPTSTR lpszComrnandLine,
LPSECURITY_ATTRIBUTES lpsaProcess,
LPSECURITY_ATTRIBLJTES lpsaThread,
BOQL flnheritHandles,
DWORD fdwCreate,
LPVOID lpvEnvironrnent,
LPCTSTR lpszcurdir,
LPSTARTUPINFO lpsistartlnfo,
5 LPPROCESS_INFQRMATION

Esta función crea un nuevo proceso y su proceso ligero principal. El nuevo proceso ejecuta el archivo ejecutable especificado en lpszlmageName. Esta cadena puede especificar el nombre de un archivo con camino absoluto o relativo, pero la función no utilizará el camino de búsqueda. Si lpszlmageName es NULL, se utilizará como nombre de archivo ejecutable la primera cadena delimitada por blancos del argumento lpszCommandLine.
El argumento lpszCommandLine especifica la línea de mandatos a ejecutar, incluyendo el

nombre del programa a ejecutar. Si su valor es NULL, la función utilizará la cadena apuntada por lpszlmageNarne como línea de mandatos. El nuevo proceso puede acceder a la línea de mandatos utilizando los parámetros argc y argv de la función main de C.

El argumento lpsaProcess determina si el manejador asociado al proceso creado y devuelto
por la función puede ser heredado por otros procesos hijos. Si es NJJLL, el manejador no puede

heredarse. Lo mismo se aplica para el argumento lpsaThread, pero relativo al manejador del proceso ligero principal devuelto por la funcion.

El argumento flnheritHandles indica si el nuevo proceso hereda los manejadores que mantiene el proceso que realiza la llamada. Si su valor es TRUE, el nuevo proceso hereda todos los manejadores, con los mismos privilegios de acceso, que tenga abiertos el proceso padre.
El argumento fdwCreate puede combinar varios valores que determinan la prioridad y la creación del nuevo proceso. Alguno de estos valores son:

- CREATE_SUS PEND: el proceso ligero principal del proceso se crea en estado suspendido y solo se ejecutará cuando se llame a la función ResumeThread descrita más adelante.

- DETACHED_PROCESS: para procesos con consola, indica que el nuevo proceso no tenga acceso a la consola del proceso padre. Este valor no puede utilizarse con el siguiente.
- CREATE_NEW_CONSOLE: el nuevo proceso tendrá una nueva consola asociada y no heredará ¡a del padre. Este valor no puede utiliLarse con el anterior.
- NORMAL PRIORITY CLASE indica un proceso sin necesidades especiales de planificación.

• HIGH PRIORITY CLASE: indica que el proceso se cree con una prioridad alta de planificación.
• IDLE PRIORITYCLASS: especifica que los procesos ligeros del proceso sólo ejecuten cuando no haya ningún otro proceso ejecutando en el sistema.
• REALTIME_PRIORITY_CLASS: indica un proceso con la mayor prioridad posible. Los procesos ligeros del nuevo proceso serán capaces de expulsar a cualquier otro proceso, incluyendo ¡os procesos del sistema operativo.

Fi parámetro lpEnvironment apunta al bloque del entorno del nuevo proceso. Si el valor es NULL, el nuevo proceso obtiene el entorno del proceso que realiza la llamada.
El argumento lpszcuraír apunta a una cadena de caracteres que indica el directorio actual de trabajo para el nuevo proceso. Si el valor de este argumento es NULL, el proceso creado tendrá el nhismo directorio que el padre.
El parámetro lpStartuplnfo apunta a una estructura de tipo STARTPINFO que especifica la apariencia de la ventana asociada al nuevo proceso. Para especificar los manejadores para la entrada, salida y error estándar deben utilizarse los campos hStdln, hStdøut y hStdErr. En este caso, el campo dwFlags de esta estructura debe contener el valor STARTFUSESTDHANDLES
Por último, en el argumento lpProcesslnformatjon puntero a una estructura de tipo PROCESE INFORMATION, se almacenará información sobre el nuevo proceso creado. Esta estructura tiene la siguiente definición:

```
typedef struct PROCESS_INFORMATION
HANDLE hProcess;
HANDLE hThread;
DWORD dwProcessld;
DWORD dwThreadld;
PROCESSINFOP1\4ATION;
```

En los campos hProcess y hThread se almacenan los manejadores del nuevo proceso y del proceso ligero principal de él. En dwProcessld se almacena el identificador del nuevo proceso y en dwThreadld el identificador del proceso ligero principal del proceso creado.
FI Programa 3.20 crea un proceso que ejecuta un mandato pasado en la línea de argumentos. El proceso padre no espera a que finalice, es decir, el nuevo proceso se ejecuta en *background.*

Programa 3.20. Programa que crea un proceso que ejecuta la línea de mandatos pasada como argumento.

```
¡
#include <windows.h>
#include <stdio.h>


void main(int argc, char argv)


STARTUPINFO si;
PRUCESSINFORMATION pi;


i f (!Createprocess
                                    NULL,    /~ utiliza la línea de mandatos */
```

argv [1], NTJLL, NULL, FALSE, o, NULL, NTJLL, &s1, &pj)
/~‘ línea de mandatos pasada como argumentos */ /* manejador del proceso no heredable*/ /* manejador del thread no heredable */ /* no hereda manejadores */
/* sin flags de creación */
/* utiliza el entorno del proceso */
/~ utiliza el directorio de trabalo del padre */
printf <“Error al crear el proceso. Error: %x\n”, GetLastError <)>;
ExitProcess<O>

/* el proceso acaba ~/
exit (O)

Terminación de procesos

Los servicios relacionados con la terminación de procesos se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso y servicios para esperar la terminación de un proceso. Estos servicios se describen a continuación:

a) Terminar la ejecución de un proceso

Un proceso puede finalizar su ejecución de forma voluntaria de tres formas:

- Ejecutando dentro de la función main la sentencia return.
- Ejecutando el servicio Exitprocess.
- La función de la biblioteca de C exit. Esta función es similar a ExitProcess.

El prototipo de la función ExitProcess es el siguiente:

VOID **ExitProcess** <UINT nExitCode);

La llamada cierra todos los manejadores abiertos por el proceso y especifica el código de salida del proceso. Este código lo puede obtener otro proceso mediante el siguiente servicio:

BOOL **GetExitCodeprocess** <HANDLE hProcess, LPDWQRD lpdwExitCode);

Esta función devuelve el código de terminación del proceso con manejador hProcess. El proceso especificado por hProcess debe tener el acceso PROCESSQUERYINFORIVIATION (veáse OpenProcess). Si el proceso todavía no ha terminado, la función devuelve en lpdwExitCode el valor STILL_ALIVE en caso contrario almacenará en este valor el código de terminación.
Además, un proceso puede finalizar la ejecución de otro mediante el servicio:

BOOLTerminateProcess <HANDLEhProcess, UlNTuExitCode);

Este servicio aborta la ejecución del proceso con manejador hProcess. El código de terminación para el proceso vendrá dado por el argumento uExitCode. La función devuelve TRUE si se ejecuta con éxito.

h) *Esperar por la finalización de un proceso*

En Win32 un proceso puede esperar la terminación de cualquier otro proceso siempre que tenga permisos para ello y disponga del manejador correspondiente. Para ello, se utilizan las funciones de espera de propósito general, las cuales también se tratarán en el Capítulo *5.* Estas funciones son:

DWORD **WaitForSingleObject(HANDLE** høbject, D~PJQRD dwTimeOut);
DWORD **WaitForMultipleObjects(DWORD** cobjects, LPHANDLE lphObjects,

BOOL fWaitAll, DWORD dwTimeout>;

La primera función bloquea al proceso hasta que el proceso con manejador hObject finalice su ejecución. El argumento dwTimeOut especifica el tiempo máximo de bloqueo expresado en rnilisegundos. Un valor de O hace que la función vuelva inmediatamente después de comprobar si el proceso finalizó la ejecución. Si el valor es INFINITE, la función bloquea al proceso hasta que el
proceso acabe su ejecución.
La segunda función permite esperar la terminación de varios procesos. El argumento cObj ects especifica el número de procesos (el tamaño del vector 1phob~ects) por los que se desea esperar. El argumento lphObjects es un vector con los manejadores de los procesos sobre los que se quiere esperar. Si el parámetro fWaitAll es TRUE, entonces la función debe esperar por todos los procesos, en caso contrario la función vuelve tan pronto como un proceso haya acabado. El paráme1ro dwTimeOut tiene el significado descrito anteriormente.
Estas funciones, aplicadas a procesos, pueden devolver los siguientes valores:

- WAlT OBJECT_o: indica que el proceso terminó en el caso de la función WaitForSin gleObject, o todos los procesos terminaron si en WaitForMultipleObjects el parámetro fWaitAll es TRUE.
- WAlT_OBJECT_O+n donde O <= n <= cobjects. Restando este valor de WAlT_OBJECT_O se puede determinar el número de procesos que han acabado.
- WAlT TIMEOUT: indica que el tiempo de espera expiró antes de que algún proceso acabará.

Las funciones devuelven OxFFFFFFFF en caso de error.
El Programa 3.21 crea un proceso que ejecuta el programa pasado en la línea de mandatos. El programa espera a continuación a que el proceso hijo termine imprimiendo su código de salida.

Programa 3.21. Programa que crea un proceso que ejecuta la línea de mandatos pasada como argumento y espera por él.

```
#include <windows. h>
#include <stdio.h>
```

```
void main(int argc, char arqv)


STARTUPINFO si;
PROCESS_INFORMATION pi;
IDWORD code;
```



```
if   (!Createprocess( NTJLL,
          argv[l],      /*
          NULL,
          NULL,
          FALSE,
          Q
          NULL,
          NULL,
          &sj,
          &pi))
utiliza la línea de mandatos */
línea de mandatos pasada como argumentos */ manejador del proceso no heredable*/
manejador del thread no heredable */ no hereda manejadores */
sin flags de creación */ utiliza el entorno del proceso */
utiliza el directorio de trabajo del padre */

printf                    ("Error al crear el proceso. Error: %x\n", GetLastError W;
ExitProcess(O);

/" Espera a que el proceso creado acabe */
WaitForSingleObject(pi.hProcess, INFINITE);

/'~ Imprime el código de salida del proceso hijo */
if <!GetExitCodeProcess (pi.hProcess, &code)
printf ("Error al acceder al codigo. Error: %x\n",
GetLastError W;
ExitProcess (O>
}
printf("codigo de salida del proceso %d\n", code); ExitProcess(O);
```

3.12.2. Servicios de Win32 para la gestión de procesos ligeros

Los procesos ligeros son la unidad básica de ejecución en Windows. Los servicios de Win32 para la gestión de procesos ligeros pueden agruparse en las siguientes categorlas:

- Identificación de procesos ligeros.

- Creación de procesos ligeros.
- Terminación de procesos ligeros.

A continuación, se presentan los servicios incluidos en las categorías anteriores.

Identificación de procesos ligeros

En Win32, los procesos ligeros, al igual que los procesos, se identifican mediante identificadores de procesos ligeros y manejadores. Estos presentan las mismas características que los identificadores y manejadores para procesos, ya descritos en la Sección 3.12.1.

Existen dos funciones que permiten obtener la identificación del proceso ligero que realiza la
llamada.

HANDLE **GetCurrentThread** (VQID>;
DWORD **GetCurrentThreadld** (yO ID);
r

La primera devuelve el manejador del proceso ligero que realiza la llamada y la segunda su
identificador de proceso ligero.

Conocido el identificador de un proceso ligero, se puede obtener su manejador mediante el
siguiente servicio:

HANDLE **OpenThread<DWORD** fdwAccess, BOOL flnherit, DWORD Idthread);

El primer argumento especifica el modo de acceso al objeto que identifica al proceso **COfl**
identificador Idthread. Algunos de los posibles valores para este argumento son:

- THREAD_ALL_ACCESS: especifica todos los modos de acceso al objeto.
- SYNCHRONIZE: permite que el proceso que obtiene el manejador pueda esperar la terminación del proceso con identificador IdThread.
- THREAD_TERNINATE: permite al proceso que obtiene el manejador finalizar la ejecución del proceso con identificador IdThread.
- THREAD_QUERY INFORI~IATION: el manejador se puede utilizar para obtener información sobre el proceso.

El argumento fInherit especifica si el manejador devuelto por la función puede ser heredado por los nuevos procesos creados por el que realiza la llamada. Si su valor es TRUE, cl manejador se puede heredar.

La función devuelve el manejador del proceso en caso éxito o NULL si se produjo algún error.

Creación de procesos ligeros

En Win32, los procesos ligeros se crean mediante la función CreateThread. El prototipo de esta función es:

BOOL **CreateThread**
LPSECURITY_ATTRIBUTES lpsa,
DWORD cbStack,
LPTHREAD_START_ROUTINE lpStartAddr;
LPVOID lpxiThreadParam,
DWORD fdwCreate,
LPDWORD lpldThread);

Esta función crea un nuevo proceso ligero. El argumento lpsa contiene la estructura con los atributos de seguridad asociados al nuevo proceso ligero. Su significado es el mismo que el utilizado en la Iunción CreateProcess. El argumento cbStack especifica el tamaño de la pila asociada al proceso ligero. Un valor de o especifica el tamaño por defecto (1 MB). lpStartAddr apunta a la función a ser ejecutada por el proceso ligero. Esta función tiene el siguiente prototipo:

DWORDWINAPI ThreadFunc(LPVOID>;

La función a ejecutar acepta un argumento de 32 bits y devuelve un valor de 32 bits. El proceso ligero puede interpretar el argumento como un DWORD o un puntero. El parámetro lpvThreadParam almacena el parámetro pasado al proceso ligero. Si fdwCreate es O, el proceso ligero ejecuta inmediatamente después de su creación. Si su valor es CREATE_SUSPENDED, el pro**CC5O** ligero se crea en estado suspendido. En lpldThread se almacena el identificador del nuevo P~OCC5() ligero creado.
La función Crea teThread devuelve el manejador para el nuevo proceso ligero creado o bien
NULL en caso de error.

Una vez creado un proceso ligero, éste puede suspenderse mediante la función:

DWORD **SuspendThread(HANDLE** hThread);

Un proceso ligero suspendido puede ponerse de nuevo en ejecución mediante la función:

DWORD **ResunieThread** (HANDLE hThread);

Terminación de procesos ligeros

Al igual **que** con los procesos en Win32, los servicios relacionados con la terminación de procesos ligeros se agrupan en dos categorías: servicios para finalizar la ejecución de un proceso ligero y

servicios para esperar la terminación de un proceso ligero. Estos últimos son los mismos que los empleados para esperar la terminación de procesos (waitForSingleøbject y WaitForMultipleObjects) y no se volverán a tratar.

a) *Terminar la ejecución de un proceso*

Un proceso ligero puede finalizar su ejecución de forma voluntaria de dos formas:

Ejecutando dentro de la función principal del proceso ligero la sentencia return

- Ejecutando el servicio ExitThread.

El prototipo de la función ExitThread es:

VOID **ExitThread(DWORD** dwExitCcde>;

Con esta función un proceso ligero finaliza su ejecución especificando su código de salida
mediante el argumento dwExitCode. Este código puede consultarse con la función GetExitCcdeThread, similar a la función GetExitCodePrccess.

Un proceso ligero puede también abortar la ejecución de otro proceso ligero mediante el servicio.

BOOL **TerminateThread(HANDLE** hThread, DWORD dwExitCode);

Similar a la función TerminateProcess.

3.12.3. Servicios de planificación en Win32

Win32 ofrece servicios para que los usuarios puedan modificar aspectos relacionados con la prioridad y planificación de los procesos y de los procesos ligeros. Como se describió en la Sección 3.7.2, la prioridad de ejecución va asociada a los procesos ligeros, ya que son éstos las unidades básicas de ejecución. La clase de prioridad va asociada a los procesos.
La prioridad de cada proceso ligero se determina según los siguientes criterios:
- La clase de prioridad de su proceso.
- El nivel de prioridad del proceso ligero dentro de la clase de prioridad de su proceso.

En Windows NT existen seis clases de prioridad que se fijan inicialmente en la llamada createprocess. Estas seis clases son:

- IDLE_PRIORITY_CLASS, con prioridad base 4.
- BELOW_NORMAL_PRIORITY_CLASS, con prioridad base 6.
- NORMAL_PRIORITY_CLASS, con prioridad base 9.

Proceso

- ABOVE_NORMAL_PRIORITY_CLASS, con pnoridad base 10.
- HIGH_PRIORITY_CLASS, con prioridad base 13.
- REAL_TIME_PRIORITY_CLASE, con prioridad base 24.

La clase de prioridad por defecto para un proceso es NORMAL_PRIORITY_CLASS. Un proceso puede modificar o consultar su clase o la de otro proceso utilizando los siguientes
servicios:

BOOL SetPriorityclass(HANDLE hProcess, DWORD fdwPricrityClass); DWORD GetPriorityClass (HANDLE hProcess);

Las prioridades de los procesos ligeros son relativas a la prioridad base del proceso al que pertenecen. Cuando se crea un proceso ligero, éste toma la prioridad de su proceso. La prioridad de cada proceso ligero varía dentro del rango ±2 desde la clase de prioridad del proceso. Estos cinco valores vienen determinados por:

- THREAD_PRIORITY_LOWEST.
- THREAD_PRIORITY_BELOW_NORMAL.
- THREAD_PRIORITY_NORMAL.
- THREAD_PRIORITY_ABOVE_NORMAL.
- THREAD_PRIORITY_HIGHEST.

Todos los procesos ligeros se crean utilizando THREAD_PRIORITY NORMAL, lo que significa que su prioridad coincide con la clase de prioridad de su proceso.
Para modificar o consultar el nivel de prioridad de un proceso ligero se deben utilizar las siguientes funciones:

BOOL SetThreadPriority(HANDLE hThread, DWORD fdwPriority); DWORD GetThreadPriority(HANDLE hProcess);

Además de los valores relativos anteriores, existen dos valores absolutos que son:

- THREAD_PRIORITY_IDLE con valor 1 o 16 para procesos de tiempo real.
- THREAD_PRIORITY_TIME_CRITICAL, con valor 15 o 31 para procesos de tiempo real.

3.12.4. Servicios de Win32 para el manejo de excepciones

Como se vio en la Sección 3.8, una excepción es un evento que ocurre durante la ejecución de un

1
programa y que requiere la ejecución de un código situado fuera del flujo normal de ejecución. Win32 ofrece un *manejo de excepciones estructurado,* que permite la gestión de excepciones software y hardware. Este manejo permite la especificación de un bloque de código o *manejador de*

1~
excepción a ser ejecutado cuando se produce la excepción.
El manejo de excepciones en Windows necesita del soporte del compilador para llevarla a cabo. El compilador de C desarrollado por Microsoft ofrece a los programadores dos palabras reservadas que pueden utilizarse para construir manejadores de excepción. Estas son: __try y
except. La palabra reservada __try identifica el bloque de código que se desea proteger de errores. La palabra __except identifica el manejador de excepciones.

En las siguientes secciones se van a describir los tipos y códigos de excepción y el uso de un rnanejador de excepción.

‘1

Tipos y códigos de excepción

El código de excepción puede obtenerse utilizando la siguiente función:

DWORD **GetExceptionCode** (VOID);

Esta función debe ejecutarse justo después de producirse una excepción. La función devuelve
cl valor asociado a la excepción. Existen muchos tipos de excepciones, algunos de ellos son:

- EXCEPTION_ACCESS_VIOLATION: se produce cuando se accede a una dirección de memoria inválida.
- EXCEPTION_DATATYPE_MISAL IGMENT: se produce cuando se accede a datos no alineados.
- EXCEPTION_INT_DIVIDE_BY_ZERO: se genera cuando se divide por cero.

- EXCEPTION_PRIV_INSTRUCTION: ejecución de una instrucción ilegal.

Uso de **un manejador de excepciones**

La estructura básica de un manejador de excepciones es:

try
/~ bloque de código a proueger */

except (expresión)

/~ manejador de excepciones ~/

El bloque de código encerrado dentro de la sección try representa el código del programa que se quiere proteger. Si ocurre una excepción mientras se está ejecutando este fragmento de código, el sistema operativo transfiere el control al manejador de excepciones. Este manejador se encuentra dentro de la sección _except. La expresión asociada a _except se evalúa inmediatamente después de producirse la excepción. La expresión debe devolver alguno de los siguientes val ores:

- EXCEPTION_EXECUTE_HANDLER: el sistema ejecuta el bloque except.
- EXCEPTION_CONTINUE_SEARCH: el sistema no hace caso al manejador de excepciones, continuando hasta que encuentra uno.
- EXCEPTION_CONTINUE_EXECUTION: el sistema devuelve inmediatamente el control al punto en el que ocurrió la excepción.

El Programa 3.22 muestra una versión mejorada de la función de biblioteca de C strcpy. Esta
nueva función detecta la existencia de punteros inválidos devolviendo NULL en tal caso.

Programa 3.22. Versión mejorada de strcpy utilizando un manejador de excepciones.

```
LPTSTR StrcpySeguro(LPTSTR si, LPTSTR s2)
t ry
return strcpy(sl, s2);

1
except<GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
EXCEPTION_EXECUTE_HANDLER EXCEPTION_CONTINUE SEARCH)
return NULL;
```


Si se produce un error durante la ejecución de la función strcpy, se elevará una excepción y
se pasará a evaluar la expresión asociada al bloque except.
Una hrma general de manejar las excepciones que se producen dentro de un bloque try.
utilizando la función GetExceptionCode. es la que se muestra en el Programa 3.23.
Cuando sc produce una excepción en el programa 3.23, se ejecuta la función GetExcept ion— C'ode que devuelve el código de excepción producido. Este valor se convierte en el argumento para la función filtrar. La función filtrar analiza el código devuelto y devuelve el valor adecuado para la expresión de _except. En el Programa 3.23 se muestra que en el caso de que la excepción se hubiese producido por una división por cero, se devolvería el valor EXCEPTION_EXECUTE_HANDLER que indicaría la ejecución del bloque except.

Programa 3.23. Manejo general de excepciones.

```
~try
7k bloque de código a proteger */


except (Filtrar(GetExceptioncode())) f
/~ manejador de excepciones */



1)WORD Filcrar (DWORD Code)
switch(Code)


case EXCEPTION_INTDIVIDEBYZERO
                        return EXCEPTIQN_EXECUTEHANDLER;
```

3.12.5. Servicios de temporizadores

Esta sección describe los servicios de Win32 relativos a los temporizadores.

Activación de **temporizadores**

Para crear un temporizador, el programador debe utilizar en primer lugar el servicio SetTimer **CUYo** prototipo es el siguiente:

UINT **SetTimer(HwND** hWnd, UINT nlDEvent, UINT uElapse, TIMERPROC lpTimerFunc);

El parámetro hwnd representa la ventana asociada al temporizador. Su valor es NULL a no ser que el gestor dc ventanas esté en uso. El segundo argumento es un identificador de evento distinto de cero. Su valor se ignora si es NULL. El argumento uElapse representa el valor del temporizador en milisegundos. Cuando venza el temporizador se ejecutará la función especificada en el cuarto argumento. La función devuelve el identificador del temporizador o cero en caso de error.

El prototipo de la función a invocar cuando vence el temporizador es:

XJOID CALLBACK **TimerFunc(HWND** hwnd, UINT uMsg, UINT idEvent, DWORD dwTime);

Los dos primeros argumentos pueden ignorarse cuando no se está utilizando un gestor de ventanas. El parámetro idEvent es el mismo identificador de evento proporcionado en la función SetTimer. El parámetro dwTime representa el tiempo del sistema en formato UTC *(Coordinated Universal Time,* tiempo universal coordinado).
La función SetTimer crea un temporizador periódico, es decir, si el valor de uElapse es 5 ms, la función lpTimerFunc se ejecutará cada *5* ms.
La siguiente función permite desactivar un temporizador:

BOOLKillTimer(HWNDhWnd, UINTuIdEvent);

El argumento uldEvent es el valor devuelto por la función setTimer. Esta función devuelve TRUE en caso de éxito, lo que significa que se desactiva el temporizador creado con SetTimer.
El Programa 3.24 imprime un mensaje por la pantalla cada 10 ms.

Programa 3.24. Programa que imprime un mensaje cada lO ms.

```
#include <windows. h>
#include <stdio.h>

void Mensaje(HWND hwnd, UJINT ms, UINT ide, DWORD time)

printf(”Ha vencido el temporizador\n”);
```

return;

void mamo

UINT idEvent = 2;
UINT intervalo = 10;
UINT tid;

tid = SetTimer(NULL, idEvent, intervalo, Mensaje); while (TRUE);

Suspender la ejecución de un proceso

Para suspender la ejecución de un proceso ligero un número determinado de milisegundos se utiliza el servicio:

VOID Sleep(DWORD cMilliseconds);

El proceso se suspende durante un número de milisegundos recibido como parámetro. El **P~OCCS()** despierta cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.

3.13. PUNTOS A RECORDAR

Un proceso es un programa en ejecución o de forma más precisa una unidad de procesamiento gestionada
por el sistema operativo.

~ El entorno del proceso consiste en un conjunto de varia-bies que se le pasan al proceso en el momento de su creación.

~ La multitarea se basa en tres características: paralelismo entre la E/S y el procesador, alternancia en los procesos de fases de E/S y de procesamiento, y memoria principal capat de almacenar varios procesos.

~ El planificador es el elemento del sistema operativo que se encarga de seleccionar el proceso que ha de ejecutar a continuación.

~ El activador es el elemento del sistema operativo que se encarga de poner en ejecución el proceso seleccionado por el planificador.

~ Se denomina grado de multiprogramación al número de procesos activos que mantiene el sistema.

~ Se denomina hiperpaginación a la situación de alta paginación que ocurre en sistemas con memoria virtual
cuando los conjuntos residentes de los procesos son demasiados pequeños.

~ Los elementos de información asociados a un proceso son: el estado del procesador, la imagen de memoria y las tablas del sistema operativo.

~ El estado del procesador está formado por el contenido de todos sus registros.

~ La imagen de memoria del proceso está formada por los espacios de memoria que está autorizado a utilizar.

~ En el modelo típico de imagen de memoria de un proceso, éste tiene un número fijo de segmentos de tamaño variable que son: el texto o código, los datos y la pila.
~ Los sistemas operativos actuales permiten que un pro**C~S()** tenga un número variable de segmentos de tamaño variable.
~ El bloque de control de proceso o BCP contiene la información básica del proceso. Esta incluye información
de identificación, el estado del procesador e información de control.
~ Un ~~OC~S() durante su ejecución pasa por una serie de estados: ejecución, bloqueado y listo para ejecutar.
~ Además de los estados básicos anteriores, los procesos pueden estar en los estados de espera y de suspendido.
Un proceso en espera todavía no ha entrado en el sistema y está a la espera de hacerlo.
Un proceso suspendido es aquel que no tiene ninguna página en memoria principal (están en la zona de intercambio).
U Se denomina cambio de contexto al conjunto de las dos operaciones siguientes: salvar el estado del procesador
en el BCP correspondiente y pasar a ejecutar otro programa. bien de otro proceso, bien del sistema operativo.

U Un proceso ligero es un flujo de ejecución que comparte la imagen de memoria y otras informaciones con otros procesos ligeros.
U La información propia de cada proceso ligero es: contador de programa, pila, registros y estado del proceso ligero (ejecutando, bloqueado o listo).
U El uso de procesos ligeros presenta varias ventajas: permite la separación de tareas que pueden ejecutarse de forma independiente, facilita la modularidad y aumenta la velocidad de ejecución de un trabajo, ya que se pueden aprovechar los tiempos de bloqueo de unos procesos ligeros para ejecutar otros.
U El objetivo de la planificación es el reparto del procesador entre los procesos o procesos ligeros que deseen y puedan ejecutar.
U La planificación a largo plazo tiene por objetivo añadir al sistema nuevos procesos, tomándolos de la lista de espera.
U La planificación a medio plazo decide qué procesos pasan a estar suspendidos y cuáles dejan de estarlo. Por tanto, cambia el grado de multiprogramación del sistema.
U La planificación a corto plazo se encarga de seleccionar el proceso en estado de listo que pasa a estado de ejecucion. Es la que se encarga de asignar el procesador.
U En una planificación sin expulsión un proceso conserva el procesador mientras no acabe o mientras no solicite un servicio al sistema operativo que lo bloquee.
U En una planificación con expulsión, el sistema operativo puede quitar a un proceso del estado de ejecución aunque éste no lo solicite.
U El algoritmo de planificación cíclico realiza un reparto equitativo del procesador, dejando que los procesos ejecuten durante unidades de tiempo denominadas rodajas.
U Un algoritmo de planificación FIFO ejecuta procesos según una política FIFO. Un proceso pasa al final de la cola cuando hace una llamada al sistema que lo bloquea. Si esto no ocurre, el proceso se ejecutará de forma indefinida hasta que acabe.
U En un algoritmo con prioridades se selecciona para ejecutar el proceso en estado listo con mayor prioridad. Este tipo de algoritmos suelen ser con expulsión, ya que un proceso abandona el procesador cuando pasa a listo un proceso de mayor prioridad.
U En una planificación basada en el algoritmo primero el trabajo más corto se elige para ejecutar el proceso **COfl** tiempo de ejecución más corto, lo que exige conocer *u priori* el tiempo de ejecución de los procesos.

U Un sistema operativo conforme a la norma POS IX debe implementar una política de planificación con expulsion basada en prioridades y al menos dos políticas de planificación: cíclica y FIFO.

U **Windows** NT realiza una planificación cíclica con prioridades y con expulsión.
U Las señales y las excepciones permiten a los sistemas operativos notificar a los procesos de la ocurrencia de determinados eventos que ocurren durante la ejecución de los mismos. POSIX utiliza el modelo de señales y Windows NT utiliza el modelo de excepciones.
U un servidor es un proceso que está pendiente de recibir órdenes de trabajo provenientes de otros procesos denominados clientes.
U POSIX dispone de una serie de servicios para la gestión de procesos, procesos ligeros y para planificación. También ofrece servicios para trabajar con señales y temporizadores.
U Win32 ofrece servicios para gestionar procesos, procesos ligeros y su planificación. También presenta servicios para trabajar con excepciones y con temporizado-res.

3.14. LECTURAS RECOMENDADAS

Son muchos los libros de sistemas operativos que cubren los temas tratados en este capítulo. Algunos de ellos son iCrow)ey, 1997], IMilenkovic, 1992], lSilberchatz, 1999], [Stallings, 1998] y lTanenbaum, 1995]. ISoloilion, 1998] describe en detalle la gestión de procesos de Windows NT y en [IEEE, 19961 puede encontrarse una descripción completa de todos los servicios POSIX descritos en este capítulo.

3.15. EJERCICIOS

3.1.¿Cuál de los siguientes mecanismos hardware no es un requisito para construir un sistema operativo multiprogramado con protección entre usuarios? Razone su respuesta.

a) Memoria virtual.
b) Protección de memoria.
e) Instrucciones de *FIS* que sólo pueden ejecutarse en modo kernel.
d) Dos modos de operación: kemel y usuario.

3.2.¿Puede degradarse el rendimiento de la utilización del procesador en un sistema sin memoria virtual siempre que aumenta el grado de multiprogramaci ón?
3.3.Indique cuál de estas operaciones no es ejecutada por el **activador:**

a) Restaurar los registros de usuario con los valores almacenados en la tabla del proceso.
h Restaurar el contador de programa.
e) Restaurar el puntero que apunta a la tabla de páginas del proceso.
d) Restaurar la imagen de memoria de un proCeso.

3.4.¿Siempre se produce un cambio de proceso cuando se produce un cambio de contexto? Razone su respuesta.
3.5.¿Cuál es la información que no comparten los pro**CCS05** ligeros de un mismo proceso?

.1

3.6.**¿Puede** producirse un cambio de contexto en un sistema con un planificador basado en el algoritmo primero el trabajo más corto además de cuando se bloquea o se termina el proceso? Razone su respuesta.

3.7.¿Qué algoritmo de planificación será más conveniente para optimizar el rendimiento de la UCPen un sistema que sólo tiene procesos en los cuales no hay entrada/salida?

3.8.¿Cuál de las siguientes políticas de planificación es más adecuada para un sistema de tiempo compartido?

a) Primero el trabajo más corto
b) Round-rohin.
e) Prioridades.
d) ElFO

3.9.¿Cuál es el criterio de planificación más relevante en un sistema de tiempo compartido, el tiempo de respuesta o la optimización en el uso del procesador? Razone su respuesta.

3.10. ¿Cuál de las siguientes transiciones entre los estados de un proceso no se puede producir en un sistema con un algoritmo de planificación no expulsivo?

a) Bloqueado a listo.
h) Ejecutando a listo.
e) Ejecutando a bloqueado
d) Listo a ejecutando.

Sea un sistema que usa un algoritmo de planificación de **~~OCCSOS** *round-rohin* con una rodaja de tiempo de lOO ms. En este sistema ejecutan dos procesos. El primero no realiza operaciones de *EIS* y el segundo solicita una operación de *EIS* cada 50 ms. ¿Cuál será el porcentaje de uso de la UCP?

Considere el siguiente conjunto de procesos planificados con un algoritmo *round-rohin* con 1 u.t. de rodaja. ¿Cuánto tardan en acabar todos ellos?

Proceso Llegada
Pl 2
P2
P3
Duración
8
5
4
3

En un sisteína que usa un algoritmo de planificación de procesos *round-rohin,* ¿cuántos procesos como máximo pueden cambiar de estado cuando se produce una interrupción del disco que indica que se ha terminado una operación sobre el mismo?

Se tienen los siguientes trabajos a ejecutar:

Trabajos Unidades de tiempo Prioridad
2
2 4

3	2	2
4	7	3

Los trabajos llegan en el orden 1, 2, 3 y 4 y la prioridad más alta es la de valor 1. Se pide:

a) Escribir un diagrama que ilustre la ejecución de estos trabajos usando:

1. Planificación de prioridades no expulsiva.
2. Planificación cíclica con una rodaja de tiempo de 2.
3. FIFO.

b) Indicar cuál es el algoritmo de planificación con menor tiempo medio de espera.

3.15. ¿Qué sucede cuando un proceso recibe una señal? *¿,* Y cuando recibe una excepción?
3.16. ¿Cómo se hace en POSIX para que un proceso cree otro proceso que ejecute otro programa? ¿Y en W in3 2'?
3.17. ¿Qué información comparten un proceso y su hijo después de ejecutar el siguiente código'?

```
if (fork() !=0)
wait (&status)
el se
execve (B, parámetros, 0)
```

3.18. En un sistema operativo conforme a la norma POSIX, ¿cuándo pasa un proceso a estado *zombie?*
3.19. Tras la ejecución del siguiente código, ¿cuántos procesos se habrán creado'?

```
for (1=0; 1 < n; i+±)
fork()
```

3.20. Cuando un proceso ejecuta una llamada fork y luego el proceso hijo un exec, ¿qué información comparten ambos procesos?
3.21. ¿Qué diferencia existe entre bloquear una señal e ignorarla en POS IX?
3.22. Escribir un programa en lenguaje C que active unos manejadores para las señales SIGINT, SIGQUIT y SIGILL. Las acciones a ejecutar por dichos manej adores serán:

a) Para SIGINT y SIGQUIT, abortar el proceso con un estado de error.
h) Para SIGILL, imprimir un mensaje de instrucción ilegal y terminar.

3.23. Dado el siguiente programa en lenguaje C:

```
void main(int argc, char argv)
mt 1;
for (1 =1; 1 <= argc; i+±)
fork ()
```

se pide:

a) Dibujar un esquema que muestre la jerarquía de procesos que se crea cuando se ejecuta el programa con argc igual a 3.
h) ¿Cuántos procesos se crean si argc vale *n'?*
3.24. Responder a las siguientes preguntas sobre las llamadas al sistema walt de POSIX y WaitFor-SlngleObject de Win32 cuando ésta se aplica sobre un manejador de proceso.

a) ¿Cuál es la semántica de estas llamadas'?

b) Indicar en qué situaciones puede producirse cambio de proceso y en qué casos no.

c) Describir los pasos que se realizan en estas llamadas al sistema desde que se llama a la rutina de biblioteca hasta que ésta devuelve el control al programa de usuario. Indicar cómo se transfiere el control y parámetros al sistema operativo, cómo realizarÇa su labor el sistema operativo y cómo devuelve el control y resultados al programa de usuario.

3.25. Escribir un programa similar al Programa 3. 15. que utilice los servicios de Win32 para temporizar la ejecución de un proceso.

3.11.

3.12.

(1
P4 3

3.13.

3.14.

8
5

4

Gestión de memoria

La memoria es uno de los recursos más importantes de la computadora y, en consecuencia, la parte del sistema operativo responsable de tratar con este recurso, el gestor de memoria, es un componente básico del mismo. El gestor de memoria del sistema operativo debe hacer de puente entre los requisitos de las aplicaciones y los mecanismos que proporciona el hardware de gestión de memoria. Se trata de una de las partes del sistema operativo que está más ligada al hardware. Esta estrecha colaboración ha hecho que tanto el hardware como el software de gestión de memoria hayan ido evolucionando juntos. Las necesidades del sistema operativo han obligado a los diseñadores del hardware a incluir nuevos mecanismos que, a su vez, han posibilitado el uso de nuevos esquemas de gestión de memoria. De hecho, la frontera entre la labor que realiza el hardware y la que hace el software de gestión de memoria es difusa y ha ido también evolucionando.

Por lo que se refiere a la organización del capítulo, en primer lugar se presentarán los requisitos que debe cumplir la gestión de memoria en un sistema con multiprogramación. A continuación, se mostrarán las distintas fases que conlleva la generación de un ejecutable y se estudiará cómo es el mapa de memoria de un proceso. En las siguientes secciones, se analizará cómo ha sido la evolución de la gestión de la memoria, desde los sistemas multiprogramados más primitivos hasta los sistemas actuales basados en la técnica de memoria virtual. Por último, se presentará el concepto de proyección de archivos y se estudiarán algunos de los servicios POSIX y Win32 de gestión de memoria. El índice del capítulo es el siguiente:

- *Objetivos del sistema de gestión de memoria.*
- *Modelo de memoria de un proceso.*
- *Esquemas de memoria basados en asignación contigua.*
- *Intercambio.*
- *Memoria virtual.*
- *Archivos proyectados en memoria.*
- *Servicios de gestión de memoria.*

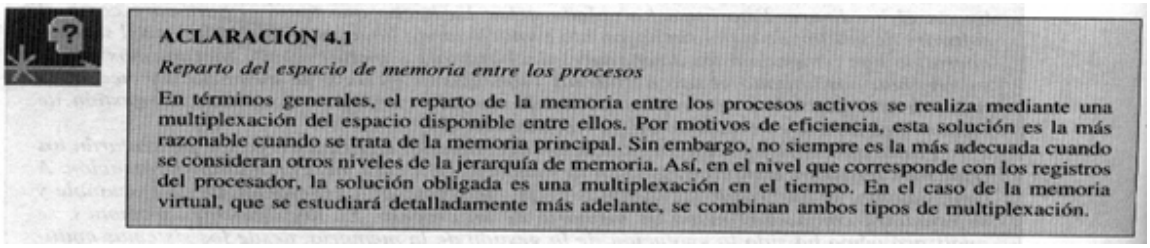
4.1. OBJETIVOS DEL SISTEMA DE GESTIÓN DE MEMORIA

En un sistema con multiprogramación, el sistema operativo debe encargarse de realizar un reparto transparente, eficiente y seguro de los distintos recursos de la máquina entre los diversos procesos, de forma que cada uno de ellos crea que «tiene una máquina para él solo». Esto es, el operativo debe permitir que los programadores desarrollen sus aplicaciones sin verse afectados por la posible coexistencia de su programa con otros durante su ejecución.

Como se ha analizado en capítulos anteriores, en el caso del procesador esta multiplexación se logra almacenando en el bloque de control de cada proceso el contenido de los registros del procesador correspondientes a dicho proceso, salvándolos y restaurándolos durante la ejecución del mismo.

En el caso de la memoria, el sistema operativo, con el apoyo del hardware de gestión memoria del procesador, debe repartir el almacenamiento existente proporcionando un espacio de memoria independiente para cada proceso y evitando la posible interferencia voluntaria o involuntaria de cualquier otro proceso.

Se podría considerar que, en el caso del procesador, se realiza un reparto en el tiempo, mientras que en el de la memoria, se trata de un reparto en el espacio (Aclaración 4.1). La acción combinada de estos dos mecanismos ofrece a los programas una abstracción de procesador virtual que les independiza del resto de los procesos.



ACLARACIÓN 4.1

Reparto del espacio de memoria entre los procesos

En términos generales, el reparto de la memoria entre los procesos activos se realiza mediante una multiplexación del espacio disponible entre ellos. Por motivos de eficiencia, esta solución es la más razonable cuando se trata de la memoria principal. Sin embargo, no siempre es la más adecuada cuando se consideran otros niveles de la jerarquía de memoria. Así, en el nivel que corresponde con los registros del procesador, la solución obligada es una multiplexación en el tiempo. En el caso de la memoria virtual, que se estudiará detalladamente más adelante, se combinan ambos tipos de multiplexación.

Sea cual sea la política de gestión de memoria empleada en un determinado sistema, se pueden destacar las siguientes características como objetivos deseables del sistema de gestión de memoria:

- Ofrecer a cada proceso un espacio lógico propio.
- Proporcionar protección entre los procesos.
- Permitir que los procesos compartan memoria.
- Dar soporte a las distintas regiones del proceso.
- Maximizar el rendimiento del sistema.
- Proporcionar a los procesos mapas de memoria muy grandes.

Espacios lógicos independientes

En un sistema operativo multiprogramado de propósito general no se puede conocer a *priori* la posición de memoria que ocupará un programa cuando se cargue en memoria para proceder a su ejecución, puesto que dependerá del estado de ocupación de la memoria, pudiendo variar, por tanto, en sucesivas ejecuciones del mismo.

El código maquina de un programa contenido en un archivo ejecutable incluirá referencias a memoria, utilizando los diversos modos de direccionamiento del juego de instrucciones del proce

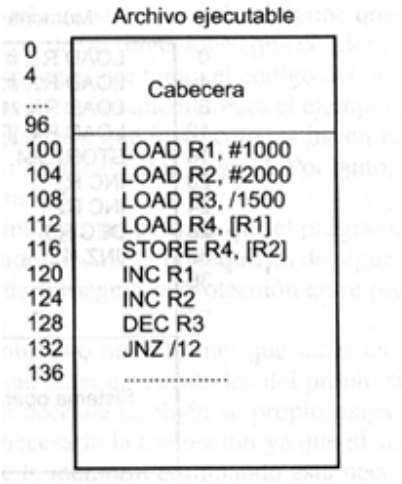


Figura 4.1, *Archivo ejecutable hipotético.*

sador, tanto para acceder a sus operandos como para realizar bifurcaciones en la secuencia de ejecución. Estas referencias típicamente estarán incluidas en un intervalo desde 0 hasta un valor máximo *N*.

Supóngase, por ejemplo, un fragmento de un programa que copia el contenido de un vector almacenado a partir de la dirección 1000 en otro almacenado a partir de la 2000, estando el tamaño del vector almacenado en la dirección 1500. El código almacenado en el archivo ejecutable, mostrado en un lenguaje ensamblador hipotético, seria el mostrado en la Figura 4.1, donde se ha supuesto que la cabecera del ejecutable ocupa 100 bytes y que cada instrucción ocupa 4. Observe que, evidentemente, tanto en el ejecutable como posteriormente en memoria, se almacena realmente código máquina. En esta figura y en las posteriores se ha preferido mostrar un pseudocódigo ensamblador para facilitar la legibilidad de las mismas.

El código maquina de ese programa incluye referencias a direcciones de memoria que corresponden tanto a operandos (las direcciones de los vectores y su tamaño) como a instrucciones (la etiqueta de la bifurcación condicional, JNZ).

En el caso de un sistema con monoprogramación, para ejecutar este programa sólo será necesario cargarlo a partir de la posición de memoria 0 y pasarle el control al mismo. Observe que, como se puede apreciar en la Figura 4.2, se esta suponiendo que el sistema operativo estará cargado en la parte de la memoria con direcciones más altas.

En un sistema con multiprogramación es necesario realizar un proceso de traducción (**reubicación**) de las direcciones de memoria a las que hacen referencia las instrucciones de un programa (**direcciones lógicas**) para que se correspondan con las direcciones de memoria principal asignadas al mismo (**direcciones físicas**). En el ejemplo planteado previamente, si al programa se le asigna una zona de memoria contigua a partir de la dirección 10000, habría que traducir todas las direcciones que genera el programa añadiéndoles esa cantidad.

Este proceso de traducción crea, por tanto, **un espacio lógico** (o **mapa**) independiente para cada proceso proyectándolo sobre la parte correspondiente de la memoria principal de acuerdo con una función de traducción:

Traducción(dirección lógica) → dirección física

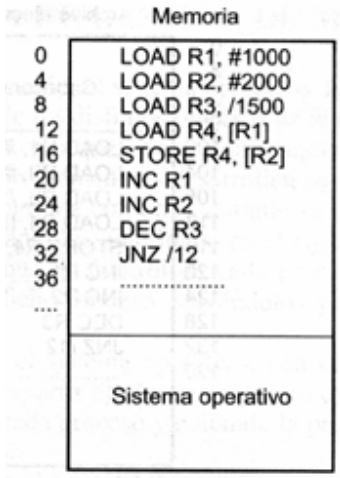


Figura 4.2. *Ejecución del programa en un sistema con monoprogramación.*

Dado que hay que aplicar esta función a cada una de las direcciones que genera el programa, es necesario que sea el procesador el encargado de realizar esta traducción. Como se explicó en el Capitulo 1, la función de traducción la lleva a cabo concretamente un módulo específico del procesador denominado unidad de gestión de memoria (MMU, *Memory Management Unit).* El sistema operativo deberá poder especificar a la MMU del procesador que función de traducción debe aplicar al proceso que está ejecutando en ese momento. En el ejemplo planteado, el procesador, a instancias del sistema operativo, debería sumarle 10000 a cada una de las direcciones que genera el programa en tiempo de ejecución. Observe que el programa se cargaría en memoria desde el ejecutable sin realizar ninguna modificación del mismo y, como se muestra en la Figura 4.3, seria durante su ejecución cuando se traducirían adecuadamente las direcciones generadas. El sistema operativo debería almacenar asociado a cada proceso cuál es la función de traducción que le corresponde al mismo y, en cada cambio de proceso, debería indicar al procesador qué función debe usar para el nuevo proceso activo.

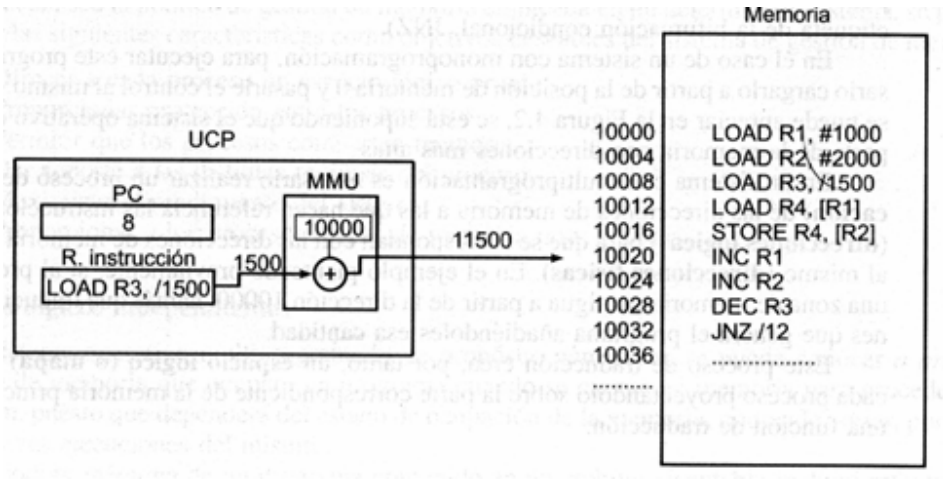


Figura 4.3. *Ejecución en un sistema con reubicación hardware.*

Una alternativa software, usada en sistemas más antiguos que no disponían de hardware de traducción, es realizar la reubicación de las direcciones que contiene el programa en el momento de su carga en memoria. Con esta estrategia, por tanto, el código del programa cargado en memoria ya contiene las direcciones traducidas apropiadamente. Para el ejemplo planteado, durante su carga en memoria, el sistema operativo detectaría que instrucciones hacen referencia a direcciones de memoria y las modificaría añadiendo 10000 al valor original. Por tanto, e1 código cargado en memoria resultaría el mostrado en la Figura 4.4.

Esta segunda solución, sin embargo, no permite que el programa o un fragmento del mismo se pueda mover en tiempo de ejecución, lo cual es un requisito de algunas de las técnicas de gestión de memoria como la memoria virtual, ni asegura la protección entre procesos como se apreciara en el siguiente apartado.

A la hora de analizar este objetivo no solo hay que tener en cuenta las necesidades de los procesos, sino que también hay que tener en cuenta las del propio sistema operativo. Al igual que los procesos, el sistema operativo necesita también su propio mapa de memoria. En este caso, sin embargo, no sería estrictamente necesaria la traducción ya que el sistema operativo puede situarse de forma contigua al principio de la memoria eliminando esta necesidad. De cualquier manera, el uso de una función de traducción puede ser interesante puesto que proporciona mas flexibilidad.

Como gestor de los recursos del sistema, el sistema operativo no sólo requiere utilizar su mapa sino que necesita acceder a toda la memoria del sistema. Es importante resaltar que, además de poder acceder a la memoria física, el sistema operativo necesita «ver» el espacio lógico de cada proceso. Para aclarar esta afirmación, considérese, por ejemplo, un proceso que realiza una llamada al sistema en la que pasa como parámetro la dirección donde esta almacenada una cadena de caracteres que representa el nombre de un archivo. Observe que se tratará de una dirección lógica dentro del mapa del proceso y que, por tanto, el sistema operativo, ya sea con la ayuda de la MMU o «a mano», deberá transformar esa dirección usando la función de traducción asociada al proceso.

Protección

En un sistema con monoprogramación es necesario proteger al sistema operativo de los accesos que realiza el programa en ejecución para evitar que, voluntaria o involuntariamente, pueda interferir en

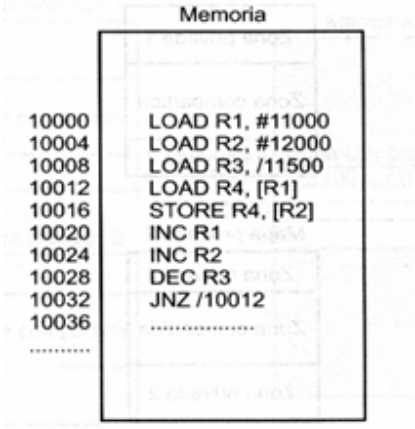


Figura 4.4. *Ejecución en un sistema con reubicación software*

el correcto funcionamiento del mismo. Todos los usuarios que han trabajado en un sistema que no cumple este requisito de protección, como por ejemplo MS-DOS, han experimentado cómo un error de programación en una aplicación puede causar que todo el sistema se colapse durante la ejecución de la misma al producirse una alteración imprevista del código o las estructuras de datos del sistema operativo.

En un sistema con multiprogramación el problema se acentúa ya que no sólo hay que proteger al sistema operativo sino también a los procesos entre sí. El mecanismo de protección en este tipo de sistemas necesita del apoyo del hardware puesto que es necesario validar cada una de las direcciones que genera un programa en tiempo de ejecución. Este mecanismo está típicamente integrado en el mecanismo de traducción: la función de traducción debe asegurar que los espacios lógicos de los procesos sean disjuntos entre sí y con el del propio sistema operativo.

Observe que en el caso de un sistema que use un procesador con mapa de memoria y E/S común, el propio mecanismo de protección integrado en el proceso de traducción permite impedir que los procesos accedan directamente a los dispositivos de E/S, haciendo simplemente que las direcciones de los dispositivos no formen parte del mapa de ningún proceso.

Compartimiento de memoria

Para cumplir el requisito de protección, el sistema operativo debe crear espacios lógicos independientes y disjuntos para los procesos. Sin embargo, en ciertas situaciones, bajo la supervisión y control del sistema operativo, puede ser provechoso que los procesos puedan compartir memoria. Esto es, la posibilidad de que direcciones lógicas de dos o más procesos, posiblemente distintas entre sí, se correspondan con la misma dirección física. Nótese que, como puede observarse en la Figura 4.5*,* la posibilidad de que dos o más procesos compartan una zona de memoria implica que el sistema de gestión de memoria debe permitir que la memoria asignada a un proceso no sea contigua. Así, por ejemplo, una función de traducción como la comentada anteriormente, que únicamente sumaba una cantidad a las direcciones generadas por el programa, obligaría a que el espacio asignado al proceso fuera contiguo, imposibilitando, por tanto, el poder compartir memoria.

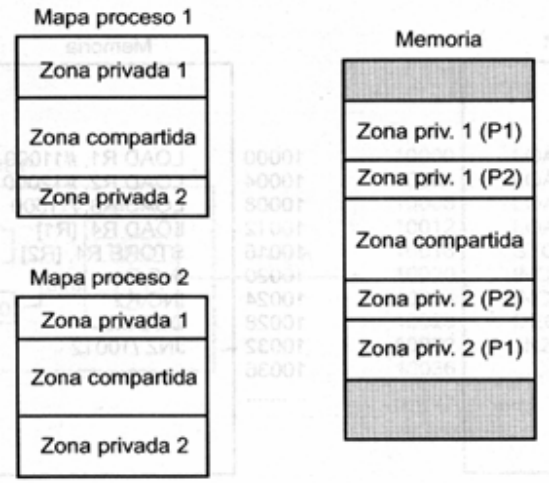


Figura 4.5.. *Dos procesos compartiendo una zona de memoria.*

El compartimiento de memoria puede ser beneficioso en varios aspectos. Por un lado, permite que cuando se estén ejecutando múltiples instancias del mismo programa (p. ej.: el intérprete de los mandatos), los procesos correspondientes compartan el código, lo que resulta

en un mejor aprovechamiento de la memoria disponible. Adicionalmente, como se analizará en el Capitulo *5,* la memoria compartida permite una forma de comunicación muy rápida entre procesos cooperantes.

La posibilidad de que los procesos compartan una zona de memoria haciéndola corresponder con rangos diferentes de su espacio lógico presenta problemas en determinadas circunstancias. Considérese el caso de que una posición de memoria de la zona compartida tenga que contener la dirección de otra posición de dicha zona (o sea, una referencia). Como se puede apreciar en la Figura 4.6, no es posible determinar qué dirección almacenar en la posición origen puesto que cada proceso ve la posición referenciada en una dirección diferente de su mapa de memoria.

Esta situación se puede presentar tanto en zonas compartidas de código como de datos. En el caso del código, considérese que una posición de la zona contiene una instrucción de bifurcación a otra posición de la misma. Por lo que respecta al caso de los datos, supóngase que la zona compartida contiene una estructura de lista basada en punteros.

Soporte de las regiones del proceso

Como se analizará con más detalle en secciones posteriores, el mapa de un proceso no es homogéneo sino que está formado por distintos tipos de regiones con diferentes características y propiedades. Dado que el sistema operativo conoce que regiones incluye el mapa de memoria de cada proceso, el gestor de memoria con el apoyo del hardware debería dar soporte a las características especificas de cada región.

Así, por ejemplo, el contenido de la región que contiene el código del programa no debe poder modificarse. Se debería detectar cualquier intento de escritura sobre una dirección incluida en dicha región y tratarlo adecuadamente (p. ej.: mandando una señal al proceso). Observe que lo que se está detectando en este caso es un error de programación en la aplicación. El dar soporte al carácter de solo lectura de la región permite detectar inmediatamente el error, lo que facilita considerablemente la depuración del programa.

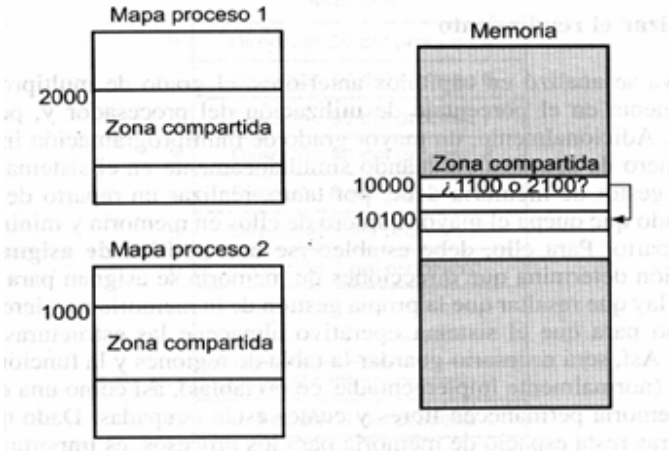


Figura 4.6. Problemas al compartir una zona en diferentes direcciones del mapa de cada proceso.

Otro aspecto a resaltar es que el mapa del proceso no es estático. Durante la ejecución de un programa puede variar el tamaño de una región o, incluso, pueden crearse nuevas regiones o eliminarse regiones existentes. El sistema de memoria debe controlar qué regiones están presentes en el mapa de memoria y cual es el tamaño actual de cada una. Tómese como ejemplo el comportamiento dinámico de la región de pila del proceso. El gestor de memoria debe asignar y liberar memoria para estas regiones según evolucionen la. mismas. Este carácter dinámico del

mapa de un proceso implica la existencia de zonas dentro del mapa de memoria del proceso que en un determinado momento de su ejecución no pertenecen a ninguna región y que, por tanto, cualquier acceso a las mismas implica un error de programación. El gestor de memoria debería detectar cuándo se produce un acceso a las mismas y, como en el caso anterior, tratarlo adecuadamente. Con ello además de facilitar la depuración, se elimina la necesidad de reservar memoria física para zonas del mapa que no estén asignadas al proceso en un determinado instante sistema operativo debe almacenar por cada proceso una **tabla de regiones** que indique las características de las regiones del proceso (tamaño, protección, etc.),

De acuerdo con los aspectos que se acaban de analizar, la función de traducción se generaliza para que tenga en cuenta las situaciones planteadas pudiendo, por tanto, devolver tres valores alternativos:

Traducción(dirección lógica tipo de operación) →
(Dirección física correspondiente,
Excepción por acceso a una dirección no asignada,
Excepción por operación no permitida}

Por último, hay que resaltar que el mapa de memoria del propio sistema operativo, como programa que es, también está organizado en regiones. Al igual que ocurre con los procesos de usuario, el sistema de memoria debería dar soporte a las regiones del sistema operativo. Así, se detectaría inmediatamente un error en el código del sistema operativo, lo que facilitaría su depuración a la gente encargada de esta ardua labor. Observe que 1a detección del error debería detener inmediatamente la ejecución del sistema operativo y mostrar un volcado de la información del sistema.

Maximizar el rendimiento

Como ya se analizó en capítulos anteriores, el grado de multiprogramación del sistema influye directamente en el porcentaje de utilización del procesador y, por tanto, en el rendimiento del sistema. Adicionalmente, un mayor grado de multiprogramación implica que puede existir un mayor numero de usuarios trabajando simultáneamente en el sistema.

El gestor de memoria debe, por tanto, realizar un reparto de la memoria entre los procesos intentando que quepa el mayor numero de ellos en memoria y minimizando el desperdicio inherente al reparto. Para ello, debe establecerse una **política de asignación** adecuada. La política de asignación determina qué direcciones de memoria se asignan para satisfacer una determinada petición. Hay que resaltar que la propia gestión de la memoria requiere un gasto en espacio de almacenamiento para que el sistema operativo almacene las estructura. de datos implicadas en dicha gestión. Así, será necesario guardar la tabla de regiones y la función de traducción asociada a cada proceso (normalmente implementadas como tablas), así como una estructura que refleje qué partes de la memoria permanecen libres y cuales están ocupadas. Dado que el almacenamiento de estas estructuras resta espacio de memoria para los procesos, es importante asegurar que su consumo se mantiene en unos términos razonables.

Así, por ejemplo, la situación óptima de aprovechamiento se obtendría si la función de traducción de direcciones permitiese hacer corresponder cualquier dirección lógica de un proceso con cualquier dirección física, como muestra la Figura 4.7. Observe que esto permitiría que cualquier palabra de memoria libre se pudiera asignar a cualquier proceso. Esta función de traducción es, sin embargo, irrealizable en la práctica puesto que implica tener unas tablas de traducción enormes, dado que habría que almacenar una entrada en la tabla por cada dirección de memoria del mapa de cada proceso.

Como ya se estudio en el Capitulo 1, la solución adoptada en la mayoría de los sistemas operativos actuales para obtener un buen aprovechamiento de la memoria, pero manteniendo las

tablas de traducción dentro de unos valores razonables, es la paginación que se volverá a tratar en las secciones siguientes.

Para optimizar el rendimiento, la mayoría de los sistemas operativos actuales no sólo intentan aprovechar la memoria lo mejor posible, sino que además utilizan la técnica de la memoria virtual presentada en el Capitulo 1, Con esta técnica, el grado de multiprogramación puede aumentar considerablemente, ya que no es preciso que todo el mapa de memoria del proceso tenga que residir en memoria principal para poder ejecutarse, sino que se irá trayendo de la memoria secundaria bajo demanda. Observe que el incremento del grado de multiprogramación no puede ser ilimitado puesto que, como ya se analizó en el Capitulo 1, cuando se supera un determinado umbral, el rendimiento del sistema decae drásticamente.

Como se analizara mas adelante, antes de la aparición de la memoria virtual, los sistemas operativos de tiempo compartido utilizaban una técnica denominada intercambio *(swapping).* Este mecanismo permite que en el sistema existan mas procesos de los que caben en memoria. Sin embargo, a diferencia de la memoria virtual, esta técnica sigue requiriendo que la imagen completa de un proceso resida en memoria para poder ejecutarlo. Los procesos que no caben en memoria en un determinado instante están suspendidos y su imagen reside en un dispositivo de almacenamiento secundario. Por tanto, el grado de multiprogramación sigue estando limitado por el tamaño de la memoria, por lo que el uso de esta técnica no redunda directamente en la mejoría del rendimiento del sistema. Sin embargo, proporciona un mejor soporte multiusuario, que fue el motivo principal para el que se concibió.

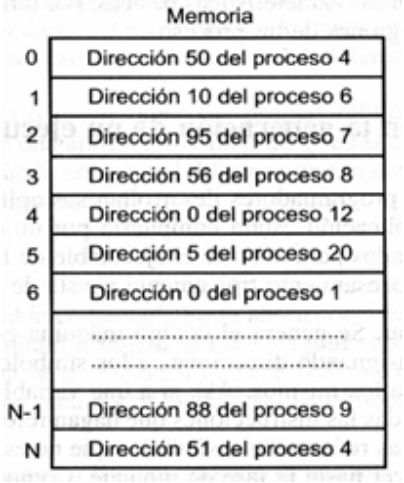


Figura 4.7. *Aprovechamiento óptimo de la memoria.*

Mapas de memoria muy grandes para los procesos

En los tiempos en los que 1a memoria era muy cara y, en consecuencia, los equipos poseían memoria bastante reducida, se producían habitualmente situaciones en las que las aplicaciones se veían limitadas por el tamaño de la memoria, Para solventar este problema, los programadores usaban la técnica de los *overlays.* Esta técnica consiste en dividir el programa en una serie de fases que se ejecutan sucesivamente, pero estando en cada momento residente en memoria sólo una fase. Cada fase se programa de manera que, después de realizar su labor, carga en memoria la siguiente fase y le cede el control. Es evidente que esta técnica no soluciona el problema de forma general dejando en manos del programador todo el trabajo.

Por ello, se ideo la técnica de la memoria virtual, que permite proporcionar a un proceso de

forma transparente un mapa de memoria considerablemente mayor que la memoria física existente en el sistema.

A pesar del espectacular abaratamiento de la memoria y la consiguiente disponibilidad generalizada de equipos con memorias apreciablemente grandes, sigue siendo necesario que el sistema de memoria, usando la técnica de la memoria virtual, proporcione a los procesos espacios lógicos más grandes que la memoria realmente disponible. Observe que a la memoria le ocurre lo mismo que a los armarios: cuanto más grandes son, más cosas metemos dentro de ellos. La disponibilidad de memorias mayores permite a los programadores obtener beneficio del avance tecnológico, pudiendo incluir características más avanzadas en sus aplicaciones o incluso tratar problemas que hasta entonces se consideraban inabordables,

4.2. MODELO DE MEMORIA DE UN PROCESO

El sistema operativo gestiona el mapa de memoria de un proceso durante la vida del mismo. Dado que el mapa inicial de un proceso esta muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo, esta sección comenzara estudiando como se genera un archivo ejecutable y cuál es la estructura típica del mismo. A continuación, se analizará como evoluciona el mapa a partir de ese estado inicial y qué tipos de regiones existen típicamente en el mismo identificando cuáles son sus características básicas. Por último, se expondrán cuáles son las operaciones típicas sobre las regiones de un proceso.

4.2.1. Fases en la generación de un ejecutable

Habitualmente los programadores desarrollan sus aplicaciones utilizando lenguajes de alto nivel. En general, una aplicación estará compuesta por un conjunto de módulos de código fuente que deberán ser procesados para obtener el ejecutable de la aplicación. Como se puede observar en la Figura 4.8, este procesamiento típicamente consta de dos fases:

- **Compilación**. Se genera el código máquina correspondiente a cada módulo fuente de la aplicación asignando direcciones a los símbolos definidos en el módulo y resolviendo las referencias a los mismos. As, si a una variable se le asigna una determinada posición de memoria, todas las instrucciones que hagan referencia a esa variable deben especificar dicha dirección. Las referencias a símbolos que no están definidos en el módulo quedan pendientes de resolver hasta la fase de montaje. Como resultado de esta fase se genera modulo objeto por cada archivo fuente.

Figura 4.8. *Fases en la generación de un ejecutable.*

- **Montaje o enlace.** Se genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos, o sea, haciendo que las referencias a un determinado símbolo apunten a la dirección asignada al mismo. Además de este tipo de referencias, pueden existir referencias a símbolos definidos en otros archivos objeto

previamente compilados agrupados normalmente en bibliotecas. El montador, por tanto, debe generalmente incluir en el ejecutable otros objetos extraídos de las bibliotecas correspondientes. Así, por ejemplo, si la aplicación usa una función matemática como la que calcula el coseno, el montador deberá extraer de la biblioteca matemática el objeto que contenga la definición de dicha función, incluirlo en el ejecutable y resolver 1a referencia a dicha función desde la aplicación de manera que se corresponda con la dirección de memoria adecuada.

Bibliotecas de objetos

Una biblioteca es una colección de objetos normalmente relacionados entre sí. En el sistema existe un conjunto de bibliotecas predefinidas que proporcionan servicios a las aplicaciones. Estos servicios incluyen tanto los correspondientes a un determinado lenguaje de alto nivel (el API del lenguaje) como los que permiten el acceso a los servicios del sistema operativo (el API del sistema operativo).

Asimismo, cualquier usuario puede crear sus propias bibliotecas para de esta forma poder organizar mejor los módulos de una aplicación y facilitar que las aplicaciones compartan módulos. Así, por ejemplo, un usuario puede crear una biblioteca que maneje números complejos y utilizar esta biblioteca en distintas aplicaciones.

Bibliotecas dinámicas

La manera de generar el ejecutable comentada hasta ahora consiste en compilar los módulos fuente de la aplicación y enlazar los módulos objeto resultantes junto con los extraídos de las bibliotecas correspondientes. De esta forma, se podría decir que el ejecutable es autocontenido: incluye todo el código que necesita el programa para poder ejecutarse. Este modo de trabajo presenta varias desventajas:

- El archivo ejecutable puede ser bastante grande ya que incluye, además del código propio de la aplicación, todo el código de las funciones «externas» que usa el programa.

- Todo programa en el sistema que use una determinada función de biblioteca tendrá una copia del código de la misma, Como ejemplo, el código de la función printf, utilizada en casi todos los programas escritos en C, estará almacenado en todos los ejecutables que la usan.
- Cuando se estén ejecutando simultáneamente varias aplicaciones que usan una misma función de biblioteca, existirán en memoria múltiples copias del código de dicha función aumentando el gasto de memoria.
- La actualización de una biblioteca implica tener que volver a generar los ejecutables que la incluyen por muy pequeño que sea el cambio que se ha realizado sobre la misma. Supóngase que se ha detectado un error en el código de una biblioteca o que se ha programado una versión más rápida de una función de una biblioteca. Para poder beneficiarse de este cambio, los ejecutables que la usan deben volver a generarse a partir de los objetos correspondientes. Observe que esta operación no siempre puede realizarse ya que dichos objetos pueden no estar disponibles.

Para resolver estas deficiencias se usan las bibliotecas dinámicamente enlazadas *(Dinamic Link Libraries)* o, simplemente, **bibliotecas dinámicas.** Por contraposición, se denominarán **bibliotecas estáticas** a las presentadas hasta el momento.

Con este nuevo mecanismo, el proceso de montaje de una biblioteca de este tipo se difiere y

en vez de realizarlo en la fase de montaje se realiza en tiempo de ejecución del programa. Cuando en la fase de montaje el montador procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que ésta sea cargada y enlazada en tiempo de ejecución.

Como parte del montaje, se incluye en el ejecutable un módulo de montaje dinámico que encargará de realizar en tiempo de ejecución la carga y el montaje de la biblioteca cuando se haga referencia por primera vez a algún símbolo definido en la misma. En el código ejecutable original del programa, las referencias a los símbolo de la biblioteca, que evidentemente todavía pendientes de resolver, se hacen corresponder con símbolos en el modulo de montaje dinámico. De esta forma, la primera referencia a uno de estos símbolos produce la activación del módulo que realizará en ese momento la carga de la biblioteca y el proceso de montaje necesario. Como parte del mismo, se resolverá la referencia a ese símbolo de manera que apunte al objeto real de la biblioteca y que, por tanto, los posteriores accesos al mismo no afecten al módulo de montaje dinámico. Observe que este proceso de resolución de referencias afecta al programa que hace uso de la biblioteca dinámica ya que implica modificar en tiempo de ejecución alguna de sus instrucciones para que apunten a la dirección real del símbolo. Esto puede entrar en conflicto con el típico carácter de no modificable que tiene la región de código.

El uso de bibliotecas dinámicas resuelve las deficiencias identificadas anteriormente:

- El tamaño de los archivos ejecutables disminuye considerablemente ya que en ellos no se almacena el código de las funciones de bibliotecas dinámicas usadas por el programa.
- Las rutinas de una biblioteca dinámica estarán almacenadas únicamente en archivo de la biblioteca en vez de estar duplicadas en los ejecutables que las usan.
- Cuando se estén ejecutando varios proceso que usan una biblioteca dinámica, éstos podrán compartir el código de la misma, por lo que en memoria habrá solo una copia de la misma.
- La actualización de una biblioteca dinámica es inmediatamente visible a los programas que la usan sin necesidad de volver a montarlos.

Con respecto a este ultimo aspecto, es importante resaltar que cuando la actualización implica un cambio en la interfaz de la biblioteca (p. ej.:una de las funciones tiene un nuevo parámetro), el programa no debería usar esta biblioteca sino la versión antigua de la misma. Para resolver este problema de incompatibilidad, en muchos sistemas se mantiene un número de versión asociado a cada biblioteca. Cuando se produce un cambio en la interfaz, se crea una nueva versión con un nuevo número. Cuando en tiempo de ejecución se procede a la carga de una biblioteca, se busca la versión de la biblioteca que coincida con la que requiere el programa, produciéndose un error en caso de no encontrarla. Observe que durante el montaje se ha almacenado en el ejecutable el nombre y la versión de la biblioteca que utiliza el programa.

Por lo que se refiere al compartimiento del código de una biblioteca dinámica entre los procesos que la utilizan en un determinado instante, es un aspecto que hay que analizar más en detalle. Como se planteó en la sección anterior, si se permite que el código de una biblioteca pueda estar asociado a un rango de direcciones diferente en cada proceso, surgen problemas con las referencias que haya en el código de la biblioteca a símbolos definidos en la misma. Ante esta situación se presentan tres alternativas:

- Establecer un rango de direcciones predeterminado y específico para cada biblioteca dinámica de manera que todos los procesos que la usen la incluirán en dicho rango dentro de su mapa de memoria. Esta solución permite compartir el código de la biblioteca, pero es poco flexible ya que limita el numero de bibliotecas que pueden existir en el sistema y puede causar que el mapa de un proceso sea considerablemente grande presentando amplias zonas sin utilizar.
- Reubicar las referencias presentes en el código de la biblioteca durante la carga de la misma de manera que se ajusten a las direcciones que le han correspondido dentro del mapa de memoria del proceso que la usa. Esta opción permite cargar la biblioteca en cualquier zona libre del mapa del proceso. Sin embargo, impide el poder compartir su código (o, al menos, la totalidad de su código) al estar adaptado a la zona de memoria donde le ha tocado vivir.
- Generar el código de la biblioteca de manera que sea independiente de la posición (en ingles se suelen usar las siglas PIC, *Position Independent Code).* Con esta técnica, el compilador genera código que usa direccionamientos relativos a un registro (p. ej.: al contador de programa) de manera que no se ve afectado por la posición de memoria donde ejecuta. Esta alternativa permite que la biblioteca resida en la zona del mapa que se considere conveniente y que, además, su código sea compartido por los procesos que la usan. El único aspecto negativo es que la ejecución de este tipo de código es un poco menos eficiente que el convencional (Gingell, 1987), pero, generalmente, este pequeño inconveniente queda compensado por el resto de las ventajas.

El único aspecto negativo del uso de las bibliotecas dinámicas es que el tiempo de ejecución del programa puede aumentar ligeramente debido a que con este esquema el montaje de la biblioteca se realiza en tiempo de ejecución. Sin embargo, este aumento es perfectamente tolerable en la mayoría de los casos y queda claramente contrarrestado por el resto de los beneficios que ofrece este mecanismo.

Otro aspecto que conviene resaltar es que el mecanismo de bibliotecas dinámicas es transparente al modo de trabajo habitual de un usuario. Dado que este nuevo mecanismo sólo atañe a la fase de montaje, tanto el código fuente de un programa como el código objeto no se en afectados por el mismo. Además, aunque la fase de montaje haya cambiado considerablemente, los mandatos que se usan en esta etapa son típicamente lo mismos. De esta forma, el usuario no detecta ninguna diferencia entre usar bibliotecas estática. y dinámicas excepto, evidentemente, los beneficios antes comentados.

En la mayoría de los sistemas que ofrecen bibliotecas dinámicas, el sistema tiene una

versión estática y otra dinámica de cada una de las bibliotecas predefinidas. Dados los importantes beneficios que presentan las bibliotecas dinámicas, el montador usará por defecto la versión dinámica, Si el usuario, por alguna razón, quiere usar la versión estática, deberá pedirlo explícitamente.

Montaje explícito de bibliotecas dinámicas

La forma de usar las bibliotecas dinámicas que se acaba de exponer es la mas habitual: se especifica en tiempo de montaje que bibliotecas se deben usar y se pospone la carga y el montaje hasta el tiempo de ejecución. Este esquema se suele denominar **enlace dinámico implícito**. Sin embargo no es la única manera de usar las bibliotecas dinámicas.

En algunas aplicaciones no se conoce en tiempo de montaje qué bibliotecas necesitará programa. Supóngase, por ejemplo, un navegador de Internet que maneja hojas que contienen archivos en distintos formatos y que usa las funciones de varias bibliotecas dinámicas para procesar cada uno de los posibles formatos. En principio, cada vez que se quiera dotar al navegador de la capacidad de manejar un nuevo tipo de formato, sería necesario volver a montarlo especificando también la nueva biblioteca que incluye las funciones para procesarlo. Observe que esto sucede aunque se usen bibliotecas dinámicas.

Lo que se requiere en esta situación es poder decidir en tiempo de ejecución que biblioteca dinámica se necesita y solicitar explícitamente su montaje y carga. El mecanismo de bibliotecas dinámicas presente en Windows NT y en la mayoría de las versiones de UNIX ofrece esta funcionalidad, denominada típicamente **enlace dinámico explícito**. Un programa que pretenda usar funcionalidad deberá hacer uso de los servicios que ofrece el sistema para realizar esta solicitud explícita. Observe que, en este caso, no habría que especificar en tiempo de montaje el nombre de la biblioteca dinámica. Pero, como contrapartida, el mecanismo de carga y montaje de la biblioteca dinámica deja de ser transparente a la aplicación.

Formato del ejecutable

Como parte final del proceso de compilación y montaje, se genera un archivo ejecutable que contiene el código máquina del programa. Distintos fabricantes han usado diferentes formatos para este tipo de archivos. En el mundo UNIX, por ejemplo, uno de los formatos más utilizados actualmente denominado *Executable and Linkable Format* (ELF). A continuación, se presentará de simplificada cómo es el formato típico de un ejecutable.

Como se puede observar en la Figura 4.9, un ejecutable está estructurado como una cabecera y un conjunto de secciones.

La cabecera contiene información de control que permite interpretar el contenido del ejecutable. En la cabecera típicamente se incluye, entre otras cosas, la siguiente información:

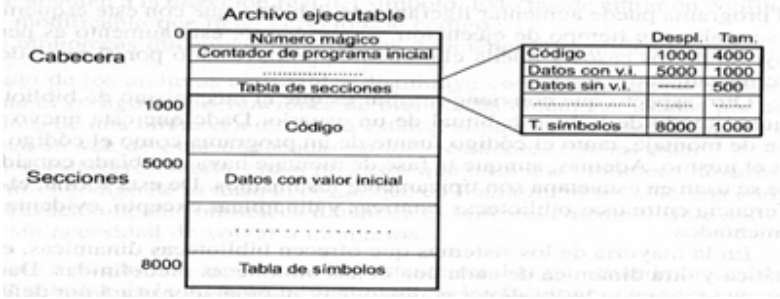


Figura 4.9. Formato simplificado de un ejecutable.

- Un «número mágico» que identifica al ejecutable. Así, por ejemplo, en el formato ELF, el primer byte del archivo ejecutable debe contener el valor hexadecimal *7* f y los tres siguientes los caracteres 'E', 'L' y 'F'.

- La dirección del punto de entrada del programa, esto es, la dirección que se almacenará inicialmente en el contador de programa del proceso.
- Una tabla que describe las secciones que aparecen en el ejecutable. Por cada una de ellas, se especifica, entre otras cosas, su tipo, la dirección del archivo donde comienza y su tamaño.

En cuanto a las secciones, Cada ejecutable tiene un conjunto de secciones. El contenido de estas secciones es muy variado. Una sección puede contener información para facilitar la depuración (p. ej.: una tabla de símbolos). Otra sección puede contener la lista de bibliotecas dinámicas que requiere el programa durante su ejecución. Sin embargo, esta exposición se centrará en las tres que más influyen en el mapa de memoria de un proceso:

- Código (texto). Contiene el código del programa.
- Datos con valor inicial. Almacena el valor inicial de todas las variables globales a las que se les ha asignado un valor inicial en el programa.
- Datos sin valor inicial. Se corresponde con todas las variables globales a las que no se les ha dado un valor inicial. Como muestra la figura, esta sección aparece descrita en la tabla de secciones de la cabecera, pero, sin embargo, no se almacena normalmente en el ejecutable ya que el contenido de la misma es irrelevante. En la tabla únicamente se especifica su tamaño.

Observe que en el ejecutable no aparece ninguna sección que corresponda con las variables locales y parámetros de funciones. Esto se debe a que este tipo de variables presentan unas características muy diferentes a las de Las variables globales, a saber:

- Las variables globales tienen un carácter estático. Existen durante toda la vida del programa. Tienen asociada una dirección fija en el mapa de memoria del proceso y, por tanto, también en el archivo ejecutable puesto que éste se corresponde con la imagen inicial del proceso. En el caso de que la variable no tenga un valor inicial asignado en el programa, no es necesario almacenarla explícitamente en el ejecutable, sino que basta con conocer el tamaño de la sección que contiene este tipo de variables.
- Las variables locales y parámetros tienen carácter dinámico. Se crean cuando se invoca la función correspondiente y se destruyen cuando termina la llamada. Por tanto, estas variables no tienen asignado espacio en el mapa inicial del proceso ni en el ejecutable. Se crean dinámicamente usando para ello la pila del proceso. La dirección que corresponde a una variable de este tipo se determina en tiempo de ejecución ya que depende de la secuencia de llamadas que genere el programa Observe que, dada la posibilidad de realizar llamadas recursivas directas o indirectas que proporciona la mayoría de los lenguajes, pueden existir en tiempo de ejecución varias instancias de una variable de este tipo.

En el Programa 4.1 se muestra el esqueleto de un programa en C que contiene diversos tipos de variables. En él se puede observar que las variables **x** e ***y*** son globales, mientras que **z** es local y es **t** un parámetro de una función. La primera de ellas tiene asignado un valor inicial, por lo que dicho valor estará almacenado en la sección del ejecutable que corresponde a este tipo de variables. Con respecto a la variable **y,** no es necesario almacenar ningún valor en el ejecutable

asociado a la misma, pero su existencia quedará reflejada en el tamaño total de la sección de variables globales sin valor inicial. En cuanto a la variable local **z** y al parámetro **t**, no están asociados a ninguna sección del ejecutable y se les asignará espacio en tiempo de ejecución cuando se produzcan llamadas a la función donde están definidos.

Programa 4.1. Esqueleto de un programa.

```
int x=8;
int y;
f(int t){
     int Z;
    .............
}
main( ) {
 .............
}
```

4.2.2. Mapa de memoria de un proceso

Como se comentó previamente, el mapa de memoria de un proceso no es algo homogéneo sino que está formado por distintas regiones o segmentos. Una región tiene asociada una determinada información. En este capítulo se denominará **objeto de memoria** a este conjunto de información relacionada. La asociación de una región de un proceso con un objeto de memoria permite al proceso tener acceso a la información contenida en el objeto.

Cuando se activa la ejecución de un programa (servicio exec en POSIX y CreateProcess en Win32), se crean varias regiones dentro del mapa a partir de la información del ejecutable. Cada sección del ejecutable constituye un objeto de memoria. Las regiones iniciales del proceso se van a corresponder básicamente con las distintas secciones del ejecutable.

Cada región es una zona contigua que está caracterizada por la dirección dentro del mapa de proceso donde comienza y por su tamaño. Además, tendrá asociadas una serie de propiedades características especificas como las siguientes:

- Soporte de la región. El objeto de memoria asociado a la región. En él está almacenado el contenido inicial de la región. Se presentan normalmente dos posibilidades:

— *Soporte en archivo.* El objeto está almacenado en un archivo o en parte del mismo.
— *Sin soporte.* El objeto no tiene un contenido inicial. En algunos entornos se les denomina objetos anónimos.

- Tipo de uso compartido:

— *Privada.* El contenido de la región sólo es accesible al proceso que la contiene. Las modificaciones sobre la región no se reflejan en el objeto de memoria.
— *Compartida.* El contenido de la región puede ser compartido por varios procesos. Las modificaciones en el contenido de la región se reflejan en el objeto de memoria.

- Protección. Tipo de acceso permitido. Típicamente se distinguen tres tipos:
  — *Lectura.* Se permiten accesos de lectura de operandos de instrucciones.
  — *Ejecución.* Se permiten accesos de lectura de instrucciones *(fetch).*
  — *Escritura.* Se permiten accesos de escritura.
- Tamaño fijo o variable. En el caso de regiones de tamaño variable, se suele distinguir si la región crece hacia direcciones de memoria menores o mayores.

Como se puede observar en la Figura 4.10, las regiones que presenta el mapa de memoria inicial del proceso se corresponden básicamente con las secciones del ejecutable más la pila inicial del proceso, a saber:

- Código (o texto). Se trata de una región compartida de lectura/ejecución. Es de tamaño fijo (el indicado en la cabecera del ejecutable). El soporte de esta región está en la sección correspondiente del ejecutable.
- Datos con valor inicial. Se trata de una región privada ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo. Es de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). El soporte de esta región está en la sección correspondiente del ejecutable.
- Datos sin valor inicial. Se trata de una región privada de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). Como se comento previamente, esta región no tiene soporte en el ejecutable ya que su contenido inicial es irrelevante. En muchos sistemas se le da un valor inicial de cero a toda la región por motivos de confidencialidad.
- Pila, Esta región es privada y de lectura/escritura. Servirá de soporte para almacenar los registros de activación de las llamadas a funciones (las variables locales, parámetros, dirección de retorno, etc.). Se trata, por tanto, de una región de tamaño variable que crecerá cuando se produzcan llamadas a funciones y decrecerá cuando se retorne de las mismas. Típicamente, esta región crece hacia las direcciones más bajas del mapa de memoria. De manera similar a la región de datos sin valor inicial por razones de confidencialidad de la información, cuando crece la pila se rellena a cero la zona expandida. En el mapa inicial existe ya esta región que contiene típicamente los argumentos especificados en la invocación del programa (en el caso de POSIX, estos argumentos son los especificados en el segundo parámetro de la función execvp).

Los sistemas operativos modernos ofrecen un modelo de memoria dinámico en el que el mapa de un proceso está formado por un número variable de regiones que pueden añadirse o eliminarse durante la ejecución del mismo. Además de las regiones iniciales ya analizadas, durante la ejecución del proceso pueden crearse nuevas regiones relacionadas con otros aspectos tales como los siguientes:

.

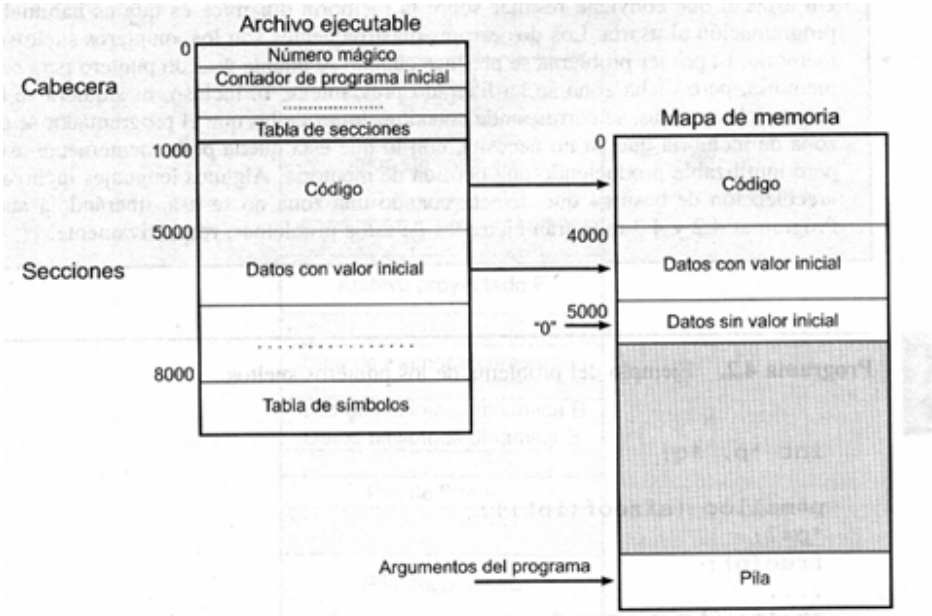


Figura 4.10. *Mapa de memoria inicial a partir del ejecutable*

• *Heap.* La mayoría de los lenguajes de alto nivel ofrecen la posibilidad de reservar espacio en tiempo de ejecución. En el caso del lenguaje C se usa la función malloc para ello. Esta región sirve de soporte para la memoria dinámica que reserva un programa en tiempo de ejecución. Comienza, típicamente, justo después de la región de datos sin valor inicial (de hecho, en algunos sistemas se considera parte de la misma) y crece en sentido contrario a la pila (hacia direcciones crecientes), Se trata de una región privada de lectura/escritura, sin soporte (se rellena inicialmente a cero), que crece según el programa vaya reservando memoria dinámica y decrece según la vaya liberando. Normalmente, cada programa tendrá un único *heap.* Sin embargo, algunos sistemas, como Win32, permiten crear múltiples *heaps.*

En la Aclaración 4.2 se intenta clarificar algunos aspectos de la memoria dinámica.

• Archivos proyectados. Cuando se proyecta un archivo, se crea una región asociada al mismo. Este mecanismo será realizado con más profundidad al final del capítulo, pero, a *priori,* se puede resaltar que se trata de una región compartida cuyo soporte es el archivo que se proyecta.

• Memoria compartida. Cuando se crea una zona de memoria compartida y se proyecta, como se analizará detalladamente en el Capitulo 4, se origina una región asociada a la misma. Se

trata, evidentemente, de una región de carácter compartido, cuya protección la especifica el programa a la hora de proyectarla.

• Pilas de *threads.* Como se estudió en el Capítulo 3, cada *thread* necesita una pila propia que normalmente corresponde con una nueva región en el mapa. Este tipo de región tiene las mismas características que la región correspondiente a la pila del proceso.

ACLARACIÓN 4.2

El sistema operativo sólo realiza una gestión básica de la memoria dinámica. Generalmente, las aplicaciones no usan directamente estos servicios. La biblioteca de cada lenguaje de programación utiliza estos servicios básicos para construir a partir de ellos unos más avanzados orientados a las aplicaciones. Un aspecto que conviene resaltar sobre la memoria dinámica es que es habitual cometer errores de programación al usarla. Los dos errores más frecuentes son los «punteros sueltos» y las «goteras» en memoria. El primer problema se produce cuando se intenta usar un puntero para acceder a una zona de memoria, pero dicha zona se ha liberado previamente o, incluso, ni siquiera se había reservado. En cuanto a las goteras, se corresponde con situaciones en las que el programador se olvida de liberar una zona de memoria que ya no necesita, con lo que ésta queda permanentemente asignada al programa, pero inutilizable produciendo una pérdida de memoria. Algunos lenguajes incluyen un mecanismo de «recolección de basura» que detecta cuando una zona no se usa, liberándola automáticamente. Los Programas 4.2 y 4.3 muestran ejemplos de estos problemas, respectivamente.

Programa 4.2. Ejemplo del problema de los punteros suelto..

```
int *p, *q ;

p=malloc (sizeof (int)) ;
*p=7;
free(p);
.....
*q=*p;  /* p y q son dos punteros sueltos */
```

Programa 4.3. Ejemplo del problema de las goteras.

```
int *p, *q;

p=malloc (sizeof(int) );
q=malloc (sizeof(int) );
p=q;
/* la primera zona reservada queda inutilizable */
```

En la Figura 4.11 se muestra un hipotético mapa de memoria que contiene algunos de los tipos de regiones comentadas en esta sección.

Como puede apreciarse en la figura, la carga de una biblioteca dinámica implicará la creación de un conjunto de regiones asociadas a la misma que contendrán las distintas secciones de la biblioteca (código y datos globales).

Hay que resaltar que, dado el carácter dinámico del mapa de memoria de un proceso (se crean y destruyen regiones, algunas regiones cambian de tamaño, etc.), existirán, en un determinado instante, zonas sin asignar (**huecos**) dentro del mapa de memoria del proceso. Cualquier acceso a estos huecos representa un error y debería ser detectado y tratado por el sistema operativo.

Por último, hay que recalcar que, dado que el sistema operativo es un programa, su mapa de memoria contendrá también regiones de código, datos y *heap* (el sistema operativo también usa memoria dinámica).

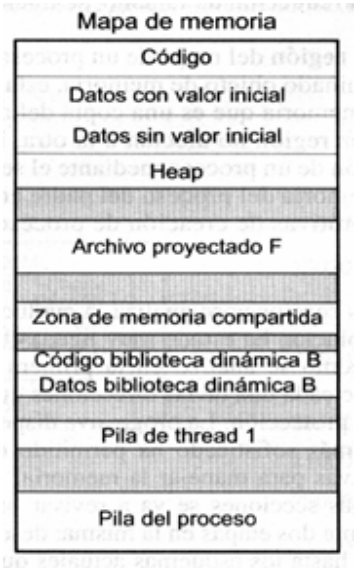


Figura 4.11. *Mapa de memoria de un proceso hipotético.*

4.2.3. Operaciones sobre regiones

Durante la vida de un proceso, su mapa de memoria va evolucionando y con el las regiones incluidas en el mismo. En aras de facilitar el estudio que se realizará en el resto de este capítulo, sección se identifican que operaciones se pueden realizar sobre una región dentro del mapa proceso. En esta exposición se plantean las siguientes operaciones genéricas:

- **Crear una región** dentro del mapa de un proceso asociándola un objeto de sistema operativo crea una nueva región vinculada al objeto en el lugar correspondiente del mapa asignándola los recursos necesarios y estableciendo las características y propiedades de la misma (tipo de soporte, carácter privado o compartido, tipo de protección y tamaño fijo o variable). En la creación del mapa de un proceso asociado a un determinado programa se realiza esta operación por cada una de las regiones iniciales (código, datos con valor, datos sin valor inicial y pila). Durante la ejecución del programa, también se lleva a diferentes situaciones como, por ejemplo, cuando se carga una biblioteca dinámica o se proyecta un archivo.
- **Eliminar una región del mapa de un proceso.** Esta operación libera todos los recursos vinculados a la región que se elimina. Cuando un proceso termina, voluntaria o involuntariamente, se liberan implícitamente todas sus regiones. En el caso de una región que corresponda con una zona de memoria compartida o un archivo proyectado el proceso puede solicitar explícitamente su eliminación del mapa.
- **Cambiar el tamaño de una región.** El tamaño de la región puede cambiar ya sea por una petición explícita del programa, como ocurre con la región del *heap*, o de forma implícita, como sucede cuando se produce una expansión de la pila. En el caso de un aumento de tamaño, el sistema asignará los recursos necesarios a la región comprobando previamente que la expansión no provoca un solapamiento, en cuyo caso no se llevaría a cabo. Cuando se trata de una reducción de tamaño, se liberan los recursos vinculados al fragmento liberado.
- **Duplicar una región** del mapa de un proceso en el mapa de otro. Dada una región

asociada a un determinado objeto de memoria, esta operación crea una nueva región asociada a un objeto de memoria que es una copia del anterior. Por tanto, las

modificaciones que se realizan en una región no afectan a la otra. Esta operación serviría de base para lleva a cabo la creación de un proceso mediante el servicio fork de POSIX, que requiere duplicar el mapa de memoria del proceso del padre en el proceso hijo. En sistemas operativos que no tengan primitivas de creación de procesos de este estilo no se requeriría esta operación.

En las siguientes secciones se analiza la evolución de la gestión de memoria en los sistemas operativos. Dicha evolución ha estado muy ligada con la del hardware de gestión de memoria del procesador ya que, como se analizo en la primera sección del capítulo, se necesita que sea el procesador el que trate cada una de las direcciones que genera un programa para cumplir los requisitos de reubicación y protección. La progresiva disponibilidad de procesadores con un hardware de gestión de memoria más sofisticado ha permitido que el sistema operativo pueda implementar estrategias más efectivas para manejar la memoria.

En estas próximas secciones se va a revisar brevemente esta evolución, distinguiendo por simplicidad, únicamente dos etapas en la misma: desde los esquemas que realizaban una asignación contigua de memoria hasta los esquemas actuales que están basados en la memoria virtual. Como etapa intermedia entre estos dos esquemas, se presentará la técnica del intercambio.

4.3. ESQUEMAS DE MEMORIA BASADOS EN ASIGNACIÓN CONTIGUA

Un esquema simple de gestión de memoria consiste en asignar a cada proceso una zona contigua de memoria para que en ella resida su mapa de memoria. Dentro de esta estrategia general hay diversas posibilidades. Sin embargo, dado el alcance y objetivo de este tema, la exposición se centrará sólo en uno de estos posibles esquemas: la gestión contigua basada en particiones dinámicas. Se trata de un esquema que se usó en el sistema operativo OS/MVT de IBM en la década de los sesenta.

Con esta estrategia, cada vez que se crea un proceso, el sistema operativo busca un hueco en memoria de tamaño suficiente para alojar el mapa de memoria del mismo. El sistema operativo reservara la parte del hueco necesaria, creara en ella el mapa inicial del proceso y establecerá una función de traducción tal que las direcciones que genera el programa se correspondan con la zona asignada.

En primer lugar, se presentará qué hardware de gestión de memoria se requiere para realizar este esquema para, a continuación, describir cuál es la labor del sistema operativo.

Hardware

Como ya se estudió en el Capítulo 1, este esquema requiere un hardware de memoria relativamente simple. Típicamente el procesador tendrá dos registros *valla,* únicamente accesibles en modo privilegiado, que utilizará para tratar cada una de las direcciones que genera un programa. Como se puede observar en la Figura 4.12, la función de estos registros será la siguiente:

- **Registro límite**. El procesador comprueba que cada dirección que genera el proceso no es mayor que el valor almacenado en este registro. En caso de que lo sea, se generará una excepción.

- **Registro base.** Una vez comprobado que la dirección no rebasa el limite permitido, el procesador le sumará el valor de este registro, obteniéndose con ello la dirección de memoria física resultante.

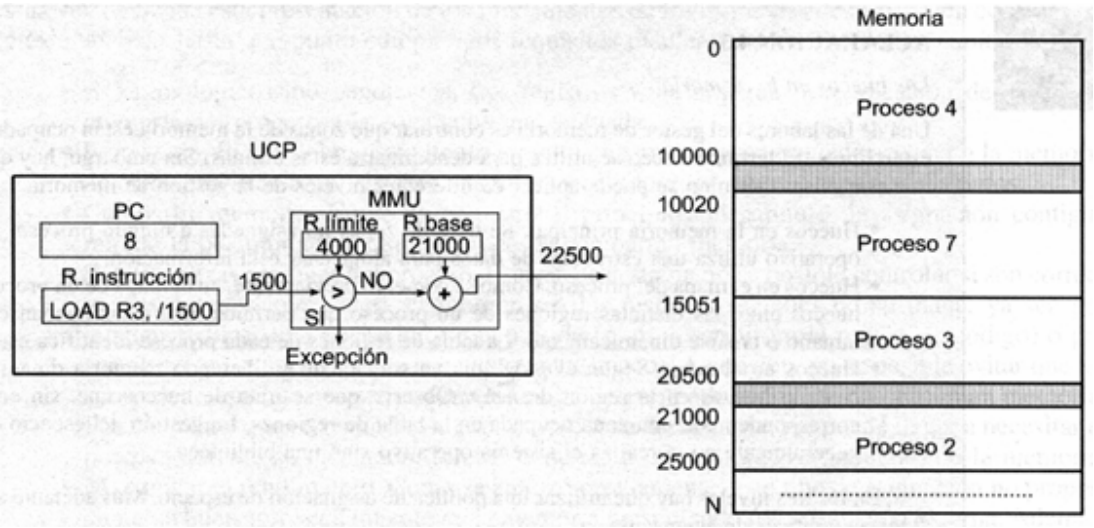


Figura 4.12. *Estado de la MMU durante la ejecución del proceso* 2.

Observe que los registros valla estarán desactivados cuando el procesador está en modo privilegiado. De esta forma, el sistema operativo podrá acceder a toda la memoria del sistema. Sin embargo, a veces el sistema operativo necesitará traducir una dirección lógica de un proceso que recibe como argumento de una llamada al sistema. En ese caso, el mismo deberá aplicar a esa dirección los registros valla.

Gestión del sistema operativo

El sistema operativo únicamente tendrá que almacenar en el bloque de control de cada proceso cuales son los valores que deben tener estos dos registros para dicho proceso. Observe que este par de valores son los que definen la función de traducción que corresponde al proceso y que, evidente- mente, su almacenamiento apenas consume espacio. Cuando se produzca un cambio de proceso, el sistema operativo deberá cargar los registros valla del procesador con los valores correspondiente al proceso que se va a activar.

El sistema operativo mantiene información sobre el estado de la memoria usando una estructura de datos que identifica qué partes de la memoria están libres. Típicamente usa una almacena la dirección inicial y el tamaño de cada zona libre o hueco (véase la Aclaración 4 analiza los distintos usos de este término en este entorno). Observe que en la gestión de esta lista, cada vez que una zona pasa de ocupada a libre hay que comprobar si el nuevo espacio libre se puede fundir con posibles zonas libres contiguas. Además, deberá mantener una tabla de regiones para identificar qué partes de la memoria otorgada al proceso están asignadas a regiones y cuales están libres.

Con esta estrategia, como muestra la figura anterior, según se van ejecutando distintos procesos, van quedando «fragmentos» en la memoria que, dado su pequeño tamaño, no podrían ser asignados a ningún proceso. A este problema se le denomina **fragmentación externa** y conlleva una mala utilización de la memoria por la progresiva fragmentación del espacio de almacenamiento. Una posible solución a este problema sería compactar la memoria moviendo los mapas de memoria de los procesos para que

queden contiguos, dejando así el espacio libre también contiguo Esta transferencia implicaría también el reajuste de los registros valla de acuerdo a la nueva posición de cada proceso. Sin embargo, esta compactación es un proceso bastante ineficiente.

ACLARACIÓN 4.3

Los huecos en la memoria

Una de las labores del gestor de memoria es controlar qué zonas de la memoria están ocupadas y cuáles están libres. El término hueco se utiliza para denominar a estas últimas. Sin embargo, hay que resaltar que este mismo término se puede aplicar en diferentes niveles de la gestión de memoria.

- Huecos en la memoria principal. Se trata de zonas no asignadas a ningún proceso. El sistema operativo utiliza una estructura de datos para almacenar esta información.
- Huecos en el mapa del proceso. Como se comentó previamente, en el mapa de un proceso existen huecos entre las distintas regiones de un proceso que permiten que éstas puedan cambiar de tamaño o crearse dinámicamente. La tabla de regiones de cada proceso identifica estos huecos.
- Huecos en el *heap.* Según el programa va solicitando y liberando memoria dinámica se van creando huecos en la región de *heap.* Observe que se trata de huecos que, sin embargo, se corresponden con una zona ocupada en la tabla de regiones. La gestión del espacio en el *heap* generalmente no la realiza el sistema operativo sino una biblioteca.

En los tres niveles hay que utilizar una política de asignación de espacio. Más adelante se analizan diversas políticas de asignación.

Política de asignación de espacio

El sistema operativo debe llevar a cabo una política de asignación de espacio. Cuando se precisa crear el mapa de memoria de un proceso que ocupa un determinado tamaño, esta política decide cuál de las zonas libres se debería usar, intentando conjugar dos aspectos: un buen aprovechamiento de la memoria y un algoritmo de decisión eficiente.

Realmente, el problema de la asignación dinámica de espacio es un problema más general que se presenta también fuera del ámbito de la informática. Se trata, por tanto, de un problema ampliamente estudiado. Típicamente, existen tres estrategias básicas:

- El mejor ajuste *(best-fit).* Se elige la zona libre más pequeña donde quepa el mapa del proceso. *A priori,* puede parecer la mejor solución. Sin embargo, esto no es así. Por un lado, se generan nuevos espacios libres muy pequeños. Por otro lado, la selección del mejor hueco exige comprobar cada uno de ellos o mantenerlos ordenados por tamaño. Ambas soluciones conducen a un algoritmo ineficiente,
- El peor ajuste *(worst-fit).* Se elige el hueco más grande, Con ello se pretende que no se generen nuevos huecos pequeños. Sin embargo, sigue siendo necesario recorrer toda la lista de huecos o mantenerla ordenada por tamaño.
- El primero que ajuste *(first-fit).* Aunque pueda parecer sorprendente a *priori,* ésta suele ser la mejor política. Es muy eficiente ya que basta con encontrar una zona libre de tamaño suficiente y proporciona un aprovechamiento de la memoria aceptable.

Una estrategia de asignación interesante es el sistema *buddy* [Knowlton, *1965].* Está basado en el uso de listas de huecos cuyo tamaño son potencias de dos, La gestión interna de la memoria del propio sistema operativo en UNIX 4,3 BSD y en Linux utiliza variantes de este algoritmo. Dada su relativa complejidad, se remite al lector a la referencia antes citada.

Es interesante resaltar que estas estrategias de asignación también se utilizan en la gestión del *heap,* tanto en el de los procesos como en el del propio sistema operativo.

Valoración del esquema contiguo

Una vez realizada esta presentación de los fundamentos de los esquemas de asignación

contigua, se puede analizar hasta qué punto cumplen los requisitos planteados al principio del capítulo:

- Espacios lógicos independientes. Los registros valla realizan la reubicación del mapa de memoria del proceso a la zona contigua asignada.
- Protección. El uso del registro límite asegura que un proceso no pueda acceder a la

memoria de otros procesos o del sistema operativo, creándose, por tanto, espacios disjuntos.

- Compartir memoria. Como se comentó al principio del capítulo, la asignación contigua impide la posibilidad de que los procesos compartan memoria.
- Soporte de las regiones del proceso. Con esta estrategia no es posible controlar si son correctos los accesos que genera el programa a las distintas regiones de su mapa, ya sea por intentar realizar una operación no permitida (p. ej.: escribir en la región de código) o por acceder a una zona no asignada actualmente (hueco). Así mismo, no es posible evitar que las zonas del mapa que no estén asignadas en un determinado momento no consuman memoria, Por tanto, hay que reservar desde el principio toda la memoria que puede llegar a necesitar el proceso durante su ejecución, lo cual conduce a un mal aprovechamiento de la memoria,
- Maximizar el rendimiento, Como se analizó previamente, este tipo de asignación no proporciona un buen aprovechamiento de la memoria presentando fragmentación externa. Además, no sirve de base para construir un esquema de memoria virtual.
- Mapas de memoria grandes para los procesos. Este esquema no posibilita el uso de mapa grandes, ya que quedan limitados por el tamaño de la memoria física.

Operaciones sobre regiones

Las limitaciones del hardware condicionan como se gestionan las regiones con este esquema. No se puede compartir memoria ni se pueden controlar los accesos erróneos.

Cuando se crea un proceso se le otorga una zona de memoria de tamaño fijo que quedará asignada al proceso hasta que este termine. El tamaño de esta región deberá ser suficiente para albergar las regiones iniciales del proceso y un hueco que permita que las regiones puedan crecer, que se incluyan nuevas regiones (p. ej.: bibliotecas dinámicas). Dado que el tamaño de la zona asignada al proceso no va a cambiar durante su ejecución, es importante establecer un tamaño adecuado para el hueco. Si es demasiado pequeño, puede agotarse el espacio durante la ejecución del proceso debido al crecimiento de una región o a la inclusión de una nueva región. Si es demasiado grande, se produce un desperdicio ya que la memoria asociada al hueco no podrá aprovecharse mientras durante la ejecución del programa.

Cuando se crea una región, ya sea en el mapa inicial o posteriormente, se le da una parte de espacio asignado al proceso actualizando la tabla de regiones adecuadamente. En este espacios carga el contenido inicial de la región (de un archivo o con ceros).

Cuando se elimina una región se libera su espacio quedando disponible para otras regiones. Al terminar el proceso se liberan todas sus regiones liberando todo el espacio del mapa para que se pueda usar para crear otros procesos.

El cambio de tamaño de una región implica la asignación o liberación de la zona afectada. En el caso de un aumento, antes se comprueba que hay espacio libre contiguo a la región sin producirse un solapamiento con otra región. Esta comprobación e puede realizar cuando la región crece por una solicitud del programa (como ocurre con el *heap).* Sin embargo, no se puede 11evar a cabo en caso de una expansión de la pila, ya que esta se realiza sin intervención del sistema operativo, por 1o que la pila puede desbordarse y afectar a otra región.

Por último, la operación de duplicar una región implica crear una región nueva y copiar el contenido de la región original. Este esquema no permite optimizar esta operación necesaria para implementar el servicio fork de POSIX.

4.4. INTERCAMBIO

Como se comentó previamente, la técnica del intercambio *(swapping)* significó en su momento una manera de permitir que en los sistemas del tiempo compartido existieran más procesos de los que caben en memoria. Se puede considerar que se trata de un mecanismo antecesor de la virtual. En esta sección se presentarán de forma breve los fundamentos de esta técnica.

El intercambio se basa en usar un disco o parte de un disco (dispositivo de s*wap)* como respaldo de la memoria principal . Cuando no caben en memoria todos los procesos activos (p. ej. debido a que se ha creado uno nuevo), se elige un proceso residente y se copia en *swap* su memoria. El criterio de selección puede tener en cuenta aspectos tales como la prioridad del proceso, el tamaño de su mapa de memoria, el tiempo que lleva ejecutando y, principalmente. Se debe intentar expulsar *(swap out)* procesos que estén bloqueados. Cuando se expulsa un proceso no es necesario copiar toda su imagen al *swap.* Los huecos en el mapa no es preciso copiarlos ya que su contenido es intrascendente. Tampoco se tiene que copiar el código, ya que se puede volver a recuperar directamente del ejecutable.

Evidentemente, un proceso expulsado tarde o temprano debe volver a activarse y cargarse en memoria principal *(swap in).* Sólo se deberían volver a cargar aquellos procesos que estén listos para ejecutar. Esta readmisión en memoria se activará cuando haya espacio de memoria disponible (p. ej.: debido a que se ha terminado un proceso) o cuando el proceso lleve un cierto tiempo expulsado. Observe que al tratarse de un sistema de tiempo compartido, se debe repartir el procesador entre todos los procesos. Por ello, en numerosa ocasiones hay que expulsar un proceso para poder traer de nuevo a memoria a otro proceso que lleva expulsado un tiempo suficiente.

En cuanto al dispositivo de *swap,* hay dos alternativas en la asignación de espacio:

- **Preasignación.** Al crear el proceso ya se reserva espacio de *swap* suficiente para albergarlo.
- **Sin preasignación**. Sólo se reserva espacio de s*wap* cuando se expulsa el proceso.

Un último aspecto a tener en cuenta es que no debería expulsarse un proceso mientras se estén realizando operaciones de entrada/salida por DMA vinculadas a su imagen de memoria ya que esto causaría que el dispositivo accediera al mapa de otro proceso.

4.5. MEMORIA VIRTUAL

En prácticamente todos los sistemas operativos modernos se usa la técnica de memoria virtual. En esta sección se analizarán los conceptos básicos de esta técnica. En el Capítulo 1 ya se presentaron los fundamentos de la memoria virtual, por lo que no se volverá a incidir en los aspectos estudiados en el mismo.

Como se estudió en dicho capítulo, la memoria en un sistema está organizada como una jerarquía de niveles de almacenamiento, entre los que se mueve la información dependiendo de la necesidad de la misma en un determinado instante, La técnica de memoria virtual se ocupa de la transferencia de información entre la memoria principal y la secundaria, La memoria secundaria está normalmente soportada en un disco (o partición). Dado que, como se verá más adelante, la memoria virtual se implementa sobre un esquema de paginación, a este dispositivo se le denomina **dispositivo de paginación.** También se usa el término **dispositivo de *swap.*** Aunque este término no es muy adecuado, ya que proviene de la técnica del intercambio, por tradición se usa frecuentemente y se utilizará indistintamente en esta exposición.

Es importante recordar en este punto que, como se explicó en el Capítulo 1, el buen rendimiento del sistema de memoria virtual está basado en que los procesos presentan la propiedad de **proximidad de referencias.** Esta propiedad permite que un proceso genere muy pocos fallos aunque tenga en memoria principal solo una parte de su imagen de memoria (**conjunto residente**). El objetivo del sistema de memoria virtual es intentar que la información que está usando un proceso en un determinado momento (**conjunto de trabajo**) esté residente en memoria principal. O sea, que el conjunto residente del proceso contenga a su conjunto de trabajo.

Algunos beneficios del uso de memoria virtual son los siguientes:

- Se produce un aumento del grado de multiprogramación al no ser necesario que todo el

mapa de memoria de un proceso este en memoria principal para poder ejecutarlo. Este aumento implica una mejora en el rendimiento del sistema. Sin embargo, como se analizó en el Capítulo 2, si el grado de multiprogramación se hace demasiado alto, el número de fallos de página se dispara y el rendimiento del sistema baja drásticamente. A esta situación se le denomina **hiperpaginación** y se estudiará más adelante,

- Se pueden ejecutar programas más grandes que la memoria principal disponible.

Hay que resaltar que el uso de la memoria virtual no acelera la ejecución de un programa, sino que puede que incluso la ralentice debido a la sobrecarga asociada a las transferencias entre la memoria principal y la secundaria. Esto hace que esta técnica no sea apropiada para sistemas de tiempo real.

En esta sección se estudia, en primer lugar, el hardware requerido por esta técnica presentando los esquema de paginación, segmentación y segmentación paginada. Hay que resaltar que estos esquemas también se pueden usar sin memoria virtual, aportando beneficios apreciables con respecto a los esquemas de asignación contigua. De hecho, algunas versiones de UNIX usaban paginación e intercambio pero no memoria virtual. Sin embargo, en la actualidad el uso de estos esquemas está ligado a la memoria virtual. Después de presentar el hardware, se mostrará cómo construir un esquema de memoria virtual sobre el mismo estudiando las diferentes políticas de gestión de memoria virtual.

4.5.1. Paginación

Como se ha analizado previamente, los sistemas de gestión de memoria basados en asignación contigua presentan numerosas restricciones a la hora de satisfacer los requisitos que debe cumplir el gestor de memoria del sistema operativo. La paginación surge como un intento de paliar estos problemas sofisticando apreciablemente el hardware de gestión de memoria del procesador aumentado considerablemente la cantidad de información de traducción que se almacena por e proceso.

Aunque el objetivo de este capítulo no es estudiar en detalle cómo es el hardware de gestión memoria (MMU) de los sistemas basados en paginación, ya que es un tema que pertenece a la materia de estructura de computadoras, se considera que es necesario presentar algunos conceptos básicos del mismo para poder comprender adecuadamente su funcionamiento. Estos esquemas ya fueron presentados en el Capítulo 1, por lo que en esta sección no se volverá a incidir en los aspectos estudiados en el mismo.

Como su nombre indica, la unidad básica de este tipo de esquema es la página. La pagina corresponde con una zona de memoria contigua de un determinado tamaño. Por motivos de eficiencia en la traducción, este tamaño debe ser potencia de 2 (un tamaño de página de 4 KB es un valor bastante típico).

El mapa de memoria de cada proceso se considera dividido en páginas. A su vez, la memoria principal del sistema se considera dividida en zonas del mismo tamaño que se

denominan marcos de página. Un marco de página contendrá en un determinado instante una página de memoria de proceso. La estructura de datos que relaciona cada página con el marco donde esta almacenada tabla de páginas. El hardware de gestión de memoria usa esta tabla para traducir todas las direcciones que genera un programa. Como se analizó en el Capítulo 1, esta traducción consiste en detectar a que pagina del mapa corresponde una dirección lógica y acceder a la tabla de páginas para obtener el numero de marco donde está almacenada dicha página. La dirección física tendrá un desplazamiento con respecto al principio del marco igual que el que tiene la dirección con respecto al principio de la página. La Figura 4.13 muestra cómo es este esquema de traducción.

Típicamente, la MMU usa dos tablas de páginas. Una tabla de páginas de usuario para traducir las direcciones lógicas del espacio de usuario (en algunos procesadores se corresponden con direcciones que empiezan por un 0) y una tabla de páginas del sistema para las direcciones

lógicas del espacio del sistema (las direcciones lógicas que empiezan por un 1). Estas últimas sólo pueden usarse cuando el procesador está en modo privilegiado.

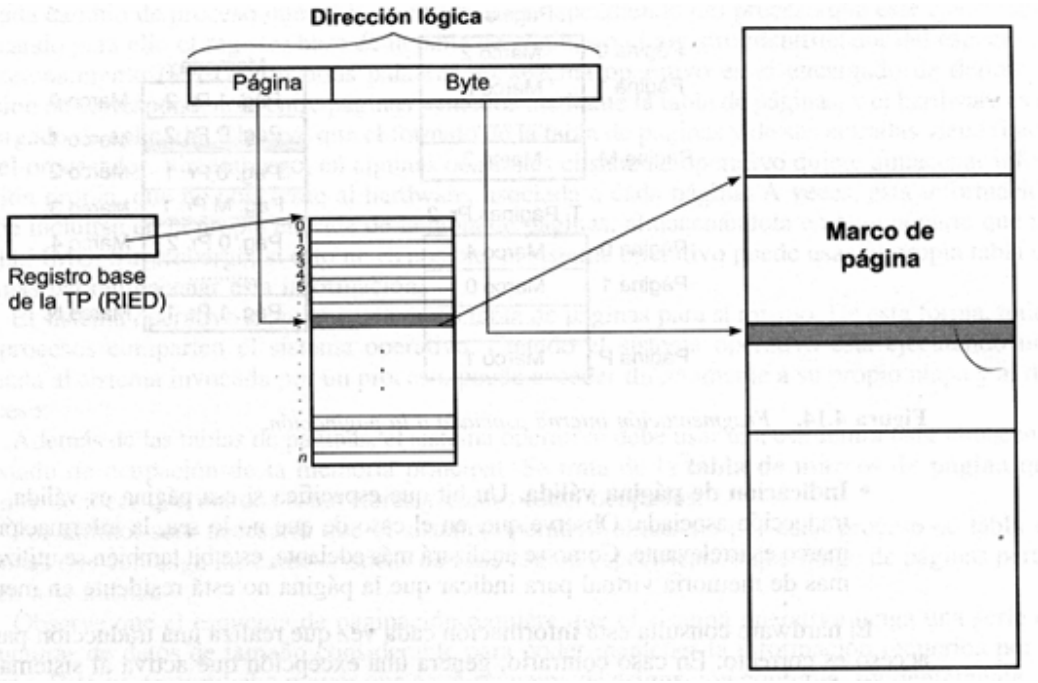


Figura 4.13. *Esquema de traducción de la paginación.*

Hay que resaltar que si se trata de un procesador con mapa común de entrada/salida y de memoria, las direcciones de entrada/salida también se accederán a través de la tabla de páginas. Para impedir que los procesos de usuario puedan acceder a ellas, estas direcciones de entrada/salida estarán asociadas a direcciones lógicas del espacio del sistema.

La paginación no proporciona un aprovechamiento óptimo de la memoria como lo haría un esquema que permitiese que cada palabra del mapa de memoria de un proceso pudiera corresponder con cualquier dirección de memoria. Sin embargo, esta estrategia, como se analizó al principio del capítulo, es irrealizable en la práctica dada la enorme cantidad de información de traducción que implicaría. La paginación presenta una

solución más realista permitiendo que cada página del mapa de un proceso se pueda corresponder con cualquier marco de memoria. Este cambio de escala reduce drásticamente el tamaño de la tabla de traducción, pero, evidentemente, proporciona un peor aprovechamiento de la memoria, aunque manteniéndose en unos términos aceptables. Observe que con la paginación se le asigna a cada proceso un número entero de marcos de pagina, aunque, en general, su mapa no tendrá un tamaño múltiplo del tamaño del marco. Por tanto, se desperdiciará parte del último marco asignado al proceso, lo que correspondería con las zonas sombreadas que aparecen en la Figura 4.14. A este fenómeno se le denomina **fragmentación interna** e implica que por cada proceso de desperdiciará, en término medio, la mitad de una página, lo cual es un valor bastante tolerable.

Cada entrada de la tabla de paginas, además del número de marco que corresponde con esa página, contiene información adicional tal como la siguiente:

- **Información de protección.** Un conjunto de bits que especifican qué tipo de accesos están permitidos. Típicamente, se controla el acceso de lectura, de ejecución y de escritura.

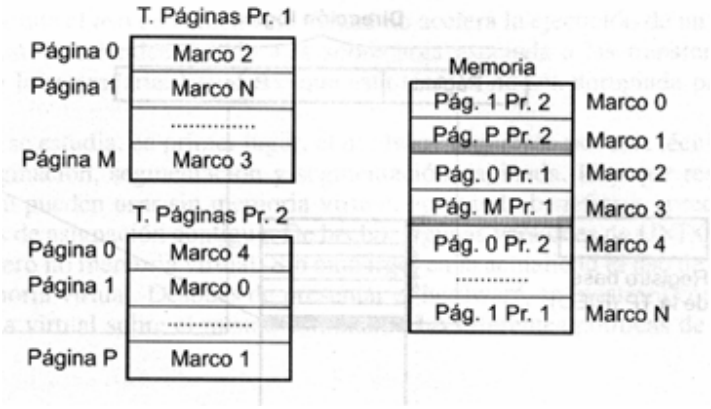


Figura 4.14. *Fragmentación interna asociada a la paginación.*

- **Indicación de página válida.** Un bit que especifica si esa página es válida, o sea, tiene traducción asociada, Observe que, en el caso de que no lo sea, la información del número del marco es irrelevante. Como se analizará más adelante, este bit también se utilizará en los esquemas de memoria virtual para indicar que la página no está residente en memoria principal.

El hardware consulta esta información cada vez que realiza una traducción para verificar si el acceso es correcto. En caso contrario, genera una excepción que activa al sistema operativo. En la entrada de la tabla de páginas puede haber información adicional, tal como la siguiente:

- **Indicación de página accedida.** La MMU activa este bit indicador cuando se dirección lógica que pertenece a esa página.
- **Indicación de pagina modificada.** La MMU activa este bit indicador cuando se escribe una dirección lógica que pertenece a esa página.
- **Desactivación de cache.** Este bit indica que no debe usarse la cache de la memoria principal para acelerar el acceso a las direcciones de esta página. En sistemas con mapa común de memoria y de entrada/salida, se activaría en las páginas que contienen direcciones

asociadas a dispositivos de entrada/salida, ya que para estas direcciones no se debe usar la cache sino acceder directamente al dispositivo.

Un aspecto importante en el rendimiento de un sistema de paginación es el tamaño de la página. Evidentemente, su tamaño debe ser potencia de 2 y, dado que va a servir de base memoria virtual, múltiplo del tamaño del bloque de disco. El tamaño típico puede estar entre 2KB y 16 KB. Hay que tener en cuenta que la determinación del tamaño optimo es un compromiso entre diversos factores:

- Un tamaño pequeño reduce la fragmentación y, como se verá más adelante, cuando se usa para implementar memoria virtual, permite ajustarse mejor al conjunto de trabajo del proceso.
- Un tamaño grande implica tablas más pequeñas y, cuando se usa para implementar memoria virtual, un mejor rendimiento en los accesos a disco.

Gestión del sistema operativo

La tabla de páginas hace la función de traducción que se analizó al principio del capítulo. El

sistema operativo mantendrá una tabla de páginas para cada proceso y se encargará de notificar al hardware.

en cada cambio de proceso qué tabla tiene que usar dependiendo del proceso que esté ejecutando, utilizando para ello el registro base de la tabla de páginas o el registro identificador del espacio de direccionamiento (RIED). En otras palabras, el sistema operativo es el encargado de definir la función de correspondencia entre paginas y marcos mediante la tabla de páginas, y el hardware es el encargado de aplicarla. Observe que el formato de la tabla de páginas y de sus entradas viene fijado por el procesador. Sin embargo, en algunas ocasiones el sistema operativo quiere almacenar información propia, que no concierne al hardware, asociada a cada página. A veces, esta información puede incluirse en la propia entrada de la tabla de páginas, almacenándola en alguna parte que no usa la MMU. Sin embargo, si esto no es posible, el sistema operativo puede usar su propia tabla de páginas para almacenar esta información.

El sistema operativo mantiene una única tabla de páginas para sí mismo. De esta forma, todos los procesos comparten el sistema operativo. Cuando el sistema operativo está ejecutando una llamada al sistema invocada por un proceso, puede acceder directamente a su propio mapa y al del proceso.

Además de las tablas de páginas, el sistema operativo debe usar una estructura para almacenar el estado de ocupación de la memoria principal. Se trata de la **tabla de marcos de página** que permite conocer qué marcos están libres y cuales están ocupados.

Por último, será necesario que el sistema operativo almacene por cada proceso su tabla de regiones que contenga las características de cada región especificando qué rango de páginas pertenecen a la misma.

Observe que el esquema de paginación requiere que el sistema operativo tenga una serie de estructuras de datos de tamaño considerable para poder mantener la información requerida por el mismo. Este gasto es mucho mayor que en el esquema de asignación contigua. Evidentemente, es el precio que hay que pagar para obtener una funcionalidad mucho mayor.

Valoración de la paginación

Una vez realizada esta presentación de los fundamentos de los esquemas de paginación, se puede analizar cómo cumplen los requisitos planteados al principio del capítulo:

- Espacios lógicos independientes. Cada proceso tiene una tabla de paginas que crea un espacio lógico independiente para el mismo. La tabla es, por tanto, una función de traducción que hace corresponder las páginas del mapa de memoria del proceso

con los marcos que tiene asignados.

- Protección. La tabla de páginas de un proceso restringe qué parte de la memoria puede ser accedida por el mismo, permitiendo asegurar que los procesos usan espacios disjuntos.
- Compartir memoria. Bajo la supervisión del sistema operativo, que es el único que puede manipular las tablas de paginas, dos o mas procesos pueden tener una página asociada al mismo marco de página. Observe que el compartimiento se realiza siempre con la granularidad de una página.
- Soporte de las regiones del proceso. La información de protección presente en cada entrada de la tabla de páginas permite controlar que los accesos a la región son del tipo que ésta requiere. Asimismo, la información de validez detecta cuándo se intenta acceder a huecos dentro del mapa del proceso y, además, elimina la necesidad de gastar memoria física para estos huecos.
- Maximizar el rendimiento. En primer lugar, la paginación, a pesar de la fragmentación interna, obtiene un buen aprovechamiento de la memoria, ya que elimina la necesidad de que el mapa de memoria de un proceso se almacene de forma contigua en memoria principal. Así, cualquier marco que esté libre se puede usar como contenedor de cualquier página

de cualquier proceso. Por otro lado, y más importante todavía, la paginación puede servir como

base para construir un esquema de memoria virtual.

- Mapas de memoria grandes para los procesos. Gracias a que los huecos no consumen espacio de almacenamiento, el sistema operativo puede crear un mapa de memoria para el proceso que ocupe todo su espacio lógico direccionable. De esta forma, habrá suficientes huecos para que las regiones crezcan o se incluyan nuevas regiones, sin penalizar con ello el gasto en espacio de almacenamiento. Por otro lado, y más importante todavía, la paginación permite implementar un esquema de memoria virtual.

Implementación de la tabla de paginas

El esquema básico de paginación que acaba de describirse, aunque funciona de manera correcta presenta serios problemas a la hora de implementarlo directamente, Estos problemas surgen debido a la necesidad de mantener las tablas páginas en memoria principal. Esto conlleva problemas de eficiencia y de consumo de espacio.

Por lo que se refiere a los problemas de eficiencia, dado que para acceder a la posición de memoria solicitada, la MMU debe consultar la entrada correspondiente de la tabla de páginas, se producirán dos accesos a memoria por cada acceso real solicitado por el programa. Esta sobrecargado es intolerable, ya que reduciría a la mitad el rendimiento del sistema. Para solventar este problema la MMU incluye internamente una especie de cache de traducciones llamada TLB *(Translation, Lookaside Buffer),* cuyo modo de operación se mostrará en la siguiente sección.

Por otro lado, existe un problema de gasto de memoria. Las tablas de páginas son muy grandes y hay una por cada proceso activo. Como se comentó previamente, la paginación permite construir mapas de memoria muy grandes para los procesos, ya que los huecos no consumen espacio. Sin embargo, si se usa todo el espacio de direccionamiento, como es deseable para conseguir que los procesos no tengan problemas de memoria durante su ejecución, la tabla de páginas debe tener tantas entradas como páginas hay en el espacio lógico, aunque muchas de ellas estén marcadas como invalidas al corresponder con huecos. Para resaltar esta circunstancia, a continuación se plantea un ejemplo.

Sea un procesador con una dirección lógica de 32 bits, un tamaño de página de 4 KB y tal que cada entrada de la tabla de páginas ocupa 4 bytes. Si se le asigna a cada proceso

todo el espacio direccionable, resulta á un mapa de memoria de 2^{32} bytes que corresponde con 2^{20} páginas. La tabla de páginas de cada proceso ocupa 4 MB (2^{20} entradas x 4bytes). Por tanto, se requerirían 4 MB de memoria principal para cada tabla, lo que resulta en un gasto de memoria inadmisible. Observe que

la mayoría de las entradas de la tabla de páginas de un proceso estarían vacías. Así, por ejemplo si las regiones de un proceso requieren 16 MB, lo cual es una cantidad bastante considerable, solo estarían marcadas como válidas 4.096 entradas de las más de un millón que hay en la tabla de

páginas. Observe que este problema se acentúa enormemente en los procesadores modernos que usan direcciones de 64 bits.

La solución a este problema son las tablas de paginas multinivel y las tablas invertidas.

***Translation Lookaside Buffer* (TLB)**

Para hacer que un sistema de paginación sea aplicable en la práctica es necesario que 1a mayoría de los accesos a memoria no impliquen una consulta a la tabla de páginas, sino que únicamente requieran el acceso a la posición solicitada. De esta forma, el rendimiento será similar al de un sistema sin paginación.

Como se comentó previamente, esto se logra mediante el uso de la TLB. Se trata de una pequeña memoria asociativa interna a la MMU que mantiene información sobre las últimas páginas accedidas, Cada entrada en la TLB es similar a la de la tabla de páginas (número de marco, protección, bit de referencia, etc.), pero incluye también el numero de la página para permitir realizar una búsqueda asociativa. Existen dos alternativas en el diseño de una TLB

dependiendo de si se almacenan identificadores de proceso o no.

- **TLB sin identificadores de proceso.** La MMU accede a la TLB sólo con el número de página. Por tanto, cada vez que hay un cambio de proceso el sistema operativo debe invalidar la TLB ya que cada proceso tiene su propio mapa.
- **TLB con identificadores de proceso.** La MMU accede a la TLB con el número de página y un identificador de proceso. En cada entrada de la TLB, por tanto, se almacena también este identificador, La MMU obtiene el identificador de un registro del procesador. El sistema operativo debe encargarse de asignarle un identificador a cada proceso y de rellenar este registro en cada cambio de proceso. De esta forma, no es necesario que el sistema operativo invalide la TLB en cada cambio de proceso, pudiendo existir en la TLB entradas correspondientes a varios procesos.

Tradicionalmente, la TLB ha sido gestionada directamente por la MMU sin intervención del sistema operativo. La MMU consulta la TLB y, si se produce un fallo debido a que la traducción de esa página no está presente, la propia MMU se encarga de buscar la traducción en la tabla de páginas e insertarla en la TLB. De hecho, la TLB es casi transparente al sistema operativo, que sólo debe encargarse en cada cambio de proceso de solicitar a la MMU su volcado y, en caso de que no use identificadores de proceso, su invalidación, Observe que es necesario realizar un volcado de la TLB a la tabla de páginas, ya que la información de los bits de página accedida o modificada se actualizan directamente en la TLB, pero no en la tabla de páginas.

Algunos procesadores modernos (como, por ejemplo, MIPS o Alpha) tienen un diseño alternativo en el que se traspasa parte de la gestión de la TLB al sistema operativo. A este esquema se le denomina **TLB gestionada por software.** La MMU se encarga de buscar la traducción en la TLB, pero si no la encuentra produce una excepción que activa al sistema operativo. Este se debe encargar de buscar «a mano» en la tabla de páginas e insertar en la TLB la traducción. Observe que, con este esquema, la MMU se simplifica considerablemente, ya que no tiene que saber nada de las tablas de páginas. Además,

proporciona mas flexibilidad, ya que el sistema operativo puede definir las tablas de página a su conveniencia, sin ninguna restricción impuesta por el hardware. Como contrapartida, el sistema será menos eficiente, ya que parte del proceso de traducción se realiza por software.

Tabla de páginas multinivel

Una manera de afrontar el problema del gasto de memoria de las tablas de páginas es utilizar tablas de página multinivel, Con este esquema, en vez de tener una unida tabla de páginas por proceso, hay una jerarquía de tablas. Existe una única tabla de paginas de primer nivel. Cada entrada de esta tabla apunta a tablas de páginas de segundo nivel. A su vez, las tablas de página de segundo nivel apuntan a tablas de página de tercer nivel. Así, sucesivamente, por cada nivel de la jerarquía. Las tablas de páginas del último nivel apuntan directamente a marcos de página. En la práctica, esta jerarquía se limita a dos o tres niveles.

A la hora de traducir una dirección lógica, el número de página contenido en la misma se considera dividido en tantas partes como niveles existan. En la Figura 4.15 se muestra cómo se realiza la traducción en un esquema con dos niveles.

Las entradas de las tablas de página de los distintos niveles tienen una estructura

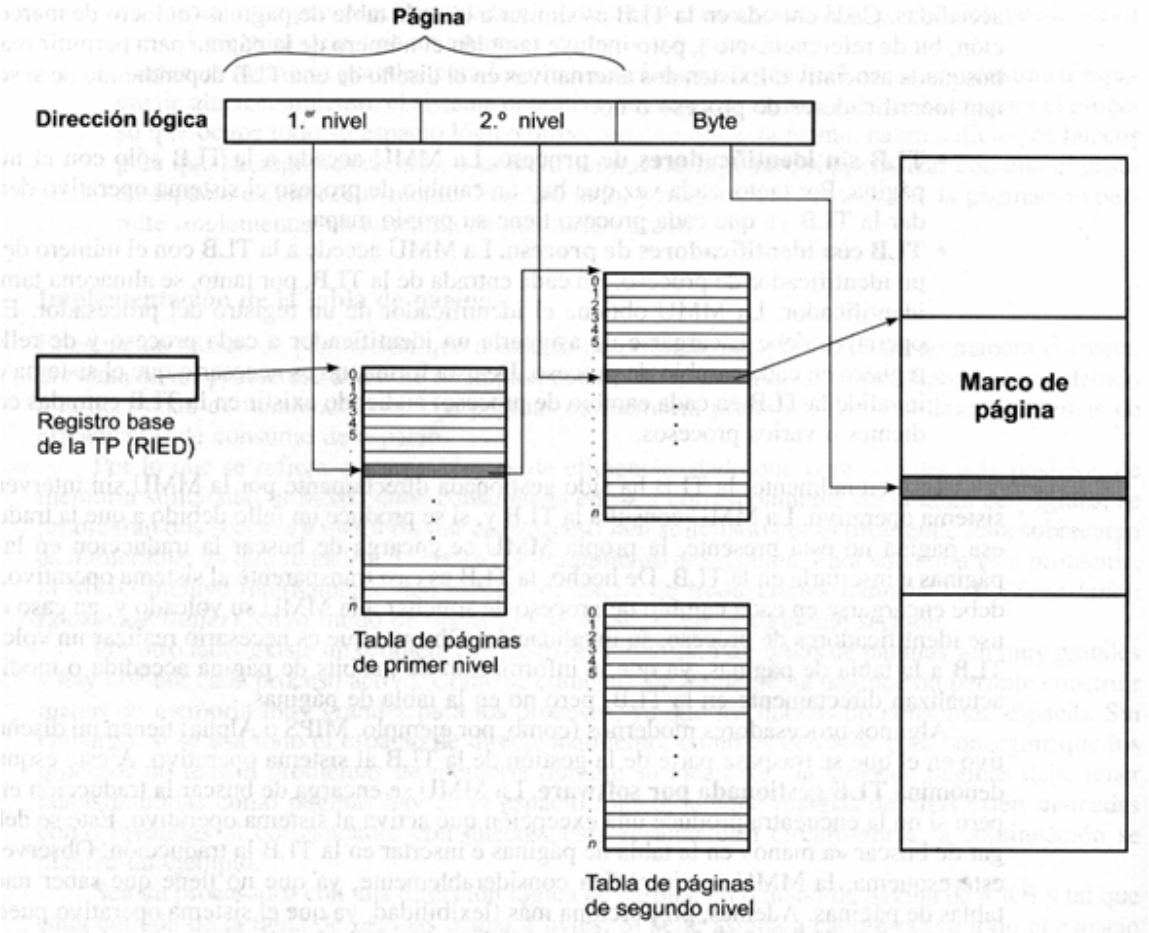


Figura 4.15. *Esquema de traducción con un esquema de paginación de dos niveles.*

similar, conteniendo información de protección y de validez, La diferencia entre las entradas de las tablas de niveles intermedios y las de las tablas de último nivel es que las primeras contienen la dirección de una tabla del siguiente nivel, mientras que las segundas incluyen la dirección del marco de página correspondiente.

La ventaja de este modelo es que si todas las entradas de una tabla de páginas de cualquier nivel están marcadas como inválidas, no es necesario almacenar esta tabla de páginas. Bastaría con marcar como inválida la entrada de la tabla de páginas de nivel superior que corresponde con esa tabla vacía. Observe que, dado que las tablas de paginas tradicionales de un solo nivel están llenas de entradas inválidas, lo que se está logrando con este esquema es ahorrar el almacenamiento de estas entradas inválidas.

Retomando el ejemplo expuesto anteriormente de un procesador con una dirección lógica de 32 bits, un tamaño de pagina de 4 KB y tal que cada entrada de la tabla de páginas ocupa 4 bytes, supóngase además que el procesador utiliza un esquema de dos niveles con 10 bits de la dirección dedicados a cada nivel. Con estos datos, la tabla de páginas de primer nivel tendría un tamaño de 4 KB (2^{10} entradas de 4 bytes) que podrían apuntar a 1,024 tablas de paginas de segundo nivel, A su vez, cada tabla de páginas de segundo nivel ocupará también 4 KB (2^{10} entradas de 4 bytes) que podrían apuntar a 1.024 marcos. Cada tabla de segundo nivel cubre un espacio de direccionamiento de 4 MB (1.024 marcos oui* 4 KB).

Si el proceso utiliza solo los 12 MB de la parte superior de su mapa (la de direcciones más bajas) y 4 MB de la parte inferior (la de direcciones más altas), el espacio gastado en tablas de páginas sería el correspondiente a la tabla de páginas de primer nivel (4 KB) más 4 tablas de páginas de segundo nivel para cubrir los 16 MB. Sólo 4 entradas de la tabla de páginas de primer nivel estarán marcadas como válidas. Esto hace un total de 20 KB (5 tablas de páginas 4 KB que ocupa cada una) dedicados a tablas de páginas. Comparando este resultado con el obtenido para un esquema con un solo nivel, se obtiene un ahorro apreciable. Se ha pasado de gastar 4 MB a sólo 20 KB. En la Figura 4.16 se puede apreciar este ahorro. Observe que, en el caso hipotético de que un proceso ocupara todo su mapa, las tablas de paginas multinivel no aportarían ningún ahorro, más bien al contrario ya que ocuparían más espacio.

Hay que resaltar que el uso de la TLB es todavía más imprescindible con las tablas multinivel, ya que si no habría que realizar siempre un acceso por cada nivel.

El esquema de tablas de página multinivel proporciona dos ventajas adicionales:

- Permite compartir tablas de páginas intermedias. Así, por ejemplo, en el caso de un sistema con dos niveles, si dos procesos comparten una región que se corresponde con una tabla de segundo nivel, pueden compartir directamente dicha tabla. O sea, cada proceso tendrá una entrada en su tabla de primer nivel que apunte a la tabla de segundo nivel compartida. De esta forma, se ahorra espacio en almacenar entradas repetidas. Observe que esto requiere que la región que sé comparte se corresponda con un número entero de tablas de segundo nivel.
- Sólo es preciso que este residente en memoria la tabla de páginas de primer nivel. Las restantes podrían almacenarse en el disco y traerlas bajo demanda.

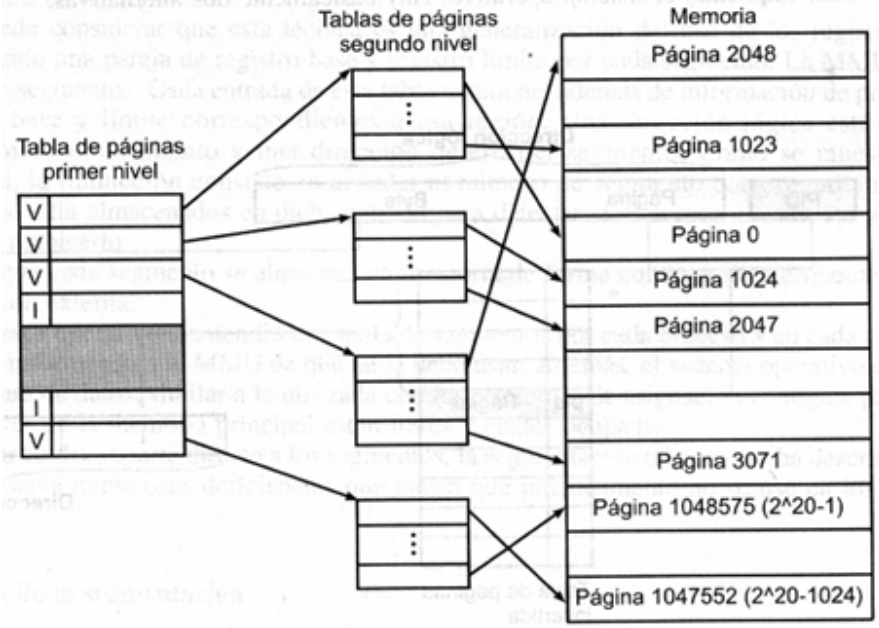


Figura 4.16. *Ventajas de las tablas de páginas multinivel.*

Tabla de páginas invertida

Otra alternativa para reducir el gasto en tablas de paginas es usar una tabla invertida. Una tabla de páginas invertida contiene una entrada por cada marco de página. Cada entrada identifica qué página está almacenada en ese marco y cuáles son sus características. Para ello, contiene el número de página y un identificador de proceso al que pertenece la página. Observe que, con este esquema hay una única tabla de paginas cuyo tamaño es proporcional al tamaño de la memoria principal. La Figura 4.17 ilustra cómo se realiza una traducción con este esquema.

La MMU usa una TLB convencional, pero cuando no encuentra en ella la traducción debe acceder a la tabla invertida para encontrarla, Dado que la tabla está organizada por marcos, no se puede hacer una búsqueda directa. En principio, se deberían acceder a todas las entradas buscando la página solicitada. Para agilizar esta búsqueda, generalmente la tabla de páginas se organiza como una tabla *hash.* En esta exposición no se entrará en más detalles sobre esta organización (p. ej.: explicar cómo se resuelven las colisiones de la función *hash).*

Esta organización dificulta el compartimiento de memoria ya que, en principio, no se pueden asociar dos paginas al mismo marco. Hay algunas soluciones a este problema, pero quedan fuera del alcance de esta exposición.

Un último punto que hay que resaltar es que la tabla invertida reduce apreciablemente el gasto de memoria en tablas, ya que sólo almacena información de las páginas válidas, Sin embargo, como se analizará más adelante, cuando se usa este esquema para implementar memoria virtual, el sistema operativo debería mantener sus propias tablas de pagina para guardar información de las páginas que no están presentes en memoria principal.

Para terminar con la paginación, es conveniente hacer notar que hay numerosos aspectos avanzados sobre este tema que no se han explicado debido al alcance de esta presentación y a que algunos de ellos son transparentes al sistema operativo. Entre ellos,

se puede destacar el tema de cómo se relaciona la paginación con el uso de una cache de memoria principal, ya que en él puede estar implicado el sistema operativo. Hay básicamente dos alternativas:

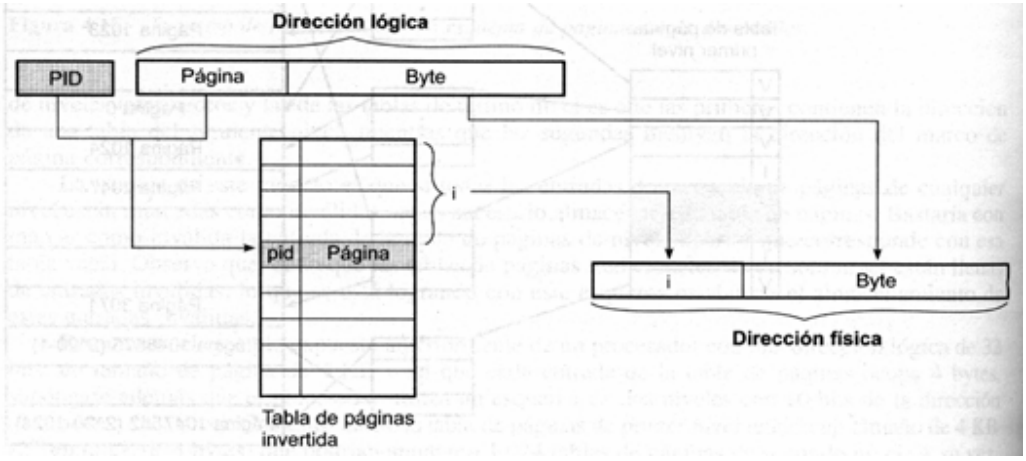


Figura 4.17. *Esquema de traducción usando una tabla de páginas invertida.*

- Procesadores (p. ej.: PA-RISC) que acceden a la cache usando la dirección virtual (o parte de ella). Esto permite realizar la búsqueda en la TLB y en la cache en paralelo, pero obliga a que el sistema operativo tenga que invalidar la cache en cada cambio de proceso para evitar incoherencias,
- Procesadores (p. ej.: Alpha) que acceden a la cache usando la dirección física (o parte de ella). Con este esquema el sistema operativo no tiene que invalidar la cache, pero no permite que el acceso a la TLB y a la cache se hagan en paralelo (aunque si el tamaño de la cache es igual al de la página, si se podría hacer en paralelo).

4.5.2. Segmentación

Con la paginación, la MMU no sabe nada sobre las distintas regiones de los procesos. Sólo entiende de paginas. El sistema operativo debe guardar para cada proceso una tabla de regiones que especifique qué páginas pertenecen a cada región. Esto tiene dos desventajas:

- Para crear una región hay que rellenar las entradas de las páginas pertenecientes a la región con las mismas características. Así, por ejemplo, si se trata de una región de código que ocupa diez páginas, habrá que rellenar diez entradas especificando una protección que no permita modificarlas.
- Para compartir una región, hay que hacer que las entradas correspondientes de dos procesos apunten a los mismos marcos.

En resumen, lo que se está echando en falta es que la MMU sea consciente de la existencia de las regiones y que permita tratar a una región como una entidad.

La segmentación es una técnica hardware que intenta dar soporte directo a las regiones. Para ello, considera el mapa de memoria de un proceso compuesto de múltiples segmentos. Cada región se almacenará en un segmento.

Se puede considerar que esta técnica es una generalización del uso de los registros valla, pero utilizando una pareja de registro base y registro límite por cada segmento. La MMU maneja una tabla de segmentos. Cada entrada de esta tabla mantiene, además de información de protección, el registro base y límite correspondientes a esa región. Una dirección lógica está formada por un número de segmento y una dirección dentro del

segmento. Como se muestra en la Figura 4.18, la traducción consiste en acceder al numero de segmento correspondiente y usar los registros valla almacenados en dicha entrada para detectar si el acceso es correcto y, en caso afirmativo, reubicarlo.

Dado que cada segmento se almacena en memoria de forma contigua, este esquema presenta fragmentación externa.

El sistema operativo mantendrá una tabla de segmentos por cada proceso y en cada cambio de proceso irá informando a la MMU de qué tabla debe usar. Además, el sistema operativo debe usar una estructura de datos, similar a la utilizada con los esquemas de asignación contigua, para conocer qué partes de la memoria principal están libres y cuáles ocupadas.

A pesar de dar soporte directo a los segmentos, la segmentación tal como se ha descrito en esta sección presenta numerosas deficiencias que hacen que prácticamente no se use en los sistemas reales.

Valoración de la segmentación

A continuación, se analiza si la segmentación cumple los requisitos planteados al principio del capítulo:

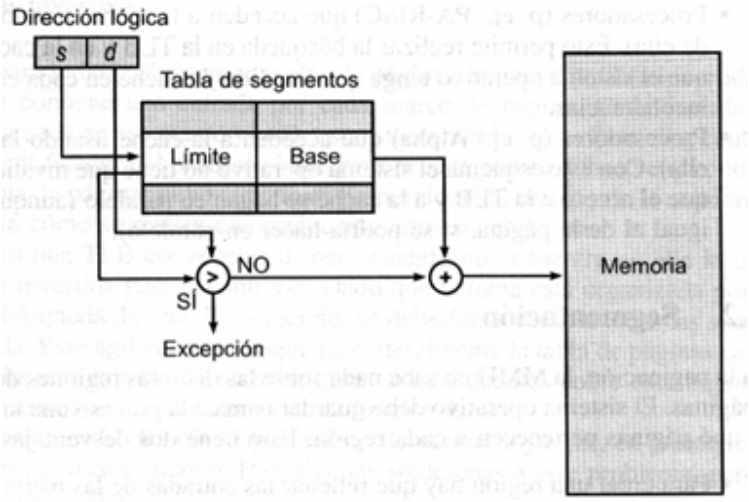


Figura 4.18. *Esquema de traducción usando segmentación.*

- Espacios lógico independientes. Cada proceso tiene una tabla de segmentos que crea espacio lógico independiente para el mismo,
- Protección. La tabla de segmentos de un proceso restringe que parte de la memoria puede ser accedida por el mismo, permitiendo asegurar que los procesos usan espacios disjuntos.
- Compartir memoria. Bajo la supervisión del sistema operativo, que es el único que manipula las tablas de segmentos, dos o más procesos pueden tener un segmento asociado a la misma zona de memoria.
- Soporte de las regiones del proceso. Este es precisamente el punto fuerte de este sistema.
- Maximizar el rendimiento. Esta es una de sus deficiencias. Por un lado, presenta fragmentación externa, por lo que no se obtiene un buen aprovechamiento de la memoria. Por un lado, y mas importante todavía, esta técnica no facilita la implementación de esquemas de memoria virtual debido al tamaño variable de los segmentos.

- Mapas de memoria grandes para los procesos. No contempla adecuadamente esta característica al no permitir implementar eficientemente un sistema de memoria virtual.

4.5.3. Segmentación paginada

Como su nombre indica, la segmentación paginada intenta aun lo mejor de los dos esquemas anteriores. La segmentación proporciona soporte directo a las regiones del proceso y la paginación permite un mejor aprovechamiento de la memoria y una base para construir un esquema de memoria virtual.

Con esta técnica, un segmento está formado por un conjunto de página, y, por tanto, no tiene que estar contiguo en memoria. La MMU utiliza una tabla de segmentos, tal que cada entrada de la tabla apunta a una tabla de paginas. La Figura 4.19 ilustra el proceso de traducción en este esquema.

Valoración de la segmentación paginada

La valoración de esta técnica se corresponde con la unión de los aspectos positivos de los esquemas anteriores:

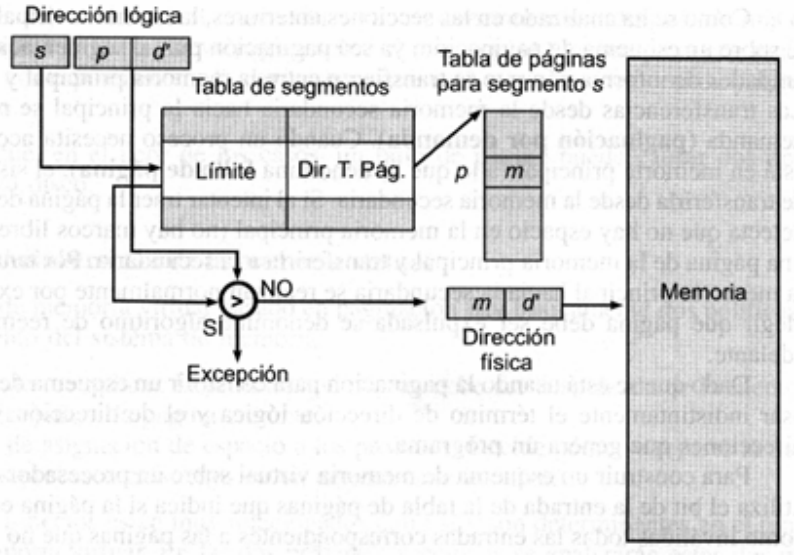


Figura 4.19. *Esquema de traducción usando segmentación paginada.*

- Espacios lógicos independientes. Cada proceso tiene una tabla de segmentos que crea un espacio lógico independiente para el mismo.
- Protección. La tabla de segmentos de un proceso restringe qué parte de la memoria puede ser accedida por el mismo, permitiendo asegurar que los procesos usan espacios disjuntos.
- Compartir memoria. Bajo la supervisión del sistema operativo, que es el único que puede manipular las tablas de segmentos, dos o más procesos pueden tener un segmento asociado a la misma zona de memoria.
- Soporte de las regiones del proceso, gracias a la segmentación.
- Maximizar el rendimiento, La paginación proporciona un buen aprovechamiento de la memoria y puede servir como base para construir un esquema de memoria virtual.
- Mapas de memoria grandes para los procesos. La paginación permite implementar un esquema de memoria virtual.

Es importante resaltar que, aunque la segmentación paginada ofrece más funcionalidad que la paginación, requiere un hardware más complejo que, además, no está presente en la mayoría de los procesadores. Por ello, la mayoría de los sistemas operativos están construidos suponiendo que el procesador proporciona un esquema de paginación.

4.5.4. Paginación por demanda

Una vez analizados los diversos esquemas hardware, en esta sección se plantea cómo construir el mecanismo de la memoria virtual tomando como base dichos esquemas. Como se comentó previamente, estos esquemas también pueden usarse sin memoria virtual, ya que de por sí proporcionan numerosas ventajas sobre los esquemas de asignación contigua. Sin embargo, en la actualidad su uso siempre está ligado a la memoria virtual.

Como se ha analizado en las secciones anteriores, la memoria virtual se construye generalmente sobre un esquema de paginación, ya sea paginación pura o segmentación paginada. Por tanto, las unidades de información que se transfieren entre la memoria principal y la secundaria son páginas. Las transferencias desde la memoria secundaria hacia la principal se realizan normalmente bajo demanda (**paginación por demanda**). Cuando un proceso necesita acceder a una página que no están en memoria principal (a lo que se denomina **fallo de página**), el sistema operativo se encarga de transferirla desde la memoria secundaria. Si al intentar traer la página desde memoria secundaria se detecta que no hay espacio en la memoria principal (no hay marcos libres), será necesario expulsar una página de la memoria principal y transferirla a la secundaria. Por tanto, las transferencias desde la memoria principal hacia la secundaria se realizan normalmente por expulsión. El algoritmo para elegir qué página debe ser expulsada se denomina algoritmo de reemplazo y se analizará más adelante.

Dado que se está usando la paginación para construir un esquema de memoria virtual, se puede usar indistintamente el termino de dirección lógica y el de dirección virtual para referirse a las direcciones que genera un programa.

Para construir un esquema de memoria virtual sobre un procesador que ofrezca paginación, se utiliza el bit de la entrada de la tabla de paginas que indica si la pagina es válida. Estarán marcadas como inválidas todas las entradas correspondientes a las páginas que no están residentes en memoria principal en ese instante. Para estas páginas, en vez de guardarse la dirección del marco, se almacenará la dirección del bloque del dispositivo que contiene la página. Cuando se produzca un acceso a una de estas paginas, se producirá una excepción (fallo de página) que activará al sistema operativo que será el encargado de traerla desde la memoria secundaria.

Observe que dado que se utiliza el bit validez para marcar la ausencia de una página y este mismo bit también se usa para indicar que una página es realmente inválida (una página que corresponde con un hueco en el mapa), es necesario que el sistema operativo almacene información asociada a la pagina para distinguir entre esos dos casos.

Por ultimo, hay que comentar que algunos sistemas de memoria virtual usan la técnica de la **prepaginaciòn.** En un fallo de página no sólo se traen la página en cuestión, sino también 1as páginas adyacentes, ya que es posible que el proceso las necesite en un corto plazo de tiempo. La efectividad de esta técnica va a depender de si hay acierto en esta predicción.

Tratamiento del fallo de página

La paginación por demanda esta dirigida por la ocurrencia de excepciones de fallo de página que indican al sistema operativo que debe traer una página de memoria secundaria a primaria puesto que un proceso la requiere. A continuación, se especifican los pasos

típicos en el tratamiento de un fallo de página:

- La MMU produce una excepción y típicamente deja en un registro especial la dirección que causó el fallo.
- Se activa el sistema operativo que comprueba si se trata de una dirección correspondiente una página realmente invalida o se corresponde con una página ausente de memoria. Si 1a pagina es invalida, se aborta el proceso o se le manda una señal. En caso contrario, se realizan los pasos que se describen a continuación.
- Se consulta la tabla de marcos para buscar uno libre.
- Si no hay un marco libre, se aplica el algoritmo de reemplazo para seleccionar una página para expulsar El marco seleccionado se desconectará de la página a la que esté asociado poniendo como inválida la entrada correspondiente. Si la página está modificada, previamente hay que escribir su contenido a la memoria secundaria.
- Una vez que se obtiene el marco libre, ya sea directamente o después de una expulsión, se inicia la lectura de la nueva página sobre el marco y, al terminar la operación, se rellena la entrada correspondiente a la pagina para que esté marcada como válida y apunte al marco utilizado.

Observe que, en el peor de los casos, un fallo de pagina puede causar dos operaciones de entrada/salida al disco.

Políticas de administración de la memoria virtual

En un sistema de memoria virtual basado en paginación hay básicamente dos políticas que definen el funcionamiento del sistema de memoria:

- Política de reemplazo. Determina qué pagina debe ser desplazada de la memoria principal para dejar sitio a la pagina entrante.
- Política de asignación de espacio a los procesos. Decide cómo se reparte la memoria física entre los procesos existentes en un determinado instante.

Estos dos aspectos están muy relacionados entre sí y son determinantes en el rendimiento del sistema de memoria virtual. En las dos próximas secciones se analizarán estas dos políticas.

4.5.5. Políticas de reemplazo

Las estrategias de reemplazo se pueden clasificar en dos categorías: reemplazo global y reemplazo local. Con una estrategia de reemplazo global, se puede seleccionar, para satisfacer el fallo de página de un proceso, un marco que actualmente tenga asociada una página de otro proceso. Esto es, un proceso puede quitarle un marco de pagina a otro. La estrategia de reemplazo local requiere que, para servir el fallo de página de un proceso, sólo puedan usarse marcos de páginas libres o marcos ya asociados al proceso.

Existen numerosos trabajos, tanto teóricos como experimentales, sobre algoritmos de reemplazo de páginas. En esta sección se describirán los algoritmos de reemplazo más típicos. Para profundizar en aspectos teóricos avanzados se recomienda [Maekawa, 1987].

Todos estos algoritmos: pueden utilizarse tanto para estrategias globales como locales. Cuando se aplica un algoritmo determinado utilizando una estrategia global, el criterio de evaluación del algoritmo se aplicará a todas las páginas en memoria principal. En el caso de una estrategia local, el criterio de evaluación del algoritmo se aplica sólo a las páginas en memoria principal que pertenecen al proceso que causó el fallo de página. La descripción de los algoritmos, por tanto, se realizara sin distinguir entre los dos tipos de estrategias.

Por último, hay que comentar que el objetivo básico de cualquier algoritmo de reemplazo es minimizar la tasa de fallos de página, intentando además que la sobrecarga asociada a la ejecución del algoritmo sea tolerable y que no se requiera una MMU con

características especificas.

Algoritmo de reemplazo óptimo

Un algoritmo óptimo debe generar el mínimo número de fallos de página. Por ello, la página que se debe reemplazar es aquella que tardará más tiempo en volverse a usar.

Evidentemente, este algoritmo es irrealizable, ya que no se puede predecir cuáles serán las siguientes páginas accedidas, El interés de este algoritmo es que sirve para comparar el rendimiento de otros algoritmos realizables.

Algoritmo FIFO *(First Input-First Output,* primera en entrar-primera en salir)

Una estrategia sencilla e intuitivamente razonable es seleccionar para la sustitución la página que lleva más tiempo en memoria. La implementación de este algoritmo es simple. Además, no necesita ningún apoyo hardware especial. El sistema operativo debe mantener una lista de las páginas que están en memoria, ordenada por el tiempo que llevan residentes. En el caso de una estrategia local, se utiliza una lista por cada proceso. Cada vez que se trae una nueva página a memoria, se pone a1 final de la lista. Cuando se necesita reemplazar, se usa la página que está al principio de la lista.

Sin embargo, el rendimiento del algoritmo no es siempre bueno. La página que lleva mas tiempo residente en memoria puede contener instrucciones o datos que se acceden con frecuencia. Además, en determinadas ocasiones, este algoritmo presenta un comportamiento sorprendente conocido como la anomalía de Belady [Belady, 1969]. Intuitivamente parece que cuantos mas marcos de página haya en el sistema, menos fallos de página se producirán. Sin embargo, ciertos patrones de referencias causan que este algoritmo tenga un comportamiento opuesto. El descubrimiento de esta anomalía resultó al principio sorprendente y llevó al desarrollo de modelos teóricos para analizar los sistemas de paginación. En la práctica, esta anomalía es más bien una curiosidad que demuestra que los sistemas pueden tener a veces comportamientos inesperados.

Algoritmo de la segunda oportunidad o algoritmo del reloj

El algoritmo de reemplazo con segunda oportunidad es una modificación sencilla del FIFO que evita el problema de que una página muy utilizada sea eliminada por llevar mucho tiempo residente.

En este algoritmo, cuando se necesita reemplazar una página, se examina el bit de referencia de la página más antigua (la primera de la lista). Si no está activo, se usa esta página para el reemplazo. En caso contrario, se le da una segunda oportunidad a la página poniéndola al final de la lista y desactivando su bit de referencia. Por tanto, se la considera como si acabara de llegar a memoria. La búsqueda continuará hasta que se encuentre una página con su bit de referencia desactivado. Observe que si todas las paginas tienen activado su bit de referencia, el algoritmo degenera en un FIFO puro.

Para implementar este algoritmo, se puede usar una lista circular de las páginas residentes en memoria, en vez de una lineal (en el caso de una estrategia local, se utiliza una lista circular por cada proceso). Existe un puntero que señala en cada instante al principio de la lista. Cuando llega a memoria una página, se coloca en el lugar donde señala el puntero y, a continuación, se avanza el puntero al siguiente elemento de la lista. Cuando se busca una página para reemplazar, se examina el bit de referencia de la página a la que señala el puntero. Si está activo, se desactiva y se avanza el puntero al siguiente elemento. El puntero avanzará hasta que se encuentre una página con el bit dc referencia desactivado. Esta forma de trabajo imita al comportamiento de un reloj donde el puntero que recorre la lista se comporta como la aguja del reloj. Debido a ello, a esta estrategia también se 1e denomina **algoritmo del reloj.**

Algoritmo LRU *(Least Recently Used,* menos recientemente usada)

El algoritmo LRU está basado en el principio de proximidad temporal de referencias: si

es probable que se vuelvan a referenciar las páginas accedidas recientemente, la página

que se debe reemplazar es la que no se ha referenciado desde hace más tiempo.

El algoritmo LRU no sufre la anomalía de Belady. Pertenece a una clase de algoritmos denominados algoritmos de pila. La propiedad de estos algoritmos es que las páginas residentes en memoria para un sistema con marcos de página son siempre un subconjunto de las que habría en un sistema con $n + 1$ marcos.

Esta propiedad asegura que un algoritmo de este tipo nunca sufrirá la anomalía de Belady.

Hay un aspecto sutil en este algoritmo cuando se considera su versión global. A la hora de seleccionar una página, no habría que tener en cuenta el tiempo de acceso real, sino el tiempo lógico de cada proceso. O sea, habría que seleccionar la pagina que haya sido menos recientemente usada teniendo en cuenta el tiempo lógico de cada proceso.

A pesar de que el algoritmo LRU es realizable y proporciona un rendimiento bastante bueno, su implementación eficiente es difícil y requiere un considerable apoyo hardware. Una implementación del algoritmo podría basarse en utilizar un contador que se incremente por cada referencia a memoria. Cada posición de la tabla de paginas ha de tener un campo de tamaño suficiente para que quepa el contador. Cuando se referencia a una página, el valor actual del contador se copia por hardware a la posición de la tabla correspondiente a esa página. Cuando se produce una fallo de página, el sistema operativo examina los contadores de todas las páginas residentes en memoria y selecciona como víctima aquella que tiene el valor menor. Esta implementación es factible aunque requiere un hardware complejo y muy especifico.

***Buffering* de paginas**

Una situación que intentan evitar la mayoría de los sistemas es la que se produce cuando la página seleccionada para reemplazar está modificada. En este caso, el tratamiento del fallo de pagina implica dos operaciones al disco, aumentando considerablemente el tiempo de servicio del fallo.

Una solución utilizada con cierta frecuencia es el *buffering* de páginas. Esta estrategia consiste en mantener un conjunto de marcos de página libres. Cuando se produce un fallo de página, se usa un marco de página libre, pero no se aplica el algoritmo de reemplazo. Esto es, se consume un marco de página pero no se libera otro. Cuando el sistema operativo detecta que el número de marcos de página disminuye por debajo de un cierto umbral, aplica repetidamente el algoritmo de reemplazo hasta que el numero de marcos libres sea suficiente. Las páginas liberadas que no están modificadas pasan a la lista de marcos libres. Las páginas que han sido modificadas pasan a la lista de modificadas.

Las páginas que están en cualquiera de las dos listas pueden recuperarse si vuelven a referenciarse. En este caso, la rutina de fallo de página recupera la página directamente de la lista y actualiza la entrada correspondiente de la tabla de páginas para conectarla. Observe que este fallo de pagina no implicaría operaciones de entrada/salida.

Las paginas en la lista de modificadas se pueden escribir en tandas al dispositivo para obtener un mejor rendimiento. Cuando la página modificada se ha escrito al dispositivo, se la incluye en la lista de marcos libres.

Esta estrategia puede mejorar el rendimiento de algoritmos de reemplazo que no sean muy efectivos. Así, si el algoritmo de reemplazo decide revocar una página que en realidad está siendo usada por un proceso, se producirá inmediatamente un fallo de página que la recuperará de las listas.

Retención de paginas en memoria

Para acabar esta sección en la que se han presentado diversos algoritmos de reemplazo, hay que resaltar que no todas las paginas residentes en memoria son candidatas al reemplazo. Se puede considerar que algunas páginas están «atornilladas» a la memoria principal.

En primer lugar, están las páginas del propio sistema operativo. Por simplicidad, la mayoría de los sistemas operativos tienen su mapa de memoria fijo en memoria principal.

Además, si se permite que los dispositivos de entrada/salida que usan DMA realicen transferencias directas a la memoria de un proceso, será necesario marcar las paginas implicadas como no reemplazables hasta que termine la operación.

Por último, algunos sistemas operativos ofrecen servicios a las aplicaciones que les permiten solicitar que una o más página de su mapa queden retenidas en memoria. Este servicio puede ser útil para procesos de tiempo real que necesitan evitar que se produzcan fallos de página imprevistos. Sin embargo, el uso indiscriminado de este servicio puede afectar gravemente al rendimiento del sistema.

4.5.6. Política de asignación de marcos de página

En un sistema con multiprogramación existen varios procesos activos simultáneamente que comparten la memoria del sistema. Es necesario, por tanto, determinar cuántos marcos de página asignan a cada proceso. Existen dos tipos de estrategias de asignación: asignación fija o asignación dinámica.

Asignación fija

Se asigna a cada proceso un numero fijo de marcos de página. Normalmente, este tipo de asignación lleva asociada una estrategia de reemplazo local. El número de marcos asignados no varía, que un proceso sólo usa para reemplazo los marcos que tiene asignados.

La principal desventaja de esta alternativa es que no se adapta a las diferente, necesidades de memoria de un proceso a lo largo de su ejecución . Una característica positiva es que el comportamiento del proceso es relativamente predecible.

Existen diferentes criterios para repartir los marcos de las páginas entre los procesos existentes. Puede depender de múltiples factores tales como el tamaño del proceso o su prioridad.

Por otra parte, cuando se usa una estrategia de asignación fija, el sistema operativo determina cuál es el número máximo de marcos asignados al proceso. Sin embargo, la arquitectura de máquina establece el número mínimo de marcos que deben asignarse a un proceso.

Por ejemplo, si la ejecución de una instrucción puede generar seis fallos de página y el sistema operativo asigna cinco marcos de página a un proceso que incluya esta instrucción, el proceso podría no terminar de ejecutar esta instrucción. Por tanto, el número mínimo de marcos de página para una determinada arquitectura quedará fijado por la instrucción que pueda generar el máximo número de fallos de página.

Asignación dinámica

El número de marcos asignados a un proceso varía según la necesidades que tenga el proceso(y posiblemente el resto de procesos del sistema) en diferentes instantes de tiempo. Con este tipo asignación se pueden usar tanto estrategias de reemplazo locales como globales.

- Con reemplazo local, el proceso va aumentando o disminuyendo su conjunto residente dependiendo de sus necesidades en las distintas fases de ejecución del programa.
- Con reemplazo global, los procesos compiten en el uso de la memoria quitándose entre sí las páginas.

La estrategia de reemplazo global hace que el comportamiento del proceso en ejecución sea difícilmente predecible. El principal problema de este tipo asignación es que la tasa de fallos página de un programa puede depender de las características de los

otros procesos que estén activos en el sistema.

4.5.7. Hiperpaginación

Si el número de marcos de página asignados a un proceso no es suficiente para almacenar las páginas referenciadas activamente por el mismo, se producirá un número elevado de fallos de página. A esta situación se le denomina **hiperpaginación** *(thrashing).* Cuando se produce la hiperpaginación, el proceso pasa más tiempo en la cola de servicio del dispositivo de paginación que ejecutando. Dependiendo del tipo de asignación utilizado, este problema puede afectar a procesos individuales o a todo el sistema.

En un sistema operativo que utiliza una estrategia de asignación fija, si el numero de marcos asignados al proceso no es suficiente para albergar su conjunto de trabajo en una determinada fase de su ejecución, se producirá hiperpaginación en ese proceso. Esto traerá consigo un aumento considerable de su tiempo de ejecución, pero, sin embargo, el resto de los procesos del sistema no se ven afectados directamente.

Con una estrategia de asignación dinámica, el numero de marcos asignados a un proceso se va adaptando a sus necesidades, por lo que, en principio, no debería presentarse este problema. Sin embargo, si el número de marcos de página existentes en el sistema no son suficientes para almacenar los conjuntos de trabajo de todos los procesos, se producirían fallos de página frecuentes y, por tanto, el sistema sufrirá hiperpaginación. La utilización del procesador disminuirá, puesto que a ya aumenta el tiempo que dedica al tratamiento de fallos de página. Como se puede observar en la Figura 4.20, no se trata de una disminución progresiva, sino más bien drástica, que se debe a que al aumentar el número de procesos aumenta, por un lado, la tasa de fallos de página de cada proceso (hay menos marcos de pagina por proceso) y, por otro lado, aumenta el tiempo de servicio del dispositivo de paginación (crece la longitud de la cola de servicio del dispositivo).

Cuando se produce esta situación, se deben suspender uno o varios procesos liberando sus páginas. Es necesario establecer una estrategia de control de carga que ajuste el grado de multiprogramación en el sistema para evitar que se produzca hiperpaginación. A continuación, se plantean algunas políticas de control de carga.

Estrategia del conjunto de trabajo

Como se comentó previamente, cuando un proceso tiene residente en memoria su conjunto de trabajo, se produce una baja tasa de fallos de página. Una posible estrategia consiste en determinar los conjuntos de trabajo de todos los procesos activos para intentar mantenerlos residentes en memoria principal.

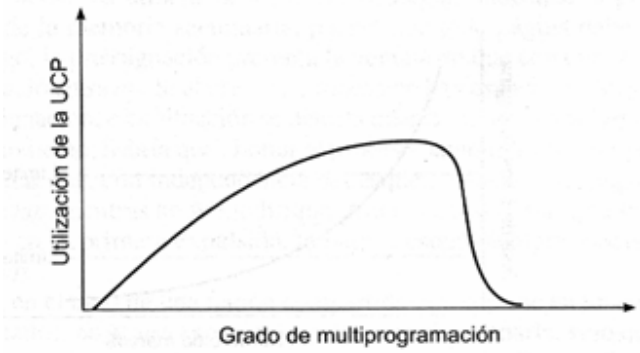


Figura 4.20. *Hiperpaginación.*

Para poder determinar el conjunto de trabajo de un proceso es necesario dar una definición más formal de este término. El conjunto de trabajo de un proceso es el conjunto de paginas asignadas por un proceso en las últimas *n* referencias. El numero *n* se denomina la ventana del conjunto trabajo. El valor de *n* es un factor critico para el funcionamiento efectivo de esta estrategia .Si es demasiado grande, la ventana podría englobar varias fases de ejecución del proceso llevando a una estimación excesiva de las necesidades del proceso. Si es demasiado pequeño, la ventana podría no englobar la situación actual del proceso con lo que se generarían demasiados fallos de página.

Suponiendo que el sistema operativo es capaz de detectar cual es el conjunto de trabajo de cada proceso, se puede especificar una estrategia de asignación dinámica con reemplazo local y control de carga.

- Si el conjunto de trabajo de un proceso decrece, se liberan los marcos asociados a las páginas que ya no están en el conjunto de trabajo.
- Si el conjunto de trabajo de un proceso crece, se asignan marcos para que puedan contener las nuevas páginas que han entrado a formar parte del conjunto de trabajo. Si no hay marcos libres, hay que realizar un control de carga suspendiendo uno o más procesos y liberando sus páginas.

El problema de esta estrategia es cómo poder detectar cuál es el conjunto de trabajo de cada proceso. Al igual que ocurre con el algoritmo LRU, se necesitaría una MMU específica que fuera controlando las páginas accedidas por cada proceso durante las últimas *n* referencias.

Estrategia de administración basada en la frecuencia de fallos de página

Esta estrategia busca una solución más directa al problema de la hiperpaginación. Se basa en controlar la frecuencia de fallos de página de cada proceso. Como se ve en la Figura 4.21, se establecen una cuota superior y otra inferior de la frecuencia de fallos de página de un proceso. Basándose en esa idea, a continuación se describe una estrategia de asignación dinámica con reemplazo local y control de carga.

- Si la frecuencia de fallos de un proceso supera el límite superior, se asignan página adicionales al proceso. Si la tasa de fallos crece por encima del límite y no hay marcos libres, se suspende algún proceso liberando sus páginas.
- Cuando el valor de la tasa de fallos es menor que el limite inferior, se liberan marcos asignados al proceso seleccionándolos mediante un algoritmo de reemplazo.

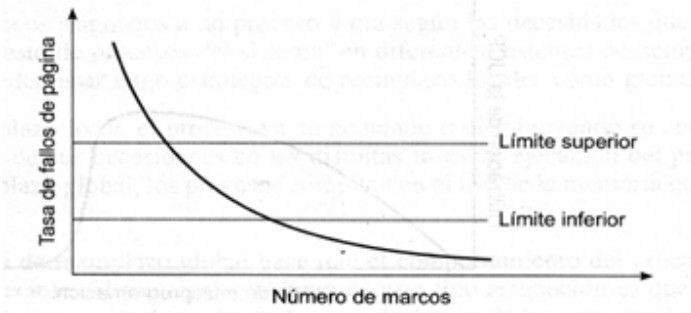


Figura 4.21. *Estrategia de administración basada en la frecuencia de fallos de página.*

Estrategia de control de carga para algoritmos de reemplazo globales

Los algoritmos de reemplazo globales no controlan la hiperpaginación. Necesitan trabajar conjuntamente con un algoritmo de control de carga. A continuación, como ejemplo, se describe la gestión de memoria en el sistema operativo UNIX BSD.

El sistema usa la técnica del *buffering* de páginas, manteniéndose un conjunto de marcos de página libres. Existe un proceso (*page daemon)* que se despierta periódicamente y comprueba si hay suficientes marcos de pagina libres. Si no es así, aplica el algoritmo de reemplazo y libera el número necesario de marcos de página.

La estrategia de reemplazo es global y utiliza una modificación del algoritmo de reloj denominada reloj con dos manecillas, puesto que utiliza dos punteros en vez de uno. Cuando se ejecuta el algoritmo de reemplazo, se desactiva el bit de referencia de la página a la que apunta el primer puntero y se comprueba el bit de referencia de la página a la que señala el segundo. Si está desactivado se libera esa página, escribiéndola previamente al disco si esta modificada, El algoritmo avanza ambos punteros y repite el proceso hasta que se liberan suficientes marcos. Las páginas que están en la lista de páginas libres pueden recuperarse si se referencian antes de ser reutilizadas.

Cuando la tasa de paginación en el sistema es demasiado alta y el número de marcos libres está frecuentemente por debajo del mínimo, otro proceso especial *(swapper)* selecciona uno o más procesos para suspenderlos y liberar sus marcos de página. El *swapper* comprueba periódicamente si alguno de los procesos expulsados debe reactivarse. La reactivación de los procesos seleccionados sólo se llevará a cabo si hay suficientes marcos de página libres.

4.5.8. Gestión del espacio de *swap*

El sistema operativo debe gestionar el espacio de *swap* reservando y liberando zonas del mismo según evolucione el sistema. Existen básicamente dos alternativas a la hora de asignar espacio de *swap* durante la creación de una nueva región:

- **Preasignación de *swap.*** Cuando se crea la nueva región, se reserva espacio de *swap* para la misma. Con esta estrategia, cuando se expulsa una página ya tiene reservado espacio en *swap* para almacenar su contenido.
- **Sin preasignación de *swap.*** Cuando se crea una región, no se hace ninguna reserva en el *swap.* Las páginas de la región se irán trayendo a memoria principal por demanda desde el soporte de la región. Sólo se reserva espacio en el *swap* para una página cuando es expulsada por primera vez.

La tendencia actual es utilizar la segunda estrategia, dado que la primera conlleva un peor aprovechamiento de la memoria secundaria, puesto que toda página debe tener reservado espacio en ella. Sin embargo, la preasignación presenta la ventaja de que con ella se detecta anticipadamente cuándo no hay espacio en *swap.* Si al crear un proceso no hay espacio en *swap,* éste no se crea. Con un esquema sin preasignación, esta situación se detecta cuando se va a expulsar una página y no hay sitio para ella, En ese momento, habría que abortar el proceso, aunque ya había realizado parte de su labor.

Hay que resaltar que, con independencia del esquema usado, una página no tiene que copiarse al dispositivo de *swap* mientras no se modifique. Además, una vez asignado espacio de *wap,* ya sea anticipadamente o en su primera expulsión, la página estará siempre asociada al mismo bloque del dispositivo de *swap.*

Observe que, en el caso de una región compartida con soporte en un archivo (como el código o un archivo proyectado), no se usa espacio de *swap* para almacenarla, sino que se utiliza directamente el archivo que la contiene como almacenamiento secundario. De esta forma, los cambios realizados sobre la región son visibles a toda las aplicaciones que la

comparten.

Algunos sistemas operativos permiten añadir dispositivos de *swap* dinámicamente e incluso usar archivos como soporte del s*wap.* Sin embargo, hay que tener en cuenta que el acceso a los archivos es más lento que el acceso a los dispositivos. Esta posibilidad es muy interesante ya que alivia al administrador de la responsabilidad de configurar correctamente a *priori* el dispositivo *swap,* puesto que si hay necesidad, se puede añadir mas espacio de *swap* en tiempo de ejecución.

4.5.9. Operaciones *sobre* las regiones de un proceso

A continuación se analiza cómo se realizan las diversas operaciones sobre las regiones en un sistema con memoria virtual.

Creación de una región

Cuando se crea una región, no se le asigna memoria principal puesto que se cargará por demanda. Todas las páginas de la región se marcan como no residentes, o sea, invalidas pero válidas para sistema operativo. De esta forma, el primer acceso causará un fallo de página.

El sistema operativo actualiza la tabla de regiones para reflejar la existencia de la nueva región y guarda la información de las características de las páginas de la región rellenando las entradas de la tabla de páginas de acuerdo a las mismas. Algunas de estas características son relevantes a MMU, como por ejemplo la protección. Sin embargo, otras solo le conciernen al sistema operativo como por ejemplo si las páginas de la región son privadas o compartidas.

Las páginas de una región con soporte en un archivo (p. ej.: las de código o las correspondientes a la región de datos con valor inicial) se marcan para indicar esta situación (**bit de cargar de archivo**), almacenándose también la dirección del bloque correspondiente del archivo. Las páginas sin soporte (p. ej.: la paginas correspondientes a la región de datos sin valor inicial) se marcan con un valor especial que indica que hay que rellenarlas a cero (**bit de rellenar con ceros**). Observe que un fallo sobre una página de este tipo no provocaría una operación de entrada/salida al disco.

Si en el sistema hay preasignación de *swap* y se trata de una región privada, hay que reservar en el momento de la creación la zona correspondiente del *swap.*

El caso de la creación de la región de pila del proceso es un poco diferente, ya que esta región tiene un contenido previo (los argumentos del programa) que no está almacenado en un archivo. Típicamente, se reserva uno o más bloques en el *swap* y se copia en ellos el contenido inicial de la pila.

Una vez creada una región, el tratamiento que se le da a la página cuando se expulsa y es modificada va a depender de si es privada o compartida:

- Si la región es privada, se escribe en el *swap*. En el caso de un sistema sin preasignación, será necesario reservar espacio en swap cuando se expulsa por primera vez.
- Si la región es compartida, se escribe directamente en el soporte para que todos los procesar puedan ver las modificaciones.

En la creación de la imagen inicial del proceso, se crean todas las regiones iniciales siguiendo el procedimiento que se acaba de describir y se marcan los huecos como páginas inválidas, tanto para el hardware como para el sistema operativo. En la Figura 4.22 se muestra cómo es la imagen de memoria en un sistema sin preasignación de *swap.*

Liberación de una región

Cuando se libera una región, se debe actualizar la tabla de regiones para reflejar este

cambio. Las páginas asociadas a la región hay que marcarlas como inválidas tanto para la

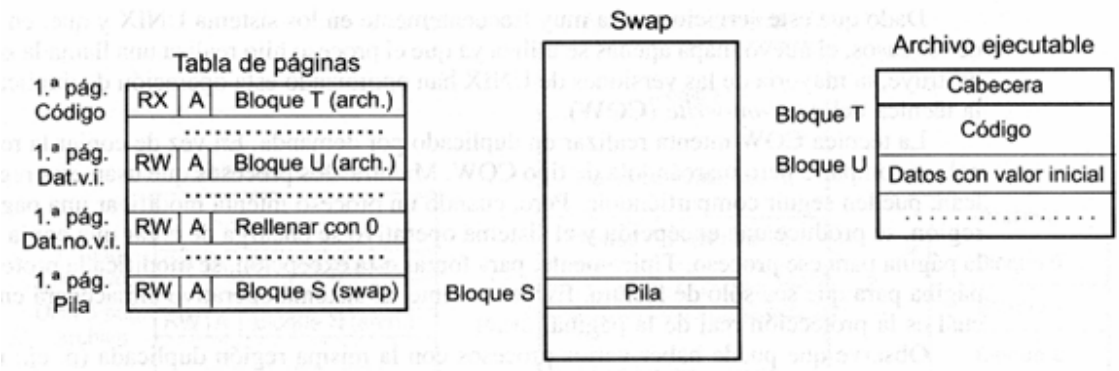


Figura 4.22. *Estado inicial de ejecución en un sistema sin preasignación de swap.*

MMU como para el sistema operativo. Además, si se trata de una región privada, se libera el espacio de swap asociada a la misma.

La liberación puede deberse a una solicitud explícita (como ocurre cuando se desproyecta un de archivo) o a la finalización del proceso que conlleva la liberación de todas sus regiones. Observe que en POSIX, el servicio exec también produce una liberación del mapa de un proceso.

Cambio del tamaño de una región

Con respecto a una disminución de tamaño en una región, esta operación implica una serie de acciones similares a la liberación, pero que solo afectan a la zona liberada. Por lo que se refiere a un aumento de tamaño, hay que comprobar que la región no se solapa con otra región y establecer las nuevas páginas como no residentes y con las mismas características que el resto de las páginas de la región.

En el caso de una expansión del heap, se marcan las nuevas paginas como de lectura y escritura, de carácter privado y a rellenar con ceros.

El tratamiento de la expansión de la pila es algo mas complejo, ya que no proviene de una solicitud del proceso, sino de la propia evolución de la pila. Observe que, cuando se produce una expansión de pila, se genera un fallo de página asociado a una dirección que se corresponde con un hueco. El sistema operativo podría pensar, en principio, que se trata de un acceso erróneo. Para diferenciarlo, debe comparar la dirección que causó el fallo con el puntero de pila. Si la dirección es mayor, está dentro de la pila. Se trata de una expansión de la pila, que implica marcar adecuadamente las páginas involucradas en la expansión. En caso contrario, se trata de un acceso erróneo.

Duplicado de una región

Esta operación está asociada al servicio fork de POSIX y no se requiere en sistemas operativos que tengan un modelo de creación de procesos más convencional.

Cuando se produce una llamada a este servicio, se debe crear un nuevo proceso que sea un duplicado del proceso que la invoca. Ambos procesos compartirán las regiones de carácter compartido que hay en el mapa del proceso original (como, por ejemplo, las regiones de código, los archivos proyectados o las zonas de memoria compartida). Sin embargo, las regiones de carácter privado (como, por ejemplo, las regiones de datos con valor inicial, de datos sin valor inicial, la región de heap y la pila) deben ser un duplicado de las regiones originales. Esta operación de duplicado es costosa, ya que implica crear una nueva región y copiar su contenido desde la región original.

