

SILBERSCHATZ
GALVIN
GAGNE

Fundamentos de sistemas operativos

SÉPTIMA
EDICIÓN

Mc
Graw
Hill

Fundamentos de sistemas operativos

Séptima edición

Fundamentos de sistemas operativos

Séptima edición

ABRAHAM SILBERSCHATZ
Yale University

PETER BAER GALVIN
Corporate Technologies, Inc.

GREG GAGNE
Westminster College

Traducción
VUELAPLUMA, S. L.

Revisión técnica
JESÚS SÁNCHEZ ALLENDE
Doctor Ingeniero de Telecomunicación
Dpto. de Electrónica y Sistemas
Universidad Alfonso X El Sabio



MADRID • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA
MÉXICO • NUEVA YORK • PANAMÁ • SAN JUAN • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI • PARÍS
SAN FRANCISCO • SIDNEY • SINGAPUR • ST. LOUIS • TOKIO • TORONTO

La información contenida en este libro procede de una obra original publicada por John Wiley & Sons, Inc. No obstante, McGraw-Hill/Interamericana de España no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

FUNDAMENTOS DE SISTEMAS OPERATIVOS, 7^a EDICIÓN

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por 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.

 McGraw-Hill / Interamericana
de España, S. A. U.

DERECHOS RESERVADOS © 2006, respecto a la séptima edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S. A. U.
Edificio Valrealty, 1^a planta
Basauri, 17
28023 Aravaca (Madrid)

<http://www.mcgraw-hill.es>
universidad@mcgraw-hill.com

Traducido de la séptima edición en inglés de
OPERATING SYSTEM CONCEPTS
ISBN: 0-471-69466-5

Copyright © 2005 por John Wiley & Sons, Inc.

ISBN: 84-481-4641-7
Depósito legal: M. 7.957-2006

Editor: Carmelo Sánchez González
Compuesto por: Vuelapluma, S. L.
Impreso en: Cofás, S. A.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

A mis hijos, Lemor, Sivan y Aaron

Avi Silberschatz

*A mi esposa Carla,
y a mis hijos, Gwen Owen y Maddie*

Peter Baer Galvin

*A la memoria del tío Sonny,
Robert Jon Heileman 1933 – 2004*

Greg Gagne

Abraham Silberschatz es catedrático de Informática en la Universidad de Yale. Antes de entrar en esa universidad, fue vicepresidente del laboratorio Information Sciences Research Center de Bell Laboratories, Murray Hill, New Jersey, Estados Unidos. Con anterioridad, fue profesor en el Departamento de Ciencias de la Computación de la Universidad de Texas, en Austin. Entre sus intereses de investigación se incluyen los sistemas operativos, los sistemas de bases de datos, los sistemas de tiempo real, los sistemas de almacenamiento, la gestión de red y los sistemas distribuidos.

Además de los puestos que ha ocupado dentro del mundo académico e industrial, el profesor Silberschatz ha sido miembro del Panel de Biodiversidad y Ecosistemas del Comité de Asesores Científicos y Tecnológicos del Presidente Clinton, además de ser Consejero de la organización National Science Foundation y consultor para diversas empresas industriales.

El profesor Silberschatz es socio de ACM y del IEEE. En 2002, recibió el premio IEEE Taylor L. Booth Education Award, en 1998 el ACM Karl V. Karlstrom Outstanding Educator Award y en 1997 el ACM SIGMOND Contribution Award; también ha sido galardonado por la organización IEEE Computer Society por el artículo "Capability Manager" que apareció en la revista *IEEE Transactions on Software Engineering*. Sus escritos han sido publicados en numerosas revistas de ACM y del IEEE, además de en otras revistas y conferencias de carácter profesional. Es coautor del libro de texto *Fundamentos de bases de datos* publicado también en castellano por McGraw-Hill.

Greg Gagne es jefe del departamento de Informática y Matemáticas de Westminster College en Salt Lake City (Estados Unidos), donde ha estado impartiendo clases desde 1990. Además de enseñar sistemas operativos, también imparte clases sobre redes informáticas, sistemas distribuidos, programación orientada a objetos y estructuras de datos. También imparte seminarios a profesores de informática y profesionales de la industria. Los intereses de investigación actuales del profesor Gagne incluyen los sistemas operativos de nueva generación y la informática distribuida.

Peter Baer Galvin es Director técnico de Corporate Technologies (www.cptech.com). Con anterioridad, Peter era administrador de sistemas del Departamento de Informática de la Universidad de Brown. También es editor asociado de la revista *SysAdmin*. Peter Galvin ha escrito artículos para *Byte* y otras revistas y, anteriormente, se encargaba de las columnas sobre seguridad y administración de sistemas en *ITWorld*. Como consultor y formador, Peter ha impartido numerosas conferencias y cursos en todo el mundo sobre seguridad y administración de sistemas.

Prefacio

Los sistemas operativos son una parte esencial de cualquier sistema informático. Del mismo modo, un curso sobre sistemas operativos es una parte esencial de cualquier carrera de Informática. Este campo está cambiando muy rápidamente, ya que ahora las computadoras se encuentran prácticamente en cualquier aplicación, desde juegos para niños hasta herramientas de planificación extremadamente sofisticadas para los gobiernos y las grandes multinacionales. Sin embargo, los conceptos fundamentales siguen siendo bastante claros y es en ellos en los que se basa este libro.

Hemos escrito esta obra como libro de texto para un curso de introducción a los sistemas operativos para estudiantes universitarios de primer y segundo ciclo. Esperamos asimismo que los profesionales también lo encuentren útil, ya que proporciona una clara descripción de los *conceptos* que subyacen a los sistemas operativos. Como prerrequisitos, suponemos que el lector está familiarizado con las estructuras de datos básicas, la organización de una computadora y algún lenguaje de alto nivel, como por ejemplo C. En el Capítulo 1 se han incluido los conceptos sobre hardware necesarios para poder comprender los sistemas operativos. En los ejemplos de código se ha utilizado fundamentalmente el lenguaje C, con algo de Java, pero el lector podrá comprender los algoritmos sin tener un conocimiento profundo de estos lenguajes.

Los conceptos se presentan mediante descripciones intuitivas. El libro aborda algunos resultados teóricos importantes, aunque las demostraciones se han omitido. Las notas bibliográficas proporcionan información sobre los libros o artículos de investigación en la que aparecieron y se demostraron los conceptos por primera vez, así como referencias a material de lectura adicional. En lugar de demostraciones, se han utilizado figuras y ejemplos para indicar por qué debemos esperar que el resultado en cuestión sea cierto.

Los conceptos y algoritmos fundamentales cubiertos en el libro están basados, a menudo, en aquéllos que se emplean en los sistemas operativos comerciales existentes. Nuestro objetivo ha sido presentar estos conceptos y algoritmos para una configuración general que no estuviera ligada a un sistema operativo concreto. Hemos presentado gran cantidad de ejemplos que pertenecen a los más populares e innovadores sistemas operativos, incluyendo Solaris de Sun Microsystems; Linux; Mach; MS-DOS, Windows NT, Windows 2000 y Windows XP de Microsoft; VMS y TOPS-20 de DEC; OS/2 de IBM y Mac OS X de Apple.

En este texto, cuando utilizamos Windows XP como sistema de operativo de ejemplo, queremos hacer referencia tanto a Windows XP como a Windows 2000. Si existe una característica en Windows XP que no está disponible en Windows 2000, se indica de forma explícita. Si una característica existe en Windows 2000 pero no en Windows XP, entonces hacemos referencia específicamente a Windows 2000.

Organización de este libro

La organización de este texto refleja nuestros muchos años de impartición de cursos sobre sistemas operativos. También hemos tenido en cuenta las aportaciones proporcionadas por los revisores del texto, así como los comentarios enviados por los lectores de las ediciones anteriores. Además, el contenido del texto se corresponde con las sugerencias de la recomendación *Computing Curricula 2001* para la enseñanza de sistemas operativos, publicado por la Joint Task Force de la sociedad IEEE Computing Society y la asociación ACM (Association for Computing Machinery).

En la página web en inglés complementaria de este texto proporcionamos diversos planes de estudios que sugieren varios métodos para usar el texto, tanto en cursos de introducción como avanzados sobre sistemas operativos. Como regla general, animamos a los lectores a avanzar de forma secuencial en el estudio de los sistemas operativos. Sin embargo, utilizando un plan de estudios de ejemplo, un lector puede seleccionar un orden diferente de los capítulos (o de las subsecciones de los capítulos).

Contenido del libro

El texto está organizado en ocho partes:

- **Introducción.** Los Capítulos 1 y 2 explican lo que *son* los sistemas operativos, lo que *hacen* y cómo se *diseñan* y *construyen*. Se explican las características comunes de un sistema operativo, lo que un sistema operativo hace por el usuario y lo que hace para el operador del sistema informático. La presentación es de naturaleza motivadora y explicativa. En estos capítulos hemos evitado exponer cómo ocurren las cosas internamente. Por tanto, son adecuados para lectores individuales o para estudiantes de primer ciclo que deseen aprender qué es un sistema operativo sin entrar en los detalles de los algoritmos internos.
- **Gestión de procesos.** Los Capítulos 3 a 7 describen los conceptos de proceso y de concurrencia como corazón de los sistemas operativos modernos. Un *proceso* es la unidad de trabajo de un sistema. Un sistema consta de una colección de procesos que se ejecutan *concurrentemente*, siendo algunos de ellos procesos del sistema operativo (aquellos que ejecutan código del sistema) y el resto, procesos de usuario (aquellos que ejecutan código del usuario). Estos capítulos exponen los métodos para la sincronización de procesos, la comunicación interprocesos y el tratamiento de los interbloqueos. También se incluye dentro de este tema una explicación sobre las hebras.
- **Gestión de memoria.** Los Capítulos 8 y 9 se ocupan de la gestión de la memoria principal durante la ejecución de un proceso. Para mejorar tanto la utilización de la CPU como su velocidad de respuesta a los usuarios, la computadora debe mantener varios procesos en memoria. Existen muchos esquemas diferentes de gestión de memoria, que se reflejan en diversas técnicas de gestión de memoria. Además, la efectividad de cada algoritmo concreto depende de la situación.
- **Gestión de almacenamiento.** Los Capítulos 10 a 13 describen cómo se gestionan el sistema de archivos, el almacenamiento masivo y las operaciones de E/S en un sistema informático moderno. El sistema de archivos proporciona el mecanismo para el almacenamiento y el acceso en línea a los datos y programas que residen en los discos. Estos capítulos describen los algoritmos y estructuras clásicos internos para gestión del almacenamiento. Proporcionan un firme conocimiento práctico de los algoritmos utilizados, indicando las propiedades, las ventajas y las desventajas. Puesto que los dispositivos de E/S que se conectan a una computadora son muy variados, el sistema operativo tiene que proporcionar un amplio rango de funcionalidad a las aplicaciones, para permitirlas controlar todos los aspectos de los dispositivos. Se expone en profundidad la E/S del sistema, incluyendo el diseño del sistema de E/S, las interfaces y las estructuras y funciones internas del sistema. En muchos sentidos, los dispositivos de E/S también constituyen los componentes más lentos

de la computadora. Puesto que son un cuello de botella que limita las prestaciones del sistema, abordamos también estos problemas de prestaciones. También se explican los temas relativos a los almacenamientos secundario y terciario.

- **Protección y seguridad.** Los Capítulos 14 y 15 se ocupan de los procesos de un sistema operativo que deben protegerse frente a otras actividades. Con propósitos de protección y seguridad, utilizamos mecanismos que garanticen que sólo los procesos que hayan obtenido la apropiada autorización del sistema operativo puedan operar sobre los archivos, la memoria, la CPU y otros recursos. La protección es un mecanismo que permite controlar el acceso de procesos, programas o usuarios a los recursos definidos por el sistema informático. Este mecanismo debe proporcionar un medio de especificar los controles que se han de imponer, así como un medio de imponerlos. Los mecanismos de seguridad protegen la información almacenada en el sistema (tanto los datos como el código) y los recursos físicos del sistema informáticos frente a accesos no autorizados, destrucción o modificación maliciosas e introducción accidental de incoherencias.
- **Sistemas distribuidos.** Los Capítulos 16 a 18 se ocupan de una colección de procesadores que no comparten la memoria, ni tampoco un reloj: un *sistema distribuido*. Proporcionando al usuario acceso a los diversos recursos que mantiene, un sistema distribuido puede aumentar la velocidad de cálculo y la disponibilidad y fiabilidad de los datos. Una arquitectura de este tipo también proporciona al usuario un sistema de archivos distribuido, que es un sistema del servicio de archivos cuyos usuarios, servidores y dispositivos de almacenamiento se encuentran distribuidos por los distintos nodos de un sistema distribuido. Un sistema distribuido debe proporcionar diversos mecanismos para la sincronización y comunicación de procesos y para tratar el problema de los interbloqueos y diversos tipos de fallos que no aparecen en los sistemas centralizados.
- **Sistemas de propósito especial.** Los Capítulos 19 y 20 se ocupan de los sistemas utilizados para propósitos específicos, incluyendo los sistemas de tiempo real y los sistemas multimedia. Estos sistemas tienen requisitos específicos que difieren de los de los sistemas de propósito general, en los que se centra el resto del texto. Los sistemas de tiempo real pueden necesitar no sólo que los resultados calculados sean "correctos", sino que también se obtengan dentro de un plazo de tiempo especificado. Los sistemas multimedia requieren garantías de calidad de servicio, para garantizar que los datos multimedia se entreguen a los clientes dentro de un periodo de tiempo específico.
- **Casos de estudio.** Los Capítulos 21 a 23 del libro y los Apéndices A y C (disponibles en inglés en el sitio web), integran los conceptos descritos en el libro abordando el estudio de sistemas operativos reales. Entre estos sistemas se incluyen Linux, Windows XP, FreeBSD, Mach y Windows 2000. Hemos elegido Linux y FreeBSD porque UNIX (en tiempos) resultaba lo suficientemente pequeño como para resultar comprensible, sin llegar a ser un sistema operativo "de juguete". La mayor parte de sus algoritmos internos fueron seleccionados por su *simplicidad*, en lugar de por su velocidad o su sofisticación. Tanto Linux como FreeBSD pueden obtenerse fácilmente y a un bajo coste, por lo que puede que muchos estudiantes tengan acceso a estos sistemas. Hemos seleccionado Windows XP y Windows 2000 porque nos proporcionan una oportunidad de estudiar un sistema operativo moderno con un diseño y una implementación drásticamente diferentes a los de UNIX. El Capítulo 23 describe de forma breve algunos otros sistemas operativos que han tenido históricamente cierta influencia.

Entornos de sistemas operativos

Este libro utiliza ejemplos de muchos sistemas operativos del mundo real para ilustrar los conceptos fundamentales. Sin embargo, se ha puesto una atención especial en la familia de sistemas operativos de Microsoft (incluyendo Windows NT, Windows 2000 y Windows XP) y varias versiones de UNIX (Solaris, BSD y Mac). También hemos cubierto una parte significativa del sistema opera-

tivo Linux, utilizando la versión más reciente del kernel, que era la versión 2.6 en el momento de escribir este libro.

El texto también proporciona varios programas de ejemplo escritos en C y en Java. Estos programas pueden ejecutarse en los siguientes entornos de programación:

- **Sistemas Windows.** El entorno de programación principal para los sistemas Windows es la API (application programming interface) Win32, que proporciona un amplio conjunto de funciones para gestionar los procesos, las hebras, la memoria y los dispositivos periféricos. Facilitamos varios programas en C que ilustran el uso de la API Win32. Los programas de ejemplo se han probado en sistemas Windows 2000 y Windows XP.
- **POSIX.** POSIX (Portable Operating System Interface, interfaz portable de sistema operativo) representa un conjunto de estándares implementados principalmente en sistemas operativos basados en UNIX. Aunque los sistemas Windows XP y Windows 2000 también pueden ejecutar ciertos programas POSIX, aquí nos hemos ocupado de POSIX principalmente centrandonos en los sistemas UNIX y Linux. Los sistemas compatibles con POSIX deben implementar el estándar fundamental de POSIX (POSIX.1); Linux, Solaris y Mac son ejemplos de sistemas compatibles con POSIX. POSIX también define varias extensiones de los estándares, incluyendo extensiones de tiempo real (POSIX.1.b) y una extensión para una biblioteca de hebras (POSIX.1.c, más conocida como Pthreads). Proporcionamos también varios ejemplos de programación escritos en C con el fin de ilustrar la API básica de POSIX, así como Pthreads y las extensiones para programación de tiempo real. Estos programas de ejemplo se han probado en sistemas Debian Linux 2.4 y 2.6, Mac OS X y Solaris 9 utilizando el compilador gcc 3.3.
- **Java.** Java es un lenguaje de programación ampliamente utilizado con una rica API y soporte integrado de lenguaje para la creación y gestión de hebras. Los programas Java se ejecutan sobre cualquier sistema operativo que permita el uso de una máquina virtual Java (JVM, Java virtual machine). Hemos ilustrado los diversos sistemas operativos y los conceptos de redes con varios programas Java que se han probado utilizando la JVM Java 1.4.

Hemos elegido tres entornos de programación porque, en nuestra opinión, son los que mejor representan los dos modelos más populares de sistemas operativos: Windows y UNIX/Linux, junto con el popular entorno Java. La mayor parte de los ejemplos de programación están escritos en C, y esperamos que los lectores se sientan cómodos con este lenguaje; los lectores familiarizados con ambos lenguajes, C y Java, deberían comprender con facilidad la mayor parte de los programas proporcionados en este texto.

En algunos casos, como por ejemplo con la creación de hebras, ilustramos un concepto específico utilizando todos los entornos de programación, lo que permite al lector comparar el modo en que las tres bibliotecas diferentes llevan a cabo una misma tarea. En otras situaciones, sólo hemos empleado una de las API para demostrar un concepto. Por ejemplo, hemos ilustrado el tema de la memoria compartida usando sólo la API de POSIX; la programación de *sockets* en TCP/IP se expone utilizando la API de Java.

La séptima edición

A la hora de escribir la séptima edición de *Fundamentos de sistemas operativos* nos hemos guiado por los muchos comentarios y sugerencias que hemos recibido de los lectores de las ediciones anteriores; también hemos procurado reflejar los cambios tan rápidos que se están produciendo en los campos de los sistemas operativos y las comunicaciones por red. Hemos reescrito el material de la mayor parte de los capítulos, actualizando el material más antiguo y eliminando aquél que actualmente ya no tiene interés o es irrelevante.

Se han hecho importantes revisiones y cambios en la organización de muchos capítulos. Los cambios más destacables son la completa reorganización del material de introducción de los Capítulos 1 y 2 y la adición de dos capítulos nuevos sobre los sistemas de propósito especial (los

sistemas integrados de tiempo real y los sistemas multimedia). Puesto que la protección y la seguridad han cobrado una gran relevancia en los sistemas operativos, hemos decidido incluir el tratamiento de estos temas más al principio del texto. Además, se ha actualizado y ampliado sustancialmente el estudio de los mecanismos de seguridad.

A continuación se proporciona un breve esquema de los principales cambios introducidos en los distintos capítulos:

- El **Capítulo 1, Introducción**, se ha revisado por completo. En las versiones anteriores, el capítulo abordaba desde un punto de vista histórico el desarrollo de los sistemas operativos. Ahora, el capítulo proporciona una exhaustiva introducción a los componentes de un sistema operativo, junto con información básica sobre la organización de un sistema informático.
- El **Capítulo 2, Estructuras de los sistemas operativos**, es una versión revisada del antiguo Capítulo 3 en la que se ha incluido información adicional, como una exposición mejorada sobre las llamadas al sistema y la estructura del sistema operativo. También se proporciona información actualizada sobre las máquinas virtuales.
- El **Capítulo 3, Procesos**, es el antiguo Capítulo 4. Incluye nueva información sobre cómo se representan los procesos en Linux e ilustra la creación de procesos usando las API de POSIX y Win32. El material dedicado a la memoria compartida se ha mejorado mediante un programa que ilustra la API de memoria compartida disponible para los sistemas POSIX.
- El **Capítulo 4, Hebras**, se corresponde con el antiguo Capítulo 5. El capítulo presenta y amplía el tema de las bibliotecas de hebras, incluyendo las bibliotecas de hebras de POSIX, la API Win32 y Java. También proporciona información actualizada sobre las hebras en Linux.
- El **Capítulo 5, Planificación**, es el antiguo Capítulo 6. El capítulo ofrece una exposición significativamente actualizada sobre temas de planificación en sistemas multiprocesador, incluyendo los algoritmos de afinidad del procesador y de equilibrado de carga. También se ha incluido una nueva sección sobre la planificación de hebras, incluyendo Pthreads e información actualizada sobre la planificación dirigida por tablas en Solaris. La sección dedicada a la planificación en Linux se ha revisado para incluir el planificador utilizado en el *kernel* 2.6.
- El **Capítulo 6, Sincronización de procesos**, se corresponde con el antiguo Capítulo 7. Se han eliminado las soluciones para dos procesos y ahora sólo se explica la solución de Peterson, ya que los algoritmos para dos procesos no garantizan su funcionamiento en los procesadores modernos. El capítulo también incluye nuevas secciones sobre cuestiones de sincronización en el kernel de Linux y en la API de Pthreads.
- El **Capítulo 7, Interbloqueos**, es el antiguo Capítulo 8. El nuevo material incluye un programa de ejemplo que ilustra los interbloqueos en un programa multihebra Pthread.
- El **Capítulo 8, Memoria principal**, se corresponde con el antiguo Capítulo 9. El capítulo ya no se ocupa del tema de las secciones de memoria superpuestas. Además, el material dedicado a la segmentación se ha modificado considerablemente, incluyendo una explicación mejorada sobre la segmentación en los sistemas Pentium y una exposición sobre el diseño en Linux para tales sistemas segmentados.
- El **Capítulo 9, Memoria virtual**, es el antiguo Capítulo 10. El capítulo amplía el material dedicado a la memoria virtual, así como a los archivos mapeados en memoria, incluyendo un ejemplo de programación que ilustra la memoria compartida (a través de los archivos mapeados en memoria) usando la API Win32. Los detalles sobre el hardware de gestión de memoria se han actualizado. Se ha añadido una nueva sección dedicada a la asignación de memoria dentro del *kernel*, en la que se explican el algoritmo de descomposición binaria y el asignador de franjas.

- El **Capítulo 10, Interfaz del sistema de archivos**, se corresponde con el antiguo Capítulo 11. Se ha actualizado y se ha incluido un ejemplo sobre las listas ACL de Windows XP.
- El **Capítulo 11, Implementación de los sistemas de archivos**, es el antiguo Capítulo 12. Se ha añadido una descripción completa sobre el sistema de archivos WAFL y se ha incluido el sistema de archivos ZFS de Sun.
- El **Capítulo 12, Estructura de almacenamiento masivo**, se corresponde con el antiguo Capítulo 14. Se ha añadido un tema nuevo, las matrices de almacenamiento modernas, incluyendo la nueva tecnología RAID y características tales como las redes de área de almacenamiento.
- El **Capítulo 13, Sistemas de E/S**, es el antiguo Capítulo 13 actualizado, en el que se ha incluido material nuevo.
- El **Capítulo 14, Protección**, es el antiguo Capítulo 18, actualizado con información sobre el principio de mínimo privilegio.
- El **Capítulo 15, Seguridad**, se corresponde con el antiguo Capítulo 19. El capítulo ha experimentado una revisión importante, habiéndose actualizado todas las secciones. Se ha incluido un ejemplo completo sobre el ataque por desbordamiento de búfer y se ha ampliado la información sobre herramientas de seguridad, cifrado y hebras.
- Los **Capítulos 16 a 18** se corresponden con los antiguos Capítulos 15 a 17, y se han actualizado con material nuevo.
- El **Capítulo 19, Sistemas de tiempo real**, es un capítulo nuevo centrado en los sistemas informáticos integrados de tiempo real, sistemas que tienen requisitos diferentes a los de los sistemas tradicionales. El capítulo proporciona una introducción a los sistemas informáticos de tiempo real y describe cómo deben construirse estos sistemas operativos para cumplir los plazos de temporización estrictos de estos sistemas.
- El **Capítulo 20, Sistemas multimedia**, es un capítulo nuevo que detalla los desarrollos en el área, relativamente nueva, de los sistemas multimedia. Los datos multimedia se diferencian de los datos convencionales en que los primeros, como por ejemplo imágenes de vídeo, deben suministrarse de acuerdo con ciertas restricciones de tiempo. El capítulo explora cómo afectan estos requisitos al diseño de los sistemas operativos.
- El **Capítulo 21, El sistema Linux** se corresponde con el antiguo Capítulo 20. Se ha actualizado para reflejar los cambios en el *kernel* 2.6, el *kernel* más reciente en el momento de escribir este libro.
- El **Capítulo 22, Windows XP**, se ha actualizado también en esta versión.
- El **Capítulo 23, Sistemas operativos influyentes**, se ha actualizado.

El antiguo Capítulo 21 (Windows 2000) se ha transformado en el Apéndice C. Al igual que en las versiones anteriores, los apéndices están disponibles en línea, aunque solamente en su versión original en inglés.

Proyectos y ejercicios de programación

Para reforzar los conceptos presentados en el texto, hemos añadido diversos proyectos y ejercicios de programación que utilizan las API de POSIX y Win32, así como Java. Hemos añadido 15 nuevos ejercicios de programación que se centran en los procesos, las hebras, la memoria compartida, la sincronización de procesos y las comunicaciones por red. Además, hemos añadido varios proyectos de programación que son más complicados que los ejercicios de programación estándar. Estos proyectos incluyen la adición de una llamada al sistema al *kernel* de Linux, la creación de una *shell* de UNIX utilizando la llamada al sistema *fork()*, una aplicación multihebra de tratamiento de matrices y el problema del productor-consumidor usando memoria compartida.