Dado que este servicio se usa muy frecuentemente en los sistema UNIX y que, en la mayoría de los casos, el nuevo mapa apenas se utiliza ya que el proceso hijo realiza una llamada exec que lo destruye, la mayoría de las versiones de UNIX han optimizado esta operación de duplicado usando la técnica del copy-on-write (**COW**).

La técnica COW intenta realizar un duplicado por demanda. En vez de copiar la región original, se comparte pero marcándola de tipo COW. Mientras los procesos que usan esta región solo la lean, pueden seguir compartiéndola. Pero, cuando un proceso intenta modificar una página de esta región, se produce una excepción y el sistema operativo se encarga de crear una copia privada de la página para ese proceso. Típicamente, para forzar esta excepción, se modifica la protección de la página para que sea sólo de lectura. Evidentemente, el sistema operativo almacenará en sus tablas cuál es la protección real de la pagina.

Observe que puede haber y varios procesos con la misma región duplicada (p. ej.: un proceso que ha creado tres hijos). Por tanto, asociado a cada página de tipo COW hay un contador que indica cuántos procesos están compartiéndola. Cada vez que se produce una excepción por acceso de escritura a una página de este tipo, el sistema operativo, además de crear una copia privada de la página para el proceso, decrementa el contador de uso. Cuando el contador indique que sólo hay un proceso asociado a la página, se puede quitar la marca de COW ya que la página no tiene duplicados.

Como resultado de aplicar la técnica COW, se optimiza considerablemente la ejecución de un servicio fork, ya que solo es necesario compartir las regiones del padre, o sea, duplicar su tabla de páginas. Las regiones de carácter compartido son realmente compartidas, pero en las de tipo privado se trata de un falso compartimiento mediante la técnica COW.

4.6. ARCHIVOS PROYECTADOS EN MEMORIA

La generalización de la técnica de memoria virtual permite ofrecer a los usuarios una forma alternativa de acceder a los archivos. Como se ha analizado en la sección anterior, en un sistema de memoria virtual se hacen corresponder directamente entradas de la tabla de páginas con bloques del archivo ejecutable. La técnica de proyección de archivos en memoria plantea usar esa misma idea, pero aplicada a cualquier archivo. -El sistema operativo va a permitir que un programa solicite que se haga corresponder una zona de su mapa de memoria con los bloques de un archivo cualquiera, ya sea completo o una parte del mismo. En la solicitud, el programa especifica el tipo de protección asociada a la región.

Como se puede observar en la Figura 4.23, el sistema operativo deberá manipular la tabla de paginas del proceso para que se corresponda con los bloques del archivo proyectado.

Una vez que el archivo está proyectado, si el programa accede a una dirección de memoria perteneciente a la región asociada al archivo, estará accediendo al archivo. El programa ya no tiene que usar los servicios del sistema operativo para leer (read) y escribir (write) en el archivo. Así si en el ejemplo de la figura el programa lee un byte de la dirección de memoria 10240, estará leyendo el primer byte del archivo, mientras que, si escribe un byte en la dirección 10241, estará escribiendo en el segundo byte del archivo.

Observe que el proyectar un archivo no implica que se le asigne memoria principal. Simplemente, se actualizan las entradas de la tabla de páginas para que referencien al archivo. El propio mecanismo de memoria virtual será el que se encargue de ir trayendo a memoria principal los bloques del archivo cuando se produzca un fallo de pagina al intentar acceder a la región asociada al mismo y de escribirlos cuando la pagina sea

expulsada estando modificada.

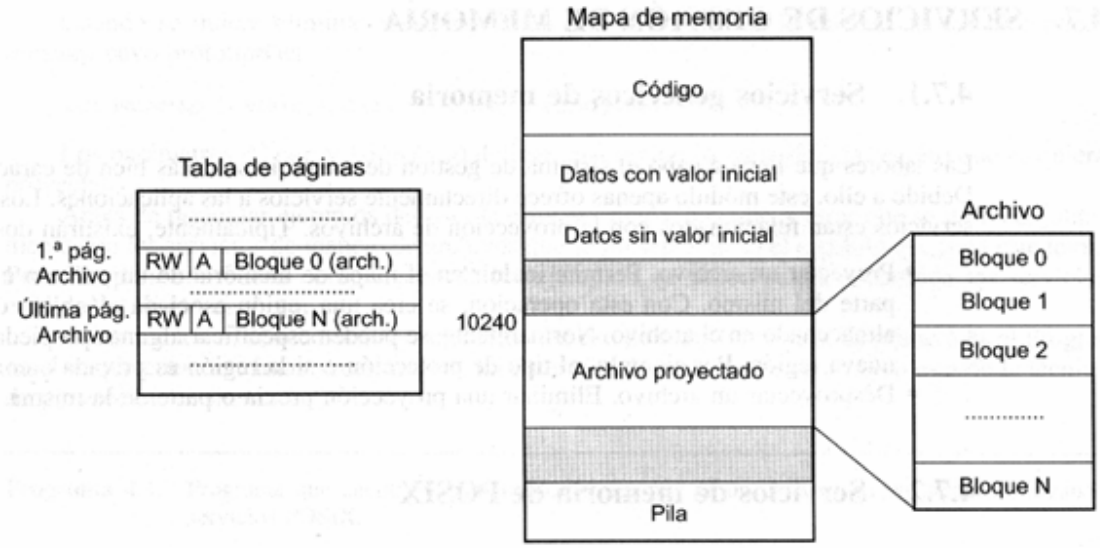


Figura 4.23. *Proyección de un archivo.*

El acceso a un archivo mediante su proyección en memoria presenta numerosas ventajas sobre el acceso convencional basado en los servicios de lectura y escritura. A continuación, se detallan algunas de estas ventajas:

- Se disminuye considerablemente el numero de llamada al sistema necesarias para acceder a un archivo. Con esta nueva técnica, una vez que el archivo está proyectado, no es necesario realizar ninguna llamada adicional. Esta reducción implica una mejora considerable en los tiempos de acceso puesto que, como ya es conocido, la activación de una llamada al sistema tiene asociada una considerable sobrecarga.
- Se evitan copias intermedias de la información. Esto repercute también en un acceso más eficiente. Con esta técnica, el sistema operativo transfiere directamente la información entre la región de memoria y el archivo. Con la forma de acceso convencional, todas las transferencias se realizan pasando por un almacenamiento intermedio del sistema de archivos.
- Se facilita la forma de programar los accesos a los archivos. Una vez proyectado el archivo, este se accede de la misma forma que cualquier estructura de datos en memoria que haya declarado el programa. No es necesario utilizar ninguna primitiva especial del sistema operativo para acceder al mismo. Así, por ejemplo, dado un programa que realiza un cierto procesamiento sobre una matriz de datos almacenada en memoria, su modificación para que leyera la matriz de un archivo sólo implicaría añadir al principio del programa la proyección del archivo. No sería necesario modificar el código restante.

En algunos sistemas se permite también establecer proyecciones en las que la región se marca como privada. Con este tipo de proyección, los cambios realizados en la región no afectan al archivo, sino que se creará una copia privada que estará vinculada al dispositivo de swap. Típicamente, las bibliotecas dinámicas se cargan usando este tipo de proyección sobre las secciones del ejecutable que tienen carácter privado.

4.7. SERVICIOS DE GESTIÓN DE MEMORIA

4.7.1. Servicios genéricos de memoria

Las labores que lleva a cabo el sistema de gestión de memoria son más bien de carácter interno. Debido a ello, este modulo apenas ofrece directamente servicios a las aplicaciones. Los principales servicios están relacionados con la proyección de archivos. Típicamente, existirán dos servicios:

- Proyectar un archivo. Permite incluir en el mapa de memoria de un proceso un archivo o parte del mismo. Con esta operación, se crea una región asociada al objeto de memoria almacenado en el archivo. Normalmente, se pueden especificar algunas propiedades de esta nueva región. Por ejemplo, el tipo de protección o si la región es privada o compartida.
- Desproyectar un archivo. Eliminar una proyección previa o parte de la misma.

4.7.2. Servicios de memoria de POSIX

El estándar POSIX define un relativamente pequeño conjunto de servicios de gestión de memoria. Los servicios de gestión de memoria más frecuentemente usados son los que corresponden con la proyección y desproyección de archivos (mmap, munmap). El servicio mmap tiene el siguiente prototipo:

```
caddrt  mmap (caddr_t direc, size_t longitud, int protec,
                    int indicador, int descriptor, off_t  despl);
```

El primer parámetro indica la dirección del mapa donde se quiere que se proyecte el archivo Generalmente, se especifica un valor nulo para indicar que se prefiere que sea el sistema el que decida dónde proyectar el archivo. En cualquier caso, la función devolverá la dirección de proyección utilizada.

El parámetro descriptor se corresponde con el descriptor del archivo que se pretende proyectar (que debe estar previamente abierto) y los parámetros despl y longitud establecen que zona del archivo se proyecta: desde la posición despl hasta desp + longitud.

El argumento protec establece la protección sobre la región que puede ser de lectura (PROT_READ), de escritura (PROT_WRITE), de ejecución (PROT_EXEC) o cualquier combinación ellas. Esta protección debe ser compatible con el modo de apertura del archivo. Por último, el parámetro indicador permite establecer ciertas propiedades en la región:

- MAP_SHARED. La región es compartida. Las modificaciones sobre la región afectarán al archivo. Un proceso hijo compartirá esta región con el padre.
- MAP_PRIVATE. La región es privada. Las modificaciones sobre la región no afectaran al archivo. Un proceso hijo no compartirá esta región con el padre, sino que obtendrá un duplicado de la misma.
- MAP_FIXED. El archivo debe proyectarse justo en la dirección especificada en el primer parámetro, siempre que éste sea distinto de cero.

En el caso de que se quiera proyectar una región sin soporte (región anónima), en algunos sistemas se puede especificar el valor MAP_ANOM en el parámetro indicador. Otros sistemas UNIX ofrecen esta opción, pero permiten proyectar el dispositivo /dev/ zero para lograr el mismo objetivo, Esta opción se puede usar para cargar la región de

datos sin valor inicia de una biblioteca dinámica.

Cuando se quiere eliminar una proyección previa o parte de la misma, se usa el servicio munmap cuyo prototipo es:

int **munmap** (caddr_t direc, size_t longitud);

Los parámetros direc y longitud definen una región (o parte de una región) que se quiere proyectar.

Antes de presentar ejemplos del uso de estos servicios, hay que aclarar que se usan conjuntamente con los servicios de manejo de archivos que se presentarán en el capítulo que trata este tema. Por ello, para una buena comprensión de los ejemplos, se deben estudiar también los servicios explicados en ese capítulo.

A continuación, se muestran dos ejemplos del uso de estas funciones. El primero es el Programa 4.4, que cuenta cuantas veces aparece un determinado carácter en un archivo usando la técnica de proyección en memoria.

Programa 4.4. Programa que cuenta el número de apariciones de un carácter en un archivo utilizando servicios POSIX.

```
#include<sys/types .h>
#include <sys/stat .h>
#include<sys/rnman ,h>
#include<sys/rnman ,h>
#include <fcntl .h>
#include<stdio.h
#include <unistd .h>.

int  main(int argc, char **argv) {
     int i , fd , contador=0;
     char caracter;
     char *org, *p;
      struct stat bstat;

      if (argc!=3) {
         fprintf (stderr, "Uso: %s caracter archivo\n", argv[0]);
          return(l);
       }
        /* Por simplicidad , se supone que el carácter a contar corresponde con el
           primero                    del primer argumento */
         carácter = argv [1] [0];

        /* Abre el archivo para lectura */
         if ((fd=open(argvL2], O_RDONLY))<0) {
         perror ("No puede abrirse el archivo");
         return(l);
        }
/* Averigua la longitud del archivo */
 if (fstat(fd, &bstat)<0)
    perror("Error en fstat del archivo");
    close (fd);
     return(l);
```

```
}

/* Se proyecta el archivo */
if ((org=mmap((caddr_t) 0, bstat.st_size, PROT_READ,
                                    MAP_SHARED, fd, 0)) = = MAP_FAILED) {
     perror("Error en la proyección del archivo");
     close(fd);
      return(l);
 }
/* Se cierra el archivo */
close(fd);

/* Bucle de acceso */
p=org;
for (i=0; i<bstat,st_size; i++)
      if (*p ++ = = carácter ) contador + +;

/* Se elimina la proyección */
munmap(org, , bstat.st_size);

printf ("%d \n", contador);
return(0);
}
```

El segundo ejemplo corresponde con el Programa 4.5, que usa la técnica de proyección para realizar la copia de un archivo. Observe el uso del servicio ftruncate para asignar espacio al archivo destino.

Programa 4.5. Programa que copia un archivo utilizando servicios POSIX.

```
#include <sys/types .h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

void main(int argc. char **argv) {
int i, fdo, fdd;
char *org, , *dst, *p, *q;
struct stat bstat;
if (argc!=3) {
    fprintf (stderr, "Uso: %s orig dest\n". argv[0]);
   exit(l);
}
/* Abre el archivo origen para lectura */
if ((fdo=open(argv[l], O_RDONLY))<O) {
     perror ("No puede abrirse el archivo origen");
     exit (l);
```

```
}
/* Crea el archivo destino */
if ((fdd=open(argv[2]. O_CREAT | O_TRUNC | O_RDWR. O640))<O) {
     perror("No puede crearse el archivo destino");
    close(fdo);
    exit (1)
      }
/* Averigua la longitud del archivo origen */
if (fstat(fdo, &bstat)<0)
{ perror("Error en fstat del archivo origen");
close(fdo);
close(fdd);
unlink (argv[2]);
exit(l);
}

/* Establece que la longitud del archivo destino es del origen,*/
if (ftruncate(fdd, bstat.st_size)<0) {
    perror("Error en ftruncate del archivo destino"); close(f do);
close (fdd);
unlink (argv[2]); exit(l);
}
/* Se proyecta el archivo origen */
if ((org=mmap((caddrt) O, bstat.st_size, PROTREAD,
                                MAP_SHARED, fdo, 0)) = = MAP_FAILED) {
   perror("Error n la proyección del archivo origen"); close(fdo);
   close(fdd);
   unlink (argv[2]);
   exit(l);
}

/* Se proyecta el archivo destino */
if ((dst=nunap((caddrt) , bstat.st_size, PROTWRITE,
                              MAP_SHARED, fdd, O)) = = MAP_FAILED) {
   perror("Error en la proyección del archivo destino");
 close(fdo);
close(fdd);
unlink (argv[2]);
exit(l);
}

/* Se cierran los archivos */
close(fdo);
close(fdd);

/* Bucle de copia  */
p = org; q = dst;
for (i=0; i<bstat.st_size; i++)
     *q++ = *p++ ;
```

```
/* Se eliminan las proyecciones */
munmap(org, bstaLst_size);
munmap(dst, bstat.stsize);
}
```

4.7.3. Servicios de memoria de Win32

Los servicios de memoria más utilizados son, nuevamente, los de proyección de archivos. A diferencia de POSIX, la proyección de un archivo se realiza en dos pasos. En primer lugar, hay que crear una proyección del chivo usando la primitiva CreateFileMapping:

```
HANDLE CreateFi1eMapping (HANDLE archivo,
        LPSECURITY_ATTRIBUTES segur, DWORD prot, DWORD
        tamanyo_max_alta , DWORD tamanyo_max_baja, LPCTSTR
        nombre_proy);
```

Esta función devuelve un identificador de la proyección y recibe como parámetros el nombre del archivo, un valor de los atributos de seguridad, la protección, el tamaño del objeto a proyectar (especificando la parte alta y la parte baja de este valor en dos parámetros independientes) y un nombre para la proyección.

En cuanto a la protección, puede especificarse de sólo lectura (PAGE READONLY), de lectura y escritura (PAGE_READWRITE) o privada (PAGE_WRITECOPY).

Con respecto al tamaño, en el caso de que el archivo pueda crecer, se debe especificar el tamaño esperado para el chivo. Si se especifica un valor 0, se usa el tamaño actual del archivo.

Por último, el nombre de la proyección permite a otros procesos acceder a la misma. Si se especifica un valor nulo, no se asigna nombre a la proyección.

Una vez creada la proyección, se debe crear una región en el proceso asociada a la misma. Esta operación se realiza mediante la función MapViewOf File cuyo prototipo es el siguiente:

```
LPVOID MapViewOfFile (HANDLE id_proy, DWORD acceso, DWORD
                      desp_alta, DWORD desp_baj a, DWORD tamanyo);
```

Esta función devuelve la dirección del mapa donde se ha proyectado la región. Recibe como parámetros el identificador de la proyección devuelto por CreateFileMapping, el acceso solicitado (FILE_MAP_WRITE, FILE_MAP_READ y FILE_MAP_ALL_ACCESS), que debe ser compatible con la protección especificada en la creación, el desplazamiento con respecto al inicio del chivo a partir del que se realiza la proyección y el tamaño de la zona proyectada (el valor cero indica todo el archivo).

Por último, para desproyectar el archivo se usa UnmapViewOfFile:

```
BOOL UnmapViewof File (LPVOID dir);
```

donde el parámetro indica la dirección de comienzo de la región que se quiere desproyectar.

Como se comentó en la sección que presentaba los servicios POSIX, es necesario conocer los servicios básicos de archivos para entender completamente los siguientes programas.

A continuación, se muestran los dos mismos ejemplos que se plantearon en la sección dedicada a POSIX. El primero es el Programa 4.6, que cuenta cuántas veces aparece un

determinado carácter en un archivo usando la técnica de proyección en memoria.

Programa 4.6. Programa que cuenta el número de apariciones de un carácter en un archivo utilizando servicios Win32.

```
#include <windows .
#include <stdio.h>

int main (mt argc, LPTSTR argv [])
{
HANDLE hArch, hProy;
LPSTR base, puntero;
DWORD tam;
int contador = 0;
char caracter;

if (argc!=3){
   fprintf (stderr, "Uso: %s carácter archivo\n", argv[0]);
   return(l);
}

/*Por simplicidad, se supone que el carácter a contar corresponde con el primero del primer argumento */
carácter =argv[l] [0];
/* Abre el archivo para lectura */          -
hArch = CreateFile (argv[2], GENERIC_READ, O, NULL,
                          OPEN_EXISTIMO, FILEATTRIBUTE_NORMAL, NULL);
if (hArch = = INVALID_HANDLE_VALUE) {
   fprintf(stderr, "No puede abrirse el archivo\n");
   return(l);
}
/* se crea la proyección del archivo */
hProy = CreateFileMapping (hArch, NTJLL, PAGE_READONLY, O, O, NULL);
 if (hProy = = INVALID_HANDLE_VALUE) {
fprintf(stderr, "No puede crearse la proyección \n");
return(l);
}
/* se realiza la proyección */
base = MapViewOfFile (hProy, FILE_MAP_READ, 0, 0, 0);
tam = GetFileSize (hArch, NULL);
/* bucle de acceso */
 puntero = bas
while (puntero < base + tam)
        if (*puntero++= =caracter) contador++;
          printf("%d\n", contador);
/* se elimina la proyección y se cierra el archivo */
UnmapViewOfFile (base);
CloseHandle (hProy);
Closehandle (hArch);
 return 0;
```

```
}
```

El segundo ejemplo corresponde con el Programa 4.7, que usa la técnica de proyección realizar la copia de un archivo.

Programa 4.7. Programa que copia un archivo utilizando servicios Win32.

```
#include <windows .
#include <stdio,h>

int main (int argc, LPTSTR argv [])
{

    HANDLE hEnt, hSal;
    HANDLE hProyEnt, hProySal;
    LPSTR base_orig, puntero_orig;
    LPSTR base_dest, punterodest;
    DWORD tam;

    if (argc!=3) {
    fprintf (stderr, "Uso: %s origen destino \n", argv[0]);
    return(l);
    }
    /* se abre archivo origen */
    hEnt = CreateFile (argv[l], GENERIC_READ,0, NULL, OPEN_EXISTING,
                       FILE_ATTRIBUTE_NORMAL, NULL);
    if (hEnt = = INVALID_HANDLE_VALUE) {
       fprintf(stderr, "No puede abrirse el archivo origen \n");
    return(l);
    }

    /* se crea proyección de archivo origen */
    hProyEnt = CreateFi1eMapping (hEnt, NULL, PACE_READONLY, 0, 0, NULL);
    if (hProyEnt = = INVALID_HANDLE VALUE) {
       fprintf(stderr, "No puede crearse proyección del archivo origen \n);
       return(l);
    }
    /* se proyecta archivo origen */
     base_orig = MapViewOfFile (hProyEnt, FILE_MAP_READ, 0, 0, 0);
    tam = GetFileSize (hEnt, NULL);
    /* se crea el archivo destino */
    hSal = CreateFile (argv[2], GENERIC_REAl) | GENERIC_WRITE,
                  0, NULL, CREATE_ALWAYS, FILE ATTRIBUTE NORMAL, NULL);
    if (hSal = = INVALID_HANDLE_VALUE) {
       fprintf(stderr, "No puede crearse el archivo destino \n");
       return(l);
       }

    /* se crea proyección de archivo destino */
    hProySal = CreateFileMapping (hSal, NULL, PACE_READWRITE, O, tam, NULL);
```

```
if (hProySal == INVALID_HAJ\JDLE_VALUE) { fprintf(stderr, "No puede
crearse proyección return(l);
1
del archivo destino");

7* se proyecta archivo destino *7
base_dest = MapViewOfFile (hProySal, FILE_MAP_WRITE, O,


/~ bucle de copia *7 puntero_orig = baseorig; puntero_dest = baseAest;
for ( ; puntero_orig < baseorig + tam; puntero_orig++, puntero_dest++)
*puntero_dest = *punteroorig;


/* se eliminan proyecciones y se cierran
UxuuapViewOfFile (base_dest);
UnmapViewOfFile (base_orig);
C1oseI~and1e (hProyEnt);
CloseHaudle (hEnt>;
CloseHandle (hProySal);
CloseHandle (hSal);
return O;
archivos *7
```

4.8. PUNTOS A RECORDAR

- El sistema de memoria debe ofrecer a cada proceso un espacio lógico propio.
- El sistema de memoria debe proporcionar protección entre los procesos.
- El sistema de memoria debe permitir que los procesos compartan memoria.
- El sistema de memoria debe dar soporte a las regiones del proceso.
- El sistema de memoria debe maximizar el rendimiento del sistema.
- El sistema de memoria debe proporcionar a los procesos mapas de memoria muy grandes.
- Las bibliotecas dinámicas ofrecen múltiples ventajas con respecto a las estáticas. Entre otras, disminuyen el tamaño del ejecutable y permiten una actualización dinámica.
- Un ejecutable está formado por una cabecera y un conjunto de secciones. Las secciones principales son el código, datos con valor inicial y sin valor inicial.
- Las variables locales y parámetros no aparecen en el ejecutable, ya que se crean en tiempo de ejecución.
- El mapa de memoria está formado por varias regiones. Las regiones iniciales se corresponde con las secciones del ejecutable más la pila inicial.
- Otras regiones se crean posteriormente de forma dinámica como el *heap*, las zonas de memoria compartida y los archivos proyectados.
- Una región se caracteriza por su tamaño, soporte, protección, carácter privado o compartido y longitud fija o variable.
- Se identifican las siguientes operaciones sobre una región: creación, liberación, cambio de tamaño y duplicado.
- Los esquemas de memoria más primitivos son los de asignación contigua que requieren que la MMU tenga registros valla.
- En los esquemas contiguos se aplica una estrategia de asignación de espacio. Las más típicas son: *best-fit*, *worst-fit* y *first-fit* (la más adecuada).
- El esquema de asignación contigua no da soporte a las regiones, no permite compartir y tiene un mal aprovechamiento de la memoria (fragmentación externa).

- El intercambio es una técnica predecesora de la memoria virtual que permite que haya más procesos de los que caben en memoria usando un disco (dispositivo de *swap*) como respaldo.
- La paginación considera el mapa del proceso dividido en páginas y la memoria principal en marcos del mismo tamaño. La tabla de páginas hace corresponder una página con el marco que la contiene.
- La paginación sufre fragmentación interna.
- La entrada de la tabla de páginas contiene información de validez, de protección, de acceso y el número de marco asociado.
- La paginación cumple todos los objetivos deseables en un esquema de memoria. Además, sirve como base para construir un sistema de memoria virtual.
- Para implementar eficientemente la paginación es necesario usar una TLB.
- Para reducir los gastos de memoria en tablas de página se pueden usar tablas de páginas multinivel y tablas invertidas.
- La segmentación considera el mapa del proceso dividido en segmentos continuos. No es adecuada por sufrir fragmentación externa y no permitir implementar eficientemente memoria virtual.
- La segmentación paginada combina las ventajas de ambos esquemas.
- La memoria virtual se construye sobre esquemas basados en paginación por demanda. Las páginas no residentes se marcan como inválidas para forzar el fallo de página.
- El tratamiento de un fallo de página implica la lectura de la página en un marco libre. Si no hay ninguno, hay que aplicar antes un reemplazo que puede requerir una escritura al disco si la página expulsada estaba modificada.
- La administración de la memoria virtual conlleva una política de reemplazo y una asignación de espacio a los procesos.
- Las políticas de reemplazo pueden ser locales o globales.
- La política de reemplazo LRU es la más adecuada pero es difícil de implementar sin un hardware específico.
- La política de reemplazo de la segunda oportunidad proporciona una solución con un rendimiento aceptable sin requerir un hardware específico.
- No todas las páginas son objeto de reemplazo. Algunas quedan retenidas en memoria.
- La política de asignación de marcos a los procesos puede ser fija o dinámica.
- Las políticas de asignación fija tienen asociados algoritmos de reemplazo locales. No se adaptan al comportamiento dinámico del proceso.
- Las políticas de asignación dinámica pueden usarse con reemplazos locales o globales.
- La hiperpaginación se produce cuando hay una elevada tasa de fallos de página.
- Con asignación fija, la hiperpaginación sólo afecta al proceso implicado. Con asignación dinámica, afecta a todo el sistema y hace falta disminuir el grado de multiprogramación.
- Para evitar la hiperpaginación puede usarse la estrategia del conjunto de trabajo, la estrategia basada en la frecuencia de fallos de página o algoritmos de control de carga específicos.
- Existen dos estrategias a la hora de asignar espacio de *swap*: asignación cuando se crea la región (preasignación) o asignación en la primera expulsión.
- Cuando se crea una región en un sistema de memoria virtual, se marcan sus páginas como no residentes y se almacenan sus propiedades.
- Las regiones privadas se expulsan a *swap*, mientras que las compartidas se escriben sobre el propio soporte.
- El aumento de tamaño de una región debe controlarse para evitar solapamiento.
- El duplicado de una región requerido por el servicio `fork` de POSIX se optimiza mediante la técnica del *copy-on-write*.
- El mecanismo de proyección de archivos en memoria proporciona una manera alternativa de acceder a los archivos con numerosas ventajas sobre la forma tradicional.
- Los principales servicios de gestión de memoria son los relacionados con la proyección de archivos en memoria.

4.9. LECTURAS RECOMENDADAS

Todos los libros generales de sistemas operativos incluyen uno o más capítulos dedicados a la gestión de memoria. Algunos de ellos dedican un capítulo a los esquemas sin memoria virtual y otro a los esquemas con memoria virtual ([Silberschatz, 1998] y [Stallings, 1998]). Sin embargo, otros libros dedican un solo capítulo a este tema [Tanenbaum, 1992].

Hay otros libros que estudian cómo se implementa la gestión de memoria en UNIX System V [Bach, 1986], en UNIX BSD [McKusick, 1998], en Linux [Beck, 1997], en Windows-NT [Custer, 1993] o en MINIX [Tanenbaum, 1997].

En cuanto a los servicios de gestión de memoria, en [Stevens, 1992] se presentan los de POSIX y en [Hart, 1997] los de Win32. Para un estudio de los aspectos más formales de la memoria virtual, se recomienda [Maekawa, 1987].

4.10. EJERCICIOS

4.1. Sea un sistema de paginación con un tamaño de página `P`. Especifique cuál sería la fórmula matemática que determina la dirección de memoria física `F` a partir de la dirección virtual `D`, siendo la función `MARCO(X)` una que devuelve el número de marco almacenado en la entrada `X` de la tabla de páginas.

4.2. ¿Es siempre el algoritmo de reemplazo LRU mejor que el FIFO? En caso afirmativo, plantee una demostración. En caso negativo, proponga un contraejemplo.

4.3. ¿Cuál de las siguientes técnicas favorece la proximidad de referencias?

— Un programa multiflujo.
— El uso de listas.
— La programación funcional.
— La programación estructurada.

4.4. El lenguaje C define el calificador `volatile` aplicable a variables. La misión de este calificador es evitar problemas de coherencia en aquellas variables que se acceden tanto desde el flujo de ejecución normal como desde flujos asíncronos, como por ejemplo una rutina asociada a una señal POSIX. Analice qué tipo de problemas podrían aparecer y proponga un método para resolver los problemas identificados para las variables etiquetadas con este calificador.

4.5. Algunas MMU no proporcionan un bit de página accedida. Proponga una manera de simularlo. Una pista: se pueden forzar fallos de página para detectar accesos a una página.

4.6. Algunas MMU no proporcionan un bit de página modificada. Proponga una manera de simularlo.

4.7. Escriba un programa que use los servicios POSIX de proyección de archivos para comparar dos archivos.

4.8. Escriba un programa que use los servicios Win32 de proyección de archivos para comparar dos archivos.

4.9. ¿Por qué una cache que se accede con direcciones virtuales puede producir incoherencias y requiere que el sistema operativo la invalide en cada cambio de proceso y, en cambio, una que se accede con direcciones físicas no lo requiere?

4.10. ¿Por qué una cache que se accede con direcciones virtuales permite que el acceso a la TLB y a la cache se hagan en paralelo y, en cambio, una que se accede con direcciones físicas no lo permite? ¿Por qué es conveniente que las caches que se acceden con direcciones físicas tengan el mismo tamaño que la página?

4.11. La secuencia que se utiliza típicamente como ejemplo de la anomalía de Belady es la siguiente:

```
1 2 3 4 1 2 5 1 2 3 4 5
```

Analice cuántos fallos de página se producen al usar el algoritmo FIFO teniendo 3 marcos y cuántos con 4 marcos. Compárelo con el algoritmo LRU. ¿Qué caracteriza a los algoritmos de reemplazo de pila?

4.12. Exponga uno o más ejemplos de las siguientes situaciones:

— Fallo de página que no implica operaciones de entrada/salida.
— Fallo de página que implica sólo una operación de lectura.
— Fallo de página que implica sólo una operación de escritura.
— Fallo de página que implica una operación de lectura y una de escritura.

4.13. Sea un sistema de memoria virtual sin *buffering* de páginas. Analice la evolución de una página en este sistema dependiendo de la región a la que pertenece. Estudie los siguientes tipos:

— Página de código.
— Página de datos con valor inicial.
— Página de datos sin valor inicial.
— Página de un archivo proyectado.
— Página de una zona de memoria compartida.

4.14. Resuelva el ejercicio anterior suponiendo que hay *buffering* de páginas.

4.15. Consulte las páginas de manual de los servicios `dlopen` y `dlsym` disponibles en algunos sistemas UNIX que permiten solicitar la carga de una biblioteca dinámica y acceder a un símbolo de la misma. Escriba un programa que use esta función para cargar una biblioteca dinámica y ejecutar una función de la misma.

4.16. Consulte el manual de los servicios `LoadLibrary` y `GetProcAddress` disponibles en

En cuanto a los servicios de gestión de memoria, en [Stevens, 1992] se presentan los de POSIX y en [Hart, 1997] los de Win32. Para un estudio de los aspectos más formales de la memoria virtual, se recomienda [Maekawa, 1987].

4.10. EJERCICIOS

4.1. Sea un sistema de paginación con un tamaño de página `P`. Especifique cuál sería la fórmula matemática que determina la dirección de memoria física `F` a partir de la dirección virtual `D`, siendo la función `MARCO(X)` una que devuelve el número de marco almacenado en la entrada `X` de la tabla de páginas.

4.2. ¿Es siempre el algoritmo de reemplazo LRU mejor que el FIFO? En caso afirmativo, plantee una demostración. En caso negativo, proponga un contraejemplo.

4.3. ¿Cuál de las siguientes técnicas favorece la proximidad de referencias?

— Un programa multiflujo.
— El uso de listas.
— La programación funcional.
— La programación estructurada.

4.4. El lenguaje C define el calificador `volatile` aplicable a variables. La misión de este calificador es evitar problemas de coherencia en aquellas variables que se acceden tanto desde el flujo de ejecución normal como desde flujos asíncronos, como por ejemplo una rutina asociada a una señal POSIX. Analice qué tipo de problemas podrían aparecer y proponga un método para resolver los problemas identificados para las variables etiquetadas con este calificador.

4.5. Algunas MMU no proporcionan un bit de página accedida. Proponga una manera de simularlo. Una pista: se pueden forzar fallos de página para detectar accesos a una página.

4.6. Algunas MMU no proporcionan un bit de página modificada. Proponga una manera de simularlo.

4.7. Escriba un programa que use los servicios POSIX de proyección de archivos para comparar dos archivos.

4.8. Escriba un programa que use los servicios Win32 de proyección de archivos para comparar dos archivos.

4.9. ¿Por qué una cache que se accede con direcciones virtuales puede producir incoherencias y requiere que el sistema operativo la invalide en cada cambio de proceso y, en cambio, una que se accede con direcciones físicas no lo requiere?

4.10. ¿Por qué una cache que se accede con direcciones virtuales permite que el acceso a la TLB y a la cache se hagan en paralelo y, en cambio, una que se accede con direcciones físicas no lo permite? ¿Por qué es conveniente que las caches que se acceden con direcciones físicas tengan el mismo tamaño que la página?

4.11. La secuencia que se utiliza típicamente como ejemplo de la anomalía de Belady es la siguiente:

```
1 2 3 4 1 2 5 1 2 3 4 5
```

Analice cuántos fallos de página se producen al usar el algoritmo FIFO teniendo 3 marcos y cuántos con 4 marcos. Compárelo con el algoritmo LRU. ¿Qué caracteriza a los algoritmos de reemplazo de pila?

4.12. Exponga uno o más ejemplos de las siguientes situaciones:

— Fallo de página que no implica operaciones de entrada/salida.
— Fallo de página que implica sólo una operación de lectura.
— Fallo de página que implica sólo una operación de escritura.
— Fallo de página que implica una operación de lectura y una de escritura.

4.13. Sea un sistema de memoria virtual sin *buffering* de páginas. Analice la evolución de una página en este sistema dependiendo de la región a la que pertenece. Estudie los siguientes tipos:

— Página de código.
— Página de datos con valor inicial.
— Página de datos sin valor inicial.
— Página de un archivo proyectado.
— Página de una zona de memoria compartida.

4.14. Resuelva el ejercicio anterior suponiendo que hay *buffering* de páginas.

4.15. Consulte las páginas de manual de los servicios `dlopen` y `dlsym` disponibles en algunos sistemas UNIX que permiten solicitar la carga de una biblioteca dinámica y acceder a un símbolo de la misma. Escriba un programa que use esta función para cargar una biblioteca dinámica y ejecutar una función de la misma.

4.16. Consulte el manual de los servicios `LoadLibrary` y `GetProcAddress` disponibles en

Win32 que permiten solicitar la carga de una biblioteca dinámica y acceder a un símbolo de la misma. Escriba un programa que use esta función para cargar una biblioteca dinámica y ejecutar una función de la misma.

4.17. Como se comentó en la explicación del algoritmo de reemplazo LRU, el tiempo que se debe usar para seleccionar la página menos recientemente usada es el tiempo lógico de cada proceso y no el tiempo real. Modifique la implementación basada en contadores propuesta en el texto para que tenga en cuenta esta consideración.

4.18. Un algoritmo de reemplazo no descrito en el texto es el MFU (menos frecuentemente utilizada). Este algoritmo elige para el reemplazo aquella página que se haya utilizado menos frecuentemente. Analice cuál son los puntos fuertes y débiles de este algoritmo y plantee una implementación de este algoritmo.

4.19. Codifique cómo sería la rutina de tratamiento de fallo de página en un sistema con memoria virtual que no usa *buffering* de páginas ni preasignación de *swap*. Tenga en cuenta la influencia de las distintas características de las regiones (p. ej.: si es privada o compartida).

4.20. Resuelva el problema anterior para un sistema con *buffering* de páginas y preasignación de *swap*.

4.21. En la descripción de la técnica COW se explicó que para implementar esta técnica generalmente se pone la página con una protección de sólo lectura. Analice cómo sería la rutina de tratamiento de la excepción que se produce al escribir en una página de este tipo para implementar la técnica COW.

4.22. Muchas implementaciones de UNIX realizan la carga de las bibliotecas dinámicas utilizando el servicio `mmap`. Explique qué parámetros deberían especificarse para cada una de las secciones de una biblioteca.

4.23. Acceda en un sistema Linux al archivo `/proc/self/maps` y analice su contenido.

4.24. En Win32 se pueden crear múltiples *heaps*. Analice en qué situaciones puede ser interesante esta característica.

4.25. Algunas versiones de UNIX proporcionan una llamada denominada `vfork` que crea un hijo que utiliza directamente el mapa de memoria del proceso padre que queda bloqueado hasta que el hijo ejecuta un `exec` o termina. En ese momento el padre recupera su mapa. Analice qué ventajas y desventajas presenta el uso de este nuevo servicio frente a la utilización del `fork` convencional. En este análisis suponga primero que el `fork` se implementa sin usar la técnica COW para a continuación considerar que sí se utiliza.

4.26. ¿Por qué es necesario mantener al menos una página inválida entre la región de pila y la región que está situada justo encima?

4.27. Analice qué puede ocurrir en un sistema que usa paginación por demanda si se recompila un programa mientras se está ejecutando. Proponga soluciones a los problemas que pueden surgir en esta situación.

4.28. En POSIX se define el servicio `msync` que permite forzar la escritura inmediata de una región al soporte. ¿En qué situaciones puede ser interesante usar esta función?

4.29. Cuando se produce un fallo en una página que pertenece a una región compartida, se trae a memoria secundaria la página y se actualiza la entrada de la tabla de páginas del proceso que causó el fallo. ¿Cómo se enteran el resto de los procesos que comparten la página de que ésta ya está en memoria?

4.30. El mecanismo de *buffering* permite recuperar una página que está en la lista de libres ya que todavía no se ha reutilizado el marco que la contiene. ¿Cómo se puede implementar esta búsqueda en la lista para que se haga de forma eficiente?

4.31. Analice qué situaciones se pueden producir en el tratamiento de un fallo de TLB en un sistema que tiene una gestión software de la TLB.

4.32. Con el uso de la técnica de proyección de archivos se produce una cierta unificación entre el sistema de archivos y la gestión de memoria. Puesto que, como se verá en el capítulo dedicado a los archivos, el sistema de archivos usa una cache de bloques con escritura diferida para acelerar el acceso al disco, analice qué tipo de incoherencias pueden producirse si se accede a un archivo usando la proyección y las primitivas convencionales del sistema de archivos.

5

Comunicación y sincronización de procesos

En este capítulo se presentan los problemas que surgen cuando los procesos concurrentes que ejecutan en un sistema compiten por recursos o se comunican entre sí. Para ello, se describen los clásicos problemas de comunicación y sincronización entre procesos. Asimismo se introducen y analizan los principales mecanismos que ofrecen los sistemas operativos para la comunicación y sincronización de procesos y se muestra cómo utilizar éstos para resolver los problemas anteriores. Los temas que se cubren en este capítulo son:

- *Procesos concurrentes.*
- *Problemas clásicos de comunicación y sincronización.*
- *Mecanismos que ofrecen los sistemas operativos para la comunicación y sincronización entre procesos, así como algunos aspectos de su diseño e implementación.*
- *Interbloqueos que pueden aparecer cuando se abordan distintos problemas de comunicación y sincronización de procesos.*
- *Servicios POSIX y Win32 para la comunicación y sincronización entre procesos.*

5.1. PROCESOS CONCURRENTES

En el Capitulo 3 se presentó el concepto de proceso como un programa en ejecución que consta de un espacio de direcciones de memoria y un bloque de control de proceso con diversa información asociada al mismo. También se vio el concepto de proceso ligero como un flujo de ejecución único e independiente dentro de un proceso. Un sistema operativo multitarea permite que coexistan vario procesos activos a la vez, ejecutando todos ellos de forma concurrente. Existen tres modelos de computadora en los que se pueden ejecutar procesos concurrentes:

Multiprogramación con un único procesador

En este modelo todos los procesos concurrentes ejecutan sobre un único procesador. El sistema operativo se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, *intercalando* la ejecución de los mismos p a d así una *apariencia* de ejecución simultánea. En la Figura *5.1* se presenta un ejemplo de ejecución de tres procesos en un sistema multiprogramado con un único procesador. Como puede verse, los tres procesos van avanzando su ejecución de forma aparentemente simultánea, pero sin coincidir en ningún momento sus fases de procesamiento.

Multiprocesador

Como se vio en el Capítulo 1, un multiprocesador es una maquina formada por un conjunto de procesadores que comparten memoria principal. En este tipo de arquitecturas, los procesos concurrentes no sólo pueden intercalar su ejecución sino también superponerla. En este caso sí existe una verdadera ejecución simultánea de procesos, al coincidir las fases de procesamiento de distinto. proceso En un instante dado se pueden ejecutar de forma simultánea tantos procesos como procesadores haya. En la Figura *5.2* se muestra un ejemplo de ejecución de cuatro procesos en un multi-procesador con dos procesadores.

Multicomputadora

Una multicomputadora es una maquina de memoria distribuida, en contraposición con el multiprocesador que es de memoria compartida. Está formada por una serie de computadoras completas con su UCP, memoria principal y, en su caso, periferia. Cada uno de estos procesadores completos se denomina nodo. Los nodos se encuentran conectados y se comunican entre sí a través de una red de interconexión, empleando el método de paso de mensajes, que analizaremos más adelante. En este tipo de arquitecturas también es posible la ejecución simultánea de los procesos sobre los diferentes procesadores.

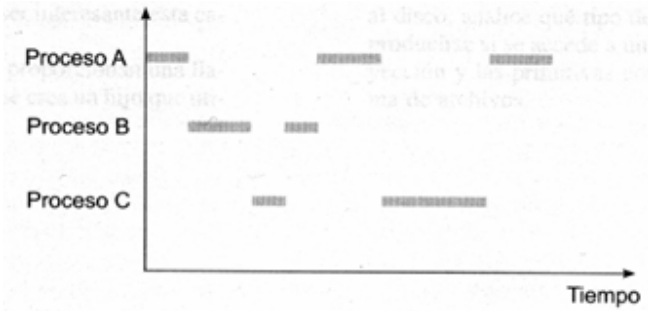


Figura 5.1. *Ejemplo de ejecución en un sistema multiprogramado con una única UCP.*

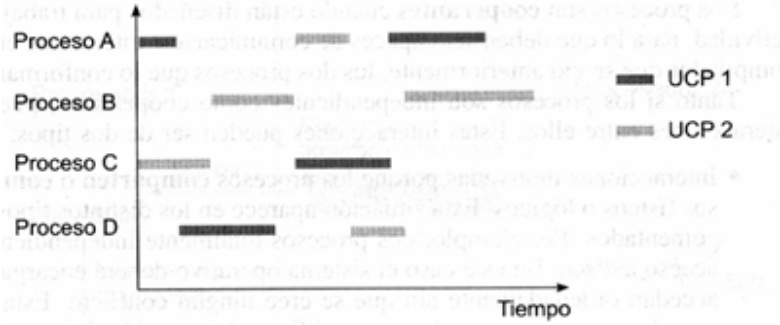


Figura 5.2. *Ejemplo de ejecución en un sistema multiprocesador.*

En general, la concurrencia será aparente siempre que el número de procesos sea mayor que el de procesadores disponibles, es decir, cuando haya mas de un proceso por procesador. La concurrencia será real cuando haya un proceso por procesador.

Aunque puede parecer que la intercalación y la superposición de la ejecución de procesos presentan formas de ejecución distintas, se verá que ambas pueden contemplase como ejemplos de procesos concurrentes y que ambas presentan los mismos problemas, los cuales pueden resolverse utilizando los mismos mecanismos.

Existen diversas razones que motivan la ejecución de procesos concurrentes en un sistema. Éstas son:

- **Facilita la programación** de aplicaciones al permitir que éstas se estructuren como un conjunto de procesos que cooperan entre sí para alcanzar un objetivo común. Por ejemplo, un compilador se puede construir mediante dos procesos: el compilador propiamente dicho, que se encarga de generar código ensamblador, y el proceso ensamblador que obtiene código en lenguaje máquina a partir del ensamblador. En este ejemplo puede apreciarse la necesidad de comunicar a los dos procesos.
- **Acelera los cálculos**. Si se quiere que una tarea se ejecute con mayor rapidez, lo que se puede hacer es dividirla en procesos, cada uno de los cuales se ejecuta en paralelo con los demás. Hay que hacer notar, sin embargo, que esta división a veces es difícil y no siempre es posible. Además el aumento en la velocidad de ejecución de la tarea sólo se puede conseguir si ejecutamos los distintos procesos en un multiprocesador o una multicomputadora.
- **Posibilita el uso interactivo** a múltiples usuarios que trabajan de forma simultánea desde varios te males.
- **Permite un mejor aprovechamiento de los recursos**, en especial de la UCP ya que, como se vio en el Capítulo 2, se pueden aprovechar las fases de entrad salida de unos procesos p a realizar las fases de procesamiento de otros.

5.1.1. *Tipos* de procesos concurrentes

Los procesos que ejecutan de forma concurrente en un sistema se pueden clasificar como procesos independientes o cooperantes.

Un proceso independiente es aquel que ejecuta sin requerir la ayuda o cooperación de otros procesos. Un claro ejemplo de procesos independientes son los diferentes intérpretes de mandatos que se ejecutan de forma simultánea en un sistema.

Los procesos son **cooperantes** cuando están diseñados para trabajar conjuntamente en alguna actividad, para lo que deben ser capaces de comunicarse e interactuar entre ellos. En el ejemplo del compilador que se vio anteriormente, los dos procesos que lo conforman son procesos cooperantes.

Tanto silos procesos son independientes como cooperantes, puede producirse una serie de interacciones entre ellos. Estas interacciones pueden ser de dos tipos:

- Interacciones motivadas porque los procesos **comparten** o **compiten** por el acceso a recursos físicos o lógicos. Esta situación aparece en los distintos tipos de procesos anteriormente comentados. Por ejemplo, dos procesos totalmente independientes pueden competir por el acceso a disco. En este caso el sistema operativo deberá encargarse de que los dos procesos accedan ordenadamente sin que se cree ningún conflicto. Esta situación también aparece cuando varios procesos desean modificar el contenido de un registro de una base de datos. Aquí es el gestor de la base de datos el que se tendrá que encargar de ordenar los distintos accesos al registro.
- Interacción motivada porque los procesos se **comunican** y **sincronizan** entre sí para alcanzar un objetivo común, Por ejemplo, los procesos compilador y ensamblador descritos anteriormente son dos procesos que deben comunicarse y sincronizarse entre ellos con el fin de producir código en lenguaje máquina.

Estos dos tipos de interacciones obligan al sistema operativo a incluir unos servicios que permitan la comunicación y la sincronización entre procesos, servicios que se presentarán a lo largo de este capítulo.

5.2. PROBLEMAS CLÁSICOS DE COMUNICACIÓN Y SINCRONIZACIÓN

La interacción entre procesos se plantea en una serie de situaciones clásicas de comunicación y sincronización. Estas situaciones junto con sus problemas se presentan a continuación para demostrar la necesidad de comunicar y sincronizar procesos. Todos ellos se resolverán a lo largo del presente capítulo mediante los diferentes mecanismos de comunicación y sincronización que ofrecen los sistemas operativos.

5.2.1. El problema de la sección crítica

Éste es uno de los problemas que con mayor frecuencia aparece cuando se ejecutan procesos concurrentes, tanto si son cooperantes como independientes. Considérese un sistema compuesto por ***n*** procesos {P_1, P_2, Pn} en el que cada uno tiene un fragmento de código, que se denomina *sección* ***crítica.*** Dentro de la sección crítica los procesos pueden estar accediendo y modificando y variables comunes, registros de una base de datos, un archivo, en general cualquier recurso compartido. La característica mas importante de este sistema es que, cuando un proceso se encuentra ejecutando código de la sección crítica, ningún otro proceso puede ejecutar en su sección.

Para ilustrar este problema se van a presentar dos ejemplos en los que existe un fragmento de código que constituye una sección critica.

Considérese en primer lugar un sistema operativo que debe asignar un identificador de proceso (PID) a dos procesos en un sistema multiprocesador. Esta situación se presenta en la Figura *5.3.* Cada vez que se crea un nuevo proceso, el sistema operativo le asigna un PID. El valor del último PID asignado se almacena en un registro o variable. Para asignar un nuevo PID, el sistema operativo debe llevar a cabo las siguientes acciones:

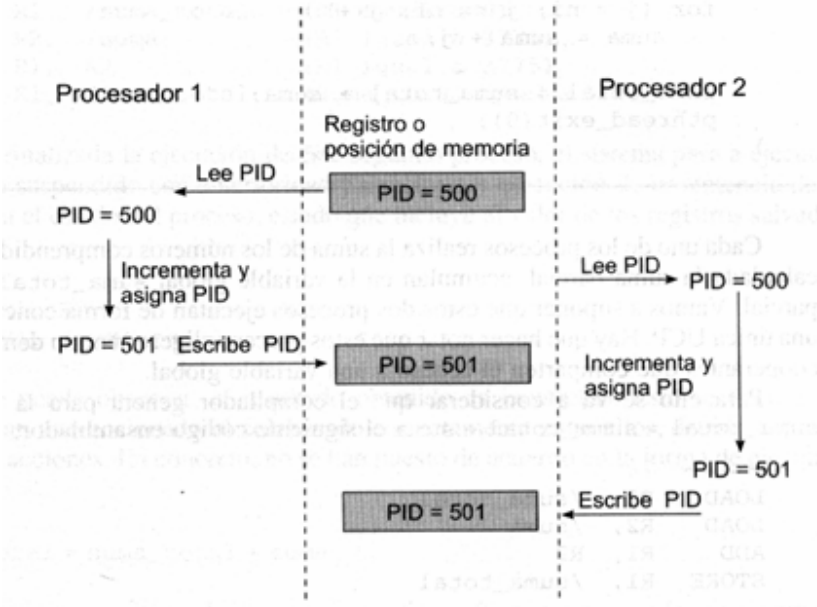


Figura 5.3. *Generación de PID en un sistema multiprocesador.*

- Leer el último PID asignado.
- Incrementar el valor del último PID. El nuevo valor será el PID a asignar al proceso.
- Almacenar el nuevo PID en el registro o variable utilizado a tal efecto.

Cuando el sistema operativo ejecuta las operaciones anteriores en dos procesadores de forma simultánea sin ningún tipo de control, se pueden producir errores, ya que se puede asignar el mismo PID a dos procesos distintos. Este problema

se debe a que las acciones anteriormente descritas constituyen una sección crítica, que debe ejecutarse de forma **atómica,** es decir, de forma completa e indivisible. Esto es, cuando un proceso empiece a ejecutar código de la sección crítica, ningún otro proceso podrá ejecutar dicho código mientras el primero no haya acabado su sección.

Vamos a describir a continuación un ejemplo con mayor detalle. Supongamos que se quiere calcular la suma de los ***n*** primeros números naturales de forma paralela utilizando múltiples procesos ligeros, cada uno de los cuales se va a encargar de la suma de un subconjunto de todos los números. En la Figura *5.4* se presentan de forma gráfica los dos procesos ligeros que se encargan de realizar la suma de los 200 primeros números naturales.

Un proceso ligero crea los dos procesos encargados de realizar las sumas parciales, cada uno de los cuales ejecuta el fragmento que se muestra en el Programa 5.1.

Programa 5.l. Código de la función suma_parcial.

```
int suma_total=0;

void suma_parcial(int ni, int nf) {
int j=0;
int suma =0;
```



```
for (j= ni; j<= nf; j++)
  suma =suma + j;

suma_total = suma_total + suma;
pthread._exit(0);
}
```

Cada uno de los procesos realiza la suma de los números comprendidos entre ni y nf. Una vez calculada la suma parcial, acumulan en la variable global sumatotal el resultado de su parcial. Vamos a suponer que estos dos procesos ejecutan de forma concurrente en un una única UCP. Hay que hacer notar que estos procesos ligeros entran dentro de la clase de cooperantes que comparten el acceso a una variable global.

Para ello se va a considerar que el compilador genera para la sentencia en suma_total =suma_total + suma el siguiente código ensamblador;

```
LOAD    Rl,   /suma_total
LOAD    R2,   /suma
ADD     Rl,   R2
STORE   Rl,   /suma_total
```

Inicialmente, la variable suma_total toma valor cero. Consideremos que los procesos entrelazan sus ejecuciones. de la siguiente manera: inicialmente ejecuta el primer proceso calculando su suma parcial igual a *1275.* Para almacenar el resultado en la variable suma_total ejecuta siguientes sentencias:

```
LOAD    Rl, /suma_total        (Rl igual a 0)
LOAD    R2, /suma              (R2igualal275)
```

Llegados a este punto, el proceso consume su rodaja de tiempo y el sistema operativo decide pasar a ejecutar el otro proceso ligero. Para ello salva el estado del primero y comienza a ejecutar el segundo. En este segundo proceso el valor de la variable local suma es *3775.*

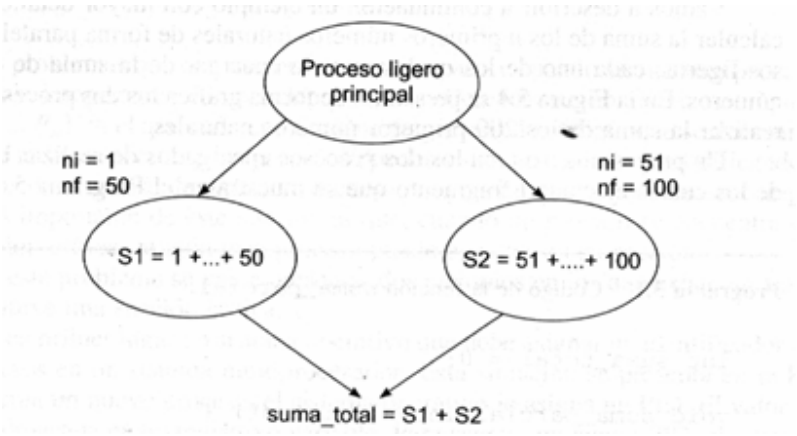


Figura ***5.4.*** *Suma en paralelo realizada por dos procesos ligeros.*


```
LOAD    Rl,   /suma_total    (Rl igual a 0)
LOAD    R2,  /suma           (R2 igual a 3775)
ADD     Rl,   R2             (Rl igual a 3775)
STORE Rl,     /suma_total    (suma_total igual a 3775)
```

Una vez finalizada la ejecución de este segundo proceso, el sistema pasa a ejecutar el primer proceso ligero suspendido con anterioridad y concluye la ejecución de la sentencia de suma. Para ello se restaura el estado del proceso, estado que incluye el valor de los registros salvados anteriormente.

```
ADD         Rl, R2              (Rl en este proceso es 1275)
STORE Rl,   /suma total         (suma_total igual a 1275)
```

Como se puede observar, el resultado obtenido es incorrecto. Esto se debe a que los dos procesos no han interaccionado ni colaborado correctamente entre ellos, debido a que no han sincronizado sus acciones, En concreto, no se han puesto de acuerdo en la forma de ejecutar la sentencia:

```
suma _total =suma total + suma
```

Esta sentencia constituye de nuevo una sección crítica que no puede ser ejecutada

de forma simultánea por más de un proceso. Esta sección debe ejecutarse de forma atómica. Para ello los procesos deben sincronizar sus ejecuciones: unos deben esperar a ejecutar el código de la sección crítica mientras otro lo esté ejecutando.

Para resolver el problema de la sección crítica es necesario utilizar algún *mecanismo de sincronización* que permita a los procesos cooperar entre ellos sin problemas. Este mecanismo debe proteger el código de la sección critica y su funcionamiento básico es el siguiente:

- Cada proceso debe solicitar permiso para entrar en la sección crítica, mediante algún fragmento de código que se denomina de forma genérica *entrada en la sección crítica.*
- Cuando un proceso sale de la sección critica debe indicarlo mediante otro fragmento de código que se denomina *salida de la sección crítica.* Este fragmento permitirá que otros procesos entren a ejecutar el código de la sección crítica.

La estructura general, por tanto, de cualquier mecanismo que pretenda resolver el problema de la sección critica es la siguiente:

Entrada en la sección crítica
Código de la sección crítica
Salida de la sección crítica

Cualquier solución que se utilice para resolver este problema debe cumplir los tres requisitos siguientes:

- **Exclusión mutua:** si un proceso está ejecutando código de la sección crítica, ningún otro proceso lo podrá hacer. En caso contrario se llegaría a situaciones como las que se han descrito en los ejemplos anteriores.
- **Progreso**: si ningún proceso está ejecutando dentro de la sección crítica, la decisión de qué proceso entra en la sección se hará sobre los procesos que desean entrar. Los procesos que no quieren entra no pueden formar parte de esta decisión. Además, esta decisión debe realizarse en tiempo finito.

- **Espera acotada:** debe haber un límite en el número de veces que se permite que los demás procesos entren a ejecutar código de la sección crítica después de que un proceso haya efectuado una solicitud de entrada y antes de que se conceda la suya.

En el ejemplo anterior si el código que conforma la sección crítica se hubiera ejecutado en exclusión mutua se habría obtenido e] resultado correcto. El primer proceso acumularía en la variable suma_total el valor 1275 y a continuación lo haría el siguiente proceso dando un valor final de 5050.

5.2.2. Prob1ema del productor-consumidor

El problema del productor-consumidor es uno de los problemas más habituales que surge cuando se programan aplicaciones utilizando procesos concurrentes. En este tipo de problemas uno o más procesos, que se denominan *productores,* generan cierto tipo de datos que son utilizados o consumidos por otros procesos que e denominan

consumidores. Un claro ejemplo de este tipo de problemas es el del compilador que se describió en la Sección *5.1.* En este ejemplo, el compilador hace las funciones de productor al generar el código ensamblador que consumirá el proceso ensamblador para generar el código máquina. En la Figura *5.5* se representa la estructura clásica de este tipo de procesos.

En esta clase de problemas es necesario disponer de algún mecanismo de comunicación que permita a los procesos productor y consumidor intercambiar información. Ambos procesos, además, deben sincronizar su acceso al mecanismo de comunicación p a que la interacción entre ellos no sea problemática: cuando el mecanismo de comunicación se llene, e] proceso productor se deberá quedar bloqueado hasta que haya hueco para seguir insertando elementos. A su vez, el proceso consumidor deberá quedarse bloqueado cuando el mecanismo de comunicación este vacío, ya que en este caso no podrá continuar su ejecución al no disponer de información a consumir. Por tanto, este tipo de problema requiere servicios para que los procesos puedan comunicarse y servicios para que se sincronicen a la hora de acceder al mecanismo de comunicación.

5.2.3. El problema de los lectores-escritores

En este problema existe un determinado objeto (Fig. *5.6),* que puede ser un archivo, un registro dentro de un archivo, etc., que va a ser utilizado y compartido por una serie de procesos *con-*

Figura 5.5. *Estructura clásica de procesos productor-consumidor.*

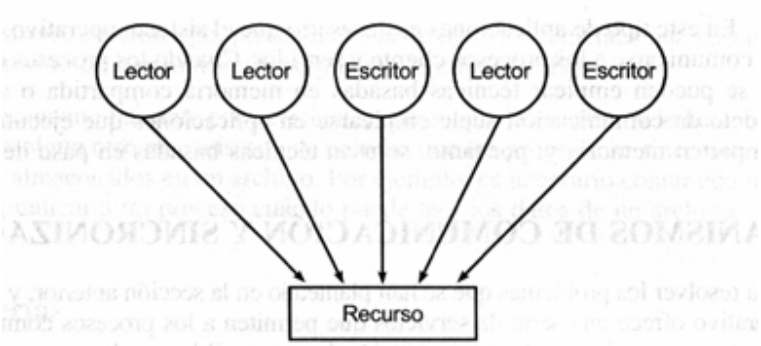


Figura ***5.6 Procesos lectores*** y *escritores.*

currentes. Algunos de estos procesos sólo van a acceder al objeto sin modificarlo, mientras que otros van a acceder al objeto para modificar su contenido. Esta actualización implica leerlo, modificar su contenido y escribirlo. A los primeros procesos se les denomina *lectores* y a los segundos se les denomina *escritores.* En este tipo de problemas existen una serie de restricciones que han de seguirse:

- Sólo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras el escritor esté accediendo al objeto, ningún otro proceso lector ni escritor podrá acceder a él.
- Se permite, sin embargo, que múltiples lectores tengan acceso al objeto, ya que ellos nunca van a modificar el contenido del mismo.

En este tipo de problemas es necesario disponer de servicios de sincronización que permitan a los procesos lectores y escritores sincronizarse adecuadamente en el acceso al objeto.

5.2.4. Comunicación cliente-servidor

En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos que se denominan clientes (Fig. *5.7).* El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión. Muchas aplicaciones y servicios de red, como el correo electrónico y la transferencia de archivos, se basan en este modelo.

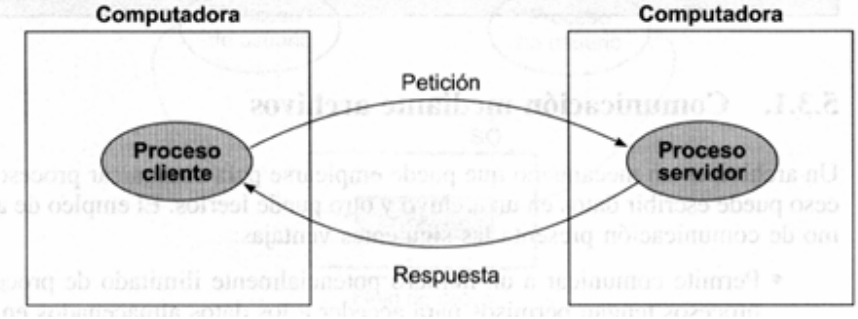


Figura ***5.7.*** *Comunicación cliente-servidor.*

En este tipo de aplicaciones es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor, Cuando los procesos ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o archivos. Sin embargo, este modelo de comunicación suele emplearse en aplicaciones que ejecutan en computadoras que no comparten memoria y, por tanto, se usan técnicas basadas en paso de mensajes.

5.3. MECANISMOS DE COMUNICACIÓN Y SINCRONIZACIÓN

Para resolver los problemas que se han planteado en la sección anterior, y otros muchos más, el sistema operativo ofrece una serie de servicios que permiten a los procesos comunicarse y sincronizarse.

Los mecanismos de comunicación hacen posible que los procesos intercambien datos entre ellos. Los principales mecanismos de comunicación que ofrecen los sistemas operativos son los siguientes:

- Archivos.
- Tuberías.
- Variables en memoria compartida.
- Paso de mensajes.

En los problemas de sincronización, un proceso debe esperar la ocurrencia de un determinado evento. Así, por ejemplo, en problemas de tipo productor-consumidor el proceso consumidor debe esperar mientras no haya datos que consumir. Para que los procesos puedan sincronizarse es necesario disponer de servicios que permitan bloque o suspender bajo determinadas circunstancias la ejecución de un proceso. Los principales mecanismos de sincronización que ofrecen los sistemas operativos son:

- Señales.
- Tuberías,
- Semáforos,
- Mutex y variables condicionales,
- Paso de mensajes.

En las siguientes secciones se describen cada uno de los mecanismos anteriores (Aclaración 5.1).

ACLARACIÓN 5.1

Dado que existen mecanismos que pueden utilizarse tanto para comunicar como para sincronizar procesos, los autores han preferido no dedicar una sección diferente para los mecanismos de comunicación y otra para los de sincronización.

5.3.1, Comunicación mediante archivos

Un archivo es un mecanismo que puede emplearse para comunicar procesos. Por ejemplo, un proceso puede escribir datos en un archivo y otro puede leerlos. El empleo

de archivos como mecanismo de comunicación presenta las siguientes ventajas:

- •Permite comunicar a un número potencialmente ilimitado de procesos. Basta con que los procesos tengan permisos para acceder a los datos almacenados en un chivo,
- •Los servidores de archivos ofrecen servicios sencillos y fáciles de utilizar.

Este mecanismo, sin embargo, presenta una serie de inconvenientes que hacen que en general no sea un mecanismo de comunicación ampliamente utilizado. Estos son:

- Es un mecanismo bastante poco eficiente, puesto que la escritura y lectura en disco es lenta.
- Necesitan algún otro mecanismo que pe ita que los procesos se sincronicen en el acceso a los datos almacenados en un archivo. Por ejemplo, es necesario contar con mecanismos que permitan indicar un proceso cuando puede leer los datos de un archivo.

5.3.2. Tuberías

Una tubería es un mecanismo de comunicación y sincronización. Desde el punto de vista de su utilización, es como un pseudoarchivo mantenido por el sistema operativo. Conceptualmente, cada proceso ve la tubería como un conducto con dos extremos, uno de los cuales se utiliza para escribir o insertar datos y el otro para extraer o leer datos de la tubería. La escritura se realiza mediante el servicio que se utiliza para escribir datos en un chivo. De igual forma, la lectura se lleva a cabo mediante el servicio que se emplea para leer de un archivo.

El flujo de datos en la comunicación empleando tuberías es unidireccional (Aclaración *5.2)* y FIFO, esto quiere decir que los datos se extraen de la tubería (mediante la operación de lectura) en el mismo orden en el que se insertaron (mediante la operación de escritura). La Figura *5.8* representa dos procesos que se comunican de forma unidireccional utilizando una tubería.

ACLARACIÓN 5.2

El flujo de datos hace referencia al sentido en el que circulan los datos en el mecanismo de comunicación. De acuerdo a este sentido, el flujo de datos puede ser:

— Unidireccional. Los datos que se comunican entre los procesos fluyen en un solo sentido. En este caso uno de los procesos será el emisor y el otro el receptor. Si se quiere que los dos procesos que intervienen en la comunicación actúen de emisor y de receptor habrá de utilizarse dos mecanismos de comunicación. Cada proceso utilizará uno de ellos para enviar datos y el otro para recibir.

— Bidireccional. Los datos pueden viajar en ambos sentidos y, por tanto, todos los procesos que utilicen el mecanismo de comunicación pueden hacer de receptor y de emisor.

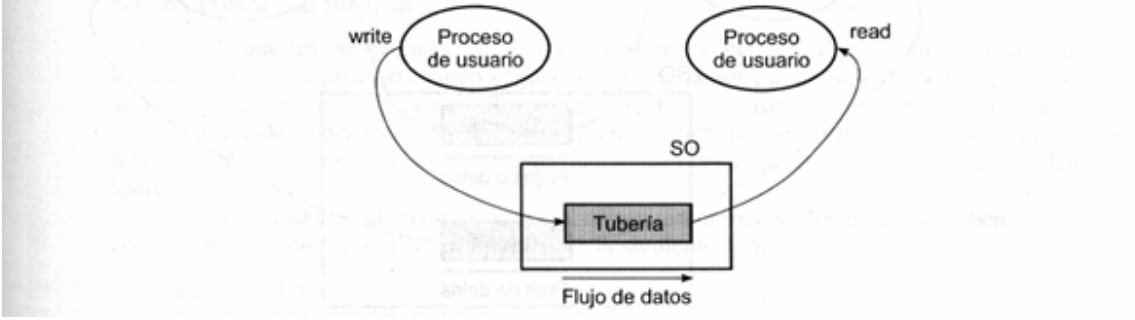


Figura ***5.8.*** *Comunicación unidireccional empleando una tubería.*

Cuando se desea disponer de un flujo de datos bidireccional es necesario crear dos tuberías como se muestra en la Figura *5.9.*

Una tubería es un mecanismo de comunicación con almacenamiento, El tamaño de una tubería varía en cada sistema operativo, aunque el tamaño típico e. de 4 KB. Sobre una tubería puede haber múltiples procesos lectores y escritores. Las tuberías se implementan normalmente como regiones de memoria compartida entre los procesos que utilizan la tubería.

A continuación se describe la semántica de las operaciones de lectura y escritura sobre una tubería,

Escritura en una tubería

Una operación de escritura sobre una tubería introduce datos en orden FIFO en la misma. La semántica de esta operación es la siguiente:

- Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.
- Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error.
- Una operación de escritura sobre una tubería se realiza de forma atómica, es decir, si dos procesos intentan escribir de forma simultánea en una tubería, sólo uno de ellos lo hará. El otro se bloque á hasta que finalice la primera escritura.

Lectura de una tubería

Una operación de lectura de una tubería obtiene los datos almacenados en la misma. Estos datos, además, se eliminan de la tubería. Las operaciones de lectura siguen la siguiente semántica:

- Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma,
- Si la tubería almacena *M* bytes y se quieren leer *n* bytes, entonces:
  - Si $M \geq n$, la llamada devuelve *n* bytes y elimina de la tubería los datos solicitados,
  - Si $M < n$, la llamada devuelve *M* bytes y elimina los datos disponibles en la tubería.

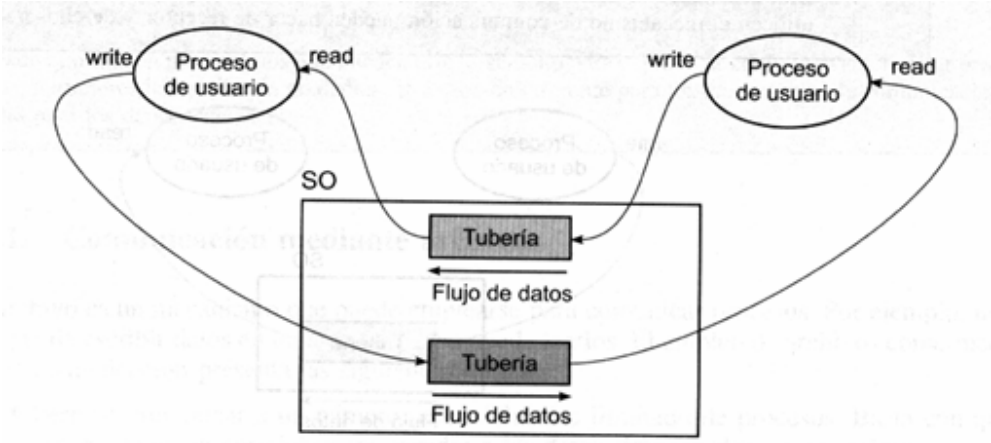


Figura 5.9. *Comunicación bidireccional empleando do, tubería*

- Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo (en este caso la operación no bloquea al proceso).
- Al igual que las escrituras, las operaciones de lectura sobre una tubería son atómicas (Aclaración *5.3).*

ACLARACIÓN 5.3

En general, la atomicidad en las operaciones de lectura y escritura sobre una tubería se asegura siempre que el número de datos involucrados en las anteriores operaciones sea menor que el *tamaño de la misma.*

Como puede apreciarse, una lectura de una tubería nunca bloquea al proceso si hay datos disponibles en la misma.

Existen dos tipos de tuberías: sin nombre y con nombre (Aclaración 5.4). Una tubería **sin nombre** solamente se puede utilizar entre los procesos que desciendan del proceso que creó la tubería. Una tubería **con nombre** se puede utilizar para comunicar y sincronizar procesos independientes.

ACLARACIÓN 5.4

Existen tres formas, en general, de identificar a un mecanismo de comunicación o sincronización:

— Sin nombre: en este caso el servicio no tiene un nombre asociado y, por tanto, sólo puede ser utilizado por el proceso que lo crea y por aquellos procesos que lo hereden.
— Con un nombre local: en este caso el mecanismo tiene un nombre al que pueden acceder todos los procesos que ejecuten en la misma máquina y tengan permisos de acceso. Esto permite comunicar y sincronizar a procesos que residan en la misma máquina siempre que conozcan el nombre dado al mecanismo de comunicación o sincronización y tengan permisos para su utilización.
— Con nombre de red: en este caso el mecanismo tiene un nombre que lo identifica de forma única dentro de una red de computadoras y, por tanto, permite comunicar y sincronizar a procesos que ejecuten en computadoras distintas.

A continuación se describe como solucionar algunos de los problemas de comunicación y sincronización, vistos en la Sección *5.2,* mediante el uso de tuberías.

Sección crítica con tuberías

El hecho de que las operaciones de lectura y escritura sobre una tubería sean atómicas y que la lectura bloquee a un proceso cuando se encuentra vacía la tubería, permite que este mecanismo sea utilizado para sincroniza procesos. Recuérdese que toda sincronización implica un bloqueo.

La forma de resolver el problema de la sección crítica mediante tuberías consiste en crear una tubería en la que se inserta inicialmente, mediante una operación de escritura, un dato que hace de testigo.

Una vez creada la tubería e insertado el testigo en la misma, los procesos deben proteger el código correspondiente a la sección crítica de la siguiente forma:

Leer el dato de la tubería
Código correspondiente a la sección crítica
Escribir el dato en la tubería

Cuando varios procesos intenten ejecutar el fragmento de código anterior, sólo uno de ellos leerá y consumirá el dato que hace de testigo de la tubería. El resto de procesos se quedar bloqueados hasta que se inserte de nuevo el testigo en la tubería, operación que se realiza a la salida de la sección crítica. Debido a que las operaciones de lectura y escritura sobre una tubería son atómicas, se asegura que nunca dos procesos leerán de la misma de forma simultánea, consumiendo de esta forma el testigo y entrando los dos a ejecutar el código de la sección crítica.

Además, con la solución expuesta anteriormente existe progreso, puesto que sólo los procesos que quieren entrar en la sección crítica son los que ejecutan la sentencia de lectura. En la decisión de qué procesos entran en la sección crítica sólo toman partido los procesos que ejecuten dicha sentencia. En general existirá espera limitada, ya que en algún momento el proceso bloqueado en la operación de lectura será elegido por el planificador del sistema operativo para ejecutar y, por tanto, podrá acceder a ejecutar el código de la sección crítica.

Productor-consumidor con tuberías

Dado que una tubería es un mecanismo de comunicación y sincronización, se puede utilizar p a resolver problemas de tipo productor-consumidor. La comunicación entre los procesos productor consumidor se realiza a través de la tubería. Cuando el productor ha elaborado algún elemento, inserta en la tubería mediante una operación de escritura. Cuando el consumidor quiere algún elemento, lee de la tubería utilizando una operación de lectura. Además, la semántica de las operaciones de lectura y escritura sobre una tubería asegura que el consumidor se quede bloqueado cuando no haya datos que consumir. Por su parte, si el proceso productor es mas consumidor, se bloqueará cuando la tubería se encuentre llena, Tanto el proceso productor como el consumidor pueden realizar operaciones de lectura y escritura de diferentes tamaños.

La estructura general de un proceso productor-consumidor utilizando tuberías se muestra en Programa *5.2*.

Programa 5.2. Estructura de los procesos productor y consumidor utilizando tuberías.

```
Productor() {
    for (;;) {
        < Producir un dato >
        < Escribir el dato a la tubería >
    }
}

Consumidor() {
    for (;;) {
        < Leer un dato a la tubería >
        < Consumir el dato leído >

    }
}
```

Ejecución de mandatos con tuberías

Aunque en las secciones anteriores se han presentado dos posibles utilizaciones de las tuberías uso mas extendido se encuentra en la ejecución de mandatos con tuberías. De esta forma, se pueden conectar programas sencillos para realizar una tarea más compleja.

Cuando un usuario en UNIX desea contabilizar el número de usuarios que están conectados al sistema en un instante determinado, lo puede hacer ejecutando el mandato who wc | -1 (Aclaración *5.5).* Con el mandato anterior se crean dos procesos que se comunican entre sí mediante una tubería. El primer proceso ejecuta el programa who y el segundo ejecuta el programa wc -1. La salida del mandato who pasa a ser la entrada del mandato wc -1.

ACLARACIÓN 5.5

El mandato who de UNIX imprime información sobre los usuarios conectados al sistema. El mandato wc -l lee datos de la entrada estándar y calcula el número de líneas existentes.

La ejecución de mandatos con tuberías forma parte de la clase de problemas de tipo productor-consumidor.

5.3.3. Sincronización mediante señales

Las señales descritas en el Capítulo 3 pueden utilizarse para sincronizar procesos. Si se utilizan, por ejemplo, señales POSIX, un proceso puede bloquearse en el servicio pause esperando la recepción de una señal. Esta señal puede ser enviada por otro proceso mediante el servicio kill. Con este mecanismo se consigue que unos procesos esperen a que se cumpla una determinada condición y que otros los despierten cuando se cumple la condición que les permite continuar su ejecución.

El empleo de señales, sin embargo, no es un mecanismo muy apropiado para sincronizar procesos debido a las siguiente razones:

- Las señales tienen un comportamiento asíncrono. Un proceso puede recibir una señal en cualquier punto de su ejecución, aunque no esté esperando su recepción.
- Las señales no se encolan, Si hay una señal pendiente de entrega a un proceso y se recibe una señal del mismo tipo, la primera se perderá. Sólo se entregará la última. Esto hace que se puedan perder eventos de sincronización importantes.

5.3.4. Semáforos

Un semáforo [Dijkstra, 1965] es un mecanismo de sincronización que se utiliza generalmente en sistemas con memoria compartida, bien sea un monoprocesador o un multiprocesador. Su uso en una multicomputadora depende del si tema operativo en particular.

Un semáforo es un objeto con un valor entero, al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: **wait** y **signal** (Aclaración *5.6).* Las definiciones de estas dos operaciones son las siguientes:

```
wait (s) {
    s = s –1;
    if (s < 0)
       Bloquear al proceso;
                                    1
}
signal (s) {
```



```
    s = s + 1;
    if ( s <= 0 )
       Desbloquear a un proceso bloqueado en la operación wait;
}
```

ACLARACIÓN 5.6

La operación `wait` también recibe otros nombres como `down` o `P`. La operación `signal` recibe también otros nombres como `up` o `V`.

Cuando el valor del semáforo es menor o igual que cero, cualquier operación wait que se realice sobre el semáforo bloqueará al proceso. Cuando el valor del semáforo es positivo, cualquier proceso que ejecute una operación wait no se bloqueará.

El número de procesos, que en un instante determinado se encuentran bloqueados en una operación wait, viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación signal, el valor del semáforo se incrementa. En el caso de que haya algún proceso bloqueado en una operación wait anterior, se desbloqueará a un solo proceso.

A continuación se presenta el uso de los semáforos en la resolución de alguno de los problemas vistos en la Sección 5.2.

Sección crítica con semáforos

Para resolver el problema de la sección crítica utilizando semáforos debemos proteger el código que constituye la sección crítica de la siguiente forma:

wait (s);
Sección crítica;
signal(s);

El valor que tiene que tomar el semáforo inicialmente es 1, de esta forma solo se permite a un único proceso acceder a la sección crítica. Si el valor inicial del semáforo fuera, por ejemplo, 2, entonces dos procesos podrían ejecutar la llamada wait sin bloquearse y por tanto se permitiría que ambos ejecutaran de forma simultánea dentro de la sección crítica.

En la Figura 5.10 se representa la ejecución de tres procesos *(P_0 , P_1* y *P_2)* que intentan acceder a una sección crítica. Los tres procesos se sincronizan utilizando un semáforo con valor inicial 1.

Productor-consumidor con semáforos

Una posible forma de solucionar el problema del productor-consumidor es utilizar un almacén o buffer circular compartido por ambos procesos. El proceso productor fabrica un determinado dato y lo inserta en un buffer (Fig. 5.11). El proceso consumidor retira del buffer los elementos insertados por el productor.

A continuación se presenta una solución al problema del productor-consumidor utilizando procesos ligeros y semáforos. El buffer utilizado para esta solución se trata como una cola circular Cuando el buffer tiene un tamaño limitado, como suele ser lo habitual, se dice que el problema es de tipo productor-consumidor con buffer circular y acotado.

En este tipo de problemas es necesario evitar que ocurra alguna de las siguientes situaciones:

- El consumidor saca elementos cuando el buffer está vacío.
- El productor coloca elementos en el buffer cuando éste se encuentra lleno.

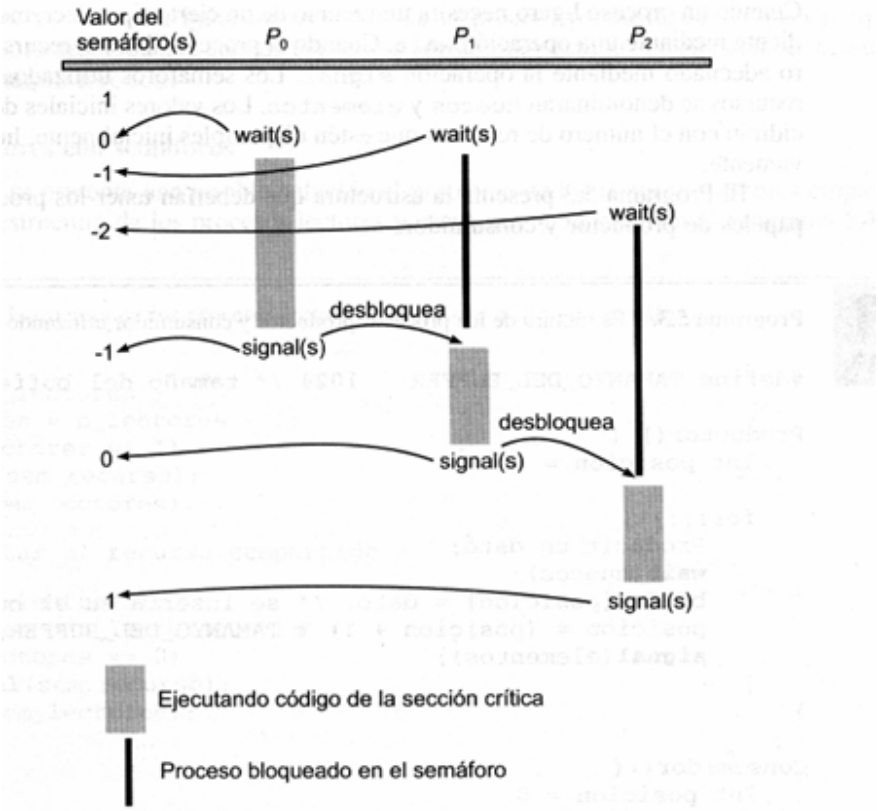


Figura 5.10. ***Sección crítica con semáforos.***

- El productor sobrescribe un elemento que todavía no ha sido sacado del buffer.
- El consumidor saca elementos del buffer que ya fueron sacados con anterioridad,
- El consumidor saca un elemento mientras el productor lo está insertando.

En este problema existen dos tipos de recursos: los elementos situados en el buffer y los huecos donde situ nuevos elementos. Cada uno de estos recursos se representa mediante un semáforo.

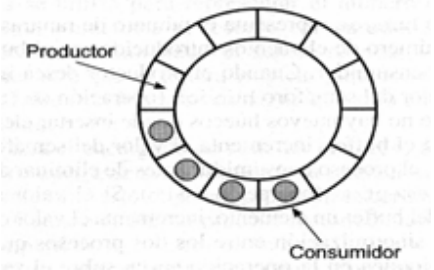


Figura 5.11.*Productor-consumidor con buffer circular,*

Cuando un proceso ligero necesita un recurso de un cierto tipo, decrementa el semáforo correspondiente mediante una operación wait. Cuando el proceso libera el recurso, se incrementa el semáforo adecuado mediante la operación signal. Los semáforo. utilizados para representar estos dos recursos se denominarán huecos y elementos. Los valores iniciales de estos dos semáforos coincidirán con el número de recursos que estén disponibles inicialmente, huecos y elementos respectivamente.

El Programa ***5.3*** presenta la estructura que deberían tener los procesos ligeros que hagan los papeles de productor y consumidor.

Programa 5.3. Estructura de los procesos productor y consumidor utilizando semáforos y procesos ligeros.

```
#define TAMANYO_DEL_BUFFER      1024 / * tamaño del buffer *1

Productor () {
    int posicion =;

    for (;;) {
        Producir un dato;
        wait(huecos);
        buffer[posición] = dato; /* se inserta en el buffer */
        posicion = (posicion + 1) % TAMANYQ_DELBUFFER;
        signal. (elementos);
    }
}

Consumidor () {
    int posicion O;

    for(;;) {
        wait(elementos);
        dato = buffer[posicion]; / * se extrae del buffer */
        posicion = (posicíon + 1) % TAMANYO_DEL BUFFER;
        signal (huecos);
        Consumir el dato extraido;
    }
}
```

El semáforo huecos representa el numero de ranuras libres que hay en el buffer y el semáforo elementos el número de elementos introducidos en el buffer por el productor que aún no han sido retirados por el consumidor. Cuando el productor desea introducir un nuevo elemento en el buffer decrementa el valor del semáforo huecos (operación wait). Si el valor se hace negativo, el proceso se bloquea ya que no hay nuevos huecos donde insertar elementos. Cuando el productor ha insertado un nuevo dato en el buffer, incrementa el valor del semáforo elementos (operación signal).

Por su parte, el proceso consumidor antes de eliminar del buffer un elemento decrementa el valor del semáforo elementos (operación wait). Si el Valor se hace negativo, el proceso se bloquea. Cuando elimina del buffer un elemento, incrementa el

valor del semáforo huecos (operación signal).

La correcta sincronización entre los dos procesos queda asegurada ya que cuando el proceso consumidor se bloquea en la operación wait sobre el semáforo huecos se despertará cuando el proceso consumidor inserte un nuevo elemento en el buffer e incremente el valor de dicho semáforo

con la operación signal. De igual manera, cuando el proceso productor se bloquea porque el buffer está vació, se desbloqueara cuando el proceso consumidor extraiga un elemento e incremente el valor del semáforo huecos.

Lectores-escritores con semáforos

En esta sección se presenta una posible solución al problema de los lectores escritores empleando semáforos. La estructura de los procesos lectores y escritores se muestra en el Programa ***5.4.***

Programa 5.4. Estructura de los procesos lectores y escritores utilizando semáforos.

```
Lector( ) {
      wait (sem_lectores);
      int_lectores = n_lectores + 1;
      if (n_lectores = = 1)
        wait (sem_recurso);
        signal (sem-lectores);

        < consultar el recurso compartido >

        wait (sem_lectores);
        n_lectores = n_lectores _ 1;
        if (n-lectores = = 0)
            signal (sen_recurso);
        signal(sem-lectores);
 }
  Escritor ( ) {
      wait (sem_recurso);
      /* se puede modificar el recurso */
      signal(sem_recurso);
  }
```

En esta solución, el semáforo sem_recurso se utiliza para asegurar la exclusión mutua en el acceso al dato a compartir. Su valor inicial debe ser 1, de esta manera en cuanto un escritor consigue decrementar su valor puede modificar el dato y evitar que ningún otro proceso, ni lector ni escritor acceda al recurso compartido.

La variable n_lectores se utiliza para representar el número de procesos lectores que se encuentran accediendo de forma simultánea al recurso compartido. A esta

variable acceden los procesos lectores en exclusión mutua utilizando el semáforo sem_lectores. El valor de este semáforo, como el del cualquier otro que se quiera emplear para acceder en exclusión mutua a un fragmento de código debe ser 1. De esta forma se consigue que sólo un proceso lector modifique el valor de la variable n_lectores,

El primer proceso lector será el encargado de solicitar el acceso al recurso compartido decrementando el valor del semáforo sem_recurso mediante la operación wait. El resto de procesos lectores que quieran acceder mientras esté el primero podrán hacerlo sin necesidad de solicitar el acceso al recurso compartido. Cuando el ultimo proceso lector abandona la sección de código que permite acceder al recurso compartido n_lectores se hace 0. En este caso deberá incrementar el valor del semáforo sem_recurso para permitir que cualquier proceso escritor pueda acceder para modificar el recurso compartido.

Esta solución, tal y como se ha descrito, permite resolver el problema de los lectores-escritores pero concede prioridad a los procesos lectores. Siempre que haya un proceso lector, consultando el valor del recurso, cualquier proceso lector podrá acceder sin necesidad de solicitar el acceso. Sin embargo, los procesos escritores deberán esperar hasta que haya abandonado la consulta el ultimo lector. Existen soluciones que permiten dar prioridad a los escritores, soluciones que se dejan al lector.

5.3.5. Memoria compartida

La memoria compartida es un paradigma que permite comunicar a procesos que ejecutan en la misma máquina, bien sea un monoprocesador o un multiprocesador. Con este modelo de comunicación un proceso almacena un valor en una determinada variable, y otro proceso puede acceder a ese valor sin más que consultar la variable. De esta forma se consigue que los dos procesos puedan comunicarse entre ellos.

En el Capítulo 3 se presento el concepto de proceso ligero. Los procesos ligeros que se crean dentro de un proceso comparten memoria de forma natural, y utilizan ésta como mecanismo de comunicación. En el ejemplo del productor-consumidor con semáforos se empleó esta técnica para comunicar al proceso productor y consumidor a través de un buffer común.

Cuando se quiere emplear memoria compartida entre procesos creados con fork es necesario recurrir a servicios ofrecidos por el sistema operativo que permitan que los procesos que se quieren comunicar creen un segmento de memoria compartida al que ambos pueden acceder a través de posiciones de memoria situadas dentro de su espacio de direcciones.

En la Figura 5.12 se representa el concepto de segmento de memoria compartida, ya presentado en el capitulo Capítuto 4. Algún procesos debe encargarse de crear ese segmento, mientras que el resto de los procesos que quieran utilizarlo simplemente tienen que acceder a él. Cada uno de los procesos pueden acceder a este segmento de memoria compartida utilizando direcciones diferentes.

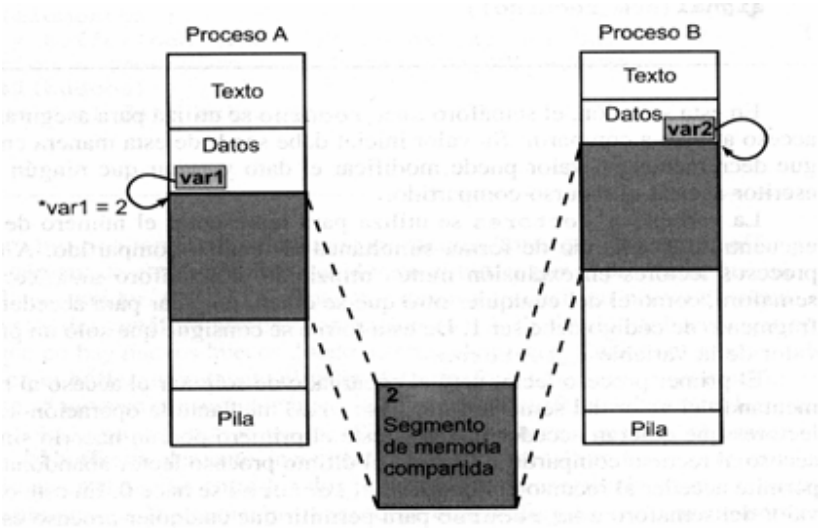


Figura 5.12. ***Memoria compartida entre procesos.***

El proceso A accede al segmento de memoria compartida utilizando la dirección almacenada en la variable de tipo puntero var1, mientras que el proceso B lo hace a través de la dirección almacenada en la variable de tipo puntero var2.

5.3.6. Mutex *y* variables condicionales

Los mutex y las variables condicionales son mecanismos especialmente concebidos para la sincronización de procesos ligeros.

Un mutex es el mecanismo de sincronización de procesos ligeros más sencillo y eficiente. Los mutex se emplean para obtener acceso exclusivo a recursos compartidos y para asegurar la exclusión mutua sobre secciones críticas.

Sobre un mutex se pueden realizar dos operaciones atómicas básicas:

- lock: intenta bloquear el mutex. Si el mutex ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. En caso contrario se bloquea el mutex sin bloquear al proceso.
- unlock: desbloquea el mutex. Si existen procesos bloqueados en él, se desbloqueará a uno para de ellos que será el nuevo proceso que adquiera el mutex. La operación unlock sobre un mutex debe ejecutarla el proceso ligero que adquirió con anterioridad el mutex mediante la operacion operación lock. Esto es diferente a lo que ocurre con las operaciones wait y signal sobre un semáforo.

Sección crítica con mutex

El siguiente segmento de pseudocódigo utiliza un mutex para proteger una sección

crítica:

```
lock (m) ;      / *solicita la entrada en la sección crítica */
< sección crítica >
unlock(m) ;  /* salida de la sección critica */
```

En la Figura 5.13 se representa de forma gráfica una situación en la que dos procesos ligeros intentan acceder de forma simultánea a ejecutar código de una sección crítica utilizando un mutex para protegerla.

Dado que las operaciones lock y unlock son atómicas, solo un proceso conseguirá bloquear el mutex y podrá continuar su ejecución dentro de la sección crítica. El segundo proceso se bloqueara hasta que el primero libere el mutex mediante la operación unlock.

Una **variable condicional** es una variable de sincronización asociada a un mutex que se utiliza para bloquear a un proceso hasta que ocurra algún suceso. Las variables condicionales tienen dos operaciones atómicas para esperar y señalizar:

- c_wait: bloquea al proceso que ejecuta la llamada y le expulsa del mutex dentro del cual se ejecuta y al que está asociado la variable condicional, permitiendo que algún otro proceso adquiera el mutex. El bloqueo del proceso y la liberación del mutex se realiza de forma atómica.
- c_signal: desbloquea a uno o varios procesos suspendidos en la variable condicional. El proceso que se despierta compite de nuevo por el mutex.

A continuación se va a describir una situacion típica en la que se utilizan los mutex y las variables condicionales de forma conjunta. Supóngase que una serie de procesos compiten por el.

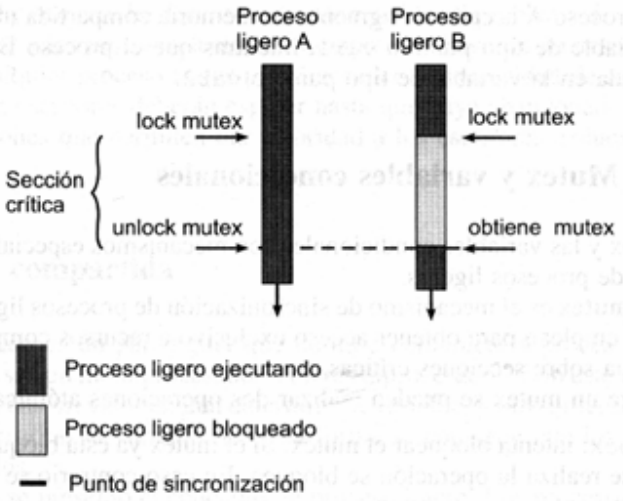


Figura 5.13. *Ejemplo de mutex en una sección crítica.*

acceso a una sección crítica. En este caso, es necesario un mutex para proteger la ejecución de dicha sección crítica. Una vez dentro de la sección crítica puede ocurrir que un proceso no pueda continuar su ejecución dentro de la misma, debido a que no se cumple una determinada condición, por ejemplo, se quiere insertar elementos en un buffer común y este se encuentra lleno. En esta situación el proceso debe bloquearse

puesto que no puede continuar su ejecución. Además debe liberar el mutex para permitir que otro proceso entre en la sección crítica y pueda modificar la situación que bloqueó al proceso, en este caso eliminar un elemento del buffer común para hacer hueco.

Para conseguir este funcionamiento es necesario utilizar una o más variables compartidas que se utilizarán como predicado lógico y que el proceso consultará para decidir su bloqueo o no. El fragmento de código que se debe emplear en este caso es el siguiente:

```
lock(m) ;
/* código de la sección crítica */
while (condicion = = FALSE)
    c_wait(c, m) ;
/* resto de la sección crítica */
unlock(m) ;
```

En el fragmento anterior m es el mutex que se utiliza para proteger el acceso a la sección crítica y c la variable condicional que se emplea para bloquear el proceso y abandonar la sección crítica,

Cuando el proceso que está ejecutando dentro de la sección evalúa la condición y ésta es falsa, se bloquea mediante la operación c_wait y libera el mutex permitiendo que otro proceso entre e ella.

El proceso bloqueado permanecerá en esta situación hasta que algún otro proceso modifique alguna de las variables compartidas que le permitan continuar, El fragmento de código que debe ejecutar este otro proceso debe seguir el modelo siguiente:

```
lock(m) ;
/* código de la sección crítica */
/*se modifica la condición y ésta se hace TRUE */
```



```
condicion = TRUE;
c-signal(c);
unlock (m);
```

En este caso, el proceso que hace cierta la condición ejecuta la operación c_signal sobre la variable condicional despertando a un proceso bloqueado en dicha variable. Cuando el proceso ligero que espera en una variable condicional se desbloquea, vuelve a competir por el mutex. Una vez adquirido de nuevo el mutex debe comprobar si la situación que le despertó y que le permitía continuar su ejecución sigue cumpliéndose, de ahí la necesidad de emplear una estructura de control de tipo while, Es necesario volver a evaluar la condición ya que entre el momento en el que la condición se hizo cierta y el instante en el que comienza a ejecutar de nuevo el proceso bloqueado en la variable condicional puede haber ejecutado otro proceso que a su vez puede haber hecho falsa la condición.

El empleo de mutex y variables condicionales que se ha presentado es similar al concepto de monitor [Hoare, 1974], y en concreto a la definición de monitor dada para el lenguaje Mesa [Lampson, 1980].

En la Figura *5.14* se representa de forma grafica el uso de mutex y variables condicionales entre dos procesos tal y como se ha descrito anteriormente.

Productor-consumidor con mutex y variables condicionales

A continuación se presenta una posible solución al problema del productor-consumidor con buffer acotado, utilizando mutex y variables condicionales. En este problema el recurso compartido es el buffer y los procesos deben acceder a él en exclusión mutua. Para ello se utiliza un mutex sobre el que los procesos ejecutarán operaciones lock y unlock,

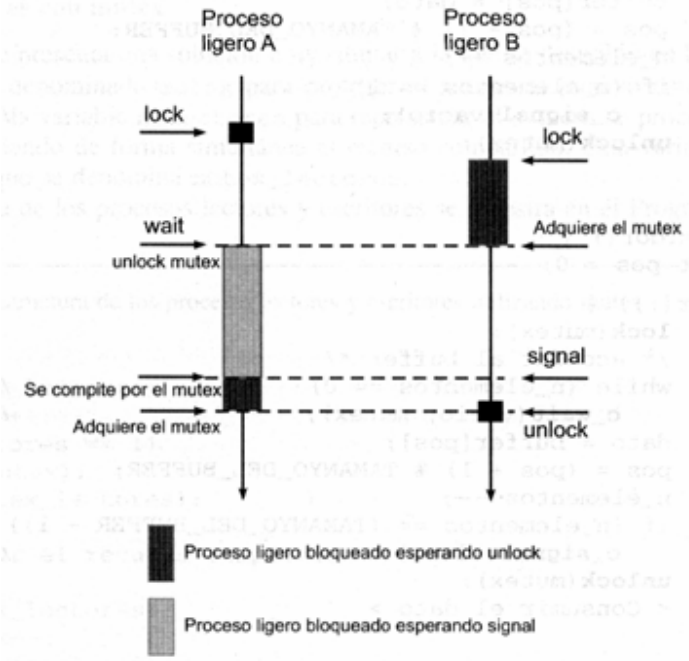


Figura ***5.14****. Ejemplo de mutex y variables condicionales,*

Hay dos situaciones en las que el proceso productor y el consumidor no pueden continuar su ejecución**,** una vez que han comenzado la ejecución del código correspondiente a la sección crítica:

- El productor no puede continuar cuando el buffer está lleno. Para que este proceso pueda bloquearse es necesario que ejecute una operación c_wait sobre una variable condicional que se denomina lleno,
- El proceso consumidor debe bloquearse cuando el buffer se encuentra vacío. En este caso se utilizará una variable condicional que se denomina vacio.

Para que ambos procesos puedan sincronizarse correctamente es necesario que ambos conozcan el número de elementos que hay en el buffer. Cuando el número de elementos es 0, el proceso consumidor deberá bloquearse. Por su parte, cuando el número de elementos coincide con el tamaño del buffer, el proceso productor deberá bloquearse. La variable n_elementos se utiliza para conocer el número de elementos insertados en el buffer.

El Programa *5.5* presenta, en pseudo código, la estructura general de los procesos

productor y consumidor utilizando mutex y variable, condicionales,

Programa *5.5.* Estructura de los procesos productor y consumidor utilizando mutex y variables condicionales

```
Productor ( ){
      int pos = 0;
      for(;;) {
            < Producir un dato >
            1ock (mutex);
              /* acceder al buffer */
            while (n_elementos = = TAMANYQ_DEL_BUFFER)          /* si buffer lleno
                  */
            c_wait (lleno, mutex);                                /* se bloquea */
            buffer[pos] = dato;
            pos = (pos + 1) % TAMANYQ_DEL_BUFFER;
            if( n_elementos = = 1)
               c_signal(vacio);                                   /* buffer no vacio
*/
            unlock(mutex);
      }
}

Consumidor( ) {
      int pos = 0;

      for(;;) (
          lock(mutex);
          /* acceder al buffer */
          while (n_elementos = = 0)                             /* si buffer vacio
*/
              c_wait(vacio, mutex);                             /* se bloquea */
          dato = buffer[pos];
          pos = (pos + 1) % TAMANYQ_DEL_BUFFER;
          n_elementos - - ;
          if (n_elementos = = (TAMANYO_DEL_BUFFER  – 1));
             c_signal(lleno);                                   /* buffer no vacio
*/
          unlock(mutex>;
          < Consumir el dato >
      }
}
```


El proceso productor evalúa la condición n_elementos == TAMANYO_DEL_BUFFER para determinar si el buffer está lleno. En caso de que sea así se bloquea ejecutando la función c_wait sobre la variable condicional lleno.

De igual forma, el proceso consumidor evalúa la condición n_elementos == 0 para determinar si el buffer esta vacío. En dicho caso se bloquea en la variable condicional vacio.

Cuando el productor inserta un primer elemento en el buffer, el consumidor podrá continuar en caso de que estuviera bloqueado en la variable vacio. Para despertar al proceso consumidor el productor ejecuta el siguiente fragmento de código:

```
if (n_elementos = = 1)
   c_signal (vacio);      /* buffer no vacío */
```

Cuando el proceso consumidor elimina un elemento del buffer y éste deja de estar lleno, despierta al proceso productor en caso de que estuviera bloqueado en la variable condicional lleno. Para ello el proceso consumidor ejecuta:

```
pos = (pos + 1) % TAMANYQ_DEL_BUFFER;
n_elementos - -;
if (n_elementos = = (TAMANYO_DEL_BUEFER – 1));
   c_signal(lleno);                    /* buffer no lleno */
```

Recuerde que, cuando se despierta un proceso de la operación c_wait, vuelve de nuevo a competir por el mutex, por tanto, mientras el proceso que le ha despertado no abandone la sección crítica y libere el mutex no podrá acceder a ella.

Lectores-escritores con mutex

En esta sección se presenta una solución muy similar a la que se describió en la Sección 5.3.4. Se utiliza un mutex denominado mutex para proteger el acceso al recurso compartido. De forma análoga se utiliza la variable n_lectores para representar el número de procesos lectores que se encuentran accediendo de forma simultánea al recurso compartido, Esta variable se protege mediante el mutex que se denomina mutex_lectores.

La estructura de los procesos lectores y escritores se muestra en el Programa *5.6*

Programa *5.6*. Estructura de los procesos lectores y escritores utilizando mutex.

```
Lector ( ) {
     lock(mutex_lectores);
     n_lectores++;
       if (n_lectores = = 1)
          lock(mutex);
     unlock(mutex_lectores>;

     < Consultar el recurso compartido >

     lock (mutex lectores);
     n_lectores --;
     if (n_lectores = = O)
          unlock (mutex);
```

```
            unlock(mutex_lectores);
    }
Escritor( ) {
     lock(mutex);
     < Modificar el recurso compartido >
     unlock(mutex);
}
```

5.4. PASO DE MENSAJES

Todos los mecanismos vistos hasta el momento necesitan que los procesos que quieren intervenir en la comunicación o quieren sincronizarse ejecuten en la misma máquina. Cuando se quiere comunicar y sincronizar procesos que ejecutan en máquinas distintas es necesario recurrir al *paso mensajes.* En este tipo de comunicación los procesos intercambian *mensajes* entre ellos. Es obvio que este esquema también puede emplearse para comunicar y sincronizar procesos que ejecutan en la misma máquina, en este caso los mensajes son locales a la máquina donde ejecutan los procesos

Utilizando paso de mensajes como mecanismo de comunicación entre procesos no es necesario recurrir a variables compartidas, únicamente debe existir un *enlace de comunicación* entre ellos. Los procesos se comunican mediante dos operaciones básicas:

- send(destino, mensaje): envía un mensaje al proceso destino.
- receive (origen, mensaje): recibe un mensaje del proceso origen.

De acuerdo con estas dos operaciones, las tuberías se pueden considerar en cierta medida como un mecanismo de comunicación basado en paso de mensajes. Los procesos pueden enviar un mensaje a otro proceso por medio de una operación de escritura y puede recibir mensajes de otros a través mediante una operación de lectura. En este caso, el enlace que se utiliza para comunicar a procesos es la propia tubería.

Existen múltiples implementaciones de sistemas con paso de mensajes. A continuación se describen algunos aspectos de diseño relativos a este tipo de sistemas.

Tamaño del mensaje

Los mensajes que envía un proceso a otro pueden ser de tamaño fijo o tamaño variable. En mensajes de longitud fija la implementación del sistema de paso de mensajes es mas sencilla, sin embargo, dificulta la tarea del programador ya que puede obligar a éste a descomponer los mensajes grandes en mensajes de longitud fija más pequeños.

Flujo de datos

De acuerdo al flujo de datos la comunicación puede ser **unidireccional** o **bidireccional.** Un enlace es unidireccional cuando cada proceso conectado a él únicamente puede enviar o recibir mensaje pero no ambas cosas. Si cada proceso puede enviar o recibir mensajes, entonces el paso de mensajes bidireccional.

Nombrado

Los procesos que utilizan mensajes para comunicarse o sincronizarse deben tener alguna forma de referirse unos a otros. En este sentido, la comunicación puede ser directa o indirecta.

La comunicación es **directa** cuando cada proceso que desea enviar o recibir un mensaje de otro debe nombrar de forma explícita al receptor o emisor del mensaje. En este esquema de comunicación, las operaciones básicas send y receive se definen de la siguiente manera:

- **send**(P, mensaje): envía un mensaje al proceso P.
- **receive** (Q, mensaje): espera la recepción de un mensaje por parte del proceso *Q*.

Existen modalidades de paso de mensajes con comunicación directa que permiten especificar al receptor la posibilidad de recibir un mensaje de cualquier proceso. En este caso, la operación receive se define de la siguiente forma:

receive(ANY, mensaje);

La comunicación es indirecta cuando los mensajes no se envían directamente del emisor al receptor, sino a unas estructuras de datos que se denominan *colas de mensajes* o *puertos*. Una cola de mensajes es una estructura a la que los procesos pueden enviar mensajes y de la que se pueden extraer mensajes. Cuando dos procesos quieren comunicarse entre ellos, el emisor sitúa el mensaje en la cola y el receptor lo extrae de ella. Sobre una cola de mensajes puede haber múltiples emisores y receptores.

Un puerto es una estructura similar a una cola de mensajes, sin embargo, un puerto se encuentra asociado a un proceso y por tanto únicamente puede recibir de un puerto un proceso. En este caso, cuando dos procesos quieren comunicarse entre si, el receptor crea un puerto y el emisor envía mensajes al puerto del receptor. La Figura *5.15* presenta la forma de comunicación utilizando colas de mensajes y puertos.

Utilizando comunicación indirecta las operaciones send y receive toman la siguiente forma:

- send(Q, mensaje): envía un mensaje a la cola o al puerto *Q*.
- receive (Q, mensaje): recibe un mensaje de la cola o del puerto *Q*.

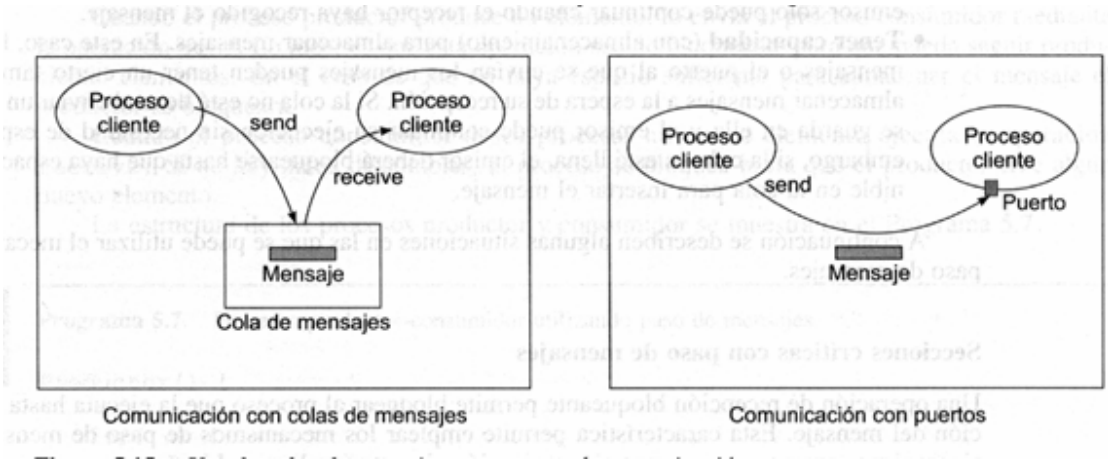


Figura *5.15*. *Uso de colas de mensajes y puertos en la comunicación entre procesos.*

Cualquiera que sea el método utilizado, el paso de mensajes siempre se realiza en exclusión mutua. Si dos procesos ejecutan de forma simultánea una operación send los mensajes no entrelazan, primero se envía uno y a continuación el otro. De igual forma, sí dos procesos desean recibir un mensaje de una cola, sólo se entregará el mensaje a uno de ellos.

Sincronización

La comunicación entre dos procesos es síncrona cuando los dos procesos han de ejecutar los servicios de comunicación al mismo tiempo, es decir, el emisor debe estar ejecutando la operación send y el receptor ha de estar ejecutando la operación receive. La comunicación es asíncrona en caso contrario.

En general, son tres las combinaciones más habituales que implementan los distintos tipos de paso de mensajes.

- **Envío y recepción bloqueante**. En este caso, tanto el emisor como el receptor se bloque hasta que tenga lugar la entrega del mensaje. Ésta es una técnica de paso de mensajes totalmente síncrona que se conoce como *cita.*
- **Envió no bloqueante y recepción bloqueante**. Ésta es la combinación generalmente mas utilizada. El emisor no se bloquea hasta que tenga lugar la recepción y por tanto puede continuar su ejecución. El proceso que espera el mensaje, sin embargo, se bloquea hasta que llega.
- **Envío y recepción no bloqueante**. Se corresponde con una comunicación totalmente asíncrona en la que nadie debe esperar. En este tipo de comunicación es necesario disponer servicios que permitan al receptor saber si se ha recibido un mensaje.

Almacenamiento

Este aspecto hace referencia a la capacidad del enlace de comunicaciones. El enlace y por tanto el paso de mensajes pueden:

- **No tener capacidad** (sin almacenamiento) para almacenar mensajes. En este caso, el mecanismo utilizado como enlace de comunicación no puede almacenar ningún mensaje y por tanto la comunicación entre los procesos emisor y receptor debe ser síncrona, es decir el emisor sólo puede continuar cuando el receptor haya recogido el mensaje.
- **Tener capacidad** (con almacenamiento) para almacenar mensajes. En este caso, la cola de mensajes o el puerto al que se envían los mensajes pueden tener un cierto tamaño para almacenar mensajes a la espera de su recepción. Si la cola no está llena al enviar un mensaje, se guarda en ella y el emisor puede continuar su ejecución sin necesidad de esperar. Sin embargo, si la cola ya esta llena, el emisor deberá bloquearse hasta que haya espacio disponible en la cola para insertar el mensaje.

A continuación se describen algunas situaciones en las que se puede utilizar el mecanismo de paso de mensajes.

Secciones críticas con paso de mensajes

Una operación de recepción bloqueante permite bloquear al proceso que la ejecuta hasta la recepción del mensaje. Esta característica permite emplear los mecanismos de paso de mensajes para sincronizar procesos, ya que una sincronización siempre implica un bloqueo.

Para resolver el problema de la sección crítica utilizando paso de mensajes se va a recurrir al empleo de colas de mensajes con una solución similar a la utilizada con las tuberías. Se creará una cola de mensajes con capacidad para almacenar un único mensaje que hará las funciones de testigo. Cuando un proceso quiere acceder al código de la sección crítica ejecutará la función receive para extraer el mensaje que hace de testigo de la cola. Si la cola esta vacía, el proceso se bloquea ya que en este caso el testigo lo posee otro proceso.

Cuando el proceso finaliza, la ejecución del código de la sección crítica inserta de nuevo el mensaje en la cola mediante la operación send.

Inicialmente, alguno de los procesos que van a ejecutar el código de la sección crítica deberá crear la cola e insertar el testigo inicial ejecutando algún fragmento con la siguiente estructura:

```
< Crear la cola de mensajes >
send(cola, testigo);                    /* insertaren la cola el testigo */
```

Una vez que todos los procesos tienen acceso a la cola, sincronizan su acceso a la sección crítica ejecutando el siguiente fragmento de código:

```
receive(cola, testigo);
< Código de la sección crítica >
send(cola, testigo);
```

De esta forma, el primer proceso que ejecuta la operación receive extrae el mensaje de la cola y la vacía, de tal manera que el resto de procesos se bloqueará hasta que de nuevo vuelva a haber un mensaje disponible en la cola. Este mensaje lo inserta el proceso que lo extrajo mediante la operación send. De esta forma, alguno de los procesos bloqueados se despertará y volverá a extraer el mensaje de la cola vaciándola.

Productor-consumidor con colas de mensajes

A continuación se presenta una posible solución al problema del productor-consumidor utilizando por paso de mensajes. Esta solución vuelve a ser similar a la que se presentó en la sección empleando tuberías.

Cuando el proceso productor produce un elemento, lo envía al proceso consumidor mediante operación send, Lo ideal es que esta operación sea no bloqueante para que pueda seguir prduciendo elementos. En el caso de que no haya espacio suficiente para almacenar el mensaje productor se bloqueará.

Cuando el proceso consumidor desea procesar un nuevo elemento ejecuta la operación receive. Si no hay datos disponibles, el proceso se bloquea hasta que el productor cree algún nuevo elemento.

La estructura de los procesos productor y consumidor se muestra en el Programa 5.7.

Programa *5.7.* Procesos productor-consumidor utilizando paso de mensajes.

```
Productor( ) {

    for(;;) {
        < Producir un dato >
```

```
        send(Consumidor, dato);
        }
}

Consumidor( ) {

    for(;;)
        receive(Productor, dato);
        < Consumir el dato >
    }
}
```

Paso de mensajes en esquemas cliente-servidor

El empleo más típico del paso de mensajes se encuentra en los esquemas cliente-servidor. En tipo de situaciones el proceso servidor se encuentra en un bucle infinito esperando la recepción de las peticiones de los clientes (operación receive). Los clientes solicitan un determinado enviando un mensaje al servidor (operación send).

Cuando el servidor recibe el mensaje de un cliente lo procesa, sirve la petición y devuelve el resultado mediante una operación send. Según se sirva la petición, los servidores se pueden clasificar de la siguiente manera:

- **Servidores secuenciales**. En este caso es el propio servidor el que se encarga de petición del cliente y devolverle los resultados. Con este tipo de servidores, sólo se puede atender a un cliente de forma simultánea.
- **Servidores concurrentes**. En este tipo de servidores, cuando llega una petición de un te, el servidor crea un proceso hijo que se encarga de servir al cliente y devolverle los resultados. Con esta estructura mientras un proceso hijo está atendiendo a un cliente, el proceso servidor puede seguir esperando la recepción de nuevas peticiones por parte de otros clientes. De esta forma se consigue atender a más de un cliente de forma simultánea

En la Figura 5.16 se representa de forma gráfica el funcionamiento de ambos tipos de servidores.

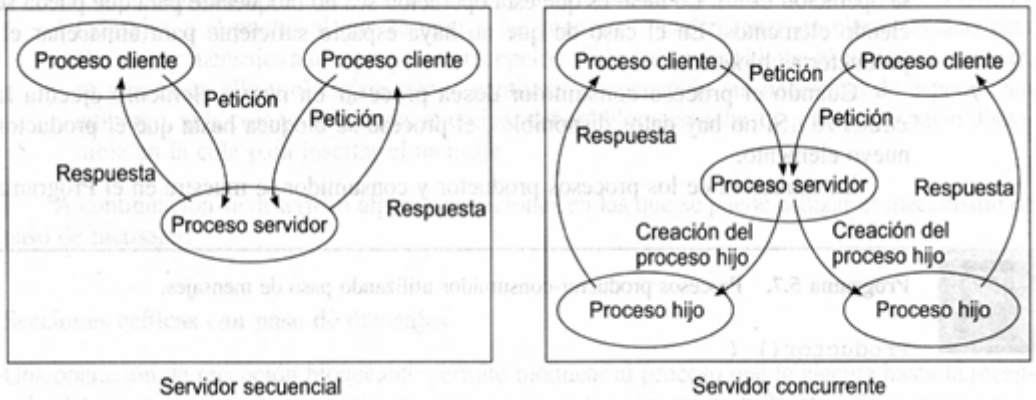


Figura 5.16. *Estructura de un servidor secuencial y concurrente.*

5.5. ASPECTOS DE IMPLEMENTACIÓN DE LOS MECANISMOS DE SINCRONIZACIÓN

Como se ha venido comentando a lo largo de este capítulo, todo mecanismo de sincronización conlleva un bloqueo bajo determinadas circunstancias. Este bloqueo se puede conseguir de dos formas. Estas dos posibilidades se van a ilustrar con las operaciones relativas a los semáforos.

Una posible forma de implementar las operaciones wait y signal sobre un semáforo es la que se muestra a continuación.

```
wait(s) {
    s = s – 1;
    while (s < 0)
        ;
}
signal (s) {
    s = s + 1;
}
```

Esta implementación coincide además con la definición original que se dio a los semáforos. En esta implementación se debe asegurar que las modificaciones del valor entero del semáforo en las operaciones wait y signal se ejecutan de forma indivisible. Además, en el caso de la operación wait la comparación debe realzarse también de forma atómica. Con esta implementación, todo proceso que se encuentra con un valor del semáforo negativo entra en un bucle en el que evalúa de forma continua el valor del semáforo. A esta situación se le denomina **espera activa.**

Cuando se emplea espera activa para bloquear a los procesos, éstos deben ejecutar un ciclo continuo hasta que puedan continuar . La espera activa es obviamente un problema en un sistema multiprogramado real, ya que se desperdicia ciclos del procesador en aquellos procesos que no puede ejecutar y que no realizan ningún trabajo útil.

Para solucionar el problema que plantea la espera activa se puede modificar la implementación de las operaciones wait y signal, utilizando la que se vio en la Sección *5.7.*

```
wait (s) {
    s = s – 1;
    if (s < 0)
        Bloquear al proceso;
}

signal (s) {
    s = s + 1;
    if ( s <= 0)
        Desbloquear a un proceso bloqueado un la operación wait;
}
```

En este caso, cuando un proceso no puede continuar la ejecución se **bloquea** suspendiendo su ejecución. A este modelo de espera se le denomina **espera pasiva**. La operación de bloqueo coloca al proceso es una cola de espera asociada al semaforo. Este bloqueo transfiere el control al planificador del sistema operativo, que seleccionara otro proceso para su ejecución. De esta forma no se desperdician ciclos de procesador en ejecutar operaciones inútiles.

Cuando un proceso ejecuta una operación signal y se encuentra con un valor menor o igual que cero, se elegirá al primer proceso de la cola asociado al semáforo y se

cambiará su estado de bloqueado a listo para ejecutar.

Estos conceptos, que se han visto para un semáforo, se pueden aplicar de igual forma para el resto de mecanismos de sincronización.

5.5.1. Implementación de la espera pasiva

Para implementar un semáforo, o cualquier otro tipo de mecanismo de sincronización, utilizando espera pasiva, basta con asignar al semáforo (o al mecanismo de sincronización concreto) una lista de procesos, en la que se irán introduciendo los procesos que se bloqueen. Una forma sencilla de implementar esta lista de procesos es mediante una lista cuyos elementos apunten a las entradas correspondientes de la tabla de procesos.

Cuando debe bloquear un proceso en el semáforo, el sistema operativo realiza los siguientes pasos:

- Inserta al proceso en la lista de procesos bloqueados en el semáforo,
- Cambia el estado del proceso a bloqueado.
- Llama al planificador para elegir otro proceso a ejecutar y a continuación al activador para ponerlo a ejecutar.

En la Figura 5.17 se muestra lo que ocurre cuando un proceso (con identificador de proceso 11) ejecuta una operación wait sobre un semáforo con valor -l (Fig. 5.17a). Como el proceso de bloquearse, el sistema operativo añade este proceso a la lista de procesos bloqueados en el semáforo y pone su estado como bloqueado (Fig. *5.* 17b).

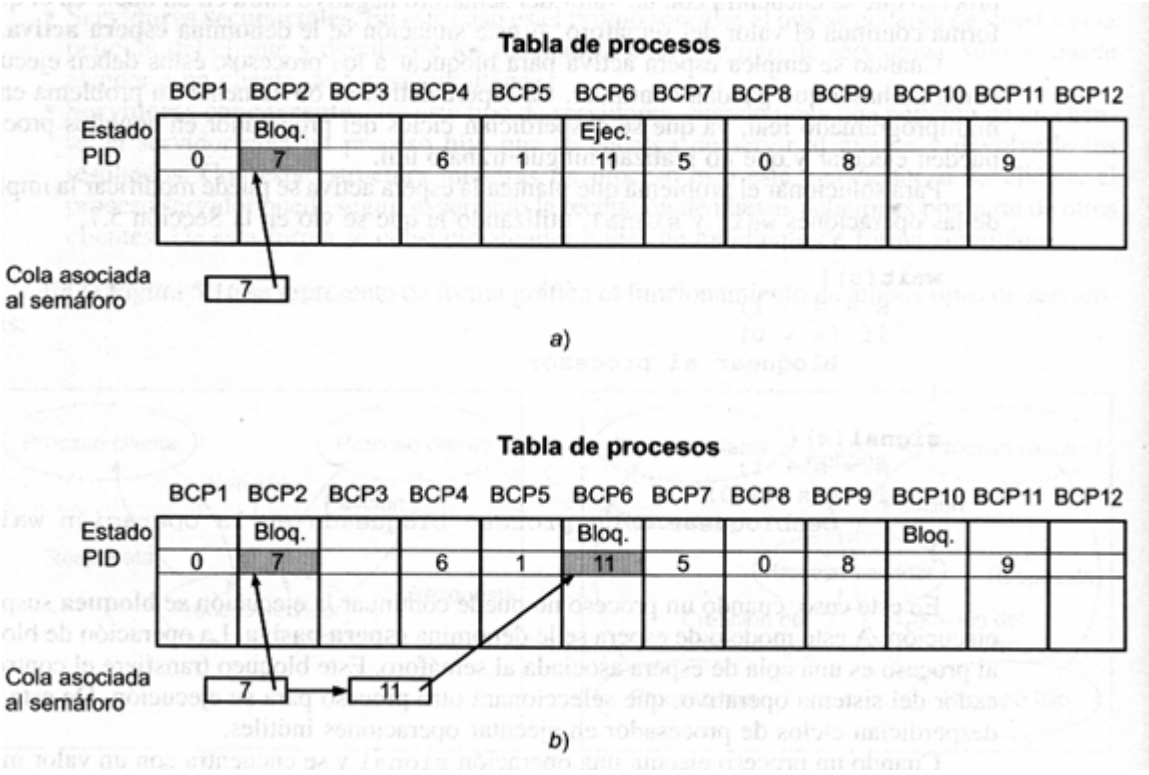


Figura 5.17. *Acciones realizadas por el sistema operativo cuando se debe bloquear a un proceso en semáforo.*

Cuando se realiza una operación signal sobre el semáforo, el sistema operativo realiza las siguientes acciones:

- Extrae al primer proceso de la lista de procesos bloqueados en el semáforo.
- Cambia el estado del proceso a listo para ejecutar.
- Llama al planificador para elegir a otro proceso (que puede ser el que llama a signal o el que se despierta) y a continuación al activador.

Estas acciones se describen en la Figura 5.18.

Aparte de estas operaciones, un aspecto importante en la implementación de un semáforo (y de otros mecanismos de sincronización) es que las operaciones wait y signal deben ejecutarse de forma atómica, es decir, en exclusión mutua. Para conseguir esta exclusión mutua, los sistemas operativos utilizan normalmente instrucciones hardware especiales que permiten resolver el problema de la sección crítica. Dos ejemplos de instrucciones hardware de este tipo son: test-and-set y swap. Ambas operaciones se ejecutan de forma atómica. La definición de la instrucción test-and-set es la siguiente:

```
int test-and-set(lnt *valor) {
    int temp;

    temp=*valor;
    *valor              = 1;        /*True*/
    return temp;
}
```

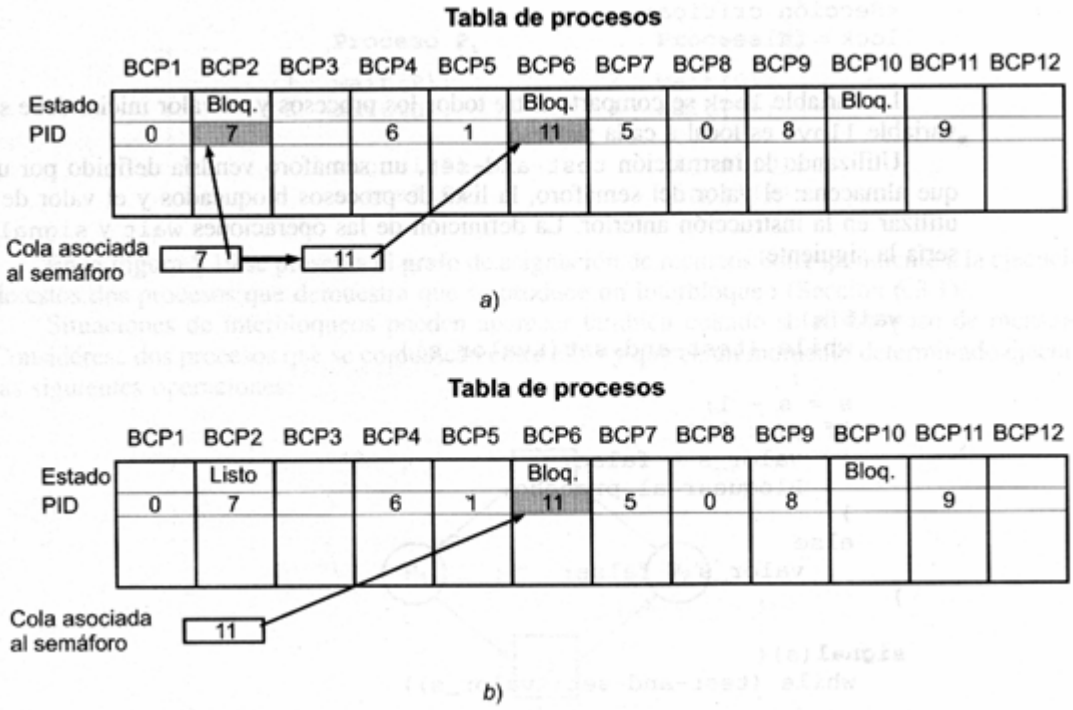


Figura 5.18. *Acciones realizadas por el sistema operativo en una operación signal.*

La definición de la instrucción swap es:

```
void swap(lnt *a, lnt *b) {
    int temp;
    temp = *a;
    *b= temp;
    return;
}
```

Estas instrucciones se pueden utilizar para resolver el problema de la sección crítica. Utilizando la instrucción test-and-set, el fragmento de código que resuelve el problema de la sección crítica es el siguiente:

```
while (tast-and-set(&lock))
    ;
<Sección crítica >
lock = false;
```

Si se utiliza la instrucción swap, el problema de la sección crítica debe resolverse de la siguiente manera:

```
llave = true;
do
    swap(lock, llave);
    while (llave != false);
    <Sección crítica>
    lock = false;
```

La variable lock se comparte entre todos los procesos y su valor inicial debe ser **false** .La variable llave es local a cada proceso.

Utilizando la instrucción test-and-set, un semáforo vendría definido por una estructura que almacena: el valor del semáforo, la lista de procesos bloqueados y el valor de la variable a utilizar en la instrucción anterior. La definición de las operaciones wait y signal en este sería la siguiente:

```
wait(s) {
    while (test-and-set (&valors))
        ;
    s = s – 1;

    if (s < 0){
        valor_s = false;
        Bloquear al proceso;
    }
    else
        valor_s = false;
}

signal (s) {
    while (test-and-set(&valors))
        ;
    s = s + 1;
```

```
        if ( s <= 0)
          Desbloquear a un proceso bloqueado en la operación wait;
        valor_s = false;
}
```

Con la implementación anterior no se ha eliminado del todo la espera activa, pero, sin embargo, se ha reducido a porciones de código muy pequeñas de la sección crítica de los códigos de las operaciones wait y signal. Lo importante es que los procesos, cuando se bloquean en la operación wait, no realizan espera activa.

5.6. INTERBLOQUEOS

En el Capítulo 2 se presentó el concepto de ínterbloqueo. Un ínterbloqueo supone un bloqueo permanente de un conjunto de procesos que compiten por recursos o bien se comunican o sincronizan entre sí.

Los interbloqueos que aparecen cuando se utilizan mecanismos de comunicación y sincronización se deben a un mal uso de los mismos, A continuación se van a presentar ejemplos de mal utilización de mecanismos de comunicación y sincronización que llevan a los procesos a un estado de ínterbloqueo.

Considérese en primer lugar una situación en la que se utilizan dos semáforos P y *Q,* ambos con un valor inicial 1. Si dos procesos utilizan estos dos semáforos de la siguiente manera, se produce un ínterbloqueo.

	Proceso P_1		**Proceso P_2**
1	wait(P);	2	wait (Q);
3	wait(Q);	4	wait(P);
	…..		….
	signal(P>;		signal(Q);
	signal(Q);		signal(P);

En la Figura 5.19 se presenta el grafo de asignación de recursos correspondiente a la ejecución de estos dos procesos que demuestra que se produce un ínterbloqueo (Sección 6.3.1).

Situaciones de interbloqueos pueden aparecer también cuando se utiliza paso de mensajes. Considérese dos procesos que se comunican entre ellos y que en un momento determinado ejecutan las siguientes operaciones:

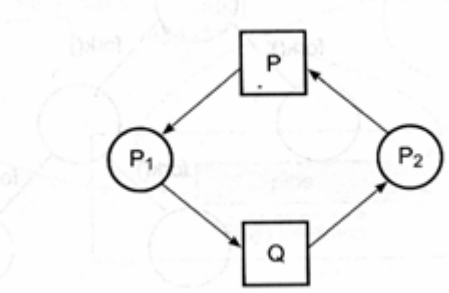


Figura 5.19. *Ejemplo de ínterbloqueo utilizando semáforos.*

Proceso. P_1	**Proceso P_2**
.....	....
receive (P_2, m)	receive(P_1,m);
send(P_2,m)	send(P_1,m);

Esta situación también lleva a un ínterbloqueo, ya que los dos procesos se encuentran do la recepción de un mensaje que nunca llegará.

Las dos situaciones descritas anteriormente se deben a errores en el diseño de los Ello obliga a un diseño muy cuidadoso de las aplicaciones. En el Capítulo 6 se analizará con detalle el problema de los interbloqueos y la forma de tratarlos.

5.7. SERVICIOS POSIX

En esta sección se presentan los servicios que ofrece POSIX para los distintos mecanismos comunicación y sincronizacion de procesos que se han ido presentando a lo largo del c Unicamente se van a trat los mecanismos más adecuados p a la comunicación y sincronizac~ de procesos. Los servicios de sincronización pura incluyen los semáforos, los mutex y las var condicionales. Para comunicar procesos se pueden emplear tuberías y colas de mensajes.

5.7.1. Tuberias

En POSIX existen tuberías sin nombre, o simplemente pipe~, y tuberías con nombre, o FIFOS.

Un pipe no tiene nombre y, por tanto, sólo puede ser utilizado entre los procesos que hereden a través de la llamada fork () . La Figura *5.20* muestra lajer quía de procesos que pue utilizar un mismo pipe.

Para leer y escribir de una tubería en POSIX se utilizan descriptores de archivo. Las tub sin nombre tienen asociados dos descriptores de archivo. Uno de ellos se emplea para leer y el para escribir. Un FIFO sólo tiene asociado un descriptor de archivo que se puede utilizar p a escribir. A continuación se describen los servicios POSIX relacionados con las tuberías,

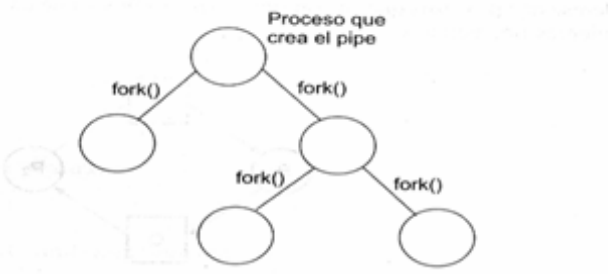


Figura 5.20. *Jerarquía de proceso. que pueden compartir un mismo pipe en POSJX.*

Crear una tubería sin nombre

Este servicio permite crear una tubería. Su prototipo es el siguiente:

int **pipe**(int fildes[2];

Esta llamada devuelve dos descriptores de chivos (Fig. 5.21) que se utilizan como identificadores:

- fildes [0]: descriptor de archivo que se emplea p a leer del pipe.
- fildes [1]: descriptor de archivo que se utiliza para escribir en el pipe.

La llamada pipe devuelve 0 si fue bien y —1 en caso de error.

Crear una tubería con nombre

En POSIX, las tuberías con nombre se conocen como FIFO. Los PIFO tienen un nombre local que lo identifican dentro de una misma máquina. El nombre que se utiliza corresponde con el de un archivo. Esta característica permite que los PIFO puedan utilizarse para comunicar y sincronizar procesos de la misma máquina, sin necesidad de que lo hereden por medio de la llamada fork. El prototipo del servicio que permite cre una tubería con nombre es el siguiente:

int **mkfifo**(char *fifo mode_t mode);

El primer argumento representa el nombre del FIFO. El segundo argumento representa los permisos asociados al FIFO. La llamada devuelve 0 si se ejecutó con éxito o -1 en caso de error.

Abrir una tubería con nombre

El servicio que permite abrir una tubería con nombre es open. Este servicio también se emplea para abrir archivos. Su prototipo es el siguiente:

int open(char *fifo, mt flag);

El primer argumento identifica el nombre del PIFO que se quiere abrir y el segundo la forma en la que se va a acceder al PIFO. Los posibles valores de este segundo argumento son:

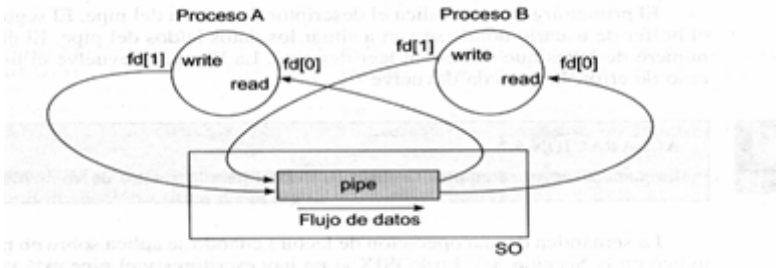


Figura 5.21. *Tuberías POSIX entre dos procesos.*

- O_RDONLY: se abre el FIFO para realizar solo operaciones de lectura.
- O_WRQNLY: se abre FIFO para realizar sólo operaciones de escritura.
- O_RDWR: se abre el FIFO para lectura y escritura.

El servicio open devuelve un descriptor de archivo que se puede utilizar para leer y escribir del FIFO. En caso de error devuelve -1. La llamada open bloquea al proceso que la ejecuta hasta; que haya algún otro proceso en el otro extremo del FIFO.

Cerrar una tubería

Este servicio cierra un descriptor de archivo asociado a una tubería con o sin nombre. También se emplea para cerrar cualquier archivo. Su prototipo es:

int **close**(int fd);

El argumento de e lose indica el descriptor de archivo que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error devuelve -1.

Borrar una tubería con nombre

Permite borrar un FIFO. Esta llamada también se emplea para borrar archivos. Su prototipo es:

int **unlink**(char *fifo);

Esta llamada pospone la destrucción del FIFO hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función close. En el caso de una tubería sin nombre, ésta se destruye cuando se cierra el último descriptor que tiene asociado.

Leer de una tubería

Para leer datos de un pipe o un FIFO se utiliza el siguiente servicio:

int read(int fd, char *butfer mt n); (Aclaración *5.7)*

El primer argumento indica el descriptor de lectura del pipe. El segundo argumento especifica el buffer de usuario donde se van a situar los datos leídos del pipe. El último argumento indica número de bytes que se desean leer del pipe. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve -1.

ACLARACIÓN 5.7

La llamada `read` se emplea también en POSIX para leer datos de un archivo.

La semántica de una operación de lectura cuando se aplica sobre un pipe en POSIX es la que indicó en la Sección 5.3. En POSIX si no hay escritores y el pipe está vacío, la llamada devuelve cero, indicando fin de archivo (en este caso la llamada no bloquea al proceso).

La lectura sobre un pipe en POSIX es atómica cuando el número de datos que se desean leer es menor que el tamaño del pipe.

Escribir en una tubería

El servicio para escribir datos en una tubería en POSIX es:

int **write**(int fd, char *buffer int n); (Aclaración 5.8)

El primer argumento representa el descriptor de archivo que se emplea para escribir en un pipe. El segundo argumento especifica el buffer de usuario donde se encuentran los datos que se van a escribir al pipe. El último argumento indica el número de bytes a escribir. Los datos se escriben en el pipe en orden FIFO.

ACLARACIÓN 5.8

La llamada `write` también se utiliza en POSIX para escribir archivos.

La semántica del servicio write aplicado a un pipe es la que se vio en la Sección *5.3*. En POSIX, cuando no hay lectores y se intenta escribir en una tubería, el sistema operativo envía la señal SIGPIPE al proceso.

Al igual que las lecturas, las escrituras sobre un pipe son atómicas. En general esta atomicidad se asegura siempre que el número de datos involucrados en la operación sea menor que el tamaño del pipe.

A continuación se van a emplear las tuberías POSIX para resolver alguno de los problemas descritos en la Sección 5.2,

Sección crítica con tuberías POSIX

En esta sección se describe la forma de resolver el problema de la sección crítica utilizando tuberías de POSIX.

Uno de los procesos debe encargarse de crear la tubería e introducir el testigo en la misma. Como testigo se utilizará un simple carácter. El fragmento de código que se debe utilizar es el siguiente:

```
int fildes[2];                          /* tubería utilizada para sincronizar */
char testigo;                           /*se declara un carácter como testigo */

pipe(fildes);                           /* se crea la tubería */
write(fildes [1], &testigo, 1);         /* se inserta el testigo en la tubería */
```

Una vez creada la tubería e insertado el testigo en ella, los procesos deben proteger el código correspondiente a la sección crítica de la siguiente forma:

```
read (fildes[0], &testigo, 1);
< código correspondiente a la sección critica >
write(fildes[l], &testígo, 1);
```

La operación read eliminará el testigo de la tubería y la operación write lo insertara de nuevo en ella.

Productor-consumidor con tuberías

El Programa *5.8* muestra un fragmento de ejemplo que se puede utilizar para resolver problemas de tipo productor-consumidor mediante las tuberías que ofrece POSIX. En este ejemplo se crea un proceso hijo por medio de la llamada fork. A continuación, el proceso hijo hará las veces productor y el proceso padre de consumidor.

Programa 5.8. Productor-consumidor con tuberías POSIX.

```
#include <stdio,h>
#include <unistd.h>
struct elemento dato;                    /* dato a producir */
int fildes[2];                           /* tubería */

if (pipe(fildes) < 0){
    perror("Error al crear la tubería");
    exit(l);
}
if (fork ( )= =0){                       /* proceso hijo: productor */
    for (;;) {
        < produce algún dato de tipo struct elemento >
          write(fildes[l],      (char *) &dato, sizeof(struct elemento));
    }
}else {                                  /* proceso padre: consumidor */
    for(;;) {
        read(fildes[0], (char *) &dato. sizeof(struct elemento));
        < consumir el dato leído >
    }
}
```

Ejecución de mandatos con tuberías

Aunque en las secciones anteriores se han presentado dos posibles utilizaciones de las tuberías, uso más extendido se encuentra en la ejecución de mandatos con tuberías. A continuación se presenta un programa que permite ejecutar el mandato `ls | wc`. La ejecución de este mandato supone la ejecución de los programas `ls` y `wc` de UNIX y su conexión mediante una tubería. El código que permite la ejecución de este mandato es el que se muestra en el Programa 5.9.

Programa 5.9. Programa que ejecuta ls 1 wc.

```
#include <sys/types .h>
 #include <stdio.h>
#include <unistd.h>
```

```
void main(void) {
      int fd[2];
    pid_t pid;

      /*se crea la tubería */
      if (pipe(fd) < 0) {
          perror("Error al crear la tubería");
          exit(0);
      }

pid = fork( );
switch (pid) {
case -1: /* error */
      perror("Error en el fork")
      exit (0);
case 0: /* proceso hijo ejecuta ls */
      close (fd[0]);
      close(STDOUT_FILENO);
      dup(fd [1])
      close(fd [1]);
      execlp("ls". "ls", NULL);
      perror("Error en el exec");
       break;
default: /* proceso padre ejecuta wc */
      close(fd [1]);
      close(STDIN_FILENO);
      dup(fd[0])
      close(fd[0]);
      execlp("'wc", "wc", NULL);
      perror("Error en el exec");
}
}
```

El proceso hijo (Fig. 5.22) redirige su salida estándar a la tubería. Por su parte, el proceso padre redirecciona su entrada estándar a la tubería. Con esto se consigue que el proceso que ejecuta el programa ls escriba sus datos de salida en la tubería y el proceso que ejecuta el programa wc lea sus datos de la tubería.

Los pasos que realiza el proceso hijo para redirigir su salida estándar a la tubería son los siguientes:

- Cierra el descriptor de lectura de la tubería, fd E 0], ya que no lo utiliza (close (fd [0])).
- Cierra la salida estándar **,** que inicialmente en un proceso referencia el terminal (close (STDOUT_FILENO)). Esta operación libera el descriptor de archivo 1, es decir, el descriptor STDOUT_FILENO.
- Duplica el descriptor de escritura de la tubería mediante la sentencia dup (fd [1]) (Aclaración 5.9). Esta llamada devuelve y consume un descriptor, que será el de número más bajo disponible, en este caso el descriptor 1 que coincide en todos los procesos como el descriptor de salida estándar. Con esta operación se consigue que el descriptor de archivo 1 y el descriptor almacenado en fd [l] sirvan para escribir datos en la tubería. De esta forma se ha conseguido redirigir el descriptor de salida estándar en el proceso hijo a la tubería.

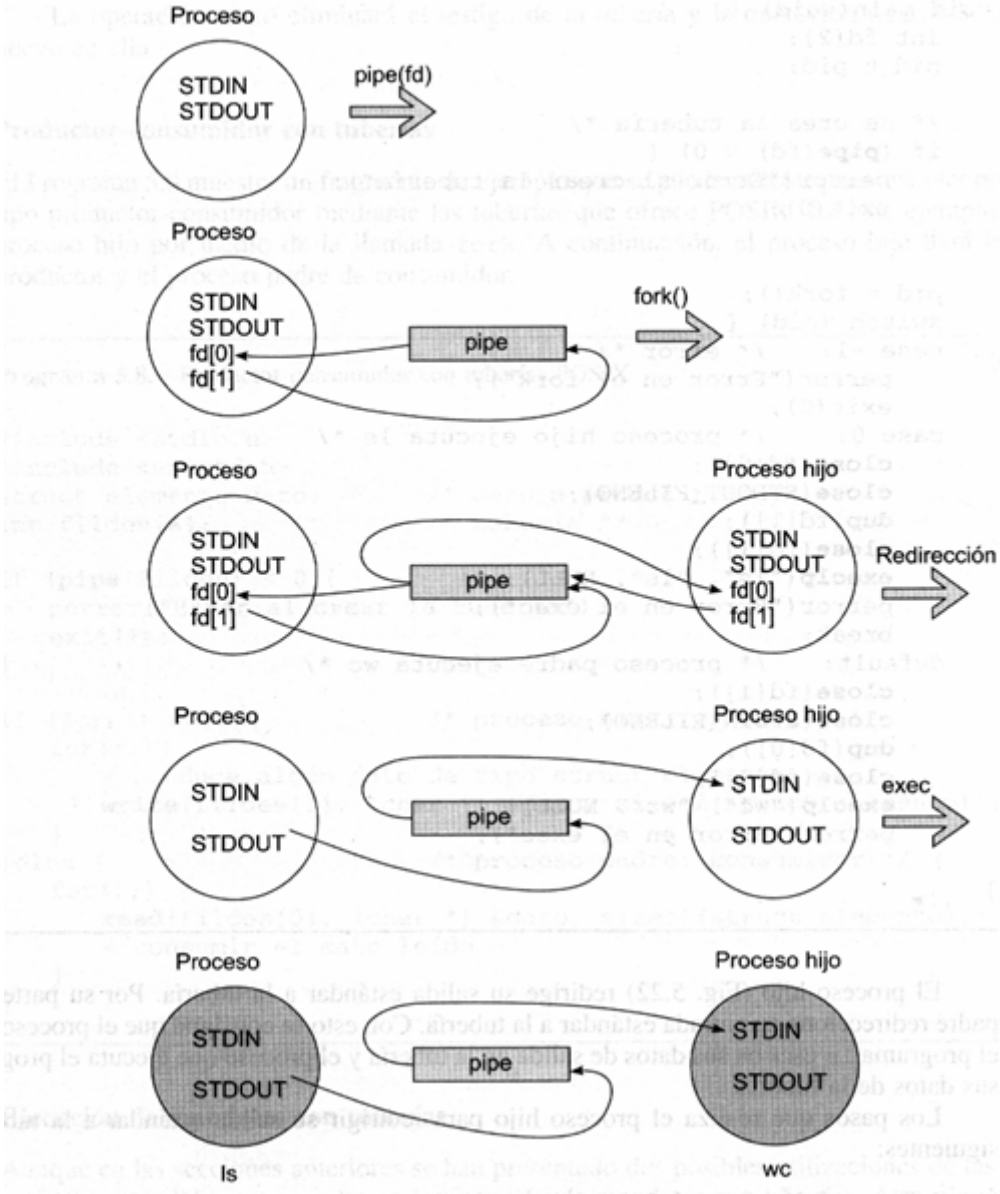


Figura 5.22 Ejecución de mandatos con tuberías.

- Se cierra el descriptor fd [1], ya que el proceso hijo no lo va a utilizar en adelante. Recuérdese que el descriptor 1 sigue siendo válido para escribir datos en la tubería.
- Cuando el proceso hijo invoca el servicio exec para ejecutar un nuevo programa, se conser-va en el BCP la tabla de descriptores de archivos abiertos y, en este caso, el descriptor de salida estándar 1 está referenciando a la tubería. Cuando el proceso comienza a ejecutar el código del programa ls, todas las escrituras que se hagan sobre el descriptor de salida estándar se harán realmente sobre la tubería.

ACLARACIÓN 5.9

El servicio **dup** de POSIX duplica un descriptor de archivo abierto. Su prototipo es:

```
int dup(int fd);
```

El servicio dup duplica el descriptor de archivo fd. La llamada devuelve el nuevo descriptor de archivo. Este descriptor de archivo referencia al mismo archivo al que referencia fd.

5.7.2. Semáforos POSIX

Las operaciones wait y signal son dos operaciones genéricas que deben particularizarse en cada sistema operativo. A continuación se presentan los servicios que ofrece el estándar POSIX para trabajar con semáforos.
En POSIX un semáforo se identifica mediante una variable del tipo sem_t. El estándar POS IX define dos tipos de semáforos:

- Semáforos sin nombre. Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso, o a los procesos que lo heredan a través de la llamada fork.
- Semáforos con nombre. En este caso, el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden
  sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada fork.

La diferencia que existe entre los semáforos con nombre y sin nombre es similar a la que existe entre los pipes y los FIFOS.

Los servicios POSIX para manejar semáforos son los siguientes:

Crear un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función sem mit permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

mt sem init(sem t ~ mt shared, mt val);

Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar procesos ligeros o cualquier otro tipo de proceso. Si shared es O, el semáforo sólo puede utilizarse entre los procesos ligeros creados dentro del proceso que inicia el semáforo. Si shared es distinto de O, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada fork. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

Destruir un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada sem_init. Su prototipo es el siguiente:

int sem destroy(sem _t *sem);

Crear y abrir un semáforo con nombre

El servicio sem_open permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades según se utilice para crear el semáforo o simplemente abrir uno existente. Estas modalidades son las siguientes:

sem_t ***sem_open**(char *name, int flag, mode_t mode, int val);
sem_t ***sem_open**(char *name, mt flag);

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo. El nombre de un semáforo es una cadena de caracteres que sigue la convención de nombrado de un archivo. La función sem_open establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función sem_open accede a un semáforo previamente creado o si crea uno nuevo. Un valor 0 en flag indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función sem_open. Si flag tiene un valor o CREAT, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la llamada open para archivos. El cuarto parámetro especifica el valor inicial del semáforo.

POSIX no requiere que los semáforos con nombre se correspondan con entradas de directorio en el sistema de archivos, aunque sí pueden aparecer.

Cerrar un semáforo con nombre

Cierra un semáforo con nombre, rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

int **sem_close**(sem t *sem);

Borrar un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función sem_close. El prototipo de este servicio es:

int **sem_unlink**(char *name)

Operación wait

La operación wait en POSIX se consigue con el siguiente servicio:

ini sem_wait(sem t *sem)

Operación signal

Este servicio se corresponde con la operación signal sobre un semáforo. El prototipo de este servicio es:

int **sem_post**(sem t *sem);

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o -1 en caso de error. En este caso se almacena en la variable errno el código que identifica el error.

Productor-consumidor con semáforos POSIX

A continuación se presenta la solución a un problema de tipo productor-consumidor utilizando semáforos POSIX y memoria compartida mediante archivos proyectados en memoria. El proceso productor genera números enteros. El consumidor consume estos números imprimiendo su valor por la salida estándar.

En este ejemplo se emplean procesos convencionales creados con la llamada fork. Dado que este tipo de procesos no comparten memoria de forma natural, es necesario crear y utilizar un segmento de memoria compartida.

En este caso se va a utilizar un buffer que reside en un segmento de memoria compartida y semáforos con nombre.

En la solución propuesta el productor se va a encargar de:

- Crear los semáforos mediante el servicio sem_open.
- Crear la zona de memoria compartida utilizando un archivo proyectado en memoria mediante la llamada open.
- Asignar espacio al archivo creado. Para ello se emplea el servicio ftruncate, que permite asignar espacio a un archivo o a un segmento de memoria compartida.
- Proyectar el segmento de memoria compartida sobre su espacio de direcciones utilizando la llamada mmap.
- Acceder a la región de memoria compartida para insertar los elementos que produce.
- Desproyectar la zona cuando ha finalizado su trabajo mediante el servicio munmap.
- Por último, este proceso se encarga de cerrar, mediante la llamada close, y destruir el archivo de memoria compartida previamente creado utilizando el servicio unlink.

Los pasos que deberá realizar el proceso consumidor en la solución propuesta son los siguientes:

- Abrir los semáforos que se van a utilizar.
- Abrir el archivo proyectado en memoria creado por el productor (open). Esta operación debe realizarse una vez creada la región. En caso contrario, la función open devolvería un

error.

- Proyectar la zona de memoria compartida en su espacio de direcciones (mmap).
- Acceder a la región de memoria compartida para eliminar los elementos de buffer.
- Desproyectar el segmento de memoria de su espacio de direcciones (munmap).
- Cerrar el objeto de memoria compartida (close).

El código a ejecutar por el proceso productor es el que se presenta en el Programa 5.10.

Programa 5.10. Código del proceso productor utilizando memoria compartida y semáforos POSIX.

```
#include <sys/mmap.h>
#include <stdio.h>
```

```
#include <pthread.h>
#include <semaphore.h>

#define MAX_BUFFER                1024        / * tamaño del buffer */
#define DATOS_A_PRODUCIR          100000          /* datos a producir */

sem_t     *huecos;
sem_t      *elementos;
       int buffer;                              /*puntero al buffer de números enteros */

void main(void){
int shd;

/*se crean e inician semáforos */
huecos = sem_open("HUECOS", OCREAT, 0700 MAX_BUFFER);

elementos = sem_open("ELEMENTOS", OCREAT, 0700, 0);

if (huecos = = -1 ‖ elementos == -1) {
perror("Error en sem_open")
exit (1)
}

/* se crea el segmento de memoria compartida utilizado como
   buffer circular */
shd = open("BUFFER" O_CREATIOWRONLY, 0700);
      if (shd = = -1)
         perror ( "Error en open");
         exit (1)
}

ftruncate(shd, MAX_BUFFER*sizeof (int));
buffer = (int *)mmap(NULL, MAXBUFFER*sjzeof(int), PROT_WRITE,
                                    MAP_SHARED, shd, 0);
if (buffer = = NULL)
perror ("Error en mmap");
exit (1);
}

productor ( )     /* se ejecuta el código del productor */

/*se desproyecta el buffer */
munmuap(buffer, MAX_EUFFER*sizeof (int));
close(shd);
unlink("BUFFER");

/* cierran y se destruyen los semáforos */
sem_close (huecos);
sem_close (elementos);
sem_unlink("HUECOS";
sem_unlink("ELEMENTOS");
exit(0)
```

```
}
```



```
/* código del proceso productor */
void productor (void)
        int dato;                   /* dato a producir */
        int posicion = 0;           /* posición donde insertar el elemento */
        int j;


for (j = 0; j<DATOS_A_PRODUCIR; j++)
        dato = j;
        sem_wait (huecos);              /* un hueco menos */
        buffer[posición] =dato;
        posición =(posición +l) % MAXBUFFER; /* nueva posición */
        sem_post(elementos);/* un elemento más */;
}
return;

}
```

El proceso productor se encarga de crear una región de memoria compartida que denomina BUFFER. También crea los semáforos con nombre HUECOS y ELEMENTOS. Los permisos que asigna a la región de memoria compartida y a los semáforos vienen dados por el valor 0700, lo que significa que sólo los procesos que pertenezcan al mismo usuario del proceso que los creó podrán acceder a ellos para utilizarlos.

El código que ejecuta el proceso consumidor se muestra en el Programa 5.11.

Programa 5.11. Código del proceso consumidor utilizando memoria compartida y semáforos POSIX.

```
#include <sys/mmap.h> #include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define MAX_BUFFER                1024        /* tamaño del buffer */
#define DATOS_A_PRODUCIR     100000          /* datos a producir */

sem_t         *huecos;
sem_t        *elementos;
int *buffer ; /* buffer de números enteros */

void main(void){
int shd;

/* se abren los semáforos */
huecos = sem_open(”HUECOS”, 0);
elementos = sem_open(”ELEMENTOS”, 0);
if (huecos = = -1 ▏▏ elementos = = -1)
   perror (“Error en sem_open”)
   exit(l)
}
```

```
/*se abre el segmento de memoria compartida utilizado como
     buffer circular */
shd = open(”BUFFER”, O_RDONLY);
if (shd = = -1)
perror(”Error en open”)
}
exit (0);
buffer = (int *)mmap(NULL MAX_BUFFER*sizeof(int),
PROTREAD,MAP_SHARED, shd, O);
if (buffer = = NULL){
perror (“Error en mmap”)
exit (1)


consumidor( ); /* se ejecuta el código del consumidor */


/* se desproyecta el buffer */
 munmap(buffer, MAX_BUFFER*sizeof (int));
close(shd);

/*se cierran semáforos */
sem_close (huecos);
sem_close(elementos);
exit (0)


/*código del proceso productor */
void consumidor (void){
   int dato;              /*dato a consumir */
    int posición = 0;  /* posición que indica el elemento a extraer */
    int j;


for ( j = 0; j<DATOS_A_PRODUCIR; j++)
     dato =j ;
     sem_wait(elementos); /* un elemento menos */
     dato = buffer[posicion];
     posición = (posición +l) % MAXBUFFER; /* nueva posición */
     sem_post (huecos); /* un hueco más);
     }
return;


}
```

5.7.3. Mutex y variables condicionales en POSIX

En esta sección se describen los servicios POSIX que permiten utilizar mutex y variables condicio-nales.

Para utilizar un mutex un programa debe declarar una variable de tipo pthread mutex_t (definido en el archivo de cabecera pthread.h) e iniciarla antes de utilizarla.

- Acceder al segmento de memoria compartida descrito por el archivo para eliminar los elementos que produce el productor.
- Desproyectar el archivo del espacio de direcciones (unmapViewofFile).
- Cerrar el archivo (CloseHandle).

El Programa 5.20 muestra el código que ejecuta el proceso productor.

Programa 5.20, Código del proceso productor utilizando semáforos Win32 y memoria compartida.

```
#include <windows .
#include <stdio,h>

#define MAX_BUFFER                    1024      /* tamaño del buffer */
#define DATOS_A_PRODUCIR            100000      /*datos a producir */

int main(void)
{
   HANDLE huecos, elementos;
   HANDLE hIN, hINMap;
   int *buffer;

   huecos = CreateSemaphore(NULL, MAX_BUFFER, MAXBUFFER, “HUECOS”);
   elementos = CreateSemaphore(NULL, 0, MAIQBUFFER, “ELEMENTOS”);
   if (huecos = = NULL || elementos = = NULL) {
      printf (“Error al crear los semáforos. Error: %x\n”)
         GetLastError W;
      return -1;
   }

     /* Crear el archivo que se utilizará como segmento de memoria
     compartida y asignar espacio */
     hIn = CreateFile(”ALMACEN”, GENERIC_WRITE, 0, NULL,
           CREATE_ALWAYS, FILEATTRIEUTENORJYIAL, NULL);
     SetFileSize(hIN, MAX_EUFFER * sizeof(int));

       /* proyectar el archivo en memoria */
       hlnMap = CreateFileMapping(hIN, NU , PACE_WRITEONLY, 0, 0, NULL);
       buffer = (mt *) MapViewofFile(hlnMap, FILE_MAP_WRITE, 0, 0, 0);

       if (buffer = NULL) {
         printf (“Error al proyectar el archivo. Error: %x\n”,
              GetLastError ( ));
       return -1;
       }

         productor(buffer);
         UnmapViewOfFile (buffer);
         CloseHaridle (hlnMap);
         CloseHandle (hIn);
         CloseHandle (huecos);
         CloseHandle (elementos);
```

```
        DeleteFile ( "ALMACEN");
      }
```



```
/* función productor */
void productor(int *buffer) {
      int dato;                    /* dato a producir */
      int posicion = 0; /*posición donde insertar el elemento */
      int j;

      for (j = 0; j < DATOS_A_PRODUCIR; j++){
          dato = j;
          WaitForSingleObject(huecos, INFINITE); /* un hueco menos */
          buffer  [posicion] = dato;
          posicion = (posicion+l) % MAX_BUFFER; /* nueva posición */
          ReleaseSemaphore(elementos, 1, NULL);
      }
      return;
}
```

El Programa 5.21 muestra el código del proceso consumidor.

Programa 5.21.Código del proceso consumidor utilizando semáforos y memoria compartida entre en Win32,

```
#include <windows .h>
#include <stdio.h>

#define MAX_BUFFER              1024 /* tamaño del buffer */
#define DATOS_A_PRODUCIR        100000 /* datos a producir */
 int main(void)
{
    HANDLE huecos, elementos;
    HANDLE hIN, hlnMap;
     int *buffer;

     huecos = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, "HUECOS");
     elementos = OpenSemaphore (SEMAPHORE_ALL_ACCESS, FALSE,
     "ELEMENTOS");
      if (huecos = = NULL ||elementos = = NULL) {
          printf ("Error al crear los semñaforos, Error: %x\n",
                  GetLastError lh;
          return -1;
      }

     /* Abrir el archivo que se utilizará como segmento de memoria compartida y asignar
     espacio */
     hIn = CreateFile("ALMACEN", GENERIC_READ, 0, NULL,
          OPEN_EXISTIMG, FILE_ATTRIBUTE_NORMAL, NULL);

     /* proyectar el archivo en memoria */
      hlnMap = CreateFi1eMapping(hIN, NULL, PACE_READEONLY, 0, 0, NULL);
     buffer = (int *) MapViewofFile(hlnMap FILE_MAPREAD, 0, 0, 0);

     if (buffer = NULL)
          printf ("Error al proyectar el archivo, Error: %x\n",
```

```
        GetLastError ( );
    return -1;
    }

    consumidor (buffer);

    UrnnapViewOfFile (buffer);
    CloseHandle (hlnMap);
    ClosaHaudle (hIn);
    CloseHandle (huecos);
    CloseHandle (elementos);
}
/ * función consumidor */
void consumidor(int *buffer) {
     int dato;        /* dato a producir */
     int posicion = 0;        /* posición que indica el elemento a extraer */
     int j;

     for (j = 0; j < DATOS_A_PRODUCIR; j++){
          WaitForSingleObject(elementos, INFINITE); /* un elemento menos */
           dato = buffer[posición];
           posición = (posición +l) % MAXBUFFER; /* nueva posición */
           ReleaseSemaphore (huecos, 1, NULL); /* un hueco max */

    }
    return;
}
```

5.8.4. Mutex *y* eventos

Los mutex, al igual que los semáforos, son objetos con nombre. Los mutex sirven para implementar secciones críticas. La diferencia en e las secciones criticas y los mutex de Win32 radica en que las secciones críticas no tienen nombre y sólo se pueden utilizar entre procesos ligeros de un mismo proceso. Un mutex se manipula utilizando manejadores.

Los servicios utilizados en Win32 para trabajar con mutex son los siguientes:

Crear un mutex

Para crear un mutex se utiliza el siguiente servicio:

```
HANDLE CreateMutex(LPSECURITY_ATTRIBUTES
                   lpsa,BOOLfInitialOwner,LPCTSTR lpszMutexName);
```

Esta función crea un mutex con atributos de seguridad lpsa. Si flnitialOwner es TRUE, el propietario del mutex será el proceso ligero que lo crea. El nombre del mutex viene dado por el tercer argumento. En caso de éxito la llamada devuelve un manejador de mutex válido y en caso de error NULL.

Abrir un mutex

Para abrir un mutex se utiliza:

HANDLE **OpenMutex**(LONG dwDesiredAccess, LONG BineheritHandle,
lpszName SemName);

Los parámetros y su comportamiento son similares a los de la llamada openSemaphore

Cerrar un mutex

Para cerrar un semáforo se utiliza el siguiente servicio:

BOOL **CloseHandle** (HANDLE hObject);

Si la llamada termina correctamente, se cierra el mutex. Si el contador del manejador es cero se liberan los recursos ocupados por el mutex. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Operación lock

Se utiliza el siguiente servicio de Win32.

DWORD **WaitForSingieObject** (HANDLE hMutex, DWORD dwTimeOut);

Ésta es la función general de sincronización que ofrece Win32, Cuando se aplica a un mutex implementa la función lock. El parámetro dwTimeOut debe tomar en este caso valor INFINITE.

El prototipo de este servicio es:

BOOL **ReleaseMutex** (HANDLE hMutex);

Los eventos en Win32 son comparables a las variables condicionales, es decir, se utilizan para notificar que alguna operación se ha completado o que ha ocurrido algún suceso. variables condicionales se encuentran asociadas a un mutex y los eventos no.

Los eventos en Win32 se clasifican en manuales y automáticos. Los primeros se pueden utilizar para desbloquear varios threads bloqueados en un evento. En este caso, el evento permanece en estado de notificación y debe eliminarse este estado de forma manual. Los automáticos se utilizan para desbloquear a un único thread, es decir, el evento notifica a un único thread y a continuación deja de estar en este estado de forma automática. POSIX no dispone de estados manuales. En esta sección sólo se trataran los automáticos, ya que estos son los más similares a las variables condicionales descritas en la Sección 53.6.

Crear un evento

Para crear un evento se utiliza el siguiente servicio:

HANDLE **CreateEvent**(LPSECURITY_ATTRIBUTES lpsa, BOOL fManua
BOOL fInitialState, LPTCSTR lpszEventName);

Esta función crea un evento de nombre lpszEventName con atributos de seguridad lpsa. Si fManualReset es TRUE, el evento será manual, en caso contrario será automático. Si fInitialState es TRUE, el evento se creará en estado de notificación, en caso contrario no. La llamada devuelve un manejador de evento válido en caso de éxito o NULL si se produjo algún fallo,

Esta función es similar al servicio pthread_cond_init de POSIX. Para emular el comportamiento de esta llamada se debería invocar a CreateEvent de la siguiente forma:

CreateEvent(NULL, FALSE, FALSE, name);

Siendo name el nombre dado al evento.

Destruir un evento

Para destruir un evento se utiliza:

BOOL **CloseHandle**(HANDLEhObject);

Si la llamada termina correctamente, se cierra el evento. Si el contador del manejador es cero, se liberan los recursos ocupados por el evento y se destruye. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Esperar por un evento

Ésta operación es similar a la operación c_wait sobre variables condicionales.

DWORD **WaitForSingleObject** (HANDLE hEvent, DWORD dwTimeOut);

Ésta es la función general de sincronización que ofrece Win32. Cuando se aplica a un evento implementa la función c_wait. El parámetro dwTimeOut debe tomar en este caso valor INFINITE.

Notificar un evento

Para notificar un evento se pueden utilizar dos servicios:
BOOL **SetEvent** (HANDLE hEvent);
BOOL **PulseEvent** (HANDLE hEvent);

El primero notifica un evento y despierta a un único thread bloqueado en waitForSingleObject. La segunda función notifica un evento y despierta a todos los threads bloqueados en él. Cuando se notifica un evento automático, después de la llamada y una vez desbloqueado el thread o threads correspondientes, el evento pasará a estado de no notificación. Si ningún thread está esperando el evento cuando se ejecuta alguna de estas llamadas, el evento permanecerá señalizando hasta que algún thread ejecute una llamada de espera.

Secciones críticas con eventos

El problema de la sección critica puede resolverse utilizando eventos (Prestaciones *5.1).* Para ello ha de crearse un evento cuyo estado inicial sea notificado:

HANDLE hEvent;

```
hEvent = CreateEvent(NULL,          /* sin atributos de seguridad */
                     FALSE,         /* evento automático */
```



```
                     TRUE,          /* evento inicialmente notificado */
                     "evento";)     /* nombre del evento */
```

Para acceder a la sección crítica basta con protegerla de la siguiente manera:

```
WaitForsingleObject(hEvent, INFINITE>;
<Sección crítica >
SetEvent (hEvent);
```

RECORDATORIO 5.1

Estos servicios también se utilizan en Win32 para esperar la terminación de uno o más procesos ligeros, como se vio en el Capítulo 4.

Variables condicionales utilizando eventos

Como se dijo anteriormente, los eventos de Win32 son similares a las variables condicionales, sin embargo, los eventos no se encuentran asociados a ningún mutex, como ocurre con las variables condicionales. A continuación se va a describir cómo utilizar los mutex y los eventos de Win32 para emular el comportamiento de los mutex y las variables condicionales,

Para implementar el siguiente fragmento de código (típico cuando se utilizan mutex y variables condicionales):

```
lock(m);
/* código de la sección crítica */
while (condicion == FALSE>
      c_wait(c, m);
/* resto de la sección crítica */
unlock(m>;
```

En Win32 debe crearse un mutex y un evento en estado inicial de no notificación. Una vez creados, el código anterior se convertirá en el siguiente:

```
WaitForSingleObject(mutex, INFINITE);
/* código de la sección crítica */
while (condicion = = FALSE) {
    ReleaseMutex(mutex);            /* se libera el mutex */
     WaitForMultipleobjects (2, hHandles, TRUE, INFINITE);
}
/* resto de la sección crítica */
ReleaseMutex (mutex);
```

donde hHandles es un vector:

```
HANDLE hHandles[2];
hHandles[0] = mutex;
hHandles [1] = evento;
```

Recuérdese que una operación c_wait sobre una variable condicional bloquea al proceso y libera de forma atómica el mutex para permitir que otros procesos puedan ejecutar. Utilizando eventos es necesario en primer lugar liberar la sección crítica (ReleaseMutex) y esperar a continuación por el mutex y el evento (waitForMultipleObjects).

Para implementar la siguiente estructura:

```
lock(m);
/* código de la sección crítica */
/*se modifica la condición y ésta se hace TRUE */
condicion = TRUE;
c_signal(c);
unlock(m);
```

se debe utilizar el siguiente fragmento de código:

```
WaitForSingleObject(mutex, INFINITE>;
/*código de la sección crítica */
/ se modifica la condición y ésta se hace TRUE */
condicion = TRUE;
SetEvent(evento);
ReleaseMutex(mutex);
```

Sólo cuando se ejecute el último ReleaseMutex se desbloqueará a un proceso bloqueado en WaitForMultipleObjects (sobre el mutex y el evento). Cuando el proceso bloqueado en esta función despierta, continuará la ejecución con el mutex adquirido y el evento en estado de no notificación.

5.8.5. Mailslots

Los mailslots de Win32 son parecidos a las tuberías con nombre de Win32 y a las colas de mensajes de POSIX. Sus principales características son:

- Un mailslot es un pseudoarchivo unidireccional que reside en memoria. El tamaño máximo de los mensajes que se pueden enviar a un mailslot es de 64 KB.
- Pueden tener múltiples lectores y múltiples escritores, Sin embargo, sólo los procesos que creen el mailslot o lo hereden pueden leer mensajes de él,
- Pueden utilizarse entre procesos que ejecutan en máquinas de un mismo dominio de red.

Las principales funciones relacionadas con los mailslots son las siguientes:

Crear un mailslot

El prototipo de la función que permite crear un mailslot es el siguiente:

```
HANDLE CreateMailslot(LPCTSTR lpName, DWORD Ma essSize, DWORD
                                    lR adTimeout, LPSECURITY_ATTRIBUTES lpsa);
```

La función crea un mailslot de nombre lpName. MaxMessSize especifica el tamaño máximo del mensaje. Un valor de 0 indica que el mensaje puede ser de cualquier tamaño. El parámetro

lReadTimeout especifica el tiempo en milisegundos que una operación de lectura del mailslot bloqueará al proceso hasta obtener un mensaje. El valor MAILSLOT_WAlT_FOREVER bloquea al proceso que lee hasta que obtenga un mensaje.

El nombre del mailslot debe seguir la siguiente convención:

\ \.\ mailslot\ nombre

La función devuelve un manejador de mailslot válido si se ejecutó con éxito o INVALID_HANDLE_VALUE en caso contrario.

Abrir un mailslot

Para abrir un mailslot se utiliza la función CreateFile ya descrita anteriormente. El formato del nombre debe ser de la siguiente forma:

- **\\.** \mailslot\nombre: para abrir un mailslot local.
- \ **\.** \máquina\mailslot\nombre: para abrir un mailslot creado en una máquina remota
- \ **\.**\nombredominio\mailslot\nombre para abrir un mailslot creado en un dominio.

Una aplicación debe especificar FILE_SHARE_READ para abrir un mailslot existente.

Cerrar un mailslot

Para cerrar un mailslot se utiliza CloseHandle. Cuando el mailslot deja de tener manejador abiertos, se destruye.

Leer y escribir en un mailslot

Para leer y escribir se utilizan los servicios **ReadFile** y **WriteFile** respectivamente.

Asignar atributos a un mailslot

Los únicos atributos que se pueden modificar sobre un mailslot ya creado es el tiempo de bloqueo de la operación **ReadFile**. El prototipo de esta función es:

BOOL **SetMailSlotlnfo** (HANDLE hMailslot, DWORD lReadTimeout);

Obtener los atributos de un mailslot

Para obtener los atributos asociados a un mailslot se debe utilizar la siguiente función:

```
BOOL GetNailslotlnfo (HANDLE hMailslot, LPDWORD lpMaxMess,
                      LPDWORD lpNexSize, LPDWORD lpMessCount,
                      LPDWORD lpReadTimeout);
```

En lpMaxMess se almacenará el tamaño máximo del mensaje, en lpNextSize el tamaño del siguiente mensaje. Si no hay ningún mensaje en el mailslot, el parámetro anterior tomará el valor lpNextSize. El argumento lpMessCount almacenará el numero de mensajes pendientes de lectura. En el último argumento se almacenará el tiempo de

bloqueo de la función ReadFile.

El desarrollo de una aplicación cliente-servidor utilizando mailslots es similar al uso

Tabla 5.3. *Características de los mecanismos de comunicación de Win32*

Mecanismo	Nombrado	Identificador	Almacenamiento	Flujo de datos
Tubería	Sin nombre	Manejador	Sí	Unidireccional
	Con nombre			Bidireccional
Mailslot	Nombre dentro de un dominio de Windows NT	Manejador	Sí	Bidireccional

Las Tablas 5.3 y 5.4 presentan las características de los mecanismos de comunicación y sincronización respectivamente de Win32.

Tabla 5.4. *Características de los mecanismos de sincronización de POSIX*

Mecanismo	Nombrado	Identificador
Sección crítica	Sin nombre	Variable de tipo sección crítica
Tubería	Sin nombre	Manejador
	Con nombre	
Semáforo	Con nombre	Manejador
Mutex	Con nombre	Manejador
Evento	Con nombre	Manejador
Mailslot	Con nombre	Manejador

5.9. PUNTOS A RECORDAR

- Existen varias razones que motivan la ejecución de procesos concurrentes en un sistema: se facilita la programación, se permite acelerar los cálculos, se posibilita el uso interactivo a múltiples usuarios y se consigue un mejor aprovechamiento de los recursos de la computadora.
- Los procesos concurrentes que ejecutan en un sistema se pueden clasificar en independientes o cooperantes.
- Un proceso independiente es aquel que se ejecuta sin requerir la ayuda o cooperación de otros procesos.
- Los procesos son cooperantes cuando están diseñados para trabajar conjuntamente en alguna actividad. Estos procesos necesitan, por tanto, de mecanismos de comunicación y de sincronización.
- Una sección crítica es un fragmento de código que debe ejecutarse en exclusión mutua, es decir, cuando un proceso esté ejecutando código de la sección crítica, ningún otro proceso puede ejecutar en la sección.
- Cualquier solución que se utilice para resolver el problema de la sección crítica debe cumplir tres requisitos: exclusión mutua, progreso y espera acotada.
- Para resolver el problema de la sección crítica se necesitan mecanismos de sincronización que permitan a los procesos acceder a la sección crítica en exclusión mutua.
- En el problema del productor-consumidor, uno o más procesos, denominados productores, generan cierto tipo de datos que son utilizados o consumidos por otros procesos denominados consumidores.
- En el problema de los lectores-escritores existen dos tipos de procesos: lectores y escritores. Los procesos lectores acceden a un recurso compartido para consultar su estado, mientras que los procesos escritores acceden al recurso compartido para modificar su contenido. En este problema, cuando haya un proceso escritor accediendo al recurso, no se permitirá el acceso a ningún proceso lector o escritor.
- En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos denominados clientes. Este tipo de comunicación suele emplearse en computadoras conectadas por una red.

- Los mecanismos que pueden emplearse para comunicar procesos son: archivos, tuberías, variables en memoria compartida y paso de mensajes.
- Para sincronizar la ejecución de procesos se pueden emplear señales, tuberías, semáforos, mutex, variables condicionales y esquemas de paso de mensajes.
- Una tubería es un mecanismo de sincronización y de comunicación unidireccional sobre el que se pueden realizar operaciones de lectura y escritura. Estas operaciones son atómicas cuando el tamaño de los datos solicitados es menor que el tamaño de la tubería.
- Un semáforo es un objeto con un valor entero que se puede emplear para sincronizar procesos. Existen dos operaciones atómicas asociadas a un semáforo: `wait` y `signal`. La primera permite bloquear al proceso que la ejecuta cuando el valor del semáforo es menor o igual que cero. La segunda incrementa el valor del semáforo y permite desbloquear a algún proceso bloqueado en el mismo.
- Un mutex es un mecanismo de sincronización con dos operaciones: `lock` y `unlock`. La operación `lock` intenta bloquear al mutex; si éste ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. La operación `unlock` desbloquea el mutex y despierta a uno de los procesos que se encuentren bloqueado en él.
- Una variable condicional es una variable de sincronización asociada a un mutex que se utiliza para bloquear a un proceso hasta que ocurra algún evento. Este tipo de variables tienen dos operaciones atómicas: `c_wait` y `c_signal`. La primera bloquea al proceso que ejecuta la llamada y desbloquea el mutex dentro del cual se ejecute. La segunda desbloquea a uno o varios procesos suspendidos en una variable condicional.
- Los esquemas de paso de mensajes utilizan dos primitivas denominadas `send` y `receive`. La operación `send` permite enviar datos a un proceso y mediante la operación `receive` un proceso puede recibir datos de otro proceso. En estos esquemas la operación `receive` suele ser síncrona, es decir, su ejecución bloquea al proceso que la ejecuta hasta que llegue un mensaje. Esto permite utilizar los esquemas de paso de mensajes para sincronizar procesos.
- Los mecanismos de sincronización pueden implementarse utilizando espera pasiva o espera activa.
- Con espera activa un proceso no suspende su ejecución, sino que entra en un bucle en el que consulta de forma continua un valor que le permite continuar su ejecución o no. Con este modelo se desperdician ciclos de procesador en procesos que no pueden ejecutar.
- Utilizando espera pasiva, el proceso que no puede continuar se bloquea, suspendiendo su ejecución.
- Los mecanismos de sincronización, tales como los semáforos, suelen implementarse utilizando instrucciones especiales como `test-and-set` y `swap`.
- POSIX dispone de los siguientes mecanismos de comunicación y sincronización: tuberías, semáforos, mutex, variables condicionales y colas de mensajes.
- Los mecanismos que ofrece Win32 para la comunicación y sincronización de procesos son: tuberías, secciones críticas, semáforos, mutex, eventos y mailslots.

5.10. LECTURAS RECOMENDADAS

El tema de la comunicación y sincronización entre procesos se analiza en multitud de libros. Uno de los primeros en tratar los algoritmos para resolver el problema de la exclusión mutua fue Dijkstra [Dijkstra, 1965]. En [Ben Ari, 1990] se realiza un estudio sobre la concurrencia de procesos y los mecanismos de comunicación y sincronización. El estudio de la comunicación y sincronización entre procesos puede completarse también en [Stallings, 1998] y [Silberschatz, 1999]. En [Beck, 1997] se analiza la implementación de los mecanismos de comunicación y sincronización en Linux, en [Solomon, 1998] se estudia la implementación en Windows NT y en [McKusick, 1996] y [Bach, 1986] la implementación en diversas versiones de UNIX.

5.11. EJERCICIOS

5.1. Dos procesos se comunican a través de un archivo, de forma que uno escribe en el archivo y el otro lee del mismo. Para sincronizarse, el proceso escritor envía una señal al lector. Proponga un esquema del código de ambos procesos. ¿Qué problema plantea la solución anterior?

5.2. El siguiente fragmento de código intenta resolver el problema de la sección crítica para dos procesos P0 y P1.

```
while (turno != i)
;
SECCIÓN CRÍTICA;
turno = j;
```

La variable `turno` tiene valor inicial 0. La variable *i* vale 0 en el proceso P0 y 1 en el proce-

so P1. La variable *j* vale 1 en el proceso P0 y 0 en el proceso P1. ¿Resuelve este código el problema de la sección crítica?

5.3. ¿Cómo podría implementarse un semáforo utilizando una tubería?

5.4. Un semáforo binario es un semáforo cuyo valor sólo puede ser 0 o −1. Indique cómo puede implementarse un semáforo general utilizando semáforos binarios.

5.5. Suponga un sistema que no dispone de mutex y variables condicionales. Muestre cómo pueden implementarse éstos utilizando semáforos.

5.6. Considérese un sistema en el que existe un proceso agente y tres procesos fumadores. Cada fumador está continuamente liando un cigarrillo y fumándolo. Para fumar un cigarrillo son necesarios tres ingredientes: papel, tabaco y cerillas. Cada fumador tiene reserva de un solo ingrediente distinto de los otros dos. El agente tiene infinita reserva de los otros tres ingredientes, poniendo cada vez dos de ellos sobre la mesa. De esta forma, el fumador con el tercer ingrediente puede liar el cigarrillo y fumárselo, indicando al agente cuándo ha terminado, momento en que el agente pone en la mesa otro par de ingredientes y continúa el ciclo.

a) Escribir un programa que sincronice al agente y a los tres fumadores utilizando semáforos.
b) Resolver el mismo problema utilizando mensajes.

5.7. Implemente la aplicación cliente-servidor desarrollada en la Sección 5.7.4 utilizando los mailslots de Win32.

5.8. Modifique el Programa 5.4 de manera que sean los escritores los procesos prioritarios.

5.9. Muestre cómo podrían implementarse los semáforos con nombre de POSIX en un sistema que no dispone de ellos pero sí dispone de colas de mensajes.

5.10. Resuelva el problema de los lectores-escritores utilizando las colas de mensajes de POSIX.

5.11. Resuelva el problema de los lectores-escritores utilizando los mailslot de Win32.

5.12. Una barbería está compuesta por una sala de espera, con *n* sillas, y la sala del barbero, que tiene un sillón para el cliente que está siendo atendido. Las condiciones de atención a los clientes son las siguientes:

— Si no hay ningún cliente, el barbero se va a dormir.
— Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente abandona la barbería
— Si hay sitio y el barbero está ocupado, se sienta en una silla libre.
— Si el barbero estaba dormido, el cliente le despierta.

Escriba un programa que coordine al barbero y a los clientes utilizando colas de mensajes POSIX para la comunicación entre procesos.

5.13. Resuelva el Problema 8 utilizando mailslots.

5.14. Resuelva el Problema 8 utilizando las tuberías con nombre que ofrece Win32.

5.15. Un puente es estrecho y sólo permite pasar vehículos en un sentido al mismo tiempo. El puente sólo permite pasar un coche al mismo tiempo. Si pasa un coche en un sentido y hay coches en el mismo sentido que quieren pasar, entonces éstos tienen prioridad frente a los del otro sentido (si hubiera alguno esperando). Suponga que los coches son los procesos y el puente el recurso. El aspecto que tendrá un coche al pasar por el lado izquierdo del puente sería:

```
entrar_izquierdo
pasar puente
salir izquierdo
```

y de igual forma con el derecho.
Se pide escribir las rutinas `entrar_izquierdo` y `salir_izquierdo` usando como mecanismos de sincronización los semáforos.

5.16. Resuelva el problema anterior utilizando mutex y variables condicionales.

6
Interbloqueos

En un sistema informático se ejecutan concurrentemente múltiples procesos que normalmente no son independientes, sino que compiten en el uso exclusivo de recursos y se comunican y sincronizan entre sí. El sistema operativo debe encargarse de asegurar que estas interacciones se llevan a cabo apropiadamente proporcionando la exclusión mutua requerida por las mismas. Sin embargo, generalmente no basta con esto. Las necesidades de algunos procesos pueden entrar en conflicto entre sí causando que éstos se bloqueen indefinidamente. A esta situación se le denomina interbloqueo (en inglés se suele utilizar el término deadlock*). En este capítulo se estudiará este problema y se analizarán sus posibles soluciones.*

Se trata de un problema ya identificado en la década de los sesenta y estudiado ampliamente desde entonces. Sin embargo, aunque pueda parecer sorprendente a priori, ese notable interés en su estudio teórico no ha tenido una repercusión apreciable en los sistemas operativos reales. La mayoría de los sistemas operativos actuales ignoran en gran parte este problema o, al menos, no lo solucionan de forma general. Es importante resaltar que este problema ha sido también analizado en otras materias ajenas a los sistemas operativos, como es el caso de las bases de datos.