Suplementos y página web

La página web (en inglés) asociada al libro contiene material organizado como un conjunto de diapositivas para acompañar al libro, planes de estudios para el curso modelo, todo el código fuente en C y Java y una relación actualizada de las erratas. La página web también contiene los apéndices de los tres casos de estudio del libro y un apéndice dedicado a la comunicación distribuida. La URL es:

<http://www.os-book.com>

Lista de correo

Hemos cambiado el sistema de comunicación entre los usuarios de *Fundamentos de sistemas operativos*. Si desea utilizar esta función, visite la siguiente URL y siga las instrucciones para suscribirse:

<http://mailman.cs.yale.edu/mailman/listinfo/os-book-list>

El sistema de lista de correo proporciona muchas ventajas, como un archivo de mensajes y varias opciones de suscripción, incluyendo suscripciones con envío de resúmenes o con acceso Web. Para mandar mensajes a la lista, envíe un mensaje de correo electrónico a:

os-book-list@cs.yale.edu

Dependiendo del mensaje, le responderemos personalmente o enviaremos el mensaje a toda la lista de correo. La lista está moderada, por lo que no recibirá correo inapropiado.

Los estudiantes que utilicen este libro como texto en sus clases no deben utilizar la lista para preguntar las respuestas a los ejercicios, ya que no se les facilitarán.

Sugerencias

Hemos intentado eliminar todos los errores en esta nueva edición pero, como ocurre con los sistemas operativos, es posible que queden algunas erratas. Agradeceríamos que nos comunicaran cualquier error u omisión en el texto que puedan identificar.

Si desea sugerir mejoras o contribuir con ejercicios, también se lo agradecemos de antemano. Puede enviar los mensajes a os-book@cs.yale.edu.

Agradecimientos

Este libro está basado en las ediciones anteriores y James Peterson fue coautor de la primera de ellas. Otras personas que ayudaron en las ediciones anteriores son Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Racsit Eskicioğlu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Bruce Hillyer, Mark Holliday, Ahmed Kamel, Richard Kieburz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Yoichi Muraoka, Jim M. Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles Qualline, John Quarterman, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J. C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L. Wear, John Werth, James M. Westall, J. S. Weston y Yang Xiang.

Partes del Capítulo 12 están basadas en un artículo de Hillyer y Silberschatz [1996]. Partes del Capítulo 17 están basadas en un artículo de Levy y Silberschatz [1990]. El Capítulo 21 está basado en un manuscrito no publicado de Stephen Tweedie. El Capítulo 22 está basado en un manus-

crito no publicado de Dave Probert, Cliff Martin y Avi Silberschatz. El Apéndice C está basado en un manuscrito no publicado de Cliff Martin. Cliff Martin también ha ayudado a actualizar el apéndice sobre UNIX para cubrir FreeBSD. Mike Shapiro, Bryan Cantrill y Jim Mauro nos han ayudado respondiendo a diversas cuestiones relativas a Solaris. Josh Dees y Rob Reynolds han contribuido al tratamiento de Microsoft .NET. El proyecto de diseño y mejora de la interfaz *shell* de UNIX ha sido aportado por John Trono de St. Michael's College, Winooski, Vermont.

Esta edición contiene muchos ejercicios nuevos, con sus correspondientes soluciones, los cuales han sido proporcionadas por Arvind Krishnamurthy.

Queremos dar las gracias a las siguientes personas que revisaron esta versión del libro: Bart Childs, Don Heller, Dean Hougen Michael Huangs, Morty Kewstel, Euripides Montagne y John Sterling.

Nuestros editores, Bill Zobrist y Paul Crockett, han sido una experta guía a medida que íbamos preparando esta edición. Simon Durkin, ayudante de nuestros editores, ha gestionado muy amablemente muchos de los detalles de este proyecto. El editor de producción senior ha sido Ken Santor. Susan Cyr ha realizado las ilustraciones de la portada y Madelyn Lesure es la diseñadora de la misma. Beverly Peavler ha copiado y maquetado el manuscrito. Katrina Avery es la revisora de estilo externa que ha realizado la corrección de estilo del texto y la realización del índice ha corrido a cargo de la colabora externa Rosemary Simpson. Marilyn Turnamian ha ayudado a generar las figuras y las diapositivas de presentación.

Por último, nos gustaría añadir algunas notas personales. Avi está comenzando un nuevo capítulo en su vida, volviendo al mundo universitario e iniciando una relación con Valerie. Esta combinación le ha proporcionado la tranquilidad de espíritu necesaria para centrarse en la escritura de este libro. Pete desea dar las gracias a su familia, a sus amigos y sus compañeros de trabajo por su apoyo y comprensión durante el desarrollo del proyecto. Greg quiere agradecer a su familia su continuo interés y apoyo. Sin embargo, desea dar las gracias de modo muy especial a su amigo Peter Ormsby que, no importa lo ocupado que parezca estar, siempre pregunta en primer lugar “¿Qué tal va la escritura del libro?”.

Abraham Silberschatz, New Haven, CT, 2004

Peter Baer Galvin, Burlington, MA, 2004

Greg Gagne, Salt Lake City, UT, 2004

Contenido

PARTE UNO ■ INTRODUCCIÓN

Capítulo 1 Introducción

1.1	¿Qué hace un sistema operativo?	3	1.9	Protección y seguridad	24
1.2	Organización de una computadora	6	1.10	Sistemas distribuidos	25
1.3	Arquitectura de un sistema informático	11	1.11	Sistemas de propósito general	26
1.4	Estructura de un sistema operativo	14	1.12	Entornos informáticos	28
1.5	Operaciones del sistema operativo	16	1.13	Resumen	31
1.6	Gestión de procesos	19		Ejercicios	32
1.7	Gestión de memoria	19		Notas bibliográficas	34
1.8	Gestión de almacenamiento	20			

Capítulo 2 Estructuras de sistemas operativos

2.1	Servicios del sistema operativo	35	2.7	Estructura del sistema operativo	52
2.2	Interfaz de usuario del sistema operativo	37	2.8	Máquinas virtuales	58
2.3	Llamadas al sistema	39	2.9	Generación de sistemas operativos	63
2.4	Tipos de llamadas al sistema	42	2.10	Arranque del sistema	64
2.5	Programas del sistema	49	2.11	Resumen	65
2.6	Diseño e implementación del sistema operativo	50		Ejercicios	65
				Notas bibliográficas	70

PARTE DOS ■ GESTIÓN DE PROCESOS

Capítulo 3 Procesos

3.1	Concepto de proceso	73	3.6	Comunicación en los sistemas cliente-servidor	96
3.2	Planificación de procesos	77	3.7	Resumen	103
3.3	Operaciones sobre los procesos	81		Ejercicios	104
3.4	Comunicación interprocesos	86		Notas bibliográficas	111
3.5	Ejemplos de sistemas ipc	92			

Capítulo 4 Hebras

4.1	Introducción	113	4.5	Ejemplos de sistemas operativos	128
4.2	Modelos multihebra	115	4.6	Resumen	130
4.3	Bibliotecas de hebras	117		Ejercicios	130
4.4	Consideraciones sobre las hebras	123		Notas bibliográficas	135

Capítulo 5 Planificación de la CPU

5.1	Conceptos básicos 137	5.6	Ejemplos de sistemas operativos 156
5.2	Criterios de planificación 140	5.7	Evaluación de algoritmos 161
5.3	Algoritmos de planificación 142	5.8	Resumen 165
5.4	Planificación de sistemas multiprocesador 151		Ejercicios 166
5.5	Planificación de hebras 153		Notas bibliográficas 169

Capítulo 6 Sincronización de procesos

6.1	Fundamentos 171	6.7	Monitores 186
6.2	El problema de la sección crítica 173	6.8	Ejemplos de sincronización 194
6.3	Solución de Peterson 174	6.9	Transacciones atómicas 197
6.4	Hardware de sincronización 175	6.10	Resumen 204
6.5	Semáforos 178		Ejercicios 205
6.6	Problemas clásicos de sincronización 182		Notas bibliográficas 214

Capítulo 7 Interbloqueos

7.1	Modelo de sistema 217	7.6	Detección de interbloqueos 232
7.2	Caracterización de los interbloqueos 219	7.7	Recuperación de un interbloqueo 235
7.3	Métodos para tratar los interbloqueos 223	7.8	Resumen 236
7.4	Prevención de interbloqueos 224		Ejercicios 237
7.5	Evasión de interbloqueos 226		Notas bibliográficas 239

PARTE TRES ■ GESTIÓN DE MEMORIA

Capítulo 8 Memoria principal

8.1	Fundamentos 243	8.6	Segmentación 269
8.2	Intercambio 249	8.7	Ejemplo: Intel Pentium 271
8.3	Asignación de memoria contigua 252	8.8	Resumen 275
8.4	Paginación 255		Ejercicios 276
8.5	Estructura de la tabla de páginas 264		Notas bibliográficas 278

Capítulo 9 Memoria virtual

9.1	Fundamentos 279	9.8	Asignación de la memoria del kernel 314
9.2	Paginación bajo demanda 282	9.9	Otras consideraciones 317
9.3	Copia durante la escritura 289	9.10	Ejemplos de sistemas operativos 323
9.4	Sustitución de páginas 290	9.11	Resumen 325
9.5	Asignación de marcos 302		Ejercicios 326
9.6	Sobrepaginación 305		Notas bibliográficas 330
9.7	Archivos mapeados en memoria 309		

PARTE CUATRO ■ GESTIÓN DE ALMACENAMIENTO

Capítulo 10 Interfaz del sistema de archivos

10.1	Concepto de archivo 333	10.6	Protección 361
10.2	Métodos de acceso 342	10.7	Resumen 365
10.3	Estructura de directorios 345		Ejercicios 366
10.4	Montaje de sistemas de archivos 354		Notas bibliográficas 367
10.5	Compartición de archivos 355		

Capítulo 11 Implementación de sistemas de archivos

- | | |
|---|---|
| 11.1 Estructura de un sistema de archivos 369
11.2 Implementación de sistemas de archivos 371
11.3 Implementación de directorios 377
11.4 Métodos de asignación 378
11.5 Gestión del espacio libre 386
11.6 Eficiencia y prestaciones 388
11.7 Recuperación 392 | 11.8 Sistemas de archivos con estructura de registro 393
11.9 NFS 395
11.10 Ejemplo: el sistema de archivos WAFL 400
11.11 Resumen 402
Ejercicios 403
Notas bibliográficas 405 |
|---|---|

Capítulo 12 Estructura de almacenamiento masivo

- | | |
|---|--|
| 12.1 Panorámica de la estructura de almacenamiento masivo 407
12.2 Estructura de un disco 409
12.3 Conexión de un disco 410
12.4 Planificación de disco 412
12.5 Gestión del disco 418
12.6 Gestión de espacio de intercambio 421
12.7 Estructuras RAID 423 | 12.8 Implementación de un almacenamiento estable 432
12.9 Estructura de almacenamiento terciario 433
12.10 Resumen 442
Ejercicios 444
Notas bibliográficas 447 |
|---|--|

Capítulo 13 Sistemas de E/S

- | | |
|---|--|
| 13.1 Introducción 449
13.2 Hardware de E/S 450
13.3 Interfaz de E/S de las aplicaciones 458
13.4 Subsistema de E/S del kernel 464
13.5 Transformación de las solicitudes de E/S en operaciones hardware 471 | 13.6 Streams 473
13.7 Rendimiento 475
13.8 Resumen 478
Ejercicios 478
Notas bibliográficas 479 |
|---|--|

PARTE CINCO ■ PROTECCIÓN Y SEGURIDAD

Capítulo 14 Protección

- | | |
|--|---|
| 14.1 Objetivos de la protección 483
14.2 Principios de la protección 484
14.3 Dominio de protección 485
14.4 Matriz de acceso 489
14.5 Implementación de la matriz de acceso 493
14.6 Control de acceso 495 | 14.7 Revocación de derechos de acceso 496
14.8 Sistemas basados en capacidades 497
14.9 Protección basada en el lenguaje 500
14.10 Resumen 505
Ejercicios 505
Notas bibliográficas 506 |
|--|---|

Capítulo 15 Seguridad

- | | |
|--|--|
| 15.1 El problema de la seguridad 509
15.2 Amenazas relacionadas con los programas 513
15.3 Amenazas del sistema y de la red 520
15.4 La criptografía como herramienta de seguridad 525
15.5 Autenticación de usuario 535
15.6 Implementación de defensas de seguridad 539 | 15.7 Cortafuegos para proteger los sistemas y redes 546
15.8 Clasificaciones de seguridad informática 547
15.9 Un ejemplo: Windows XP 549
15.10 Resumen 550
Ejercicios 551
Notas bibliográficas 552 |
|--|--|

PARTE SEIS ■ SISTEMAS DISTRIBUIDOS

Capítulo 16 Estructuras de los sistemas distribuidos

16.1	Motivación	557	16.7	Robustez	575
16.2	Tipos de sistemas distribuidos	559	16.8	Cuestiones de diseño	577
16.3	Estructura de una red	563	16.9	Un ejemplo: conexión en red	579
16.4	Topología de red	565	16.10	Resumen	581
16.5	Estructura de comunicaciones	567		Ejercicios	581
16.6	Protocolos de comunicaciones	572		Notas bibliográficas	583