El planteamiento que se va a realizar en este capítulo intentará conjugar dentro de lo posible los aspectos teóricos del tema con aquellos relacionados con su aplicación en un sistema real. En primer lugar, se presentará el problema de una manera informal mostrando varios ejemplos del mismo tanto procedentes del ámbito de la informática como de otros ajenos a la misma. Después de esta introducción se planteará un modelo general que permitirá un estudio formal de los interbloqueos. A continuación se estudiarán las tres estrategias usadas para tratarlos: la detección y recuperación, la prevención y la predicción. Por último, se analizará cómo manejan este problema los sistemas operativos reales. Este desarrollo del tema se desglosa en las siguientes secciones:

- *Los interbloqueos: una historia basada en hechos reales.*
- *Los interbloqueos en un sistema informático.*
- *Un modelo del sistema.*
- *Definición y caracterización del interbloqueo.*
- *Tratamiento del interbloqueo.*
- *Detección y recuperación del interbloqueo.*
- *Prevención del interbloqueo.*
- *Predicción del interbloqueo.*
- *Tratamiento del interbloqueo en los sistemas operativos.*

6.1. LOS INTERBLOQUEOS: UNA HISTORIA BASADA EN HECHOS REALES

El problema de los interbloqueos no se circunscribe únicamente al mundo de la informática, si que aparece en muchos otros ámbitos incluyendo el de la vida cotidiana. De hecho, algunos de ejemplos utilizados por los investigadores en este tema están inspirados en situaciones cotidiana Un ejemplo de ello es el ampliamente conocido problema de la cena de los filósofos propuesto por Dijkstra [Dijkstra, *1965]*. En esta sección se presentarán algunos ejemplos de interbloqueos extraídos de la vida real para que sirvan como una introducción intuitiva a este tema.

Uno de los ámbitos que proporciona mas ejemplos es el del tráfico de vehículos. De hecho, uno de los objetivos de algunas señales de tráfico, e incluso de algunas normas de circulación, resolver los posibles interbloqueos entre los vehículos. Observe que, en este ámbito, el interbloqueo causaría la detención permanente de los vehículo implicados. Al fin y al cabo, un atasco de tráfico en una ciudad es un caso de interbloqueo.

Como ejemplo, considérese una carretera con dos sentidos de circulación que atraviesa largo puente estrecho por el que sólo cabe un vehículo. El interbloqueo se produciría dos cuando vehículos atravesando simultáneamente el puente en sentido contrario se enconaran el uno frete al otro sin posibilidad, por tanto, de continuar. Observe que cada vehículo posee un recurso (el trozo de puente que ya ha cruzado hasta el momento) pero necesita otro recurso (el trozo de puente que queda por cruzar) para terminar su labor (cruzar el puente). El interbloqueo surge debido a que produce un conflicto entre las necesidades de los dos vehículos: el recurso que necesita cada vehículo lo posee el otro. Hay que resaltar que otros vehículos que intentaran cruzar el puente ese momento en cualquiera de los dos sentidos se quedarían detenidos detrás de ellos viéndose, tanto, implicados también en el interbloqueo. Sobre el ejemplo anterior se puede plantear una primera idea general de cuáles son las posibles estrategias para tratar el problema de los interbloqueos:

- **Detección y recuperación.** Una vez detectada la situación de interbloqueo, uno de vehículos debe liberar el recurso que posee para dejar que el otro lo utilice. Una posible recuperación de esta situación consistiría en seleccionar uno de los sentidos de circulación hacer que el vehículo o vehículos detenidos en ese sentido dieran marcha a tras hasta el principio del puente, liberando así el paso en el otro sentido (se esta suponiendo que vehículo tiene capacidad para avanzar marcha atrás, si no fuera así, habría que tomar i acción más drástica como tirarlo al río que pasa por debajo del puente). Debería existir una política para determina qué vehículos deben retroceder. Para ello se podrían tener en cu aspectos tan diversos como cuánta distancia ha recorrido cada vehículo dentro del puente, que velocidad tiene cada vehículo, cuál de ellos tiene mayor prioridad (p. ej: en el caso de una ambulancia) o cuantos vehículos hay en cada sentido. Observe que cada vehículo al afectado incurriría en un gasto extra de combustible y de tiempo debido al primer intento frustrado de cruzar el puente y al recorrido de vuelta hasta el principio del mismo. Esta estrategia, por tanto, tiene un carácter que se podría considerar destructivo ya que se pie parte del trabajo realizado por una entidad.
- **Prevención o predicción.** Un punto importante a resaltar en este ejemplo y, en gene sobre los interbloqueos es que, antes de producirse el interbloqueo

propiamente dicho vehículos detenidos frente a frente), existe un «punto de no retomo» a partir del cual interbloqueo es inevitable. En el ejemplo, habiendo uno o mas vehículos atravesando puente en un determinado sentido, el punto de no retorno ocurriría en el momento en que un vehículo entrase en el puente en sentido contrario. A partir de ese momento, tarde o temprano, se produciría un interbloqueo. Las estrategias basadas en la prevención, o predicción,

evitan el interbloqueo asegurando que no se llega a este punto de no retorno. En el ejemplo se podría usar para ello un sistema de señalización basado en semáforos. Más adelante, en este capitulo, se explicara la diferencia que existe en e las estrategias de prevención y las de predicción. Un último aspecto a destacar es que, aunque estas estrategias no tienen el carácter destructivo de la anterior, suelen implicar en muchos casos una infrautilización de los recursos. En el ejemplo planteado podría ocurrir que una es estrategia preventiva basada en el uso de semáforos hiciese que un vehículo tuviera que detenerse en el semáforo aunque el puente estuviese libre.

Además de poder aparecer interbloqueos en el uso exclusivo de recursos, también pueden ocurrir en escenarios donde existe comunicación y sincronización entre entidades. Supóngase un sistema de comunicación telefónica en el que cuando se realiza una llamada a un determinado número esta llamada queda bloqueada si el aparato telefónico receptor de la misma está ocupado intentando establecer una llamada o manteniendo una llamada previa. Cuando el aparato receptor queda libre de la llamada en curso, se establece automáticamente la llamada pendiente desbloqueando al usuario que realizó la llamada. En este entorno, dos usuarios podrían verse implicados en un interbloqueo si intentan llamada se entre sí simultáneamente. Se podría producir también un interbloqueo si, por error, un usuario intentase llamarse a sí mismo. Este último ejemplo muestra dos aspectos interesantes sobre los interbloqueos:

- No es necesario que intervengan do: o más entidades para que se produzca un interbloqueo.
- En algunas situaciones, los interbloqueos se deben a un error de operación.

6.2 LOS INTERBLOQUEOS EN UN SISTEMA INFORMÁTICO

Como se ha podido apreciar en los ejemplos anteriores, un escenario donde pueden aparecer interbloqueos se caracteriza por la existencia de un conjunto de entidades activas (los vehículos o los usuarios del teléfono) que utilizan un conjunto de recursos (el puente estrecho o el sistema de comunicación telefónica) para 11ev a cabo su labor. De manera similar, en un sistema informático existirán estos dos papeles:

- Las entidades: activas que corresponden, evidentemente, con los procesos existentes en el sistema. Es importante resaltar que en un sistema operativo que proporcione *threads,* éstos representarán las entidades activas, ya que son la unidad de ejecución del sistema.
- Los recursos existentes en el :sistema que serán utilizados por los procesos para 11ev a cabo su labor. En un sistema existe una gran variedad de recursos. Por un lado, recursos físicos tales como procesadores, memoria o dispositivos. Por otro, recursos lógicos tales como chivos, semáforos, *mutex,* cerrojos, mensajes o

señales,

Dada la diversidad de recursos existentes en un sistema, y su diferente comportamiento con respecto al interbloqueo, a continuación se establecerá una clasificación de los distintos recursos que permitirá delimitar el alcance del estudio de los interbloqueos que se planteará en este capítulo. Asimismo, se mostraran algunos ejemplos de interbloqueos en un sistema informático.

6.2.1. Tipos de recursos

Desde el punto de vista del estudio del interbloqueo, los recursos presentes en un sistema se pueden clasificar siguiendo varios criterios:

- El recurso sigue existiendo después de que un proceso lo use *(recurso reutilizable)* o rece una vez utilizado *(recurso consumible).*
- Los procesos pueden compartir el uso de un recurso o lo deben usar en modo exclusivo o dedicado.
- Hay un único ejemplar de cada recurso o existen múltiples unidades de cada uno.
- Es factible expropiar el recurso al proceso que lo está utilizando.

Recursos reutilizables o consumibles

Como se comentó previamente, un **recurso reutilizable** se caracteriza porque el recurso existiendo después de que un proceso lo use quedando disponible para otros procesos. Por tanto, la «vida» del recurso es independiente de su utilización. El recurso, o bien existe desde el (en el caso de que se trate de un recurso físico), o una vez creado sigue existiendo hasta destruya explícitamente (como, por ejemplo, un archivo). Dentro de esta categoría se todos los recursos físicos y recursos lógicos, como los archivos, los cerrojos o los semáforos de *mutex.* A continuación se muestra un ejemplo sencillo de interbloqueo usando este tipo de recurso

Supóngase que existen dos procesos, P_1 y P_2, tal que ambos usan una cinta (c) y una impresora (I) y que tienen la siguiente estructura:

Proceso P_1	Proceso P_2
Solicita(C)	Solicita(I)
Solicita (I)	Solicita(C)
Uso de los recursos	Uso de los recursos
Libera(I)	Libera(C)
Libera(C)	Libera(I)

Durante la ejecución de estos procesos en un sistema multiprogramado se puede producir interbloqueo, ya que se puede llegar a una situación en la que el primer proceso tiene asignada cinta y está bloqueado esperado por la impresora, mientras que el segundo tiene la impresora pero esta esperando por la cinta. Dado que se trata del primer ejemplo de interbloqueo en un informático que se presenta, se mostrará a continuación un posible orden de ejecución de los procesos que produciría el interbloqueo:

1. P_1: solicita (c)

2. P_2: solicita(I)
3. P_2 :solícita (C) → se bloquea puesto que el recurso no está disponible
4. P_1: solícita (1) → se bloquea ya que el recurso no está disponible: se produce interbloqueo

En un sistema monoprocesador, este orden de ejecución entrelazado característico de los interbloqueos se debería a que se producen cambios de contexto en los instantes correspondientes. En un multiprocesador podrán ocurrir simplemente debido a que cada proceso ejecuta en un procesador diferente.

Es interesante resaltar que no todos los posibles órdenes de ejecución de estos dos procesos causarían un interbloqueo. Así, por ejemplo, si el orden de ejecución fuera el siguiente:

1. P_1: solícita (C)
2. P_1: solícita (I)
3. P_2: solícita (I) → se bloquea puesto que el recurso no está disponible
4. P_1: libera (I) → se desbloquea P_2 ya que el recurso ya está disponible
5. P_2: solícita (C) → se bloquea puesto que el recurso no está disponible
6. P_1: libera (C) → se desbloquea m'$_2$ porque el recurso ya está disponible
7. P_2: libera (C)
8. P_2: libera (I)

Los dos procesos habrían terminado realizando su labor sin producirse interbloqueos. Observe que hay que diferenciar claramente entre el bloqueo de un proceso debido a la falta de un recurso (como ocurre con en los pasos 3 y *5* anteriores) que terminará cuando dicho recurso esté disponible y el interbloqueo que implica el bloqueo indefinido de los procesos involucrados,

En cuanto a los **recursos consumibles**, éstos se caracterizan porque dejan de existir una vez que un proceso los usa. Un proceso genera o produce el recurso y el o o lo utiliza consumiéndolo, Dentro de esta categoría se encuentran básicamente recursos lógicos relacionados con la comunicación y sincronización de procesos. Algunos ejemplos serían los mensajes, las señales o los semáforos generales (más adelante se explicará por qué se considera que los semáforos de tipo *mutex* son recursos reutilizables y, en cambio, los semáforos generales son recursos consumibles). Así, por ejemplo, en el caso de un mensaje, existe un proceso que genera el recurso (el emisor del mensaje) y otro que lo consume cuando lo recibe (el receptor del mensaje). A continuación se muestra un ejemplo de interbloqueo n e procesos que se comunican mediante mensajes. Se ha elegido deliberadamente una situación con tres procesos ya que hasta ahora sólo se habían planteado ejemplos en os que intervenían dos y, como se ha comentado previamente, el interbloqueo puede afectar un número ilimitado de procesos.

Supóngase un sistema de comunicación que proporciona primitivas para enviar y recibir mensajes en las que se especifica como primer parámetro el proceso al que se le quiere enviar o del que se quiere recibir, respectivamente, y como segundo el mensaje. Además, en este sistema el envío del mensaje no es bloqueante pero la recepción sí lo es. Considérese que se ejecutan en este sistema los siguientes tres procesos:

Proceso P_1	**Proceso P_2**	**Proceso P_3**
Enviar(P_3,A)	Recibír(P_1,D)	Recibir(P_2, C)
Recíbir(P_3, B)	Enviar(P_3,E)	Enviar(P_1,G)
Enviar (P_2, C)		Recibir (P_1, H)

La ejecución de estos tres procesos produciría un interbloqueo de los mismos con independencia de cual sea su orden de ejecución. El proceso P_3 se quedaría bloqueado indefinidamente esperando el mensaje de p_2, ya que éste no lo puede mandar hasta que reciba un mensaje de P_1 que, a su vez, no puede hacerlo porque debe antes recibir un mensaje de P_3. Cada proceso se queda bloqueado esperando un mensaje que sólo puede enviar otro de los procesos implicados, lo que no puede ocurrir ya que dicho proceso está también bloqueado esperando un mensaje. Es importante resaltar que en este ejemplo, a diferencia del anterior, se produce el interbloqueo con independencia del orden en que se ejecuten los procesos. Se ata, por tanto, de un interbloqueo que se podría denominar *estructural* puesto que se debe a un error en el diseño del patrón de comunicaciones entre los procesos. En una aplicación relativamente compleja, formada por múltiples procesos comunicándose entre sí, pueden d se errores de este tipo que sean bastante difíciles de detectar.

En un sistema general, los procesos usarán tanto recursos reutilizables como consumibles y. por lo tanto, como se puede apreciar en el siguiente ejemplo, pueden aparecer interbloqueos en los que estén implicados recursos de ambos tipos.

Proceso P_1	**Proceso P_2**
Solícita(C)	Solícita(C)
Enviar(P_2,A)	Recíbir(P_1, 5)
Libera(C)	Libera(C)

Si durante la ejecución concurrente de estos dos procesos el proceso P_2 obtiene el recurso o, se producirá un interbloqueo entre los dos procesos, ya que P_2, nunca recibirá el mensaje puesto que P, está bloqueado esperando que se libere el recurso o. Observe que si, en cambio, fuera el proceso P_1, el que obtuviera el recurso o, no se produciría un interbloqueo.

No hay una solución general eficiente para tratar el problema de los interbloqueos con ambos tipos de recursos debido, principalmente, a las características especificas que presentan los recursos consumibles. Por ello, a partir de este momento, la exposición se centrara en los recursos reutilizables, El lector interesado puede consultar [Maekawa, 1987] para estudiar las estrategias que se utilizan para tratar los interbloqueos con recursos consumibles.

Recursos compartidos o exclusivos

Aunque hasta el momento no se haya expresado explícitamente, se ha estado suponiendo que cuando un proceso está usando un recurso, ningún otro lo puede usar. O sea, se ha considerado que los recurso. se usan de forma exclusiva o dedicada. Sin embargo, esto no es así para todos los recursos. Algunos recursos pueden ser usados

simultáneamente por varios procesos: son recursos compartidos, Considérese, como ejemplo de recurso de este tipo, un chivo al que pueden acceder simu1táneamente múltiples procesos.

Como es evidente, los recursos de tipo compartido no se ven afectados por los interbioqueos ya que los procesos que quieran usarlos podrán hacerlo inmediatamente sin posibilidad de quedarse bloqueados. Por tanto, las estrategias de tratamiento de interbloqueos que se estudiarán posteriormente no tratarán este tipo de recursos, centrándose únicamente en los *recursos reutilizables de uso exclusivo* (a los que también se les denomina *recursos reutilizables en serie)*.

Es interesante resaltar que en un sistema pueden existir recursos que tengan ambos modos de uso (compartido y exclusivo). Cuando un proceso quiere usar un recurso de este tipo, debe especificar en su solicitud si desea utilizarlo en modo exclusivo o compartido. El sistema permitirá que varios procesos usen el recurso silo hacen todos ellos en modo compartido, pero, sin embargo, sólo permitirá que un único proceso lo use en modo exclusivo. Así, una solicitud de un recurso para su uso en modo compartido se satisfará inmediatamente siempre que el recurso no esté siendo usado en modo exclusivo. En cambio, una solicitud de uso en modo exclusivo sólo se concederá si el recurso no está siendo utilizado en ese momento.

Un ejemplo de este tipo de recursos lo proporciona el servicio `fcntl` de POSIX que permite establecer un cerrojo sobre un archivo (o una región del mismo). Se pueden especificar dos tipos de cerrojos: de lectura (que correspondería con un uso compartido del recurso) o de escritura (que implicaría un uso exclusivo). Este tipo de recursos t bien aparece frecuentemente en las bases de datos.

Por simplicidad, no se considerará dentro de esta exposición el tratamiento de los interbloqueos para este tipo de recursos. En el Ejercicio 6.29 se estudia este problema específico, analizando cómo se pueden adapta los algoritmos que se presentan en las secciones posteriores para que puedan tratar recursos de este tipo.

Recursos con un único ejemplar o con múltiples

Hasta ahora se ha considerado que cada recurso es una entidad única. Sin embargo, en un sistema pueden existir múltiples instancias o ejemplares de un determinado recurso. Una solicitud de ese recurso por parte de un proceso podría satisfacerse con cualquier ejemplar del mismo. Así, por ejemplo, en un sistema en el que haya cinco impresoras, cuando un proceso solicita una impresora, se le podría asignar cualquier unidad que esté disponible. La existencia de múltiples unidades de un mismo recurso también permite generalizar los servicios de solicitud de forma que un proceso pueda pedir simultáneamente varios ejemplares de un recurso. Evidentemente, el numero de unidades solicitadas nunca debería ser mayor que el número de unidades existentes. Las soluciones al problema del ínterbloqueo que se plantearán en este capítulo serán aplicables a este modelo general en el que existe un conjunto de recursos, cada uno de los cuales esta formado por una o más unidades.

Observe que, a veces, puede ser discutible si dos elementos constituyen dos instancias de un recurso o se trata de dos recursos diferentes. Incluso distintos usuarios pueden querer tener una visión u otra de los mismos. Considérese, por ejemplo, un equipo que tiene conectadas dos impresoras láser con la misma calidad de impresión pero tal que una de ellas es algo mas rápida. Un usuario puede querer usar indistintamente cualquiera de estas impresoras con lo que preferiría considerarlas como dos ejemplares del mismo recurso. Sin embargo, otro usuario que necesite con urgencia imprimir un documento requeriría verlas como dos recursos separados para poder especificar la impresora más rápida. Lo razonable en este tipo de situaciones

sería proporcionar a los usuarios ambas vistas de los recursos, Sin embargo, las soluciones clásicas del ínterbloqueo no contemplan esta posibilidad de un doble perfil: o son recursos independientes o son ejemplares del mismo recurso. En esta exposición, por tanto, se adoptará también esta restricción.

A los usuarios de un sistema operativo convencional les puede parecer que este tipo de organización de los recursos no se corresponde con su experiencia de trabajo habitual en la que, generalmente, hay que solicitar una unidad específica de un recurso. Sin embargo, aunque no sea de forma evidente, este tipo de situaciones se dan hasta cierto punto en todos los sistemas operativos. Considérese el caso de la memoria. Se trata de un único recurso con múltiples unidades: cada palabra que forma parte de la misma. Cuando un proceso solicita una reserva de memoria de un determinado tamaño, está solicitando el número de unidades de ese recurso que corresponde con dicho tamaño. Observe que en este caso existe una restricción adicional, ya que las unidades asignadas, sean cuales sean, deben corresponder con posiciones de memoria contiguas. A continuación se muestra un ejemplo de interbloqueo en el uso de la memoria.

Considérese la ejecución de los dos procesos siguientes, suponiendo que se dispone de *450* KB de memoria:

Proceso P_1	**Proceso P_1**
Solicita (l00 K)	Solicita(200 K)
Solicita(l00 K)	Solicita(l00 K)
Solicita (l00 K)	

Si se produce un orden de ejecución tal que se satisfacen las dos primeras solicitudes del primer proceso y la primera del segundo, se produciría un ínterbloqueo puesto que la cantidad de memoria libre *(50* KB) no es suficiente para cumplir con ninguna de las peticiones pendientes.

Aunque ya se comentó previamente que los recursos consumibles quedaban fuera del alcance de esta exposición, es interesante resaltar que también pueden existir múltiples unidades asociadas a un recurso de este tipo. Así, por ejemplo, un semáforo se puede considerar un recurso consumible que tiene tantas instancias como indica el contador asociado al mismo (Aclaración 6.1).

ACLARACIÓN 6.1

Los semáforos: ¿Recursos reutilizables o consumibles?

Puede parecer sorprendente que, a la hora de clasificar los recursos, se hayan considerado de forma diferente los semáforos de tipo *mutex* y los semáforos generales. Los primeros se han clasificado como reutilizables mientras que los segundos como consumibles. ¿A qué se debe esta diferencia?

Un semáforo tipo *mutex* responde al patrón de un recurso reutilizable exclusivo con un solo ejemplar asociado. Una vez creado, los procesos lo usan de forma exclusiva hasta que se destruye. Con este tipo de semáforos se producen situaciones de interbloqueo similares a la planteada en el ejemplo anterior que mostraba dos procesos que usaban una cinta y una impresora. En el siguiente ejemplo, simplemente se han sustituido esos dos dispositivos por dos semáforos de tipo *mutex*.

Proceso P_1	**Proceso P_2**
`lock(M1)`	`lock(M2)`
`lock(M2)`	`lock(M1)`
`.........`	`.........`
`unlock(M2)`	`unlock(M1)`
`unlock(M1)`	`unlock(M2)`

Como en el caso anteriormente citado, puede darse un interbloqueo si los dos procesos consiguen cerrar el primer *mutex* y se quedan bloqueados esperando por el segundo. Observe que este interbloqueo se produciría con independencia de lo que puedan hacer otros procesos existentes en el sistema.

En cuanto a los semáforos generales, su patrón de comportamiento corresponde con un recurso consumible con tantas unidades asociadas como indique el contador del semáforo. Cuando un proceso realiza una operación `signal` sobre un semáforo se está produciendo una nueva unidad de ese recurso, mientras que cuando hace un `wait` se está consumiendo una. El comportamiento, por tanto, de este tipo de recursos es muy diferente al de los semáforos de tipo *mutex*. Así, considérese qué ocurriría si en el ejemplo anterior se sustituyen los *mutex* por semáforos iniciados con un contador igual a 1:

Proceso P_1	**Proceso P_2**
`wait(S1)`	`wait(S2)`
`wait(S2)`	`wait(S1)`
`.........`	`.........`
`signal(S2)`	`signal(S1)`
`signal(S1)`	`signal(S2)`

Aparentemente, los procesos mantienen el mismo comportamiento. Sin embargo, hay un aspecto sutil muy importante. En este caso, el comportamiento puede depender de otros procesos externos que podrían generar nuevas unidades de estos dos recursos (o sea, podrían incluir llamadas a `signal` sobre cualquiera de los dos semáforos). Por tanto, el posible interbloqueo detectado en el ejemplo anterior podría no darse en este caso siempre que algún proceso externo lo rompiera. Observe que éste es uno de los motivos que complican el tratamiento general de los interbloqueos con recursos consumibles.

Recursos expropiables o no

Algunas de las estrategias para el tratamiento de los ínterbloqueos que se estudiarán a lo este capítulo implicarán la expropiación de recursos, o sea, la revocación de un recurso un proceso, mientras éste lo está usando, para otorgárselo a otro proceso. Para evitar la pérdida de información, esta expropiación implica salvar de alguna forma el trabajo que llevaba hecho el

proceso con el recurso expropiado. La posibilidad de llevar a cabo esta expropiación de forma relativamente eficiente va a depender de las características específicas del recurso, pudiéndose, por tanto, clasificar los recursos de acuerdo a este criterio.

Algunos recursos tienen un carácter no expropiable ya que o bien no es factible esta operación o, en caso de serlo, sería ineficiente, Por ejemplo, no tendría sentido quitarle a un proceso un trazador gráfico cuando lo está usando ya que con ello se perdería todo el trabajo realizado.

Otros recursos, sin embargo, pueden expropiarse de manera relativamente eficiente. Considérese, por ejemplo, el caso de un procesador. Cada vez que se produce un cambio de proceso, el sistema operativo está expropiando el procesador a un proceso para asignárselo a otro. La expropiación de un procesador implicaría únicamente salvar el estado del mismo en el bloque de control del proceso correspondiente para que así éste pueda seguir ejecutando normalmente cuando se le vuelva a asignar. Observe que, conceptualmente, el procesador, como cualquier otro recurso reutilizable de uso exclusivo, puede verse implicado en ínterbloqueos. Suponga, por ejemplo, una situación en la que existe un proceso que tiene asignada una cinta y está en estado listo para ejecutar (o sea, no tiene asignado el procesador). Si el proceso que está en ejecución (o sea, que tiene asignado el procesador) solicita la cinta, se producirá un ínterbloqueo ya que cada proceso necesita un recurso que posee el otro. Este ínterbloqueo potencial no ocurre en la práctica ya que en un sistema multiprogramado, cuando el proceso en ejecución se bloquea, se asigna automáticamente el procesador a otro proceso (gracias al carácter expropiable de este recurso).

En los sistemas que utilizan un dispositivo de almacenamiento secundario (generalmente un disco) como respaldo de la memoria principal (como son los sistemas con intercambio o con memoria virtual que se estudiaron en el Capitulo 4), el recurso memoria también es expropiable. Cuando se requiera expropiar a un proceso p te o toda la memoria que esta usando, se copia el contenido de la misma en el dispositivo de respaldo dejándola libre para que pueda usarla otro proceso. El ejemplo de ínterbloqueo en el uso de la memoria planteado previamente podría resolverse de esta forma. Observe que dicha operación de copia tiene un coste asociado que afectara al rendimiento del sistema,

El análisis realizado en esta sección ha permitido clasificar los diferentes tipos de recursos y delimitar cuáles de ellos van a considerarse en el resto de esta exposición, a saber, los recursos reutilizables exclusivos compuestos por una o más unidades, En la siguiente sección se definirá un modelo formal del sistema bajo estas restricciones, que servirá de mareo de referencia para poder caracterizar el problema del ínterbloqueo.

6.3. UN MODELO DEL SISTEMA

Desde el punto de vista del estudio del ínterbloqueo, en un sistema se pueden distinguir las siguientes entidades y relaciones:

- Un conjunto de procesos (o *threads).*
- Un conjunto de recursos reutilizables de uso exclusivo, Cada recurso consiste a su vez de un conjunto de unidades,
- Un primer conjunto de relaciones entre procesos y recursos que define qué asignaciones de recursos están vigentes en el sistema en un momento dado, Esta relación define si un proceso tiene asignadas unidades de un determinado recurso

y, en caso afirmativo, cuántas.
• Un segundo conjunto de relaciones entre procesos y recursos que define qué solicitudes de recursos están pendientes de satisfacerse en el sistema en un momento dado. Esta relación define si un proceso tiene unidades de un determinado recurso pedidas y no concedidas y, en caso afirmativo, cuántas.

Estos dos conjuntos de relaciones no pueden tomar cualquier valor sino que deben cumplir siguientes **restricciones de coherencia** para que un estado de asignación de recursos sea válida

1. *Restricción de asignación:* el número total de unidades asignadas de un recurso tiene que ser menor o igual que el numero de unidades existentes del mismo (no se puede dar mas de lo que hay).
2. *Restricción de solicitud:* el número total de unidades de un recurso solicitadas (tanto asignadas como pendientes de asignar) por un proceso en un determinado momento tiene que ser menor o igual que el número de unidades existentes del recurso. Observe que esta restricción asegura que ningún proceso quiere usar en un momento dado mas de un recurso que las existentes.

De manera general, e independientemente del tipo de cada recurso en particular o de un sistema operativo específico, se van a considerar dos primitivas abstractas para trabajar con los que, dado su carácter genérico, permiten representar cualquier operación específica presente en un sistema operativo determinado, A continuación, se especifican estas primitivas.

- *Solicitud* (s (R_1 [U_1], ... **,** R_n [U_n])): permite que un proceso que, evidentemente, bloqueado pida varias unidades de diferentes recursos (u_1 unidades del recurso 1, u_2 des del recurso 2, etc.). Si **todos** los recursos solicitados están disponibles, se concederá la petición, asignando al proceso dichos recursos. En caso contrario, se bloqueara el proceso sin reservar ninguno de los recursos solicitados, aunque algunos de ellos estén disponibles. El proceso se desbloqueará cuando todos estén disponibles (como resultado operaciones de liberación), Observe que se asume un modo de operación que se podría calificar como de *todo-o-nada:* hasta que no estén todos los recursos disponibles no se asigna ninguno (el Ejercicio *6.5* plantea al lector analizar las repercusiones de modificar este comportamiento).
- *Liberación* (L (R_1 [U_1] *,...,* R_n [U_n])): permite que un proceso que, evidentemente, bloqueado libere varias unidades de diferentes recursos que tenga asignadas en ese momento. La liberación de estos recursos puede causar que se satisfagan solicitudes pendientes de otros procesos, provocando su desbloqueo.

La generalidad de estas primitivas supera a lo que normalmente proporcionan los si
operativos, como se analiza en la Aclaración 6.2,

Existen diversas maneras de representa esta información. Aunque todas las representaciones de este modelo son lógicamente equivalentes, es conveniente mostrar diferentes alternativas que un esquema de representación puede ser mas adecuado que otro para expresar un determinado algoritmo de tratamiento del interbloqueo. En

concreto, se han seleccionado las dos representaciones mas habituales: el grafo de asignación de recursos [Holt, 972]y la representación matricial.

6.3.1. Representación mediante un grafo de asignación de recursos.

Un grafo de asignación de recursos G consiste de un conjunto de nodos N y un conjunto de aristas A: G={{N}, {A}}.

- El conjunto de nodos N se descompone a su vez en dos subconjuntos disjuntos que se corresponden con los procesos P y los recursos R. Cada recurso R_1 tiene asociado un valor que representa cuántas unidades del recurso existen (su inventario).

ACLARACIÓN 6.2

Las primitivas de solicitud y liberación en los sistemas reales

En un sistema real generalmente no hay una única primitiva para solicitar recursos o para liberarlos. Dada la gran variedad y heterogeneidad de los recursos, existen servicios específicos para distintos tipos de recursos. Algunos ejemplos en POSIX serían los siguientes: `open` y `close` para el acceso a dispositivos, `malloc` y `free` para la reserva y liberación de memoria, `pthread_mutex_lock` y `pthread_mutex_unlock` para manejar semáforos de tipo *mutex* y `fcntl` para establecer y quitar cerrojos a archivos.

Además de esta diversidad, otro factor que diferencia a las primitivas genéricas propuestas de las reales es su mayor funcionalidad. Normalmente, los servicios ofrecidos por los sistemas operativos no permiten solicitar o liberar múltiples recursos simultáneamente. Existen algunos ejemplos reales de servicios que sí tienen ese comportamiento, aunque generalmente no son tan genéricos no pudiéndose aplicar a todos los tipos de recursos del sistema.

Así, por ejemplo, como se vio en el Capítulo 5, el servicio de Win32 `WaitForMultipleObjects` permite solicitar múltiples recursos, tanto correspondientes con elementos de sincronización como relacionados con la terminación de procesos o *threads* (observe que la sincronización de un proceso con otro que termina se corresponde también con una situación de uso de recursos y, por tanto, puede implicar interbloqueos). Sin embargo, a diferencia de la primitiva de solicitud abstracta especificada anteriormente, este servicio permite especificar dos modalidades de comportamiento: un modo de operación *todo-o-nada* similar al de la primitiva abstracta y un modo alternativo en el que la solicitud se satisface en cuanto haya al menos un recurso disponible. Se podría considerar que el primer modo de operación es una solicitud con un comportamiento lógico *Y*, mientras que el segundo sigue un modelo lógico *O*. Dado que la literatura sobre interbloqueos no ha tratado este segundo modelo, esta exposición se centrará únicamente en el primero.

En el caso de UNIX, un ejemplo de operaciones que permiten solicitar o liberar múltiples recursos simultáneamente serían las proporcionadas por los semáforos de *SystemV* que permiten trabajar con vectores de semáforos. Por lo que se refiere a servicios que tengan un comportamiento lógico *O*, se podría considerar como ejemplo la primitiva `select` que permite esperar que se produzca un determinado evento en al menos un descriptor del conjunto especificado.

Un último aspecto a destacar es que, como se analizará más adelante, el uso de primitivas que permitan solicitar múltiples recursos simultáneamente favorece el desarrollo de programas libres de interbloqueos.

- El conjunto de aristas también se descompone en dos subconjuntos que se corresponden con las dos relaciones antes planteadas:

  — Aristas de asignación que relacionan recursos con procesos. Una arista entre un recurso R_1 y un proceso P_j indica que el proceso tiene asignada una unidad de dicho recurso.
  — Aristas de solicitud que relacionan procesos con recursos. Una arista entre un

proceso P_1y un recurso R_j indica que el proceso está esperando la concesión de una unidad de dicho recurso.

Las restricciones de coherencia anteriormente especificadas se plasmarían en el grafo de asignación de recursos de la siguiente manera:

1. *Restricción de asignación:* el numero de aristas que salen de un recurso debe ser menor o igual que su inventario,
2. *Restricción de solicitud:* se de e cumplir que por cada pareja proceso *i* y recurso *j,* el número de aristas de asignación que van de Rj a P_1 mas el número de aristas de solicitud que conectan P_1 a Rj sea menor o igual que el inventario.

A partir de la especificación de las primitivas genéricas de solicitud y liberación, se puede analizar cómo afectaría su procesamiento al grafo que representa el estado del sistema.

- Solicitud del proceso i de u_1 unidades del recurso 1, u_2 del recurso 2, etc. Se presentan dos situaciones dependiendo de si todos los recursos pedidos están disponibles o no.

— Sí no lo están, se bloquea el proceso añadiendo al grafo, por cada recurso pedido j, tantas aristas desde el nodo P_1 hasta R_j como unidades se hayan solicitado (u_j). Cuando el proceso se desbloquee posteriormente, al quedar disponibles todos los recursos requeridos, se eliminaran del grafo todas estas aristas de solicitud y se añadirán las aristas de asignación que en el caso de que los recursos hubiesen estado disponibles desde el principio.
— Si están disponibles, ya sea desde el principio o posteriormente, se añaden al grafo por cada recurso pedido *j* tantas aristas desde el nodo R_1 hasta P_i como unidades se hayan solicitado (u_j).

- Liberación por parte del proceso *i* de u_1 unidades del recurso 1, u_2 del recurso 2, etc. Por cada recurso liberado j se eliminan del grafo tantas aristas desde el nodo R_1 hasta P_1 como unidades se hayan dejado libres (u_1),

Observe que sólo se añaden aristas en el grafo durante las solicitudes, tanto si se trata de solicitud como de asignación. En cuanto a la eliminación de aristas, en la liberación se quitan aristas de asignación, mientras que las de solicitud se retiran en el desbloqueo de un proceso que realizó una petición.

A continuación se muestran dos ejemplos del uso de un grafo de asignación de recursos representar un sistema.

En primer lugar, supóngase que en un sistema con 3 recursos, R_1 (2 unidades), R_2 *(3* unida y R_3 (2 unidades), se ejecutan tres procesos que realizan la siguiente traza de peticiones:

1. P_1: solicita(R_1[2]) → solícita 2 unidades del recurso

2 P_2:solicita(R_2[1])
3. P_2: solicita (R_1 11]) → se bloquea puesto que el recurso no está disponible
4. P_3: solicita(R_2[1])

El grafo que representa la situación del sistema después de ejecutarse esta secuencia es el siguiente:

•N ={P_1,P_2,P_3,R_1(2) ,R_2(3) ,R_3(2) }

•A ={R_1→ P_1, R_1 →P_1,R_2 →P_2,P_2→ R_1, R_2 →P_3,}

Para poder entender de forma intuitiva el estado de un sistema es conveniente establecer una representación gráfica del grafo de asignación de recursos. La convención que se suele utilizar es la siguiente:

- Cada proceso se representa con un circulo,
- Cada recurso con un cuadrado. Dentro del cuadrado que representa a un determinado recurso se dibuja un círculo por cada unidad existente del recurso.
- Las aristas de solicitud se representan como arcos que van desde el proceso hasta el cuadrado que representa al recurso, mientras que las aristas de asignación se dibujan como arcos que unen el circulo que representa una unidad determinada del recurso con el proceso correspondiente.

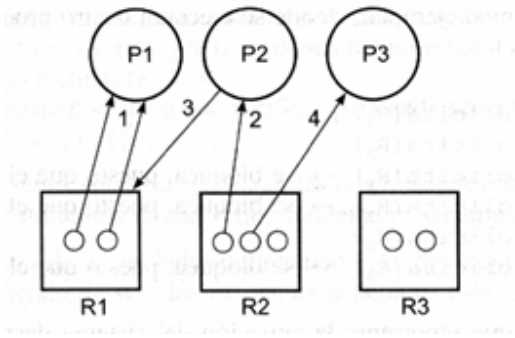


Figura 6.1. *Grafo de asignación de recursos después de la primera secuencia de peticiones.*

Siguiendo esta convención, en la Figura 6.1 se muestra la representación gráfica correspondiente al grafo del ejemplo anterior.
Continuando con el ejemplo, considere que a continuación se realizan las siguientes peticiones:

5. P_3: solicita(R_2[1])
6. P_1: solicita (R_2 [11 , R_3 [2]) →se bloquea, pues uno de los recursos no está disponible

El grafo que representa la situación del sistema después de ejecutarse esa secuencia es el siguiente:

• N={P_1, P_2, P_3, R_1(2),R_2(3),R_3(2)}

•A ={R_1→ P_1, R_1 →P_1,R_2 →P_2,P_2→ R_1, R_2 →P_3, R_2 →P_3 ,P_1→R_2 ,P_1→ R_3, P_1→ R_3}

La Figura 6.2 muestra la representación grafica de este grafo resultante,

Generalmente, si en el sistema sólo hay una unidad de cada recurso, se prescinde de los pequeños círculos que representan los ejemplares del recurso y se dibujan las aristas de asignación, como arcos que unen el cuadrado que representa un determinado recurso con el proceso correspondiente. Así, como segundo ejemplo, considere un sistema con 3 recursos, R_1, R_2 y R_3, compuestos todos

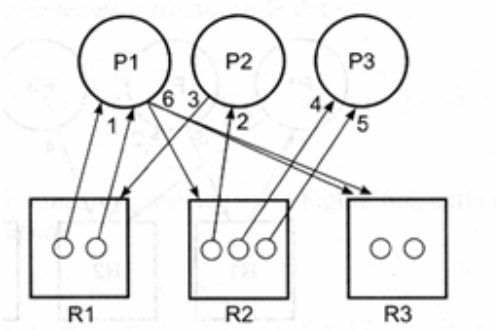


Figura 6.2. *Grafo de asignación de recursos después de la segunda secuencia de peticiones.*

ellos por un único ejemplar, donde se ejecutan cuatro procesos que realizan la siguiente traza de peticiones:

1. P_1:solicita(R_1)
2. P_2:solicita(R_2)
3. P_2:solicita (R_1) → se bloquea, puesto que el recurso no está disponible
4. P_3:solicita (R_2) → se bloquea, puesto que el recurso no está disponible
5. P_4:solicita(R_3)
6. P_1:solicita (R_2) → se bloquea, puesto que el recurso no está disponible

El grafo que representa la situación del sistema después de ejecutarse esa secuencia es el siguiente:

•N = {P_1, P_2, P_3, P_4,R_1(l),R_2(1) R_3(l)}

• A = {R_1→P_1,R_2→P_2, P_2→R_1, P_3→R_2, R_3→P_4, P_1 →R_2}

La Figura 6.3 muestra la representación gráfica de este grafo de asignación de recursos,

6.3.2. Representación matricial

Una forma alternativa de representar esta información es mediante el uso de matrices. Para representar el estado del sistema se usan dos matrices: una matriz de solicitud S y una de asignación A .Además, por cada recurso se debe guardar el numero de unidades existentes del mismo: un vector E que contiene el inventario de cada recurso. Siendo p el número de procesos existentes (o sea p = | P |) y *r* el número de recursos diferentes que hay en el sistema (o sea, *r* = | R |). el significado de las estructuras de datos es el siguiente:

- **Matriz de asignación**, A, de dimensión p x *r*. La componente A [i, j] de la matriz especifica cuántas unidades del recurso *j* están asignadas al proceso *i.*
- **Matriz de solicitud,** s, de dimensión p x *r*. La componente s [i, j] de la matriz especifica cuántas unidades del recurso *j* está esperando el proceso i que se le concedan.
- **Vector de recursos existentes,** E, de dimensión *r*. La componente R [1] especifica cuantas unidades del recurso *i* existen.

En este tipo de representación, las restricciones de coherencia del sistema implicarían lo siguiente:

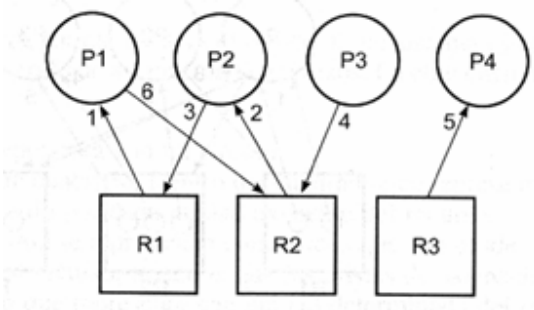


Figura 6.3. *Grafo de asignación para recursos con un único ejemplar.*

1. *Restricción de asignación:* se debe cumplir para cada recurso i que la suma de la columna i de la matriz A (ΣA [j ,i], para j = 1, *..., p)* sea menor o igual que el número de unidades existentes de dicho recurso (E [j]).
2. *Restricción de solicitud:* se tiene que verificar para cada proceso i y recurso j la siguiente expresión A[i, j]+S[i,j] <= E[j].

Cuando se utiliza una representación matricial, las repercusiones de las operaciones de solicitud y liberación sobre las estructuras de datos que modelan el sistema serian las siguientes:

- Solicitud del proceso i de u_1 unidades del recurso 1, u_2 del recurso 2, etc. Se presentan dos situaciones, dependiendo de si todos los recursos pedidos están disponibles o no.
  - — Si no lo están, se bloquea el proceso y se actualiza la matriz de solicitud: por cada recurso pedido *j,* [j, j] = S [i, j] + U_j, Cuando el proceso se desbloquee posteriormente, al quedar disponibles todos los recursos requeridos, se elimina el efecto de esta suma realizando la resta correspondiente (por cada recurso pedido j, *s* [j, j] = S [i, j] - u_j) y se actualiza la matriz de asignación de la misma manera que se hace cuando los recursos están disponibles desde

el principio.
— Si están disponibles, ya sea desde el principio o posteriormente, se actualiza la matriz de asignación: por cada recurso pedido j, A[i, j] = A[j, ji + U_j.

• Liberación por parte del proceso i de u_1 unidades del recurso 1, u_2 unidades del recurso 2, etcétera. Por cada recurso liberado j se sustraen de la matriz de asignación las unidades liberadas: A[j,]=A[i ,j]- U_j .

Estas estructuras son suficientes para reflejar el estado del sistema. Sin embargo, para simplificar la especificación de algunos algoritmos que se expondrán más adelante es útil usar un vector de recursos disponibles D que refleje el numero de unidades de cada recurso disponibles en un momento dado. Observe que este vector es innecesario, ya que su valor se puede deducir directamente a partir de la matriz de asignación A y del vector de recursos existentes E: D [i] = E [i] - **Σ**[j , i] , para j = 1, ...,p.

Para hacer más concisa la especificación de los algoritmos se utilizará a partir de ahora la siguiente notación compacta:

• Dada una matriz A, el valor A [i] representa un vector que corresponde con la fila i-ésima de dicha matriz.
• Dados dos vectores A y B de longitud *n,* se considera que A <= B si A [i] <= B [i] para todo j = 1, ..., *n.*

Si se utiliza este modo de representación para el ejemplo representado en la Figura 6.1, se obtiene como resultado las siguientes estructuras de datos:

$$A=\begin{pmatrix}2&0&0\\0&1&0\\0&1&0\end{pmatrix}\quad S=\begin{pmatrix}0&0&0\\1&0&0\\0&0&0\end{pmatrix}\quad E=[232]\quad D=[0\ 0\ 2]$$

La evolución de este sistema, representada en la Figura 6.2, haría que las estructuras de datos quedaran de la siguiente manera:

$$A=\begin{pmatrix}2&0&0\\0&1&0\\0&2&0\end{pmatrix}\quad S=\begin{pmatrix}0&0&0\\1&0&0\\0&0&0\end{pmatrix}\quad E=[2\ 3\ 2]\quad D=[0\ 0\ 2]$$

Por lo que se refiere al ejemplo con un único ejemplar por recurso, correspondiente a la Figura 6.3, las matrices resultantes serían:

$$A=\begin{pmatrix}1&0&0\\0&1&0\\0&0&0\\0&0&1\end{pmatrix}\quad S=\begin{pmatrix}0&1&0\\1&0&0\\0&1&0\\0&0&0\end{pmatrix}\quad E=[1\ 1\ 1]\quad D=[0\ 0\ 0]$$

Con independencia del esquema de representación usado, es importante resaltar que un sistema real tiene un carácter dinámico. En un sistema informático se están continuamente creando y destruyendo tanto procesos como recursos (considere, por ejemplo, archivos). Por tanto, las estructuras de información usada: para modelar el sistema tendrán también una evolución dinámica. En el caso de un grafo, no sólo se añadirán y eliminarán aristas, sino también nodos. En cuanto a la representación

matricial, las dimensione de las matrices cambiarán dinámicamente. Este aspecto dificulta considerablemente el tratamiento de este problema. De hecho, la mayoría de las solucione planteadas en la literatura que estudia este tema obvian este comportamiento dinámico del sistema.

6.4. DEFINICIÓN Y CARACTERIZACIÓN DEL INTERBLOQUEO

Una vez establecido el modelo del sistema, parece que ya es momento de intentar definir más formalmente el interbloqueo para poder así caracterizarlo. Una posible definición de interbloqueo inspirada en [Tanenbaum, 1992] seria la siguiente:

Un conjunto de procesos está en interbloqueo si cada proceso está esperando un recurso que sólo puede liberar (o generar, en el caso de recursos consumibles) otro proceso del conjunto.

Normalmente, cada pro eso implicado en el interbloqueo estará bloqueado esperando un recurso, pero esto no es estrictamente necesario ya que el interbloqueo también existiría aunque los procesos involucrados realizasen una espera activa, La espera activa tiene como consecuencia un uso innecesario del procesador, pero, por lo que se refiere a los interbloqueos, no implica ninguna diferencia.

A partir de esta definición, es preciso caracterizar un interbloqueo. Coffman [Coffman, 1971] identifica las siguientes condiciones como necesarias para que se produzca un interbloqueo:

1. **Exclusión mutua.** Los recursos implicados deben usarse en exclusión mutua, o sea, debe tratarse de recursos de uso exclusivo. Como se analizó previamente en esta exposición, los recursos compartidos no están involucrados en interbloqueos.
2. **Retención y espera**. Cuando no se puede satisfacer la petición de un proceso, este se bloquea manteniendo los recursos que tenía previamente asignados. Se trata de una condición que refleja una forma de asignación que corresponde con la usada prácticamente en todos los sistemas reales,
3. **Sin expropiación**. No se deben expropiar los recursos que tiene asignado un proceso. Un proceso sólo libera sus recursos voluntariamente.
4. **Espera circular.** Debe existir una lista circular de procesos tal que cada proceso en la lista esté esperando por uno o más recursos que tiene asignados el siguiente proceso.

Estas cuatro condiciones no son todas de la misma índole. Las tres primeras tienen que ver con aspectos estáticos del sistema, como que características deben tener los recursos implicados o cómo

debe ser la política de asignación de recursos. Sin embargo, la condición de espera circular refleja cómo debe ser el comportamiento dinámico de los procesos para que se produzca el interbloqueo.

Es importante resaltar que se trata de condiciones necesarias pero no suficientes (o sea, que la ocurrencia de las condiciones no asegura la presencia del interbloqueo, pero para que exista un interbloqueo tienen que cumplirse). Sirvan como ejemplo las dos situaciones estudiadas previamente. En ambos casos se cumplen las cuatro condiciones (las tres primeras se satisfacen debido a que corresponden con las suposiciones que se han hecho sobre el sistema) y, sin embargo, como se analizará más adelante, sólo en el

segundo existe un interbloqueo. En los dos ejemplos, correspondientes a las Figuras 6.2 y 6.3 respectivamente, se puede identificar la siguiente lista de espera circular: P_1 está esperando por un recurso que mantiene P_2 que, a su vez, está esperando por un recurso asignado a P_1.

Además de por tratarse de una reseña histórica casi obligatoria, estas condiciones necesarias son interesantes porque, como se analizará más adelante, sirven de base para el desarrollo de estrategias de prevención de interbloqueos. Sin embargo, no bastan para poder caracterizar el interbloqueo adecuadamente: se precisa establecer las condiciones necesarias y suficientes para que se produzca un interbloqueo.

6.4.1. Condición necesaria y suficiente para el interbloqueo

La caracterización del interbloqueo se basa en mirar hacia el futuro del sistema de una manera «optimista», siguiendo la idea que se expone a continuación.
Dado un sistema con un determinado estado de asignación de recursos, un proceso cualquiera que no tenga peticiones pendientes (por tanto, desbloqueado) debería devolver en un futuro más o menos cercano todos los recursos que actualmente tiene asignados. Esta liberación tendría como consecuencia que uno o más procesos que estuvieran esperando por estos recursos se pudieran desbloquear. Los procesos desbloqueados podrían a su vez devolver más adelante los recursos que tuvieran asignados desbloqueando a otros procesos, y así sucesivamente. Si todos los procesos del sistema terminan desbloqueados al final de este análisis del futuro del sistema, el estado actual está libre de interbloqueo. En caso contrario existirá un interbloqueo en el sistema estando implicados en el mismo lo. procesos que siguen bloqueados al final del análisis, A este proceso de análisis se le suele denomina **reducción.** A continuación se define de una forma más precisa:

Se dice que el estado de un sistema se puede reducir por un proceso P si se pueden satisfacer las necesidades del proceso con los recursos disponibles.

Como parte de la reducción, el proceso devolverá los recursos asignados, tanto los que tenía previamente como los que acaba de obtener, añadiéndolos al sistema creándose un nuevo estado hipotético. Gracias al aporte de recursos fruto de la reducción por P, ese nuevo estado podrá a su vez reducirse por uno o más procesos. A partir del concepto de reducción se puede establecer la condición necesaria y suficiente p a que se produzca un interbloqueo.

La condición necesaria y suficiente para que un sistema esté libre de interbloqueos es que exista una secuencia de reducciones del estado actual del sistema que incluya a todos los procesos del sistema. En caso contrario, hay un interbloqueo en el que están implicados los procesos que no están incluidos en la secuencia de reducciones.

Observe que, en un determinado paso de una secuencia de reducción, podría haber varios procesos a los que aplicar la siguiente reducción ya que se satisfacen sus necesidades de recursos. En esta situación se podría elegir cualquiera de ellos, ya que el proceso de reducción no depende del orden. Para poder demostrar esta propiedad sólo es necesario darse cuenta de que el proceso de reducción es acumulativo, esto es, en cada paso de reducción se mantienen los recursos disponibles

que había hasta entonces, añadiéndose los liberados en la reducción actual. Por tanto, si en un determinado punto de la secuencia se cumplen las condiciones para poder aplicar la reducción por un proceso, éstas se seguirán cumpliendo aunque se realice la reducción por otro proceso.

En la sección que analiza el tratamiento de los interbloqueos basándose en la

detección y recuperación se presentarán algoritmos que se basan en este principio. A continuación se aplicará los ejemplos planteados hasta ahora.

En el ejemplo representado en la Figura 6,2 se puede identifica la siguiente secuencia reducciones:

1. Se puede reducir el estado por P_3 que no está pendiente de ningún recurso. Se liberan dos unidades de R_2.
2. Gracias a esas dos unidades, se puede reducir por P_1 ya que están disponibles todos los recursos que necesita (2 unidades de R_3 y una de R_2). Como resultado de la reducción, se liberan dos unidades de R_1.
3. Se produce la reducción por P_2 puesto que ya está disponible la unidad de R_1 que necesitaba.

Dado que la secuencia de reducciones *(*P_3 ,P_1, P_2*)* incluye a todos los procesos, el sistema esta libre de interbloqueos.

Por lo que se refiere al ejemplo correspondiente a la Figura 6.3, la secuencia de reducciones sería la siguiente: se puede reducir el estado por P_4 que no está pendiente de ningún recurso liberándose una unidad de R_3.

Sólo se puede realizar esta reducción, por tanto, existe un interbloqueo en el sistema en el están involucrados el resto de los procesos.

Como se verá más adelante, la aplicación directa de este principio lleva a algoritmos relativamente complejos. Sin embargo, si se consideran sistemas con algún tipo de restricción, se reduce apreciablemente el orden de complejidad de los algoritmos. Así, en el caso de un sistema sola unidad de cada recurso, la caracterización del interbloqueo se simplifica ya que las con necesarias de Coffman son también suficientes.

6.5. TRATAMIENTO DEL INTERBLOQUEO

Como se vio al principio del capítulo, las técnicas para tratar el interbloqueo pueden clasificarse en tres categorías:

- Estrategias de detección y recuperación.
- Estrategias de prevención.
- Estrategias de predicción.

Antes de analizar en detalle cada una de ellas, es interesante comentar que este tipo de soluciones se emplea también en otros ámbitos diferentes como puede ser en el equipo o en el mantenimiento de una enfermedad. Así, por ejemplo, en el caso del mantenimiento de un equipo, la estrategia basada en la detección y recuperación consistiría en esperar a que determinado componente para sustituirlo, con la consiguiente parada del sistema mientras se produce la reparación. Una estrategia preventiva, sin embargo, reemplazaría periódicamente los componentes del equipo para asegurar que no se averían. Observe que esta sustitución periódica podría implicar que se descarten en componentes que todavía estén en buenas condiciones. Para paliar esta situación, la predicción se basaría en conocer *a priori* que síntomas muestra un determinado componente cuando se está acercando al final de su «vida» (p. ej., presenta una temperatura excesiva o produce una vibración). Esta estrategia consistiría en supervisar periódicamente el comportamiento

del componente y proceder a su sustitución cuando aparezcan los síntomas predeterminados. Aunque, evidentemente, este ejemplo no presenta las mismas características que el problema de los interbloqueos, permite identificar algunas de las ideas básicas sobre su tratamiento como, por ejemplo, el mal uso de los recursos que pueden implicar las técnicas preventivas o la necesidad de un conocimiento *a priori* que requieren las estrategias basadas en la predicción.

A continuación, se comentan los tres tipos de estrategias utilizadas para tratar el interbloqueo:

- **Detección y recuperación**: Se podría considerar que este tipo de estrategias conlleva una visión optimista del sistema. Los procesos realizan sus peticiones sin ninguna restricción pudiendo, por tanto, producirse interbloqueos. Se debe supervisar el estado del sistema para detectar el interbloqueo mediante algún algoritmo basado en las condiciones analizadas en la sección anterior. Observe que la aplicación de este algoritmo supondrá un coste que puede afectar al rendimiento del sistema. Cuando se detecta, hay que eliminarlo mediante algún procedimiento de recuperación que, normalmente, conlleva una pérdida del trabajo realizado hasta ese momento por algunos de los procesos implicados.
- **Prevención:** Este tipo de estrategias intenta eliminar el problema de raíz fijando una serie de restricciones en el sistema sobre el uso de los recursos que aseguran que no se pueden producir interbloqueos. Observe que estas restricciones se aplican a todos los procesos por igual, con independencia de qué recursos use cada uno de ellos. Esta estrategia suele implicar una infrautilización de los recursos puesto que un proceso, debido a las restricciones establecidas en el sistema, puede verse obligado a reservar un recurso mucho antes de necesitarlo.
- **Predicción** (en ingles, se suele usar el término *avoidance):* Esta estrategia evita el interbloqueo basándose en un conocimiento *a priori* de qué recursos va a usar cada proceso. Este conocimiento permite definir algoritmos que aseguren que no se produce un interbloqueo. Como ocurre con las estrategias de detección, a la hora de aplicar esta técnica es necesario analizar la repercusión que tiene la ejecución del algoritmo de predicción sobre el rendimiento del sistema. Además, como sucede con la prevención, generalmente provoca una infrautilización de los recursos.

Existe una cuarta alternativa que consiste en no realizar ningún tratamiento, o sea, **ignorar los interbloqueos**. Aunque parezca sorprendente *a priori,* muchos sistemas operativos usan frecuentemente esta estrategia de «esconder la cabe a debajo del ala», denominada «**política del avestruz**». Esta opción no es tan descabellada si se tiene en cuenta, por un lado, las restricciones que se han ido identificando a lo largo del capitulo que limitan considerablemente el tratamiento general de los interbloqueos en un sistema real y, por otro, las consecuencias negativas que conllevan las técnicas de tratamiento (como la baja utilización de los recursos o el coste de los algoritmos de tratamiento). Observe que ignorar el problema trae como consecuencia que, cuando se produce un interbloqueo, los procesos implicados seguirán indefinidamente bloqueados y, lo que puede ser más grave todavía, los recursos involucrados quedan permanentemente inutilizables. En la Sección 6.9 se analizará cuáles pueden ser las repercusiones de esta situación dependiendo de diversos factores.

En las siguientes secciones se estudian en detalle las tres estrategias presentadas: detección y recuperación, prevención y predicción.

6.6. DETECCIÓN Y RECUPERACIÓN DEL INTERBLOQUEO

Esta técnica de tratamiento de los interbloqueos presenta, como su nombre indica, dos fases:

- **Fase de detección:** Debe ejecutarse un algoritmo que determine si el estado actual del sistema esta libre de interbloqueos y que, en caso de que no lo este, identifique qué procesos.

están implicados en el interbloqueo. Dado que, como se analizará más adelante, los algoritmos de detección pueden tener una repercusión apreciable sobre el rendimiento del sistema un aspecto importante es establecer con qué frecuencia se ejecutará dicho algoritmo. En e caso de que el algoritmo detecte un interbloqueo, se activará la fase de recuperación d sistema.

- **Fase de recuperación**: Una vez detectado el interbloqueo, se debe aplicar una acción que lo elimine. Como se analizara más adelante, esto implica generalmente abortar la ejecución algunos de los procesos implicados liberando de esta forma los recursos que tuvieran asignados. Debe existir, por tanto, un algoritmo que determine a qué procesos se les aplica esa medida drástica.

6.6.1. Detección del interbloqueo

A partir de la condición necesaria y suficiente para el interbloqueo basada en el concepto de reducción que se presento en la sección anterior, se pueden diseñar algoritmos que detecten si un sistema está libre de interbloqueos. Se trata simplemente de trasladar dicho concepto a las características específicas de cada tipo de representación del sistema.

Algoritmo de detección para una representación mediante un grafo de recursos

En primer lugar, se va a definir cómo se aplica el concepto de reducción a la representación mediante un grafo:

Dado un grafo de asignación de recursos, éste se puede reducir por un proceso s_I *si. recursos disponibles satisfacen sus necesidades (el proceso, por tanto, está desbloqueado). reducción consiste en eliminar las aristas de solicitud que salen del nodo* P_i ($P_i \rightarrow R_i$ *,puesto que hay suficientes recursos disponibles) y las de asignación que llegan a dicho nodo* ($R_i \rightarrow P_i$ *,ya que se supone que el proceso devolverá todos sus recursos). Como fruto de la reducción, se obtiene nuevo grafo donde podrá haber nuevos proceso desbloqueados gracias a los recursos liberado los que se les podrá aplicar una nueva reducción.*

A partir de esta definición, se puede especificar directamente un algoritmo que detecte si 1 un interbloqueo en el grafo de asignación de recursos:

```
S = Ø;              /* secuencia de reducción. Inicialmente vacía */

D ={Conjunto de procesos desbloqueados y no incluidos en S};

Mientras (D! = ø) {

    /* se puede reducir por cualquier proceso de D: elige el primero */
    P1 =primer elemento de D;
    Reducir grafo por Pi
    Añadir Pi  a S y eliminarlo de D;
```

```
    Determinar qué procesos se desbloquean por la reducción (sus
    solicitudes pueden satisfacerse en el nuevo grafo) y añadirlos a D;
}
Si (S = =P)
/* si la secuencia contiene todos los procesos del sistema (P) */
```

No hay interbloqueo
Si no
Los procesos en el conjunto P-S están en un interbloqueo

Este algoritmo tiene una complejidad $O(p^2r)$, siendo p el número de procesos en el sistema y r el número de recursos, ya que el bucle principal puede ejecutarse una vez por cada proceso (p veces) y, en cada iteración del bucle, la operación que determina qué procesos se desbloquean por la reducción implica comparar las solicitudes de recursos de cada proceso que todavía no esté en la secuencia con los recursos disponible. en el grafo reducido (del orden de $p \times r$ comparaciones).

Como ejemplo, se va a aplicar a continuación este algoritmo al grafo dibujado en la Figura 6.2, cuya representación formal era la siguiente:

- $N=\{P_1,P_2,P_3,R_1(2),R_2(3),R_3(2)\}$
- $A = \{R_1\rightarrow P_1, R_1\rightarrow P_1,R_2\rightarrow P_2, P_2 \rightarrow R_1 ,R_2\rightarrow P_3, R_2\rightarrow P_3, P_1 \rightarrow R_2,P_1\rightarrow R_3, P_1\rightarrow R_3\}$

Cuando se aplica el algoritmo de detección a este grafo, el resultado es el siguiente:

1. Estado inicial: =Ø y D =$\{P_3\}$.
2. Reducción del grafo por P_3. El grafo reducido resultante, representado en la Figura 6.4, es:

$$A = \{R_1\rightarrow P_1, R_1\rightarrow P_1,R_2\rightarrow P_2, P_2 \rightarrow R_1 ,P_1 \rightarrow R_2 ,P_1\rightarrow R_3, P_1\rightarrow R_3\}$$

3. Se añade el proceso a la secuencia de reducción: S = $\{ P_3\}$.
4. Como resultado de la reducción se desbloquea P_1 ya que están disponibles todos los recursos que necesita (2 unidades de R_3 y una de R_2). D = $\{ P_1\}$.
5. Reducción del grafo por P_1. El grafo reducido resultante, representado en la Figura *6.5,* es:

$$A=\{ R_2\rightarrow P_2, P_2\rightarrow R_3\}$$

6. Se añade el proceso a la secuencia de reducción: $s = \{P_3,P_1\}$.
7. Como resultado de la reducción se desbloquea P_2. D = $\{ P_2 \}$.
8. Reducción del grafo por P3~ El grafo queda completamente reducido: A = 0 (Fig. 6.6).
9. Se añade el proceso a la secuencia de reducción: S = $\{ P_3,P_1, P_2\}$.
10. Como no se desbloquea ningún proceso, D está vacío por lo que termina el bucle,
11. Como S incluye a todos los procesos, el sistema está libre de interbloqueos.

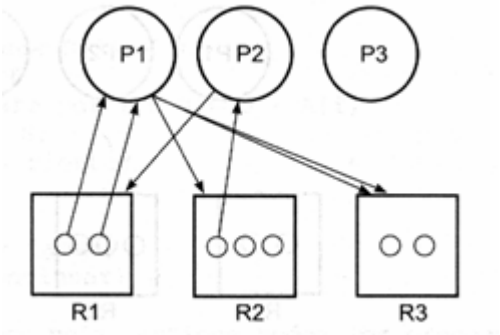


Figura 6.4. *Grafo de asignación de recursos después de la reducción por* P_3.

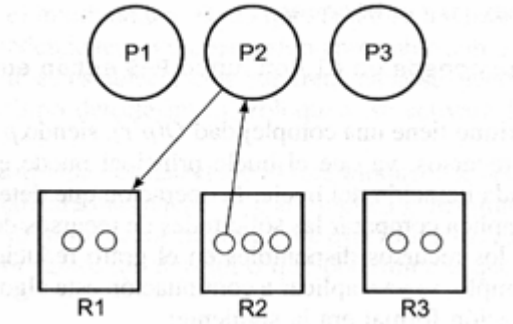


Figura *6.5*. *Grafo de asignación de recursos después de la reducción por* P_1.

A continuación, como segundo ejemplo, se aplica el algoritmo e detección al grafo de Figura 6.3, que tenía la siguiente representación formal:

- $N = \{P_1, P_2, P_3, P_4, R_1(1), R_2(1), R_3(1)\}$
- $A = \{R_1 \rightarrow P_1, R_2 \rightarrow P_2, P_2 \rightarrow R_1, P_3 \rightarrow R_2, R_3 \rightarrow P_4, P_1 \rightarrow R_2\}$

El resultado es el siguiente:

1. Estado inicial: $s = \emptyset$ y $D = \{P_4\}$.
2. Reducción del grafo por P_4. El grafo reducido resultante, representado en la Figura 6.7 es:

$$A = \{R_1 \rightarrow P_1, R_2 \rightarrow P_2, P_2 \rightarrow R_1, P_3 \rightarrow R_2, P_1 \rightarrow R_2\}$$

3. Se añade el proceso a la secuencia de reducción: $S = \{P_4\}$.
4. Como no se desbloquea ningún proceso, D está vacío por lo que termina el bucle.
5. Como S no incluye a todos los procesos, en el sistema hay un interbloqueo que afecta a los procesos del sistema que no están en S: P_1, P_2 y P_3.

Como se comentó previamente, cuando se consideran sistemas con algún tipo de restricción se pueden diseñar algoritmos de menor complejidad. Así, en el caso de un sistema con unidad de cada recurso, para asegurar que un sistema esta libre de interbloqueos sólo es necesario comprobar que no hay un ciclo en el grafo , dado que en este caso las condiciones de Coffman son necesarias y suficientes para que exista un interbloqueo. Por tanto, se puede usar para ello cual-

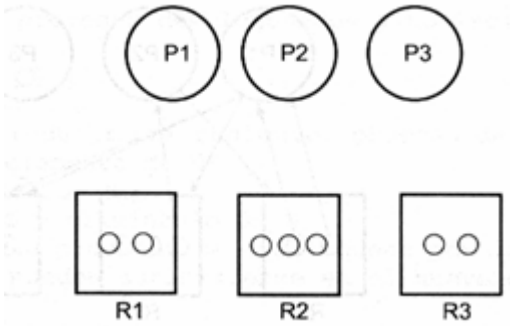


Figura 6.6. *Grafo de asignación de recursos después de la reducción por* P_2.

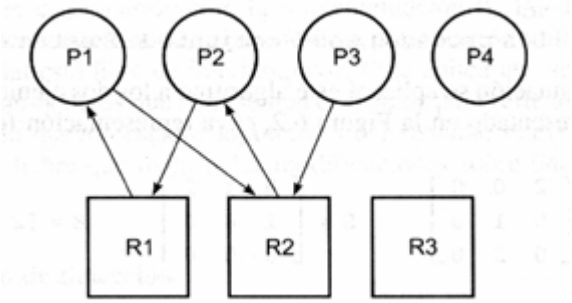


Figura 6.7. *Grafo de asignación de recursos después. de la reducción por* P_4.

quier algoritmo de detección de ciclos en un grafo. El orden de complejidad de este tipo de algoritmos es proporcional al número de aristas en el grafo. Como en un grafo de asignación de recursos, el número de aristas es proporcional a *p x r,* el algoritmo tiene una complejidad de *O(pr).*

Algoritmo de detección para una representación matricial

Es relativamente directo trasladar las ideas expuestas para el caso de una representación mediante un grafo a una matricial. Por ello, la exposición que trata este caso se hará de una forma más sucinta. Por lo que se refiere a la operación de reducción aplicada a una representación matricial, se podría definir de la siguiente forma:

Un sistema se puede reducir por un proceso P_i si los recursos disponibles satisfacen sus necesidades:

S[i] <=D (recuerde que esta notación compacta equivale a S[i, j] <=D[j] para j = 1, ..., *r)*

La reducción consiste en devolver los recurso asignado al proceso P_i:

$$D = D + A[i]$$

Una vez hecha esta definición, se puede especificar un algoritmo de detección similar al presentado para el caso de un grafo y con la misma complejidad *(O(p²r))*.

```
S = Ø; /* secuencia de reducción. Inicialmente vacía */

Repetir {
    Busca, Pi tal que S[i] <=D;
    Si Encontrado {
        Reducir grafo por Pi : D = D + A[i]
        Añadir Pi a S;
        Continuar = cierto;
    }
    else
        Continuar = falso;

    }Mientras (Continuar)
    Si (S = =P)
        /* si la secuencia contiene todos los procesos del sistema (P) */
```

No hay ínterbloqueo

Si no

Los procesos en el conjunto P-S están en un interbloqueo

A continuación se aplicará este algoritmo a los dos ejemplos considerados. En primer lugar sistema representado en la Figura 6.2, cuya representación formal era la siguiente:

$$A=\begin{pmatrix}2&0&0\\0&1&0\\0&2&0\end{pmatrix}\quad S=\begin{pmatrix}0&1&2\\1&0&0\\0&0&0\end{pmatrix}\quad E=[2\ 3\ 2]\quad D=[0\ 0\ 2]$$

El resultado es el siguiente:

1. Estado inicial: S = 0.
2. Se puede reducir por P_3 ya que S[3] <= D([012]<=[002]),dando como resultado:

$$D = D + A[3] = [0\ 0\ 2] + [0\ 2\ 0] = [0\ 2\ 2]$$

3. Se añade el proceso a la secuencia de reducción: S = { P_3}, y se pasa a la siguiente iteración:
4. Reducción por P_2 dado que S[2]<=D([1 0 0]<=[2 2 2]), que da como resultado:

$$D = D + A[1] = [0\ 2\ 2] + [2\ 0\ 1] = [2\ 2\ 2]$$

5. Se añade el proceso a la secuencia de reducción: S = { P_3, P_1}, y se pasa a la siguiente iteración.
6. Reducción por P_2 dado que S[2]<=D ([1 0 0]]<=[2 2 2]), que da como resultado:

$$D = D + A[2] = [0\ 1\ 0] + [2\ 2\ 2] = [2\ 2\ 3]$$

7. Se añade el proceso a la secuencia de reducción: S = { { P_3, P_1, P_2}, y se termina el bucle
8. Como S incluye a todos los procesos, el sistema está libre de interbloqueos.

En cuanto al segundo ejemplo, que corresponde con el sistema representado en la Figura su representación matricial era la siguiente:

$$A=\begin{pmatrix}1&0&0\\0&1&0\\0&0&0\\0&0&1\end{pmatrix}\quad S=\begin{pmatrix}0&1&0\\1&0&0\\0&1&0\\0&0&0\end{pmatrix}\quad E=[1\ 1\ 1]\quad D=[0\ 0\ 0]$$

La aplicación del algoritmo a este sistema produciría el siguiente resultado:

1. Estado inicial: *S* =0.
2. Se puede reducir por P_4 ya que S[4] <=D ([0 0 0]<=[0 0 0]), dando como resultado:

$$D = D + A[4] = [0\ 0\ 1] +[0\ 0\ 0\ ] = [0\ 0\ 1]$$

3. Se añade el proceso a la secuencia de reducción: S = { P_4}, y se pasa a la siguiente iteración.
4. No hay ningún P_i tal que S [i]<= D. Termina la ejecución del bucle.
5. Como S no incluye a todos los procesos, en el sistema hay un interbloqueo que

afecta a procesos del sistema que no es en S: P_1, P_2 y P_3.

Es necesario hacer un comentario sobre la implementación de los algoritmos de detección propuestos. Con independencia de cuál sea el modo de representación utilizado, cuando se intenta averiguar si el estado actual esta libre de interbloqueos, se le aplica una secuencia de reducciones que modifica las estructuras de datos que representan el sistema para reflejar las sucesivas reducciones. O sea, se van creando nuevos estados hipotéticos del sistema. Puesto que el estado real del sistema no debe cambiar, habrá que realizar las modificaciones sobre una «copia de trabajo» del estado del sistema.

Activación del algoritmo de detección

Una vez seleccionado un algoritmo de detección, es necesario decidir cuándo se ejecuta el mismo. Evidentemente, lo ideal sería poder detectar un interbloqueo justo cuando se produce, para poder tratarlo inmediatamente. Téngase en cuenta que mientras no se arregle el problema, los procesos y recursos involucrados están bloqueados pudiéndose, además, incorporar progresivamente más procesos al interbloqueo. Sin embargo, no hay que olvidar que el algoritmo tiene un coste de ejecución apreciable y que, por tanto, esta supervisión continua puede ser inabordable.

Volviendo al símil del mantenimiento de los equipos de una empresa, lo idóneo sería poder tener una supervisión continua del funcionamiento de todos los equipos p a poder detectar inmediatamente el fallo de alguno. Sin embargo, esta estrategia puede requerir más recursos humanos de los que dispone la empresa. Ante esta situación, una solución alternativa sería una comprobación periódica del buen funcionamiento de los sistemas. El periodo de comprobación debería fijarse teniendo en cuenta las estadísticas sobre la frecuencia de fallos de cada componente. Además de esta estrategia periódica, también podría a activarse la comprobación de un equipo cuando se detecta algún síntoma en el sistema global que haga pensar que algún componente ha fallado.

En el tratamiento de los interbloqueos basado en su detección se presentan alternativas similares:

- Se puede realizar una **supervisión continua** del estado del sistema con respecto a la asignación de recursos para comprobar que está libre de interbloqueos. Observe que un interbloqueo sólo puede aparecer cuando no puede satisfacerse una petición de un proceso. Por tanto, sólo sería necesario ejecutar el algoritmo en ese momento. Hay que tener en cuenta también que el algoritmo de detección queda simplificado en esta situación puesto que, una vez que se detecte que el proceso que realizo la petición que activó el algoritmo no está involucrado en un interbloqueo, se puede detener la búsqueda. Por tanto, en cuanto aparezca dicho proceso en una secuencia de reducción, el sistema está libre de interbloqueos. La viabilidad de esta alternativa dependerá de con qué frecuencia se active el algoritmo y del tiempo que consuma su ejecución, que dependerá del numero de procesos y recursos existentes en el sistema.
- Se puede realizar una **suspensión periódica**. Ésta sería la opción adecuada cuando, dadas las características del sistema, la repercusión sobre el rendimiento del sistema de la ejecución del algoritmo por cada petición insatisfecha fu ra intolerable. El valor del período debe reflejar un compromiso entre dos factores: debe ser suficientemente alto para que el tiempo de ejecución del algoritmo no afecte apreciablemente al rendimiento del sistema, pero suficientemente bajo para que sea aceptable el tiempo que pasa entre que se produce un interbloqueo y su detección. Este valor dependerá de las características del sistema y de las peticiones que realizan los procesos activos en un momento dado, siendo, por tanto, difícil de determinar. Observe que el algoritmo podría también activarse cuando se detecta algún síntoma que pudiera indicar un posible interbloqueo. Por ejemplo, cuando el grado de utilización del procesador baja de un determinado

umbral.

6.6.2. Recuperación del interbloqueo

Una vez detectado un interbloqueo es necesario tomar una serie de medidas para eliminarlo, esto es recuperar al sistema del mismo. Para ello se deberían seleccionar uno o más de lo: procesos implica dos y quitarles algunos de los recursos que tienen asignados de forma que se rompa el interbloqueo Habría que hacer que los procesos elegidos «retrocedan en el tiempo su ejecución», al menos hasta justo antes de que solicitasen dichos recursos. Este «viaje hacia atrás en el tiempo» es complicado puesto que implica restaurar el estado del proceso tal como estaba en ese instante previo. Esta operación solo sería factible, pero no fácil de implementar , en sistemas que tengan algún mecanismo d puntos de recuperación, lo que no es habitual en los sistemas de propósito general. Observe que, su embargo, esta estrategia podría ser adecuada p a el atamiento de interbloqueos en una base de datos con transacciones, gracias al carácter atómico de las mismas. Dada esta dificultad, lo más habitual e realizar «un retroceso total» abortando directamente los procesos elegidos, con la consiguiente pérdida del trabajo realizado por los mismos.

El algoritmo de recuperación, por tanto, tomaría como punto de partida el conjunto de procesos que están implicados en interbloqueos, tal como ha determinado el algoritmo de detección, e iría sucesivamente abortando procesos de este conjunto hasta que los interbloqueos desaparezcan de sistema, Observe que después de abortar un proceso habría que volver a aplicar el algoritmo de detección para ver si siguen existiendo interbloqueos y, en caso afirmativo, qué procesos están implicados en los mismos.

El criterio, a la hora de seleccionar qué procesos del conjunto se abortarán, debería intentar minimizar el coste asociado a la muerte prematura de los mismos, En esta decisión pueden intervenir numerosos factores tales como la prioridad de lo procesos implicados, el número de recursos que tiene asignados cada proceso o el tiempo que lleva ejecutando cada proceso.

Un último aspecto a destacar es que si se realiza una supervisión continua del sistema (o sea, se aplica el algoritmo de detección siempre que se produce una petición que no puede satisfacerse), se podría tomar la decisión de abortar precisamente al proceso que ha realizado la petición que ha generado el interbloqueo. No es un método óptimo, ya que este proceso puede tener gran prioridad o llevar mucho tiempo ejecutando, pero es eficiente ya que elimina la necesidad de aplicar nuevamente el algoritmo de detección.

6.7. PREVENCIÓN DEL INTERBLOQUEO

Con la estrategia de prevención se intenta eliminar el problema de raíz, asegurando que nunca se pueden producir interbloqueos. Dado que, como se analizó previamente, es necesario que se cumpla las cuatro condiciones de Coffman para que se produzca un interbloqueo, bastaría únicamente con asegurar que una de estas condiciones no se puede satisfacer para eliminar los interbloqueos en sistema. A continuación se analiza sucesivamente cada una de estas cuatro condiciones para ver si es posible establecer estrategias que aseguren que no puede cumplirse [Havender, 1968].

6.7.1. Exclusión mutua

Esta condición establece que para que se produzca un interbloqueo, los recursos implicados en mismo deben ser de uso exclusivo. Para asegurar que no se puede satisfacer esta condición, habría que conseguir que todos los recursos del sistema fueran de tipo compartido. Sin embargo, como se comen-

tó previamente, esto no es posible puesto que hay recursos que son intrínsecamente de carácter exclusivo. Por tanto, esta primera condición no permite definir estrategias de prevención.

6.7.2. Retención y espera

Esta condición identifica que para que ocurra un interbloqueo tiene que haber procesos que tengan asignados recursos pero que estén bloqueados esperando por otros recursos. Una primera estrategia de prevención basada en asegurar que no se cumple esta condición consistiría en hacer que cada programa al principio de su ejecución solicitase simultáneamente todos los recursos que va a necesitar. De esta forma se evita el interbloqueo ya que el proceso sólo se bloquea esperando recursos al principio cuando no tiene ninguno asignado. Una vez satisfecha la solicitud, el programa ya no se bloqueara en espera de recursos puesto que dispone de todos los que necesita,

Como ejemplo, supóngase un programa que necesita usar cuatro recursos (A, B, C y D) en distintos intervalos de tiempo a lo largo de su ejecución de acuerdo con el diagrama de la Fig a 6.8.

Siguiendo esta estrategia de prevención, el programa debería solicitar todos los recursos al principio, aunque realmente la mayoría de ellos sólo los necesite en fases posteriores del programa, como se aprecia a continuación:

- t_1: solicita(A,B,C,D).
- (t_1, t_2): sólo utiliza A.
- (t_2, t_3): utiliza A y B.
- t_3: Libera(A).
- (t_3, t_4): sólo utiliza B.
- (t_4, t_5): utiliza B y C.
- t_5: Libera(B).
- (t_5, t_6): sólo utiliza C.

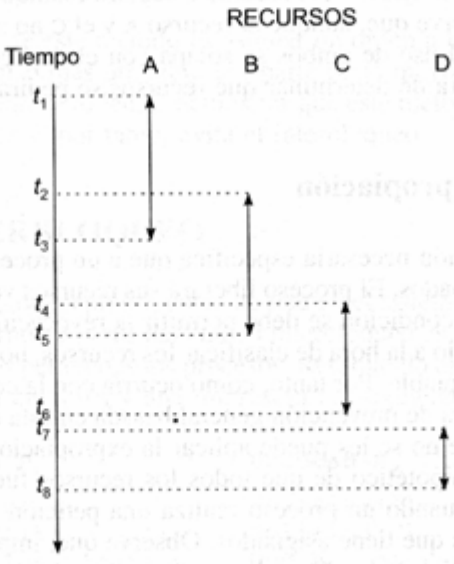


Figura 6.8. *Diagrama de uso de recursos a lo largo del tiempo del programa ejemplo.*

- t_6 :Libera(C).
- (t_7,t_8): sólo utiliza D.
- t_8: Libera(D),

Esta estrategia conlleva una tasa muy baja de utilización de los recursos, Observe que, por ejemplo, el recurso D está reservado desde el instante de tiempo t_1 hasta el t_8 aunque realmente solo se usa en el intervalo de t_7 hasta t_8. Además, esta solución retrasa considerablemente el inicio programa, ya que éste tiene que esperar que todos los recursos estén libres para comenzar cuando en realidad podría comenzar en cuanto estuviese disponible el recurso A.

Una estrategia más refinada consistiría en permitir que un proceso pueda solicitar un recurso solo si no tiene ninguno asignado. Con esta segunda alternativa, un programa sólo se vería obligado a pedir simultáneamente dos recursos si se solapa en el tiempo el uso de los mismos. En el ejemplo anterior, la solicitud del recurso D puede realizarse en el momento que realmente se necesita puesto que dicho recurso no se usa simultáneamente con ningún otro. Sin embargo, el resto de recursos deben seguir pidiéndose al principio.

- t_1: solicita(A ,B ,C).
- (t_1,t_2): sólo utiliza A.
- (t_2,t_3): utiliza Ay B.
- t_3: Libera(A).
- (t_3,t_4): sólo utiliza B,
- (t_4,t_5):utiliza B y C.
- t_5: Libera(B).
- (t_5,t_6): sólo utiliza C.
- t_6**:** Libera(C).
- t_7: solicita(D).
- (t_7,t_8): sólo utiliza D.
- t_8: Libera(D).

Aunque esta nueva política supone una mejora sobre la anterior, sigue presentando los misa problemas. Así, por ejemplo, el recurso C seguirá estando desaprovechado en el intervalo de tiempo entre t_1 y t_4. Observe que, aunque el recurso A y el e no se usan de forma simultanea, se piden vez puesto que el uso de ambos se solapa con el del recurso B. Se produce, por tanto, un cie transitivo a la hora de determinar que recursos se pedirán juntos.

6.7.3. Sin expropiación

La tercera condición necesaria especifica que a un proceso no se le pueden expropiar los recursos que ya tiene asignados. El proceso liberará sus recursos voluntariamente cuando ya no los necesite. Para romper esta condición se debe permitir la revocación involuntaria de recursos. Sin embargo como ya se discutió a la hora de clasificar los recursos, no todos los recursos tienen intrínsecamente un carácter expropiable. Por tanto, como ocurría con la condición de exclusión mutua, no se puede definir una política de prevención general basada en esta condición ya que existen muchos tipos recursos a los que no se les puede aplicar la expropiación.

En el caso hipotético de que todos los recursos fueran expropiables, una posible estrategia consiste en que cuando un proceso realiza una petición que no puede satisfacerse, se le revocan todos los recursos que tiene asignados. Observe que, implícitamente, este tratamiento es el se que aplica al uso del procesador. Cuando un proceso en ejecución se bloquea esperando un recurso, se le expropia el procesador hasta que el recurso esté disponible.

Otra alternativa más «agresiva» es que, ante una petición por parte de un proceso de un recurso que está asignado a otro, se le quite el recurso a su actual poseedor y se le asigne al solicitante. El reemplazo de una página en un sistema con memoria virtual se puede catalogar como una aplicación de esta estrategia, ya que el proceso que causa el fallo de página le está «robando» el marco a otro proceso,

Por último, es importante recordar que la expropiación de recursos conlleva algún tipo de salvaguarda de información de estado del recurso y una posterior restauración de dicha información. Hay que tener en cuenta que estas operaciones tienen un coste que puede afectar al rendimiento del sistema.

6.7.4. Espera circular

La última condición plantea la necesidad de que exista una lista circular de dependencias entre procesos para que exista el interbloqueo. De manera intuitiva se ha podido apreciar en los ejemplos planteados hasta ahora que a esta situación se llega debido a que los procesos piden los recursos en diferente orden.

Basándose en esa idea, una estrategia de prevención es el **método de las peticiones ordenadas**, Esta estrategia requiere establecer un orden total de los recursos presentes en el sistema y fijar la restricción de que un proceso debe pedir los recursos que necesita en orden creciente. Observe que esta restricción hace que un proceso tenga que pedir algunos recursos de forma anticipada.

En el caso del ejemplo de la Figura 6.8, si el orden fijado p a los recursos es A < B <C <D, el programa podría solicitar los recursos justo en el momento que los necesita (A en t_1, B en t_2 ,C en t_4 y D en t_7). Sin embargo, si el orden establecido es el inverso (D < C <*B* <A), el programa debería solicitar en el instante t_1 sucesivamente los recursos D, C, B y A, lo que causaría una infrautilización de los mismos. Observe que sólo sería necesario pedir en orden aquellos recursos que se usan de forma solapada (directamente o debido a un cierre transitivo debido a un tercer recurso). Esto es, un proceso sólo podrá solicitar recursos cuyo orden sea mayor que el de los que tiene actualmente asignados. Así, en el ejemplo, la solicitud del recurso D puede realizarse en el momento en que realmente se necesita *(t_7)*.

Un aspecto fundamental de este método es asignar un orden a los recursos de manera que se corresponda con el orden de uso más probable por parte de la mayoría de los programas.

En el Ejercicio 6,16 se plantea al lector demostrar que este método asegura que no se produce la condición de espera circular y, por tanto, evita el interbloqueo.

6.8. PREDICCIÓN DEL INTERBLOQUEO

Como se comentó al principio del capitulo, antes de que en el sistema aparezca un interbloqueo se produce un «punto de no retorno» a partir del cual el interbloqueo es inevitable, con independencia del orden en el que realicen sus peticiones los procesos. Retomando el primer ejemplo planteado, en el que dos procesos usan una cinta (e) y una impresora *(i)* siguiendo la siguiente estructura:

Proceso P_1	**Proceso P_2**
Solicita(C)	Solicita(I)
Solicita(I)	Solicita(C)
Uso de los recursos	Uso de los recursos
Libera(I)	Libera(C)
Libera(C)	Libera(I)

Si se permite que cada proceso obtenga su primer recurso solicitado, se habrá atravesado el umbral que conduce inevitablemente al interbloqueo. Observe que en ese instante ninguno de los dos procesos está bloqueado, pero, a partir de entonces, sea cual sea la secuencia de ejecución que se produzca, el interbloqueo es ineludible. ¿Cómo se podría detectar cuando el sistema se acerca a este punto de no retorno para evitar entrar en el mismo?. La solución es «muy sencilla»: sólo es necesario conocer el futuro. Si se conociera a *priori* qué recursos van a solicitar los procesos durante su ejecución, se podría controlar la asignación de recursos a los procesos de manera que se evite el ínterbloqueo. En el ejemplo, si después de asignarle la cinta al primer proceso, se produce la solicitud de la impresora por parte del segundo, no se satisfará esta petición bloqueando al segundo proceso aunque realmente el recurso está libre.

Aunque se trata de un ejemplo muy sencillo, ha permitido apreciar cuáles son las claves de los algoritmos de predicción de interbloqueos: un conocimiento a *priori* de las necesidades de los procesos y el no conceder las peticiones que pueden conducir hacia el interbloqueo, aunque los recursos solicitados estén disponibles.

Los algoritmos de predicción se basarán, por tanto, en evitar que el sistema cruce el punto de no retomo que conduce al interbloqueo. Para ello se necesitará conocer a *priori* las necesidades máximas de recursos que tiene cada programa. A partir de esta información, se deberá determinar si el estado del sistema en cada momento es seguro.

6.8.1. Concepto de estado seguro

Se considera que un determinado estado es seguro si, suponiendo que todos los procesos solicitan en ese momento sus necesidades máximas, existe al menos un orden secuencial de ejecución de los procesos tal que cada proceso pueda obtener sus necesidades máximas. Así, para que un estado sea seguro tiene que haber, en primer lugar, un proceso cuyas necesidades máximas puedan satisfacerse. Cuando, hipotéticamente, este proceso terminase de ejecutar devolvería todos sus recursos (o sea, sus necesidades máximas). Los recursos disponibles en ese instante podrían permitir que se satisficieran las necesidades máximas de al menos un proceso que terminaría liberando sus recursos, lo que a su vez podría hacer que otro proceso obtuviera sus necesidades máximas. Repitiendo este proceso, se genera u a secuencia de ejecución de procesos tal que cada uno de ello, puede obtener sus necesidades máximas. Si esta secuencia incluye todos los procesos del sistema, el estado es seguro. En caso contrario es inseguro.

El lector habrá podido apreciar que este proceso de análisis «mirando hacia el futuro» es similar al usado para la reducción de un sistema en el algoritmo de detección de interbloqueos. La única diferencia es que en el algoritmo de reducción se tienen en cuenta sólo las peticiones actuales de los procesos mientras que en este algoritmo se consideran también como peticiones las necesidades máximas de cada proceso. Esta similitud permite especificar una segunda definición del estado seguro:

Un estado es seguro si el estado de asignación de recursos que resulta al considerar que todos los procesos realizan en ese instante todas sus posibles peticiones está libre de interbloqueos.

Gracias a esta nueva definición, se puede especificar directamente un algoritmo para determinar si un estado es seguro: aplicar un algoritmo de detección de interbloqueos al estado resultante de considerar que todos los procesos solicitan sus necesidades básicas.

Retomando el ejemplo anterior, el estado al que se llega si se permite que cada proceso obtenga el primer recurso solicitado (el primer proceso la cinta y el segundo la impresora) es inseguro

puesto que en ese momento ninguno de los dos procesos puede satisfacer sus necesidades máximas (el primer proceso no podría obtener la impresora, ni el segundo la cinta).

Es importante resaltar que este tipo de algoritmos tienen un carácter «conservador» debido a que los datos que se poseen a *priori* sobre el uso de recursos de cada proceso no incluye información sobre la utilización real de los mismos. Esto puede hacer que se consideren como inseguros estados que realmente nunca pueden llevar a un interbloqueo. Así, por ejemplo, considérese el siguiente ejemplo que tiene una estructura similar al anterior:

Proceso P_1	**Proceso P_2**
Solicita(C)	Solicita(I)
Uso del recurso C	Solicita(C)
Libera(C)	Uso de los recursos
Solicita(I)	Libera(C)
Uso del recurso I	Libera(I)
Libera(I)	

En este caso no puede haber nunca interbloqueo, pero, sin embargo, la información sobre el uso máximo de recursos por cada proceso es la misma que en el ejemplo anterior. Por tanto, la situación que corresponde con que cada proceso haya obtenido su primer recurso se considerará como un estado inseguro.

Se puede, por tanto, afirmar que una condición necesaria, pero no suficiente, para que un sistema evolucione hacia un interbloqueo es que su estado actual sea inseguro. Los algoritmos de predicción se basan en evitar que el estado del sistema se convierta en inseguro eliminando de esta forma la posibilidad del interbloqueo.

6.8.2. Algoritmos de predicción

Una vez identificado el concepto de estado seguro es relativamente directo establecer una estrategia de predicción. Cada vez que un proceso realice una solicitud de recursos que estén disponibles, se calcula provisionalmente el nuevo estado del sistema resultante de esa petición y se aplica el algoritmo para determinar si el nuevo estado es seguro. Si lo es, se asignan los recursos solicitados haciendo que el estado provisional se convierte en permanente. En caso de que no lo sea, se bloquea al proceso sin asignarle los recursos quedando, por tanto, el sistema en el estado previo.

Dado que el sistema esta inicialmente en estado seguro, puesto que ningún recurso está asignado, este algoritmo asegura que el sistema siempre se encuentra en un estado seguro eliminando, por tanto, el interbloqueo.

A continuación se plantean algoritmos de predicción para los dos tipos de representación considerados: el grafo de recurso y la representación matricial. Observe que no se trata de algoritmos nuevos ya que, como se ha explicado previamente, se aplican los algoritmos de detección de interbloqueos ya presentados. Por ello, la exposición de los mismos no será exhaustiva.

Algoritmo para una representación mediante un grafo de recursos

Además de las aristas de asignación y las de solicitud, sería necesario un nuevo tipo de arista que refleje las necesidades máximas de cada proceso:

- Una *arista de necesidad* entre un proceso P_i y un recurso R_1 indica que el proceso puede solicitar durante su ejecución una unidad de dicho recurso.

La evolución de este nuevo tipo de aristas sería la siguiente:

- Inicialmente, en la creación de un proceso P_1 se establecerían aristas de necesidad desde el proceso a los recursos correspondientes de manera que queden reflejadas las necesidades máximas requeridas por el proceso.
- En una solicitud, las aristas de necesidad correspondientes se convertirían en aristas d solicitud si los recursos correspondientes no están disponibles, o de asignación, en caso contrario.
- Cuando se produce una liberación de recursos, las aristas de asignación correspondientes se transforman a su estado inicial, o sea, en aristas de necesidad.

Dado este comportamiento de las aristas de necesidad, se puede apreciar que el número total de aristas, sean del tipo que sean, que hay en cualquier momento entre un proceso y un recurso debe corresponder con las necesidades máximas del proceso con respecto a ese recurso.

Como se comento previamente, el algoritmo para determinar si un estado es seguro consiste directamente en comprobar que no hay interbloqueos en dicho estado pero teniendo en cuenta las peticiones máximas, o sea, tanto las aristas de solicitud como las de necesidad. Para ello se pueden usar los algoritmos de detección de interbloqueos para la representación mediante un grafo ya estudiados previamente.

La estrategia de predicción consiste, por tanto, en que cuando un proceso realiza una solicitud de recursos que están disponibles, se calcula un nuevo estado provisional transformando las aristas de necesidad correspondientes en aristas de asignación y se aplica el algoritmo para determinar si el nuevo estado es seguro. Si lo es, se asignan los recursos solicitados haciendo que el estado provisional se convierta en permanente. En caso contrario se bloquea al proceso sin asignarle los recursos restaurando, por tanto, el sistema al estado previo.

A continuación se va a aplicar esta estrategia de predicción al ejemplo de los dos procesos que usan la cinta y la impresora. Observe que, dado que se trata de un sistema con una sola unidad de cada tipo de recurso, bastaría con comprobar que no hay un ciclo en el grafo para asegurar que no hay un interbloqueo.

En la Figura 6.9 se muestra el estado inicial del sistema justo cuando se acaban de crear los dos procesos. Observe que en el mismo sólo hay aristas de necesidad representando las necesidades máximas puesto que todavía no se ha realizado ninguna petición. Evidentemente, este estado inicial es seguro como lo demuestra la ausencia de un ciclo en el grafo que representa al mismo.

Supóngase que el primer proceso realiza su solicitud de la cinta. Puesto que está disponible, e construirá de forma provisional el estado resultante de la asignación del recurso, lo que queda reflejado en la Figura 6.10. En este instante habría que aplicar el algoritmo de detección teniendo en

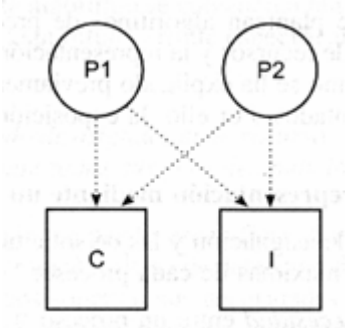


Figura 6.9. *Estado inicial del sistema.*

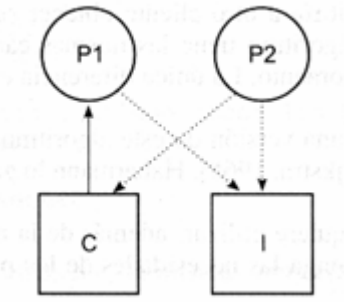


Figura 6.10. *Estado del sistema resultante de aceptar la primera petición.*

cuenta las aristas de necesidad. Como se aprecia en la figura, no hay ciclos, por lo que el estado es seguro y puede hacerse permanente.

Si a continuación el segundo proceso solicita la impresora, al estar disponible, se genera el estado provisional representado en la Figura 6.11. El grafo resultante de transformar la arista de necesidad en una arista de asignación presenta un ciclo, por lo que hay un interbloqueo y el estado es inseguro. Por tanto, no se asignaría el recurso y se bloquearía al proceso restaurando el estado previo (el que corresponde con la Figura 6.10).

Observe que, como se comentó previamente, esta misma situación se produciría también en el ejemplo modificado de los procesos donde no podría haber interbloqueo.

Algoritmo del banquero

El algoritmo más conocido de predicción para una representación matricial es el del banquero. Su nombre proviene de que se utiliza como modelo un banquero que presta dinero a sus clientes. Cada cliente (proceso) tiene asignado un determinado crédito máximo (necesidad máxima del proceso). El banquero tiene disponible una determinada cantidad de dinero en caja (numero de unidades del recurso disponible). Los clientes irán pidiendo y devolviendo dinero (o sea, solicitando y liberando unidades del recurso) de acuerdo a sus necesidades. El algoritmo del banquero controla las peticiones de los clientes de manera que el sistema este siempre en un estado seguro. O sea, se asegura que si en un determinado momento todos los clientes solicitaran su crédito máximo, al menos uno de ellos satisfaría esta petición pudiendo posteriormente terminar su labor y devolver su dinero

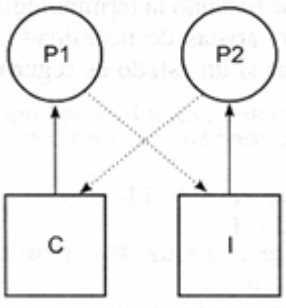


Figura 6.11. *Estado del sistema resultante de aceptar la segunda petición.*

prestado, lo que permitiría a otro cliente obtener sus necesidades y así sucesivamente. Como puede apreciar, este algoritmo tiene las mismas características que los algoritmos de predicción presentados hasta el momento. La única diferencia es que utiliza una representación matricial modelar el sistema.

Dijkstra propuso una versión de este algoritmo aplicable sólo a un único tipo de recurso múltiples unidades [Dijkstra, 1965]. Habermann lo generalizó para múltiples recursos [Habermann 1969].

Este algoritmo requiere utilizar, además de la matriz de solicitud S y la de asignación A, una nueva matriz que contenga las necesidades de los procesos:

- **Matriz de necesidad**, N, de dimensión *p x r*. Siendo *p* el numero de procesos existen el número de recursos diferentes que hay en el sistema, la componente N [i,j] de la matriz especifica cuántas unidades del recurso *j* puede necesitar el proceso *i*. Este valor se corresponde con la diferencia entre las necesidades máximas del proceso para dicho recurso número de unidades actualmente asignadas. Observe que, por tanto, en esta matriz r quedan reflejadas las posibles peticione futuras de cada proceso solicitando recursos hasta la cantidad máxima, sino también las peticiones actuales que no han podido satisfacerse. Inicialmente, esta matriz contiene las necesidades máximas de cada proceso.

El procesamiento de las peticiones de solicitud y liberación de recursos, además de afectar al vector de recursos disponibles y a las matrices de asignación y solicitud como se mostró Sección 6.3.2, modificará también la matriz de necesidad de la siguiente forma:

- Cuando se satisface una solicitud, se restarán las unidades pedida: de cada recurso matriz de necesidad y se sumaran a la matriz de asignación. Observe que una petición puede satisfacerse por el momento no altera esta matriz, sólo modifica la de solicitud mando las unidades correspondientes a la misma. En el momento que se pueda alterará la matriz de necesidad.
- Cuando se produce una liberación de recursos, se sustraen de la matriz de asignación unidades liberadas y se añaden a la matriz de necesidad.

Como el algoritmo de predicción basado en un grafo, el algoritmo del banquero determinar un estado es seguro comprobando que no hay interbloqueos en dicho estado pero ten cuenta las peticiones máximas, en este caso las representadas por la matriz de necesidad. Por el algoritmo es igual que el de detección de interbloqueos para la representación matricial en la Sección 6.6.1, pero usando la matriz de necesidad en vez de la solicitud a la hora de n si se puede reducir el estado del sistema por un determinado proceso. Es interesante res en este algoritmo no se usa la matriz de solicitud para determinar si el estado es seguro ya matriz de necesidad refleja tanto las peticiones futuras del proceso como las peticiones actuales no han podido satisfacerse (usando la terminología de la representación mediante un grafo de recursos, la matriz incluye las aristas de necesidad y las de reserva). A continuación, se muestra el algoritmo para determinar si un estado es seguro:

```
S = Ø;        /* secuencia de reducción, Inicialmente vacía */
Repetir {
      Buscar Pi  tal que N[i];<= D;
      Si Encontrado {
           Reducir grafo por Pi: D = D + A[i]
           Añadir Pi a S;
           Continuar = cierto;
}
```

```
else
    Continuar = falso;
}                Mientras (Continuar)
Si (S = =P)
    /* si la secuencia contiene todos los procesos del sistema (P) */
    El estado es seguro
Si no
    El estado no es seguro
```

Este algoritmo tendrá, evidentemente, la misma complejidad *($O(p^2r)$)* que la versión que se usaba para detectar interbloqueos.

Una vez especificado el algoritmo que determina si el estado es seguro, la definición de la estrategia de predicción es casi directa. Cuando un proceso realiza una solicitud de recursos que están disponibles, se calcula un nuevo estado provisional transformando las matrices de necesidad y de asignación de acuerdo a la petición realizada. Sobre este estado provisional se aplica el algoritmo para determinar si es seguro. Silo es, se asignan los recursos solicitados haciendo que el estado provisional se convierta en permanente. En caso contrario se bloquea al proceso sin asignarle los recursos restaurando, por tanto, el sistema al estado previo.

A continuación, se va a aplicar esta estrategia de predicción a un ejemplo. Considérese un sistema con tres tipo de recursos (R_1, R_2 y R_3) y tres procesos (P_1, P_2 y P_3). Supóngase que se está usando el algoritmo del banquero y que el estado actual del sistema, que se asume como seguro, es el siguiente:

$$A=\begin{pmatrix}1&1&0\\0&1&2\\1&0&0\end{pmatrix}\quad N=\begin{pmatrix}3&0&2\\2&2&0\\1&1&2\end{pmatrix}\quad D=[2\ 1\ 2]$$

Observe que no se ha especificado la matriz de solicitud, puesto que no es relevante para el algoritmo. Observando con detalle los datos del estado actual, se puede obtener información adicional sobre el sistema. Por ejemplo, se puede apreciar que las necesidades máximas de P_1 con respecto al recurso R_1 son cuatro unidades (N [1, 1] +A [1, 1]), lo que coincide con el número total de unidades de dicho recurso que existen en el sistema (A [1, 1] +A [2, 1] +A [3, 1] +D [1]).

Supóngase que, estando el sistema en ese estado, llega una petición de P_3 solicitando una unidad de R_3.En primer lugar, dado que el recurso implicado en la petición está disponible, habría que calcular el estado provisional resultante de satisfacer esta solicitud:

$$A=\begin{pmatrix}1&1&0\\0&1&2\\1&0&1\end{pmatrix}\quad N=\begin{pmatrix}3&0&2\\2&2&0\\1&1&1\end{pmatrix}\quad D=[2\ 1\ 1]$$

A continuación, habría que comprobar si el nuevo estado es seguro. El resultado de aplicar el algoritmo es el siguiente:

1. Estado inicial: S = 0
2. Se puede reducir por P_3 ya que N[3] <= D ([1 1 l]<=[2 1 1]), dando como resultado:

$$D = D + A[3] = [2\ 1\ 1] + [1\ 0\ 1] = [3\ 1\ 2]$$

3. Se añade el proceso a la secuencia de reducción: S= { P_3 }, y se pasa a la siguiente iteración
4. Reducción por P_1 dado que N[l] <= D ([3 0 2] <= D[3 12]), que da como resultado:

$$D = D + A[1] = [3\ 1\ 2] + [1\ 1\ 0] = [4\ 2\ 2]$$

5. Se añade el proceso a la secuencia de reducción: S = { P_3,P_1} y se pasa a la siguiente iteración.
6. Reducción por P_2 dado que N[2] <=D([2 2 0]<= [4 2 2]),queda como resultado:

$$D = D + A[2] = [4\ 2\ 2] + [0\ 1\ 2] = [4\ 3\ 4]$$

7. Se añade el proceso a la secuencia de reducción: S = { P_3 ,P_1 ,P_2} , y se termina el bucle.
8. Como S incluye a todos los procesos, el estado del sistema es seguro.

Por tanto, se aceptaría la petición consolidando el estado provisional como nuevo estado de sistema.

Supóngase que, a continuación, el proceso P_2 solicita una unidad de R_1. Puesto que el recurso implicado está disponible, habría que, en primer lugar, calcular el estado provisional resultante:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 2 \\ 1 & 0 & 1 \end{pmatrix} \quad N = \begin{pmatrix} 3 & 0 & 2 \\ 2 & 2 & 0 \\ 1 & 1 & 1 \end{pmatrix} \quad D = [1\ 1\ 1]$$

Al aplicar al algoritmo que determina si el estado es seguro, el resultado sería el siguiente

1. Estado inicial: s =0.
2. Se puede reducir por P_3 ya que N[3] <=D([11 1]<=[1 11]),dando como resultado:

$$D = D + A[3] = [111] + [101] = [212]$$

3. Se añade el proceso a la secuencia de reducción: S = { P_3}, y se pasa a la siguiente iteración
4. No hay ningún tal que N [i] <= D. Termina la ejecución del bucle.
5. Como S no incluye a todos los procesos, el estado del sistema no es seguro.

Por tanto, no se satisfaría la petición bloqueando al proceso y restaurando el estado anterior cl sistema.

Sea cual sea el algoritmo de predicción utilizado, es conveniente resumir algunas de las deficiencias que presentan este tipo de algoritmos:

- Conocimiento de las necesidades máximas de los procesos. Esta información no siempre puede conocerse a *priori* y, en caso de poderse, siempre deberá corresponder con el «pe caso posible» en cuanto al uso de recursos que puede tener un programa.
- Las necesidades máximas no pueden expresar el uso exacto de los recursos durante la ejecución de los programas. Por ello, como se vio en uno de los ejemplos previos, se puede detectar como inseguros estados que realmente no pueden conducir a un interbloqueo ya que en ellos no hay un uso solapado de recursos.
- Infrautilización de los recursos. Estos algoritmos pueden denegar la concesión de un recursos aunque este disponible produciéndose el consiguiente desaprovechamiento del mismo.

6.9. TRATAMIENTO DEL INTERBLOQUEO EN LOS SISTEMAS OPERATIVOS

La exposición realizada a lo largo de este capitulo se ha centrado en los aspectos teóricos del tema, Sin embargo, en esta última sección se va a analizar la aplicación de las técnicas presentadas hasta ahora en el diseño de los sistemas operativos reales. Hay que decir a *priori* que, aunque resulte sorprendente y un poco decepcionante, la mayoría de los sistemas operativos actuales ignoran en gran parte este problema o, al menos, no lo solucionan de forma general.

Un primer aspecto que hay que tener en cuenta, cuando se considera el problema de los interbloqueos en un sistema real, es la diferencia que existe entre los recursos internos del sistema operativo y los recursos disponibles a las aplicaciones de usuario [Howard, 1973].

- **Recursos internos del sistema.** Son recursos que el sistema operativo necesita para llevar a cabo su labor. La reserva y liberación de este tipo de recursos se realiza desde el código del sistema operativo y, generalmente, el uso del recurso está comprendido dentro de la ejecución de una única llamada al sistema. Por tanto, el bloqueo del proceso se produce cuando éste está ejecutando dentro del sistema operativo. Algunos ejemplos de este tipo de recursos son un semáforo interno del sistema operativo que permite controlar el acceso concurrente a la tabla de procesos, un descriptor interno de un chivo que una llamada al sistema necesita usar en modo exclusivo o un buffer de la cache de bloques del sistema de chivos,
- **Recursos de usuario.** Se corresponden con recursos que usan las aplicaciones de usuario para realizar su trabajo. La reserva y liberación de este tipo de recursos se hace, por tanto, desde el código de las aplicaciones. Se pueden considerar como ejemplos de este tipo de recursos un semáforo utilizado por un conjunto de procesos de usuario cooperantes para sincronizar su ejecución o un dispositivo dedicado como una unidad de cinta.

La repercusión de un interbloqueo va a ser muy diferente dependiendo de esta clasificación. En cuanto a los recursos internos, si se produce un interbloqueo entre un conjunto de procesos que están ejecutando llamadas al sistema (o sea, usando recursos internos), tanto los procesos como los recursos internos afectados quedaran indefinidamente «fuera de servicio». Una llamada al sistema que implicase el uso de alguno de estos recursos causaría que el proceso que la invoca se viese también implicado en el interbloqueo. Para apreciar las consecuencias fatales de esta situación, solo es necesario imaginar lo que sucedería si uno de los recursos afectados fuera, por ejemplo, un semáforo que controla el acceso a la tabla de proceso o el descriptor interno del directorio raíz del sistema de chivos: prácticamente cualquier llamada al sistema causaría el bloqueo indefinido del proceso que la invoca provocando el colapso total del sistema. Es importante hacer notar que la mayoría de los sistemas operativos (entre ellos, las distintas versiones de UNIX), p a evitar problemas de coherencia en sus estructuras de datos internas, no permiten abogar la ejecución de un proceso que está bloqueado esperando usar un recurso interno. Por tanto, no se podría resolver la situación abortando algunos de los procesos implicados. La única «solución» seria volver a arrancar la máquina.

En el caso de los recursos de usuario, como sucede con los internos, la ocurrencia de un interbloqueo inutilizaría los recursos de usuario implicados haciendo que cualquier proceso que los intente usar se vea también involucrado en el interbloqueo. El da o causado es, evidentemente, importante. Así, por ejemplo, una unidad de cinta podría quedar permanentemente inutilizada. Sin embargo, la repercusión no es tan grave. Por un lado, solo se verían afectados los procesos que usan el recurso explícitamente. Por otro lado, al tratarse de recursos externos, los sistemas operativos permiten, generalmente, abortar la ejecución de algunos de los procesos implicados, no siendo necesario volver a iniciar el sistema. Observe que, de manera intuitiva, los usuarios aplican una política de detección y recuperación de interbloqueos: cuando el usuario considera que su programa

está tardando demasiado tiempo en realizar su labor (detección de posible interbloqueo), ejecución (recuperación del interbloqueo) liberando los recursos que tenía asignados. - mente, esta estrategia «de andar por casa» sólo se puede aplicar a los recursos de usuario.

Además de tener una repercusión diferente, el atamiento que realizan los sistemas operativos de estos dos tipos de recursos es muy distinto. Por lo que se refiere a los recursos internos, gravedad del problema, el sistema operativo debería asegurar que no se producen. Un aspecto que hay que tener en cuenta es que el código del sistema operativo, a diferencia del de las aplicaciones del usuario, es algo relativamente cerrado que prácticamente no se modifica (o, en caso de modificación, la realizan programadores «expertos» siguiendo unas reglas estrictas). Por tanto, es estudiar a *priori* el uso de recursos internos de las distintas llamadas y ajustarlo para prevenir interbloqueos. De hecho, la ocurrencia de un interbloqueo se debería considerar un error de programación del sistema operativo. Dados estos condicionantes, las estrategias más adecuadas para el tratamiento de interbloqueos en el uso de recursos internos son las de prevención: establecer unas restricciones a la hora de pedir los recursos internos para asegurar que nunca se produzca ínterbloqueos.

Hay que resaltar que, a diferencia de lo que ocurre con los recursos de usuario, los internos se usan durante un tiempo limitado y conocido a *priori,* ya que corresponde con la duración de las llamadas al sistema. Por ello, en este caso el problema de la infrautilización de provocado por los algoritmos de prevención tiene un efecto muy limitado. La estrategia decisión más frecuentemente usada es la petición ordenada de recursos. Para ello se establece entre los recursos i ternos del sistema y se fija la restricción de que el código del sistema debe reservar los recursos internos en el orden establecido. Así, por ejemplo, en el UNIX BSD [McKusick, 1996], durante la traducción del nombre de ruta de un archivo, la propia árbol de chivos define el orden en el que se tienen que realizar los cerrojos sobre los É"'componentes de la ruta para prevenir el interbloqueo.

En el caso de los recursos de usuario, prácticamente ningún sistema operativo de general proporciona una solución global al problema de los interbloqueos. Para entender esta situación hay que tener en cuenta varios factores, En primer lugar, a diferencia de lo que ocurre con el sistema operativo, el uso de recursos por parte de las aplicaciones en un sistema propósito general es impredecible. Además, como se analizó previamente, la repercusión de los interbloqueos que afectan a este tipo de recursos es menos grave que en el caso de los in que los usuarios pueden abortar la ejecución de los procesos involucrados. La aplicación técnicas de prevención conduce generalmente a una infrautilización de los recursos ya que de los intervalos de uso de los mismos es normalmente mucho mayor que en el caso de los internos, cuya utilización se limita al ámbito de una única llamada sistema. En cuanto a las estrategias de prevención, además de presentar el mismo problema de infrautilización, son difíciles de implantar debido a que requieren conocer a *priori* el comportamiento de los procesos, lo que, normalmente, no es factible en un sistema operativo de propósito general.

Por lo que se refiere a las políticas de detección y recuperación, la ejecución del detección, ya sea periódicamente o en cada petición insatisfecha, supone un esfuerzo que puede afectar negativamente al rendimiento del sistema. Puesto que el coste de la ejecución del & depende del tamaño del «estado del sistema>> y puede hacerse intolerable si se aplica a recursos del sistema, muchos sistemas operativos no usan este tipo de estrategias de sino que las limitan a un único tipo de recurso. Así, por ejemplo, el sistema 4.4BSD utiliza un algoritmo de detección para tratar los interbloqueos sólo en el uso de cerrojos sobre archivos (curiosamente, no lo hacía en la versión 4.3). La suma de todas estas consideraciones conduce a la falta de un tratamiento global del los interbloqueos en la mayoría operativos de propósito general.

Debido a sus peculiaridades, merece mención aparte el análisis de los recursos con el almacenamiento de los programas de usuario, o sea, la memoria principal y la secundaria.En

el caso de la memoria principal, como se explico en la Sección 6.2.1, al tratarse de un recurso expropiable, cuando un proceso necesita memoria principal (p. ej.: como consecuencia de un fallo de página), se le puede expropiar a otro proceso parte de la memoria que tiene asignada (un marco de página) salvando su contenido en memoria secundaria, Se trata, por tanto, de una estrategia de prevención basada en la expropiación.
Con respecto a la gestión de la memoria secundaria, en sistemas sin preasignación de espacio pueden d se situaciones de interbloqueo. Supóngase que en un determinado instante no hay espacio disponible en el dispositivo de almacenamiento secundario y todas las páginas en memoria están modificadas y no tienen todavía reservado espacio en el mismo. Si en ese momento ocurre un fallo de página que requiere leerla de la memoria secundaria, se producirá una situación de interbloqueo ya que no hay espacio en el dispositivo para salvar una de las páginas residentes antes de traer la página implicada en el fallo. El sistema operativo debería abortar la ejecución de alguno de los procesos implicados para liberar espacio en memoria principal. La alternativa que evita este tipo de situaciones es la preasignación de memoria secundaria. Observe que, desde el punto de vista del atamiento de lo: interbloqueos, la preasignación se puede considerar una técnica de prevención basada en solicitar todos los recursos de forma conjunta y, como ocurre con este tipo de estrategias, tiene como desventaja la infrautilización de los recursos: el espacio reservado en memoria secundaria para una determinada página no se utilizará hasta que esa página sea expulsada por primera vez, situación que, además, puede no ocurrir.
La situación descrita no solo afecta a los sistema con memoria virtual. En lo. sistemas sin memoria virtual basados en el intercambio *(swapping)* se produce un problema simil [Bach, 1986] pero que, evidentemente, afecta a todo el mapa de un proceso en vez de a una sola página. Nuevamente la preasignación de espacio elimina la posibilidad de que se produzcan interbloqueos.
Como conclusión de las ideas expuestas en esta sección, se puede expresar que el desencuentro aparente entre los estudios teóricos sobre el tratamiento de los interbloqueos y su aplicación a los sistemas operativos reales tiene un final feliz: el tratamiento de los interbloqueos es un aspecto muy importante en el diseño de un sistema operativo, estando presente, aunque en ocasiones de forma implícita, en casi todos los ámbitos del mismo.

6.10. PUNTOS A RECORDAR

- Se denomina interbloqueo (en inglés, *deadlock*) a la situación que se produce cuando las necesidades de unos procesos entran en conflicto entre sí causando que éstos se bloqueen indefinidamente.
- El problema de los interbloqueos no se circunscribe únicamente al mundo de la informática, sino que aparece en muchos otros ámbitos como, por ejemplo, en el tráfico de vehículos o en la comunicación entre entidades.
- Un escenario donde pueden aparecer interbloqueos se caracteriza por la existencia de un conjunto de entidades activas que utilizan un conjunto de recursos para llevar a cabo su labor.
- En un sistema informático las entidades activas se corresponden con los procesos o *threads*.
- En un sistema informático existe una gran variedad de recursos. Recursos físicos tales como procesadores, memoria o dispositivos. Recursos lógicos tales como archivos, semáforos, *mutex*, cerrojos, mensajes o señales.
- Los recursos presentes en un sistema se pueden clasificar siguiendo varios criterios: recursos reutilizables o consumibles, recursos compartidos o exclusivos, recursos con un único ejemplar o con múltiples y recursos expropiables o no expropiables.
- Un *recurso reutilizable* se caracteriza porque sigue existiendo después de que un proceso lo use quedando disponible para otros procesos. Dentro de esta categoría se engloban todos los recursos físicos y recursos lógicos tales como los archivos o los cerrojos.
- Un *recurso consumible* se caracteriza porque deja de existir una vez que un proceso lo usa. Un proceso genera o produce el recurso y el otro lo utiliza consumiéndolo. Dentro de esta categoría se encuentran básicamente recursos lógicos relacionados con la comunicación y sincronización de procesos.
- No hay una solución general eficiente para tratar el problema de los interbloqueos con ambos tipos de recursos

debido, principalmente, a las características específicas que presentan los recursos consumibles. Las soluciones presentadas en el capítulo se centran en los recursos consumibles de uso exclusivo.

- Los *recursos de tipo compartido* no se ven afectados por los interbloqueos.
- En un sistema pueden existir múltiples instancias o ejemplares de un determinado recurso.
- Un recurso tiene un carácter *expropiable* si permite que se le revoque a un proceso mientras lo está usando, para otorgárselo a otro proceso. Para evitar la pérdida de información, esta expropiación implica salvar de alguna forma el trabajo que llevaba hecho el proceso con el recurso expropiado.
- El procesador y la memoria son ejemplos de recursos expropiables.
- En un sistema se pueden distinguir las siguientes entidades y relaciones: Un conjunto de procesos (o *threads*), un conjunto de recursos reutilizables de uso exclusivo, un conjunto de relaciones entre procesos y recursos que define qué asignaciones de recursos están vigentes en el sistema en un momento dado, y un conjunto de relaciones entre procesos y recursos que define qué solicitudes de recursos están pendientes de satisfacerse en el sistema en un momento dado.
- Una petición de recursos genérica permitirá solicitar simultáneamente varias unidades de diferentes recursos.
- Sólo se concederá una petición de recursos si **todos** los solicitados están disponibles, asignando al proceso dichos recursos.
- Las dos representaciones más habituales del estado de un sistema son el grafo de asignación de recursos y la representación matricial.
- Un conjunto de procesos está en interbloqueo si cada proceso está esperando un recurso que sólo puede liberar (o generar, en el caso de recursos consumibles) otro proceso del conjunto.
- Las condiciones necesarias para que se produzca un interbloqueo son: exclusión mutua, retención y espera, sin expropiación y espera circular. Estas condiciones son necesarias y suficientes si en el sistema sólo hay una unidad de cada recurso.
- Se dice que el estado de un sistema se puede reducir por un proceso si se pueden satisfacer las necesidades del proceso con los recursos disponibles.
- La condición necesaria y suficiente para que un sistema esté libre de interbloqueos es que exista una secuencia de reducciones del estado actual del sistema que incluya a todos los procesos del sistema.
- Los tres tipos de estrategias utilizadas para tratar el interbloqueo son: la detección y recuperación, la prevención y la predicción.
- Los algoritmos de detección se basan en aplicar el concepto de reducción al estado del sistema.
- La activación del algoritmo de detección puede hacerse en cada petición insatisfecha o periódicamente, dependiendo de las características del sistema.
- La recuperación generalmente se basa en abortar algunos de los procesos implicados en el interbloqueo.
- Las estrategias de detección suponen un coste que puede afectar al rendimiento del sistema. Asimismo, las técnicas de recuperación conllevan una pérdida del trabajo realizado hasta ese momento por algunos de los procesos implicados.
- El criterio a la hora de seleccionar qué procesos se abortan debería intentar minimizar el coste asociado a la muerte prematura de los mismos.
- Las técnicas de prevención se basan en asegurar que no se puede satisfacer una de las condiciones necesarias del interbloqueo.
- Las dos estrategias de prevención más frecuentemente usadas son la petición simultánea de recursos y la petición ordenada.
- Las técnicas de prevención suelen implicar una infrautilización de los recursos.
- Antes de que en el sistema aparezca un interbloqueo se produce un «punto de no retorno» a partir del cual el interbloqueo es inevitable. Los algoritmos de predicción se basan en evitar que el sistema cruce el punto de no retorno que conduce al interbloqueo a partir de un conocimiento *a priori* de las necesidades máximas de cada proceso.
- Un estado es seguro si el estado de asignación de recursos que resulta al considerar que todos los procesos realizan en ese instante todas sus posibles peticiones está libre de interbloqueos.
- La estrategia de predicción consiste en que cada vez que un proceso realiza una solicitud de recursos que están disponibles, sólo se le concede si el estado resultante es seguro. En caso de que no lo sea, se bloquea al proceso sin asignarle los recursos.
- El algoritmo del banquero es una estrategia de predicción para una representación matricial del sistema.
- Los algoritmos de predicción presentan las siguientes deficiencias: requieren un conocimiento de las necesidades máximas de los procesos que, además, no reflejan el uso exacto de los recursos durante la ejecución de los mimos, y causan una infrautilización de los recursos.
- La mayoría de los sistemas operativos actuales ignoran en gran parte el problema de los interbloqueos o, al menos, no lo solucionan de forma general.
- En el caso de recursos internos del sistema (usados por el propio sistema operativo), un interbloqueo puede causar el colapso del sistema. Se debe analizar el códi-

go del sistema operativo para asegurar que no se producen. Generalmente se aplican técnicas de prevención.

- Los recursos de usuario (usados por las aplicaciones) son menos críticos puesto que el usuario puede abortar «manualmente» a los procesos afectados. Por ello, la mayoría de los sistemas operativos ignoran los interbloqueos para este tipo de recursos o no les dan una solución general.
- La memoria usada por las aplicaciones es un caso que requiere un tratamiento específico. Para la memoria principal se puede usar la expropiación y para la memoria secundaria la preasignación.
- El diseñador de un sistema operativo tiene que tener continuamente presente el problema del interbloqueo para asegurar que no se produzca durante la ejecución de los distintos módulos del sistema operativo.

6.11. LECTURAS RECOMENDADAS

Como se comentó al principio del capítulo, el problema del interbloqueo ya se identificó en la década de los sesenta y, desde entonces, se han publicado un gran número de estudios sobre el mismo. De hecho, han aparecido artículos que recopilan referencias bibliográficas sobre el tema [Newton, 1979] y [Zoble, 1983].

Dado el carácter «clásico» del tema, cualquier libro general de sistemas operativos incluye un capítulo que trata sobre el mismo (como, por ejemplo, [Stallings, 1998], [Silberschatz, 1999] o [Tanenbaum, 1992]). En cuanto a los aspectos de su aplicación práctica, en los libros dedicados a presentar el diseño interno detallado de un determinado sistema operativo suele haber diseminadas múltiples referencias al tratamiento de los interbloqueos en las distintas partes del mismo. Así ocurre, por ejemplo, en el caso de UNIX (p. ej.: [McKusick, 1996» y [Bach, 1986]).

6.12. EJERCICIOS

6.1. Enumere algunos ejemplos adicionales de interbloqueos que ocurren en entornos ajenos a la informática.

6.2. El problema de la inanición y el de los interbloqueos se tratan en algunos textos de forma conjunta. Analice qué similitudes y diferencias existen entre estos dos problemas. Muestre ejemplos de las siguientes posibilidades:

a) Una situación donde se produzca inanición pero no haya interbloqueo.
b) Una situación donde se produzca inanición e interbloqueo.

6.3. ¿Cómo se arreglaría el error de programación del ejemplo de interbloqueo entre procesos que se comunican mostrado en la Sección 6.2.1?

6.4. Clasifique según los cuatro criterios especificados en la Sección 6.2.1 los siguientes tipos de recursos:

a) Semáforo.
b) Mutex.
c) Memoria.
d) Procesador.
e) Archivo.
f) Cerrojo.
g) Paso de mensajes.
h) Tubería (*pipe*).

6.5. Si en una operación de petición de recursos no están disponibles todos los recursos solicitados, se bloquea el proceso sin reservar ninguno de ellos. Analice qué ocurriría si, ante una petición de múltiples recursos, el sistema fuera asignando los recursos al proceso según éstos fueran quedando disponibles, desbloqueando al proceso cuando todos estén asignados al mismo. Dicho de otra forma, estudie si $S(U_1, \ldots, U_n)$ es equivalente a $S(U_1) + \ldots + S(U_n)$.

6.6. Plantee un ejemplo de interbloqueos entre procesos que se comunican usando tuberías.

6.7. Identifique por qué no es coherente el siguiente estado de asignación de recursos.

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 2 & 0 \end{bmatrix} \qquad S = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$E = [2 \quad 3 \quad 2] \qquad D = [0 \quad 0 \quad 2]$$

6.8. Identifique por qué no es coherente el siguiente estado de asignación de recursos.

$$A = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 0 \end{bmatrix} \qquad S = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$E = [2 \quad 3 \quad 2] \qquad D = [0 \quad 0 \quad 2]$$

6.9. Supóngase un sistema con un único tipo de recurso con `U` unidades disponibles donde se ejecutan `P` procesos que utilizan el mismo número de unidades del recurso. ¿Cuál es el número máximo de unidades que puede solicitar cada proceso de manera que se asegure que no puede haber interbloqueo?

6.10. Supóngase un sistema con un único tipo de recurso con `U` unidades disponibles donde se ejecutan procesos tal que cada uno de ellos puede necesitar `K` unidades del recurso. ¿Cuál es el número máximo de procesos que puede existir de forma que se asegure que no puede haber interbloqueo?

6.11. Supóngase un sistema con un único tipo de recurso que consta de múltiples unidades. En este sistema se ejecutan `P` procesos tal que cada uno de ellos puede necesitar `K` unidades del recurso. ¿Cuántas unidades del recurso deben existir como mínimo para asegurar que no puede haber interbloqueo?

6.12. Sea un sistema con tres procesos y cuatro recursos (`A`, `B`, `C` y `D`). Cada proceso utiliza tres de estos recursos: el primer proceso `A`, `B` y `C`, el segundo `B`, `C` y `D`, y el tercero `A`, `B` y `D`. El primer proceso solicita los recursos en el orden `ABC`. El segundo en el orden `BCD`. Analice cuáles de los posibles órdenes de solicitud de recursos por parte del tercer proceso (o sea, `ABD`, `DBA`,...) pueden generar interbloqueos.

6.13. Supóngase un sistema con dos tipos de recursos con `3` unidades disponibles de cada uno de ellos. En este sistema se ejecutan procesos de forma tal que cada uno de ellos necesita una unidad de cada tipo de recurso. ¿Cuál es el número máximo de procesos que puede existir de forma que se asegure que no puede haber interbloqueo?

6.14. Demuestre que la presencia de un ciclo en un grafo de asignación de recursos es una condición necesaria pero no suficiente para que exista un interbloqueo.

6.15. Demuestre que si no existe un ciclo en un grafo de asignación de recursos el sistema se puede reducir.

6.16. Demuestre que el algoritmo de prevención basado en la ordenación de recursos asegura que no se producen interbloqueos.

6.17. Demuestre formalmente que no se puede producir un interbloqueo entre procesos POSIX que ejecuten la llamada `wait`.

6.18. Demuestre que se puede producir un interbloqueo entre *threads* POSIX que ejecuten la llamada `pthread_join`.

6.19. Demuestre que un estado de interbloqueo es un estado inseguro.

6.20. Razone por qué un sistema inicialmente está en estado seguro.

6.21. Demuestre que no es posible que un sistema evolucione de un estado inseguro a uno seguro.

6.22. Aplique el algoritmo de predicción basado en la representación mediante un grafo al ejemplo representado en la Figura 6.2.

6.23. Aplique el algoritmo de predicción basado en la representación mediante un grafo al ejemplo representado en la Figura 6.3.

6.24. Aplique el algoritmo del banquero al ejemplo representado en la Figura 6.2.

6.25. Aplique el algoritmo del banquero al ejemplo representado en la Figura 6.3.

6.26. Resuelva el ejemplo de predicción correspondiente al sistema representado en las Figuras 6.9, 6.10 y 6.11 usando el algoritmo del banquero.

6.27. Compruebe que el estado de partida del ejemplo del algoritmo del banquero presentado en la Sección 6.9.2 es seguro.

6.28. Resuelva el ejemplo del algoritmo del banquero presentado en la Sección 6.9.2 mediante un algoritmo de predicción basado en una representación mediante un grafo.

6.29. Desarrolle un algoritmo de detección de interbloqueos para el mecanismo de cerrojos correspondientes al servicio `fcntl` de POSIX. Esta llamada permite establecer dos tipos de cerrojos sobre un archivo (o una región del mismo): de lectura (que correspondería con un uso compartido del recurso) o de escritura (que implicaría un uso exclusivo).

7

Entrada/salida

En este capítulo se presentan los conceptos básicos de entrada/salida (FIS), se describe brevemente el hardware de E/S y su visión lógica desde el punto de vista del sistema operati vo, se muestra cómo se organizan los módulos de E/S en el sistema operativo y los servicios de E/S que proporciona éste.

El capítulo tiene tres objetivos básicos: mostrar al lector dichos conceptos desde el punto de vista del sistema, los servicios de E/S que da el sistema operativo y los aspectos de diseño de los sistemas de E/S. Para alcanzar este objetivo el capítulo se estructura en los siguientes grandes apartados:

- Introducción.
- Caracterización de los dispositivos de E/S.
- Arquitectura del sistema de E/S.
- Interfaz de aplicaciones.
- Almacenamiento secundario.
- Almacenamiento terciario.
- El reloj.
- El terminal.
- La red.
- Servicios de E/S.

7.1. INTRODUCCIÓN

El corazón de una computadora lo constituye la UCP. Esta unidad se encarga de procesar los datos y las instrucciones para conseguir el fin deseado por una aplicación. Ahora bien, esta unidad no serviría de nada sin otros dispositivos que almacenaran los datos y que permitieran interactuar con los usuarios y los programadores de las computadoras. Los primeros son básicamente dispositivos de almacenamiento secundario (discos) y terciario (cintas y sistemas de archivo). Los segundos son los denominados dispositivos periféricos de interfaz de usuario, porque generalmente están fuera de la computadora y se conectan a ella mediante cables, y son los teclados, ratones, micrófonos, cámaras y cualquier otro dispositivo de E/S que se le ocurra conectar a una computadora. La Figura 7.1 muestra una arquitectura típica de una computadora personal con sus dispositivos principales de E/S.

Todos estos dispositivos de E/S se pueden agrupar en tres grandes grupos:

- Dispositivos de interfaz de usuario. Se llama así a los dispositivos que permiten la comuni cación entre los usuarios y la computadora. Dentro de este grupo se incluyen todos los dispositivos que sirven para proporcionar interfaz con el usuario, tanto para entrada (ratón, teclado, etc.) como para salida (impresoras, pantalla, etc.). Existen periféricos menos habi tuales, pero más sofisticados, tales como un escáner, lectores de huella digital, lectores de cinta magnética, instrumentos musicales digitales (MIDI), etc.
- Dispositivos de almacenamiento. Se usan para proporcionar almacenamiento no volátil de datos y memoria. Su función primordial es abastecer de datos y almacenamiento a los pro gramas que se ejecutan en la UCP. Según su capacidad y la inmediatez con que se puede acceder a los datos almacenados en estos dispositivos, se pueden dividir en almacenamiento secundario (discos y disquetes) y terciario (cintas).•
- Dispositivos de comunicaciones. Permiten conectar a la computadora con otras computa doras a través de una red. Los dos tipos de dispositivos más importantes de esta clase son los

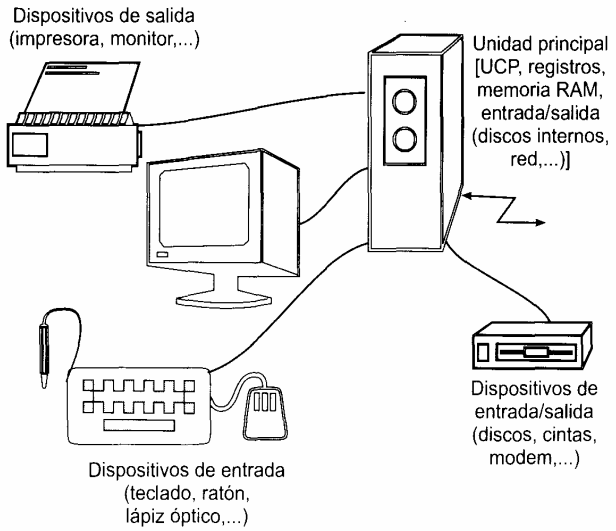


Figura 7.1. *Configuración típica de una computadora personal.*

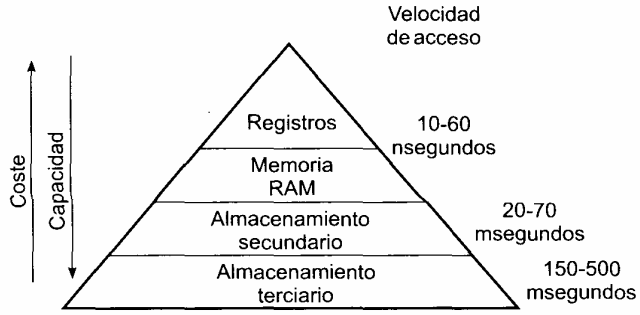


Figura 7.2. *Jerarquía de dispositivos de E/S según su velocidad de acceso.*

módem, para comunicación vía red telefónica, y las tarjetas de interfaz a la red, para conec tar la computadora a una red de área local.

El gran problema de todos estos dispositivos de E/S es que son muy lentos. Piense que mien tras la UCP procesa instrucciones a casi 1 GHz y la memoria RAM tiene un tiempo de acceso de nanosegundos, los dispositivos de E/S más rápidos tienen una velocidad de acceso del orden de milisegundos (Fig. 7.2). Esta diferencia en la velocidad de acceso, y el hecho de que las aplicacio nes son cada vez más interactivas y necesitan más E/S, hace que los sistemas de E/S sean el cuello de botella más importante de los sistemas de computación y que todos los sistemas operativos dediquen un gran esfuerzo a desarrollar y optimizar todos los mecanismos de E/S. Piense, por ejemplo, que el mero hecho de seguir el curso de un ratón supone inspeccionar su posición varias veces por segundo. Igualmente, los dispositivos de comunicaciones interrumpen continuamente el flujo de ejecución de la UCP para comunicar la llegada de paquetes de datos.

El sistema de E/S es la parte del sistema operativo que se ocupa de facilitar el manejo de los dispositivos de EIS ofreciendo una visión lógica simplificada de los mismos que pueda ser usada por otros componentes del sistema operativo (como el sistema de archivos) o incluso por el usuario. Mediante esta visión lógica se ofrece a los usuarios un mecanismo de abstracción que oculta todos los detalles relacionados con los dispositivos físicos, así como del funcionamiento real de los mis mos. El sistema operativo debe controlar el funcionamiento de todos los dispositivos de EIS para alcanzar los siguientes objetivos:

- Facilitar el manejo de los dispositivos de E/S. Para ello debe ofrecer una interfaz entre los dispositivos y el resto del sistema que sea sencilla y fácil de utilizar.
- Optimizar la E/S del sistema, proporcionando mecanismos de incremento de prestaciones donde sea necesario.
- Proporcionar dispositivos virtuales que permitan conectar cualquier tipo de dispositivo físi co sin que sea necesario remodelar el sistema de E/S del sistema operativo.
- Permitir la conexión de dispositivos nuevos de E/S, solventando de forma automática su instalación usando mecanismos del tipo plug&plav.

En este capítulo se caracterizan los componentes que constituyen el hardware de E/S de una computadora, se define la arquitectura del sistema de E/S de una computadora y se estudian algunos de los componentes más importantes del sistema de E/S, ta'es como os sistemas de a to secundario y terciario, los terminales, los relojes, los dispositivos de red, etc. Para terminar se muestran los servicios de E/S más habituales en sistemas operativos, con ejemplos de programación que involucran llamadas al sistema de E/S

Digitalización realizada con propósito académico

7.2. CARACTERIZACIÓN DE LOS DISPOSITIVOS DE E/S

La visión del sistema de E/S puede ser muy distinta dependiendo del nivel de detalle necesario en su estudio. Para los programadores, el sistema de E/S es una caja negra que lee y escribe datos en dispositivos externos a través de una funcionalidad bien definida. Para los fabricantes de dispositi vos, un dispositivo es un instrumento muy complejo que incluye cientos o miles de componentes electrónicos o electromecánicos. Los diseñadores de sistemas operativos se encuentran en un lugar intermedio entre los dos anteriores. Les interesa la funcionalidad del dispositivo, aunque a un nivel de detalle mucho más grande que la funcionalidad que espera el programador de aplicaciones, pero también les interesa conocer la interfaz física de los dispositivos y su comportamiento interno para poder optimizar los métodos de acceso a los mismos.

En esta sección se estudia brevemente cómo se conecta un dispositivo de E/S a una computa dora y se lleva a cabo una caracterización de los dispositivos de E/S según sus métodos y tamaño de acceso, su forma de programación, etc.

7.2.1. Conexión de un dispositivo de E/S a una computadora

La Figura 7.3 muestra el esquema general de conexión de periféricos a una computadora IDe Miguel, 1998]. En el modelo de un periférico se distinguen dos elementos:

- **Periféricos o dispositivos de E/S.** Elementos que se conectan a la unidad central de proceso a través de las unidades de entradalsalida. Son el componente mecánico que se conecta a la computadora.
- **Controladores de dispositivos o unidades de E/S.** Se encargan de hacer la transferencia de información entre la memoria principal y los periféricos. Son el componente electrónico a través del cual se conecta el dispositivo de E/S. Tienen una conexión al bus de la computa dora y otra para el dispositivo (generalmente mediante cables internos o externos).

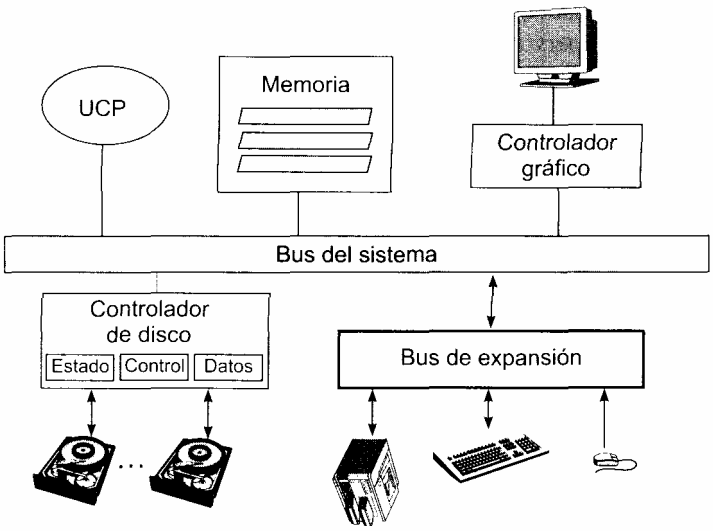


Figura 7.3. *Conexión de dispositivos de E/S a una computadora.*

Los controladores son muy variados, casi tanto como los dispositivos de E/S. Muchos de ellos, como los de disco, pueden controlar múltiples dispositivos. Otros, como los de canales de E/S, incluyen su propia UCP y bus para controlar la E/S por programa y evitar interrupciones en la UCP de la computadora. De cualquier forma, en los últimos años ha existido un esfuerzo importante de estandarización de los dispositivos, lo que permite usar un mismo controlador para dispositivos de distintos fabricantes. Un buen ejemplo lo constituyen los dispositivos SCSI (SrnaIl Computer Svstern interftice), cuyos controladores ofrecen una interfaz común independientemente de que se trate de un disco, una cinta, un CD-ROM, etc. Otro buen ejemplo son los controladores IDE (In tegrated Drive Electronics), que suelen usarse para conectar los discos en todas las computadoras personales. En cualquier caso, y sea como sea el controlador, su misión es convertir los datos del formato interno del dispositivo a uno externo que se ofrezca a través de una interfaz de programa ción bien definida.

El controlador es el componente más importante desde el punto de vista del sistema operativo. ya que constituye la interfaz del dispositivo con el bus de la computadora y es el componente que se ve desde la UCP. Su programación se lleva a cabo mediante una interfaz de muy bajo nivel que proporciona acceso a una serie de registros del controlador (Fig. 7.3), incluidos en el mapa de E/S de la computadora, que se pueden acceder mediante instrucciones de máquina de E/S. Hay tres registros importantes en casi todos los controladores: **registro de datos, estado y control.** El re gistro de datos sirve para el intercambio de datos. En él irá el controlador cargando los datos leídos y de él irá extrayendo los datos para su escritura en el periférico. Un bit del registro de estado sirve para indicar que el controlador puede transferir una palabra. En las operaciones de lectura esto significa que ha cargado en el registro de datos un nuevo valor, mientras que en las de escritura significa que necesita un nuevo dato. Otros bits de este registro sirven para que el controlador indique los problemas que ha encontrado en la ejecución de la última operación de E/S. El registro de control sirve para indicarle al controlador las operaciones que ha de realizar. Los distintos bits de este registro indican distintas acciones que ha de realizar el periférico. Para empezar una operación de E/S, la UCP tiene que escribir sobre los registros anteriores los datos de la operación a través de una dirección de E/S o de memoria asignada únicamente al controlador. Este modelo vale tanto para los terminales o la pantalla como para los discos.

Las características del controlador son muy importantes, ya que definen el aspecto del pe riférico para el sistema operativo. Atendiendo a las características del hardware de los dispositivos, se pueden observar los siguientes aspectos distintivos:

- **Dirección de E/S.** En general hay dos modelos de direccionamiento de E/S, los que usan puertos y los que proyectan los registros en memoria.
- **Unidad de transferencia**. Los dispositivos suelen usar unidades de transferencia de tamaño fijo. Hay dos modelos clásicos de dispositivos: de caracteres y de bloques.
- **Interacción computadora-controlador.** La computadora tiene que interaccionar con la computadora para realizar las operaciones de E/S y saber cuándo terminan.

7.2.2. Dispositivos conectados por puertos o proyectados en memoria

Para empezar una operación de E/S, la UCP tiene que escribir sobre los registros anteriores los datos de la operación a través de una dirección de E/S o de memoria asignada únicamente al controlador. Según se haga de una u otra forma, se distingue entre dispositivos conectados por puertos o proyectados en memoria.

El modelo de **dispositivos por puertos** es clásico en las arquitecturas de Intel. En ellas, cuando se instala un dispositivo, a su controlador se le asigna un puerto de E/S, una interrupción

Digitalización realizada con propósito académico

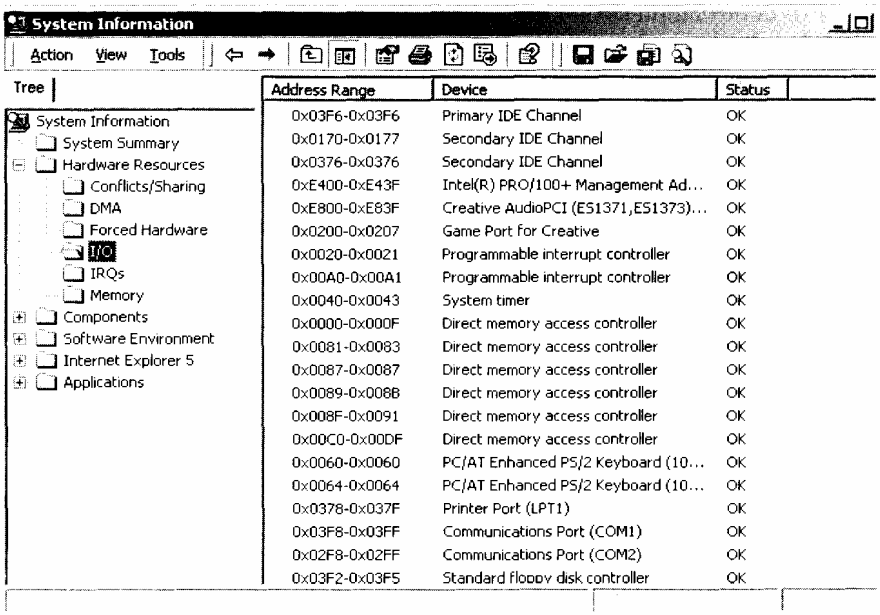


Figura 7.4. *Direcciones de E/S de algunos controladores en un PC con Windows 2000.*

hardware y un vector de interrupción. La Figura 7.4 muestra las direcciones de E/S asignadas a algunos de los dispositivos de E/S de una computadora persona! con el sistema operativo Windows 2000. Para efectuar una operación de E/S la UCP ejecuta operaciones por o portout con la dirección de puerto del dispositivo y con parámetros para indicar qué registro se quiere manipular. Todas las operaciones de entradalsalida (pantalla gráfica, impresoras, ratón, discos, etc.) se realizan usando esas dos instrucciones de lenguaje máquina con los parámetros adecuados. El problema de este tipo de direccionamiento es que exige conocer las direcciones de E/S y programar las instruc ciones especiales de E/S, lo que es significativamente distinto del modelo de memoria de la compu tadora.

El otro modelo de direccionamiento de E/S es el modelo **proyectado en memoria.** Este mo delo, típico de las arquitecturas de Motorola, asigna a cada dispositivo de E/S un rango de di recciones de memoria a través de las cuales se escribe sobre los registros del controlador. En este modelo no hay instrucciones específicas de E/S, sino que las operaciones se llevan a cabo mediante instrucciones máquina de manejo de memoria, lo que permite gestionar un mapa único de direccio nes de memoria. Sin embargo, para no tener conflictos con otros accesos a memoria y para optimi zar las operaciones, se reserva una zona de memoria física para asignar las direcciones de E/S.

7.2.3. Dispositivos de bloques y de caracteres

Los dispositivos de almacenamiento secundario y terciario manejan la información en unidades de tamaño fijo, denominadas **bloques**, por lo que a su vez se denominan **dispositivos de bloques**. Estos bloques se pueden direccionar de manera independiente, lo que permite leer o escribir un bloque con independencia de los demás. Los dispositivos de bloque lo son porque el hardware

Digitalización realizada con propósito académico

fuerza la existencia de accesos de un tamaño determinado. Un disco, por ejemplo, se divide en sectores de 512 bytes o de 1 KB, siendo un sector la unidad mínima de transferencia que el con trolador del disco puede manejar.

Los dispositivos de caracteres, como los terminales, impresoras, tarjetas de red, módems, etcétera, no almacenan información en bloques de tamaño fijo. Gestionan flujos de caracteres de forma lineal y sin ningún tipo de estructura de bloque. Un teclado es un buen ejemplo de estos dispositivos. Está conectado a una UART (Universal Asvnchronous Receiver/Transmiter) que recibe un carácter del teclado cada vez que se pulsa una tecla. No es posible leer un bloque de teclas de un golpe o buscar dentro del dispositivo por ninguna unidad. Un terminal por línea serie también es un dispositivo de caracteres. Su controlador se limita a enviar al periférico el flujo de caracteres que debe representar en la pantalla y a recibir del mismo los caracteres tecleados por el usuario.

7.2.4. E/S programada o por interrupciones

Un controlador de dispositivo o unidad de E/S se encarga de controlar uno o más dispositivos de! mismo tipo y de intercambiar información entre ellos y la memoria principal o unidad central de proceso de la computadora. El controlador debe encargarse además de sincronizar la velocidad del procesador con la del periférico y de detectar los posibles errores que se produzcan en el acceso a los periféricos. En el caso de un controlador de disco, éste debe encargarse de convertir un flujo de bits procedente del disco a un bloque de bytes detectando y corrigiendo, si es posible, los errores que se produzcan en esta transferencia. Una vez obtenido el bloque y comprobado que se encuentra libre de errores, deberá encargarse de transferirlo a memoria principal.

La información entre los controladores de dispositivo y la unidad central de proceso o me moria principal se puede transferir mediante un programa que ejecuta continuamente y lee o escribe los datos del (al) controlador. Con esta técnica, que se denomina **E/S programada**, la transferencia de información entre un periférico y el procesador se realiza mediante la ejecución de una instruc ción de E/S. Con esta técnica, es el procesador el responsable de extraer o enviar datos entre el procesador y el controlador de dispositivo, lo que provoca que el procesador tenga que esperar mientras se realiza la transferencia entre el periférico y el controlador. Dado que los periféricos son sensiblemente más lentos que el procesador, éste deberá esperar una gran cantidad de tiempo hasta que se complete la operación de E/S. En este caso no existe ningún tipo de concurrencia entre la E/S y el procesador ya que éste debe esperar a que finalice la operación.

Aunque esta técnica es muy antigua, ya que proviene del tiempo en que los controladores no tenían interrupciones, actualmente los canales de E/S y algunos multiprocesadores usan esta técnica para evitar que lleguen a la UCP de la computadora muchas interrupciones de E/S. En ambos casos, la técnica es la misma: dedicar una UCP especial para la E/S. La forma de hacerlo es **muestrear** continuamente los registros de estado de los controladores para ver si están disponibles y, en ese caso, leer o escribir los registros. Imagine un canal de E/S al que hay conectados múltiples buses de E/S que, a su vez, tienen múltiples dispositivos de E/S. Si la UCP quiere escribir en uno de ellos, debe mirar su registro de estado hasta que los bits indiquen que no está ocupado. Cuando esto ocurra, escribirá un bloque en los registros del controlador y esperará hasta que los bits de estado indiquen que está disponible. Imagine que quiere leer de otro controlador, deberá esperar a que los bits de estado le indiquen que está disponible, programar la operación y esperar a que se indique que los datos están disponibles. Evidentemente, incluso aunque la UCP esté controlando varios dispositivos de E/S, siempre existe pérdida de ciclos debido a la existencia de las esperas. Sin embargo, existen situaciones en que esto no es así. En algunos sistemas de tiempo real, como por

Digitalización realizada con propósito académico

ejemplo un satélite, la velocidad de E/S es tan rápida (byte/microsegundos) que sería imposible efectuarla con interrupciones, debido al coste de tratar cada interrupción. En estos casos, la E/S programada es la técnica de elección.

Un bucle de E/S programada en la que se controlan las lecturas y escrituras sobre múltiples dispositivos podría tener la estructura que se muestra en el Programa 7. 1. Como se puede observar, el bucle hace un muestreo de todos los dispositivos en orden, siempre que tengan peticiones de E/S pendientes. Este criterio de muestreo se puede alterar usando una prioridad distinta para cada dispo sitivo, por ejemplo.

Programa 7.1. Bucle de E/S progamada para multiples dispositivos.

```
Numero-dispositivos d;                  /* Numero de dispositivos a controlar  */
Numero-dispositivos k;                  /* Numero de dispositivo particular  */
Tamanyo-datos m [d];                    /* Buffer por dispositivo */
Tipo-operación t [d];                    /* lectura, escritura  */
Tamanyo-datos n [d];                     /* Posición de operación por dispositivo */

for (k=0;  k<=d; k++)
        n[ k] = 0;                        /*  Posición al inicio de los buffers */
k = 0;                                    /*  Primer dispositivo */
while (TRUE) {
  if
```

Falta un pedazo de codigo

Digitalización realizada con propósito académico

Con E/S programada el procesador tiene que esperar hasta que el controlador esté listo para recibir o enviar datos, y mientras tanto no realiza ningún trabajo útil. Empleando **E/S dirigida por interrupciones** el procesador envía la orden de E/S al controlador de dispositivo y no espera a que éste se encuentre listo para enviar o transmitir los datos, sino que se dedica a otras tareas hasta que llega una interrupción del dispositivo que indica que se ha realizado la operación solicitada.

El modelo de interrupciones está íntimamente ligado a la arquitectura del procesador. Casi todas las UCP actuales incluyen interrupciones vectorizadas y enmascarables. Es decir, un rango de interrupciones entre 0 y 255, por ejemplo, alguna de las cuales se pueden inhibir temporalmente para no recibir interrupciones de su vector correspondiente. Cada interrupción se asigna a un dis positivo, o un rango de ellos en caso de un controlador SCSI o una cadena de dispositivos tipo daisv chain, que usa el vector correspondiente para indicar eventos de E/S a la UCP. Cuando se programa una operación en un dispositivo, como por ejemplo una búsqueda en un disco, éste contesta con un ACK indicando que la ha recibido, lo que no significa que haya terminado. En este caso existe concurrencia entre la E/S y el procesador, puesto que éste se puede dedicar a ejecutar código de otro proceso, optimizando de esta forma el uso del procesador. Al cabo de un cierto tiempo, cuando el disco ha efectuado la búsqueda y las cabezas del disco están sobre la posición deseada, genera una interrupción (poniendo un 1 en el vector correspondiente). La rutina de tratamiento de la interrup ción se encargará de leer o enviar el dato al controlador. Obsérvese que tanto la tabla de interrupcio nes como la rutina de tratamiento de la interrupción se consideran parte del sistema operativo. Esto suele ser así por razones de seguridad; en concreto, para evitar que los programas que ejecuta un usuario puedan perjudicar a los datos o programas de otros usuarios. Véase el Capítulo 1 para ver más en detalle cómo se trata una interrupción.

Las computadoras incluyen varias señales de solicitud de interrupción, cada una de las cuales tiene una determinada prioridad. En caso de activarse al tiempo varias de estas señales, se tratará la de mayor prioridad, quedando las demás a la espera de ser atendidas. Además, la computadora incluye un mecanismo de **inhibición** selectiva que permite detener todas o determinadas señales de interrup ción. Las señales inhibidas no son atendidas hasta que pasen a estar desinhibidas. La información de inhibición de las interrupciones suele incluirse en la parte del registro de estado que solamente es modificable en nivel de núcleo, por lo que su modificación queda restringida al sistema operativo.

¿Quién asigna las interrupciones a los dispositivos? Normalmente, el sistema operativo se hace cargo de esa asignación cuando instala el dispositivo. Ahora bien, también suele existir la posibilidad de que el administrador fije las interrupciones manualmente (Advertencia 7.1). La Fi gura 7.5 muestra la asignación de interrupciones a dispositivos en un PC con Windows 2000.

ADVERTENCIA 7.1

Nunca asigne interrupciones manualmente si no tiene experiencia con el sistema operativo y la arquitec tura de la computadora. Si origina conflictos entre interrupciones, varios dispositivos usarán el mismo

¿ Quien proporciona la **rutina de tratamiento de interrupción**? Las rutinas de nterrupc suelen tener dos partes: una genérica y otra particular para el dispositivo.La parte genérica pernite:

1. Capturar la interrupción.
2. Salvaguardar el estado del procesador.
3. Activar la rutina de manejo de la interrupción.
4. Indicar al planificador que debe poner lista para ejecutar la rutina particular.
5. Desactivar la interrupción (Advertencia 7.2).

Digitalización realizada con propósito académico

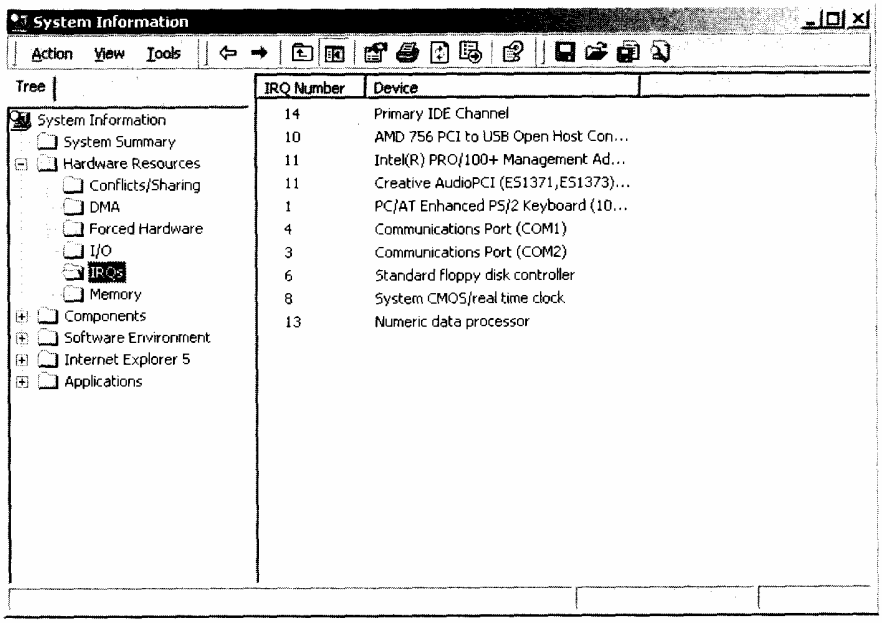


Figura 7.5. *Interrupciones asociadas a algunos controladores en un PC con Windows 2000.*

6. Restaurar el estado del procesador.
7. Ceder el control (RETI).

La rutina de tratamiento particular indica al planificador que encole la rutina particular, que se activa cuando le llega su turno de planificación, posiblemente más tarde. La rutina genérica la propor ciona el sistema operativo y es independiente del dispositivo. Se limita a preparar el entorno de ejecución de la interrupción, salvar los datos y parámetros, llamar a la rutina particular del manejador y restaurar el estado del proceso. La rutina particular la proporciona el fabricante del dispositivo o del sistema operativo, si se trata de un dispositivo estándar. Cuando se compra un dispositivo de EIS, como por ejemplo un ratón, es habitual encontrar un disquete o un CD-ROM con los manejadores del dispositivo. El usuario o el administrador debe instalar estos manejadores en el sistema operativo y reiníciarlo antes de que sea posible acceder al dispositivo.

ADVERTENCIA 7.2

Es importante desactivar las interrupciones después de activar su tratamiento para evitar que se presenten nuevas interrupciones antes de terminar el tratamiento y perder alguna de ellas.

Un caso especial en la arquitectura Intel es la controladora gráfica, que se encarga de gestionar la salida a los dispositivos de mapas de bits (pantallas gráficas). Estas controladoras suelen tener su propia memoria, sobre la cual se llevan a cabo las operaciones de E/S. Aunque la memoria de la controladora se escribe también a partir de un puerto de E/S, sus prestaciones son muy altas (na nosegundos), por lo que el tratamiento de las operaciones de E/S se desvía del estándar en el sistema operativo, ya que estos dispositivos no interrumpen, por lo que se efectúa E/S programada.

Digitalización realizada con propósito académico

Observe que aunque la controladora gráfica tiene asociada una dirección de E/S en la Figura 7.4, no tiene una interrupción asociada en la Figura 7.5.

7.2.5. Mecanismos de incremento de prestaciones

A medida que la tecnología de fabricación de controladores ha ido mejorando, la capacidad de efectuar operaciones autónomas en los mismos se ha incrementado considerablemente. Actualmen te es muy frecuente que un controlador de dispositivo tenga capacidad de procesamiento, memoria interna (hasta 16 MB en controladoras gráficas, por ejemplo) y capacidad de solapar búsquedas en unos dispositivos con transferencias en otros. Estas mejoras convierten al controlador en un auténti co procesador intermedio entre la UCP y el dispositivo, lo que le permite proporcionar varios servicios para incrementar las prestaciones de EIS del dispositivo. En esta sección se comentan los más importantes.

Acceso directo a memoria

Tanto en la E/S programada como la basada en interrupciones, la UCP debe encargarse de la transferencia de datos una vez que sabe que hay datos disponibles en el controlador. Una mejora importante para incrementar la concurrencia entre la UCP y la E/S consiste en que el controlador del dispositivo se pueda encargar de efectuar la transferencia de datos, liberando de este trabajo a la UCP, e interrumpir a la UCP sólo cuando haya terminado la operación completa de EIS. Esta técnica se denomina **acceso directo a memoria** (DMA, Direct Memory Access).

Cuando se utiliza acceso directo a memoria, es el controlador el que se encarga directamente de transferir los datos entre el periférico y la memoria principal, sin requerir intervención alguna por parte del procesador. Esta técnica funciona de la siguiente manera: cuando el procesador desea I Qe un bloq de datos, envía una orden al controlador indicándole la siguiente infor mación (Fig 7.6):

- Tipo de operación: lectura o escritura.
- Periferico involucrado en la operación.
- La dirección de memoria desde la que se va a leer o a la que va a escribir ciirectamente ci controlador de dispositivo (dirección).
- El número de bytes a transferir (contador).

Una vez emitida la orden, el procesador continúa realizando otro trabajo sin necesidad de transferir el bloque de datos. Es el propio controlador el que se encarga de transferir el bloque de datos del periférico a memoria. La transferencia se realiza palabra a palabra. Cuando el controlador ha completado la transferencia, genera una interrupción que activa la rutina de tratamiento correspon diente, de tal manera que se sepa que la operación ha concluido.

Utilizando acceso directo a memoria el procesador únicamente se ve involucrado al inicio y al final de la transferencia. Por tanto, cuando el sistema operativo despierta al proceso que pidió la E/S, no tiene que copiar el bloque a memoria porque ya está allí. El DMA requiere una etapa de almacenamiento intermedio en el controlador del dispositivo para armonizar la velocidad del dis positivo de EIS con la copia de los datos en memoria principal (Fig. 7.6). La razón para este almacenamiento intermedio reside en que una vez que el dispositivo empieza la transferencia de datos, ésta debe hacerse a velocidad constante para evitar transferencias parciales y nuevas esperas de posicionamiento del dispositivo sobre los datos (latencia). Una vez transferidos los datos a la memoria del controlador, éste los copia en memoria principal aprovechando el ancho de banda libre del bus.

Digitalización realizada con propósito académico

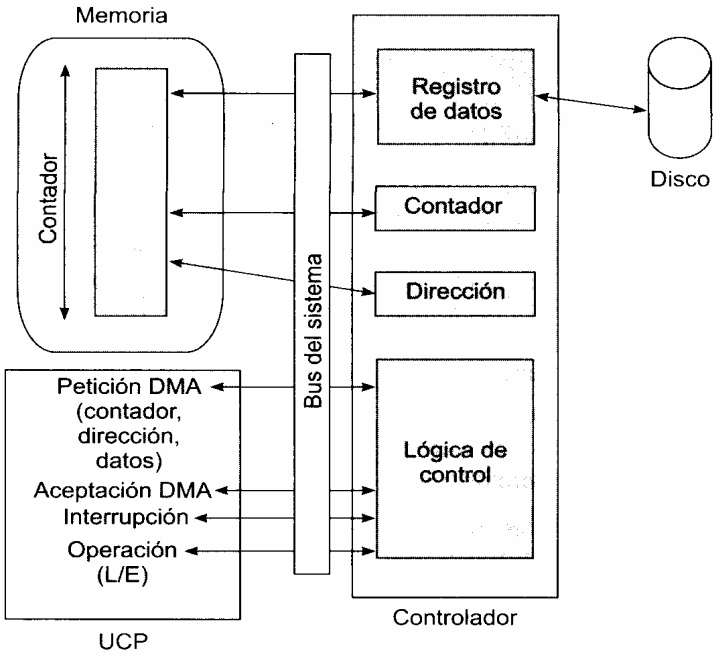


Figura 7.6. *Una operación de E/S con DMA.*

Los pasos a seguir en una operación de E/S con DMA son los siguientes:

1. Programación de la operación de E/S. Se indica al controlador la operación, los datos a transferir y la dirección de memoria sobre la que se efectuará la operación.
2. El controlador contesta aceptando la petición de E/S.
3. El controlador le ordena al dispositivo que lea (para operación de lectura) una cierta

   cantidad de datos desde una posición determinada del dispositivo a su memoria interna. 7.3.
4. Cuando los datos están listos, el controlador los copia a la posición de memoria que tiene en sus registros, incrementa dicha posición de memoria y decrementa el contador de datospendientes de transferir.
5. Los pasos 3 y 4 se repiten hasta que no quedan más datos por leer.
6. Cuando el registro de contador está a cero, el controlador interrumpe a la UCP para in dicar que la operación de DMA ha terminado.

Canales de EIS con DMA

Un canal de E/S se puede mejorar si se incluye el concepto de DMA que permite al controlador ejecutar instrucciones de E/S. Con estos sistemas, las instrucciones de EIS se almacenan en me moria principal y son ejecutadas ordenando al procesador del canal que ejecute un programa en memoria. Dicho programa se encarga de designar dispositivos y zonas de memoria de E/S.

Hay dos tipos principales de canales de E/S: canal selector y canal multiplexor. Ambos pueden interaccionar con varios dispositivos de E/S, pero mientras el canal selector sólo puede transferir datos de un dispositivo a la vez, el canal multiplexor puede transferir datos de varios dispositivos simultáneamente.

Digitalización realizada con propósito académico

Caches de disco en el controlador

Las caches de datos, tan populares en sistemas operativos, han irrumpido en el mundo de los contro'adores de disco con mucha fuerza. La idea es aprovechar la memoria interna de los controladores para leer los datos por adelantado, evitando muchas operacines de usqueda en el disco y sobre todo los tiempos de latencia necesarios para esperar a que los datos pasen de nuevo bajo las cabezas del disco [Biswas, 19931.

La proximidad espacial permite optimizar la E/S en el ámbito de controlador, ya que en lugar de leer un sector, o un grupo de ellos, se leen pistas enteras en cada vuelta de disco, lo que permite traer múltiples bloques de datos en una única operación. En los canales de E/S, donde suele haber mucha memoria interna, se guardan en memoria varias pistas por cada dispositivo de E/S.

Estos mecanismos permiten optimizar mucho la E/S, especialmente en operaciones de lectura con un comportamiento conocido. Para evitar afectar al rendimiento de las operaciones que no responden a patroues de proximidad espacial predecibles, los controladores incluyen instrucciones para desactivar este mecanismo, siempre que el sistema operativo lo crea conveniente.

Solapamiento de búsquedas y transferencias

Los controladores de disco actuales permiten la conexión de varios dispositivos de E/S y tienen un canal de comunicaciones con ellos de varios MB. Un controlador SCSI-2 permite conectar hasta ocho dispositivos y tiene un ancho de banda de 40 MB/segundo. Un problema grave de los disposi tivos es que las operaciones de búsqueda son lentas y, mientras el controlador espera la respuesta, el bus de comunicaciones está vacío porque no está siendo usado por ningún dispositivo.

Para optimizar el uso del conjunto de los dispositivos, muchos controladores actuales pro graman las operaciones de búsqueda en los dispositivos y mientras reciben la respuesta transfieren datos de otros dispositivos listos para leer o escribir. De esta forma existe paralelismo real entre los dispositivos, lo que permite explotar al máximo el canal de comunicaciones. ¿Cómo sabe el con trolador cuándo ha terminado la espera? Pues con un sistema similar al de la E/S no bloqueante: programa un temporizador y cuando vence le pregunta al dispositivo si ya está listo para transmitir.

7.3. ARQUITECTURA DEL SISTEMA DE ENTRADA/SALIDA

El sistema de entradalsalida está construido como un conjunto de manejadores apilados, cada uno de los cuales está asociado a un dispositivo de entradalsalida (archivos, red, etc.). Ofrece a las aplicaciones y entornos de ejecución servicios genéricos que permiten manejar los objetos de en trada/salida del sistema. A través de ellos se puede acceder a todos los manejadores de archivos y de dispositivos tales como discos, cintas, redes, consola, tarjetas de sonido, etc.

En esta sección se estudia la estructura y los componentes del sistema de E/S y el software del sistema de É/S.

7.3.1. Estructura y componentes del sistema de E/S

La arquitectura del sistema de entrada/salida (Fig. 7.7) es compleja y está estructurada en capas, cada una de las cuales tiene una funcionalidad bien definida:

- **Interfaz del sistema operativo para entrada/salida.** Proporciona servicios de E/S síncrona y asíncrona a las aplicaciones y una interfaz homogénea para poderse comunicar con losmanejadores de dispositivo ocultando los detalles de bajo nivel.

Digitalización realizada con propósito académico

- **Sistemas de archivos.** Proporcionan una interfaz homogénea, a través del sistema de ar chivos virtuales, para acceder a todos los sistemas de archivos que proporciona el sistema operativo (FFS, SV, NTFS, FAT, etc.). Permite acceder a los manejadores de los dispositi vos de almacenamiento de forma transparente, incluyendo en muchos casos, como NFS o NTFS, accesos remotos a través de redes. En algunos sistemas, como Windows NT, los servidores para cada tipo de sistema de archivos se pueden cargar y descargar dinámica- mente como cualquier otro manejador.
- **Gestor de redes.** Proporciona una interfaz homogénea para acceder a todos los sistemas de red que proporciona el sistema operativo (TCP/IP, Novell, etc.). Además, permite acceder a
- los manejadores de cada tipo de red particular de forma transparente.
- **Gestor de bloques**. Los sistemas de archivos y otros dispositivos lógicos con acceso a nivel de bloque se suelen limitar a traducir las operaciones del formato del usuario al de bloques que entiende el dispositivo y se las pasan a este gestor de bloques. El gestor de bloques admite únicamente operaciones a nivel de bloque e interacciona con la cache de bloques para optimi zar la E/S.
- **Gestor de cache.** Optimiza la entradalsalida mediante la gestión de almacenamiento inter medio en memoria para dispositivos de E/S de tipo bloque [1985]. Aunque en el Capítulo 8 se comenta con más detalle la estructura de la cache de bloques, es importante saber que su tamaño varía dinámicamente en función de la memoria RAM disponible y que los bloques se escriben a los dispositivos según una política bien definida, que en UNIX y WINDOWS NT es la de escritura retrasada.
- **Manejadores de dispositivo.** Proporcionan operaciones de alto nivel sobre los dispositivos y las traducen en su ámbito interno a operaciones de control de cada dispositivo particular. Como ya se ha dicho, los manejadores se comunican con los dispositivos reales mediante puertos o zonas de memoria especiales.

Cada uno de los componentes anteriores se considera un objeto del sistema, por lo que habi tualmente todos los sistemas operativos permiten modificar el sistema operativo de forma estática o dinámica para reemplazar, añadir o quitar manejadores de dispositivos. Sin embargo, habitualmen te, y por razones de seguridad, no se permite a las aplicaciones de usuario acceder directamente a los dispositivos, sino a través de la interfaz de llamadas al sistema operativo.

7.3.2. Software de E/S

Una vez examinada la arquitectura de E/S de una computadora y las técnicas posibles de transferen cia entre el procesador y los periféricos, en esta sección se va a presentar la forma en la que estructura el sistema operativo el software de gestión de E/S. Este software se organiza en una serie de capas que se muestran en la Figura 7.8. Estas capas se corresponden, en general, con los niveles de la arquitectura de E/S.

Como puede verse en dicha figura, los procesos de usuario emiten peticiones de entrada/salida al sistema operativo. Cuando un proceso solicita una operación de E/S, el sistema operativo prepara dicha operación y bloquea al proceso hasta que se recibe una interrupción del controlador del dispositivo indicando que la operación está completa. Las peticiones se procesan de forma estruc turada en las siguientes capas:

- Manejadores de interrupción.
- Manejadores de dispositivos o drivers.

Digitalización realizada con propósito académico

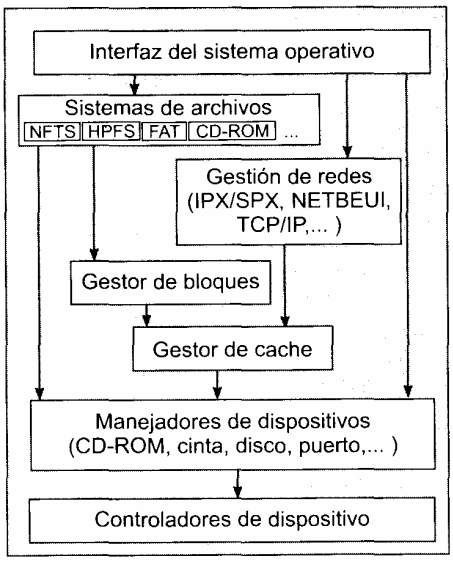


Figura 7.7. *Arquitectura del sistema de entrada/salida.*

- Software de EIS independiente de los dispositivos. Este software está formado por la parte de alto nivel de los manejadores, el gestor de cache, el gestor de bloques y el servidor de archivos.
- Interfaz del sistema operativo. Llamadas al sistema que usan las aplicaciones de usuario.

El uso de capas conlleva la realización de varias copias de datos, alguna de las cuales son inevita bles. En algunos casos, la copia que se realiza en el núcleo del sistema operativo puede ser innecesaria, por lo que existen mecanismos para acceder directamente a los controladores desde la interfaz de E/S del sistema a los manejadores. Sin embargo, y como norma general, esa copia existe siempre.

El sistema operativo estructura el software de gestión de E/S de esta forma para ofrecer a los usuarios una serie de servicios de E/S **independientes de los dispositivos**. Esta independencia implica que deben emplearse los mismos servicios y operaciones de E/S para leer datos de un disquete, de un disco duro, de un CD-ROM o de un teclado, por ejemplo. Como se verá más adelante, el servicio read de POSIX puede utilizarse para leer datos de cualquiera de los dispositi vos citados anteriormente. Además, la estructuración en capas permite hacer frente a la gestión de errores que se pueden producir en el acceso a los periféricos en el nivel de tratamiento más adecua do. A continuación, se describe más en detalle cada uno de los componentes.

Manejadores de interrupción

Los manejadores de interrupción se encargan de tratar las interrupciones que generan los controla dores de dispositivos una vez que éstos están listos para la transferencia de datos o bien han leído o escrito los datos de memoria principal en caso de acceso directo a memoria. Para tratar dicha interrupción se ejecuta el correspondiente manejador de interrupción cuyo efecto es el de salvar los registros, comunicar el evento al manejador del dispositivo y restaurar la ejecución de un proceso (que no tiene por qué ser el interrumpido). En la sección anterior se mostró más en detalle cómo se trata una interrupción.

Digitalización realizada con propósito académico

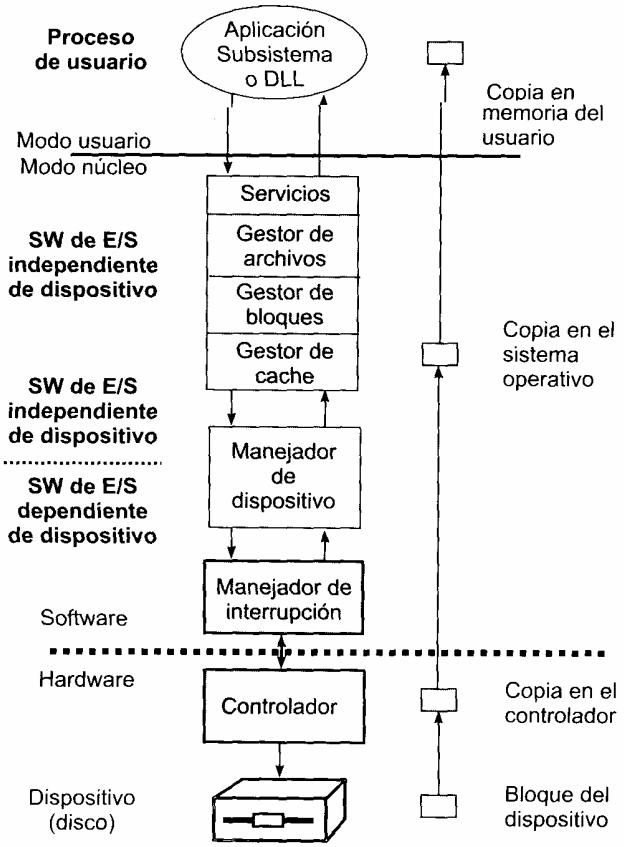


Figura 7.8. *Estructuración del software de E/S y flujo de una operación de E/S.*

Los manejadores de interrupción suelen hacer algo más que comunicar el evento al manejador de dispositivo. Cuando una interrupción ocurre muy frecuentemente, caso del reloj, o cuando la cantidad de información a transferir es muy pequeña, caso del teclado, sería muy costoso comunicar siempre el evento al manejador de dispositivo asociado. En estos casos, el propio manejador de interrupción registra la ocurrencia del evento, bien mediante el incremento de una variable global para el reloj o la acumulación de caracteres en un buffer del teclado. La notificación al manejador se hace únicamente cada cierto número de ocurrencias del evento, en el caso del reloj, o activando unflag que indica que hay datos en el buffer del teclado.

Manejadores de dispositivos

Cada dispositivo de E/S, o cada clase de dispositivos, tiene un manejador asociado en el sistema operativo. Dicho manejador incluye: código independiente del dispositivo para proporcionar al nivel superior del sistema operativo una interfaz de alto nivel y el código dependiente del dis positivo necesario para programar el controlador del dispositivo a través de sus registros y man datos. La tarea de un manejador de dispositivo es aceptar peticiones en formato abstracto, de la parte del código de E/S independiente del dispositivo, traducir dichas peticiones a términos que entienda el controlador, enviar al mismo las órdenes adecuadas en la secuencia correcta y esperar a que se cumplan. La Figura 7.9 muestra un diagrama de flujo con las operaciones de un manejador.

Digitalización realizada con propósito académico

Todos los manejadores tienen una lista de peticiones pendientes por dispositivo donde se encolan las peticiones que llegan de niveles superiores. El manejador explora la lista de peticiones, extrae una petición pendiente y ordena su ejecución. La política de extracción de peticiones de la lista es dependiente de manejador y puede ser FIFO, con pr etc. Una cz enviada la pet al controlador, el manejador se bloquea o no, dependiendo de la velocidad del dispositivo. Para los lentos (discos) se bloquea esperando una interrupción. Para los rápidos (pantalla, discos RAM, etcétera) responde inmediatamente. Después de recibir el fin de operación, controla la existencia de errores y devuelve al nivel superior el estado de terminación de la operación. Si tiene operaciones pendientes en la cola de peticiones, atiende a la siguiente, en caso de que le toque ejecutar después de la operación de E!S. En caso contrario se bloquea.

En los sistemas operativos modernos, como Windows NT [Solomon, 19981, los manejadores se agrupan en clases. Para cada clase existe un manejador genérico que se encarga de las operacio nes de E/S para una clase de dispositivos, tales como el CD-ROM, el disco, la cinta o un teclado. Cuando se instala un dispositivo particular, como por ejemplo el disco SEAGATE S300, se crea una instanclación de manejador de c eoii ‘ius pai tiiiciro o dc c objetn. Todaç laç. funciones comunes al manejador de una clase se llevan a cabo en el manejador genérico y las

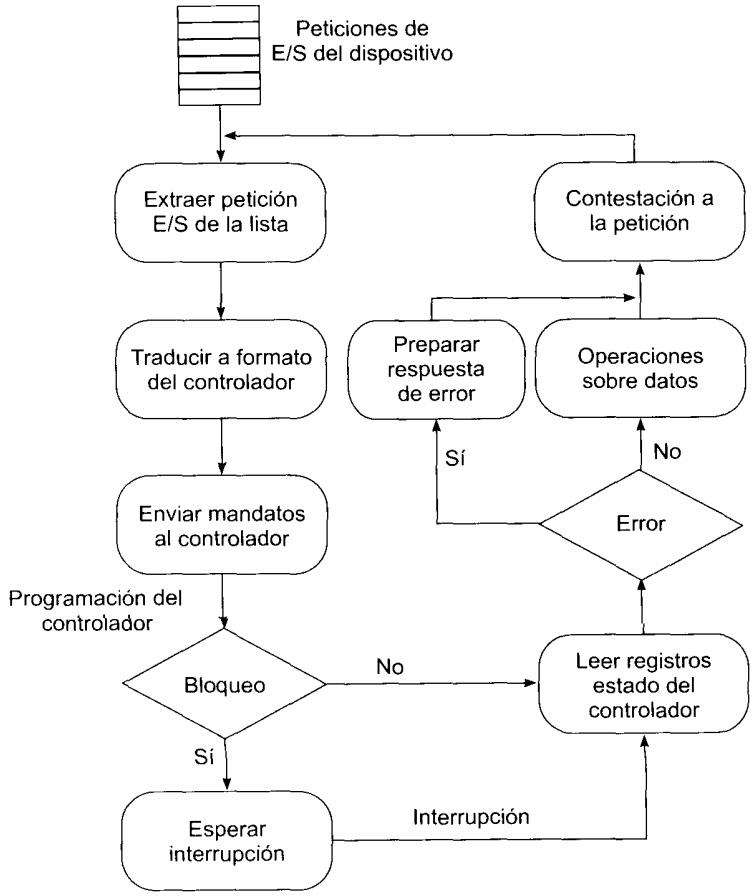


Figura 7.9. *Diagrama de flujo con las operaciones de un manejador.*

Digitalización realizada con propósito académico

particulares en el del objeto. De esta forma se crea un apilamiento de manejadores que refleja muy bien qué operaciones son independientes del dispositivo y cuáles no.

Software de E/S independiente del dispositivo

La mayor parte del sistema de E/S es software independiente de dispositivo. Como se puede ver en la Figura 7.8, este nivel incluye el sistema de archivos y el de gestión de red, el gestor de bloques, la cache de bloques y una parte de los manejadores de dispositivo. La principal función de esta capa de software es ejecutar las funciones de E/S que son comunes a todos los dispositivos a través de una interfaz uniforme, Internamente, en este nivel se proporciona acceso a nivel de bloques o caracteres, almacenamiento intermedio, gestión de los dispositivos, planificación de la E/S y con trol de errores.

El **tamaño de acceso** a nivel de bloques se hace usando tamaños de bloque de acceso comunes para todo un sistema de archivos, lo que permite ocultar que cada dispositivo puede tener distinto tamaño de sector y distinta geometría. Estos detalles quedan ocultos por la capa de software in dependiente de dispositivo que ofrece una interfaz sobre la base de bloques lógicos del sistema de archivos. Lo mismo ocurre con los dispositivos de caracteres, algunos de los cuales trabajan con un carácter cada vez, como el teclado, mientras otros trabajan con flujos de caracteres, como el modem o las redes.

Para optimizar la E/S y para armonizar las peticiones de usuario, que pueden ser de cualquier tamaño, con los bloques que maneja el sistema de archivos, el software de E/S proporciona **almacenamiento intermedio** en memoria del sistema operativo. Esta facilidad se usa para tres cosas:

- Optimizar la E/S evitando accesos a los dispositivos.
- Ocultar las diferencias de velocidad con que cada dispositivo y usuario manejan los datos. 7
- Facilitar la implementación de la semántica de compartición, al existir una copia única de los datos en memoria.

El sistema de E/S mantiene buffers en distintos componentes. Por ejemplo, en la parte del manejador del teclado independiente del dispositivo existe un buffer para almacenar los caracteres que teclea el usuario hasta que se pueden entregar a los niveles superiores. Si se usa una línea serie para leer datos de un sistema remoto y almacenarlos en el disco, se puede usar un buffer para guardar temporalmente los datos hasta que tengan una cierta entidad y sea rentable escribirlos a disco. Si se está leyendo o escribiendo un archivo, se guardan copias de los bloques en memoria para no tener que acceder al disco si se vuelven a leer.

La **gestión de los dispositivos** agrupa a su vez tres servicios: nombrado, protección y control de acceso. El nombrado permite traducir los nombres de usuario a identificadores del sistema. Por ejemplo, en UNIX, cada dispositivo tiene un nombre (p. ej.: /dev/cdrom) que se traduce en un único identificador interno (o nodo-i), que a su vez se traduce en un único número de dispositivo principal (clase de dispositivo) y secundario (elemento de la clase). Cada dispositivo tiene asociada una información de protección (en UNIX mediante 3 bits para dueño, grupo y mundo) y este nivel de software asegura que los requisitos de protección se cumplen. Además proporciona control de acceso para que un dispositivo dedicado, como una impresora, sea accedido por un único usuario cada vez.

Una de las funciones principales del sistema de E/S es la **planificación de la E/S** de los distintos componentes. Para ello se usan colas de peticiones para cada clase de dispositivo, de las que se extraen las peticiones de cada dispositivo en particular. Cada una de estas colas se ordena siguiendo una política de planificación, que puede ser distinta en cada nivel. Imagine el caso de LINUX, donde existe una cola global de peticiones de E/S, ordenadas en orden FIFO, para los discos instalados. Cuando un manejador de disco queda libre, busca la cola global para ver si hay

Digitalización realizada con propósito académico

peticiones para él y, si existen, las traslada a su cola de peticiones particular ordenadas según la política SCAN, por ejemplo. Este mecanismo permite optimizar la E/S al conceder a cada me canismo la importancia que, ajuicio de los diseñadores del sistema operativo, se merece. En el caso de Windows NT, por ejemplo, el ratón es el dispositivo de E/S más prioritario del sistema. La razón que hay detrás de esta política es conseguir un sistema muy interactivo. En otros sistemas, como UNIX, las operaciones de disco son más prioritarias que las del ratón para poder desbloquear rápidamente a los procesos que esperan por la E/S. Sea cual sea el criterio de planificación, todos los sistemas de E/S planifican las actividades en varios lugares.

Por último, este nivel proporciona **gestión de errores** para aquellos casos que el manejador de dispositivo no puede solucionar. Un error transitorio de lectura de un bloque se resuelve en el manejador reintentando su lectura. Un error permanente de lectura no puede ser resuelto y debe ser comunicado al usuario para que tome las medidas adecuadas. En general, todos los sistemas operativos incluyen alguna forma de control de errores internos y de notificación al exterior en caso de que esos errores no se puedan resolver. Imagine, por ejemplo, que una aplicación quiere leer de un dispositivo que no existe. El sistema de E/S verá que el dispositivo no está y lo notificará a los niveles superiores hasta que el error llegue a la aplicación. Sin embargo, es importante resaltar que los sistemas operativos son cada vez más robustos y cada vez incluyen más control y reparación de errores, para lo cual usan métodos de paridad, checksums, códigos correctores de error, etc. Además, la información que proporcionan cuando hay un error es cada vez mayor. En Windows NT, por ejemplo, existen monitores que permiten ver el comportamiento delas operaciones de E/S.

7.4. INTERFAZ DE APLICACIONES

Las aplicaciones tienen acceso al sistema de E/S a través de las llamadas al sistema operativo relacionadas con la gestión de archivos y con la E/S, como íoctl por ejemplo. En muchos casos, las aplicaciones no acceden directamente a las llamadas del sistema, sino a utilidades que llaman al sistema en representación del usuario. Las principales utilidades de este estilo son:

- Las **bibliotecas** de los lenguajes, como la libc. so de C, que traducen la petición del usua rio a llamadas del sistema, convirtiendo los parámetros allí donde es necesario. Ejemplos de utilidades de biblioteca en C son fread, fwríte o printf. Las bibliotecas de enlace dinámico (DLL) de Windows. Por ejemplo, Kernel32 .dll incluye llamadas para la ges tión de archivos y otros componentes de E/S.
- Los **demonios** del sistema, como los de red o los spooler de las impresoras. Son programas privilegiados que pueden acceder a recursos que las aplicaciones normales tienen vetados. Así, por ejemplo, cuando una aplicación quiere acceder al puerto de telnet, llama al de monio de red (inetd) y le pide este servicio. De igual forma, cuando se imprime un archivo, no se envía directamente a la impresora, sino que se envía a un proceso spooler que lo copia a unos determinados directorios y, posteriormente, lo imprime.

Esta forma de relación a través de representantes existe principalmente por razones de se guridad y de control de acceso. Es fácil dejar que un proceso spooler, generalmente desarrollado por el fabricante del sistema operativo y en el que se confía, acceda a la impresora de forma controlada, lo que evita problemas de concurrencia, filtrando los accesos del resto de los usuarios.

La interfaz de E/S de las aplicaciones es la que define el modelo de E/S que ven los usuarios, por lo que, cuando se diseña un sistema operativo, hay que tomar varias decisiones relativas a la funcionalidad que se va a ofrecer al mundo exterior en las siguientes cuestiones:

Digitalización realizada con propósito académico

- Nombres independientes de dispositivo.
- E/S bloqueante y no bloqueante.
- Control de acceso a dispositivos compartidos y dedicados.
- Indicaciones de error.
- Uso de estándares.

La elección de unas u otras características determina la visión del sistema de E/S del usuario.

A continuación, se estudian brevemente cada una de ellas.

Nombres independientes de dispositivo

Usar nombres independientes de dispositivo permite construir un árbol completo de nombres lógi cos, sin que el usuario vea en ningún momento los dispositivos a los que están asociados. La utilidad mount de UNIX es un buen ejemplo de diseño. Si se monta el dispositivo /dev/hda3 sobre el directorio lógico /users, a partir de ese instante todos los archivos del dispositivo se pueden acceder a través de /users, sin que el nombre del dispositivo se vea en ningún momento. Es decir, que el archivo /dev/hda.3/pepe pasa a ser /users/pepe después de la operación de montado.

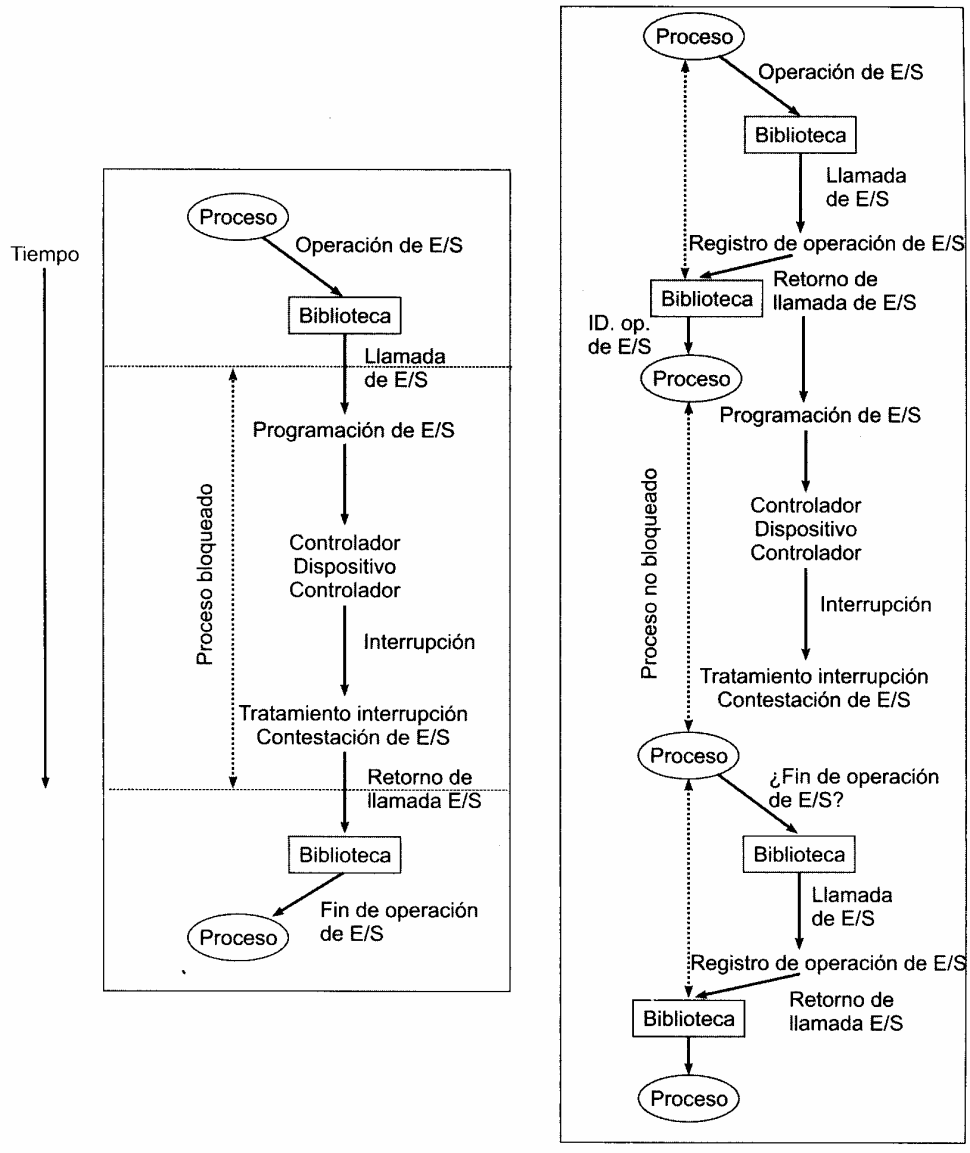
Usar un árbol de nombres único complica la traducción de nombres, por lo que algunos sis temas operativos, como Windows NT, no la incluyen. En Windows, cuando se accede a un dis positivo con un nombre completo, siempre hay que escribir el nombre del dispositivo al que se accede (C:, D:, etc.). En las últimas versiones se enmascaran los dispositivos con unidades de red, pero siempre hay que saber a cuál se quiere acceder. Por ejemplo, Condor\users\profesores (Z:) identifica a una unidad de red que está en la máquina Condor y que está montada en la computadora local sobre el dispositivo lógico z:. No hay en este sistema un árbol de nombres único tan claramente identificado como en UNIX o LINUX.

E/S bloqueante y no bloqueante

La mayoría de los dispositivos de E/S son no bloqueantes, también llamados asíncronos, es decir, reciben la operación, la programan, contestan e interrumpen al cabo de un cierto tiempo. Sólo los dispositivos muy rápidos o algunos dedicados fuerzan la existencia de operaciones de E/S blo queantes (también llamadas síncronas). Sin embargo, la mayoría de las aplicaciones efectúan ope raciones de E/S con lógica bloqueante, lo que significa que emiten Ja operación y esperan hasta tener el resultado antes de continuar su ejecución. En este tipo de operaciones, el sistema operativo recibe la operación y bloquea al proceso emisor hasta que la operación de E/S ha terminado (Fig. 7.lüa), momento en que desbloquea a la aplicación y le envía el estado del resultado de la opera ción. En este caso, la aplicación puede acceder a los datos inmediatamente, ya que los tiene disponi bles en la posición de memoria especificada, a no ser que hubiera un error de E/S. Este modelo de programación es claro y sencillo, por lo que las principales llamadas al sistema de E/S. como read o write en POSEX y ReadFile y WriteFile en Win32, bloquean al usuario y completan la operación antes de devolver el control al usuario.

Las llamadas de E/S no bloqueantes se comportan de forma muy distinta, reflejando mejor Ja propia naturaleza del comportamiento de los dispositivos de E/S. Estas llamadas permiten a la aplicación seguir su ejecución, sin bloquearla, después de hacer una petición de E/S (Fig 7. lOb). El procesamiento de la llamada de E/S consiste en recuperar los parámetros de la misma, asignar un identificador de operación de E/S pendiente de ejecución y devolver a la aplicación este identifica dor. Las llamadas de POSIX aioread aiowrite permiten realizar operaciones no bloqueantes. A continuación, el sistema operativo ejecuta la operación de E/S en concurrencia con la apli-

Digitalización realizada con propósito académico



a) E/S bloqueante *b*) E/S no bloqueante

Figura 7.10. *Flujo de las operaciones de E/S bloqueantes y no bloqueantes.*

cación, que sigue ejecutando su código. Es responsabilidad de la aplicación preguntar por el estado de la operación de EIS, usando una llamada al sistema especial para realizar esta consulta (alo walt), o cancelarla si ya no le interesa o tarda demasiado (aiocancel). En Windows se puede conseguir este mismo efecto indicando, cuando se crea el archivo, que se desea E/S no bloqueante (FILE_FLAG y usando las llamadas ReadFileEx y WriteFileEx.

Digitalización realizada con propósito académico

Este modelo de programación es más complejo, pero se ajusta muy bien al modelo de algunos sistemas que emiten peticiones y reciben la respuesta después de un cierto tiempo. Un programa que esté leyendo datos de varios archivos, por ejemplo, puede usarlo para hacer lectura adelantada de datos y tener los datos de un archivo listos en memoria en el momento de procesarlos. Un programa que escuche por varios canales de comunicaciones podría usar también este modelo (Consejo de programación 7.1).

CONSEJO DE PROGRAMACIÓN 7.1

El modelo de E/S no bloqueante, o asíncrono, es complejo y no apto para programadores o usuarios noveles del sistema operativo. Si lo usa, debe tener estructuras de datos para almacenar los descriptores de las operaciones que devuelve el sistema y procesar todos ellos, con espera o cancelación. Tenga en cuenta que almacenar el estado de estas operaciones en el sistema tiene un coste en recursos y que el espacio es finito. Por ello, todos los sistemas operativos definen un máximo para el número de operaciones de E/S no bloqueantes que pueden estar pendientes de solución. A partir de este límite, las llamadas no bloqueantes devuelven un error.

Es interesante resaltar que, independientemente del formato elegido por el usuario, el sistemaoperativo procesa siempre las llamadas de E/S de forma no bloqueante, o asíncrona, para permitir la implementación de sistemas de tiempo compartido.

Control de acceso a dispositivos

Una de las funciones más importantes de la interfaz de usuario es dar indicaciones de control de acceso a los dispositivos e indicar cuáles son compartidos y cuáles dedicados. En general, las llamadas al sistema no hacen este tipo de distinciones, que, sin embargo, son necesarias. Imagine qué ocurriría si dos archivos se escribieran en la impresora sin ningún control. Ambos saldrían mezclados, siendo el resultado del trabajo inútil.

Para tratar de resolver este problema se usan dos tipos de mecanismos:

- Mandatos externos (como el lpr para la impresora) o programas especiales (demonios) que se encargan de imponer restricciones de acceso a los mismos cuando es necesario.
- Llamadas al sistema que permiten bloquear (lock) y desbloquear (unlock) el acceso a un dispositivo o a parte de él. Usando estas llamadas, una aplicación se puede asegurar acceso exclusivo bloqueando el dispositivo antes de acceder y desbloqueándolo al terminar sus accesos. Para evitar problemas de bloqueos indefinidos, sólo se permiten bloqueos acon sejados (advisory), nunca obligatorios. Además, es habitual que el único usuario que puede bloquear un dispositivo de E/S como tal sea el administrador del sistema. Para el resto de usuarios, este privilegio se restringe a sus archivos.

La seguridad es un aspecto importante del control de accesos. No basta con que se resuelvan los conflictos de acceso. Hay que asegurar que el usuario que accede al sistema de E/S tiene de rechos de acceso suficientes para llevar a cabo las operaciones que solicita. En UNIX y LINUX, los aspectos de seguridad se gestionan en el gestor de archivos. En Windows NT existe un servidor de seguridad que controla los accesos a los objetos.

Indicaciones de error

Las operaciones del sistema operativo pueden fallar debido a cuestiones diversas. Es importante decidir cómo se va a indicar al usuario esos fallos.

Digitalización realizada con propósito académico

En UNIX, por ejemplo, las llamadas al sistema que fallan devuelven —l y ponen en una variable global errno el código de error. La descripción del error se puede ver en el archivo /usr/include/sys/errno.h o imprimirlo en pantalla mediante la función perrorO. A con tinuación, se muestran algunos códigos de error de UNIX junto con sus descripciones

```
#define   EPERNM    1    /* Not super-user              */
#define   ENOENT    2    /* No such file or directory   */
#define   ESRCH     3    /* No such process             */
#define   EINTR     4    /* interrupted system cali     */
#define   ElO       5    /* I/O error                   */
#define   ENXIO     6    /* No such device or address   */
#define   E2BIG     7    /*k Arg iist too long         */
#define   ENOEXEC   8    /* Exec format error           */
#define   EBAOF     9    /* Bad file number             */
```

En Windows se devuelven más códigos de error en las llamadas a función, e incluso en algu nos parámetros de las mismas. Igualmente, se puede obtener más información del error mediante la función GetLastError ().

Uso de estándares

Proporcionar una interfaz de usuario estándar garantiza a los programadores la portabilidad de sus aplicaciones, así como un comportamiento totalmente predecible de las llamadas al sistema en cuanto a definición de sus prototipos y sus resultados. Actualmente, el único estándar definido para la interfaz de sistemas operativos es POSIX (Portable Operating Systeni interface) [19881, basado principalmente en la interfaz del sistema operativo UNIX. Todos los sistemas operativos modernos proporcionan este estándar en su interfaz, bien como interfaz básica (en el caso de UNIX y LINUX) o bien como un subsistema POSIX (caso de Windows) que se ejecuta sobre la interfaz básica (Win32).

No se hace más énfasis en la interfaz POSIX porque se va estudiando a lo largo de los distintos capítulos.

7.5. ALMACENAMIENTO SECUNDARIO

El sistema de almacenamiento secundario se usa para guardar los programas y datos en dispositivos rápidos, de forma que sean fácilmente accesibles a las aplicaciones a través del sistema de archivos. Esa es la razón por la que en la jerarquía de EIS (Fig. 7,2) los dispositivos de almacenamiento se e jtv ha aia. m RAM.

La Figura 7.11 muestra el sistema de E/S en el sistema operativo LINUX y la parte del mismo correspondiente a la gestion del sistema de almacenamiento secundario. Como se puede ver, hay dos elementos principales involucrados en este sistema.

- **Discos.** El almacenamiento secundario se lleva a cabo casi exclusivamente sobre discos, por lo que es interesante conocer su estructura y cómo se gestionan.
- **Manejadores de disco**. Controlan todas las operaciones que se hacen sobre los discos, entre las que son especialmente Importantes las cuestiones de planificacion de peticiones a disco.

Digitalización realizada con propósito académico

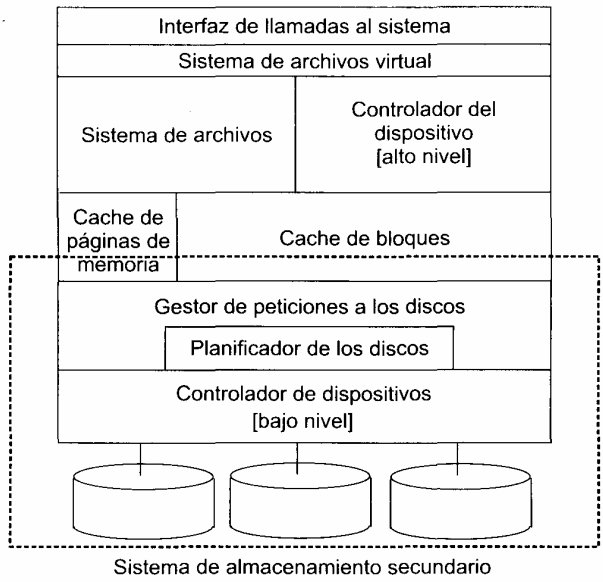


Figura 7.11. *Estructura del sistema de E/S del sistema operativo LINUX.*

En esta sección se tratan los principales aspectos relativos a estos dos componentes, así como algunos otros relativos a tolerancia a fallos y fiabilidad. Los componentes que conforman los nive les superiores del sistema de E/S (sistemas de archivos y caches) se estudian en el Capítulo 8.

7.5.1. Discos

Los discos son los dispositivos básicos para llevar a cabo almacenamiento masivo y no volátil de datos. Además se usan como plataforma para el sistema de intercambio que usa el gestor de me moria virtual. Son dispositivos electromecánicos (HARD DISK) u optomecánicos (CD-ROM y DVD), que se acceden a nivel de bloque lógico por el sistema de archivos y que, actualmente, se agrupan en dos tipos básicos, atendiendo a la interfaz de su controlador:

- **Dispositivos SCSI** (Small Computer System Inteiface), cuyos controladores ofrecen una interfaz común independientemente de que se trate de un disco, una cinta, un CD-ROM, etc. Un controlador SCSI puede manejar hasta ocho discos y puede tener una memoria interna de varios MB.
- **Dispositivos IDE** (Integrated Drive Electronics), que suelen usarse para conectar los discos en todas las computadoras personales. Actualmente estos controladores se han extendido al sistema EIDE, una mejora de IDE que tiene mayor velocidad de transferencia. Puede mane jar hasta cuatro discos IDE. Es barato y muy efectivo.

Y a tres tipos básicos atendiendo a su tecnología de fabricación:

- **Discos duros** (Winchester). Dispositivos de gran capacidad compuestos por varias super ficies magnetizadas y cuyas cabezas lectoras funcionan por efecto electromagnético. Actual mente su capacidad máxima está en los 30 GB, pero se anuncian discos de 100 GB y más.

Digitalización realizada con propósito académico

- **Discos ópticos.** Dispositivos de gran capacidad compuestos por una sola superficie y cuyas cabezas lectoras funcionan por láser. Actualmente su capacidad máxima está en los 700 MB. Hasta hace muy poco, la superficie se agujereaba físicamente y no se podían regrabar. Sin embargo, actualmente existen discos con superficie magnética regrabables.
- **Discos extraíbles.** Dispositivos de poca capacidad similares a un disco duro, pero cuyas cabezas lectoras se comportan de forma distinta. En este tipo se engloban los disquetes (floppies), discos ZIP y JAZZ.

Independientemente del tipo al que pertenezcan, las estructuras física y lógica de todos los discos son muy similares, como se muestra a continuación.

Estructura física de los discos

La Figura 7. 12 muestra la estructura física de un disco duro y los parámetros de un disco comercial. un modelo de la familia Barracuda ATA II de Seagate ISeagate, 2000], tomado como ejemplo. Como se puede ver, un disco duro es un dispositivo de gran capacidad compuesto por varias su perficies magnetizadas y cuyas cabezas lectoras funcionan por efecto electromagnético. Las ca bezas se mueven mediante un motor de precisión para poder viajar por las superficies que com ponen el disco.

En el ámbito organizativo, las superficies del disco están divididas en cilindros, con una pista para cada cabeza y un cierto número de sectores por pista. En el caso del ejemplo, 1023, 256 y 83 respectivamente. Los cilindros se llaman así por ser la figura geométrica que forman las cabezas al moverse en paralelo y simultáneamente sobre los discos. El tamaño del sector es 512 bytes. por lo que multiplicando los números anteriores se debe poder obtener la capacidad del disco:

Capacidad = cilindros * pistas * sectores * tamaño sector

Particularizando esta ecuación para el caso del ejemplo:

Capacidad = 1023 * 256 * 83 * 512 = 10,3 GB

Los parámetros de la estructura física del disco son muy importantes para el manejador del mismo, ya que las operaciones de EIS se calculan y optimizan mediante dos parámetros fundamen tales: **tiempo de búsqueda** (lo que se tarda en ir de una pista a otra) y **tiempo de latencia** (el tiempo medio que tardan en llegar los datos debajo de las cabezas, una vez posicionadas en la pista) [vy, 19951. A grosso modo, se puede calcular el tiempo de acceso con un desplazamiento de a cilindros como:

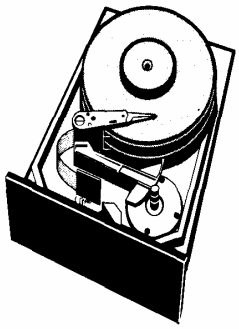
T_acceso = n * T_búsqueda + T_latencia + T_transferencia

donde el tiempo de transferencia para un sector depende de la velocidad rotacional del disco, que en este caso es 7.200 rpm, y del tamaño del sector, en este caso 512 bytes.

Los parámetros de configuración física de un disco dependen de dos cosas:

1. El diseño del fabricante, que define su velocidad de rotación, ancho de banda del bus, etc.
2. La operación de formato físico, que define el tamaño del sector y la posición de los sectores en las pistas. Los discos duros suelen venir con formato físico de fábrica. En los disquetes se lleva a cabo el formato físico al mismo tiempo que el lógico. De cualquier forma, todos los sistemas operativos incluyen utilidades del administrador para dar for mato físico a un dispositivo, como el mandato fdformat de LINUX (Advertencia 7.3).

Digitalización realizada con propósito académico



SEAGATE Barracuda ATA II ST330630A

Parámetro	Valor
Capacidad:	10,2 GB
Cilindros:	1023
Cabezas:	256
Sectores:	83
Velocidad:	7.200 rpm
T. búsqueda:	8,5 mseg
Latencia:	4,16 mseg
Memoria:	2 MB

Figura 7.12. *Estructura física de un disco y parámetros arquitectónicos.*

ADVERTENCIA 7.3

Nunca dé formato físico a un disco duro a no ser que esté muy seguro de lo que hace y conozca muy bien el dispositivo. Para efectuar esta operación se piden parámetros muy detallados de la estructura física del disco, que a veces no es fácil conocer.

Las ecuaciones anteriores se ven influenciadas por varios factores:

• Densidad de cada pista. En los discos modernos las pistas tienen distinta densidad, por lo que el número de sectores varía de unas pistas a otras. Si se mantiene el número de bytes constante por pista, las pistas más pequeñas tienen mayor densidad, lo que afecta mucho al controlador. Este problema se resuelve en el ámbito del controlador. Los controladores SCSI e IDE son de este tipo.

• Intercalado de sectores. El controlador debe emplear tiempo en copiar los datos leídos a memoria principal. Durante este tiempo no puede estar transfiriendo datos del disco, que se sigue moviendo a velocidad constante, lo que significa que si quiere leer el sector siguiente deberá esperar a que las cabezas lectoras den una vuelta y lleguen de nuevo a ese bloque. Una solución a este problema es intercalar sectores del disco de forma que no sean consecutivos. Por ejemplo, en un disco se pueden colocar los bloques con un factor de entrelazado de 2, de forma que el orden en una pista de 8 sectores sería: 0, 3, 6, 1, 4, 7, 2, 5. Así, una vez transmitidos los datos del controlador, las cabezas están colocadas sobre el siguiente bloque a leer. El factor de entrelazado se puede definir cuando se hace el formato físico del disco.

• Almacenamiento intermedio en el controlador. La existencia de memoria en el controla dor permite optimizar mucho las operaciones de E/S, ya que en lugar de leer sectores sueltos de una pista se puede optar por leer la pista entera y mantenerla en memoria del controlador. Si las peticiones de E/S son para archivos contiguos en disco, el rendimiento se puede incrementar considerablemente.

• Controladores inteligentes. Cuando controlan varios dispositivos, permiten efectuar opera ciones de búsqueda de forma solapada, como si funcionaran guiados por interrupciones. De esta forma pueden solapar varios tiempos de búsqueda mejorando mucho el rendimiento global del sistema. Obviamente, esta optimización no es posible mientras se efectúa trans ferencia de datos ya que el bus del controlador está ocupado.

Digitalización realizada con propósito académico

Todos los detalles expuestos en esta sección, concernientes a la estructura física de los discos,

no son visibles desde el exterior debido a que el controlador se encarga de ofrecer una visión lógica del dispositivo como un conjunto de bloques, según se describe en la sección siguiente.

Estructura lógica de los discos

Una vez que el disco ha sido formateado en el ámbito físico, se procede a crear su estructura lógica, que es la realmente significativa para el sistema operativo. Como parte del formato físico sc crea un sector especial, denominado sector O, aunque no siempre es el primero del disco, que se reserva para incluir en él la información de distribución lógica del disco en subconjuntos denominados volúmenes o particiones. Esta información de guarda en una estructura denominada tabla de par ticiones (Fig. 7. 13). Una partición es una porción contigua de disco delimitada por un sector inicial y final. No existe ninguna limitación más para crear particiones, si bien algunos sistemas operativos exigen tamaños de partición mínimos en algunos casos. En el caso de la Figura 7.13, el dispositivo /dev/hda se ha dividido en dos particiones primarias usando la utilidad fdisk de LINUX:

• Una de intercambio (sivap), llamada /dev/hda

• Una extendida (que tiene el sistema de archivos ext2 de LINUX), llamada /dev/hda2.

A su vez, la partición extendida se ha dividido en tres subparticiones lógicas /dev/hda5.

/dev/hda6 y /dev/hda7.

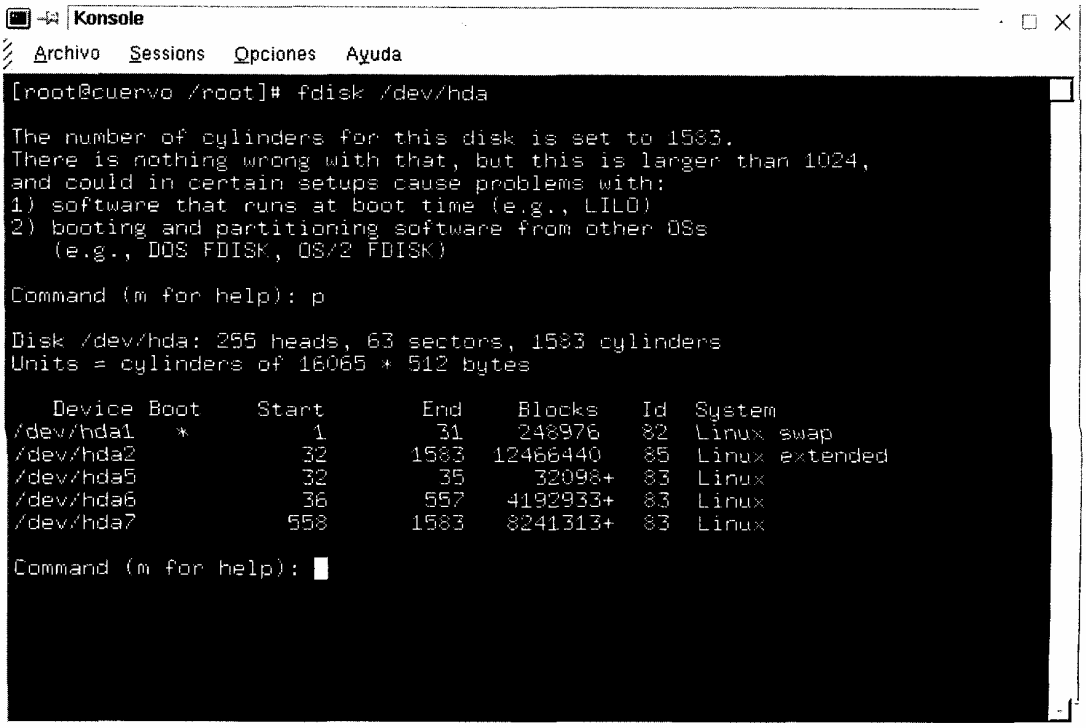


Figura 7.13. *Tabla de particiones de un disco con el sistema operativo LINUX.*

Digitalización realizada con propósito académico

FALTA LA PAG. 378

Digitalización realizada con propósito académico

en memoria. Cuando ha terminado salta a la dirección O de memoria (donde puso el núcleo) y se arranca el sistema operativo.

La segunda tarea importante que) a ca ia operac de Yomia 6gko es ‘ta detecc de ¡os bloques lógicos defectuosos y su inclusión en una lista de bloques defectuosos. Estos bloques no son asignados nunca por el sistema de E/S, para lo que se marcan siempre como ocupados y no se liberan nunca. Ahora bien, ¿cómo se sabe que un bloque es defectuoso? Un bloque es defectuoso porque alguno de los sectores que lo componen son defectuosos. ¿Cómo se sabe que un sector es defectuoso? Pues se sabe porque cuando se efectúa el formateo físico se crea una estructura de sector que contiene, al menos, tres elementos: cabecera, área de datos y cola. La cabecera incluye, entre otras cosas, el número de sector y un código de paridad. La cola suele incluir el número de sector y un código corrector de errores (ECC, Error Correcting Code). Para verificar el estado del sector se escribe una información y su ECC. A continuación se lee y se recalcula su ECC. Si el sector está defectuoso, ambos ECC serán distintos, lo que indica que el ECC es incapaz de corregir los errores existentes en el sector. En ese instante, el bloque al que pertenece el sector se incluirá en la lista de bloques defectuosos.

La tercera tarea consiste en elaborar una lista de bloques de repuesto para el caso en que algún bloque del dispositivo falle durante el tiempo de vida del mismo. Los discos están sujetos a rozamiento, suciedad y otros elementos que pueden dañar su superficie. En este caso, algunos bloques que antes eran válidos pasarán a la lista de bloques defectuosos. Para evitar que se reduzca el tamaño del dispositivo y para ocultar este defecto en el ámbito del controlador, se suplanta este bloque por uno de los de la lista de repuesto, que a todos los efectos sustituye al anterior sin tener que modificar en absoluto el manejador.

7.5.2. El manejador de disco

El manejador de disco se puede ver como una caja negra que recibe peticiones de los sistemas de archivos y del gestor de bloques y las traslada a un formato que entiende el controlador del dis positivo. Sin embargo, el diseñador de sistemas operativos debe entrar en los detalles del interior de un controlador y determinar sus funciones principales, entre las que se incluyen:

1. Proceso de la petición de E/S de bloques.

2. Traducción del formato lógico a mandatos del controlador.

3. Inserción de la petición en la cola del dispositivo, llevando a cabo la política de planifica ción de disco pertinente (FIFO, SJF, SCAN, CSCAN, EDF, etc.).

4. Envío de los mandatos al controlador, programando el DMA.

5. Bloqueo en espera de la interrupción de E/S.

6. Comprobación del estado de la operación cuando llega la interrupción.

7. Gestión de los errores, si existen, y resolverlos si es posible.

8. Indicación del estado de terminación al nivel superior del sistema de E/S.

En LINUX y Windows NT (Fig. 7.8), el paso 1 se lleva a cabo en un manejador para la clase de dispositivo disco, pero independiente del tipo de disco en particular, denominado manejador genérico. Sin embargo, el paso 2 se lleva a cabo en un manejador dependiente del dispositivo, denominado manejador particular. ¿Cómo distingue el manejador el tipo de dispositivo y el dispositivo en particular? Los distingue porque en la petición de E/S viene información que lo indica. En el caso de UNIX y LINUX, por ejemplo, en cada descriptor de archivo hay dos números, denominados número primario (majornumber) y número secundario (minornumber) [19861, que indican el tipo de dispositivo (p. ej.: disco) y el número de dispositivo (p. ej.: 3 para hda3). En Windows NT hay conceptos equivalentes.

Digitalización realizada con propósito académico

A continuación, se inserta la petición en una cola de peticiones pendientes. Dependiendo del diseño del manejador, existe una cola global para cada tipo de dispositivo, una cola para dispositivo particular o ambas. De cualquier forma, una vez insertadas las peticiones en las colas, el manejador se encarga de enviarlas al controlador cuando puede ejecutar operaciones porque este último está libre y se bloquea en espera de la interrupción del dispositivo que indica el fin de la operación de E/S. Esta operación puede terminar de forma correcta, en cuyo caso se indica al nivel superior eJ estado de terminación, o de forma incorrecta, en cuyo caso hay que analizar los errores para ver si se pueden resolver en el manejador o deben comunicarse al nivel superior.

A continuación, se estudia más en detalle la estructura de un manejador de disco, así como las

técnicas de planificación de disco y de gestión de errores en el manejador de disco.

Estructura de un manejador de disco

Los manejadores de disco actuales están compuestos en realidad por un conjunto de manejadores apilados, como se muestra en la Figura 7.14, que refleja una estructura similar a la de los ma nejadores de disco de LINUX [19961. En Windows NT esta tendencia es todavía más acusa da, ya que los manejadores están diseñados como objetos y existe herencia de propiedades y com portamiento.

En cada nivel de la pila se lleva a cabo una funcionalidad específica, que es más dependiente del dispositivo a medida que se profundiza en la jerarquía. En el caso de LINUX, el manejador de dispositivos de bloque incluye operaciones genéricas como open, close, read y write para dispositivos de tipo bloque. Este manejador accede a dos módulos:

• Módulo de planificación, que se encarga de ordenar las peticiones de bloque a un dis positivo con política CSCAN.

• Manejador de disco, que se encarga de proporcionar una interfaz común a todos los dis positivos y demultiplexar las peticiones a cada tipo de dispositivo en particular.

Cuando le llega una operación de cerrar un dispositivo, el planificador mira si tiene peticiones pendientes y «encola» la petición de desconexión del dispositivo hasta que hayan terminado. Una vez procesadas las peticiones, el planificador las redirige al manejador de tipo de dispositivo ade cuado (IDE, CD-ROM, SCSI, floppy, etc.), usando para ello el major number, que se encarga de llevar a cabo las operaciones específicas para ese dispositivo y de enviarlas al dispositivo espe cífico, usando el minor number.

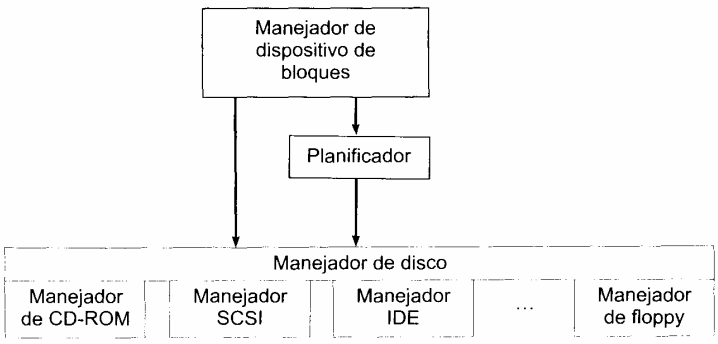


Figura 7.14. *Estructura de un manejador de disco.*

Uno de los manejadores de tipo de dispositivo existente en LINUX es el de dispositivos IDE, algunas de cuyas operaciones se indican a continuación:

```
bloc k_read,                    /* operación de lectura de bloque */
block_write,                    /* operación de escritura de bloque */
ide_octl,                       /* ioct1 */
ide_open,                       /* abrir */
ide_release,                    /* cerrar */
block_fsync,                    /* sincronización */
ide_check_media_change,         /* comprobar el medio */
revalidate_disk,                /* vuelca datos a disco */
ide_requests,                   /* llamadas del planificador */
```

Como se puede ver, el manejador proporciona muy pocas funciones de alto nivel. Además de las típicas de lectura y escritura, merece la pena resaltar las de control del dispositivo (ide_ioctl), las de sincronización (bl ock_f sync), las de comprobación del medio (ide_check_medi a_change). Internamente, el manejador IDE se estructura en dos niveles:

1. Nivel independiente de dispositivo, definido por algunas de las funciones de la interfaz.Estas funciones acaban llamando a funciones internas de bajo nivel. Una de las más un- portantes es la autodetección de los desplazamientos de bloques en el disco a partir de los datos de la tabla de particiones.
2. Nivel dependiente de dispositivo. Funciones de bajo nivel que se encargan de contactar con el controlador del dispositivo. Algunas de estas funciones son:

• ide_setup, que calcula los datos del disco a partir de la BIOS.

• controller_busy, que comprueba si el controlador está libre u ocupado.

• Status_OK, que comprueba el estado del disco.

• ide_out, que emite los mandatos al controlador para escribir en el disco.

• reset_controller y reset_hd, que reinician el controlador y el dispositivo respec tivamente.

• recalibrate, que ordena al controlador recalibrar el motor del disco.

• identify intr, que lleva a cabo el tratamiento de interrupción.

El Programa 7.2 muestra una versión simplificada de la rutina del manejador IDE que escribe datos al controlador del dispositivo. Observe que es similar al programa canal que se mostró en el Programa 7.1. Consiste básicamente en una operación de inicio del controlador para cargar el registro de datos, la dirección de memoria y el contador de datos. Un bucle que escribe los datos, incrementando el contador de memoria, y una etapa de fin de escritura.

Programa 7.2. Rutina de escritura de datos en un controlador IDE.

```
void ide_output_data
        (ide_drive_t *drive, void *buffer, unsigned int wcount)
{
         unsigned short io_base = HWIF(drive)->io_base;
         unsigned short data_reg = io_base+IDE_DATA_OFFSET;
         byte io_32bit = drive->io_32bit;
         unsigned short *ptr = (unsigned short *) buffer;

         /* Inicio de escritura */
```

Digitalización realizada con propósito académico

```
            outsl(data_reg, buffer, wcount);
            /* Escritura de datos */
            while (wcount--) {
                outw_p(*ptr++, data_reg);
                outw_p(*ptr++, data_reg);
            }
            /* Fin de escritura */
            outsw(data_reg, buffer, wcount<<1);
}
```

Planificación del disco

El rendimiento de un disco puede ser muy distinto dependiendo del orden en que reciba las peticio nes de EIS. La razón principal es que un disco, a diferencia de una tarjeta de red por ejemplo, tiene una geometría que fuerza a mover las cabezas de unas posiciones del disco a otras. Por ello es fundamental usar políticas de planificación que minimicen el movimiento de cabezas para obtener un buen rendimiento medio del disco.

El manejador de disco es el responsable de llevar a cabo la política de planificación adecuada dependiendo del tipo de petición, de la posición de las cabezas del disco y de la posición de la petición a realizar, teniendo en cuenta fundamentalmente el tiempo de búsqueda del disco. Sin embargo, es responsabilidad del diseñador elegir la política, o políticas, de planificación existentes en el sistema, para lo cual debe conocer las ventajas e inconvenientes de las más populares, algunas de las cuales se describen a continuación.

Una opción es usar la política FCFS (First Come First Served), según la cual las peticiones se sirven en orden de llegada. Esta política no permite optimizar el rendimiento del disco ni ordenar las peticiones según la geometría del disco. Para ver el comportamiento de este algoritmo, con sidere que las cabezas del disco están en el cilindro 6 y que hay encoladas peticiones para los cilindros 20, 2, 56 y 7. El número de desplazamientos necesario para servir todas las peticiones sería:

Desplazamientos = (20 — 6) + (20 — 2) + (56 — 2) + (56 — 7) = 14 + 18 + 54 + 49 = 135

Una forma de mejorar la planificación del disco es usar la política SSE (Shortest Seek First), que como su nombre indica trata de minimizar el tiempo de búsqueda sirviendo primero las pe ticiones que están más cerca de la posición actual de las cabezas. Así, en el ejemplo anterior las peticiones se atenderían en el siguiente orden: 7, 2, 20 y 56. El número de desplazamientos sería:

Desplazamientos = (7 — 6) + (7 — 2) + (20 — 2) + (56 — 20) 1 + 5 + 18 + 36 60

Sin embargo, este algoritmo tiene un problema serio: el retraso indefinido de peticiones por inanición o hambruna. Si el sistema está congestionado, o se produce alguna circunstancia de proximidad espacial de las peticiones, se da prioridad a las peticiones situadas en la misma zona del disco y se relega a las restantes. Imagine un conjunto de peticiones como las siguientes: 1, 2, 3, 56, 4, 2, 3, 1, 2,... La petición del cilindro 56 podría morir de inanición.

Sería pues conveniente buscar una política que siendo relativamente justa minimizara los movimientos de las cabezas y maximizara el rendimiento. Esta política, o políticas, es la política del ascensor con todas sus variantes. Su nombre se debe a que es la política que se usa en edificios altos para controlar eficientemente los ascensores y asegurar que todo el mundo puede subir al

Digitalización realizada con propósito académico

ascensor. El fundamento de la política básica del ascensor, denominada SCAN, consiste en mover las cabezas de un extremo a otro del disco, sirviendo todas las peticiones que van en ese sentido. Al volver se sirven todas las del sentido contrario. Con esta política, el ejemplo anterior, en el que las cabezas están en el cilindro 6 con movimiento ascendente y las peticiones encoladas son 20, 2, 56 y 7, causaría los siguientes desplazamientos, asumiendo un disco de 70 cilindros:

Desplazamientos = (7—6) + (20 —7) + (56—20) +(70 — 56)+ (56— 2)= 1 + 13 + 36 + 14 +

+54=118

Como se puede ver, el número de desplazamientos es menor que el FCFS, pero considerable mente mayor que el SSF, debido fundamentalmente al último desplazamiento que obliga a recorrer el disco entero. El comportamiento de este algoritmo es óptimo cuando las peticiones se distribuyen uniformemente por todo el disco, cosa que casi nunca ocurre. Por ello se optimizó para mejorar el tiempo de búsqueda aprovechando una propiedad de los discos que también tienen los ascensores:

es mucho más costoso subir un edificio parando que ir de un golpe de arriba abajo, o viceversa. E! ascensor de las torres Sears de Chicago, por ejemplo, sube los 110 pisos sin parar en menos de 2 minutos, mientras que parando dos o tres veces tarda más de 10. Esto es lógico, porque los motores tienen que frenar, acelerar y ajustarse al nivel de los pisos en que para el ascensor. En un disco pasa lo mismo: cuesta casi lo mismo ir de un extremo a otro que viajar entre dos pistas. Por ello se puede mejorar el tiempo de respuesta del disco usando una versión de la política del ascensor que viaje en un solo sentido. Con esta política, denominada CSCAN (Cvclic Scan), las cabezas se mueven del principio al final del disco, atienden las peticiones en ese sentido sin parar y vuelven al principio sin atender peticiones. Con esta política, el número de desplazamientos del ejemplo anterior sería:

Desplazamientos = (7—6) + (20—7) + (56—20) + (70—56) + 70 + (2—0) = 1 + 13 + 36 +

+ 14+70+2= 136

Muy alto, como puede observar. Sin embargo el resultado es engañoso, ya que los desplaza mientos realmente lentos, con paradas en cilindros, son 66. El tiempo de respuesta de este algoritmo es pues mucho menor que el del SCAN y casi similar al de SSF. En dispositivos con muchas peticiones de E/S, esta política tiene un rendimiento medio superior a las otras.

La política CSCAN todavía se puede optimizar evitando el desplazamiento hasta el final del disco, es decir, volviendo de la última petición en un sentido a la primera en el otro. En el caso del ejemplo se iría de 56 a 2 con un desplazamiento rápido. Los desplazamientos del ejemplo con esta política, denominada LOOK [1998], serían:

Desplazamientos = (7—6) + (20—7) + (56—20) + (56—2) 1 + 13 + 36 + 54 = 104

donde los desplazamientos realmente lentos son 50.

El algoritmo CSCAN es el más usado actualmente. Se usa en prácticamente todas las ver siones de UNIX, LINUX y Windows, debido a que da el mejor rendimiento en sistemas muy cargados. Sin embargo, su rendimiento depende mucho de la distribución de las operaciones de E/S por el disco y de la situación de los bloques que se acceden con más frecuencia. Por ello, suele ser una técnica habitual en sistemas operativos situar en el centro del disco aquellos datos que se acceden más a menudo.

Estos algoritmos, sin embargo, no resuelven el problema de algunas aplicaciones modernas que requieren un tiempo de respuesta determinado. Es el caso de las aplicaciones de tiempo real y multimedia, en las que es importante realizar las operaciones de E/S antes de que termine un plazo

Digitalización realizada con propósito académico

determinado (deadline). Para este tipo de sistemas se pueden usar otras técnicas de planificación, como la política EDF (Earliest Deadline First), SCAN-EDF o SCAN-RT. Para tratar de satisfacer los requisitos de sistemas de propósito general y los de multimedia se han propuesto algoritmos que optimi zan simultáneamente geometría y tiempo de respuesta, tales como Cello y 2-Q [20001.

Gestión de errores

Otra de las funciones básicas del manejador de disco es gestionar los errores de EIS producidos en el ámbito del dispositivo. Estos errores pueden provenir de:

• Las aplicaciones. Por ejemplo, petición para un dispositivo o sector que no existe.

• Del controlador. Por ejemplo, errores al aceptar peticiones o parada del controlador.

• De los dispositivos. Por ejemplo, fallos transitorios o permanentes de lectura o escritura y fallos en la búsqueda de pistas.

Algunos de ellos, como los errores transitorios, se pueden resolver en el manejador. Otros, como los errores de aplicación o fallos permanentes del dispositivo, no se pueden resolver y se deben comunicar al nivel superior de E/S.

Los errores transitorios pueden ser debidos a la existencia de partículas de polvo en la su perficie del disco cuando se efectúa la operación de E/S, a pequeñas variaciones eléctricas en la transmisión de datos o a radiaciones que afecten a la memoria del controlador, entre otras cosas. Esos errores se detectan porque el ECC de los datos no coincide con el calculado y se resuelven repitiendo la operación de E/S. Si después de un cierto número de repeticiones no se resuelve el problema, el manejador concluye que la superficie del disco está dañada y lo comunica al nivel superior de E/S. Otro tipo de errores transitorios tiene su origen en fallos de búsqueda debidos a que las cabezas del disco están mal calibradas. Se puede intentar resolver este problema recalibrando el disco, lo que puede hacerse mediante la operación del controlador reca1íbra

Los errores permanentes se tratan de distintas formas. Ante errores de aplicación el ma nejador tiene poco que hacer. Si se pide algo de un dispositivo o bloque que no existe, lo único que se puede hacer es devolver un error. Ante errores del controlador, lo único factible es tratar de reiniciar el controlador mediante sus instrucciones específicas para ello. Si al cabo de un cierto número de repeticiones no se resuelve el problema, el manejador concluye que la superficie del disco o el controlador están dañados y lo comunica al nivel superior de E/S. Ante errores permanen tes de la superficie del dispositivo, lo único que se puede hacer es sustituir el bloque por uno de repuesto, si existe una lista de bloques de repuesto, y comunicar el error al nivel superior.

7.5.3. Discos en memoria

Los discos en memoria RAM son una forma popular de optimizar el almacenamiento secundario en sistemas operativos convencionales y de proporcionar almacenamiento en sistemas operativos de tiempo real, donde las prestaciones del sistema exigen dispositivos más rápidos que un disco con vencional.

Hay dos formas básicas de proporcionar discos en memoria:

• Discos RAM.

• Discos sólidos.

Los **discos RAM** son dispositivos de bloques que proporciona el sistema operativo y que se almacenan, generalmente, en la propia memoria del sistema operativo (/dev/kmem en UNIX). Un disco RAM es una porción de memoria de un tamaño arbitrario dividida en bloques, a nivel lógico,

Digitalización realizada con propósito académico

por el sistema operativo que muestra una interfaz de disco similar al de cualquier disco de dis positivo secundario. El manejador de esos dispositivos incluye llamadas open, read, write, etc., a nivel de bloque y se encarga de traducir los accesos del sistema de archivos a posiciones de memoria dentro del disco RAM. Las operaciones de transferencia de datos son copias de memoria. No tienen ningún hardware especial asociado y se implementan de forma muy sencilla. Pero tienen un problema básico: si falla la alimentación se pierden todos los datos almacenados.

Los **discos sólidos** son sistemas de almacenamiento secundario no volátiles fabricados co locando chips de memoria RAM en placas. Las placas se conectan al bus del sistema y se acceden a la misma velocidad que la memoria principal (nanosegundos). Hay una diferencia fundamental entre estos sistemas y los discos RAM: tienen hardware, y puede que un controlador, propio. La interfaz sigue siendo la de un dispositivo de bloques, encargándose el manejador de traducir pe ticiones de bloque a memoria. Las transferencias, sin embargo, se pueden convertir en operaciones de lectura o escritura a puertos, dependiendo de la instalación del disco sólido.

Los discos en chips son la última tendencia en discos sólidos implementados con memoria. Combinan memoria flash de alta densidad y una interfaz sencilla, proyectada en memoria o a través de puertos, para proporcionar almacenamiento secundario en un chip que se parece a una memoria ROM de 32 pines. Usando estos sistemas se pueden crear discos de estado sólido con almacena miento no volátil con una capacidad de hasta 256 MB. Además, estos chips se pueden combinar en placas para proporcionar varios GB de capacidad, como por ejemplo la PCM-3840 de Advantech. Una vez instalados en el sistema, se ven como un objeto más del sistema de E/S que se manipula a través del manejador correspondiente, por lo que proporciona las mismas operaciones que cualquier otro dispositivo de almacenamiento secundario. De tal forma que cuando se crea sobre el disco un sistema de archivos, se puede montar en el árbol de nombres sin ningún problema.

7.5.4. Fiabilidad y tolerancia a fallos

El sistema de E/S es uno de los componentes del sistema con mayores exigencias de fiabilidad, debido a que se usa para almacenar datos y programas de forma permanente. Las principales téc nicas usadas para proporcionar esta fiabilidad son:

• Códigos correctores de error, como los existentes en las cabeceras y colas de los sectores.

• Operaciones fiables, cuya corrección se puede verificar antes de dar el resultado de la opera ción de EIS. Esta técnica se implementa mediante técnicas de almacenamiento estable.

• Redundancia, tanto en datos replicados [19921 como en código de paridad para detectar errores y recuperarlos. Esta técnica se implementa mediante el uso de dispositivos

RAID (Redundaní Array of Inexpensive Disks).

• Redundancia hardware, como el sistema de Windows NT que permite conectar un disco a través de dos controladores.

Todas estas técnicas contribuyen a incrementar la fiabilidad del sistema de E/S y proporcionan tolerancia a fallos para hacer frente a algunos de los errores que antes se calificaron de permanen tes. A continuación se estudian con más detalle los mecanismos de almacenamiento estable y los discos RAID.

Almacenamiento estable

La técnica clásica de almacenamiento redundante es usar discos espejo ITanenbaurn, 1992], es decir, dos discos que contienen exactamente lo mismo. Sin embargo, usar dos discos no es su-

Digitalización realizada con propósito académico

ficiente, hay que modificar el manejador, o el gestor de bloques, para tener almacenamiento fiable, de forma que las operaciones de escritura y de lectura sean fiables.

Una **escritura fiable** supone escribir en ambos discos con éxito. Para ello se escribe primero en un disco. Cuando la operación está completa, se escribe en el segundo disco. Sólo si ambas tienen éxito se considera que la operación de escritura ha tenido éxito. En caso de que una de las dos escrituras falle, se considera un error parcial. Si ambas fallan, se considera que hay fallo total del almacenamiento estable.

En el caso de la **lectura fiable**, basta con que uno de los dispositivos esté disponible. Ahora bien, si se quiere verificar el estado de los datos globales, habrá que leer ambos dispositivos y comparar los datos. En caso de error, habrá que elegir alguno de ellos según los criterios definidos por el sistema o la aplicación.

Este tipo de redundancia tiene dos problemas principales:

• Desperdicia el 50 por 100 del espacio de almacenamiento.

• Reduce mucho el rendimiento de las operaciones de escritura. Ello se debe a que una es critura no se puede confirmar como válida hasta que se ha escrito en ambos discos espejo, lo

que significa dos operaciones de escritura.

Sin embargo, tiene la ventaja de ser una técnica barata y sencilla de implementar. Además, si se lee simultáneamente de ambos discos, se puede incrementar mucho el rendimiento del sistema. En Windows NT, por ejemplo, existen manejadores para proporcionar almacenamiento estable en el ámbito del sistema y de forma transparente al usuario.

Dispositivos RAID

Una técnica más actual para proporcionar fiabilidad y tolerancia a fallos consiste en usar dispositi vos RAID (Redundant Array of Independent Disks) a nivel hardware [19951 o software EChen, 1995]. Estos dispositivos usan un conjunto de discos para almacenar la información y otro conjunto para almacenar información de paridad del conjunto anterior (Fig. 7.15). En el ámbito físico se ven como un único dispositivo, ya que existe un único controlador para todos los discos. Este controlador se encarga de reconfigurar y distribuir los datos como es necesario de forma transparente al sistema de FIS.

Se han descrito hasta siete niveles de RAID, pero solamente los cinco primeros están realmen te operativos. Estos niveles son los siguientes:

• RAID 1. Son discos espejo en los cuales se tiene la información duplicada. Tiene los pro blemas y las ventajas del almacenamiento estable.

• RAID 2. Distribuye los datos por los discos, repartiéndolos de acuerdo con una unidad de distribución definida por el sistema o la aplicación. El grupo de discos se usa como un disco lógico, en el que se almacenan bloques lógicos distribuidos según la unidad de reparto.

• RAID 3. Reparte los datos a nivel de bit por todos los discos. Se puede añadir bits con códigos correctores de error. Este dispositivo exige que las cabezas de todos los discos estén

sincronizadas, es decir, que un único controlador controle sus movimientos.

• RAID 4. Reparto de bloques y cálculo de paridad para cada franja de bloques, que se al macena en un disco fijo. En un grupo de cinco discos, por ejemplo, los cuatro primeros serían de datos y el quinto de paridad. Este arreglo tiene el problema de que el disco de paridad se convierte en un cuello de botella y un punto de fallo único.

Digitalización realizada con propósito académico

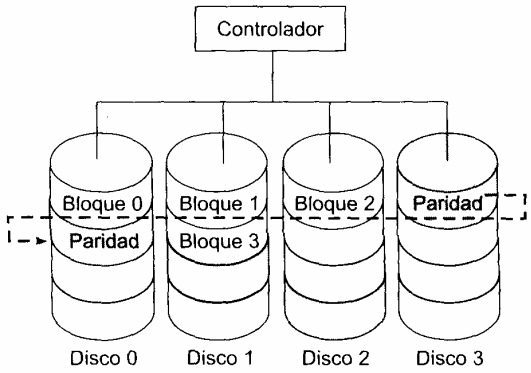


Figura 7.15. *Distribución de bloques en un dispositivo RAID 5.*

• RAID 5. Reparto de bloques y paridad por todos los discos de forma cíclica. Tiene la ventaja de la tolerancia a fallos sin los inconvenientes del RAID 4. Actualmente existen múltiples

dispositivos comerciales de este estilo y son muy populares en

fiabilidad.

La Figura 7.15 muestra un dispositivo de tipo RAID 5, donde toda la paridad se reparte por todos los discos de forma cíclica. La paridad se calcula por hardware en el controlador haciendo el X-OR de los bloques de datos de cada franja. Esto conlieva problemas si las escrituras no llenan todos los discos de datos (escrituras pequeñas), ya que hay que leer los datos restantes para calcular la paridad. Sin embargo, los RAID proporcionan un gran ancho de banda para lecturas y escrituras grandes (Prestaciones 7.1).

PRESTACIONES 7.1

Actualmente los RAID son los dispositivos de elección en un sistema que requiera fiabilidad y altas prestaciones. Sus controladores están muy optimizados y permiten convertir la información de unos formatos de RAID a otros de forma transparente al usuario para optimizar su rendimiento. Eso es lo que hace, por ejemplo, el HP-AUTORAID [Wilkes, 1995], que permite cambiar los archivos de RAID 1 a RAID 5 de forma automática o bajo demanda del sistema de E/S.

7.6. ALMACENAMIENTO TERCIARIO

Los sistemas de almacenamiento secundario están bien para acceso rápido a datos y programas, sin embargo tienen tres problemas serios:

1. Poca capacidad.

2. Alto coste.

3. No se pueden extraer de la computadora.

Digitalización realizada con propósito académico

En algunos sistemas de almacenamiento es necesario disponer de dispositivos extraíbles y de alta capacidad para poder hacer copias de respaldo de datos o para archivar datos que se usan poco frecuentemente. Estas dos necesidades justifican la existencia de almacenamiento terciario, que se puede definir como un sistema de almacenamiento de alta capacidad, bajo coste y con dispositivos extraíbles en e] que se almacenan los datos que no se necesitan de forma inmediata en el sistema.

En esta sección se estudia brevemente la tecnología para sistemas de almacenamiento ter ciario, la estructura y los componentes del sistema de almacenamiento terciario y se estudia el sistema HPSS, un sistema de almacenamiento terciario de altas prestaciones.

7.6.1. Tecnología para almacenamiento terciario

La **tecnología** de almacenamiento terciario no ha evolucionado mucho en los últimos años. Los dispositivos de elección son los CD-ROM, los DVD y, sobre todo, las cintas magnéticas. En cuanto al soporte usado, se usan jukeboxes y sistemas robotizados para las cintas.

Los jukeboxes son torres de CD-ROM o DVD similares a las máquinas de música que se ven en algunos bares y salas de baile. Cuando se quiere leer algo, las cabezas lectoras suben o bajan por la torre, seleccionan el disco adecuado y lo leen. Este método es adecuado para bases de datos medianas o centros de distribución de música o de películas. Sin embargo, estos dispositivos se usan principalmente como discos WORM (Write Once ReadMany), lo que significa que es difícil o costoso rescribir información sobre ellos. Además, tienen poca capacidad, ya que un CD-ROM almacena 700 MB y un DVD unos 18 GB.

Hasta ahora, la alternativa como sistema de almacenamiento son las cintas magnéticas, que son más baratas, igual de rápidas y tienen más capacidad (Tabla 7.1). Estas cintas se pueden mani pular de forma manual o mediante robots que, en base a códigos de barras, las almacenan una vez escritas y las colocan en las cabezas lectoras cuando es necesario.

Un buen ejemplo de sistema de almacenamiento terciario lo constituye el existente en el CERN (Centro Europeo para la Investigación Nuclear), donde los datos de los experimentos se almacenan en cintas magnéticas que controlan un sistema de almacenamiento controlado por un robot. Cuando se va a procesar un experimento, sus datos se cargan en el almacenamiento se cundario. La gran ventaja de este sistema es que el un único robot puede manipular un gran número

Tabla 7.1. *Evolución de las cintas magnéticas*

Unidad	DLT 2000	3590	SD-3	3570	9490	9840	3590E
Fecha	*1993*	*1995*	*1995*	1997	1998	1998	1999
Capacidad (MB)	10.000	10.000	50.000	5.000	1.600	20.000	40.000
Velocidad transferencia (MB/sec)	1,2	9	11	7	6	10	14
Longitud (pies)	1.200	1.100	1.200	550	2.200	900	2.200
Pistas	128	128	Helicoidal	128	36	288	256

Digitalización realizada con propósito académico

de cintas con pocas unidades lectoras. Su gran desventaja es que los datos no están inmediatamente disponibles para los usuarios, que pueden tener que esperar segundos u horas hasta que se instalan en el sistema de almacenamiento secundario.

Una tecnología novedosa para el almacenamiento secundario y terciario la constituyen las SAN (Storage Area Networks), que son redes de altas prestaciones a las que se conectan directa mente dispositivos de almacenamiento. Presentan una interfaz uniforme para todos los dispositivos y proporcionan un gran ancho de banda. Actualmente empiezan a ser dispositivos de elección para sistemas de almacenamiento secundarios y terciarios de altas prestaciones.

En sistemas de altas prestaciones, como el descrito más adelante, se usan servidores de EIS

muy complejos, incluidos multiprocesadores, para crear sistemas de almacenamiento terciario cuyo tiempo de respuesta es muy similar a los sistemas de almacenamiento secundario.

7.6.2. Estructura y componentes de un sistema de almacenamiento terciario

Desde el punto de vista del sistema operativo, el principal problema de estos sistemas de almacena miento terciario es que se crea una jerarquía de almacenamiento secundario y terciario, por la que deben viajar los objetos de forma automática y transparente al usuario. Además, para que los usua rios no estén descontentos, se debe proporcionar accesos con el mismo rendimiento del nivel más rápido y el coste aproximado del nivel más bajo.

La existencia de un sistema de almacenamiento terciario obliga al diseñador del sistema opera tivo a tomar cuatro decisiones básicas relacionadas con este sistema:

• ¿Qué estructura de sistema de almacenamiento terciario es necesaria?

• ¿Cómo, cuándo y dónde se mueven los archivos del almacenamiento secundario al terciario?

• ¿Cómo se localiza un archivo en el sistema de almacenamiento terciario?

• ¿Qué interfaz de usuario va a estar disponible para manejar el sistema de almacenamiento terciario?

Los sistemas de almacenamiento terciario existentes en los sistemas operativos actuales, tales como UNIX, LINUX y Windows, tienen una estructura muy sencilla consistente en un nivel de dispositivos extraíbles, tales como cintas, disquetes o CD-ROM. Estos dispositivos se tratan exacta mente igual que los secundarios, exceptuando el hecho de que los usuarios o administradores deben insertar los dispositivos antes de trabajar con ellos. Solamente en grandes centros de datos se pue den encontrar actualmente sistemas de almacenamiento terciario con una estructura compleja, como la de HPSS que se estudia en la sección siguiente.

La cuestión de la **migración de archivos** del sistema secundario al terciario, y viceversa, depende actualmente de las decisiones del administrador (en el caso de las copias de respaldo) o de los propios usuarios (en el caso de archivos personales). En los sistemas operativos actuales no se define ninguna política de migración a sistemas terciarios, dado que estos sistemas pueden no existir en muchas computadoras. Sin embargo, se recomienda a los administradores tener una plani ficación de migración para asegurar la integridad de los datos del sistema secundario y para migrar aquellos archivos que no se han usado durante un período largo de tiempo.

La **política de migración** define las condiciones bajo las que se copian los datos de un nivel de la jerarquía a otro, u otros. En cada nivel de la jerarquía se puede aplicar una política distinta. Además, en el mismo nivel se puede aplicar una política distinta dependiendo de los datos a migrar. Imagine que se quiere mover un archivo de usuario cuya cuenta de acceso a la computadora se ha cerrado. Las posibilidades de volver a acceder a estos datos son pequeñas, por lo que lo más proba

Digitalización realizada con propósito académico

ble es que la política los relegue al último nivel de la jerarquía de EIS (p. ej.: cintas extraíbles). Imagine ahora que se quiere migrar un archivo de datos del sistema que se accede con cierta frecuencia, lo más probable es que la política de migración decida copiarlo a un dispositivo de alta integridad y disponibilidad, como un RAID. En los sistemas automatizados, como HPSS, hay mo nitores que rastrean los archivos del sistema de almacenamiento secundario y crean estadísticas de uso de los mismos. Habitualmente, los archivos usados son los primeros candidatos para la política de migración.

Cómo y dónde migrar los archivos depende mucho de la estructura del sistema de almacena miento terciario. Existen dos opciones claras para mantener los archivos en almacenamiento terciario:

• Usar dispositivos extraíbles y mantenerlos o/j-line.

• Usar dispositivos extraíbles o no, pero mantenerlos en-une.

En el primer caso, los archivos se copian en el sistema terciario y se eliminan totalmente del secundario. En el segundo se copian en el sistema terciario pero se mantienen enlaces simbólicos desde el sistema secundario, de forma que los archivos estén automáticamente accesibles cuando se abren. Un caso especial lo constituyen las copias de respaldo, en las que se copian los datos a un terciario extraíble por cuestiones de seguridad y no de prestaciones. En este caso, los datos siguen estando completos en el sistema secundario de almacenamiento. Con cualquiera de las opciones, es habitual comprimir los datos antes de copiarlos al sistema terciario, para aumentar la capacidad del mismo. La compresión se puede hacer transparente a los usuarios incorporando herramientas de compresión al proceso de migración.

La **localización de archivos** en un sistema de almacenamiento terciario es un aspecto muy importante de los mismos. En un sistema operativo de una instalación informática convencional, lo normal es que el usuario deba preguntar al administrador del sistema por ese archivo si éste lo ha migrado a un sistema terciario extraíble, ya que no estará en el almacenamiento secundario. En el caso de que el sistema terciario sea capaz de montar dinámicamente los dispositivos extraíbles (robots de cintas,jukehoxes, etc.), se pueden mantener enlaces desde el sistema secundario al ter ciario, por lo que se migrará el archivo automáticamente (en cuestión de segundos). Un sistema terciario de altas prestaciones como HPSS intenta que los usuarios no puedan distinguir si los archivos están en el sistema secundario o terciario, tanto por la transparencia de nombres COfl1() por el rendimiento. Para ello usa servidores de nombres, localizadores de archivos y, si es necesario, bases de datos para localizar los archivos.

La **interfaz de usuario** de los sistemas de almacenamiento terciario puede ser la misma que la del sistema secundario, es decir, las llamadas al sistema de POSIX o Win32 para manejar dispositi vos de EIS. Sin embargo, es habitual que los sistemas operativos incluyan mandatos, tales conio el tar de UNIX o el backup de Windows, para llevar a cabo copias de archivos en dispositivos extraíbles de sistema terciario. Por ejemplo:

```
tar cvf /usr/jesus /dev/rmtO
```

Hace una copia de los datos del usuario jesús al dispositivo de cinta rrntO de una computado ra UNIX. Igualmente:

```
tar cvf / /dev/rmtO
```

Hace una copia de todos los sistemas de archivos de una computadora UNIX al dispositivo de cinta rmt0.

Digitalización realizada con propósito académico

Estos mandatos permiten hacer copias totales de tos datos o parciales, es decir, copiar sólo los archivos modificados desde la copia anterior. En los sistemas de almacenamiento terciario de alto rendimiento, como HPSS, se encuentran interfaces muy sofisticadas para configurar y manipular los datos, como se muestra a continuación.

7.6.3. Estudio de caso: sistema de almacenamiento de altas prestaciones (HPSS)

El sistema de almacenamiento de altas prestaciones (HPSS, High Performance Storage Svstern) [19991 es un sistema de E/S diseñado para proporcionar gestión de almacenamiento jerár quico, secundario y terciario, y servicios para entornos con necesidades de almacenamiento me dianas y grandes. Está pensado para ser instalado en entornos de EIS con alto potencial de creci miento y muy exigentes en términos de capacidad, tamaño de archivos, velocidad de transferencia, número de objetos almacenados y número de usuarios. Se distribuye como parte de IJ(I (Lfls trihuted Computing Environment).

Uno de los objetivos de diseño primarios de HPSS fue mover los archivos grandes de dis positivos de almacenamiento secundario a sistemas terciarios, que pueden ser incluso sistemas de E/S paralelos, a velocidades mucho más altas que los sistemas de archivo tradicionales (cintas. jukehoxes. etc.), de forma más fiable y totalmente transparente al usuario. El resultado del diseño fue un sistema de almacenamiento de altas prestaciones con las siguientes caracterísi cas (Fig. 7.16):

• Arquitectura distribuida a través de redes, siempre que es posible, para proporcionar escala bilidad y paralelismo.

• Jerarquías múltiples de dispositivos que permiten incorporar todo tipo de dispositivos ac tuales (RAID, SAN, jukebox, cintas, etc.) y experimentar con medios de almacenamiento nuevos. Los dispositivos se agrupan en clases, cada una de las cuales tiene sus propios servicios. Existen servicios de migración entre clases para mover archivos de una clase de dispositivo a otra.

• Transferencia de datos directa entre los clientes y los dispositivos de almacenamiento. sin copias intermedias de memoria. Para ello se usan reubicadores de archivos que se encargan

de las transferencias de archivos de forma distribuida.

• Velocidad de transferencia muy alta, debido a que los reubicadores extraen los datos directa mente de los dispositivos y los transmiten a la velocidad de éstos (p. ej.: 5() MB/segundo

para algunos RAID).

• Interfaz de programación con acceso paralelo y secuencial a los dispositivos de E/S. Diseño interno multithread que usa un protocolo FTP (File Tran,sfer Protocol) paralelo.

• Uso de componentes estandarizados que le permiten ejecutar sobre POSIX sin ninguna mo- dificación. Está escrito en ANSI C e incorpora varios estándares del IEEE para sistemas de

almacenamiento.

• Uso de transacciones para asegurar la integridad de los datos y de Kerberos (Capítulo 9) como sistema de seguridad.

Estructura de HPSS

La Figura 7. 16 muestra una estructura muy simplificada de HPSS. Básicamente se pueden distin guir cinco módulos principales en el sistema de almacenamiento terciario:

Digitalización realizada con propósito académico

• **Interfaz.** Utilidades para administrar y controlar el sistema de almacenamiento y para de finir políticas de actuación.

• **Servidor de nombres**. Traduce nombres de usuario a identificadores de objetos HPSS, tales como archivos o directorios. También proporciona control de acceso a los objetos.

• **Localizador de archivos.** Una vez resuelto el nombre, este módulo se encarga de localizar el archivo físicamente mediante los descriptores proporcionados por el sistema de nombres.

• **Reubicador de archivos.** Se encarga de la transferencia de datos desde el origen al destino. Interacciona con el servidor de nombres y el localizador de archivos para determinar si el origen y el destino son correctos. Usa un modelo de trabajadores, en el que se crea un trabajador para cada transferencia de datos, que realiza directamente entre el dispositivo origen y el destino.

•**Gestor del sistema de almacenamiento terciario**. Proporciona la jerarquía de objetos de almacenamiento, traduce las peticiones de la interfaz de usuario a sus correspondientes dis positivos físicos y proporciona toda la funcionalidad de gestión del sistema de E/S, incluyendo instalación, configuración, terminación y accesos a dispositivos. Incluye también facilidades de monitorización que permiten conocer la situación del sistema de almacenamiento y la

dpon de recursos en las d clases de d Por ú incluye fac dades de acceso a los datos.

El sistema de almacenamiento físico incluye volúmenes físicos y virtuales que pueden estar en

cualquier tipo de soporte (discos, RAID, cintas, robots de cintas, jukeboxes, SAN, etc.). Los vo lúmenes virtuales son internos a HPSS y se forman agrupando varios volúmenes físicos bajo una

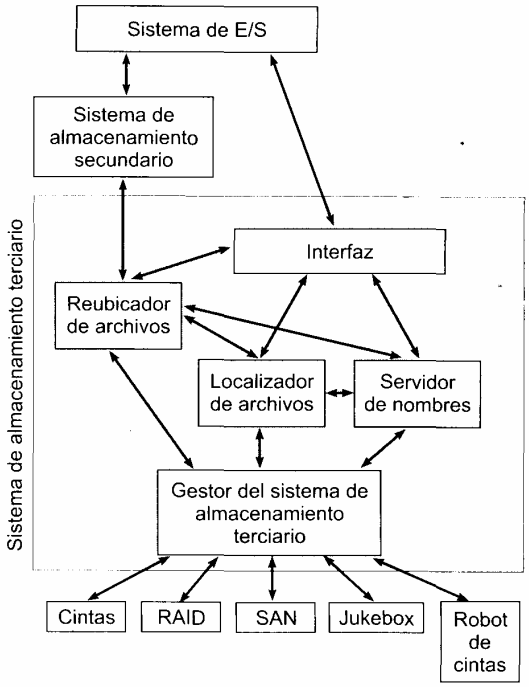


Figura 7.16. *Estructura del sistema de almacenamiento terciario de HPSS.*

misma entidad, sobre la cual se reparten los datos. Sobre ellos se proyectan los segmentos de almacenamiento, entidades asociadas a una clase de almacenamiento y con localización de archivos transparentes. Estas clases de almacenamiento se organizan en jerarquías a través de las cuales pueden migrar los archivos de acuerdo con la disponibilidad de recursos y la política de migración definida.

Interfaz de usuario de HPSS

HPSS es un sistema complejo que proporciona varias interfaces de usuario, si bien la interfaz primaria es la basada en el estándar POSIX, para facilitar el uso de aplicaciones ópo 1 para explotar la nueva funcionalidad de HPSS se han añadido bibliotecas nuevas que permiten optimizar la gestión y el uso de los sistemas de almacenamiento que usan HPSS.

Además de esta inter de usuario, 1-WSS p1opoTcwrn \a

• Acceso remoto a través del protocolo FTP estándar y una versión paralela PFTP.

• Acceso a dispositivos remotos a través de NFS.

• Acceso paralelo a través del sistema de archivos paralelo PIOFS.

• Acceso a través de la parte de E/S del estándar MP! (MP!-IO).

En conjunto los usuarios de HPSS disponen de una amplia funcionalidad para instalar y operar

un sistema de almacenamiento secundario y terciario de altas prestaciones.

7.7. EL RELOJ

El reloj es un elemento imprescindible en cualquier sistema informático. Es necesario aclarar desde el principio que se trata de un término que presenta varias acepciones en este entorno:

- El reloj del procesador, que marca el ritmo con el que se ejecutan las instrucciones.
- El reloj del sistema, que mantiene la fecha y la hora en el mismo.
- El reloj temporizador, que hace que el sistema operativo se active periódicamente para realizar las labores correspondientes.

La primera acepción queda fuera de esta presentación ya que no involucra al sistema operati vo, consistiendo generalmente en una señal generada por un cristal de cuarzo que alimenta directa mente el procesador. Este apartado se centra en los dos úhiinos sentidos del término ya que en ellos sí que interviene el sistema operativo.

El estudio del reloj se ha englobado en este capítulo puesto que presenta características si av a 1 d.sp' d e vttás c ks s' pv .rn normalmente escribir en los registros de entradalsalida y usa interrupciones para notificar sus even tos. Sin embargo, la mayoría de los sistemas operativos no lo consideran como un dispositivo más, sino que le dan un tratamiento específico.

En esta sección se presentarán en primer lugar los aspectos hardware del reloj para pasar a continuación a analizar las cuestiones relacionadas con el software.

7.7.1. El hardware del reloj

Para medir el tiempo sólo se requiere un componente que genere una señal periódica que sirva como base de tiempo. Normalmente se dispone de un circuito temporizador que, a partir de las

Digitalización realizada con propósito académico

oscilaciones producidas por un cristal de cuarzo, genera periódicamente interrupciones (a cada una de estas interrupciones del reloj se las suele denominar en inglés tick). Este elemento está conectado generalmente a una línea de interrupción de alta prioridad del procesador debido a la importancia de los eventos que produce.

La mayoría de estos temporizadores permiten programar su frecuencia de interrupción. Típi camente contienen un registro interno que actúa como un contador que se decrementa por cada oscilación del cristal generando una interrupción cuando llega a cero. El valor que se carga en este contador determina la frecuencia de interrupción del temporizador, que se comporta, por tanto, como un divisor de frecuencia.

Además de ser programable la frecuencia de interrupción, estos dispositivos frecuentemente presentan varios modos de operación seleccionables. Suelen poseer un modo de «un solo disparo» en el que cuando el contador llega a cero y se genera la interrupción, el temporizador se desactiva hasta que se le reprograme cargando un nuevo valor en el contador. Otro típico modo de operación es el de «onda cuadrada» en el que al llegar a cero el contador y generar la interrupción el propio temporizador vuelve a recargar el valor original en el contador sin ninguna intervención externa.

Hay que resaltar que generalmente un circuito de este tipo incluye varios temporizadores independientes, lo que permite, en principio, que el sistema operativo lo use para diferentes labores. Sin embargo, en muchos casos no todos ellos son utilizables para esta misión ya que no están conectados a líneas de interrupción del procesador.

Un ejemplo típico de este componente es el circuito temporizador 8253 que forma parte de un PC estándar. Tiene tres temporizadores pero sólo uno de ellos está conectado a una línea de in terrupción (en concreto, a la línea IRQ número O). Los otros están destinados a otros usos. Por ejemplo, uno de ellos está conectado al altavoz del equipo. La frecuencia de entrada del circuito es de 1,193 MHz. Dado que utiliza un contador interno de 16 bits, el temporizador se puede programar en el rango de frecuencias desde 18,5 Hz (con el contador igual a 65536) hasta 1,193 MHz (con el contador igual a 1). Cada temporizador puede programarse de forma independiente presentando diversos modos de operación, entre ellos, el de «un solo disparo» y el de «onda cuadrada».

Otro elemento hardware presente en prácticamente la totalidad de los equipos actuales es un reloj alimentado por una batería (denominado en ocasiones reloj CMOS) que mantiene la fecha y la hora cuando la máquina está apagada.

7.7.2. El software del reloj

Dado que la labor fundamental del hardware del reloj es la generación periódica de interrupciones, el trabajo principal de la parte del sistema operativo que se encarga del reloj es el manejo de estas interrupciones. Puesto que, como se analizará a continuación, las operaciones asociadas al trata miento de una interrupción de reloj pueden implicar un trabajo considerable, es preciso establecer un compromiso a la hora de fijar la frecuencia de interrupción del reloj de manera que se consiga una precisión aceptable en la medición del tiempo, pero manteniendo la sobrecarga debida al trata miento de las interrupciones en unos términos razonables. Un valor típico de frecuencia de interrup ción, usado en muchos sistemas UNIX, es de 100 Hz (o sea, una interrupción cada 10 ms).

Hay que resaltar que, dado que las interrupciones del reloj son de alta prioridad, mientras se procesan no se aceptan interrupciones de otros dispositivos menos prioritarios. Se debe, por tanto. minimizar la duración de la rutina de tratamiento para asegurar que no se pierden interrupciones de menor prioridad debido a la ocurrencia de una segunda interrupción de un dispositivo antes de que se hubiera tratado la primera. Para evitar esta posibilidad, muchos sistemas operativos dividen el trabajo asociado con una interrupción de reloj en dos partes:

Digitalización realizada con propósito académico

- Operaciones más urgentes que se ejecutan en el ámbito de la rutina de interrupción.
- Operaciones menos urgentes que lleva a cabo una rutina que ejecuta con una prioridad más baja que cualquier dispositivo del sistema. Esta función es activada por la propia rutina de interrupción de reloj mediante un mecanismo generalmente denominado interrupción software (en el caso de Linux se denomina «mitad inferior»).

Aunque la labor principal del sistema operativo con respecto al manejo del reloj es ci trata miento de sus interrupciones, hay que hacer notar que también debe realizar su iniciación y llevar a cabo las llamadas al sistema relacionadas con el mismo.

Con independencia de cual sea el sistema operativo específico, se pueden identificar las si guientes operaciones como las funciones principales del software de manejo del reloj:

- Mantenimiento de la fecha y de la hora.
- Gestión de temporizadores.
- Contabilidad y estadísticas.
- Soporte para la planificación de procesos.

Mantenimiento de la fecha y de la hora

En el arranque del equipo, el sistema operativo debe programar el circuito temporizador del sistema cargándole en su contador interno el valor correspondiente a la frecuencia deseada y estableciendo un modo de operación de «onda cuadrada», si es que el temporizador dispone del mismo. Asimis mo, debe leer el reloj mantenido por una batería para obtener la fecha y hora actual.

A partir de ese momento, el sistema operativo se encargará de actualizar la hora según se vayan produciendo las interrupciones.

La principal cuestión referente a este tema es cómo se almacena internamente la información de la fecha y la hora. En la mayoría de los sistemas la fecha se representa como el número de unidades de tiempo transcurridas desde una determina fecha en el pasado. Por ejemplo, los sistemas UNIX toman como fecha base el 1 de enero de 1970. Algunos de ellos almacenan el número de segundos transcurridos desde esta fecha, mientras que otros guardan el número de microsegundos que han pasado desde entonces. Por lo que se refiere a Windows, la fecha se almacena Corno el número de centenas de nanosegundos (l0- segundos) transcurridos desde la fecha de referencia del Ide enero de 1601.

Sea cual sea la información almacenada para mantener la fecha y la hora, es muy importante que se le dedique un espacio de almacenamiento suficiente para que se puedan representar fechas en un futuro a medio o incluso a largo plazo. De esta forma se asegura que la vida del sistema operativo no quede limitada por este aspecto. Así, por ejemplo, un sistema UNIX que use una variable de 32 bits para guardar el número de segundos desde la fecha de referencia (1- 1-1970) permitiría representar fechas hasta principios del siglo XXII.

Otro aspecto importante es cómo tratar las peculiaridades existentes en el horario de cada país. Esto no sólo implica las diferencias que hay dependiendo del huso horario al que pertenezca el país, sino también la existencia en algunos países de políticas de cambio de horario con el objetivo de ahorrar energía. En el caso del sistema UNIX, el sistema operativo almacena la hora en el sistema de tiempo estándar UTC (Universal Ceordinated Time), con independencia de las peculiaridades del país donde reside la máquina. La conversión al horario local no la realiza el sistema operativo sino las bibliotecas del sistema.

Por último, es interesante resaltar que algunos sistemas operativos, además de un servicio para cambiar la hora en el sistema de forma inmediata, ofrecen un servicio que permite hacer este ajuste de forma de progresiva, para así no perturbar bruscamente el estado del sistema. Observe que un cambio inmediato de la hora que implique un retraso de la misma puede causar situaciones proble máticas en el sistema (evidentemente, el tiempo real nunca va hacia atrás).

Digitalización realizada con propósito académico

Gestión de temporizadores

En numerosas ocasiones un programa necesita esperar un cierto plazo de tiempo antes de realizar una determinada acción. El sistema operativo ofrece servicios que permiten a los programas es tablecer temporizaciones y se encarga de notificarles cuando se cumple el plazo especificado (en caso de UNIX mediante una señal). Pero no sólo los programas de usuario necesitan este meca nismo, el propio sistema operativo también lo requiere. El módulo de comunicaciones del sistema operativo, por ejemplo, requiere establecer plazos de tiempo para poder detectar si un mensaje se pierde. Otro ejemplo es el manejador del disquete que, una vez arrancado el motor del mismo, requiere esperar un determinado tiempo para que la velocidad de rotación se estabilice antes de poder acceder al dispositivo.

Dado que en la mayoría de los equipos existe un único temporizador (o un número muy reducido de ellos), es necesario que el sistema operativo lo gestione de manera que sobre él puedan crearse los múltiples temporizadores que puedan requerirse en el sistema en un determinado momento.

El sistema operativo maneja generalmente de manera integrada tanto los temporizadores de los procesos de usuario como los internos. Para ello mantiene una lista de temporizadores activos. En cada elemento de la lista se almacena típicamente el número de unidades de tiempo (para facilitar el trabajo, generalmente se almacena el número de interrupciones de reloj requeridas) que falta para que se cumpla el plazo y la función que se invocará cuando éste finalice. Así, en el ejemplo del disquete se corresponderá con una función del manejador de dicho dispositivo. En el caso de una temporización de un programa de usuario, la función corresponderá con una rutina del sistema operativo encargada de mandar la notificación al proceso (en UNIX, la señal).

Existen diversas alternativas a la hora de gestionar la lista de temporizadores. Una organiza ción típica consiste en ordenar la lista de forma creciente según el tiempo que queda por cumplirse el plazo de cada temporizador. Además, para reducir la gestión asociada a cada interrupción, el plazo de cada temporizador se guarda de forma relativa a los anteriores en la lista. Así se almacenan las unidades que quedarán pendientes cuando se hayan cumplido todas las temporizaciones corres pondientes a elementos anteriores de la lista. Con este esquema se complica la inserción, puesto que es necesario buscar la posición correspondiente en la lista y reajustar el plazo del elemento situado a

continuación de la posición de inserción. Sin embargo, se agiliza el tratamiento de la interrupción 1 7.8 ya que sólo hay que modificar el elemento de cabeza, en lugar de tener que actualizar todos los

temporizadores.

En último lugar hay que resaltar que generalmente la gestión de temporizadores es una de las operaciones que no se ejecutan directamente dentro de la rutina de interrupción sino, como se comentó previamente, en una rutina de menor prioridad. Esto se debe a que esta operación puede conllevar un tiempo considerable por la posible ejecución de las rutinas asociadas a todos aquellos temporizadores que se han cumplido en la interrupción de reloj en curso.

Contabilidad y estadísticas

Puesto que la rutina de interrupción se ejecuta periódicamente, desde ella se puede realizar un muestreo de diversos aspectos del estado del sistema llevando a cabo funciones de contabilidad y estadística. Es necesario resaltar que, dado que se trata de un muestreo del comportamiento de una determinada variable y no de un seguimiento exhaustivo de la misma, los resultados no son exactos, aunque si la frecuencia de interrupción es suficientemente alta, pueden considerarse aceptables.

Dos de las funciones de este tipo presentes en la mayoría de los sistemas operativos son las siguientes:

• Contabilidad del uso del procesador por parte de cada proceso.

• Obtención de perfiles de ejecución.

Digitalización realizada con propósito académico

Por lo que se refiere a la primera función, en cada interrupción se detecta qué proceso está ejecutando y a éste se le carga el uso del procesador en ese intervalo. Generalmente, el sistema operativo distingue a la hora de realizar esta contabilidad si el proceso estaba ejecutando en modo usuario o en modo sistema. Algunos sistemas operativos utilizan también esta información para implementar temporizadores virtuales, en los que el reloj sólo «corre» cuando está ejecutando el proceso en cuestión, o para poder establecer límites en el uso del procesador.

Con respecto a los perfiles, se trata de obtener información sobre la ejecución de un programa que permita determinar cuánto tiempo tarda en ejedutarse cada parte del mismo. Esta información permite que el programador detecte los posibles cuellos de botella del programa para poder así optimizar su ejecución. Cuando un proceso tiene activada esta opción, el sistema operativo toma una muestra del valor del contador de programa del proceso cada vez que una interrupción en $ cuentra que ese proceso estaba ejecutando.

La acumulación de esta información durante toda la ejecución del proceso permite que el sistema operativo obtenga una especie de histograma de las direcciones de las instrucciones que ejecuta el programa.

Soporte a la planificación de procesos

La mayoría de los algoritmos de planificación de procesos tienen en cuenta de una forma u otra el tiempo y, por tanto, implican la ejecución de ciertas acciones de planificación dentro de la rutina de interrupción.

En el caso de un algoritmo round-robin, en cada interrupción de reloj se le descuenta el tiempo correspondiente a la rodaja asignada al proceso. Cuando se produce la interrupción de reloj que consume la rodaja, se realiza la replanificación.

Otros algoritmos requieren que cada cierto tiempo se recalcule la prioridad de los procesos teniendo en cuenta el uso del procesador en el último intervalo. Nuevamente, estas acciones estarán

asociadas con la interrupción de reloj.

7.8. EL TERMINAL

Se trata de un dispositivo que permite al usuario comunicarse con el sistema y que está presente en todos los sistemas de propósito general actuales. Está formado típicamente por un teclado que permite introducir información y una pantalla que posibilita su visualización.

Hay una gran variedad de dispositivos de este tipo, aunque en esta sección se analizan los dos más típicos: los terminales serie y los proyectados en memoria. Por lo que se refiere al tipo de información usada por el terminal, esta exposición se centra en la información de tipo texto, de jando fuera de la misma el tratamiento de información gráfica, puesto que la mayoría de los sis temas operativos no dan soporte a la misma.

En primer lugar se expondrá cómo es el modo de operación básico de un terminal con indepen dencia del tipo del mismo. A continuación, se analizarán las características hardware de los termi nales. Por último, se estudiarán los aspectos software identificando las labores típicas de un maneja dor de terminal.

7.8.1. Modo de operación del terminal

El modo de operación básico de todos los terminales es bastante similar a pesar de su gran diversi dad. La principal diferencia está en qué operaciones se realizan por hardware y cuáles por software. En todos ellos existe una relativa independencia entre la entrada y salida.

Digitalización realizada con propósito académico

Entrada

Cuando el usuario pulsa una tecla en un terminal, se genera un código de tecla que la identifica. Este código de tecla debe convertirse en el carácter ASCII correspondiente teniendo en cuenta el estado de las teclas modificadoras (típicamente, Control, Shift y Alt). Así, por ejemplo, si está pulsada la tecla Shift al teclear la letra "a", el carácter resultante será "A".

Salida

Una pantalla de vídeo está formada por una matriz de pixels. Asociada a la misma existe una memoria de vídeo que contiene información que se visualiza en la pantalla. El controlador de vídeo es el encargado de leer la información almacenada en dicha memoria y usarla para refrescar el contenido de la pantalla con la frecuencia correspondiente. Para escribir algo en una determinada posición de la pantalla sólo es necesario modificar las direcciones de memoria de vídeo correspon dientes a esa posición.

Cuando un programa solicita escribir un determinado carácter ASCII en la pantalla, se debe obtener el patrón rectangular que representa la forma de dicho carácter, lo que dependerá del tipo de fuente de texto utilizado. El controlador visualizará dicho patrón en la posición correspondiente de la pantalla.

Además de escribir caracteres, un programa necesita realizar otro tipo de operaciones, tales

como borrar la pantalla o mover el cursor a una nueva posición. Este tipo de operaciones están generalmente asociadas a ciertas secuencias de caracteres.

Cuando un programa escribe una de estas secuencias, no se visualiza información en la pan talla sino que se lleva a cabo la operación de control asociada a dicha secuencia. Típicamente, por razones históricas, estas secuencias suelen empezar por el carácter Escape.

7.8.2. Hardware del terminal

Como se comentó previamente, se van a considerar dos tipos de terminales: terminales proyectados en memoria y terminales serie.

Terminales proyectados en memoria

Como se puede apreciar en la Figura 7.17, un terminal de este tipo está formado realmente por dos dispositivos totalmente independientes: el teclado y la pantalla.

El teclado genera una interrupción cuando se aprieta una tecla (en algunos sistemas también se genera cuando se suelta). Cuando se produce la interrupción, el código de la tecla pulsada queda almacenado en un registro de entradalsalida del controlador del teclado. Observe que tanto la con versión desde el código de tecla hasta el carácter ASCII como el tratamiento de las teclas modifica doras los debe realizar el software.

En este tipo de terminales, la memoria de vídeo está directamente accesible al procesador. Por tanto, la presentación de información en este tipo de terminales implica solamente la escritura del dato que se pretende visualizar en las posiciones correspondientes de la memoria de vídeo, no requiriéndose el uso de interrupciones para llevar a cabo la operación.

Con respecto a la información que se escribe en la memoria de vídeo, va a depender de si el

modo de operación del terminal es alfanumérico o gráfico.

En el modo alfanumérico se considera la pantalla como una matriz de caracteres, por lo que la

memoria de vídeo contiene el código ASCII de cada carácter presente en la pantalla. Durante una

Digitalización realizada con propósito académico

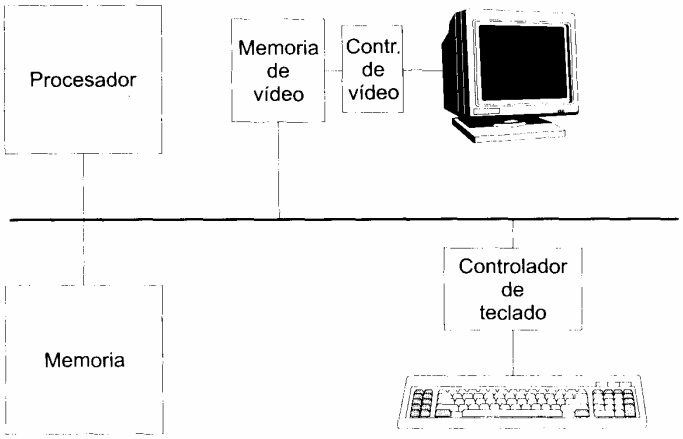


Figura 7.17. *Esquema de un terminal proyectado en memoria.*

operación de refresco de la pantalla, el controlador va leyendo de la memoria cada carácter y el mismo se encarga de obtener el patrón de bits correspondiente al carácter en curso y lo visualiza en la pantalla.

Por lo que se refiere al modo gráfico, en este caso la pantalla se considera una matriz de pixeis y la memoria de vídeo contiene información de cada uno de ellos. Cuando un programa solicita escribir un carácter, debe ser el software el encargado de obtener el patrón de bits que define dicho carácter.

El trabajo con un terminal de este tipo no se limita a escribir en vídeo. El controlador de vídeo contiene un conjunto de registros de entradalsalida que permiten realizar operaciones como mover la posición del cursor o desplazar el contenido de la pantalla una o varias líneas.

Un último aspecto que conviene resaltar es que en este tipo de terminales el tratamiento de las secuencias de escape debe ser realizado por el software.

Terminales serie

En este tipo de terminales, como se puede apreciar en la Figura 7.18, el terminal se presenta ante el resto del sistema como un único dispositivo conectado, típicamente a través de una línea serie RS-232. al controlador correspondiente denominado UART (Universal Asvnchronous Receiver Transmitter, Transmisor-Receptor Universal Asíncrono). Además de la pantalla y el teclado, el terminal tiene algún tipo de procesador interno que realiza parte de la gestión del terminal y que permite también al usuario configurar algunas de las características del mismo. Para poder dialogar con el terminal, se deben programar diversos aspectos de la UART tales como la velocidad de transmisión o el número de bits de parada que se usarán en la transmisión.

Al igual que ocurre con los terminales proyectados en memoria, la entrada se gestiona me diante interrupciones. Cuando se pulsa una tecla, el terminal envía a través de la línea serie el carácter pulsado. La UART genera una interrupción al recibirlo. A diferencia de los terminales proyectados en memoria, el carácter que recoge de la UART ya es el código ASCII de la tecla pulsada puesto que el procesador del terminal se encarga de pasar del código de la tecla al código ASCII y de comprobar el estado de las teclas modificadoras.

Para la visualización de información en este tipo de terminales se debe cargar el código ASCII del carácter deseado en un registro de la UART y pedir a ésta que lo transmita. Una vez

Digitalización realizada con propósito académico

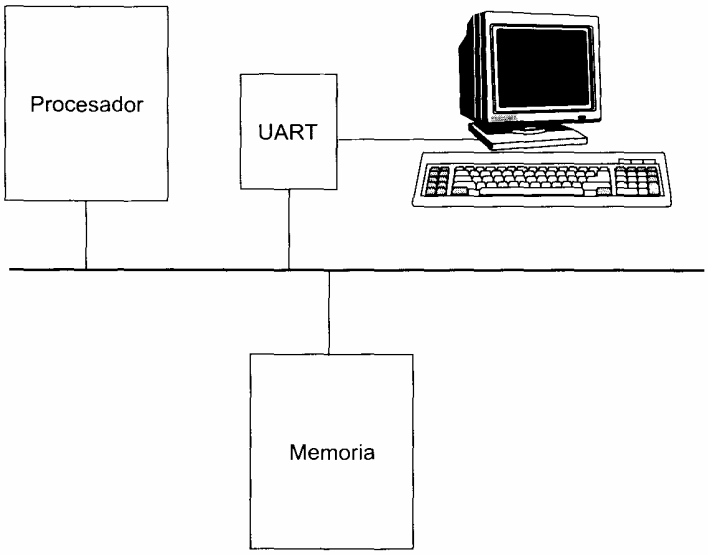


Figura 7.18. *Esquema de un terminal serie.*

transmitido, lo que puede llevar un tiempo considerable debido a las limitaciones de la transmisión serie, la UART produce una interrupción para indicar este hecho. Por lo que se refiere al terminal, cuando recibe el carácter ASCII se encarga de obtener su patrón y visualizarlo en la pantalla, manejando asimismo las secuencias de escape.

7.8.3. El software del terminal

En esta sección se van analizar las labores principales de un manejador de terminal. Algunas de estas labores son específicas del tipo de terminal. Así, por ejemplo, el manejador de un terminal serie debe encargarse de procesar interrupciones asociadas al envío de información al terminal, mientras que uno proyectado en memoria no lo tiene que hacer. Como ejemplo del caso contrario, el manejador de un terminal proyectado en memoria tiene que encargarse de obtener el código ASCII a partir del código de la tecla pulsada, mientras que, en el caso de un terminal serie, esta labor la realiza el hardware. Hay, sin embargo, numerosas labores del manejador independientes del tipo de terminal.

En esta sección se presentarán primero los aspectos relacionados con el software que maneja la entrada para a continuación estudiar el software que gestiona la salida. De todas formas, es necesario hacer notar que hay algunos aspectos que tienen que ver tanto con la entrada como la salida. Así, por ejemplo, el manejador de un terminal serie debe encargarse de programar los pa rámetros de transmisión de la UART (velocidad de transmisión, número de bits de parada, etc.), que corresponde con una labor que tiene que ver tanto con la entrada como con la salida.

Software de entrada

Como se comentó antes, la lectura del terminal está dirigida por interrupciones. En el caso de un terminal proyectado en memoria, el manejador debe realizar más trabajo ya que debe convertir el

Digitalización realizada con propósito académico

código de tecla en el código ASCII correspondiente. Como contrapartida, este esquema propor ciona más flexibilidad al poder ofrecer al usuario la posibilidad de configurar esta traducción a su conveniencia.

Una característica que debe proporcionar todo manejador de terminal es el «tecleado anti cipado» (en inglés, tvpe ahead). Esta característica permite que el usuario teclee información antes de que el programa la solicite, lo que proporciona al usuario una forma de trabajo mucho más cómoda. Para implementar este mecanismo, se requiere que el manejador use una zona de alma cenamiento intermedio de entrada para guardar los caracteres tecleados hasta que los solicite un proceso.

Otro aspecto importante es la edición de los datos de entrada. Cuando un usuario teclea unos datos puede, evidentemente, equivocarse y se requiere, por tanto, un mecanismo que le permita corregir el error cometido. Surge entonces la cuestión de quién se debe encargar de proporcionar esta función de edición de los datos introducidos. Se presentan, al menos, dos alternativas. Una posibilidad es que cada aplicación se encargue de realizar esta edición. La otra alternativa es que sea el propio manejador el que la lleva a cabo. Para poder analizar estas dos opciones es necesario tener en cuenta los siguientes factores:

- La mayoría de las aplicaciones requieren unas funciones de edición relativamente sencillas.
- No parece razonable hacer que cada programa tenga que encargarse de tratar con la edición de su entrada de datos. Un programador desea centrarse en resolver el problema que le concierne y, por tanto, no quiere tener que ocuparse de la edición.
- Es conveniente que la mayoría de las aplicaciones proporcionen al usuario la misma forma de editar la información.
- Hay aplicaciones, como por ejemplo los editores de texto, que requieren unas funciones de edición complejas y sofisticadas.

Teniendo en cuenta estas condiciones, la mayoría de los sistemas operativos optan por una solución que combina ambas posibilidades:

- El manejador ofrece un modo de operación en el que proporciona unas funciones de edición relativamente sencillas, generalmente orientadas a líneas de texto individuales. Este suele ser el modo de trabajo por defecto puesto que satisface las necesidades de la mayoría de las aplicaciones. En los sistemas UNIX, este modo se denomina elaborado ya que el ma nejador procesa los caracteres introducidos.
- El manejador ofrece otro modo de operación en el que no se proporciona ninguna función de edición. La aplicación recibe directamente los caracteres tecleados y será la encargada

de realizar las funciones de edición que considere oportunas. En los entornos UNIX se califica a este modo como crudo ya que no se procesan los caracteres introducidos. Esta forma de trabajo va a ser la usada por las aplicaciones que requieran tener control de la edición.

Por lo que se refiere al modo orientado a línea, hay que resaltar que una solicitud de lectura de una aplicación no se puede satisfacer si el usuario no ha introducido una línea conipleta, aunque se hayan tecleado caracteres suficientes. Observe que esto se debe a que, hasta que no se completa una línea, el usuario todavía tiene las posibilidades de editarla usando los caracteres de edición correspondientes.

Con respecto al modo crudo, en este caso no existen caracteres de edición aunque, como se verá a continuación, pueden existir otro tipo de caracteres especiales. Cuando un programa solicita datos, se le entregan aunque no se haya tecleado una línea completa. De hecho, en este modo el manejador ignora la organización en líneas realizando un procesado carácter a carácter.

Digitalización realizada con propósito académico

En el modo elaborado existen unos caracteres que tienen asociadas funciones de edición, pero éstos no son los únicos caracteres que reciben un trato especial por parte del manejador. Existe un conjunto de caracteres especiales que normalmente no se le pasan al programa que lee del terminal, como ocurre con el resto de los caracteres, sino que activa alguna función del manejador. Estos caracteres se pueden clasificar en las siguientes categorías:

• **Caracteres de edición**. Tienen asociadas función de edición tales como borrar el último carácter tecleado, borrar la línea en curso o indicar el fin de la entrada de datos. Estos caracteres sólo se procesan si el terminal está en modo línea. Supóngase, por ejemplo, que el carácter backspace tiene asociada la función de borrar el último carácter tecleado. Cuando el usuario pulsa este carácter, el maiiejador lo detecta y no lo encola en la zona donde almacena la línea en curso, sino que elimina de dicha zona el último carácter encolado. Dentro de esta categoría, es conveniente analizar cómo se procesa el carácter que indica el final de una línea. Observe que, estrictamente, un cambio de línea implica un retorno de carro para volver al principio de la línea actual y un avance de la línea para pasar a la siguiente. Sin embargo, no parece que tenga sentido obligar a que el usuario teclee ambos caracteres. Así, en los sistemas UNIX, el usuario puede teclear tanto un retorno de carro como un carácter de nueva línea para indicar el final de la línea. El manejador se encarga de mantener esta equivalencia y, por convención, en ambos casos entrega un carácter de nueva línea al programa que lee del terminal.

• **Caracteres para el control de procesos.** Todos los sistemas proporcionan al usuario algún carácter para abortar la ejecución de un proceso o detenerla temporalmente.

• **Caracteres de control de flujo**. El usuario puede desear detener momentáneamente la sa lida que genera un programa para poder revisarla y, posteriormente, dejar que continúe apareciendo en la pantalla. El manejador gestiona caracteres especiales que permiten rea lizar estas operaciones.

• **Caracteres de escape**. A veces el usuario quiere introducir como entrada de datos un ca rácter que está definido como especial. Se necesita un mecanismo para indicar al manejador que no trate dicho carácter, sino que lo pase directamente a la aplicación. Para ello, gene ralmente, se define un carácter de escape cuya misión es indicar que el carácter que viene a continuación no debe procesarse. Evidentemente, para introducir el propio carácter de escape habrá que teclear otro carácter de escape justo antes.

Por último hay que resaltar que la mayoría de los sistemas ofrecen la posibilidad de cambiar

qué carácter está asociado a cada una de estas funciones o incluso desactivar dichas funciones si se considera oportuno.

Software de salida

La salida en un terminal no es algo totalmente independiente de la entrada. Por defecto, el inane jador hace eco de todos los caracteres que va recibiendo en las sucesivas interrupciones del teclado. Así, la salida que aparece en el terminal es una mezcla de lo que escriben los programas y del eco de los datos introducidos por el usuario. Esta opción se puede desactivar, en cuyo caso el manejador no escribe en la pantalla los caracteres que va recibiendo.

A diferencia de la entrada, la salida no está orientada a líneas de texto, sino que se escriben

directamente los caracteres que solicita el programa, aunque no constituyan una línea (Aclara ción 7.1).

Digitalización realizada con propósito académico

ACLARACIÓN 7.1

Un programador en C que trabaja en un sistema UNIX puede pensar que la salida en el terminal está orientada a líneas ya que cuando usa en la sentencia `printf` una cadena de caracteres que no termina en un carácter de nueva línea, ésta no aparece inmediatamente en la pantalla. Sin embargo, este efecto no es debido al manejador sino al almacenamiento intermedio que se produce en la biblioteca estándar de entrada/salida del lenguaje C.

El software de salida para los terminales serie es más sencillo que para los proyectados en memoria. Esto se debe a que en los primeros el propio terminal se encarga de realizar muchas labores liberando al manejador de las mismas. Por ello se estudiarán estos dos casos de I separada.

Terminal serie

En este tipo de terminal la salida está dirigida por interrupciones. Cuando un programa solicita escribir una cadena de caracteres. el manejador los copia en un espacio de almacenamiento in termedio de salida. A continuación, el manejador copia el primer carácter en el registro corres pondiente de la UART y le pide a ésta que lo envíe al terminal. Cuando la UART genera la interrupción de fin de transmisión, el manejador copia en ella el segundo carácter y así sucesi vamente.

Cuando recibe el carácter a través de la línea serie, el propio terminal se encarga de todas las labores implicadas con la presentación del carácter en pantalla. Esto incluye aspectos tales como la obtención del patrón que representa al carácter y su visualización, el manejo de caracteres que tienen alguna presentación especial (p. ej.: el carácter campanada), la manipulación de la posición del cursor o el manejo e interpretación de las secuencias de escape.

Terminal proyectado en memoria

La salida en este tipo de terminales implica leer del espacio del proceso los caracteres que éste desea escribir, realizar el procesado correspondiente y escribir en la memoria de vídeo el resultado del mismo. No se requiere, por tanto, el uso de interrupciones o de un espacio de almacenamiento de salida. Si la pantalla está en modo alfanumérico, en la memoria de vídeo se escribe el carácter a visualizar, siendo el propio controlador de vídeo el que se encarga de obtener el patrón que lo representa. En el caso del modo gráfico, el propio manejador debe obtener el patrón y modificar la memoria de vídeo de acuerdo al mismo.

A diferencia de los terminales serie, en este caso el manejador debe encargarse de la presen tación de los caracteres. Para la mayoría de ellos, su visualización implica simplemente la actua lización de la memoria de vídeo. Sin embargo, algunos caracteres presentan algunas dificultades específicas como, por ejemplo, los siguientes:

- Un tabulador implica mover el cursor el número de posiciones adecuadas.
- El carácter campanada (control-G) requiere que el altavoz del equipo genere un sonido.
- El carácter de borrado debe hacer que desaparezca de la pantalla el carácter anterior.
- Los caracteres de salto de línea pueden implicar desplazar el contenido de la pantalla una línea hacia arriba cuando se introducen estando el cursor en la línea inferior. El manejador debe escribir en el registro correspondiente del controlador de vídeo para realizar esta ope ración.

Digitalización realizada con propósito académico

Por último, hay que recordar que en este tipo de terminales, el manejador debe encargarse de interpretar las secuencias de escape, que normalmente requerirán modificar los registros del con trolador que permiten mover el curso o desplazar el contenido de la pantalla.

7.9. LA RED

El dispositivo de red se ha convertido en un elemento fundamental de cualquier sistema informá tico. Por ello, el sistema operativo ha evolucionado para proporcionar un tratamiento más exhaus tivo y sofisticado de este dispositivo.

Se trata de un dispositivo que presenta unas características específicas que generalmente im plican un tratamiento diferente al de otros dispositivos. Así, muchos sistemas operativos no propor cionan una interfaz basada en archivos para el acceso a la red, aunque sí lo hacen para el resto de dispositivos. En Linux, por ejemplo, no hay archivos en /dev que representen a los dispositivos de red. Los manejadores de estos dispositivos de red se activan automáticamente cuando en el arranque del sistema se detecta el hardware de red correspondiente, pero no proporcionan acceso directo a los programas de usuario.

Típicamente, el software de gestión de red del sistema operativo se organiza en tres niveles:

- Nivel de interfaz a las aplicaciones.
- Nivel de protocolos. Este nivel, a su vez, puede estar organizado en niveles que se corres ponden con la pila de protocolos.
- Nivel de dispositivo red.

Con respecto al nivel superior, una de las interfaces más usadas es la que corresponde con los sockets del UNIX BSD, que incluso se utiliza en sistemas Windows (Winsock). Esta interfaz, que se describirá en el Capítulo 10, tiene servicios específicos que sólo se usan para acceder a la red (como, por ejemplo, connect, bind, accept y sendto), aunque usa descriptores de archivo para representar a las conexiones y permite utilizar servicios convencionales de archivo (como read, write y close). Se puede considerar que se trata de un nivel de sesión de OS!.

El nivel intermedio consta de una o más capas que implementan los niveles de transporte y de red de OS! (o los niveles TCP e IP en el caso de Internet). Este nivel se encarga también de funciones de encaminamiento. En el nivel inferior están los manejadores de los dispositivos de red que se corresponden con el nivel de enlace de OS!. En este nivel puede haber manejadores para distintos tipos de hardware de red (Ethernet, SLIP, etc.).

Todos estos niveles trabajan de manera independiente entre sí, lo que facilita la coexistencia de distintos protocolos y dispositivos de red. Para lograr esta independencia se definen interfaces estándar entre los distintos niveles.

La información viajará del nivel superior a los inferiores como resultado de una llamada de una aplicación solicitando transferir un mensaje. El flujo inverso de información se activa con la ocurrencia de una interrupción del dispositivo de red que indica la llegada de un mensaje.

Según la información va pasando por los distintos niveles se va añadiendo y eliminando in formación de control. Un aspecto fundamental para conseguir una implementación eficiente del software de gestión de red es intentar reducir lo máximo posible el número de veces que se copia la información del mensaje.

La Figura 7. 19 muestra cómo están organizados los distintos niveles de software del sistema operativo que gestiona la red.

Digitalización realizada con propósito académico

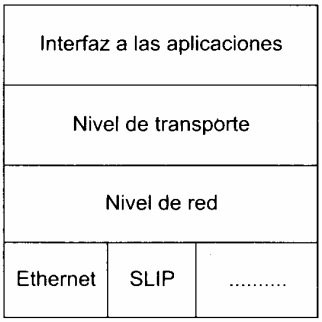


Figura 7.19. *Niveles típicos del software de gestión de red.*

7.10. SERVICIOS DE ENTRADA/SALIDA

7.10.1. Servicios genéricos de entrada/salida

En esta sección se intentan generalizar los servicios de entradalsalida que presentan los distintos sistemas operativos identificando un conjunto de servicios representativos de los existentes en los sistemas reales. Dado que, como se comentó previamente, los sistemas operativos no tratan al reloj de la misma manera que al resto de los dispositivos, se expondrán en primer lugar los servicios relacionados con el mismo para, a continuación, presentar los servicios correspondientes a los dispositivos de entrada/salida convencionales.

Servicios relacionados con el reloj

Se pueden clasificar en las siguientes categorías:

• **Servicios de hora y fecha.** Debe existir un servicio para obtener la fecha y la hora, así como para modificarla. Este último sólo lo podrán usar procesos privilegiados.

• **Temporizaciones.** El sistema operativo suele ofrecer servicios que permiten establecer tem porizaciones síncronas, o sea, que bloqueen al proceso el tiempo especificado, y asíncronas,

esto es, que no bloqueen al proceso pero le manden algún tipo de evento cuando se cumpla el plazo.

• **Servicios para obtener la contabilidad y las estadísticas recogidas por el sistema ope rativo.** Por ejemplo, en casi todos los sistemas operativos existe algún servicio para obtener información sobre el tiempo de procesador que ha consumido un proceso.

Servicios de entrada/salida

La mayoría de los sistemas operativos modernos proporcionan los mismos servicios para trabajar con dispositivos de entrada/salida que los que usan con los archivos. Esta equivalencia es muy beneficiosa ya que proporciona a los programas independencia del medio con el que trabajan. Así, para acceder a un dispositivo se usan los típicos servicios para abrir, leer, escribir y cerrar archivos.

Sin embargo, en ocasiones es necesario poder realizar desde un programa alguna operación dependiente del dispositivo. Por ejemplo, suponga un programa que desea solicitar al usuario una contraseña a través del terminal. Este programa necesita poder desactivar el eco en la pantalla para

Digitalización realizada con propósito académico

que no pueda verse la contrasefia mientras se teclea. Se requiere, por tanto, algún mecanismo para poder realizar este tipo de operaciones dependiendo del dispositivo. Existen al menos dos posibi lidades no excluyentes:

- Permitir que este tipo de opciones se puedan especificar como indicadores en el propio servicio para abrir el dispositivo.
- Proporcionar un servicio que permita invocar estas opciones dependientes del dispositivo.

7.10.2. Servicios de entrada/salida en POSIX

Debido al tratamiento diferenciado que se da al reloj, se presentan separadamente los servicios relacionados con el mismo de los correspondientes a los dispositivos de entrada/salida conven cionales.

En cuanto a los servicios del reloj, se van a especificar de acuerdo a las tres categorías antes planteadas: fecha y hora, temporizaciones y contabilidad. Con respecto a los servicios destinados a dispositivos convencionales, se presentarán de forma genérica haciendo especial hincapié en los relacionados con el terminal.

Servicio de fecha y hora

El servicio para obtener la fecha y hora es time, cuyo prototipo es el siguiente:

time_t time (time_t *t)

Esta función devuelve el número de segundos transcurridos desde el 1 de enero de 1970 en UTC. Si el argumento no es nulo, también lo almacena en el espacio apuntado por el mismo.

Algunos sistemas UNIX proporcionan también el servicio gettimeofday que proporciona una precisión de microsegundos.

La biblioteca estándar de C contiene funciones que transforman el valor devuelto por time a un formato más manejable (año, mes, día, horas, minutos y segundos) tanto en UTC, la función gmtime, como en horario local, la función localtime.

Para cambiar la hora y fecha se proporciona el servicio stime, cuyo prototipo es:

mt stime (timet *t)

Esta función fija la hora del sistema de acuerdo al parámetro recibido, que se interpreta como el número de segundos desde el 1 de enero de 1970 en UTC. Se trata de un servicio sólo disponible para el superusuario.

En algunos sistemas UNIX existe la función settimeofday que permite realizar la misma función pero con una precisión de microsegundos.

Todas estas funciones requieren el archivo de cabecera time . h. Como ejemplo del uso de estas funciones se presenta el Programa 7.3 que imprime la fecha y la hora en horario local.

Programa 7.3. Programa que imprime la fecha y hora actual.

```
#include <stdio.h>

#inciude <time.h>
```

mt mamo

timet tiempo;

struct tm *fecha.

tiempo= **time** (NULL); fechas localtime(&tiempo)

/* hay que ajustar el año ya que lo devuelve respecto a 1900 */ printf (“%02d/%02d/%04d %02d:%02d:%02d\n”,

fecha->tmmday, fecha->tm_mon, fecha->tm_year+1900, fecha->tmhour, fecha—>tmmin, fecha->tmsec)

return 0;

Servicios de temporización

El servicio alarm permite establecer un temporizador en POSIX. El plazo debe especiflcarse en segundos. Cuando se cumple dicho plazo, se le envía al proceso la señal SIGALRM. Sólo se permite un temporizador activo por cada proceso. Debido a ello, cuando se establece un temporizador, se desactiva automáticamente el anterior. El prototipo de esta función es el siguiente:

unsigned mt **alarm** (unsigned mt segundos);

El argumento especifica la duración del plazo en segundos. La función devuelve el número

de segundos que le quedaban pendientes a la alarma anterior, si había alguna pendiente. Una alarma con un valor de cero desactiva un temporizador activo.

En POSIX existen otro tipo de temporizadores que permiten establecer plazos con una mayor

resolución y con modos de operación más avanzados. Este tipo de temporizadores se activa con la función setitimer.

Servicios de contabilidad

POSIX define diversas funciones que se pueden englobar en esta categoría. Esta sección va a presentar una de las más usadas. El servicio times que devuelve información sobre el tiempo de ejecución de un proceso y de sus procesos hijos. El prototipo de la función es el siguiente:

clockt **times** (struct tms *info)

ble Esta función rellena la zona apuntada por el puntero recibido como argumento con informa ción sobre el uso del procesador, en modo usuario y en modo sistema, tanto del propio proceso como de sus procesos hijos. Además, devuelve un valor relacionado con el tiempo real en el sistema

de (típicamente, el número de interrupciones de reloj que se han producido desde el arranque del

sistema). Observe que este valor no se usa de manera absoluta. Normalmente, se compara el valor

devuelto por dos llamadas a times realizadas en distintos instantes para determinar el tiempo real transcurrido entre esos dos momentos.

El Programa 7.4 muestra el uso de esta llamada para determinar cuánto tarda en ejecutarse un programa que recibe como argumento. La salida del programa especifica el tiempo real, el tiempo de procesador en modo usuario y el tiempo de procesador en modo sistema consumido por el programa especificado.

Digitalización realizada con propósito académico

Programa 7.4. Programa que muestra el tiempo real, el tiempo en modo usuario y en modo sistema que se consume durante la ejecución de un programa.

```
#include <stdio.h> #include <unistd.h> #include <time.h> #include <sys/times.h>
int main(int argc, char *argv struct tms Infolnicio, InfoFin; clock_t tinicio, tfin; long tickporseg;
if (argc<2)
fprintf(stderr, “Uso: %s programa [ argv
exit(l)
/ obtiene el número de interrupciones de reloj por segundo /
tickporseg= sysconf (_SC_CLK_TCK);
t_inicio= times (&Infolnicio)
if (fork()==O)
execvp(argv &argv
perror (“error ej ecutando el programa”)
exit(l)
wait(NULL);
t_fin= times(&InfoFin)
printt(”Tiempo real: %7.2f\n”, (float) (tfin — t inicio) /tickporseg)
printf( de usuario: %7.2f\n”,
(float) (InfoFin. tms_cutime— Infolnicio.tmscutime) /tickporseg)
printf(”Tiempo de sistema: %7.2f\n”,
(float) (InfoFin.tmscstime— Infolnicio.tms_cstime) /tickporseg)
return O;
}
```

Servicios de entrada/salida sobre dispositivos

Corno ocurre en la mayoría de los sistemas operativos, se utilizan los mismos servicios que para acceder a los archivos. En el caso de POSIX, por tanto, se trata de los servicios open, read, write y close. Estas funciones se estudian detalladamente en el Capítulo 8.

Es necesario, sin embargo, un servicio que permita configurar las opciones específicas de cada dispositivo o realizar operaciones que no se pueden expresar como un read o un write (corno, por ejemplo, rebobinar una cinta). En UNIX existe la función ioctl, cuyo prototipo es el siguiente:

```
intiocti (mt descriptor, intpeticion, ..j;
```

Digitalización realizada con propósito académico

El primer argumento corresponde con un descriptor de archivo asociado al dispositivo corres pondiente. El segundo argumento es el tipo de operación que se pretende realizar sobre el dis positivo, que dependerá del tipo de dispositivo. Los argumentos restantes, normalmente sólo Uflo, especifican los parámetros de la operación que se pretende realizar.

El uso más frecuente de esta función es para trabajar con terminales. Sin embargo, aunque esta función se incluya en prácticamente todas las versiones de UNIX, el estándar POSIX no la define. En el caso del terminal, POSIX especifica un conjunto de funciones destinadas sólo a este tipo de dispositivo. Las dos más frecuentemente usadas son tcgetattr y tcsetattr, destinadas a obtener los atributos de un terminal y a modificarlos, respectivamente. Sus prototipos son los siguientes:

int **tcgetattr** (mt descriptor, struct termios *atrjb);

int **tcsetattr** (mt descriptor, mt opcion, struct termios *atrib);

La función tcgetattr obtiene los atributos del terminal asociado al descriptor especificado. La función tcsetattr modifica los atributos del terminal que corresponde al descriptor pasado como parámetro. Su segundo argumento establece cuándo se producirá el cambio: inmediatamente o después de que la salida pendiente se haya transmitido.

La estructura termios contiene los atributos del terminal que incluye aspectos tales como el tipo de procesado que se realiza sobre la entrada y sobre la salida, los parámetros de transmisión en el caso de un terminal serie o la definición de qué caracteres tienen un significado especial. Dada la gran cantidad de atributos existentes (más de 50), en esta exposición no se entrará en detalles sobre cada uno de ellos, mostrándose simplemente un ejemplo de cómo se usan estas funciones. Como ejemplo, se presenta el Programa 7.5 que lee una contraseña del terminal desac tivando el eco en el mismo para evitar problemas de seguridad.

Programa 7.5. Programa que lee del terminal desactivando el eco.

```
#include <stdio.h>

*include <termios.h> #include <unistd.h> #define TAMMAX 32

mt main(int argc, char **argv)

struct termios termatrib;

char linea

/* Obtiene los atributos del terminal */

if (tcgetattr(O, &term_atrib)<O);

perror ("Error obteniendo atributos del terminal");

exit(l)

/ Desactiva el eco. Con TCSANOW el cambio sucede inmediatamente */ term_atrib.cjflag &= —ECHO;

tcsetattr(O, TCSANOW, &termatrib)

printf ("Introduzca la contraseña:");
```

Digitalización realizada con propósito académico

```
/* Lee la línea y a continuación la muestra en pantalla */
if (fgets(linea, TAN_MAX, stdin))
linea / Eliminando fin de línea */
printf("\nHas escrito %s\n", linea);
/ Reactiva el eco /
term_atrib.cjflag 1= ECHO;
tcsetattr(O, TCSANOW, &term_atrib);
exit(O);
```

7.10.3. Servicios de entrada/salida en Win32

Se van a presentar clasificados en las mismas categorías que se han usado para exponer los servicios POSIX.

Servicios de fecha y hora

El servicio para obtener la fecha y hora es GetSystemTime, cuyo prototipo es el siguiente:

BOOL **GetSystemTime** (LPSYSTEMTIME tiempo)

Esta función devuelve en el espacio asociado a su argumento la fecha y hora actual en una estructura organizada en campos que contienen el año, mes, día, hora, minutos, segundos y mi lisegundos. Los datos devueltos corresponden con el horario estándar UTC. La función GetLocal Time permite obtener información según el horario local.

La función que permite modificar la fecha y hora en el sistema es SetSystemTime, cuyo prototipo es:

BOOL **SetSystemTime** (LPSYSTEMTIME tiempo);

Esta función establece la hora del sistema de acuerdo al parámetro recibido que tiene la misma estructura que el usado en la función anterior (desde el año hasta los milisegundos actuales).

Como ejemplo se plantea el Programa 7.6 que imprime la fecha y hora actual en horario local.

Programa 7.6. Programa que imprime la fecha y hora aLtual

```
#include <windows.h>
include <stdio.h>
mt main (mt argc, LPTSTR argv [
SYSTEMTIME Tiempo;
GetLocalTime (&Tiempo);
printf("%02d/%02d/%04d %02d:%02d:%02d\n",
Tiempo .wDay, Tiempo .wMonth,
Tiempo .wYear, Tiempo .wHour,
```

Tiempo.wMinute, Tiempo.wSecond);

return O;

}

Servicios de temporización

La función setTimer permite establecer un temporizador con una resolución de milisegundos. Un proceso puede tener múltiples ternporizadores activos simultáneamente. El prototipo es:

UINT **SetTimer** (HWND ventana, UINT evento, UINT plazo,

TIMERPROC funcion);

Esta función devuelve un identificador del temporizador y recibe como parámetros un iden tificador de la ventana asociada con el temporizador (si tiene un valor nulo no se le asocia a ninguna), un identificador del evento (este parámetro se ignora en el caso de que el temporizador no esté asociado a ninguna ventana), un plazo en milisegundos y la función que se invocará cuando se cumpla el plazo.

Servicios de contabilidad

En Win32 existen diversas funciones que se pueden encuadrar en esta categoría. Como en el caso de POSIX, se muestra como ejemplo el servicio que permite obtener información sobre el tiempo de ejecución de un proceso. Esta función es GetProcessTimes y su prototipo es el siguiente:

BOOL **GetProcessTimes** (HANDLE proceso, LPFILETIME t_creacion,

LPFILETIME t_fin, LPFILETIME tsistema,

LPFILETIME tusuario);

Esta función obtiene para el proceso especificado su tiempo de creación y finalización, así como cuánto tiempo de procesador ha gastado en modo sistema y cuánto en modo usuario. Todos estos tiempos están expresados como centenas de nanosegundos transcurridas desde el 1 de enero de 1601 almacenándose en un espacio de 64 bits. La función FileTimeToSystemTime permite transformar este tipo de valores en el formato ya comentado de año, mes, día, hora, minutos, segundos y milisegundos.

Como ejemplo del uso de esta función, se presenta el Programa 7.7 que imprime el tiempo que tarda en ejecutarse el programa que recibe como argumento especificando el tiempo real de ejecución, así como el tiempo de procesador en modo usuario y en modo sistema consumido por dicho programa.

Programa 7.7. Programa que muestra el tiempo real, el tiempo en modo usuario y en modo sistema que se consume durante la ejecución de un programa.

finclude <windows . h> #include <stdio.h>

fdefine MAX 255

mt main (mt argc, LPTSTR argv [ ])

}

Digitalización realizada con propósito académico

```
int i;
BOOL result; STARTUPINFO Infolnicíal; PROCESS_INFORMATION InfoProc union
LONGLONG numero;
FILETIME tiempo;
TiempoCreacion, TiempoFin, TiempoTranscurrido;
FILETIME TiempoSistema, TiempoUsuario;
SYSTEMTIME TiempoReal, TiempoSistema, TiempoUsuario;
CHAR mandato
/* Obtiene la línea de mandato saltándose el primer argumento */ sprintf(mandato,"%s", argv{l]);
for (i=2; i<argc; i++)
sprintf (mandato, "%s %s", mandato, argv
GetStartuplnfo(&]info
result= CreateProcess(NULL, mandato, NULL, NULL, TRUE, NORMAL PRIORITY CLASS, NULL, NULL, &Infolnicial,
&InfoProc)
if (! result)
fprintf(stderr, "error creando el proceso\n");
return(l);
WaitForSingleObject(InfoProc.hProcess, INFINITE);
GetProcessTixnes(InfoProc.hProcess, &TiempoCreación. tiempo, &TiempoFin.tiempo, &TiempoSístema, &TiempoUsuario);
TiempoTranscurrido numero= TiempoFin .numero— TiempoCreación numero;
FileTimeToSystemTime(&TiempoTranscurrido.tiempo, &TiempoReal); FileTimeToSystemTime (&TiempoSistema, &TiempoSistema) FileTimeToSysteinTirne(&Tiempousuario, &TiempoUsuario); príntf ("Tiempo real: %02d:%02d:%02d:%03d\n",
TiempoReal .wHour, TiempoReal .wMinute, TiempoReal .wSecond, TiempoReal .wMilliseconds);
printf ("Tiempo de usuario: %02d:%02d:%02d:%03d\n", TiempoUsuario.wHour, TiempoUsuario .wMinute,
TiempoUsuario.wSecond, TiempoUsuario.wMilliseconds); printf ("Tiempo de sistema: %02d:%02d:%02d:%03d\n",
TiempoSistema.wHour, TiempoSistema.wMinute,
TiempoSistema.wSecond, TiempoSistema.wMilliseconds);
 return O;
}
```

Digitalización realizada con propósito académico

Servicios de entradalsalida

Al igual que en POSIX, para acceder a los dispositivos se usan los mismos servicios que se utilizan para los archivos. En este caso, createFile, CloseHandie, ReadFile y WriteFile.

Sin embargo, como se ha analizado previamente, estos servicios no son suficientes para cubrir

todos los aspectos específicos de cada tipo de dispositivo. Centrándose en el caso del terminal, Win32 ofrece un servicio que permite configurar el modo de funcionamiento del mismo. Se de nomina SetConsoleMode y su prototipo es el siguiente:

BOOL **SetConsoleMode** (HANDLE consola, DWORD modo);

Esta función permite configurar el modo de operación de una determinada consola. Se pueden especificar, entre otras, las siguientes posibilidades: entrada orientada a líneas (ENABLE_LINE_IN PUT), eco activo (ENABLE_ECHO_INPUT), procesamiento de los caracteres especiales en la salida (ENABLE_PROCESSED_OUTPUT) y en la entrada (ENABLE_PROCESSED_INPUT).

Es importante destacar que para que se tengan en cuenta todas estas opciones de configuración hay que usar para leer y escribir del dispositivo las funciones ReadConsole y WriteConsole, respectivamente, en vez de utilizar ReadFile y WriteFile. Los prototipos de estas dos nuevas funciones son casi iguales que las destinadas a archivos:

BOOL **ReadConsole** (HANDLE consola, LPVOID mem, DWORD a_leer,

LPDWORD leidos, LPVOID reservado);

BOOL **WriteConsole** (HANDLE consola, LPVOID mem, DWORD a_escribir,

LPVOID escritos, LPVOID reservado);

El significado de los argumentos de ambas funciones es el mismo que los de ReadFile y WriteFile, respectivamente, excepto el último parámetro que no se utiliza.

Como muestra del uso de estas funciones, se presenta el Programa 7.8 que lee la información del terminal desactivando el eco del mismo.

Programa 7.8. Programa que lee del terminal desactivando el eco.

#include <windows . h> #include <stdio.h> #define TAMMAX 32

mt main (mt argc, LPTSTR argv E])

HANDLE entrada, salida;

DWORD leídos, escritos;

BOOL result;

BYTE clave [

entrada= **CreateFile**(”CONIN$”, GENERIC_READ GENERIC_WRITE, O, NULL, OPENALWAYS, FILE_ATTRIBUTE NORMAL, NULL);

if (entrada == INVALID HANDLE VALUE)

fprintf (stderr, “No puede abrirse la consola para lectura\n”);

return(l);

}

Digitalización realizada con propósito académico

```
salida= CreateFile (“CONOUT$”, GENERICWRITE, O,
MJLL, OPENALWAYS, FILE ATTRIBUTE NORMAL, NULL);
if (salida == INVALID_HANDLE_VALUE)
fprintf(stderr, “No puede abrirse la consola para escritura\n”)
return(l);
result = SetConsoleMode(entrada, ENABLE_LINE_INPUT
        ENABLEPROCESSEDINPUT)
&& SetConsoleMode (salida, ENABLE_WRAP_AT_EOL_OUTPUT
         ENABLE_PROCESSEDOUTPUT)
&& WriteCorisole(salida, “Introduzca la contraseña: “, 26,
      &escritos, NULL)
&& ReadConsole(entrada, clave, TAM_MAX, &leidos, NULL);
if (! result) {
fprintf(stderr, “error accediendo a la consola\n”)
return(l);
}
CloseHandle (entrada);
CloseHandle(salida);
return result;
}
```

7.11. PUNTOS A RECORDAR

- ☑ Los dispositivos de E/S almacenan datos y permiten interactuar con los usuarios y los programadores de las computadoras. Los del primer tipo son básicamente los dispositivos de almacenamiento secundario (discos) y terciario (cintas y sistemas de archivo). Los segundos son los denominados dispositivos periféricos de interfaz de usuario.
- ☑ Los dispositivos de E/S se pueden clasificar en perifé ricos de interfaz de usuario, dispositivos de almacena miento y dispositivos de comunicaciones.
- ☑ Los dispositivos de E/S son actualmente seis órdenes de magnitud más lentos que la UCP. los registros y lamemoria. Su tiempo de acceso del orden de milisegun dos, mientras que el de los últimos es del orden de na nosegundos.
- ☑ El sistema de E/S es la parte del sistema operativo que se ocupa dc facilitar el manejo de los dispositivos de E/S ofreciendo una visión lógica simplificada de los mismos que pueda ser usada por otras componentes del sistema operativo (como el sistema de archivos) o in cluso por el usuario.
- ☑ Los diseñadores de sistemas operativos se encuentran en un lugar intermedio entre los programadores de apli caciones y los fabricantes de dispositivos. Les interesa la funcionalidad del dispositivo, aunque a un nivel de detalle mucho más grande que la funcionalidad que es pera el

Digitalización realizada con propósito académico

programador de aplicaciones, pero también les interesa conocer la interfaz física de los dispositivos y su comportamiento interno para poder optimizar los métodos de acceso a los mismos.

- ☑ Un dispositivo de E/S está compuesto por el dispositivo en sí y por el controlador, componente electrónico que se conecta al bus de la computadora.
- ☑ El controlador es el componente más importante des de el punto de vista del sistema operativo, ya que constituye la interfaz del dispositivo con el bus de la computadora y es el componente que se ve desde la UCP.
- ☑ Existen dispositivos conectados por puertos o proyec tados en memoria. En los primeros, para empezar una operación de E/S, la UCP tiene que escribir sobre los registros del controlador los datos de la operación a

Digitalización realizada con propósito académico

través de una instrucción de EIS. En los segundos, lo hace sobre direcciones de memoria asignadas única mente al controlador.

- ☑ Según la unidad de acceso, los dispositivos pueden ser bloques o de caracteres.
- ☑ La información entre los controladores de dispositivo y la unidad central de proceso o memoria principal se puede transferir mediante un programa que ejecuta con tinuamente. técnica que se denomina **E/S programada**, o programando el dispositivo y esperando a que notifi que el fin de la operación, técnica que se denomina **E/S por interrupciones**.
- ☑ El DMA supone una mejora importante al incrementar la concurrencia entre la UCP y la E/S. Con esta técnica, el controlador del dispositivo se encarga de efectuar la transferencia de datos a memoria, liberando de este tra bajo a la UCP.
- ☑ La arquitectura del sistema de entrada/salida es comple ja y está estructurada en capas, cada una de las cuales tiene una funcionalidad bien definida.
- ☑ El software de E/S del sistema operativo se estructura en las siguientes capas: manejadores de interrupción, manejadores de dispositivos o drivers, software de E/S independiente de los dispositivos e interfaz del sistema operativo.
- ☑ Cada dispositivo de E/S. o cada clase de dispositivos, tiene un manejador asociado en el sistema operativo. Dicho manejador incluye: código independiente del dispositivo para proporcionar al nivel superior del sis tema operativo una interfaz de alto nivel y el código dependiente del dispositivo necesario para programar el controlador del dispositivo a través de sus registros y mandatos.
- ☑ La mayor parte del sistema de E/S es software indepen diente de dispositivo.
- ☑ La interfaz de E/S de las aplicaciones lo constituyen las bibliotecas de E/S y los procesos demonio.
- ☑ La interfaz de E/S de las aplicaciones es la que define el modelo de E/S que ven los usuarios en cuanto a nom bres. control de acceso, bloqueos, indicaciones de error y uso de estándares.
- ☑ El sistema de almacenamiento secundario se usa para guardar los programas y datos en dispositivos rápidos, de forma que sean fácilmente accesibles a las aplicacio nes a través del sistema de archivos.
- ☑ Los discos son los dispositivos básicos para llevar a cabo almacenamiento masivo y no volátil de datos. Sus controladores más populares son SCSI e IDE. Los dis cos pueden ser duros, ópticos o extraíbles.
- ☑ Un disco duro es un dispositivo de gran capacidad com puesto por varias superficies magnetizadas y cuyas ca bezas lectoras funcionan por efecto electromagnético. Su formato físico lo orgafliLa en cilindros, pistas y sectores.
- ☑ Una partición es una porción contigua de disco delimi tada por un sector inicial y final. La información de dis tribución lógica del disco en subconjuntos denominados volúmenes o particiones se almacena en una estructura denominada **tabla de particiones**.
- ☑ El **formato lógico** de la partición crea un sistema de archivos dentro de dicha partición.
- ☑ En LINUX y Windows NT existe un manejador gené rico para la clase de dispositivo disco y un manejador dependiente del dispositivo para cada tipo de disco en particular.
- ☑ En los discos es fundamental usar políticas de planifi cación que minimicen el movimiento de cabezas para obtener un buen rendimiento medio del disco.
- ☑ La política de planificación de disco CSCAN es la más usada en todos los sistemas operativos.

Digitalización realizada con propósito académico

- Una de las funciones básicas del manejador de disco es gestionar los errores de E/S producidos en el ámbito del dispositivo. Estos errores pueden provenir de las apli caciones (p. ej.: petición para un dispositivo o sector que no existe), del controlador (p. ej.: errores al aceptar peticiones o parada del controlador) o de los dispositi vos (p. ej.: fallos transitorios o permanentes de lectura o escritura y fallos en la búsqueda de pistas).
- El sistema de E/S es uno de los componentes del siste ma con mayores exigencias de fiabilidad, debido a que se usa para almacenar datos y programas de forma per manente.
- La técnica clásica de almacenamiento redundante es usar **discos espejo** [Tanenbaum, 19921, es decir. dos discos que contienen exactamente lo mismo.
- Actualmente los dispositivos RAID son los dispositivos de elección en un sistema que requiera fiabilidad y altas prestaciones.
- El sistema de almacenamiento terciario se puede definir como un sistema de almacenamiento de alta capacidad. bajo coste y con dispositivos extraíbles en el que se al macenan los datos que no se necesitan de forma innie diata en el sistema.
- La tecnología de almacenamiento terciario no ha evolucionado mucho en los últimos años. Los dispo sitivos de elección son los CD-ROM. los DVD y. so bre todo, las cintas magnéticas. En cuanto al soporte usado, se usan jukeboxes y sistemas robotizados para las cintas.
- Una tecnología novedosa para el almacenaniiento se cundario y terciario la constituyen las SAN (Siorage Area Networks), que son redes de altas prestaciones a las que se conectan directamente dispositivos de alma cenam iento.
- Los sistemas de almacenamiento terciario existentes en los sistemas operativos actuales. UNIX, LINUX y Win dows, tienen una estructura muy sencilla consistente en

Digitalización realizada con propósito académico

un nivel de dispositivos extraíbles, tales como cintas, disquetes o CD-ROM.

- La cuestión de la **migración de archivos** del sistema secundario al terciario, y viceversa, depende actual mente de las decisiones del administrador de usuario (en el caso de las copias de respaldo) o de los propios usuarios (en el caso de archivos personales). En los sis temas operativos actuales no se define ninguna política de migración a sistemas terciarios, dado que estos sis temas pueden no existir en muchas computadoras.
- La **interfaz de usuario** de los sistemas de almacenamien to terciario puede ser la misma que la del sistema secun dado, es decir, las llamadas al sistema de POSIX o Win32 para manejar dispositivos de E/S. Sin embargo, es ha bitual que los sistemas operativos incluyan mandatos, tales como el tar de UNIX o el backup de Windows.
- El sistema de almacenamiento de altas prestaciones (HPSS, High Pertormance Storage Svstem) ICoine. 19991 es un sistema de E/S diseñado para proporcionar gestión de almacenamiento jerárquico, secundario y terciario, y servicios para entornos con necesidades de almacenamiento medianas y grandes.
- El hardware del reloj consiste en un temporizador pro gramable que interrumpe periódicamente, así como de un reloj alimentado por una batería que mantiene la hora cuando el equipo está apagado.
- Las principales labores del manejador de reloj son: man tenimiento de la hora, gestión de temporizadores, conta bilidad y soporte para la planificación.
- En el arranque del equipo el sistema operativo lee del reloj hardware la hora actual y la almacena internamen te. A partir de entonces, la incrementa en cada interrup ción de reloj.
- En un sistema se necesitan temporizaciones para los procesos y temporizaciones internas.
- El sistema operativo debe gestionar varios temporiza dores software usando normalmente un único tempori zador hardware.
- Dos de las labores de contabilidad típicas son el cóm puto del uso del procesador de cada proceso y la obten ción de perfiles de ejecución.
- Numerosos algoritmos de planificación de procesos re quieren información sobre el tiempo.
- Hay una gran variedad de tipos de terminales. Los dos más típicos son los terminales serie y los proyectados en memoria.
- Los terminales proyectados en memoria están formados por dos dispositivos independientes: el teclado y la pan talla.
- La entrada en los terminales proyectados en memoria está dirigida por interrupciones. El software debe encar garse de pasar del código de la tecla al carácter ASCII. así como de manejar las teclas modificadoras.
- La salida en los terminales proyectados en memoria im plica copiar información en la memoria de vídeo. No se usan interrupciones.
- Los terminales serie se comportan como un único dis positivo conectado a través de la línea serie.
- La entrada en los terminales serie está dirigida por in terrupciones. El terminal proporciona directamente el carácter ASCII.
- La salida en los terminales serie está dirigida por inte rrupciones que se producen cada vez que se transmite al terminal un carácter.

Digitalización realizada con propósito académico

- ☑ El manejador de terminal proporciona típicamente dos modos de operación: uno orientado a líneas en el que proporciona funciones de edición y otro orientado a ca rácter en el que no procesa la entrada.
- ☑ El dispositivo de red suele tener un tratamiento especí fico en la mayoría de los sistemas operativos, ofrecien do además una interfaz especial no orientada a servicios de archivos.
- ☑ El software de gestión de red suele estar estructurado en niveles: un nivel de interfaz a las aplicaciones. un nivel de protocolos y un nivel de dispositivos de red.
- ☑ Tanto POSIX como Win32 ofrecen servicios específi cos relacionados con el reloj.
- ☑ POSIX y Win32 proporcionan los mismos servicios para trabajar con dispositivos de entrada/salida que con archivos. Asimismo, ofrecen algunos servicios para realizar operaciones dependientes del dispositivo.

7.12. LECTURAS RECOMENDADAS

Para aquellos usuarios interesados en ver otros enfoques a los sistemas de E/S. se recomiendan los libros de Silberschatz [19981, Stallings IStallings. 19981 y Tanenhaum [19921. Todos ellos son libros generales de sistemas operativos que muestran una panorámica del sistema de E/S. Para una visión más detallada del sistema de E/S del sistema operativo Windows, se puede consultar el libro de Solomon [19981.

Para ampliar conocimientos sobre el terna de los manejadores de dispositivo y especialmente de los de los dispositivos de almacenamiento secundario y terciario del sistema operativo UNIX, se recomiendan los libros de Goodheart IGoodheart. 19941. Bach Bach. 19861 y lMcKusick, 19961. Para el sistema operativo LINUX se puede consultar el libro de Beck IBeck. 19981, titulado Linux Kernel internais. Para saber más

Digitalización realizada con propósito académico

acerca de los manejadores del sistema operativo Windows NT, se pueden consultar los libros de Solomon [Solomon, 19981, Nagar [Nagar, 19971 y Dekker [Dekker, 19991.

Para ver cómo es realmente el código de los manejadores de dispositivo, es recomendable el libro de sistemas operativos basado en MINIX Tanenbaum, 19971, que incluye el listado completo de este sistema operativo.

En cuanto a los servicios de entradalsalida, en [Stevens, 19921 se presentan los servicios de POSIX y en [Hart, 1997] los de Win32.

7.13. EJERCICIOS

7.1 Calcule las diferencias en tiempos de acceso entre los distintos niveles de la jerarquía de EIS. Razone la respuesta.

7.2 ¿Qué es el controlador de un dispositivo? ¿Cuáles son sus funciones?

7.3 ¿Es siempre mejor usar sistemas de E/S por inte rrupciones que programados? ¿Por qué?

7.4 ¿Qué ocurre cuando llega una interrupción de EIS? ¿Se ejecuta siempre el manejador del dispositivo? Razone la respuesta.

7.5 Imagine un sistema donde sólo hay llamadas de E/S bloqueantes. En este sistema hay 4 impresoras,3 cintas y 3 trazadores. En un momento dado hay 3 procesos en ejecución A, B y C que tienen la siguiente necesidad de recursos actual y futura:

	I	C	T	I	C	T
A	l	1	O	1	1	2
B	1	0	1	0	1	1
C	I	I	I	2	2	2

Determinar si el estado actual es seguro. Deter minar si las necesidades futuras se pueden satisfa cer, usando el algoritmo del banquero. ¿Iría todo mejor con E/S no bloqueante?

7.6 Indique dos ejemplos en los que sea mejor usar E/S no bloqueante que bloqueante.

7.7 ¿Qué problemas plantea a los usuarios la E/S blo queante? ¿Y la no bloqueante?

7.8 ¿Se le ocurre alguna estructura para el sistema de E/S distinta de la propuesta en este capítulo? Razone la respuesta.

7.9 ¿Es mejor tener un sistema de E/S estructurado por capas o monolítico? Explique las ventajas y des ventajas de ambas soluciones?

7.10 ¿En qué componentes del sistema de E/S se llevan a cabo las siguientes tareas?

a) Traducción de bloques lógicos a bloques del dispositivo.

b) Escritura de mandatos al controlador del dispositivo.

c) Traducir los mandatos de E/S a mandatos del dispositivo.

d) Mantener una cache de bloques de E/S.

7.11 ¿En que consiste el DMA? ¿Para que sirve?

Digitalización realizada con propósito académico

7.12 En un centro de cálculo tienen un disco Winchester para dar soporte a la memoria virtual de una com putadora. ¿Tiene sentido? ¿Se podría hacer lo mis mo con un disquete o una cinta? Razone la respuesta.

7.13 Suponga que un manejador de disco recibe peticio nes de bloques de disco para las siguientes pistas:2, 35, 46, 23, 90, 102, 3, 34. Si el disco tiene 150 pistas, el tiempo de acceso entre pistas consecuti vas es 4 ms y el tiempo de acceso de la pista 0 a la 150 es 8 ms. Calcule los tiempos de acceso para los algoritmos de planificación de disco SSF, FCFS, SCAN y CSCAN.

7.14 Sea un disco con 63 sectores por pista, intercalado simple de sectores, tamaño de sector de 5 12 bytes y una velocidad de rotación de 3.000 rpm. ¿Cuánto tiempo costará leer una pista completa? Tenga en cuenta el tiempo de latencia.

El almacenamiento estable es un mecanismo hard ware/software que permite realizar actualizaciones atómicas en almacenamiento secundario. ¿Cuál es el tamaño máximo de registro que este mecanismo permite actualizar atómicamente (sector, bloque. pista....)? Si la corriente eléctrica falla justo a mi tad de una escritura en el segundo disco del alma cenamiento estable, ¿cómo funciona el procedi miento de recuperación?

¿Es lo mismo un disco RAM que una cache de dis co? ¿Tienen el mismo efecto en la E/S? Razone la respuesta.

Suponga un controlador de disco SCSI. al que se pueden conectar hasta ocho dispositivos simultá neamente. El bus del controlador tiene 40 MB/scg. de ancho de banda y puede solapar operaciones de búsqueda y transferencia, es decir, puede ordenar una búsqueda en un disco y transferir datos mien tras se realiza la búsqueda. Si los discos tienen un ancho de banda medio de 2 MB/seg. y el tiempo.

Digitalización realizada con propósito académico

medio de búsqueda es de 6 mseg., calcule el má ximo número de dispositivos que el controlador podría explotar de forma eficiente si hay un 20 por 1(X) de operaciones de búsqueda y un 80 por lOO de transferencias, con un tamaño medio de 6 KB, repartidas uniformemente por los ocho dispositivos.

7.18. ¿Cuáles son las principales diferencias, desde el punto de vista del sistema operativo, entre un sis tema de copias de respaldo y un sistema de alma cenamiento terciario complejo como HPSS?

7.19. Suponga que los dispositivos extraíbles, como las cintas, fueran tan caros como los discos. ¿Tendría sentido usar estos dispositivos en la jerarquía de almacenamiento?

7.20. Suponga un gran sistema de computación al que se ha añadido un sistema de almacenamiento ter ciario de alta capacidad. ¿De qué forma se puede saber si los archivos están en el sistema secundario o en el terciario? ¿Se podrían integrar todo el árbol de nombres? Razone la respuesta.

7.2 1. En algunos sistemas, como por ejemplo Linux, se almacena en una variable el número de interrup ciones de reloj que se han producido desde el arra nque del equipo, devolviéndolo en la llamada ti mes. Si la frecuencia de reloj es de lOO Hz y se usa una variable de 32 bits, ¿cuánto tiempo tardará en deshordarse ese contador? ¿Qué consecuencia puede tener ese desbordamiento?

7.22. ¿Qué distintas cosas puede significar que una fun ción obtenga un valor elevado en un perfil de eje cución de un programa?

7.23. Proponga un método que permita a un sistema ope rativo proporcionar un servicio que ofrezca tempo rizaciones de una duración menor que la resolu ción del reloj.

7.24. Algunos sistemas permiten realizar perfiles de eje cución de! propio sistema operativo. ¿De qué par tes del código del sistema operativo no podrán ob tener perfiles?

7.25. Escriba el pseudocódigo de una rutina de interrup ción de reloj.

7.26. Suponga un sistema que no realiza la gestión de temporizadores directamente desde la rutina deinterrupción sino desde una rutina que ejecuta con menor prioridad que las interrupciones de todos los dispositivos. ¿En qué situaciones puede tomar un valor negativo el contador de un temporiza dor?

7.27. Analice para cada uno de estos programas si en su consumo del procesador predomina el tiempo gas tado en modo usuario o en modo sistema.

• Un compilador.

• Un programa que copia un archivo.

• Un intérprete de mandatos que sea un programa que comprime un archivo.

• Un programa que resuelve una compleja fórmula matemática.

7.28. Enumere algunos ejemplos de situaciones proble máticas que podrían ocurrir si un usuario cambia la hora del sistema retrasándola. ¿Podría haber pro blemas si el cambio consiste en adelantarla?

7.29. Escriba el pseudocódigo de las funciones de lectu ra, escritura y manejo de interrupciones para un terminal serie.

7.30. Lo mismo que el ejercicio anterior pero en e! caso de un terminal proyectado en memoria.

7.31. Enumere ejemplos de tipos de programas que re quieran que el terminal esté en modo carácter (modo crudo).

7.32. Escriba un programa usando servicios POSIX que lea un único carácter del terminal.

7.33. Realice el mismo programa utilizando servicios Win32.

Digitalización realizada con propósito académico

7.34. ¿Por qué cree que es conveniente en los terniina!es serie copiar los datos que se desean escribir desde la memoria del proceso a una zona de almacena miento intermedio en vez de enviarlos directanien te desde allí?

7.35. Analice si es razonable permitir que múltiples pro cesos puedan leer simultáneamente del mismo ter minal. ¿Y escribir?

7.36. Proponga métodos para intentar reducir e! máximo posible el número de veces que se copia un men saje tanto durante su procesamiento en la máquina emisora como en la máquina receptora.

Digitalización realizada con propósito académico

8

Gestión de archivos y directorios

En este capítulo se presentan los conceptos relacionados con archivos y directorios. El capí tulo tiene tres objetivos básicos: mostrar al lector dichos conceptos desde el punto de vista de usuario, los servicios que da el sistema operativo y los aspectos de diseño de los sistemas de archivos y del servidor de archivos. De esta forma se pueden adaptar los contenidos del tema a distintos niveles de conocimiento.

Para alcanzar este objetivo el capítulo se estructura en las siguientes grandes secciones:

- Visión de usuario del sistema de archivos.
- Archivos.
- Directorios.
- Servicios de archivos y directorios.
- Sistemas de archivos.
- El servidor de archivos.

Digitalización realizada con propósito académico

8.1. VISIÓN DE USUARIO DEL SISTEMA DE ARCHIVOS

Desde el punto de vista de los usuarios y las aplicaciones, los archivos y directorios son los elemen tos centrales del sistema. Cualquier usuario genera y usa información a través de las aplicaciones que ejecuta en el sistema. En todos los sistemas operativos de propósito general, las aplicaciones y sus datos se almacenan en archivos no volátiles, lo que permite su posterior reutilización. La visión que los usuarios tienen del sistema de archivos es muy distinta de la que tiene el sistema operativo en el ámbito interno. Como se muestra en la Figura 8.1, los usuarios ven los archivos como un conjunto de información estructurada según sus necesidades o las de sus aplicaciones, mientras que el sistema operativo los contempla como conjuntos de datos estructurados según sus necesidades de almacenamiento y representación. Además, puede darse la circunstancia de que distintos usuarios vean un mismo archivo de forma distinta [19811. Un archivo de empleados, por ejemplo, puede tratarse como un conjunto de registros indexados o como un archivo de texto sin ninguna estructura. Cuando en un sistema existen múltiples archivos, es necesario dotar al usuario de algún mecanismo para estructurar el acceso a los mismos. Estos mecanismos son los directorios, agrupa ciones lógicas de archivos que siguen criterios definidos por sus creadores o manipuladores. Para facilitar el manejo de los archivos, casi todos los sistemas de directorios permiten usar nombres lógicos, que, en general, son muy distintos de los descriptores físicos que usa internamente el sistema operativo.

Cualquiera que sea la visión de los archivos, para los usuarios su característica principal es que no están ligados al ciclo de vida de una aplicación en particular. Un archivo y un directorio tienen su propio ciclo de vida. Para gestionar estos objetos, todos los sistemas operativos de propósito general incluyen servicios de archivos y directorios [19961, junto con programas de utilidad que facilitan el uso de sus servicios [19841 y Horspool, 19921. El servidor de archivos es la parte del sistema operativo que se ocupa de facilitar el manejo de los dispositivos periféricos, ofreciendo una visión lógica simplificada de los mismos en forma de archivos. Median te esta visión lógica se ofrece a los usuarios un mecanismo de abstracción que oculta todos los detalles relacionados con el almacenamiento y distribución de la información en los dispositivos periféricos, así como el funcionamiento de los mismos. Para ello se encarga de la organización, almacenamiento, recuperación, gestión de nombres, coutilización y protección de los archivos de un sistema.

8.2. ARCHIVOS

Las aplicaciones de una computadora necesitan almacenar información en soporte permanente, tal como discos o cintas. Dicha información, cuyo tamaño puede variar desde unos pocos bytes hasta

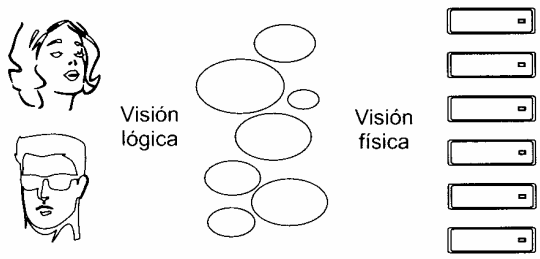


Figura 8.1. *Visión lógica y física de un archivo.*

Digitalización realizada con propósito académico

terabytes, puede tener accesos concurrentes desde varios procesos. Además, dicha información tiene su ciclo de vida que normalmente no está ligado a una única aplicación.

En esta sección se va a estudiar la visión externa del sistema de archivos. Para ello se mostrará el concepto de archivo, la gestión de nombres de archivos, las estructuras de archivo posibles, las semánticas de acceso, las semánticas de coutilización y los servicios que proporcionan los sistemas operativos para gestionar archivos.

8.2.1. Concepto de archivo

Todos los sistemas operativos proporcionan una unidad de almacenamiento lógico, que oculta los detalles del sistema físico de almacenamiento, denominada **archivo**. Un archivo es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de información relacionada entre sí ha jo un mismo nombre I 1973]. Desde el punto de vista del usuario, el archivo es la única forma de gestionar el almacenamiento secundario, por lo que es importante en un sistema operativo definir cómo se nombran los archivos, qué operaciones hay disponibles sobre los archivos, cómo perciben los usuarios los archivos, etc. Internamente, todos los sistemas operativos dedican una parte de sus funciones, agrupada en el **sistema de archivos**, a gestionar los archivos. En este componente del sistema operativo se define cómo se estructuran los archivos, cómo se identifican, cómo se implementan, acceden, protegen, etc.

Desde el punto de vista del usuario, el sistema de archivos es la parte más visible del sistema operativo ya que a través de él accede a los programas y los datos almacenados en archivos de distinto tipo (código fuente, programas objeto, bibliotecas, programas ejecutables, texto ASCII, etcétera) agrupados en directorios. Para el usuario, los archivos son contenedores de información de un tipo definido por su creador, aunque todos ellos se pueden agrupar en dos grandes clases: **archivos ASCII y archivos binarios**. Los archivos ASCII, formados por líneas de texto, pueden ser editados o impresos directamente, cosa que no suele ocurrir con archivos binarios que suelen alma cenar archivos ejecutables, objetos y datos no textuales. En el sistema operativo UNIX [Ritchie, 19741 existe un tipo peculiar de archivos, denominado **archivos especiales**[McKusick, 19961, que permiten modelar cualquier dispositivo de E/S como un archivo más del sistema. Los archivos especiales pueden serlo de caracteres (para modelar terminales, impresoras, etc.) o de bloques (para modelar discos y cintas).

Desde el punto de vista del sistema operativo, un archivo se caracteriza por tener una serie de atributos Smith, 19941. Dichos atributos varían de unos sistemas operativos a otros, pero habitual mente incluyen los siguientes:

- **Nombre**: identificador del archivo en formato comprensible para el usuario. Definido por su creador.
- **Identificador único**: en el ámbito interno, el sistema operativo no usa el nombre para iden tificar a los archivos, sino un identificador único fijado con criterios internos del sistema operativo. Este identificador suele ser un número y habitualmente es desconocido por los usuarios.
- **Tipo de archivo**: útil en aquellos sistemas operativos que soportan tipos de archivos en el ámbito interno. En algunos casos el tipo de archivo se identifica por el denominado número mágico, una etiqueta del sistema operativo que le permite distinguir entre distintos formatos de almacenamiento de archivos.
- **Mapa del archivo**: formado normalmente por apuntadores a los dispositivos, y a los blo ques dentro de éstos, que albergan el archivo. Esta información ‘se utiliza para localizar el dispositivo y los bloques donde se almacena la información que contiene el archivo.

Digitalización realizada con propósito académico

- **Protección**: información de control de acceso que define quién puede hacer qué sobre el archivo, la palabra clave para acceder al archivo, el dueño del archivo, su creador, etc.
- **Tamaño del archivo**: número de bytes en el archivo, máximo tamaño posible para el archi vo, etc.
- **Información temporal:** tiempo de creación, de último acceso, de última actualización, etc. Esta información es muy útil para gestionar, monitorizar y proteger los sistemas de archivos.
- **Información de control del archivo**: que indica si es un archivo oculto, de sistema, normal o directorio, cerrojos, etc.

El sistema operativo debe proporcionar, al menos, una estructura de archivo genérico que dé soporte a todos los tipos de archivos mencionados anteriormente, un mecanismo de nombrado. facilidades para proteger los archivos y un conjunto de servicios que permita explotar el almacena miento secundario y el sistema de E/S de forma sencilla y eficiente [1987]. Dicha estructura debe incluir los atributos deseados para cada archivo, especificando cuáles son visibles y cuáles están ocultos a los usuarios. La Figura 8.2 muestra tres formas de describir un archivo: nodo-i de UNIX, registro de MFT en Windows NT y entrada de directorio de MS-DOS. Todas ellas sirven como mapas para acceder a la información del archivo en los dispositivos de almacenamiento.

El **nodo-i** de UNIX [Bach, 1986] contiene información acerca del propietario del archivo, de su grupo, del modo de protección aplicable al archivo, del número de enlaces al archivo, de valores de fechas de creación y actualización, el tamaño del archivo y el tipo del mismo. Además incluye un mapa del archivo mediante apuntadores a bloques de dispositivo que contienen datos del archi vo. A través del mapa de bloques del nodo-i se puede acceder a cualquiera de sus bloques con un número muy pequeño de accesos a disco. Cuando se abre un archivo, su nodo-i se trae a memoria. En este proceso se incrementa la información del nodo-i almacenada en el disco con datos referen tes al uso dinámico del mismo, tales como el dispositivo en que está almacenado y el número de veces que el archivo ha sido abierto por los procesos que lo están usando.

El **registro de MFT** de Windows NT [Solomon, 1998] describe el archivo mediante los si guientes atributos: cabecera, información estándar, descriptor de seguridad, nombre de archivo y datos (Fig. 8.2). A diferencia del nodo-i de UNIX, un registro de Windows NT permite almacenar hasta 1,5 KB de datos del archivo en el propio registro, de forma que cualquier archivo menor de ese tamaño debería caber en el registro. Si el archivo es mayor, dentro del registro se almacena infor mación del mapa físico del archivo incluyendo punteros a grupos de bloques de datos (Vclusters),

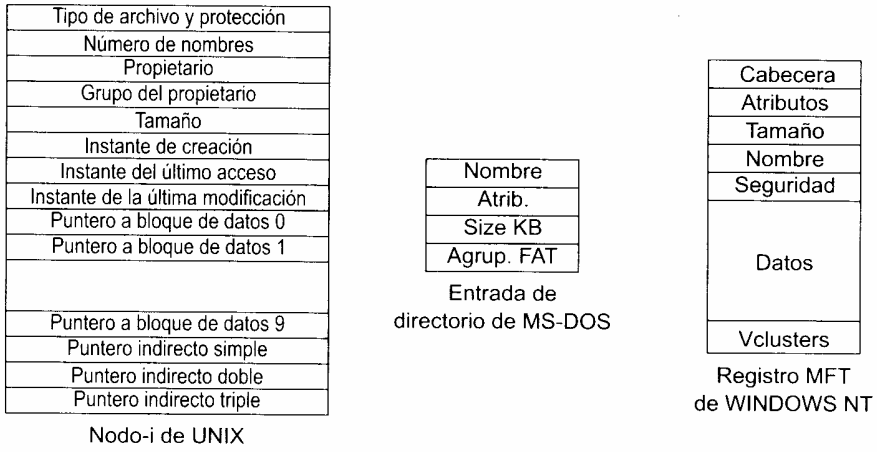


Figura 8.2. *Distintas formas de representar un archivo.*

Digitalización realizada con propósito académico

cada uno de los cuales incluye a su vez datos y punteros a los siguientes grupos de bloques. Cuando se abre el archivo, se trae el registro a memoria. Si es pequeño, ya se tienen los datos del archivo. Si es grande hay que acceder al disco para traer bloques sucesivos. Es interesante resaltar que en este sistema todos los accesos a disco proporcionan datos del archivo, cosa que no pasa en UNIX. donde algunos accesos son sólo para leer información de control.

En el caso de **MS-DOS** [Patterson, 19891, la representación del archivo es bastante más senci lla debido principalmente a ser un sistema operativo monoproceso y monousuario. En este caso, la información de protección no existe, limitándose a unos atributos mínimos que permiten ocultar el archivo o ponerlo como de sólo lectura. El nombre se incluye dentro de la descripción, así como los atributos básicos y el tamaño del archivo en KB. Además, se especifica la posición del inicio del archivo en la tabla FAT (file allocation table), donde se almacena el mapa físico del archivo.

8.2.2. Nombres de archivos

Una de las características principales de un sistema operativo, de cara a los usuarios, es la forma en que se nombran los archivos. Todo objeto archivo debe tener un nombre a través del que se pueda acceder a él de forma inequívoca [19731. Por ello, todos los sistemas operativos propor cionan mecanismos de nombrado que permiten asignar un nombre a un archivo en el momento de su creación. Este nombre acompañará al objeto mientras exista y le identificará de forma única.

El tipo de nombres que se usan para los archivos varía de un sistema operativo a otro. Sin embargo, y para ser más amigables con los usuarios, todos ellos permiten asignar a los archivos nombres formados por combinaciones de caracteres alfanuméricos y algunos caracteres especiales. La longitud de los nombres puede ser variable. Por ejemplo, MS-DOS permite nombres con una longitud máxima de ocho caracteres, mientras UNIX permite nombres de hasta 4.096 caracteres. Asimismo, algunos sistemas operativos, como MS-DOS o Windows, no distinguen entre mayúscu las y minúsculas, mientras otros, como UNIX, sí lo hacen. Por las razones anteriores, un nombre de archivo como CATALINAPEREZ no sería válido en MS-DOS pero sí en UNIX y los nombres CATALI - NA y catalina denominarían al mismo archivo en MS-DOS pero a dos archivos distintos en UNIX.

Muchos sistemas operativos permiten añadir una o más **extensiones** al nombre de un archivo. Dichas extensiones se suelen separar del nombre con un punto (p. ej.: . txt) y sirven para indicar al sistema operativo, a las aplicaciones, o a los usuarios, características del contenido del archivo [19981. Las extensiones de los archivos tienen formato y significado muy distinto para cada sistema operativo. En MS-DOS, por ejemplo, un archivo sólo pueden tener una extensión y ésta debe ser una etiqueta de tres letras como máximo. En UNIX y Windows NT, un archivo puede tener cualquier número de extensiones y de cualquier tamaño. La Figura 8.3 muestra algunas de las extensiones más habituales en los nombres de archivos, tales como pdf, PS o dvi para archivos ASCII en krmato visible o imprimible, zip para archivos comprimidos, c o cpp para código fuente en C y C++ respectivamente. htm o html para hipertexto, etc. Como se puede ver, Windows asocia a cada tipo de archivo el icono de la aplicación que lo puede manejar (si conoce el tipo de archivo y la aplicación).

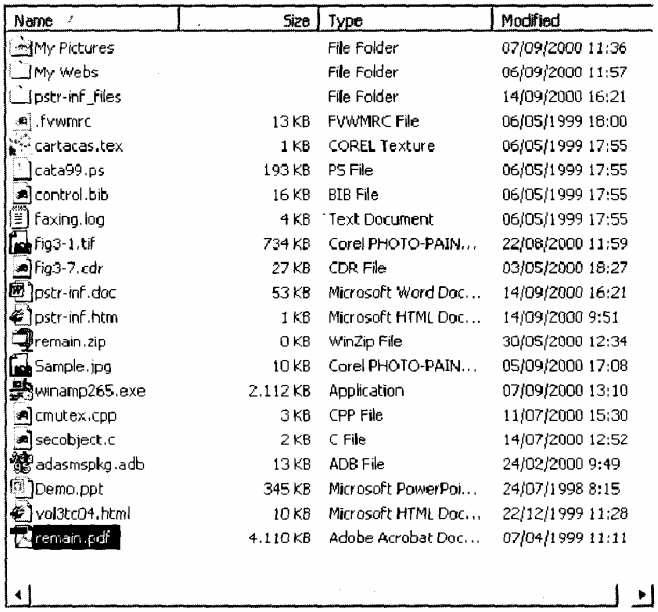
Habitualmente, las extensiones son significativas sólo para las aplicaciones de usuario. pero

existen casos en que el sistema operativo reconoce, y da soporte específico. a distintos tipos de archivos según sus extensiones. Ambos enfoques tienen sus ventajas e inconvenientes:

• Si el sistema operativo reconoce tipos de archivos, puede proporcionar utilidades específicas y muy optimizadas para los mismos. Sin embargo, el diseño del sistema se complica mucho.

• Si no los reconoce, las aplicaciones deben programar las utilidades específicas para manejar su tipo de archivos. Pero el diseño interno del sistema operativo es mucho más sencillo.

Digitalización realizada con propósito académico



Name	Size	Type	Modified
My Pictures		File Folder	07/09/2000 11:36
My Webs		File Folder	06/09/2000 11:57
pstr-inf_files		File Folder	14/09/2000 16:21
.fvwmrc	13 KB	FVWMRC File	06/05/1999 18:00
cartacas.tex	1 KB	COREL Texture	06/05/1999 17:55
cata99.ps	193 KB	PS File	06/05/1999 17:55
control.bib	16 KB	BIB File	06/05/1999 17:55
faxing.log	4 KB	Text Document	06/05/1999 17:55
fig3-1.tif	734 KB	Corel PHOTO-PAIN...	22/08/2000 11:59
fig3-7.cdr	27 KB	CDR File	03/05/2000 18:27
pstr-inf.doc	53 KB	Microsoft Word Doc...	14/09/2000 16:21
pstr-inf.htm	1 KB	Microsoft HTML Doc...	14/09/2000 9:51
remain.zip	0 KB	WinZip File	30/05/2000 12:34
Sample.jpg	10 KB	Corel PHOTO-PAIN...	05/09/2000 17:08
winamp265.exe	2.112 KB	Application	07/09/2000 13:10
cmutex.cpp	3 KB	CPP File	11/07/2000 15:30
secobject.c	2 KB	C File	14/07/2000 12:52
adasmspkg.adb	13 KB	ADB File	24/02/2000 9:49
Demo.ppt	345 KB	Microsoft PowerPoi...	24/07/1998 8:15
vol3tc04.html	10 KB	Microsoft HTML Doc...	22/12/1999 11:28
remain.pdf	4.110 KB	Adobe Acrobat Doc...	07/04/1999 11:11

Figura 8.3. *Extensiones frecuentes en los nombres de archivo.*

Un ejemplo de la primera opción fue el sistema operativo TOPS-20, que proporcionaba distin tos tipos de archivos en su ámbito interno y ejecutaba operaciones de forma automática dependien do del tipo de archivo. Por ejemplo, si se modificaba un archivo con código fuente Pascal, y exten sión .pas, el sistema operativo recompilaba automáticamente la aplicación creando una nueva versión de la misma. La segunda opción tiene un exponente claro en UNIX, sistema operativo para el que las extensiones del nombre no son significativas, aunque sí lo sean para las aplicaciones externas. Por ello, UNIX no almacena entre los atributos de un archivo ninguna información acerca del tipo del mismo, ya que este sistema operativo proporciona un único tipo de archivo cuyo mode lo se basa en una tira secuencial de bytes. Unicamente distingue algunos formatos, como los archi vos ejecutables, porque en el propio archivo existe una cabecera donde se indica el tipo del mismo mediante un número, al que se denomina **número mágico**. Por supuesto, esta característica está totalmente oculta al usuario. Por ejemplo, un compilador de lenguaje C [Kernighan, 1978] puede necesitar nombres de archivos con la extensión . c y la aplicación compress puede necesitar nom bres con la extensión . z. Sin embargo, desde el punto de vista del sistema operativo no existe ninguna diferencia entre ambos archivos. Windows tampoco es sensible a las extensiones de archi vos, pero sobre l existen aplicaciones del sistema (como el explorador) que permiten asociar dichas extensiones con la ejecución de aplicaciones. De esta forma, cuando se activa un archivo con el ratón, se lanza automáticamente la aplicación que permite trabajar con ese tipo de archivos.

8.2.3. Estructura de un archivo

Los archivos pueden estructurarse de formas distintas Folk, 1987], tanto en el ámbito de usuario como del sistema operativo. Desde el punto de vista del sistema operativo, algunos archivos (como los ejecutables o las bilbiotecas dinámicas) deben tener una cierta estructura para que su informa-

Digitalización realizada con propósito académico

ción pueda ser interpretada. Desde el punto de vista del usuario, la información de un archivo puede estructurarse como una lista de caracteres, un conjunto de registros secuencial o indexado, etcétera [Stal1ings, 1988]. Desde el punto de vista de algunas aplicaciones, como por ejemplo un compilador, un archivo de biblioteca está formado por una serie de módulos acompañados por una cabecera descriptiva para cada módulo. La Figura 8.4 muestra algunas de estas estructuras de archivo.

Una cuestión distinta es si el sistema operativo debe proporcionar funcionalidad para múltiples estructuras de archivos en el ámbito interno. A priori, un soporte más extenso facilita las tareas de programación de las aplicaciones. Sin embargo, esta solución tiene dos serios inconvenientes:

• La interfaz del sistema operativo se complica en exceso.

• El diseño y la programación del propio sistema operativo son mucho más complejos.

Es por ello que los sistemas operativos más populares, como UNIX o Windows, proporcionan una estructura interna de archivo y una interfaz de programación muy sencillos pero polivalentes, permitiendo a las aplicaciones construir cualquier tipo de estructura para sus archivos sin que el sistema operativo sea consciente de ello. Con la estructura de archivo que proporciona POSIX, una tira secuencial de bytes, cualquier byte del archivo puede ser accedido directamente si se conoce su desplazamiento desde el origen del archivo. La traslación desde estas direcciones lógicas a direc ciones físicas de los dispositivos que albergan el archivo es distinta en cada sistema operativo, pero se basa en el mapa del archivo almacenado como parte de los atributos del archivo. La Figura 8.5 muestra la estructura lógica y física de un archivo en el sistema operativo UNIX.

Todos los sistemas operativos deben reconocer al menos un tipo de archivo: sus propios archi vos ejecutables. La estructura de un archivo ejecutable está íntimamente ligada a la forma de gestio nar la memoria y la E/S en un sistema operativo. En algunos sistemas, como VMS Kenah, 19881, los archivos ejecutables se reconocen por su extensión. En otros, como en UNIX, porque así se

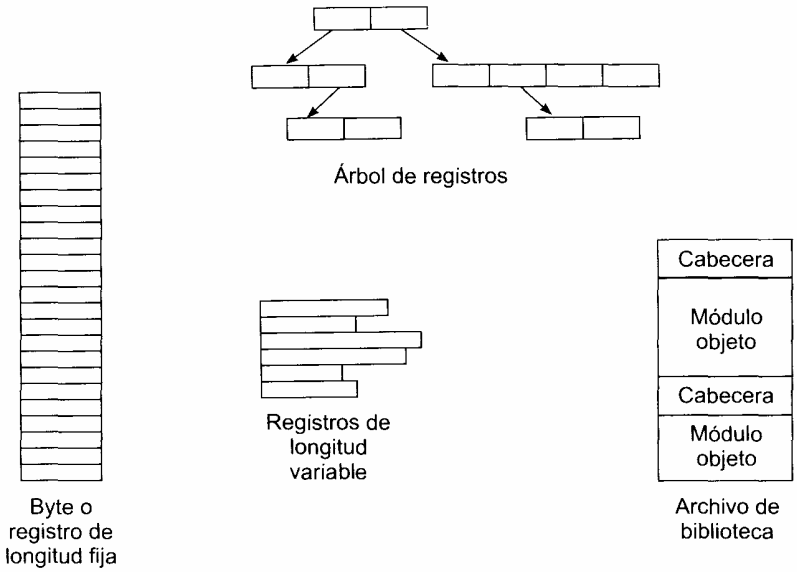


Figura 8.4. *Ejemplos de estructuras de archivo.*

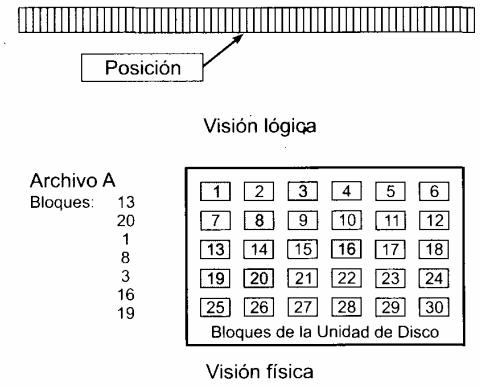


Figura 8.5. *Estructura lógica y física de un archivo.*

indica en el número mágico que identifica el tipo de archivo. Aunque existen distintos fii de archivos ejecutables, como el ELF, la Figura 8.6 muestra un ejemplo simplificado de estructura de un archivo ejecutable en el sistema operativo UNIX IChristian, 19881. Como puede verse, el siste ma operativo espera que dicho formato incluya, al menos, cinco secciones: cabecera, texto, datos, información de carga y tabla de símbolos. Dentro de la cabecera está la información necesaria para que el sistema operativo pueda interpretar la estructura del ejecutable y encontrar cada uno de sus elementos (número mágico, tamaño de las otras secciones, opciones de ejecución, etc.). Cada siste ma operativo usa un número mágico característico para reconocer sus ejecutables.

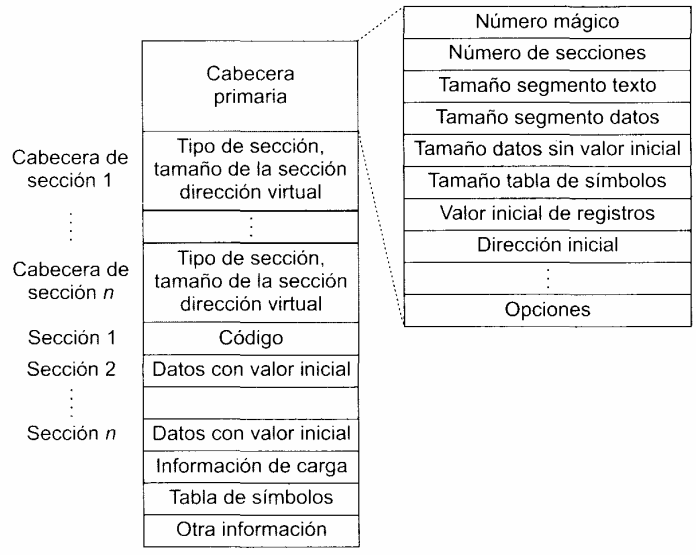


Figura 8.6. *Estructura de un archivo ejecutable en UNIX.*

Digitalización realizada con propósito académico

8.2.4. Métodos de acceso

Para poder utilizar la información almacenada en un archivo, las aplicaciones deben acceder a la misma y almacenarla en memoria. Hay distintas formas de acceder a un archivo [l9 aunque no todos los sistemas operativos proporcionan todas ellas. Para una aplicación, elegir ade cuadamente la forma de acceder a un archivo suele ser un aspecto importante de diseño, ya que, en muchos casos, el método de acceso tiene un impacto significativo en el rendimiento de la misma. Dependiendo de que se pueda saltar de una posición a otra de un archivo, se distinguen dos métodos de acceso principales: acceso secuencial y acceso directo.

Cuando se usa el método de **acceso secuencial**, lo único que se puede hacer es leer los bytes del archivo en orden, empezando por el principio. No puede saltar de una posición del archivo a otra o leerlo desordenado. Si se quiere volver atrás, hay que volver al principio y releer todo el archivo hasta el punto deseado. Las operaciones más comunes son lecturas y escrituras. En una lectura, un proceso lee una porción del archivo, como resultado de la cual se actualiza el apuntador de posición de dicho archivo a la siguiente posición de lectura. En una escritura, un proceso puede añadir información al archivo y avanzar hasta el final de los datos nuevos. Este modo de acceso. muy fácil de implementar, se usa muy frecuentemente en aplicaciones que necesitan leer los archi vos completos y de forma contigua, como ocurre con editores y compiladores. La forma de acceso secuencial se basa en un modelo de archivo almacenado en cinta magnética, pero funciona igual mente bien en dispositivos de acceso directo. El principal problema de este método es su falta de flexibilidad cuando hay que acceder de forma no secuencial a los datos. Para tratar de paliarlo, el sistema MVS incluye archivos ISAM (Indexed Sequential Access Method), es decir, archivos se cuenciales a partir de los cuales se crea un índice que indica la posición de determinados elementos y que se usa para acceder a ellos rápidamente sin tener que estudiar todos los que hay delante de ellos. Imagine que tiene un archivo de clientes ordenado por orden alfabético y quiere buscar a Juan y a Pepe. Deberá leer todos los nombre y ver si son ellos. Con el método ISAM existe un archivo de índice que le dice, por ejemplo, dónde empiezan los clientes cuyo nombre comienza por A, por B, etc. Para leer los datos de Juan puede ir directamente a la posición del índice para la J y luego comparar los datos para buscar Juan dentro de ese bloque. ¡Observe que, en cualquier caso, es necesario leer los datos hasta Juan!

Con la llegada de los dispositivos de acceso directo (como los discos magnéticos), surgió la forma de **acceso directo**, o **aleatorio** [London, 19731, a un archivo. El archivo se considera como un conjunto de registros, cada uno de los cuales puede ser un byte. Se puede acceder al mismo desordenadamente moviendo el apuntador de acceso al archivo a uno u otro registro. Esta forma de acceso se basa en un modelo de archivo almacenado en disco, ya que se asume que el dispositivo se puede mover aleatoriamente entre los distintos bloques que componen el archivo. Para proporcio nar este método de acceso a las aplicaciones, los sistemas operativos incluyen llamadas del sistema de archivos con las que se puede modificar la posición dentro del archivo (seek) o en las que se puede especificar el número de registro o bloque a leer o escribir, normalmente relativos al princi pio del archivo. El sistema operativo relaciona estos números relativos con números absolutos en los dispositivos físicos de almacenamiento. Este método de acceso es fundamental para la imple mentación de muchas aplicaciones.. Las bases de datos, por ejemplo, usan fundamentalmente archi vos de este tipo. Imagine que tiene la lista de clientes anterior en un dispositivo que permite acceso directo. En lugar de leer los datos hasta Juan basta con leer la posición en el índice y saltar hasta dicha posición. Actualmente, todos los sistemas operativos usan esta forma de acceso, mediante la cual se puede también acceder secuencialmente al archivo de forma muy sencilla, Igualmente, sobre sistemas de acceso directo se pueden construir fácilmente otros métodos de acceso como los de índice, registros de tamaño fijo o variable, etc. (Prestaciones 8. 1).

Digitalización realizada con propósito académico

PRESTACIONES 8.1

Seleccionar adecuadamente la forma de acceso a un archivo influye mucho en el rendimiento de las operaciones sobre archivos. Por ejemplo, usar accesos secuenciales en una base de datos cuyas operaciones se hacen en el ámbito de los registros sería pésimo. Sin embargo, usar accesos secuenciales para copiar un archivo sobre otro sería óptimo.

8.2.5. Semánticas de coutilización

El uso de cualquiera de las formas de acceso anteriores no resuelve el problema del uso concurrente de un archivo, sobre todo en el caso de que alguno de los procesos que lo accede esté modificando la información existente en dicho archivo. En situaciones de **coutilización** de un archivo, las accio nes de un proceso pueden afectar a la visión que los otros tienen de los datos o a los resultados de su aplicación. La **semántica de coutilización** [Levy, 19901 especifica qué ocurre cuando varios proce sos acceden de forma simultánea al mismo archivo y especifica el momento en el que las modifica ciones que realiza un proceso sobre un archivo pueden ser observadas por los otros. Es necesario que el sistema operativo defina la semántica de coutilización que proporciona para que las aplica ciones puedan estar seguras de que trabajan con datos coherentes. Piense qué ocurriría si varios usuarios modificasen simultáneamente los mismos registros de reservas de un tren sin ningún con trol. El estado final de la base de datos de reservas sería totalmente impredecible. A continuación se describen las tres semánticas de coutilización más clásicas en los sistemas operativos actuales: semántica de UNIX, de sesión y de archivos inmutables.

La **semántica de UNIX** consiste en que cualquier lectura de archivo vea los efectos de todas las escrituras previas sobre ese archivo. En caso de accesos concurrentes de lectura, se deben obte ner los mismos datos. En caso de accesos concurrentes de escritura, se escriben los datos en el orden

de llegada al sistema operativo, que puede ser distinto al que piensan los usuarios. Con esta semán- 8 tica cada proceso trabaja con su propia imagen del archivo y no se permite que los procesos inde pendientes compartan el apuntador de posición dentro de un archivo. Este caso sólo es posible para procesos que heredan los archivos a través del servicio fork. En este caso, la coutilización no sólo afecta a los datos, sino también a los metadatos del archivo, ya que una operación de lectura o escritura modificará el apuntador de posición de acceso al archivo. Si los usuarios quieren estar seguros de que los datos se escriben sin concurrencia, o en el orden correcto, deben usar cerrojos para bloquear los accesos al archivo.

El principal problema de la semántica de UNIX es la sobrecarga que genera en el sistema operativo, que debe usar cerrojos a nivel interno, para asegurar que las operaciones de un usuario no modifican los metadatos de otros y que las operaciones de escritura de los usuarios se realizan en su totalidad. Para relajar esta política se definió la **semántica de sesión** [Walker, 19831, que permite que las escrituras que realiza un proceso sobre un archivo se hagan sobre su copia y que no sean visibles para los demás procesos que tienen el archivo abierto hasta que se cierre el archivo o se emita una orden de actualización de la imagen del archivo. Sólo en esas situaciones los cambios realizados se hacen visibles para futuras sesiones. El principal problema de esta semántica es que cada archivo tiene temporalmente varias imágenes distintas, denominadas **versiones**. Imagine que se están modificando los datos de los clientes de una empresa simultáneamente en varias sedes de la misma. Cuando se termina la actualización en cada una de ellas se pide actualizar la versión defini tiva. En caso de que haya procesos que han trabajado ya con el mismo cliente, se pueden encontrar con que su copia está obsoleta sin saberlo. Es más, pueden escribir su copia después y guardar los

Digitalización realizada con propósito académico

datos obsoletos. Por ello, en caso de que un proceso necesite acceder a los datos que escribe otro proceso, ambos deben sincronizarse explícitamente abriendo y cerrando el archivo, o forzando actualizaciones de la versión, lo que puede ser poco eficiente en aplicaciones formadas por un conjunto de procesos cooperantes. Este tipo de semánticas se utilizan en algunos sistemas de archi vos con versiones de archivos y en sistemas de archivos distribuidos (Aclaración 8. 1).