Capítulo 17 Sistemas de archivos distribuidos

17.1	Conceptos esenciales	585	17.6	Un ejemplo: AFS	598
17.2	Nombrado y transparencia	586	17.7	Resumen	602
17.3	Acceso remoto a archivos	590		Ejercicios	603
17.4	Servicios con y sin memoríes del estado	594		Notas bibliográficas	604
17.5	Replicación de archivos	597			

Capítulo 18 Coordinación distribuida

18.1	Ordenación de sucesos	605	18.6	Algoritmos de elección	623
18.2	Exclusión mutua	607	18.7	Procedimientos de acuerdo	625
18.3	Atomicidad	610	18.8	Resumen	627
18.4	Control de concurrencia	612		Ejercicios	627
18.5	Gestión de interbloqueos	616		Notas bibliográficas	629

PARTE SIETE ■ SISTEMAS DE PROPÓSITO ESPECIAL

Capítulo 19 Sistemas de tiempo real

19.1	Introducción	633	19.5	Planificación de la CPU en tiempo real	641
19.2	Características del sistema	634	19.6	VxWorks 5.x	645
19.3	Características de un <i>kernel</i> de tiempo real	636	19.7	Resumen	648
19.4	Implementación de sistemas operativos de tiempo real	637		Ejercicios	649
				Notas bibliográficas	650

Capítulo 20 Sistemas multimedia

20.1	¿Qué es la multimedia?	651	20.6	Gestión de red	660
20.2	Compresión	654	20.7	Un ejemplo: CineBlitz	663
20.3	Requisitos de los <i>kernels</i> multimedia	655	20.8	Resumen	665
20.4	Planificación de la CPU	658		Ejercicios	666
20.5	Planificación de disco	658		Notas bibliográficas	667

PARTE OCHO ■ CASOS DE ESTUDIO

Capítulo 21 El sistema Linux

21.1	Historia de Linux	671	21.5	Planificación	684
21.2	Principios de diseño	675	21.6	Gestión de memoria	688
21.3	Módulos del kernel	678	21.7	Sistemas de archivos	696
21.4	Gestión de procesos	681	21.8	Entrada y salida	702

21.9	Comunicación interprocesos	704
21.10	Estructura de red	705
21.11	Seguridad	707
21.12	Resumen	709
	Ejercicios	710
	Notas bibliográficas	711

Capítulo 22 Windows XP

22.1	Historia	713
22.2	Principios de diseño	714
22.3	Componentes del sistema	717
22.4	Subsistemas de entorno	739
22.5	Sistema de archivos	742
22.6	Conexión de red	749
22.7	Interfaz de programación	756
22.8	Resumen	763
	Ejercicios	763
	Notas bibliográficas	764

Capítulo 23 Sistemas operativos influyentes

23.1	Sistemas pioneros	765
23.2	Atlas	771
23.3	XDS-940	771
23.4	THE	772
23.5	RC 4000	773
23.6	CTSS	773
23.7	MULTICS	.774
23.8	IBM OS/360	774
23.9	Mach	776
23.10	Otros sistemas	777
	Ejercicios	777

Bibliografía 779

Créditos 803

Índice 805

Parte Uno

Introducción

Un *sistema operativo* actúa como un intermediario entre el usuario de una computadora y el hardware de la misma. El propósito de un sistema operativo es proporcionar un entorno en el que el usuario pueda ejecutar programas de una manera *práctica y eficiente*.

Un sistema operativo es software que gestiona el hardware de la computadora. El hardware debe proporcionar los mecanismos apropiados para asegurar el correcto funcionamiento del sistema informático e impedir que los programas de usuario interfieran con el apropiado funcionamiento del sistema.

Internamente, los sistemas operativos varían enormemente en lo que se refiere a su configuración, dado que están organizados según muchas líneas diferentes. El diseño de un nuevo sistema operativo es una tarea de gran envergadura. Es fundamental que los objetivos del sistema estén bien definidos antes de comenzar el diseño. Estos objetivos constituyen la base para elegir entre los distintos algoritmos y estrategias.

Dado que un sistema operativo es un software grande y complejo, debe crearse pieza por pieza. Cada una de estas piezas debe ser una parte perfectamente perfilada del sistema, estando sus entradas, salidas y funciones cuidadosamente definidas.

Introducción

Un **sistema operativo** es un programa que administra el hardware de una computadora. También proporciona las bases para los programas de aplicación y actúa como un intermediario entre el usuario y el hardware de la computadora. Un aspecto sorprendente de los sistemas operativos es la gran variedad de formas en que llevan a cabo estas tareas. Los sistemas operativos para *mainframe* están diseñados principalmente para optimizar el uso del hardware. Los sistemas operativos de las computadoras personales (PC) soportan desde complejos juegos hasta aplicaciones de negocios. Los sistemas operativos para las computadoras de mano están diseñados para proporcionar un entorno en el que el usuario pueda interactuar fácilmente con la computadora para ejecutar programas. Por tanto, algunos sistemas operativos se diseñan para ser *prácticos*, otros para ser *eficientes* y otros para ser ambas cosas.

Antes de poder explorar los detalles de funcionamiento de una computadora, necesitamos saber algo acerca de la estructura del sistema. Comenzaremos la exposición con las funciones básicas del arranque, E/S y almacenamiento. También vamos a describir la arquitectura básica de una computadora que hace posible escribir un sistema operativo funcional.

Dado que un sistema operativo es un software grande y complejo, debe crearse pieza por pieza. Cada una de estas piezas deberá ser una parte bien diseñada del sistema, con entradas, salida y funciones cuidadosamente definidas. En este capítulo proporcionamos una introducción general a los principales componentes de un sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Proporcionar una visión general de los principales componentes de los sistemas operativos.
- Proporcionar una panorámica sobre la organización básica de un sistema informático.

1.1 ¿Qué hace un sistema operativo?

Comenzamos nuestra exposición fijándonos en el papel del sistema operativo en un sistema informático global. Un sistema informático puede dividirse a grandes rasgos en cuatro componentes: el *hardware*, el *sistema operativo*, los *programas de aplicación* y los *usuarios* (Figura 1.1).

El **hardware**, la **unidad central de procesamiento**, UCP o CPU (central processing unit), la **memoria** y los **dispositivos de E/S** (entrada/salida), proporciona los recursos básicos de cómputo al sistema. Los **programas de aplicación**, como son los procesadores de texto, las hojas de cálculo, los compiladores y los exploradores web, definen las formas en que estos recursos se emplean para resolver los problemas informáticos de los usuarios. El sistema operativo controla y coordina el uso del hardware entre los diversos programas de aplicación por parte de los distintos usuarios.

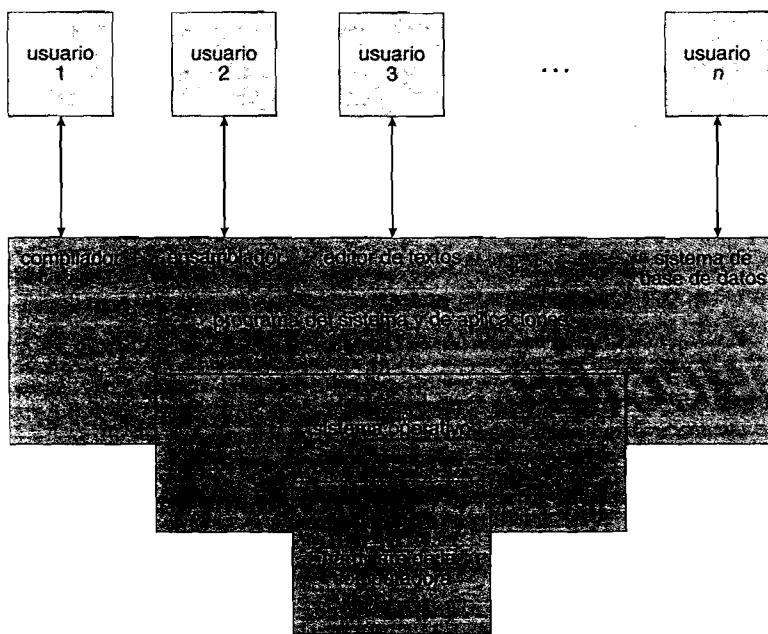


Figura 1.1 Esquema de los componentes de un sistema informático.

También podemos ver un sistema informático como hardware, software y datos. El sistema operativo proporciona los medios para hacer un uso adecuado de estos recursos durante el funcionamiento del sistema informático. Un sistema operativo es similar a un gobierno. Como un gobierno, no realiza ninguna función útil por sí mismo: simplemente proporciona un entorno en el que otros programas pueden llevar a cabo un trabajo útil.

Para comprender mejor el papel de un sistema operativo, a continuación vamos a abordar los sistemas operativos desde dos puntos de vista: el del usuario y el del sistema.

1.1.1 Punto de vista del usuario

La visión del usuario de la computadora varía de acuerdo con la interfaz que utilice. La mayoría de los usuarios que se sientan frente a un PC disponen de un monitor, un teclado, un ratón y una unidad de sistema. Un sistema así se diseña para que un usuario monopolice sus recursos. El objetivo es maximizar el trabajo (o el juego) que el usuario realice. En este caso, el sistema operativo se diseña principalmente para que sea de **fácil uso**, prestando cierta atención al rendimiento y ninguna a la **utilización de recursos** (el modo en que se comparten los recursos hardware y software). Por supuesto, el rendimiento es importante para el usuario, pero más que la utilización de recursos, estos sistemas se optimizan para el uso del mismo por un solo usuario.

En otros casos, un usuario se sienta frente a un terminal conectado a un **mainframe** o una **microcomputadora**. Otros usuarios acceden simultáneamente a través de otros terminales. Estos usuarios comparten recursos y pueden intercambiar información. En tales casos, el sistema operativo se diseña para maximizar la utilización de recursos, asegurando que todo el tiempo de CPU, memoria y E/S disponibles se usen de forma eficiente y que todo usuario disponga sólo de la parte equitativa que le corresponde.

En otros casos, los usuarios usan **estaciones de trabajo** conectadas a redes de otras estaciones de trabajo y **servidores**. Estos usuarios tienen recursos dedicados a su disposición, pero también tienen recursos compartidos como la red y los servidores (servidores de archivos, de cálculo y de impresión). Por tanto, su sistema operativo está diseñado para llegar a un compromiso entre la usabilidad individual y la utilización de recursos.

Recientemente, se han puesto de moda una gran variedad de computadoras de mano. La mayor parte de estos dispositivos son unidades autónomas para usuarios individuales. Algunas se conectan a redes directamente por cable o, más a menudo, a través de redes y modems inalám-

bricos. Debido a las limitaciones de alimentación, velocidad e interfaz, llevan a cabo relativamente pocas operaciones remotas. Sus sistemas operativos están diseñados principalmente en función de la usabilidad individual, aunque el rendimiento, medido según la duración de la batería, es también importante.

Algunas computadoras tienen poca o ninguna interacción con el usuario. Por ejemplo, las computadoras incorporadas en los electrodomésticos y en los automóviles pueden disponer de teclados numéricos e indicadores luminosos que se encienden y apagan para mostrar el estado, pero tanto estos equipos como sus sistemas operativos están diseñados fundamentalmente para funcionar sin intervención del usuario.

1.1.2 Vista del sistema

Desde el punto de vista de la computadora, el sistema operativo es el programa más íntimamente relacionado con el hardware. En este contexto, podemos ver un sistema operativo como un **asignador de recursos**. Un sistema informático tiene muchos recursos que pueden ser necesarios para solucionar un problema: tiempo de CPU, espacio de memoria, espacio de almacenamiento de archivos, dispositivos de E/S, etc. El sistema operativo actúa como el administrador de estos recursos. Al enfrentarse a numerosas y posiblemente conflictivas solicitudes de recursos, el sistema operativo debe decidir cómo asignarlos a programas y usuarios específicos, de modo que la computadora pueda operar de forma eficiente y equitativa. Como hemos visto, la asignación de recursos es especialmente importante cuando muchos usuarios acceden al mismo *mainframe* o minicomputadora.

Un punto de vista ligeramente diferente de un sistema operativo hace hincapié en la necesidad de controlar los distintos dispositivos de E/S y programas de usuario. Un sistema operativo es un programa de control. Como **programa de control**, gestiona la ejecución de los programas de usuario para evitar errores y mejorar el uso de la computadora. Tiene que ver especialmente con el funcionamiento y control de los dispositivos de E/S.

1.1.3 Definición de sistemas operativos

Hemos visto el papel de los sistemas operativos desde los puntos de vista del usuario y del sistema. Pero, ¿cómo podemos definir qué es un sistema operativo? En general, no disponemos de ninguna definición de sistema operativo que sea completamente adecuada. Los sistemas operativos existen porque ofrecen una forma razonable de resolver el problema de crear un sistema informático utilizable. El objetivo fundamental de las computadoras es ejecutar programas de usuario y resolver los problemas del mismo fácilmente. Con este objetivo se construye el hardware de la computadora. Debido a que el hardware por sí solo no es fácil de utilizar, se desarrollaron programas de aplicación. Estos programas requieren ciertas operaciones comunes, tales como las que controlan los dispositivos de E/S. Las operaciones habituales de control y asignación de recursos se incorporan en una misma pieza del software: el sistema operativo.