ACLARACIÓN 8.1

En un sistema de archivos distribuido se puede acceder a archivos situados en máquinas remotas conectadas a través de una red de interconexión, lo que complica extraordinariamente el problema de las versiones.

La tercera semántica de coutilización, la semántica de archivos inmutables[Shroeder, 1985], se basa en que cuando un archivo es creado por su dueTio no puede se n más. Para escribir algo en el archivo es necesario crear uno nuevo y escribir en él los datos. Un archivo inmutable se caracteriza por dos propiedades: su nombre no puede ser reutilizado sin borrar previamente el archivo y su contenido no puede ser alterado. El nombre del archivo se encuentra indisolublemente ligado al contenido del mismo. Por tanto, esta semántica solo permite accesos concurrentes de lectura. Para optimizar la implementación de esta semántica se usa una tecnica similar al copy-on-write de la memoria. Con esta tecnica, el nuevo archivo tiene el mismo mapa que el anterior en su descriptor. Cada vez que se modifica la información de un bloque, se le asigna un nuevo bloque y se sustituye el identificador del bloque anterior por el del nuevo bloque en el mapa de este archivo. Ademas, cuando los sistemas operativos usan esta semántica, suelen tener una forma de crear automáticamente los nombres de las nuevas versiones de un archivo (como, por ejemplo, añadir un entero creciente al nombre).

8.3. DIRECTORIOS

Un sistema de archivos puede ser muy grande. Para poder acceder a los archivos con facilidad, todos los sistemas operativos proporcionan formas de organizar los nombres de archivos mediante **directorios**. Un directorio puede almacenar otros atributos de los archivos tales como ubicación, propietario, etc., dependiendo de cómo haya sido diseñado. Habitualmente, los directorios se implementan como archivos, pero se tratan de forma especial y existen servicios especiales del sistema operativo para su manipulación. En esta sección se va a estudiar el concepto de directorio, las estructuras de directorios más comunes, su relación con los nombres de archivo y las formas más frecuentes de construir las jerarquías de directorios.

8.3.1. Concepto de directorio

Un **directorio** es un objeto que relaciona deforma unívoca el nombre de usuario de un archivo y el kia descriptor interno del mismo usado por el sistema operativo [Chapin, 1969]. Los directorios sirven para organizar y proporcionar información acerca de la estructuración de los archivos en los sistemas de archivos. Para evitar ambigüedades, un mismo nombre no puede identificar nunca a dos archivos distintos, aunque varios nombres se pueden referir al mismo archivo. Habitualmente, un

Digitalización realizada con propósito académico

directorio contiene tantas entradas como archivos son accesibles a través de él, siendo la función principal de los directorios presentar una **visión lógica** simple al usuario, escondiendo los detalles de implementación del sistema de directorios. Un ejemplo de visión lógica del esquema de directo rios de Windows NT, en este caso un esquema jerárquico, puede verse en la Figura 8.7, donde los archivos se representan mediante iconos gráficos y los directorios se representan mediante carpetas. Esta representación permite visualizar gráficamente la estructura de los directorios a través de aplicaciones del sistema, en este caso Microsoft Explorer. Además permiten visualizar con prefe rencia ciertos tipos de archivos, como ejecutables o archivos de procesadores de textos, y asociarlos con aplicaciones que se ejecutan de forma automática cuando se pulsa con el ratón sobre dichos archivos.

Cuando se abre un archivo, el sistema operativo busca en el sistema de directorios hasta que encuentra la entrada correspondiente al nombre del archivo. A partir de dicha entrada, el sistema operativo obtiene el identificador interno del archivo y, posiblemente, algunos de los atributos del mismo. Esta información permite pasar del nombre de usuario al objeto archivo que maneja el sistema operativo. Todas las referencias posteriores al archivo se resuelven a través de su descriptor (nodo-i, registro MFT, etc.).

Al igual que un archivo, un directorio es un objeto y debe ser representado por una estructura de datos. Una cuestión importante de diseño es decidir qué información debería ir asociada a una entrada del directorio. La Figura 8.8 muestra varias entradas de directorio usadas en sistemas opera tivos. Como puede verse en la figura, hay dos alternativas principales:

- Almacenar los atributos del archivo en su entrada del directorio.
- Almacenar únicamente el identificador del descriptor de archivo (en UNIX, el nodo-i) y colocar los atributos del objeto archivo dentro de la estructura de datos asociada a su des criptor.

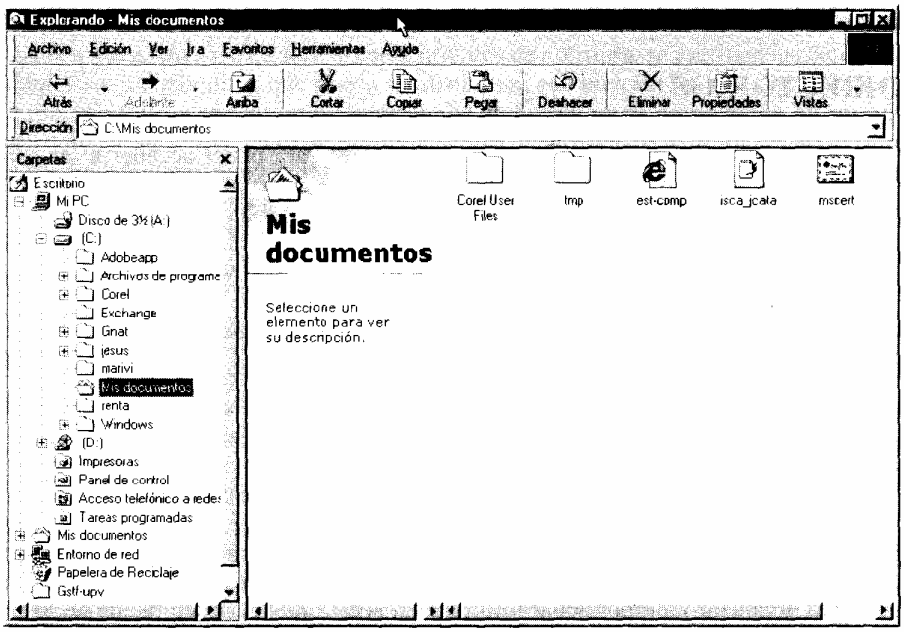


Figura 8.7. *Visión lógica de los directorios en Windows NT.*

Digitalización realizada con propósito académico

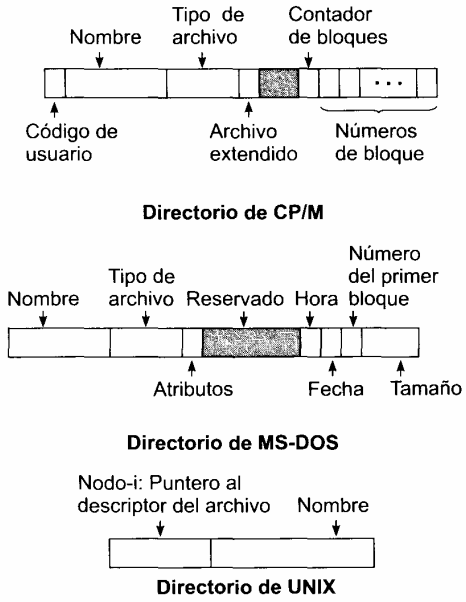


Figura 8.8. *Ejemplos de entradas de directorio.*

Los sistemas operativos modernos usan la última solución, porque tiene varias ventajas que la primera. Sin embargo, existen sistemas operativos, como CPíM o MS-DOS, que almacenan atribu tos del archivo en su entrada del directorio. **CP/M** [Zaks, 1980] tiene un único directorio, en el que cada entrada incluye toda la información suficiente para acceder al archivo asociado (Fig. 8.8). Por tanto, una vez encontrado el nombre del archivo dentro del directorio, se dispone de sus atributos, su tipo y un número máximo de bloques de disco asociados al archivo. Si se necesitan más bloques de datos es necesario asignar una nueva entrada de directorio y fijar su posición dentro del archivo (1 a, 2. etc.) mediante el campo extendido. Esta solución sólo permite esquemas de nombres de un único nivel. **MS-DOS** [Norton, 1988] tiene entradas de directorio de 32 bytes que contienen el nombre del archivo, sus atributos, valores de tiempos y el primer bloque de disco del archivo. A partir de este bloque, el sistema de archivos accede a una tabla de bloques del dispositivo (FAT), dentro de la cual puede acceder a bloques sucesivos del archivo por hallarse encadenados mediante una lista. La principal diferencia entre esta solución y la de CPÍM es que en MS-DOS una entrada del directorio puede a su vez representar a otro directorio, con lo que se pueden definir esquemas jerárquicos de nombres. Además, el límite en la extensión del archivo ya no está incluido en la entrada del directorio, sino que depende del tamaño de la FAT.

En el diseño del sistema operativo UNIX se adoptó la segunda alternativa, por lo que la entrada de directorio es una estructura de datos muy sencilla (Fig. 8.8) que contiene únicamente el nombre del archivo asociado a ella y el identificador del descriptor de archivo, número de nodo-i, usado por el sistema operativo [19941. Toda la información relativa a los atributos del archivo se almacena en su nodo-i. El uso de una entrada de directorio como la de UNIX tiene varias ventajas:

• La entrada de directorio no se ve afectada por los cambios de atributos del archivo.

• Los nodos-i pueden representar a su vez directorios o archivos, con lo que se pueden cons truir esquemas de nombres jerárquicos de forma sencilla.

Digitalización realizada con propósito académico

• La longitud de los nombres no está predeterminada, pudiendo ser variable hasta un límite grande (4.096 caracteres en las versiones actuales).

• La interpretación de nombres es muy regular, lo que aporta sencillez al sistema de archivos.

• Facilita la creación de sinónimos para el nombre del archivo.

8.3.2. Estructuras de directorio

Independientemente de cómo se defina la entrada de un directorio, es necesario organizar todas las entradas de directorio para poder manejar los archivos existentes en un sistema de almacenamiento de forma sencilla y eficiente. La forma de estructurar dichas entradas varía de unos sistemas a otros dependiendo de las necesidades de cada sistema.

En sistemas operativos antiguos, como CP/M, se usaron estructuras de directorio de **nivel único y directorios con dos niveles** (Fig. 8.9). En la primera existe un único directorio que contie ne todas las entradas de archivos. Es fácil entender e implementar, pero asignar nombre a un archi vo nuevo no es fácil por la dificultad de recordar todos los nombres o la imposibilidad de ver los de otros usuarios. En CP/M, para evitar este problema, el número máximo de archivos que podía haber en un sistema de archivos estaba significativamente limitado. Esto reduce el problema de gestión de nombres, pero limita el número de archivos en un dispositivo de almacenamiento. Cuando creció la capacidad de los dispositivos de almacenamiento, se resolvió el problema anterior extendiendo la estructura a un directorio de dos niveles, donde cada usuario disponía de su propio directorio, reduciendo así una parte de la confusión inherente a los directorios de nivel único. Cuando se daba de alta un nuevo usuario en el sistema, las utilidades de administración del sistema

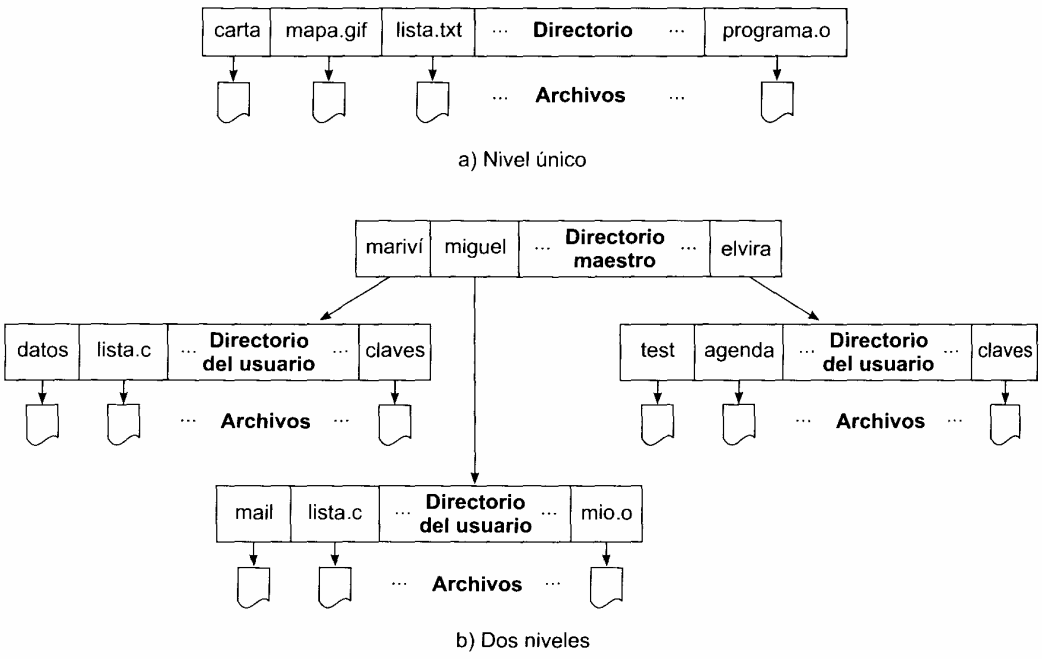


Figura 8.9. *Estructuras de directorio de uno y dos niveles.*

Digitalización realizada con propósito académico

operativo creaban el directorio para los archivos del usuario con un nombre fijado con criterios internos al sistema. Cuando el usuario entra al sistema, se le ponía dentro de su directorio. Para acceder a sus archivos podía hacer dos cosas: especificar sólo el nombre del archivo relativo a su directorio (p. ej.: claves) o especificar el nombre completo del archivo incluyendo dispositivo. directorio del usuario y nombre de archivo (p. ej.: C: \miguel\c laves). Esta estructura mejoraba la seguridad al aislar a los usuarios, pero no se evitaba otros problemas, tales como la gestión de nombres propios de un usuario, la imposibilidad de agrupar los archivos de un mismo usuario con distintos criterios o el uso compartido de archivos. Supongamos, por ejemplo, que un alumno tiene varios archivos de prácticas. Todos ellos estarían bajo el mismo directorio, pero no podría agrupar los de las prácticas de sistemas operativos o las de arquitectura de computadoras. Suponga ahora que la practica se lleva a cabo por un grupo de alumnos, ¿Corno pueden compartir los archivos de la misma? Con esta estructura sería necesario copiar los archivos en los directorios de todos ellos, con el peligro consiguiente de incoherencias entre las distintas copias, o crear un nuevo usuario (p. ej.: grupo-l) y poner en su directorio los archivos compartidos. Esta solución era la adoptada para permitir que todos los usuarios vieran las utilidades del sistema operativo sin tener que hacer múltiples copias: crear un usuario especial, denominado sistema, en cuyo directorio se incluían dichas utilidades con permisos de lectura y ejecución para todos los usuarios.

A medida que la capacidad de los dispositivos se fue incrementando, fueron también creciendo el número de archivos almacenados por los usuarios en el sistema de archivos. Con lo que el problema de la complejidad del nombrado, paliado por la estructura de dos niveles, volvió a parecer

a nivel de usuario. Fue pues necesario generalizar la estructura de dos niveles para obtener una estructura jerárquica más general que permitiera a los usuarios ordenar sus archivos con criterios lógicos en sus propios subdirectorios, sin depender de las limitaciones de los niveles de la estructura de directorios [Organick, 1972], lo que llevó a la **estructura de árbol**. Una estructura de árbol representa todos los directorios y subdirectorios del sistema partiendo de un **directorio raíz**, exis tiendo un camino único (**path**) que atraviesa el árbol desde la raíz hasta un archivo determinado. Los nodos del árbol son directorios que contiene un conjunto de subdirectorios o archivos. Las hojas del árbol son siempre archivos. Normalmente, cada usuario tiene su propio directorio **home** a partir del cual se cuelgan sus subdirectorios y archivos y en el que le sitúa el sistema operativo cuando entra a su cuenta. Este directorio lo decide el administrador, o los procesos de creación de nuevos usuarios, cuando el usuario se da de alta en el sistema y está almacenado junto con otros datos del usuario en una base de datos o archivo del sistema operativo. En el caso de UNIX, por ejemplo, el archivo /etc/password contiene una línea por usuario del sistema, similar a la siguiente:

miguel: * :Miguel: /users/miguel: /etc/bin

donde el directorio home es /users/miguel. MS-DOS y Windows NT tienen registros de usua rios que también definen el directorio home.

Los usuarios pueden subir y bajar por el árbol de directorios, mediante servicios del sistema operativo, siempre que tengan los permisos adecuados. Por ello, tanto el propio usuario como el sistema operativo deben conocer dónde están en cada instante. Para solventar este problema se definió el concepto de **directorio de trabajo**, que es el directorio en el que un usuario se encuentra en un instante determinado. Para crear o bonar un archivo o directorio se puede indicar su nombre relativo al directorio de trabajo o completo desde la raíz a las utilidades del sistema que llevan a cabo estas operaciones. Un problema interesante con este tipo de estructura es cómo borrar un directorio no vacío. Hay dos soluciones posibles:

• Un directorio no se bona si no está vacío. Si tiene archivos o subdirectorios, el usuario debe borrarlos previamente. Esta es la solución adoptada habitualmente en las llamadas a los sistemas operativos.

Digitalización realizada con propósito académico

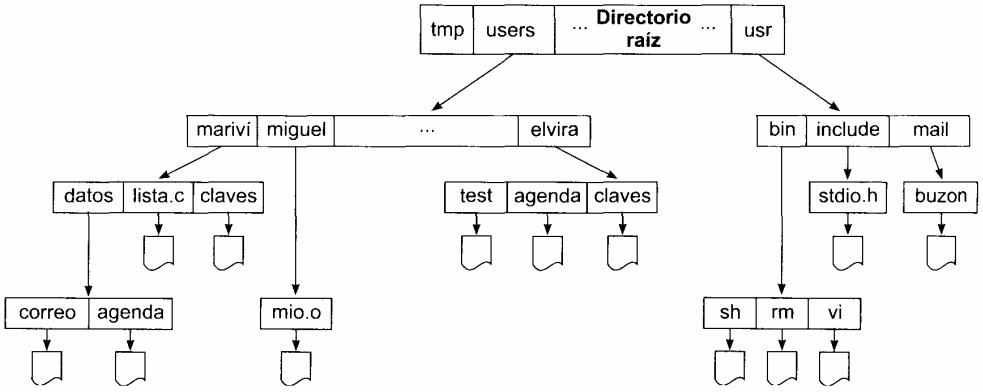


Figura 8.10. *Estructura de árbol jerárquico.*

• Un directorio, y todo el subárbol de directorios que cuelga de él, es borrado, aunque el subárbol no esté vacío. Esta solución existe en UNIX y Windows NT, aunque no como llamada al sistema sino como mandato de usuario. Para evitar que un usuario borre archivos por error se solicita una confirmación del usuario, vía opción en el mandato UNIX o confir mación en un menú gráfico en Windows NT. Si la respuesta es afirmativa, se borra todo el subárbol recursivamente.

La estructura de árbol es muy general, pero no proporciona los servicios requeridos en algunos entornos. Por ejemplo, puede ser interesante que varios programadores trabajando en el mismo proyecto compartan archivos o subdirectorios llegando a los mismos por sus respectivos directorios de usuario para no tener problemas de seguridad y protección. Esta forma de acceso no existe en la estructura de árbol porque exige que a un archivo se llegue con único nombre. El modelo descrito, sin embargo, rompe la relación uno a uno entre el nombre y el archivo, al requerir que un mismo archivo pueda ser accedido a través de varios caminos. Este problema puede resolverse generali zando la estructura del árbol de directorio para convertirla en un **grafo acíclico** (Fig. 8. 1 1) en el cual el mismo archivo o subdirectorio puede estar en dos directorios distintos, estableciendo una relación unívoca nombre-archivo. La forma más habitual de compartir archivos es crear un **enlace** al archivo compartido [Bach, 1986] de uno de los dos tipos siguientes:

• **Físico.** Un apuntador a otro archivo o subdirectorio, cuya entrada de directorio tiene el mismo descriptor de archivo (en UNIX, el nodo-i) que el archivo enlazado. Ambos nombres apuntan al mismo archivo. Resolver cualquiera de los nombres de sus enlaces nos devuelve el descriptor del archivo.

• **Simbólico.** Un nuevo archivo cuyo contenido ese! nombre del archivo enlazado. Resolver el nombre del enlace simbólico no da el descriptor del destino, sino el descriptor del archivo en el que está el nombre del destino. Para acceder al destino, hay que abrir el archivo del enlace, leer el nombre del destino y resolverlo.

Para tener constancia de los enlaces físicos que tiene un archivo, se ha añadido un nuevo atributo al nodo-i denominado **contador de enlaces**. Cuando se crea un enlace a un archivo, se incrementa en su nodo-i del archivo un contador de enlaces físicos. Cuando se rompe el enlace, se decrementa dicho contador. La existencia de enlaces introduce varios problemas en este tipo de estructura de directorio:

Digitalización realizada con propósito académico

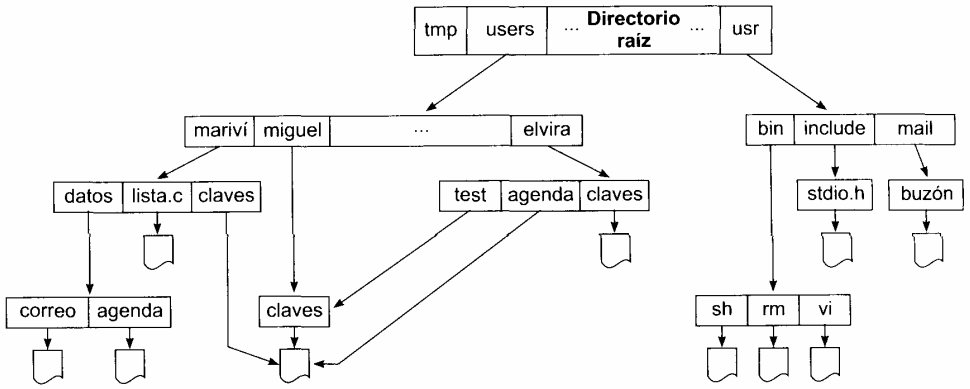


Figura 8.11. *Estructura de grafo acíclico.*

• Existen varios nombres para el mismo archivo. Si se quiere recorrer todo el árbol, es impor tante evitar los bucles (Advertencia 8.1).

• El borrado de archivos se complica, ya que un mismo archivo puede ser borrado por varios caminos. Es necesario pues determinar cuándo se puede borrar el archivo. En UNIX, el archivo no se borra hasta que no existe ninguna referencia al mismo, lo que significa que el valor del contador de enlaces es cero. Para ello, cuando se borra un archivo, se borra la entrada de directorio que referencia al archivo y se decrementa su contador de enlaces. Sólo en el caso de que el contador de enlaces sea cero y de que nadie tenga abierto el archivo se borra el archivo realmente y se liberan sus recursos.

ADVERTENCIA 8.1

¡Cuidado con los enlaces! Es importante evitar la existencia de bucles en el árbol de directorios. Dichos bucles originan dos problemas: no permiten recorrer el árbol de directorios completo y pueden hacer que una operación de traducción de nombre no acabe nunca.

Para evitar los problemas anteriores, algunos sistemas operativos, como MS-DOS, no permi ten la existencia de directorios compartidos o enlaces.

8.3.3. Nombres jerárquicos

La especificación del nombre de un archivo en un árbol de directorios, o en un grafo acíclico, toma siempre como referencia el directorio raíz (1 en UNIX, \ en MS-DOS). A partir de este directorio, es necesario viajar por los sucesivos subdirectorios hasta encontrar el archivo deseado. Para ello el sistema operativo debe conocer el nombre completo del archivo a partir del directorio raíz. Hay dos posibles formas de obtener dicho nombre:

• Que el usuario especifique el nombre completo del archivo, denominado **nombre absoluto**.

• Que el usuario especifique el nombré de forma relativa, denominado **nombre relativo**, a algún subdirectorio del árbol de directorios.

Digitalización realizada con propósito académico

El **nombre absoluto** de un archivo proporciona todo el camino a través del árbol de directo rios desde la raíz hasta el archivo. Por ejemplo, en UNIX, un nombre absoluto de uno de los archivos existentes en la Figura 8.11 sería /users/míguel/claves. Este nombre indica al siste ma de archivos que a partir del directorio raíz (1) se debe atravesar el directorio users y, dentro de este último, el subdirectorio miguel para llegar al archivo claves. En MS-DOS, dicho nombre absoluto se representaría como C: \users\miguel\claves. Un nombre absoluto es autoconteni do, en el sentido de que proporciona al sistema de archivos toda la información necesaria para llegar al archivo, sin que tenga que suponer o añadir ninguna información de entorno del proceso o interna al sistema operativo. Algunas aplicaciones necesitan este tipo de nombres para asegurar que sus programas funcionan independiente del lugar del árbol de directorios en que estén situados. Por ejemplo, un compilador de lenguaje C, en una máquina que ejecuta el sistema operativo UNIX, necesita saber que los archivos con definiciones de macros y tipos de datos están en el directorio /usr/include. Es necesario especificar el nombre absoluto porque no es posible saber en qué directorio del árbol instalará cada usuario el compilador.

La Figura 8.12 muestra, para el sistema operativo UNIX, las tablas que almacenan algunos de los directorios especificados en la Figura 8.1 1. Como puede verse, el directorio raíz tiene un núme ro de nodo-i predefinido en UNIX, el 2. A partir de sus entradas se puede interpretar cualquier nombre absoluto. Por ejemplo, el nombre absoluto /users/miguel/claves se interpretaría de la siguiente forma:

1. Se traen a memoria las entradas existentes en el archivo con nodo-i 2.

2. Se busca dentro de ellas el nombre users y se extrae su nodo-i, el 342.

3. Se traen a memoria las entradas del archivo con nodo-i 342.

4. Se busca dentro de ellas el nombre miguel y se extrae su nodo-i, el 256.

5. Se repite el proceso con este nodo-i hasta que se encuentra el archivo claves y se obtiene su nodo-i, el 758.

La interpretación de estos nombres en MS-DOS sería similar, pero, en lugar de usar el nodo-i para acceder a los bloques que contienen entradas de directorios, se usa la tabla FAT, por lo que los números de nodo-i reseñados en el ejemplo se sustituirían por números de bloques de la FAT.

¿Cómo se sabe cuándo parar la búsqueda? Hay tres criterios bien definidos:

• Se ha encontrado el archivo cuyo nombre se ha especificado, en cuyo caso se devuelve el nodo-i del archivo.

• No se ha encontrado el archivo y estamos en el último subdirectorio especificado en el nombre absoluto (miguel). Este caso devuelve error.

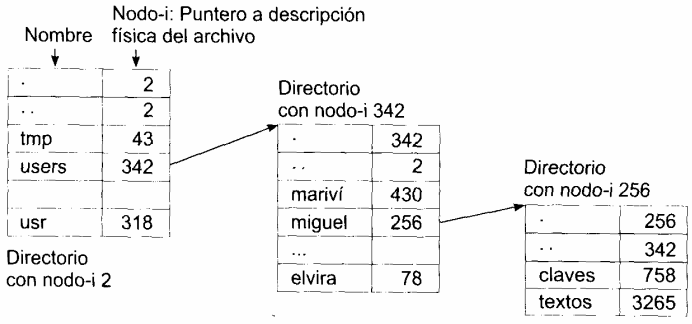


Figura 8.12. *Tablas de directorios en UNIX.*

Digitalización realizada con propósito académico

• Estamos en un directorio y la siguiente entrada de subdirectorio especificada en el nombre no existe o no se tiene permiso de acceso. Este caso también devuelve un error.

Puesto que la profundidad del árbol de directorios puede ser muy grande, resultaría muy incó modo tener que especificar constantemente nombres absolutos de archivos. Además, algunas aplicaciones usan directorios relativos a su situación para almacenar archivos. Es por ello que la mayo ría de los sistemas operativos modernos permiten definir **nombres relativos**, es decir, nombres de archivo que sólo especifica una porción del nombre absoluto a partir de un determinado suhdirec tono del árbol de nombres. Por ejemplo, miguel/claves. Los nombres relativos no se pueden interpretar si no se conoce el directorio del árbol a partir del que empiezan, para ello existe un **directorio de trabajo**, o **actual**, a partir del cual se interpretan siempre los nombre relativos. Por ejemplo, si el directorio de trabajo es /usr y se especifica el nombre relativo miguel/claves se obtendrá un error. Pero si el directorio de trabajo es /users, la interpretación del nombre relativo será correcta. Es responsabilidad del usuario colocarse en el directorio de trabajo adecuado antes de usar nombres relativos al mismo.

Muchos sistemas operativos con directorios jerárquicos tienen dos entradas especiales, . y

• , en cada directorio para referirse a sí mismos y a su directorio padre en la jerarquía. Estas entradas especiales son muy útiles para especificar posiciones relativas al directorio actual y para viajar por el árbol. Por ejemplo, si el directorio actual es /users/miguel, el mandato UNIX

is . /

mostrará las entradas de directorio de su padre en el árbol, es decir, del directorio users. Pero el mandato UNIX

cp /usr/include/stdio.h

copiará el archivo stdio.h al directorio actual, es decir, miguel.

8.3.4. Construcción de la jerarquía de directorios

Una de las decisiones más importantes de diseño de un sistema operativo, que acaba siendo visible al usuario, es ofrecer o no un árbol único de directorios para todo el sistema, independientemente de los dispositivos físicos o lógicos en que éstos estén almacenados. En MS-DOS y en Windows NT, el árbol de directorios es único por dispositivo lógico, pero el usuario debe conocer los nombres de dispositivo (c :, H:, etc.) cuando quiere cambiar de uno a otro. Esa es la razón por la que la especifi cación en Windows N de un nombre completo, tal corno c ‘\In exige espec el nombre de dispositivo. En todos los sistemas operativos derivados de UNIX, la existencia de un **árbol de directorios único** es un principio de diseño fundamental. Para que ello sea posible, el sistema debe ocultar la existencia de los dispositivos al usuario, que debe poder cambiar dc uno a otro de forma transparente sin más que usar nombres lógicos. Por ejemplo, suponga que en la Figura 8.12 el directorio raíz estuviera en el dispositivo /dev/hdal (C: en Windows) y que los usuarios (/users) estuvieran en el dispositivo /dev/hda3 (D: en Windows). Si se quiere acceder al archivo /users/miguel/teXtOS, en UNIX bastaría con especificar dicho nombre. Sin embar go, en Windows sería necesario especificar D: \miguel\textos. .

Para ofrecer una imagen única del sistema, el sistema operativo debe ofrecer servicios que

permitan asociar, y desasociar, unos sistemas de otros de forma transparente en un árbol de nom bres único. Además, las utilidades de interpretación de nombres deben ser capaces de saltar de un

Digitalización realizada con propósito académico

sistema de archivos a otro sin que sea aparente en ningún momento el nombre del dispositivo físico o lógico que almacena el sistema de archivos. Las dos llamadas al sistema de UNIX que realizan estas operaciones son mount y umount [19851. La operación de montado permite conectar un sistema de archivos, almacenado en un volumen o partición, a una entrada de directorio del árbol de directorios existente. A partir de dicha operación, el sistema de archivos en el nuevo dispositivo aparece como un subárbol del árbol de directorios, sin que exista diferencia con el resto del mismo. Sus archivos y directorios son accesibles a través del nombre lógico. Por ejemplo, para montar /dev/hda3 de forma que se construya un árbol de directorios como el de la Figura 8.12 se ejecuta ría el siguiente mandato:

mount /dev/hda3 /users

Desmontar (umount) un sistema de archivos es sencillo. Por ejemplo, para desconectar el

dispositivo /dev/hda3 montado en el directorio /users del sistema de archivos raíz bastaría con ejecutar el mandato:

umount /users

Si no se está usando ningún archivo o directorio del sistema de archivos existente en

/dev/hda3, el sistema operativo lo desconecta. A partir de ese instante, el subárbol de directorios del dispositivo no aparecerá en la jerarquía de directorios y sus datos no estarán accesibles.

Las operaciones anteriores tienen dos ventajas:

1. Ofrecen una imagen única del sistema.

2. Ocultan el tipo de dispositivo que está montado sobre un directorio.

Sin embargo, también tienen inconvenientes:

1. Complican la traducción de nombres lógicos de archivos.

2. Dan problemas cuando se quiere establecer un enlace físico entre dos archivos. Debido a estos problemas, en UNIX sólo se puede establecer un enlace físico entre dos archivos que se encuentran en el mismo sistema de archivos. Nunca de un archivo a otro existente en un sistema de archivos distinto.

En la Sección 8.6, dedicada al servidor de archivos, se explica con más detalle cómo se imple mentan las operaciones anteriores.

8.4. SERVICIOS DE ARCHIVOS Y DIRECTORIOS

Un archivo es un tipo abstracto de datos. Por tanto, para que esté completamente definido, es necesario definir las operaciones que pueden ejecutarse sobre un objeto archivo. En general, los sistemas operativos proporcionan operaciones para crear un archivo, almacenar información en él y recuperar dicha información más tarde.

En la mayoría de los sistemas operativos modernos los directorios se implementan como archi vos que almacenan una estructura de datos definida: entradas de directorios. Por ello, los servicios de archivos pueden usarse directamente sobre directorios. Sin embargo, la funcionalidad necesaria para los directorios es mucho más restringida que la de los archivos, por lo que los sistemas operati vos suelen proporcionar servicios específicos para satisfacer dichas funciones de forma eficiente y sencilla.

Digitalización realizada con propósito académico

En esta sección se muestran los servicios genéricos más frecuentes para archivos y directorios. A continuación, se estudia la concreción de dichos servicios propuesta en el estándar POSIX [IEEE, 19881, y en Win32, y se muestran ejemplos de uso de dichos servicios.

8.4.1. Servicios genéricos para archivos

Los sistemas operativos proporcionan llamadas para crear, borrar, abrir, cerrar, leer, escribir, mo- verse dentro del archivo y ver y/o modificar los atributos del objeto. A continuación, haremos una breve descripción de las mismas.

- **Crear**. Este servicio crea un archivo vacío asegurándose de que hay espacio para el archivo el dentro del sistema de archivos, de que se puede crear una entrada de directorio para su ;on nombre y de que no existe ya un archivo con el mismo nombre.
- **Borrar.** Busca el nombre del archivo en los directorios. Si existe, libera la entrada del directorio y los recursos del sistema de archivos asignados al archivo.
- **Abrir**. Un archivo debe ser abierto antes de usarlo. Este servicio comprueba que el archivo existe y que el usuario tiene derechos de acceso. Si es así, trae a memoria información del en objeto para optimizar el acceso al mismo. Además, devuelve al usuario un identificador dos temporal del objeto para su manipulación. Normalmente, todos los sistemas operativos tie nen un límite máximo para el número de archivos que puede tener abierto un usuario.
- **Cerrar.** Este servicio permite al usuario indicar que ya no necesita el archivo. Cuando ocurre esto, se elimina el identificador temporal, se liberan los recursos de memoria que ocupa el archivo y se actualiza su información en el disco.
- **Leer.** Este servicio permite traer información del archivo a memoria. Para ello se especifica el identificador obtenido en la apertura, la posición en memoria y la cantidad de información a leer. Normalmente se lee a partir de la posición que indica el apuntador de posición del archivo. En algunos sistemas operativos se puede especificar la posición de lectura.
- **Escribir**. Este servicio permite llevar información de memoria al archivo. Para ello se espe que cifica el identificador obtenido en la apertura, la posición en memoria y la cantidad de nun información a escribir. Normalmente se escribe a partir de la posición que indica el apunta dor de posición del archivo. Si está en medio, se sobrescriben los datos. Si está al final del archivo, su tamaño crece. En algunos sistemas operativos se puede especificar la posición de escritura.
- **Cambiar el apuntador**. Este servicio permite cambiar el valor del apuntador de posición del archivo. Normalmente no conlleva transferencia de datos, sino que se usa para colocarse en algún punto del archivo antes de leer o escribir. Este servicio es necesario cuando se realizan accesos aleatorios al archivo.
- **Manipulación de atributos**. En muchos casos las aplicaciones necesitan conocer los atrihu o, es tos de un archivo antes de ejecutar otras operaciones de E/S. Por ejemplo, un sistema de ,los realización de copias de respaldo incrementales necesita conocer el tiempo de última modi el y ficación de los archivos. Estos servicios permiten ver o modificar los atributos de un archi yo. La información de control de acceso a un archivo (protección) puede ser modificada gracias a estos servicios.

Sobre la base de los servicios anteriores se pueden implementar otros como añadir intorma erati- ción al final del archivo, renombrar archivos, bloquear un archivo o proyectar un archivo sobre una nte y zona de memoria. Además, se pueden implementar mandatos que permitan copiar, mover o compi lar archivos.

Digitalización realizada con propósito académico

8.4.2. Servicios POSIX para archivos

POSIX proporciona una visión lógica de archivo equivalente a una tira secuencial de bytes. Para acceder al archivo se mantiene un apuntador de posición, a partir del cual se ejecutan las operacio nes de lectura y escritura sobre el archivo. Para identificar a un archivo, el usuario usa nombres al estilo de UNIX, como por ejemplo /users/miguel/datos. Cuando se abre un archivo, se de vuelve un descriptor de archivo, que se usa a partir de ese momento para identificar al archivo en otras llamadas al sistema. Estos descriptores son números enteros de O a n y son específicos para cada proceso. Cada proceso tiene su tabla de descriptores de archivo (tda), desde la que se apunta a los descriptores internos de los archivos. El descriptor de archivo fd indica el lugar en la tabla, que se rellena de forma ordenada, de manera que siempre se ocupa la primera posición libre de la misma. Cuando se realiza una operación open, el sistema de archivos busca desde la posición O hasta que encuentra una posición libre, siendo ésa la ocupada. Cuando se cierra un archivo (close), se libera la entrada correspondiente. En los sistemas UNIX, cada proceso tiene tres descriptores de archivos abiertos por defecto. Estos descriptores ocupan las posiciones O a 2 y reciben los siguientes nombres:

- Entrada estándar, fd = O.
- Salida estándar, fd = 1.
- Error estándar, fd = 2.

El objetivo de estos descriptores estándar es poder escribir programas que sean independientes de los archivos sobre los que han de trabajar.

Cuando se ejecuta una llamada fork se duplica el BCP del proceso. Cúando se cambia la imagen de un proceso con la llamada exec, uno de los elementos del BCP que se conserva es la tabla de descriptores de archivo. Por tanto, basta con que un proceso coloque adecuadamente los descriptores estándar y que luego invoque la ejecución del mencionado programa, para que éste utilice los archivos previamente seleccionados (Aclaración 8.2).

ACLARACIÓN 8.2

Los mandatos del intérprete de mandatos de UNIX están escritos para leer y escribir de los descriptores estándar, por lo que se puede hacer que ejecuten sobre cualquier archivo sin más que redireccionar los descriptores de archivos estándares.

Usando servicios de POSIX se pueden consultar los atributos de un archivo. Estos atributos son una parte de la información existente en el descriptor interno del archivo (nodo-i). Entre ellos se encuentran el número de nodo-i, el sistema de archivos al que pertenece, su dispositivo, tiempos de creación y modificación, número de enlaces físicos, identificación de usuario y grupo, tamaño óptimo de acceso, modo de protección, etc. El modo de protección es especialmente importante porque permite controlar el acceso al archivo por parte de su dueño, su grupo y el resto del mundo. En POSIX, estos permisos de acceso se especifican usando máscaras de 9 bits con el siguiente formato:

dueño grupo mundo

rwx rwx rwx

Variando los valores de estos bits, se pueden definir los permisos de acceso a un archivo. Por ejemplo, 755 indica rwxr-xr-x. En el Capítulo 9 se estudia este tema con más profundidad.

Digitalización realizada con propósito académico

En esta sección se describen los principales servicios de archivos descritos en el estándar POSIX [IEEE, 1988]. Estos servicios están disponibles en prácticamente todos los sistemas operati vos modernos.

Crear un archivo

Este servicio permite crear un nuevo objeto archivo. El prototipo de la llamada es:

int **creat**(const char *path, modet mode)

Su efecto es la creación de un archivo con nombre path y modo de protección mode. El resultado es un descriptor de archivo interno al proceso que creó el archivo y que es el descriptor de archivo más bajo sin usar en su tabla de descriptores. El archivo queda abierto sólo para escritura. En caso de que el archivo exista, se trunca. En caso de que el archivo no pueda ser creado, devuelve -1 y pone el código de error adecuado en la variable errrno.

Borrar un archivo

Este servicio permite borrar un archivo indicando su nombre. Si el archivo no está referenciado por ningún nombre más (lo que en UNIX equivale a decir que el contador de los enlaces del nodo-i es cero) y nadie lo tiene abierto, esta llamada libera los recursos asignados al archivo, por lo que queda inaccesible. Si el archivo está abierto, se pospone su eliminación hasta que todos los procesos lo hayan cerrado. El prototipo de este servicio es:

int **unlink**(const char *path)

El argumento path indica el nombre del archivo a borrar.

Abrir un archivo

Este servicio establece una conexión entre un archivo y un descriptor de archivo abierto, interno al proceso que lo ha abierto. El resultado de su ejecución es el descriptor de archivo libre más bajo de su tabla de descriptores. El prototipo de la llamada es:

int **open**(const char *path, mt oflag, /* modo t mode */ ..) ;

El argumento path define el nombre del archivo a abrir. El argumento of lag permite especi ficar qué tipo de operación se quiere hacer con el archivo: lectura (O_RDONLY), escritura (O_WRONLY), lectura-escritura (O_RDWR). añadir información nueva (O_ APPEND), creación, trunca do, escritura no bloqueante, etc. En caso de especificar creación del archivo (O_CREAT) se debe especificar el modo de protección del mismo, siendo el efecto de la llamada equivalente al servicio creat. Por ejemplo, creat (“archivo”, 0751) es equivalente a operi (“archivo”, O_WRONLY O_CREAT (O_TRUNC, 0751). Si el archivo no existe, no se puede abrir con las caracte rísticas especificadas o no se puede crear, la llamada devuelve -l y un código de error en la variable errno.

Cerrar un archivo

La llamada close libera el descriptor de archivo obtenido cuando se abrió el archivo, dejándolo disponible para su uso posterior por el proceso. El prototipo de la llamada es el siguiente:

int **close**(int fildes);

Digitalización realizada con propósito académico

Si la llamada termina correctamente, se liberan los posibles cerrojos sobre el archivo fijados por el proceso y se anulan las operaciones pendientes de E/S asíncrona. Además, si ningún proceso lo tiene abierto (es decir, en UNIX el contador de aperturas del nodo-i es cero) se liberan los recursos del sistema operativo ocupados por el archivo, incluyendo posibles proyecciones en me moria del mismo.

Leer datos de un archivo

Este servicio permite a un proceso leer datos de un archivo, que debe abrirse previamente, y copiar- los a su espacio de memoria. El prototipo de la llamada es:

ssize_t **read**(int fildes, void *buf, sizet nbyte);

El descriptor de archivo se indica en fildes, la posición de memoria donde copiar los datos se especifica en el argumento buf y el número de bytes a leer se especifica en nbyte. La lectura se lleva a cabo a partir de la posición actual del apuntador de posición del archivo. Si la llamada se ejecuta correctamente, devuelve el número de bytes leídos realmente, que pueden ser menos que los pedidos, y se incrementa el apuntador de posición del archivo con esa cantidad. Si se intenta leer más allá del fin de archivo, la llamada devuelve un cero. Si se intenta leer en una posición interme dia de un archivo que no ha sido escrita previamente, la llamada devuelve bytes con valor cero.

Escribir datos a un archivo

Esta llamada permite a un proceso escribir en un archivo una porción de datos existente en el espacio de memoria del proceso. Su prototipo es:

ssize_t **write**(int fildes, const void *buf, size_t nbyte);

El descriptor de archivo se indica en fildes, la posición de memoria donde copiar los datos se especifica en el argumento buf y el número de bytes a escribir se especifica en nbyte. La escritura se lleva a cabo a partir del valor actual del apuntador de posición del archivo. Si la llamada se ejecuta correctamente, devuelve el número de bytes escritos realmente y se incrementa el valor del apuntador de posición del archivo. Si la escritura va más allá del fin de archivo, se incrementa el tamaño del mismo. Si se especificó O_APPEND en la apertura del archivo, antes de escribir se pone el apuntador de posición al final del archivo, de forma que siempre se añade la información al final.

Cambio del apuntador de un archivo

Esta llamada permite cambiar el valor del apuntador de posición de un archivo abierto, de forma que posteriores operaciones de E/S se ejecuten a partir de esa posición. Su prototipo es:

off_t **lseek**(int fildes, offt offset, mt whence);

El descriptor de archivo se indica en Ludes, el desplazamiento se indica en offset y el lugar de referencia para el desplazamiento se indica en whence. Hay tres formas de indicar la posición de referencia para el cambio de apuntador: SEEK_SET, el valor final del apuntador es offset; SEEK CUR, el valor final del apuntador es el valor actual más offset; SEEK el valor final es la posición de fin de archivo más offset. Si la llamada se ejecuta correctamente. devuelve el nuevo valor del apuntador de posición del archivo. Esta llamada permite ir más allá del fin del archivo, pero no modifica el tamaño del mismo. Para que esto ocurra hay que escribir algo en dicha posición.

Faltan desde la pag 443 hasta la 445

Digitalización realizada con propósito académico

rrespondiente a la representación de ese objeto archivo. Al igual que en POSIX, los procesos pue den obtener manejadores de objetos estándar para entradalsalida (GetstdHandle, Set StdHand— le). El objetivo de estos descriptores estándares es poder escribir programas que sean independien tes de los archivos sobre los que han de trabajar.

Usando servicios de Win32 se pueden consultar los atributos de un archivo. Estos atributos son una parte de la información existente en el manejador del objeto. Entre ellos se encuentran el nombre, tipo y tamaño del archivo, el sistema de archivos al que pertenece, su dispositivo, tiempos de creación y modificación, información de estado, apuntador de posición del archivo, información sobre cerrojos, protección, etc. El modo de protección es especialmente importante porque permite controlar el acceso al archivo por parte de los usuarios. Cada objeto archivo tiene asociado un descriptor de seguridad, que se describe detalladamente en el Capítulo 9.

A continuación, se muestran los servicios más comunes para gestión de archivos que propor ciona Win32 [19971. Como se puede ver, son similares a los de POSIX, si bien los prototipos de las funciones que los proporcionan son bastante distintos.

Crear y abrir un archivo

Este servicio permite crear, o abrir, un nuevo archivo. El prototipo de la llamada es:

HANDLE **CreateFile**(LPCSTR lpFileName, DWORD dwDesíredAccess,

DWORD dwShareMode, LPVOID lpSecurityAttributes,

DWORD CreationDísposition, DWORD dwFlagsAndAttributes,

HANDLE hTemplateFile);

Su efecto es la creación, o apertura, de un archivo con nombre lpFi leName, modo de acceso

dwDesiredAccess (GENERIC_READ, GENERIC_WRITE) y modo de compartición dwShareMode (NO_SHARE, SHARE_READ, SHARE WRITE). lpSecurítyAttributes define los atributos de se guridad para el archivo, cuya estructura se comentará en el Capítulo 9. La forma de crear o abrir el archivo se especifica en CreationDisposition. Modos posibles son: CREATE_NEW, CREA TE_ALWAYS, OPEN EXISTING, OPEN_ALWAYS, TRUNCATE EXISTING. Estas primitivas indican que se debe crear un archivo si no existe, sobrescribirlo si ya existe, abrir uno ya existente, crearlo y abrirlo si no existe y truncarlo si ya existe, respectivamente. El parámetro dwFlagsAndAttribu tes especifica acciones de control sobre el archivo. Por su extensión, se refiere al lector a manuales específicos de Win32. hTemplateFíle proporciona una plantilla con valores para la creación por defecto. Si el archivo ya está abierto, se incrementa el contador de aperturas asociado al manejador. El resultado de la llamada es un manejador al nuevo archivo. En caso de que haya un error devuelve un manejador nulo.

Borrar un archivo

Este servicio permite borrar un archivo indicando su nombre. Esta llamada libera los recursos asignados al archivo. El archivo no se podrá acceder nunca más. El prototipo de este servicio es:

BOOL **DeleteFile**(LPCTSTR lpszFileName);

El nombre del archivo a borrar se indica en lpszFileName. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Digitalización realizada con propósito académico

Cerrar un archivo

La llamada CloseHandie libera el descriptor de archivo obtenido cuando se abrió el archivo, dejándolo disponible para su uso posterior. El prototipo de la llamada es el siguiente:

BOOL **CloseHandle**(HANDLE hObject);

Si la llamada termina correctamente, se liberan los posibles cerrojos sobre el archivo fijados por el proceso, se invalidan las operaciones pendientes y se decrementa el contador del manejador del objeto. Si el contador del manejador es cero, se liberan los recursos ocupados por el archivo. Dv TR en caso de éxito o FALSE en caso de error.

Leer datos de un archivo

Este servicio permite a un proceso leer datos de un archivo abierto y copiarlos a su espacio de memoria. El prototipo de la llamada es:

BOOL **ReadFile**(HANDLE hFile, LPVOID]pBuffer,

DWORD nNumberOfBytesToRead, LPD lpNumberOfBytesRead,

LPVOID ILpOverlapped);

El manejador del archivo se indica en hFile, la posición de memoria donde copiar los datos se especifica en el argumento lpBuffer y el número de bytes a leer se especifica en lpNumbe— rüfBytesToRead. La lectura se lleva a cabo a partir de la posición actual del apuntador de posi ción del archivo. Si la llamada se ejecuta correctamente, devuelve el número de bytes leídos real mente, que pueden ser menos que los pedidos, en]pNumberOfBytesRead y se incrementa el apuntador de posición del archivo en esa cantidad. lpOverlapped sirve para indicar el manejador de evento ue se debe usar cuando se hace una escritura no bloqueante (asíncrona). Si ese paráme tro tiene algún evento activado, cuando termina la llamada se ejecuta el manejador de dicho evento. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Escribir datos a un archivo

Esta llamada permite a un proceso escribir en un archivo una porción de datos existente en el espacio de memoria del proceso. Su prototipo es:

BOOL **WriteFile**(HANDLE hFile, LPVOID lpBuffer,

DWORD nNumberOfBytesToWríte, LPDWORD lpNumberOfByteswritten,

LPVOID lpOverlapped);

El manejador del archivo se indica en hFile, la posición de memoria donde copiar los tiatos se especifica en el argumento lpBuffer y el número de bytes a escribir se especifica en lpNumbe— rOfBytesTowrite. La escritura se lleva a cabo a partir de la posición actual del apuntador de posición del archivo. Si la llamada se ejecuta correctamente, devuelve el número de bytes escritos realmente, que pueden ser menos que los pedidos, en pNumberofByteswritten y se incrementa el apuntador de posición del archivo en esa cantidad.]pOverlapped sirve para indicar el maneja dor de evento que se debe usar cuando se hace una escritura no bloqueante (asíncrona). Si ese parámetro tiene algún evento activado, cuando termina la llamada se ejecuta el manejador de dicho evento. Devuelve TRUE en caso de éxito o FALSE en caso de error.

Digitalización realizada con propósito académico

Cambio del apuntador de un archivo

Esta llamada permite cambiar el valor del apuntador de posición de un archivo abierto previamente, de forma que operaciones posteriores de E/S se ejecuten a partir de esa posición. Su prototipo es:

DWORD **SetFilePointer**(HANDLE hFile, LONG iDistanceToMove,

LONG FAR *lpDistanceToMoveHigh, DWORD dwMoveMethod);

El manejador de archivo se indica en hFile, el desplazamiento (positivo o negativo) se indica en iDistanceToMove y el lugar de referencia para el desplazamiento se indica en dwMoveMet hod. Hay tres posibles formas de indicar la referencia de posición: FILE_BEGIN, el valor final del

apuntador es iDistanceToMove; FILE_CURRENT, el valor final del apuntador es el valor actual más (o menos) iDistanceToMove; FILE_END, el valor final es la posición de fin de archivo más (o menos) iDistanceToMove. Si la llamada se ejecuta correctamente, devuelve el nuevo valor del apuñtador de posición del archivo (Consejo de Programación 8.2).

CONSEJO DE PROGRAMACIÓN 8.2

La llamada **`SetFilePointer`** permite averiguar la longitud del archivo especificando `FILE_END` y desplazamiento `0`.

Consulta de atributos de un archivo

Win32 especifica varios servicios para consultar los distintos atributos de un archivo, si bien la llamada más genérica de todas es GetFileAttributes. Sus prototipos son:

DWORD **GetFileAttributes**(LPCTSTR lpszFileName);

BOOL **GetFileTime** (HANDLE hFile, LPFILETIME lpftCreation,

LPFILETIME lpftLastAccess, LPFILETIME lpftLastwrite);

BOOL **GetFileSize** (HANDLE hFile, LPDWORD lpdwFileSize);

La llamada GetFiieAttributes permite obtener un subconjunto reducido de los atributos de un archivo que indican si es temporal, compartido, etc. Dichos atributos se indican como una máscara de bits que se devuelve como resultado de la llamada. La llamada GetFileTime permite obtener los atributos de tiempo de creación, último acceso y última escritura de un archivo abierto. La llamada GetEileSize permite obtener la longitud de un archivo abierto.

Otros servicios de archivos

Win32 describe detalladamente muchos más servicios para archivos que los descritos previamente. Permite manejar descriptores de archivos de entradalsalida estándar (**GetStdHandle, SetS tdHandle**), trabajar con la consola (**SetconsoleMode, ReadConsole, FreeConsole, Allocconsole**), copiar un archivo (copyFi].e), renombrarlo (MoveFile) y realizar operaciones con los atributos de un archivo (**SetFileTime, GetFileTime, GetFileAttributes**).

Se remite al lector a la información de ayuda del sistema operativo Windows NT (botón help), y a los manuales de descripción de Win32, para una descripción detallada de dichos servi ci os.

Digitalización realizada con propósito académico

8.4.5. Ejemplo de uso de servicios Win32 para archivos

Para ilustrar el uso de los servicios de archivos que proporciona el sistema operativo se presenta en esta sección el ejemplo de copia de un archivo sobre otro, pero usando llamadas de Win32.

El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el Programa 8.2. Para ver más ejemplos de programación, consulte [Hart, 1997] y [Andrews, 1996].

Programa 8.2. Copia de un archivo sobre otro usando llamadas Win32.

```
#include <windows . h>
#include <stdio.h>
#define TAMANYOALM 1024
int main (mt argc, LPTSTR argv [ ] )
{ HANDLE hEnt, hSal;
DWORD nEnt, nSal;
CHAR Almacen [TAMANYO_ALM];   /* Se comprueba que el número de argumentos sea el adecuado. En caso contrario, termina y devuelve un error*/
if (argc !=3) {
fprintf (stderr, "Uso: copiar archivol archivo2\n");
return  -1;  }  /* Abrir el archivo origen. Opción: OPEN_EXISTING */
hEnt = CreateFile (argv [ 1 ], GENERICREAD, FILE_SHARE_READ, NULL,
OPENEXISTING, FILE ATTRIBUTE_NORMAL, NULL);
if (hEnt == INVALID_HANDLEVALUE){
fprintf (stderr,
"No se puede abrir el archivo de entrada. Error: %x\n",
GetLastError); /* Consejo de programación 8.3 */  }
return -1; }
/* Crear el archivo destino. Opción: CREATE_ALWAYS */ hSal = CreateFile (argv[2],GENERIC_WRITE, 0, NULL,CREATEALWAYS, FILE ATTRIBUTE NORMAL, NULL);
if (hSal == INVALIDHANDLE_VALUE) {
fprintf (stderr, "No se puede abrir el archivo de salida. Error: %x\n", GetLastError )
return -1; } /* Bucle de lectura del archivo origen y escritura en el destino. Se ejecuta mientras no se llegue al final del archivo origen */
while (ReadFile (hEnt, Almacen, TAMANYO_ALM, &nEnt, NULL)
&& nEnt > 0) { /* Sólo se escriben los datos que se han leído realmente */
WriteFile (hSal, Almacen, nEnt, &nSal, NULL)
if (nEnt nSal) fprintf (stderr,
```

Digitalización realizada con propósito académico

```
                "Error fatal de escritura. Error: %x\n",
               GetLastError () );
            return -1;
      }
  }
CloseHandle (hEnt);
CloseHandle (hSal);
 return O;
}
```

CONSEJO DE PROGRAMACIÓN 8.3

La función `GetLastError` permite imprimir el código de error que el sistema operativo Windows NT devuelve en caso de error de una llamada al sistema. Esta llamada es muy útil para depurar programas.

Para ilustrar el uso de las funciones de conveniencia de servicios de archivos que proporciona Win32 se resuelve también en esta sección el ejemplo de copia de un archivo sobre otro COfl la llamada CopyFile. El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el Programa 8.3. Como se puede ver, comparando con el Programa 8.2, esta función se encarga de abrir y crear los archivos, leer y escribir los datos y cerrar los archivos (Prestaciones 8.2).

PRESTACIONES 8.2

Normalmente las funciones complejas, como `CopyFile`, que proporciona el sistema operativo, son más eficientes que su equivalente programado. Existen varias razones para este comportamiento, pero dos muy importantes son que cada llamada al sistema operativo tiene un coste de ejecución alto y que a nivel interno el sistema operativo usa directamente servicios optimizados e incluso instrucciones máquina complejas que resuelven las tareas muy rápidamente.

Programa 8.3. Copia de un archivo sobre otro usando funciones de conveniencia de Win32.

```
finclude <windows . h>
#include <stdio.h>
mt main (mt argc, LPCTSTR argv [
if (argc 3)
fprintf (stderr, "Uso: copiar archivol archivo2 \n"fl;
return -1;
if (!CopyFile (aigv argv FALSE)) {
   fprintf (stderr, "Error de copia: %x \n", GetLastError());
 }
return O;
}
```

Digitalización realizada con propósito académico

8.4.6. Servicios genéricos de directorios

Un objeto directorio está formado básicamente por un conjunto de entradas que relacionan nombres y archivos y las operaciones para manejar dichas entradas. Estas entradas pueden tener diferente estructura, pero en cualquier caso deben poder ser creadas, borradas, abiertas, cerradas y leídas. Además, es importante poder recorrer la estructura de directorios y cambiar nombres de archivos. En algunos sistemas, como UNIX, se puede también crear y romper enlaces a archivos y directo ri os.

A continuación, se hace una breve descripción de los principales servicios genéricos para directorios.

- **Crear un directorio**. Crea un objeto directorio y lo sitúa en el lugar del árbol de directorios donde se especifique en el nombre, absoluto o relativo, del nuevo directorio. En el caso de POSIX, se añaden automáticamente al nuevo directorio las entradas "." y
- **Borrar un directorio**. Elimina el objeto directorio de forma que nunca más pueda ser acce sible y borra su entrada del árbol de directorios. Habitualmente, y salvo casos especiales. sólo se puede borrar un directorio vacío (sin entradas).
- **Abrir un directorio.** Abre el archivo que contiene el directorio para leer los datos del mismo. Al igual que un archivo, un directorio debe ser abierto para poder manipular su contenido.
- **Cerrar un directorio.** Cierra el archivo que contiene el directorio, liberando los recursos de memoria y del sistema operativo relativos al mismo.
- **Leer un directorio.** Extrae la siguiente entrada de un directorio abierto. Devuelve una es tructura de datos como la que define la entrada de directorio. Por tanto, es específica de cada interfaz o sistema operativo.
- **Cambiar directorio**. Este servicio permite al usuario cambiar su directorio de trabajo.
- **Enlazar.** Permite acceder a un archivo o directorio existente mediante un nuevo nombre. Existe una operación inversa que permite desenlazar una entrada de un directorio. No están disponibles en todos los sistemas operativos, aunque sí en el estándar POSIX.

Los servicios anteriores son los más comunes en distintos sistemas operativos, si bien existen algunos otros que no vamos a .describir en este libro. Se remite al lector a los manuales de cada sistema operativo en particular para ver información acerca de otras llamadas.

8.4.7. Servicios POSIX de directorios

Un directorio en POSIX tiene la estructura lógica de una tabla, cada una de cuyas entradas relaciona un nombre de archivo con su nodo-i (como se describe en la Sección 8.3). Cada entrada de la tabla es una estructura dirent como la siguiente:

```
struct dirent{
    char *d_name; /* nombre del archivo */
    ...
}
```

La gestión de las entradas de directorios se complica debido a que los nombres de archivo pueden ser de longitud variable en POSIX. Para optimizar dicha gestión se almacena información relacionada con la longitud total del registro que alberga cada entrada de directorio. Para facilitar

Digitalización realizada con propósito académico

dicha gestión, y evitar el problema de gestión de las entradas de directorio, el estándar POSIX define los servicios específicos que debe proporcionar un sistema operativo para gestionar los di rectorios. En general, se ajustan a los servicios genéricos descritos en la sección anterior. A conti nuación se describen los servicios de directorios más comunes del estándar POSIX.

Crear un directorio

El servicio mkdir permite crear un nuevo directorio en POSIX. Su prototipo es:

```
int mkdir (const char *path, mode_t mode);
```

Esta llamada al sistema crea el directorio especificado en path con el modo de protección especificado en mode. El dueño del directorio es el dueño del proceso que crea el directorio. En caso de éxito devuelve un cero. En caso de error un -1.

Borrar un directorio

El estándar POSIX permite borrar un directorio especificando su nombre. El prototipo de la llama da es:

int **rmdir** (const char *path);

El directorio se borra únicamente cuando está vacío. Para borrar un directorio no vacío es obligatorio borrar antes sus entradas. Esta llamada devuelve cero en caso de éxito y —l en caso de error. El comportamiento de la llamada no está definido cuando se intenta borrar el directorio raíz o el directorio de trabajo. Si el contador de enlaces del directorio es mayor que cero, se decrementa dicho contador y se borra la entrada del directorio, pero no se liberan los recursos del mismo hasta que el contador de enlaces es cero.

Abrir un directorio

La función opendir abre el directorio de nombre especificado en la llamada y devuelve un identi ficador de directorio. Su prototipo es.

DIR ***opendir**(const char *dirname);

El apuntador de posición indica a la primera entrada del directorio abierto. En caso de error

devuelve NULL.

Cerrar un directorio

Un directorio abierto, e identificado por dirp, puede ser cerrado ejecutando la llamada:

int **closedir** (DIR *dirp);

En caso de éxito devuelve O y -l en caso de error.

Digitalización realizada con propósito académico

Leer un directorio

Este servicio permite leer de un directorio abierto, obteniendo como resultado la siguiente entrada del mismo. Su prototipo es:

struct dirent ***readdir** (DIR *dirp);

En caso de error, o de alcanzar el final del directorio, devuelve NULL. Nunca devuelve entradas cuyos nombres están vacíos.

Cambiar de directorio

Para poder viajar por la estructura de directorios, POSIX define la llamada chdir, cuyo prototipo es:

int **chdir** (const char *path);

El directorio destino se especifica en path. Si se realiza con éxito, esta llamada cambia el directorio de trabajo, es decir, el punto de partida para interpretar los nombres relativos. Si falla, no se cambia al directorio especificado.

Crear un enlace a un directorio

POSIX permite que un archivo o directorio pueda ser accedido usando nombres distintos. Para ello, proporciona las llamadas link y symlink para enlazar un archivo o directorio a otro usando enlaces físicos o simbólicos, respectivamente. Sus prototipos son:

int **link** (const char *exmstmng, const char *new);

int **symlink** (const char *exjstjng, const char *new);

La llamada link establece un enlace físico desde una nueva entrada de directorio, new, a un archivo ya existente, existing. Ambos nombres deben pertenecer al mismo sistema de archivos. En caso de éxito, crea la nueva entrada de directorio, incrementa el contador de enlaces del nodo-i del directorio o archivo existente y devuelve un cero.

La llamada syinlink establece un enlace simbólico desde una nueva entrada de directorio, new, a un archivo ya existente, existing. No es necesario que ambos nombres pertenezcan al mismo sistema de archivos. En caso de éxito, crea la nueva entrada de directorio del enlace y devuelve un cero.

En caso de error, las llamadas no crean el enlace y devuelven -1.

Otras llamadas POSIX

El estándar POSIX define algunas llamadas más pala servicios de directorios. La llamada **rewinddir** permite volver al principio del directorio. **getcwd** devuelve el nombre absoluto del directorio de trabajo actual. **unlink** puede usarse para borrar entradas de un directorio. Por último, la llamada **rename** permite cambiar el nombre de un directorio, creando una entrada nueva y eliminando el nombre anterior.

Es interesante resaltar la inexistencia de una llamada para escribir en un directorio, equivalen te a **readdir**. Esto es así para que el sistema operativo pueda garantizar la coherencia del árbol de directorios, permitiendo únicamente añadir o borrar entradas, y cambiar nombres, a través de llama das específicas para ello (Recordatorio 8.1).

Digitalización realizada con propósito académico

RECORDATORIO 8.1

Como se dijo en secciones anteriores, los directorios se suelen implementar mediante archivos. Por ello, aunque existen las llamadas específicas descritas para manejar directorios, se pueden obtener los mismos resultados usando llamadas de acceso a archivos y manipulando las estructuras de datos de los directorios.

8.4.8. Ejemplo de uso de servicios POSIX para directorios

En esta sección se presentan dos pequeños programas en lenguaje C que muestran las entradas de un directorio, cuyo nombre se recibe como argumento.

El Programa 8.4 muestra dichas entradas por la salida estándar del sistema. Para ello, el pro grama ejecuta la siguiente secuencia de acciones:

1. Muestra el directorio actual.
2. Imprime el directorio cuyo contenido se va a mostrar.
3. Abre el directorio solicitado.
4. Mientras existan entradas en el directorio, lee una entrada e imprime el nombre de la misma.
5. Cierra el directorio.

El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el

Programa 8.4.

Programa 8.4. Programa que muestra las entradas de un directorio usando llamadas POSIX.

#include <sys/types .h> #include <dirent .h> #include <stdio.h> #include <error.h>

*define TAMANYOALM 1024

void main (mt argc, char **argv)

DIRP *dirp; struct dirent *dp; char almacen /* Comprueba que el número de

En caso contrario, termina

if (argc 2)

fprintf (stderr, “Uso: mi_ls directorio\n”);

exit(1)

/* Obtener e imprimir el directorio actual */ **getcwd** (almacen, TAMANYO_ALM);

printf (“Directorio actual: %s \n”, almacen); /* Abrir el directorio recibido como argumento *1 dirp = **opendir** (argv [1]);

Digitalización realizada con propósito académico

```
if (dírp == NULL) {
   fprintf (stderr, "No se pudo abrir el directorio: %s \n",
   argv [ 1]);
   perror ();
   exit (1);
}
else
{
    printf ("Entradas en el directorio: %s \n". argv /* Toer el directorio entrada a entrada */
    while ((dp = readdir(dirp)) MILL)
    printf ("%s\n", dp->d_name); /* Imprimir nombre* /
 closedir (dirp);
}
exit (O);
}
```

Obsérvese que la ejecución de este programa ejemplo no cambia el directorio de trabajo ac tual. Por ello, si se usan nombres absolutos, y se tienen los permisos adecuados, se pueden ver los contenidos de cualquier directorio sin movernos del actual. Además del nombre, la estructura de directorio incluye otra información que puede verse sin más que añadir las instrucciones para impri mir su contenido. El mandato ls de UNIX se implementa de forma similar al ejemplo.

El Programa 8.5 copia el contenido del directorio a un archivo, cuyo nombre se recibe como parámetro. Para ello, ejecuta acciones similares a las del Programa 8.4, pero además redirige el descriptor de salida estándar hacia dicho archivo. El mandato ls > archivo de UNIX se imple menta de forma similar al ejemplo.

Programa 8.5. Programa que copia las entradas de un directorio a un archivo.

```
#include <sys/types.h>
#include <dirent .h>
#include <stdio.h> #include <error.h>
#define TAMANYO_ALM 1024
void main (mt argc, char **argv)
D]1RP *dirp;
struct dirent *dp;
char almacen{TANANYO_ALM]; mt fd;   /* Comprueba que el número de parámetros sea el adecuado. En caso contrario, termina con un error */
if (argc  |= 4)
fprintt (stderr, "Uso: mi_ls directorio > archivo \n"); exit(l)
/* Crear el archivo recibido como argumento /
```

Digitalización realizada con propósito académico

```
fd = creat (argv , 0600);
if (fd == —1)
fprintf (stderr, “Error al crear archivo de salida: %s \n”. argv
perror ()
exit (1);
/ Redirigir la salida estándar al archivo */
close(stdout);    /* Cierra la salida estándar */
dup(fd);          /* Redirige fd a stdout /
close(fd);        /* Cierra fd */
```

/* Ahora “archivo” está accesible por la salida estándar */ /* El resto del programa sigue igual que el Programa 8.4 */ /* Obtener e imprimir el directorio actual */ **getcwd** (almacen, TANANYOALM); printf (“Directorio actual: %s \n”, almacén); /* Abrir el directorio recibido como argumento */

```
 dirp = opendir (argv [ 1] );
if (dirp == NULL)
{
    fprintf (stderr, “No se pudo abrir el directorio: %s \n”,
     argv [ 1] );
     perror ();
     exit (1);
}
else  {
     printf (“Entradas en el directorio: %s \n”, argv / Leer el directorio entrada a entrada /
     while ((dp = readdir(dirp)) NULL)
      printf (“%s\n”, dp->d_name); /* Imprimir nombre */
      closedir (dirp);
}  exit (0);
}
```

Obsérvese que el programa es idéntico al 8.4, excepto en el bloque inicial, en el que se redirige la salida estándar al archivo solicitado. El mismo mecanismo se usa en POSIX cuando hay redirec ción de la salida de un mandato a un archivo.

8.4.9. Servicios Win32 para directorios

A continuación se describen los servicios más comunes de Win32 para gestión de directorios.

Crear un directorio

El servicio CreateDirectory permite crear un nuevo directorio en Win32. Su prototipo es:

```
BOOL CreateDirectory (LPCSTR lpPathName,
        LPVOID lpSecurityAttributes);
```

Esta llamada al sistema crea el directorio especificado en lpPathName con el modo de protec ción especificado en lpSecurityAt tributes. El nombre del directorio puede ser absoluto para el dispositivo en que está el proceso o relativo al directorio de trabajo. En caso de éxito devuelve

TRUE.

Borrar un directorio

Win32 permite borrar un directorio especificando su nombre. El prototipo de la llamada es:

BOOL **RemoveDirectory** (LPCSTR lpszPath);

El directorio lpszpath se borra únicamente cuando está vacío. Para borrar un directorio no

vacío es obligatorio borrar antes sus entradas. Esta llamada devuelve TRUE en caso de éxito.

Leer un directorio

En Win32 hay tres servicios que permiten leer un directorio: FindFirstFile, FindNextFiie y FindClose. Su prototipos son:

```
HANDLE      FindFirstFile (LPCSTR lpFileName,
                LPWin32_FIND_DATA lpFindFileData);
BOOL        FindNextFi1e(HAJ hFindFile,
                LPWin32_FIND_DATA lpFindFileData);
BOOL  FindClose(HANDLE hFindFile);
```

FindFirstFile es equivalente al opendir de POSIX. Permite obtener un manejador para buscar en un directorio. Además, busca la primera ocurrencia del nombre de archivo especificado en lpFileName. FindNextFile es equivalente al readdir de POSIX, permite leer la siguiente entrada de archivo en un directorio. FindClose es equivalente a closedir de POSIX y permite cerrar el manejador de búsqueda en el directorio.

Cambiar el directorio de trabajo

Para poder viajar por la estructura de directorios, Win32 define la llamada SetCurrentDirec tory, cuyo prototipo es:

BOOL **SetCurrentDirecto**ry (LPCTSTR lpszcurDir);

El directorio destino se especifica en lpszCurDir. Si se realiza con éxito, esta llamada cam bia el directorio de trabajo, es decir, el punto de partida para interpretar los nombres relativos. Si falla, no se cambia al directorio especificado. Si se especifica un nombre de dispositivo, como A: \ O D: \, se cambia el directorio de trabajo en ese dispositivo.

Obtener el directorio de trabajo

Win32 permite conocer el directorio de trabajo actual mediante la llamada GetCurrentDirec tory, cuyo prototipo es:

DWORD **GetCurrentDirectory** (DWORD cchCurDir, LPCTSTR lpszCurDir);

Digitalización realizada con propósito académico

el directorio se devuelve en lpsCurDir. La longitud del string se indica en cchCurDir. Si el almacen es de longitud insuficiente, la llamada indica como deberia ser de grande.

8.4.10. Ejemplo de uso de servicios Win32 para directorios

Para ilustrar el uso de los servicios de archivos que proporciona Win32, se resuelve en esta sección un ejemplo que muestra las entradas del directorio de trabajo. El código fuente, en lenguaje C, del ejemplo propuesto se muestra en el Programa 8.6.

Programa 8.6. Programa que muestra las entradas del directorio de trabajo usando llamadas Win32.

*include <windows . h>

*inclLude <direct.h> #include <stdio.h>

#define LONGITUD NOMBRE MAX_PATH + 2

mt main (mt argc, LPTSTR argv {])

BOOL Flags [LPTSTR pSlash, pNombreArchivo; mt i, IndiceArchivo; /* Almacén para el nombre del directorio */ TCHAR Almacenpwd [DWORD CurDirLon; / Manejador para el directorio */ HANDLE SearchHandle; Win32_FINDDATA FindData;

/* Obtener el nombre del directorio de trabajo *1

CurDirLon = **GetCurrentDirectory** (LONGITUD_NOMBRE, Almacenpwd);

if (CurDirLon == O) printf(”Eallo al obtener el directorio. Error: %x\n”, GetLastError) return —1;

if (CurDirLon > LONGITUD_NOMBRE) printf(de directorio demasiado largo. Error: %x\n”, GetLastError ());

return -1;

/* Todo está bien. Leer e imprimir entradas / / Encontrar el \ del final del nombre, si lo hay. Si no, añadirlo y restaurar el directorio de trabajo */ pSlash = strrchr (Almacenpwd, ‘\\‘); if (pSlash NULL) {

*pSlash = ‘\O’;

```
SetCurrentDirectory (Almacenpwd);
*pSlash =
pNombreArchivo = pSlash + 1;
else
pNombreArchivo = Almacenpwd;
/* Abrir el manejador de búsqueda del directorio y obtener
el primer archivo. */
SearchHandle = FindFirstFile (Almacenpwd, &FindData);
if (SearchHandle == INVALID HANDLE VALUE)
printf ("Error %x abriendo el directorio.\n".
GetLastError (H;
return —1;
/* Buscar el directorio /
do
printf (" %s \n", FindData.cFileName);
/* Obtener el siguiente nombre de directorio. */
while (FindNextFile (SearchHandle, &FindData));
return O;
}
```

La estructura FindData contiene bastante más información que la mostrada en este programa.

Se deja como ejercicio la extensión del Programa 8.5 para que muestre todos los datos de cada entrada de directorio.

8.5. SISTEMAS DE ARCHIVOS

El sistema de archivos permite organizar la información dentro de los dispositivos de almacena miento secundario en un formato inteligible para el sistema operativo. Habitualmente, cuando se instala el sistema operativo, los dispositivos de almacenamiento están vacíos. Por ello, previa mente a la instalación del sistema de archivos es necesario dividir física o lógicamente los discos en **particiones** o **volúmenes** [Pinkert, 1989]. Una **partición** es una porción de un disco ala que se la dota de una identidad propia y que puede ser manipulada por el sistema operatn'() como una entidad lógica independiente. Admite formato, instalación de sistemas de archivos, comproba ciones, etc. Este objeto no es utilizable directamente por la parte del sistema operativo que gestio na los archivos y directorios, que debe instalar un sistema de archivos dentro de dicha partición (Aclaración 8.3). Además, en el caso de discos de arranque del sistema, los sistemas operativos exigen que existan algunas particiones dedicadas a propósitos específicos de los mismos, tales como arranque del sistema (partición raíz), intercambio de páginas de memoria virtual (partición de intercambio), etc. La Figura 8.13 muestra la diferencia entre partición y disco. Un disco puede dividirse en varias particiones o puede contener una única partición. Además, algunos sistemas operativos modernos permiten construir particiones extendidas que engloban varias unidades de disco.

Digitalización realizada con propósito académico

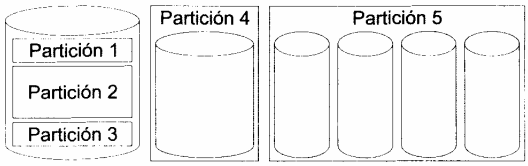


Figura 8.13. *Tipos de particiones.*

ACLARACIÓN 8.3

El sistema operativo puede acceder directamente a la partición como si fuera un dispositivo orientado a caracteres, pero con acceso a bloque. En este caso no es necesario que la partición tenga instalado un sistema de archivos. Este tipo de accesos, habituales en el mundo de las bases de datos, son muy raros a nivel de usuario de sistema operativo de propósito general. En muchos casos hay restricciones de seguridad que hacen que sólo el software de sistema pueda llevar a cabo este tipo de accesos. Además, cuando se accede a una partición de esta forma, es obligatorio alinear los accesos para que empiecen siempre en un múltiplo del tamaño de sector de la partición. Sin embargo, son habituales en aplicaciones como servidores de bases de datos, que gestionan los discos de forma especial.

Una vez creadas las particiones, el sistema operativo debe crear las estructuras de los sistemas de archivos dentro de esas particiones. Para ello se proporcionan mandatos como format o mkfs al usuario. Los siguientes mandatos de UNIX crean dos sistemas de archivos, uno para intercambio y otro para datos, dentro de dos particiones del disco duro a: /dev/hda2 y /dev/hda3:

#mkswap -c /dev/hda2 20800

#mkfs —c /dev/hda3 —b 8192 123100

El tamaño del sistema de archivos, por ejemplo 20800, se define en bloques. Un **bloque** se define como una agrupación lógica de sectores de disco y es la unidad de transJ mínima que usa el sistema de archivos. Se usan para optimizar la eficiencia de la entradalsalida de los dispositi vos secundarios de almacenamiento. Aunque todos los sistemas operativos proporcionan un tamaño de bloque por defecto, en UNIX, los usuarios pueden definir el tamaño de bloque a usar dentro de un sistema de archivos mediante el mandato mkfs. Por ejemplo, -b 8192 define un tamaño de bloque de 8 KB para /dev/hda3 (Prestaciones 8.3). El tamaño de bloque puede variar de un sistema de archivos a otro, pero no puede cambiar dentro del mismo sistema de archivos. En mu chos casos, además de estos parámetros, se puede definir el tamaño de la **agrupación**, es decir, el conjunto de bloques que se gestionan como una unidad lógica de gestión del almacenamiento. El problema que introducen las agrupaciones, y los bloques grandes, es la existencia de fragmentación interna. Por ejemplo, si el tamaño medio de archivo es de 6 KB, un tamaño de bloque de 8 KB introduce una fragmentación interna media de un 25 por 100. Si el tamaño de bloque fuera de 32 KB, la fragmentación interna alcanzaría casi el 80 por 100.

Digitalización realizada con propósito académico

PRESTACIONES 8.3

Aunque no es obligatorio, el tamaño de bloque suele ser múltiplo par del número de sectores del disco por cuestiones de rendimiento del disco y de sencillez de implementación del sistema de archivos.

Todos los sistemas operativos de propósito general incluyen un componente, denominado servidor de archivos, que se encarga de gestionar todo lo referente a los sistemas de archivos. Este componente se estudia en detalle en la Sección 8.6.

8.5.1. Estructura del sistema de archivos

Cuando se crea un sistema de archivos en una partición de un disco, se crea una entidad lógica autocontenida con espacio para la información de carga del sistema de operativo, descripción de su estructura, descriptores de archivos, información del estado de ocupación de los bloques del sistema de archivos y bloques de datos Goodheart, 1994]. La Figura 8.14 muestra las estructuras de un sistema de archivos para MS-DOS, UNIX y Windows NT, así como la disposición de sus distintos componentes, alwacenado en una partición.

En UNIX, cada sistema de archivos tiene un bloque de carga (Boot) que contiene el código que ejecuta el programa de arranque del programa almacenado en la ROM de la computadora. Cuando se arranca la máquina, el iniciador ROM lee el bloque de carga del dispositivo que almace na al sistema operativo, lo carga en memoria, salta a la primera posición del código y lo ejecuta. Este código es el que se encarga de instalar el sistema operativo en la computadora, leyéndolo desde el disco. No todos los sistemas de archivos necesitan un bloque de carga. En MS-DOS o UNIX, por ejemplo, sólo los dispositivos de sistema, es decir, aquellos que tienen el sistema opera tivo instalado contienen un bloque de carga válido. Sin embargo, para mantener la estructura del sistema de archivos uniforme, se suele incluir en todos ellos un bloque reservado para carga. El

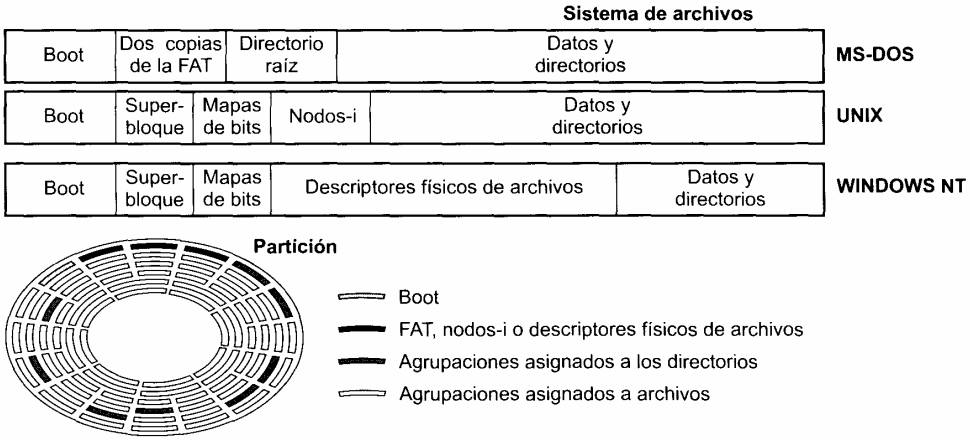


Figura 8.14. *Estructura de distintos sistemas de archivos.*

Digitalización realizada con propósito académico

sistema operativo incluye un número, denominado **número mágico**, en dicho bloque para indicar que es un dispositivo de carga. Si se intenta cargar de un dispositivo sin número mágico, o con un valor erróneo del mismo, el monitor ROM lo detecta y genera un error.

A continuación del bloque de carga está la metainformación del sistema de archivos (superblo que, nodos-i,...). **La metainformación** describe el sistema de archivos y la distribución de suis componentes. Suele estar agrupada al principio del disco y es necesaria para acceder al sistema de archivos. El primer componente de la metainformación en un sistema de archivos UNIX es el **superbloque** (el Bloque de Descripción del Dispositivo en Windows NT), que contiene la informa ción que describe toda la estructura del sistema de archivos. La información contenida en el super bloque indica al sistema operativo las características del sistema de archivos dónde están los distin tos elementos del mismo y cuánto ocupan. Para reducir el tamaño del superbloque, sólo se indica la información que no puede ser calculada a partir de otra. Por ejemplo, si se indica que el tamaño del bloque usado es 4 KB, que el sistema tiene 16 K nodos-i y que el nodo-i ocupa 512 bytes, es sencillo calcular que el espacio que ocupan los nodos-i en el sistema de archivos es 2.048 bloques. Asimis mo, si se usan mapas de bits para representar el estado de ocupación de los nodos-i, necesitaremos 16 KB, lo que significa que el mapa de bits de los nodos-i ocuparía 4 bloques del sistema de archivos. La Figura 8.15 muestra una parte del superbloque de un sistema de archivos de LINUX. Como se puede ver, además de la información reseñada, se incluye información común para todos los sistemas de archivos que proporciona el sistema operativo y una entrada para cada tipo de archivos en particular (MINIX, MS-DOS, ISO, NFS, SYSTEM V, UFS, genérico, etc.). En este caso se incluye información de gestión tal como el tipo del dispositivo, el tamaño de bloque, núme

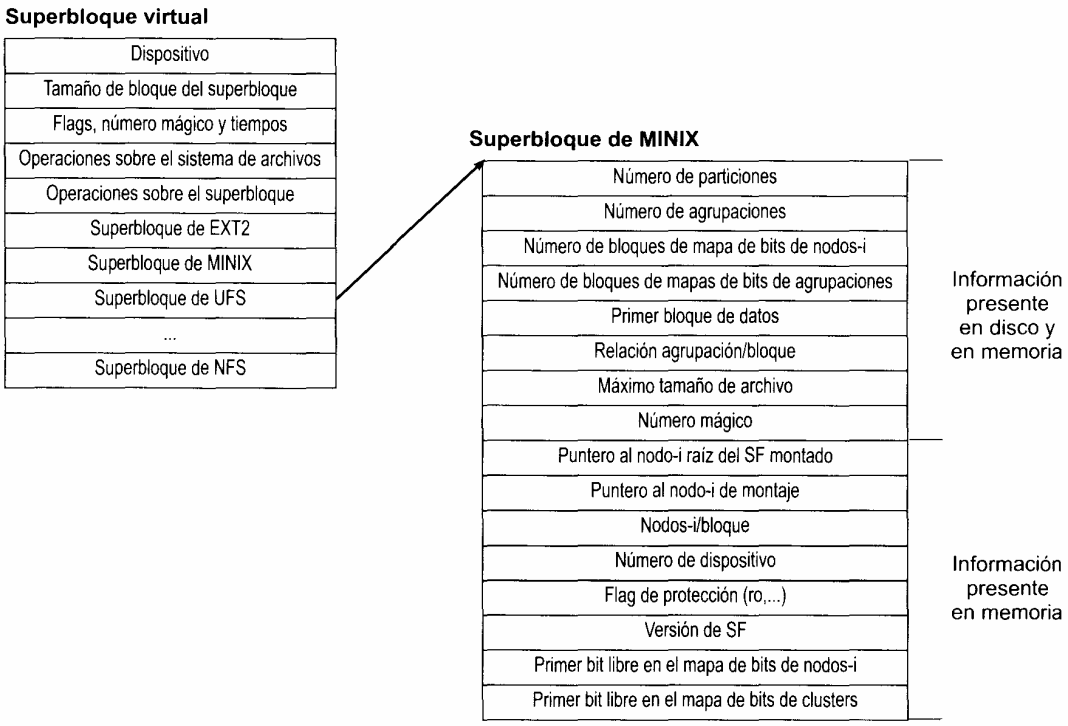


Figura 8.15. *Superbloque de sistemas de archivos en LINUX.*

Digitalización realizada con propósito académico

ro mágico, tipo de archivo, operaciones sobre el superbloque y de cuota de disco y apuntadores a los tipos de archivo que soporta. Para cada uno de ellos se incluye una estructura de datos para su superbioque particular, el máximo tamaño de archivo posible, la protección que se aplica al sistema de archivos, etc. Además, para optimizar aspectos como la búsqueda de espacio libre, se incluye información sobre la situación del primer bloque libre y del primer descriptor de archivos libre. Como ejemplo de superbloque particular se muestra el de MINIX.

Cuando arranca la computadora y se carga el sistema operativo, el superbioque del dispositivo de carga, o sistema de archivos raíz, se carga en memoria en la **tabla de superbioques**. A medida que otros sistemas de archivos son incorporados a la jerarquía de directorios (en UNIX se dice que son montados) sus superbloques se cargan en la tabla de superbloques existente en memoria. La forma de enlazar unos con otros se verá más adelante.

Tras el superbioque, el sistema de archivos incluye **información de gestión de espacio** en el disco. Esta información es necesaria por dos razones: para permitir al servidor de archivos imple- mentar distintas políticas de asignación de espacio y para reutilizar los recursos liberados para nuevos archivos y directorios. Normalmente, los sistemas de archivos incluyen dos mapas de espa cio libre:

• información de **bloques de datos**, en la que se indica si un bloque de datos esta ibre o no. En caso de que el espacio de datos se administre con agrupaciones para optimizar 'la gestión de espacio libre, esta información se refiere a as agrupaciones (Prestaciones .4).

• Información de la **descripción física de los archivos**, como nodos-i en UNIX o registros de Windows NT, en la que se indica si un descriptor de archivo está libre o no.

PRESTACIONES 8.4

En algunos sistemas se puede trabajar en el ámbito de *fragmento* de agrupación o bloque para disminuir la fragmentación interna del sistema de archivos. En estos sistemas, además de guardar información acerca de las agrupaciones, es necesario guardar información acerca de las agrupaciones de la que sólo se han usado ciertos fragmentos. Esta característica se explicará en detalle cuando se explique el *Fast File System* (Sección 8.5.2).

Existen distintas formas de representar el espacio existente en un sistema de archivos. Las más populares son los mapas de bits y las listas de recursos libres. Estos mecanismos se estudian en detalle en una sección posterior.

Después de los mapas de recursos del sistema de archivos, se encuentran los descriptores físicos de archivos. Estos descriptores, sean nodos-i de UNIX o registros de Windows NT, tienen una estructura y tamaño variable dependiendo de cada sistema operativo, según se vio en secciones anteriores. Por ejemplo, en LINUX ocupa 128 bytes EBokhari, 19951 y en Windows NT el registro ocupa todo un bloque de 4 KB. El tamaño del área de descriptores de archivo es fácilmente calcula ble si se conoce el tamaño del nodo-i y el número de nodos-i disponibles en el sistema de archivos. Normalmente, cuando se crea un sistema de archivos, el sistema operativo habilita un número de descriptores de archivo proporcional al tamaño del dispositivo. Por ejemplo, en LINUX se crea un nodo-i por cada 2 bloques de datos. Este parámetro puede ser modificado por el usuario cuando crea un sistema de archivos, lo que en el caso de UNIX se hace con el mandato mkfs (Advertencia 8.2).

Digitalización realizada con propósito académico

ADVERTENCIA 8.2

¡Cuidado! Puesto que cada archivo necesita un descriptor, el número de descriptores es el número máximo de archivos que puede haber en un sistema de archivos. Una vez creado el sistema de archivos no se puede modificar este parámetro, por lo que el usuario que lo define debe saber algo respecto al uso esperado para el dispositivo. Si va a haber muchos archivos pequeños, necesitará muchos descriptores. En caso contrario, puede limitar el número de los mismos para ganar espacio de datos.

El último componente del sistema de archivos son los **bloques de datos**. Estos bloques, bien tratados de forma individual o bien en grupos, son asignados a los archivos por el servidor de archivos, que establece una correspondencia entre el bloque y el archivo a través del descriptor del archivo. Tanto si se usan bloques individuales como agrupaciones, el tamaño de la unidad de acceso que se usa en el sistema de archivos es uno de los factores más importantes en el rendi miento de la entrada/salida del sistema operativo. Puesto que el bloque es la mínima unidad de transferencia que maneja el sistema operativo, elegir un tamaño de bloque pequeño, por ejemplo 5 1 2 bytes, permite aprovechar al máximo el tamaño del disco. Así, el archivo prueba de 1,2 KB ocuparía 3 bloques y sólo desperdiciaría 1/2 bloque o el 20 por 100 del espacio de disco si todos los archivos fuesen de ese tamaño. Si el bloque fuese de 32 KB, el archivo ocuparía un único bloque y desperdiciaría el 90 por 100 del bloque y del espacio de disco. Ahora bien, transferir el archivo prueba, en el primer caso, necesitaría la transferencia de 3 bloques, lo que significa buscar cada bloque en el disco, esperar el tiempo de latencia y hacer la transferencia de datos. Con bloques de 32 KB sólo se necesitaría una operación. La Figura 8.16 muestra la relación entre tamaño de bloque, velocidad de transferencia y porcentaje de uso del sistema de archivos, con un tamaño medio de archivo de 14 KB. Como puede verse, los dos últimos parámetros son contra dictorios, por lo que es necesario adoptar una solución de compromiso que proporcione rendi mientos aceptables para ambos. En el caso del dispositivo cuyos resultados se muestran en la figura, el tamaño de bloque puede estar entre 4 KB y 8 KB. Es interesante resaltar que esta curva cambia con la tecnología de los discos y con el tamaño medio de los archivos, que ha cambiado desde 1 KB en los sistemas UNIX de hace 15 años hasta los 14 KB medidos en estudios más actuales. Además, el uso masivo de información multimedia tenderá a incrementar mucho el tamaño medio de los archivos en un futuro próximo, por lo que es necesario evaluar los parámetros cuidadosamente y

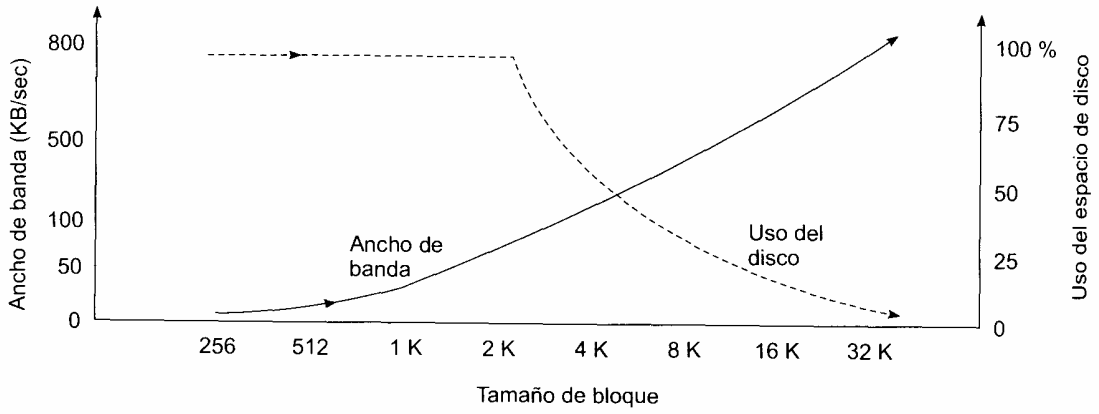


Figura 8.16. *Relación entre el tamaño de bloque, el ancho de banda y el uso del disco.*

ajustarlos al tipo de almacenamiento que se llevará a cabo en cada dispositivo (general, científico, multimedia, etc.).

Como se puede ver en la Figura 8.14, el **sistema de archivos de Windows NT** [Nagar, 1997] tiene una estructura similar al de UNIX, pero los descriptores físicos de archivos ocupan mucho más espacio en la partición. Esto se debe a que los descriptores de Windows NT tienen un tamaño de 4 KB y no unos pocos bytes como en UNIX. Esta estrategia permite almacenar datos en el descriptor de archivos, lo que puede optimizar el acceso a los archivos pequeños al evitar accesos a disco para obtener el descriptor y después los datos. Esta cuestión se estudia en detalle en la Sección 8.6.

8.5.2. Otros tipos de sistemas de archivos

En algunos sistemas operativos se incluyen sistemas de archivos con una estructura diferente a las presentadas en la sección anterior. Los objetivos de estos nuevos sistemas de archivos son variados e incluyen optimizaciones del rendimiento de la entradalsalida, incrementar la fiabilidad u ofrecer información acerca de objetos internos del sistema operativo (como los procesos). La estructura de sistema de archivos mostrada anteriormente tiene varios problemas:

- La metainformación del sistema de archivos (superbioque, nodos-i,...) está agrupada al prin 1 cipio del disco y es única. Ello tiene dos consecuencias. Primero, el tiempo de búsqueda de bloques es muy largo. Segundo, si se conompe algún bloque de metainformación, todo el sistema de archivos queda inutilizado. El sistema **FFS** (fast file system) de UNIX y el **EXT2** e (extended file system) de LINUX resuelven este problema de forma similar.
- El sistema de archivos no se puede extender a varias particiones de forma natural, con lo que el tamaño máximo de un archivo está limitado por la partición. Por ejemplo, en una partición de 2 GB de UNIX, ése sería el máximo tamaño de archivo, aun cuando el sistema pudiera manejar archivos de mayor tamaño. Los sistemas de archivos como los de los dispositivos **RAID**, los **sistemas de archivos paralelos** o los sistemas de **archivos con bandas** de Win dows NT resuelven este problema.
- No se explotan adecuadamente las distintas características de las operaciones de lectura y de escritura con distintos tamaños. Un problema especialmente importante son las escrituras más pequeñas que un bloque, habitualmente denominadas **escrituras parciales**. Para que la información quede coherente, hay que leer la información del bloque, modificarla en memo ria y escribirla de nuevo al disco. Dado que en los sistemas de propósito general la mayoría de las escrituras son pequeñas, la estructura de sIstema de archIvos tradIcIonal es muy poco eficiente para este tipo de operaciones. El sistema LFS (log structureed file system) resuelve este problema.
- En los antiguos sistemas operativos, la información de los procesos estaba oculta a los usuanos y sólo se podía ver mediante mandatos del sistema operativo o usando monitores. Los sistemas operativos modernos incluyen un sistema de archivos de tipo especial, denominado **proc**, dentro del cual incluyen directorios con la información de cada proceso identificada mediante el **pid**. Estos sistemas de archivos sólo existen en memoria y se crean cuando arranca el sistema operativo. La información de cada proceso se construye dinámicamente cuando es necesario.

Fast File System (FFS)

La primera version del FFS apareció en 1984 como parte del UNIX-BSD, desarrollado en la Uni versidad de Berkeley en 1984 [McKusick, 1984]. Actualmente se ha convertido en el formato

Digitalización realizada con propósito académico

estándar de gran parte de versiones del sistema operativo UNIX. El objetivo de este sistema de archivos es doble: reducir tiempos de búsqueda en el disco e incrementar la fiabilidad. Para ello se divide la partición en varias áreas, denominadas **grupos de cilindros o de bloques**. Cada grupo de cilindros contiene una copia del superbioque, nodos-i y mapas de bits para los bloques de ese cilindro. La motivación para usar grupos de cilindros es repartir los datos y asociar a dichos datos la metainformación correspondiente, de forma que esté más cercana a los bloques referenciados en ella. Además, la replicación del superbioque permite restaurar posibles fallos en uno de los super bloques. La Figura 8. 17 muestra la disposición del sistema de archivos en una partición.

Cuando se crea el sistema de archivos, el administrador debe especificar el tamaño de los grupos de cilindros, el número de nodos-i dentro de cada grupo y el tamaño del bloque. El servidor de archivos usa valores por defecto en otro caso. Además de los grupos de cilindros, el FFS incluyó en su momento otras optimizaciones:

• El servidor de archivos intenta asignar los bloques de un archivo cercanos a su nodo- y dentro del mismo grupo de cilindros. Además, los nodos-i de los archivos se intentan colocar

en el mismo grupo de cilindros que el directorio donde se encuentran.

• Los nodos-i de los archivos creados se distribuyen por toda la partición, para evitar la con gestión de un único grupo de cilindros.

• El tamaño de bloque por defecto se aumentó de 1 KB, usado normalmente hasta ese momen to, hasta 4 KB.

Con estas optimizaciones. el FFS consiguió duplicar el rendimiento del servidor de archivos que usaba UNIX System V, la versión más popular de UNIX del momento. Sin embargo, la funcio nalidad interna del servidor de archivos se complicó considerablemente, debido fundamentalmente a dos problemas:

• Cuando se llena un grupo de bloques, hay que colocar los datos restantes en otros grupos. Ello conlleva tener una política de dispersión de bloques e información de control extra para saber dónde están dichos bloques.

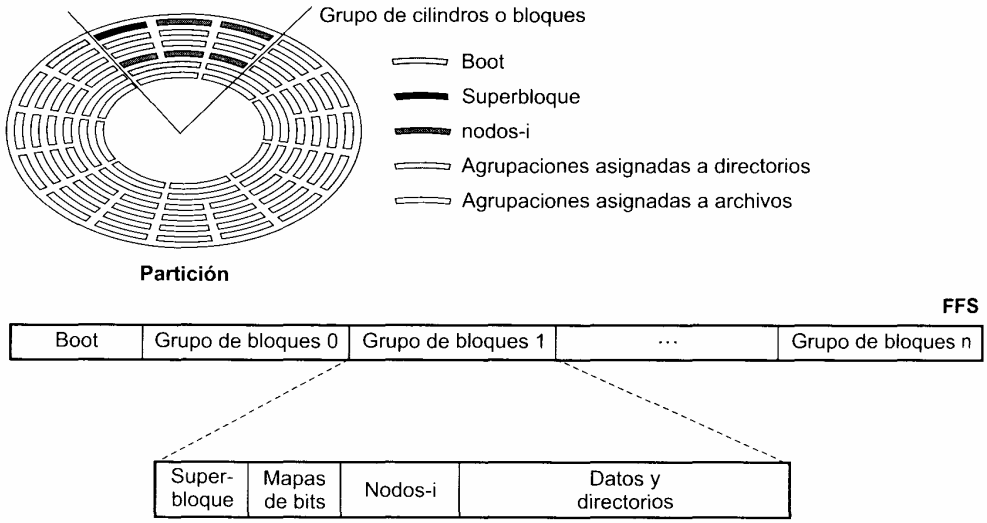


Figura 8.17. *Estructura de un sistema de archivos del FFS.*

Digitalización realizada con propósito académico

• Al incrementar el tamaño del bloque se incrementa la fragmentación interna. En 1984, el tamaño medio de archivo en UNIX era de 1,5 KB FOusterhout, 1985}, lo que significaba que, si se dedicaba un bloque de 4 KB a un archivo, más del 50 por 100 del bloque quedaba sin usar. Pasa solventar este problema fue necesario introducir el concepto de fragmento de un bloque.

Un **fragmento** es una porción de bloque del sistema de archivos que se puede asignar de fbi-ma independiente. Obsérvese que cuando hay fragmentos, no siempre es cierto que la unidad mínima de asignación es un bloque, por lo que es necesario gestionar también dichos fragmentos. La gestión de fragmentos está muy relacionada con la asignación de bloques nuevos a un archivo. Cuando un archivo crece, necesita más espacio del servidor de archivos, en cuyo caso existen dos posibilidades:

- El archivo no tiene ningún bloque fragmentado. En este caso, silos datos ocupan un bloque o más, se le asignan bloques completos. Si los datos se ajustan al tamaño de los bloques, la operación termina aquí. En caso de que al final quede una porción de datos menor que un bloque, se asigna al archivo los fragmentos de bloque que necesite para almacenar dicha porción de datos.
- El archivo tiene un bloque fragmentado al final. En este caso, si los datos ocupan más que el tamaño de bloque restante, se asigna al archivo un nuevo bloque, se copian los fragmentos al principio del bloque y los datos restantes a continuación. En esta situación ya se puede aplicar el caso 1. En caso de que los datos a almacenar no llenen un bloque, se asignan más fragmentos al archivo.

Observe que un archivo sólo puede tener un bloque fragmentado al final, ya que el último será el único bloque sin llenar totalmente. Por tanto, el servidor de archivos sólo debe mantener informa ción de fragmentos para ese último bloque. Para seguir la pista a los fragmentos, el FFS mantiene una tabla de descriptores de fragmentos, en la que se indica el estado de los fragmentos en los bloques fragmentados.

Sistemas de archivos con bandas (Stripes)

El sistema de archivos con bandas de Windows NT Nagar, 19971 permite crear sistemas de archi vos que ocupan varias particiones. Su estructura distribuye los bloques de datos de forma cíclica por los discos que conforman la partición lógica, repartiendo la carga de forma equitativa. Para optimi zar la eficiencia del sistema de archivos tWeikum, 19921 se puede definir una unidad de almacena miento en cada banda con un tamaño mayor que el del bloque del sistema de archivos. Esta unidad, denominada unidad de distribución (stripe unit) es la unidad de información que se escribe de forma consecutiva en cada banda, siendo 64 KB el valor por defecto en Windows NT. La Figura 8.18 muestra la estructura de un sistema de archivos con bandas distribuido por cuatro particiones.

Además, este tipo de sistemas de archivos permite incrementar la fiabilidad del sistema de archivos insertando bloques de paridad con información redundante. De esa forma si falla un dispo sitivo, se puede reconstruir la información mediante los bloques de los otros dispositivos y median te la información de paridad. Además, se puede hacer que la partición sea más tolerante a fallos distribuyendo también la información de la partición del sistema.

Log Structured File System (LFS)

La evolución de la tecnología de computadoras ha permitido incrementar el tamaño de la memoria de forma exponencial. Sin embargo, los tiempos de búsqueda de los dispositivos de entrada/salida siguen siendo muy lentos en comparación a la velocidad del procesador y de los accesos a memoria.

Digitalización realizada con propósito académico

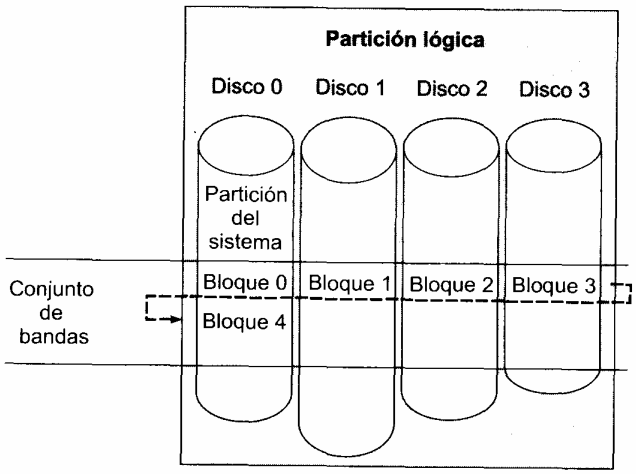


Figura 8.18. *Estructura de un sistema de archivos con bandas.*

Como resultado de esta evolución, los sistemas de entradalsalida disponen actualmente de grandes almacenes intermedios en memoria que permiten optimizar las operaciones de lectura de forma espectacular. Esta optimización no se puede aplicar en la misma medida a las operaciones de escritura, porque para mantener la fiabilidad del sistema es necesario escribir los datos a los discos, al menos, en períodos determinados. Las escrituras parciales son especialmente ineficientes, ya que es necesario leer el bloque afectado, modificarlo en memoria y escribirlo posteriormente a disco. La idea básica de LFS [Douglis, 1989] y [Ousterhout, 1989] es hacer todas las operaciones de escritura de forma secuencial en un **archivo de registro** intermedio del disco y, posteriormente, escribirlas en cualquier otro formato que sea necesario. Periódicamente, el sistema recolecta las escrituras pendientes en los almacenes de memoria, los agrupa en un segmento único de datos y los escribe a disco secuencialmente al final del registro.

Esta política permite explotar de forma óptima el ancho de banda del disco, pero presenta problemas de rendimiento graves cuando hay que buscar elementos dentro del registro de forma aleatoria. Por ejemplo, si se quiere leer el bloque 342 del nodo-i 45, es necesario buscar el nodo-i dentro del registro. Una vez obtenido, se calcula la posición del bloque de forma habitual. Como el sistema de archivos no tiene una estructura con grupos de datos situados en lugares fijos, es imposi ble calcular la posición del nodo-i. En el peor de los casos, se deberá recorrer el registro completo para encontrarlos. Para aliviar este problema, LFS mantiene en el disco, y en memoria, un mapa de nodos-i, ordenados por número de nodo-i, a bloques del registro. Estos detalles, y las operaciones de control de bloques libres, complican mucho las operaciones de mantenimiento del sistema de archivos [Blackwell 1995]. Es necesario mantener información de control de segmentos, un resu men de contenido por segmento y control de distintas versiones del mismo archivo.

8.6. EL SERVIDOR DE ARCHIVOS

Hasta el momento se han estudiado los archivos y directorios desde el punto de vista del usuario de un sistema operativo. Los usuarios necesitan saber cómo se manipulan los archivos, cómo se estruc turan en directorios, cuál es el método de nombrado usado por el sistema operativo, las operaciones

Digitalización realizada con propósito académico

permitidas sobre archivos y directorios, cómo se crea un sistema de archivos en un dispositivo de almacenamiento y con qué criterios deben ajustarse los parámetros del mismo, etc. Para proporcio nar un acceso eficiente y sencillo a los dispositivos de almacenamiento, todos los sistemas operati vos tienen un servidor de archivos que permite almacenar, buscar y leer datos fácilmente [1991]. Dicho servidor de archivos tiene dos tipos de problemas de diseño muy distintos entre sí:

- Definir la visión de usuario del sistema de entrada/salida, incluyendo servicios, archivos, directorios, sistemas de archivos, etc.
- Definir los algoritmos y estructuras de datos a utilizar para hacer corresponder la visión del usuario con el sistema físico de almacenamiento secundario.

El primer tipo de problemas se ha discutido ya en secciones anteriores. En esta sección se va a estudiar cuál es la estructura del servidor de archivos, cómo se implementan las entidades lógicas vistas hasta ahora, cómo se gestiona el espacio en los dispositivos, los elementos que se incluyen en el servidor de archivos para optimizar las operaciones de entrada/salida, etc.

8.6.1. Estructura del servidor de archivos

La importancia de una buena gestión de los archivos y directorios se ha subestimado a menudo. Desde el punto de vista de los usuarios, los directorios son una forma de organización de la infor mación y los archivos son flujos de datos de los que en general se requiere velocidad y aleatori edad en los accesos. Para satisfacer estas demandas, los servidores de archivos tienen una estructura intema que, en general, permite acceder a los distintos dispositivos del sistema mediante archivos de distintos tipos, escondiendo estos detalles a los usuarios.

Un servidor de archivos está compuesto por varias capas de software [Goodheart, 1994] y [Goscinski, 1991]. Cada capa usa las características de los niveles inferiores para crear un nivel más abstracto, hasta llegar a los servicios que se proporcionan a los usuarios. La Figura 8.19 muestra la arquitectura de un servidor de archivos como el del sistema operativo LINUX.

El **sistema de archivos virtual** es el encargado de proporcionar la interfaz de llamadas de entrada/salida del sistema y de pasar al módulo de organización de archivos la información necesa ria para ejecutar los servicios pedidos por los usuarios. Dentro de este nivel se suele incluir: manejo de directorios, gestión de nombres, algunos servicios de seguridad, integración dentro del servidor de archivos de distintos tipos de sistemas de archivos y servicios genéricos de archivos y directo rios. Para ello, en casi todos los sistemas operativos se usa una estructura de información que incluye las características mínimas comunes a todos los sistemas de archivos subyacentes y que enlaza con un descriptor de archivo de cada tipo particular. Por ejemplo, en UNIX esta estructura se denomina **nodo-v** (por nodo virtual). El nodo virtual es un objeto que contiene información genéri ca útil, independientemente del tipo de sistema de archivos particular al que representa el objeto. Esta información incluye:

- Atributos, tales como estado, información de protección, contadores de referencia, informa ción acerca del tipo de sistema de archivos subyacente al que en realidad pertenece el objeto, etcétera.
- Un apuntador al nodo-i real del objeto (existente en su sistema de archivos específico).
- Un apuntador a las funciones que realmente ejecutan los servicios específicos de cada siste ma de archivos.

Digitalización realizada con propósito académico

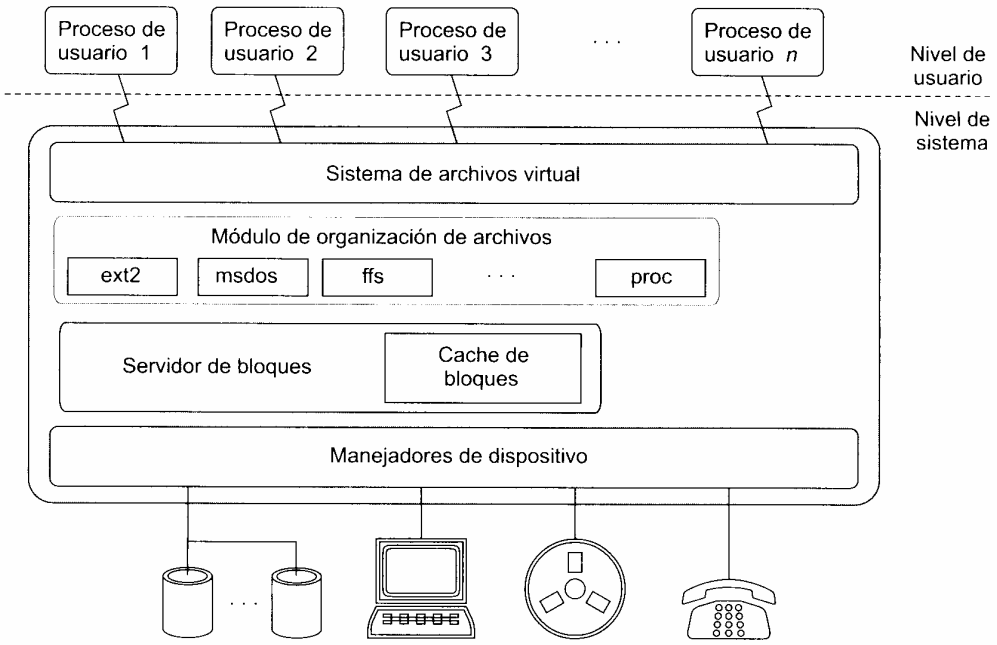


Figura 8.19. *Estructura de un servidor de archivos.*

La Figura 8.20 muestra la estructura de la información dentro de un nodo-y. Dentro del siste ma de archivos virtuales se incluyen operaciones que son independientes del tipo de sistema de archivos, tales como el mantenimiento de una cache de nombres, gestión de nodos virtuales, gestión de bloques en memoria, etc. Cuando las operaciones son específicas del tipo de sistema de archivos subyacente, el sistema de archivos virtual se limita a traducir los parámetros necesarios y a llamar a la operación adecuada del tipo de archivos afectado, cuyo servicio es provisto por el módulo de organización de archivos.

El **módulo de organización de archivos** proporciona el modelo del archivo del sistema opera tivo y los servicios de archivos. Es en este nivel donde se relaciona la imagen lógica del archivo con su imagen física, proporcionando algoritmos para trasladar direcciones lógicas de bloques a sus correspondientes direcciones físicas. Además, en este nivel se gestiona el espacio de los sistemas de archivos, la asignación de bloques a archivos y el manejo de los descriptores de archivo (nodos-i de UNIX o registros de Windows NT). Puesto que un mismo sistema operativo puede dar servicio para varios tipos de archivos, existirá un módulo de este estilo por cada tipo de archivo soportado (UNIX, AFS. Windows NT, MS-DOS, EFS, MINIX, etc.). Dentro de este nivel también se propor cionan servicios para pseudoarchi vos, tales como los del sistema de archivos proc.

Las llamadas de gestión de archivos y de directorios particulares de cada sistema de archivos se resuelven en el módulo de organización de archivos. Para ello se usa la información existente en el nodo-i del archivo afectado por las operaciones. Para realizar la entrada/salida de datos a disposi tivos de distintos tipos, este nivel se apoya en un servidor de bloques, que proporciona entra da/salida independiente de los dispositivos a nivel de bloques lógicos.

El **servidor de bloques** se encarga de emitir los mandatos genéricos para leer y escribir blo ques a los manejadores de dispositivo. La FIS de bloques de archivo, y sus posibles optimizacio -

Digitalización realizada con propósito académico

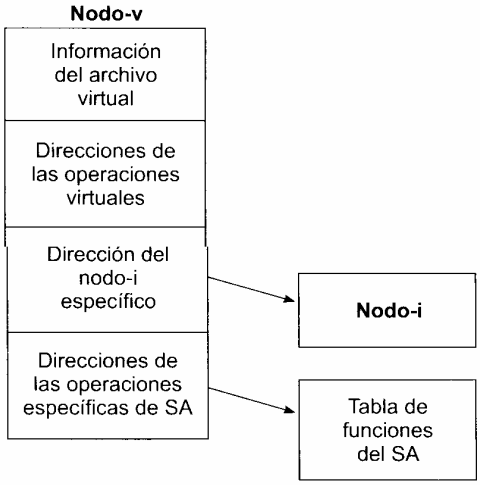


Figura 8.20. *Organización de un nodo virtual (***nodo-v***).*

nes, se lleva a cabo en este nivel del servidor de archivos. Las operaciones se traducen a llamadas de los manejadores de cada tipo de dispositivo específico y se pasan al nivel inferior del sistema de archivos. Esta capa oculta los distintos tipos de dispositivos, usando nombres lógicos para los mismos. Por ejemplo, /dev/hda3 será un dispositivo de tipo hard disk (hd), cuyo nombre princi pal es a y en el cual se trabaja sobre su partición 3. Los mecanismos de optimización de la FIS. como la cache de bloques, se incluye en este nivel.

El nivel inferior incluye los **manejadores de dispositivo**. Existe un manejador por cada dispo sitivo, o clase de dispositivo, del sistema. Su función principal es recibir órdenes de EIS de alto nivel, tal como move_to_block 234, y traducirlas al formato que entiende el controlador del dispositivo, que es dependiente de su hardware. Habitualmente, cada dispositivo tiene una cola de peticiones pendientes, de forma que un manejador puede atender simultáneamente a varios disposi tivos del mismo tipo. Por tanto, una de las principales funciones de los manejadores de dispositivos es recibir las peticiones de entradalsalida y colocarlas en el lugar adecuado de la cola de peticiones del dispositivo afectado. La política de inserción en cada cola puede ser diferente, dependiendo del tipo de dispositivo o de la prioridad de los dispositivos. Para un disco, por ejemplo, se suele usar la política CSCAN.

La Figura 8.21 muestra el **flujo de datos en el servidor de archivos** debido a una llamada read, así omnb1snperaLzinnasna ar1aspara diíibn fl El usuario ve tun vPctnr de hyte yue san

colectivamente descritos mediante un descriptor de archivo, un apuntador de posición en el archivo y el número de bytes a leer. Los datos a leer pueden no estar alineados con el principio de un bloque lógico y el número de bytes pedidos puede no ser múltiplo del tamaño de bloque lógico. Cuando el sistema de archivos virtual recibe la petición de lectura, comprueba el tipo de sistema de archivos al que pertenece el archivo (p. ej.: FES) y llama a la rutina de lectura particular del mismo. El módulo servidor de FFS comprueba 'la dirección de'l apuntador y e'l tamaño de buffer y ca'icu 'ius rnoques a leer. Como puede verse en la figura, es habitual que sólo se necesite un fragmento del bloque inicial y final de la petición. Sin embargo, puesto que el sistema de archivos trabaja con bloques, es necesario leer ambos bloques completos. Una vez calculados los bloques, se envía la petición al servidor de bloques. A continuación se solicitan los bloques al manejador de disco. Una vez terminada la opera ción de lectura, los bloques se copian al espacio del usuario.

Digitalización realizada con propósito académico

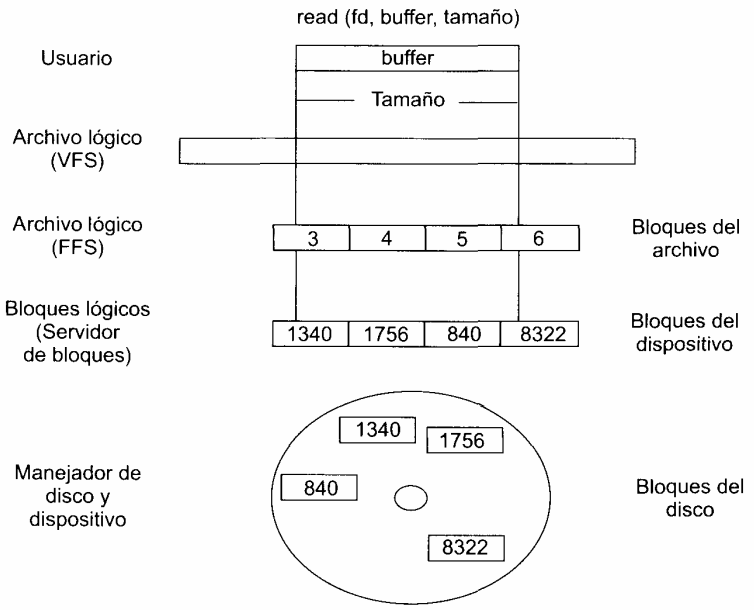


Figura 8.21. *Flujo de datos en el servidor de archivos.*

8.6.2. Estructuras de datos asociadas con la gestión de archivos

Para crear un nuevo archivo, las aplicaciones llaman al sistema de archivos virtual mediante la llamada al sistema crea Con esta llamada, el sistema crea un descriptor virtual para el nuevo archivo, en el que incluye la información de protección del mismo. El módulo de organización de archivos del tipo requerido, a petición del sistema de archivos virtual, se encarga de crear un des criptor de archivo de ese tipo, incluyendo en él la información recibida del nivel superior. A conti nuación modifica la estructura de los directorios. Para ello lee la información del directorio donde se quiere crear el archivo y la trae a memoria. Para ello llama al módulo de organización de archi vos correspondiente al directorio (los directorios no son sino archivos especiales), que a su vez se encarga de llamar al gestor de bloques para leer los bloques del directorio a memoria. Si todo es correcto, la actualiza con la entrada del nuevo directorio y la escribe inmediatamente al disco. La escritura del directorio hace un recorrido similar al de la lectura, pero los bloques van de memoria al disco.

En este momento ya existe el nuevo objeto archivo. En el caso de UNIX, el archivo está vacío y, por tanto, no tiene ningún bloque asociado. En el caso de Windows NT, el archivo ya dispone de la parte de datos de su registro para almacenar información válida. El descriptor asignado al archivo es el primero que se encuentra libre en el mapa de bits, o lista de descriptores libres, del sistema de archivos al que pertenece el archivo creado. Para trabajar con dicho archivo es necesario empezar una sesión abriendo el archivo (open). La implementación de las llamada open en un sistema operativo multiproceso, como UNIX, en el que varios procesos pueden abrir y cerrar el mismo archivo simultáneamente, es más complicada que en sistemas operativos monoproceso como MS-DOS. Si los procesos pueden compartir un mismo archivo, ¿qué valor tendrá en cada momento

Digitalización realizada con propósito académico

el apuntador de posición del archivo? Si los procesos son independientes, cada uno debe tener su propio apuntador de posición. ¿Dónde se almacena dicho valor? La solución más obvia es ponerlo en el nodo-y que representa al archivo, dentro de la **tabla de nodos-v** [Goodheart, 1994]. Dicha tabla almacena en memoria los nodos-v de los archivos abiertos. En ella se almacena la información del nodo-y existente en el disco y otra que se usa dinámicamente y que sólo tiene sentido cuando el archivo está abierto. El problema es que, si sólo hay un campo de apuntador, cada operación de un proceso afectaría a todos los demás. Una solución a este problema podría ser la inclusión de un apuntador por proceso con el archivo abierto. Pero entonces, ¿cuál sería el tamaño del nodo-y? Y ¿cuál debería ser el número de apuntadores disponibles? Se puede concluir que esa información no se puede incluir en el nodo-y sin crear problemas de diseño e implementación importantes en el sistema operativo. Parece pues necesario desacoplar a los procesos que temporalmente usan el objeto de la representación del objeto en sí.

Una posible solución para desacoplar a los procesos que usan el archivo de la representación interna del mismo podría ser incluir la información relativa al archivo dentro del **bloque de descripción del proceso** (BCP). En este caso, dentro del BCP de un proceso se incluiría una **tabla de archivos abiertos** (tdaa) con sus descriptores temporales y el valor del apuntador de posición del archivo para ese proceso. El tamaño de esta tabla define el máximo número de archivos que cada proceso puede tener abierto de forma simultánea. La Figura 8.22 muestra las tablas de descriptores de archivos abiertos por varios procesos en el sistema operativo UNIX. El descriptor de archivo fd indica el lugar de tabla. La tdaa se rellena de forma ordenada, de forma que siempre se ocupa la primera posición libre de la tabla. Cuando se realiza una operación open, el sistema de archivos busca desde la posición o hasta que encuentra una posición libre, siendo ésa la ocupada. Cuando se cierra un archivo (close), se marca como nula la correspondiente posición de la tdaa. En los sistemas UNIX cada proceso tiene tres descriptores de archivos abiertos por defecto. Estos descrip tores ocupan las posiciones O a 2 y reciben los siguientes nombres:

• Entrada estándar, fd = O.

• Salida estándar, fd = 1.

• Error estándar, fd = 2.

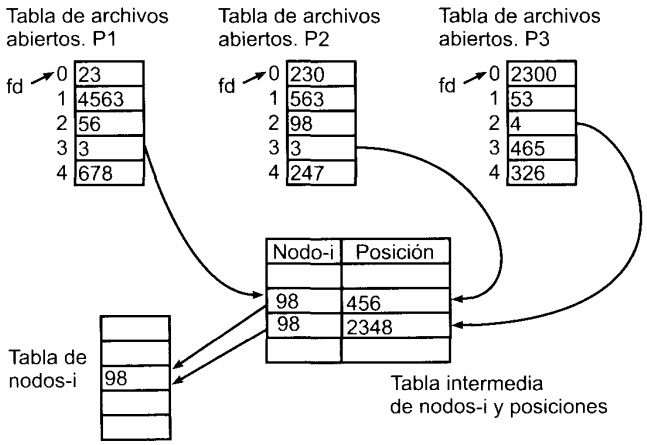


Figura 8.22. *Tablas relacionadas con la gestión de archivos en el sistema operativo UNIX.*

Digitalización realizada con propósito académico

El objetivo de estos descriptores estándar es poder escribir programas que sean independientes de los archivos sobre los que han de trabajar. En efecto, uno de los elementos del BCP que se conserva cuando se cambia el programa de un proceso (con el servicio POSIX exec) es la tabla de descriptores de archivo. Por tanto, basta con que un proceso coloque adecuadamente los descripto res estándar y que luego invoque la ejecución del mencionado programa, para que éste utilice los archivos previamente seleccionados.

Con la daa, el nodo-i queda libre de los problemas anteriores, pero surgen problemas cuando dos o más procesos comparten el mismo archivo y su apuntador de posición. Esta situación se da cuando se ejecuta una llamada fork con semántica POSIX, llamada de la que nace un nuevo proceso hijo que comparte con el padre su tdaa, lo que debe incluir el apuntador de posición de los archivos. Ahora bien, al duplicar el BCP si el hijo y el padre hacen operaciones de EIS distintas, los apuntadores difieren porque cada uno tiene su propio apuntador. Para solucionar este problema, algunos sistemas operativos, como UNIX, introducen una tabla intermedia entre la tabla de archi vos del BCP y la tabla de nodos-i (Fig. 8.22). Dicha tabla incluye, entre otras cosas:

- La entrada del nodo-i del archivo abierto en la tabla de nodos-i.
- El apuntador de posición correspondiente al proceso, o procesos. que usan el archivo durante esa sesión.
- El modo de apertura del archivo.

La introducción de esta tabla permite resolver los problemas anteriores, ya que cada proceso puede tener sus propios descriptores de archivo, evitando tener dicha información en el nodo-i, y permite que varios procesos puedan compartir no sólo el mismo archivo, sino también el mismo apuntador de posición dentro de un archivo.

Un problema similar al anterior surge cuando se implementan otras llamadas. Por ejemplo, un proceso puede bloquear todo o parte de un archivo, siendo esa información propia del proceso pero a su vez compartida con sus posibles procesos hijos. Un caso similar ocurre si se permite a los procesos proyectar archivos en memoria. Normalmente, todos estos casos se resuelven introdu ciendo nuevas tablas intermedias que desacoplen los procesos de la descripción del archivo en sí (nodo-i), al estilo de las tablas anteriores.

Cuando la aplicación quiere añadir información al archivo, el servidor de archivos debe deci dir dos cosas:

1. Cómo hacer corresponder los bloques de disco con la imagen del archivo que tiene la aplicación.

2. Cómo asignar bloques de disco libres para almacenar la información del archivo.

A continuación, se presentan las soluciones más frecuentes a estos dos problemas.

8.6.3. Mecanismos de asignación y correspondencia de bloques a archivos

Una de las cuestiones más importantes del diseño y la implementación de un servidor de archivos es cómo asignar los bloques de disco a un archivo y cómo hacerlos corresponder con la imagen del archivo que tiene la aplicación. Este problema se resuelve con lo que tradicionalmente se conoce como mecanismos de asignación [Koch, 1987]. En distintos sistemas operativos se han propuesto varias políticas, existiendo dos variantes claras:

Digitalización realizada con propósito académico

• Asignación de **bloques contiguos**. Con este método, un archivo de 64 KB ocuparía 16 bloques consecutivos en un sistema de archivos que use un tamaño de bloque de 4 KB. Es muy sencillo de implementar y el rendimiento de la E/S es muy bueno, pero si no se conoce

el tamaño total del archivo cuando se crea, puede ser necesario buscar un nuevo hueco de bloques consecutivos cada vez que el archivo crece [19891. Además, la necesidad de buscar huecos contiguos origina una gran fragmentación externa en el disco, ya que hay muchos huecos no utilizables debido a la política de asignación. Sería pues necesario com pactar el disco muy frecuentemente. Debido a estas desventajas, ningún sistema operativo moderno usa este método, a pesar de que la representación interna del archivo es muy senci lla: basta con conocer el primer bloque del archivo y su longitud.

• Asignación de **bloques no contiguos**. Con este método se asigna al archivo el primer bloque que se encuentra libre. De esta forma se elimina el problema de la fragmentación externa del disco y el de la búsqueda de huecos. Además, los archivos pueden crecer mientras exista espacio en el disco. Sin embargo, este método complica la implementación de la imagen de archivo que usa el servidor de archivos. La razón es que se pasa de un conjunto de bloques contiguos a un conjunto de bloques dispersos, debiendo conocer dónde están dichos blo ques y en qué orden deben recorrerse.

Para resolver el problema de mantener el mapa de bloques discontiguos de un archivo, casi todos los sistemas operativos han propuesto usar una lista de bloques, una lista en combinación con un índice o un índice multinivel para optimizar el acceso. En una **lista enlazada** desde cada bloque de un archivo existe un apuntador al siguiente bloque del mismo. En el descriptor del archivo se indica únicamente el primer bloque del archivo. Este método tiene varias desventajas:

- No es adecuado para accesos aleatorios, ya que hay que leer todos los bloques para recorrer la cadena de enlaces hasta el destino.
- Puesto que el apuntador al bloque siguiente ocupa espacio (digamos 4 bytes) y están inclui dos en el archivo, el cálculo de la longitud real del archivo es más complicado.
- La inclusión de los apuntadores en los bloques de datos hace que el tamaño de éstos deje de ser múltiplo de 2 (cosa que ocurre habitualmente), complicando mucho el cálculo del núme ro de bloque en el que está un determinado byte del archivo.
- Es muy poco fiable, ya que la pérdida de un bloque del archivo supone la pérdida de todos los datos del archivo que van detrás de dicho bloque.

Las desventajas de la lista enlazada se pueden eliminar si se quitan los apuntadores de los bloques del archivo y se almacenan en un **índice enlazado** gestionado por el servidor de archivos. Cuando se crea un sistema de archivos, se almacena en una parte especial del mismo una tabla que contiene una entrada por cada bloque de disco y que está indexada por número de bloque. Usando esta tabla, cada vez que se crea un archivo se incluye en su descriptor el descriptor de la tabla que apunta al primer bloque del archivo. A medida que se asignan nuevos bloques al archivo se apunta a ellos desde la última entrada de la tabla asociada al archivo. Esta es la solución usada en la FAT (file allocation table) de los sistemas operativos MS-DOS y OS/2, que se muestra en la Fi 8.23. La gran desventaja de esta solución es que la FAT puede ocupar mucho espacio si el dispositivo es grande. Por ejemplo, un disco de 4 GB, con 4 KB como tamaño de bloque, necesitaría una FAT con 1 Mega entradas. Si cada entrada ocupa 4 bytes, la FAT ocuparía 4 MB. Para buscar un bloque de un archivo muy disperso podría ser necesario recorrer toda la FAT y, por tanto, tener que traer todos los bloques de la FAT a memoria. Imagine qué pasaría si la computadora tuviese ocho dispositivos como éste: se necesitarían 32 MB de memoria sólo para las FAT. Este método es pues inviable si la FAT no puede estar continuamente en memoria [19831, lo cual ocurre en cuanto los dispo sitivos alcanzan un tamaño medio. Para aminorar este problema, los bloques se juntan en agrupa ciones, lo que permite dividir el tamaño de la FAT (Prestaciones 8.5). Si se usaran grupos de 4 bloques, el tamaño de la FAT anterior se reduciría a 1 MB.

Digitalización realizada con propósito académico

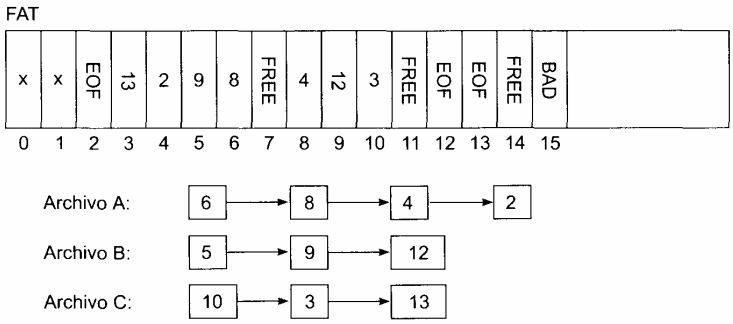


Figura 8.23. *Mapa de bloques en la FAT de MS-DOS.*

PRESTACIONES 8.5

La FAT de MS-DOS es muy lenta cuando se accede aleatoriamente a un archivo grande. La razón es que no se sabe dónde está un bloque de un archivo si no se sigue toda la cadena bloques del mismo. Por eso, si una aplicación salta de un lado a otro del archivo, puede ser necesario empezar cada vez desde el principio. Este comportamiento se debe a que el mecanismo está diseñado pensando en accesos secuenciales y en archivos pequeños.

Los problemas anteriores pueden resolver estos problemas de forma eficiente mediante el uso de un **índice multinivel** cuyos bloques son apuntados desde el nodo-i que describe al objeto archi vo, como los de UNIX y LINUX. Con esta solución, cada archivo tiene sus **bloques de índice** que incluyen apuntadores a los bloques de disco del archivo. El orden lógico se consigue mediante la inserción de los apuntadores en orden creciente, a partir del primero, en los bloques de índices. Por ejemplo, el byte 5672 de un archivo, almacenado en un sistema de archivos que usa bloques de 4 KB, se encontrará en el segundo apuntador del índice. La ventaja de usar índices es que basta con traer a memoria el bloque de índices donde está el apuntador a los datos para tener acceso al bloque de datos. Además, si un apuntador de bloque ocupa 4 bytes y el bloque es de 4 KB, con un único acceso a disco tendremos 1 .024 apuntadores a bloques del archivo. Existe sin embargo un proble ma: el espacio extra necesario para los bloques de índices. Imagine que un archivo tiene 1 byte. Necesita un bloque de índices asociado a su bloque de datos. Sin embargo, sólo usa una entrada del bloque de índices (4 bytes) y 1 byte del bloque de datos. Ese problema fue resuelto en UNIX BSD combinando un sistema de índices puros con un sistema de índices multinivel, que es el que se usa actualmente en UNIX y LINUX. La Figura 8.24 muestra un ejemplo del nodo-i de UNIX con sus bloques de índice. Como puede verse, cada descriptor de archivo (nodo-i) tiene varios apuntadores directos a bloques de datos y tres apuntadores a bloques de índices de primero, segundo y tercer nivel. Este método tiene dos ventajas:

• Permite almacenar archivos pequeños sin necesitar bloques de índices.

• Permite accesos aleatorios a archivos muy grandes con un máximo de tres accesos a bloques de índices.

Sin embargo, sigue teniendo una desventaja: hay que acceder al nodo-i para tener las direccio nes de los bloques de disco y luego leer los datos. Los sistemas operativos tipo UNIX tratan de paliar este problema manteniendo los nodos-i de los archivos abiertos en una tabla de memoria.

Digitalización realizada con propósito académico

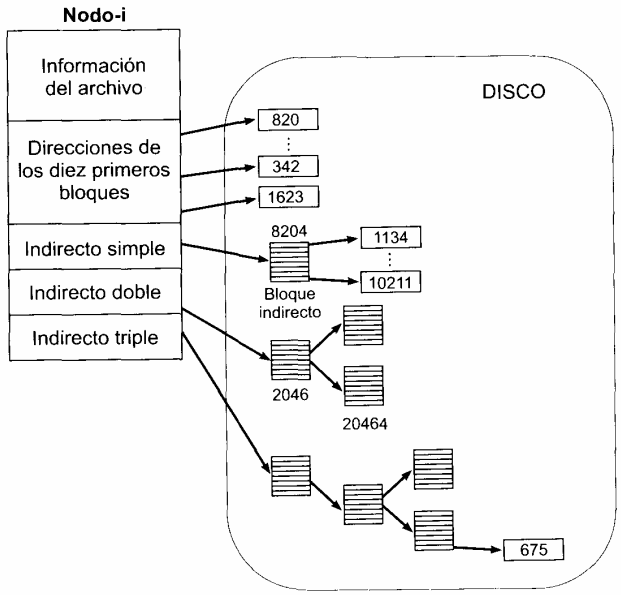


Figura 8.24. *Mapa de bloques en un nodo-i.*

Para evitar ese acceso extra, Windows NT almacena datos en el mismo descriptor de archivo. De esta forma, muchos archivos pequeños pueden leerse completamente con un único acceso a o e servidor de archivos mantiene una estructura jerárquica en forma de árbol, donde en cada bloque se almacenan datos y apuntadores aXosb\oques s S\ el archivo es muy grande, algunos de estos bloques pueden a su vez estar llenos de apuntadores a nuevos bloques. Este esquema permite un acceso muy rápido a los datos, ya que cada acceso trae datos, pero complica la implementación del modelo de archivos en el servidor de archivos porque el primer bloque necesita un cálculo de dirección especial [Nagar, 1997].

8.6.4. Mecanismos de gestión de espacio libre

El espacio de los dispositivos debe ser asignado a los objetos de nueva creación y a los ya existentes que requieran más espacio de almacenamiento. Es pues necesario conocer el estado de ocupación de los bloques de los dispositivos para poder realizar las operaciones de asignación de forma efi ciente Oldehoeft, 19851. Por ello, todos los sistemas de archivos mantienen mapas de recursos, habitualmente construidos como mapas de bits o listas de recursos libres.

Los **mapas de bits**, o vectores de bits, incluyen un bit por recurso existente (descriptor de archivo, bloque o agrupación). Si el recurso está libre, el valor del bit asociado al mismo es 1, si está ocupado es O. Por ejemplo, sea un disco en el que los bloques 2, 3, 4, 8, 9 y 10 están ocupados y el resto libres, y en el que los descriptores de archivo 2, 3 y 4 están ocupados. Sus mapas de bits serían:

MB de bloques: 1100011100011....

Digitalización realizada con propósito académico

MB de descriptores: 1100011...

La principal ventaja de este método es que es fácil de implementar y sencillo de usar. Además es muy eficiente si el dispositivo no está muy lleno o muy fragmentado, ya que se encuentran zonas contiguas muy fácilmente. Sin embargo, tiene dos inconvenientes importantes: es difícil e inefi ciente buscar espacio si el dispositivo está fragmentado y el mapa de bits ocupa mucho espacio si el dispositivo es muy grande. Por ejemplo, un sistema de archivos de 4 GB, con un tamaño de bloque de 4 KB, necesita 1 MB para el mapa de bits de bloques. Para que la búsqueda sea eficiente, los mapas de bits han de estar en memoria, ya que sería necesario recorrer casi todo el mapa de bits para encontrar hueco para un archivo. Imagine que un sistema tuviera montados ocho dispositivos como el anterior, necesitaría 1 MB de memoria sólo para los mapas de bits. Si se aplicara una agrupación de 4 bloques, el tamaño final del mapa de cada dispositivo sería de 256 KB (32 KB), lo que equivale a 256 KB para los ocho dispositivos. Estas cantidades, con ser grandes, son mucho menores que los 32 MB y 8 MB que serían respectivamente necesarios con la FAT de MS-DOS.

Las **listas de recursos libres** permiten resolver el problema de forma completamente distinta. La idea es mantener enlazados en una lista todos los recursos disponibles (bloques o descriptores de archivos) manteniendo un apuntador al primer elemento de la lista. Este apuntador se mantiene siempre en memoria. Cada elemento de la lista apunta al siguiente recurso libre de ese tipo. Cuando el servidor de archivos necesita recursos libres, recorre la lista correspondiente y desenlaza elemen tos, que ya no estarán libres. Como el lector puede comprender, este método no es eficiente, excep to para dispositivos muy llenos y fragmentados, donde las listas de bloques libres son muy peque ñas. En cualquier otro caso, recorrer las listas requiere mucha entrada/salida. La FAT del sistema operativo MS-DOS es una lista enlazada. La Figura 8.25 muestra un dispositivo cuyos bloques libres se mantienen en una lista con una entrada por bloque libre (caso A). Este método se puede optimizar incluyendo dentro de la lista la dirección de un bloque libre y el número de bloques consecutivos al mismo que también están libres. El caso B de la Figura 8.25 muestra la lista de bloques libres del mismo dispositivo anterior optimizada de la forma descrita.

Una posible optimización de la gestión de bloques libres es reunir los bloques en **agrupaciones** y mantener mapas de bits o listas de recursos libres para ellas. Por ejemplo, si en el dispositivo anterior se usan agrupaciones de 64 KB (16 bloques), el mapa de bits se reduce a 8 KB. Sin embargo, esto complica la política de gestión de espacio libre en el servidor de archivos [Akyurek, 1995] y [Davy, 1995]. Los sistemas operativos UNIX, LINUX y Windows NT usan este método. Igualmente, se puede usar la agrupación de bloques en las listas de recursos libres. Obviamente, cuando se usan agrupaciones, la mínima unidad que se puede asignar es una agrupación, por lo que es muy importante decidir cuál va a ser el tamaño de la misma. Cuanto mayor sea, más grande puede ser la fragmentación interna del dispositivo. Para paliar en parte este problema, los sistemas que usan agrupaciones suelen usar también el mecanismo de gestión de fragmentos descrito en la sección del Fast File Svstem (FFS).

Digitalización realizada con propósito académico

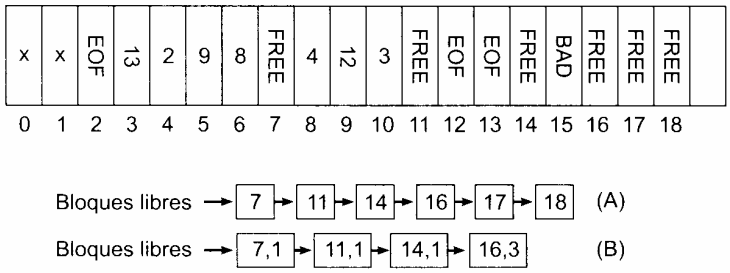


Figura 8.25. *Gestión de espacio con listas de bloques libres.*

8.6.5. Mecanismos de incremento de prestaciones

Los dispositivos de entradalsalida son habitualmente más lentos que la UCP y los CCCSOS a memo ria. Cada acceso a un dato en memoria cuesta pocos nanosegundos, mientras que acceder a un bloque de disco cuesta varios milisegundos. La mayoría de los servidores de archivos tratan de reducir esta diferencia, de seis órdenes de magnitud, optimizando los mecanismos de que disponen para acceder a los datos. Prácticamente todos los parámetros del servidor de archivos son suscepti bles de optimización. Ya se ha visto que la asignación de bloques libres se puede optimizar usando grupos de bloques, que las estructuras de datos en uso (como nodos-i o superbloques) se guardan en tablas de memoria y que la correspondencia de los bloques lógicos con los físicos se puede optimi zar mediante el uso de índices de múltiple nivel. También se ha visto que existen diferentes estruc turas de sistema de archivos que permiten optimizar distintos tipos de accesos.

Además de los métodos anteriores, los sistemas operativos incluyen mecanismos de incremen to del rendimiento de la entradalsalida basados en el uso de **almacenamiento intermedio** de datos de entradalsalida en memoria principal. Estos mecanismos son de dos tipos:

- **Discos RAM**, cuyos datos están almacenados sólo en memoria LTanenbaum, 19971. Estos discos aceptan todas las operaciones de cualquier otro sistema de archivos y son gestionados por el usuario. Excepto por su rapidez, el usuario no percibe ninguna diferencia cori cual quier otro tipo de disco. Estos pseudodispositivos se usan habitualmente para almacena miento temporal o para operaciones auxiliares del sistema operativo. Su contenido es volátil, ya que se pierde cuando se apaga el sistema.
- **Cache de datos**, instaladas en secciones de memoria principal controladas por el sistema operativo, donde se almacenan datos para optimizar accesos posteriores. Es importante ob servar que este mecanismo se basa en la existencia de **proximidad espacial y temporal** en las referencias a los datos de entradalsalida [1985]. Al igual que ocurre con las ins trucciones que ejecuta un programa, el sistema operativo asume que los datos de uii archivo que han sido usados recientemente serán reutilizados en un futuro próximo. 1-lay dos caches importantes dentro del servidor de archivos: **cache de nombres y cache de bloques**.

Windows NT incluye otro mecanismo de incremento de prestaciones para incrementar la ca pacidad de los dispositivos de almacenamiento. Se trata de la **compresión de datos** de los archivos y directorios de usuario.

Digitalización realizada con propósito académico

Cache de nombres

Los sistemas operativos modernos incluyen una cache de nombres en memoria principal para acele rarla interpretación de los nombres de archivo [Goodheart, 1994]. Esta cache almacena las entradas de directorio de aquellos archivos que han sido abiertos recientemente. En UNIX. cada elemento de la cache es una pareja (nombre, nodo- i). Cuando se abre un archivo, las funciones que interpre tan el nombre miran en la cache para ver si los componentes del mismo se encuentran presentes. En caso positivo, no acceden a los bloques de directorio, obteniendo el nodo-i del archivo. En caso negativo, acceden a los bloques de directorio y. a medida que interpretan el nombre, incluyen los nuevos componentes en la cache.

El mantenimiento de una cache de nombres puede complicarse debido a problemas de seguri dad. Imagine qué ocurriría si se borra un archivo y se reutiliza su nombre con otro nodo-i. Los accesos a la cache conducirían a un archivo erróneo. En el mejor de los casos, el nodo-i estaría sin usar y se obtendría un error. En el peor de los casos, se accedería a datos de otro archivo. Para evitar este problema, casi todas las operaciones de creación, destrucción y cambio de características de los

archivos actualizan la cache de nombres con el fin de mantenerla **coherente** con la situación real del sistema (Prestaciones 8.6).

PRESTACIONES 8.6

Actualmente, las caches de nombres no sólo almacenan los nombres de los archivos que existen, sino los de los archivos que no existen pero a cuyo nombre se ha intentado acceder en el sistema. Esta característica, denominada *cache negativa*, es muy importante para optimizar los accesos erróneos, ya que éstos suelen requerir búsquedas casi completas de cada directorio accedido para ver si el archivo existe.

Cache de bloques

La técnica más habitual para reducir el acceso a los dispositivos es mantener una colección de bloques leídos o escritos, denominada cache de bloques, en memoria principal durante un cierto período de tiempo [Braunstein, 19891 y [Smith, 19921. Aunque los bloques estén en memoria, el servidor de archivos considera su imagen de memoria válida a todos los efectos, por lo que es necesario mantener la **coherencia** entre dicha imagen y la del disco. Como se vio anteriormente, el flujo de datos de entradalsalida de los dispositivos de bloque pasa siempre a través de la cache de bloques. Cuando se lee un archivo, el servidor busca primero en la cache. Sólo aquellos bloques no encontrados se leen del dispositivo correspondiente. Cuando se escribe un archivo, los datos se escriben primero en la cache y posteriormente, si es necesario, son escritos al disco. La Figura 8.26

Digitalización realizada con propósito académico

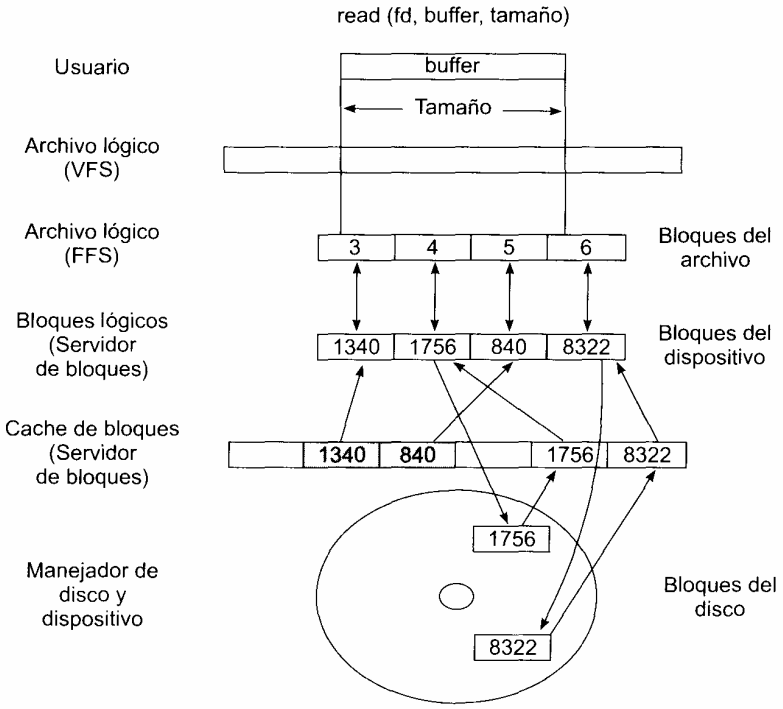


Figura 8.26. *Flujo de datos optimizado con cache de bloques.*

muestra un esquema del flujo de datos en el servidor de archivos cuando hay una cache de bloques. Como puede verse, el número de accesos al dispositivo se reduce drásticamente si hay aciertos en la cache. La tasa de aciertos depende mucho del comportamiento de las aplicaciones que se ejecutan sobre el sistema operativo. Si éstas no respetan la proximidad espacial o temporal de las referencias en sus patrones de acceso, o si ejecutan peticiones de entradalsalida tan grandes que sustituyen todos los bloques de la cache constantemente, la tasa de aciertos es tan baja que en muchas aplica ciones conviene desactivar la cache de bloques. Esta opción está disponible en los sistemas operati vos de propósito general más populares, como UNIX o Windows NT. En el caso de UNIX, si se akwe ‘ d s de. baques como si fuera de caracteres, los accesos a este dispositivo no pasan por la cache. En Windows NT se puede especificar si se desea usar la cache o no en la llamada de creación y apertura del archivo.

El formato de la cache de bloques es complejo [1996] y [1994]. La mayoría de los servidores de archivos usan listas doblemente encadenadas con hashing para optimizar la búsqueda y la inserción de bloques. Estas listas conectan estructuras de datos que contienen, entre otras cosas:

- La identificación del bloque.
- Los atributos del bloque (datos, especial, etc.).
- El tipo de sistema de archivos al que pertenece el bloque.
- El tamaño del bloque referenciado.
- Un apuntador a los datos del bloque.

Esta última característica es fundamental para permitir que la cache pueda manejar bloques de datos de distintos tamaños. El tamaño de la cache puede ser fijo o variable. En las versiones anti guas de UNIX, la cache era una estructura definida como un vector de bloques. Sus dimensiones eran, por tanto, fijadas en la instalación del sistema. En los sistemas operativos modernos, la cache es una estructura dinámica y su tamaño es mayor o menor dependiendo de la memoria que deja libre el resto del sistema operativo.

Digitalización realizada con propósito académico

La gestión de la cache se puede llevar a cabo usando varios algoritmos, pero lo habitual es comprobar si el bloque a leer está en la cache. En caso de que no esté, se lee del dispositivo y se copia a la cache. Si la cache está llena, es necesario hacer hueco para el nuevo bloque reemplazando uno de los existentes. Existen múltiples **políticas de reemplazo** que se pueden usar en una cache de bloques, tales como el FIFO (First in First Out), segunda oportunidad, MRU (Most Recently Used), LRU (Least Recently Used), etc. La política de reemplazo más frecuentemente usada es la **LRU**. Esta política reemplaza el bloque que lleva más tiempo sin ser usado, asumiendo que no será referenciado próximamente. Los bloques más usados tienden a estar siempre en la cache y, por tanto, no van al 1 jj d pialítica pii crear problemas de fiabilidad en el sistema de archivos si la computadora falla. Imagine que se creó un directorio con tres archivos que están siendo referenciados constantemente El bloque del directorio estará siempre en la cache. Si la computado ra falla, esa entrada de directorio se habrá perdido. Imagine que el bloque se ha escrito, pero su imagen en la cache ha cambiado antes del fallo. El sistema de archivos queda inconsistente. Por ello, la mayoría de los servidores de archivos distinguen entre **bloques especiales y bloques de datos**. Los bloques especiales, tales como directorios, nodos-i o bloques de apuntadores, contienen información crucial para la coherencia del sistema de archivos, por lo que es conveniente salva guardar la información en disco tan pronto como sea posible. Sin embargo, con un criterio LRU puro, los bloques de este tipo que son muy usados irían a disco con muy poca frecuencia.

Para tratar de solventar el problema anterior, el sistema de archivos puede proporcionar distin tas políticas de escritura a disco de los bloques de la cache que tienen información actualizada (se dice que están sucios). En el caso de MS-DOS, los bloques se escriben a disco cada vez que se modifica su imagen en la cache de bloques. Esta política de escritura se denomina de **escritura**

inmediata (write-through). Con esta política no hay problema de fiabilidad, pero se reduce ci rendimiento del sistema porque si un bloque se puede volcar varias veces a disco. Imagine que un usuario escribe un bloque byte a byte. Con política de escritura inmediata se escribiría el bloque a disco cada vez que se modifique un byte. En el lado opuesto está la política de **escritura diferida** (write-back), en la que sólo se escriben los datos a disco cuando se eligen para su reemplazo por falta de espacio en la cache. Esta política optimiza el rendimiento, pero genera los problemas de fiabilidad anteriormente descritos. Si los bloques de datos se gestionan con política LRU pura, también se origina un problema de fiabilidad. Imagine que se crea un archivo pequeño cuyos blo ques se usan de forma habitual. Sus datos estarán siempre en la cache. Si la computadora falla, los datos del archivo se habrán perdido.

Para tratar de establecer un compromiso entre rendimiento y fiabilidad, algunos sistemas ope rativos, como UNIX, usan una política de **escritura retrasada** (delaved-write), que consiste en escribir a disco los bloques de datos modificados en la cache de forma periódica cada cierto tiempo (30 segundos en UNIX). De esta forma se reduce la extensión de los posibles daños por pérdida de datos. En el caso de UNIX, estas operaciones son ejecutadas por un proceso de usuario denominado sync. Además, para incrementar todavía más la fiabilidad, los bloques especiales se escriben inme diatamente al disco. Una consecuencia de la política de escritura retrasada es que. a diferencia de MS-DOS. en UNIX no se puede quitar un disco del sistema sin antes volcar los datos de la cache. En caso contrario se perderían datos y el sistema de archivos se quedaría inconsistente o corrupto [19931. Por último, en algunos sistemas operativos se aplica un nuevo criterio de escritura, denominado escritura al cierre (write-on-close). Con esta política, cuando se cierra un archivo, se vuelcan al disco los bloques del mismo que tienen datos actualizados. Esta política suele ser com plementaria de las otras y permite tener bloques de la cache listos para ser reemplazados sin tener que escribir al disco en el momento de ejecutar la política de reemplazo, con lo que se reduce el tiempo de respuesta de esta operación. En la mayoría de los casos, el sistema operativo ejecuta esta política en background, es decir, marca los bloques y los vuelca a disco cuando tiene tiempo para ello.

Compresión de datos

Digitalización realizada con propósito académico

Windows NT, en el sistema de archivos NTFS ISolomon, 19881, proporciona compresión de datos dentro del servidor de archivos de forma transparente al usuario. La compresión se puede llevar a cabo en el ámbito de archivos, de directorios o de volumen o dispositivo lógico. En este último caso no se aplica a los metadatos del sistema de archivos. La compresión permite reducir considerable mente el espacio ocupado por los datos, aun a costa de complicar la gestión en e! servidor de archivos.

La técnica de compresión usada consiste en eliminar conjuntos contiguos de ceros (caracteres nulos) de los archivos. Si el archivo se crea indicando que es comprimido, o el volumen donde se crea lo es, el servidor de archivos filtra los datos de los usuarios y almacena sólo los caracteres no nulos. Si se comprime a posteriori, las utilidades de compresión crean una nueva copia del archivo de la que se han eliminado esos caracteres. El rendimiento de esta técnica se mide mediante la **razón de compresión**, que se define de la siguiente manera:

$$\text{Razón de compresión} = \frac{\text{Tamaño inicial de archivo}}{\text{Tamaño final de archivo}}$$

Esta razón depende mucho del grado de **dispersión** de los datos a almacenar. Si los datos están muy dispersos, gran parte del archivo está lleno de ceros, la razón de compresión es alta y se reduce

considerablemente el tamaño del archivo. Si los datos no son dispersos, en cuyo caso se dice que son densos, la razón de compresión será peor.

El gran problema de comprimir un archivo es mantener mecanismos de asignación y de mapas de bloques que sean compatibles con los formatos no comprimidos. Observe que el usuario puede pensar que sus datos ocupan n KB y que un usuario experto puede incluso intentar optimizar la entrada/salida en base al tamaño de bloque. En Wiindows NT, si se compime un archivo, las peticiones que hace el usuario llegan a la cache de bloques con el mapa de bloques como si fuera descomprimido. Es a partir de este momento donde se incluye, y oculta, 1a compresión. Para ello, en el MFT del archivo se asignan bloques únicamente para los que tienen datos comprimidos. Los bloques que se han comprimido se marcan como vacíos y no se les asigna bloques. De esta forma. cuando se pide el bloque 36, por ejemplo, de un archivo comprimido se mira en el MFT si estaba lleno de ceros o no. Si lo estaba, se le devuelve al usuario un bloque lleno de ceros. En otro caso se accede al bloque comprimido donde se encuentra el bloque 36, se descomprime y se devuelve al usuario la parte que está pidiendo.

Windows NT proporciona la llamada al sistema GetVolumelnformatiOn para ver si un volumen está comprimido o no. Además, se puede ver el tamaño real de un archivo comprimido usando la llamada de Win32 GetCompressedFileSiZe (Advertencia 8.3).

ADVERTENCIA 8.3

¡Cuidado con los sistemas de archivos comprimidos! Cuando se comprime un volumen, por ejemplo `C:\`, se crea un dispositivo virtual, por ejemplo `G:\`, que contiene información relacionada con la compresión de `C:\`. Si ocurre algún error en el volumen `G:\`, la recuperación es muy difícil. Si `G:\` tiene errores, es prácticamente imposible descomprimir `C:\`.

8.6.6. Montado de sistemas de archivos e interpretación de nombres

Digitalización realizada con propósito académico

Una vez descrita la arquitectura del servidor de archivos y sus mecanismos de asignación de bloques y de incremento de prestaciones, se puede pasar a estudiar en detalle cómo se interpreta un nombre y cómo se resuelve dicho problema cuando para interpretar el nombre hay que pasar a un sistema de archivos montado. Para ello se usa como caso de estudio el sistema operativo

UNIX.

La Figura 8.27 muestra un ejemplo de montaje de un sistema de archivos en un directorio de otro sistema de archivos. Como se puede ver, en el montaje se usan dos estructuras de datos de forma primordial: la tabla de nodos-i y la tabla de superbloques del sistema. Esta última tabla contiene apuntadores a los superbloques de los sistemas de archivos montados y es fundamental para ocultar el punto de montaje de un dispositivo, como /dev/hda3, sobre un directorio, como /home/root/distr/lib

La implementación del montado de sistemas de archivos es sencilla cuando se dispone, como en UNIX, de las tablas de estructuras de datos adecuadas. Cuando se monta un sistema de archivos. se trae su superbloque a memoria y se colocan en él dos apuntadores: uno al nodo-i del directorio raíz del sistema de archivos montado y otro al nodo-i del directorio sobre el que está montado. Además se indica en el nodo-i del directorio sobre el que se monta que tiene un sistema de archivos colgando de él y se incluye la entrada de la tabla de superbloques en la que se almacenó el superblo que del sistema de archivos montado. En la operación de desmontado se desconectan dichos apun tadores [Goodheart, 1994] y [McKusick, 1996].

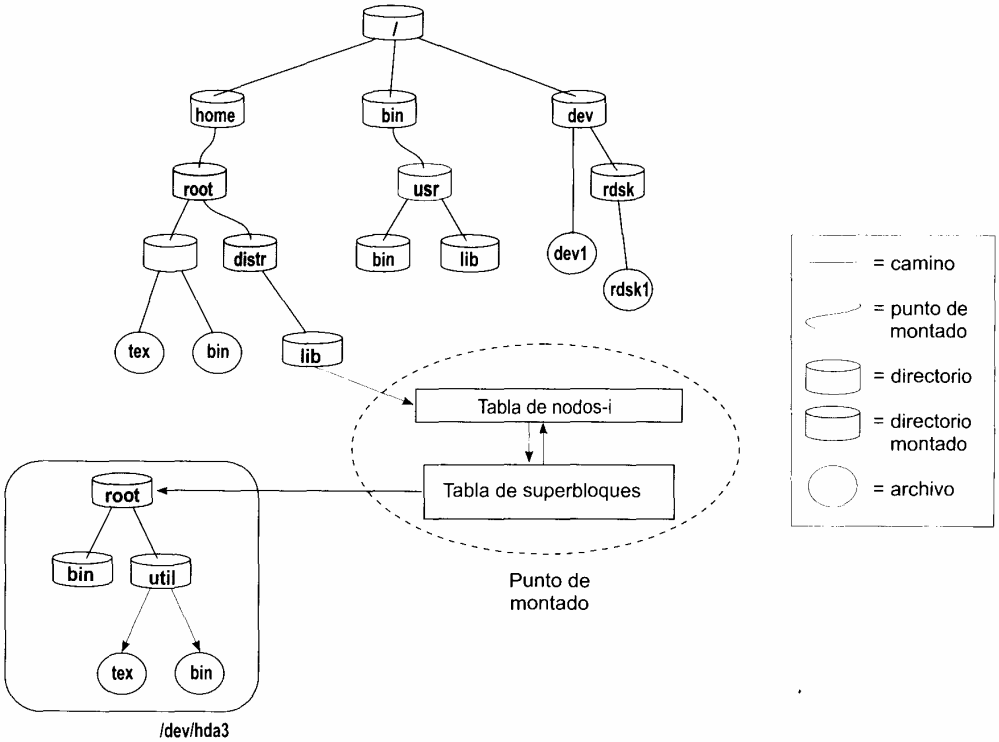


Figura 8.27. *Montado de un sistema de archivos.*

Para poder acceder a cualquier archivo por su nombre, las facilidades de interpretación de nombres en conocer los detalles anteriores. Así, cuando en el ejemplo de la figura se quiera acceder al archivo /home/ root /distr/ lib/util / tex, las funciones de interpretación del nom bre llegarán al directorio lib y verán en su nodo-i que tiene un sistema de archivos montado y la entrada donde está su superbloque en la tabla de superbloques. A continuación, acceden a dicha tabla y en el superbloque del sistema de archivos montado obtienen el apuntador al nodo-i de su raíz. Con este valor ya pueden acceder al directorio raíz y buscar la entrada del directorio util. En caso de que se usen nombres relativos que impliquen subir por el árbol de directorios, tal como . . /. . /distr a partir de util, la solución también funciona puesto que del nodo-i de la raíz del sistema de archivos montado se sube a la tabla de superbioques y desde aquí al nodo-i del directorio lib (Advertencia 8.4).

ADVERTENCIA 8.4

La existencia de enlaces *físicos* entre archivos plantea un problema grave para la decodificación de nombres. En UNIX y sus derivados sólo se pueden establecer enlaces entre archivos que pertenecen al mismo sistema de archivos. Si se quiere establecer un enlace entre dos archivos que pertenecen a distintos directorios, hay que usar un *enlace simbólico* (Sección 8.4).

El concepto de sistema de archivos montados, tal como se ha explicado aquí, no existe en Windows NT, por lo que la solución adoptada es muy distinta a la de UNIX. En este caso, en función del nombre de dispositivo lógico al que se quiere acceder, se busca en la tabla de superblo ques y se cambia directamente al sistema de archivos de este dispositivo. En un sistema de este estilo es típico tener múltiples **unidades lógicas** accesibles a nivel local o remoto. Una unidad logica esun dispositivo de almacenamiento sobre el que hay instalado un sistema de archivos, bien sea de Windows o de cualquier otro tipo. El usuario, o el programador de sistema, debe saber en qué unidad logica (C:\, D:\, E:\, J:\,...) está el archivo que busca, ya que pueden existir archivos con el mismo nombre que sólo se pueden discriminar añadiendo la unidad lógica al nombre. C: \pepe y D: \pe por ejemplo, çon irchivo digtinto En Windows NT, ia unidad es LuIIsustancIal al nombre absoluto de un archivo. Los nombres relativos lo son siempre dentro de una unidad lógica.

8.6.7. Fiabilidad y recuperación

La destrucción de un sistema de archivos es a menudo mucho peor que la destrucción de una computadora. La recuperación de los datos de un sistema de archivos muy dañado es una operación difícil, lenta y muchas veces imposible {Anyawun, 19861. En algunos casos, la destrucción de ciertos datos archivados, como la lista de clientes de una empresa o su contabilidad, pueden causar un daño irreparable a la organización afectada, lo que en el caso de las empresas puede llevar a la quiebra o el cese de actividades. ¿Imagina qué podría ocurrir si una universidad perdiera todos los expedientes de sus alumnos por un fallo del sistema de a'macenamiento? Podría costar años recupe rarlos del soporte en pape', suponiendo que éste exista. En esta sección se muestra cómo afrontan los servidores de archivos estos problemas, aunque muchas soluciones a los problemas de fiabilidad y recuperación son externas al sistema operativo y entran en los campos de administración del sistema, seguridad, etc.

Existen dos razones fundamentales por las que se pueden perder los datos de un sistema de archivos:

Digitalización realizada con propósito académico

• Destrucción física del medio de almacenamiento.

• Corrupción de los datos almacenados.

La destrucción física del medio de almacenamiento puede ser debida a defectos de fabrica ción, que originan la existencia de **bloques dañados** en el dispositivo, o debida a catástrofes o fallos hardware que dañan bloques con información almacenada. Actualmente, los propios controladores de disco son capaces de ocultar los sectores dañados, evitando así que sean usados. Además, si se detectan bloques dañados cuando se crea un sistema de archivos, se marcan como ocupados y se incluyen en un archivo oculto que contiene todos los bloques dañados. Este archivo es propiedad del sistema, está oculto y no se puede usar. La destrucción física de los datos debida a catástrofes o fallos del hardware no se puede solventar usando estas técnicas. Dos técnicas habituales para hacer frente a este tipo de fallos son las copias de respaldo y el almacenamiento de datos con redundancia.

Copias de respaldo

Si un dispositivo de almacenamiento falla, los datos se perderán irremisiblemente. Por ello es necesario hacer copias de respaldo (backups) de los sistemas de archivo con cierta frecuencia Frish, 19961. Las copias de respaldo consisten en copiar los datos de un dispositivo a otro de forma total o parcial. Las copias totales suelen requerir mucho tiempo y espacio extra, por lo que una

política muy popular es hacer copias de respaldo **incrementales**. Un respaldo incremental contiene únicamente aquellos archivos modificados desde la última vez que se hizo una copia de respaldo. Una de las principales tareas del administrador del sistema operativo es dictar la política de elabora ción de copias de respaldo en el sistema. Esta política depende de las necesidades de los usuarios del sistema, pero es muy habitual hacer copias totales cada semana o cada mes y hacer copias incrementales cada día.

Normalmente, los dispositivos sobre los que se hacen las copias de respaldo son cintas u otros medios de almacenamiento terciario. Estos dispositivos son más lentos, pero tienen más capacidad que los de almacenamiento secundario y, generalmente, se pueden extraer de la unidad de almace namiento. Si se quiere tener un sistema de copias de respaldo más rápido, se pueden hacer estas copias sobre discos magnéticos. Debido a la gran cantidad de espacio que requiere una copia tota!, lo habitual es usar este tipo de almacenamiento para las copias de respaldo incrementales. Actual mente, sin embargo, existen discos extraíbles baratos y de gran capacidad, por lo que se está usando este tipo de dispositivos para hacer copias de respaldo de sistemas pequeños, especialmente com putadoras personales que no suelen tener una unidad de cinta conectada.

En caso de que existan requisitos fuertes de seguridad, las copias de respaldo se guardan en cajas fuertes o en edificios distintos a los del sistema original. Además, se mantienen varias copias de los datos.

Almacenamiento con redundancia

En los sistemas donde se necesita mucha disponibilidad, se puede almacenar la información de forma redundante en múltiples discos. Además, en caso de que se necesite fiabilidad, se puede añadir información de paridad para poder reconstruir los datos en caso de fallo de una unidad de almacenamiento.

La técnica clásica de almacenamiento redundante es usar **discos espejo** [Tanenbaum, 1992], es decir, dos discos que contienen exactamente lo mismo. En este caso, toda la información se escribe en ambos discos, pero se lee sólo de uno. En caso de que una lectura falle, siempre se tiene la otra copia de los datos. Este tipo de redundancia tiene dos problemas principales:

- Desperdicia el 50 por 100 del espacio de almacenamiento.

Digitalización realizada con propósito académico

- Reduce mucho el rendimiento de las operaciones de escritura. Ello se debe a que una escritu ra no se puede confirmar como válida hasta que se ha escrito en ambos discos espejo, lo que significa dos operaciones de escritura.

Una técnica más actual consiste en usar dispositivos **RAID** (Redundant Arrav of Independent Disks)a nivel hardware [Wilkes, 1995] o software [Chen, l995]. Estos dispositivos usan un conjun to de discos para almacenar la información y otro conjunto para almacenar información de paridad del conjunto anterior. La Figura 8.28 muestra un dispositivo de este estilo, donde toda la paridad se almacena en el mismo disco. Se puede conseguir todavía más fiabilidad, y eficiencia, distribuyendo la paridad por todos los dispositivos. En el Capítulo 7 se estudian más a fondo estos dispositivos.

Recuperación del sistema de archivos

Las técnicas anteriores no sirven de mucho si los sistemas de archivos quedan en estado incoherente debido a fallos de la computadora o a operaciones inadecuadas de los usuarios. Por ejemplo, su ponga que un usuario de un sistema UNIX apaga la computadora mediante el interruptor eléctrico sin ejecutar previamente los procedimientos de apagado del sistema operativo. En este caso, toda la información nueva de los bloques de la cache modificados durante los 30 segundos anteriores se

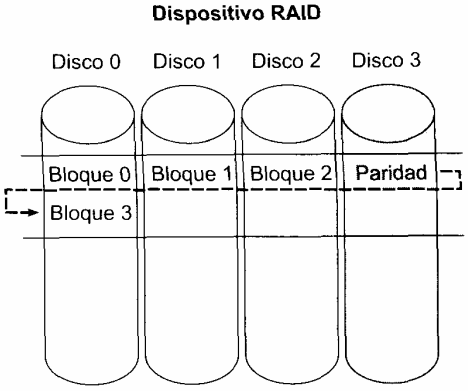


Figura 8.28. *Información redundante en un dispositivo RAID tipo IV.*

habrá perdido. Este problema es especialmente crítico si algunos de los bloques perdidos contienen datos de directorios o nodos-i, ya que la imagen del archivo o directorio afectado existente en el disco estará incoherente. Imagine el caso de un archivo escrito frecuentemente. Se le han asignado bloques nuevos y se han marcado como ocupados. Su nodo-i está en la cache y el mapa de bits también. Si se apaga la computadora cuando se ha escrito el nodo-i pero no el mapa de bits, habrá bloques marcados como libres en el disco que estarán asignados al nodo-i de un archivo. Este problema puede existir incluso en sistemas que usan escritura inmediata a disco, ya que la mayoría de las aplicaciones del sistema de archivos implican la actualización de varias estructuras de infor mación. Para tratar de resolver este tipo de problemas, los servidores de archivos proporcionan operaciones para comprobar, y a ser posible recuperar, la coherencia de los datos. Estas operaciones se acceden mediante un programa de utilidad, que en el caso del sistema operativo UNIX se denomina fsck [Foxley 1985].

Digitalización realizada con propósito académico

Cuando se arranca una computadora o se monta un dispositivo después de un fallo, el sistema operativo comprueba el estado de coherencia del sistema, o sistemas, de archivos. Hay dos aspectos importantes en estas comprobaciones:

• Comprobar que la estructura física del sistema de archivos es coherente.

• Verificar que la estructura lógica del sistema de archivos es correcta.

La comprobación de la estructura física del sistema de archivos se hace mediante la verifica ción del estado de los bloques del mismo. Para ello se comprueba la superficie del dispositivo de almacenamiento. En caso de detección de errores, los sistemas operativos suelen incluir utilidades para intentar reparar los bloques dañados o sustituirlos por bloques en buen estado.

La comprobación de la estructura fisica del sistema de archivos se hace mediante la verificación del estado de los bloques del mismo para ello se comprueba la superficie del dispositivo de almacenamiento. En caso de deteccion de errores, los sistemas opererativos suelen incluir utilidades para intentar reparar los bloques dañados o sustituirlos por bloques en buen estado. La comprobación de la estructura logica requiere la ejecución de varios pasos para verificar todos los componentes del sistema de archivos. Si el sistema falla cuando se estaba modificando un archivo, hay varias piezas de información que pueden estar en fase de modificación y que pueden fallar: bloques, nodo-i, bloques indirectos, directorios y mapas de bits. Puesto que el sistema opera tivo no tiene forma de saber dónde puede estar la anomalía, es necesario reconstruir el estado del sistema de archivos completo. Para ello:

1. Se comprueba que el contenido del superbloque responde a las características del sistema de archivos.

2. Se comprueba que los mapas de bits de nodos-i se corresponden con los nodos-i ocupados en el sistema de archivos.

3. Se comprueba que los mapas de bits de bloques se corresponden con los bloques asigna dos a archivos.

4. Se comprueba que ningún bloque esté asignado a más de un archivo.

5. Se comprueba el sistema de directorios del sistema de archivos, para ver que un mismo nodo-i no está asignado a más de un directorio.

Para llevar a cabo las **comprobaciones del estado de los bloques**, el programa lleva a cabo las siguientes acciones:

1. Construye dos tablas, cada una con un contador para cada bloque. La primera indica las veces que un bloque es referenciado desde los nodos-i. La segunda indica si un bloque está libre u ocupado.

2. Lee los nodos-i de cada archivo e incrementa los contadores de los bloques referenciados.

3. Lee los mapas de bits de bloques e incrementa el contador de los que están libres.

A continuación compara ambas tablas. La Figura 8.29 muestra varios estados posibles de los bloques de un sistema de archivos:

1. Si el sistema de archivos está **coherente**, los contadores de las tablas tendrán un 1 en una u otra tabla, pero no en ambas a la vez (caso 1 de la figura).
2. En caso contrario, puede ocurrir que un bloque ocupado no esté asignado a un archivo (caso 2 de la figura). Se dice que ése es un **bloque perdido o sin propietario.** En este caso es posible que exista pérdida de información en algún archivo, por lo que la herramienta de recuperación los almacena en un directorio predefinido, que en UNIX es típicamente lost+found. Si el usuario es afortunado, la información le permitirá recuperar los datos perdidos. A continuación se indica que el bloque está libre.
3. Otra anomalía posible es que un bloque ocupado esté referenciado desde dos archivos (caso 3 de la figura). En este caso se dice que éste es un bloque reutilizado. La solución correcta en este caso es

Digitalización realizada con propósito académico

asignar un bloque libre, copiar los contenidos del reutilizado al nuevo bloque y sustituir la referencia en uno de los archivos por la del bloque nuevo.

4. El último error posible es que algún bloque esté dos veces en la lista de bloques libres (caso 4 de la figura). La solución de este problema consiste en reducir el contador de bloques libres a 1 (Aclaración 8.4).

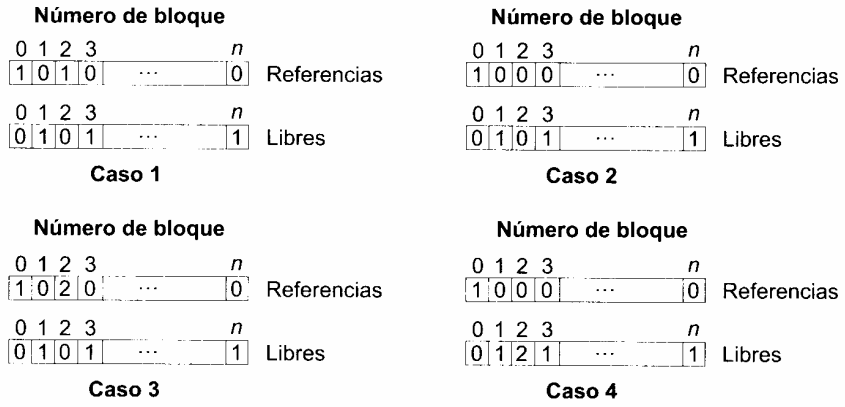


Figura 8.29. *Posibles estados de los mapas de bloques de un sistema de archivos.*

ACLARACIÓN 8.4

El caso 4 sólo tiene sentido si se usan listas de bloques libres. Si se usan mapas de bits para representar el estado de los recursos, el caso 4 de la figura no tiene sentido. Si se libera un bloque se pone a 1. Si se vuelve a liberar, se queda a 1. De esta forma, la aplicación de verificación nunca puede contar un bloque dos veces como bloque libre.

Un procedimiento similar al anterior se usa para verificar los mapas de bits de los nodos-i (Recordatorio 8.2).

RECORDATORIO 8.2

Este método se puede usar para verificar otros tipos de recursos de los que existe información acerca de cuáles están libres y cuáles usados.

Por último, la utilidad de recuperación comprueba el estado de los directorios. Para ello usa una tabla de contadores de nodos-i que indican los enlaces existentes a cada nodo-i. Para calcular dicho número, el programa recorre todo el árbol de directorios e incrementa el contador de cada nodo-i cuando encuentra un enlace al mismo. Cuando ha terminado, compara los contadores con el número de enlaces que existe en el nodo-i almacenado en disco. Si ambos valores no coinciden, se ajusta el del nodo-i al valor calculado.

8.6.8. Otros servicios

Digitalización realizada con propósito académico

Dada la importancia que para algunos sistemas tiene la coherencia de los datos almacenados en el sistema de archivos y la disponibilidad de los mismos, algunos servidores de archivos incluyen servicios orientados a satisfacer estas necesidades. Aunque existen múltiples mecanismos de este tipo, los tres más populares son la actualización atómica, las transacciones y la replicación.

Un servidor de archivos que proporciona **actualización atómica** [Goscinski, 1991], o indivisi ble, asegura a los usuarios que sus operaciones están libres de interferencia con las de otros usuarios y que la operación se realiza completamente o no tiene ningún efecto en el sistema. Esta semántica todo o nada permite asegurar que una operación no tenga efecto a no ser que se complete satisfacto riamente. Existen varias formas de implementar esta semántica. Algunas muy populares son usar mecanismos de almacenamiento estable, archivos de registro o versiones de archivos. De esta for ma, todas las operaciones se pueden ejecutar en estado provisional, hasta que el sistema está seguro de que concluyen satisfactoriamente. En ese momento se convierten en definitivas. En caso de fallo, se descartan las acciones provisionales.

En muchos casos no basta con asegurar al usuario que una operación sea atómica. La razón es que el usuario puede querer que un conjunto de operaciones relacionadas se ejecuten de forma atómica. En estos casos, el servidor de archivos puede ofrecer servicios de **transacciones** [Gray, 1981]. Un servidor de archivos transaccional permite ejecutar operaciones atómicas que agrupan a varias operaciones de entradalsalida y que se ejecutarán con semántica todo o nada. Desde el punto de vista del usuario, una transacción es una secuencia de operaciones que definen un paso único para transformar los datos de un estado incoherente a otro. Todas las operaciones que forman parte de una transacción se agrupan entre dos etiquetas, tales como begin y endtrans, que

definen el principio y el fin de la operación. Una vez más, todo sistema con atomicidad debe ser capaz de deshacer un conjunto de operaciones en caso de fallo. La reversibilidad es consustancial a los sistemas atórnicos. Los servidores de archivos transaccionales deben resolver además problemas de aislamiento de operaciones, serialización, control de concurrencia, etc. Todos ellos se escapan del ámbito de este libro, por lo que se refiere al lector interesado en bibliografía específica del tema.

La **replicación** es un mecanismo importante para incrementar el rendimiento de las lecturas del servidor de archivos y la disponibilidad de los datos. Consiste en mantener varias copias de los datos y otros recursos del sistema [Bereton, 1986]. Silos datos están replicados, en dos servidores o dispositivos distintos, se puede dar servicio a los usuarios aunque uno de ellos falle. Además, si varios servidores de archivos procesan los mismos datos, las posibilidades de error se reducen drásticamente. Sin embargo, proporcionar replicación transparente en los servidores de archivos no es inmediato. Para que los usuarios no sean conscientes de que existen varias copias de los datos, el servidor de archivos debe proporcionar acceso a los mismos a nivel lógico, y no físico. Cuando exista una petición de lectura, el usuario debe recibir un único conjunto de datos. Además, el sistema debe asegurar que la actualización de los datos se lleva a cabo en todas las réplicas en un tiempo aceptable. Para satisfacer estos requisitos se han propuesto dos modelos arquitectónicos básicos: copia primaria y gestión colectiva (Fig. 8.30). En el **modelo de copia primaria**, todos los servicios de escritura se piden a un servidor maestro de réplicas. Este gestor se encarga de propagar

Digitalización realizada con propósito académico

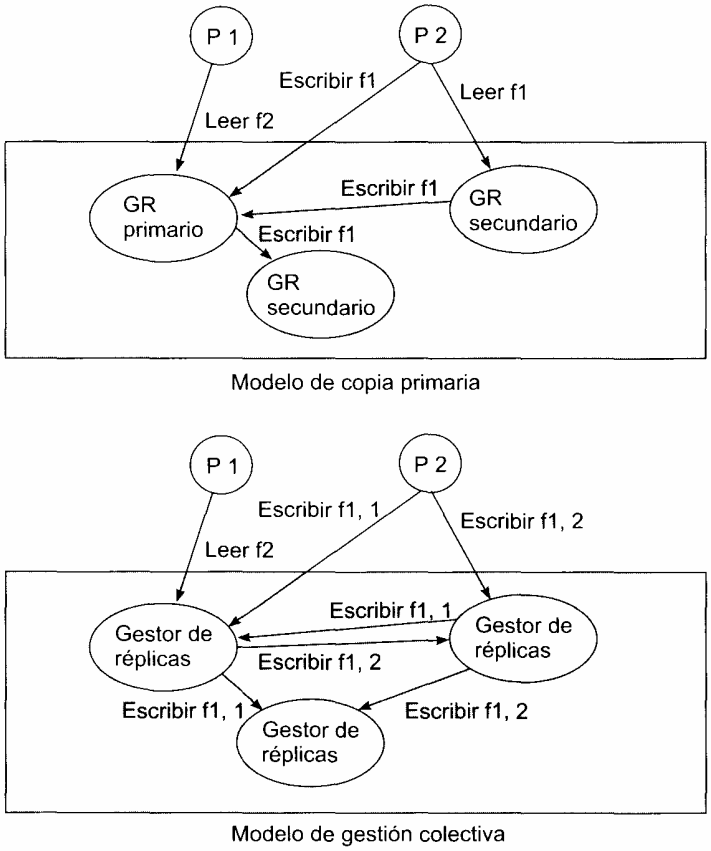


Figura 8.30. *Modelos de gestión de réplicas.*

las operaciones a los esclavos que tienen las réplicas. El mayor problema de este modelo es que el maestro es un cuello de botella y un punto de fallo único. Como puede verse en la figura, con este modelo todas las escrituras deben ejecutarse en el primario, que posteriormente se encarga de actualizar las réplicas de otros gestores. En el **modelo de gestión colectiva**, todos los servidores son paritarios y no existe la relación maestro-esclavo. Las peticiones de lectura y escritura se sirven en cualquier servidor, pero las últimas deben ser propagadas a todos los servidores con copia de los datos afectados. Como puede verse en la figura, con este modelo, las escrituras pueden ir a cual quier gestor de réplicas, pero éste debe propagar las modificaciones a todos los gestores de réplicas restantes. Este modelo es más eficiente y tolerante a fallos, pero su diseño e implementación es muy complicada.

8.7. PUNTOS A RECORDAR

- Desde el punto de vista de los usuarios y las aplicacio nes, los archivos y los directorios son los elementos centrales del sistema. Cualquier usuario genera y usa información a través de as aplicaciones que ejecuta en el sistema. En todos los sistemas operativos de propósi to general, las aplicaciones y sus datos se almacenan en archivos no volátiles, lo que permite su posterior reutili zación.

Digitalización realizada con propósito académico

- Un archivo es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de inft)rmación relacio nada entre sí bajo un mismo nombre. Desde el punto de vista del usuario, el archivo es la única forma de gestio nar el almacenamiento secundario.
- Todos los sistemas operativos tienen un descriptor de archivo que almacena atributos del mismo tales como nombre, identificador, tipo, mapa de bloques, tiempos de acceso, etc. El acceso al archivo se realiza, en el ám bito del sistema, usando este descriptor. Descriptores de este estúo son e'! nodo-i de UNIX y una entrada MFT de Windows NT.
- Todos los sistemas operativos proporcionan mecanis mos de nombrado que permiten asignar un nombre a un archivo en el momento de su creación. Este nombre acompaña al objeto mientras existe y le identifica de forma única. Los nombres de usuario se expresan como cadenas de caracteres y pueden estar ordenados con al gún tipo de jerarquía.
- Desde el punto de vista del usuario, la información de un archivo puede estructurarse como una lista de carac teres, un conjunto de registros secuencial o indexado, etcétera. Desde el punto de vista del sistema operativo, la estructura interna de archivo está oculta al usuario y suele ser distinta para cada sistema operativo. Una es tructura de usuario muy habitual es la tira secuencial de caracteres.
- Hay distintas formas de acceder a un archivo, aunque no todos los sistemas operativos proporcionan todas ellas. Para una aplicación elegir adecuadamente la forma de acceder a un archivo suele ser un problema im portante de diseño. Las dos formas de acceso más hahi tuales son el acceso secuencial y el directo (o aleatorio).
- La **semántica de coutilización** especifica qué ocurre cuando varios procesos acceden de forma simultánea al mismo archivo y especifica el momento en el que las modificaciones que realiza un proceso sobre un archivo pueden ser observadas por los otros. Tres semánticas que se usan frecuentemente son la de UNIX. la de se sión y la de archivos inmutables.
- Un **directorio** es un objeto que relaciona de forma uní voca el nombre de usuario de un archivo y el descriptor interno del mismo usado por el sistema operativo. Para evitar ambigüedades, un mismo nombre no puede iden tificar nunca a dos archivos distintos, aunque varios nombres se pueden referir al mismo archivo. Los direc torios se suelen implementar como archivos del sistema que sirven para organizar y proporcionar información acerca de la esifuctrnac de los archivos ea l siste mas de archivos.
- Un directorio es un objeto y debe ser representado por una estructura de datos. Esta estructura es característica de cada sistema de directorios. En MS-DOS, la entrada de directorio almacena también características del archi vo que representa. En UNIX, la entrada de directorio es una tupla <nombre, nodo-i>. La organización de es tas entradas se puede hacer mediante directorios de un nivel, de un nivel por usuario, de árbol y de grafo acíclico.
- Hay dos. posibles formas de especificar un nombre: nom bre completo del archivo, denominado nombre absolu to, o nombre de forma relativa a algún subdirectorio del árbol de directorios, denominado nombre relativo.
- Una de las decisiones más importantes de diseño de un sistema operativo es ofrecer o no un árbol único de di rectorios para todo el sistema, independientemente de los dispositivos físicos o lógicos en que éstos estén al macenados. En UNIX se puede montar sistemas de ar chivos y ofrecer esta imagen única. En MS-DOS y en

Digitalización realizada con propósito académico

Windows NT, el árbol de directorios es único por dispo sitivo lógico.

- Las principales llamadas al sistema para gestión de ar chivos definidas en POSIX son: creat, open, do se, link, unlink, read, write y lseek.
- Las principales llamadas al sistema para gestión de ar chivos definidas en Win32 son: CreateFi le, DeleteFile, CloseHandle, ReadFile, WriteFile y SetFilePointer.
- Las principales llamadas al sistema para gestión de di rectorios definidas en POSIX son: mkdir, rmdir, opendir, closedir, readdir y chdir.
- Las principales llamadas al sistema para gestión de di rectorios definidas en Win32 son: CreateDirec tory, RemoveDirectory, FindFirstFile, FindNextFile, FindClose, SetCurrentDi rectory y GetCurrentDirectory.
- Previamente a la instalación del sistema operativo es necesario dividir física o lógicamente los discos en **particiones** o **volúmenes**. Una partición es una porción de un disco a la que se la dota de una identidad propia y que puede ser manipulada por el sistema operativo como una entidad lógica independiente.

Digitalización realizada con propósito académico

- ☑ Una vez creadas las particiones, el sistema operativo debe crear las estructuras de los sistemas de archivos dentro de esas particiones. Para ello se proporcionan mandatos como format o mkfs al usuario. Cuando se crea un sistema de archivos en una partición de un dis co, se crea una entidad lógica autocontenida con espa cio para la información de carga del sistema operativo, descripción de su estructura, descriptores de archivos, información del estado de ocupación de los bloques del sistema de archivos y bloques de datos.
- ☑ El superbloque, los mapas de bloques y los mapas de descriptores constituyen la metainformación de un sis tema de archivos. Si el superbloque se estropea, todo el sistema de archivos queda inaccesible. Si se estropean bloques de mapas, los bloques o descriptores afectados quedan inaccesibles. Si se estropean los bloques de des criptores en sí, varios archivos quedarán inaccesibles. Si se estropean bloques de datos, todo o parte de al gún archivo quedará inaccesible. Como se puede ob servar, los errores en la metainformación son muy graves.
- ☑ Existen distintos tipos de estructuras de sistemas de ar chivos: tradicional, FFS, LFS, con bandas, etc. Cada una de ellas es más apropiada para determinados propó sitos. El FFS es muy usado en la actualidad.
- ☑ Los servidores de archivos tienen una estructura interna que, en general, permite acceder a los distintos disposi tivos del sistema mediante archivos de distintos tipos, escondiendo estos detalles a los usuarios. Los principa les bloques estructurales del servidor de archivos son: sistema de archivos virtuales, módulo de organización de archivos, servidor de bloques y manejadores de dis positivo.
- ☑ El sistema de archivos virtuales proporciona una inter faz de llamadas de entrada/salida genéricas válidas para todos los tipos de sistemas de archivos instalados en el sistema operativo. Resuelve internamente aquellas que no son específicas de un tipo de sistema de archivos, como cambiar los tiempos de acceso, y llama al módulo de organización de archivos para que resuelva la parte específica de cada sistema de archivos.
- ☑ Los mecanismos de asignación hacen corresponder la imagen lógica del archivo con la imagen física que exis te en el almacenamiento secundario. Hay dos políticas de asignación básicas: bloques contiguos y bloques dis contiguos. En el descriptor de un archivo debe estar in cluida información de asignación. Cada sistema opera tivo usa un método distinto. MS-DOS usa una lista enlazada (FAT) y Windows NT y UNIX usan un índice multinivel. La información de asignación de un archivo es crítica para acceder a dicho archivo.
- ☑ Los dispositivos son finitos. Su espacio se asigna y libe ra a medida que las aplicaciones ejecutan peticiones de entrada/salida. Para conocer el estado de los recursos disponibles, todos los sistemas operativos tienen mapas de recursos y políticas de gestión de los mismos. Dos mecanismos populares para gestionar el espacio libre son los mapas de bits y la lista de bloques libres.
- ☑ Los mapas de recursos pueden ocupar mucho espacio. Además, su gestión puede ser costosa en tiempo. Para tratar de optimizar estas operaciones se juntan los recur sos, especialmente los bloques, en agrupaciones.
- ☑ Para optimizar el rendimiento del sistema de entra da/salida, el servidor de archivos incorpora mecanis mos de incremento de prestaciones tales como discos RAM, caches de nombres, caches de bloques y compre sión de datos.
- ☑ La destrucción de un sistema de archivos es, a menudo, mucho peor que la destrucción de una computadora. Es importante salvaguardar los datos en copias de respaldo y en sistemas redundantes. En casos de instalaciones de alta seguridad, las copias de respaldo se pueden almace nar en edificios o instalaciones distintas a las de la com putadora original.
- ☑ Un sistema de archivos puede quedar en estado incohe rente por mal uso, caídas de tensión, apagados indebi dos del sistema operativo, etc. Para tratar de resolver este tipo de problemas, los servidores de archivos inclu yen funcionalidad para comprobar, y a ser posible recu perar, la coherencia de los datos.

Digitalización realizada con propósito académico

- ☑ Actualmente, algunos servidores de archivos incorpo ran servicios avanzados tales como actualizaciones ató micas, transacciones o replicación.

8.8. LECTURAS RECOMENDADAS

Existe mucha bibliografía sobre sistemas de archivos y directorios que puede servir como lectura complemen taria a este libro. [Silberschatz, 1998], [19981 y [Tanenbaum, 1997] son libros generales de sistemas operativos en los que se puede encontrar una explicación completa del tema con un enfoque docente.

[Grosshans, 1986] contiene descripciones de las estructuras de datos que maneja el sistema de archivos y presenta aspectos de acceso a archivos. Con esta información, el lector puede tener una idea clara de lo que es un archivo y de cómo valorar un sistema de archivos. [Folks, 1987] incluye abundante información sobre estructuras de archivo, mantenimiento, búsqueda y gestión de archivos. [Abernathy, 1973] describe los princi pios de diseño básico de un sistema operativo, incluyendo el sistema de archivos. En [McKusick, 1996] se estudia el diseño del Fast File Svstem con gran detalle, así como las técnicas de optimización usadas en el mismo. [Smith, 19941 estudia la influencia de la estructura del sistema de archivos sobre el rendimiento de las operaciones de entrada/salida. [Staelin, 1988] estudia los tipos de patrones de acceso que usan las aplicaciones para acceder a los archivos. Esta información es muy importante si se quiere

Digitalización realizada con propósito académico

optimizar el rendimiento del sistema de archivos. Para otros mecanismos de incremento de prestaciones, consulte [Smith, 1985], [Krieger, 19941, [Davy, 1995] y [Ousterhout, 1989].

Para programación de sistemas operativos puede consultar los libros de [Kernighan, 1978] para aprender el lenguaje C, [Kernighan, 1984] y [Rockind, 1985] para la programación de sistemas con UNIX y [Andrews. 1996] para la programación de sistemas con Windows NT.

Para los ejemplos y casos de estudio se recomienda consultar [Beck, 1996] para el caso de LINUX, [Goodheart, 1994] para UNIX y 1 [Solomon, 1998] para Windows NT. Una explicación detallada del sistema de archivos de Windows NT puede encontrarse en [Nagar, 1997]. Una explicación detallada del sistema de archivos de VMS puede encontrarse en [Kenah, 1988].

8.9. EJERCICIOS

8.1. ¿Se puede emular el método de acceso aleatorio con el secuencial? ¿Y viceversa? Explique las ven tajas e inconvenientes de cada método.

8.2. ¿Cuál es la diferencia entre la semántica de coutili zación UNIX y la de versiones? ¿Podría emularse la semántica UNIX con la de versiones?

8.3. ¿Podrían establecerse enlaces al estilo de los UNIX en Windows NT? ¿Por qué?

8.4. ¿Es UNIX sensible a las extensiones del nombre de archivo? ¿Lo es Windows NT?

8.5. ¿Cuál es la diferencia entre nombre absoluto y re lativo? lndique dos nombres relativos para /users/miguel/datos. Indique el directorio respecto al que son relativos.

8.6. ¿Cuál es la ventaja de usar un grafo acíclico frente a usar un árbol como estructura de los directorios? ¿Cuál puede ser su principal problema?

8.7. En UNIX existe una aplicación mv que permite re- nombrar un archivo. ¿Hay alguna diferencia entre implementarla usando link y unlink y hacerlo copiando el archivo origen al destino y luego bo rrando este último?

8.8. Modifique el Programa 8.1 para que en lugar de copiar un archivo sobre otro lea un archivo y lo saque por la salida estándar. Su programa es equi valente al mandato cat de UNIX.

8.9. Modifique el Programa 8.2 para que en lugar de copiar un archivo sobre otro lea un archivo y lo saque por la salida estándar. Su programa es equi valente al mandato type de Windows o MS-DOS.

8.10. Usando llamadas POSIX, programe un mandato que permita leer un archivo en porciones, especifi cando el formato de la porción. Para incrementar el rendimiento de su mandato, debe hacer lectura adelantada de la siguiente porción del archivo.

8.11. Modifique el Programa 8.4 (mijs) para que, además de mostrar el nombre de las entradas del directorio, muestre algo equivalente a la salida del mandato ls -l de UNIX.

8.12. Haga lo mismo que en el Ejercicio 8.11 para el Programa 8.5 del entorno Windows NT.

8.13. ¿Qué método es mejor para mantener los mapas de bloques: mapas de bits o listas de bloques libres? Indique por qué. ¿Cuál necesita más espacio de al macenamiento? Explíquelo.

8.14. ¿Qué problema tiene usar bloques grandes o agru paciones? ¿Cómo puede solucionarse?

8.15. ¿Qué es mejor en un sistema donde hay muchas escrituras: un sistema de archivos convencional o uno de tipo LFS? ¿Y para lecturas aleatorias?

8.16. ¿Es conveniente mantener datos de archivo dentro del descriptor de archivo, como hace Windows NT?

Digitalización realizada con propósito académico

¿Es mejor tener un descriptor como el nodo-i de UNIX?

8.17. Tener una buena tasa de aciertos en la cache (por centaje de bloques que se encuentran en la cache) es fundamental para optimizar la entradalsalida. Suponga que el tiempo medio de acceso de un dis co es de 17 milisegundos y satisfacer una petición desde la cache cuesta 0,5 milisegundos. Elabore una fórmula para calcular el tiempo necesario para satisfacer una petición de n bloques. Considere la tasa de aciertos de la cache como un parámetro más de dicha fórmula. Calcule el tiempo de servicio para cinco bloques y una tasa de aciertos del 85 por lOO.

8.18. ¿Qué problemas de seguridad puede plantear la ca che de nombres? ¿Cómo se pueden solucionar?

8.19. Imagine que está comprobando el estado del siste ma de archivos y observa que el nodo-i 342 tiene cuatro referencias pero aparece libre en el mapa de bits de nodos-i. ¿Cómo resolvería el problema?

8.20. Imagine el problema inverso al del Ejercicio 8.19. El nodo-i no tiene referencias, pero aparece como ocupado. ¿Cómo lo solucionaría?

Digitalización realizada con propósito académico

8.21. ¿Cómo se puede mejorar el tiempo de asignación de bloques de un dispositivo muy fragmentado que sólo admite acceso aleatorio? ¿Valdría su solución para dispositivos de acceso secuencial?

8.22. Determine cuántos accesos físicos a disco serían necesarios, como mínimo, en un sistema UNIX para ejecutar la operación: fd= open("/lib/agenda/direccjones", ORDONLY); Explique su respuesta. Suponga que la cache del sistema de archivos está inicialmente vacía.

8.23. Realice el Ejercicio 8.22 para el archivo "C: \lib\ agenda\direcciones". ¿Por qué el número de accesos puede ser tan distinto en ambos sistemas?

8.24. Se quiere diseñar un sistema de archivos para un sistema operativo que dará servicio a un entorno del que se sabe lo siguiente:

- El tamaño medio de los archivos es de 1,5 KB.

- El número medio de bloques libres es el 7 por lOO del total.

- Se usan 16 bits para la dirección del bloque.

Para este sistema se selecciona un disco duro con bloques físicos de 1 kB y con una capacidad igual a la del máximo bloque direccionable. Teniendo en cuenta que se debe optimizar el uso del disco y la velocidad de acceso (por este orden) y que la memoria física de que se dispone es sufi ciente, conteste razonadamente a las siguientes preguntas:

a) ¿Cuál será el tamaño de bloque lógico más adecuado?

b) ¿Cuál será el método más adecuado para lle var el control de los bloques lógicos del disco?

8.25. Un proceso de usuario ejecuta operaciones de en tradalsalida en las que pide los siguientes bloques de un sistema de archivos: 1 2 3 4 1 3 10 2 3 2 10 1

Suponiendo que en la cache de bloques del sistema operativo caben 4 bloques y que inicialmente está vacía. Se pide:

a) Hacer una traza de la situación de los bloques de la cache para cada petición de bloque del proceso, suponiendo que se usa un política de reemplazo LRU (Least Recentlv Used).

b) Hacer una traza de la situación de los bloques de la cache para cada petición de h1oq proceso, suponiendo que se usa un política ifr reemplazo MRU (Most Recently Used).

8.26. En un sistema de archivos tradicional de UNIX, cada partición tiene un superbloque donde se guar da información acerca de la estructura del sistema de archivos. Sin este superbloque es imposible ac ceder al sistema de archivos. ¿Qué problemas pre senta una estructura de archivos como la anterior? ¿Cómo podrían resolverse? Indique un sistema de archivos en el que se han propuesto soluciones al problema anterior.

8.27. ¿Qué operaciones se hacen en UNIX para montar un sistema de archivos sobre un directorio? ¿Y en

Windows NT?

8.28. En Windows NT existe una utilidad in dentro de las utilidades del sistema. Cree un archivo ar chivo . orig y use ln para crear un enlace físico entre dos archivos de Windows NT, ejecutando el mandato:

in archivo.orig archivo,dest

Use el explorador para ver la entrada de ambos ar chivos. Borre el archivo. orig y describa lo que ocurre.

8.29. Los archivos de metadatos de un sistema de archi vos de Windows NT son archivos normales, deno minados MFT, pero están ocultos a los usuarios. Ejecute el mandato

C:\>dir /A:h

e indique cuál es el tamaño del archivo de metada tos del dispositivo lógico C:.

8.30. ¿Cuánto espacio de disco se necesita para tener re dundancia en un disco espejo? ¿Cuánto hace falta

Digitalización realizada con propósito académico

en un sistema RAID IV con cuatro discos, incluyendo los de paridad?

8.31.¿Cuál es la sobrecarga para las escrituras en un disco espejo? ¿Cuál es la sobrecarga para las escrituras en un sistema RAID con tres discos de datos y uno de paridad, asumiendo que se escribe de un golpe sobre todos los dispositivos?

Digitalización realizada con propósito académico

9

Seguridad y protección

En este capítulo se presentan los conceptos de seguridad y protección. Su objetivo es hacer comprender al lector la problemática de seguridad existente en los sistemas de computación y, en concreto, en los sistemas operativos. En este capítulo se tratan los siguientes temas:

- *Conceptos de seguridad y protección.*
- *Problemas de seguridad.*
- *Políticas de seguridad.*
- *Diseño de sistemas operativos seguros.*
- *Criptografía.*
- *Clasificaciones de seguridad.*
- *Seguridad y protección en sistemas operativos de propósito general.*
- *Servicios de protección y seguridad.*
- *El sistema de seguridad en Windows NT.*
- *Kerberos.*

Los servicios de seguridad son la parte del sistema operativo que se ocupa de controlar el acceso y de proteger los recursos, proporcionando mecanismos adecuados para implementar la política de protección adecuada. En los sistemas operativos antiguos, la política de seguridad está dispersa por el sistema. Sin embargo, en los modernos sistemas orientados a objetos (caso de Windows NT) el servidor de seguridad se encarga de todo lo relativo a la seguridad de los mismos.

9.1. CONCEPTOS DE SEGURIDAD Y PROTECCIÓN

Los sistemas de una computadora manejan información que suele ser valiosa para sus propietarios por lo que la seguridad de dichos sistemas es un elemento importante en el diseño de los sistemas operativos. En este capítulo se estudian los principales conceptos de seguridad y los mecanismos i protección que pueden usarse para proporcionar dicha seguridad a través del sistema operativa Aunque tradicionalmente los términos seguridad y protección se han usado indistintamente, actualmente ambos conceptos se distinguen claramente

La **seguridad** de un sistema tiene múltiples facetas, como se muestra en la Figura 9.1, incluyendo desde aspectos tales como la protección ante posibles daños físicos de los datos (fuegos terremotos, etc.) hasta el acceso indebido a los mismos (intrusos, fallos de confidencialidad, etc Los ataques contra la confidencialidad, la integridad o la disponibilidad de recursos en un sistet deben prevenirse y solventarse mediante la política y los mecanismos de seguridad de un sistema En el caso de un sistema informático hay varios elementos susceptibles de sufrir dichos ataques,) siendo suficiente proteger sólo alguno de ellos o protegerlos parcialmente. Como se puede ver en Figura 9.1, el hardware, el software y los datos de un sistema informático pueden sufrir ataques internos o externos al sistema. Por tanto, la seguridad debe tener en cuenta eventos externos provenientes del entorno en que opera el sistema. De nada sirve tener mecanismos de protección hiten muy buenos, si el sistema operativo no es capaz de identificar a los usuarios que acceden al sistema o si no existe una política de salvaguarda de datos ante la rotura de un disco.

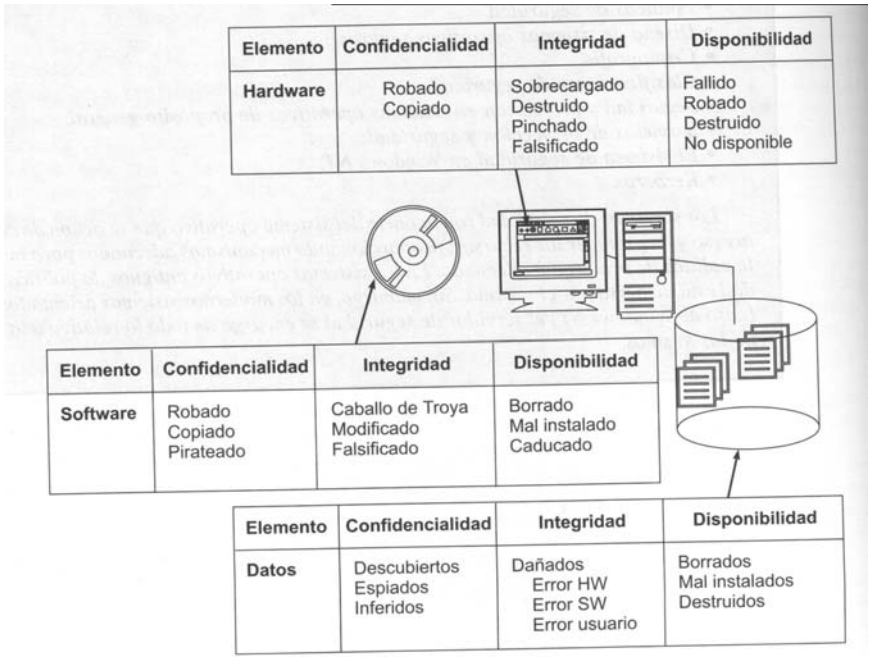


Figura 9.1. Problemas de seguridad de una instalación informática.

La **protección,** sin embargo, consiste en evitar que se haga un uso indebido de los recursos que están dentro del ámbito del sistema operativo. Para ello deben existir mecanismos y políticas que aseguren que los usuarios sólo acceden a sus propios recursos (archivos, zonas de memoria, etc.). Además, es necesario comprobar que los recursos sólo se usan por aquellos usuarios que tienen derechos de acceso a los mismos. Las políticas de protección y seguridad de hardware, software y datos deben incluirse dentro del sistema operativo, pudiendo afectar a uno o varios componentes del mismo. En cualquier caso, el sistema operativo debe proporcionar medios para implementar la política de protección deseada por el usuario, así como medios de hacer cumplir dicha política.

La seguridad de un sistema operativo se basa principalmente en tres aspectos de diseño:

- Evitar la pérdida de datos.
- Controlar la confidencialidad de los datos.
- Controlar el acceso a los datos y recursos.

La pérdida de datos puede deberse a catástrofes naturales o artificiales que afecten al sistema (terremotos, guerras, etc.), a errores del hardware o del software de la computadora (rotura de un disco, por ejemplo) o a errores humanos (p. ej.: borrado accidental de archivos). La protección frente a fallos físicos, para conseguir que el sistema sea fiable, está más relacionada con la gestión de datos que con el sistema operativo. Una solución frecuente para estos problemas es hacer que los administradores del sistema mantengan varias copias de los datos almacenadas en distintos lugares.

En el ámbito interno del sistema operativo hay operaciones que pueden violar la confidencialidad de los datos. Una simple asignación de bloque de disco libre a un usuario le proporcionará el bloque con el contenido del usuario anterior si el sistema operativo no tiene una política definida para este tipo de situaciones. En estos casos, siempre hay que limpiar los recursos de los datos anteriormente existentes. Sin embargo, controlar la confidencialidad de los datos es un problema de seguridad que sobrepasa el ámbito de los sistemas operativos, aunque una parte del problema puede resolverse en su ámbito interno. De nada sirve controlar muy bien el acceso a la base de datos de nóminas, si un operador de una compañía distribuye listas de personal con sus nóminas y datos personales. Otro ejemplo es la realización de transacciones comerciales a través de Internet, en muchas de las cuales se envían números de tarjeta de crédito sin protección. La solución de este tipo de problemas requiere actuaciones externas al sistema operativo, que pueden ser incluso policiales.

El control del acceso a datos y recursos sí es competencia directa del sistema operativo. Es necesario asegurar que los usuarios no acceden a archivos para los que no tienen permisos de acceso, a cuentas de otros usuarios o a páginas de memoria o bloques de disco que contienen información de otros usuarios (aunque ya estén en desuso). Para conseguirlo hay que aplicar criterios de diseño rigurosos y ejecutar pruebas de seguridad exhaustivas para todos los elementos del sistema, aunque se consideren seguros. El control de acceso incluye dos problemas que estudiaremos más adelante: autenticación de usuarios y protección frente a accesos indebidos.

A continuación se estudian los problemas de seguridad más importantes que conciernen al sistema operativo, aunque el lector debe tener en cuenta que los mecanismos de protección del sistema operativo sólo resuelven una mínima parte de los problemas de seguridad existentes en una instalación informática.

9.2. PROBLEMAS DE SEGURIDAD

Todos los sistemas operativos comerciales desarrollados hasta el momento han tenido algún problema de seguridad. Algunos ejemplos notorios son:

• Los fallos de UNIX que permitían, en su momento, que un usuario pudiera abortar un mandato con permisos de súper usuario, como mkdir, y quedarse con la identificación de súper-usuario.

• En versiones antiguas de UNIX, cualquier usuario podía leer el archivo passwd donde se almacenaban los usuarios y sus palabras clave cifradas. Con esta información se le podían aplicar a la clave todos los algoritmos de descifrado que se quisiera hasta romperla.

• En TENEX, un sistema operativo en desuso, se comprobaba la palabra clave carácter a carácter. Cuando se cometía un error se indicaba al usuario. De esta forma era muy sencillo introducir la clave carácter a carácter para buscar patrones válidos hasta descifrarla completamente. Como máximo era necesario probar todos los caracteres del alfabeto por cada carácter de la clave.

Muchos fallos no se deben al sistema de protección sino a la candidez de los usuarios. Poner como palabra de acceso al sistema el nombre de su esposa, de algún hijo o la matrícula de su coche o poner " . " en la cabecera del *path* (camino de búsqueda de archivos ejecutables) es postularse como candidato a un acceso indebido.

A continuación se describen los problemas más frecuentes en un sistema informático.

9.2.1. Uso indebido o malicioso de programas

Algunos problemas de seguridad están relacionados con el uso indebido, o malicioso, de programas de un usuario [Hoffman, 1990]. Dos formas frecuentes de generar fallos de seguridad usando estas técnicas son el caballo de Troya y la puerta de atrás. El **caballo de Troya** se basa en la idea de crear un programa para que haga cosas no autorizadas en el sistema cuando actúa en el entorno adecuado. Algunos ejemplos incluyen: un editor de texto modificado para que haga copias de los archivos a los directorios del intruso o un programa de login modificado para grabar los nombres y palabras de acceso de los usuarios. La **puerta de atrás** consiste en crear un agujero de seguridad al sistema a través de un programa privilegiado que lo permite. Algunos ejemplos incluyen:

• Convencer a un programador de sistema para que le deje acceder con privilegios anormales.

• Permitir que el diseñador de un programa se reserve el acceso por medios no conocidos, tales como una identificación reservada.

• Meter parches en un compilador para que añada código no autorizado a cualquier programa.

Los problemas de seguridad generados por programas maliciosos son generalmente difíciles de detectar y corregir. En los años sesenta dos programadores robaron varios millones de dólares en un banco americano por el método de traspasar a una cuenta suya las fracciones de centavo que se redondeaban en las operaciones. Sólo fueron detectados cuando el balance de caja empezó a fallar de forma escandalosa.

Las puertas de atrás no siempre son perjudiciales o maliciosas. Pueden tener varias causas:

• Se dejan a propósito para probar el sistema.

• Se dejan para mantener el sistema.

• Se dejan como un medio de acceso encubierto, o canal encubierto, que permite extraer información del sistema sin que el dueño sea consciente de ello.

Sólo este último caso es realmente peligroso, si bien cualquier puerta de atrás es un peligro potencial para la seguridad del sistema. La Figura 9.2 muestra un ejemplo de inserción de un canal encubierto en un programa.

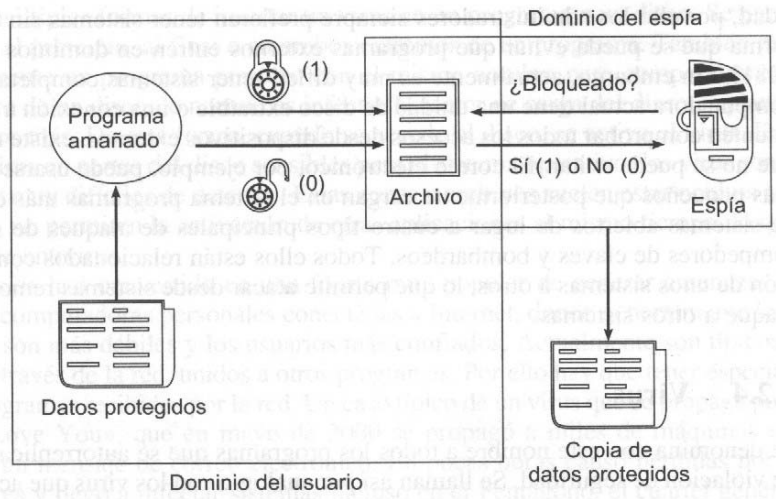


Figura 9.2. *Ejemplo de canal encubierto usando cerrojos sobre un archivo.*

9.2.2. Usuarios inexpertos o descuidados

Los usuarios inexpertos o descuidados son potencialmente muy peligrosos. Cosas tales como borrar archivos no deseados, dejar abiertos los accesos al sistema durante largo tiempo o escribir la palabra clave en un papel junto a la computadora son más frecuentes de lo que parece.

Los problemas de seguridad debidos a usuarios descuidados deben tener atención especial por parte del administrador del sistema, que puede conseguir paliar muchos de ellos. Por ejemplo, ciertos mandatos, como delete o rm (borrar archivos), se pueden configurar para pedir confirmación de cada acción que realizan, el acceso a un sistema se puede cerrar si se detecta que lleva un cierto tiempo inactivo, etc.

9.2.3. Usuarios no autorizados

Los sistemas operativos, como UNIX o Windows NT, mantienen **cuentas** para los usuarios autorizados. El acceso a dichas cuentas se protege mediante contraseñas, o palabras clave, que sólo debe conocer el dueño de las mismas. Uno de estos usuarios, denominado **administrador o súper-usuario** puede acceder a todo el sistema saltándose las protecciones del resto de los usuarios.

Para poder modificar o ejecutar algo en la computadora es necesario acceder a una cuenta de usuario. El proceso de reconocimiento del usuario, denominado **autenticación,** es muy importante para evitar que usuarios no autorizados accedan al sistema. Un usuario no autorizado, o *pirata,* trata de saltarse o romper el proceso de autenticación para entrar en el sistema de computación con la identidad de un usuario legítimo, especialmente con la del administrador. Si lo consigue, puede hacer todo aquello a lo que su identidad falsa le conceda derecho. En caso de que haya conseguido acceder como administrador puede hacer lo que desee, desde borrar datos hasta crearse una cuenta verdadera con identidad falsa o cambiar la contraseña del administrador para que sólo pueda acceder él.

La posibilidad de cargar programas externos en un sistema o de que existan usuarios no autorizados se incrementa mucho en sistemas con arquitectura abierta conectados en redes. La existencia de sistemas con arquitectura abierta origina muchas situaciones azarosas en el entorno de segu-

ridad, por ello los administradores siempre prefieren tener sistemas sin interferencias externas, de forma que se pueda evitar que programas externos entren en dominios de otros usuarios [Parker, 1981]. Sin embargo, actualmente es muy difícil tener sistemas completamente aislados. Cualquier computadora actual tiene una unidad de disco extraíble o una conexión a la red. Además, aunque se intenten comprobar todos los accesos desde dispositivos extemos, existe un cierto grado de apertura que no se puede evitar. El correo electrónico, por ejemplo, puede usarse como portador de programas pequeños que posteriormente cargan en el sistema programas más destructores. La existencia de sistemas abiertos da lugar a cuatro tipos principales de ataques de seguridad: virus, gusanos, rompedores de claves y bombardeos. Todos ellos están relacionados con la capacidad de propagación de unos sistemas a otros, lo que permite atacar desde sistemas remotos o enviar programas de ataque a otros sistemas.

9.2.4. Virus

Se denomina con este nombre a todos los programas que se autorreplican con fines destructivos o de violación de seguridad. Se llaman así por analogía con los virus que actúan sobre los seres vivos. Como ellos, necesitan de un programa que transporte el virus y de un agente que los transmita para infectar a otros programas. En todos los casos se trata de un programa o un trozo de código que se añade a otro y que se activa cuando se ejecuta el programa portador. Cuando esto ocurre, además de ejecutarse el portador, se ejecuta el virus.

La Figura 9.3 muestra el proceso descrito. El fabricante de un virus genera un programa y lo almacena en un disquete. Cuando se copia dicho archivo en una computadora y se ejecuta, el virus se instala en el disco duro del sistema. A partir de este momento, infecta a todos los discos extraíbles que se instalen en el sistema. En un determinado momento, el virus se activa y destruye los datos del sistema. Un caso típico de virus con caballo de Troya es el Viernes 13. Este virus sólo se activa en los días que son viernes y 13.

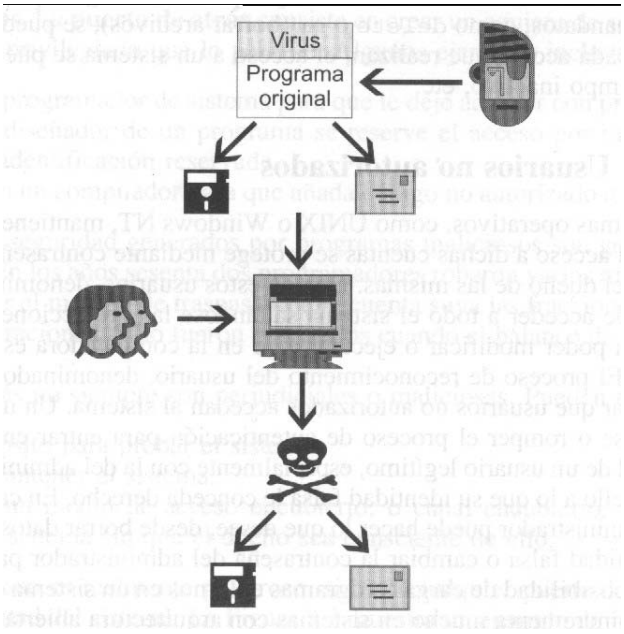


Figura 9.3. Instalación y propagación de virus.

Existen múltiples formas de insertar un virus en un programa o un disco. Se puede insertar en medio, añadir al principio, al final o en ambos extremos de un programa. También puede reemplazar completamente un programa por otro con el mismo nombre pero comportamiento totalmente distinto. Dentro de un disco, lo normal es infectar el sector de carga del disco, suplantándolo con el virus, y algunos más. De esta forma, cuando se accede al disco, se activa el virus. En el caso de la inserción de virus en partes del disco sensibles para el sistema operativo, los virus son especialmente peligrosos y muy difíciles de detectar. Estas zonas o archivos suelen estar ocultos al usuario y, en muchos casos, no permiten la activación de otras aplicaciones al mismo tiempo, lo que hace difícil la ejecución de anti virus.

Los virus se han convertido en una forma muy popular de generar amenazas de seguridad, sobre todo en computadoras personales conectadas a Internet, donde generalmente los mecanismos de protección son más débiles y los usuarios más confiados. Actualmente son típicos los virus que se propagan a través de la red, unidos a otros programas. Por ello hay que tener especial cuidado con los datos o programas recibidos por la red. Un caso típico de un virus que se propaga por Internet es el del virus «I Love You», que en mayo de 2000 se propagó a miles de máquinas en Internet incorporado en un mensaje de correo electrónico. En pocas horas causó pérdidas por miles de millones de dólares y llegó a infectar

sistemas incluso en el Pentágono, el cuartel general del ejército americano.

La solución para una infección por un virus puede adoptar dos formas:

• Comprobación manual de todos los dispositivos de almacenamiento para limpiarlos del virus.

• Creación de un antídoto que también se propague y limpie el virus.

En cualquier caso, lo mejor es practicar técnicas preventivas tales como comprobación rutinaria de discos del sistema, comprobación de cualquier disco extraíble que se instale, comprobación de cualquier programa recibido antes de ejecutarlo, etc.

ADVERTENCIA 9.1

¡Nunca ejecute un programa o abra un archivo recibido por la red sin pasar antes controles antivirus! Puede ser portador de virus. Aunque no se haya infectado de forma maliciosa, la acción del virus será igualmente destructiva.

9.2.5. Gusanos

El 2 de noviembre de 1988, R. T. Morris, un estudiante de Cornell, liberó un programa gusano en Internet. Anteriormente había descubierto dos fallos de seguridad en el UNIX de Berkeley que le permitían acceder a miles de máquinas conectadas a Internet, especialmente aquellas que permitían accesos por ftp a usuarios anónimos. Aprovechando esta circunstancia, fabricó un gusano que explotaba los fallos de seguridad para acceder a sistemas remotos y, además, se replicaba a sí mismo para acceder a más sistemas. El gusano estaba compuesto por dos programas: un pequeño programa de 99 líneas en C y el gusano principal. El programa pequeño era un cargador que, una vez compilado y ejecutado en el sistema destino, cargaba el gusano principal. El gusano leía las tablas de encaminamiento del sistema infectado y enviaba el cargador a todos los sistemas remotos que podía, usando para ello tres mecanismos de penetración:

• Intérprete de mandatos remoto (rsh).

• **Demonio del** finger.

• sendmail.

A través de estos mecanismos, el gusano provocó la caída de miles de computadoras en la red, antes de ser detectado y eliminado. Desde entonces se han liberado varios programas de este tipo por la red.

A pesar de sus efectos negativos se han propuesto varias formas de usar el mecanismo de autorreplicación para fines lícitos. Algunas posibles utilizaciones son buscar computadoras desocupadas para distribuir carga en el sistema, proporcionar tolerancia a fallos permitiendo que un segmento del gusano arranque otro si falla su

computadora, realizar tareas en paralelo, etc. El principal problema de los gusanos no es que sean destructivos, que la mayoría no lo son, sino que colapsan las redes de comunicaciones.

9.2.6. Rompedores de sistemas de protección

Estos programas llevan a cabo distintas pruebas sobre sistemas, generalmente remotos, para tratar de romper la seguridad de los mismos y poder ejecutar accesos ilegales. Para ello prueban con los mecanismos que dieron fallos de seguridad anteriormente en virus y gusanos: archivos con se-tuid, intérpretes de mandatos remotos, ftp anónimo, finger, etc. Una de las pruebas que ejecutan típicamente consiste en tratar de adivinar las palabras clave que tienen los usuarios para acceder al sistema. Esta prueba, que se basa en distintos tipos de programas de cifrado y un diccionario de palabras clave, compara las del diccionario con las de los usuarios del sistema. Es sorprendente comprobar que las palabras clave que se usan para acceder al sistema son en muchos casos palabras de uso común, nombres propios o incluso la misma identificación del usuario. Por ello, la mayoría de los sistemas operativos exigen actualmente el uso de palabras clave con una longitud mínima, con caracteres especiales, etc.

Satán *(Security Administrator Tools for Analyzing Networks)* es un buen ejemplo de este tipo de programas. Según sus autores. Satán es un simple recolector de información que está disponible para que cualquiera pueda comprobar el estado de seguridad de una red. El problema surge en que «cualquiera» puede ser un intruso desde el exterior de la red. Para solventar este problema, la red debería estar aislada adecuadamente y proporcionar sólo información limitada a los usuarios externos. Como se puede ver, estos sistemas son armas de doble filo. Por una parte, permiten mejorar la seguridad de la red. Por otra, permiten que los intrusos puedan encontrar puntos débiles en la seguridad. Para tratar de limitar las desventajas y explotar los beneficios de estas utilidades, surgió COPS [Farmer, 1990], un sistema similar a Satán que sólo puede ser usado por los administradores del sistema y desde dentro del mismo.

9.2.7. Bombardeo

Un ataque de seguridad con mucho éxito en Internet es el que consiste en llevar a cabo bombardeos masivos con peticiones de servicio o de establecimiento de conexión a un servidor determinado. Estos ataques masivos provocan que el servidor deniegue sus servicios a los clientes legales y pueden llegar a bloquear al servidor. Este problema causó a principios del año 2000 el bloqueo de servidores de Internet famosos como *yahoo o altavista.* Para lograr que estos ataques tengan éxito, los atacantes se enmascaran con las direcciones e identidades de otros usuarios *(spoofing) y* llevan a cabo los ataques desde múltiples clientes.

Ataques similares pueden ocurrir en cualquier computadora, ya que todas ellas tienen recursos limitados. La tabla de procesos, por ejemplo, es crítica, ya que si se llena, la computadora no puede crear ni siquiera el proceso de apagar el sistema. Un usuario descuidado, o malicioso, que cree

procesos de forma recursiva y sin límite colapsará el sistema continuamente. Para evitarlo, los administradores deben poner límites, siempre que sea posible, a los recursos que cada usuario puede tener simultáneamente (número de procesos, impresoras, etc.).

POLÍTICAS DE SEGURIDAD

Los requisitos de seguridad son siempre una cuestión importante en las organizaciones. Por ejemplo, la ley obliga a mantener como privados los datos de los clientes de una empresa. Por ello, cuando se demuestra que se han difundido estos datos, se imponen multas a las compañías que violan la ley. Esto se debe generalmente a que el acceso a dichos datos no está controlado o a que las reglas de acceso son tan laxas que todos los compartimentos se encuentran entremezclados. Otro ejemplo interesante es el de la información reservada. Imagine que dos bancos se quieren fusionar. Conocer esta información a tiempo puede permitir hacer grandes negocios o evitar dicha fusión. Para evitar la difusión de la información, se mantiene en compartimentos pequeños que requieren una acreditación de acceso muy grande. La existencia de un sistema seguro pasa porque exista una **política de seguridad** [Neumann, 19901 que defina claramente la seguridad que proporciona el sistema, independientemente de los mecanismos usados para implementarla.

Es interesante resaltar que los requisitos de seguridad varían de unos sistemas a otros, e incluso entre usuarios distintos dentro del sistema. Imagine que una universidad tiene una computadora compartida entre alumnos y profesores, de forma que los alumnos no puedan tener acceso a Internet pero los profesores sí. El sistema operativo de esta computadora necesita una política de control de acceso a los recursos según el usuario que solicite dichos accesos. Por eso, cuando se instala un sistema operativo en una computadora con restricciones de seguridad, es necesario saber primero si la política de seguridad que ofrece dicho sistema operativo satisface los requisitos de seguridad de la instalación. Es decir, si es **confiable.** Ahora bien, un sistema operativo sólo es confiable respecto a una política de seguridad, es decir, a las características de seguridad que se esperen del sistema.

Existen distintas políticas de seguridad, todas las cuales describen políticas de seguridad generales para cualquier organización. A continuación, se estudian brevemente algunas de ellas, aplicándolas al campo de los sistemas operativos.

9.3.1. Política militar

Ésta es una de las políticas más popularmente conocidas y también de las más estrictas, por lo que casi nunca se aplica en su totalidad. Se basa en la clasificación de todos los objetos con requisitos de seguridad en uno de los cinco **niveles de seguridad** siguientes:

- Desclasificado.
- Restringido.
- Confidencial.
- Secreto.
- Alto secreto.

Y en clasificar también a los usuarios según el nivel al que pueden acceder [NCSC, 1985]. Según muestra la Figura 9.4, los cinco niveles se estructuran lógicamente como un conjunto de círculos concéntricos en cuyo interior está el alto secreto y en cuyo exterior están los documentos públicos (o desclasificados).

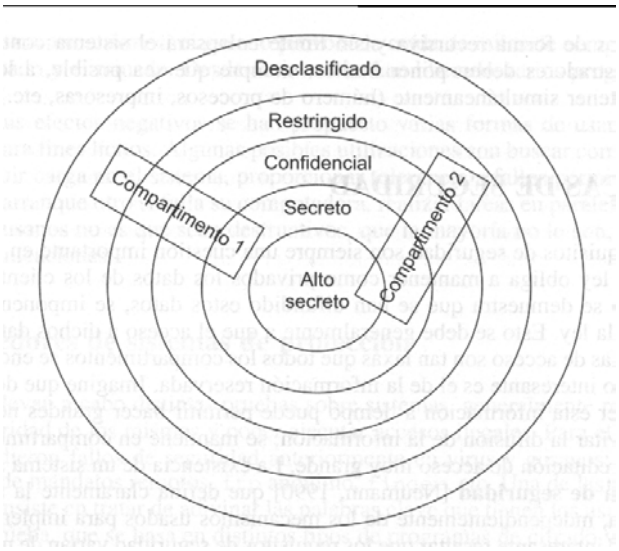


Figura 9.4. *Niveles de seguridad de la política militar.*

En general, los usuarios que tienen acceso a objetos de nivel i (p. ej.: secreto) también lo tienen a los de i+1 (confidenciales). Sin embargo, el acceso a la información se controla por la regla de **lo que** se **necesita saber.** Sólo se permite el acceso a datos sensibles a quien los necesita para hacer su trabajo. De esta forma se puede **compartimentar** a los usuarios, haciendo más estricta la regla general de acceso. Un **compartimento** se puede extender a varios niveles y dentro del mismo se aplica también la regla general de acceso. Usar compartimentos permite establecer conjuntos de usuarios que acceden a la información y ocultar documentos a usuarios con el mismo nivel de seguridad. Imagine que en el sistema operativo, el manejador de disco y el del teclado tienen la misma clasificación de seguridad. Con la regla general de acceso, cada uno de ellos tendría acceso a los datos internos del otro. Esta situación es típica en sistemas monolíticos, como UNIX, donde toda la información de nivel de núcleo es accesible a todos los componentes del núcleo. Sin embargo, parece claro que el manejador del teclado no tiene mucho que ver con el del disco y que, en un sistema bien diseñado, la información interna de uno debería estar oculta para el otro. Esta situación se da en sistemas orientados a objetos, como Windows NT, donde cada objeto define un compartimento de seguridad para su información interna.

Con esta política, cada pieza de información se debe **clasificar** usando la combinación <nivel, compartimento. Con los mismos criterios, hay que **acreditar** que un usuario puede tener acceso a un cierto nivel de seguridad dentro de su compartimento. Existe una relación de orden (<=), denominada **dominancia,** que se aplica sobre objetos (o) y sujetos (s) sensibles para definir la posibilidad de acceso o no.

S -^Cx^nivel <=nivel AND compartimento c compartimento

Usando esta relación se puede decidir que un sujeto puede acceder a un objeto sólo si su dominancia es mayor. Lo que equivale a decir que el sujeto debe estar suficientemente acreditado y que su compartimiento debe estar dentro de aquellos que tienen el acceso permitido al objeto. Imagine una empresa con dos secciones: personal y contabilidad. El presidente tiene acceso a

ambos y acreditación O, mayor que los jefes de dichas secciones (compartimentos) que tienen acreditación 1. El jefe de personal y el presidente tienen acceso a los expedientes de los trabajadores (clasificación nivel 1), pero el de contabilidad no. Su acreditación es suficiente, pero su sección no está dentro del compartimento de las que tienen acceso a esos documentos.

Una política similar a ésta se usó en el núcleo de seguridad del VAX, un núcleo que daba soporte de ejecución seguro a los sistemas operativos VMS y ULTRIX. El diseño de este núcleo tenía 16 niveles, cada uno de los cuales exportaba servicios a los niveles superiores e importaba los del nivel inmediatamente inferior. Dichos niveles incluían desde una simulación del hardware hasta la memoria virtual y la entrada/salida. La Figura 9.5 muestra los dominios de seguridad del sistema operativo VMS.

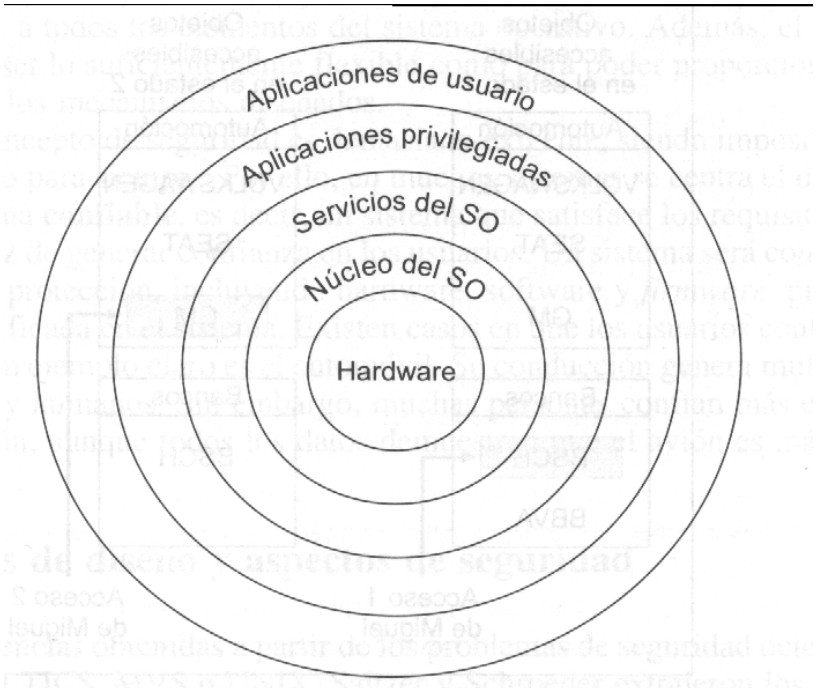
9.3.2. Políticas comerciales

Basándose en la política militar, pero generalmente debilitando las restricciones de seguridad, se han diseñado varias políticas de uso comercial. Algunos ejemplos de ellas son las de Clark-Wilson [Clark, 1987], separación de deberes [Lee, 1988] y [Nash, 1990] o la muralla china [Brewer, 1989]. Aunque son menos rígidas que la militar, todas ellas usan los principios de compartimentación de los usuarios y de clasificación de la información. Además definen reglas similares para el trasvase de información entre los distintos niveles y compartimentos.

Como ejemplo de política de seguridad en uso, se estudia brevemente la política de la muralla china.

La política de la **muralla china** [Brewer, 1989] clasifica a objetos y usuarios en tres niveles de abstracción:

- Objetos.
- Grupos.
- Clases de conflicto.



Cada objeto pertenece a un único gmpo y cada gmpo a una única clase de conflicto. Una clase de conflicto, sin embargo, puede incluir a varios grupos. Por ejemplo, suponga que existe información de tres fabricantes de automóviles (Volkswagen, Seat y General Motors) y dos bancos (BSCH y BBVA). Con la muralla china existen 5 grupos y 2 clases de conflicto (bancos y automóviles). La política de control de acceso es sencilla. Una persona puede acceder a la información siempre que antes no haya accedido a otro grupo de la clase de conflicto a la que pertenece la información a la que quiere acceder. La Figura 9.6 muestra el bloqueo de accesos para el caso en que un usuario, denominado Miguel, manipula información en ambas clases de conflicto. En su primer acceso, Miguel conoce información de la clase de conflicto Bancos, específicamente del BSCH. Eso le invalida para conocer más información de otros grupos de esa clase de conflicto. En el acceso 2 Miguel conoce información de la clase de conflicto Automoción, específicamente de General Motors. Por tanto, para posteriores accesos podrá acceder a ambas clases de conflicto, pero sólo a los grupos que ya ha accedido.

9.3.3. Modelos de seguridad

Un modelo es un mecanismo que permite hacer explícita una política de seguridad [Landwher, 1981]. En seguridad, los modelos se usan para probar la completitud y la coherencia de la política de seguridad. Además, pueden ayudar en el diseño del sistema y sirven para comprobar si la imple-mentación cumple los requisitos de seguridad exigida. Dependiendo de los requisitos, los modelos de seguridad se pueden clasificar en dos grandes tipos: multinivel [Bell, 1983] y limitados [Harri-son, 1985].

Los **modelos de seguridad multinivel** permiten representar rangos de sensibilidad y reflejar la necesidad de separar rigurosamente los sujetos de los objetos a los que no tienen acceso. Suelen ser modelos abstractos y muy generales, lo que los convierte en muy complejos, difíciles de verificar y muy costosos de implementar. Ejemplos de modelos de este estilo son los modelos de rejilla, el modelo de confidencialidad de Bell-La Padula y el modelo de integridad de Biba [Bell, 1983],

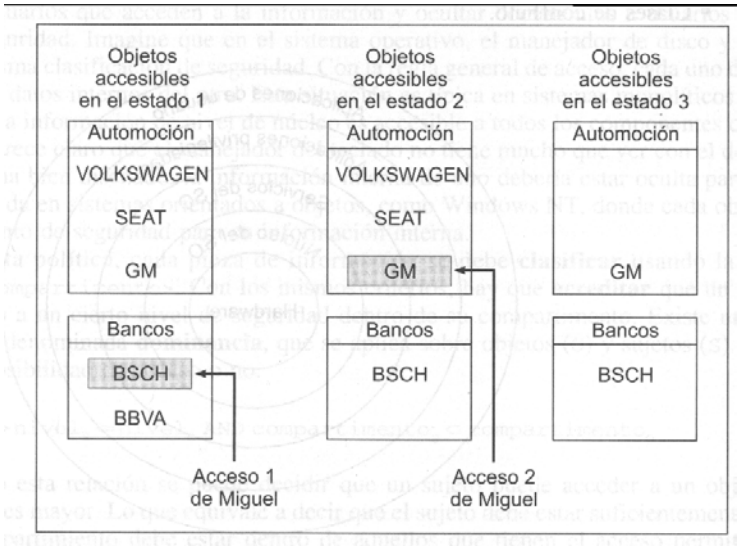


Figura 9.6. *Bloqueo de accesos por la política de la muralla china.*

Los **modelos de seguridad limitada** se centran en responder formalmente las propiedades que un sistema seguro debe satisfacer, pero introduciendo restricciones a los sistemas de seguridad multinivel. Todos ellos se basan en dos principios:

- Usan la teoría general de la computación para definir un sistema formal de reglas de protección.
- Usan una matriz de control de acceso, en cuyas filas están los sujetos y en cuyas columnas están los objetos.

Los derechos de acceso del sujeto i sobre el objeto j son los contenidos del elemento de la matriz (i, j). Ejemplos de modelos de este tipo son los de Graham-Denning, los de Harrison-Ruzzo-Hullman (HRU) y los de permiso de acceso [Harrison, 1985].

Es importante resaltar que la existencia de un modelo de seguridad no es requisito obligatorio en todos los sistemas operativos. Sin embargo, si se quiere demostrar a alguien que el sistema es confiable y seguro, tener un modelo con validación formal es una de las mejores formas de hacerlo. En los sistemas operativos con un nivel de seguridad muy alto, como el núcleo de seguridad del VAX, sí es obligatoria la existencia de un modelo de seguridad formalmente verificable. Los modelos que se usan en la mayoría de los sistemas operativos actuales se basan en modelos de seguridad limitada del tipo HRU. La implementación de una matriz de acceso y su explotación por filas o columnas es una de las formas más populares de implementar los mecanismos de protección en sistemas operativos de propósito general.

Se puede consultar bibliografía más específica de sistemas de seguridad, como [Landwher, 1981] o [Fites, 1989] para tener una explicación detallada de los modelos de protección.

9.4. DISEÑO DE SISTEMAS OPERATIVOS SEGUROS

Para dotar a un sistema operativo con mecanismos de seguridad es necesario diseñarlo para que admitan estos mecanismos desde el principio. Incluir mecanismos de seguridad dentro de un sistema operativo existente es muy difícil porque las repercusiones de los mecanismos de seguridad afectan prácticamente a todos los elementos del sistema operativo. Además, el diseño del sistema de seguridad debería ser lo suficientemente flexible como para poder proporcionar distintas políticas de seguridad con los mecanismos diseñados.

En general, el concepto de seguridad es demasiado exigente, siendo imposible diseñar y construir un sistema seguro para siempre. Por ello, en muchos sistemas se centra el diseño de seguridad en conseguir un sistema **confiable,** es decir, un sistema que satisface los requisitos de seguridad de terceros o que es capaz de generar confianza en los usuarios. Un sistema será confiable si el conjunto de mecanismos de protección, incluyendo hardware, software *yfirmware,* proporciona una política de seguridad unificada en el sistema. Existen casos en que los usuarios confían en sistemas no totalmente seguros. Un ejemplo claro es el automóvil. Su conducción genera multitud de accidentes por fallos mecánicos y humanos. Sin embargo, muchas personas confían más en su vehículo particular que en un avión, aunque todos los datos demuestran que el avión es más seguro.

9.4.1. Principios de diseño y aspectos de seguridad

Basados en las experiencias obtenidas a partir de los problemas de seguridad detectados en sistemas i operativos como MULTICS, MVS o UNIX, Saitzer y Schroeder extrajeron los criterios de diseño i siguientes para dotar a un sistema operativo de mecanismos de seguridad:

1. Diseño abierto. El diseño del sistema debería ser público para disuadir a posibles curiosos. Los mecanismos de protección no deben asumir la ignorancia de los posibles intrusos, reduciendo los aspectos ocultos al mínimo. Además, un diseño abierto puede ser estudiado y probado por cualquiera, lo que permite tener verificaciones independientes del sistema.

2. Exigir permisos. La política de acceso por defecto debería ser restrictiva, es decir, denegar el acceso. Es mejor identificar qué objetos son accesibles y cómo identificar los que no. De esta forma se tienen mecanismos para pedir permisos de acceso a todos los objetos.

3. Privilegios mínimos. Los procesos deberían tener la menor prioridad posible que les permita acceder a los recursos que necesitan. Se les debe conceder únicamente la prioridad necesaria para llevar a cabo su tarea. Este principio permite limitar los daños en casos de ataques maliciosos.

4. Mecanismos económicos. Los mecanismos de protección deberían ser sencillos, regulares y pequeños. Un sistema así se puede analizar, verificar, probar y diseñar fácilmente.

5. Intermediación completa. Cada intento de acceso al sistema debe ser comprobado, tanto los directos como los indirectos. Los mecanismos de comprobación deberían estar situados en zonas del sistema operativo donde no puedan ser evitados. Lo lógico es que estén ocultos en los niveles inferiores del sistema. Si se relaja este principio de diseño de forma que no se comprueben todos los accesos a un objeto, es necesario comprobar los permisos de los usuarios de forma periódica y no sólo cuando se accede al recurso por primera vez.

6. Compartición mínima. Los objetos compartidos pueden servir como canales de comunicación de información. Los sistemas que usan la separación física o lógica de los usuarios permiten reducir el riesgo de compartición ilegal de objetos. La memoria virtual, por ejemplo, permite separar los espacios de memoria de cada objeto y controlar los accesos a memoria con gran detalle. La separación debe aplicarse cuidadosamente cuando se asignan a un usuario objetos libres que antes fueron de otro usuario, para evitar que se conviertan en canales de comunicación encubiertos. Los archivos o páginas de memoria usados por un usuario y liberados posteriormente deben ser borrados físicamente para que otros usuarios no puedan reusar dicha información por accidente o por mala fe. Imagine, por ejemplo, que un usuario crea un archivo, lo escribe y lo borra. A continuación, otro usuario crea un archivo y pide bloques pero no escribe. Es muy probable que se le den bloques del usuario anterior con toda la información que contenían. Este fallo de seguridad ocurría en las primeras versiones de UNIX.

7. Fáciles de usar y aceptables. El esquema de protección debe ser aceptado por los usuarios y fácil de usar. Por ejemplo, un análisis de DNA sería muy fiable para identificar al usuario, pero el lector puede entender que los usuarios de una computadora no serían muy felices si se tuviesen que hacer un análisis de sangre cada vez que quieren entrar al sistema. Si un mecanismo es sencillo, y no es desagradable, existen menos probabilidades de que los usuarios traten de evitarlo.

8. Separación de privilegios. Si se quiere diseñar un sistema seguro, los accesos a cada objeto deben depender de más de un mecanismo de protección. De esta forma, si se consigue violar uno de ellos, todavía quedarán los restantes mecanismos de protección. Tener la información criptografiada añade un plus de seguridad, ya que aunque un intruso pueda acceder a la información todavía debe ser capaz de descifrarla.

Todo lo descrito anteriormente necesita ser implementado en el sistema operativo para proporcionar seguridad a los usuarios, por lo que el diseño de los aspectos de seguridad de un sistema operativo es una tarea delicada [Bell, 19831. Es necesario elegir un conjunto apropiado y coherente

de características a satisfacer y fijar cuidadosamente el grado de fiabilidad con que se implementan, e incluyen en el sistema, las características anteriores [DeLashmutt, 1979].

En general, un sistema operativo multiprogramado lleva a cabo las siguientes **tareas relacionadas con la seguridad del sistema,** debiendo satisfacer unas características de seguridad mínimas para poder ser confiable:

• **Autenticación de usuarios.** Si hay que controlar los accesos basándose en la identidad individual de cada potencial usuario del sistema, dichas identidades deben ser precisas. La existencia de identidad conlleva la necesidad de autenticar, o verificar, esa identidad. En un sistema operativo debe existir un método seguro y fiable de identificar a los usuarios y cada usuario debe tener una identidad única. Además, en caso de que un intruso consiga acceder al sistema, es necesario detectar dicha violación. Esta característica todavía no existe en muchos sistemas operativos.

• **Asignación de recursos.** Los objetos de un sistema operativo deben ser asignados a los procesos que los solicitan sin que haya conflictos, violaciones de normas de seguridad, problemas de compartición o fallos de confidencialidad a través de recursos reusables. Las páginas de memoria, por ejemplo, pertenecen a un proceso o a otro dependiendo del estado de la memoria física. La reutilización de recursos puede originar fallos de seguridad, que pueden ser maliciosos o no. Para evitar este problema, el sistema operativo debe borrar la información existente dentro de un objeto antes de permitir que otro usuario lo utilice.

• **Control de accesos a los recursos** del sistema. El dueño de un objeto tiene un cierto gradcC de libertad para establecer quién y cómo puede acceder a un objeto. Además, esta situación puede cambiar dinámicamente. Los sistemas operativos de propósito general usan este tipo de políticas discrecionales. Los seguros usan políticas con controles obligatorios.

• **Control de la compartición y la comunicación** entre procesos. Para que se puedan aplicar controles de acceso, hay que controlar todos los accesos, incluyendo la comunicación entre procesos y la reutilización de elementos. La complejidad del sistema de seguridad aumenta a medida que existen más formas de acceder al sistema (memoria, archivos, puertos, redes, etcétera).

• **Protección de los datos** del sistema de protección en sí mismo. Los sistemas operativos almacenan la información de seguridad en recursos internos del sistema. Es necesario tener estos recursos bien protegidos para que el acceso a los mismos no facilite violaciones de seguridad. En versiones antiguas de UNIX, el archivo de usuarios con las claves cifradas era legible para todo el mundo. A través de este archivo se podía conocer las identidades de los usuarios y tratar de romper sus claves. Además, muchos sistemas operativos registran todos los eventos que son relevantes para la seguridad, tales como accesos, fallos de acceso, cambios de protecciones, etc. Este registro se lleva a cabo en uno o más archivos, que deben estar protegidos de accesos no autorizados. La mayoría de los sistemas registran el primer y último acceso a un objeto, pero aun así los registros de eventos suelen ser grandes y difíciles de analizar.

La Figura 9.7 relaciona las tareas de seguridad anteriores con las funciones más tradicionales del sistema operativo. Como se puede ver, el control de acceso es fundamental para servicios, ; utilidades y recursos físicos. La autenticación de usuarios está intrínsecamente relacionada con la *[•:* interfaz de usuario en un sistema operativo tradicional. Aunque en sistemas conectados a redes I también es necesario autenticar a usuarios que acceden a través de la red. La asignación de recursos ; afecta al reparto de tiempo de la UCP, espacio en dispositivos de E/S, espacio en memoria, etc. [s Además de estas funciones, el sistema operativo debe controlar que la compartición de recursos del s sistema sea segura.

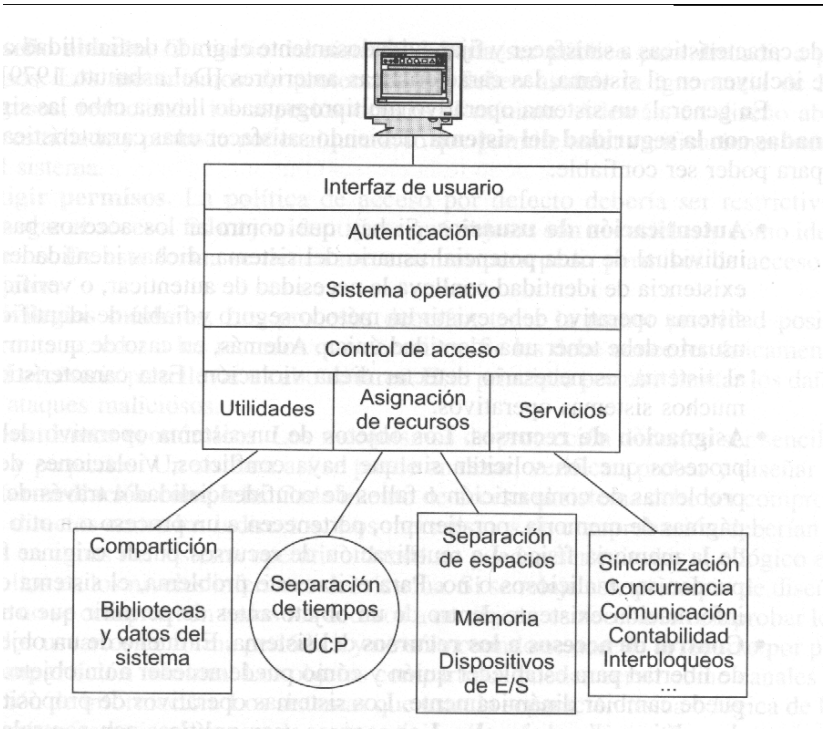


Figura 9.7. *Tareas de seguridad y componentes del sistema operativo.*

9.4.2. Técnicas de diseño de sistemas seguros

Existen distintas técnicas de diseño que se pueden usar para dotar a un sistema operativo con los principios y características de seguridad descritas anteriormente. En esta sección se describen las tres principales:

- Separación de recursos.
- Uso de entornos virtuales.
- Diseño por capas.

Separación de recursos

Una de las formas más seguras y eficaces de evitar problemas de seguridad es separar los recursos de los distintos usuarios o dominios, de forma que no puedan compartirlos o que la forma de compartirlos esté completamente controlada a través de un medio de comunicación fiable. Hay cuatro formas básicas de separación entre procesos:

• **Física.** Los procesos ejecutan en distintas plataformas hardware, dependiendo de sus restricciones de seguridad. Por ejemplo, se puede instalar el software nuevo en una máquina aislada para prevenir la extensión de virus o ejecutar los procesos con restricciones fuertes de seguridad en entornos de computación restringidos. Es importante resaltar que gran parte del hardware usado actualmente proporciona separación física para recursos tales como memoria, ejecución de procesos ligeros, sistema operativo y aplicaciones [Wiikes, 1982].

- **Temporal.** Ocurre cuando los procesos se ejecutan a distintas horas. Algunos sistemas operativos, como el VMS de DEC, permiten especificar a qué horas puede ejecutar un proceso, cuándo se puede acceder a un sistema desde un terminal, etc. En sistemas grandes se permite ejecutar trabajos interactivos durante una ventana horaria (p. ej.: de 9 a 18 horas) y a partir de esa hora sólo se permite la ejecución de trabajos por lotes *(batch).*
- **Criptográfica.** Usa la criptografía para asegurar que los datos de distintos usuarios no sean inteligibles para los demás, aunque puedan acceder a dichos datos.
- **Lógica.** Los sistemas proporcionan múltiples espacios lógicos de ejecución, asignando uno a cada proceso. El sistema operativo separa así los objetos y el contexto de un usuario de los de otro.

Los sistemas operativos multiprogramados proporcionan separación física y lógica, aislando cada proceso de los otros y permitiendo únicamente la comunicación a través del sistema operativo. Cada proceso, por ejemplo, tiene su espacio de memoria virtual y su propio árbol de archivos y directorios. Para evitar una rigidez excesiva, en algunos sistemas operativos, como UNIX, se dejan canales de comunicación con menos control. Por ejemplo, si un usuario quiere dar un archivo a todos los demás sólo tiene que copiarlo al directorio /tmp, al que puede acceder cualquier usuario para leer y escribir. Las separaciones temporales o criptográficas, en caso de estar disponibles, suelen ser discrecionales del administrador o incluso del propio usuario.

Uso de entornos virtuales

Este mecanismo es muy importante para diseñar sistemas operativos seguros, porque permiten proporcionar separación lógica de forma natural. Obviamente, los entornos virtuales deben apoyarse en recursos reales, pero el sistema operativo puede proporcionar a los usuarios los recursos virtuales y controlar el acceso a los mismos. Ejemplos de mecanismos virtuales importantes son:

- Los espacios múltiples de memoria virtual.
- Las máquinas virtuales.

Las máquinas virtuales proporcionan un entorno virtual completo para cada usuario. De esta forma, cada usuario debe acceder a los recursos reales a través de su máquina virtual. Las máquinas virtuales permiten mejorar el aislamiento entre los usuarios, si bien incrementan la sobrecarga en las llamadas al sistema. Un buen ejemplo de uso de máquinas virtuales se puede encontrar en el sistema operativo MVS de IBM [Seawright, 19791, cuya estructura se muestra en la Figura 9.8. Como puede verse en la figura, cada máquina virtual dota a cada usuario de un subconjunto, más o menos completo, de características del hardware, que puede ser muy distinto de la implementación física real del mismo.

Cuando un usuario accede al sistema operativo MVS se le asigna una máquina virtual. Este entorno incluye una emulación completa del hardware que, posteriormente, se proyecta sobre el hardware real. De esta forma, cada usuario sólo puede acceder a su máquina virtual. Además de proporcionar aislamiento, las máquinas virtuales permiten limitar el acceso de los usuarios a los recursos que realmente puedan necesitar. De esta forma se puede reducir sensiblemente el rango de objetos a proteger por el sistema operativo.

Diseño por capas

La visión de un sistema seguro se puede asemejar a una serie de círculos concéntricos, donde los niveles interiores son de más confianza. Está visión se puede plasmar de forma precisa en un sistema por capas, donde cada capa sólo tiene acceso a las capas adyacentes.

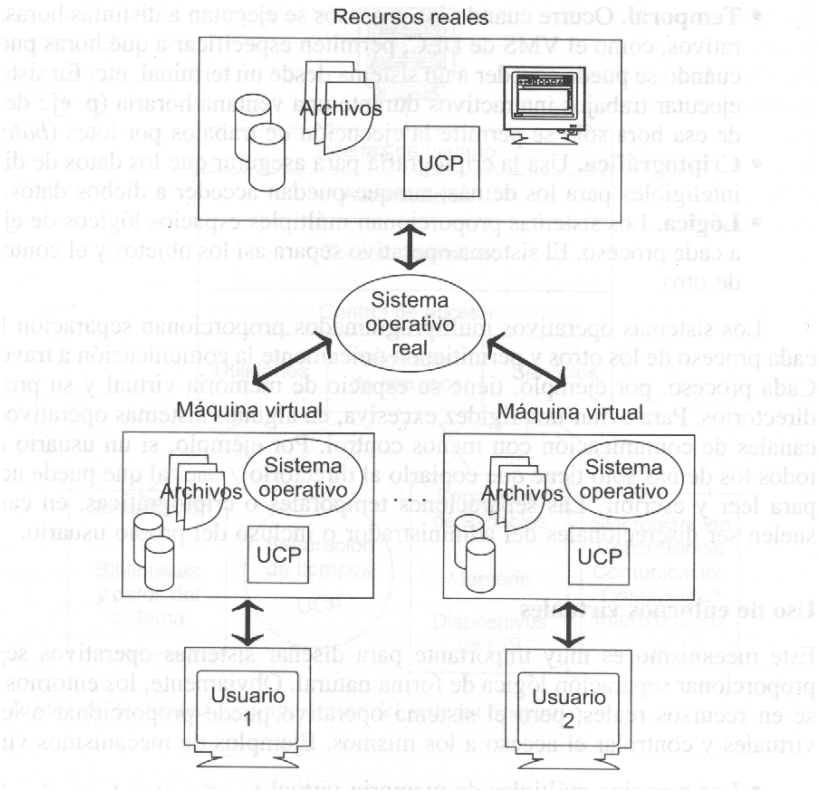


Figura 9.8. Máquinas virtuales en el sistema operativo MVS.

La Figura 9.9 muestra un sistema por capas y, dentro de él, cómo afectan a cada nivel las tareas de seguridad. Este diseño permite relajar el concepto de núcleo de seguridad, repartiendo sus funciones por las distintas capas del sistema, ocultar datos entre niveles y reducir daños en caso de fallos de seguridad. En cada capa del sistema se puede incluir toda la funcionalidad que afecta a los elementos de esta capa. En caso de que haya un fallo de seguridad, sólo esos elementos se verán afectados. Imagine lo que supondría la existencia del mismo fallo de seguridad dentro de un núcleo monolítico. El fallo podría afectar a todo el sistema sin limitación.

Ejemplos de diseño por capas de sistemas seguros son MULTICS y la versión de seguridad del sistema operativo VMS. En VMS, cada anillo del sistema es un dominio de ejecución, siendo el núcleo la capa más interna (Fig. 9.10). Los anillos se implementan como bandas concéntricas alrededor del hardware, de tal forma que los procesos más fiables ejecutan en niveles internos, mientras los de usuario se quedan en las capas extemas. Los anillos se solapan, de forma que | ejecutar en un determinado anillo significa tener los privilegios de ese nivel y los de todos los | anillos más externos. El núcleo es la parte del sistema operativo que ejecuta las operaciones del nivel más básico, tales como comunicación, planificación, gestión de interrupciones, etc. Elnúcte@| es además responsable de los servicios de seguridad para todo el sistema operativo, proporcionand&J interfaces seguras para el hardware y las partes restantes del sistema operativo.

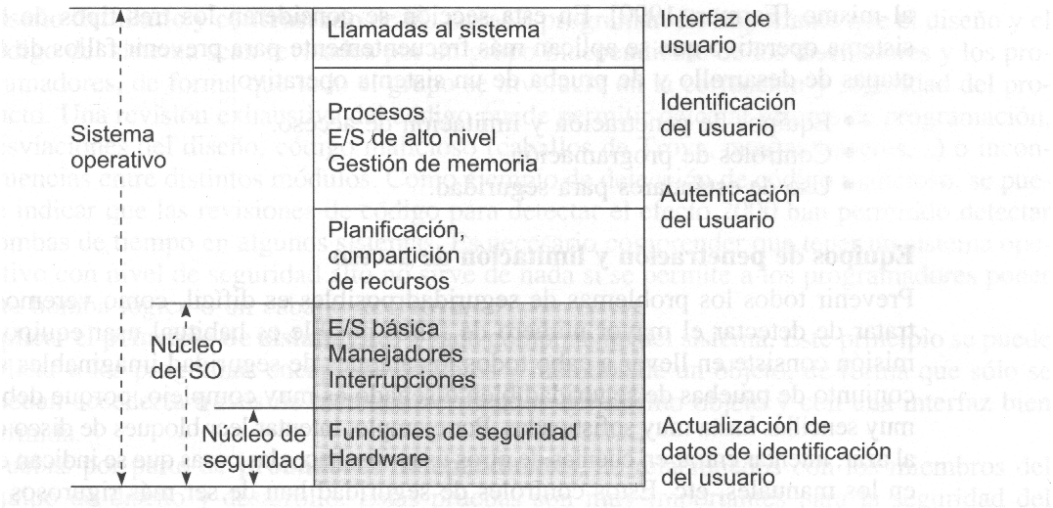

Sistema operativo
Núcleo del SO
Núcleo de seguridad
Llamadas al sistema
Procesos
E/S de alto nivel
Gestión de memoria
Planificación, compartición de recursos
E/S básica
Manejadores
Interrupciones
Funciones de seguridad
Hardware
Interfaz de usuario
Identificación del usuario
Autenticación del usuario
Actualización de datos de identificación del usuario

Figura 9.9. *Diseño por capas y tareas de seguridad.*

9.4.3. Controles de seguridad externos al sistema operativo

En las secciones anteriores se han mostrado los criterios de diseño de un sistema seguro. Obviamente, no existirían fallos de seguridad si se pudieran prevenir. Sin embargo, la prevención supone hacer frente a un problema de gran complejidad, por lo que ningún sistema aporta una solución total

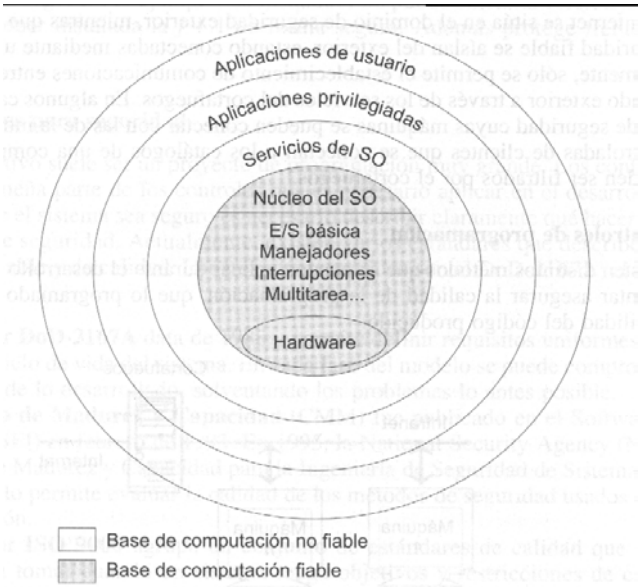


Figura 9.10. *Dominio de seguridad de VMS.*

al mismo [1990]. En esta sección se consideran los tres tipos de controles externos al sistema operativo que se aplican más frecuentemente para prevenir fallos de seguridad durante las etapas de desarrollo y de prueba de un sistema operativo:

• Equipos de penetración y limitación de acceso.
• Controles de programación.
• Uso de estándares para seguridad.

Equipos de penetración y limitación de acceso

Prevenir todos los problemas de seguridad posibles es difícil, como veremos más adelante. Para tratar de detectar el mayor número de fallos posible es habitual usar equipos de penetración. Su misión consiste en llevar a cabo todos los ataques de seguridad imaginables sobre un sistema. Un conjunto de pruebas de seguridad bien diseñado es muy complejo, porque debe incluir desde cosas muy sencillas hasta muy sofisticadas. Por ejemplo, intentar leer bloques de disco o páginas de memoria al azar, intentar entrar en cuentas de otros usuarios, hacer las cosas que se indican como no convenientes en los manuales, etc. Estos controles de seguridad han de ser más rigurosos si los sistemas estái conectados en redes por la posibilidad existente de difusión de virus o de intento de adquisición de palabras de acceso al sistema mediante programas que descifran dichos códigos (crackers).

La complejidad de las comprobaciones y el registro de los accesos aumenta en los sistemas conectados a la red. En este caso, la seguridad del sistema se enfrenta a múltiples puntos de ejecu ción y a canales de comunicación expuestos. Debido a la complejidad de los mecanismos de protec ción en sistemas distribuidos, en muchas redes de computadoras se limita el acceso y sólo se permi te el acceso a la red interna a través de una máquina determinada, denominada cortafuegos (firewall). Esta máquina separa dos dominios de seguridad: el fiable y el exterior. El cortafuegos se sitúa entre ambos y filtra todo el tráfico de la red, monitoriza y registra las conexiones. Actualmen te, Internet se sitúa en el dominio de seguridad exterior, mientras que las máquinas del dominio de seguridad fiable se aíslan del exterior, estando conectadas mediante una intranet (Fig. 9.11). Habi tualmente, sólo se permite el establecimiento de comunicaciones entre máquinas de la intranet y el mundo exterior a través de los servicios del cortafuegos. En algunos casos existe una zona interme dia de seguridad cuyas máquinas se pueden conectar con las de la intranet. Por ejemplo, máquinas controladas de clientes que se conectan a los catálogos de una compañía usando protocolos que pueden ser filtrados por el cortafuegos.

Controles de programación

Existen distintos métodos que se pueden aplicar durante el desarrollo de un sistema operativo para intentar asegurar la calidad de la programación, que lo programado se ajusta a lo diseñado y la fiabilidad del código producido:

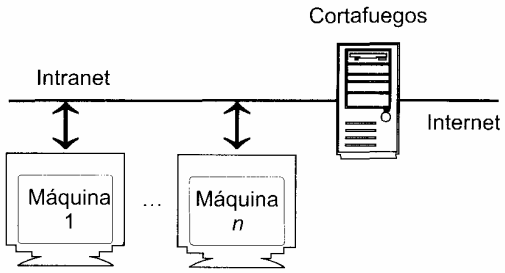


Figura 9.11. *Limitación de accesos mediante un cortafuegos.*

• **Diseño detallado y contrastado** de las tareas a programar. Es importante que el diseño y el código del sistema sean revisados por un grupo independiente de los diseñadores y los pro gramadores, de forma que todo el grupo se involucre en la corrección y seguridad del pro ducto. Una revisión exhaustiva del código puede permitir detectar errores de programación, desviaciones del diseño, código malicioso (caballos de Troya, puertas traseras,...) o incon gruencias entre distintos módulos. Como ejemplo de detección de código malicioso, se pue de indicar que las revisiones de código para detectar el efecto 2000 han permitido detectar bombas de tiempo en algunos sistemas. Es necesario comprender que tener un sistema ope rativo con nivel de seguridad alto no sirve de nada si se permite a los programadores poner una bomba lógica o un caballo de Troya.
• Aplicar el **principio de aislamiento** a cada componente del sistema. Este principio se puede aplicar a los programas encapsulando datos y métodos de un objeto, de forma que sólo se puedan acceder a través de métodos verificados del mismo objeto y con una interfaz bien definida.
• Pruebas por parte de **probadores independientes**, no relacionados con los miembros del equipo de diseño y desarrollo. Estas pruebas son muy importantes para la seguridad del sistema, porque si un programador quiere introducir código malicioso en el sistema nunca desarrollará código de prueba que detecte dicho código. En los sistemas operativos abiertos y de distribución gratuita, como LINUX, se suelen detectar y resolver antes los problemas de seguridad debido a dos razones: gran número de probadores y gran número de modificado res que estudian detalladamente el código.
• **Gestión de configuración**, de forma que cualquier cambio o instalación de software en un sistema debe ser aprobado por el administrador que debe juzgar la necesidad del cambio. Este control protege frente a amenazas no intencionadas, al evitar que el sistema quede en un estado inseguro. Por ejemplo, un instalador no puede borrar la versiónj de un programa antes de tener instalada la j + 1 de forma segura. Además protege frente a errores ma liciosos.

Uso de estándares para seguridad

Un sistema operativo suele ser un proyecto de programación muy grande. Los controles anteriores son sólo una pequeña parte de los controles que es necesario aplicar en el desarrollo del sistema. Además, para que el sistema sea seguro es necesario describir claramente qué hacer, y cómo hacer lo, en términos de seguridad. Actualmente, existen varios estándares que describen cómo conse guir un sistema fiable y de calidad. Los tres más conocidos son el DoD-2167A, el SSE-CMM y el ISO 9000.

• El estándar **DoD-2167A** data de 1988 y permite definir requisitos uniformes aplicables a lo largo del ciclo de vida del sistema. En cada fase del modelo se puede comprobar la calidad y seguridad de lo desarrollado, solventando los problemas lo antes posible.
El **Modelo de Madurez y Capacidad** (CMM) fue publicado en el Software Engineering Institute (SEI) en febrero de 1993. En 1995, la National Security Agency (NSA) publicó el Modelo de Madurez y Capacidad para la Ingeniería de Seguridad de Sistemas (SSE-CMM). Este modelo permite evaluar la calidad de los métodos de seguridad usados en un sistema u organización.
• El estándar **ISO 9000** agrupa un conjunto de estándares de calidad que especifican las acciones a tomar cuando un sistema tenga objetivos y restricciones de calidad. Uno de ellos, el 9000-1, se aplica al desarrollo de software e identifica unos requisitos de calidad mínimos.

Digitalización realizada con propósito académico

9.4.4. Controles de seguridad del sistema operativo

El sistema operativo debe llevar a cabo su parte de la política de seguridad del sistema liJones, 19781. Algunos de los controles de seguridad que se deben aplicar en el sistema operativo son:

- Ejecutar software fiable.
- Sospechar de los procesos.
- Ejecutar los procesos con confinamiento.
- Registrar los accesos.
- Buscar periódicamente agujeros de seguridad.

El software fiable es aquel que ha sido rigurosamente desarrollado y analizado, de forma que se puede confiar en que hará lo que se espera y nada más. Típicamente, el software fiable es la base sobre la que se ejecutan aplicaciones no fiables. Un sistema operativo debe ser fiable y, por tanto, se puede usar para que los programas de usuario ejecuten operaciones sensibles sin acceder a datos sensibles. Tanto las bibliotecas de interfaz del sistema operativo como los generadores de código deben ser fiables.

La sospecha mutua es un concepto que se desarrolló para describir las relaciones entre dos procesos. Los sistemas que sospechan se ejecutan como si los otros procesos fueran maliciosos. Por ello, en todos los módulos se aplica el encapsulamiento y la ocultación de la información. El sistema operativo debe sospechar de todos los procesos que se ejecutan sobre él. Sin embargo, existen muy pocos sistemas operativos que apliquen este criterio de diseño con sus componentes internos, por lo que es necesario aplicar controles de programación muy estrictos durante el desa rrollo de este tipo de sistemas. Una técnica típica de sospecha es monitorizar los procesos para ver si tienen patrones de ejecución sospechosos.

El confinamiento es una técnica usada por los sistemas operativos para tratar de reducir los daños en caso de que existan fallos de seguridad o código malicioso. Un proceso confinado tiene estrictamente limitados los recursos del sistema a los que puede acceder. Este principio es muy útil para proteger el sistema ante la existencia de virus, ya que, si se aplica estrictamente, el virus sólo puede dañar el compartimento al que tiene acceso. La separación de dominios es una técnica de seguridad muy frecuente en sistemas operativos.

El registro de accesos origina un listado de los usuarios que acceden a los objetos del sistema, especificando cuándo y cómo se han realizado dichos accesos. En un sistema operativo de propósi to general se suelen registrar las entradas y salidas al sisterha y los accesos a ciertos objetos. En un sistema seguro se registran muchos más eventos. En cualquier caso, es fundamental registrar los fallos de acceso a cualquier tipo de objetos, puesto que pueden revelar la presencia de intrusos. Asimismo, cuando se pruebe un programa nuevo sería conveniente registrar los eventos que origina para comprobar que no realiza acciones indebidas. Sobre el registro de accesos se pueden efectuar auditorías periódicas para tratar de detectar cualquier tipo de fallos de seguridad, como por ejemplo el uso de cerrojos sobre un archivo para crear un canal encubierto.

Una vez en funcionamiento, el sistema operativo tiene que comprobar que no hay intentos de violación de la seguridad en el sistema. Algunas de las comprobaciones a realizar son: palabras clave cortas o muy sencillas, programas con prioridad indebida, programas con un identificador de usuario indebido, programas no autorizados en directorios del sistema, etc. Asimismo, el sistema operativo debe registrar los accesos a los recursos por parte de los usuarios. Cuando se ha producido un problema de seguridad, se puede usar el registro de accesos para saber quién lo ha producido, facilitar la recuperación y para prevenir problemas futuros.

Los mecanismos usados dentro del sistema operativo para satisfacer los principios anteriores se estudian con más detalle en la Sección 9.7.

Digitalización realizada con propósito académico

9.5. CRIPTOGRAFÍA

En las secciones anteriores se ha mencionado varias veces la palabra criptografía en relación con la seguridad. Suponga que además de ocultar información y controlar los accesos a la misma se quiere hacer dicha información ininteligible en caso de que sea accedida. La criptografía es la técnica que permite codificar un objeto de manera que su significado no sea obvio. Es un medio para mantener datos seguros en un entorno inseguro, en el cual un objeto original (O) se puede convertir en un objeto cifrado (C) aplicando una función de cifrado (E). Obviamente, es necesario que se pueda descifrar el mensaje cifrado, para volver a obtener el mensaje original aplicando una función de descifrado (D), que puede ser o no la inversa de E. La Figura 9.12 muestra el proceso de cifrado y descifrado de un mensaje.

9.5.1. Conceptos básicos

Existen dos conceptos básicos en criptografía: algoritmos de cifrado y claves. Ambos conceptos no están indisolublemente ligados, puesto que, habitualmente, un algoritmo de cifrado admite muchas claves y una clave podría servir para muchas funciones de cifra. A continuación, se estudian breve mente ambos conceptos. Los lectores interesados en profundizar en el tema pueden leer los libros de Denning [1992] y de Pfleeger [1997].

Algoritmos de cifrado

Las funciones de cifrado y descifrado permiten cifrar y descifrar un objeto mediante la aplicación al mismo de procedimientos, generalmente repetitivos, que permiten ocultar el contenido del objeto y ponerlo en su forma original, respectivamente.

Los algoritmos de cifrado son muchos y variados. En general, todo diseñador de un sistema de criptografía busca algoritmos nuevos y mejores que los existentes. Sin embargo, la mayor parte de los algoritmos convencionales de cifrado se pueden clasificar dentro de dos tipos: sustitución, es decir, cambiar el contenido del objeto original por otro, y transposición, es decir, modificar el orden del contenido del objeto original liDenning, 1982].

Los algoritmos que se basan en la sustitución de partes del texto original por otros textos del mismo o de otro alfabetos tienen como objetivo básico aumentar la confusión. Se pueden dividir en dos grandes tipos según la complejidad del algoritmo:

• **Monoalfabéticos.** Cambian cada carácter por otro carácter o símbolo. Son muy sencillos y se conocen desde antiguo. Los griegos y persas ya codificaban mensajes usando esta técnica. Sin embargo, el primer algoritmo con nombre es el denominado Julio César, porque se dice que fue el primero en usarlo. Es muy sencillo, pero efectivo: cada letra del original se cambia por la que está un cierto número de posiciones más adelante. César usaba un despla zamiento de tres caracteres. Ejemplo: abc -> cde.

• **Polialfabéticos**. Cambian grupos de caracteres por otros caracteres o símbolos depen diendo de la frecuencia de su distribución en el texto original [19661. Ejemplo:

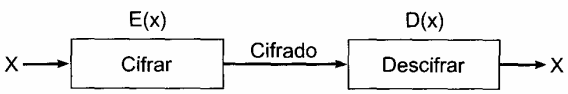


Figura 9.12. *Proceso de cifrado de un mensaje.*

Digitalización realizada con propósito académico

abc -> 112233. El objetivo de estos algoritmos es evitar un defecto de los monoalfabétic que hace aparente la distribución del alfabeto del objeto original a partir del estudio d objeto codificado. El código Morse es un ejemplo muy sencillo de algoritmo polialfabéti en el que cada carácter se sustituye por una secuencia de puntos y líneas.

Los algoritmos de transposición o permutación reordenan la estructura interna del obje original para obtener un objeto cifrado cuya estructura no es aparente. Los algoritmos más popul res de este tipo son los de transposición por columnas, que reordenan el objeto en columnas forma que un cierto número de elementos (número de columnas) se agrupan y luego se reorden aplicando el algoritmo de cifrado (p. ej.: transposición de columnas). Como ejemplo, el nomb “Julio Cesar”, organizado en 4 columnas y transpuesto sería:
Juli o Ce sar “Josu alCrie”

Además de la clasificación anterior, los algoritmos de cifrado se pueden clasificar atendien al conjunto de datos sobre el que trabajan. Existen dos tipos básicos:

• Flujo de caracteres. Convierten la entrada inmediatamente en código cifrado. La transfc mación depende únicamente del símbolo de entrada, la clave y el algoritmo de cifrado. S rápidos y tienen muy poca propagación de error. Sin embargo, son susceptibles de suf modificaciones o inserciones en línea. El código Morse es de este tipo.
• Bloques. Trabajan sobre un bloque completo del objeto original. Una transposición p columnas es un ejemplo de este tipo de algoritmos. Generan mucha difusión y son inmune inserciones en el bloque. Sin embargo, tienen el problema de ser lentos y de propagar po:
bles errores a todo el bloque. Hay sistemas de archivos que codifican bloques antes’ guardarlos en disco.

Con el incremento de la capacidad de cómputo que se produjo en la década de los setenta abordaron problemas de criptografía más complejos. Como resultado de este trabajo, a finales esa década se publicaron tres nuevos algoritmos basados en la realización de cálculos masivos p computadora:

• El de **Merkle-Hellman**, que se basaba en el uso de un conjunto de enteros positivos y u suma objetivo. Es un algoritmo NP-completo, por lo que el tiempo de solución es expone cial al número de enteros usados. Desafortunadamente, a principios de la década de 1 ochenta se encontró una forma de romper este algoritmo de cifrado.
• El **RSA** [19781, que fue inventado en 1978 y ha sido seguro hasta la fecha. Tiene 1 mismos fundamentos matemáticos que el anterior pero incorpora resultados de la teoría grandes números y la determinación de números primos. Su implementación software pi porciona un rendimiento muy pobre, por lo que es muy frecuente el uso de hardware q lleva a cabo el cifrado con este tipo de algoritmo.
• El **Data Encryption Standard** (DES) fue desarrollado por el U. S. National Institute Standards and Technology para su uso en aplicaciones comerciales [1977]. Este al ritmo combina sustitución y transposición, aplicando ambas repetidamente con un total 16 ciclos. Para ello usa únicamente aritmética estándar y operaciones lógicas, lo que perm su implementación en cualquier computadora. A pesar de la complejidad de trazar un bi través de 16 iteraciones de sustituciones y transposiciones, el DES ya no es un algoritr seguro. Además ha sido cuestionado por tener debilidades en su diseño, claves muy coi (56 bits) y por posibilitar la existencia de puertas traseras que permitirían a la Natioi Security Agency (NSA) descifrar cualquier objeto cifrado con este algoritmo. A pesar todo es un algoritmo muy popular y varias versiones del mismo son ampliamente usadas

Digitalización realizada con propósito académico

la actualidad. Además, para reforzar la seguridad del algoritmo se ha incrementado la longi tud de la clave.

En 1993, el gobierno de los Estados Unidos de América anunció el programa Clipper, que incluye un conjunto de algoritmos de criptografía conocido como Skipjack. El algoritmo se mantie ne en secreto, pero se sabe que se basa en un nuevo concepto de cifrado denominado escrutinio de claves jiDenning, 19961. La clave se divide en varios componentes, cada uno de los cuales está en poder de una agencia autorizada. Todos los componentes son necesarios para descifrar un objeto, de manera que es necesario pedirlos a las respectivas agencias autorizadas. Este método, aunque muy seguro, fue muy mal recibido en su momento porque las agencias autorizadas eran del gobierno y hubo muchas quejas respecto al control gubernamental de las comunicaciones privadas.

Claves

Un concepto básico en criptografía es el de clave. La clave (k) es el patrón que usan los algoritmos de cifrado y descifrado para manipular los mensajes en uno u otro sentido. Aunque existen sistemas criptográficos que no usan clave (keyless cipher), el uso de claves añade más seguridad a los meca nismos de cifrado porque con distintas claves se pueden obtener distintos mensajes cifrados usando la misma función de cifrado. De esta forma, para romper un sistema de cifrado es necesario conocer tanto las funciones correspondientes como la clave usada para cifrar un determinado objeto.

Obviamente, es necesario tener sistemas de criptografía en los que siempre se pueda recuperar el objeto original a partir del objeto cifrado (o=D (k, E (k, O))). Sin embargo, dependiendo de los pasos a aplicar para ejecutar el proceso de la Figura 9.12, los sistemas de criptografía se pueden clasificar en dos tipos básicos:

• **Simétricos**. En este caso, D es la función inversa de E y las clave usada en ambos casos es la misma. También se denominan sistemas de clave privada. Puesto que ambos comparten la clave, pueden trabajar de forma independiente. La simetría del proceso es muy útil siempre que se mantenga e1 secreto, ya que en ese caso se puede usar la clave como medio de autenticación. Sin embargo, estos métodos tienen problemas asociados con la filtración de las claves, su distribución, su debilidad criptográfica y el número creciente de claves.
• **Asimétricos**. En este caso existen claves distintas para cifrar y descifrar y la función de descifrado no es exactamente inversa a la de cifrado (o=D (Kd, E (Ke, O))). La asimetría permite reducir el número de claves a intercambiar entre los participantes en el proceso de cifrado.

Todos los sistemas que usan claves son sensibles a la pérdida, filtración o robo de claves [19941. La única solución posible para este problema es que cuando un usuario sospeche que una clave ha sido interceptada lo notifique a quien corresponda para que éste a su vez cambie la clave y notifique que, a partir de un determinado instante, los mensajes codificados con la clave anterior no son válidos. Este método es el mejor posible, aunque no es óptimo. Si un usuario tiene objetos cifrados con una clave y le notifican que es inválida, deberá recodificar los objetos con la nueva clave y notificar a su vez la invalidez de la clave anterior.

El uso de claves tiene varias ventajas:

• Permite que las funciones de cifrado y descifrado puedan ser públicas.
• Se pueden usar las mismas funciones para generar distintos cifrados.
• El resultado cifrado no depende únicamente del diseñador del algoritmo sino también de una clave fijada por el dueño del objeto cifrado.

Digitalización realizada con propósito académico

Sin embargo, el uso de claves también genera problemas importantes. En primer lugar, la clave debe ser conocida por el codificador y el descodificador de un objeto. En segundo lugar, la clave debe resistir los intentos de rotura, lo que significa que debe tener una cierta complejidad. En tercer lugar, limita la comunicación entre procesos a aquellos que conocen sus respectivas claves, lo que obliga a establecer protocolos de intercambio de claves que permitan llevar a cabo comuni caciones seguras, incluso aunque se ejecuten sobre canales inseguros [19811. Mediante estos protocolos se puede conseguir que dos usuarios que sospechan entre sí puedan comunicarse y estar convencidos de la validez de sus contactos. Existen distintos tipos de protocolos [19941, tales como los arbitrados, los adjudicados o los autocontrolados, pero la distribución de claves es uno de los problemas principales en todos ellos [19941. Según cómo se distribu yan las claves, se clasifican en sistemas con protocolos simétricos o asimétricos.

9.5.2. Sistemas de clave privada y sistemas de clave pública

Los sistemas de clave privada se basan en la ocultación de la clave de cifrado [19821. Se supone que la clave es conocida únicamente por el usuario que cifra el objeto original. Estos siste mas, que se han usado convencionalmente, ejecutan el cifrado (E) aplicando la clave (k) al objeto origen (o) para obtener el objeto cifrado (c):

C = E (k, o)

El problema de estos sistemas es que para que alguien descifre el objeto cifrado debe conocer la clave con que está cifrado. Hay dos soluciones para ese problema:

• Propagar la clave.
• Recodificar y añadir nuevas claves.

Imagine ahora que se desea que tres procesos (A, B y C) compartan objetos cifrados. Distribuir a todos ellos la misma clave propagando la clave original no parece muy seguro. Es mejor que cada par de procesos, por ejemplo A y B, compartan una clave kAB para cifrar sus interacciones. Lo mismo debería hacerse para las interacciones (A, C) y (B, C). En general, para n usuarios serían necesarias * (n — 1) /2 claves, lo que hace que para un número grande de usuarios sea muy difícil mantener claves confidenciales y con buenas características, debido principalmente a dos proble mas básicos:

• Es necesario tener una base de datos de claves almacenadas en un sistema seguro, por la imposibilidad de recordar todas ellas.
• Estos sistemas violan el principio de diseño que recomienda mostrar al exterior lo más posible para evitar la curiosidad y los ataques de usuarios dispuestos a romper la seguridad del sistema.

El sistema DES (Data Encription Standard) es el más popular de los de clave privada.

Para resolver los problemas de propagación de claves, Diffie y Helman [1977] propu sieron un sistema de cifrado con clave pública, en el que cada usuario tiene una clave de cifrado que puede conocer todo el mundo. De esta forma cada usuario puede publicar su clave de cifrado para que cualquiera pueda enviar un mensaje cifrado con dicha clave. Sin embargo, cada usuario tiene también una clave de descifrado secreta o privada. En cierta forma, éste es un sistema de cifrado de sentido único, donde cada usuario tiene un par de claves kpR) pública y privada, respectivamente.

Digitalización realizada con propósito académico

Ambas claves permiten aplicar sobre un objeto, o, los algoritmos de cifrado, E, y de descifrado, D, de la siguiente manera:

O = D (kpR, E O))
O = D E (KPR,))

Es decir, se puede descifrar con la clave privada algo cifrado con la clave pública y se puede descifrar algo cifrado con la clave privada sólo si se dispone de la clave pública. Estas dos propieda des implican que ambas claves se pueden aplicar en cualquier orden.
Con este método sólo se necesitan dos claves para cualquier número de usuarios: clave pública y privada. Cualquier proceso A puede enviar a B mensajes cifrados con la clave pública de B. Sólo el proceso B podrá descrifrar el mensaje, ya que es el único que posee la clave privada, necesaria para descifrar el mensaje cifrado con la clave pública. Este método asegura la confidencialidad, puesto que aunque un intruso obtenga el mensaje cifrado, no podrá descifrarlo. Además, soluciona los dos problemas principales de los sistemas de clave privada:

• No es necesario intercambiar claves para poder comunicarse con un servidor de forma segura.
• Muestran lo más posible al exterior, evitando que los intrusos tengan curiosidad por conocer las claves de los servidores.

Se han propuesto varios algoritmos de cifrado con clave pública: Merkle-Hellman Knapsacks, RSA, Hash, DAS, etc. [19961. Todos ellos han sido objeto de extensas investigaciones criptográficas que han conducido a la rotura de varios de ellos, como ocurrió con el algoritmo de Merkle-Hellman. El más popular es el RSA (Rivest-Shamir-Adelman), que fue presentado en 1978 y que se mantiene seguro hasta el momento. Este algoritmo usa operaciones de grandes números y dos claves, para cifrar (c) y descifrar (d). Cada bloque B se codifica como Bc mod n. La clave de descifrado se elige cuidadosamente de forma que (Bc) d mod n = B. Por tanto, el receptor legítimo del bloque, al conocer d, sólo tiene que llevar a cabo esa operación. Para cualquier otro usuario sería muy difícil calcular potencias del bloque cifrado hasta encontrar d puesto que el tiempo de cómputo es exponencial.

Una variante de los algoritmos de clave pública son las firmas digitales, que se usan para realizar autorizaciones en un entorno de computación donde no hay objetos tangibles (p. ej.: la pupila o la firma) para identificar al usuario. El sistema DSS (Digital Signature Standard), publica do en 1991, se basa en el algoritmo de firma digital de El Gamal. Una firma digital es un protocolo que produce el mismo efecto que una firma real. Es una marca que sólo el dueño puede proporcio nar, pero que otros pueden reconocer fácilmente como perteneciente a dicho usuario. Se puede implementar mediante:

• Sistemas de clave privada, donde la posesión de la clave garantiza la autenticidad del mensa je y su secreto.
• Sistemas de sello, donde no es necesaria una clave. Se puede hacer que el usuario tenga un sello que lo identifique, que puede ser una función matemática o una tarjeta electrónica con información grabada en la banda magnética.
• Sistemas de clave pública, donde la posesión de la clave de descifrado hace que sólo el receptor adecuado pueda descifrar un mensaje cifrado con su clave pública.

Una firma digital debe ser auténtica, no falsificable, no alterable y no reusable. Para satisfacer estos requisitos, las firmas digitales pueden incluir sellos con fechas y números de orden o números aleatorios. Los detalles relativos a firmas digitales se pueden estudiar en [1997].

Digitalización realizada con propósito académico

9.6. CLASIFICACIONES DE SEGURIDAD

La clasificación de los sistemas de computación según sus requisitos de seguridad ha sido un tema ampliamente discutido desde los años setenta. La disparidad de criterios existentes se ha ampliado más con la conexión de las computadoras para formar redes de computación que pueden compartir recursos. Algunas de las clasificaciones existentes en la actualidad son la clasificación del Departa mento de Defensa (DoD) de los Estados Unidos de América, el criterio alemán, el criterio canadien se, el ITSEC o el criterio común.

La última clasificación ha sido definida conjuntamente en Estados Unidos y Canadá, siendo publicada su primera versión en 1994 [19941. Es un sistema complejo que todavía está en fase de elaboración y discusión, por lo que hay muy pocos sistemas comerciales que se ajusten a esta norma. Sin embargo, es importante resaltar que tiene grandes posibilidades de convertirse en un estándar.

9.6.1. Clasificación del Departamento de Defensa (DoD) de Estados Unidos

Una de las clasificaciones más populares es la del Orange Book del Departamento de Defensa (DoD) de Estados Unidos [19851. Esta clasificación especifica cuatro niveles de seguridad: A, B, C y D (Fig. 9.13). A continuación, se describen estos niveles de seguridad y las características de cada uno.

Nivel D. Sistemas con protección mínima o nula

No pasan las pruebas de seguridad mínima exigida en el DoD. MS-DOS y Windows 3.1 son siste mas de nivel D. Puesto que están pensados para un sistema monoproceso y monousuario, no pro porcionan ningún tipo de control de acceso ni de separación de recursos.

Nivel C. Capacidad discrecional para proteger recursos

La aplicación de los mecanismos de protección depende del usuario, o usuarios, que tienen privile gios sobre los mismos. Esto significa que un objeto puede estar disponible para lectura, escritura o

A	Plan de seguridad acreditado	Ax	A1 + Desarrollo con instalaciones y personal fiables
		A1	B3 + Sistema de seguridad con verificación formal
B	Sistema de seguridad obligatorio	B3	B2 + ACL para denegar acceso + registro y auditoría de violaciones de seguridad
		B2	B1 + Protección obligatoria para todo recurso
		B1	C2 + Protección obligatoria para todo objeto de usuario
C	Capacidad discrecional de controlar accesos	C2	C1 + Control de acceso individual
		C1	Control de acceso por dominio de seguridad
D	No existen medidas de seguridad		

Figura 9.13. *Clasificación de seguridad del DoD.*

Digitalización realizada con propósito académico

cualquier otra operación, según el libre albedrío de su dueño. Casi todos los sistemas operativos comerciales de propósito general, como UNIX, LINUX o Windows NT, se clasifican en este nivel. Este nivel se divide a su vez en dos subniveles, dependiendo de la precisión del control de acceso:

• **Clase C1. Control de acceso por dominios**. No hay posibilidad de establecer qué elemento de un determinado dominio ha accedido a un objeto. UNIX pertenece a esta clase. Divide a los usuarios en tres dominios: dueño, grupo y mundo. Se aplican controles de acceso según los dominios, siendo todos los elementos de un determinado dominio iguales ante el sistema de seguridad.
• **Clase C2. Control de acceso individualizado**. Granularidad mucho más fina en el control de acceso a un objeto. El sistema de seguridad del ser capaz de controlar y registrar los accesos a cada objeto a nivel de usuario. Windows NT pertenece a esta clase.

Nivel B. Control de acceso obligatorio

En este nivel, los controles de acceso no son discrecionales de los usuarios o dueños de los recursos, sino que deben existir obligatoriamente. Esto significa que todo objeto controlado debe tener pro tección, sea del tipo que sea. En caso de que el dueño no defina cuál, el sistema de seguridad asigna una por defecto. Este nivel se divide a su vez en tres subniveles:

• **Clase Bi. Etiquetas de seguridad obligatorias**. Cada objeto controlado debe tener su eti queta de seguridad. Pueden existir objetos no controlados. Este modelo de seguridad se ajusta al de Bell-La Padula.
• **Clase B2. Protección estructurada**. Todos los objetos deben estar controlados mediante un sistema de seguridad con diseño formal y mecanismos de verificación. Estos mecanismos permiten probar que el sistema de seguridad se ajusta a los requisitos exigidos. Controles obligatorios, y asignados según el principio del menor privilegio posible, para objetos, suje tos y dispositivos.
• **Clase B3. Dominios de seguridad**. B2 ampliado con pruebas exhaustivas para evitar cana les encubiertos, trampas y penetraciones. Diseño probado y verificado, que usa niveles, abstracciones de datos y ocultamiento de información. El sistema debe ser capaz de detectar intentos de violaciones de seguridad, para ello debe permitir la creación de listas de control de acceso para usuarios o grupos que no tienen acceso a un objeto.

Nivel A. Sistemas de seguridad certificados

Para acceder a este nivel, la política de seguridad y los mecanismos de protección del sistema deben ser verificados y certificados por un organismo autorizado para ello. Organismos de verificación muy conocidos son el National Computer Security Center o el TEMPEST
• Clase Ai. Diseño verificado. Clase B 1 más modelo formal del sistema de seguridad. La especificación formal del sistema debe ser probada y aprobada por un organismo certifica dor. Para ello debe existir una demostración de que la especificación se corresponde con el modelo, una implementación consistente con el mismo y un análisis formal de distintos problemas de seguridad. El Vax Security Kernel pertenece a esta clase.
• Clase Ax. Desarrollo controlado. Al más diseño con instalaciones y personal controlados. Formas de control no definidas. Se podrían incluir requisitos de integridad de programas, alta disponibilidad y comunicaciones seguras.

Digitalización realizada con propósito académico

9.7. SEGURIDAD Y PROTECCIÓN EN SISTEMAS OPERATIVOS DE PROPÓSITO GENERAL

Un sistema operativo puede dar soporte de ejecución a múltiples procesos de múltiples usuarios que se ejecutan de forma concunente. Por ello, una de las funciones principales del sistema operativo es proteger los recursos de cada usuario para que pueda ejecutar en un entorno seguro. Para poder satisfacer esta función, todos los sistemas operativos deben tener mecanismos de protección que permitan implementar distintas políticas de seguridad para los accesos al sistema l 1968] y [.19741. Dichos mecanismos permiten controlar el acceso a los objetos del sistema permi tiéndolo o denegándolo sobre la base de información tal como la identificación del usuario, el tipo de recurso, la pertenencia del usuario a un cierto grupo de personas, las operaciones que puede hacer el usuario o el grupo con cada recurso, etc. La existencia de los mecanismos de seguridad obliga a mantener un compromiso constante entre separación y compartición. Este compromiso es más difícil de satisfacer a medida que la granularidad de control es más fina. En este caso es más difícil aislar recursos pero es más fácil compartirlos. Es pues necesario lograr un equilibrio entre el tipo y la intensidad de la separación y el grado de compartición de recursos [19851.
En esta sección se estudian los distintos mecanismos de protección que ofrecen los sistemas operativos de propósito general para implementar la política de seguridad deseada por los usuarios. Para mostrar su uso, se aplican a archivos o directorios, aunque se pueden aplicar a cualquier tipo de objeto. Los mecanismos de protección hardware, como registros valla o arquitecturas etiquetadas [1992], se estudiaron en el Capítulo 1, por lo que no se contemplan aquí.

9.7.1. Autenticación de usuarios

El paso previo a la aplicación de cualquier esquema de protección o confidencialidad de datos es conocer la identidad del usuario que está accediendo a dichos datos. El objetivo de la identificación del usuario, también denominada autenticación, es determinar si un usuario (persona, servicio o computadora) es quien dice ser. Para ello, la mayoría de los sistemas solicitan del usuario que, previamente al acceso a los recursos del sistema, proporcione algún tipo de información que se supone que únicamente es conocida por él y que debe ser suficiente para su identificación.

Existen diversas formas de establecer la identidad de un usuario:

• Pedir información que sólo conoce él a través de contraseñas, juegos de preguntas o algorit mos de identificación.
• Determinar características físicas del usuario tales como la pupila, la huella dactilar, el DNA, la firma, etc. Es más costoso que el método anterior pero más fiable. Actualmente 1 empiezan a ser frecuentes los sistemas de detección de la huella digital, la pupila o el tono de voz de una persona.
• Pedir un objeto que posee el usuario, como puede ser una firma electrónica, una tarjeta con banda magnética o con un chip.

Cualquiera que sea el método de identificación de usuario utilizado, en las instalaciones donde la seguridad es un aspecto importante se pueden tomar medidas suplementarias, como parte del proceso de autenticación, para evitar la existencia de intrusos o, al menos, dificultar accesos fraudu lentos al sistema. En algunos sistemas operativos, como por ejemplo VMS de DEC, se puede limitar el acceso a los recursos a determinadas horas del día o a determinados terminales de usuario. En sistemas con conexión telefónica, los usuarios solicitan conexión y, posteriormente, el sistema les llama a un número de teléfono previamente definido. En sistemas interactivos, si un usuario

Digitalización realizada con propósito académico

accede al sistema desde un terminal y dicho terminal sobrepasa un tiempo de inactividad, se expul sa al usuario del sistema después de uno o varios avisos de desconexión. Los sistemas más moder nos, especialmente los que permiten conexiones a través de redes, intercambian claves de forma dinámica cada cierto tiempo. Este método es equivalente a pedirle al usuario que reintroduzca su contraseña, tarjeta o característica física cada cierto tiempo. Además, como criterio general de seguridad, los sistemas operativos modernos dan la posibilidad de registrar todos los accesos al sistema, lo que permite hacer controles interactivos y a posteriori de dichos accesos.
Windows NT tiene un subsistema de seguridad encargado de autenticar a los usuarios. Cuando se crea un nuevo usuario en el sistema, se almacena en una base de datos de seguridad una ficha del usuario que incluye su identificador de seguridad, los grupos a los que pertenece, sus privilegios, grupo primario y enlace con su lista de control de acceso. Cuando el usuario quiere acceder al sistema introduce su contraseña a través de un proceso logon. Dicha contraseña se pasa al subsiste ma de seguridad, que verifica la identidad del usuario y, en caso positivo, construye una ficha de acceso para el usuario. Este objeto sirve como identificador oficial del proceso siempre que el usuario intente acceder a un recurso a partir de ese instante. La Figura 9.14 resume los atributos y servicios de este tipo de objeto, así como un ejemplo de ficha de acceso en Windows NT.

El proceso de autenticación

Habitualmente, cuando un usuario quiere acceder al sistema, aparece una pantalla o mensaje de entrada. En el caso de Windows NT, la pantalla pide tres valores:
• Identificación del usuario: nombre del usuario en el sistema.
• Palabra clave o contraseña: espacio para teclear la clave (el eco muestra *).
• Dominio de protección al que pertenece el usuario.

Existen varios fallos posibles en el proceso de entrada al sistema, por lo que este proceso debe ser robusto y no dar información a los intrusos. Con el paso del tiempo se ha ido dotando de mayor robustez e integridad al proceso de autenticación gracias, en parte, a la experiencia de errores previos. En las primeras versiones de UNIX, por ejemplo, el proceso login se podía matar desde el teclado. Al matarlo, el usuario se quedaba dentro del sistema con privilegios de administrador del

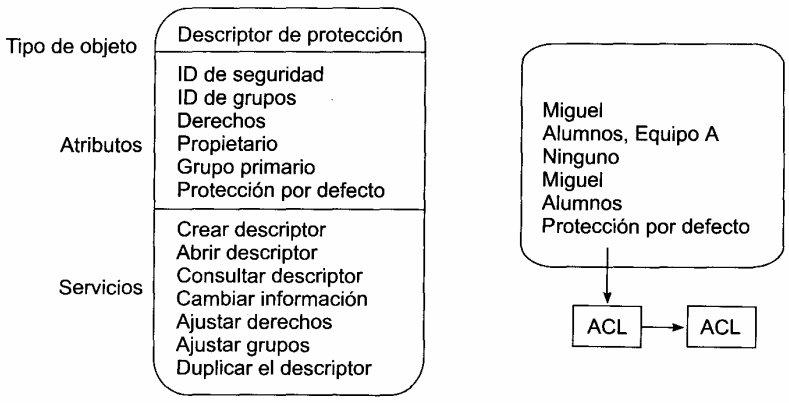


Figura 9.14. *Descriptor de seguridad en Windows NT.*

Digitalización realizada con propósito académico

sistema. Igualmente, en distintas versiones se comprobaba primero la identificación del usuario y, caso de no existir, se indicaba error sin pedir la contraseña:

Login: pepe
Error: el usuario no existe
Login:

Con este método era sencillo averiguar qué usuarios había o no en un sistema. Otro fallo habitual era comprobar la contraseña carácter a carácter, notificando un error en cuanto fallaba el primer carácter. Con este método era muy sencillo comprobar las claves carácter a carácter hasta dar con la clave correcta.

Actualmente, los sistemas piden todos los datos de autenticación, verifican el proceso y notifican el éxito o error de todo el proceso.

Login: pepe
Password: * * * * * * * * * * * * *
Acceso inválido. Inténtelo de nuevo.

En caso de error al teclear la clave, el usuario puede intentar la reinserción de la misma. Ahora bien, teniendo en cuenta que un usuario debería ser capaz de teclear su clave correctamente en cuatro o cinco intentos, es necesario tomar alguna medida especial cuando se alcanza ese número de errores. En UNIX no se penniten reintentos hasta que pasa un cierto tiempo. En Windows NT, una medida habitual es bloquear la cuenta y notificar la situación al administrador de seguridad del siste ma. En ambos casos se trata de evitar que los programas que intentan adivinar las claves del sistema se puedan ejecutar de forma normal o que lo tengan que hacer de forma tan lenta que tal detección sea inviable. Todos los sistemas operativos registran los intentos de acceso fallidos al sistema.

Un ataque a la seguridad del sistema relacionado con el proceso de autenticación consiste en suplantar al proceso que pide los datos de entrada. Imagine que un intruso entra al sistema y suplan ta dicho proceso con uno que muestra una ventana de entrada idéntica a la del sistema. Un usuario, al intentar entrar, tecleará su identidad y su clave en la ventana del intruso, que inmediatamente conocería dichos datos. Esta intrusión podría ser difícil de detectar si además de suplantar al proce so de entrada al sistema se hubiera hecho con permiso del administrador. En este caso, tras registrar los datos del usuario, el intruso podría ejecutar el proceso de entrada correcto escondiendo al usua rio la violación de seguridad. Esta suplantación se clasificaría como un caballo de Troya.

ADVERTENCIA 9.2

Evidentemente, el principal fallo de un proceso de autenticación es que el propio usuario sea descuidado. Un usuario descuidado deja su cuenta abierta, apunta la clave al lado del terminal o se la dice a cualquiera que se la pregunte. Este tipo de fallos son críticos si el usuario es el administrador del sistema. Un principio de seguridad básico es la desconfianza. Si alguien necesita un recurso suyo, déselo usted mismo o póngalo compartido, pero nunca comparta su clave de acceso al sistema.

9.7.2. Palabras clave o contraseñas

El método más usado actualmente para identificar a un usuario es el de las contraseñas, o palabras clave [19791. Una contraseña es un conjunto de caracteres alfanuméricos y especiales

Digitalización realizada con propósito académico

conocido únicamente por el usuario y por el sistema operativo sobre el que se ha llegado a un acuerdo para que sea usado como clave de acceso al sistema. Normalmente, cuando se habilita un nuevo usuario en el sistema, éste introduce su contraseña, que puede cambiar posteriormente tantas veces como quiera. Dicha contraseña se guarda cifrada en unos archivos especiales.

Cuando intenta acceder a su cuenta, el proceso que controla los accesos (login) pide al usuario que teclee su contraseña. Inmediatamente, dicha contrasefia es cifrada [19821 y comparada con la existente en el archivo de contraseñas para ese usuario. Si las dos contraseñas, ambas cifradas, coinciden, se permite acceder al usuario. En otro caso se deniega el acceso. Huelga decir que no deben existir copias sin cifrar de las contraseñas y que mientras el usuario teclea su contraseña hay que inhibir el eco en la pantalla para que otras personas no puedan ver dicha palabra clave.

Este sistema es sencillo de implementar y de usar, funcionando de forma similar en todos los sistemas operativos. Sin embargo, asumiendo que la autenticación se basa en tupias «usuario, clave», es necesario tomar cuatro decisiones de diseño básicas para un sistema como éste:

1. ¿Quién asigna las palabras clave?
2. Longitud y formato de las palabras clave.
3. ¿Dónde se almacenan las claves?
4. Duración de las claves.

Asignación de claves

Normalmente, la palabra clave es fijada por el usuario cuando entra en su cuenta y la puede cambiar tantas veces como quiera. Sin embargo, hay situaciones en que el sistema operativo elige la clave, o parte de la clave, de un usuario:

• El sistema o el administrador fijan una clave de acceso cuando se crea la cuenta [
19721.
• Si dos usuarios eligen la misma clave, el sistema debe resolver la ambigüedad. Denegar la clave por repetida daría pistas a posibles intrusos. Por ello, algunos sistemas operativos, como UNIX, añaden una extensión (un número aleatorio de 12 bits y el identificador del proceso) a la clave de cada usuario.
Si el usuario no usa caracteres especiales, usa claves cortas, palabras de diccionario o fáciles de adivinar, el sistema debe añadir algo que haga su identificación más difícil o rechazar dichas claves.
• Cuando se usan sistemas con claves seguras, el usuario introduce una clave y, basándose en ella, el sistema operativo le devuelve un conjunto de claves complejas identificadas por un número. Así, cuando el usuario intenta acceder al sistema se le pide una clave determinada. Estas claves se usan una única vez.

La asignación discrecional de claves por parte del usuario tiene problemas asociados. Es habi tual que los usuarios tengan contraseñas fáciles de adivinar, como su nombre, fecha de nacimiento, dirección, nombre de familiares, claves que se puedan buscar fácilmente con un diccionario o no tiene clave. Estos usuarios son presa fácil de cualquier programa rompedor de claves, que tardará muy poco en encontrar su clave.

Longitud y formato de las claves

La longitud y el formato de las claves han ido cambiando a través del tiempo, principalmente debido a la detección de fallos asociados a las claves usadas en cada momento. La Figura 9.15

Digitalización realizada con propósito académico

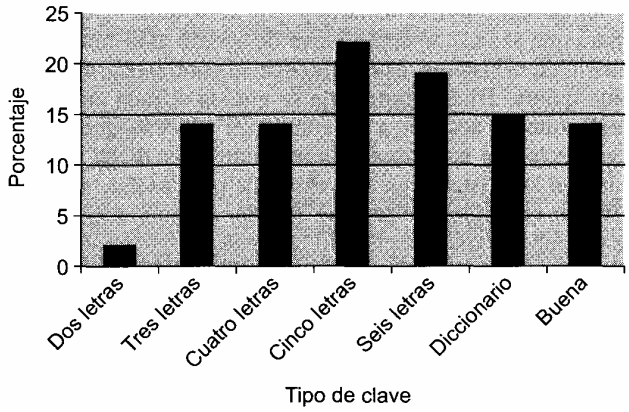


Figura 9.15. *Distribución de palabras clave según su dificultad.*

muestra una distribución de palabras clave obtenida en un estudio realizado por Monis y Thompson en 1979 [19791. Como se puede ver, la mayoría de las palabras clave eran muy cortas o fáciles de adivinar. Estudios posteriores, como los de Luby [19891 y Spafford [1992], mostraron un panorama similar. Actualmente, un buen servidor de claves trata de descartar los fallos más probables complicando el formato o los datos de las claves.

Observe que, teóricamente, un sistema que use palabras clave con un mínimo de 6 caracteres, siendo éstos los 26 alfabéticos, diez numéricos y diez caracteres especiales, tiene un número de 466 palabras clave posibles. Si la longitud de la clave puede variar entre 6 y 10 caracteres, el número posible sería 466 + 46 + 468 + 46 + 46'°. Costaría cientos de años romper una clave de este estilo, siempre que el usuario fuera cuidadoso y no cometiera fallos como los mencionados anteriormente. Para aumentar la complejidad de la clave se puede hacer que el mandato que permite cambiar la contraseña (passwd en UNIX) obligue al usuario a meter caracteres no alfanuméricos y que fuerce la existencia de contraseñas de una longitud mínima añadiendo números aleatorios a la palabra clave antes de cifrarla.

Almacenamiento de claves

Es muy importante dónde se almacenan las claves para la seguridad del sistema. En el sistema operativo UNIX, por ejemplo, las claves se almacenaban tradicionalmente en un archivo de texto denominado / etc /passwd. En cada entrada del archivo se encontraba la identidad de un usuario y su palabra clave cifrada, entre otras cosas, con el siguiente formato:

m±guel:xsfg7.5t:Miguel Alumno: /home/users/miguel:/bin/csh

Este archivo era accesible para lectura por todo el mundo, lo que permitía que cualquier usuario pudiera ejecutar programas que leían este archivo y comparaban las palabras clave con contraseñas de uso probable o, simplemente, con listados exhaustivos de contraseñas. Para paliar este problema, actualmente las contraseñas cifradas se guardan en archivos especiales que única mente son accesibles para el administrador. Son los denominados archivos sombra (/etc/sha dow). El sistema operativo tiene operaciones internas para acceder a estos archivos y manipular las

Digitalización realizada con propósito académico

contraseñas. En el caso de Windows NT, se guardan en archivos que sólo están accesibles para el administrador del sistema y que se manipulan a través de la utilidad de administración para gestión de usuarios.
Se puede incrementar la seguridad del sistema cifrando todos los directorios y archivos de palabras claves. De esta forma, aunque alguien pudiera acceder a ellos, no podría usar la informa ción. La forma más habitual de cifrar la palabra clave es usar funciones de sentido único. Con este enfoque se cifra la palabra clave, se guarda en el archivo de claves y no se descifra jamás. Cuando un usuario desea entrar al sistema se pide su clave, se cifra y se compara con la almacenada en el archivo.

Duración de las claves

Para dificultar la detección de contraseñas válidas por parte de posibles intrusos se puede configurar el sistema operativo: los sistemas operativos permiten que las contraseñas de usuarios sean válidas únicamente durante un cierto tiempo. Evidentemente, cuanto más corta sea la duración de la pala bra clave, más difícil será romper la seguridad del sistema. Actualmente, muchos sistemas operati vos obligan a cambiar las claves periódicamente, por ejemplo cada 3 días, si así lo decide el admi nistrador de seguridad. Este método per se no tiene por qué mejorar la seguridad del sistema. Suponga que un usuario tiene tres claves y siempre las usa de forma cíclica. Si las tres claves son sencillas, la seguridad del sistema no ha mejorado mucho.

Para evitar la situación anterior, empieza a ser frecuente el uso de contraseñas válidas para un único acceso (claves de una sola vez). Hay tres formas básicas de implementar esta polftica:

1. Obligar al usuario a cambiar su contraseña cada vez que entra en el sistema. Esta solución no limita posibles accesos futuros de intrusos, ya que éstos pueden cambiar la palabra
clave a su antojo, pero permite que el usuario afectado por la violación de seguridad lo detecte tan pronto como quiera entrar a su cuenta y avise al administrador del sistema.
2. Asignar una función matemática a cada usuario, de forma que cuando el usuario entre al sistema se le proporcione un argumento para la función y se le pida que introduzca el
resultado de la función para dicho argumento. Las funciones matemáticas a usar pueden ser muy variadas, pero en sistemas conectados a la red suelen ser de una gran complejidad.
Es muy frecuente usar polinomios.
3. Usar libros de contraseñas ordenadas según un cierto orden numérico. Cada vez c el usuario quiere acceder al sistema se le pide que introduzca la contraseña con un cierto
número de identificación. El libro de claves es generado por el usuario y debe estar en su posesión, sin que existan copias del mismo que no estén cifradas. Obviamente, este méto do no es útil si el intruso tiene acceso al libro de claves del usuario en versión no cifrada.
Este es el método usado por el sistema Securekey.

La conexión de sistemas mediante redes supone un problema añadido para la seguridad. Una precaución mínima a tener en cuenta, silos usuarios pueden acceder a su cuenta desde sistemas remotos, es que las contraseñas nunca deben viajar por la red sin estar cifradas.

9.7.3. Dominios de protección

Para poder implementar mecanismos de protección sobre los objetos del sistema, que pueden ser hardware (UCP, memoria, etc.) o software (procesos, archivos, semáforos, etc.), es necesario en primer lugar disponer de identificadores únicos de usuarios y de objetos. Asumiendo que dichos

Digitalización realizada con propósito académico

identificadores son proporcionados por el sistema de autenticación y el de nombrado, hay que definir:

• Los posibles conjuntos de objetos a los que se aplican los mecanismos de protección.
• La relación de operaciones que cada usuario puede hacer sobre cada conjunto de objetos.
• La forma en que se define dicha relación: por objetos, por usuarios, etc.

La relación entre objetos y derechos de acceso se define usando dominios de protección. Un dominio de protección es un conjunto de pares (objeto, derechos), donde cada par especifica un objeto y las operaciones que se pueden ejecutar sobre el mismo. Por ejemplo:

dominio_1= [RW), (/dev/lpd, W)J

Un objeto puede pertenecer a varios dominios de protección simultáneamente. En cada mo mento, un proceso ejecuta en un dominio de protección, pudiendo cambiar de un dominio a otro, si el sistema operativo lo permite. En UNIX, por ejemplo, un proceso puede ejecutar en modo usuario o en modo núcleo. Cuando un proceso hace una llamada al sistema cambia a modo núcleo y tiene acceso a unos dominios de protección distintos que en modo usuario, tales como páginas de memo ria, dispositivos especiales, etc. (Fig. 9.16). Algo similar ocurre si el sistema operativo permite aun proceso cambiar su identidad mientras está ejecutando, como ocurre con las llamadas setgid y setuid de UNIX. En este caso, cuando el proceso cambia su identidad pasa a tener acceso a los dominios de protección de su nueva identidad. Dos procesos con el mismo identificador tendrán acceso al mismo conjunto de objetos y con los mismos derechos.

Para evitar una explosión de dominios de protección, algunos sistemas operativos, como UNIX, restringen significativamente el número de dominios de protección y el tipo de derechos de acceso sobre un objeto. Por ejemplo, en UNIX sólo existen tres dominios de protección:

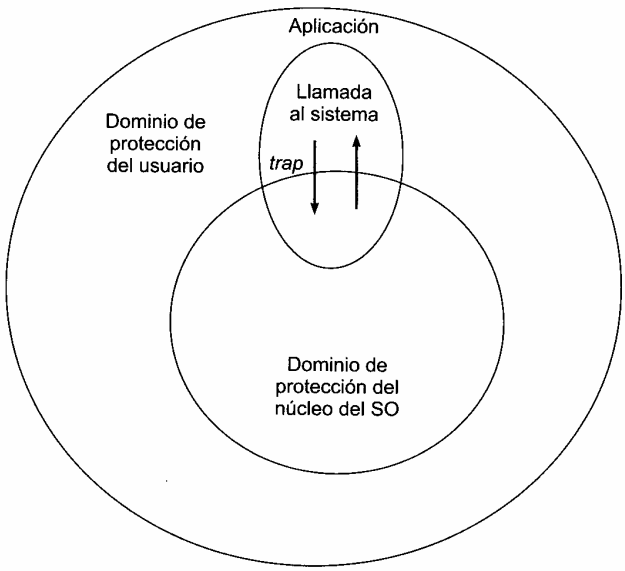


Figura 9.16. *Cambio de dominio de protección.*

• owner. Dueño del objeto. Suele ser su creador. El alumno Miguel, por ejemplo. Cada usua rio tiene un identificador (u íd) único en todo el sistema. Este identificador está asociado a todos los objetos creados por el usuario, para indicar que es su dueño.
• group. Usuarios distintos del dueño que forman parte de su grupo. Los componentes del grupo suelen estar relacionados de alguna forma, por lo que se les permite gestionar los
derechos de sus objetos con criterios comunes al grupo. Un grupo de usuarios podrían ser los alumnos de la asignatura Sistemas Operativos. Si Miguel cursa Sistemas Operativos perte nece a este grupo. Cada grupo tiene un identificador (gid) único en todo el sistema. Un grupo se define especificando los uid de los usuarios que forman parte del mismo. El gid está asociado a todos los objetos creados por cualquier usuario que pertenezca a dicho grupo.
• others. El resto de los usuarios conocidos que no son el creador ni su grupo (también llamados mundo). Suelen tener más restricciones en los derechos. En una computadora
dedicada a alumnos, para Miguel el resto del mundo serían los usuarios no matriculados en Sistemas Operativos.

Para cada dominio se permiten únicamente tres tipos de operaciones sobre un objeto:

• Leer (r).
• Escribir (w).
• Ejecutar (x).

Además, cada objeto tiene asociados dos bits extra para conceder permisos especiales del dueño y del grupo al que pertenece el dueño. El primero de ellos se denomina bit setuid y el segundo bit setgid. Estos bits permiten realizar cambios de dominio cuando se ejecutan objetos que tienen alguno de ellos activado. Cuando un usuario A ejecuta un programa del usuario B, almacena do en un archivo cuyos bits de setuid y setgid están inactivos, su identificación sigue siendo la de A y ése es el setuid del proceso. Sin embargo, si el archivo tiene el bit setuid activado, la identidad del proceso será la de B, aun cuando sea el usuario A el que está ejecutando el proceso. Asimismo, si tiene activado el setgid, la identidad del grupo del proceso será la del grupo de B. Estos bits se pueden modificar usando la llamada al sistema chxnod. Las llamadas al sistema setuid y setgid, que se explican en detalle en la sección de llamadas al sistema de POSIX, permiten cambiar de dominio a vn usuario y son utilizadas por los sistemas operativos de tipo UNIX para permitir ceder permisos de acceso a archivos del sistema o a recursos que nunca estarían disponibles a usuarios no privilegiados, pero que pueden ejecutar temporalmente para el desarrollo de sus actividades en el sistema. La mayoría de los intérpretes de mandatos y programas del sistema funcionan sobre la base de este mecanismo. En UNIX, por ejemplo, el mandato mkdir permite crear un directorio. En versiones antiguas de UNIX mkdi r no era una llamada al sistema, por lo que era necesario ejecutar la llamada al sistema mknod, que sólo podía ejecutar el superusuario. Lo que se hacía era permitir que el mandato mkdir se ejecutara con el bit setuid del superusuario activado, por lo que cualquie ra que lo ejecutara lo hacía como si fuera el superusuario y pudiera acceder a mknod.

Existen otras formas de cambiar de dominio de protección en los distintos sistemas operativos existentes. Alternativas a los bits de protección de UNIX son:
• Colocar los programas privilegiados en un directorio especial. En este caso, el sistema ope rativo es responsable de cambiar el setuid del proceso que ejecuta en ese directorio.
• Usar programas especiales, denominados demonios, para dar servicios en dispositivos con acceso restringido. Un ejemplo claro es el de la impresora. Ningún proceso de usuario puede acceder a ella directamente, sino a través del spooler, un demonio que recibe las peticiones de impresión y las envía al dispositivo.
• No permitir nunca el cambio de identidad de un proceso y ejecutar todas las operaciones privilegiadas a través de servicios del sistema operativo.

Digitalización realizada con propósito académico

En cualquier caso, es necesario ser muy cuidadoso con los programas que se ejecutan de forma privilegiada. Si un proceso del superusuario no tiene bien definidos y controlados todos los fallos de protección, se puede generar una total falta de protección en el sistema. Algunos de los fallos de protec ción más famosos, como la invasión por virus o la captura de contraseñas, han sido posibles gracias a esta cesión de identidad.

Para evitar la total desprotección del sistema, en UNIX cada proceso que se ejecuta tiene cuatro identificadores asociados:

• uid: identidad real del usuario que ejecuta el proceso.
• euid: identidad efectiva del usuario que ejecuta el proceso. Puede no coincidir con la real si ha cambiado de dominio porque el archivo del proceso tiene el bit setuid activado.
• gid: identidad real del grupo al que pertenece el usuario que ejecuta el proceso.
• egid: identidad efectiva del grupo al que pertenece el usuario que ejecuta el proceso. Puede no coincidir con la real si ha cambiado de dominio porque el archivo del proceso tiene el bit setgid activado.

Para ejecutar los procesos que tienen fuertes restricciones de seguridad se exige siempre que la identidad efectiva y la real coincidan. Un ejemplo de este tipo es el mandato mount, que permite montar un sistema de archivos en una jerarquía de archivos ya existente.

ADVERTENCIA 9.3

¡Cuidado! Nunca tenga archivos con los bits de **setuid** o **setgid** activados a no ser que sepa muy bien por qué quiere ceder su identidad a cualquier otro que pueda ejecutar esos archivos. Esta medida sólo está justificada para dar acceso controlado a recursos que únicamente están accesibles para su propio `uid`.

9.7.4. Matrices de protección

La relación entre dominios y objetos se puede definir de forma completa mediante una matriz de protección, también denominada de acceso. Los dominios de protección son las filas de la matriz y los objetos son las columnas de la misma. El elemento (i, i) expresa las operaciones que el dominio ± puede ejecutar sobre el objeto j. Si la matriz de protección está completamente definida, los mecanismos de protección pueden saber siempre qué hacer cuando un proceso de un dominio solicita determinada operación sobre un objeto.

La Figura 9.17 muestra un ejemplo de matriz de protección. Cada elemento muestra una com binación (dom_i, obj _j) y sus derechos de acceso. Los elementos vacíos significan que no hay derechos. Los dominios son un objeto más del sistema, por lo que se incluyen también los cambios de dominio permitidos. Por ejemplo, en la Figura 9.17 se muestra que se puede cambiar de Domj a Dom_2, pero no al revés.

El modelo de matriz de protección, derivado del modelo teórico HRU, es muy claro desde el punto de vista conceptual, pero tiene inconvenientes para su implementación:

• La matriz de un sistema complejo puede ser muy grande y muy dispersa.
• Una matriz tiene un número fijo de filas (dominios) y columnas (objetos), lo que es muy poco flexible para sistemas cuyo número de dominios u objetos puede cambiar.

Digitalización realizada con propósito académico

Objeto / Dominio	Arch_1	Arch_2	Modem	Impresora	Dom_1	Dom_2	Dom_3
Dom_1	RWX	R	RW	W		Switch	
Dom_2	R		RW				Switch
Dom_3		RWX		W			

Figura 9.17. *Ejemplo de matriz de protección.*

Para resolver estos problemas, la mayoría de los sistemas operativos implementan la matriz mediante estructuras dinámicas de datos (listas) a las que se puede añadir o quitar elementos sin tener que redefinir ninguna estructura de datos del sistema operativo. Además, con esta técnica se evita la dispersión de la matriz al representar únicamente los elementos no nulos de la misma. Para la implementación de la matriz mediante elementos dinámicos, los sistemas operativos usan dos enfoques:

• Almacenar la matriz por columnas, con una lista por objeto que especifica qué operaciones puede hacer cada dominio sobre ese objeto. La lista resultante se denomina lista de control de accesos (ACL, Access Control List).
• Almacenar la matriz por filas, con una lista por dominio que especifica qué operaciones se pueden hacer sobre un objeto cuando se pertenece a ese dominio. La lista resultante se deno mina lista de capacidades (capabilities).

A continuación, se estudian ambas aproximaciones más detalladamente.

9.7.5. Listas de control de accesos

Una forma frecuente de controlar los accesos a un objeto es usar el identificador del usuario como criterio. Con listas de control de acceso es necesario especificar para cada dominio de protección, e incluso para cada usuario, qué tipos de accesos al objeto son posibles. Para implementar esta solu ción, a cada objeto (archivos, directorios, procesos, etc.) se le asocia una lista de pares:

(dominio, operaciones)

Cuando un usuario pide acceso a un objeto, se determina a qué dominio de protección pertene ce y se recorre la lista para ver si se puede hacer la operación solicitada. En caso positivo se permite el acceso. En caso contrario se deniega.

Por ejemplo, sea un sistema operativo en el que hay registrados cuatro usuarios (juan, mi guel, elvira y maría) que a su vez pueden pertenecer a tres grupos (profesor, alumno, visi tante). Algunos archivos podrían tener las siguientes ACL.

datos -> (juan,profesor,RW) (elvira,alumno,R)
notas -> (juan,profesor,RW) (*,alumno,R)
corrector -> (juan,profesor,RWX
Guía -> (miguel,alumno,RW) (elvira,alumno,R) (maría,visitante,R)

Digitalización realizada con propósito académico

Usando ACL es posible especificar completamente los derechos de acceso que sobre un objeto tiene un usuario o un grupo. Por ejemplo, el archivo Guía puede ser leído y escrito por el alumno miguel, que es el encargado de su mantenimiento, pero sólo puede ser leído por la alumna elvira o la visitante maría. El archivo notas puede ser leído por todos (*) los alumnos. Si aparece un nuevo grupo de usuarios, por ejemplo asociado a una práctica de una asignatura, sería necesario definir los derechos de ese grupo sobre las herramientas necesarias para dicha práctica. Por tanto, para que este sistema funcione correctamente, es necesario que la administración de los dominios y de la pertenencia a los mismos sea muy rigurosa.

Sea cual sea su implementación, las listas de control de acceso se corresponden directamente con las necesidades de los usuarios. Cuando un usuario crea un objeto puede especificar qué domi nios tendrán acceso al mismo y qué operaciones pueden realizar sobre el objeto. Sin embargo, localizar la información relacionada con un dominio en particular puede ser costoso si no se limita el número de dominios, como en UNIX. Además, la lista de control de accesos debe ser recorrida cada vez que se accede a un objeto, lo que puede requerir bastante tiempo. Esto hace que la mayoría de los sistemas basados en ACL sólo comprueben los derechos de acceso cuando se accede al objeto por primera vez, es decir, se abre el objeto. En caso de que se conceda acceso al objeto, no se hacen más comprobaciones posteriores a medida que se usa el objeto. Esta circunstancia hace muy difícil la revocación de derechos para los procesos que ya tienen objetos abiertos, incluso aunque se cambie su ACL. Por tanto, las listas de control de acceso tienen dos problemas asociados:

• Construir y mantener las listas es costoso en tiempo y recursos.
• Es necesario disponer de estructuras de almacenamiento de tamaño variable porque las listas pueden tener longitudes distintas dependiendo del objeto.

A continuación, se estudia con más detalle la implementación de las listas de acceso en UNIX y Windows NT.

Listas de control de acceso en UNIX

En UNIX, la implementación de las listas de control de acceso es sencilla por la simplificación de dominios de protección llevada a cabo en este sistema operativo. Como se vio en la sección ante rior, en UNIX, sólo existen tres dominios de protección: dueño, grupo, otros. Para cada dominio se permiten tres tipos de operaciones sobre un objeto: leer (r), escribir (w) y ejecutar (x). De esta forma se pueden implementar las ACL usando sólo 9 bits por objeto, información que cabe en el nodo-i del mismo. Esta solución es menos general que un sistema que use ACL de forma completa, pero es suficiente y su implementación es mucho más sencilla.

Por ejemplo, suponga que los archivos datos, notas y corrector tienen las siguientes ACL:

	Dueño	Grupo	Otros
datos	rw-	r——	
notas	rw		
corrector		rwx	——x

Estos permisos indican que los archivos datos y notas pueden ser leídos y escritos por el dueño y sólo leídos por la gente de su grupo y por otros. El archivo corrector puede ser leído, escrito y ejecutado por su dueño, pero el resto del mundo sólo puede ejecutarlo.

En las operaciones de interfaz con el sistema operativo, estos permisos se indican con números en octal. Se usa un dígito para cada dominio y el valor de los bits de cada dígito se pone a 1 si la

Digitalización realizada con propósito académico

operación es posible o a O si no lo es. Los permisos del ejemplo anterior se expresan de forma equivalente como:

dat os notas corrector 6 4
Dueño Grupo 7 1
6 4

Este modelo conlleva el que haya que hacer ciertas simplificaciones en cuanto a las operacio nes no contempladas. Por ejemplo, borrar es posible si se puede escribir en el directorio que contie ne el objeto que se pretende eliminar, atravesar un directorio es posible si se tiene activado el bit x, etcétera.

Para permitir el cambio de dominio se pueden modificar los bits setuid y setgid asociados a cada objeto. En el caso de que alguno de estos bits esté activado, se muestran dos bits más asociados al objeto. En el ejemplo anterior, suponga que el archivo corrector tiene activado el setuid. Su salida en un mandato is -la sería:

corrector Otros
Dueño Grupo — —x
rws ——x

Si tuviera activado el setgid, su salida en un mandato ls -la sería:

corrector rws ——5
Dueño Grupo Otros

Los parámetros de protección de un objeto se pueden cambiar en UNIX mediante las llamadas al sistema chmod y chown, que se explican en la sección de llamadas al sistema de POSIX.

Listas de control de acceso en Windows NT

Windows NT proporciona un sistema de seguridad algo más sofisticado que el de UNIX. Todos los objetos de Windows NT tienen asignados descriptores de seguridad como parte de sus fichas de acceso. La parte más significativa de los descriptores de seguridad es la lista de control de accesos. Cada entrada de la ACL contiene los descriptores de seguridad de los distintos dominios del sistema y los derechos de uso del objeto. Normalmente, sólo el dueño del objeto puede modificar los derechos de la ACL para permitir o denegar el acceso al objeto.

El criterio de asignación de derechos en la ACL de un objeto nuevo en Windows NT es el siguiente:

1. Si el creador de un objeto proporciona una ACL de forma explícita, el sistema la incluye en la ficha de acceso de dicho objeto.
2. Si no se proporciona una ACL de forma explícita, pero el objeto tiene un nombre, el sistema comprueba si el objeto debe heredar la de los objetos de su directorio. En ese caso se incluye en la ficha de acceso del objeto la ACL heredada de los objetos de su directorio.
3. Si ninguna de. las dos condiciones anteriores se cumplen, el subsistema de seguridad apli ca al objeto una ACL por defecto.

Además, en los descriptores de seguridad de los objetos se puede activar un campo de audito ría, que indica al subsistema de seguridad que debe espiar al objeto y generar informes de seguridad cuando algún usuario intente hacer un uso incorrecto del mismo.

Digitalización realizada con propósito académico

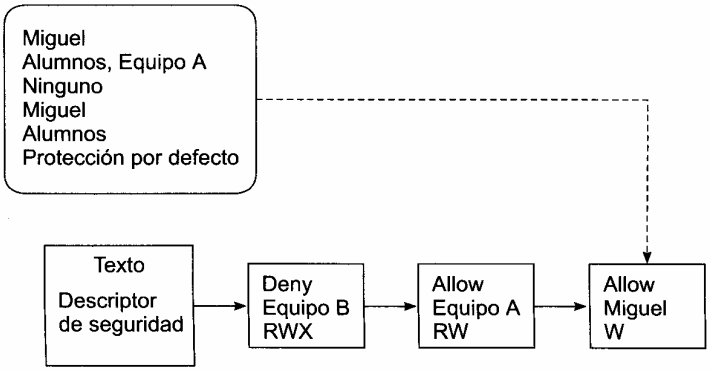


Figura 9.18. *Listas de control de acceso en Windows NT.*

La Figura 9.18 muestra un ejemplo de comprobación de derechos de acceso en Windows NT. En ella el usuario miguel pide acceso de lectura al objeto texto. El subsistema de seguridad recorre la lista de control de acceso hasta que encuentra la confirmación positiva (Allow) en la tercera posición, devolviendo al usuario un manejador al objeto. Obsérvese que los primeros ele mentos de la lista son denegaciones de derechos (Deny). Esto se hace así para evitar que si el grupo del usuario miguel tiene denegado el acceso, dicho usuario pueda acceder al elemento aunque una entrada de la lista se lo permita posteriormente.

9.7.6. Capacidades

La otra forma posible de implementar la matriz de protección es asociar a cada dominio un conjunto de descriptores que indiquen las operaciones que los componentes de ese dominio pueden efectuar sobre cada objeto del sistema. Estos descriptores se denominan capacidades (capabilities) y son una combinación de una referencia a un objeto con los permisos de acceso al mismo desde el dominio del poseedor de la capacidad. Las listas de capacidades son a su vez objetos, por lo que pueden ser incluidas dentro de otras listas de capacidades, facilitando la compartición de los objetos en domi nios y subdominios [19861.

A continuación, se muestran las capacidades del usuario juan, siguiendo los ejemplos anteriores:

Obj eto				archivo rwx	corrector
Cap-id	Tipo	Derechos		1	
O	archivo	rw-	datos	2	
archivo	rw-	notas			

La posesión por parte de un miembro del grupo de profesores, como es juan, de una capacidad del objeto datos le permite efectuar operaciones de lectura y escritura sobre el mismo.

Un problema asociado a las capacidades, desde el punto de vista de la implementación, es que están en posesión de los usuarios y no asociadas al objeto en sí, por lo que hay que distribuirlas cuando se solicite la concesión de una capacidad por parte de un usuario. Para evitar problemas de

Digitalización realizada con propósito académico

seguridad, las listas de capacidades no suelen estar nunca accesibles directamente, sino que se acceden desde métodos controlados por el sistema operativo o el sistema de seguridad. La mayoría de los sistemas que usan capacidades se basan en el hecho de que las listas de capacidades están siempre dentro del espacio de memoria del sistema operativo, sin que exista posibilidad de migrar a espacio de memoria de los procesos de usuario. Además, las capacidades sólo pueden ser accedidas a través de métodos que proporciona el sistema operativo, tales como:

Crear capacidad
Destruir capacidad
Copiar capacidad

Es habitual que una capacidad no pueda ser modificada. En casi todos los sistemas es necesa rio destruirla y crear una nueva, siguiendo un esquema de uso de una única vez.

A nivel interno se han propuesto tres métodos para proteger las listas de capacidades:

• **Arquitectura etiquetada**, en la cual cada palabra de memoria tenga un bit de etiqueta adicional diciendo si contiene o no una capacidad. En caso positivo, esa posición de memo ria sólo puede ser modificada por procesos que ejecuten dentro del núcleo del sistema opera tivo. Esta solución es cara y poco adecuada para sistemas de propósito general.
• **Capacidades cifradas**. La clave de cifrado es desconocida por los usuarios, que deben limitarse a manipular las capacidades cifradas recibidas del sistema operativo. Este sistema se ajusta bien a las necesidades de los sistemas operativos distribuidos.
• **Listas de control de acceso** asociadas a cada capacidad.

El sistema operativo Amoeba [19861 usa capacidades para proteger puertos de mensajes y otros objetos. La capacidad está formada por cuatro elementos (Fig. 9.19), de los cuales dos, los derechos de acceso y el número aleatorio, se usan para protección. Los derechos de acceso definen un bit por cada operación permitida sobre el objeto, el número aleatorio permite distinguir entre distintas versiones de la misma capacidad. Todas las capacidades se cifran antes de ser distri buidas a los usuarios.

Las capacidades no se corresponden directamente con las necesidades de los usuarios y son menos intuitivas que las ACL. Debido a ello, la mayoría de los sistemas operativos proporcionan ACL como mecanismo de protección. Sin embargo, las capacidades tienen varias ventajas:

• Son muy útiles para incluir información de protección para un proceso en particular [19841.
• El mecanismo de comprobación de derecho es muy sencillo.
• Se adaptan muy bien a sistemas distribuidos.

Su gran desventaja sigue siendo que la revocación de accesos a un objeto puede ser ineficiente si se desea hacerla con criterios de selectividad entre dominios o para revocar derechos parciales.

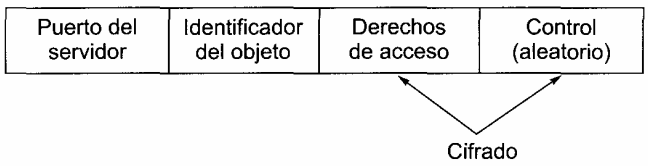


Figura 9.19. *Estructura de capacidad usada en el sistema operativo Amoeba.*

Digitalización realizada con propósito académico

Revocación de derechos de acceso

El principal problema de las capacidades es que, en un sistema dinámico, pueden existir cientos de capacidades concedidas para acceder a un objeto, lo que hace muy difícil su control al no existir una estructura centralizada en el mismo objeto como la ACL. Por ello, revocar los derechos de acceso para un objeto en particular es muy difícil, ya que el sistema debe buscar todas las capacidades existentes sobre el mismo para aplicar la revocación de derechos. El problema se complica si además se quiere tener revocación parcial de derechos.

En sistemas operativos centralizados se han propuesto métodos de control de capacidades tales como seguir la pista a la situación de las mismas, forzar su paso siempre a través del núcleo y mantener las capacidades en una base de datos centralizada, y restringida, limitar la propagación de capacidades, obligar a la readquisición periódica de la capacidad o usar claves de versión para cada capacidad [1991].

Si se implementan las capacidades como una lista de control de acceso, se puede mantener una lista desde cada objeto a todas sus capacidades. Si hay modificaciones, se recorre dicha lista y se aplican. Este método, usado en MULTICS, es muy flexible, pero costoso de implementar. Una variante es borrar las capacidades para un objeto periódicamente. Si un proceso quiere utilizarlo, debe adquirir una nueva capacidad. Si se le ha denegado el acceso, no será capaz de conseguirla.

Con el mecanismo de claves de versión, cada objeto tiene una clave maestra, generalmente definida como un número aleatorio con muchos bits, que se copia en cada nueva capacidad subte ese objeto. Los usuarios no pueden modificar dicha clave, que deben presentar junto a la capacidad cuando van a acceder al objeto. En ese momento, el sistema operativo compara la clave de la capacidad y la maestra. Si no coinciden, se deniega el acceso. Para revocar el acceso, lo único que hay que hacer es cambiar la clave maestra existente en el objeto. Normalmente, sólo el dueño de un objeto, o el sistema operativo, puede cambiar la clave maestra del mismo. Este método, usado en el sistema operativo Amoeba, tampoco permite revocaciones selectivas.

Cuando las capacidades se pueden ceder de unos usuarios a otros, para ceder permisos de acceso a un objeto, es necesario mantener un grafo o árbol que muestre el patrón de propagación de una capacidad y que permita viajar por el árbol para revocar la capacidad en todos los dominios de protección. Este mecanismo se puede implementar mediante indirecciones, de forma que las capa cidades no apunten directamente a los objetos, sino a una tabla global intermedia desde la cual se apunta al objeto. Para revocar las capacidades sobre un objeto sólo hay que eliminar el apuntador entre la tabla de objetos y el objeto en sí. No permite revocaciones selectivas.

9.8. SERVICIOS DE PROTECCIÓN Y SEGURIDAD

Los servicios de protección y seguridad de un sistema varían dependiendo de la complejidad del sistema implementado. En esta sección se muestran algunos servicios de protección genéricos en sistemas operativos clásicos, sin entrar en mecanismos de protección de sistemas distribuidos o en los problemas de distribución de claves por redes. Dichos servicios se concretarán para el estándar POSIX y para Windows NT, incluyendo además ejemplos de uso de las llamadas a ambos sistemas operativos.

9.8.1. Servicios genéricos

En general, todos los sistemas operativos crean la información de protección cuando se crea un objeto, por lo que no es muy frecuente encontrar servicios de creación y destrucción de ACL o

Digitalización realizada con propósito académico

capacidades disponibles para los usuarios. Es mucho más frecuente que los servicios de protección incluyan llamadas al sistema para cambiar características de la información de protección o para consultar dichas características. Sin embargo, para presentar una descripción más completa de ser vicios genéricos, se incluyen también los de creación y destrucción en la siguiente descripción:

• **Crear un descriptor de protección**. Este servicio permite crear un nuevo descriptor de protección que, inicialmente, puede no estar asignado a ningún objeto. Lo más usual, sin embargo, es que esté indisolublemente unido a un objeto y que su ciclo de vida vaya asocia do al del objeto, aunque puede ser más corto. En el caso de Windows NT, esta llamada crea un nuevo objeto de tipo testigo de acceso.
• **Destruir un descriptor de protección**. Elimina del sistema un descriptor de protección. No lo hace para aquellos objetos asociados al descriptor que están siendo utilizados (abiertos). En el caso de usar capacidades, éste puede ser un medio de revocación general de accesos posteriores al objeto.
• **Abrir un descriptor de protección.** Permite abrir un descriptor, creado previamente, para usarlo asociado a un objeto.
• **Obtener información de protección**. Este servicio permite a los usuarios conocer la infor mación de protección de un objeto. Para ello deben tener permisos de acceso al descriptor de protección que quieren consultar.
• **Cambiar información de protección**. Este servicio permite a los usuarios cambiar la infor mación de protección de un objeto. Para ello deben tener permisos de acceso y modificación del descriptor de protección que quieren modificar. Normalmente, los descriptores incluyen información acerca del dueño del objeto, su dominio, sus privilegios, etc. Muchas de estas características son modificables.
• **Fijar información de protección por defecto**. Este servicio permite fijar máscaras de pro tección que se aplican por defecto a los objetos de un dominio. De esta forma, los usuarios pueden asegurarse de que sus recursos tienen unas características de protección prefijadas, incluso aunque se creen con descriptores de protección vacíos.

A continuación, se detallan los servicios de protección existentes en POSIX y Windows NT.

9.8.2. Servicios POSIX

El estándar POSIX define servicios que, en general, se ajustan a los servicios genéricos descritos en la sección anterior. Sin embargo, no existen servicios específicos para crear, destruir o abrir des criptores de protección. Estos se asocian a los objetos y se crean y se destruyen con dichos objetos.

Comprobación de la posibilidad de acceder a un archivo

access comprueba si un archivo está accesible con unos ciertos privilegios. Tienen en cuenta el uid y el gid real, no los efectivos. No es necesario tener el archivo abierto.
mt access (const char *path, mt amode)

El nombre del archivo se especifica en path. amode es el OR binario de los permisos de acceso a comprobar o la constante F_OK si se quiere comprobar que el archivo existe. En caso de éxito devuelve un cero. En caso de error devuelve —1.

Digitalización realizada con propósito académico

Cambio del modo de protección de un archivo

chinoc3. cambia los derechos de acceso a un archivo. Sólo el dueño del archivo o el superusuario pueden ejecutar esta llamada. No es necesario tener el archivo abierto. Si algún proceso tiene el archivo abierto, esta llamada no afectará a sus privilegios de acceso hasta que lo cierre.

mt chmod (const char *path, mode_t mode);

El nombre del archivo se especifica en path. mode es el valor, en notación octal, de los permisos de acceso a instalar. Por ejemplo, los bits rwxrx-r-- se indican en octal con los números 764. Permite cambiar los bits setuid y setgid. En caso de éxito devuelve un cero. En caso de error un —1.

Cambio del propietario de un archivo

chown cambia el propietario y el grupo de un archivo. Sólo el dueño del archivo o el superusuario pueden cambiar estos atributos. No es necesario tener el archivo abierto. Si algún proceso tiene el archivo abierto, esta llamada no afectará a sus privilegios de acceso hasta que lo cierre.

mt chown (const char *path, uidt owner, gid_t group);

El nombre del archivo se especifica en path. owner y group son los identificadores numéri cos del nuevo dueño y del nuevo grupo. En caso de éxito devuelve un cero. En caso de error devuelve —l y no cambia nada.

Obtención de los identificadores del propietario y de su grupo

Las llamadas getuid, geteuid, getgid y getegid permiten obtener los identificadores reales y efectivos del propietario de un archivo y de su grupo. Estas llamadas son sólo de consulta y no modifican nada.

uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);

En caso de éxito devuelven el identificador solicitado. Estas funciones siempre deben resol- verse con éxito, por lo que no hay previsto caso de error. Esto se debe a que todo proceso debe i ejecutar siempre con una identificación de usuario y de grupo.

Cambio de los identificadores del propietario y de su grupo

Las llamadas setuid y setgid permiten cambiar los identificadores reales del usuario y de su grupo. A partir de ese momento, estas identidades pasan a ser la identidades efectivas de los mis mos. Sus identidades reales sólo se cambian si el usuario tiene los privilegios adecuados. En otro caso, sólo se cambia la efectiva y se mantiene la identidad real.

uid_t getuid (uid_t uid);
gid_t setgid (gid_t gid);

Digitalización realizada con propósito académico

En caso de éxito devuelven cero. En caso de error, por privilegios insuficientes o por no existir la identidad solicitada, devuelven —1.

Definir la máscara de protección por defecto

La llamada umask permite a un usuario definir una máscara de protección que será aplicada por defecto a todos sus objetos creados a partir de ese instante.

mode_t uiuask (mode_t cmask);

El parámetro cmask define el modo de protección por defecto. Cuando se crea un objeto y se define su modo de protección, por ejemplo model, el valor efectivo de protección resultante para el objeto es el resultado del OR binario exclusivo entre cmask y model. Por ejemplo:

cmask = 022 model = 777
(rwxrwxrwx)

daría un modo de protección real de 755 (rwxr-xr-x).

En caso de éxito devuelve el valor del modo de protección de los archivos anterior a este cambio.

Otras llamadas POSIX

El estándar POSIX define algunas llamadas más relacionadas con la identificación de usuarios y las sesiones de trabajo de los mismos. La llamada getgroups permite obtener la lista de grupos de un usuario. Las llamadas getiogin y getlogin_r devuelven el nombre del usuario asociado a un proceso. La llamada setsid crea un identificador de sesión para un usuario. La llamada uname permite identificar al sistema en el que se ejecuta el sistema operativo. Para una referencia más completa de estas llamadas se remite al lector a los manuales de descripción del estándar.

9.8.3. Ejemplo de uso de los servicios de protección de POSIX

Para ilustrar el uso de los servicios de protección que propQrciona POSIX, se presenta en esta sección dos pequeños programas, con su código fuente en lenguaje C. El primero (Programa 9.1) ejecuta la siguiente secuencia de acciones:

1. Comprueba que el archivo origen se puede leer y que la identificación efectiva y real del usuario son las mismas.
2. Crea el archivo destino con la máscara por defecto.
3. Copia el archivo antiguo al nuevo y restaura el modo de protección anterior.
4. Cambia el propietario y el grupo del archivo destino.

Programa 9.1. Copia con cambio del modo de protección de un archivo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
```

Digitalización realizada con propósito académico

```
#define FIC_OR        "/tmp/fic_origen"
#define FIC_DES       "/tmp/fic-destino"
#define NEW_OWN       512  /* alumno miguel */
#define NEW_GRP       500  /* alumnos */

main (int argc, char **argv)
{
   int oldmask;

   if (getuid() != geteuid()) {
      printf ("Error: las identidades no coinciden \n");
      exit(0);
   }
   if (access(FIC_OR, R_OK) == -1) {
      printf ("Error: %s no se puede leer \n", FIC_OR);
      exit(0);
   }
   /* Comprobaciones bien. Crear el destino con máscara 007 */
   oldmask = umask (0007);
   if ( creat (FIC_DES, 0755) < 0) {
      printf ("Error: %s no pudo ser creado \n", FIC_DES);
      exit(0);
   }
   /* Restaurar la máscara de protección anterior */
   umask (oldmask);
   /* Si todo fue bien, el modo de protección efectivo es 750 */
   system ("cp /tmp/fic-origen /tmp/fic-destino");
   /* Cambiar su dueño y su grupo. Sólo lo puede hacer el superusuario */
   chown (FIC_DES, NEW_OWN, NEW_GRP);
   exit (0);
}
```

El Programa 9.2 permite consultar la identidad del dueño, de su grupo y los derechos de acceso de un archivo, cuyo nombre recibe como parámetro de entrada, usando la llamada al sistema stat. Este ejemplo no modifica nada, sólo extrae los parámetros y se los muestra al usuario. Para ello, el programa ejecuta la siguiente secuencia de acciones:

1. Ejecuta la llamada stat para el archivo solicitado y comprueba si ha recibido un error.
2. Si no le ha devuelto un error, extrae de la estructura de datos devuelta como parámetro de salida (struct stat) la información pedida. El identificador del dueño está en el campo st_uid, el de su grupo en st_gid y los permisos de acceso en st_mode.
3. Da formato a los datos obtenidos y los muestra por la salida estándar.

Programa 9.2. Extracción de la identidad del dueño, de su grupo y los derechos de acceso de un archivo.

```
#include <sys/types.h>
#include <stat.h>

main (int argc, char **argv)
{
    struct stat InfArchivo;  /* Estructura para datos de stat */
    int fd, EstadoOperacion, i;
```

```
    unsigned short Modo;        /* Modo de protección */
    char NombreArchivo[128];    /* Longitud máxima arbitraria: 128 */

    /* Se copia el nombre del archivo a la variable NombreArchivo */
    sprintf (NombreArchivo, "%s", argv[1]);
    /* Se escribe el nombre por salida estándar */
    printf ("\n %s", NombreArchivo);

    EstadoOperacion = stat (NombreArchivo, &InfArchivo);
    if (EstadoOperacion != 0) {
       printf ("Error: %s nombre de archivo erróneo \n", NombreArchivo);
       exit(-1);
    } else {
       /* Se escribe el identificador de usuario por salida estándar */
       printf ("uid: %d", InfArchivo.st_uid);
       /* Se escribe el identificador de grupo por salida estándar */
       printf ("gid: %d", InfArchivo.st_gid);
       /* Se toman grupos de 3 bits del modo de protección. Los
          tres primeros son del dueño, los siguientes del grupo
          y los siguientes del mundo. Para estas operaciones se
          usan máscaras de bits y desplazamiento binario*/
       for (i=6; i>=0; i = i-3) {
          Modo = InfArchivo.st_mode >> i;
          if (Modo & 04) printf ("r"); else printf ("-");
          if (Modo & 02) printf ("w"); else printf ("-");
          if (Modo & 01) printf ("x"); else printf ("-");
       }
       printf ("\n");
    }
    exit (0);
}
```

Nota: Se puede obtener información detallada de la llamada al sistema s tat y de la estructura de datos que maneja mediante el manual de usuario del sistema (man s tat).

9.8.4. Servicios de Win32

Windows NT tiene un nivel de seguridad C2 según la clasificación de seguridad del Orange Book del DoD, lo que significa la existencia de control de acceso discrecional, con la posibilidad de permitir o denegar derechos de acceso para cualquier objeto partiendo de la identidad del usuario que intenta acceder al objeto. Para implementar el modelo de seguridad, Windows NT usa un descriptor de seguridad y listas de control de acceso (ACL), que a su vez incluyen dos tipos de entradas de control de acceso (ACE): las de permisos y las de negaciones de accesos.
Las llamadas al sistema de Win32 permiten manipular la descripción de los usuarios, los descriptores de seguridad y las ACL. A continuación se describen las llamadas de Win32 más frecuentemente usadas.

Dar valores iniciales a un descriptor de seguridad

El servicio InitializeSecurityDescriptor inicia el descriptor de seguridad especificado en psd con unos valores de protección por defecto.

Digitalización realizada con propósito académico

BOOL InitializeSecurityDescriptor
(PSECURITY_DESCRIPTOR psd, DWORD dwRevision)

dwRevision tiene el valor de la constante SECURITY_DESCRIPTOR_REVISION. Esta llamada devuelve TRUE si psd es un descriptor de seguridad correcto y FALSE en otro caso.

Obtención del identificador de un usuario

La llamada GetUserName permite obtener el identificador de un usuario que ha accedido al sistema.

BOOL GetUserNaine (LPTSTR NornbreUsuario, LPDWORD LongitudNombre);

En caso de éxito, devuelve el identificador solicitado en NornbreUsuario y la longitud del nombre en LongitudNombre. Esta función siempre debe resolverse con éxito, por lo que no hay previsto caso de error.

Obtención de la información de seguridad de un archivo

El servicio GetF±leSecur±ty extrae el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL GetFilesecurity (LPCTSTR NombreArchivo, SECURITY_INFORNATION secinf o, PSECTJRITY_DESCRIPTOR psd, DWORD cbSd, LPDWORD lpcbLongitud);

El nombre del archivo se especifica en NombreArchivo. secinfo es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. El argumento psd es el descriptor de seguridad en el que se devuelve la información de seguridad. Esta llamada devuelve TRUE si todo es correcto y FALSE en otro caso.

Cambio de la información de seguridad de un archivo

El servicio SetFileSecurity fija el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto.

BOOL SetFi].eSecurity (LPCTSTR NombreArchivo, SECURITYINFORMATION secinf o, PSECURITY_DESCRIPTORpsd);

El nombre del archivo se especifica en NombreArchivo. El argumento secinfo es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discreccional o la del sistema. El argumento psd es el descriptor de seguridad en el que se pasa la información de seguridad. Esta llamada devuelve TRUE si todo es correcto y FALSE en otro caso.

Obtención de los identificadores del propietario y de su grupo para un archivo

Las llamadas GetSecurityDescriptorOwner y GetSecurityDescriptorGroup permiten extraer la identificación del usuario de un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. Estas llamadas son sólo de con sulta y no modifican nada.

Digitalización realizada con propósito académico

```
BOOL GetSecurityDescriptorOwner (PSECURITY_IJESCRIPTOR psd,
PSID psidOwner);
BOOL GetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd, PSID psidGroup);
```

El descriptor de seguridad se especifica en psd. El argumento psidOwner es un parámetro de salida donde se obtiene el identificador del usuario y psidGroup es un parámetro de salida donde se obtiene el grupo del usuario. En caso de éxito devuelve TRUE y si hay algún error FALSE.

Cambio de los identificadores del propietario y de su grupo para un archivo

Las llamadas SetSecurityDescriptorOwner y SetSecurityDescriptorGroup permiten modificar la identificación del usuario en un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo.

```
BOOL SetsecurityDescrjptorowner (PSECURITY_DESCRIPTOR psd,
                                 PSID psidOwner, BOOL fOwnerDefaulted);
BOOL SetSecurityDescriptorGroup (PSECURITY_DESCRIPTOR psd,
                                 PSIDpsidGroup, BOOL fGroupDefaulted);
```

El descriptor de seguridad se especifica en psd. El argumento PS idOwner es un parámetro de entrada donde se indica el identificador del usuario y ps ±dGroup es un parámetro de entrada donde se indica el grupo del usuario. El último parámetro de cada llamada especifica que se debe usar la información de protección por defecto para rellenar ambos campos. En caso de éxito devuelve TRUÉ y si hay algún error FALSE.

Gestión de ACLs y ACEs

Las llamadas InitializeAcl, AddAccessAllowedAceyAddAccessDeniedAce permiten inializar una ACL y añadir entradas de concesión y denegación de accesos.

```
BOOL  Initia].izeAc]. (PACL pAci, DWORD cbAcl, DWORD dwAclRevision);
```

pAcl es la dirección de una estructura de usuario de longitud cbAcl. El último parámetro debe ser ACL_REVISION.

La ACL se debe asociar a un descriptor de seguridad, lo que puede hacerse usando la llamada SetSecurityDescriptorDacl.

```
BOOL SetSecurityDescriptorDacl (PSECURITY_DESCRIPTOR psd,
                                BOOL fDaciPresent, PACL pAci,
                                BOOL fDaclDefaulted)
```

psd incluye el descriptor de seguridad. El argumento fDaclPresent a TRUE indica que hay una ACL válida en pAci. fdaciDefaulted a TRUE indica que se debe iniciar la ACL con valores por defecto.

AddAccessAllowedAce y AddAccessDeniedAce permiten añadir entradas con concesión y denegación de accesos a una ACL.

```
BOOL AddAccessAllowedAce (PACL pAci, DWORD dwAciRevision, DWORD
                          clwAccessMask, PSID psid);

BOOL AddAccessDeniedAce (PACL pAci, DWORD dwAclRevision,
                         DWORD dwAccessMask, PSID pSid);
```

Digitalización realizada con propósito académico

pAci es la dirección de una estructura de tipo ACL, que debe estar iniciada. El argumento dwAclRevis ion debe ser ACLREVISION. El argumento pSid apunta a un identificador válido de usuario. El parámetro dwAccessMask determina los derechos que se conceden o deniegan al usua rio o a su grupo. Los valores por defecto varían según el tipo de objeto.

Otras llamadas de Win32

Win32 proporciona algunas llamadas más relacionadas con la identificación de usuarios. Looku pAccountName permite obtener el identificador de un usuario mediante un nombre de cuenta suya. LookupAccountSid permite obtener el nombre de la cuenta de un usuario a partir de su identifica dor. Además incluye llamadas para obtener información de las ACL (GetAclInformation), ob tener las entradas de una ACE (GetAce) y borrarlas (DeleteAce).

Existe un conjunto de llamadas similar al mostrado para proporcionar seguridad en objetos privados de los usuarios, tales como sockets o bases de datos propietarias. CreatePrivateOb jectSecurity, SetPrivateObjectSecurity y GetPrivateObjectSecurity son algunas de estas funciones.

También es posible proteger objetos del núcleo, tales como la memoria. CreateKernelOb jectSecurity, SetKernelObjectSecurity y GetKernelObjectSecurity son algunas de estas funciones.

9.8.5. Ejemplo de uso de los servicios de protección de Win32

Para ilustrar el uso de los servicios de protección que proporciona Win32, se presenta en esta sección dos pequeños programas en lenguaje C que consultan y manipulan los descriptores de seguridad de un objeto.

El Programa 9.3 lee los atributos de seguridad de un archivo. Para ello extrae primero la longitud del descriptor de seguridad del archivo y luego el propio descriptor de seguridad, que devuelve como salida de la función. Como ejercicio se sugiere al lector que modifique este progra ma para extraer la ACL del archivo y cada una de sus entradas (ACEs).

Programa 9.3. Lectura de los atributos de seguridad de un archivo.

```
#include <windows.h>

PSECURITY_DESCRIPTOR LeerPermisosDeUnArchivo (LPCTSTR NombreArchivo)
   /* Devuelve los permisos de un archivo */
{
   PSECURITY_DESCRIPTOR pSD = NULL;
   DWORD Longitud;
   HANDLE ProcHeap = GetProcessHeap ();
   /* Obtiene el tamaño del descriptor de seguridad. */
   GetFileSecurity (NombreArchivo, OWNER_SECURITY_INFORMATION |
      GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
      pSD, 0, &Longitud);
   /* Obtiene el descriptor de seguridad del archivo*/
   pSD = HeapAlloc (ProcHeap, HEAP_GENERATE_EXCEPTIONS, Longitud);
   if (!GetFileSecurity (NombreArchivo, OWNER_SECURITY_INFORMATION |
```

```
            GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
            pSD, Longitud, &Longitud))
        printf ("Error GetFileSecurity \n");
    return pSD;
}
```

El Programa 9.4 crea un descriptor de seguridad y le asigna los valores de protección por defecto. Para ello extrae primero la longitud del descriptor de seguridad y del componente de la ACL, los crea y posteriormente asigna valores. Se sugiere al lector que siga los comentarios del programa como guía para comprender lo que hace.

Programa 9.4. Creación de un descriptor de seguridad con protección por defecto.