Además, no hay ninguna definición universalmente aceptada sobre qué forma parte de un sistema operativo. Desde un punto de vista simple, incluye todo lo que un distribuidor suministra cuando se pide “el sistema operativo”. Sin embargo, las características incluidas varían enormemente de un sistema a otro. Algunos sistemas ocupan menos de 1 megabyte de espacio y no proporcionan ni un editor a pantalla completa, mientras que otros necesitan gigabytes de espacio y están completamente basados en sistemas gráficos de ventanas. (Un kilobyte, Kb, es 1024 bytes; un megabyte, MB, es 1024^2 bytes y un gigabyte, GB, es 1024^3 bytes. Los fabricantes de computadoras a menudo redondean estas cifras y dicen que 1 megabyte es un millón de bytes y que un gigabyte es mil millones de bytes.) Una definición más común es que un sistema operativo es aquel programa que se ejecuta continuamente en la computadora (usualmente denominado *kernel*), siendo todo lo demás programas del sistema y programas de aplicación. Esta definición es la que generalmente seguiremos.

La cuestión acerca de qué constituye un sistema operativo está adquiriendo una importancia creciente. En 1998, el Departamento de Justicia de Estados Unidos entabló un pleito contra

Microsoft, aduciendo en esencia que Microsoft incluía demasiada funcionalidad en su sistema operativo, impidiendo a los vendedores de aplicaciones competir. Por ejemplo, un explorador web era una parte esencial del sistema operativo. Como resultado, Microsoft fue declarado culpable de usar su monopolio en los sistemas operativos para limitar la competencia.

1.2 Organización de una computadora

Antes de poder entender cómo funciona una computadora, necesitamos unos conocimientos generales sobre su estructura. En esta sección, veremos varias partes de esa estructura para completar nuestros conocimientos. La sección se ocupa principalmente de la organización de una computadora, así que puede hojearla o saltársela si ya está familiarizado con estos conceptos.

1.2.1 Funcionamiento de una computadora

Una computadora moderna de propósito general consta de una o más CPU y de una serie de controladoras de dispositivo conectadas a través de un bus común que proporciona acceso a la memoria compartida (Figura 1.2). Cada controladora de dispositivo se encarga de un tipo específico de dispositivo, por ejemplo, unidades de disco, dispositivos de audio y pantallas de vídeo. La CPU y las controladoras de dispositivos pueden funcionar de forma concurrente, compitiendo por los ciclos de memoria. Para asegurar el acceso de forma ordenada a la memoria compartida, se proporciona una controladora de memoria cuya función es sincronizar el acceso a la misma.

Para que una computadora comience a funcionar, por ejemplo cuando se enciende o se reinicia, es necesario que tenga un programa de inicio que ejecutar. Este programa de inicio, o **programa de arranque**, suele ser simple. Normalmente, se almacena en la memoria ROM (read only memory, memoria de sólo lectura) o en una memoria EEPROM (electrically erasable programmable read-only memory, memoria de sólo lectura programable y eléctricamente borrable), y se conoce con el término general **firmware**, dentro del hardware de la computadora. Se inicializan todos los aspectos del sistema, desde los registros de la CPU hasta las controladoras de dispositivos y el contenido de la memoria. El programa de arranque debe saber cómo cargar el sistema operativo e iniciar la ejecución de dicho sistema. Para conseguir este objetivo, el programa de arranque debe localizar y cargar en memoria el *kernel* (núcleo) del sistema operativo. Después, el sistema operativo comienza ejecutando el primer proceso, como por ejemplo “init”, y espera a que se produzca algún suceso.

La ocurrencia de un suceso normalmente se indica mediante una **interrupción** bien hardware o bien software. El hardware puede activar una interrupción en cualquier instante enviando una señal a la CPU, normalmente a través del bus del sistema. El software puede activar una interrupción ejecutando una operación especial denominada **llamada del sistema** (o también llamada de **monitor**).

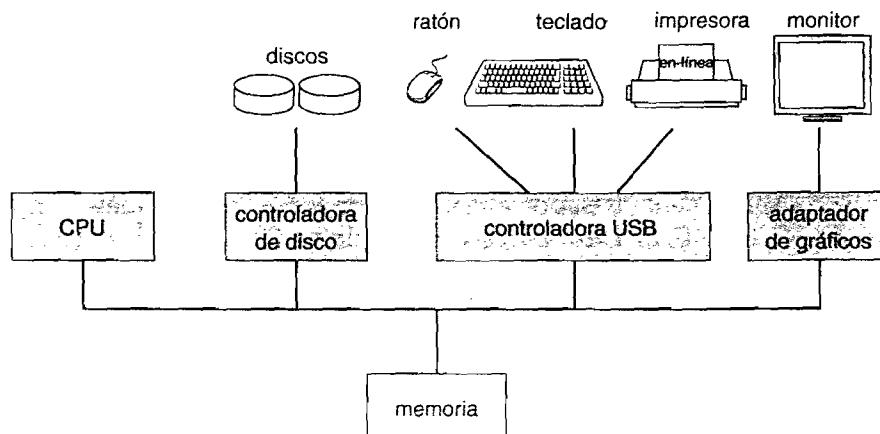


Figura 1.2 Una computadora moderna.

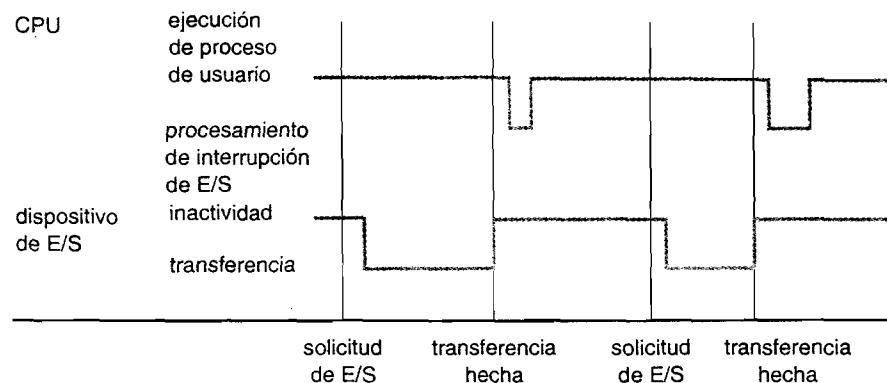


Figura 1.3 Diagrama de tiempos de una interrupción para un proceso de salida.

Cuando se interrumpe a la CPU, deja lo que está haciendo e inmediatamente transfiere la ejecución a una posición fija (establecida). Normalmente, dicha posición contiene la dirección de inicio de donde se encuentra la rutina de servicio a la interrupción. La rutina de servicio a la interrupción se ejecuta y, cuando ha terminado, la cpu reanuda la operación que estuviera haciendo. En la Figura 1.3 se muestra un diagrama de tiempos de esta operación.

Las interrupciones son una parte importante de la arquitectura de una computadora. Cada diseño de computadora tiene su propio mecanismo de interrupciones, aunque hay algunas funciones comunes. La interrupción debe transferir el control a la rutina de servicio apropiada a la interrupción.

El método más simple para tratar esta transferencia consiste en invocar una rutina genérica para examinar la información de la interrupción; esa rutina genérica, a su vez, debe llamar a la rutina específica de tratamiento de la interrupción. Sin embargo, las interrupciones deben tratarse rápidamente y este método es algo lento. En consecuencia, como sólo es posible un número predefinido de interrupciones, puede utilizarse otro sistema, consistente en disponer una tabla de punteros a las rutinas de interrupción, con el fin de proporcionar la velocidad necesaria. De este modo, se llama a la rutina de interrupción de forma indirecta a través de la tabla, sin necesidad de una rutina intermedia. Generalmente, la tabla de punteros se almacena en la zona inferior de la memoria (las primeras 100 posiciones). Estas posiciones almacenan las direcciones de las rutinas de servicio a la interrupción para los distintos dispositivos. Esta matriz, o **vector de interrupciones**, de direcciones se indexa mediante un número de dispositivo único que se proporciona con la solicitud de interrupción, para obtener la dirección de la rutina de servicio a la interrupción para el dispositivo correspondiente. Sistemas operativos tan diferentes como Windows y UNIX manejan las interrupciones de este modo.

La arquitectura de servicio de las interrupciones también debe almacenar la dirección de la instrucción interrumpida. Muchos diseños antiguos simplemente almacenaban la dirección de interrupción en una posición fija o en una posición indexada mediante el número de dispositivo. Las arquitecturas más recientes almacenan la dirección de retorno en la pila del sistema. Si la rutina de interrupción necesita modificar el estado del procesador, por ejemplo modificando valores del registro, debe guardar explícitamente el estado actual y luego restaurar dicho estado antes de volver. Después de atender a la interrupción, la dirección de retorno guardada se carga en el contador de programa y el cálculo interrumpido se reanuda como si la interrupción no se hubiera producido.

1.2.2 Estructura de almacenamiento

Los programas de la computadora deben hallarse en la memoria principal (también llamada memoria **RAM**, random-access memory, memoria de acceso aleatorio) para ser ejecutados. La memoria principal es el único área de almacenamiento de gran tamaño (millones o miles de millones de bytes) a la que el procesador puede acceder directamente. Habitualmente, se implementa con una tecnología de semiconductores denominada **DRAM** (dynamic random-access memory).

memoria dinámica de acceso aleatorio), que forma una matriz de palabras de memoria. Cada palabra tiene su propia dirección. La interacción se consigue a través de una secuencia de carga (*load*) o almacenamiento (*store*) de instrucciones en direcciones específicas de memoria. La instrucción *load* mueve una palabra desde la memoria principal a un registro interno de la CPU, mientras que la instrucción *store* mueve el contenido de un registro a la memoria principal.

Aparte de las cargas y almacenamientos explícitos, la CPU carga automáticamente instrucciones desde la memoria principal para su ejecución.

Un ciclo típico instrucción-ejecución, cuando se ejecuta en un sistema con una arquitectura de von Neumann, primero extrae una instrucción de memoria y almacena dicha instrucción en el **registro de instrucciones**. A continuación, la instrucción se decodifica y puede dar lugar a que se extraigan operandos de la memoria y se almacenen en algún registro interno. Después de ejecutar la instrucción con los necesarios operandos, el resultado se almacena de nuevo en memoria. Observe que la unidad de memoria sólo ve un flujo de direcciones de memoria; no sabe cómo se han generado (mediante el contador de instrucciones, indexación, indirección, direcciones literales o algún otro medio) o qué son (instrucciones o datos). De acuerdo con esto, podemos ignorar *cómo* genera un programa una dirección de memoria. Sólo nos interesaremos por la secuencia de direcciones de memoria generada por el programa en ejecución.

Idealmente, es deseable que los programas y los datos residan en la memoria principal de forma permanente. Usualmente, esta situación no es posible por las dos razones siguientes:

1. Normalmente, la memoria principal es demasiado pequeña como para almacenar todos los programas y datos necesarios de forma permanente.
2. La memoria principal es un dispositivo de almacenamiento *volátil* que pierde su contenido cuando se quita la alimentación.

Por tanto, la mayor parte de los sistemas informáticos proporcionan **almacenamiento secundario** como una extensión de la memoria principal. El requerimiento fundamental de este almacenamiento secundario es que se tienen que poder almacenar grandes cantidades de datos de forma permanente.

El dispositivo de almacenamiento secundario más común es el **disco magnético**, que proporciona un sistema de almacenamiento tanto para programas como para datos. La mayoría de los programas (exploradores web, compiladores, procesadores de texto, hojas de cálculo, etc.) se almacenan en un disco hasta que se cargan en memoria. Muchos programas utilizan el disco como origen y destino de la información que están procesando. Por tanto, la apropiada administración del almacenamiento en disco es de importancia crucial en un sistema informático, como veremos en el Capítulo 12.

Sin embargo, en un sentido amplio, la estructura de almacenamiento que hemos descrito, que consta de registros, memoria principal y discos magnéticos, sólo es uno de los muchos posibles sistemas de almacenamiento. Otros sistemas incluyen la memoria caché, los CD-ROM, cintas magnéticas, etc. Cada sistema de almacenamiento proporciona las funciones básicas para guardar datos y mantener dichos datos hasta que sean recuperados en un instante posterior. Las principales diferencias entre los distintos sistemas de almacenamiento están relacionadas con la velocidad, el coste, el tamaño y la volatilidad.

La amplia variedad de sistemas de almacenamiento en un sistema informático puede organizarse en una jerarquía (Figura 1.4) según la velocidad y el coste. Los niveles superiores son caros, pero rápidos. A medida que se desciende por la jerarquía, el coste por bit generalmente disminuye, mientras que el tiempo de acceso habitualmente aumenta. Este compromiso es razonable; si un sistema de almacenamiento determinado fuera a la vez más rápido y barato que otro (siendo el resto de las propiedades las mismas), entonces no habría razón para emplear la memoria más cara y más lenta. De hecho, muchos de los primeros dispositivos de almacenamiento, incluyendo las cintas de papel y las memorias de núcleo, han quedado relegadas a los museos ahora que las cintas magnéticas y las **memorias semiconductoras** son más rápidas y baratas. Los cuatro niveles de memoria superiores de la Figura 1.4 se pueden construir con memorias semiconductoras.