```
#include <windows.h>

int main (int argc, char** argv)
{
    /* Declaración de contenedores para longitudes */
    DWORD LongitudDACL;
    DWORD ÇodigoError;
    PSID SegProcSegId = NULL;
    /* Declaración de contenedores para descriptores de seguridad */
    PACL pDACL;
    PSECURITY_DESCRIPTOR m_pSD = NULL;
    /* Definición de autoridad identificadora */
    SID_IDENTIFIER_AUTHORITY siaWorld = SECURITY_WORLD_SID_AUTHORITY;
    int LongSidTemp;

    /* Asignación de memoria para descriptor de seguridad */
    m_pSD = malloc(sizeof(SECURITY_DESCRIPTOR));
    if (!m_pSD) {
        SetLastError(ERROR_NOT_ENOUGH_MEMORY);
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }/* Iniciación de valores en el descriptor de seguridad */
    if (!InitializeSecurityDescriptor
        (m_pSD,SECURITY_DESCRIPTOR_REVISION)) {
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    /* Obtención de la longitud en bytes para almacenar un identificador
    de seguridad para una autoridad de identificación y del espacio de
    memoria correspondiente para el identificador */
    LongSidTemp = GetSidLengthRequired(1);
    SegProcSegId = (PSID)malloc(LongSidTemp);
    if (!SegProcSegId) {
        SetLastError(ERROR_NOT_ENOUGH_MEMORY);
        CodigoError = GetLastError();
```

Digitalización realizada con propósito académico

```
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
}
    /* Obtención de espacio de memoria para la ACL */
    LongitudDACL = sizeof (ACL) +sizeof (ACCESS_ALLOWED_ACE) - sizeof
    (DWORD) + LongSidTemp;
    pDACL = (PACL) malloc(LongitudDACL);
    if (!pDACL) {
        SetLastError(ERROR_NOT_ENOUGH_MEMORY);
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    /* Valores iniciales para la ACL y el SID. Si cualquiera falla, se
        origina un error */
    if (!InitializeAcl(pDACL,LongitudDACL,ACL_REVISION)
        ||!InitializeSid(SegProcSegId,&siaWorld,1)) {
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    /* Extracción de la autoridad en el descriptor de seguridad */
    *(GetSidSubAuthority(SegProcSegId,0)) = SECURITY_WORLD_RID;
    /* Añadir una entrada de control de acceso a la ACL con valores por
        defecto. También se incluye el identificador de seguridad */
    if(!AddAccessAllowedAce(pDACL,ACL_REVISION,GENERIC_ALL|STANDARD_RIGHT
    S_ALL|SPECIFIC_RIGHTS_ALL,SegProcSegId)) {
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    /* Conectar la ACL creada con el descriptor de seguridad creado */
    if (!SetSecurityDescriptorDacl(m_pSD,TRUE,pDACL,FALSE)) {
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    printf ("\n Descriptor de seguridad con protección por defecto \n");
}
```

Como ejercicio se sugiere al lector que modifique este programa para asignar permisos especí ficos en el momento de la creación del descriptor de seguridad. Estos permisos se pueden pasar como parámetros del proceso.

9.9. EL SISTEMA DE SEGURIDAD DE WINDOWS NT

La seguridad se incluyó como parte de las especificaciones de diseño de Windows NT, lo que permite lograr un nivel de seguridad C2 si se configura el sistema adecuadamente. El modelo de seguridad incluye componentes para controlar quién accede a los objetos, qué accesos pueden

Digitalización realizada con propósito académico

efectuar los usuarios sobre un objeto y qué eventos se auditan. Todos estos componentes juntos componen el subsistema de seguridad de Windows NT.

Como se muestra en la Figura 9.20, el modelo de seguridad de Windows NT incluye los siguientes componentes:

• Procesos de logon, que muestran las ventanas de diálogo para que los usuarios puedan acce der al sistema, piden el identificador del usuario, su palabra clave y su dominio.
• Autoridad de seguridad local, que controla que el usuario tenga permiso para acceder al sistema. Es el corazón del sistema porque gestiona la política local, los servicios de autenti cación, política de auditoría y registro de eventos auditados.
• Gestor de cuentas de usuario, que mantiene las base de datos de usuarios y grupos. Propor ciona servicios de validación de usuarios.
• Monitor de referencia de seguridad, que controla los accesos de los usuarios a los objetos para ver si tienen los permisos apropiados aplicando la política de seguridad y genera even tos para los registros de auditoría.

El modelo de seguridad mantiene información de seguridad para cada usuario, grupo y objeto del sistema. Puede identificar accesos directos de un usuario y accesos indirectos a través de proce sos que ejecutan en representación del usuario. Permite a los usuarios asignar permisos a los objetos de forma discrecional, pero si el usuario no asigna estos permisos, el subsistema de seguridad asigna permisos de protección por defecto.

Cada usuario se identifica en este sistema mediante un identificador de seguridad único (SID, Security ID) durante la vida del usuario dentro del sistema. Cuando un usuario accede al sistema, la autoridad de seguridad local crea un descriptor de seguridad para acceso, lo que incluye una identi ficación de seguridad para el usuario y otra para cada grupo al que pertenece el usuario. Además,

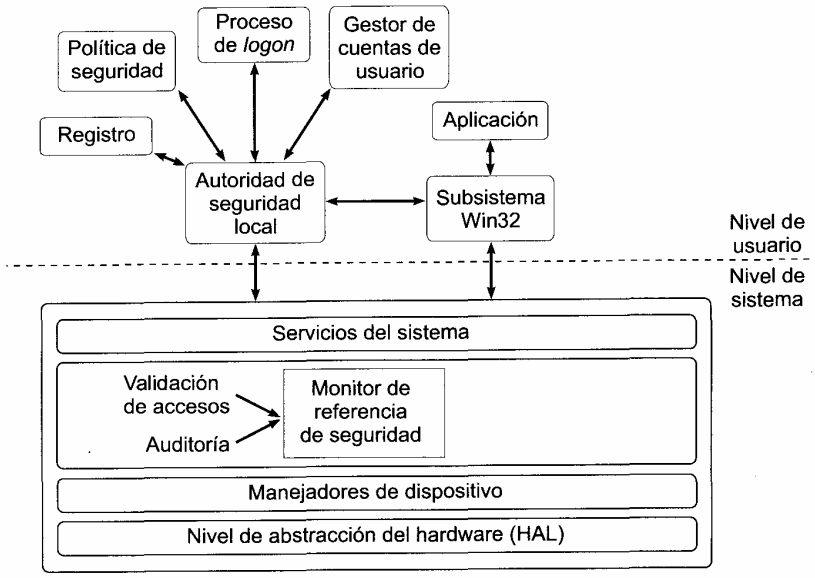


Figura 9.20. *Arquitectura del sistema de seguridad de Windows NT.*

Digitalización realizada con propósito académico

cada proceso que ejecuta en nombre del usuario tiene una copia de descriptor de seguridad del usuario.

Cuando el usuario trata de acceder a un objeto con una máscara de protección, por ejemplo, READ-WRITE, su SID se compara con los existentes en la ACL del objeto para verificar que el usuario tiene permisos suficientes para efectuar la operación de acceso. Para ello, el Monitor de Referencia de Seguridad evalúa cada entrada de la ACL, denominada ACE (Access Control Entry), de la siguiente manera:

1. Si el SID de la ACE y el del usuario no coinciden, se salta a la siguiente ACE.
2. Si coinciden, se comprueba la máscara de acceso solicitada con los derechos de acceso existentes en la ACE.
3. Si la ACE es de negación de acceso y las máscaras tienen algo en común, se deniega el acceso.
4. Si la ACE es de negación y no coinciden las máscaras, se sigue investigando.
5. Si la ACE es confirmación de acceso y ambas máscaras coinciden totalmente, se permite el acceso.

Las utilidades de registro de eventos permiten recolectar información sobre los accesos de los usuarios a los objetos y monitorizar eventos relacionados con el sistema de seguridad. El registro permite identificar fallos de seguridad y determinar la extensión de los daños, si se han producido. El nivel de auditoría se puede configurar para ajustarlo a las necesidades de cada instalación.

9.10. KERBEROS

Kerberos [19871 es un sistema de seguridad para sistemas conectados en red, aunque se puede aplicar a sistemas operativos convencionales, como Solaris y Windows 2000. Fue desarrolla do en el Massachusetts Institute of Technology (MIT) a finales de los años ochenta y se ha converti do en un estándar defacto en sistemas operativos [1993]. Se basa en el uso de claves privadas, protocolos simétricos de intercambio de claves y en la existencia de servidores de claves y tickets. Cada usuario del sistema tiene su clave y el servidor de Kerberos usa dicha clave para codificar los mensajes que envía a dicho usuario de forma que no puedan ser leídos por nadie más.

Para proporcionar los servicios de autenticación, Kerberos mantiene una base de datos con sus clientes y sus respectivas claves privadas. La clave privada es un número muy grande que sólo conoce el servidor de claves de Kerberos y el cliente dueño de dicha clave. Si el cliente es un usuario, dicha clave se almacena cifrada. Los servicios del sistema que requieren autenticación, y los clientes que quieren usar dichos servicios, deben registrarse en el sistema Kerberos. En ese momento se establecen las claves privadas para cada uno de ellos. Puesto que Kerberos tiene las claves de los usuarios, puede generar mensajes que convenzan a los mismos de que otro usuario es quien dice ser. Se dice que Kerberos es un sistema confiable en el sentido en que los usuarios creen que lo que dice Kerberos acerca de las identidades de otros usuarios es exacto. Para incrementar la seguridad, el sistema genera también claves temporales o de sesión, que se dan únicamente a los clientes que se quieren comunicar entre sí. Dichas claves se usan para cifrar los mensajes entre los dos clientes durante una sesión de trabajo.

Kerberos proporciona tres niveles de protección distintos:

• **Autenticación al inicio de una sesión**. Este servicio es el usado en sistemas operativos convencionales. Con él se asume que una vez establecida la sesión, los accesos siguientes se llevan a cabo por parte del cliente autenticado.

Digitalización realizada con propósito académico

• **Autenticación de cada acceso.** En este caso se comprueba que cada acceso es efectuado por el usuario que inició la sesión. No se preocupa de ocultar el contenido de los mensajes, sino de proporcionar mensajes seguros.
• **Autenticación y ocultación.** En este caso se comprueban todos los accesos y se cifran los mensajes con la clave de sesión de las dos partes en contacto proporcionando mensajes privados.

La arquitectura de Kerberos, que se muestra en la Figura 9.2 1, es compleja e incluye los siguientes componentes:

• **Biblioteca de aplicación**. Interfaz entre las aplicaciones de los clientes y los servidores. Incluye rutinas para crear autenticaciones, leerlas, generar mensajes seguros o privados, etcétera.

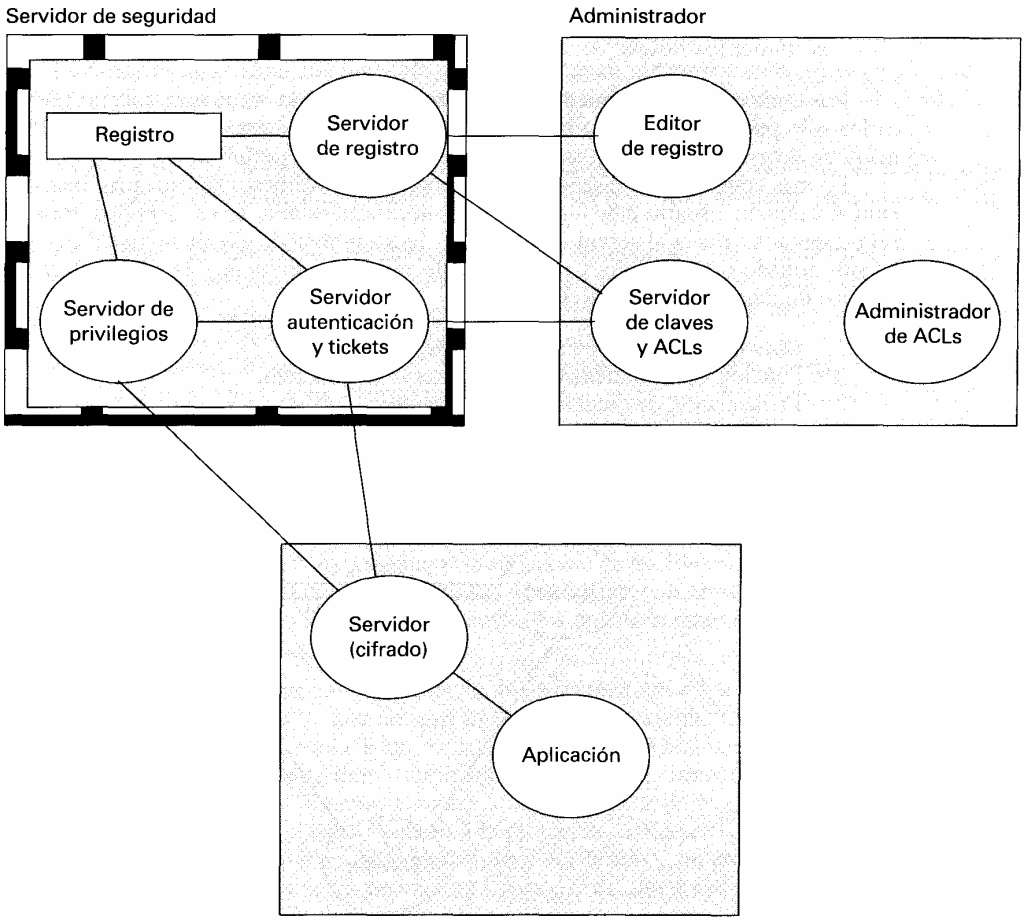


Figura 9.21. *Arquitectura de Kerberos.*

Digitalización realizada con propósito académico

• **Biblioteca de cifrado**. Incluye las rutinas para cifrar y descifrar. Se basa en el estándar DES y proporciona varios métodos alternativos de cifrado con características distintas de veloci dad y seguridad (CBC, PCBC, etc.).
• **Sistema de gestión de base de datos**. Módulo reemplazable que proporciona los servicios de bases de datos. Los servicios que necesita Kerberos son sencillos: por cada registro se guarda el nombre del usuario, la clave privada y la fecha de expiración de la misma. Además puede existir información adicional de los usuarios, tal como su nombre, teléfono, etc. En este caso, la información sensible se trata con medidas de seguridad mayores.
• **Servidor de registro,** o módulo de entrada, que proporciona la interfaz de cliente para acceder al sistema. La parte del cliente se puede ejecutar en cualquier sistema, pero la parte
del servidor debe ejecutar en la máquina donde se ha instalado la base de datos de Kerberos.
• El **servidor de autenticación** ejecuta operaciones de sólo lectura en la base de datos de Kerberos para adquirir los datos de los usuarios. Además genera claves de sesión y tickets, Puesto que no modifica la base de datos puede ejecutar en cualquier sistema que tenga una copia de sólo lectura de la base de datos.
• Existen además programas de usuario que permiten entrar a Kerberos, cambiar una clave o gestionar los tickets.

Los componentes que pueden escribir sobre la base de datos para generar registros o modifi carlos sólo pueden ejecutar en la misma máquina en la que está instalada la base de datos. Dicha máquina debe estar instalada en un entorno seguro y ser confiable.

Lo más complejo del diseño e implementación de Kerberos son sus protocolos de autentica. ción. Cuando un usuario pide un servicio, es necesario establecer su identidad. Para ello es necesa rio presentar un ticket al servidor, junto con alguna prueba de que dicho ticket es legal y de que no ha sido robado o falsificado. Como se puede ver en las Figuras 9.21 y 9.22, hay tres fases en el proceso de autenticación de Kerberos:

1. Obtención de credenciales por parte del usuario.
2. Petición de autenticación para un servicio concreto.
3. Presentación de credenciales al servidor.

La credencial básica en Kerberos es el ticket. Un ticket permite transmitir la identidad del dueño del ticket de forma segura, además de poder incluir información para identificar a los recep.

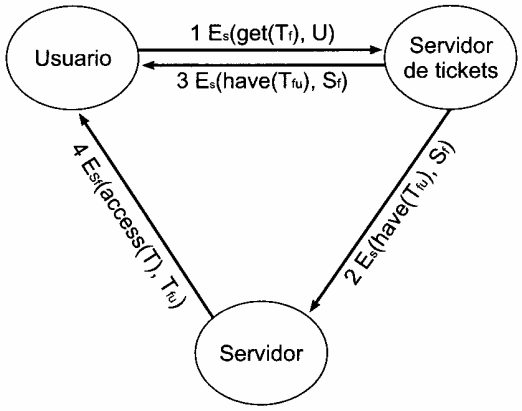


Figura 9.22. *Servidor de tickets en Kerberos.*

Digitalización realizada con propósito académico

tores del mismo. Entre otras cosas incluyen el nombre del servidor, el del cliente, un sello temporal, su duración y una clave aleatoria de sesión. Toda esta información se cifra usando la clave del servidor para el que se usa el ticket, lo que evita tener que comprobar si hay modificaciones del ticket. Los tickets tienen una duración temporal limitada, debiendo renovarse cuando expira.

Cuando un usUario quiere adquirir un ticket inicial (obtener credencial), debe acceder al servi dor de autenticación a través de un proceso de login. Cuando introduce su nombre y clave se codifican con la clave privada del cliente y se envían dichos datos al servidor de autenticación, que los contrasta con sus registros de la base de datos. Si todo es correcto, genera una clave aleatoria para. esa sesión de trabajo y un ticket para que el servidor de tickets reconozca al usuario y le conceda credenciales para servicios concretos cuando las pida. Para pedir credenciales para un servicio concreto en un servidor, el usuario envía un mensaje al servidor de tickets (paso 1 de la Figura 9.22). Este comunica con el servidor y el usuario y les envía el ticket concedido (pasos 2 y 3 de la Figura 9.22). A continuación, el servidor se pone en contacto con el usuario para indicarle que puede acceder a sus servicios usando dicho ticket (paso 4 de la Figura 9.22). Todos los mensajes van codificados con las claves privadas del servidor o de la sesión correspondiente. Cada vez que el usuario quiera acceder al servidor, debe presentar las credenciales concedidas para efectuar esas operaciones.

Kerberos es un sistema ampliamente usado en sistemas operativos convencionales y distribui dos. Sin embargo, no está exento de complicaciones y cuestiones pendientes de resolver llBellovin, 19911. Algunos de los problemas asociados con Kerberos son los siguientes:

• Los protocolos de intercambio de tickets son complicados, lo que conlieva un tiempo consi derable de ejecución si se tiene en cuenta que hay que pedir un ticket para cada servicio y servidor.
• La seguridad de las bases de datos del servidor de autenticación debe ser muy estricta para que no haya accesos indebidos ni maliciosos. Este requisito se contradice con el rendimien to. Para tratar de acercar ambas cuestiones, se replican las bases de datos con copias de sólo lectura y con acceso controlado por servidores de seguridad.
• La duración de los tickets supone también un compromiso entre seguridad y eficiencia. Por seguridad, lo mejor sería usar un ticket para cada operación. Por eficiencia, es mejor tener tickets con una vida larga para evitar las operaciones de petición de uno nuevo. Sin embargo, si el ticket fuese usado de forma indebida, los daños serían mayores cuanto más larga fuera la vida de dicho ticket.
• Es muy difícil garantizar la integridad del software que ejecuta en una computadora. La única solución posible es dificultar la modificación del software que se ejecuta en computa doras con múltiples usuarios mediante la aplicación de los controles descritos en este capítulo.
• Otro problema importante es la dificultad de delegar tareas entre servidores. Para conseguir esto, habría que modificar mucho los protocolos de intercambio de tickets o compartir infor mación sensible de las bases de datos.

A pesar de todo lo anterior, Kerberos es bien aceptado por los usuarios de un sistema de computación porque, si todo va bien, es casi totalmente transparente. Cuando accede al sistema, el usuario ve una interfaz similar a la de un sistema convencional. Igualmente, las operaciones de cam bio de palabra clave efectúan de forma automática los cambios pertinentes en los datos de Kerberos. Sólo cuando está dentro del sistema durante un tiempo mayor de lo que dura el ticket de sesión (normalmente varias horas) el usuario observa la presencia de Kerberos, que le obliga a adquirir un nuevo ticket de sesión. El trabajo del administrador del sistema, sin embargo, se complica sustan cialmente. Debe efectuar todas las operaciones para dar de alta a nuevos usuarios y servicios del sistema, replicar las bases de datos y asegurar que las máquinas que ejecutan los componentes centrales de Kerberos son físicamente seguras y confiables.

Digitalización realizada con propósito académico

9.11. PUNTOS RECORDAR

EJ La seguridad de un sistema tiene múltiples facetas, in cluyendo aspectos tales como la protección ante posi bles daños físicos de los datos (fuegos, terremotos, rotu ra del soporte de almacenamiento, etc.), hasta el acceso indebido a los mismos (intrusos, fallos de confidenciali dad, etc.).
EJ La protección, sin embargo, consiste en evitar que se haga un uso indebido de los datos cuando se está dentro del ámbito de diseño del sistema operativo.
EJ Los mecanismos de protección del sistema operativo sólo resuelven una mínima parte de los problemas de seguridad.
EJ Algunos problemas de seguridad están relacionados con el uso indebido o malicioso de programas de un usuario. Dos formas frecuentes de generar fallos de seguridad usando estas técnicas son el caballo de Troya y la puerta de atrás.
EJ Otros problemas de seguridad están relacionados con la posibilidad de cargar programas externos en un sistema. La existencia de sistemas de arquitectura abierta origina muchas situaciones azarosas en el entorno de seguridad. Los administradores siempre prefieren tener sistemas sin interferencias externas, de forma que se pueda evitar que programas externos entren en dominios de otros usuarios.
EJ La existencia de sistemas abiertos da lugar a cuatro ti pos principales de ataques de seguridad: virus, gusanos, rompedores de claves y bombardeos. Todos ellos están relacionados con la capacidad de propagación de unos sistemas a otros, lo que permite atacar desde sistemas remotos o enviar programas de ataque a otros sistemas.
U Añadir aspectos de seguridad al diseño del sistema ope rativo incrementa sustancialmente la complejidad de di cho diseño. Las consideraciones de seguridad deben ex tenderse al diseño de cada aspecto del sistema operativo si se desea que éste sea realmente seguro.
EJ Para diseñar un sistema seguro, en el que se pueda con fiar, hay que definir de forma clara y no ambigua crite rios a seguir respecto a las siguientes cuatro cuestiones:
política de seguridad, modelos, diseño del sistema de seguridad y confiabilidad del sistema.
U La política de seguridad define la seguridad que un sis tema debe proporcionar, no los mecanismos con que se proporciona dicha seguridad.
EJ Algunas políticas de seguridad en uso son la militar del DoD, la de Clark-Wilson y la de la muralla china.
EJ Existen principios de diseño básicos a seguir en siste mas seguros. Entre otros: diseño abierto, privilegios mí nimos, compartición mínima, sospecha mutua, etc.
U Un sistema operativo multiprogramado lleva a cabo las siguientes tareas relacionadas con la seguridad del sis tema: autenticación de usuarios, protección de memo ria, control de accesos a los dispositivos y a los datos, control de la compartición y la comunicación de objetos entre procesos y protección de los datos del sistema de protección en sí mismo.
EJ Las principales técnicas de diseño de sistemas seguros son: uso de núcleos de seguridad, separación físico-ló gica de recursos, uso de entornos virtuales y diseño por capas.
U La criptografía es la técnica que permite codificar un objeto de forma que su significado no sea obvio. Para ello usa algoritmos de cifrado y claves públicas o priva-
das.
EJ Es importante conocer la clasificación de seguridad de un sistema operativo y la norma con que ha sido clasifi cado. La norma del DoD es muy popular.
U Todos los sistemas operativos deben tener mecanismos de protección que permitan implementar distintas polí ticas de seguridad para los accesos al sistema. Dichos mecanismos permiten controlar el acceso a los objetos del sistema permitiéndolo o denegándolo sobre la base de información tal como la identificación del usuario, el tipo de recurso, la pertenencia del usuario a un cierto grupo de personas, las operaciones que puede hacer el usuario o el grupo con cada recurso, etc.
U El paso previo a la aplicación de cualquier esquema de protección o confidencialidad de datos es conocer la identidad del usuario que está accediendo a dichos da tos. El objetivo de la identificación del usuario, también denominada autenticación, es determinar si un usuario (persona, servicio o computadora) es quien dice ser.
EJ Existen diversas formas de establecer la identidad de un usuario: pedir información que sólo conoce él, identifi car una característica física o pedir un objeto que sólo tiene el usuario. En sistemas operativos es muy popular el uso de contraseñas o palabras clave.
EJ Es necesario tomar cuatro decisiones básicas de diseño para un sistema de contraseñas: ¿Quién asigna las pala bras clave? Longitud y formato de las palabras clave. ¿Dónde se almacenan las claves? Duración de las cia-
ves.
U La relación entre objetos y derechos de acceso se define usando dominios de protección. Un dominio de protec ción es un conjunto de pares (objeto, derechos), donde se especifican un objeto y las operaciones que se pue den ejecutar sobre el mismo.
EJ En UNIX sólo existen tres dominios de protección: dueño, grupo y otros. En cada dominio sólo se pueden especificar tres operaciones: read, write, execute.
U La relación entre dominios y objetos se puede definir de forma completa mediante una matriz de protección. Los

Digitalización realizada con propósito académico

dominios de protección son las filas de la matriz y los objetos son las columnas de la misma. El elemento (i, j) expresa las operaciones que el dominio i puede eje cutar sobre el objeto j. Estas matrices de protección suelen ser muy grandes o muy dispersas, por lo que se implementan como listas de control de acceso (ACL) o capacidades (capabilities).

U Una ACL es una lista de entradas asociadas a un objeto que especifican qué operaciones puede hacer cada do-
minio sobre ese objeto. Las ACL son sencillas de im plementar en sistemas no distribuidos, pero tienden a crecer mucho y pueden ser poco eficientes. Revocar to dos los permisos de acceso es fácil: basta con desconec tar la ACL.

U Una capacidad indica qué operaciones puede hacer un usuario sobre un objeto cuando pertenece a un determi nado dominio. Cada usuario tendrá una lista de capaci dades. Las capacidades complejas, pero se prestan muy bien a sistemas distribuidos. Revocar todos los permi sos de acceso puede ser complejo.

U Los servicios de protección y seguridad de un sistema varían dependiendo de que se usen ACL o capacidades. Además, varían dependiendo de la complejidad del sis tema implementado.

U Los servicios de seguridad de POSIX usan el modelo de ACL de UNIX directamente. Permiten manipular los permisos de un objeto y la identidad y el grupo del due ño del objeto.

U Los servicios de seguridad de Windows NT usan un modelo de ACL más general y, por tanto, más comple jo. Incluyen varias llamadas para manipular la ACL y sus entradas (ACE).

U Kerberos es un sistema de seguridad para sistemas co-
nectados en red, aunque se puede aplicar a sistemas operativos convencionales. Fue desarrollado en el Mas sachusetts Institute of Technology (MIT) a finales de los años ochenta y se ha convertido en un estándar de facto en sistemas operativos. Se basa en el uso de claves privadas, protocolos simétricos de intercambio de cla ves y en la existencia de servidores de claves y tickets.

9.12. LECTURAS RECOMENDADAS

Para extender la visión introductoria sobre el tema se recomienda acudir a otros libros de sistemas operativos tales como el de Deitel [1984], Krakowiak [1988], Tanenbaum [1997] y [1992], Silberschatz [1998] o Stallings [1995]. Para tener una visión más profunda del tema de seguridad y protección se recomienda acudir a textos más específicos, tales como [PUTER, 1893b], [1982], [1990], [19891 y [1997].

Los estudios básicos sobre temas de seguridad y protección en sistemas operativos se llevaron a cabo en la década de los setenta. Varios de estos estudios son importantes para entender el problema y las soluciones propuestas. Popek [19741 estudió distintas estructuras de protección en sistemas operativos. En [1976] se mostró el estado del arte del momento sobre estructuras que permitían dar soporte de seguridad. En [19761 se repasó la perspectiva de la seguridad de los sistemas operativos existentes. En [1979] se hizo una exploración muy completa de las posibles vulneraciones de seguridad y de la forma de controlarlas.

Las cuestiones más relevantes de diseño de sistemas seguros se estudian en [19731, [TER, 1983a], [1988] y [1990]. Para aprender más sobre la aplicación de la criptografía al diseño de sistemas seguros se puede consultar [1982] y [19821. Buenos estudios de penetra ción de sistemas son los de [1976] para el VM/370 y [1974] para MVS.

Los problemas de integridad y disponibilidad de datos se estudian en [1991] y [19921.

Para estudios sobre seguridad en sistemas operativos específicos se pueden leer los textos de [19851, [1986], [1990 [1993], [19911 y [19921, que estudian el sistema operativo UNIX, los de [1999] y [1999] para LINUX y los de [1994] y [19981 para Windows NT.

9.13. EJERCICIOS

¿Qué tipos de fallos de seguridad puede tener un sistema de computación?

¿Cuál es la diferencia entre los términos seguridad y protección?

Suponga que un empleado de una compañía genera un código malicioso que está transmitiendo infor mación no autorizada, como por ejemplo el sueldo de los empleados, a intrusos o espías. ¿Cómo se

Digitalización realizada con propósito académico

podría controlar este problema? ¿Qué solución se le ocurre para limitar sus efectos?

9.4. Suponga que un usuario malicioso quiere imple- mentar un canal encubierto usando archivos. Ex plique cómo podría conseguirlo usando cerrojos sobre un archivo ya existente. ¿Se podría imple- mentar el canal encubierto usando utilidades de creación y borrado de un archivo? Explique cómo.

9.5. ¿Cuál es la principal diferencia entre un gusano y un virus?

9.6. ¿De qué formas se puede introducir un virus en un programa? Explique cómo funciona cada mé todo.

9.7. Suponga que un sistema operativo usa palabras clave como método de autenticación y que las guarda internamente en un archivo cifrado. ¿Debe rían ser dichas palabras clave visibles a todos los usuarios? ¿Cómo se puede resolver este problema habida cuenta de que el proceso de login debe
ver las claves?

9.8. ¿Cuáles son las ventajas de almacenar los datos criptografiados? ¿Y sus desventajas?

9.9. ¿Cómo puede detectar el administrador de un siste ma que hay un intruso en el mismo?

9.10. ¿Cuál es la principal diferencia entre los cifrados por sustitución y por permutación? ¿Incluye toda sustitución una permutación?

9.11. ¿Qué características debería tener un algorit mo de cifrado para ser completamente perfecto? ¿Cuáles son los límites prácticos de dichos algorit mos?

9.12. Suponga un algoritmo de cifrado por sustitución que usa un alfabeto de 24 caracteres, 10 números y
10 caracteres especiales y cuya longitud de palabra clave es 8 bytes. ¿Cuál es el número de combina ciones posibles existentes como resultado del cif rado?

9.13. Enumere los principios de diseño más importantes de un sistema seguro.

9.14. En un sistema que obliga a cambiar la clave de ac ceso periódicamente se permite a los usuarios in troducir una clave nueva cada vez sin ninguna res tricción. ¿Cuál es el principal fallo de seguridad relacionado con el usuario que puede ocurrir en este sistema? ¿Cómo se puede tratar de paliar ese problema?

¿Se puede implementar la política de seguridad militar en un sistema de computación? ¿Existe al gún sistema que la use?

¿Qué ventajas y desventajas tiene el diseño de un sistema de seguridad con anillos concéntricos? In dique un sistema operativo que use esta técnica de diseño.

9.17. ¿Usa UNIX una política de control de acceso obli gatoria o discrecional? Y Windows NT? Explique su respuesta.

9.18. ¿La protección de memoria en sistemas operativos se lleva a cabo mediante mecanismos hardware, software o mediante una combinación de ambos? Explique su respuesta.

9.19. Indique ejemplos de separación física y lógica usa dos en sistemas operativos para proporcionar me canismos de seguridad.

9.20. ¿Es posible controlar el acceso a un objeto sin nombre? En caso de que el objeto deba tener nom bre, ¿quién debería asignar el nombre? ¿Debería dicho nombre ser a su vez seguro y gozar de inte gridad?

9.21. Suponga que desea diseñar una política de control de acceso en la que cada usuario sólo puede acce der a un objeto un número limitado de veces. Des pués de agotada su cuota de accesos, el usuario debe renovarla. Proponga mecanismos para imple-
mentar dicha política.

9.22. Una buena técnica de seguridad sería guardar to dos los archivos cifrados. Sin embargo, usar
siempre esta política podría causar graves tras tornos en el rendimiento del sistema operativo y complicar la estructura de los archivos. ¿Qué solu ción se le ocurre para los problemas anteriores? Diséñela.

9.23. ¿Es posible proteger los objetos de un sistema ope rativo en el que cualquier usuario puede acceder
directamente a los dispositivos de entrada-salida? ¿Quién debería controlar el acceso a dichos dispo sitivos?

9.24. Indique algunos de los fallos más frecuentes en contrados en los procesos de autenticación
de los sistemas operativos basados en palabras clave.

9.25. ¿Es suficiente con controlar los accesos a un archi vo de UNIX cuando se abre y se cierra dicho archi vo? ¿Qué problemas pueden surgir con este tipo de
solución?

9.26. ¿Le parece una buena política de seguridad que cualquier usuario pueda acceder a casi todos los
archivos del sistema en UNIX? Si esto no fuera así, ¿cómo podría un usuario acceder a los mandatos del sistema?

¿Cuáles son las principales diferencias entre las ca pacidades, las listas de control de acceso y los bits de protección de UNIX?

Imagine que un sistema operativo que usa ACLs tiene un archivo en el que ha concedido permisos a 1.000 usuarios. ¿Qué debería hacer para revocar los derechos de acceso a todos ellos? ¿Ocurriría lo

Digitalización realizada con propósito académico

mismo si el sistema operativo usara capacidades?, ¿Puede el alumno juan escribir el archivo? ¿Pue
¿por qué? de leerlo? ¿Qué debería hacer pepe para que
j uan pudiera ejecutarlo?

9.29. En un sistema UNIX hay un archivo denominado

datos cuyas características obtenidas con el man-
dato is —is datos son las siguientes:

```
-rw-r--r--  1 pepe alumnos 7642 archivo datos
```

9.30. Implemente un programa que cambie los permisos de acceso a un archivo UNIX, obligando a establecer una máscara de protección para que nunca se pueda ejecutar.

9.31. ¿Qué ocurriría en UNIX si un usuario tuviera per
misos de acceso r—— a un archivo y su grupo
tuviera rw—? ¿Podría el usuario escribir el archi
vo?

9.32. ¿Usa Kerberos un protocolo de intercambio de cia ves simétrico o asimétrico?

11

Estudio de casos: LINUX

En este capítulo se estudia el sistema operativo Linux. Se trata de un sistema operativo de libre distribución que proporciona una interfaz POSIX. Actualmente Linux es ampliamente utilizado y se ha convertido en uno de los pocos sistemas que puede competir hasta cierto punto con los sistemas Windows por lograr una cuota del mercado de los sistemas operativos. De hecho, no sólo se trata de dos sistemas operativos diferentes sino de dos maneras distintas de entender la informática.

Windows es un sistema cerrado y «secreto» creado por una gran multinacional, Microsoft cuyo único objetivo es vender su producto sea como sea. Linux es un sistema abierto y público que inicialmente surgió de un proyecto personal sin ánimo lucro, pero que se ha convertido en la actualidad en un sistema desarrollado por gente de todo el mundo gracias al auge de Internet.

En este capítulo se da una visión general de Linux. La exposición se organiza en las siguientes secciones:

** Historia de Linux.*
** Características y estructura de Linux.*
** Gestión de procesos.*
** Gestión de memoria.*
** Entrada/salida.*
** Sistema de archivos.*

11.1. HISTORIA DE LINUX

El origen de Linux se encuentra en el sistema operativo MINIX. MINIX fue desarrollado por Andrew S. Tanenbaum con el objetivo de que sirviera de apoyo para la enseñanza de sistemas operativos. De hecho, en su clásico libro Operating Systems: Design and Implementation [Tanenbaum, 1987], se utilizaba este sistema operativo para explicar los diferentes conceptos de esta materia, incluyéndose, además, en un apéndice un listado completo de su código escrito en lenguaje C.

Además de por su carácter pedagógico, MINIX se caracterizaba por tener una estructura basada en un microkernel. El autor intentaba demostrar al crear MINIX que se podía construir un sistema operativo más sencillo y fiable, pero a la vez eficiente, usando este tipo de organización, que era novedosa en aquel momento. Algunas otras características positivas de MINIX eran las siguientes:

- Ofrecía una interfaz basada en la de UNIX Versión 7. El grupo de trabajo de POSIX todavía no había terminado su labor en esa época.
- Tenía un tamaño relativamente pequeño. Constaba de aproximadamente 12.000 líneas de código.
- Podía trabajar en equipos que disponían de unos recursos hardware muy limitados. De hecho, incluso podía usarse en máquinas que no disponían de disco duro.

Sin embargo, también presentaba algunas deficiencias y limitaciones.

- La gestión de memoria era muy primitiva. No había ni memoria virtual ni intercambio. Además, no se aprovechaba adecuadamente el mecanismo de paginación del procesador. El autor justificaba esta limitación argumentando que la inclusión de estos mecanismos complicaría considerablemente el código del sistema operativo. Aunque esta opinión es razonable desde el punto de vista pedagógico, tenía como consecuencia que MINIX no se utilizara como un sistema para el desarrollo de aplicaciones de cierta entidad.
- Por simplicidad, algunas partes del sistema operativo, como por ejemplo el sistema de archivos, no eran concurrentes, lo que limitaba considerablemente el rendimiento del sistema.

A pesar de estos defectos, MINIX atrajo la atención de muchos usuarios de todo el mundo, que usaban el grupo de noticias comp.os.minix como punto de encuentro. Algunos de estos usuarios se ofrecían a mejorar partes del sistema o a incluir nuevas funciones al mismo. Sin embargo, el autor siempre fue reacio a estas ofertas. Entre los interesados en MINIX se encontraba un estudiante llamado Linus Torvalds, Fue en el año 1990 cuando este estudiante envió un mensaje a este grupo de noticias comentando que, por curiosidad y ganas de ampliar sus conocimientos, estaba desarrollando un nuevo sistema operativo tomando como base MINIX. Se estaba produciendo el humilde nacimiento de Linux. Cuya primera versión (enumerada como 0.01) vio la luz a mediados del año 1991.

Es importante recalcar que, en su concepción inicial, Linux tomaba prestadas numerosas características de MINIX (p. ej.: el sistema de archivos). De hecho, Linux se desarrolló usando corno plataforma de trabajo MINIX y las primeras versiones de Linux no eran autónomas sino que tenían que arrancarse desde MINIX.

Sin embargo, a pesar de esta herencia inicial, MINIX y Linux son radicalmente diferentes. Por un lado. Linux solventa muchas de las deficiencias de MINIX como, por ejemplo, la carencia de memoria virtual. Estas mejoras permiten que se trate de un sistema adecuado para trabajar de manera profesional, no limitándose su uso a un entorno académico. Pero la diferencia más impor-

tante entre ellos está en su organización Interna. Mientras que MINIX tiene una estructura moderna basada en un microkernel, Linux posee una estructura monolítica más clásica. Este diseño conservador tuvo como consecuencia que el autor de MINIX no diera el visto bueno a Linux ya que consideraba que este sistema suponía un paso atrás en la evolución de los sistemas operativos.

Desde su lanzamiento público en 1991, Linux ha evolucionado e incorporado nuevas características. Además, se ha transportado a otros procesadores como SPARC, Alpha y MIPS. En la actualidad es un sistema con unas características y prestaciones comparables a las de cualquier sistema operativo comercial.

Puesto que en esta sección se ha presentado la historia de Linux, parece el lugar adecuado para explicar cual es el convenio usado para enunciar las sucesivas versiones del sistema. El número de la versión contiene un número primario y uno secundario separados por un punto p. ej.: versión 2.2). Un incremento en el número primario corresponde con una nueva versión que incluye cambios significativos en el sistema. En el caso de una modificación de menor impacto, solo se modifica el número de versión secundario. Además un número secundario de versión que sea impar (como, por ejemplo la versión 2.1) indica que se trata de una versión inestable en la que se han incluido nuevas características que todavía hay que probar y depurar. Evidentemente un usuario normal debería instalarse una versión con un número secundario par.

11.2 CARACTERISTICAS Y ESTRUCTURA DE LINUX

Linux es un sistema de tipo UNIX y, por tanto, posee las características típicas de los sistemas UNIX. Se trata de un sistema multiusuario y multitarea de propósito general. Algunas de sus características específicas más relevantes son las siguientes:

- Proporciona una interfaz POSIX.
- Tiene un código independiente del procesador en la medida de lo posible. Aunque inicialmente se desarrolló para procesadores Intel, se ha transportado a otras arquitecturas con un esfuerzo relativamente pequeño.
- Puede adaptarse a máquinas de muy diversas características. Como el desarrollo inicial se realizó en máquinas con recursos limitados, ha resultado un sistema que puede trabajar en máquinas con prestaciones muy diferentes.
- Permite incluir de forma dinámica nuevas funcionalidades al núcleo del sistema operativo gracias al mecanismo de los módulos.
- Proporciona soporte para una gran variedad de tipos de sistemas de archivos, entre ellos los utilizados en Windows. También es capaz de manejar distintos formatos de archivos ejecutables.
- Proporciona soporte para multiprocesadores utilizando un esquema de multiproceso simétrico. Para aprovechar al máximo el paralelismo del hardware, se ha ido modificando progresivamente el núcleo con el objetivo de aumentar su concurrencia interna.

En cuanto a la estructura de Linux, como se comentó previamente, tiene una organización monolítica al igual que ocurre con la mayoría de las implementaciones de UNIX. A pesar de este carácter monolítico, el núcleo no es algo estático y cerrado sino que se pueden añadir y quitar módulos de código en tiempo de ejecución. Se trata de un mecanismo similar al de las bibliotecas dinámicas pero aplicado al propio sistema operativo. Se pueden añadir módulos que correspondan

con nuevos tipos de sistemas de archivos, nuevos manejadores de dispositivos o gestores de nuevos formatos de ejecutables.

Un sistema Linux completo no sólo está formado por el núcleo monolítico sino también incluye programas del sistema (como, por ejemplo, demonios) y bibliotecas del sistema.

Debido a las dificultades que hay para instalar y configurar el sistema, existen diversas distribuciones de Linux que incluyen el núcleo, los programas y bibliotecas del sistema, así como un conjunto de herramientas de instalación y configuración que facilitan considerablemente esta ardua labor. Hay distribuciones tanto de carácter comercial como gratuitas. Algunas de las distribuciones más populares son *Slackware, Debian, Suse y Red Hat.*

11.3. GESTIÓN DE PROCESOS

La gestión de procesos en Linux es básicamente igual que en cualquier otra variedad de UNIX. Un aspecto original de Linux es el servicio *clone* que es una extensión del clásico fork.

Este nuevo servicio permite crear un proceso que comparta con el padre su mapa de memoria, sus rutinas de manejo de señales y sus descriptores de archivos. Aunque Linux no implementa *threads* en el núcleo, se pueden construir bibliotecas de *threads* usando este nuevo servicio.

En cuanto a la sincronización dentro del núcleo, siguiendo la tradición de UNIX, Linux no permite que haya llamadas concurrentes activas dentro del sistema operativo. Así, si se produce un evento que causa un cambio de proceso mientras se está ejecutando una llamada al sistema (p. ej.: una interrupción de reloj), el cambio se difiere hasta que la llamada termina o se bloquea.

Para evitar las condiciones de carrera entre la ejecución de una llamada y el tratamiento de una interrupción, se prohíben las interrupciones en pequeñas zonas del código del sistema.

Puesto que el código de una rutina de interrupción ejecuta con una prioridad alta bloqueando el tratamiento de las interrupciones, es importante que estas rutinas sean muy cortas. Para ayudar a lograr este objetivo, Linux ofrece un mecanismo que permite dividir las operaciones asociadas a una interrupción en dos partes:

- Las operaciones de carácter más urgente se ejecutan en el entorno de la rutina de interrupción, denominada, *mitad superior* en la nomenclatura de Linux.
- Las operaciones menos urgentes las ejecuta una rutina del núcleo que tiene una prioridad inferior a la de los dispositivos. Esta rutina se denomina en Linux, *mitad inferior* y durante su ejecución no estarán bloqueadas las interrupciones de otros dispositivos. La propia mitad superior se encarga de fijar un determinado valor en una estructura de datos para indicar que la mitad inferior está activada. Justo antes de retornar a modo usuario, el sistema operativo comprueba dicha estructura, y si encuentra que hay alguna mitad inferior activada, la ejecutará.

La Figura 11.1 muestra cuáles son los niveles de prioridad de las diversas partes del núcleo. Observe que sólo se ejecutará una rutina de un determinado nivel si no está activa ninguna rutina de un nivel superior.

En cuanto a la planificación, Linux soporta tres clases de planificación: un algoritmo de tiempo compartido y dos algoritmos de planificación de tiempo real que se corresponden con los definidos por POSIX.

El servicio sched_setscheduler permite definir la clase de planificación del proceso que la invoca. Esta llamada sólo puede hacerla un proceso privilegiado. Cada proceso de tiempo real tiene

Mitades superiores (prio. máx)
Llamadas al sistema
Mitades inferiores
Procesos de usuario (prio. mín.)

Figura 11.1. *Niveles de prioridad dentro del núcleo de Linux.*

asociada una prioridad y un tipo de planificación que puede ser FIFO o Round-Robin. El planificador selecciona en cada momento el proceso listo para ejecutar que tenga mayor prioridad. Si el proceso es de tipo FIFO, seguirá ejecutando hasta que se bloquee. Si es de tipo Round-Robin, cuando termine su rodaja, el proceso pasará al final de la cola de procesos listos para ejecutar de su misma prioridad.

Los procesos de tiempo compartido sólo pueden ejecutar cuando no hay ningún proceso de tiempo real listo para ejecutar. El algoritmo de planificación para este tipo de procesos intenta conjugar la prioridad del proceso con su perfil de ejecución, favoreciendo a los procesos que realizan más operaciones de entrada/salida. A continuación, se describe este algoritmo.

Todo proceso tiene asociada una *prioridad base*. Inicialmente la prioridad del proceso es igual a su prioridad base. Cada vez que se produce una interrupción de reloj se resta una unidad a la prioridad del proceso que estaba ejecutando. El algoritmo de planificación está basado en la prioridad del proceso y tiene carácter exclusivo: el planificador elige el proceso listo para ejecutar que tenga mayor prioridad.

Cuando se produce una situación en la que todos los procesos listos para ejecutar tienen una prioridad igual a cero (todos ellos han usado el procesador un tiempo suficiente para que su prioridad haya caído a 0), se produce un reajuste de las prioridades de todos los procesos sea cual sea su estado. La nueva prioridad se calcula dividiendo por 2 la actual y sumando la prioridad base (prioridad = prioridad/2 + prioridad base). Observe que los procesos listos para ejecutar simple mente recuperan su prioridad base ya que su prioridad actual es igual a 0. Sin embargo, los procesos bloqueados obtienen una nueva prioridad mayor que la base puesto que su actual prioridad es mayor que cero. Se trata de una fórmula de tipo exponencial. Con ella, un proceso que estuviese bloqueado mucho tiempo puede llegar a tener una prioridad con un valor del doble de la prioridad base.

11.4. GESTIÓN DE MEMORIA

Linux tiene un sistema de memoria que incluye todas las características habituales en los sistemas modernos. Estas características ya fueron discutidas en el capítulo dedicado a este tema, por lo que en esta sección se presentan aquellos aspectos específicos de Linux:

- Se utiliza un modelo de memoria independiente del procesador. Utiliza un esquema de paginación con tres niveles. Existe una capa de software de bajo nivel que se encarga de adaptar este modelo abstracto al hardware de gestión de memoria real.
- Permite utilizar tanto dispositivos como archivos para soporte de la memoria secundaria.
- Se utiliza una versión modificada del algoritmo del reloj como algoritmo de reemplazo.
- Gestiona la memoria dinámica del propio sistema operativo usando un algoritmo inspirado en el sistema *buddy.*

11.5. ENTRADA/SALIDA

La entrada/salida en Linux es muy similar a la de cualquier otro sistema UNIX. Se distinguen, por tanto, dos tipos de dispositivos: dispositivos de bloques y dispositivos de caracteres.

Como el resto de los sistemas UNIX, se utiliza una cache común para todos los dispositivos de bloques. El tamaño de la cache es dinámico y crece de acuerdo a las necesidades de memoria del resto del sistema. Para gestionarla se usa básicamente una política de reemplazo LRU. En las últimas versiones esta cache trabaja coordinadamente con la utilizada por el gestor de memoria.

En cuanto al acceso a los discos, se utiliza el algoritmo del ascensor con un único sentido de servicio.

Siguiendo el modelo de UNIX, en Linux los usuarios ven los dispositivos como archivos y utilizan los servicios destinados a trabajar con archivos para acceder a los dispositivos.

La red, sin embargo, es un dispositivo que tiene un tratamiento un poco diferente. El usuario no puede acceder a este dispositivo de la misma manera que a un archivo. La parte del sistema operativo que trata la red está dividida en tres niveles:

- En el nivel inferior está el manejador del dispositivo al que el usuario no puede acceder directamente.
- En el nivel intermedio está el software que implementa la pila de protocolos (p. ej.: TCP e IP).
- En el nivel superior está la interfaz del programador que corresponde con la de los *sockets* definidos en el UNIX BSD.

11.6. SISTEMA DE ARCHIVOS

Linux da soporte a una gran variedad de tipos de sistemas de archivos entre los que se incluyen los distintos sistemas de archivos de Windows y de otros sistemas UNIX. Además, cualquier usuario puede programar un manejador de un nuevo tipo de sistema de archivos e incluirlo en el sistema como un módulo.

Esta coexistencia de distintos tipos de sistemas de archivos la posibilita el VFS (Virtual File System, Sistema Virtual de Archivos), presente en la mayoría de los sistemas UNIX actuales y suficientemente analizado en el Capítulo 8.

Aunque admite muy diferentes tipos de sistemas de archivos, Linux tiene su propio sistema de archivos que se denomina ext2fs. Este sistema evolucionó desde el sistema de archivos de MINIX .Se le fueron añadiendo nuevas características al sistema de archivos de MINIX hasta llegar al sistema extfs. Posteriormente se rediseñó dando lugar al ext2fs actual. Se trata de un sistema basado en el FFS (Fast File System, Sistema de Archivos Rápido) del UNIX BSD, que ya se estudió adecuadamente en el Capítulo 8.

Merece mención aparte un tipo de sistema de archivos muy especial: el sistema de archivos proc. Este sistema de archivos no tiene soporte en ningún dispositivo. Su objetivo es poner a disposición del usuario datos del estado del sistema en la forma de archivos. Esta idea no es original de Linux ya que casi todos los sistemas UNIX la incluyen. Sin embargo, Linux se caracteriza por ofrecer más información del sistema que el resto de variedades de UNIX. En este sistema de archivos se puede acceder a información general sobre características y estadísticas del sistema, así como a información sobre los distintos procesos existentes.

La Figura 11.2 muestra cómo se relacionan las distintas partes del sistema de archivos.

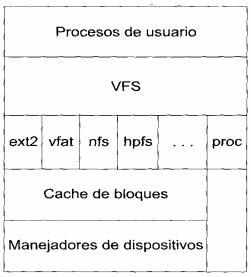


Figura11.2. *Niveles del sistema de archivos*

11.7. PUNTOS A RECORDAR

- El origen del Linux está en MINIX que es un sistema operativo de carácter pedagógico basado en un microkernel
- Linux supera muchas de las limitaciones de MINIX y tiene una organización monolítica más convencional.
- Linux comenzó en 1991, como proyecto personal de Linus Torvalds. Gracias al auge de Internet, han colaborado numerosas personas en su desarrollo.
- Proporciona una interfaz POSIX.
- Está diseñado para facilitar su transporte a distintos procesadores.
- Puede ejecutar en máquinas de muy distintas prestaciones.
- Da soporte a una gran variedad de tipos de sistemas de archivos.
- Proporciona soporte para un esquema de multiproceso simétrico.
- La gestión de procesos en Linux es muy similar a la realizada en otros sistemas UNIX.
- El núcleo de Linux no es reentrante.
- El tratamiento de una interrupción puede dividirse en una mitad superior, que ejecuta con alta prioridad, y una mitad inferior menos prioritaria.
- Linux soporta las clases de planificación de tiempo real definidas en POSIX.
- La planificación de procesos de tiempo compartido intenta conjugar la prioridad del proceso y su perfil de ejecución.
- La gestión de memoria en Linux incluye todas las características presentes en cualquier sistema operativo moderno.
- La entrada/salida es similar a la existente en cualquier otro sistema UNIX.
- Linux da soporte a una gran variedad ele tipos de sistemas de archivos gracias al VFS.
- El sistema de archivos nativo de Linux es ext2fs basado en el FFS.
- El sistema de archivos proc ofrece mucha información sobre el propio sistema y los procesos existentes en el mismo.

11.8. LECTURAS RECOMENDADAS

Algunos libros generales ele sistemas operativos incluyen un capítulo dedicado a Linux Como [Silberschatz, 1998]. Hay también libros dedicados exclusivamente a Linux como [Cornes. 1997], que presenta con bastante detalle todas las características del sistema, o [Beck, 1998] que se centra en los aspectos internos. Para comprender mejor Linux es interesante estudiar MINIX, el sistema operativo que le sirvió como punto de partida. En [Tanenbaum, 1997] se presenta la versión actual de MINIX incluyendo el listado completo de este sistema operativo.

Digitalización Realizada con propósito académico. Sistemas Operativos

12

Estudio de casos: Windows NT

En este capítulo se presenta con más detalle el sistema operativo Windows NT, haciendo énfasis en sus conceptos principales y principios de diseño. El capítulo tiene como objetivo básico mostrar al lector los aspectos de diseño del sistema operativo y sus componentes, así como las características que ofrece Windows NT a los usuarios. Para alcanzar este objetivo el capítulo se estructura en las siguientes grandes secciones:

** Principios de diseño de Windows NT.*
** Arquitectura de Windows NT.*
** El núcleo de Windows NT.*
** El ejecutivo de Windows NT.*
** Subsistemas de entorno de ejecución.*
** Sistemas de archivos de Windows NT*
** El subsistema de seguridad.*
**Mecanismos para la tolerancia a fallos en Windows NT.*

Para terminar el capítulo, se incluye un conjunto seleccionado de lecturas recomendadas que permitirán al lector interesado profundizar en este tema.

12.1. INTRODUCCIÓN

Windows NT es un sistema operativo multitarea, basado en un diseño de 32 bits, cuyas características principales son su diseño orientado a objetos, el subsistema de seguridad y los servicios de entrada/salida. Al igual que otros sistemas operativos modernos, proporciona espacios de memoria separados para cada proceso, planificación con expulsión a nivel de thread y multiprocesamiento simétrico en máquinas con dos procesadores. Se puede ejecutar tanto en procesadores con conjuntos de instrucciones complejos (CISC, *complex instruction set computer*) como en aquellos con conjuntos de instrucciones reducidas (RISC, reduced instruction set computer).

La historia del sistema operativo empezó en 1989, cuando Microsoft decidió diseñar un nuevo sistema que sustituyera a Windows 3.x y que incluyera las características adecuadas para ser usado en máquinas multiproceso de grandes dimensiones, para lo que versiones anteriores de Windows no eran válidas. Para ello se contrató a expertos en diseño de sistemas operativos, como Dave Cutler de DEC, que aportaron al diseño de Windows NT muchas de las ideas existentes en otros sistemas operativos. El diseño ha ido cambiando ligeramente, aunque la parte del núcleo se mantiene bastante estable desde la versión 3.5. Actualmente se distribuye la versión 4 de Windows NT junto con Windows 2000 y Windows 2000 Server. Esta versión se ha diseñado para su uso en grandes servidores, por lo que incluye muchas mejoras frente a las versiones anteriores (directorios activos, seguridad distribuida, Kerberos, cifrado, etc.).

12.2. PRINCIPIOS DE DISEÑO DE WINDOWS NT

Windows NT tiene un diseño moderno de tipo micronúcleo, con una pequeña capa de núcleo que da soporte a las restantes funciones del ejecutivo del sistema operativo. Dentro del ejecutivo destaca la integración del modelo de seguridad (nivel C2), de la gestión de red, de la existencia de sistemas de archivos robustos y de la aplicación exhaustiva del modelo orientado a objetos.

El diseño del sistema operativo Windows NT se hizo desde cero, pero, a pesar de ello, no incluye muchas ideas nuevas, sino que surgió como el resultado de fundir ideas ya contrastadas en otros sistemas operativos, como UNIX, VMS o MACH, y de la optimización de las mismas. Además, para mantener la compatibilidad con sistemas operativos anteriores de Microsoft, se siguieron manteniendo algunas ideas existentes en MS-DOS y Windows 3.x. Por ello, los principios de diseño fundamentales se parecen mucho a los de otros sistemas operativos:

- **Compatibilidad**. Tanto con los sistemas anteriores (interfaz gráfico y sistemas de archivos FAT de Windows 3.x) como con otros sistemas operativos (OS/2, POSIX, etc.) y con distintos entornos de red. La compatibilidad se logra mediante el uso de subsistemas que emulan los servicios de los distintos sistemas operativos. Los emuladores son similares a las máquinas virtuales de MVS.
- **Transportabilidad**. Windows NT se diseñó para tener amplia difusión comercial, por lo que se pensó desde el principio en la posibilidad de transportarlo a distintos tipos de computadoras con procesadores CISC (Intel) y RISC (MIPS, Digital Alpha, etc.). Para facilitar el transporte, Windows NT se ha construido sobre una pequeña capa de abstracción de hardware (HAL, Hardware Abstraction Layer) que proporciona toda la funcionalidad dependiente del hardware al núcleo del sistema. Para transportar el sistema operativo, sólo es necesario adaptar el HAL a cada entorno hardware. Esta capa existe en otros sistemas como MACH y MINIX, aunque no suele existir en sistemas monolíticos como UNIX y LINUX. Además de la transportabilidad para el hardware, Windows NT incluye facilidades para transportar las interfases a distintas lenguas mediante el uso del estándar *Unicode* de ISO.

Digitalización Realizada con propósito académico. Sistemas Operativos

- **Escalabilidad**. Windows NT se diseñó de forma modular sobre la base de un micronúcleo. Esta arquitectura permite repartir elementos del sistema sobre distintos procesadores de forma sencilla y extender el sistema con nuevos componentes. Actualmente, el sistema operativo se puede ejecutar en computadoras con 32 procesadores. lo que permite integrar entornos de estaciones de trabajo y servidores.
- **Seguridad**. Uno de los requisitos fundamentales de diseño de Windows NT fue proporcionar un nivel de seguridad C2 de acuerdo a la clasificación del DoD. Para ello se diseñó una arquitectura de seguridad, basada en un monitor de seguridad, que proporciona servicios de seguridad a todos los componentes del sistema operativo y a las aplicaciones externas al mismo.
- **Fiabilidad y robustez**. Los diseñadores de Windows NT han incluido servicios i dar más robustez al sistema tanto en el ámbito de procesos como en el de sistemas de archivos. Ejemplos son los sistemas de archivos con puntos de recuperación, la información redundante con técnicas de paridad, las técnicas de gestión de memoria o la existencia de depuradores internos al núcleo.
- **Procesamiento distribuido**. Al contrario que otros micronúcleos, Windows NT incluye las utilidades de gestión de redes como parte del núcleo del sistema, proporcionando múltiples protocolos de transporte, RPC, sockets, colas de mensajes, etc.
- **Eficiencia**. Los diseñadores de Windows NT se plantearon diseñar un sistema muy eficiente, tanto en monoprocesadores como en multiprocesadores. Para ello construyeron un modelo de proceso basado en procesos ligeros y un sistema de entrada/salida muy compacto en el que todos sus componentes se usan como manejadores de dispositivos.

Los principios anteriores permitieron diseñar un sistema operativo con una arquitectura muy moderna. Su uso se extendió rápidamente, sobrepasando el ámbito de Windows 3.x que estaba reducido a computadoras personales, para pasar a ser instalado en servidores medianos y estaciones de trabajo.

12.3. ARQUITECTURA DE WINDOWS NT

Windows NT tiene una arquitectura por capas muy modular, en la que cada capa está compuesta de varios módulos relativamente simples y con una funcionalidad muy específica.

Como se puede ver en la Figura 12.1, el sistema operativo está compuesto por las siguientes capas:

- **Capa de abstracción del hardware**. Proporciona una interfaz virtual del hardware, escondiendo las particularidades de cada entorno de ejecución del sistema operativo. Incluye la funcionalidad del hardware que necesita el resto del sistema y es el único módulo que es necesario transportar cuando se cambia a otra plataforma hardware. Entre sus funciones están las interfases de los controladores de E/S, interfaz con memoria y UCP, temporizadores del sistema y esconder los detalles del multiprocesamiento simétrico al núcleo del sistema. Cuando se instala el sistema operativo, se elige uno u otro HAL en función de la plataforma hardware.
- **Núcleo**. Proporciona servicios y funcionalidades básicas del sistema operativo tales como gestión de excepciones hardware, control de procesos ligeros, sincronización, etc. Sólo incluye mecanismos, nunca políticas, aunque toma las decisiones para expulsar procesos de memoria. Es pequeño (unos 60 KB), muy eficiente y altamente transportable (el 80 por 100 es independiente de la plataforma hardware). Usa memoria compartida para optimizar las comunicaciones con otros módulos del sistema, por lo que no sigue un diseño de micronúcleo puro.

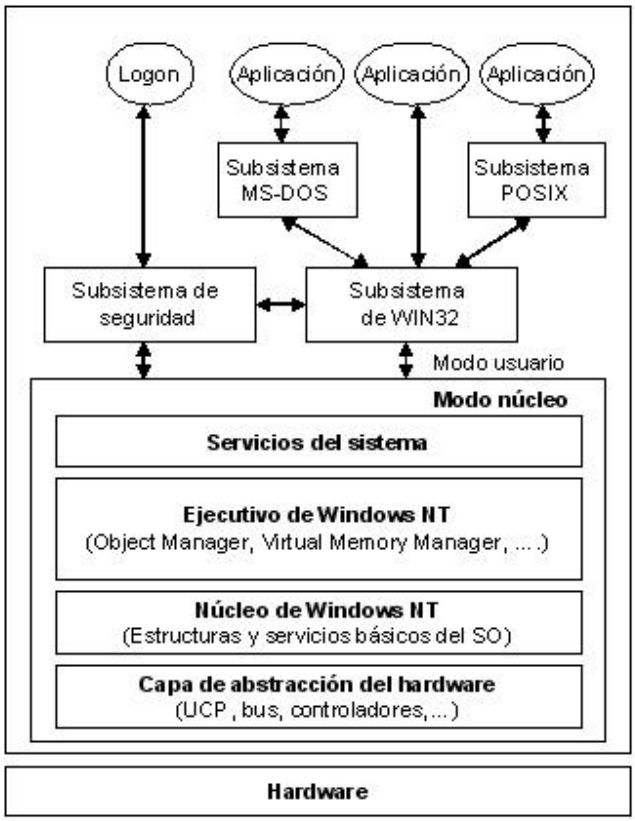


Figura 12.1 *Componentes arquitectónicos de Windows NT.* (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)

- **Ejecutivo**. Incluye los módulos que proporcionan los servicios del sistema operativo para los distintos subsistemas de ejecución. Tiene diseño orientado a objetos y, a su vez, proporciona servicios orientados a objetos. Por ello entre sus componentes incluye un gestor de objetos, además de gestores para los sistemas de seguridad, procesos, memoria virtual, etc. Su interfaz define la **capa de servicios** que el sistema operativo exporta a los subsistemas externos al núcleo. Estos servicios son las interfases entre los subsistemas, que se ejecutan en modo usuario, y el núcleo
- **Subsistemas de entorno de ejecución**. Proporcionan las llamadas al sistema operativo que usan las aplicaciones de usuario, existiendo subsistemas compatibles con distintos sistemas operativos (MS- DOS, OS/2, POSIX I, etc.). A diferencia de las capas anteriores, sus componentes se ejecutan en mudo usuario. Sus servicios se pueden acceder a través de las bibliotecas de los compiladores.

A continuación, se estudian más en detalle los componentes más importantes del sistema operativo.

12.4. EL NÚCLEO DE WINDOWS NT

El núcleo es la base de Windows NT ya que planifica las actividades, denominadas *threads*, de los procesadores de la computadora. Al igual que en UNIX, el núcleo de Windows NT se ejecuta siempre en modo seguro (modo núcleo) y no usa la memoria virtual (no paginable). El software del núcleo no se puede expulsar de la UCP, y por tanto, no hay cambios de contexto durante su ejecución. En caso de que se ejecute en un multiprocesador se puede ejecutar simultáneamente en todos los procesadores.

El núcleo proporciona las siguientes funciones al resto del sistema:

- Modelos de objeto, denominados objetos del núcleo.
- Ejecución ordenada de los *threads* según un modelo de prioridad con 32 niveles.
- Sincronización de la ejecución de distintos procesadores si es necesario.
- Gestión de excepciones hardware.
- Gestión de interrupciones y *traps*.
- Funcionalidad específica para manejar el hardware.
- Mecanismos eficientes de comunicación y sincronización.

El modelo de objetos está en la base de funcionamiento del núcleo, que proporciona dos tipos de objetos básicos:

- **Objetos de planificación**. Permiten controlar la ejecución y sincronización de operaciones del sistema mediante una señal de estado. Los eventos, mutantes, mutex, semáforos, *threads* y temporizadores pertenecen a este tipo de objetos. Los mutantes son el equivalente a los mutex, pero en el nivel de usuario y concepto de propiedad. Los mutex sólo están disponibles en el modo núcleo.
- **Objetos de control**. Permiten controlar las operaciones del núcleo, pero no la planificación. Dentro de este tipo de objeto se agrupan las interrupciones, las llamadas asíncronas a procedimiento, los procesos y los perfiles de ejecución. Todos ellos permiten controlar la ejecución de las operaciones del núcleo llamando a un procedimiento durante la ejecución de un thread, conectando interrupciones a un servicio a través de la tabla de interrupciones dDT, *Interrupt Dispatch Table*), o iniciando un proceso y capturando la información de su tiempo de ejecución en distintos bloques de código.

Un objeto contiene un nombre, un manejador, un descriptor de seguridad, una lista de manejadores de objetos abiertos, una descripción del tipo de objeto que almacena y un cuerpo que incluye información específica del objeto. Además, existe una referencia al tipo de objeto del núcleo al que se apunta, incluyendo atributos tales como si es sincronizable o no, los métodos básicos, etc. Un proceso, por ejemplo, es un objeto que se usa para representar el espacio virtual Y la información necesaria para controlar a un conjunto de objetos tipo thread. El proceso objeto contiene un apuntador a un mapa de direcciones, una lista de threads listos para ejecutar, la lista de threads del proceso, tiempo de ejecución, etc. Al igual que en todo sistema orientado a objetos, es necesario crear e iniciar los objetos más altos en la jerarquía antes de crear sus objetos hijo.

Para almacenar la información acerca de los objetos y sus atributos, el núcleo gestiona las siguientes estructuras de datos:

- **Tabla de interrupciones** (IDT, *Interrupt Dispatch Table*). Asocia las interrupciones con las rutinas que las gestionan.
- **Tabla de descriptores de proceso** (PCB. Process Control Blocks). Incluye apuntadores a los manejadores de objetos tipo proceso. Hay una tabla por cada procesador del sistema.

Asociada a ellas hay una tabla de control de regiones de memoria, cuyas entradas apuntan a las regiones de memoria donde están las otras tablas con información relativa al proceso.

- **Cola de temporizadores**. Lista global de temporizadores activos de todo el sistema. Se mantiene en el núcleo.

Además de estas estructuras se mantienen otras como las colas de dispositivos, las de petición de procesadores y recursos, etc.

12.5. EL EJECUTIVO DE WINDOWS NT

La capa más compleja del sistema operativo es el ejecutivo, un conjunto de servicios comunes que pueden ser usados por todos los subsistemas de entorno de ejecución existentes en Windows NT a través de la capa de servicios del sistema.

Cada grupo de servicios es manejado por uno de los siguientes componentes (Fig. 12.2):

- Gestor de objetos.
- Gestor de procesos.
- Gestor de memoria virtual.
- Monitor de seguridad.
- Utilidad para llamadas a procedimientos locales.
- Gestor de entrada/salida.

En esta sección se estudian todos ellos, excepto el monitor de seguridad, al que se dedica una sección posterior.

12.5.1. Gestor de objetos

El gestor de objetos es el componente del ejecutivo de Windows NT que se encarga de proporcionar servicios para todos los aspectos de los objetos, incluyendo reglas para nombres y seguridad. Todo el funcionamiento del sistema operativo se basa en los **objetos**,

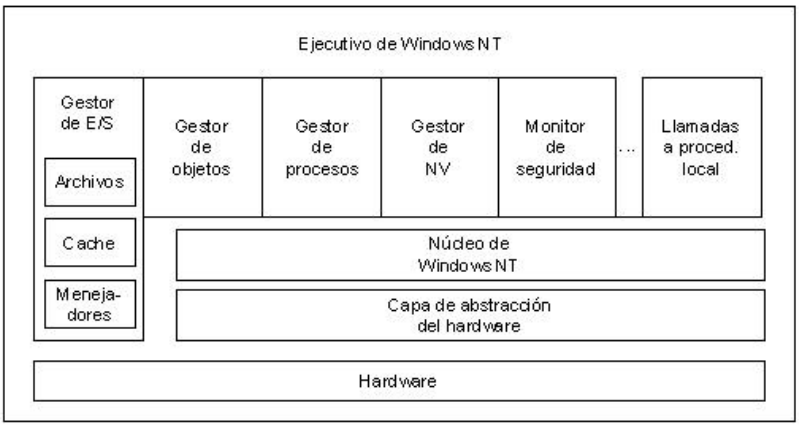


Figura 12.2 *Estructura del ejecutivo de Windows NT.* (Fuente: Microsoft Windows T ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)

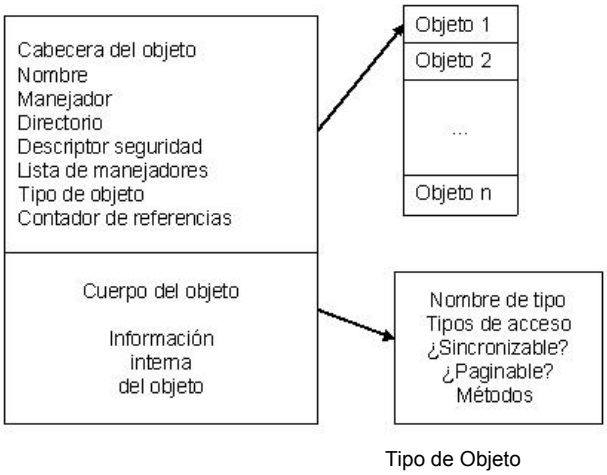


Figura 12.3. *Estructura de un objeto en Windows NT.*

que son instancias en tiempo de ejecución de un tipo objeto particular que pueden ser manipuladas por algún componente del sistema operativo. Un **tipo objeto** está compuesto por un tipo de datos estructurado definido por el sistema y las operaciones que se pueden hacer sobre este tipo de datos y un conjunto de atributos de datos para dichas operaciones.

La Figura 12.3 muestra la estructura de un objeto en Windows NT. Como puede verse, todos los objetos tienen atributos comunes almacenados en la cabecera del objeto, tales como el nombre, el manejador, la lista de manejadores de otros objetos a los que dan paso, el tipo objeto con que están relacionados y un contador de referencias al objeto. Además, incluyen un cuerpo del objeto en el que se almacena información específica de cada objeto. Antes de poder usar un objeto es necesario adquirir un **manejador** del mismo mediante operaciones específicas del sistema operativo que se ejecutan a través del gestor de objetos. El manejador de un objeto incluye información de control y un puntero al objeto en sí mismo.

Para poder identificar un objeto de forma inequívoca es necesario que cada objeto tenga un nombre único en todo el sistema. El gestor de objetos se encarga de gestionar el espacio global de objetos con nombre (directorios, procesos, threads, puertos, archivos, semáforos, etc.). El espacio de nombres se modela como un sistema de archivos jerárquico, como el de la Figura 12.4. Todos los objetos del sistema tienen una parte incluida en el núcleo, definida como un apuntador al objeto correspondiente del núcleo. Esos objetos básicos no pueden ser vistos por el usuario y no tienen nombre lógico.

12.5.2. Gestor de procesos

El gestor de procesos se encarga de gestionar dos tipos de objetos básicos para el sistema operativo:

- **Proceso**. Objeto ejecutable complejo que incluye un espacio de direcciones, un conjunto de objetos para los recursos que ha abierto el proceso, sus manejadores y un conjunto de threads que ejecutan dentro del contexto del proceso.
- **Thread**. Unidad planificable más pequeña del sistema. Siempre están asociados a un proceso, pero su constitución es mucho más ligera: registros, pila y entorilo.

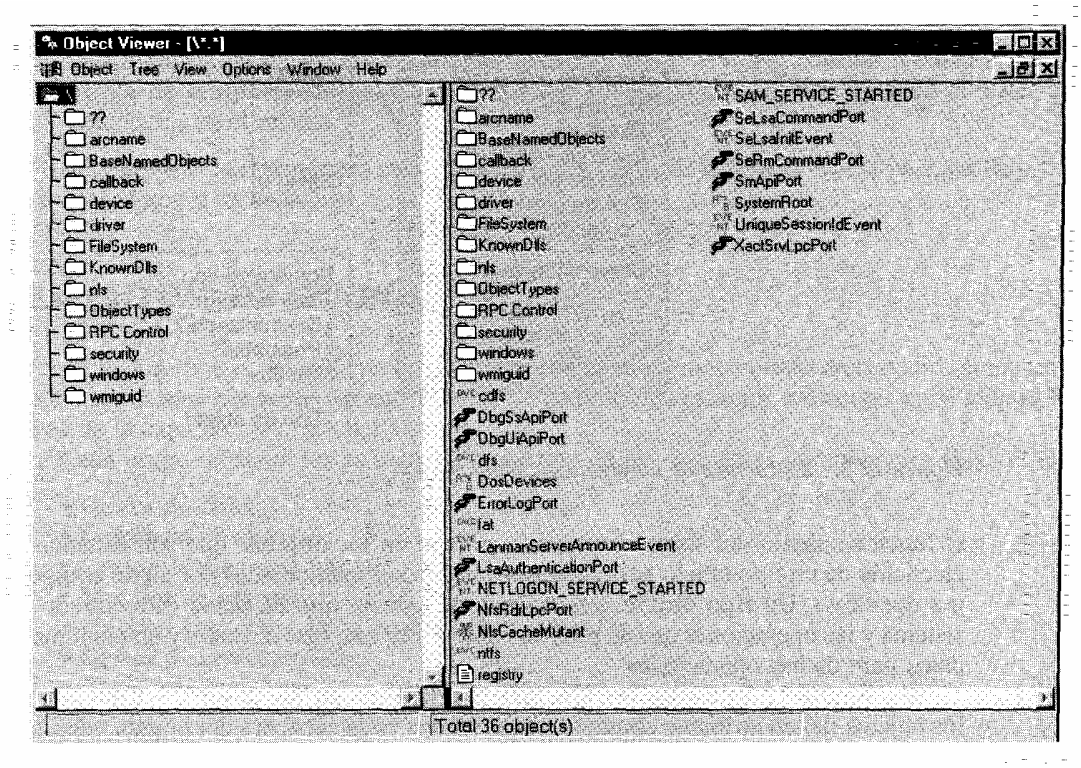


Figura 12.4. *Espacio de nombres de objetos en Windows NT.*

Además, gestiona las operaciones asociadas con ambos objetos (creación, destrucción, etc.) a través del conjunto de servicios para procesos y threads existentes en cada subsistema de entorno de ejecución. Sin embargo, el gestor de procesos no impone ninguna jerarquía a los mismos ni fuerza la existencia de una relación padre-hijo, como ocurre en UNIX o LINUX.

La Figura 12.5 muestra el bloque descriptor de un proceso. Como se puede ver, los datos de descripción del proceso están compuestos por dos estructuras de datos: descriptor en el ejecutivo y descriptor en el núcleo. En la parte del ejecutivo, además de la identificación de descriptor del núcleo, está la identificación del proceso y de su padre, el estado del proceso, los tiempos de creación y terminación, bloques del proceso, descriptor de seguridad, prioridad, etc. En suma, toda la información del proceso que necesitan los otros componentes del ejecutivo y que, en muchos casos, pueden acceder los usuarios a través de los servicios del sistema. En la parte del núcleo se encuentra la información relacionada con la planificación (prioridad, rodaja, etc.) y con la ejecución del proceso (threads, tiempo de ejecución de usuario y sistema, afinidad con el procesador, etcétera).

Los procesos y los threads se planifican de la misma manera: planificación con expulsión y rodaja de tiempo. De hecho, un proceso es visto como un thread especial. Cuando se crea un objeto proceso se le asigna una prioridad determinada de entre la cuatro siguientes: *Iddle, Normal, High, Real-Time.* Existen 32 niveles de prioridad, de los cuales los niveles 1 a 32 se dedican a los procesos de tiempo real. Además, el sistema operativo puede variar el nivel de prioridad mediante calificaciones tales como normal, por debajo de lo normal o por encima de lo normal. Con esos criterios el gestor de objetos consulta una tabla de prioridades y asigna la prioridad inicial al objeto. Además, cada objeto puede tener su propia rodaja de ejecución, asignada cuando se crea el obje-

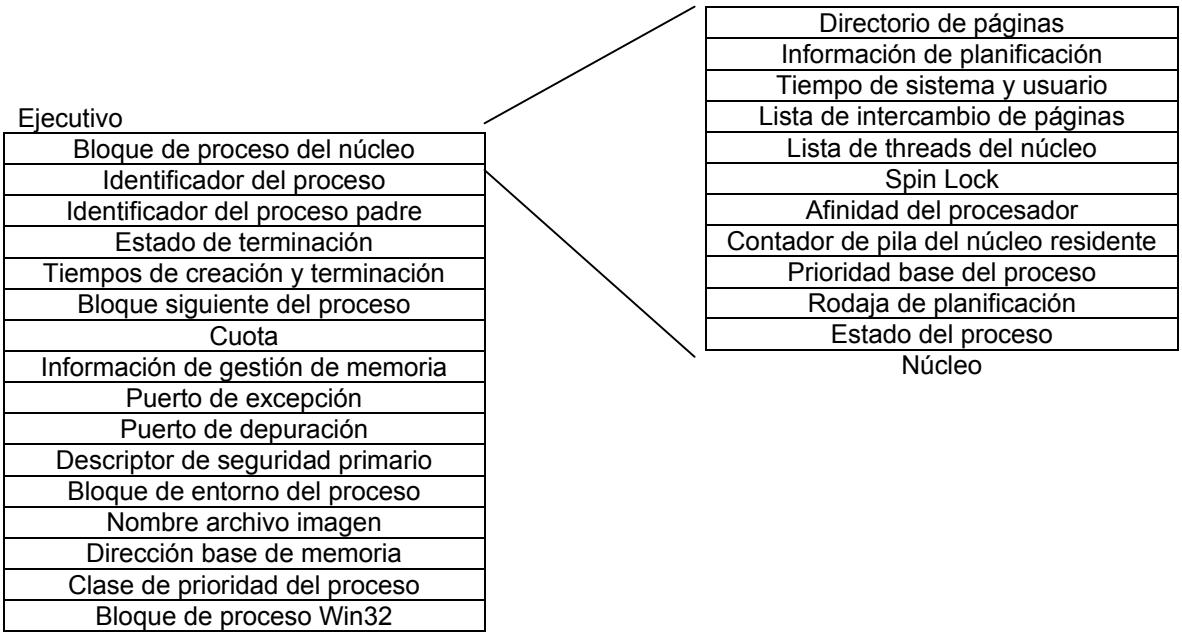


Figura 12.5. Bloque descriptor de un proceso.

to. La roda se indica en el descriptor del objeto como un valor de 6 bits, divididos en grupos de 2 de la siguiente manera:

Bits 0 y 1. Modificaciones de la rodaja. Sirve para activar la ejecución de threads en primer plano.
Bits 2 y 3. Rodaja de planificación con valor variable o fijo.
Bits 4 y 5. Rodaja de planificación corta o larga.

Con estos parámetros el sistema planifica los threads, pero no existe un planificador central que esté siempre activo, sino que las rutinas de planificación son manejadores de eventos disparados por los threads (bloqueo voluntario), los temporizadores (rodaja de planificación) o interrupciones (entrada/salida) del sistema.

ACLARACIÓN 12.1

Los bits que se usan para describir o cualificar la rodaja de planificación se usan como máscaras de bits en pares. El efecto de estas máscaras es complementario, lo que permite definir una rodaja corta con valor variable o una rodaja corta con valor fijo. El valor de la rodaja varía dependiendo de la versión del sistema operativo, siendo mayor en configuraciones para servidores de datos o de proceso.

12.5.3. Gestor de memoria virtual

Windows NT proporciona un modelo de memoria virtual con paginación por demanda y sin preasignación de espacio de intercambio. Cada proceso dispone de una capacidad de direccionamiento de 4 GB (32 bits), de los cuales 2 GB son para el programa de usuario y 2 GB se

reservan para el sistema. La Figura 12.6 muestra el modelo de memoria de un proceso en Windows NT. Este modelo es muy similar al existente desde hace años en el sistema operativo VMS de DEC. El espacio del proceso contiene la aplicación en ejecución (ejecutables y bibliotecas con enlace dinámico), un módulo de pila para cada thread para almacenamiento dinámico y el almacenamiento estático definido por la aplicación. El espacio de sistema contiene el ejecutivo, el núcleo, los manejadores de dispositivo, pilas de ejecución del ejecutivo, una pila para cada thread en ejecución, el HAL, etc. El espacio del proceso es único por proceso y todo lo que se ejecuta dentro de él se hace en modo usuario. El espacio del sistema, sin embargo, es global a todos los procesos y se divide en dos Partes: 1 GB para información compartida por los usuarios (DLL, memoria compartida, etc.), que se ejecuta en modo usuario, y 1 GB para el sistema operativo, que se ejecuta en modo núcleo.

La memoria física disponible en Windows NT depende del tamaño de la tabla de páginas que se puede direccionar con cada arquitectura. Actualmente se dispone de 4 GB en arquitecturas Intel x86 y 8 GB en procesadores Alpha. La última versión de Windows NT incluye **direccionamiento** extendido que permite usar hasta 4 bits más para direccionar memoria. Con este método se puede disponer de hasta 64 GB direccionables en versiones del sistema operativo para servidores de datos. Este tamaño permite mantener más procesos en memoria al mismo tiempo (hasta 32 procesos con ocupación máxima de 2 GB) sin hacer uso de la memoria virtual y sus archivos de páginas.

El gestor de memoria virtual es responsable de relacionar direcciones virtuales del espacio de direcciones de cada proceso con direcciones físicas, protegiendo la memoria que usan los threads del proceso para asegurar que no pueden acceder a la de otros procesos si no tienen la debida autorización. Para relacionar espacio lógico y físico, el sistema operativo mantiene una **tabla de páginas** (PTE, *Page Table Entry*). Cada una de sus entradas está vacía o hace referencia a un marco de memoria física o un bloque del archivo de páginas. Para optimizar el acceso a la tabla de páginas se usa un **directorio de página**, que no es otra cosa que un índice a la tabla de páginas. La Figura 12.7 muestra cómo se traduce una dirección de memoria virtual en la arquitectura. La dirección virtual está compuesta por 32 bits, divididos en tres grupos de 10, 10 y 12 respectivamente. Los diez primeros bits de la dirección se usan para identificar en la tabla de directorios la tabla de páginas dentro del espacio del proceso (hasta 512 páginas) o del sistema (hasta 512 páginas) a la que se refiere la dirección virtual.

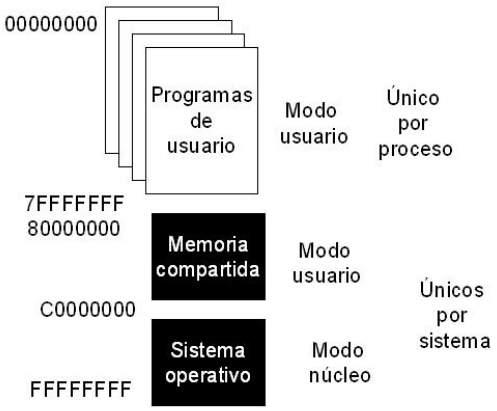


Figura 12.6. *Modelo de memoria de un proceso.*

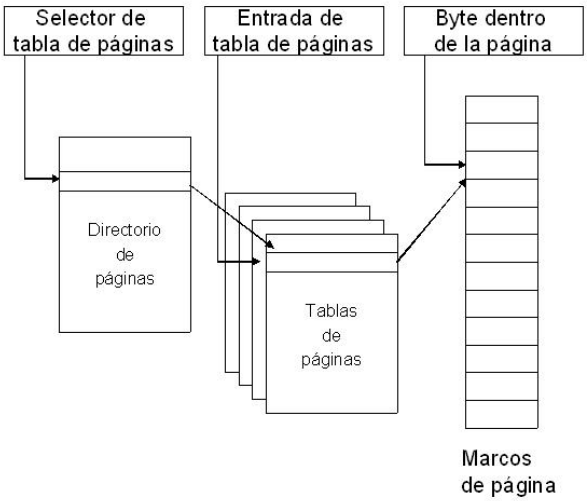


Figura 12.7. *Traducción de una dirección de memoria virtual.*

Los 10 bits siguientes indican la entrada de página dentro de la tabla de páginas seleccionada, hasta un máximo de 1.024 páginas. Los 12 bits siguientes indican la posición dentro de la página virtual, cuto tamaño es de 4 KB.

ACLARACIÓN 12.2

Este esquema es similar a la traducción de direcciones de memoria con tres niveles que se proponen en otras arquitecturas. Lo que varía es la denominación del nivel superior (directorio, nivel primario, etc.).

Cada proceso tiene un conjunto de páginas de trabajo en memoria, representado por una lista de páginas que se define como su conjunto de trabajo. Además de esta estructura de datos, el gestor de memoria mantiene varias listas asociadas a la política de gestión:

- **Lista de páginas libres.** Almacena las páginas libres, es decir, que han sido asignadas alguna vez y liberadas posteriormente. Existen en el archivo de intercambio.
- **Lista de páginas a cero.** Páginas sin iniciar. Cuando una página se referencia por primera vez se asigna desde esta lista.
- **Lista de páginas modificadas.** Páginas reemplazadas en memoria y cuyos contenidos han sido modificados (escritos).
- **Lista de páginas en espera.** Lista que aglutina páginas modificadas que todavía no se liberan para evitar sobrescribirlas de forma prematura.

La Figura 12.8 muestra la **gestión de páginas** de memoria en Windows NT. Cuando hay un fallo de página en un proceso, el gestor de memoria aplica una política de reemplazo LRU sobre el conjunto de trabajo del proceso. Si la página ha sido modificada, la almacena en la lista de páginas modificadas (paso 1). Si no, pasa a la lista de páginas en espera (paso 2). En el caso de que haya

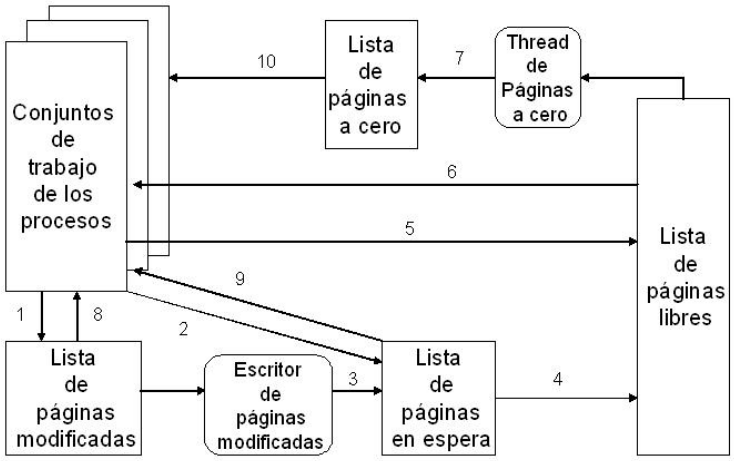


Figura 12.8. *Gestión de páginas de memoria.*

páginas modificadas, existe un thread escritor de páginas modificadas que las lee de la lista anterior, las escribe y las pasa a la lista de páginas en espera (paso 3). En cualquiera de los dos casos, cuando hace falta espacio o se cumple un determinado tiempo sin ser referenciadas, las páginas de la lista en espera pasan a la lista de bloques libres (paso 4). A esta lista pueden llegar también páginas cuando se libera un proceso o un conjunto de páginas de memoria de forma explícita (paso 5). Si después de pasar un cierto tiempo las páginas libres siguen sin ser usadas (o si hay recolección de basura en memoria), el thread de páginas a cero libera los recursos de estas páginas y las considera no asignadas (paso 7). Existen varias posibilidades para la página nueva referenciada: estar en la lista de páginas modificadas o en espera, en cuyo caso se trae tal cual (pasos 8 y 9); que se hubiera liberado anteriormente, en cuyo caso se traerá de la lista de páginas libres (paso 6); que sea la primera vez que se referencia, en cuyo caso se trae de la lista de páginas a cero (paso 10).

12.5.4. Llamada a procedimiento local

Las aplicaciones de usuario y los subsistemas de entorno de ejecución no se comunican directamente en Windows NT sino que pasan a través del ejecutivo del sistema operativo por criterios de protección y de compartición de recursos. El ejecutivo se encarga de optimizar estas comunicaciones mediante una utilidad de Llamada a procedimiento local dPC, *Local Procedure Call*), algo muy similar al concepto de llamada a procedimiento remoto pero optimizada para procesos que comparten memoria. De esta forma, cuando una aplicación se quiere comunicar, con otra, sus mensajes pasan a través del subsistema de su entorno hasta el gestor de LPC (Fig. 12.9).

Las aplicaciones no son conscientes de esta forma de comunicación porque el uso de las LPC está oculto dentro de las DLL (*Dynamic-Link Libraries*) que proporciona el sistema operativo. Cuando se genera una aplicación que usa un determinado sistema de ejecución se enlaza con la DLL de este subsistema, que incluye los mecanismos de comunicación a través de LPC. Estos mecanismos incluyen el uso de suplentes que se encarga de empaquetar, o desempaquetar, los parámetros y de la comunicación. Esta última se optimiza mediante el uso de memoria compartida.

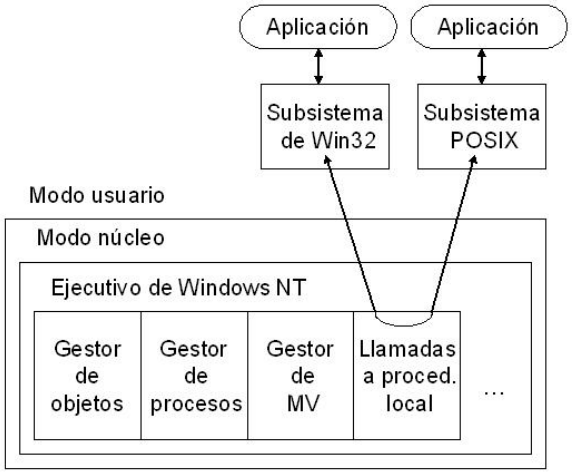


Figura 12.9. *Comunicación mediante LPC.*

PRESTACIONES 12.1

Los sistemas operativos con diseño por capas han tenido siempre fama de ser más lentos que los monolíticos debido a la sobrecarga de tiempo que en las llamadas al sistema constituyen los mensajes de unos niveles a otros. Para reducir esta sobrecarga, los sistemas operativos comerciales «cortocircuitan» los niveles locales para proporcionar mecanismos de comunicación más rápidos basados en el uso de memoria compartida.

12.5.5. Gestor de entrada/salida

El sistema de entrada/salida de Windows NT está construido Como un conjunto de manejadores apilados, cada uno de los cuales está asociado a un objeto de entrada/salida (archivos, red, etc.). Ofrece a las aplicaciones y entornos de ejecución servicios genéricos (*CreateFile, ReadFile WriteFile, CloseHandle,* etc.) que permiten manejar los objetos de entrada/salida del sistema. A través de ellos se puede acceder a todos los manejadores de archivos y de dispositivos tales como discos, cintas, redes, consola, tarjetas de sonido, etc.

La arquitectura del sistema de entrada/salida (Fig. 12.10) es compleja y está estructurada en capas, cada una de las cuales tiene una funcionalidad bien definida:

- **Gestor de entrada/salida**. Proporciona servicios de E/S síncrona y asíncrona a las aplicaciones y una interfaz homogénea para poderse comunicación con los manejadores de dispositivo sin saber cómo funcionan realmente.
- **Gestor de cache**. Optimiza la entrada/salida mediante la gestión de almacenamiento intermedio en memoria, tanto para los sistemas de archivos como para las redes. El tamaño de la cache varía dinámicamente en función de la memoria RAM disponible. La política de escritura es la de escritura perezosa, según la cual se retrasa la escritura de los bloques sucios hasta que la UCP tenga una tasa de utilización baja. Las aplicaciones, sin embargo, pueden decidir no usar la cache o cambiar la política por otra de escritura inmediata.

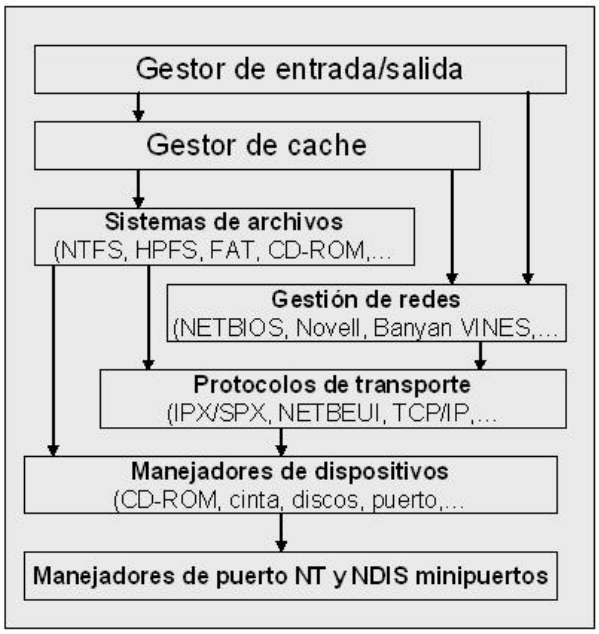


Figura 12.10. *Arquitectura del sistema de entrada/salida.*

- **Manejador de sistemas de archivos.** Proporciona una interfaz homogénea para acceder a todos los sistemas de archivos que proporciona Windows NT (NTFS, HPFS, FAT, etc.). Además, permite acceder a los manejadores de los dispositivos de almacenamiento de forma transparente, incluyendo, si es necesario, accesos remotos a través de redes. Además, los servidores para cada tipo de sistema de archivos se pueden cargar y descargar dinámicamente como cualquier otro manejador.
- **Gestor de redes.** Proporciona una interfaz homogénea para acceder a todos los sistemas de red que proporciona Windows NT (TCP/IP, Novell, etc.). Además permite acceder a los manejadores de cada tipo de red particular de forma transparente.
- **Manejadores de dispositivo.** Proporcionan operaciones de alto nivel sobre los dispositivos y las traducen en su ámbito interno a operaciones de control de cada dispositivo particular.
- **Gestores de puertos y minipuertos.** Toda la entrada/salida de Windows NT se comunica con los dispositivos locales mediante dos puertos: NT para dispositivos locales y NDIS para dispositivos remotos. Estos a su vez usan minipuertos (SCSI, CD-ROM, NIC, etc.) específicos para cada dispositivo y configurados para la plataforma hardware específica.

Cada uno de los componentes anteriores se considera un objeto del sistema, por lo que es muy sencillo crear el sistema de entrada/salida de forma dinámica, así como reemplazar manejadores de archivos y dispositivos. Además, para mantener la compatibilidad con aplicaciones de l6 bits y permitir que piensen que tienen acceso directo a los puertos de entrada/salida, se proporcionan manejadores virtuales para puertos serie, paralelos, ratón, teclado, etc.

La entrada/salida en Windows NT es inherentemente asíncrona, aunque las aplicaciones puedan pensar que es síncrona porque se bloquean los procesos que ejecutan tales instrucciones. A partir del gestor de entrada/salida los manejadores se comunican intercambiando paquetes de petición de E/S (IRP, *I/O Request Packets*).

Estos paquetes describen las peticiones de E/S e incluyen, entre otras cosas, el tipo de operación, la dirección de memoria en el destino, la cantidad de datos a transferir o un apuntador al objeto manejador que necesitan. Todas ellas se encolan en una lista global desde la cual se distribuyen posteriormente a la lista particular de cada dispositivo, cuya gestión es guiada por eventos de interrupción. A partir de esta lista de peticiones, los manejadores de dispositivo acceden a cada dispositivo a través de las rutinas que proporciona Windows NT, rutinas con una interfaz común pero con un cuerpo distinto dependiendo de la plataforma hardware en que se haya instalado el sistema. El procesador se ve como un dispositivo más del sistema. La Figura 12.11 muestra una traza de una petición de E/S síncrona en Windows NT.

PRESTACIONES 12.2

Al igual que con la comunicación, en el sistema de entrada/salida de Windows NT hay una forma de saltarse los pasos que se muestran en la Figura 12.11 y acceder directamente a los dispositivos. Se denomina E/S Rápida (*Fast I/O*).

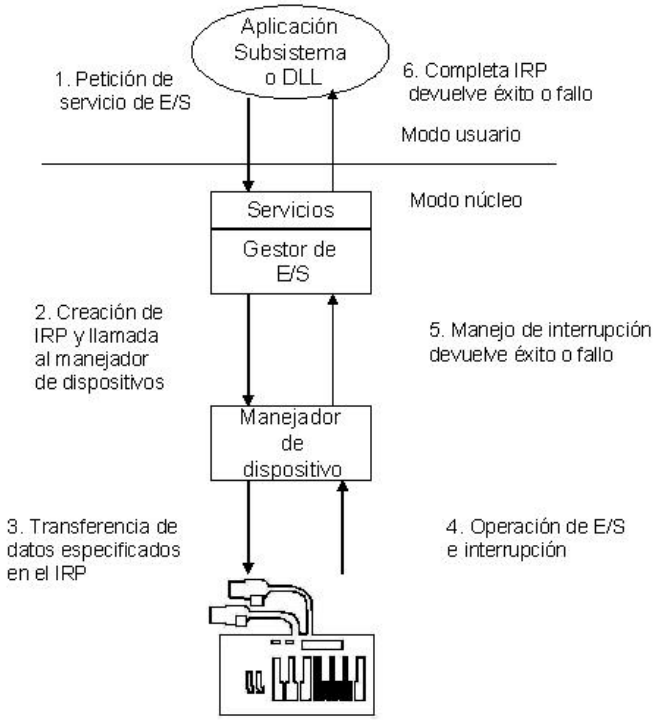


Figura 12.11. *Pasos de una petición de E/S síncrona.*

El gestor de cache

Este componente es Fundamental para la optimización de la E/S del sistema operativo Windows NT. Su misión es gestionar la **cache de archivos** de Windows NT, una cache única en el ámbito del sistema y común para todos los tipos de sistemas de archivos locales y remotos. Esta cache no se gestiona sobre la base de bloques de archivos, sino mediante **bloques virtuales**, agrupaciones de bloques de archivo de 256 KB que se proyectan en zonas de memoria virtual. Su tamaño varía dinámicamente en función de la memoria RAM disponible, siendo esta característica controlada por el gestor de memoria virtual. Con esta característica, cada vez que se abre un objeto archivo se le habilitan apuntadores a los bloques virtuales de la cache, lo que le permite trabajar con su propia imagen de cache. Sin embargo, todas estas representaciones virtuales se proyectan sobre una única representación de marcos de memoria física, a través de los cuales se hace el acceso final a los datos de un archivo.

La estructura de la cache se representa mediante varias estructuras de datos, como se puede ver en la Figura 12.12. La primera es un vector que representa a los bloques virtuales en la cache del sistema, denominado Direcciones virtuales de bloques de control (VACB, *Virtual Address Control Block*). Representa el estado de las vistas de archivos en la cache del sistema que incluye para cada VACB:

- La dirección virtual en la cache del sistema de la vista del archivo.
- Apuntador al mapa de cache compartida del sistema, que contiene el mapa de cada archivo en la cache.
- Posición en el archivo de los datos de la vista.
- Contador de activaciones, es decir, el número de procesos o threads que están usando simultáneamente esa vista del archivo.

La segunda es una estructura que relaciona el objeto archivo con la cache, indicando qué vistas del objeto están presentes en la cache. Se denomina Apuntadores a secciones de objetos y es una estructura del sistema a la que pueden apuntar varios objetos archivo, propiciando así el

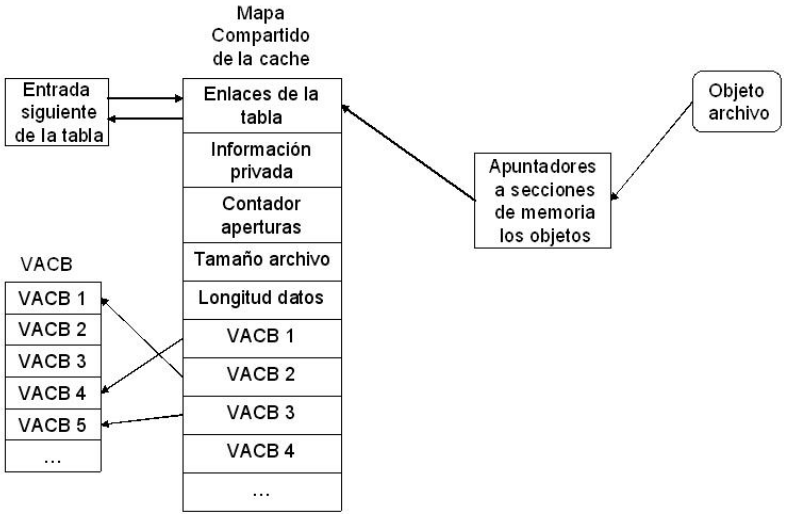


Figura 12.12. *Estructura de la cache de Windows NT.*

uso compartido y coherente de las vistas. A través de esta estructura, la tabla de objetos archivo en el sistema enlaza con la cache.

La tercera estructura de datos es una lista doblemente encadenada que se denomina Mapa compartido de cache. Cada entrada de esta lista muestra el mapa de un archivo cuyos datos están presentes en la cache. Para ello se incluyen en dichas entradas datos como un contador de aperturas, el tamaño del archivo o el conjunto de VACB del archivo presentes en la cache. A través de los VACB se enlaza con la primera estructura de datos.

Para la gestión de la cache descrita se aplica **política de escritura retrasada perezosa** y **semántica de coherencia tipo UNIX.** La primera consiste en acumular los datos de escritura en memoria hasta que son volcados a disco por un thread del sistema, denominado escritor perezoso, de forma agrupada para reducir el número de operaciones de escritura. Este thread está controlado por el gestor de memoria virtual, que lo arranca cuando necesita memoria, cuando el número de páginas escritas sobrepasa un cierto número o de forma periódica cada segundo. En este último caso, se vuelcan a disco una cuarta parte de las páginas que han sido escritas. ¿Cómo se calcula este valor? Cuando arranca el sistema, en función de la memoria RAM disponible y de las opciones de configuración del sistema operativo (cache estándar, grande, servidor, etc.), se calcula un umbral de escritura que sirve como referencia al gestor de memoria. La semántica de tipo UNIX permite que cada proceso vea siempre la versión más reciente de los datos. El gestor de memoria virtual es el encargado de proporcionar esta semántica, para lo que mantiene siempre una única copia de los datos en memoria física a la que apuntan las tablas de páginas de los objetos que comparten el archivo.

Las políticas anteriores son las que se instalan por defecto en el sistema. Sin embargo, Windows NT es muy flexible y permite elegir para cada archivo desde una política de escritura inmediata hasta no hacer uso de la cache, pasando por no volcar nunca los datos a disco.

12.6. SUBSISTEMAS DE ENTORNO DE EJECUCIÓN

Uno de los principales objetivos de diseño de Windows NT era su compatibilidad con entornos de ejecución provistos anteriormente por Microsoft (como MS-DOS o Windows 3.x) y con otras interfases de sistemas operativos (como POSIX u OS/2). La solución adoptada para proporcionar esta compatibilidad fue el uso de entornos virtuales, implementados como procesos de Windows NT que emulan el entorno de cada sistema operativo específico. Estos componentes, que se denominan subsistemas de entorno de ejecución, se relacionan entre sí y con el sistema como se muestra en la Figura 12.13. Actualmente, Windows NT proporciona los siguientes entornos:

- Subsistema OS/2.
- Subsistema POSIX.
- Subsistema Win32.
- Máquinas virtuales para emular el entorno de MS-DOS y aplicaciones de Windows 3.x de 16 bits (Win16).
- Subsistema de seguridad.

Como se puede ver, cada entorno es un proceso independiente que ejecuta en modo usuario, por lo que su fallo no afectará a otros subsistemas o al sistema operativo. Todos ellos son opcionales excepto el subsistema de Win32, que es la interfaz mínima que necesita el sistema operativo para manejar teclado, ratón y pantalla.

El subsistema de Win32 es la interfase que controla toda la interacción con los usuarios. Entre sus funciones principales se encuentran la implementación de colas de E/S para los dispositivos que maneja el usuario y la creación de objetos, para lo que interacciona con

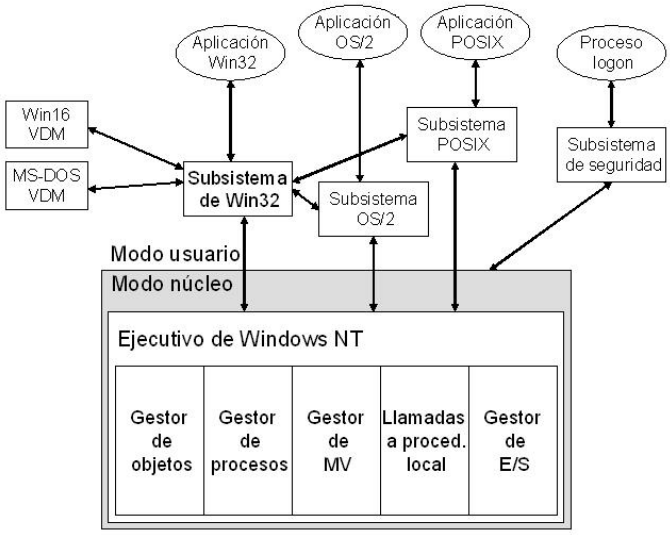


Figura 12.13. *Visión conceptual de los subsistemas de entorno de ejecución. (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)*

prácticamente todos los componentes del ejecutivo de Windows NT. Los otros subsistemas contactan con él para llevar a cabo las dos tareas anteriores. Cuando se crea un proceso se lo indican al subsistema de Win32, que se encarga de contactar con el gestor de objetos para crear el objeto de tipo proceso, con el gestor de procesos para crear el bloque descriptor del proceso, con el gestor de memoria virtual para cargar el ejecutable, etc. Cuando se carga el ejecutable comprueba a qué tipo de entorno de ejecución pertenece y, si no está activo, lo arranca. Además contacta con las utilidades de llamadas a procedimiento lo local para crear los suplentes necesarios para el proceso y con el gestor de E/S para habilitar puertos de comunicación con dicho proceso. Las aplicaciones de tipo Win32 disponen de una cola específica para E/S asíncrona, mientras las de otros entornos se reconducen a través de una única cola por entorno.

ACLARACIÓN 12.3

Todas las llamadas al sistema mostradas en capítulos anteriores, para el caso del sistema operativo Windows NT, se refieren al subsistema de entorno de ejecución de Win32. Sin embargo, las llamadas de POSIX mostradas son perfectamente compatibles con el subsistema POSIX 1 que incorpora Windows NT.

12.7. SISTEMAS DE ARCHIVOS DE WINDOWS NT.

Windows NT permite la existencia de varios tipos sistemas de archivos, ya que su diseño modular le capacita para incluir sin ningún problema todos los tipos de sistemas de archivos que desee el usuario. El origen de los sistemas de archivos de Windows NT está en lograr la compatibilidad con

otros sistemas existentes en el mercado. La primera versión de Windows NT incluía sistemas de archivos tipo FAT, provenientes de MS-DOS y Windows 3.x. Estos sistemas de archivos tienen dos problemas principales: bajo rendimiento e incapacidad de gestionar dispositivos grandes. Para tratar de resolver este problema, en 1990. Microsoft incluyó un nuevo tipo de sistemas de archivos: el HPFS (*High Performance File System*). Este sistema de archivos permitía manejar dispositivos grandes de forma eficiente en las aplicaciones para el subsistema de OS/2. A estos sistemas de archivos se unió, a mediados de los noventa, el sistema de archivos NTFS (*NT File System*). Es el sistema de archivos más moderno del sistema operativo y permite explotar adecuadamente discos de gran tamaño y arquitecturas multiprocesador.

En esta sección se describen los tres tipos de sistemas de archivos principales que se incluyen en Windows NT: FAT, HPFS y NTFS. Además de estos sistemas de archivos, Windows NT proporciona otros para CD-DOM, para servidores, para comunicación entre procesos (como el NPFS, *Named Pipes File System*, y el MSFS. *Mailslot File System*), para registro (como el LFS. *Log File System*) y para sistemas distribuidos (como el DFS. *Distributed File System*).

12.7.1. Sistemas de archivos tipo FAT

Estos sistemas de archivos usan la FAT (File Allocation Table) como medio para representar los archivos. Cada archivo es una lista enlazada de bloques de la FAT, como se vio en el Capítulo 8. Los sistemas de archivos de este tipo se usan habitualmente a través del subsistema de entorno de MS-DOS y proporcionan direccionamiento con 16 bits, permitiendo particiones de hasta 32 MB. Para incrementar su capacidad de direccionamiento se usan agrupaciones de bloques, como se vio en el Capítulo 8, y en la última versión de Windows NT se ha incluido el sistema de archivos FAT para 32 bits.

La Figura 12.14 muestra la estructura de un sistema de archivos tipo FAT. En primer lugar se encuentra el **bloque de carga** de la partición con los parámetros que le indican a la BIOS dónde se encuentra el sistema operativo en caso de que sea una partición activa. A continuación hay dos copias de la **información de la FAT**. La segunda es redundante y se incluye para dotar con más tolerancia a fallos al sistema de archivos, ya que en caso de fallo de acceso a la FAT gran parte de los archivos quedarían inaccesibles. El tamaño que ocupa la FAT puede ser considerable. Para un disco de 8 GB, usando direcciones de bloque de 32 bits y bloques de 4 KB, sería necesaria una FAT de 8 MB. El tercer componente del sistema de archivos es el directorio raíz, que incluye una entrada de 32 bytes para cada directorio del sistema, a través de la cual se puede acceder a subdirectorios y archivos. En el Capítulo 8 se comentó en detalle el formato de esta entrada de directorio. Los bloques restantes del sistema de archivos constituyen la denominada área de datos o de archivos, es decir, los bloques que contienen los datos de los archivos.

El sistema de archivos tipo FAT presente en Windows NT tiene varias mejoras respecto a los que había en MS-DOS y Windows 3.x:

- Atributos temporales que indican tiempo de creación y modificación del archivo.
- Extensión de los bits de atributos de archivo (archivo, sistema, escondido y sólo lectura) para indicar entradas de subdirectorios y volúmenes.
- Nombres de archivos de hasta 256 caracteres (frente a los 8 de MS-DOS).
- Direccionamiento con 32 bits.

La gran desventaja de esta solución es que la FAT puede ocupar mucho espacio si el dispositivo es grande. Ya hemos visto que un volumen de 8 GB, con 4 KB como tamaño de bloque, necesitaría una FAT de 8 MB. Para buscar un bloque de un archivo muy disperso podría ser necesario recorrer toda la FAT y, por tanto, tener que traer todos los bloque de la FAT a memoria.

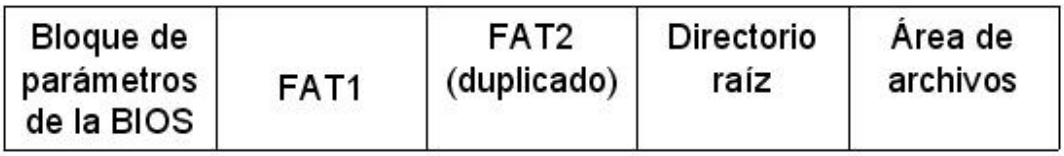


Figura 12.14. *Un sistema de archivos tipo FAT. (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)*

En general, la FAT de MS-DOS es muy lenta cuando se accede aleatoriamente a un archivo grande. La razón es que no se sabe dónde está un bloque de un archivo si no se sigue toda la cadena de bloques del mismo. Si una aplicación salta de un lado a otro del archivo, es necesario empezar cada vez desde el principio. Imagine qué pasaría si la computadora tuviese ocho dispositivos como éste: se necesitarían 64 MB de memoria sólo para las FAT. Este método es pues inviable si la FAT no puede estar continuamente en memoria, lo cual ocurre en los dispositivos de tamaño medio.

12.7.2. Sistema de archivos de alto rendimiento (HPFS)

Los sistemas de archivos tipo FAT fueron diseñados pensando en accesos secuenciales y en volúmenes pequeños. Con la incorporación del entorno de OS/2 se vio que era necesario disponer de un tipo de sistemas de archivos que pudiese gestionar volúmenes más grandes de forma más eficiente. Para lograr este objetivo, se diseñaron los Sistemas de archivos de alto rendimiento (HPFS, *High Performance File System*).

Los HPFS tienen una estructura completamente distinta a la de la FAT (Fig. 12.15) ya que el volumen se divide en **bandas**, cada una de las cuales tiene su propio mapa de bits junto a ella. Cuando se da formato a un volumen, se reservan 18 sectores para el bloque de carga, el superbloque y un «bloque de repuesto», que sirve para duplicar el superbloque y aumentar la tolerancia a fallos. A partir de estos bloques se colocan las bandas, definidas por espacios de 16 MB para datos y 2 KB para mapas de bits de la zona de datos adjunta. Los mapas de bits se colocan a los extremos de las bandas, pero de Forma alternativa para permitir una zona de datos contiguos de hasta 16 MB. Esta idea, muy similar a la de los grupos de cilindros del FFS de UNIX (Capítulo 8), permite reducir la zona de búsqueda de los archivos, pero presenta problemas de fragmentación de las bandas y de extensión de los archivos. ¿Qué ocurre si un archivo no cabe en una banda de 16 MB? La solución de HPFS es buscarle un hueco adecuado, para lo que mantiene listas de huecos en memoria.

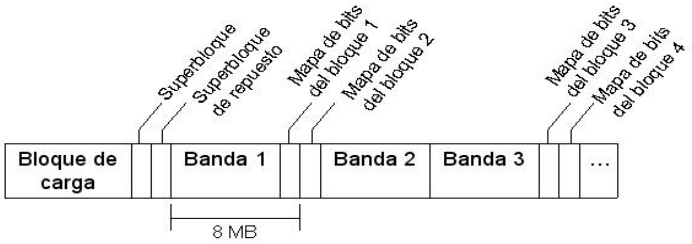


Figura 12.15. *Un sistema de archivos tipo HPFS. (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)*

Estas listas se elaboran cuando se abre el sistema de archivos y se mantienen actualizadas con las operaciones de creación y borrado de archivos.

El sistema de archivos tipo HPFS presente en Windows NT tiene varias mejoras respecto a los sistemas de archivos de tipo FAT:

- Nombres de archivos de hasta 256 caracteres (frente a los 8 de MS-DOS).
- Estructuración del volumen en bandas.
- Árboles binarios de directorios cuyos nodos se denominan *Fondees*. Son estructuras de 5 12 bytes que contienen un nombre de archivo, su longitud, atributos, ACL y situación de los datos del archivo (número de banda).

Sin embargo, HPFS tiene dos problemas serios:

1. Fragmentación externa, cuya incidencia depende del tamaño de archivo de los usuarios y de su disposición en las bandas.
2. Asignación de espacio usando sectores de disco como unidad de asignación. Esto con lleva que la mayoría de los bloques lógicos se compondrán de varios sectores y que el sistema de archivos se debe encargar de ocultar estos detalles. Por tanto, si se quiere tener bloques mayores de un sector, o agrupaciones de sectores, es necesario gestionarlo en la capa del gestor de archivos.

PRESTACIONES 12.3

Resolver la fragmentación no es sencillo debido a que las políticas de asignación que tratan de llevar a cabo ajustes óptimos de huecos son muy lentas. La solución más plausible es ejecutar de forma periódica algún thread de desfragmentación del disco para compactar las bandas.

12.7.3. NTFS

Los dos sistemas de archivos anteriores tienen serias limitaciones si se quieren usar en grandes instalaciones, donde es necesario tener archivos de gran capacidad y muy eficientes. NTFS (*NT File System*) es el sistema de archivos más moderno de Windows NT e incluye soluciones de diseño nuevas, lo que permite resolver los inconvenientes de los sistemas anteriores y proporcionar una combinación de rendimiento, fiabilidad y compatibilidad ausente en sistemas anteriores. Es el sistema de archivos asociado al subsistema de entorno de Win32.

Las principales características de diseño de NTFS son:

- Operaciones de alto rendimiento sobre archivos y discos muy grandes. Usa agrupaciones como unidad de asignación y 64 bits para numerar los bloques o grupos.
- Nuevas características de seguridad, incluyendo ACL sobre archivos individuales. recuperación de archivos, integridad de datos, etc. Toda la seguridad se gestiona a través del monitor de referencia de seguridad de Windows NT.
- Implementación de todos los componentes del volumen como objetos archivos (concepto similar a UNIX) que tienen atributos de usuario y de sistema (nombre. tiempos. contenidos, etcétera).
- Archivos con múltiples flujos de datos que pueden tener nombre (archivo: flujo) y ser manipulados de forma totalmente independiente. Además cada hijo tiene sus propios atributos de tiempo, tamaño, asignación, etc. Esta característica permite manejar como única unidad de datos relacionados (p. ej.: metadatos y datos) aunque estén en dispositivos distintos.

- Modelo transaccional que permite efectuar operaciones de E/S de forma atómica y recuperar un estado coherente del sistema de archivos en caso de fallos. Se puede complementar con mecanismos de almacenamiento redundante de datos.
- Funcionalidad compatible con POSIX 1, lo que incluye diferenciar los nombres según usen mayúsculas y minúsculas y proporcionar enlaces «físicos», entre otras cosas.
- Nombres largos de hasta 256 caracteres en formato Unicode de 16 bits. Esta característica hace que los nombres de NTFS puedan ser compatibles con los de MS-DOS y los de HPFS.

Para satisfacer estos objetivos de diseño, el sistema de archivos y los archivos de NTFS tienen una estructura muy distinta de tos de tipo FAT o HPFS. A continuación se describen ambas brevemente.

Estructura del sistema de archivos y los archivos de NTFS

Un sistema de archivos de NTFS es una organización lógica que permite almacenar archivos de tipo NTFS en un volumen de disco. Un volumen no es sino una forma de combinar múltiples fragmentos de disco para formar una unidad lógica. En el caso de Windows NT, un volumen puede tener hasta 32 extensiones, pertenecientes a uno o más discos. Sobre estos volúmenes se crea un sistema de archivos de NTFS, cuya estructura (Fig. 12.16) se describe en un registro de un archivo especial contenido al principio del volumen que se denomina **MFT** (*Master File Table*). Se puede pues decir que el sistema de archivos de NTFS es únicamente un archivo, en el que los primeros 16 registros contienen información especial del sistema. A continuación hay una copia de repuesto del MFT, del cual hay otra copia redundante en el centro del volumen para incrementar la tolerancia a fallos. En caso de volúmenes extendidos a múltiples particiones o discos, estos datos del MFT están repetidos en todas las particiones para permitir interpretar el sistema de archivos desde cualquiera de ellos.

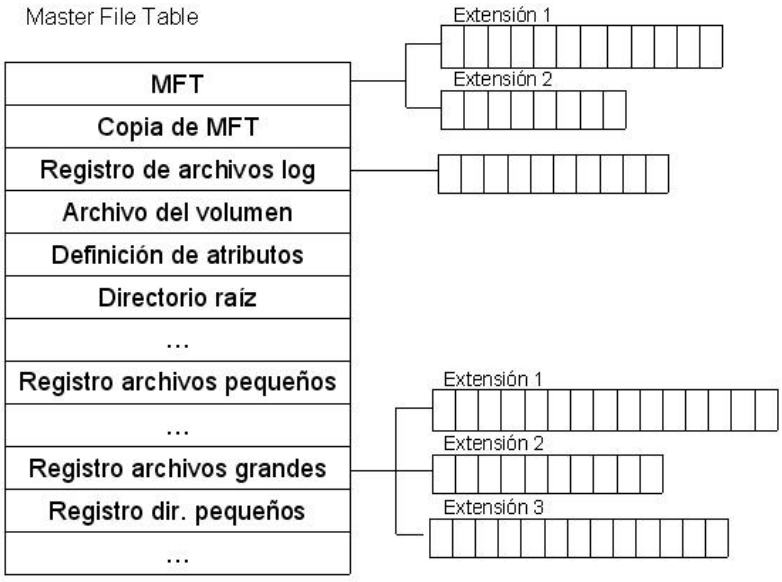


Figura 12.16. *Master File Table. (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)*

A continuación, hay dentro del MFT varios **archivos de metadatos**, que incluyen:

- **Registro** d*og*), usado para almacenar todas las operaciones que afectan a los metadatos del sistema de archivos. Básicamente es un registro transaccional que usa el *Log File System* para recuperar el sistema de archivos en caso de fallo.
- **Volumen**, que incluye información de los atributos del volumen (nombre, versión, creador, fechas, etc.).
- **Definición de atributos**, con los nombres de atributos y los valores definidos en el volumen, así como definiciones de lo que significa cada atributo.
- **Directorio raíz** ("\") del sistema de archivos, con información acerca de los archivos y directorios que cuelgan directamente de la raíz.
- **Mapa de bits del volumen**, con un bit por cada grupo de bloques (cluster) que indica si está libre o asignado a un archivo.
- **Carga**, con el programa cargador para el volumen, en caso de que sea un dispositivo de arranque. En otro caso existe igualmente, pero está vacío.
- **Grupos defectuosos** (*bad clusters*), que contiene una lista de los bloques defectuosos y su posición en el volumen. Este archivo se genera cuando se da formato a la partición del volumen o cuando se comprueba la situación de la superficie del disco (p. ej.: con *chkdsk*).

Estos archivos están ocultos y se denominan habitualmente archivos de sistema porque son usados por el sistema de archivos para almacenar los metadatos y otros datos de gestión o control del mismo.

A continuación, desde el registro 17 en adelante, se incluyen los **archivos y directorios** de usuario en el volumen. Para optimizar los accesos a disco se distingue en NTFS entre archivos y directorios pequeños, existiendo dentro de la MFT un archivo para describir cada uno de estos tipos, incluyendo una entrada distinta para cada extensión, en caso de que existan extensiones. La razón para esta distinción es que un registro de MFT puede almacenar:

- Atributos del archivo.
- Nombre del archivo.
- Descriptor de seguridad.
- Hasta 1,5 KB de datos.

Por ello, muchos archivos pequeños y directorios caben directamente en el registro de la MFT, sin que sean necesarias más estructuras de datos ni operaciones para acceder a los mismos. Esta característica, ya presente durante los años ochenta en sistemas de archivos de VMS o de Amoeba, permite optimizar mucho el acceso a archivos pequeños, ya que basta con un acceso a disco frente a dos o tres de UNIX o varios de los sistemas tipo FAT. Los archivos y directorios grandes también usan la zona de datos, pero en este caso para guardar datos y apuntar a otros dos registros de la MFT. Esta solución permite generar árboles binarios, que pueden tener apuntadores a otros archivos o directorios.

La parte del registro de la MFT que no contiene datos se usa para almacenar los **atributos** residentes que describen al archivo o directorio, cada uno de los cuales es identificado por un código y un nombre (accesibles en el archivo de metadatos de definición de atributos). Si son necesarios más atributos para representar las características del archivo, se les sitúa en otras zonas del volumen y se les denomina atributos no residentes. Algunos de los atributos existentes en archivos de tipo NTFS son: tiempos, contador de enlaces, nombre del archivo, descriptor de seguridad, índices para directorios largos, etc.

12.7.4. Comparación de tos sistemas de archivos FAT, HPFS y NTFS

La Tabla 12.1 resume las características principales de los tres sistemas de archivos descritos y existentes en la última versión de Windows NT.

12.8. EL SUBSISTEMA DE SEGURIDAD

La seguridad se incluyó como parte de las especificaciones de diseño de Windows NT, lo que permite lograr un nivel de seguridad C2 si se configura el sistema adecuadamente. El modelo de seguridad incluye componentes para controlar quién accede a los objetos, qué accesos pueden efectuar los usuarios sobre un objeto y qué eventos se auditan. Todos estos componentes juntos forman el subsistema de seguridad de Windows NT. Varios componentes del cual se ejecutan en modo núcleo para tener acceso a información interna de seguridad del sistema operativo de forma controlada. Esta filosofía permite aplicar los mismos procedimientos de seguridad a todos los objetos del sistema operativo.

Como se muestra en la Figura 12.17, el modelo de seguridad de Windows NT incluye los siguientes componentes:

- Procesos de *logon*, que muestran las ventanas de diálogo para que los usuarios puedan acceder al sistema, piden el identificador del usuario, su palabra clave y su dominio.
- Autoridad de seguridad local, que controla que el usuario tenga permiso para acceder al sistema. Es el corazón del sistema porque gestiona la política local, los servicios de autenticación, política de auditoria y registro de eventos auditados.

Tabla 12.1. *Comparación de sistemas de archivos FAT, HPFS, y NTFS.*

	Sistemas de archivos FAT	HPFS	NTFS
Nombre de archivo	Ocho.tres caracteres ASCII; En Windows NT 3.5, 255 caracteres formato Unicode y múltiples delimitadores	255 caracteres sin signo (8 bits útiles) y múltiples delimitadores	255 caracteres formato Unicode y múltiples delimitadores
Tamaño de archivo	2^{32} bytes	2^{32} bytes	2^{64} bytes
Partición	2^{32} bytes	2^{41} bytes	2^{64} bytes
Longitud del nombre	64 bytes; En Windows NT 3.5 sin límites	Sin límite	Sin límite
Atributos	Unos pocos bits para *flags* y unos pocos bits de atributos extendidos en Windows NT 3.5	Bit de flags y hasta 64 K de información para atributos extendidos	Todo, incluyendo los datos, se trata como atributos de archivos
Directorios	Lineales	Árbol binario	Árbol binario
Filosofía	Simple	Eficiente para discos grandes	Rápido, recuperable y seguro
Sistema de seguridad interno	No	No	Si

Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation

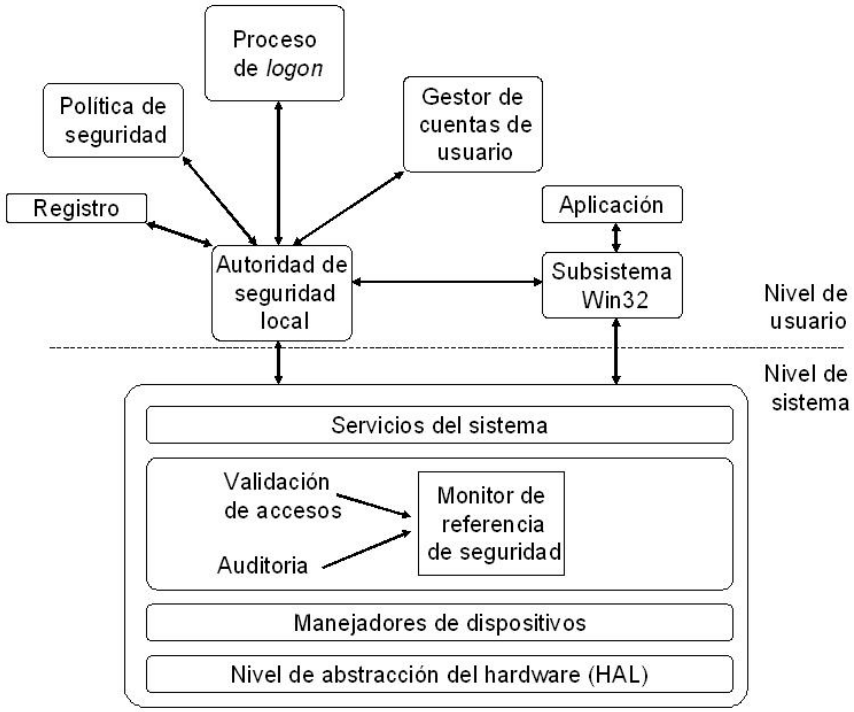


Figura 12.17. *Sistema de seguridad en Windows NT.*

- Gestor de cuentas de usuario, que mantiene las base de datos de usuarios y grupos. Proporciona servicios de validación de usuarios.
- Monitor de referencia de seguridad, que controla los accesos de los usuarios a los objetos para ver si tienen los permisos apropiados aplicando la política de seguridad y genera eventos para los registros de auditoría. Estos registros permiten al administrador seguir la pista a las acciones de los usuarios y detectar posibles violaciones de seguridad.

El modelo de seguridad mantiene información de seguridad para cada usuario, grupo y objeto del sistema y proporciona control de acceso discrecional para todos los objetos. Puede identificar accesos directos de un usuario y accesos indirectos a través de procesos que ejecutan en representación del usuario. Permite a los usuarios asignar permisos a los objetos de forma discrecional, pero si el usuario no asigna estos permisos, el subsistema de seguridad asigna permisos de protección por defecto. Cada usuario se identifica en este sistema mediante un identificador de seguridad único (SID, *Security ID*) durante la vida del usuario dentro del sistema. Cuando un usuario accede al sistema, la autoridad de seguridad local crea un descriptor de seguridad para acceso, lo que incluye una identificación de seguridad para el usuario y otra para cada grupo al que pertenece el usuario. Además, cada proceso que ejecuta en nombre del usuario tiene una copia de descriptor de seguridad del usuario.

12.8.1. Autenticación de usuarios

Windows NT tiene un subsistema de seguridad encargado de autenticar a los usuarios. Cuando se crea un nuevo usuario en el sistema, se almacena en una base de datos de seguridad una ficha del

usuario que incluye su identificador de seguridad, los grupos a los que pertenece, sus privilegios, grupo primario y enlace con su lista de control de acceso. Cuando el usuario quiere acceder al sistema introduce su contraseña a través de un proceso logon. Dicha contraseña se pasa al subsistema de seguridad, que verifica la identidad del usuario y, en caso positivo, construye una ficha de acceso para el usuario. Este objeto sirve como identificador oficial del proceso siempre que el usuario intente acceder a un recurso a partir de ese instante. La Figura 12.18 resume los atributos y servicios de este tipo de objeto, así como un ejemplo de Í'icha de acceso en Windows NT.

Habitualmente, cuando un usuario quiere acceder al sistema teclea CTRL-ALT-DEL. Esta combinación de teclas activa el proceso de logon, que presenta una pantalla o mensaje de entrada (paso 1 de la Figura 12.19) que pide tres valores:

- Identificación del usuario: nombre del usuario en el sistema.
- Palabra clave o contraseña: espacio pata teclear la clave (el eco muestra *).
- Dominio de protección al que pertenece el usuario.

Si hay error al introducir la clave durante un cierto número de veces, definido por el administrador del sistema, una medida habitual es bloquear la cuenta y notificar la situación al administrador de seguridad del sistema. En ambos casos se trata de evitar que los programas que intentan adivinar las claves del sistema se puedan ejecutar de forma normal o que lo tengan que hacer de forma tan lenta que tal detección sea inviable.

Tras capturar los datos del usuario, el proceso de *logon* (paso 2) se los envía al subsistema de seguridad, que activa el paquete de autenticación adecuado de entre los existentes (pasó 3). La misión de estos componentes es contrastar los datos de seguridad de los usuarios con los existentes en la base de datos de seguridad (paso 4). En caso de usuarios remotos es necesario contactar con el subsistema remoto adecuado. De cualquier forma, si los datos son satisfactorios, el gestor de la base de datos de seguridad devuelve el SID del usuario (paso 5). En caso contrario devuelve un error de validación. Con estos datos, el paquete de autenticación crea un identificador de seguridad para la sesión y se lo pasa, junto con el SID, al subsistema de seguridad (paso 6). Éste comprueba que los datos sean válidos, crea un descriptor de acceso con los datos anteriores y se lo pasa al proceso de *logon* (paso 7). En caso de error notifica el error al proceso de *logon*, y borra los datos de la sesión.

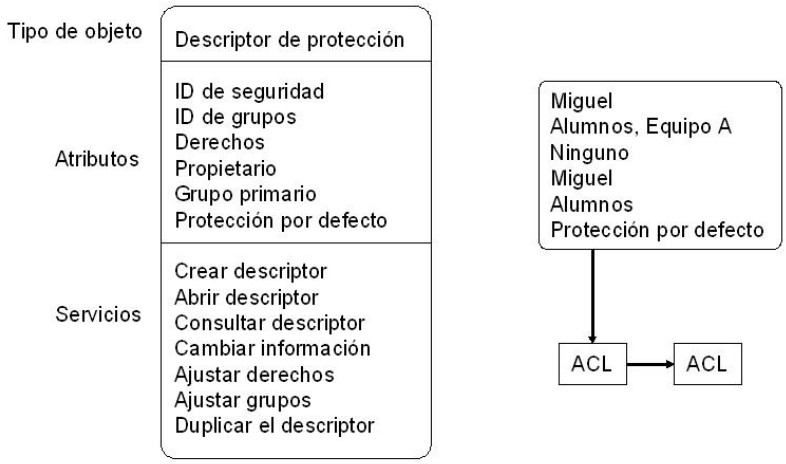


Figura 12.18. *Descriptor de seguridad en Windows NT. (Fuente: Microsoft Windows NT ServerResource Kit, Copyright © 2000 by Microsoft Corporation.)*

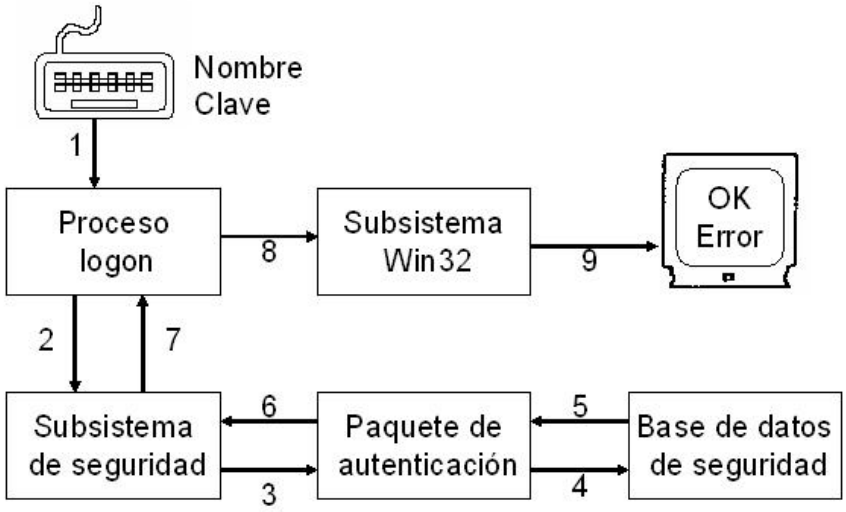


Figura 12.19. *Proceso de autenticación de un usuario en Windows NT.*

Por último, el proceso de *logon* abre una ventana gráfica en la que indica al usuario que el acceso ha sido concedido, en cuyo caso arranca el *Program Manager*, o denegado.

Una vez identificado y autenticado el usuario, cada vez que intenta acceder a un objeto protegido el Monitor de Referencia de Seguridad ejecuta procedimientos de validación de acceso en los que se comprueba si los permisos del usuario son suficientes para acceder al objeto. Estas rutinas de validación incluyen comprobaciones del descriptor de seguridad para comprobar la identidad y de las entradas de la lista de control de accesos asociada al objeto para ver si tiene una entrada (ACE, *Access Control Entry*) en la que permita al usuario efectuar la operación solicitada. Si la comprobación no es satisfactoria, se deniega el acceso al objeto.

12.8.2. Listas de control de acceso en Windows NT

Todos los objetos de Windows NT tienen asignados descriptores de seguridad como parte de sus fichas de acceso. La parte más significativa de los descriptores de seguridad es la lista de control de acceso. Normalmente, sólo el dueño del objeto puede modificar los derechos de la ACL para permitir o denegar el acceso al objeto. Cada entrada de la ACL contiene los descriptores de seguridad de los distintos dominios del sistema y los derechos de uso del objeto. El criterio de asignación de derechos en la ACL de un objeto nuevo en Windows NT es el siguiente:

1. Si el creador de un objeto proporciona una ACL de forma explícita, el sistema la incluye en la ficha de acceso de dicho objeto.
2. Si no se proporciona una ACL de forma explícita, pero el objeto tiene un nombre, el sistema mira si el objeto debe heredar la de los objetos de su directorio. En ese caso incluye en la ficha de acceso del objeto la ACL heredada de los objetos de su directorio.
3. Si ninguna de las dos condiciones anteriores se cumplen, el subsistema de seguridad aplica al objeto una ACL por defecto.

Además, en los descriptores de seguridad de los objetos se puede activar un campo de auditoría, que indica al subsistema de seguridad que debe espiar al objeto y generar informes de seguridad cuando algún usuario intente hacer un uso incorrecto del mismo. La Figura 12.20 muestra un ejemplo de comprobación de derechos de acceso en Windows NT. En ella el usuario miguel pide acceso de lectura al objeto texto.

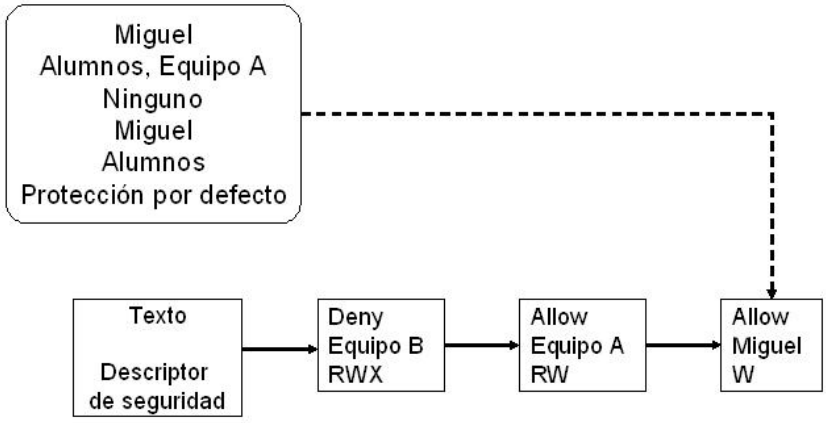


Figura 12.20. *Listas de control de acceso en Windows NT.*

El subsistema de seguridad recorre la lista de control de acceso hasta que encuentra la confirmación positiva en la tercera posición, devolviendo al usuario un manejador al objeto. Obsérvese que los primeros elementos de la lista son denegaciones de derechos. Esto se hace así para evitar que si el grupo del usuario miguel tiene denegado el acceso, dicho usuario no pueda acceder al elemento aunque una entrada de la lista se lo permita.

Las ACL están compuestas por entradas de control de acceso (ACE, *Access Control Entry*), de las que existen tres tipos básicos: dos para control de acceso discrecional y una para el sistema de seguridad. Las dos primeras sirven para permitir (*AccessAllowed*) o denegar (*AccessDenied*) accesos a un usuario o grupo de usuarios. La tercera se usa para guardar información de eventos de seguridad y para identificar al objeto en el registro de eventos de seguridad.

La implementación de los derechos de acceso se lleva a cabo mediante máscaras de bits, denominadas máscaras de acceso, que incluyen dos tipos de derechos:

- Estándar, que se aplican a todos los objetos (sincronización, dueño, escritura en ACL, borrado, etc.).
- Específicos, que sólo se aplican a un tipo de objeto determinado. Por ejemplo, para un objeto archivo, estos derechos incluyen permiso de lectura, de escritura, de ejecución, para añadir datos, etc.

Todos estos valores se rellenan cuando se crea un objeto con los valores que proporcione su creador, con valores heredados de la clase del objeto o con unos valores por defecto del sistema. Además, se pueden modificar dinámicamente usando los servicios de seguridad de Win32.

12.9. MECANISMOS PARA TOLERANCIA A FALLOS EN WINDOWS NT

El sistema operativo Windows NT incluye varios mecanismos para proporcionar tolerancia a fallos, algunos de los cuales se han mencionado ya. Estos mecanismos permiten salvaguardar el estado del sistema y de los datos para protegerlos ante fallos de dispositivos de almacenamiento.

Los principales mecanismos de tolerancia a fallos son:

- Utilidades para hacer copias de respaldo a cintas magnéticas.
- Registros transaccionales.

- Discos espejo.
- Discos con reparto de datos cíclico y paridad nivel RAID 5.
- Duplicación de discos.

Las utilidades para hacer **copias de respaldo** están pensadas para los administradores de sistema, ya que permiten hacer copias masivas de datos a cintas magnéticas de forma muy sencilla. Es similar a las utilidades de *backup* existentes en UNIX pero con una interfaz más amigable. Permiten hacer copias totales o parciales de volúmenes o cuentas de usuario.

Para dotar con capacidades de recuperación a los sistemas de archivos de Windows NT, NTFS incluye **procesamiento transaccional** de las peticiones que afectan a los metadatos de los sistemas de archivos. Con esta facilidad, todas las operaciones de este estilo se almacenan en un registro d*og*) de forma que si el sistema falla se pueda volver a un estado coherente usando la información de dicho registro. Este archivo de registro se gestiona mediante un conjunto de rutinas internas al ejecutivo denominadas LFS d*og File Services*). Para asegurar la coherencia de los sistemas de archivos se usan técnicas de escritura cuidadosa con verificación en el registro y en los sistemas de archivos, de forma que las operaciones se reflejan primero en el registro y luego en el sistema de archivos. Ambas operaciones se hacen primero en la cache, por lo que el gestor de cache escribe inmediatamente los datos del registro a la zona del disco donde se guarda dicho registro.

Los **discos espejo** son una técnica de tolerancia a fallos muy popular y sencilla de implementar. pero costosa y poco eficiente. Consiste en tener dos volúmenes idénticos y actualizar los datos de forma cuidadosa en ambos. Una escritura no se valida hasta que no se ha hecho en los dos discos. Las lecturas, sin embargo, son válidas desde cualquiera de ellos. Las actualizaciones se mantienen coherentes en ambos discos de forma transparente al usuario.

La técnica más actual de tolerancia a fallos consiste en usar **dispositivos tipo RAID 5** a nivel software. Con esta técnica se usa un conjunto de discos para almacenar la información de los usuarios y la información de paridad del conjunto anterior como si fueran un único disco lógico. Los datos se reparten en grupos, cada uno de los cuales se escribe a un disco. Cuando se ha escrito un bloque en cada disco, se calcula su paridad y se almacena en el disco siguiente (*disk striping with parity*). Para que todos los datos de paridad no estén en el mismo disco, se almacenan de forma cíclica en discos distintos. Este mecanismo necesita un mínimo de tres discos y gestiona un máximo de 32 discos, que pueden estar en el mismo controlador o en controladores distintos.

PRESTACIONES 12.4

La escritura de datos con paridad conlleva el coste del cálculo de la misma, lo que puede ser oneroso en caso de escrituras que no llenan todos los discos (escrituras pequeñas). Por ello, este método se ajusta mejor a archivos con unidades de escritura grandes.

La **duplicación de discos** (Disk duplexing) es una técnica hardware que consiste en conectar un par de discos espejos, pero cada uno con su propio controlador. Esta técnica permite fallos software y hardware (discos y controlador). Para el sistema de archivos no hay diferencia entre discos espejos y duplicados, ya que la distinción se hace a muy bajo nivel.

Todos los servicios de tolerancia a fallos para almacenamiento se obtienen a través del manejador de disco tolerante a fallos *FtDisk.sys.* Este manejador está en un nivel más bajo que los manejadores de disco normales y proporciona abstracciones para discos espejo, duplicados o con paridad, así como procedimientos de recuperación dinámica del estado de los dispositivos.

12.10. PUNTOS A RECORDAR

- Windows NT es un sistema operativo multitarea, basado en un diseño de 32 bits, cuyas características principales son su diseño orientado a objetos, el subsistema de seguridad y los servicios de entrada/salida.
- Windows NT tiene un diseño moderno de tipo micronúcleo, con una pequeña capa de núcleo que da soporte a Las restantes funciones del ejecutivo del sistema operativo. Dentro del ejecutivo destaca la integración del modelo de seguridad (nivel C2), de la gestión de red, de la existencia de sistemas de archivos robustos y de la aplicación exhaustiva del modelo orientado a objetos.
- Las capas de Windows NT son la capa de abstracción de hardware (HAL), núcleo, el ejecutivo y los subsistemas de entorno.
- El núcleo es la base de Windows NT ya que planifica las actividades, denominadas *threads*, de los procesadores de la computadora. Al igual que en UNIX, el núcleo de Windows NT se ejecuta siempre en modo seguro (modo núcleo) y no usa la memoria virtual (no paginable).
- El núcleo proporciona las siguientes funciones al resto del sistema: modelos de objeto, ejecución ordenada de los threads, sincronización de la ejecución de distintos procesadores, gestión de excepciones hardware, gestión de interrupciones y *traps,* funcionalidad específica para manejar el hardware y mecanismos eficientes de comunicación y sincronización.
- La capa más compleja del sistema operativo es el ejecutivo, un conjunto de servicios comunes que pueden ser usados por todos los subsistemas de entorno de ejecución existentes en Windows NT a través de la capa de servicios del sistema.
- El ejecutivo tiene varios componentes: gestor de objetos, gestor de procesos, gestor de memoria virtual, monitor de seguridad, llamadas a procedimientos locales y gestor de entrada/salida.
- El gestor de objetos es el componente del ejecutivo de Windows NT que se encarga de proporcionar servicios para todos los aspectos de los objetos, incluyendo reglas para nombres y seguridad.
- El gestor de procesos se encarga de gestionar dos tipos de objetos básicos para el sistema operativo: procesos y threads.
- El gestor de memoria virtual proporciona un modelo de memoria con paginación por demanda y sin preasignación de espacio de intercambio. Cada proceso dispone de una capacidad de direccionamiento de 4 GB (32 bits), de los cuales 2 GB son para el programa de usuario y 2 GB se reservan para el sistema.
- El sistema de entrada/salida de Windows NT está construido como un conjunto de manejadores apilados, cada uno de los cuales está asociado a un objeto de entra da/salida (archivos, red, etc.).
- El gestor de la cache proporciona una cache única en el ámbito del sistema y común para todos los tipos de sistemas de archivos locales y remotos. Esta cache se gestiona sobre la base de bloques virtuales.
- Los subsistemas de entorno de ejecución proporcionan compatibilidad con entornos de ejecución provistos anteriormente por Microsoft (como MS-DOS o Windows 3.x) y con otras interfases de sistemas operativos (como POSIX u OS/2).
- Windows NT proporciona tres tipos de sistemas de archivos principales: FAT, HPFS y NTFS. Además de es tos sistemas de archivos, Windows NT proporciona otros para CD-ROM, para servidores, para comunicación entre procesos (como el NPFS, *Named Pipes File System*, y el MSFS, *Mailslot File System*), para registro (como el LFS, *Log File System)* y para sistemas distribuidos (como el DFS, *Distributed File System*).
- Los sistemas de archivos FAT16 son idénticos a los usados en MS-DOS y Windows 3.x.
- El sistema de archivos HPFS se diseñó para OS/2. Su principal característica es que divide el volumen en bloques de 8 MB que tienen su mapa de bits adyacente.
- NTFS es el sistema de archivos más moderno de Windows NT e incluye soluciones de diseño nuevas, proporcionando una combinación de rendimiento, fiabilidad y compatibilidad ausente en sistemas anteriores. Es el sistema de archivos asociado al subsistema de entorno de Win32.
- La estructura de un sistema de archivos de NTFS se describe en un registro de un archivo especial contenido al principio del volumen que se denomina **MFT** (*Master File Table*).
- La seguridad se incluyó como parte de las especificaciones de diseño de Windows NT, lo que permite lograr un nivel de seguridad C2 si se configura el sistema adecuadamente.
- El modelo de seguridad incluye componentes para controlar quién accede a los objetos, qué accesos pueden efectuar los usuarios sobre un objeto y qué eventos se auditan. Todos estos componentes juntos componen el subsistema de seguridad de Windows NT.
- Todos los objetos de Windows NT tienen asignados descriptores de seguridad como parte de sus fichas de acceso. La parle más significativa de los descriptores de segundad es la lista de control de accesos.
- Windows NT proporciona mecanismos de tolerancia a fallos tales como: utilidades para hacer copias de respaldo a cintas magnéticas, registros transaccionales, discos espejo, discos con reparto de datos cíclico y paridad nivel RAID 5 o duplicación de discos.

Digitalización Realizada con propósito académico. Sistemas Operativos

12.11. LECTURAS RECOMENDADAS

Los lectores interesados en saber más sobre Windows NT y Windows 2000 disponen actualmente de muy buena documentación. En esta relación de lecturas recomendadas se incluyen un conjunto seleccionado de libros que permitirán al lector interesado profundizar en este tema.

Para ampliar conocimientos sobre la estructura general del sistema operativo, se recomiendan los libros de Solomon [Solomon, 1998], Pearce [Pearce, 1977], Martínez [Martínez, 1999] y Custer [Custer, 1993].

Para profundizar más en aspectos de sistemas de entrada/salida y dispositivos, se recomiendan los libros de Baker [Baker, 1997] y Dekker [Dekker, 1999].

Para sistemas de archivos y directorios se puede acudir a los libros de Custer [Custer, 1995], Mitchell [Mitchell, 1977], Nagar [Nagar, 1977] y Lowe-Norris [Lowe-Norris, 2000].

Para aspectos concretos del sistema de seguridad se puede consultar el libro de Ivens [Ivens, 2000].

Digitalización Realizada con propósito académico. Sistemas Operativos

Digitalización Realizada con propósito académico. Sistemas Operativos

A

Comparación de los servicios POSIX y Win32

Tema	POSIX	Win32	Comentarios
Señales	pause		Suspende proceso hasta recepción de señal
Señales	kill		Manda una señal
Señales	sigempty, sigfillset, sigaddset, sigdelset, sigismember		Manipulación de conjuntos de señales
Señales	sigprocmask		Consulta o modificación de la máscara de señales
Señales	sigpending		Obtiene qué señales están pendientes de entregar
Señales	sigaction		Gestión detallada de señales
Señales	sigsetjmp, siglongjmp		Realizan saltos no locales
Señales	sigsuspend		Especifica máscara y suspende proceso hasta señal
Gestión de memoria	mmap	CreateFileMapping, MapviewOfFile	Proyecta en memoria un archivo. En Win32 requiere el uso de dos funciones (CreateFileMapping y MapViewOfFile)
Gestión de memoria	munmap	UnmapViewOfFile	Desproyecta un archivo
Memoria compartida	shmget	CreateFileMapping	Crea o asigna un segmento de memoria compartida
Memoria compartida	shmat	MapViewOfFile	Proyecta un segmento de memoria compartida
Memoria compartida	shmdt	UnmapViewOfFile	Desproyecta un segmento de memoria compartida

Tema	POSIX	Win32	Comentarios
Cerrojos de archivos y E/S	fentl (emd = F_SETLK, …)	LockFile,LockFileEx	Establece un cerrojo a un archivo
Cerrojos de archivos y E/S	fentl (emd = F_SETLK, …)	UnlockFile,UnlockFileEx	Elimina un cerrojo de un archivo
Procesos	fork y después exec	CreateProcess	Crea proceso (CreateProcess equivale a fork + exec
Procesos	_exit	ExitProcess	Termina el proceso
Procesos	getpid	GetCurrentProcess, GetCurrentProcessId	Obtiene identificador del proceso
Procesos	wait, waitpid	GetExitCodeProcess	Obtiene información de proceso ya terminado
Procesos	excel, execv, execle, execve, execlp, execvp		Ejecuta un programa (no hay equivalente en Win32)
Procesos	fork		Crea proceso duplicado (no hay equivalente en Win32)
Procesos	getppid		Obtiene ID del padre (en Win32 no hay relación padre/hijo)
Procesos	getgid, getegid		Obtiene ID del grupo (en Win32 no hay grupos de procesos)
Procesos	kill	TerminateProcess	Finaliza la ejecución de un proceso
Procesos	waitpid	WaitForMultipleObjects (manejadores de procesos)	Espera la terminación de un proceso (en Win32 de múltiples procesos)
Procesos	wait, waitpid	WaitForSingleObject (manejador de proceso)	Espera la terminación de un proceso
Comunicación	close	CloseHandle (manejador de tubería)	Cierra una tubería
Comunicación	mq_open	CreateFile (mailslot)	Abre una cola de mensajes en POSIX y un mailslot en Win32
Comunicación	mq_open	CreateMailslot	Crea una cola de mensajes en POSIX y un mailslot en Win32
Comunicación	mq_close	CloseHandle (manejador de mailslot)	Cierra una cola de mensajes en POSIX y un mailslot en Win32
Comunicación	mq_send	WriteFile (manejador de mailslot)	Envía datos a una cola de mensajes en POSIX y a un mailslot en Win32
Comunicación	mq_receive	ReadFile (manejador de mailslot)	Recibe datos de una cola de mensajes en POSIX y de un mailslot en Win32
Comunicación	mq_unlink	CloseHandle (manejador de mailslot)	Borra una cola de mensajes en POSIX y un mailslot en Win32 cuando deja de estar referenciado
Comunicación	mq_getattr	GetMailSlotInfo	Obtiene atributos de una cola de mensajes en POSIX y un mailslot en Win32
Comunicación	mq_setattr	SetMailSlotInfo	Fija los atributos de una cola de mensajes en POSIX y un mailslot en Win32

Tema	POSIX	Win32	Comentarios
Comunicación	mkfifo	CreateNamedPipe	Crea una tubería con nombre
Comunicación	pipe	CreatePipe	Crea una tubería sin nombre
Comunicación	dup or dup2 or fentl	DuplicateHandle	Duplica un manejador de archive
Comunicación	read (tubería)	ReadFile (tubería)	Lee datos de una tubería
Comunicación	write (tubería)	WriteFile (tubería)	Escribe datos en una tubería
Comunicación	close	CloseHandle	Cierra una tubería
Procesos ligeros	pthread_create	Create Thread	Crea un proceso ligero
Procesos ligeros	pthread_exit	ExitThread	Finaliza la ejecución de un proceso ligero
Procesos ligeros		GetCurrentThread	Devuelve el manejador del proceso ligero que ejecuta
Procesos ligeros		GetCurrenteThreadId	Devuelve el identificador del proceso ligero que ejecuta
Procesos ligeros	pthread_join	GetExitCodeThread	Obtiene el código de finalizaciónj de un proceso ligero
Procesos ligeros		ResumeThread	Pone en ejecución un proceso ligero suspendido
Procesos ligeros		SuspendThread	Suspende la ejecución de un proceso ligero
Procesos ligeros	pthread_join	WaitForSingleObject (manejador de proceso ligero)	Espera la terminación de un proceso ligero
Procesos ligeros		WaitForMultipleObject (manejador de proceso ligero)	Espera la terminación de múltiples procesos ligeros en Win32
Procesos ligeros		GetPriorityClass	Devuelve la clase de prioridad de un proceso
Procesos ligeros	sched_getparam	GetThreadPriority	Devuelve la prioridad de un proceso ligero
Procesos ligeros		SetPriorityClass	Fija la clase de prioridad de un proceso
Procesos ligeros	sched_setparam	SetThreadPriority	Fija la prioridad de un proceso ligero
Sincronización	pthread_cont_destroy	CloseHandle (manejador de evento)	Destruye una variable condicional en POSIX y un evento en Win32 cuando deja de estar referenciado
Sincronización	pthread_cond_init	CreateEvent	Inicia una variable condicional y un evento
Sincronización	pthread_cond_ broadcast	PulseEvent	Despierta a los procesos ligeros bloqueados en una variable condicional o un evento
Sincronización	pthread_cond_signal	SetEvent	Despierta a un proceso ligero bloqueado en una variable condicional o evento
Sincronización	pthread_cond_wait	WaitForSingleObject (manejador de evento)	Bloquea a un proceso en una variable condicional o evento
Semáforos binarios	pthread_mutex_ destroy	CloseHandle (manejador de mutex)	Destruye un mutex

Tema	POSIX	Win32	Comentarios
Semáforos binarios	pthread_mutex_init	CreateMutex	Inicia un mutex
Semáforos binarios	pthread_mutex_unlock	ReleaseMutex	Operación unlock sobre un mutex
Semáforos binarios	pthread_mutex_lock	WaitForSingleObject (manejador de mutex)	Operación lock sobre un mutex
Semáforos	sem_open	CreateSemaphore	Crea un semáforo con nombre
Semáforos	sem_init		Inicia un semáforo sin nombre
Semáforos	sem_open	OpenSemaphore	Abre un semáforo con nombre
Semáforos	sem_close	CloseHandle (manejador de semáforo)	Cierra un semáforo
Semáforos	sem_post	ReleaseSemaphore	Operación signal sobre un semáforo
Semáforos	sem_wait	WaitForSingleObject (manejador de semáforo)	Operación wait sobre un semáforo
Manejo de errores	errno	GetLastError	Almacena información sobre la última llamada al sistema
Tiempo	time	GetSystemTime	Obtiene el tiempo de calendario
Tiempo	localtime	GetLocalTime	Obtiene el tiempo de calendario en horario local
Tiempo	stime	SetSystemTime	Establece la hora y fecha
Tiempo	alarm	SetTimer	Establece un temporizador
Tiempo	times	GetProcessTimes	Obtiene los tiempos del proceso
Archivo y E/S	tcgetattr, tcsetattr	SetConsoleMode	Establece el modo de operación del terminal
Archivo y E/S	read, write	ReadConsole, WriteConsole	Lectura y escritura en el terminal
Archivo y E/S	close	CloseHandle	No está limitada a archivos
Archivo y E/S	open, creat	CreateFile	Crea o abre un archivo
Archivo y E/S	unlink	DeleteFile	Borra un archivo
Archivo y E/S	fsync	FlushFileBuffers	Vuelca la cache del archivo a disco
Archivo y E/S	stat, fstat	GetFileAttributes	Obtiene los atributos de un archivo
Archivo y E/S	stat, fstat	GetFileSize	Longitud del archivo en bytes
Archivo y E/S	stat, fstat	GetFileTime	Fechas relevantes para el archivo
Archivo y E/S	stat, fstat	GetFileType	Archivo o dispositivo de caracteres
Archivo y E/S	stdin, stdout, stderr	GetStdHandle	Devuelve un dispositivo de E/S estándar
Archivo y E/S	link, symlink		Win32 no proporciona enlaces
Archivo y E/S	readv		Lectura simple
Archivo y E/S	writev		Escritura simple
Archivo y E/S	read	ReadFile	Lee datos de un archivo
Archivo y E/S	truncate, ftruncate	SetEndOfFile	Fija la longitud de un archivo
Archivo y E/S	fentl	SetFileAttributes	Cambia los atributos de un archivo
Archivo y E/S	lseek	SetFilePointer	Devuelve el apuntador de posición del archivo

Tema	POSIX	Win32	Comentarios
Archivo y E/S	utime	SetFileTime	Modifica las fechas de un archivo
Archivo y E/S		SetStdHandle	Define un manejador de E/S estándar
Archivo y E/S	write	WriteFile	Escribe datos a un archivo
Archivo y E/S		CreateFileMapping	Define la proyección de un archivo en memoria
Archivo y E/S	mmap	MapViewOfFile	Proyecta un archivo en memoria
Archivo y E/S		OpenFileMapping	Abre un archivo proyectado en memoria
Archivo y E/S	munmap	UnmapViewOfFile	Elimina la proyección en memoria de un archivo
Directorios	mkdir	CreateDirectory	Crea un nuevo directorio
Directorios	closedir	FindClose	Cierra un directorio
Directorios	opendir, readdir	FindFirstFile	Busca una entrada en un directorio
Directorios	readdir	FindNextFile	Extrae la siguiente entrada de directorio
Directorios	getcwd	GetCurrentDirectory	Devuelve el nombre del directorio de trabajo
Directorios	rmdir, unlink	RemoveDirectory	Borra un directorio
Directorios	chdir, fchdir	SetCurrenteDirectory	Cambia el directorio de trabajo
Seguridad		DeleteAce	Borra una entrada de control de acceso de una ACL
Seguridad	stat, fstat, lstat	GetAce	Devuelve una entrada de control de acceso de una ACL
Seguridad	stat, fstat, lstat	GetAcInformation	Extrae la información de una ACL
Seguridad	stat, fstat, lstat, access	GetFileSecurity	Devuelve el descriptor de seguridad de un archivo
Seguridad	stat, fstat, lstat	GetSecurityDescriptor	Devuelve el descriptor de seguridad de un usuario
Seguridad	getlogin	GetUserName	Devuelve el nombre de sistema de un usuario
Seguridad		InitializeAcl	Inicia la información de una ACL
Seguridad	umask	IniatilzeSecurityDescriptor	Inicia el descriptor de seguridad de un usuario
Seguridad	getpwnam, getgrnam	LookupAccountName	Devuelve el nombre de sistema de una cuenta
Seguridad	getpwuid, getuid, geteuid	LookupAccountSid	Devuelve el identificador de sistema de una cuenta
Seguridad	setuid, seteuid, setreuid		Activan los distintos UID de un archivo en UNIX
Seguridad	setgid, setegid, setregid		Activan los distintos GID de un archivo en UNIX
Seguridad	getgroups, setgroups, initgroups	OpenProcessToken	Grupos suplementarios
Seguridad	chmod, fchmod	SetFileSecurity	Cambian permisos de archivo

Tema	POSIX	Win32	Comentarios
Seguridad		SetPrivateObjectSecurity	Cambian permisos de objetos privados
Seguridad	umask	SetSecurityDescriptorDacl	Cambian máscara de protección por defecto
Seguridad	chown, fchown, lchown	SetSecurityDescriptorGroup	Cambian el dueño de un archivo

B

Entorno de programación de sistemas operativos

La programación de aplicaciones sobre sistemas operativos supone conocer y usar las bibliotecas con las llamadas al sistema operativo. Para hacer una aplicación con llamadas al sistema operativo es necesario indicar en los programas los archivos con:

- La definición de prototipos y tipos de datos, por ejemplo <windows.h>, para que puedan compilarse los módulos del programa, así como el directorio, o directorios, en que se encuentran
- Los archivos de bibliotecas del sistema que deben enlazarse con la aplicación para crear un archivo ejecutable, incluyendo el camino donde localizar dichas bibliotecas.
- Las opciones de compilación que deben activarse para compilar y enlazar la aplicación.

Cada compilador proporciona un entorno de programación, más o menos integrado, para simplificar el proceso de construcción del software. Dentro de este entorno existe habitual mente un proyecto, o *makefile*, que almacena la información que específica cómo compilar y enlazar una aplicación.

En este apéndice se muestra brevemente el entorno de programación en el sistema operativo Windows y en UNIX/LINUX. Ambos incluyen herramientas para desarrollar programas, pero el nivel de integración de las mismas suele ser muy distinto. Los entornos que se estudian en este apéndice corresponden al compilador de *Visual C++,* de Microsoft, y al compilador gcc del lenguaje C para UNIX/LINUX. Como ejemplo de trabajo se utilizará una versión simplificada del programa Reloj, correspondiente a un trabajo práctico de Sistemas Operativos expuesto en el Apéndice C.

B.1. MAKEFILES DE UNIX

El archivo *makefile* describe, en UNIX y LINUX, cómo construir una aplicación, incluyendo las dependencias de bibliotecas y archivos. El programa make utiliza esta información para determinar qué archivos deben compilarse y enlazarse para producir una unidad ejecutable de la aplicación, de forma que sólo se compilen aquellos que han cambiado o que dependen de un archivo que ha cambiado.

B.1.1. Estructura de un archivo makefile

El Listado B.1 contiene el *makefile* del programa Reloj, que específica cómo construir el ejecutable `reloj`. Por convención, una especificación de *makefile* se almacena típicamente en un archivo llamado `Makefile` o `makefile`. Para ejecutar el programa make y construir una aplicación, sólo hay que teclear el mandato `make` dentro de un intérprete de mandatos (shell). El programa make busca en el directorio actual un archivo llamado `Makefile` o `makefile` y lo procesa.

Listado B.1. *Makefile de Reloj*.

```
#
# Las siguientes líneas especifican que los archivos de C
# tendrán una extensión c.

.SUFFIXES:
.SUFFIXES:.c $(SUFFIXES)

# Asignar a CC el nombre del compilador de C usado.

CC = gcc
# Asignar a DIR_APOYO la ruta del directorio que contiene
# los directorios del material de apoyo.
# Las bibliotecas del sistema se incluyen por defecto.

DIR_APOYO = ./apoyo

#
# La variable CFLAGS especifica dónde encontrar
# los archivos a incluir desde el material de apoyo.

CFLAGS = -I$ (DIR_APOYO)/include -g

# La siguiente regla le indica a make cómo procesar los
# archivos con extensión c. Normalmente esto no es necesario
# porque la extensión .c está definida para make.

.c.o:
        $(CC) $(CFLAGS -c $<

# La variable LDFLAGS especifica dónde encontrar
# los archivos de las bibliotecas. Las bibliotecas del
# sistema se incluyen por defecto.

LDFLAGS = L$ d){DIR_APOYO)/lib

#
# La variable LIBS especifico al compilador qué bibliotecas
# de archivos objeto se deben usar para construir la aplicación, además
# de las del sistema.
#
```

```
LIBS = -lapoyo

#
# La variable OBJS especifica al compilador qué archivos
# objeto se deben crear para construir la aplicación

OBJS = reloj.o clock_task.o hardware.o

# La siguiente regla especifica cómo construir
# el ejecutable del programa, así como las dependencias
# de archivos #include (.h)
#

reloj: $(OBJS)
        $(CC)  $(OBJS)  $[LDFLAGS)  $[LIBS) -o reloj

reloj.o: reloj.h
clock_task.o: reloj.h
hardware.o: reloj.h

# Regla clean. La ejecución de 'make clean' borra todos los archivos
# objeto y el ejecutable
 clean:
 rm -f *.o reloj
```

Para compilar en UNIX O LINUX hay que ir al directorio donde se encuentra el *makefile* de la aplicación y ejecutar el mandato make, como se muestra en la Figura 13.1.

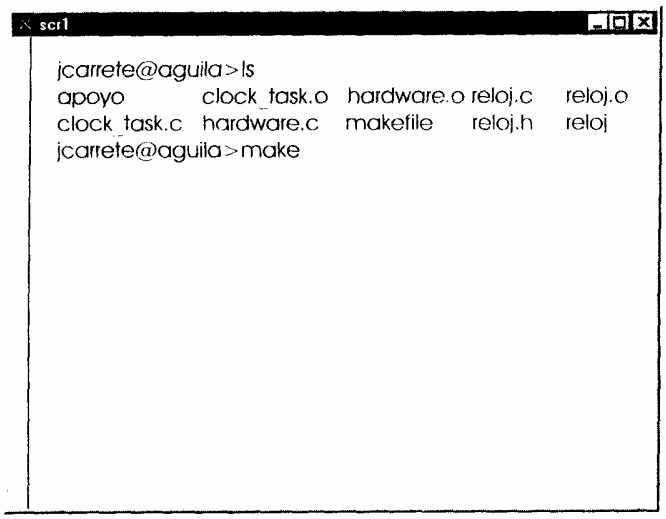


Figura B.1. *Compilando con make en UNIX*.

En un *makefile*, un comentario comienza con el carácter #. La especificación de un, makefile puede ser muy oscura, y difícil de mantener, si no se comenta adecuadamente.

Las líneas siguientes del, *makefile* especifican al programa make las extensiones de los archivos que se van a compilar. En este caso se indica que los archivos de C tienen la extensión .c.

- SUFFIXES:
- SUFFIXES: .c $(SUFFIXES)

Algunos lenguajes, por ejemplo C++, esperan archivos que tengan la extensión .cpp o .cc. Si esto es lo que sucede, se debería cambiar el .c de la segunda línea a .cpp o .cc, o añadir dichos sufijos a la línea de definición. Todos los archivos de C proporcionados en este libro tienen la extensión .c.

La siguiente línea define el compilador que se debe usar para compilar los programas:

```
CC = gcc
```

En este ejemplo se asigna a la variable CC el nombre del mandato que corresponde con el compilador de C que traduce el código fuente. En este fragmento se le da el valor gcc que es el compilador de GNU para el lenguaje C en UNIX y LINUX.

La línea:

```
DIR APOYO = ./apoyo
```

asigna la ruta que se corresponde con los directorios de inclusión y de la biblioteca del material de apoyo para la construcción del programa reloj. Es siempre mejor usar variables relativas a la situación del programa a construir, porque así se puede instalar la aplicación en distintos directorios cada vez sin que haya que modificar el archivo *makefile*.

La línea:

```
CFLAGS= -I $(DIR_APOYO)
```

establece una variable que informa al compilador de C dónde debe buscar los archivos de inclusión del material de apoyo del usuario y los del sistema. Además de buscar en este directorio, el compilador siempre busca en los directorios del sistema, que suelen ser /usr/include y sus subdirectorios.

De manera similar la asignación siguiente:

```
LDFLAGS= - L $ (DIR APOYO)/lib
```

define una variable que informa al cargador de UNIX, ld, dónde debe buscar los archivos de biblioteca. Además de buscar en este directorio, el compilador siempre busca en los directorios del sistema, que suelen ser /usr/lib y sus subdirectorios.

La variable LIBS especifica al compilador qué bibliotecas de archivos objeto se deben usar para construir la aplicación.

```
LIBS= -lapoyo
```

Las líneas siguientes:

```
.c.o:
        $(CC) $ (CFLAGS) –c $<
```

son una regla de dependencia implícita que especifica que un tipo de archivo se construye a partir de otro y describe cómo rechazar su construcción. En este ejemplo, las líneas anteriores especifican que los archivos .o se construyen a partir de los archivos .c mediante el compilador de C.

La asignación siguiente:

```
OBJS=reloj.o clock_task.o hardware.o
```

da valor a la variable OBJS con los nombres de los archivos objeto que forman la aplicación. La aplicación reloj tiene tres módulos objeto de aplicación: reloj.o, clock_task.o y hardware.o. Para una aplicación diferente habría que modificar esta línea para asignar a OBJS los módulos objeto de dicha aplicación.

El siguiente conjunto de líneas es el corazón del ,makefile. Estas líneas son reglas de dependencia que especifican Cómo construir la aplicación. Por ejemplo, las siguientes líneas:

```
reloj: $(OBJS)
        $(CC) $(OBJS) $(LDFLAGS) $(LIBS) -o reloj
```

del ,*makefile* de Reloj especifican que reloj depende de OBJS, que tiene el valor reloj.o, clock_task.o y hardware.o. Además depende de LIBS, que tiene el valor libapoyo.a. Si cualquiera de estos archivos objeto es más reciente que reloj, make ejecuta la línea de mandato que produce una nueva versión de reloj más reciente que los archivos objeto. Si lodos los archivos objeto son más antiguos que reloj, significa que reloj está actualizado y, por lo tanto, no se necesita realizar ninguna acción.

Después de que se hayan construido todos los archivos objeto necesarios, su fecha de actualización será más reciente que la de reloj, por lo que se ejecuta el mandato que construye reloj.

La lista completa de opciones del programa make se puede obtener mediante el siguiente mandato:

```
man make
```

Para **ejecutar** una aplicación en UNIX basta con teclear el nombre de la aplicación en el *prompt* del intérprete de mandatos (Fig. B.2).

B.1.2. Gestor de bibliotecas

Existe en UNIX y LINUX una utilidad para la creación y mantenimiento de bibliotecas de archivos. Su principal uso es crear bibliotecas de objetos, es decir, agrupar un conjunto de objetos relacionados dentro una entidad lógica que se puede usar como un elemento de compilación.

En la línea de compilación se especifica la biblioteca (por convención libnombre.a) en vez de los objetos que hay dentro de ella. El enlazador extraerá de la biblioteca los objetos que contienen las variables y funciones requeridas y los insertará dentro del programa ejecutable o incluirá referencias dinámicas a dichos objetos.

Formato del mandato:

```
ar opciones biblioteca archivos…
```

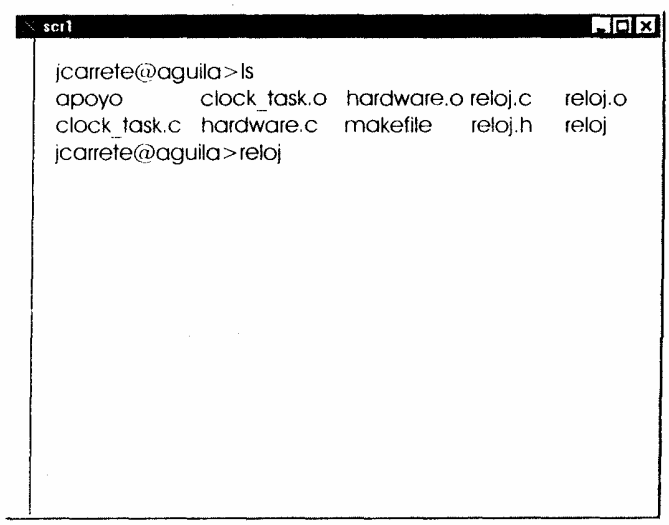


Figura B.2. *Ejecución de reloj en UNIX.*

Algunas opciones de este mandato son:

- -d. Elimina de la biblioteca los archivos especificados.
- -r. Añade (o reemplaza si ya existe) a la biblioteca los archivos especificados. Si no existe la biblioteca, se crea.
- -ru. Igual que -r pero sólo reemplaza si el archivo es más nuevo.
- -t. Muestra una lista de los archivos contenidos en la biblioteca.
  -v. *Verbose*.
- -x. Extrae de la biblioteca los archivos especificados.

A continuación, se muestran algunos ejemplos de aplicación de este mandato:

- Obtención de la lista de objetos contenidos en la biblioteca estándar de C.

```
ar -tv /usr/lib/libc.a
```

- Creación de una biblioteca con objetos que manejan distintas estructuras de datos.

```
ar -rv $HOME/lib/libest.a pila.o lista.o
ar –rv $HOME/lib/libest.a arbol.o hash.o
```

- Creación de la biblioteca con material de apoyo que incluye el objeto del programa que simula el dispositivo reloj.

```
ar -rv $HOME/apoyo/libapoyo.a dispositivo.o
```

Hay dos formas posibles de compilar un programa que use una biblioteca: con nombre absoluto y con nombre relativo. En este último caso es necesario tener una variable de entorno para indicar dónde está la biblioteca, como se ha hecho en el makefile *anterior*. A continuación, se muestran ejemplos de uso de ambas formas:

```
gcc -o reloj reloj.c clock_task.c hardware.c
        $HOME/apoyo/libapoyo.a
gcc –o reloj reloj.c clock_task.c hardware.c
        -L$HOME/apoyo -lapoyo
```

Observe que la forma de especificar la biblioteca es distinta en ambos casos. Cuando se especifica el nombre absoluto, se indica el nombre completo del archivo donde está la biblioteca. Cuando se especifica un nombre relativo, sólo se indica una porción de dicho nombre: se asume una extensión válida y se elimina el prefijo lib.

B.1.3. Depuración de una aplicación en UNIX o LINUX

En todas las versiones del sistema operativo UNIX, incluyendo LINUX, existe un programa de depuración de programas. En el caso de LINUX existe un depurador denominado gdb. El depurador permite que el usuario pueda controlar la ejecución de un programa y observar su comporta miento interno mientras ejecuta. Estos programas son muy útiles cuando se programa o prueban aplicaciones, como las prácticas de alumnos. Su uso puede ahorrar mucho tiempo de desarrollo y facilitar la tarea de los programadores.

Para poder depurar un programa, el compilador debe incluir información especial dentro del mismo. Por ello, para poder depurar un programa compilado con el gcc, se debe compilar con la opción -g.

A continuación se describen algunas de las Funciones de un depurador genérico:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*).
- Examinar el valor de variables.
- Ejecutar el programa línea a línea.

El formato del mandato para ejecutar el depurador en LINUX es:

```
gdb programa_ejecutable
```

En el caso del *makefile* del ejemplo se ha creado un ejecutable denominado reloj. Para depurarlo habría que ejecutar el mandato:

```
gdb reloj
```

Algunos mandatos internos del gdb son:

- run. Arranca la ejecución del programa.
- break. Establece un *breakpoint* (un número de línea o el nombre de una Función).
- list. Imprime las líneas de código especificadas.
- print. Imprime el valor de una variable.
- continue. Continúa la ejecución del programa después de un *breakpoint*.
- next. Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa.
- stop. Ejecuta la siguiente línea. Si se trata de una llamada a función. ejecuta sólo la llamada y se para al principio de la misma.
- quit. Termina la ejecución del depurador.

B.2. ENTORNO DE PROGRAMACIÓN DEL VISUAL C++ DE MICROSOFT

Este compilador proporciona un entorno de programación y desarrollo totalmente integrado. A través del mismo se puede acceder a todas las utilidades necesarias para escribir, compilar y probar un programa. En la Figura B.3 se muestra la ventana principal del compilador Visual C++ de Microsoft. En la parte superior de la ventana hay un menú con mandatos tales como File, Edit, View y Help. Debajo de los elementos del menó hay una barra de herramientas con varios botones, que proporcionan acceso rápido a muchos de los mandatos usados frecuentemente: open, close, help, etc. Se puede obtener ayuda de cualquiera de las características del entorno, ejecutando el mandato Help.

B.2.1. Creación de un espacio de trabajo

El primer paso para crear un archivo de proyecto para una aplicación de Sistemas Operativos es crear un espacio de trabajo y un proyecto, mediante el menú: File ->New. Dentro de la caja de diálogo New debe especificarse el tipo de aplicación que se está construyendo. La elección correcta depende de que la aplicación requiera una consola. Cualquier aplicación de Sistemas Operativos c acepte entrada de teclado del usuario mediante el objeto de iostream cin o escriba en la pantalla mediante el objeto de iostream cout requiere una consola. Para este tipo de aplicación de Sistemas Operativos la selección apropiada es Win32 Console Application. Si la aplicación no utiliza la biblioteca iostream, la selección apropiada es win32 Application.

El paso final es especificar la posición del proyecto. En el ejemplo se quiere que el proyecto resida en c:\jesus\docencia\sos2\apendice2\reloj, por lo que habrá que navegar hasta c:\jesus\docencia\sos2\apendice2\reloj y teclear reloj en el campo Project name:. Para crear el proyecto se debe pulsar el botón OK.

Para compilar el archivo reloj.c sólo hay que indicarlo en la opción Build, que se muestra en la Figura B.4.

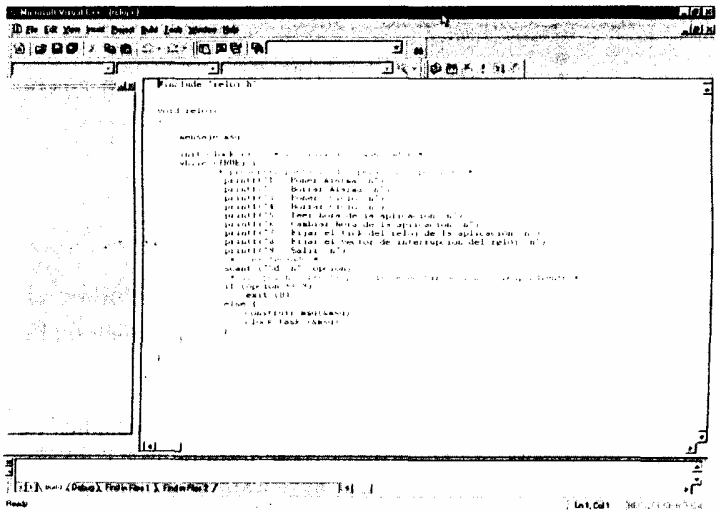


Figura B.3. *Ventana principal de Visual C++ de Microsoft*

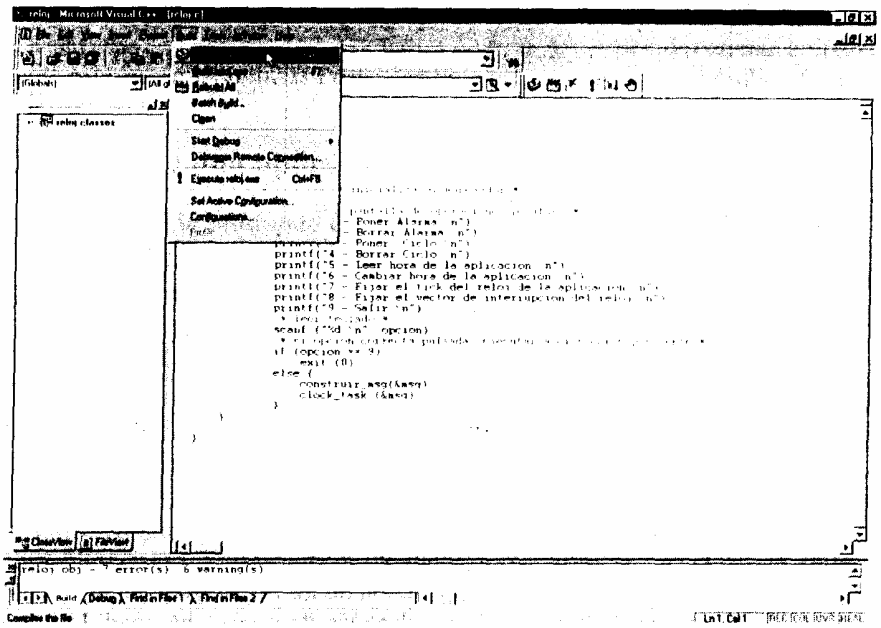


Figura B.4. *Compilación de reloj en Visual C++ de Microsoft*

El compilador de Visual C++ de Microsoft almacena la información de cómo construir una aplicación en un archivo de proyecto. Los archivos de proyecto tienen la extensión .dsp. En el caso del ejemplo habría un nuevo archivo denominado reloj.dsp. Para salvar un archivo de proyecto se usa el menú: File—>Sabe Works pace.

Después de crear el archivo de proyecto, inicialmente, el proyecto está vacío y habrá que añadirle los archivos fuente del proyecto. El programa de ejemplo, Reloj, consta de varios módulos fuente: reloj.c, clock_task.c y hardware.c. Además, como material de apoyo a los alumnos se proporciona una biblioteca y un archivo de definición de estructuras de datos (*.h). La Figura B.5 muestra un diagrama simplificado de la estructura de módulos de Reloj.

Para activar la caja de diálogo para añadir archivos al proyecto se ejecuta el mandato: Project->Ad Lo Project->Files. Este paso activa la caja de diálogo que se muestra en la Figura B.6.

Para poder compilar y enlazar el proyecto es necesario definir correctamente las opciones del proyecto para poder encontrar los archivos a incluir y las bibliotecas que se van a usar. Esto se

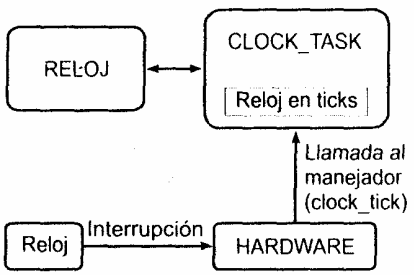


Figura B.5. *Estructura de módulos del programa reloj*

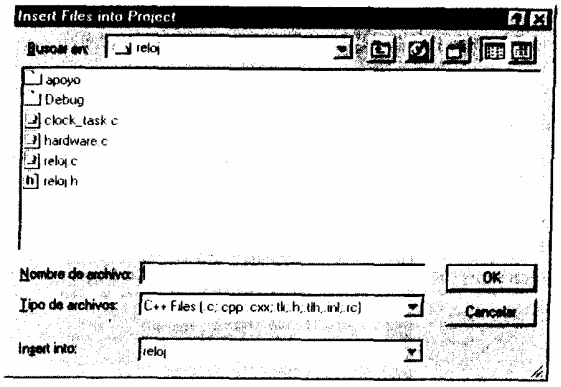


Figura B.6. *Caja de diálogo «Insert Files into Project» del Visual C++ de Microsoft*

consigue mediante el mandato: Tools ->Options. Para ello se ejecuta el menú Show directories for-> Include files, y se teclea la ruta en la ventana Directories. En el ejemplo del reloj, esta ruta es c:\jesus\docencia\sos2\apendice2\reloj.apoyo. El valor por defecto del campo Directorias indica la posición de los archivos de inclusión del sistema. La Figura B.7 muestra la caja de di Options después de que se ha añadido la entrada del directorio de inclusión del programa reloj. De igual manera se puede definir el directorio para la biblioteca de apoyo, pero usando el menú Show directories for -> Libraries.

Para compilar y enlazar la aplicación se usa el menú: Build -> Build reloj.exe. Este

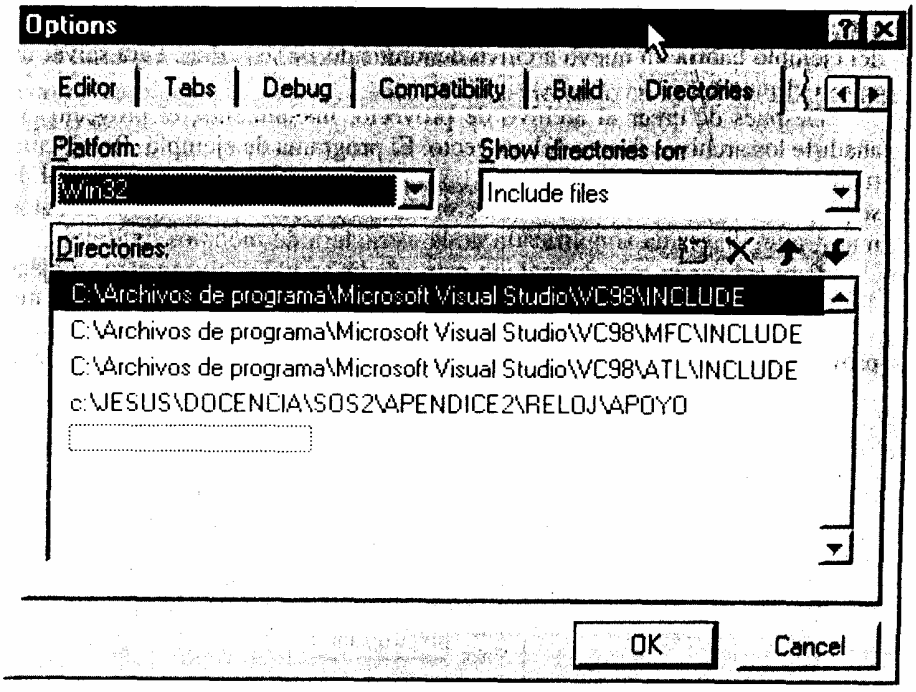


Figura B.7. *Caja de diálogo «Options» del Visual C++ de Microsoft*

mandato hace que se compilen todos los módulos que han cambiado desde la última compilación y que después se enlacen todos los archivos objeto y las bibliotecas pedidas en una unidad ejecutable. El ejecutable se escribe en el archivo reloj .exe puesto que ése es el nombre del proyecto.

8.2.2. Ejecución de una aplicación en Visual C++

Para ejecutar una aplicación utilizando el C++ de Microsoft existen dos opciones:

- Ejecución desde el entorno de Visual C++.
- Ejecución desde una consola de MS-DOS.

Para ejecutar desde el entorno de Visual C++, lo único que hay que hacer es ejecutar la opción: Build ->Execute reloj.exe Este mandato solicita al entorno que ejecute el archivo reloj.exe usando para ello todos los recursos que sean necesarios. Si se usa entrada/salida por consola, se abre una ventana consola. La Figura B.8 muestra la ventana Build y la opción a usar para ejecutar el archivo reloj .exe.

La otra opción es ejecutar el programa desde una ventana consola o de MS-DOS, que se puede crear activando el icono de MS-DOS en el menú Inicio. Una vez en la ventana consola, se debe cambiar el directorio a la posición del ejecutable de la aplicación y a continuación ejecutar la aplicación tecleando su nombre.

B.2.3. Depuración de una aplicación en Visual C++

Depurar una aplicación utilizando el C++ de Microsoft, usando el entorno de Visual C++, es real mente fácil. Hay dos opciones básicas:

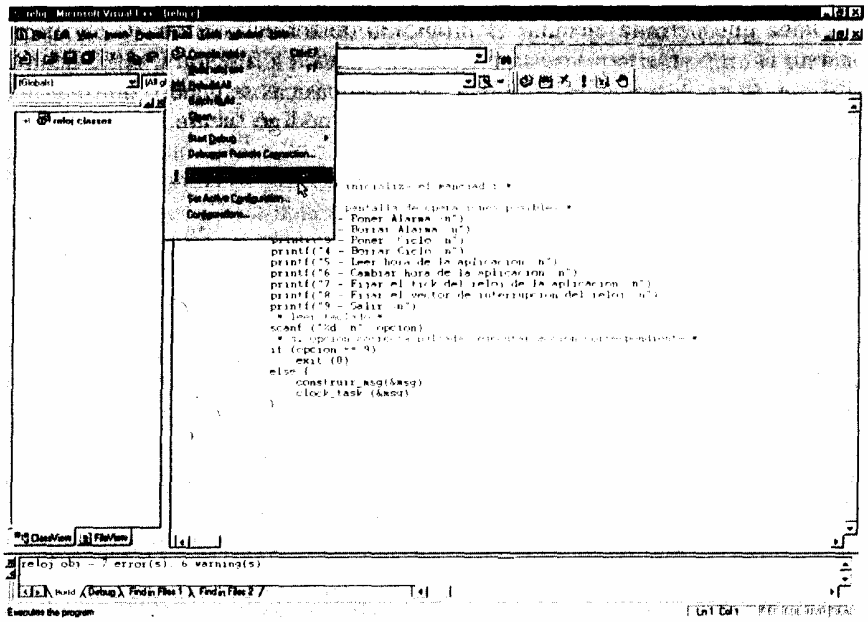


Figura B.8. *Ejecución de relo.exe en el Visual C++ de Microsoft*

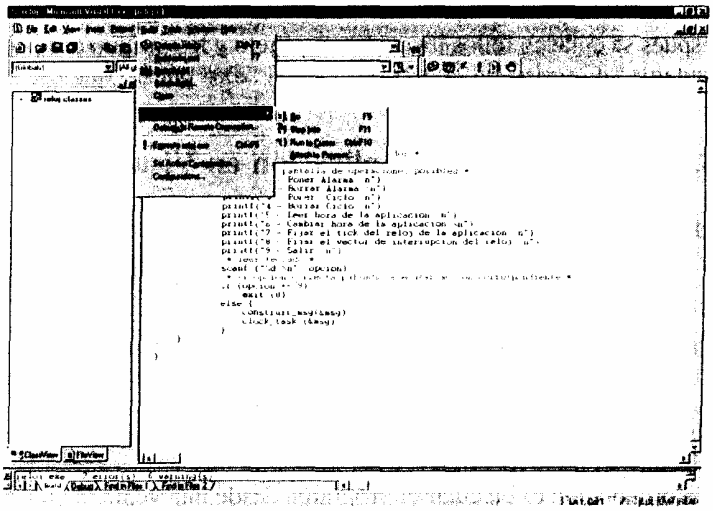


Figura B.9. *Depuración de reloj.exe en el Visual C++ de Microsoft*

- Depurar una vez enlazada la aplicación.
- Enlazar y depurar al mismo tiempo.

La primera es más aconsejable, porque en este caso se puede estar seguro de que no hay errores de en lazado.
Para ejecutar desde el entorno de Visual C++, lo único que hay que hacer es ejecutar la opción: Build -> Start Debug. Este mandato solicita al entorno que ejecute el archivo reloj .exe de forma controlada por el depurador (Fig. B.9). Cuando se ejecuta esta opción, aparece una nueva ventana que permite ejecutar el programa de forma continua (Go), ejecutar hasta donde está el cursor (Run to cursor) o pararse en una llamada a función y saltar dentro del código fuente de la misma (Step into). Con estas tres llamadas básicas se puede controlar la ejecución del programa.- Los errores salen en la pantalla de debajo del código fuente, mientras que el flujo de ejecución se controla en esta ventana.

C

Trabajos prácticos de sistemas operativos

Las prácticas que se presentan en este apéndice fueron diseñadas como trabajos de laboratorio para estudiantes de las asignaturas de Sistemas Operativos de la Universidad Politécnica de Madrid y de la Universidad Carlos III de Madrid. Todas las prácticas se han desarrollado en uno u otro entorno y fueron pensadas para ellos, por lo que es posible que existan todavía en los enunciados dependencias o referencias cercanas o derivadas de dichos en tornos. No obstante, se ha hecho un importante esfuerzo para generalizar dichos enunciados. De forma que puedan desarrollarse fácilmente sobre sistemas operativos de amplia difusión como Linux, UNIX o Windows NT. De cualquier forma, es importante que los profesores que pretendan usar estos trabajos de laboratorio revisen cuidadosamente cada uno de ellos y lleven a cabo una etapa previa de adaptación a sus entornos de prácticas. Esto evitará errores y confusión a los alumnos.

En casi todos los trabajos prácticos expuestos se hace referencia al material de apoyo existente para las prácticas. Este material se puede conseguir en las páginas WWW del libro, cuya referencia es: http://datsi.fi.upm.es/~ssoo-va y http://arcos.inf.uc3m.es/~ssoo-va

Los trabajos prácticos que se proponen en este apéndice se describen brevemente a continuación:

1. **Programación de scripts.** El objetivo de esta práctica es familiarizarse con el intérprete de mandatos de UNIX que. Además de ser la interfaz de usuario del sistema. Ofrece una poderosa herramienta de programación que permite a los usuarios, tanto ordinarios como administradores, construir nuevos mandatos (*shell scripts*) sin tener que utilizar directamente los servicios proporcionados por el sistema operativo.
2. **Llamadas al sistema para la gestión de procesos**. Esta práctica permite al alumno familiarizarse con los servicios para la gestión de procesos que proporciona POSIX. Asimismo, se pretende que conozca cómo es el funcionamiento interno de un intérprete de mandatos en UNIX/Linux. El alumno deberá diseñar y codificar, en lenguaje C y sobre sistema operativo UNIX/Linux, un programa que actúe como intérprete de mandatos.

3. **Llamadas al sistema para la gestión de archivos**. Esta práctica permite al alumno familiarizarse con los servicios para la gestión de archivos que proporciona POSIX. El alumno deberá diseñar y codificar, en lenguaje C y sobre un sistema operativo UNIX/Linux, dos programas: uno que permita copiar un archivo sobre otro y otro que permita lisiar el contenido de un directorio.
4. **Gestión de interrupciones de un reloj.** En esta práctica se pretende que el alumno se familiarice con los mecanismos de gestión de interrupciones. Para ello se propone que el alumno desarrolle un manejador muy simplificado de un reloj, que se simula con un pseudodispositivo disponible en el material de apoyo.
5. **Comunicación de procesos con sockets**. En esta práctica se pretende que el alumno se familiarice con los mecanismos de comunicación entre procesos, usando sockets como mecanismo de comunicación. Además, el alumno tendrá ocasión de experimentar con un sistema cliente-servidor.
6. **Diseño de un sistema multiprogramado**. El objetivo de esta práctica es llegar a conocer los principales conceptos relacionados con la gestión de procesos y la multiprogramación. El trabajo va a Consistir básicamente en convertir un sistema con monoprogramación en uno multiprogramado.
7. **Diseño de un sistema de archivos con bandas**. Esta práctica tiene tres objetivos principales: que el alumno entienda que se puede optimizar el almacenamiento en archivos haciendo uso de la concurrencia y del paralelismo, enseñar al alumno a diseñar las estructuras que representan un archivo y familiarizar al alumno con los mecanismos de programación de sistemas de archivos.

Los trabajos prácticos descritos se pueden dividir en dos grupos:

- Prácticas generalistas de sistemas operativos. Prácticas 1 a la 5. Son adecuadas para cualquier curso de Sistemas Operativos. Para desarrollarlas es necesario haber seguido en clase de teoría las secciones del libro que, en cada capítulo, muestran los aspectos de introducción a los Sistemas Operativos.
- Prácticas de diseño de sistemas operativos. Prácticas 6 y 7. Estas prácticas se han pensado específicamente para un curso de Diseño de Sistemas Operativos. Para desarrollarlas es necesario haber seguido en clase de teoría las secciones del libro que, en cada capítulo, muestran los aspectos de diseño de Sistemas Operativos.

En cuanto al lenguaje de programación a utilizar para llevar a cabo los trabajos prácticos, aunque se recomiendan los lenguajes C y C++, () es estrictamente necesario, ya que las prácticas se pueden desarrollar en cualquier otro lenguaje. Es importante tener en cuenta que los materiales de apoyo sólo están disponibles en lenguaje C.

PRÁCTICA C.1. PROGRAMACIÓN DE SCRIPTS

C.1.1. Objetivo de la práctica

El objetivo de esta práctica es familiarizarse con el intérprete de mandatos de UNIX que, además de ser la interfaz de usuario del sistema, ofrece una poderosa herramienta de programación que permite a los usuarios, tanto ordinarios como administradores, construir nuevos mandatos (shell-scripts) sin tener que utilizar directamente los servicios proporcionados por el sistema operativo.

Un aspecto que conviene resaltar desde el principio es la dificultad de programación en este entorno. La sintaxis y el modelo de programación son bastante inusuales, lo que hace un poco

difícil escribir los primeros programas. Sin embargo. El esfuerzo merece la pena puesto que permite llegar a conocer mejor cómo funciona el sistema.

C.1.2. Descripción de la funcionalidad a desarrollar por el alumno

La práctica consiste en desarrollar un *script* denominado dirmix que lleve a cabo la unión del contenido de un conjunto de directorios. El script recibirá como argumentos:

- Un criterio de selección, cuyo objetivo se explicará un poco más adelante.
- Un directorio destino.
- Un conjunto de directorios origen.

Después de la ejecución de dirmix. En el directorio destino habrá una copia de todos los archivos (sólo de los archivos, no de los directorios) de los directorios origen especificados, teniendo en cuenta también los contenidos inicialmente existentes en el propio directorio destino.
Cada copia debe preservar las características del archivo original, tales como su propietario o las fechas asociadas al mismo (mandato cp —p). En el caso de que exista un archivo con el mismo nombre en dos o más de los directorios (incluyendo también el directorio destino), se aplicará el criterio (le selección, que se recibe como primer argumento, para determinar cuál (le ellos se copia al directorio destino. De esta forma, el criterio de selección no está incluido en el script, sino que se trata de un programa que éste recibe como argumento.
Como parte de la práctica se deberá también construir un script. Denominado más nuevo. Que realice la selección teniendo en cuenta la fecha de última modificación (le los archivos.

Descripción de masnuevo

Este script recibirá como argumentos los archivos a los archivos se pretende aplicar el criterio de comparación y escribirá por su salida estándar cuál ha sido el archivo elegido al aplicar el criterio de selección correspondiente. El criterio consistirá en seleccionar el archivo cuya fecha de última modificación sea más reciente (una posible forma (le hacerlo es usando alguna (le las opciones del mandato ls).

Descripción de dirmix

El mandato dirmix se usará de la siguiente forma:

dirmix criterio dir_dest $\text{dir_org}_1 \ldots \text{dir_org}_n$

Como se explicó antes, después de la ejecución (le este mandato, en el directorio dir_dest deberá quedar tina copia de todos los archivos que había inicialmente en cada dir_org junto con los que había en dir_dest habiéndose aplicado el mandato criterio para determinar cuál debe copiarse en caso de coincidencia de nombres.
El script dirmix deberá:

- Comprobar que el número de argumentos recibidos es mayor o igual que 3 y que el segundo es un directorio. Si no se cumple alguna de estas dos condiciones, el programa deberá terminar inmediatamente imprimiendo un mensaje (le error por la salida de error estándar y devolviendo un valor igual a 1

- Si durante el procesado de un argumento correspondiente a un directorio origen se detecta que no es un directorio, se mostrará un mensaje de error por la salida de error estándar y se continuara con el procesado de los siguientes. Al final el programa deberá devolver un valor igual a 2.
- En el caso de que a la hora de copiar un archivo exista un directorio con el mismo nombre en el directorio destino, no se realizará la operación, se imprimirá un mensaje de error y se continuará tratando el siguiente archivo. En este caso, el programa no devolverá al final ningún valor de error (o sea, devolverá un O).
- Cuando se precise aplicar el criterio de selección para determinar qué archivo se debe copiar, se invocará al programa recibido como primer argumento pasándole como argumentos los archivos que se quieren comparar. El programa escribirá por su salida estándar el nombre del archivo elegido.

Hay muchas maneras de estructurar este .script. El esquema más intuitivo, aunque no más eficiente, sería ir procesando cada directorio origen realizando la copia de los archivos que cumplan las condiciones. Este esquema puede implicar copiar un archivo que luego es sobrescrito por otro de un directorio posterior.
Una versión más eficiente que elimina la necesidad de realizar estas operaciones de copia innecesarias consistiría en. Luego el script sólo copiara un archivo al directorio destino si está seguro que es el que debe quedar allí al final del proceso. Observe que esta versión sería bastante más complicada de programar que la anterior.

C.1.3. Recomendaciones generales al alumno

- Desarrolle primero el script más nuevo.
- Continúe introduciendo la funcionalidad de dirmix en el orden en que se describe en la sección anterior.
- Nunca ejecute mandatos complejos para probar la primera vez. Pruebe primero copiando de un único directorio con criterios bien definidos.

C.1.4. Documentación a entregar

Se recomienda que el alumno entregue los siguientes archivos:

- autores: archivo con los datos de los autores.
- memoria.txt: memoria de la práctica.
- masnuevo y dirmix: códigos fuentes de los scripts con toda la funcionalidad que requieren.

C.1.5. Bibliografía

S. R. Bourne. The UNIX System. Addison-Wesley, 1983
K. Havilland, D. Cray y 13. Salama. UNIX System / Addison—Wesley. 1999.
13. Kernighan y R. Pike. The UNIX Programing Enviromen 2 edición. Prentice-Hall, 1 994
M. J. Rocking. Advanced UNIX Programming Prentice—Hall, 1 9
SUN Microsystems. /', Programming Utilizes and Libreries. Sum Microsystems Microsystems, 1990.

PRÁCTICA C.2. LLAMADAS AL SISTEMA PARA LA GESTIÓN DE PROCESOS

C.2. 1. Objetivo de práctica

Esta práctica permite al alumno fan con los servicios para la gestión de procesos que proporciona POSIX. Asimismo, se pretende que conozca cómo es el funcionamiento interno del un intérprete de mandatos en UNIX/Linux.

Esta práctica cubre los objetivos de programación de llamadas al sistema y del capítulo de procesos.

El alumno deberá diseñar y codificar, en lenguaje C y sobre sistema operativo UNIX/Linux, un programa que actúe como intérprete de mandatos. El programa deberá seguir estrictamente las especificaciones y requisitos contenidos en este documento.

Con la realización de este programa el alumno adquirirá valiosos conocimientos de programación en entorno POSIX. Tanto en el uso de las llamadas al sistema operativo (FORK, EXEC, SIGNAL, PIPE, DUP, etc.), como en el manejo de herramientas como el visualizador de páginas de manual man, el compilador de C gcc. el regenerador de programas make, etc.

Nota: Durante la lectura de este documento encontrará la notación “man -s# xxxxx”. Que sugiere usar el mandato man de UNIX/Linux para obtener información sobre la orden xxxxx de la sección #. Por favor, siga las recomendaciones.

C.2.2. Descripción de la funcionalidad a desarrollar por el alumno

El intérprete de mandatos a desarrollar o, minishell utiliza la entrada estándar (descriptor de archivo ()), para leer las líneas de mandatos que interpreta y ejecuta. Utiliza la salida estándar (descriptor de archivo 1) para presentar el resultado de las órdenes internas. Y utiliza la salida de error estándar (descriptor de archivo 2) para mostrar las variables especiales promp y bqpid (véase el epígrafe Variables especiales), así como para notificar los errores que puedan dar. Si ocurre un error en alguna llamada al sistema, se utiliza para notificarlo la función de librería perror.

Para el desarrollo de esta práctica se recomienda proporcionar al alumno un *parser* que le permita leer los mandatos introducidos por el usuario. Así, el alumno sólo deberá preocuparse de implementar el intérprete de mandatos.

La sintaxis que se propone para este p es similar a la sintaxis que comprende el parser del intérprete de mandatos del sistema operativo UNIX:

- Blanco: es un carácter tabulador o espacio.
- Separador: es un carácter con significado especial (<>&). El fin de línea o el fin de archivo (por teclado CTRL-D).
- Texto: es cualquier secuencia de caracteres delimitada por blanco o separador.
- Mandato: es una secuencia de textos separados por blancos. El primer texto especifica el nombre del mandato a ejecutar. Las restantes son los argumentos del mandato invocado. El nombre del mandato se pasa como argumento 0 (man execvp) Cada mandato se ejecuta como un proceso hijo directo del mininisheil (man Cork). El valor de un mandato es su estado de determinación (man —s2 wait). Si la ejecución falla. Se nodifica el error (por el estándar error).
- Secuencia: es una secuencia de dos o más mandatos separados por ‘1 ‘La salida estándar de cada mandato se conecta por una tubería (man pipe) a la entrada estándar del siguiente. El, minishell normalmente espera la terminación del último mandato de la secuencia antes de solicitar la siguiente línea de entrada. El valor de una secuencia es el valor del último man dato de la misma.

- Redirección: la entrada o la salida de un mandato o secuencia puede ser redirigida añadiendo tras él la siguiente nutación. En caso de cualquier error durante las redirecciones, se notifica (por la salida de error estándar) y se suspende la ejecución de la línea.
  — < archivo: usa archivo como entrada estándar abriéndolo para lectura (man open).
  — > archivo: usa archivo como salida estándar. Si el archivo no existe se crea, si existe se trunca (man creat), modo de creación 0666.
  — >& archivo: usa archivo como estándar error. Si el archivo no existe se crea, si existe se trunca (man creat), modo de creación 0666.
- Background (&): un mandato o secuencia terminado en '& 'supone la ejecución asíncrona del mismo, esto es, el minishell no queda bloqueado esperando su terminación. Ejecuta el mandato sin esperar por él imprimiendo por pantalla el identificador del proceso por el que habría esperado con el siguiente formato: [\n".
- Prompt: mensaje de apremio antes de leer cada línea. Por defecto será "msh>".

Obtención de la línea de mandatos leída

Para obtener la línea de mandatos tecleada por el usuario se recomienda proporcionar al alumno una función obtain_order cuyo prototipo es el siguiente:

```
Int obtain_order (char ***argvv, char **filev, int *bg);
```

El código fuente de esta Función puede encontrarse en el material complementario existente en la página Web del libro. A continuación, describimos su comportamiento. La llamada devuelve O en caso de teclear Control-D (EOF), —1 si se encontró un error. Si se ejecuta con éxito la llamada, devuelve el número de mandatos más uno. Así:

- Para 1s -1 devuelve 2.
- Para 1s | sort devuelve 3.

El argumento argvv permite tener acceso a todos los mandatos introducidos por el usuario. Con el argumento f 11ev se pueden obtener los archivos utilizados en la redirección:

- Filev [0] apuntará al nombre del archivo a utilizar en la redirección de entrada en caso de que exista o NEJLL si no hay ninguno.
- Filev [1] apuntará al nombre del archivo a utilizar en la redirección (le salida en caso de que exista o N U LL si no hay ninguno.
- Filev [2] apuntará al nombre del archivo a utilizar en la redirección (le la salida de error en Caso de t exista o NULL S no hay ninguno.

El argumento bg es 1 si el mandato o secuencia (le mandatos debe ejecutarse en background. Si el usuario teclea ls —1 sort > archivo, los argumentos anteriores tendrán la disposición que se muestra en la Figura C. 1

Desarrollo del minishell

Para desarrollar el minishell el alumno debe seguir una serie de pasos de tal forma que se construyan el, minishell de forma incremental. En cada paso se añadirá nueva funcionalidad sobre el anterior.

1. Ejecución de mandatos simples del tipo 1 s — 1, who, etc. Para esta sección deberá usar as llamadas al sistema fork, wait y execve.

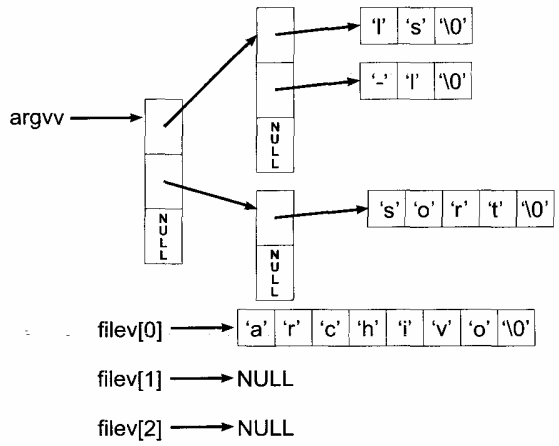


Figura C.1. *Disposición de los argumentos en argvv*

2. Ejecución de mandatos simples con redirecciones (entrada, salida y de error). Además de las llamadas anteriores deberá usar las llamadas dup o dup2.
3. Ejecución de mandatos simples en back No se debe usar wait.
4. Ejecución de secuencias de mandatos conectados por pipe. Usar la llamada pipe.
5. Tratamiento de señales. Ni el minishell ni las órdenes lanzadas en background deben morir por señales generadas desde teclado (SIGINT, SIGQUIT) (man signal), por tanto, éstas son ignoradas. Por contra, las órdenes lanzadas en foreground deben morir si le llegan estas señales, por tanto, mantienen la acción por defecto.
6. Ejecución de mandatos internos. Un mandato interno es aquel que bien se corresponde directamente con una llamada al sistema o bien es un complemento que ofrece el propio, minishell. Para que su efecto sea permanente, ha de ser implementado y ejecutado dentro del propio, minishell. Será ejecutado en un, subshell (man fork) sólo si se invoca en background o aparece en una secuencia y no es el último.

Todo mandato interno comprueba el número de argumentos con que se le invoca, y si encuentra este o cualquier otro error, lo notifica (por la salida de error estándar) y termina con valor distinto de cero.

Los mandatos internos del minishell son:

- cd [Cambia el directorio por defecto (man s2 chdir). Si aparece Directorio debe cambiar al mismo. Si no aparece, cambia al directorio especificado en la variable de entorno HOME. Presenta (por la salida estándar) como resultado el camino absoluto al directorio actual de trabajo (man getcwd) con el formato: "%s\n".
- umask [Cambia la máscara de creación de archivos (man s2 umask). Presenta (por la salida estándar) como resultado el valor de la actual máscara con el formato: "%o \n". Además, si aparece Valor (dado en octal, man strtol), cambia la máscara a dicho valor.

C.2.3. Código fuente de apoyo

Para Facilitar la realización de esta práctica, se recomienda proporcionar a los alumnos un parser que les evite tener que programar el suyo propio. Esto es importante, porque nuestra experiencia

nos dice que si no se proporciona dicho parser, los alumnos dedican la mayor parte del tiempo ha programado y no se centran en los aspectos de sistemas operativos. En concreto, a nuestros alumnos se les proporciona el siguiente código de apoyo, que puede encontrarse en la página Web del libro:

- Makefile. Archivo fuente para la herramienta make. Con él se consigue la recopilación automática sólo de los archivos fuente que se modifiquen.
- y. c. Archivo fuente de C. Define funciones básicas para usar la herramienta lex.
- Scanner. 1. Archivo fuente para la herramienta lex. Con él se genera automáticamente código C que implementa un analizador lexicográfico (scanner) que permite reconocer al tokens, considerando los posibles separadores (\t 1
- parser.y. Archivo fuente para la herramienta yacc. Con él se genera automáticamente código C que implementa un analizador gramatical (parser) que permite reconocer sentencias correctas de la gramática de entrada del minishell.
- main. c. Este archivo es el que debe modificar el alumno para incluir en él su práctica. Se recomienda estudiar detalladamente la función obtain_order para la correcta comprensión del uso de la función de interfaz. La versión que se ofrece hace eco de las líneas tecleadas que sean sintácticamente correctos. Esta funcionalidad debe ser eliminada y sustituida por la ejecución de las líneas tecleadas.

C.2.4. Recomendaciones generales al alumno

- Desarrolle el, minishell por etapas, complicándolo progresivamente. Comience implementando una versión elemental como la que aparece en el libro de texto.
- Continúe introduciendo la funcionalidad en el orden en que se describen en la sección Desripción de la práctica.
- Lea las páginas de manual a las que se hace referencia.
- Cuando tenga una idea clara de cómo implementar lo que se le pide, codifíquelo, compile y pruebe su práctica.

C.2.5. Documentación a entregar

Se recomienda que el alumno entregue los siguientes archivos:

- autores. Archivo con los datos de los autores.
- memoria. txt. Memoria de la práctica.
- Main. c. Código fuente del minishell, implementando toda la funcionalidad que se requieren.

C.2.6. Bibliografía

S. R. l3ouriie. luí' (!Nlt' ,S'v.vte,n. Addison—Wesley, 1953.
M J. Rochkiiid. Ailiíimeí/ UNIX Prot,','cuniiiint,'. lkeniice—l-laII, 1955.
SUN Miciosystctiis. P;oi (Jtjljíje.s aiid Librarjc.s. Sun N4icrosysteiiis. 1990.

PRÁCTICA C.3. LLAMADAS AL SISTEMA PARA LA GESTIÓN DE ARCHIVOS

C.3. 1. Objetivo de la práctica

Esta práctica permitirá al alumno familiarizarse con los servicios que ofrece el estándar POSIX para la gestión (le archivos y directorios.

Esta práctica cubre los conceptos del Capítulo 8.

C.3.2. Descripción de la funcionalidad a desarrollar por el alumno

Parte obligatoria de la práctica

Se trata de desarrollar dos programas similares a los mandatos cp y 1s de UNIX. El primero se denominará copy y el segundo mus. A continuación, se describen estos dos programas.

Programa copy

Este programa copiará un archivo a otro. Su sintaxis será la siguiente:

copy archivo_origen archivo_destino

El programa deberá copiar archivo_origen en archivo_destino. Si el archivo destino no existe, deberá crearse. En caso de existir, deberá crearse su contenido. En este caso se deberán utilizar las llamadas al sistema POSIX que sean necesarias.

Programa mils

Se trata de desarrollar un programa similar al mandato 1 s de UNIX, aunque con mucha funcionalidad reducida. El programa se denominará mus y recibirá el nombre de uno o varios directorios y mostrará el contenido de cada tino de ellos. Si no recibe ningún argumento. listará el contenido del directorio actual (.). En el caso de que alguno de los argumentos no sea un directorio sino un archivo ordinario, el programa tratará directamente el archivo imprimiendo una línea que lo describa.

Cuando no recibe ninguna opción muestra sólo el nombre de los archivos contenidos en el directorio restringiéndose a aquellos cuyo nombre no empieza por el carácter . (se podría decir que en UNIX los archivos cuyo nombre empieza por dicho carácter son ocultos). El programa puede además recibir dos opciones:

- Si recibe la opción —a, mostrará todos los archivos, es decir, los ocultos y no ocultos.
- Si se especifica la opción -1, además del nombre de cada archivo se imprimirá en la misma línea algunos de sus atributos: su tipo (R para regular. 1) para directorio, C para un dispositivo de caracteres, B para un dispositivo de bloques y F para un FIFO), el número de enlaces, el tamaño y la fecha del último acceso. Para procesar este último valor se recomienda usar las funciones gmtime o localtime (se recomienda consultar el manual interactivo del sistema para aclarar su uso: man gmtime o man localtime).

Un ejemplo de ejecución del mandato mus —la sería el siguiente:

```
D       2   12456       27/1/98 .
D       6   8978        27/1/98 ..
R       1  9878         27/1/98 f1.o
R       1  1024         27/1/98 f2
R       1  2039         27/1/98 f3.c
R       1  2345         27/1/98 f5.c
R       1  19288        27/1/98 f7.c
R       1  512          27/1/98 f9.c
```

La primera columna indica el tipo de cada archivo, la segunda el número de enlaces, la tercera el tamaño, la cuarta la fecha del último acceso. La última línea contiene el nombre del archivo.

Parte opcional de la práctica

Se plantea una parte opcional que consiste en modificar el programa mil s anterior para que realice un listado recursivo (opción —R) de todo el árbol de archivos y directorios que hay por debajo de cada uno de los directorios que recibe como argumento.

Uno de los aspectos que hay que tener en cuenta a la hora de realizar el recorrido recursivo es que se debe evitar atravesar los directorios. y . . presentes en todo directorio ya que de hacerlo se entraría en un bucle infinito.

Para emular dentro de lo posible el comportamiento de este mandato en UNIX, el recorrido del árbol se hará como se describe a continuación. No se realizarán las llamadas recursivas correspondientes a los subdirectorios encontrados en un determinado directorio hasta que no se haya tratado dicho directorio. Una posible forma de llevar a cabo este recorrido es utilizar una lista. Cada vez que el programa encuentre un directorio, insertará su nombre en la lista. Cuando se haya terminado de recorrer un determinado directorio, se comenzará con los directorios que se han almacenado en la lista, cada uno de los cuales se recorrerá a su vez de la forma descrita. A continuación se presenta un posible pseudo código de dicho recorrido. Este pseudocódigo asume la existencia de dos funciones, insertar y extraer. La primera introduce el nombre de un directorio en la lista y la segunda lo extrae.

```
recorrido (root) {
        para cada nodo que cuelgue directamente de roo {
        visitar eL nodo;
        if (nodo es un directorio)
                insertar (nodo);
        }
        While (lista no sea vacía) {
                extraer (&next)
                recorrido(next)
        }
}
```

El alumno deberá modificar el programa mi1s siguiendo las siguientes formas:

- Se deberá recuperar la opción —R dentro de la función main.
- Se deberá incorporar el recorrido descrito anteriormente, para lo cual deberá implementar la lista y las funciones insertar y extraer comentadas.

C.3.3. Código fuente de apoyo

Para facilitar la realización de la práctica se recomienda proporcionar a los alumnos un archivo comprimido que contenga el código fuente de apoyo. Dentro del existente de la página web del libro. Se han incluido los siguientes archivos:

- Make file. Archivo fuente para la herramienta make. Con él Se Consigue la recopilación automática de los archivos fuente cuando se modifiquen. Basta con ejecutar el mandato make para que el programa se compile de forma automática.
- copia. Archivo fuente de C donde se incluirá el programa copy.
- mils.c. Archivo fuente de C donde se incluirá el programa mus.

C.3.4. Recomendaciones generales al alumno

Es importante estudiar previamente el funcionamiento del mandato cp y 1s de UNIX.

A continuación se listan los servicios que se pueden necesitar a la hora de desarrollar la práctica: open, read, write, close, chdir, closedir, getcwd, opendir, readdiry stat.

C.3.5. Documentación a entregar

Se recomienda que el alumno entregue los Siguientes archivos:

- autores. Archivo con los datos de los autores.
- memoria . txt. Memoria de la práctica.
- copy. c. Código fuente del programa copy.
- mils.c. Código fuente del programa mus.

C.3.6. Bibliografía

J. L. Antonakos y K. C. Mansfiled ir. Progrwnación Esi,ucuiirada en (7. Pz'cniice—Hall. 997.
B. Kcrnigham y D. Ritchie. TIu' (7 progralnining Language, 2. edición. Pre,iticc—Hall. I 9
M. J. Rochkind. Advaneed UNIX Programining. Prentice-Hall. l9

PRÁCTICA C.4. MANEJO DE INTERRUPCIONES

C.4.1. Objetivo de la práctica

En esta práctica se plantea como objetivo el manejo de las interrupciones del reloj de forma similar a como lo hace un sistema operativo. Para ello se va a implementar un manejador de reloj. La práctica se va a desarrollar sobre LINUX, lo que impone algunas restricciones, por lo que el dispositivo reloj se va a simular con un proceso que emite señales periódicas, equivalentes a las interrupciones del reloj real.

Sirve como trabajo práctico para el tema de entrada/salida.

C.4.2. Descripción de la funcionalidad a desarrollar por el alumno

La práctica tiene tres partes fundamentales:

- Gestionar la interrupción del reloj.
- Manejar el tiempo de la aplicación de la práctica en base al reloj virtual de la misma.
- Implementar un programa de prueba que permita comprobar la funcionalidad del manejador de reloj implementado.

La pieza esencial es el manejador de reloj que recibe las llamadas del manejador de interrupción y del programa de usuario, procesando las mismas de forma similar a como lo hace LINUX. La idea fundamental de la práctica consiste en construir un manejador de reloj que permita gestionar un reloj virtual en nuestra aplicación y colas de temporizadores y ciclos. Cada temporizador y ciclo puede tener asociado una función (especificable por el usuario), de forma que cuando vence alguno de estos elementos se ejecuta la función asociada.

Funcionamiento del manejador

Cada vez q llega una interrupción de reloj, se mira si tenemos un tick (unidad mínima) en el reloj de nuestra aplicación. En caso positivo, la rutina de tratamiento de interrupción avisa al manejador de reloj (CLOCK_TASK) y éste hace todo el proceso correspondiente a un tick de reloj.

La unidad mínima del reloj de la aplicación (tick) no tiene por qué coincidir con la frecuencia de las interrupciones del reloj hardware (18,7 mt, por segundo = aprox. 58 ms). Puede ser cualquier múltiplo de ellas (en milisegundos), ajustable por el usuario mediante una operación (SET_TICK) Dicha unidad debe poder ser cambiada dinámicamente durante la ejecución de la aplicación.

Además el manejador de reloj tiene que permitir mantener un reloj virtual de la aplicación. Dicho reloj virtual se inicializará con el tiempo de la máquina cuando arranque la aplicación y se mantendrá mediante los ticks del reloj fijado (horas, minutos y segundos). Este reloj permitirá dos operaciones: GET y SET Estas operaciones permiten cambiar la hora y leerla respectivamente.

Un aspecto que conviene destacar es que este reloj es relativo a la aplicación y no tiene por qué coincidir con el de la máquina.

Por último, el manejador debe permitir poner, dentro de la aplicación, Temporizadores y ciclos (SET ALARM y SET_LOOP), asociando la ejecución de una función al vencimiento de estos eventos. El manejador debe mantener dos vectores estáticos, con un máximo de diez elementos cada uno, para mantener las direcciones de las l'unciones asociadas a estos elementos (utilizar punteros a funciones). Cuando se ponga un temporizador o ciclo, el manejador lo insertará en la lista. Cada vez que venza un tick, debe mirar si alguno de estos elementos ha vencido y ejecutar la función asociada. Además, el manejador debe proporcionar dos funciones para eliminar temporizadores (RESET_ALARM) y ciclos (RESET_LOOP) fijados anteriormente.

Módulos de la práctica

A continuación, se describen los módulos que componen la práctica, el comportamiento esperado (le los mismos y una posible interfaz para sus funciones, junto con la funcionalidad que se espera de cada uno de ellos.

Módulo que gestiono la interrupción del reloj (HARDWARE)

La pieza clave de este módulo es la rutina de tratamiento de interrupción del reloj. Dicha rutina se ejecuta cada vez que llega una interrupción de reloj y debe llevar a cabo las siguientes acciones:

- Incrementar una variable temporal y compararla con el período mínimo (tick = n interrupciones hardware) del reloj software de la aplicación.
- Si hay un tick, llamar al manejador de reloj con la operación CLOCK_TICK.
- Ejecutar la rutina de interrupción de reloj del sistema para que se mantenga la hora de la máquina.

Además de esta rutina, éste módulo debe tener varias operaciones que permitan:

- Inicializar el tratamiento de interrupción, fijando la rutina definida por el usuario.
- Fijar el número de interrupciones que constituyen un tick.
- Restaurar los vectores de interrupción iniciales del sistema cuando acabe la aplicación.

Manejador de reloj (CLOCK_TASK)

Este módulo se encarga de hacer las operaciones propias de un manejador de reloj, excluyendo las de planificación.

La funcionalidad que debe admitir este módulo es la siguiente:

- Inicializar el manejador y el módulo de tratamiento de interrupción (poner un Lick de reloj software cada lOO ms).
- Poner una alarma (SET ALARM).
- Quitar una alarma (RESET ALARM).
- Poner un ciclo (SET Loop).
- Quitar un ciclo (RESET_LOOP)
- Preguntar hora del reloj (GET_TIME).
- Fijar hora del reloj (SET_TIME).
- Fijar el Lick del reloj software (SET_TIME).
- Fijar el vector de tratamiento de la interrupción (SETVECT).
- Procesamiento del tick de reloj (CLOCK_TICK).

Este módulo maneja dos tipos de elementos de tiempo distintos:

- Alarmas, que producen una única interrupción cuando expiran.
- Ciclos, que producen interrupciones periódicamente.

El manejador debe permitir poner temporizadores (SET_ALARN) y ciclos (SET_LOOP) dentro de la aplicación, asociando la ejecución de una función al vencimiento de estos elementos. El manejador debe mantener dos vectores estáticos, con un máximo de diez elementos cada uno, para mantener las direcciones de las funciones asociadas a estos elementos (utilizar punteros a funciones). Cuando se ponga un temporizador o ciclo, el manejador lo insertará en la lista, cada vez que venza un tick, debe mirar si alguno de estos elementos ha vencido y ejecutar la función asociada. Además, el manejador debe proporcionar dos funciones para eliminar temporizadores (RESET_ALARM) y ciclos (RE5ET_LOOP) fijados anteriormente. La función do_clock_set_vect permite cambiar la función que se usa para gestionar la interrupción.

La única función que exporta permite acceder a toda la funcionalidad del manejador mediante el código de las operaciones.

Su Prototipo es similar a:

void clock task message *msg)

´ Esta función es similar a la principal de un manejador de LINUX, sin embargo, tiene dos diferencias importantes:

- No es un bucle infinito.
- Usa paramatros de tipo message.

Los prototipos de las funciones que llevan a cabo las operaciones del manejador se describen a continuación:

```
int init_clock ()
int do_set_alarm        (message      *msg)
int do_reset_alarm      (message      *msg)
int do_set_loop (message      *msg)
int. do_reset_ loop     (message      *msg)
int do_set_time (message      *msg)
int do_get_time (message      *msg)
int do_set_tick  (message     *msg)
mt do_clock_set_vect    (message      *msg)
int do_clock_tick       (message      *msg)
```

A continuación, se describe, como ejemplo, el comportamiento que se espera de una de estas funciones.

int do_clock_tick (message *msg)

Esta función es llamada por el manejador de interrupción cada vez que hay un tick de reloj software.

- Revisa las colas de ciclos y alarmas y ejecuta las funciones de los que hayan vencido, pintando los resultados por pantalla.
- Reactiva los ciclos vencidos.
- Elimina las alarmas vencidas.
- Actualiza la hora del reloj de la aplicación (hora:minutos:segundos) y la escribe en la esquina superior derecha de la pantalla.

Módulo de prueba (RELOJ)

El módulo de prueba debe permitir acceder a la funcionalidad del manejador de reloj sin perder precisión en el manejo de las alarmas, ciclos y hora relativa a la aplicación (o la menos posible).

El esquema del programa de prueba es el siguiente:

Carátula de presentación (nombre,…

```
init_clock /‘inicializa el manejador */
while (TRUE)
/* Presentar pantalla de operadores posibles*/
1 - Poner alarma
2 - Borrar alarma
3 - Poner ciclo
4 - Borrar ciclo
5 - Leer hora de la aplicación
```

```
6 - Cambiar hora de la aplicación
7 - Fijar el tick del reloj de la aplicación
8 - Salir
/ * Leer teclado */
/ *Si opción correcta pulsada, ejecutar acción correspondiente*/
clock_task (&msg)
}
```

Cuando se pone un ciclo o alarma hay que indicar una función para que se ejecute.

El alumno debe proporcionar cinco funciones de prueba, todas ellas con el mismo formato, para que puedan ser ejecutadas en la aplicación.

El formato de dichas funciones de prueba es el siguiente:

```
función_i () {
escribe (“soy la función i \n”);
escribe (hora del sistema)
}
```

En la Figura C.2 puede verse el flujo de llamadas entre los distintos módulos del sistema.

Estructuras de datos

```
struct clock {
        int hora;
        int minutos;
        int segundos;
}

struct message {
        int caller;         /* identificador del origen del mensaje*/
        int operation;  /* código de operación */
        int duration;       /* duración del ciclo, alarma o tick del reloj */
        void(*function) ( ) /* puntero a función asociada a alarma, ciclo o
                                interrupción */
        int id;             /* identificador de alarma o ciclo fijado */

        struct clock time;   /* hora a escribir o leer */
        int result;         /* O = éxito; -l = error */

}
```

Vectores de ciclos y alarma

```
struct {
        int free;
        int ticks_left;
        char *function;
} Temporizadores [MAXIMO_TEMPOLRALIZADORES]

struct {
        int free;
        int ticks_left;
        char *function;
}  Ciclos [MAXIMO_CICLOS]
```

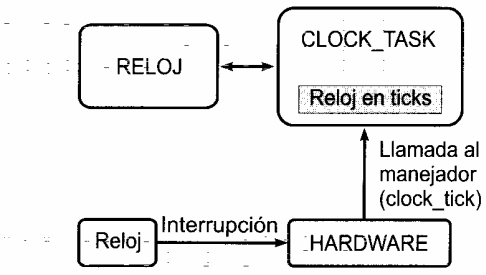


Figura C.2. *Esquema de módulos y llamadas de la práctica*

Cada vez que se ejecute do_clock_tick, el manejador de reloj debe recorrer estos vectores decrementando los ticks restantes de cada elemento (ticks_left). Si alguno llega a cero, el manejador debe ejecutar su función asociada.

C.4.3. Material de apoyo

Se sugiere proporcionar al alumno el pseudodispositivo que emula el reloj a base de señales. Si se quiere poca precisión, se puede usar la llamada al sistema alarma para dicha emulación. Si se necesita más precisión, se puede recurrir al contador de ticks de la UCP.

C.4.4. Recomendaciones generales al alumno

- Se recomienda implementar una versión sencilla de la práctica en la que el alumno pueda comprobar exclusivamente que funcionan bien las interrupciones.
- Se recomienda hacer un programa bien estructurado, evitando la linealidad.
- Se debe hacer un control exhaustivo de errores de ejecución en las llamadas al sistema y funciones.
- La salida de la práctica debe hacerse por la salida estándar. Sin embargo, los errores, y sólo los errores, deben escribirse por la salida de error estándar.
- Se debe seguir estrictamente las recomendaciones dadas en la sección de descripción.

C.4.5. Documentación a entregar

Se recomienda que el alumno entregue los siguientes archivos como solución a esta práctica:

- Memoria descriptiva de la misma, incluyendo: listados, explicación, esquemas de los nu5du— los y una traza de ejecución de la práctica como prueba de que funciona. Archivo: memoria.txt.
- Las fuentes de la aplicación, un ejecutable e indicaciones de cómo compilar los distintos módulos de la misma. Archivos:
  — reloj. c. Contendrá el código del módulo user y el programa principal.
  — hardware.c Contendrá el código del módulo hardware.
  — cock_ task.c. Contendrá el código del módulo clock_ task.

— reloj.exe. Ejecutable de prueba.
— makefile. Archivo de compilación de los módulos de la práctica.

C.4.6. Bibliografía

A. Silherschatz y P. B. Galvin. Operatin Systems' Concepis. S. edición. Addison-Wesley. 1999.
D. A. Solomon. Insule Windows NT. Microsoft Press, 1998.
W. R. Stevens. Ad Progrwnming in 11w UNIX Eni'ironmnent. Addison-Wesley. 1992.
A. S. Tancnhauin. Operating Sy.s't'mns: Desi and Implementation, 2: edición. Prentice—Hall, 1987.
—Modern Operaling Systems. Prentice-Hall, 1991.
P. S. Wang. An Introduction to Berkeley Unix. Wadsworth, lnlcrnational Student Edilion, 1988.

PRÁCTICA C.5. COMUNICACIÓN DE PROCESOS CON SOCKETS

C.5.1. Objetivo de la práctica

En esta práctica se pretende que el alumno se familiarisara con los mecanismos de comunicación entre procesos, usando sockets como mecanismo de comunicación. Además. El alumno tendrá ocasión de experimentar con un sistema cliente-servidor.

Sirve como trabajo práctico para el tema de comunicación y sincronización y í el tema de sistemas distribuidos.

C.5.2. Descripción de la funcionalidad a desarrollar por el alumno

Esta práctica permitirá al alumno estudiar los mecanismos existentes en UNIX para comunicar procesos con sockets y el concepto de relación cliente-servidor.

La práctica consiste en el desarrollo de dos programas C que definen dos procesos denomina dos csocket y ssocket respectivamente. El resultado de la ejecución conjunta de ambos procesos será similar al obtenido mediante la ejecución de la orden cat de UNIX, es decir, visualizar el contenido de un archivo cuyo nombre se indica.

El proceso csocket lee el nombre del archivo por la (‘mitrada estándar (teclado) y se lo envía al servidor por un socket de recepción de mensajes del servidor, que se supone creado por el servidor.

Si el envío es correcto, crea un socket para recibir y ejecuta un bucle de lectura del socket y de escritura de lo leído por la salida estándar (pantalla). Cuando el archivo está terminado, el servidor le envía un código de finalización.

El proceso ssocket crea un, socket de recepción y recibe del cliente el nombre del archivo a leer. Si la recepción es correcta, abre el, socket (socket) del cliente y el archivo recibido (open) y ejecuta un bucle de lectura del archivo (send/write) y envía (recv/read) por el socket del cliente recibido como parámetro. Una vez leído el archivo completo, cierra todos los archivos y descriptores (close).

Para la comunicación se usa una estructura de tipo mensaje. El uso de esta estructura permite generalizar el modelo de comunicación para su uso en un esquema cliente-servidor (le tipo general. El servidor debe poder atender a cualquier cliente de forma secuencial. Para ello supondremos que el servidor tiene una dirección de recepción de mensajes conocida por todos los clientes.

Puesto que el servidor debe conocer la identidad de la cola cliente donde debe responder, usará una estructura mensaje como la siguiente:

```
typedef struct {
        sockaddrin dirección; / dirección del emisor /
        char     datos datos que ha enviado */
        unsigned mt longdat; /* longitud de datos válidos dentro de datos /
        unsigned mt orden; /* número de orden del mensaje /
      }mensaje;
```

donde se incluye la dirección del cliente adonde debe enviarse la respuesta del servidor. Esta modificación permite al servidor atender peticiones de cualquier cliente que conozca su dirección y a un mismo cliente recibir respuestas a sus mensajes por distintos sockets (uno por cada servicio).

Aunque existen distintos tipos de sockets (INET, UNIX, SNET,...), esta práctica se limitará a los sockets de tipo INET y, dentro de ellos, a los sockets de tipo DATAGRAMA y STREAN.

Todas las llamadas al sistema devuelven un error en caso de ejecución incorrecta. El alumno debe controlar los posibles errores y mostrarlos por la salida de error.

Para construir un servidor similar a uno real de UNIX, en esta práctica se desarrollarán varios apartados.

Parte 1. Servicio secuencial

Similar a lo descrito hasta el momento.

Parte 2. Servicio concurrente basado en procesos

Para cada petición de servicio, el servidor crea un proceso hijo al que le pasa el mensaje recibido (Fig. C.3). Este es el responsable de ejecutar el servicio completo.

Parte 3. Servicio concurrente basado en procesos ligeros (PL)

Para cada petición de servicio, el servidor crea un PL al que le pasa el mensaje recibido. Este PL es el responsable de ejecutar el servicio completo. El esquema es similar al de la Figura C.3. pero en lugar de procesos convencionales se crean procesos ligeros usando threads de POSIX.

C.5.3. Introducción a los sockets

El mecanismo de comunicación a utilizar en esta práctica son los sockets. Un socket (enchufe) es un punto extremo de comunicación, que permite comunicar dos procesos punto a punto. Estos procesos pueden ejecutarse en la misma máquina o no. Permiten la utilización de protocolos de red y de transporte para la comunicación de procesos situados en distintos nodos. Esta característica hace que sean más potentes que los pipes, y un mecanismo de comunicación a considerar en sistemas distribuidos.

Para trabajar con .sockets es siempre necesaria la existencia de extremos con direcciones conocidas y que estos extremos se sincronicen adecuadamente, si bien la forma de sincronización depende del tipo de socket.

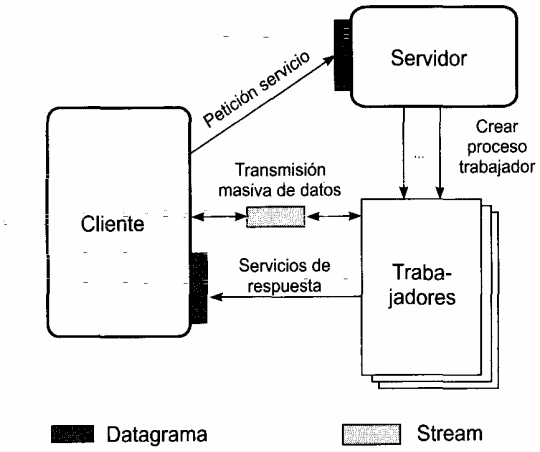


Figura C.3. *Esquema Cliente-servidor con sockets y concurrencia*

Existen sockets de distintos tipos en un sistema UNIX, diferenciándose cada tipo por la semántica que proporcionan en el protocolo de comunicaciones. En esta práctica utilizaremos las siguientes constantes:

- PF INET. Familias de sockets de internet.
- SOCKADDR_ANY. Recibir de cualquier dirección, en la estructura sockaddr_in.
- UDP. Protocolo de usuario.
- IPPORT_USERRESERVED. Numero de socket mínimo que puede utilizar un usuario; el socket a utilizar será esa constante más algo.

En esta práctica se utilizarán dos tipos de sockets de los existentes en UNIX:

- **Datagramas**. Son sockets que se caracterizan porque:

  — No necesitan una operación de conexión explícita antes de enviar.
  — No aseguran el orden de llegada de los datos si éstos se parten en distintos paquetes, por lo que no son muy indicados para grandes transferencias.

- **Streams**. Se caracterizan porque:

  -— Necesitan una operación explícita de conexión antes de enviar.
  -— Aseguran el orden de llegada de los datos si éstos se parten en distintos paquetes (son aptos para grandes cantidades de datos).

C.5.4. Llamadas al sistema operativo a usar en la práctica

Servicios de archivos

Los sockets se manejan en UNIX como descriptores de archivos, por lo que aceptan muchas llama das de archivos como read. write, close o ioctl.

Servicios de sockets

Se describe aquí el pequeño subjunto de servicios de sockets a usar en la práctica.

int socket(int domain, int type, int protocol);

Crea un punto de comunicación tipo socket y devuelve un descriptor de archivo asociado con él. Dominios de sockets: PF UNIX, PP INET y PF_IMPLINK. Tipos de sockets: SOCKSTREAN, SOCK_DGRAM, SOCK_RAW, SOCKSEQPACKET y SOCK Los protocolos son dependientes del dominio.

int bind(int s, struct sockaddr *name int namelen);

Esta llamada asigna un nombre (una dirección) a un socket sin nombre, es decir, a uno creado con socket. A partir de su ejecución, se puede acceder a la dirección a través del socket sin especificar nada más que el identificador del socket.

int. listen(int s, int n)

Escucha a la espera de conexiones al socket s. Permite n conexiones simultáneas como máximo. Si se sobrepasa este número, devuelve un error. Sólo válida para sockets tipo SOCK_STREAM.

int accept(int s, struct sockaddr *addr mt *addrlen);

Acepta conexiones sobre un socket que ha sido previamente creado (socket), enlazado (bind) a una dirección y que está a la escucha (listen). Esta llamada es bloqueante y extrae la primera conexión que exista. Crea un nuevo socket conectado y devuelve su descriptor de archivo.

int connect(int s, struct sockaddr *name, int namelen)

Intenta establecer una conexión con otro socket. Este último debe estar ejecutando una llama da accept. Generalmente sólo se puede establecer una conexión de ¡ simultánea para SOCK S'l'REAM y múltiples para SOCK_

int select (int width, fd_set *readfds, fd_set *writetds
fd_set *exceptfds struct tirneval *timeout)

Examina los descriptores de socket de readfds. writefdsn y exceptfds para ver si alguno está lisio para leer, recibir o tiene alguna excepción pendiente. El parámetro width es una máscara de bits que indica los descriptores que deben ser muestreados.

getsockname (int s, struct sockaddr *name, int *namelen

Devuelve el nombre asociado a un socket.

struct hostent *qerhostbyname(char *name)

Devuelve los datos de comunicación asociados a la máquina cuyo nombre es nane.

int recvfrorn(int s, int len, int flag f] struct sockdddr *fromm, Int *fromlen)
int send int s, char *buf, int len, int ilags)

Distintas llamadas de recepción sobre sockets. recv sólo puede usarse para sockets conecta dos. Se lee sobre buf una longitud máxima len, con la operación modificada por flags. Si to es no nulo, se introduce en dicho parámetro los datos del origen del mensaje. Obteniendo su longitud en tolen.

int sendio(int s, mt len, int flags, struc sockaddr *io, int *io);
int send(int s, char *buf, int len, int flags)

Distintas llamadas de envío sobre sockets. send sólo puede usarse para sockets conectados. Se envía buf con una longitud máxima len, con la operación modificada por flags. Si to es no nulo, se introduce en dicho parámetro los datos del destino del mensaje, incluyendo su longitud en iolen.

Llamadas de procesos ligeros (Threads)

Para resolver la sección que usa procesos ligeros, se recomienda al lector acudir al tema de Procesos, donde se describen las llamadas al sistema para procesos ligeros, tales como pihread_desiroy, pihread_yield, pthread sleep, etc.

C.5.5. Material de apoyo

Como material de apoyo se sugiere a los profesores proporcionar a los alumnos algún ejemplo de programación con sockets. Para ello se puede usar el libro de Stevens, Unix Network Programing donde hay múltiples ejemplos.

Igualmente, se les puede proporcionar un ejemplo de programa que use las llamadas al sistema
fork y exec.

Además, se sugiere que para evitar una explosión de definiciones distintas se proporcione un archivo, denominado Soc. . h, que, usando el lenguaje C, incluya las definiciones generales para el desarrollo de la práctica.

C.5.6. Material a entregar

.
Para tratar de que las distintas soluciones se puedan entregar de forma clara y bien separada, se sugiere pedir a los alumnos los siguientes archivos como resultado de su implementación de la práctica:

- csockei.c (común)
- ssockei.c (Parte 1)
- spsockei. c (Parte 2)
- sisockei.c (Parte 3)

C.5.7. Bibliografía

W. R. Stevens. UNIX Network Programming Addison-Wesley, 1990.
—Advanced Programming in the UNIX Environment Addisson—Wesley, 1992.
P. S. Wang. An Introduction to Berkeley Unix. Wadsworth, International Student Edition, 1988.
A. S. Tanenbaum. Sistemas Operativos Modernos. Prentice—Hall, 1992.

PRÁCTICA C.6. DISEÑO DE UN SISTEMA MULTIPROGRAMADO

C.6.1. Objetivo de la práctica

El objetivo de esta práctica es llegar a conocer los principales conceptos relacionados con la gestión de procesos y la multiprogramación. El trabajo va a consistir básicamente en convertir un sistema con monoprogramación en uno multiprogramado.

La multiprogramación y, en general, la concurrencia son probablemente los temas más importantes en la enseñanza de los sistemas operativos, aunque también son los más complejos de entender. Es muy difícil conseguir comprender lo que ocurre cuando se están ejecutando concurrentemente varias actividades.

Esta dificultad se acentúa notablemente cuando se está trabajando en el nivel más bajo del sistema operativo. Por un lado, en este nivel la mayoría de los eventos son asíncronos. Por otro, en él existe una gran dificultad para la depuración, dado el carácter no determinista del sistema y la falta de herramientas de depuración adecuadas.

Todas las consideraciones expuestas hasta ahora explican el motivo de que la programación de sistemas tenga una productividad tan baja y que los sistemas operativos tengan «mala fama» debido a sus «caídas» y «cuelgues» imprevistos.

Por tanto, al enfrentarse con esta problemática, es importante resaltar desde el principio las dificultades que se encontrará para la realización de este proyecto práctico. Sin embargo, aunque parezca un poco sorprendente, enfrentarse con esos problemas es a su vez un objetivo del mismo.

Por último, hay que resaltar que en esta práctica se plasman muchos de los conceptos de sistemas operativos relacionados con la concurrencia y la gestión de interrupciones tales como:

- El arranque del sistema operativo.
- El manejo de las interrupciones.
- El manejo (le las excepciones.
- El manejo de las llamadas al sistema.
- La multiprogramación.
- El cambio de contexto.
- La planificación de procesos.
- La sincronización entre procesos.

C.6.2. Entorno de desarrollo de la práctica

Uno de los problemas que surgen cuando se pretende hacer una práctica de este tipo es decidir en qué entorno se lleva cabo.

La elección más obvia es utilizar un sistema operativo real del que se dispongan las fuentes (tales como MINIX o LINUX) y realizar modificaciones en el código del mismo para incluir nuevas funcionalidades. Este enfoque es el más realista, pero, sin embargo, presenta algunos problemas que dificultan considerablemente su aplicación práctica.

En el caso de sistemas concebidos para su utilización en la vida real, como LINUX, el código de los mismos es demasiado complejo debido a las «oscuras» optimizaciones que suelen estar presentes en este tipo de sistemas.

En el caso de MINIX, al tratarse de un sistema diseñado con un objetivo pedagógico, su uso directo es más viable. Sin embargo, la necesidad de recompilar el sistema operativo en cada modificación de mismo y su dificultad de la depuración hace que sea problemática su utilización como entorno de prácticas.

Ante esta situación se han creado algunos entornos que simulan el comportamiento del sistema operativo en el contexto de uno o varios procesos convencionales. Este enfoque es más adecuado, aunque, como todo entorno simulado, se pierde parte de la experiencia que podría obtener trabajan do directamente en un sistema real.

Uno de los entornos de este tipo más conocidos y utilizados es el sistema NACHOS desarrollado en la universidad de Berkeley. Aunque este sistema proporciona un entorno con una serie de características interesantes, presenta algunas deficiencias (como, por ejemplo, la necesidad de usar un compilador cruzado o su carácter determinista) que hacen que se haya considerado que era conveniente usar un entorno propio denominado, minikernel. Algunas características de este entorno son las siguientes:

- Tiene un modo de operación que imita el comportamiento de los sistemas operativos monolíticos.
- Se puede utilizar en diferentes plataformas UNIX (p. ej.: LINUX o Digital UNIX). Para lograr esta transportabilidad, todas las operaciones del sistema dependientes de la plataforma están «escondidas» en un módulo objeto proporcionado por el entorno. El programador del sistema operativo no va a utilizar código en ensamblador.
- No requiere el uso de un compilador cruzado. Los programas se compilan usando el compilador nativo de la máquina.
- Los programas de usuario son programas convencionales, con la única peculiaridad de que usan los servicios el, minikernel en vez de los de UNIX.
- Tanto los programas de usuario corno el código del sistema operativo ejecutan sobre el procesador nativo. A diferencia de otros entornos como NACHOS donde los procesos de usuario ejecutan sobre la simulación de un procesador (un MIPS en el caso de NACHOS).
- La emulación del hardware se realiza exportando algunas características del hardware real subyacente. El sistema incluye mecanismos que emulan las interrupciones, las excepciones y las llamadas al sistema.
- En la versión actual sólo existe una fuente de interrupciones: el reloj.
- La generación de excepciones es realista: Las excepciones reales que produce el programa durante su ejecución son propagadas al minikernel como tales.

Evidentemente, como material de apoyo de la práctica, se debe proporcionar este entorno de desarrollo denominado minikernel.

C.6.3. Estructura deL entorno de desarrollo

El entorno de desarrollo de la práctica intenta imitar dentro (le lo que cabe el comportamiento y estructura de un sistema real. En una primera aproximación, en este entorno se pueden diferenciar (res componentes principales:

- El programa cargador del sistema operativo (directorio boot).
- El sistema operativo minikernel propiamente dicho (directorio kernel).
- Los programas de usuario (directorio usuario).

A continuación, se describe cada una de estas parles haciendo especial énfasis en el sistema operativo.

Carga del sistema operativo

De forma similar a lo que ocurre en un sistema real, en este entorno existe un programa de arranque que se encarga de cargar el sistema operativo en memoria y pasar el control a su punto de entrada inicial. Este procedimiento imita el modo de arranque de los sistemas operativos reales que se realiza desde un programa cargador.

El programa cargador se encuentra en el subdirectorio boot y. en un alarde de originalidad, se denomina boot. Para arrancar el sistema operativo, y con ello el entorno de la práctica, se debe ejecutar dicho programa pasándole como argumento el nombre del archivo que contiene el sistema operativo. Así, suponiendo que el directorio actual corresponde con el subdirectorio boot y que el sistema operativo está situado en el directorio kernel y se denomina minikernel, se debería ejecutar: boot. . /kernel/minikernel. Observe que no es necesario ejecutar el programa de arranque desde su directorio. Así, si se está situado en el directorio base de la práctica, se podría usar el siguiente mandato: boot/boot kernel/minikernel.

El minikernel

El código del sistema operativo está organizado como un módulo de apoyo (archivo apoyo o), del que se proporciona únicamente su interfaz, y el módulo que contiene la funcionalidad principal del sistema operativo (archivo proc.c).

El módulo de apoyo

El objetivo principal de este módulo es simular el procesador y ofrecer servicios que permitan al sistema operativo su manejo. Las principales características de este procesador son las siguientes:

- Tiene dos niveles de ejecución: usuario y kernel.
- El tratamiento de las interrupciones se realiza mediante el uso de una tabla de vectores de interrupción.
- Hay cuatro vectores disponibles que corresponden con:

  — Vector O: Excepción aritmética.
  — Vector 1: Excepción por acceso a memoria inválido.
  — Vector 2: Interrupción de reloj.
  — Vector 3: Llamada al sistema.

- Cuando se produce una interrupción, sea del tipo que sea, el procesador realiza el tratamiento habitual, esto es, almacenar el contador de programa y el registro de estado en la pila, prohibir las interrupciones, elevar el nivel del procesador y cargar en el contador de programa el valor almacenado en el vector correspondiente. Evidentemente, al tratarse de operaciones realizadas l) el hardware, todas estas operaciones no son visibles al programador del sistema operativo
- La ejecución en modo kernel de una instrucción de retorno de interrupción restaurará el nivel de procesador y el estado de las interrupciones previo. Observe que, ciado que se pretende que el código a desarrollar no incluya ensamblador, la ejecución de esta instrucción está incluida en uno de los servicios exportados por este módulo.

Además de funcionalidad relacionada con la simulación del procesador, este módulo también incluye otras funciones de más alto nivel que intentan facilitar la labor (le programación del sistema operativo. Por ejemplo, se ofrecen funciones relacionadas con la creación del mapa de memoria de cada proceso a partir del ejecutable. Con esto se pretende que el desarrollo se centre en los aspectos relacionados con la gestión de procesos y no en otros como la gestión de memoria.

Las funciones ofrecidas por el modulo de apoyo se pueden clasificar en las siguientes categorías:

- Operaciones relacionadas con las interrupciones. En esta categoría se incluyen funciones que se pueden subdividir a su vez en los siguientes apartados:

  — Iniciación del controlador de interrupciones (iniciar_cont_int) y del controlador del reloj (iniciar_cont_reloj).
  — Instalación de un manejador de interrupciones (instal_man_int)
  — Inhibición y habilitación de tas interrupciones (prohibir_int, permitir_int, obtener_estado_int y fijar_estado_int).

- Operaciones relacionadas con la salvaguarda y recuperación de información del proceso.
  — activar_pila_kernel. Permite activar la pila del kernel del proceso.
  — salvar_en_pila. Salva en la pila del proceso actual una copia de los registros del procesador.
  — recuperar_de_pila. Recupera de la pila del proceso actual la copia de los registros cargándola en los registros del procesador y ejecuta la instrucción de retorno de interrupción.
  — cambio _ contexto. Salva el contexto de un proceso y restaura el de otro. Si no se especifica el proceso cuyo contexto debe salvarse, s6lo realiza la restauracióu.

- Operaciones relacionadas con el tratamiento de los argumentos. Estas funciones permiten obtener el número de llamada solicitada (obtener_nservicío, el número se recibe en un registro), los argumentos de la llamada (obtener _argumento, se reciben en la pila de usuario) y almacenar el resultado de la misma (almacenar_resultado. se devuelve en un registro).
- Funciones relacionadas con el mapa de memoria del proceso. Permiten realizar operaciones tales como crear el mapa de memoria del proceso a partir del ejecutable (crear_imagen) y liberarla (liberar_imagen), y crear una pila de kernel para el proceso (crear_pila_kernel) y liberarla (liberar_pila_kerne]).
- Operaciones misceláneas. En este apartado se agrupan una serie de funciones de utilidad diversa:

  — obtener_nivel_previo. Devuelve cual era el nivel de ejecución del procesador previo al actual.
  — halt. Ejecuta una instrucción HALT para parar el procesador hasta que se active una interrupción.
  — panico. Escribe un mensaje por la pantalla y termina la ejecución del sistema operativo.
  — escribir_ker. Escribe en la pantalla.
  — printk. Escritura con formato en la pantalla. Esta función de conveniencia se apoya en la anterior (escribír_ker).

El módulo proc

Éste es el módulo que contiene la funcionalidad del sistema operativo. Su nombre viene a indicar cuál es la labor fundamental de este minisistema operativo: la gestión de procesos. Como parte

material de apoyo para la realización de la práctica se proporciona una versión de este módulo que incluye una funcionalidad básica que corresponde con un sistema monoprogramado. Para ser más precisos, hay que aclarar que en este sistema inicial, aunque se puedan crear y cargar en memoria múltiples programas, el proceso en ejecución continúa hasta que termina. Este sistema se deberá convertir en un sistema multiprogramado siguiendo las pautas que se detallan posteriormente.

Antes de pasar a revisar las características principales de esta versión inicial es conveniente realizar algunas aclaraciones sobre el mismo:

- Como se comentó previamente, una vez que empieza a ejecutar un proceso, éste continúa hasta que termina. El estado del proceso será en todo momento en ejecución. No existe ningún servicio que cause que el proceso pase a un estado bloqueado. Sólo se invoca a la rutina de cambio de contexto cuando el proceso termina, sea voluntaria o involuntariamente. Observe que la rutina de interrupción no cambia el estado de los procesos.
- No van a existir problemas de sincronización dentro del núcleo causados por la ejecución concurrente de varias llamadas al sistema ya que se trata de un sistema monoprogramado. Según se vaya incluyendo en el sistema la funcionalidad pedida, habrá que analizar si puede surgir este tipo de problemas.
- Los problemas de sincronización dentro del núcleo entre una llamada al sistema y la rutina de tratamiento de interrupción van a ser relativamente simples, ya que en esta versión inicial la rutina de interrupción no realiza ninguna labor. Solamente es necesario asegurar que mientras se está salvando o recuperando información de la pila, realizando un cambio de contexto, o después de liberar la pila del sistema del proceso, no se produzca una interrupción Y que la rutina de interrupción también manipula esas estructuras de datos. Para ello se prohibirán las interrupciones en la sección de código problemática. Como se comentó para el caso anterior, según se incluya nueva funcionalidad en la rutina de interrupción será necesario revisar el sistema para comprobar si pueden surgir este tipo de problemas.

A continuación, se describen las principales características del sistema inicial:

- Estructuras de datos (contenidas en proc.h). Este módulo utiliza las siguientes variables globales:

  — tabla_procs. Tabla de procesos. Cada entrada representa un BCP. Se deberán incluir nuevos campos en el BCP cuando se considere oportuno.
  — lista_listos. Cola de procesos listos. Se trata de una lista con enlace simple. El tipo lista_BCPs guarda referencias al primer y último elemento de la lista. Este tipo puede usarse p otras listas del sistema operativo (p. ej.: en un semáforo).
  — p_proc_actual. Apunta al BCP del proceso en ejecución. Este BCP está incluido en la cola de listos.
  — tabla_servicios. Tabla que guarda en cada posición la dirección de la rutina del sistema operativo que lleva a cabo la llamada al sistema cuyo código corresponde con dicha posición.

- Iniciación. Una vez cargado el sistema operativo, el programa cargador pasa control al punto de entrada del mismo (en este caso a la función main de este módulo). En este momento,- el sistema inicia sus estructuras (le datos, los dispositivos hardware e instala sus manejadores en la tabla de vectores. En último lugar, crea el proceso inicial init y lo activa pasándole el control. Observe que durante esta l las interrupciones están inhibidas. Sin embargo, cuan do se activa el proceso init restaurándose su contexto inicial, se habilitan automáticamente las interrupciones puesto que en dicho contexto inicial se ha establecido previamente que esto sea así.

- Tratamiento de interrupciones externas. La única fuente externa de interrupciones es el reloj. La rutina de tratamiento instalada únicamente muestra un mensaje por la pantalla indicando la ocurrencia del evento. En esta rutina habrá que incluir progresivamente la funcionalidad pedida. Observe que esta rutina se ejecuta con las interrupciones externas inhibidas. En un sistema con varias fuentes de interrupción lo habitual sería mantener inhibidas las interrupciones de este mismo tipo pero habilitar las más prioritarias.
- Tratamiento de excepciones. Las dos posibles excepciones presentes en el sistema tienen un tratamiento común produciendo la terminación del proceso actual y la liberación de sus recursos. La rutina se inicia con las interrupciones inhibidas pero las habilita una vez salvada la información en la pila para, por último, volver a inhibirlas cuando es preciso.
- Llamadas al sistema. Existe una única rutina de interrupción para todas las llamadas. Esta rutina obtiene de un registro el código numérico de la llamada e invoca indirectamente a través de la tabla_servicios a la función correspondiente. Esta rutina se inicia con las interrupciones inhibidas pero, una vez salvada la información en la pila, las rehabilita. Una vez terminada la función específica del servicio, esta rutina inhibe las interrupciones y restaura la información de la pila. En la versión inicial sólo hay tres llamadas disponibles. La función asociada con cada una de ellas está referenciada desde la posición correspondiente de la tabla_servicios. Estas llamadas son las siguientes:

  — crear_proceso. Crea un proceso que ejecuta el programa almacenado en el archivo especificado (parámetro prog). Para ello realiza los pasos básicos implicados en la crea ción: buscar una entrada libre, leer el ejecutable y crear el mapa de memoria, reservar la pila del sistema para el proceso, rellenar el BCP adecuadamente, poner el proceso como listo para ejecutar e insertarlo al final de la cola de listos. Esta llamada devolverá un —l si hay un error y un o en caso contrario. El prototipo de este llamada es: int crear_proceso (char *programa).
  — terminar_proceso. Termina la ejecución de un proceso liberando sus recursos (BCP, mapa de memoria, pila del kernel. etc.). El prototipo de esta llamada es: init terminar_proceso ().
  — escribir. Escribe un mensaje por la pantalla (parámetro texto) de una determinada longitud (parámetro longi) haciendo uso de la función de apoyo escribir_ker. De vuelve siempre un O. El prototipo de este llamada es: int escribir (char * texto, unsigned int longi).

En la versión inicial, el código de las tres llamadas ejecuta la mayoría del tiempo con las interrupciones habilitadas. Al ir añadiendo la funcionalidad pedida puede ser necesario revisar el código para aumentar la zona de código que ejecuta con las interrupciones inhibidas.

- Planificador (función planificador). La versión inicial de este módulo se corresponde con un sistema monoprogramado. Por tanto, el planificador no se invoca hasta que termina el proceso actual. El algoritmo que sigue el planificador es de tipo FIFO: simplemente selecciona el proceso que esté primero en la cola de listos. Observe que, si lodos los procesos existentes están bloqueados (situación imposible en la versión inicial), se invoca la rutina espera_int de la que no se vuelve hasta que se produzca una interrupción.

Los programas de usuario

En el subdirectorio usuario existe inicialmente un conjunto de programas de ejemplo que usan los servicios del minikernel. De especial importancia es el programa init puesto que es el primer programa que arranca el sistema operativo. En un sistema real este programa consulta archivos de

configuración para arrancar otros programas que, por ejemplo, se encarguen de atender a los usuarios (procesos de login). De manera relativamente similar, en este sistema este proceso hará el papel de lanzador de otros procesos, aunque en nuestro caso no se trata de procesos que atiendan al usuario puesto que el Sistema no proporciona servicios para leer del terminal. Se tratará simple mente de programas que realizan una determinada labor y terminan.

Además del programa init, en este directorio existe un conjunto de programas que sirven (le ejemplo del uso de las llamadas al sistema.

Como ocurre en un sistema real, los programas tienen acceso a las llamadas al sistema como rutinas de biblioteca. Para ello, existe una biblioteca estática denominada libserv. a que contiene las funciones de interfaz para las llamadas. Esta biblioteca almacenada en el subdirectorio usuario/lib está compuesta de dos módulos:

- serv. c. Contiene las rutinas de interfaz para las llamadas. Se apoya en una función de nominada hacer_trap que es la que realmente ejecuta la instrucción (le llamada al sistema. Para hacer accesible a los programas una nueva llamada, se deberá modificar este archivo para incluir la rutina de interfaz correspondiente.
- misc.o. Contiene funciones de utilidad comO la definición de la función printf del sistema que, evidentemente, se apoya en la llamada al sistema escribir, de la misma manera que el printf de UNIX se apoya en la llamada write.

Todos los programas de usuario utilizan el archivo de cabecera usuario/include/servicios. h que contiene los prototipos de las funciones de interfaz a las llamadas al sistema.

Inclusión de una nueva llamada al sistema

Dado que parte de la labor de la práctica es incluir nuevas llamadas al sistema, se ha considerado oportuno incluir en esta sección los pasos típicos que hay que llevar a cabo en este sistema para hacerlo. Suponiendo que el nuevo servicio se denomina nueva, éstos son los pasos a realizar:

- Incluir en kernel /proc. c una rutina (que podría denominarse sis_nueva) con el código de la nueva llamada.
- Incluir en tabla_servicios (archivo kernel/include/proc.h) la nueva llamada en la última posición de la tabla.
- Modificar el archivo para incrementar el número de llamadas disponibles y asignar el código más alto a la nueva llamada.
- Una vez realizados los pasos anteriores, el sistema operativo ya incluiría el nuevo servicio, pero sólo sería accesible desde los programas usando código ensamblador. Por tanto, es necesario modificar la biblioteca de servicios (archivo usuario/lib/serv.c) para que proporcione la interfaz para el nuevo servicio. Se debería también modificar el archivo de cabecera que incluyen los programas de usuario (usuario/ include / servicios. h) para que dispongan del prototipo de la función de interfaz.
- Por último, hay que crear programas de prueba para este nuevo servicio y, evidentemente, modificar init para que los invoque. Asimismo, se debería modificar el archivo Makefile para facilitar la compilación de este nuevo programa.

C.6.4. Descripción de la funcionalidad a desarrollar por el alumno

Como se comentó previamente, la práctica va a consistir en modificar la versión inicial que se ci1trc como material de apoyo para incluir nuevas funcionalidades y añadir multiprogramación al mismo. Se deben realizar tas siguientes modificaciones sobre la versión inicial del sistema:

- Incluir una nueva llamada (obtener_id) que devuelva el identificador del proceso que la invoca. Corno puede observarse, se trata de un servicio que realiza un trabajo muy sencillo. Sin embargo, esta primera tarea servirá para familiarizarse con el mecanismo que se usa para incluir una nueva llamada al sistema. El prototipo de la función de interfaz sería el siguiente:
  int obtener_id_pr ().
- Incluir una nueva llamada (int dormir (unsigned int segundos)) que permita que un proceso pueda quedarse bloqueado un plazo de tiempo. El plazo se especifica en segundos corno parámetro de la llamada. La inclusión de esta llamada significará que el sistema pasa a ser multiprogramado ya que cuando un proceso la invoca pasa al estado bloqueado durante el plazo especificado y se deberá asignar el procesador al proceso elegido por el planificador. Observe que en el sistema sólo existirán cambios de contexto voluntarios y, por tanto, sigue sin ser posible la existencia de llamadas al sistema concurrentes. Sin embargo. dado que la rutina de interrupción del reloj va a manipular listas de BCPs. es necesario revisar el código del sistema para detectar posibles problemas de sincronización en el manejo de estas listas y solventarlos prohibiendo las interrupciones en los fragmentos de código correspondientes. Aunque se puede implementar esta llamada corno se considere oportuno, a continuación se sugieren algunas pautas:

  — Modificar el BCP para incluir algún campo relacionado con esta llamada.
  — Definir una lista de procesos esperando plazos.
  — Incluir la llamada que, entre otras labores, debe poner al proceso en estado bloqueado, reajustar las listas de BCPs correspondientes y realizar el cambio de contexto.
  — Añadir a la rutina de interrupción la detección de si se cumple el plazo de algún proceso dormido. Si es así, debe cambiarle de estado (LISTO) y reajustar las listas correspondientes.

- Ofrecer a las aplicaciones un servicio de sincronización basado en semáforos. Se trata de un semáforo algo sui géneris. Aunque su modo de operación no se corresponde exactamente con el que típicamente poseen los semáforos proporcionados por los sistemas operativos reales y su utilidad práctica sea dudosa, su semántica está ideada para permitir practicar con distintas situaciones que aparecen frecuentemente en la programación de un sistema operativo. Las principales características de estos semáforos son las siguientes:

  — El número de semáforos disponibles es fijo (constante NUM_SEM). Un identificador de semáforo será un número que se encuentra en el intervalo de O a NUM_SEM —1.
  — Para usar un semáforo, el programa debe reservarlo y fijar el valor inicial de su contador (llamada iniciar_sem (unsigned int semid, unsigned int val_ini)). Con esta llamada, el proceso se hace dueño del semáforo hasta que lo libere (llamada liberar_sem (unsigned int semid)).
  — Una vez iniciado, cualquier proceso puede realizar las operaciones típicas de los semáforos: bajar (unsigned int semid) y subir (unsigned int semid). Estas operaciones tienen la semántica convencional ofrecida por los semáforos. Devolverán —l si hay un error (p. ej.: el identificador del semáforo no es válido o no ha sido iniciado) y O en caso contrario.
  — Si la llamada iniciar_sem encuentra que el semáforo solicitado ha sido reservado por otro proceso, bloqueará al proceso solicitante hasta que el actual dueño lo libere. Esta llamada devolverá —1 si el identificador del semáforo no es válido o el proceso ya tiene reservado el semáforo. Si no hay ningún error, devolverá un O.
  — La llamada liberar_sem debe desbloquear a todos los procesos que estaban bloquea dos en una operación bajar sobre el semáforo. Además. si hay Lino O Varios OCCSOS esperando reservar este mismo semáforo, se desbloqueará al primero de ellos. Esta

Digitalización Realizada con propósito académico. Sistemas Operativos

función devolverá —1 si el identificador del semáforo no es válido o el proceso no es el dueño del semáforo, y O en caso contrario.
— Cuando el proceso termina voluntaria o involuntariamente, se deberán liberar los semáforos que el proceso reservó y no liberó.

- Sustituir el algoritmo de planificación FIFO por round-robin (el tamaño de la rodaja es igual a la constante TICKS_POR_RODAJA). Con la inclusión de este algoritmo aparecen cambios de contexto involuntarios, lo que causa un gran impacto sobre los problemas de sincroniza ción dentro del sistema al poderse ejecutar varias llamadas de forma concurrente. Para sol ventar estos problemas, en la práctica se va a usar una solución similar a la utilizada en muchas implementaciones de UNIX: No permitir los cambios de contexto mientras el proceso está ejecutando en modo sistema. A grandes rasgos, la solución planteada debe cubrir los siguientes aspectos:

  — Al asignar el procesador a un proceso se le debe conceder una rodaja.
  — Al final de cada interrupción de reloj debe comprobarse si se ha cumplido la rodaja. Si es así y el proceso estaba ejecutando en modo usuario, se realiza un cambio de contexto pasándolo al final de la cola de listos. Si el proceso estaba en modo sistema, simplemente se anota en una variable que hay una replanificación pendiente y que, por tanto, el proceso debe abandonar el procesador justo antes de retornar a modo usuario.
  — Cada vez que termina el procesamiento de una llamada y antes de retornar al modo usuario, se debe comprobar si hay una replanificación pendiente y, si es así, llevarla a cabo. Observe que si el proceso que tiene pendiente la replanificación se bloquea como parte de la ejecución de una llamada no debe aplicársele la replanificación.

C.6.5. Material de apoyo

En esta sección se describe el árbol de archivos que corresponde con el entorno de desarrollo, minikernel.

- Makefile general del entorno. Invoca a los archivos Makefile de los subdirectorios subyacentes.
- boot. Este directorio está relacionado con la carga del sistema operativo. Contiene el pro grama de arranque del sistema operativo.
- kernel. Este directorio contiene todos los archivos necesarios para generar el sistema operativo:

  — Makefile. Permite compilar el sistema operativo.
  — minikernel. Archivo que contiene el ejecutable del sistema operativo.
  — apoyo. o. Archivo objeto que contiene las funciones de apoyo para la programación del sistema operativo.
  — proc. c. Archivo que contiene la funcionalidad del sistema operativo. En él debe incluirse la funcionalidad pedida en el enunciado.
  — include. Subdirectorio que contiene los archivos de cabecera usados por el sistema operativo:

- apoyo. h. Archivo que contiene los prototipos de las funciones de apoyo.
- const. . h. Archivo que contiene algunas constantes útiles.
- llamsis. h. Archivo que contiene los códigos numéricos asignados a cada llamada al sistema. En él deben incluirse las nuevas llamadas.

- proc. h. Contiene definiciones usadas por proc. c Como la del BCP.
- usuario. Este directorio contiene diversos programas de usuario:

— Makefile. Permite compilar los programas (le usuario.
— init. c. Primer programa que ejecuta el sistema operativo. Se puede modificar para que invoque los programas que se consideren oportunos.
— * . c. Programas de prueba.
— include. Subdirectorio que contiene los archivos de cabecera usados por los programas de usuario:

- servicios. h. Archivo que contiene los prototipos de las funciones que sirven (le interfaz a las llamadas al sistema. En él se debe incluir la interfaz a las nuevas llamadas.
- lib. Este directorio contiene los programas que permite generar la biblioteca que utilizan los programas de usuario. Su contenido es:

  — Makefile. Compila la biblioteca.
  — libserv.a. La biblioteca.
  — serv. c. Archivo que contiene la interfaz a los servicios del sistema operativo. En él se debe incluir la interfaz a las nuevas llamadas.
  — misc. o. Contiene otras funciones de biblioteca auxiliares.

C.6.6. Bibliografía

M. Beck. et al. Linux Kernel Internals, 2. edición, Addison—Wesley, 1997.
B. Kernighan y R. Pike. The UNIX Programming Environment, 2. edición. Prentice-Hall, 1984.
W. R. Stevens. UNIX Network Programming. Addison-Wesley, 1990.
—Advanced Programming in the UNIX Environment. Addison—Wesley. 1992.
A. S. Tanenbaum. Sistemas Operativos Modernos. Prentice—Hall. 1992.
P. S. Nang. An Introduction ¡o Berkeley Unix. Wadsworth, International Student Edition, 1988.

PRÁCTICA C.7. DISEÑO DE ARCHIVOS CON BANDAS

C.7.1. Objetivo de la práctica

Esta práctica tiene los siguientes objetivos principales:

- Permitir que el alumno entienda que se puede optimizar el almacenamiento en archivos haciendo uso de la concurrencia y del paralelismo.
- Enseñar al alumno a diseñar las estructuras que representan un archivo.
- Familiarizar al alumno con los mecanismos de programación de sistemas de archivos.

C.7.2. Entorno de desarrollo de la práctica

La práctica consiste en el diseño e implementación de archivos con bandas sobre el sistema de archivos de UNIX, siguiendo las directrices de este enunciado.
Para ello se proporciona al alumno una estructura de datos, denominada nodo_sfp, en la que se describe el archivo con bandas, y una descripción de cuál es la representación de un archivo con bandas.

Sistemas de archivos con bandas (FB)

Un sistema de archivos con bandas permite crear sistemas de archivos que ocupan varias particiones. Su estructura distribuye los bloques de datos de forma cíclica por los discos que con forman la partición lógica, repartiendo la carga de forma equitativa. Para optimizar la eficiencia del sistema de archivos se puede definir una unidad de almacenamiento en cada banda con un tamaño mayor que el del bloque del sistema de archivos. Esta unidad, denominada unidad de distribución (stripe unit), es la unidad de información que se escribe de forma consecutiva en cada banda. Este valor cambia de un sistema, e incluso archivo, a otro, siendo 64 KB el valor por defecto en Windows NT.

La Figura C.4 muestra la estructura de un sistema de archivos de bandas que ocupa cuatro particiones.

Además, este tipo de sistemas de archivos permite incrementar la fiabilidad del sistema de archivos insertando bloques de paridad con información redundante. De esa forma, si falta un dispositivo, se puede reconstruir la información mediante los bloques de los otros dispositivos y la información de paridad. Además, se puede hacer que la partición sea más tolerante a fallos distribuyendo también la información de la partición del sistema.

Un archivo con bandas queda definido por dos parámetros:

- **Unidad de distribución**. Es la unidad de información que se escribe de forma consecutiva en cada banda.
- **Número de subarchivos**. Número de archivos UNIX que componen el archivo con bandas.

En el caso de la práctica, tanto la unidad de distribución como el número de subarchivos son parámetros que se piden a la hora de crear el archivo con bandas.

Lógicamente, el archivo con bandas se ve como un único archivo por el usuario. Esto significa que el puntero de posición es único para el archivo con bandas, independientemente de que interna mente el sistema mantenga un puntero por cada subarchivo. Igualmente, cuando se pide informa ción de estado del archivo, o se hace un ls, la información que aparece es global para el archivo.

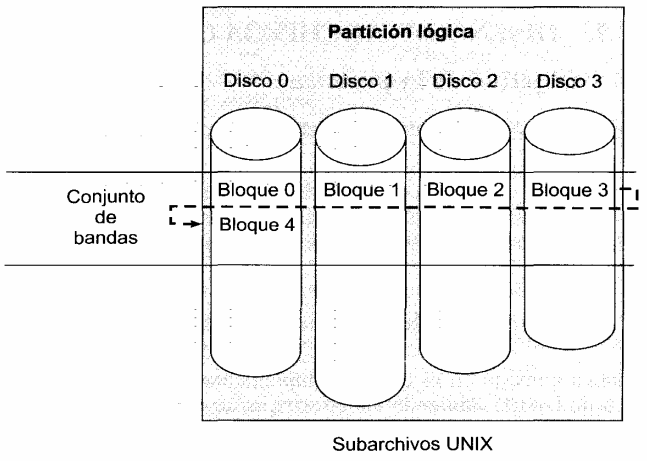


Figura C.4. *Sistema de archivos con bandas*

La forma de operar sobre un archivo con bandas es la siguiente. Cuando se va a leer del archivo, se accede al subarchivo donde indica el puntero de posición del FB. Sobre esa subunidad se lee lo que reste de la unidad de reparto y luego se pasa a los siguientes subarchivos. Evidentemente, si se leyera de varias subunidades se podría hacer en paralelo, pero teniendo en cuenta que la modificación del puntero de posición será la del puntero global.

Nodo_sfp de un archivo con bandas (nodo_sfp)

En un objeto archivo se puede almacenar información de tipo muy distinto: código fuente, pro gramas objeto, bibliotecas, programas ejecutables, texto ASCII, agrupaciones de registros, imágenes, sonidos, etc. Desde el punto de vista del sistema operativo, un archivo se caracteriza por tener una serie de atributos. Dichos atributos varían de tinos sistemas operativos a otros, pero todos ellos tienen una estructura interna que los describen. En nuestro caso, esa estructura se denominará nodo_sfp y se muestra en la Figura C.5.

El nodo_sfp usado contiene la siguiente información:

- Identificador único: a nivel interno, el sistema operativo no usa el nombre para identificar a los archivos, sino un identificador único fijado con criterios internos al sistema operativo. Este identificador suele ser un número y habitualmente es desconocido por los usuarios. En el caso de la práctica será un número entero que nunca decrece. Por tanto, hay que tener un contador para asignar el número de nodo_sfp.
- Identificador del proceso propietario y de su grupo. Son el pid y el gid del proceso creador.
- Protección: información de control de acceso que define quién puede hacer qué sobre el archivo, el dueño del archivo, su creador, etc.
- Nombre: identificador del archivo en formato comprensible para el usuario. Definido por su creador.
- Número de subarchivos que componen el archivo con bandas. Definido por su creador.
- Tamaño de la unidad de distribución, en bytes. Este dato indica la cantidad de datos consecutiva que se escribe en cada subarchivo.
- Tamaño del archivo: número de bytes en el archivo.
- Tiempos: de creación y de última actualización. Esta información es muy útil para gestionar, monitorizar y proteger los sistemas de archivos.
- Número de veces que el archivo ha sido abierto simultáneamente. Es el número de sesiones existentes sobre el archivo.

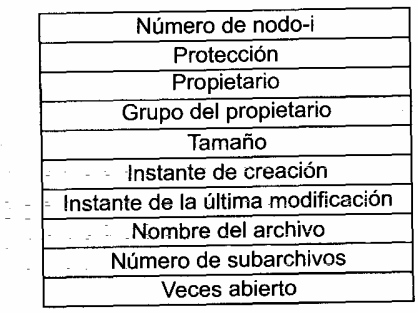


Figura C.5. *Nodo_sfp de los archivos con bandas*

La biblioteca debe mantener en memoria una tabla de nodos_sfp abiertos, denominada tnodos_sfp. En esa tabla estarán los nodo_sfp con sesiones pendientes, pero cada nodo_sfp estará únicamente una vez, independientemente de las sesiones que tenga abiertas. Las entradas de la tabla tendrán un nodo_sfp con un número identificador positivo mayor que cero si están ocupadas y con identificador — 1 si están libres.

Cuando se abre un archivo, hay que entrar en la tabla y buscar el primer hueco libre. Esa entrada es la ocupada y sobre ella se almacena el nodo_sfp leído del archivo de nodos . sfp que incluye las estructuras nodo_sfp de los archivos con bandas existentes en el sistema. El número de nodo_sfp no coincide con su posición en las filas de la tabla, por lo que cuando se busque un nodo_sfp habrá que buscar desde el principio. El valor del nodo_sfp se extrae del archivo de nodos. sfp. Si el archivo pedido ya estaba abierto, tendrá una entrada en esta tabla. Por tanto, lo único que se debería hacer es incrementar en 1 el número de veces que está abierto.

Cuando se cierre un archivo hay que decrementar el contador de sesiones abiertas y, si es cero, liberar la entrada de la tabla asociada al nodo_sfp. En el archivo nodos. s p también hay que actualizar el valor del nodo_sfp con el número de sesiones pendientes a cero.

Todas las modificaciones del nodo_sfp deben reflejarse en el archivo nodos. sfp, es decir, se debe usar la política de escritura inmediata (write-through).

Tabla de descriptores de archivos (tdf_sfp)

Para desacoplar a los procesos que usen la biblioteca de los descriptores internos de los archivos, nodo_sfp, y para permitir que un archivo se pueda abrir varias veces sin que haya conflictos con los accesos, se usará una tabla de descriptores de archivos como la que se muestra en la Figura C.6.

La tabla tiene cuatro entradas:

- Descriptor de archivo abierto. Es el que obtiene el proceso cuando ejecuta open. Este descriptor es un número entero entre 0 y MAX_FD. Teniendo en cuenta que 0, 1 y 2 están reservados para la STDIN, STDOUT y STDERR como en UNIX.
- Número de nodo sfp. Descriptor interno del archivo. Con este descriptor se pueden buscar los metadatos del archivo en la tabla tnodos_sfp.
- Puntero de posición del archivo. Para esa sesión del archivo abierto incluye la posición del archivo sobre la que se ejecutan las operaciones de E/S.
- Bandera de ocupado o libre. Está a O cuando está libre y a 1 cuando está ocupado.

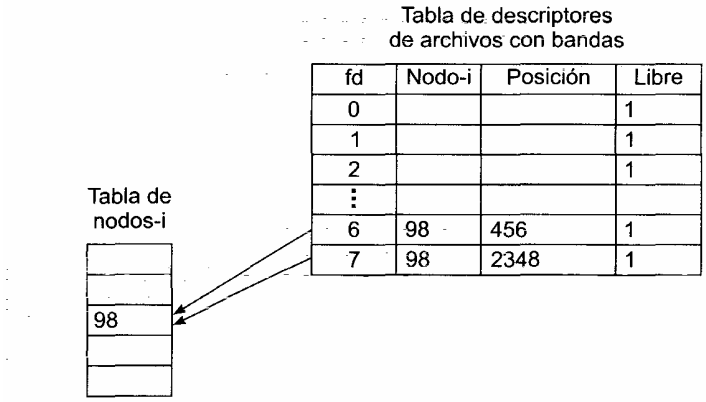


Figura C.6. *Tabla de descriptores de archivos.*

Cuando se abre un archivo hay que entrar en la tabla y buscar el primer descriptor libre. Esa entrada es la ocupada y su descriptor de archivo el devuelto. En la práctica, el número de fd coincidirá con su posición en las filas de la tabla, El valor del puntero de posición del archivo recién abierto es cero. El valor del nodo_sfp se extrae del archivo de nodos_sfp, denominado nodos.sfp, y se incluye en la columna correspondiente. Puesto que, como se ve en la figura, varias entradas de esta tabla pueden apuntar al mismo nodo E p, ya que se puede abrir un archivo repetidamente sin ningún problema.

Cuando se cierre un archivo hay que liberar la entrada del descriptor asociado al mismo y se borran los contenidos de dicha entrada.

Estructuración de la información

Con lo descrito hasta ahora, ya se puede explicar cómo y dónde debe quedar la información resultante de la ejecución de la práctica. Para simplificar la realización de la práctica, supondremos que todos los archivos con bandas se crean dentro de un subdirectorio del directorio donde se ejecuta la práctica, denominado sfp. Por tanto, todos los nombres de archivo con bandas deberían empezar por /sfp/.

Dentro del directorio . /sfp estarán:

- El archivo nodos. sfp. Todos los accesos a este archivo se hacen usando el tipo nodo_sfp definido.
- Los subarchivos resultantes de los archivos con bandas creados. Por tanto, cuando se ejecute un mandato ls. /sfp de UNIX sobre ese directorio, se verán entradas del tipo mi_fbandas_1 a mi_fbandas_n. No existirá ninguna entrada de archivo con bandas, ya que esta entelequia existe únicamente a nivel de biblioteca y de metadatos.

Para saber si las operaciones de archivos con bandas funcionan bien, bastará en muchos casos con monitorizar el directorio /sfp.

C.7.3. Descripción de la funcionalidad a desarrollar por el alumno

Para construir un servidor de archivos similar a un servidor real, se recomienda desarrollar esta práctica en varias etapas.

Primera etapa

En esta etapa, el alumno debe desarrollar una biblioteca de funciones para manipular los archivos con bandas. En esta sección se describe la funcionalidad de esta biblioteca, que debe incluir las siguientes funciones para manejar este tipo de archivos:

- **crear sfp**. Permite crear un archivo con bandas. Su prototipo es:

```
int crear_sfp (char * nombre, unsigned int permisos, unsigned int num_bandas, unsigned int
                u_distribucion);
```

Esta función crea el archivo, pero no lo abre. Para abrirlo es necesario hacer la llamada que se describe a continuación. El efecto de crear_sfp es la aparición de un archivo vacío con el nombre, protecciones, número de bandas y unidad de distribución definida en la llamada. Si el archivo ya existe, se trunca. dejándolo vacío de contenido. Además se crea el nodo_sfp y se incluye, relleno con los datos del archivo, en el archivo nodos. sfp.

En caso de que ya exista el archivo, se modifica su longitud a cero y se liberan sus datos. Si se intenta crear un archivo que está abierto, la llamada debe devolver un error. El código de error es -1.

Cuando se abra un archivo con bandas aparecerán en el subdirectorio. /sfp tantos subarchivos (archivos UNIX) como se indique en la creación. El nombre de dichos subarchivos se debe construir uniendo al nombre del archivo con bandas un entero correspondiente al número de subarchivo. Por ejemplo, para mi_fbandas, los subarchivos serían:

mi_fbandas_1 a mi_fbandas_n. Los permisos de acceso de los subarchivos deberán estar en consonancia con los solicitados para el archivo con bandas.

- **abrir_sfp** Permite abrir un archivo con bandas. Su prototipo es:

  int abrir_sfp (char * nombre, unsigned int modo);

  Esta llamada abre el archivo, y sus subarchivos asociados, y devuelve un identificador de archivo con el que trabajar. Además, lee el nodo_sfp en el archivo nodos. sfp y lo carga en la tabla de nodos_sfp en memoria, dentro de la primera entrada de la tabla que esté libre. Los modos de acceso posibles son FD_READ, FDWRITE y FD para lectura, escritura y lectura-escritura, respectivamente.

  Si el archivo ya estaba abierto, incrementa el contador de aperturas del nodo Para terminar, actualiza la tabla de descriptores de archivos con una nueva entrada para este archivo. En caso de que el archivo no exista, el modo de acceso solicitado no sea adecuado o no haya espacio en alguna tabla, la llamada devuelve un —1.
- **cerrar_sfp**. Permite cerrar un archivo con bandas. Su prototipo es:

  int cerrar_sfp (int fd);

  Esta llamada cierra el archivo con bandas y sus subarchivos asociados y decrementa el contador de aperturas del nodo_sfp. Además se libera su entrada de la tabla de descriptores de archivos. Si el contador de aperturas del archivo es igual a cero, se elimina el nodo_sfp del archivo de la tabla de nodo_sfp. Si el identificador de archivo pedido no existe, debe devolver un error.
- **borrar_sfp**. Borra un archivo con bandas. Su prototipo es:

  int borrar sfp (char * nombre);

  Esta llamada comprueba si el archivo está abierto, devolviendo error en este caso. Si el archivo no está abierto por ningún usuario, se borra el archivo con bandas y los subarchivos que lo componen y se elimina su nodo_sfp del archivo nodos. sfp.
- **leer_sfp.** Lee de un archivo con bandas a un buffer de memoria. El archivo debe estar abierto. Su prototipo es:

  int leer sfp (int fd, char * buffer, int tamanyo)

  Lee del archivo fd a un buffer la cantidad de datos tamanyo. Además adelanta el puntero de posición y lo modifica en la tabla de descriptores de archivo y en el nodo_sfp. Evidentemente, si el tamaño pedido es mayor que la unidad de distribución, hay que leer de los distintos subarchivos que contienen los datos. En caso de que el puntero de posición se encuentre en medio de una unidad de distribución, hay que leer en ese subarchivo la porción de unidad restante (módulo) y luego seguir a los otros.

Esta función devuelve la longitud de los datos leídos realmente, que será siempre menor o igual al tamaño pedido. Si el archivo no está abierto, debe ciar un error. Si se intenta leer más allá del fin de archivo, se deben devolver los datos existentes. l caso de que no haya ninguno, se devuelve cero.

- **escribir_sfp.** Escribe un buffer de memoria a un archivo con bandas. El archivo debe estar abierto. Su prototipo es:

int escribir_sfp (int fd, char * buffer, int tamanyo)

Escribe en el archivo fd desde un buffer la cantidad de datos tamanyo. Además adelanta el puntero de posición, lo actualiza en la tabla de descriptores de archivos y modifica su hora de última actualización y la de última modificación. Evidentemente, si el tamaño pedido es mayor que la unidad de distribución, hay que escribir en los distintos subarchivos que contienen los datos. En caso de que el puntero de posición se encuentre en medio de una unidad de distribución, hay que escribir en ese subarchivo la porción de unidad restante (módulo) y luego seguir a los otros.

Esta función devuelve la longitud de los datos escritos realmente, que será siempre menor o igual al tamaño pedido. Si el archivo no está abierto, debe dar un error. Si se intenta escribir más allá del fin de archivo, no hay ningún problema, simplemente se añaden los datos al archivo y se modifica su longitud. En caso de que no haya espacio en el dispositivo, se devuelve error.

- **posicionar_sfp**. Mueve el puntero de posición de un archivo con bandas. Su prototipo es:

int posicionar_sfp (int fd, off_t desplazamiento, int desde);

Esta operación se hace sobre la tabla de descriptores de archivo, donde está su puntero de posición. Con ella se puede cambiar dicho puntero mediante un desplazamiento con relación a desde. Los puntos que se pueden usar como referencia son el principio del archivo (SEEK_SET), el final (SEEK_END) y la posición actual (SEEK_CUR). Si se mueve más allá del fin de archivo, la llamada debe devolver un error. El archivo debe estar abierto.

- **estado_sfp**. Permite consultar la información de estado de un archivo. Su prototipo es:

int estado_sfp (char *nombre, struct nodo_sfp estado); \\

Se entiende por información de estado la contenida en el nodo_sfp del archivo. El archivo no tiene por qué estar abierto. Esta operación se hace sobre el nodo_sfp del archivo. Devuelve un —1 en caso de error.

Segunda etapa

Además de la biblioteca anterior, el alumno puede hacer los mandatos sfp_ls y sfp_cat para poder consultar la información de dichos archivos.

- **sfp_ls**. Permite ver los metadatos de los archivos con bandas que se han creado hasta el momento, es decir, todos los existentes en el directorio . /sfp/. La información debe re presentar al archivo con bandas y no a sus subarchivos. La información a mostrar por cada

archivo es: permisos, gid. uid, longitud, fecha de última escritura y nombre. Similar a un ls —1 de UNIX:

—rw-r—--r—- 1 jcarrete 937 Sep 23 16:05 Makefile.

El mandato admite dos formatos:

— sfp_ls . /sfp/nombre-archivo para ver información de un archivo específico.
— sfp_ls ./sfp para ver todos los archivos del directorio.

- sfp_cat. Permite ver los datos de un archivo con bandas existente en el directorio. /sfp/ por la salida estándar. El formato del mandato es sfp_cat - /sfp/nombre-archivo. Es similar al mandato cat de UNIX.

Tercera etapa

En esta parte se pide al alumno que modifique la implementación de la biblioteca y de los programas de prueba anteriores para que se comporten como un sistema cliente-servidor. Para ello debe existir un proceso servidor de archivos, con el que se comunican los clientes a través de sockets. El servidor debe ser concurrente.

C.7.4. Llamadas al sistema operativo

Para la realización de esta práctica sólo se deben usar llamadas, al sistema de archivos, tales como read, write, close, 1seek, o ioctl.

Para obtener información más detallada de estas llamadas se recomienda mirar la bibliografía de la práctica y los manuales de la máquina para los mandatos de archivos (p. ej.: man lseek).

C.7.5. Recomendaciones generales al alumno

- Se recomienda hacer un programa bien estructurado, evitando la linealidad. Se puede pro gramar mucho código en funciones.
- Se debería hacer un control exhaustivo de errores, tanto de ejecución en las llamadas al sistema y funciones como de la distribución de datos en los subarchivos.
- En caso de que tenga problemas durante el desarrollo de la práctica, intente solventarlos antes de hacer pruebas defectuosas. Use el depurador.
- Se recomienda programar funciones de utilidad para ver los contenidos de las tablas de nodo_sfp, de descriptores de archivos y del archivo de nodos. sfp.
- Existen varias pruebas que el alumno puede realizar para ver si la funcionalidad de la práctica es correcta. A continuación se sugieren algunas:
  — Después de crear un archivo con bandas se puede verificar que las entradas de los subarchivos están en. /sfp con longitud cero. Después de borrarlo, dichas entradas deberán haber desaparecido. Igualmente, cuando se cree un archivo, debe aparecer un nodo_sfp nuevo en el archivo. /sfp/nodos.sfp. En caso de horrado, la entrada deberá desaparecer. Si se crea un archivo que ya existe, compruebe que su longitud se trunca a cero.

- Después de abrir un archivo, la entrada de su nodo_sfp deberá estar en la tabla de nodos_sfp de memoria. Si ya lo estaba, su contador de aperturas se habrá incrementado en 1. después de cerrarlo, la entrada deberá desaparecer. Si el contador era mayor que 1 se decrementará en 1
- Se recomienda al alumno implementar dos rutinas. read-verify y write—verify, para comprobar que los datos se leen y escriben bien en los archivos. Como indica su nombre, leen y escriben datos y después verifican que los datos manipulados están correctos en el archivo con bandas.
- Se aconseja escribir en medio de la unidad de distribución y ver si los datos se distribuyen bien. No se limite a escribir siempre en bloques que coinciden con la unidad de distribución.
- Tras las operaciones de creación y de escritura de un archivo se debe comprobar que las fechas correspondientes del nodo-i se habrán modificado.
- Se recomienda escribir un archivo con longitud y datos conocidos. Hacer una operación de posicionar y leer para ver si se está en el lugar adecuado. A continuación se debería intentar ir más allá del fin de archivo para ver si se obtiene error.
- Se aconseja escribir un archivo con longitud y datos conocidos y ejecutar un mandato sfp_cat para ver si se obtienen los datos escritos previamente.

C.7.6. Bibliografía

A. Silberschatz y P. B. Galvin. Operating Systems Concepts. 5. edición. Addison-Wesley. 1999.
D. A. Solomon. Inside Windows NT. Microsoft Press, 1998.
W. R. Stevens. Advanced Programming in the UNIX Environment. Addison-Wesley. 1992.
A. S. Tanenbaum. Operating Systems: Design and Implementation. 2. edición. Prentice-Hall, 1987.
—Modern Operating Systems. Prentice-Hall, 1991.
P. S. Wang. An Introduction to Berkeley Unix. Wadsworth. International Student Edition. 1988.

Bibliografía

Capítulo 1. Conceptos arquitectónicos de la computadora

[Miguel, 19981 P. de Miguel. Fundamentos de los Computadores. Ed. Paraninfo, 1998.
[1994] D. Patterson y J. Hennessy. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann, 1994.
[19961 W. Stallings. Computer Organization and Architecture, 4• edición, Prentice-Hall, 1996
[1999] A. S. Tanenbaum. Structured Computer Organization, 4• edición, Prentice-Hall, 1999

Capítulo 2. Introducción a los sistemas operativos

[1986] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian y M. Young. Mach: A New Kernel Foundation for UNIX Development. Proceedings Summer 1986. USENIX Conference, págs. 93-112, 1986
[19861 M. J. Bach. The Design of the Unix Operating Systems. Prentice-Hall, 1986.
[19981 M. Beck, N. Boheme, M. Dziadzka et al. Linux Kernel Internals, 2. edición, Addison-Wesley, 1988.
ICorbato, 19621 F. Corbato, M. Merwin-Dagget y R. Dealey. A Experimental Time-Sharing System. Proced dings of the 1962 Spring Joint Computer Conference, 1962.
[1997] C. Crowley. Operating Systems, A Design-Oriented Approach. Irwin-McGraw-Hill, 1997
[1994] H. M. Deitel y M. S. Kogan. The Design of OS/2. Prentice-Hall, 1994
[19681 E. W. Disjkstra. The Structure of THE Multiprogramming System. Communications of the ACM, vol. 11, págs. 341-346, mayo 1968.
[2000] L. D. Galli. Distributed Operating Systems: Concepts and Practice. Prentice-Hall, 2000.
[1998] J. M. Hart. Win32 System Programming. Addison-Wesley, 1998.
[1996] Information technology. Portable Operating System Interfac (POSIX). System Application Pro gram Interface (API). IEEE Computer Society, 1996.
[1996] M. McKusick, K. Bostic, M. Karels y J. Quaterman. The Design and Implementation of the 4.4 BSD UNIX Operating System. Addison-Wesley, 1996
[1966] G. H. Mealey, B. 1. Witt y W. A. Clark. The Strucrural Structure of 05/360. IBM System Journal, vol. 5, núm 1, 1966.
[1992] M. Milenkovic. Operating Systems: Concepts and Design. McGraw-Hill, 1992
[19901 S. J. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse y H. Van Staveren. Amoeba: A Distributed Operating Systems for the]990s. IEEE Computer, vol. 23, págs. 44-53 mayo, 1990.

[Organik.1972] E. 1. Organick.The Multics System: An Examination of its Structure. MIT. Press, Cambridge, MA, 1972.
[19961 R. Otte, P. Patrick y M. Roy. Understanding CORBA. Prentice-Hall, 1996.
IIQNX, 1997] QNX Software Systems Ltd. QNX Operating System. System Architecture, 1997.
[1986] J. F. Ready. VRTX: A Real-Time Operating Systemfor Embedded Microprocessor Applications. MICRO, vol. 6, núm. 4, págs. 8-17, agosto 1986.
[19881 M. Roizer, V. Abroxximov, F. Armand, 1. Boule, M. Gien, M. Guillemont, F. Hen-mann, C. Kaiser, P. Leonard, 5. Langlois y W. Neuhauser. Chorus Distributed Operating System. Computing Systems, vol. 1, págs. 305-379, octubre 1988
[19921 W. Rosenberry, D. Kenney y G. Fisher. Understanding DCE. O'Reilly, 1992
[1999] W. Rubin, M. Brain y R. Rubin. Understanding DCOM. Prentice-Hall, 1999.
[1990] 5. Samson. MVS Performance Management. McGraw-Hill, 1990.
[19991 A. Silbertchatz y P. Galvin. Operating Systems Concepts, 5• edición, Addison-Wesley, 1999.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.
[1998] W. Stallings. Operating Systems, Internais and Design Principies, 3.' edición, Prentice-Hall, 1998.
[19921 A. 5. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[1995] A. 5. Tanenbaum. Distributed Operating Systems. Prentice-Hall, 1995.
[1997] A. 5. Tanenbaum y A. 5. Woodhull. Operating Systems: Design and Impiementation, 2. edición, Prentice-Hall, 1997.

Capítulo 3. Procesos

Crow1ey, 1997] C. Crowley. Operating Systems, A Design-Oriented Approach. Irwin, 1997
[19961 Information technology. Portabie Operating System Interfac (POSIX). System Appiication Pro gram Interface (API). IEEE Computer Society, 1996.
[1992] M. Milenkovic. Operating Systems: Concepts and Design. McGraw-Hill, 1992
[19991 A. Silberstchatz y P. Galvin. Operating Systems Concepts, 5 edición, Addison-Wesley, 1999.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.
[19981 W. Stallings. Operar/ng Systems, Internais and Design Principies, 3• edición, Prentice-Hall, 1998.
[19951 A. S. Tanenbaum. Distributed Operating Systems. Prentice-Hall, 1995.

Capítulo 4. Gestión de memoria

[1986] M. J. Bach. The Design of the UNIX Operating System. Prentice-Hall, 1986.
[19961 M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus y D. Verwomer. Linux Kernel Inter nais. Addison-Wesley, 1996.
[1969] L. A. Belady et al. «An Anomaly in Space-Time Characteristics of Certain Programms Run ning in a Paging Machine», Communications of the ACM, vol. 12, núm. 6, 1969.
[1993] H. Custer. Inside Windows/NT. Microsoft Press, 1993.
[1987] R. A. Gingeil et al. «Shared Libraries in SunOS», Summer Conference Proceedings. USENIX Association, 1987.
[19971 J. M. Hart. Win32 System Programming. Addison-Wesley, 1997.
[1965] K. C. Knowlton. «A Fast Storage Allocator». Communications of ihe ACM, vol. 8, núm. 10, 1965.
[1987] M. Maekawa et al. Operating Systems: Advanced Concepts. Benjamin-Cummings, 1987. [1996] M. K. McKusick, K. Bostic, M. J. Karels y J. 5. Quaterman. The Design and Implementa tion of the 4.4 BSD Operating System. Addison-Wesley, 1996.

Digitalización realizada con propósito académico

[19981 A. Silberschatz y P. B. Galvin. Operating System Concepts, 5. edición, Addison-Wesley, 1998.
[19981 W. Stallings, Operating Systems, 3 edición, Prentice-Hall, 1998.
[19921 W. Stevens. Advanced Programming in the UNIX Environment. Addison-Wesley, 1992.
[19921 A. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[1997] A. Tanenbaum y A. Woodhull. Operating Systems Design and Impiementation, 2. edi ción, Prentice-Hall, 1997.

Capítulo 5. Comunicación y sincronización de procesos

[19861 M. J. Bach. The Design of the Unix Operating System. Prentice-Hall 1986.
[19971 M. Beck et al. Linux Kernel Internais, 2. edición. Addison-Wesley, 1997.
[Ari, 19901 Ben Ari. Principies of Concurrent and Distributed Programming. Prentice-Hall, 1990
[1965] E. W. Dijkstra. Cooperating Sequential Processes. Technical Report EWD- 123, Technologi cal University, Eindhoven, Holanda, 1965. Reimpreso en [1968], págs. 43-112.
[1974j C. A. R. Hoare. Monitors: An Operating Systems Structuring Concept. Communications of the ACM, vol. 17, núm 10, págs. 549-557, octubre 1974.
[19801 B. Lampson y D. Redeil. Experience with Processes and Monitors in Mesa. Communica tions of the ACM, febrero 1980.
[1996] M. K. McKusick, K. Bostic, M. J. Karels y J. S. Quaterman. The Design and impiementa tion of the 4.4 BSD Operating System. Addison-Wesley, 1996
[19991 A. Silberschatz y J. Peterson. Operating Systems Concepts, 5• edición, Addison-Wesley, 1999.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998
[1998] W. Stallings. Operating Systems, Internais and Design Principies, edición, Prentice-Hall, 1998.

Capítulo 6. Interbloqueos

[1986] M. J. Bach. The Design of the Unix Operating System. Prentice-Hall, 1986.
[1971] E. G. Coffman, M. J. Elphick y A. Shoshani. System Deadlocks. Computing Surveys, vol. 3, núm. 2, págs. 67-78, junio 1971.
[19651 E. W. Dijkstra. Cooperating Sequentiai Processes. Technical Report EWD- 123, Technologi cal University, Eindhoven, Holanda, 1965. Reimpreso en [1968], págs. 43-112.
[1969] A. N. Habermann. Prevention of System Deadlocks. Communications of the ACM, vol. 12, núm. 7, págs. 373-377, julio 1969.
[1968] J. W. Havender. Avoiding Deadlocks in Multitasking Systems. IBM Systems Journal, vol. 7, núm. 12, págs. 74-84, 1968.
[19721 R. C. Holt. Sorne Deadiock Properties of Computer Systems. Computing Surveys, vol. 4, núm. 3, págs. 179-196, septiembre 1972.
[1973] J. H. Howard. Mixed Soiutionsfor the Deadiock Problem. Communications of the ACM, vol. 16, núm. 7, págs. 427-430, julio 1973.
[19871 M. Maekawa, A. E. Oldehoeft y R. R. Oldehoeft. Operating Systems, Advanced Concepts. The BenjaminlCummings Publishing Company, 1987.
[19961 M. K. McKusick, K. Bostic, M. J. Karels y J. S. Quaterman. The Design and Impiementa tion of the 4.4 BSD Operating System. Addison-Wesley, 1996.
[19791 G. Newton. Deadiock Prevention, Detection and Resolution: An Annotated Bibiiography. Operating Systems Review, vol. 13, núm. 2, págs. 33-44, abril 1979.
[1999] A. Silberschatz y P. B. Galvin. Operating Systems Concepts, 5. edición, Addison-Wes ley, 1999.
[1998] W. Stallings. Operating Systems, Internais and Design Principies, 3. edición, Prentice-Hall, 1998.

Digitalización realizada con propósito académico

[1992] A. S. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[1983] D. Zobel. The DeadlockProblem: A Class Bibliography. Operating Systems Review, vol. 17, núm. 4, págs. 6-16, octubre 1983.

Capítulo 7. Entrada/salida

[1995] S. Akyurek y K. Salem. Adaptive Biock Rearrangement. ACM Transactions on Computer Systems, 13(2), págs. 89-121, mayo 1995.
[1986] M. J. Bach. The Design of the UNIX Operating System. Prentice-Hall, 1986.
[1996] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus y D. Verworner. Linux Kernel Ínter nais. Addison-Wesley, 1996.
[1999] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus y D. Verwomer. Linux Kernel ínter nais, 2. edición, Addison-Wesley, 1996.
[1993] P. Biswas, K. K. Ramakrishnan y D. Towsley. «Trace Driven Analysis of Write Caching Policies for Disks». Proceedings of the 1993 ACM SIGMETRICS Conference Qn Measurement, and Modeling of Computer Systems, págs. 13-23, mayo 10-14, 1993.
[20001 J. Carretero, J. Fernández y F. García. «Enhancing Parallel Multimedia Servers through New Hierarchical Disk Scheduling Algorithms». VECPAR '2000, 4th International Meeting on Vector and Parallel Processing, Oporto, junio 2000.
[1995] P. M. Chen y E. K. Lee. «Striping in a RAID Level 5 Disk Array». Proceedings of the 1995 ACM SIGMETRICS Joint International Conference Qn Measurement and Modeling of Computer Systems, págs. 136-145, mayo 15-19, 1995.
[1999] R. A. Come. HPSS Tutorial. Technical Report, IBM Global Government Industry. http://www. sdsc.edu/hpss, 1999.
[1995] W. Davy. «Method for Eliminating File Fragmentation and Reducing Average, Seek Times in a Magnetic Disk Media Environment». USENIX, 1995.
[1999] E. N. Dekker y J. M. Newcomer. «Developing Windows NTDevice Drivers: A Programmer's Handbook». Addison-Wesley, 1999.
[1994] B. Goodheart y J. Cox. The Magic Garden Explained: The internais of UNIX System V Release 4, an Open Systems Design, págs. 664, Prentice-Hall, 1994.
{Hart, 1997] J. M. Hart. Win32 System Programming. Addison-Wesley, 1997.
[1988] IEEE. IEEE Standard Portable Operating System Interfacefor Computer Environments, IEEE Computer Society Press, 1988.
[1996] M. K. McKusick, K. Bostic, M. J. Karels y J. S. Quaterman. The Design and Implementa tion of the 4.4 BSD Operating System. Addison-Wesley, 1996.
[1997] R. Nagar. Windows NT File System Internals, 774 pág., O'Reilly & Associates Inc., 1997.
[2000] Seagate. Barracuda ATA II Family: Product Manual. Seagate, 2000.
[1998] A. Silberschatz y P. B. Galvin. Operating System Concepts, a edición, Addison-Wesley, 1998.
[19851 A. J. Smith. Disk Cache -Miss Ratio Analysis and Design Considerations. ACM Transactions on Computer Systems, vol. 3, núm. 3, págs. 161-203, agosto 1985.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.
[1998] W. Stallings. Operating Systems, edición, Prentice-Hall, 1998.
[1992] W. Stevens. Advanced Programming in the UNIX Environment. Addison-Wesley, 1992.
[1992] A. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[1997] A. Tanenbaum y A. Woodhull. Operating Systems Design and Implementation, 2. edi ción, Prentice-Hall, 1997.
[1992] G. Weikum y P. Zabback. «Tuning of Stnping Units in Disk-Array-Based File Systems». Proceedings of the 2nd International Workshop cm Research Issues in Data Engineering, págs. 80-87, febrero 2-3, 1992.
[1995] J. Wilkes, R. Golding, C. Staelin y T. Sullivan. «The HP AutoRAID Hierarchical Storage System». Proceedings of the Fifteenth ACM Symposiuin Qn Operating Systems Principies, págs. 96-108, diciembre 3-6, 1995.

Digitalización realizada con propósito académico

Capítulo 8. Gestión de archivos y directorios

[19731 D. H. Abernathy et al. Survey of Design Goalsfor Operating Systems, Operating Systems. Review, vol. 7, núm. 2, págs. 29-48, abril 1973; OSR, vol. 7, núm. 3, págs. 19-34, julio 1973; OSR, vol. 8, núm. 1, págs. 25-35, enero 1974.
[19951 5. Akyurek y K. Salem. Adaptive Biock Rearrangement. ACM Transactions on Computer Systems, 13(2), págs. 89-121, mayo 1995.
[19811 D. A. Anderson. Operating Systems. COMPUTER, vol. 14, núm. 6, págs. 69-82, junio 1981.
[19961 M. Andrews. C++ Windows NT Programming, 735 páginas, M&T Press, 1996.
[1986] J. A. Anyanwu y L. F. Marshall. A Crash Resistant UNIX File System, Software - Practice and experience, vol. 16, núm. 1, págs. 107-118, febrero 1986.
[1990J American Telephone y Telegraph Company. UNIX System V release 4: programmer's guide:
POSIX conformance. Prentice-Hall, 1990.
[1986] M. J. Bach. The Design of the UNIX Operating System. Prentice-Hall, 1986.
Bailey, 1981] K. A. Bailey etal. UserDefined Files. Operating Systems Review, vol. 15, núm. 4, págs. 75-81, octubre 1981.
[1996] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus y D. Verwomer. Linux Kernel ínter nais. Addison-Wesley, 1996.
[1993] P. Biswas, K. K. Ramakrishnan y D. Towsley. «Trace Driven Analysis of Wnte Caching Policies for Disks». Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement, and Modeling of Computer Systems, págs. 13-23, mayo 10-14, 1993.
[1995] T. Blackwell, J. Harris y M. Seltzer. «Heuristic Cleaning Algorithms in Log-Structured File Systems». Proceedings of the USENIX 1995 Technical Conference, págs. 277-288, enero 16-20, 1995.
[1995] S. H. Bokhari. The Linux Operating System, Computer, vol. 28, núm. 8, págs. 74-79, agosto 1995.
[19891 A. Braunstein, M. Riley y J. Wilkes. «Improving the Efficiency of UNIX File Buffer Ca ches». Proceedings of the Twelfth ACM Symposium on Operating Systems, Principies, págs. 71-82, 1989.
[1989] R. Brent. Efflcient implementation of the First-Fit Strategy for Dynamic Storage Allocation. ACM Transactions on Programming Languages and Sytems, julio 1989.
[19861 0. P. Brereton, Management of Replicated Files in a UNIX Environment, Software - Practice and Experience, vol. 16, núm. 4, págs. 771-780, agosto 1986.
[1969] N. Chapin. Common File Organization Techniques Compared. FJCC, vol. 35, págs. 413-422, 1969.
[1995] P. M. Chen y E. K. Lee. «Striping in a RAID Level 5 Disk Array». Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, págs. 136-145, mayo 15-19, 1995.
[1988] K. Christian. The UNIX Operating System. Wiley-Interscience, 1988.
[19851 D. Davcev y W. A. Burkhard. «Consistency and Recovery Control for Replicated Files». Proceedings of the Tenth Sysposium on Operating Systems Principies, vol. 19, núm. 5, págs. 87-96, diciembre 1985.
[1995] W. Davy. «Method for Eliminating File Fragmentation and Reducing Average, Seek Times in a Magnetic Disk Media Environment». USENIX, 1995.
[1989] F. Douglis y J. Ousterhout. «Log-Structured File Systems», Proceedings of the 34th COMP-CON Conference, págs. 124-129, febrero 1989.
[1987] M. J. Folk y B. Zoellick. File Structures. Addison-Wesley, 1987.
[1985] E. Foxley. UNIX for super-users. International Computer Science Series, págs. 213, Addison- Wesley, 1985.
[1996] A. Frish. Essential UNIX Administration, 2. edición, O'Reilly, 1996.
[1994] B. Goodheart y J. Cox. The Magic Garden Explained: The Internais of UNIX System V Release 4, an Open Systems Design, págs. 664, Prentice-Hall, 1994.
[1991] A. Goscinski. Distributed Operating Systems: The Logical Design. Addison-Wesley, 1991.
[198 lj J. N. Gray. «The Transaction Concept: Virtues and Limitations». Proceedings of the internatio nal Conference on Very Large Data Bases, págs. 144-154, 1981.

Digitalización realizada con propósito académico

[1986] D. Grosshans. File Systems Design and Implementation. Prentice-Hall, 1986.
[19971 J. M. Hart. Win32 System Programming. Addison-Wesley, 1997.
[1992] N. Horspool. The Berkeley UNIX Environment. Prentice-Hall, 1992.
[19881 IEEE. IEEE Standard Portable Operating System Interfacefor Computer Environments, IEEE Computer Society Press, 1988.
[19831 S. H. Kaisler. The Design of Operating Systems for Small Computer Systems. John Wiley & Sons, Nueva York, 1983.
[1988] L. J. Kenah, R. E. Goldenberg y 5. F. Bate. VAX/VMS Internals and Data Structures, Digital Press, 1988.
[19781 B. Kemighan y D. Ritchie. The C Programming Language. Prentice-Hall, 1978.
[1984] B. Kernighan y R. Pike. The UNIX Programming Environment. Prentice-Hall, 1984.
[1997] P. D. L. Koch. Disk File Allocation Based on the Buddy System. ACM Transactions on Computer Systems, vol. 5, núm. 4, págs. 352-370, noviembre 1987.
[1994] 0. Krieger, M. Stumm y Ron Unrau. The Al/oc Stream Faciliiy: A Redesign of Application Level Stream I/O. Computer, vol. 27, núm. 3, págs. 75-82, marzo 1994.
[1990] E. Levy y A. Silberschatz. Distributed File Systems, ACM Computer Systems, vol. 22, núm. 4, págs. 321-374, diciembre 1990.
[1973] K. R. London. Techniques for Direct Access. Auerbach Publishers, 1973.
[1984] M. K. McKusick, W. N. Joy, 5. J. Leffler y R. S. Fabry. A Fast File Systemfor UNIX, ACM Transactions on Computer Systems, vol. 2, núm. 3, págs. 181-197, agosto 1984.
[1996] M. K. McKusick, K. Bostic, M. J. Karels y J. 5. Quaterman. The Design and Impiementa tion of the 4.4 BSD Operating System. Addison-Wesley, 1996.
[1997] R. Nagar. Windows NT File System Internals, 774 pág., O'Reilly & Associates Inc., 1997.
[1988] P. Norton y R. Wilton. The New Peter Norton Programmer's Guidefor the IBM PC & PS/2. Microsoft Press, 1988.
[1985] R. R. Oldehoeft y S. J. Allan. Adaptive Exact-fit Storage Management. Communications of the ACM, vol. 28, núm. 5, págs. 506-511, mayo 1985.
[1972] E. 1. Organick. The Multics System: An Examination of its Structure. MIT Press, 1972.
[1989] J. K. Ousterhout y F. Douglis. Beating the I/O Bottieneck: A case for Log Structured File Systems. ACM Opearing Systems Review, vol. 23, núm. 1, págs. 11-28, enero 1989.
[1985] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. D. Kupfer y J. G. Thompson. «A Trace-Driven Analysis of the UNIX 4.2 BSD File System». Proceedings of the Tenth ACM Sympo sium on Operating Systems Principies, págs. 15-24, diciembre 1-4,, 1985.
[1983] T. Patterson. An Inside Look at MS-DOS. BYTE, vol. 8, núm. 6, págs. 230-252, junio 1983.
[1989] J. Pinkert y L. Wear. Operating Systems: Concepts, Policies y Mechanisms. Prentice-Hall, 1989.
[1974] D. M. Ritchie y K. Thompson. The Unix Timesharing System. Communications of the Associa tion for Computing Machinery, vol. 17, núm. 7, págs. 365-375, julio 1974.
[1985] M. J. Rockind. Advanced UNIX Programming. Prentice-Hall, 1985.
[1985] M. D. Schroeder, D. K. Gifford y R. M. Needham. «A Caching File System for a Program mer' s Worsktation». Proceedings of the Tenth Symposium on Operating Systems Principies, págs. 25-32, diciembre 1985.
[1998] A. Silberschatz y P. B. Galvin. Operating System Concepts, a edición, Addison-Wesley, 1998.
[1982] A. J. Smith. Cache Memories. ACM Computing Surveys, vol. 14, núm. 5, págs. 473-530, septiembre 1982.
[1985] A. J. Smith. Disk Cache-Miss Ratio Analysis and Design Considerations. ACM Transactions on Computer Systems, vol. 3, núm. 3, págs. 161-203, agosto 1985.
[1994] K. Smith y M. Seltzer. File Layout and Fi/e System Performance. Technical Report, Harvard University, Number TR-35-94, 1994.
[19981 D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.
[1988] C. Staelin. File Access Patterns. Technical Report, Department of Computer Science, Prince ton University, Number CD-TR-179-88, septiembre 1988.

Digitalización realizada con propósito académico

[19981 W. Stallings. Operating Systems, 3• edición. Prentice-Hall, 1998.
tTanenbaum, 19871 A. S. Tanenbaum. Operating Systems: Design and Implementation. Prentice-Hall, Engle wood Cliffs, Ni, 1987.
Tanenbaum, 19921 A. S. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[19971 A. S. Tanenbaum y A. Woodhull. Operating Systems Design and Implementation, 2. edi ción. Prentice-Hall, 1997.
Walker, 19831 B. Walker et al. The LOCUS Distributed Operating System. SOSP-9, págs. 49-70, octubre 1983.
Weikum, 19921 G. Weikum y P. Zabback. «Tuning of Striping Units in Disk-Array-Based File Systems». Proceedings of the 2nd International Workshop on Research Issues in Data Engineering, págs. 80-87, febrero 2-3, 1992.
[19911 B. Welch. «The File System Belongs in the Kemel». Proceedings of the Second USENIX Mach Symposium, págs. 233-250, noviembre 20-22, 1991.
[19951 J. Wilkes, R. Golding, C. Staelin y T. Sullivan. «The HP AutoRAID Hierarchical Storage System». Proceedings of the Fifleenth ACM Symposium on Operating Systems Principies, págs. 96-108, diciembre 3-6, 1995.
[19801 R. Zaks. The CP/M Handbook with MP/M. SYBEX Inc. Berkeley, California, 1980.

Capítulo 9. Seguridad y protección

[1994] M. Abadi y R. Needham. «Prudent Engineenng Practice for Cryptographic Protocols». Proc, of the IEEE Symp. On Security and Privacy, págs. 122-136, IEEE Computer Society Press, 1994.
[1973j D. 1-1. Abernathy et al. Survey of Design Goalsfor Operating Systems. Operating Systems Review, vol. 7, núm. 2, págs. 29-48, abril 1973; OSR, vol. 7, núm. 3, págs. 19-34, julio 1973; OSR, vol. 8, núm. 1, págs. 25-35, enero 1974.
[1993] D. Arnold. UNIX Security. A Practical Tutorial. McGraw-Hill, 1993.
[et al., 19761 C. R. Attanasion et al. Penetrating an Operating System: A Study of VM/370 Inte grity, 113M Systems Journal, vol. 15, núm. 1, págs. 102-116, 1976.
[1986] M. J. Bach. The Design of the UNIX System. Prentice-Hall, 1986.
[1994] M. Blaze. «Key Management in an Encrypting File System». Proceedings of the USENIX Summer 1994 Technical Conference, págs. 27-35, junio 6-10, 1994.
[19831 D. Bell. «Secure Computer Systems: a Retrospective». Proc. of the IEEE Symp. on Security and Privacy, págs. 161-162, IEEE Computer Society Press, 1993.
[1991] S. Bellovin y M. Merrit. «Limitations of the Kerberos Authentication System». Proc, of the Usenix Conference, págs. 253-267, invierno 1991.
[1995] A. Berman, y. Bourassa y E. Selberg. «TRON: Process-Specific File Protection for the UNIX Operating System». Proceedings of the USENIX 1995 Technical Conference, págs. 165-175, enero 16- 20, 1995.
[19891 D. Brewer y M. Nash. «The Chinese Wall Security Policy». Proc, of the JEEE Symp. on Securily and Privacy, págs. 206-214, IEEE Computer Society Press, 1989.
[1994] Common Criteria Editorial Board. Common Criteriafor Information Technology Security Eva luations, versión 0,6, abril 1994.
[1987] D. Clark y D. Wilson. «A Comparison of Conimercial and Military Computer Security Policies». Proc, of the IEEE Symp. on Securily and Privacy, págs. 184-194, IEEE Computer Society Press, 1987.
[l983a] IEEE Computer. Special issue: Data Security in Computer Networks, IEEE Computer, vol. 16, núm. 2, febrero 1983.
[l983b1 IEEE Computer. Speciai issue: Computer Security Technology, IEEE Computer, vol. 16, núm. 7, julio 1983.
[19761 Computer Surveys. Special issue: Reliable Software II: Fault-Tolerant Software. Computer Surveys, vol. 8, núm. 4, diciembre 1976.
[1992] D. Curry. UNIX System Security. A Guidefor Users and System Administrators. Addison-Wes ley, 1992.

Digitalización realizada con propósito académico

[1993] B. DeDecker. Unix Securily and Kerberos. Lecture Notes in Computer Science, vol. 741, págs. 257-274, 1993.
[1984] H. M. Deitel. An Introduction to Operating Systems. Addison-Wesley, 1984.
[19791 L. F. DeLashmutt. Steps toward a pro vably secure operating system. COMPCON, págs. 40-43, verano 1979.
[1982] D. Denning. Cryptography and Data Security. Addison-Wesley, 1982.
[1990] D. Denning. Computers Under Attack. Addison-Wesley, 1990.
[19961 D. Denning y D. Branstad. A taxonomy of Key Escrow Encryption Systems. Communication of the ACM, vol. 39 núm. 3, págs. 34-40, marzo 1996.
[1990] D. Farmer y E. Spafford. «The COPS Security Checker System». Proc, of the Usenix Summer Conference, págs. 165-170, 1990.
[1990] R. Farrow. UNIX System Security. How to Protect your Data and Prevent Intruders. Addison- Wesley, 1990.
[1989] P. Fites et al. Control and Securily of Computer Information Systems. Computer Science Press, 1989.
[1982] C. Foster. Cryptoanalysis for Microcomputers. Hayden Books, 1982.
[1991] 5. Garfinkel y G. Spafford. Practical UNIX Securily. O'Reilly & Associates, Inc., 1991.
[1997] S. Garfinkel y G. Spafford. Web Securily & Commerce. O'Reilly & Associates, Inc., 1997.
[1988] M. Gasser. Building a Secure System. Van Nostrand Reinhoid, 1988.
[1991] A. Goscinski. Distributed Operating Systems. Addison-Wesley Publishing Company, 1991.
[1968] R. Graham. Protection in an Information Processing Utility. Communications of the ACM, vol. 11, núm. 5, págs. 365-369, mayo 1968.
[1984] F. T. Grampp y R. H. Morris. UNIX Operating System Securiiy. BLTJ, vol. 63, núm. 8, parte 2, págs. 1649-1672, octubre 1984.
[1985] M. Harrison. Theoreticai Issues Concerning Protection in Operating Systems. Advances in Computers, vol. 24, págs. 61-100, 1985.
[1990] L. Hoffman. Rogue Pro grams: Viruses, Wor,ns, Trojan Horses. Van Nostrand Reinhold, 1990.
[1978] A. K. Jones. «Protection Mechanisms and the Enforcement of Security Policies». En E. Bayer et al., eds. Operating Systems: An Advanced Course, págs. 228-250, Springer-Verlag, Berlín, 1978.
[1984] P. Karger y A. Herbert. «An Augmented Capability Architecture to Support Lattice Security». Proc, of the IEEE Symp. Qn Security and Privacy, págs. 2-12, IEEE Computer Society Press, 1984.
[1990] P. Karger et al. «A VMM Security Kernel for the VAX Architecturex.. Proc, of the IEEE Symp. on Securily and Privacy, págs. 2-19, IEEE Computer Society Press, 1990.
[1988] S. Krakowiak. Principies of Operating Systems. MIT Press, 1988.
[1999] P. Lambert. Implementing Security on Linux. Journal of System Administration, vol. 8, núm. lo, págs. 67-70, octubre 1999.
[1981] L. Lamport. Password Authentication with Insecure Communication. Communications of the ACM, vol. 24, núm. 11, págs. 770-772, noviembre 1981.
[1974] B. Lampson. Protection. ACM Operating Systems Review, vol. 8, núm. 1, págs. 18-24, enero 1974
[19811 C. Landwher. Formal Models for Computer Security. Computer Surveys, vol. 13, núm. 3, págs. 247-278, septiembre 1981.
[19881 T. Lee. «Using Mandatory Integrity to Enforce Commercial Security». Proc, of the IEEE Symp. on Securiiy and Privacy, págs. 140-146, IEEE Computer Society Press, 1988.
[1976] T. A. Linden. Operating System Structures to Support Security and Reliable Software. Compu ting Surveys, vol. 8, núm. 4, págs. 409-445, diciembre 1976.
[1989] M. Luby y C. Rackoff. A study of password security. Journal of Cryptology: the Journal of the International Association for Cryptologic Research, vol. 1, núm. 3, págs. 15 1-158, 1989.
[1991] T. Mayfield. Integrity in an Automated Information System. NCSC C Technical Report, págs. 79-91, septiembre 1991.
[1974] W. S. McPhee. Operating System Integrity in OS/VS2. IBM Systems Journal, vol. 13, núm. 3, págs. 230-252, 1974.
[19821 C. H. Meyer y 5. M. Matyas. Cryptography. John Wiley & Sons, Nueva York, 1982.

Digitalización realizada con propósito académico

[19921 J. Millen. «A resource Allocation Model for Denia! of Service». Proc. of the IEEE Symp. on Security and Privacy, págs. 137-147, IEEE Computer Society Press, 1992.
[1987] S. P. Miller, B. C. Neuman, J. 1. Schiller y J. H. Saltzer. Kerberos Authentication andAuthoriza tion System. MIT Project Athena, Cambridge, Massachusetts, diciembre, 1987.
[1979] R. Monis y K. Thornpson. Password Securily: A Case History. Communications of the ACM, vol. 22, núm. 11, págs. 594-597, noviembre 1979.
[1982] R. H. Monis. Cryptographic Features of the UNIX Operating System. Cryptologia, vol. 6, núm. 3, julio 1982.
[1986] S. J. Mullender y A. S. Tanenbaum. The Design of a Capabiliiy-Based Distributed Opera ting System. IEEE Computer, vol. 29, núm. 4, págs. 289-299, 1986.
[1990] M. Nash y K. Poland. «Sorne Conundrurns Concerning Separation of Duty». Proc. of the IEEE Symp. on Security and Privacy, págs. 201-207, IEEE Cornputer Society Press, 1990.
[1977] National Bureau of Standards. Data Encryption Standard. FIPS Publication 46, enero 1977.
[1985] Nacional Computer Secunty Center. Orange Book, diciembre 1985.
[1990] P. Neurnann. «Towards Standars and Criteria for Critical Computer Systems». Proc, of the COMPASS Conference, 1990.
[19811 D. Parker. Computer Security Management. Reston Books, 1981.
[1997] C. P. Pfleeger. Security in Computing, 2. edición. Prentice-Hall, 1997.
[19741 U. J. Popek. Protection Structures. IEEE Computer, vol. 7, núm. 6, págs. 22-33, junio 1974.
[1979] G. J. Popek et al. UCLA data secure UNIX. NCC 1979, vol. 48, págs. 355-364.
[1994] J. Richter. Advanced Windows NT. Microsoft Press, 1994.
[1978] R. L. Rivest eta!. On Digital Signatures and Public Key Cryptosystems. Cornmunications of the ACM, vol. 2!, núm. 2, págs. 120-126, febrero 1978.
[1969] R. F. Rosin. Supervisory and Monitor Systems. Computing Surveys, vol. 1, núm. 1, págs. 37-54, marzo 1969.
[1979] L. H. Seawright y R. A. MacKinnon. VM/370 - a Study of Muftipliciiy & Usefu!ness. IBM Systems Journal, vol. 18, núm. 1, págs. 4-17, 1979.
[19961 B. Schneider. Applied Cryptography, 2. edición, John Wiley & Sons, 1996.
[1949] C. Shannon. Communication Theory of Secrecy Systems. Bel! Systems Technical Journal, vol. 28, págs. 659-715, octubre 1949.
[1998] A. Silberschatz y P. B. Galvin. Operating System Concepts, edición. Addison-Wesley, 1998.
[19941 G. Simmons. Cryptoana!isis and Protoco! Failures. Communications of the ACM, vol. 37, núm. 1!, págs. 54-64, noviembre 1994.
[1966] A. Sinkov. Elementary Cryptoana!isis: A Mathematical Approach. Math Association of America, 1996.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.
[1995] W. Stallings. Operating Systems, 2. edición. Prentice-Hall, 1995.
[1999] M. Taber y R. Roger. Maximum Linux securily: a hacker's guide toprotecting your Linux server and network. Macmillan Computer Publishing, 1999.
[1992] A. 5. Tanenbaum. Modern Operating Systems. Prentice-Hall, 1992.
[1997] A. S. Tanenbaum y A. Woodhull. Operating Systems Design and Implementation, 2. edi ción. Prentice-Hall, 1997.
[Tassel, 1972] D. Van Tassel. Computer Securiiy Management. Prentice-Hall, 1972.
[1979] W. Ware. Security Contro!sfor Comp uter Systems. Rand Corp Technical Report R-609-1, octu bre 1979.
[1984] W. Ware. Information System Securily and Privacy, Communications of the ACM, vol. 27, núm. 4, págs. 316-32!, abril 1984.
[1995] W. Ware. «A Retrospective on the Criteria Movement». Proc, of the National Information Systems Security Conference, págs. 582-588, 1995.
[1992] D. A. Willcox y 5. R. Bunch. «A Tool for Covert Storage Channel Analysis of the UNIX Kernel». J5th Nationa! Computer Security Conference, págs. 13-16, Baltimore Convention Center, octubre 1992.

Digitalización realizada con propósito académico

{Wilkes, 19821 M. Wilkes. «Hardware Support for Memory Protection». ACM Symp. On Architectural Sup portfor Programming Languages and Operating Systems, págs. 107-1 16, 1982.
[18951 P. H. Wood y S. G. Kochan. UNIX System Securily. Hayden UNIX System Library, Hayden Books, 1985.

Capítulo 10. Introducción a los sistemas distribuidos

[Accetta, 1986] M. Accetta, R. Baron. D. Golub, R. Rashid, A. Tevanian y M. Young. Mach: A New Kernel Foundationfor UNIX Development. Proceedings Summer 1986. USENIX Conference, págs. 93-112, 1986.
[1994] K. P. Birman y R. Van Renesse. Reliable Distributed Computing with the Isis Toolkit. Nueva York: IEEE Computer Society Press, 1994.
[1996a] K. P. Birman y R. Van Renesse. Software for Re/jable Networks. Scientific American 274:5, págs. 64-69, mayo 1996.
[1996b] K. P. Birman. Building Secure and Reliable Networks Applications. Manning Publications Co, 1996.
[19841 A. D. Bine! y B. J. Nelson. Implementig Remote Procedure Ca/ls. ACM Transactíons on Com puter Systems, vol 2, págs. 39-59, febrero 1984.
[20011 G. Coulouris, J. Dollimore y T. Kindberg. Distributed Systems. Concepts and Design, 3. edición, Addison-Wesley, 2001.
[1988] C. Fidge. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. Procee dings of the Eleventh Australian Computer Science Conference, 1988.
[20001 L. D. Galli. Distributed Operating Systems: Concepts and Practice. Prentice-Hall, 2000.
[1997] A. S. Grimshaw, Wm. A. Wulf y the Legion Team. The Legion Vision of a Wordwide Virtual Computer. Communications of the ACM, 40(1), enero 1997.
[19991 Proceedings of the IEEE, marzo 1999. Número especial sobre Memoria compartida distribuida. [19941 P. Jalote. Fault Tolerance in Distributed Systems. Prentice-Hall, 1994.
[19781 L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communica tions of the ACM 21:7, págs. 558-565, abril 1978.
[19861 K. Li. Shared Virtual Memory on Loosely Coupled Multiprocessors. Tesis Doctoral, Universidad de Yale, 1986.
[19891 F. Mattern. Tkime and Global States in Distributed Systems. Proceedings of the International Workshop on Parallel and Distributed Algorithms. Amsterdam, 1989.
[1990] S. Mullender, G. Van Rossum, A. 5. Tanenbaum, R. Van Renesse y H. Van Staveren. Amoeba: A Distributed Operating Systems for the 1990s. IEEE Computer, vol. 23, núm. 5 págs. 44-53, mayo 1990.
[1993] S. Mullender (Ed.). Distributed Systems, 2. edición, ACM Press, Nueva York, 1993.
[1999] R. Orfali, D. Harkey y J. Edwards. Client/Server Survival Guide, 3. edición, Wiley Computer Publishing, 1999.
[19961 R. Otte, P. Patrick y M. Roy. Understanding CORBA. Prentice-Hall, 1996.
[19981 J. M. Protíc, M. Tomasevic y V. Milutinovic. Distributed Shared Memory: Concepts and Systems. IEEE Computer Society Press, Los Alamitos, California, 1998.
[19811 G. Ricart, A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. Communications of the ACM, vol. 24, págs. 9-17, enero 1981.
[1988] M. Roizer, V. Abrossimov, F. Armand, 1. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois y W. Neuhauser. Chorus Distributed Operating System. Computing Systems, vol. 1, págs. 305-379, octubre 1988.
[1992] W. Rosenberry, D. Kenney y G. Fisher. Understanding DCE. O'Reilly, 1992.
[1999] W. Rubin, M. Brain y R. Rubin. Understanding DCOM. Prentice-Hall, 1999.
[20001 W. Stallings. Data and Computer Communications, 6. edición. Prentice-Hall, 2000.
[1999] W. R. Stevens. UNIX Network Programming. Prentice-Hall, 1999.
[1995] A. S. Tanenbaum. Distributed Operating Systems. Prentice-Hall, 1995.
[1996] A. S. Tanenbaum. Computer IVetworks, edición. Prentice-Hall, 1996.

Digitalización realizada con propósito académico

Capítulo 11. Estudio de casos: LINUX

[1996] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus y D. Verworner. Linux Kernel ínter nals. Addison-Wesley, 1996.
[19971 P. Comes. The Linux A-Z. Prentice-Hall, 1997.
[19981 A. Silberschatz y P. B. Galvin. Operating System Concepts, a edición, Addison-Wesley, 1998.
[1987] A. S. Tanenbaum. Operating Systems: Design and implementation, . edición. Prentice- Hall, 1987.
[1997] A. S. Tanenbaum y A. Woodhull. Opera:ing Systems: Design and Implementation, 2. edi ción. Prentice-Hall, 1997.

Capitulo 12. Estudio de casos: Windows NT

[Baker, 1997] A. Baker. The Windows NT Device Driver Book: A Guide for Programmers, Prentice-Hall,
1997.
[1993] H. Custer. Inside Windows NT, l. edición, Microsoft Press, 1993.
[1995] H. Custer. Inside Windows NT File System, Microsoft Press, 1995.
liDekker, 1999] E. N. Dekker y J. M. Newcomer. Developing Windows NT Device Drivers: A Pro grammer ‘s
Handbook. Addison-Wesley, 1999.
[2000] K. Ivens. Managing Windows NT Logons. O’Reilly & Associates, Inc., 2000.
[2000] A. G. Lowe-Norris. Windows 2000 Active Directory. O’Reilly, 2000.
[1999] J. M. Martínez Delgado y X. Martínez Balart. Windows NT 4.0 Server: Guía Avanzada. Prentice-Hall, 1999.
‘ 1997] S. Mitchell. Inside the Windows 95 File System. O’Reilly & Associates, Inc., 1997
[1997] R. Nagar. Windows NT File System Internals: A Developers Guide. O’Reilly & Associates, Inc.,
1997.
[1997] E. Pearce. Windows NT in a Nutshell: A Desktop Quick Reference for System Administration.
O’Reilly & Associates, Inc., 1997.
[1998] D. A. Solomon. Inside Windows NT, 2. edición, Microsoft Press, 1998.

Digitalización realizada con propósito académico

Digitalización realizada con propósito académico

Índice

abort, 126
accept, 574, 688
Acceso
directo, 427
secuencial, 427
Acceso directo a memoria (DMA), 3, 361
operación, 362
programación del controlador, 361
access, 541
ACE, Access Control Entry, 64
entradas de, 646
ACL, Access Control List, 535
ACL. Véase Lista de control de acceso
Activación
de un proceso, 97
del sistema operativo, 57-58
Activador, 82, 97, 102
Actualización atómica, 489
AddAccessAllowedACe, 547
AddAccessDeniedACe, 547
Administrador, 38, 501
de programas, 67
Agrupación, 459, 478
ajo_cancel, 443
aio_read, 443
aio_suspend, 443
a±o wrjte, 443
AIX, 70
alarm, 143, 407
Algoritmo del banquero, 341
Algoritmos de cifrado, 519
de bloques, 520
de flujo de caracteres, 520
de permutación, 520
de sustitución, 519
de transposición, 520
Algoritmos de planificación 105-107
aleatorio, 106
cíclico, 105
FIFO, 105
lotería, 10
por prioridades, 106
primero al trabajo más corto, 106
round-robin, 105
Algoritmos de predicción de interblo queos, 339
para una representación mediante
grafo de recursos, 339
Algoritmos de sustitución, 519
monoalfabéticos, 519
polialfabéticos, 519
Amacenamiento
con redundancia, 486
estable, 385
Almacenamiento secundario, 373
discos, 374
Almacenamiento terciario, 387
componentes, 388
estructura, 388
sistema de almacenamiento HPSS, 391
tecnología, 388
Amoeba, 44, 72
Antivirus, 503
API (Application Programming ínter-
face) del sistema operativo, 35
ar, 661
Arbol
de directorios en UNIX, 437
de directorios en Windows, 437
Archivo
ejecutable, 89
proyectado en memoria, 210-211
Archivos, 420, 421
ASCII, 421
atributos, 421
binarios, 421
de mandatos, 37
especiales, 42
estructura, 424
métodos de acceso, 427
nombres, 423
semántica de coutilización, 428
sombra, 530
visión física, 51, 420
visión lógica, 51, 420
Archivos en POSIX, 440
atributos de un archivo, 440
bits de protección, 440
descriptor de archivo, 440
argc, 123
argv, 123
Arquitectura
del sistema de E/S, 363
von Neuman, 2
Arranque
de la computadora, 38-41
del sistema operativo, 40-41
Asignación contigua, 183-186
Asignación de procesos a procesado-
res, 598
estación de trabajo inactiva, 599
migración de procesos, 599
atexjt, 126
Atributos de un archivo, 421
Atributos de una cola de mensajes PO SIX, 275
mq_getattr, 275
mq_setattr, 276
Autenticación, 55, 501, 526
Autenticación
contraseña, 528
de usuarios, 643
login, 528
proceso de, 527
Backup, 390
Bandas, 638
Batch, modo de ejecución, 46
BCP. Véase Bloque de control del
proceso
Bibliotecas
de archivos, 661
dinámicas, 173-176, 630
carga explícita de, 176
estáticas, 173
bind, 573, 688
BIOS (Basic Input-Output System), 40
Bloque, 459
de carga, 378, 451
de control del proceso (BCP), 45, 78
descriptor de un proceso, 627
Bloques
de datos, 464

de índice, 476
de repuesto, 379
defectuosos, 379
Bombardeo, 504
Boot. Véase Cargador del sistema ope rativo
Broadcast, 592
Buffering de páginas, 203
Caballo de Troya, 500
Cache
de disco en el controlador, 363
de nombres, 479
Cache de archivos
en Windows NT, 634
Cache de bloques, 479, 480
coherencia, 480
escritura diferida, 482
escritura inmediata, 482
escritura retrasada, 483
flujo de datos con, 480
LRU, 481
política de escritura en UNIX, 483
política de i-eemplazo, 481
Cache de datos, 479
coherencia, 480
Cambio
de contexto, 95-97
de estado del proceso, 96
Canal encubierto, 501
Canales de E/S, 362
Capabi/ities, 55, 538
protección de, 539
revocación de derechos y, 540
Capacidades. Véase Capabilities
Caracterización de los dispositivos de
E/S, 354
Cargador del sistema operativo, 39
chdir, 453
chmod, 542
Chorus, 72
chown, 542
Ciclo
de aceptación de interrupción, 7
de vida de un archivo, 420
Cilindros, 375
Cintas magnéticas, 388
Clasificaciones de seguridad, 524
Clave
privada, 522
pública, 522
Claves de una sola vez, 531
Claves. Véase Contraseña
done, 614
close, 260, 441, 575
closedir, 452
CloseHandie, 289, 295, 300, 301, 304, 447
Código independiente de la posición, 175
Colas de mensajes, 249
atributos, 275
comunicación cliente-servidor con
colas de mensajes POSIX, 279
mailslots de Win32, 303
mq_close, 275
rnqgetattr, 275
mq_open, 275
mq_send, 275
mq_setattr, 276
mg_receive, 276
colas de mensajes POSIX, 274
productor-consumidor con colas de
mensajes POSIX, 276
secciones críticas con colas de men sajes POSIX, 276
Compartimento de memoria, 168
Compilación, 173
Compilador de IDL, 584
Componentes del sistema operativo,
41-42
Compresión de datos en el servidor de
archivos, 482
Comunicación
asíncrona, 49
de grupos, 592
de procesos, 570
síncrona, 49
Comunicación cliente-servidor, 231
con colas de mensajes POSIX, 279
con paso de mensajes, 252
con tuberías de Win32, 291
Condición necesaria y suficiente para
que se produzca un interbloqueo, 324
Condiciones necesarias para que se
produzca un interbloqueo, 324
Conexión de un dispositivo de EIS a
una computadora, 354
controladores de dispositivos, 354
dispositivos de E/S, 354
Confinamiento, 517
Conjunto
de trabajo (working set), 22, 187
de trabajos, 629
residente, 187
connect, 574, 688
Contabilidad, 36
Contador de programa (PC), 3, 5
Contraseña, 55, 528
almacenamiento de, 530
asignación de, 529
duración de las, 531
longitud y formato de, 529
Control de acceso a dispositivos, 372
Controlador de dispositivo, 355
Controladores inteligentes, 376
Controles
de programación, 516
de seguridad externos, 514
Copias
de respaldo, 485, 646
incrementales, 486
totales, 485
COPS, 504
copy, 677
CopyFile, 448
Corba, 72
core, 126
Cortafuegos, 516
COW (Copy-on-write), 210
Crackers, 516
creat, 441
CreateDirectory, 456
CreateEvent, 300
CreateFile, 288, 304, 446
CreateFileMapping, 216
CreateMailslot, 303
CreateMutex, 299
CreateNamedPjpe, 287
CreatePipe, 286
CreateProcess, 148
CreateSemaphore, 295
CreateThread, 153
Criptografía, 519
Criterio común, 524
CTSS, 69
Cuentas de usuario, 501
DCE (Distributed Computing Envi ronment), 72, 582
DCOM, 72
DeleteCriticalSection, 294
DeleteFile, 446
Demonio, 40, 112, 369, 533

Digitalización realizada con propósito académico

Depuración de aplicaciones, 663
en UNIX, 663
en Visual C++, 667
DES (Data Encryption Standard), 520
Descriptor de archivo en POSIX, 440
de entrada estándar, 440
de error estándar, 440
de salida estándar, 440
Detección y recuperación de interblo queos, 327
activación del algoritmo de detec ción, 33
algoritmo de detección utilizando
un grafo de recursos, 328
algoritmo de detección utilizando una representación matricial, 331
detección, 328
recuperación, 334
DFS (Distributed File System), 637
Difusión de mensajes, 592
Dirección
física, 165
lógica, 165
Direccionamiento extendido, 628
Directorio, 53, 429
•de páginas, 628
de trabajo, 432
en grafo acíclico, 434
raíz, 53, 432
visión física de un, 54, 431
visión lógica de un, 53, 430
Directorios
con dos niveles, 432
de un nivel, 432
en árbol, 432
en POSIX, 451
entrada de directorio en POSIX,
451
estructura DIR, 452
Discos, 24, 374
almacenamiento intermedio en el
controlador, 376
cilindros, 375
controladores inteligentes, 376
densidad de cada pista, 376
discos RAM, 384
discos sólidos, 385
duros, 374
en memoria, 384
espejo, 385, 646
estructura física, 375
estructura lógica, 377
extraíbles, 374
gestión de errores, 383
IDE, 374
intercalado de sectores, 376
manejador, 379
ópticos, 375
pistas, 375
planificación, 382
RAM, 479
SCSI, 374
sectores, 375
tiempo de acceso, 375
tiempo de latencia, 375
tiempo de búsqueda, 375
en memoria, 384
WROM, 388
Diseño de sistemas seguros, 509
Diseño seguro por capas, 513
en MULTICS, 514
en VMS, 514
Dispositivo de paginación, 187
Dispositivos
de bloques, 356
de caracteres, 356
de E/S conectados por puertos, 355
de E/S proyectados en memoria, 355
orientados a bloques, 50
orientados a caracteres, 50
RAID, 386
de E/S, 352
de almacenamiento, 352
de comunicaciones, 352
de interfaz de usuario, 352
Disquetes, 375
Distribución de carga, 598
algoritmos de, 599
DLL (Dynamic Linked Library). Véa se Bibliotecas dinámicas
DMA. Véase Acceso directo a memo ria
DoD-2167A, 517
Dominio de protección, 532
cambio de, 532
en UNIX, 532
Duplicación de discos, 646
E/S bloqueante, 370
flujo de operación, 371
E/S no bloqueante, 370
en POSIX, 370
en Win32, 371
flujo de operación, 371
E/S por interrupciones, 357
inhibición, 359
tratamiento, 359
E/S programada, 357
EDVAC, 57
Ejecutable de Win32, 89
ELF, 176
ENIAC, 57
Enlace, 434
contador de enlaces, 434
dinámico de RPC, 586
físico, 434
simbólico, 434
EnterCriticalSection, 294
Entorno
de programación, 657
del proceso, 78, 93-97
Entornos de ejecución de Windows
NT, 635
Entornos virtuales, 513
en MVS, 514
Entrada de directorio en POSIX, 451
Entrada/salida, 23-27, 351
arquitectura del sistema, 363
caracterización de los dispositivos
de E/S, 354
en Windows NT, 631
Entrada/salida
estructura y componentes del siste ma de E/S, 363
fuera de línea (off-line), 68
objetivos del sistema operativo, 353
por DMA, 25
por interrupciones, 25
programada, 25
software de E/S, 364
síncrona, 633
Envejecimiento, 106
environ, 116
Equipos de penetración, 516
errno, 60
Escritor perezoso, 635
Escrutinio de claves, 521
Espacio de nombres global, 603
esquema centralizado, 60
esquema distribuido, 603
Espacio de trabajo de C++, 664
añadir archivos fuente, 665
creación de, 664
Espacio lógico, 165
Espera activa, 26, 253
implementación, 254

Digitalización realizada con propósito académico

Espera pasiva, 26, 253
Estado
de un proceso, 82
del procesador, 45, 84
seguro en interbloqueos, 338
Estándares para seguridad, 517
Estructura
de un archivo, 424
del sistema de EIS de LINUX, 374
física de un disco, 375
lógica de un disco, 377
del sistema operativo, 42-44
sistemas con arquitectura cliente-
servidor, 43
sistemas estructurados, 43
sistemas monolíticos, 42
sistemas por capas, 43
y componentes del sistema de E/S, 363
interfaz del sistema operativo, 363
Eventos en Win32, 299
CloseHandie, 301
Crea teEvent, 300
PulseEvent, 301
SetEvent, 301
WaitForSingleObject, 301
Excepciones, 111-112
manejo estructurado de excepcio nes, 112
Exclusión mutua en sistemas distri buidos, 596
algoritmo centralizado, 596
algoritmo de paso de testigo, 596
algoritmo distribuido, 597
exec, 121-123
exit, exit, 125-126
ExitProcess, 150
ExitThread, 154
ext2fs, 616
Fallo de página, 17, 200
tratamiento del, 200
Fases de un proceso, 81
FAT (File Allocation Table), 637
área de datos, 637
bloque de carga, 637
directorio raíz, 637
información de la FAT, 637
sistemas de archivos, 52, 423, 637
fcntl, 443
fdformat, 375
fdisk, 377
FFS (Fast File System), 616
Fiabilidad y recuperación de un siste ma de archivos, 485
FIFOS, 258
FileTimeToSystemTime, 411
FindCiose, 457
FindFirstFile, 457
FindNextFile, 457
Firewall, 516
Firmas digitales, 523
Flujo
de datos en el servidor de archivos, 471
de una operación de EIS, 366
de una operación de EIS bloquean te, 371
de una operación de E/S no blo queante, 371
FMS, 68
fork, 118
Formación de un proceso, 93
format, 378
Formato del ejecutable, 176
Formato lógico de un disco, 378
forrnat, 378
mkfs, 378
Fragmentación
externa, 184
interna, 189
fsck, 487
fstat, 443
gcc, 660
gdb, 663
Generación de un ejecutable, 173
Gestión
de archivos, 472
estructuras de datos, 472
de archivos y directorios, 419
de memoria distribuida, 606
Gestión de procesos distribuidos, 598
modelo de estaciones de trabajo, 598
modelo híbrido, 598
Gestión del espacio de libre, 477
listas de recursos libres, 478
mapas de bits, 477
Gestor
de objetos, 624
de procesos, 625
de ventanas, 67
CetCurrentDírectory, 457
GetCurrentProcess, 146
GetCurrentprocessId, 146
GetCurrentThread, 152
GetCurrentThreadId, 152
getcwd, 453
getegid, 115, 542
getenv, 117
GetEnvironmentstrjngs, 147
GetEnvironmentvariabie, 147
geteuid, 115, 542
GetExceptionCode, 156
GetExitCodeProcess, 150
GetFileSecurity, 546
GetFileSize, 448
GetFileTime, 448
getgid, 115, 542
gethostbynarrie, 572
GetMailslotInfo, 304
getpid, 114
getppid, 114
GetPriorityCiass, 155
GetProcessTimes, 411
GetSecurityDescriptor Group, 547
GetSecuri tyDescríptorOw ner, 547
GetSystemTime, 410
GetThreadPriority, 155
gettimeofday, 406
getuid, 115, 542
GetUserName, 546
Gid. Véase Identificador de grupo de
usuarios
Grado de multiprogramación, 82
Grupo
de procesos, 79
de usuarios, 38
Gráfico de asignación de recursos, 318
algoritmo de detección de interbio queos, 328
algoritmo de predicción de interblo queos, 339
asignación de recursos, 319
restricción de asignación, 319
restricción de solicitud, 319
solicitud de recursos, 319
Gusanos, 503
HAL (Hardware Abstraction Layer), 621
Hardware
del reloj, 393
del terminal, 398

Digitalización realizada con propósito académico

Heap, 180, 184
Hiperpaginación, 84, 187, 205-207
estrategia basada en la frecuencia
de fallos de página, 206
estrategia del conjunto de trabajo, 205
HP-UX, 70
HPFS (High-Performance File System), 637, 638
HPSS. Véase Sistema de almacena miento HPSS
HRU, modelo de seguridad, 534
IBYSS, 68
Identificador de grupo de usuarios, 38, 115
de grupo de usuarios efectivo, 115
de usuario, 38, 115
efectivo, 115
IDL (Interface Definition Language), 584
Imagen de memoria del proceso, 45, 78, 85
Índice
enlazado, 475
multinivel, 476
Iniciador ROM, 39
InitializeAcl, 547
InitializeCriticalSec tion, 294
InitializeSecurityDescrip tor, 546
Instrucción swap, 256
secciones críticas con swap, 256
Instrucción test-and-set, 255
Interbloqueos, 257, 309
caracterización, 324
condición necesaria y suficiente, 325
condiciones necesarias, 324
definición, 324
en la vida real, 310
en sistemas informáticos, 311
modelo de sistema, 317
recursos compartidos, 314
recursos con un único ejemplar, 315
recursos consumibles, 313
recursos de usuario, 345
recursos internos del sistema, 345
recursos exclusivos, 314
recursos expropiables, 316
recursos múltiples, 315
recursos no expropiables, 316
recursos reutilizables, 313
reducción, 325
representación matricial, 322
representación mediante un gráfico
de asignación de recursos, 318
tipos de recursos, 311
tratamiento, 326, 345
Intercalado de sectores, 376
Intercambio, 186-187
con o sin preasignación, 187
dispositivo de, 186
Interfaz
alfanumérica, 63-65
de programador, 59-6 1
de usuario del sistema operativo, 6 1-67
gráfica, 65-67
POSIX, 59-60
Win32, 60-6 1
Interpretación de nombres de archi vos, 483
Intérprete de mandatos (shell), 35, 37, 41, 63
con mandatos externos, 65
con mandatos internos, 64
Interrupciones, 7-8
ciclo de aceptación de interrupción, 7
concepto, 7
ioctl, 408
IRIX, 70
ISO 9000, 517
Jerarquía
de memoria, 10-15
de procesos, 78
de directorios, 437
árbol de directorios único, 437
Juego de instrucciones, 4
Kerberos, 552
arquitectura de, 553
claves de sesión, 552
claves privadas, 552
claves temporales, 552
protocolos de, 554 kill, 141
KiliTimer, 158
LAN (Local Area Network), 564
LeaveCriticalSection, 294
Lenguaje
de control de trabajos, 68 de definición de interfaces, 584
LFS (Log File System), 637
link, 453
Linus Torvalds, 612
Linux, historia de, 613
características, 613
estructura, 613
gestión de procesos, 614
Lista
de capacidades (capabilities), 55
de control de acceso (ACL), 55, 535
en UNIX, 536
en Windows, 537
de páginas a cero, 629
de páginas en espera, 629
de páginas libres, 629
de páginas modificadas, 629
enlazada, 475
listen, 574, 688
localtirne, 406
Login, 40
Logon, proceso de, 551, 644
LPC, 630
LRU, política de reemplazo, 202
is, 674
lseek, 442
Llamada
a procedimiento local, 630
al sistema, 35, 36
Llamadas a procedimiento remoto, 582
de SUN, 591
diseño de, 583
paquetes de las, 587
semántica de las, 586
transferencia de parámetros de, 585
Mach, 44, 72
Macintosh, 66
Mailslots, 303
CloseHandie, 304
CreateFile, 304
CreateMailslot, 303
GetMailslotInfo, 304
ReadFile, 304
SetMailslotInfo, 304
WriteFile, 304
make, 657
makefile, 657
comentarios, 660
compilador, 660
estructura de, 658

Digitalización realizada con propósito académico

Manejador de disco, 379
estructura, 379, 380
gestión de errores, 382
manejador genérico, 379
manejador particular, 379
planificación, 382 Manejador de sistema de archivos, 632
Manejadores de dispositivos, 366 operaciones de un manejador, 367
Manejadores de interrupción, 365 Mapa
de bloques en la FAT de MS-DOS, 476
de bloques en un nodo-i, 477
Mapas de bits, 477
Map\JiewOfFile, 216
Máquina
desnuda, 34
virtual, 71
Marco de página, 17, 188
Matriz
de asignación, 322
de protección, 534
de solicitud, 322
Mecanismos de asignación de blo ques, 474
bloques contiguos, 474
bloques discontiguos, 475
Mecanismos de comunicación y sin cronización, 232
archivos, 232
implementación, 253
memoria compartida, 242
mutex, 243
semáforos, 237
señales, 237
servicios POSIX, 258
servicios Win32, 285
tuberías, 233
variables condicionales, 243
Mejor ajuste (best-fit), 185
Memoria compartida, 242
Memoria virtual, 15-23, 187-210
en Windows NT, 627
operaciones sobre regiones, 208-210
política de administración, 201
política de asignación, 204
política de reemplazo, 201-203
Mensaje, 564
Mensajes multipunto, 592
Merkle-Hellman, 520
Metainformación, 53-55, 462
de un archivo, 53
de un sistema de archivos, 55
Métodos de acceso a archivos, 427
acceso directo, 427
acceso secuencial, 427
MFT (Master File Table), 637
atributos residentes, 641
Micronúcleo, 43
Middleware, 72, 569
minikernel, 691
minisheil, 674
MINIX, 44, 612
rnkdir, 452
mkfifo, 259
mkf a, 378, 463
mrnap, 212
MMU. Véase Unidad memoria
Modelo
de copia primaria, 490
de gestión colectiva, 490
de programación, 3-5
Modelo de sistema para interblo queos, 317
restricción de asignación, 318
restricción de coherencia, 318
primitivas de liberación, 318
primitivas de solicitud, 317
Modelos de seguridad, 508
limitada, 509
multinivel, 508
Modo de ejecución, 46 batch, 46
interactivo, 46
Modos de direccionamiento, 4
Módulo de organización de archivos, 470
Monitorización, 36
Monotarea, 45, 80
Monousuario, 45, 80
Montado de sistemas de archivos, 483
Montaje, 173
mount, 438
mq 275
mqgetattr, 275
mqopen, 275
m 276
mqsend, 275
mq_setattr, 276
Mu/ti cast, 592
Multicomputadora, 30, 224
procesos cooperantes, 226
Nivel
de ejecución, 4 de núcleo, 4
de usuario, 4 Nodo-i, 52, 422, 436, 463 Nodo-y, 469
tabla de, 473
de gestión de procesos independientes, 225
tipos de procesos concurrentes, 225
MULTICS, 70
Multiplexación
de la memoria, 164
del procesador, 164
Multiprocesador, 30, 69, 224
Multiproceso, 45
Multiprogramación, 69
Multítarea, 45, 79-84
Multiusuario, 45, 80
munrnap, 213
Muralla china, 507
grupos en, 508
Mutex, 243
mutex en POSIX, 270
mutex en Win32, 299
operación lock, 243
productor-consumidor con mutex y variables condicionales, 245
sección crftica con mutex, 243
Mutex en POSIX, 270
pthread mutex des troy, 271
pthreadrnutexinit, 271
pthread..jnutex_lock, 271
pthread 271
productor-consumidor con mutex y variables condicionales, 272
Mutex en Win32, 299
CloseHandie, 300
CreateMutex, 299
OpenMutex, 300
ReleaseMutex, 300
secciones críticas con mutex de Win32, 301
WaitForsingleobject 300
MVS, 70
Nombre
de archivo absoluto, 53, 435
de archivo relativo, 53, 435
Nombres de los mecanismos de comu nicación y sincronización, 235
NTFS (NT File System), 637, 639

Digitalización realizada con propósito académico

Núcleo (kernel) del sistema operativo, 34
Núcleo seguro, 514
Número
de subarchivos, 700
mágico, 424
Objeto de memoria, 178
Objetos
de control, 623
de planificación, 623
open, 259, 441
opendir, 452
OpenMutex, 300
OpenProcess, 146
OpenSemaphore, 295
OpenThread, 153
Operación
signal sobre un semáforo, 237
wait sobre un semáforo, 237
Operaciones
de un manejador de disco IDE en
LINUX, 381
sobre regiones, 182
Orange Book, 524
Ordenación de eventos, 593
OS/2, 43
OS/360, 69
Overlays, 172
Página, 188
Paginación, 17, 188-197
por demanda, 199
Páginas
de intercambio, 17
virtuales, 17
Paquete, 564
Partición, 459
activa, 378
de intercambio, 459
raíz, 459
Particiones extendidas, 459
Paso de mensajes, 248
almacenamiento, 250
colas de mensajes, 249
colas de mensajes POSIX, 274
comunicación cliente-servidor con
paso de mensajes, 252
flujo de datos, 248
mailslots de Win32, 303
nombrado, 249
operación send, 248
productor-consumidor con paso de
mensajes, 251
puertos, 249
secciones críticas con paso de men sajes, 250
sincronización, 250
pause, 143
Peor ajuste (worst-Jit), 185
Periféricos, 23-25
perror, 119
PIC (Position-Independent Code).
Véase Código independiente de
la posición
pipe, 259
Pirata, 501
Pistas, 375
Planificación
de procesos distribuidos, 601
en Windows NT, 626
Planificación de procesos, 102-109
a corto plazo, 102
a largo plazo, 102
a medio plazo, 102
algoritmos de (véase algoritmos de planificación)
en POSIX, 107
en sistemas de tiempo real, 106
en Windows, 108
Planificación del disco, 382
política CSCAN, 383
política del ascensor, 382, 383
política FCFS (First Come First
Served), 382
política LOOK, 383
política SCAN, 383
política SSF (Shortest Seek First), 382
Planificador, 82, 94, 102
Política
de actualización, 12
de asignación de marcos de página, 204-205
asignación dinámica 204-205
asignación fija, 204
de asignación de memoria, 170
mejor ajuste (best-fit), 185
peor ajuste (worst-fit), 185
primero que ajuste (first-fit), 185
de escritura diferida, 483
de escritura en la cache de UNIX, 483
de escritura inmediata, 482
de escritura retrasada, 483
de escritura retrasada perezosa, 635
de extracción, 11
de ubicación, 11
de reemplazo, 11, 201-205
FIFO, 202
LRU, 202
óptima, 201
reloj, 202
segunda oportunidad, 202
de seguridad, 505
comercial, 507
militar, 505
de seguridad militar, 505
compartimento, 506
dominancia en, 506
niveles de seguridad, 505
Políticas
de actualización de cache, 605
de reemplazo de cache, 605
de reparto de carga, 600
Prácticas de Sistemas Operativos, 669
archivos con bandas, 699
comunicación con sockets, 685
gestión de archivos, 677
gestión de procesos, 673
manejo de interrupciones, 679
programación de scripts, 670
sistema multiprogramado, 690
Predicción de interbloqueos, 337
algoritmo del banquero, 341
algoritmo para una representación me diante un grafo de recursos, 339
algoritmos de predicción, 339
estado seguro, 338
Prepaginación, 200
Prevención de interbloqueos, 334
espera circular, 337
exclusión mutua, 334
retención y espera, 335
sin expropiación, 336
Primero que ajuste (first-fit), 185
Prioridad, 626
base, 615
Privilegios, 55
Problema de los lectores-escritores, 230
resolución con mutex, 247
resolución con semáforos, 241
Problema del productor-consumidor, 230
resolución con colas de mensajes
POSIX, 277
resolución con mutex y variables
condicionales, 245
operación receive, 248

Digitalización realizada con propósito académico

resolución con mutex y variables
condicionales POSIX, 272
resolución con paso de mensajes,
251
resolución con semáforos, 238
resolución con semáforos POSIX, 267
resolución con semáforos de Win32,
296
resolución con tuberías, 236
resolución con tuberías POSIX, 262
Problemas de comunicación y sincro nización, 226
comunicación cliente-servidor, 231
el problema de la sección crítica,
225
el problema de los lectores-escrito res, 230
el problema del productor-consumi dor, 230
Problemas de segundad, 499
Procesamiento transaccional, 646
Proceso, 36, 78 cliente, 579
distribuido, 71
nulo, 81
servidor, 576
ligero, 98-102 estado de un, 99
Procesos concurrentes, 224
tipos de procesos concurrentes, 225
Procesos huérfanos, 129
Programa máquina, 3
Programas del sistema, 64
Protección, 27-30, 35
de memoria, 167
Protocolo TCP/IP, 565
Protocolos de comunicación, 565
Proximidad
espacial, 14
referencia!, 13-15
secuencial, 14
temporal, 15
Proyección en memoria de un archivo.
21 0-2 11
pthread ram, 138
pthread_at tr_getschedpo licy, 138
pthread_attr_getscope, 138
pthreadattr_setschedpa ram, 138
pthreadattrsetscope, 138
pthread_cond_broadcast, 272
pthread 271
pthreac ignal, 272
pthread_cond_wait, 271
pthreadcreate, 133
pthread_exit, 133
pthreacLjoin, 133
pthreadmutexdestroy, 271
pthreadmutexinit, 27!
pthread_mutex_lock, 27!
pthreadmutex_unlock, 271
pthreadself, 133
Puerta de atrás, 500
Puertos, 249
y minipuertos, 632
PulseEvent, 30!
Puntero
de pila (SP), 3, 5
de posición de un archivo, 5!
QNX, 71
R.T. Monis, 503
RAID (Redundant Array of Inexpensi ve Disks), 5, 385, 646
read, 260, 442, 575
ReadConsole, 413 readdir, 453
ReadFile, 289, 304, 447
Recuperación del sistema de archivos,
486
fsck, 486
Recursos
compartidos, 314
consumibles, 312
de usuario, 345
exclusivos, 314
expropiables, 316
internos del sistema, 345
no expropiables, 316
reutilizables, 312
recv, 575, 689
recvfrom, 575, 689
Red, 404, 563
interfaz de aplicaciones, 404
nivel de dispositivo de red, 404
nivel de protocolos, 404
niveles del software de gestión de red, 405
Reducción, 325 Registro
base, 28, 183 de acceso, 517
de estado, 3, 6 identificador de espacio de direc cionamiento (RIED), 7, 22, 29,
85
límite, 28, 183
MFT, 421, 463
Registros valla, 28, 183
ReleaseMutex, 300
ReleaseSemaphore, 296
Reloj, 9, 393
contabilidad y estadísticas, 396
gestión de temporízadores, 396 hardware, 393
mantenimiento de la hora y fecha,
395
software, 394
soporte a la planificación de proce sos, 397
tick, 394
Relojes
lógicos, 593
vectoriales, 595
RemoveDirectory, 457
rename, 443
Replicación, 490
modelo de copia primaria, 490
modelo de gestión colectiva, 490
Representación matricial de interblo queos, 322
algoritmo de detección de interbio queos, 331
matriz de asignación, 322
matriz de solicitud, 322
restricción de asignación, 323
restricción de solicitud, 323
vector de recursos existentes, 322
Resguardos, 582
ResumeThread, 154
Retención de páginas en memoria, 203
return, 125
Reubicación, 165
hardware, 166
software, 167
rewinddir, 453
RIED. Véase Registro identificador de
espacio de direccionamiento
rmdir, 452
Rompedores de sistemas de protec
pthreadattr_setschedpo licy, 138
ción, 504

Digitalización realizada con propósito académico

RPC (Remote Procedure Cali), 582
rpcgen, 589
RSA, 520
RTEMS, 71
R.T. Monis, 503
Rutina de tratamiento de interrupción, 359
Salvaguarda del estado del proceso, 96
Satan, 504
SCHED_FIFO, 137
sched_getparam, 137
sched_getpriority_max, 138
sched_getpriority_min, 138
sched_getscheduler, 138
SCHED_OTHER, 137
SCHED_RR, 137
sched_setparam, 137
sched_setscheduler, 137
Sección crítica, 226 requisitos para resolver el problema de, 229
resolución con colas de mensajes
POSIX, 276
resolución con la instrucción swap, 256
resolución con mutex, 239
resolución con mutex de Win32, 301
resolución con paso de mensajes, 250
resolución con secciones críticas de Win32, 294
resolución con semáforos, 238
resolución con tuberías, 235
resolución con tuberías POSIX, 261 Secciones críticas de Win32, 294
DeleteCriticalSection, 294
EnterCriticalSection, 294
InitializeCriticalSec tion, 294
LeaveCriticalSection, 294
resolución del problema de la sec ción crítica, 294
Sectores, 375
SEE-CMM, 517
Segmentación, 197-198
paginada, 198-199
Segmento de memoria, 17
Seguridad, 498, 505
polftica de, 505
requisitos de, 505
select, 688
sem_close, 266
sem_destroy, 266
sem_init, 265
sem_open, 266
sem_post, 267
sem_unlink, 266
sem_wait, 266
Semáforos, 237 lectores escritores con semáforos, 241
operación signal, 237
operación wait, 237
productor-consumidor con semáfo ros, 238
sección crítica con semáforos, 238
semáforos POSIX, 265
semáforos Win32, 295
Semáforos POSIX, 265
productor-consumidor con semáfo ros POSIX, 267
semáforos con nombre, 265
semáforos sin nombre, 265
sem_close, 266
sem_destroy, 266
sem_init, 265
sem_open, 266
sem_post, 267
sem_unlink, 266
sem_wait, 266
Semáforos Win32, 295
CloseHandie, 295
CreateSemaphore, 295
OpenSernaphore, 295
productor-consumidor con semáfo ros en Win32, 296
ReleaseSemaphore, 296
WaitForSingleObject, 296
Semántica
de archivos inmutables, 429
de coutilización, 428
de sesión, 428
de UNIX, 428
versiones, 428
send, 574, 689
sendto, 575, 689
Señales, 110-111
de proceso a proceso, 110
de sistema operativo a proceso, 111
tipos de, 111
Separación de recursos, 512
criptográfica, 513
física, 512
lógica, 513
temporal, 513
Servicio de fecha y hora en POSIX, 406
gettirneofday, 406
localtime, 406
settimeofday, 406
stime, 406 time, 406
Servicios POSIX
de gestión de señales, 139-146
de contabilidad, 407
de E/S, 408
de memoria, 212-216
de planificación, 136-139
de procesos, 114-131
de procesos ligeros, 131-136
de temporización, 407
para archivos, 441 para directorios, 452
Sevicios de contabilidad en POSIX, 407 times, 407
Servicios de contabilidad en Win32, 411
FileTimeToSystemTime, 411
GetProcessTimes, 411
Servicios de directorios, 451
genéricos, 451
POSIX, 451
Win32, 456
Servicios de E/S, 405
genéricos, 405
POSIX, 406
Win32, 410
Servicios de EIS en POSIX, 406
de contabilidad, 407
de EIS sobre dispositivos, 408
de fecha y hora, 406
de temporización, 407
Servicios de E/S en Win32, 410
de contabilidad, 411
de EIS sobre dispositivos, 413
de fecha y hora, 410
de temporización, 411
Servicios de EIS sobre dispositivos en POSIX, 408
ioctl, 408
tcgetattr, 409 tcsetattr, 409
Servicios de E/S sobre dispositivos en Win32, 413
ReadConsole, 413
SetConsoleMode, 413
WriteConsole, 413

Digitalización realizada con propósito académico

Servicios de fecha y hora en Win32, 410
GetSystemT±me, 410
SetSystemTime, 410
Servicios de protección POSIX, 541
ejemplos de, 543
Servicios de protección y seguridad, 540
genéricos, 540
POSIX, 541
Win32, 545
Servicios de temporización en POSIX, 407
alarm, 407
settimer, 407
Servicios de temporización en Win32, 411
SetTimer, 411
Servicios para archivos, 439
genéricos, 439
POSIX, 440
de Win32, 445
Servicios POSIX para archivos, 441
ajo_cancel, 443
aio_read, 443
aio_suspend, 443
aio_write, 443
close, 441
creat, 441
fcntl, 443
fstat, 443
lseek, 442
open, 441
read, 442
rename, 443
stat, 443
unl 441
write, 442
Servicios POSIX para directorios, 452
chdir, 453
closedir, 452
getcwd, 453
link, 453
mkdir, 452
opendir, 452
readdir, 453
rewinddir, 453
rmdir, 452
symlink, 453
Servicios POS IX y Win32, 651
archivos, 654
cerrojos, 652
comunicación, 652
directorios, 655
memoria compartida, 651
procesos, 652
procesos ligeros, 653
seguridad, 655
semáforos, 654
señales, 651
sincronización, 653
tiempo, 654
Servicios Win32
de gestión de procesos, 146-152
de gestión de procesos ligeros, 152-154
de memoria, 216-219
de planificación, 154-155
de temporizadores, 157-158
para manejo de excepciones, 155-157
Servicios Win32 para archivos, 445
CloseHandle, 447
CopyFile, 448
CreateFile, 446
DeleteFile, 446
GetFileSize, 448
GetFileTime, 448
ReadFile, 447
SetFilepointer, 448
WriteFile, 447
Servicios Win32 para directorios, 456
CreateDirectory, 456
FindClose, 457
FindFirstFile, 457
FindNextFile, 457
GetCurrentDirectory, 457
RemoveDirectory, 457
SetCurrentDirectory, 45
Servidor de archivos, 468
estructura, 469
fiabilidad y recuperación, 485
flujo de datos, 471
gestión del espacio libre, 477
mecanismos de asignación de blo ques, 474
mecanismos de incremento de pres taciones, 479
Servidor de bloques, 470
SetConsoleNode, 413
SetCurrentDirectory, 457
SetEnvironmentVariable, 147
SetEvent, 301
SetFilePointer, 448
SetFileSecurity, 546
setgid, 533, 542
SetMailslotInfo, 304
SetPriorityClass, 155
SetSecurityDescriptorAcl, 547
SetSecuri tyDescriptorGroup, 547
SetSecuri tyDescriptorüw ner, 547
SetSystemTirne, 410
SetThreadPriority, 155
settimeofday, 406
settimer, 407
SetTimer, 157, 411
setuid, 533, 542
set-user-id, 123
Shell. Véase Intérprete de mandatos
Shell scnpts. Véase Archivos de mandatos
sigaction, 141
sigaddset, 140
SIGCHLD, 126
sigdelset, 140
sigemptyset, 140
sigfillset, 140
sigismeruber, 140
sigpending, 143
sigprocmask, 142
Sincronización de procesos, 593
Sistema
confiable, 509
de almacenamiento HPSS, 391
estructura, 391
interfaz de usuario, 393
de archivos, 55, 421, 459
bloque de carga, 451
con bandas, 465, 467
en MS-DOS, 451, 461
en UNIX, 451, 461
en Windows-NT, 451, 461
estructura, 461
EXT2 (extended file system), 465
FFS (fast file system), 465
interpretación de nombres, 483
LFS (log file system), 465, -467
metainformación, 462
montado, 483
paralelos, 465
superbloque, 462
virtual, 469
de archivos FFS (fast file system), 465
fragmento, 466
grupos de cilindros, 466
de memoria en Linux, 615
de tiempo real, 69
de ventanas X, 66
distribuido, 562
características de, 562
definición de, 562

Digitalización realizada con propósito académico

operativo, definición, 34-3 8
como gestor de recursos, 35
como interfaz de usuario, 35
como máquina virtual extendida,
35
operativo de red, 566
Sistemas de archivos, comparación,
642
Sistemas de archivos distribuidos, 601,
602
acceso remoto en, 603
cache en, 604
nombrado en, 602 Sistemas de criptografía, 521
asimétricos, 521
simétricos, 521
Sistemas operativos de tiempo real, 71 Sistemas operativos distribuidos, 72,
567
fiabilidad, 568
flexibilidad, 569
rendimiento, 568
servicios de, 569
transparencia en, 567
Sistemas por lotes (batch), 68
sleep, 143 Sleep, 158
socket, 573, 685, 688 Sockets, 570
comunicación con, 571
direcciones de, 572
dominios de, 571
ejemplo de uso, 581
tipos de, 571
UNIX BSD 4.2, 570 Software de E/S, 364
flujo de una operación de EIS, 366
manejadores de dispositivos, 366
manejadores de interrupción, 365
Software de E/S independiente del dispositivo, 368
almacenamiento intermedio, 368
gestión de los dispositivos, 368
gestión de errores, 369
planificación de la EIS, 368
tamaño de acceso, 368
Software de entrada del terminal, 400
caracteres de control de flujo, 402
caracteres de edición, 402
caracteres de escape, 402
caracteres para el control de proce
sos, 402 tecleado anticipado, 401
Software de salida del terminal, 402 terminal proyectado en memoria, 403
terminal serie, 403
Software del reloj, 394
contabilidad y estadísticas, 396 gestión de temporizadores, 396
mantenimiento de la hora y fecha,
395
soporte a la planificación de proce sos, 397
Software del terminal, 400 de entrada, 400
Software fiable, 517
Solapamiento de búsquedas y transfe rencias, 363
Solaris, 70 sort, 674
Sospecha mutua, 517
Spoofing, 504
Spooler, 369
stat, 443
stime, 406 Stubs. Véase Suplentes
Subsistema de seguridad de Windows,
551
Superbloque, 462 en LINUX, 462
tabla de superbloques, 463
Superusuario, 38, 501
Suplentes, 582
SuspendThread, 154
Suspensión de procesos, 95
swap, 255
Swapping. Véase Intercambio
symlink, 453
Tabla
de archivos abiertos, 473
de descriptores de procesos, 623
de interrupciones, 623
de marcos de página, 191
de nodos-v, 473
de particiones, 377
de regiones, 170
de superbloques, 463 Tablas de páginas, 18-22, 188, 628
de usuario, 188
del sistema, 188
implementación, 192
invertida, 196
multinivel, 193
Tablas de directorios en UNIX, 436
Lar, 390
Tareas de seguridad, 511
y componentes del sistema operati vo, 512
Tasa de aciertos, 12
tcgetattr, 409
tcsetattr, 409
Técnicas de diseño de sistemas segu
ros, 512 Temporizadores, 112
Terminal, 397
entrada, 398
hardware, 398
modo de operación, 397
proyectados en memoria, 398
salida, 398
software, 400
software de entrada, 400
software de salida, 402
terminales serie, 399
Terminales
proyectados en memoria, 398
serie, 399
TerminateProcess, 150
TerminateThread, 154
test-and-set, 256
THE, 43
Thread. Véase Proceso ligero
Tick del reloj, 394
Tiempo
compartido, 45, 69
de acceso de un disco, 375
de búsqueda de un disco, 375
de latencia de un disco, 375
time, 406
times, 407
TLB (Transiation Look-aside Buffer), 22, 192-193
con o sin identificador de proceso,
193
gestionada por software, 193
Tolerancia a fallos, 646
Transacciones, 489
TRAP, 57
Tratamiento de los interbloqueos, 326
detección y recuperación, 327
predicción, 337
prevención, 334
Tratamiento del interbloqueo en los
sistemas operativos, 345
Traza de ejecución, 15

Digitalización realizada con propósito académico

Tuberías, 234
ejecución de mandatos con tube rías, 236
en POSIX, 258
en Win32, 286
escritura en una tubería, 234
lectura de una tubería, 234
productor-consumidor con tuberías, 236
sección crítica con tuberías, 235
Tuberías en POSIX, 258
close, 260
ejecución de mandatos con tuberías
POSIX, 262
FIFO, 258
mkfifo, 259
open, 59
pipe, 259
productor-consumidor con tuberías
POSIX, 262
read, 260
sección crítica con tuberías
POSIX, 261
unl±nk, 260
write, 261
Tuberías en Win32, 286
CloseHandie, 289
comunicación cliente-servidor con
tuberías en Win32, 291
CreateFile, 288
CreateNamedPipe, 287
CreatePipe, 286
ejecución de mandatos con tuberías
en Win32, 290
ReadFile, 289
WaitNamedPipe, 288
WriteFile, 289
UART (Universal Asynchronous Re ceiver-Transmitter), 399
Uid. Véase Identificador de usuario
uinask, 543
umount, 438
Unidad
aritmética, 3
de control, 3
de distribución, 700
Unidad de gestión de memoria, 17, 166
unlink, 260, 441
UnmapViewOfFile, 216
Usuario, 37-38
UTC (Universal Coordinated Time), 395
Variables condicionales, 243
operación c_signal, 243
operación c_wait, 243
productor-consumidor con mutex y
variables condicionales, 245
utilizando eventos de Win32, 302
variables condicionales en POSIX, 270
Variables condicionales en POSIX, 270
productor-consumidor con mutex y
variables condicionales, 272
pthreadcond_broadcas t, 272
pthreadcond_destroy, 271
pthread_cond_signal, 272
pthreadcondwa.it, 271
Vector de recursos existentes, 322
VFS (Virtual File System), 616
Violación de memoria, 20
Virus, 502
insertar un, 503
instalación y propagación de, 502
solución para, 503
Visión
de usuario del sistema de archivos, 420
física de un archivo, 420
física de un directorio, 431
ejemplos de entradas de directo rio, 431
lógica de un archivo, 420
lógica de un directorio, 430
en Windows NT, 430
Visual C++, 664
compilación en, 666
depuración, 667
ejecución en, 666
Volumen, 459
VRTX, 71
wait, 126
WaitForMultipleObjects, 151, 286
WaitForSingleobject, 151, 285, 296, 300, 301
WaitNarnedPipe, 288
waitpid, 126
WAN (Wide Area Network), 564
Windows NT
diseño de, 620
arquitectura de, 621
HAL, 621
ejecutivo de, 624
Winsockets, 571
write, 261, 442, 574
WriteConsole, 413
WriteFile, 289, 304, 447
XDR (External Data Representation), 585
tipos de datos en, 588
ZIP, 375
Zombie, 129

Digitalización realizada con propósito académico