Además de diferenciarse en la velocidad y en el coste, los distintos sistemas de almacenamiento pueden ser volátiles o no volátiles. Como hemos mencionado anteriormente, el **almacena-**

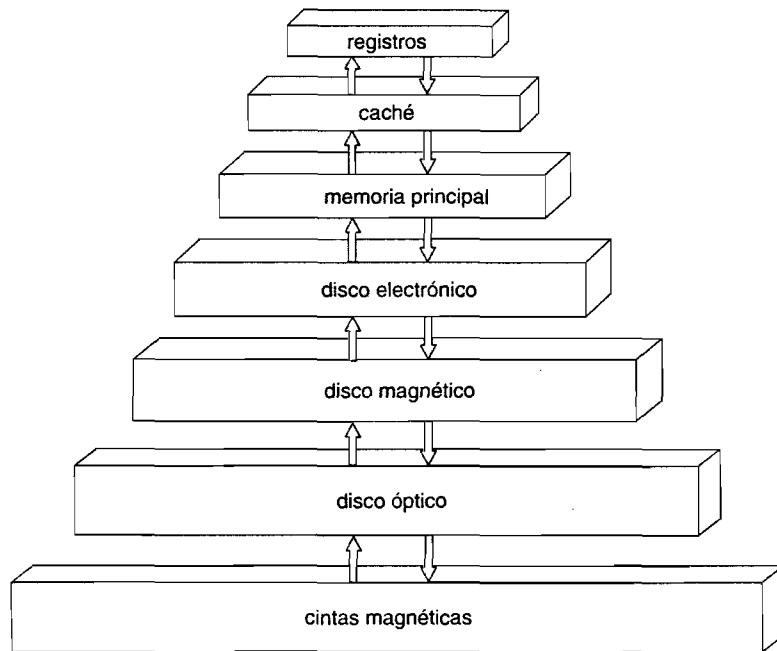


Figura 1.4 Jerarquía de dispositivos de almacenamiento.

miento volátil pierde su contenido cuando se retira la alimentación del dispositivo. En ausencia de baterías caras y sistemas de alimentación de reserva, los datos deben escribirse en **almacenamiento no volátil** para su salvaguarda. En la jerarquía mostrada en la Figura 1.4, los sistemas de almacenamiento que se encuentran por encima de los discos electrónicos son volátiles y los que se encuentran por debajo son no volátiles. Durante la operación normal, el disco electrónico almacena datos en una matriz DRAM grande, que es volátil. Pero muchos dispositivos de disco electrónico contienen un disco duro magnético oculto y una batería de reserva. Si la alimentación externa se interrumpe, la controladora del disco electrónico copia los datos de la RAM en el disco magnético. Cuando se restaura la alimentación, los datos se cargan de nuevo en la RAM. Otra forma de disco electrónico es la memoria flash, la cual es muy popular en las cámaras, los PDA (*personal digital assistant*) y en los robots; asimismo, está aumentando su uso como dispositivo de almacenamiento extraible en computadoras de propósito general. La memoria flash es más lenta que la DRAM, pero no necesita estar alimentada para mantener su contenido. Otra forma de almacenamiento no volátil es la NVRAM, que es una DRAM con batería de reserva. Esta memoria puede ser tan rápida como una DRAM, aunque sólo mantiene su carácter no volátil durante un tiempo limitado.

El diseño de un sistema de memoria completo debe equilibrar todos los factores mencionados hasta aquí: sólo debe utilizarse una memoria cara cuando sea necesario y emplear memorias más baratas y no volátiles cuando sea posible. Se pueden instalar memorias caché para mejorar el rendimiento cuando entre dos componentes existe un tiempo de acceso largo o disparidad en la velocidad de transferencia.

1.2.3 Estructura de E/S

Los de almacenamiento son sólo uno de los muchos tipos de dispositivos de E/S que hay en un sistema informático. Gran parte del código del sistema operativo se dedica a gestionar la entrada y la salida, debido a su importancia para la fiabilidad y rendimiento del sistema y debido también a la variada naturaleza de los dispositivos. Vamos a realizar, por tanto, un repaso introductorio del tema de la E/S.

Una computadora de propósito general consta de una o más CPU y de múltiples controladoras de dispositivo que se conectan a través de un bus común. Cada controladora de dispositivo se encarga de un tipo específico de dispositivo. Dependiendo de la controladora, puede haber más

de un dispositivo conectado. Por ejemplo, siete o más dispositivos pueden estar conectados a la controladora SCSI (**small computer-system interface**, interfaz para sistemas informáticos de pequeño tamaño). Una controladora de dispositivo mantiene algunos búferes locales y un conjunto de registros de propósito especial. La controladora del dispositivo es responsable de transferir los datos entre los dispositivos periféricos que controla y su búfer local. Normalmente, los sistemas operativos tienen un **controlador (driver) de dispositivo** para cada controladora (**controller**) de dispositivo. Este software controlador del dispositivo es capaz de entenderse con la controladora hardware y presenta al resto del sistema operativo una interfaz uniforme mediante la cual comunicarse con el dispositivo.

Al iniciar una operación de E/S, el controlador del dispositivo carga los registros apropiados de la controladora hardware. Ésta, a su vez, examina el contenido de estos registros para determinar qué acción realizar (como, por ejemplo, "leer un carácter del teclado"). La controladora inicia entonces la transferencia de datos desde el dispositivo a su búfer local. Una vez completada la transferencia de datos, la controladora hardware informa al controlador de dispositivo, a través de una interrupción, de que ha terminado la operación. El controlador devuelve entonces el control al sistema operativo, devolviendo posiblemente los datos, o un puntero a los datos, si la operación ha sido una lectura. Para otras operaciones, el controlador del dispositivo devuelve información de estado.

Esta forma de E/S controlada por interrupción resulta adecuada para transferir cantidades pequeñas de datos, pero representa un desperdicio de capacidad de proceso cuando se usa para movimientos masivos de datos, como en la E/S de disco. Para resolver este problema, se usa el acceso directo a memoria (**DMA, direct memory access**). Después de configurar búferes, punteros y contadores para el dispositivo de E/S, la controladora hardware transfiere un bloque entero de datos entre su propio búfer y la memoria, sin que intervenga la CPU. Sólo se genera una interrupción por cada bloque, para decir al controlador software del dispositivo que la operación se ha completado, en lugar de la interrupción por byte generada en los dispositivos de baja velocidad. Mientras la controladora hardware realiza estas operaciones, la CPU está disponible para llevar a cabo otros trabajos.

Algunos sistemas de gama alta emplean una arquitectura basada en conmutador, en lugar de en bus. En estos sistemas, los diversos componentes pueden comunicarse con otros componentes de forma concurrente, en lugar de competir por los ciclos de un bus compartido. En este caso,

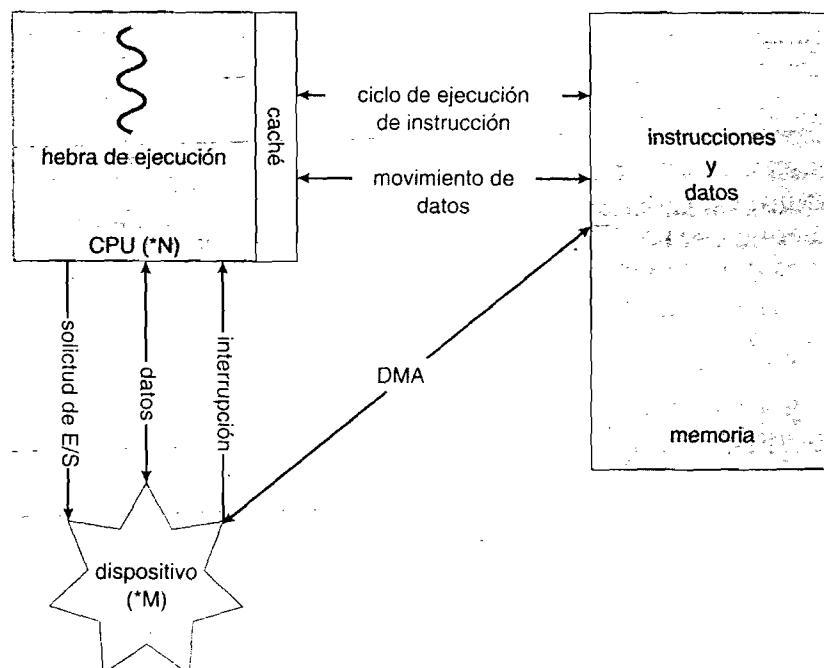


Figura 1.5 Funcionamiento de un sistema informático moderno.

el acceso directo a memoria es incluso más eficaz. La Figura 1.5 muestra la interacción de los distintos componentes de un sistema informático.

1.3 Arquitectura de un sistema informático

En la Sección 1.2 hemos presentado la estructura general de un sistema informático típico. Un sistema informático se puede organizar de varias maneras diferentes, las cuales podemos clasificar de acuerdo con el número de procesadores de propósito general utilizados.

1.3.1 Sistemas de un solo procesador

La mayor parte de los sistemas sólo usan un procesador. No obstante, la variedad de sistemas de un único procesador puede ser realmente sorprendente, dado que van desde los PDA hasta los sistemas *mainframe*. En un sistema de un único procesador, hay una CPU principal capaz de ejecutar un conjunto de instrucciones de propósito general, incluyendo instrucciones de los procesos de usuario. Casi todos los sistemas disponen también de otros procesadores de propósito especial. Pueden venir en forma de procesadores específicos de un dispositivo, como por ejemplo un disco, un teclado o una controladora gráfica; o, en *mainframes*, pueden tener la forma de procesadores de propósito general, como procesadores de E/S que transfieren rápidamente datos entre los componentes del sistema.

Todos estos procesadores de propósito especial ejecutan un conjunto limitado de instrucciones y no ejecutan procesos de usuario. En ocasiones, el sistema operativo los gestiona, en el sentido de que les envía información sobre su siguiente tarea y monitoriza su estado. Por ejemplo, un microprocesador incluido en una controladora de disco recibe una secuencia de solicitudes procedentes de la CPU principal e implementa su propia cola de disco y su algoritmo de programación de tareas. Este método libera a la CPU principal del trabajo adicional de planificar las tareas de disco. Los PC contienen un microprocesador en el teclado para convertir las pulsaciones de tecla en códigos que se envían a la CPU. En otros sistemas o circunstancias, los procesadores de propósito especial son componentes de bajo nivel integrados en el hardware. El sistema operativo no puede comunicarse con estos procesadores, sino que éstos hacen su trabajo de forma autónoma. La presencia de microprocesadores de propósito especial resulta bastante común y no convierte a un sistema de un solo procesador en un sistema multiprocesador. Si sólo hay una CPU de propósito general, entonces el sistema es de un solo procesador.

1.3.2 Sistemas multiprocesador

Aunque los sistemas de un solo procesador son los más comunes, la importancia de los **sistemas multiprocesador** (también conocidos como **sistemas paralelos** o **sistemas fuertemente acoplados**) está siendo cada vez mayor. Tales sistemas disponen de dos o más procesadores que se comunican entre sí, compartiendo el bus de la computadora y, en ocasiones, el reloj, la memoria y los dispositivos periféricos.

Los sistemas multiprocesador presentan tres ventajas fundamentales:

1. **Mayor rendimiento.** Al aumentar el número de procesadores, es de esperar que se realice más trabajo en menos tiempo. Sin embargo, la mejora en velocidad con N procesadores no es N , sino que es menor que N . Cuando múltiples procesadores cooperan en una tarea, cierta carga de trabajo se emplea en conseguir que todas las partes funcionen correctamente. Esta carga de trabajo, más la contienda por los recursos compartidos, reducen la ganancia esperada por añadir procesadores adicionales. De forma similar, N programadores trabajando simultáneamente no producen N veces la cantidad de trabajo que produciría un solo programador .
2. **Economía de escala.** Los sistemas multiprocesador pueden resultar más baratos que su equivalente con múltiples sistemas de un solo procesador, ya que pueden compartir perifé-

ricos, almacenamiento masivo y fuentes de alimentación. Si varios programas operan sobre el mismo conjunto de datos, es más barato almacenar dichos datos en un disco y que todos los procesadores los compartan, que tener muchas computadoras con discos locales y muchas copias de los datos.

3. Mayor fiabilidad. Si las funciones se pueden distribuir de forma apropiada entre varios procesadores, entonces el fallo de un procesador no hará que el sistema deje de funcionar, sino que sólo se ralentizará. Si tenemos diez procesadores y uno falla, entonces cada uno de los nueve restantes procesadores puede asumir una parte del trabajo del procesador que ha fallado. Por tanto, el sistema completo trabajará un 10% más despacio, en lugar de dejar de funcionar.

En muchas aplicaciones, resulta crucial conseguir la máxima fiabilidad del sistema informático. La capacidad de continuar proporcionando servicio proporcionalmente al nivel de hardware superviviente se denomina **degradación suave**. Algunos sistemas van más allá de la degradación suave y se denominan sistemas **tolerantes a fallos**, dado que pueden sufrir un fallo en cualquier componente y continuar operando. Observe que la tolerancia a fallos requiere un mecanismo que permita detectar, diagnosticar y, posiblemente, corregir el fallo. El sistema HP NonStop (antes sistema Tandem) duplica el hardware y el software para asegurar un funcionamiento continuado a pesar de los fallos. El sistema consta de múltiples parejas de CPU que trabajan sincronizadamente. Ambos procesadores de la pareja ejecutan cada instrucción y comparan los resultados. Si los resultados son diferentes, quiere decir que una de las CPU de la pareja falla y ambas dejan de funcionar. El proceso que estaba en ejecución se transfiere a otra pareja de CPU y la instrucción fallida se reinicia. Esta solución es cara, dado que implica hardware especial y una considerable duplicación del hardware.

Los sistemas multiprocesador actualmente utilizados son de dos tipos. Algunos sistemas usan el **multiprocesamiento asimétrico**, en el que cada procesador se asigna a una tarea específica. Un procesador maestro controla el sistema y el resto de los procesadores esperan que el maestro les dé instrucciones o tienen asignadas tareas predefinidas. Este esquema define una relación maestro-esclavo. El procesador maestro planifica el trabajo de los procesadores esclavos y se lo asigna.

Los sistemas más comunes utilizan el **multiprocesamiento simétrico (SMP)**, en el que cada procesador realiza todas las tareas correspondientes al sistema operativo. En un sistema SMP, todos los procesadores son iguales; no existe una relación maestro-esclavo entre los procesadores. La Figura 1.6 ilustra una arquitectura SMP típica. Un ejemplo de sistema SMP es Solaris, una versión comercial de UNIX diseñada por Sun Microsystems. Un sistema Sun se puede configurar empleando docenas de procesadores que ejecuten Solaris. La ventaja de estos modelos es que se pueden ejecutar simultáneamente muchos procesos (se pueden ejecutar N procesos si se tienen N CPU) sin que se produzca un deterioro significativo del rendimiento. Sin embargo, hay que controlar cuidadosamente la E/S para asegurar que los datos lleguen al procesador adecuado. También, dado que las CPU están separadas, una puede estar en un período de inactividad y otra puede estar sobrecargada, dando lugar a ineficiencias. Estas situaciones se pueden evitar si los procesadores comparten ciertas estructuras de datos. Un sistema multiprocesador de este tipo permitirá que los procesos y los recursos (como la memoria) sean compartidos dinámicamente entre los distintos procesadores, lo que permite disminuir la varianza entre la carga de trabajo de los procesadores. Un sistema así se debe diseñar con sumo cuidado, como veremos en el Capítulo 6. Prácticamente todos los sistemas operativos modernos, incluyendo Windows, Windows XP, Mac OS X y Linux, proporcionan soporte para SMP.

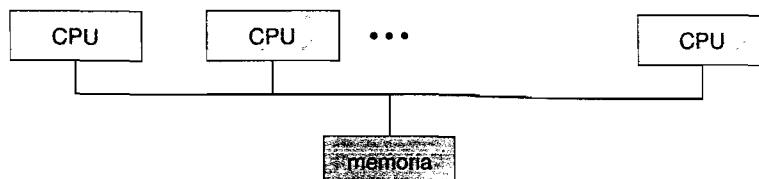


Figura 1.6 Arquitectura de multiprocesamiento simétrico.

La diferencia entre multiprocesamiento simétrico y asimétrico puede deberse tanto al hardware como al software. Puede que haya un hardware especial que diferencie los múltiples procesadores o se puede escribir el software para que haya sólo un maestro y múltiples esclavos. Por ejemplo, el sistema operativo SunOS Versión 4 de Sun proporciona multiprocesamiento asimétrico, mientras que Solaris (Versión 5) es simétrico utilizando el mismo hardware.

Una tendencia actual en el diseño de las CPU es incluir múltiples **núcleos** de cálculo en un mismo chip. En esencia, se trata de chips multiprocesador. Los chips de dos vías se están convirtiendo en la corriente dominante, mientras que los chips de N vías están empezando a ser habituales en los sistemas de gama alta. Dejando aparte las consideraciones sobre la arquitectura, como la caché, la memoria y la contienda de bus, estas CPU con múltiples núcleos son vistas por el sistema operativo simplemente como N procesadores estándar.

Por último, los **servidores blade** son un reciente desarrollo, en el que se colocan múltiples procesadores, tarjetas de E/S y tarjetas de red en un mismo chasis. La diferencia entre estos sistemas y los sistemas multiprocesador tradicionales es que cada tarjeta de procesador blade arranca independientemente y ejecuta su propio sistema operativo. Algunas tarjetas de servidor blade también son multiprocesador, lo que difumina la línea divisoria entre los distintos tipos de computadoras. En esencia, dichos servidores constan de múltiples sistemas multiprocesador independientes.

1.3.3 Sistemas en cluster

Otro tipo de sistema con múltiples CPU es el **sistema en cluster**. Como los sistemas multiprocesador, los sistemas en cluster utilizan múltiples CPU para llevar a cabo el trabajo. Los sistemas en cluster se diferencian de los sistemas de multiprocesamiento en que están formados por dos o más sistemas individuales acoplados. La definición del término *en cluster* no es concreta; muchos paquetes comerciales argumentan acerca de qué es un sistema en *cluster* y por qué una forma es mejor que otra. La definición generalmente aceptada es que las computadoras en cluster comparten el almacenamiento y se conectan entre sí a través de una red de área local (**LAN, local area network**), como se describe en la Sección 1.10, o mediante una conexión más rápida como InfiniBand.

Normalmente, la conexión en cluster se usa para proporcionar un servicio con **alta disponibilidad**; es decir, un servicio que funcionará incluso si uno o más sistemas del cluster fallan. Generalmente, la alta disponibilidad se obtiene añadiendo un nivel de redundancia al sistema. Sobre los nodos del *cluster* se ejecuta una capa de software de gestión del *cluster*. Cada nodo puede monitorizar a uno o más de los restantes (en una LAN). Si la máquina monitorizada falla, la máquina que la estaba monitorizando puede tomar la propiedad de su almacenamiento y reiniciar las aplicaciones que se estuvieran ejecutando en la máquina que ha fallado. Los usuarios y clientes de las aplicaciones sólo ven una breve interrupción del servicio.

El cluster se puede estructurar simétrica o asimétricamente. En un **cluster asimétrico**, una máquina está en **modo de espera en caliente**, mientras que la otra está ejecutando las aplicaciones. La máquina *host* en modo de espera en caliente no hace nada más que monitorizar al servidor activo. Si dicho servidor falla, el *host* que está en espera pasa a ser el servidor activo. En el **modo simétrico**, dos o más *hosts* ejecutan aplicaciones y se monitorizan entre sí. Este modo es obviamente más eficiente, ya que usa todo el hardware disponible. Aunque, desde luego, requiere que haya disponible más de una aplicación para ejecutar.

Otras formas de cluster incluyen el *cluster* en paralelo y los *clusters* conectados a una red de área extensa (**WAN, wide area network**), como se describe en la Sección 1.10. Los *clusters* en paralelo permiten que múltiples *hosts* accedan a unos mismos datos, disponibles en el almacenamiento compartido. Dado que la mayoría de los sistemas operativos no soportan el acceso simultáneo a datos por parte de múltiples *hosts*, normalmente los clusters en paralelo requieren emplear versiones especiales de software y versiones especiales de las aplicaciones. Por ejemplo, Oracle Parallel Server es una versión de la base de datos de Oracle que se ha diseñado para ejecutarse en un cluster paralelo. Cada una de las máquinas ejecuta Oracle y hay una capa de software que controla el acceso al disco compartido. Cada máquina tiene acceso total a todos los datos de la base

Cuando dicho trabajo tiene que esperar, la CPU comuta a *otro* trabajo, y así sucesivamente. Cuando el primer trabajo deja de esperar, vuelve a obtener la CPU. Mientras haya al menos un trabajo que necesite ejecutarse, la CPU nunca estará inactiva.

Esta idea también se aplica en otras situaciones de la vida. Por ejemplo, un abogado no trabaja para un único cliente cada vez. Mientras que un caso está a la espera de que salga el juicio o de que se rellenen determinados papeles, el abogado puede trabajar en otro caso. Si tiene bastantes clientes, el abogado nunca estará inactivo debido a falta de trabajo. (Los abogados inactivos tienden a convertirse en políticos, por lo que los abogados que se mantienen ocupados tienen cierto valor social.)

Los sistemas multiprogramados proporcionan un entorno en el que se usan de forma eficaz los diversos recursos del sistema, como por ejemplo la CPU, la memoria y los periféricos, aunque no proporcionan la interacción del usuario con el sistema informático. El **tiempo compartido** (o **multitarea**) es una extensión lógica de la multiprogramación. En los sistemas de tiempo compartido, la CPU ejecuta múltiples trabajos conmutando entre ellos, pero las conmutaciones se producen tan frecuentemente que los usuarios pueden interactuar con cada programa mientras éste está en ejecución.

El tiempo compartido requiere un **sistema informático interactivo**, que proporcione comunicación directa entre el usuario y el sistema. El usuario suministra directamente instrucciones al sistema operativo o a un programa, utilizando un dispositivo de entrada como un teclado o un ratón, y espera los resultados intermedios en un dispositivo de salida. De acuerdo con esto, el **tiempo de respuesta** debe ser pequeño, normalmente menor que un segundo.

Un sistema operativo de tiempo compartido permite que muchos usuarios comparten simultáneamente la computadora. Dado que el tiempo de ejecución de cada acción o comando en un sistema de tiempo compartido tiende a ser pequeño, sólo es necesario un tiempo pequeño de CPU para cada usuario. Puesto que el sistema cambia rápidamente de un usuario al siguiente, cada usuario tiene la impresión de que el sistema informático completo está dedicado a él, incluso aunque esté siendo compartido por muchos usuarios.

Un sistema de tiempo compartido emplea mecanismos de multiprogramación y de planificación de la CPU para proporcionar a cada usuario una pequeña parte de una computadora de tiempo compartido. Cada usuario tiene al menos un programa distinto en memoria. Un programa cargado en memoria y en ejecución se denomina **proceso**. Cuando se ejecuta un proceso, normalmente se ejecuta sólo durante un período de tiempo pequeño, antes de terminar o de que necesite realizar una operación de E/S. La E/S puede ser interactiva, es decir, la salida va a la pantalla y la entrada procede del teclado, el ratón u otro dispositivo del usuario. Dado que normalmente la E/S interactiva se ejecuta a la “velocidad de las personas”, puede necesitar cierto tiempo para completarse. Por ejemplo, la entrada puede estar limitada por la velocidad de tecleo del usuario; escribir siete caracteres por segundo ya es rápido para una persona, pero increíblemente lento para una computadora. En lugar de dejar que la CPU espere sin hacer nada mientras se produce esta entrada interactiva, el sistema operativo hará que la CPU conmute rápidamente al programa de algún otro usuario.

El tiempo compartido y la multiprogramación requieren mantener simultáneamente en memoria varios trabajos. Dado que en general la memoria principal es demasiado pequeña para acomodar todos los trabajos, éstos se mantienen inicialmente en el disco, en la denominada **cola de trabajos**. Esta cola contiene todos los procesos que residen en disco esperando la asignación de la memoria principal. Si hay varios trabajos preparados para pasar a memoria y no hay espacio suficiente para todos ellos, entonces el sistema debe hacer una selección de los mismos. La toma de esta decisión es lo que se denomina **planificación de trabajos**, tema que se explica en el Capítulo 5. Cuando el sistema operativo selecciona un trabajo de la cola de trabajos, carga dicho trabajo en memoria para su ejecución. Tener varios programas en memoria al mismo tiempo requiere algún tipo de mecanismo de gestión de la memoria, lo que se cubre en los Capítulos 8 y 9. Además, si hay varios trabajos preparados para ejecutarse al mismo tiempo, el sistema debe elegir entre ellos. La toma de esta decisión es lo que se denomina **planificación de la CPU**, que se estudia también en el Capítulo 5. Por último, ejecutar varios trabajos concurrentemente requiere que la capacidad de los trabajos para afectarse entre sí esté limitada en todas las fases del sistema operativo, inclu-

yendo la planificación de procesos, el almacenamiento en disco y la gestión de la memoria. Estas consideraciones se abordan a todo lo largo del libro.

En un sistema de tiempo compartido, el sistema operativo debe asegurar un tiempo de respuesta razonable, lo que en ocasiones se hace a través de un mecanismo de **intercambio**, donde los procesos se intercambian entrando y saliendo de la memoria al disco. Un método más habitual de conseguir este objetivo es la **memoria virtual**, una técnica que permite la ejecución de un proceso que no está completamente en memoria (Capítulo 9). La ventaja principal del esquema de memoria virtual es que permite a los usuarios ejecutar programas que sean más grandes que la **memoria física** real. Además, realiza la abstracción de la memoria principal, sustituyéndola desde el punto de vista lógico por una matriz uniforme de almacenamiento de gran tamaño, separando así la **memoria lógica**, como la ve el usuario, de la memoria física. Esta disposición libera a los programadores de preocuparse por las limitaciones de almacenamiento en memoria.

Los sistemas de tiempo compartido también tienen que proporcionar un sistema de archivos (Capítulos 10 y 11). El sistema de archivos reside en una colección de discos; por tanto, deberán proporcionarse también mecanismos de gestión de discos (Capítulo 12). Los sistemas de tiempo compartido también suministran un mecanismo para proteger los recursos frente a usos inapropiados (Capítulo 14). Para asegurar una ejecución ordenada, el sistema debe proporcionar mecanismos para la comunicación y sincronización de trabajos (Capítulo 6) y debe asegurar que los trabajos no darán lugar a interbloqueos que pudieran hacer que quedaran en espera permanentemente (Capítulo 7).

1.5 Operaciones del sistema operativo

Como se ha mencionado anteriormente, los sistemas operativos modernos están **controlados mediante interrupciones**. Si no hay ningún proceso que ejecutar, ningún dispositivo de E/S al que dar servicio y ningún usuario al que responder, un sistema operativo debe permanecer inactivo, esperando a que algo ocurra. Los sucesos casi siempre se indican mediante la ocurrencia de una interrupción o una excepción. Una **excepción** es una interrupción generada por software, debida a un error (por ejemplo, una división por cero o un acceso a memoria no válido) o a una solicitud específica de un programa de usuario de que se realice un servicio del sistema operativo. La característica de un sistema operativo de estar controlado mediante interrupciones define la estructura general de dicho sistema. Para cada tipo de interrupción, diferentes segmentos de código del sistema operativo determinan qué acción hay que llevar a cabo. Se utilizará una rutina de servicio a la interrupción que es responsable de tratarla.

Dado que el sistema operativo y los usuarios comparten los recursos hardware y software del sistema informático, necesitamos asegurar que un error que se produzca en un programa de usuario sólo genere problemas en el programa que se estuviera ejecutando. Con la compartición, muchos procesos podrían verse afectados negativamente por un fallo en otro programa. Por ejemplo, si un proceso entra en un bucle infinito, este bucle podría impedir el correcto funcionamiento de muchos otros procesos. En un sistema de multiprogramación se pueden producir errores más sutiles, pudiendo ocurrir que un programa erróneo modifique otro programa, los datos de otro programa o incluso al propio sistema operativo.

Sin protección frente a este tipo de errores, o la computadora sólo ejecuta un proceso cada vez o todas las salidas deben considerarse sospechosas. Un sistema operativo diseñado apropiadamente debe asegurar que un programa incorrecto (ó malicioso) no pueda dar lugar a que otros programas se ejecuten incorrectamente.

1.5.1 Operación en modo dual

Para asegurar la correcta ejecución del sistema operativo, tenemos que poder distinguir entre la ejecución del código del sistema operativo y del código definido por el usuario. El método que usan la mayoría de los sistemas informáticos consiste en proporcionar soporte hardware que nos permita diferenciar entre varios modos de ejecución.

Como mínimo, necesitamos dos modos diferentes de operación: **modo usuario** y **modo kernel** (también denominado **modo de supervisor**, **modo del sistema** o **modo privilegiado**). Un bit, denominado **bit de modo**, se añade al hardware de la computadora para indicar el modo actual: *kernel* (0) o *usuario* (1). Con el bit de modo podemos diferenciar entre una tarea que se ejecute en nombre del sistema operativo y otra que se ejecute en nombre del usuario. Cuando el sistema informático está ejecutando una aplicación de usuario, el sistema se encuentra en modo de usuario. Sin embargo, cuando una aplicación de usuario solicita un servicio del sistema operativo (a través de una llamada al sistema), debe pasar del modo de usuario al modo *kernel* para satisfacer la solicitud. Esto se muestra en la Figura 1.8. Como veremos, esta mejora en la arquitectura resulta útil también para muchos otros aspectos del sistema operativo.

Cuando se arranca el sistema, el hardware se inicia en el modo *kernel*. El sistema operativo se carga y se inician las aplicaciones de usuario en el modo *usuario*. Cuando se produce una excepción o interrupción, el hardware conmuta del modo de usuario al modo *kernel* (es decir, cambia el estado del bit de modo a 0). En consecuencia, cuando el sistema operativo obtiene el control de la computadora, estará en el modo *kernel*. El sistema siempre cambia al modo de usuario (poniendo el bit de modo a 1) antes de pasar el control a un programa de usuario.

El modo dual de operación nos proporciona los medios para proteger el sistema operativo de los usuarios que puedan causar errores, y también para proteger a los usuarios de los errores de otros usuarios. Esta protección se consigue designando algunas de las instrucciones de máquina que pueden causar daño como **instrucciones privilegiadas**. El hardware hace que las instrucciones privilegiadas sólo se ejecuten en el modo *kernel*. Si se hace un intento de ejecutar una instrucción privilegiada en modo de usuario, el hardware no ejecuta la instrucción sino que la trata como ilegal y envía una excepción al sistema operativo.

La instrucción para conmutar al modo *usuario* es un ejemplo de instrucción privilegiada. Entre otros ejemplos se incluyen el control de E/S, la gestión del temporizador y la gestión de interrupciones. A medida que avancemos a lo largo del texto, veremos que hay muchas instrucciones privilegiadas adicionales.

Ahora podemos ver el ciclo de vida de la ejecución de una instrucción en un sistema informático. El control se encuentra inicialmente en manos del sistema operativo, donde las instrucciones se ejecutan en el modo *kernel*. Cuando se da el control a una aplicación de usuario, se pasa a modo *usuario*. Finalmente, el control se devuelve al sistema operativo a través de una interrupción, una excepción o una llamada al sistema.

Las llamadas al sistema proporcionan los medios para que un programa de usuario pida al sistema operativo que realice tareas reservadas del sistema operativo en nombre del programa del usuario. Una llamada al sistema se invoca de diversas maneras, dependiendo de la funcionalidad proporcionada por el procesador subyacente. En todas sus formas, se trata de un método usado por un proceso para solicitar la actuación del sistema operativo. Normalmente, una llamada al sistema toma la forma de una excepción que efectúa una transferencia a una posición específica en el vector de interrupción. Esta excepción puede ser ejecutada mediante una instrucción genérica `trap`, aunque algunos sistemas (como la familia MIPS R2000) tienen una instrucción `syscall` específica.

Cuando se ejecuta una llamada al sistema, el hardware la trata como una interrupción software. El control pasa a través del vector de interrupción a una rutina de servicio del sistema opera-

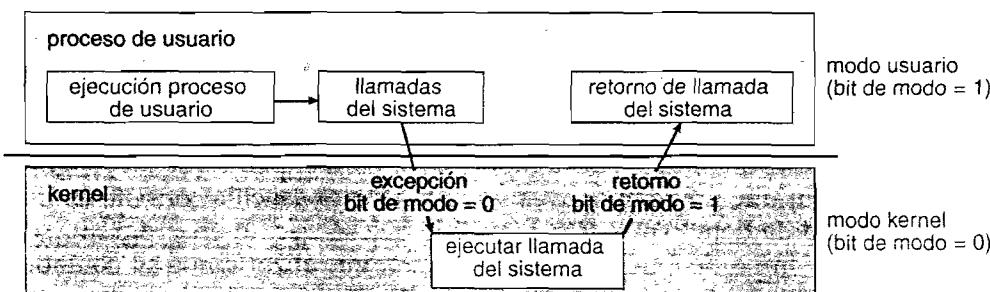


Figura 1.8 Transición del modo usuario al modo *kernel*.

tivo, y el bit de modo se establece en el modo *kernel*. La rutina de servicio de la llamada al sistema es una parte del sistema operativo. El *kernel* examina la instrucción que interrumpe para determinar qué llamada al sistema se ha producido; un parámetro indica qué tipo de servicio está requiriendo el programa del usuario. Puede pasarse la información adicional necesaria para la solicitud mediante registros, mediante la pila o mediante la memoria (pasando en los registros una serie de punteros a posiciones de memoria). El *kernel* verifica que los parámetros son correctos y legales, ejecuta la solicitud y devuelve el control a la instrucción siguiente a la de la llamada de servicio. En la Sección 2.3 se describen de forma más completa las llamadas al sistema.

La falta de un modo dual soportado por hardware puede dar lugar a serios defectos en un sistema operativo. Por ejemplo, MS-DOS fue escrito para la arquitectura 8088 de Intel, que no dispone de bit de modo y, por tanto, tampoco del modo dual. Un programa de usuario que se ejecute erróneamente puede corromper el sistema operativo sobre escribiendo sus archivos; y múltiples programas podrían escribir en un dispositivo al mismo tiempo, con resultados posiblemente desastrosos. Las versiones recientes de la CPU de Intel, como por ejemplo Pentium, sí que proporcionan operación en modo dual. De acuerdo con esto, la mayoría de los sistemas operativos actuales, como Microsoft Windows 2000, Windows XP, Linux y Solaris para los sistemas x86, se aprovechan de esta característica y proporcionan una protección mucho mayor al sistema operativo.

Una vez que se dispone de la protección hardware, el hardware detecta los errores de violación de los modos. Normalmente, el sistema operativo se encarga de tratar estos errores. Si un programa de usuario falla de alguna forma, como por ejemplo haciendo un intento de ejecutar una instrucción ilegal o de acceder a una zona de memoria que no esté en el espacio de memoria del usuario, entonces el hardware envía una excepción al sistema operativo. La excepción transfiere el control al sistema operativo a través del vector de interrupción, igual que cuando se produce una interrupción. Cuando se produce un error de programa, el sistema operativo debe terminar el programa anormalmente. Esta situación se trata con el mismo código software que se usa en una terminación anormal solicitada por el usuario. El sistema proporciona un mensaje de error apropiado y puede volcarse la memoria del programa. Normalmente, el volcado de memoria se escribe en un archivo con el fin de que el usuario o programador puedan examinarlo y quizás corregir y reiniciar el programa.

1.5.2 Temporizador

Debemos asegurar que el sistema operativo mantenga el control sobre la CPU. Por ejemplo, debemos impedir que un programa de usuario entre en un bucle infinito o que no llame a los servicios del sistema y nunca devuelva el control al sistema operativo. Para alcanzar este objetivo, podemos usar un **temporizador**. Puede configurarse un temporizador para interrumpir a la computadora después de un período especificado. El período puede ser fijo (por ejemplo, 1/60 segundos) o variable (por ejemplo, entre 1 milisegundo y 1 segundo). Generalmente, se implementa un **temporizador variable** mediante un reloj de frecuencia fija y un contador. El sistema operativo configura el contador. Cada vez que el reloj avanza, el contador se decrementa. Cuando el contador alcanza el valor 0, se produce una interrupción. Por ejemplo, un contador de 10 bits con un reloj de 1 milisegundo permite interrupciones a intervalos de entre 1 milisegundo y 1.024 milisegundos, en pasos de 1 milisegundo.

Antes de devolver el control al usuario, el sistema operativo se asegura de que el temporizador esté configurado para realizar interrupciones. Cuando el temporizador interrumpe, el control se transfiere automáticamente al sistema operativo, que puede tratar la interrupción como un error fatal o puede conceder más tiempo al programa. Evidentemente, las instrucciones que modifican el contenido del temporizador son instrucciones privilegiadas.

Por tanto, podemos usar el temporizador para impedir que un programa de usuario se esté ejecutando durante un tiempo excesivo. Una técnica sencilla consiste en inicializar un contador con la cantidad de tiempo que esté permitido que se ejecute un programa. Para un programa con un límite de tiempo de 7 minutos, por ejemplo, inicializaríamos su contador con el valor 420. Cada segundo, el temporizador interrumpe y el contador se decrementa en una unidad. Mientras que el valor del contador sea positivo, el control se devuelve al programa de usuario. Cuando el valor

del contador pasa a ser negativo, el sistema operativo termina el programa, por haber sido excedido el límite de tiempo asignado.

1.6 Gestión de procesos

Un programa no hace nada a menos que una CPU ejecute sus instrucciones. Un programa en ejecución, como ya hemos dicho, es un proceso. Un programa de usuario de tiempo compartido, como por ejemplo un compilador, es un proceso. Un procesador de textos que ejecute un usuario individual en un PC es un proceso. Una tarea del sistema, como enviar datos de salida a una impresora, también puede ser un proceso (o al menos, una parte de un proceso). Por el momento, vamos a considerar que un proceso es un trabajo o un programa en tiempo compartido, aunque más adelante veremos que el concepto es más general. Como veremos en el Capítulo 3, es posible proporcionar llamadas al sistema que permitan a los procesos crear subprocessos que se ejecuten de forma concurrente.

Un proceso necesita para llevar a cabo su tarea ciertos recursos, entre los que incluyen tiempo de CPU, memoria, archivos y dispositivos de E/S. Estos recursos se proporcionan al proceso en el momento de crearlo o se le asignan mientras se está ejecutando. Además de los diversos recursos físicos y lógicos que un proceso obtiene en el momento de su creación, pueden pasársele diversos datos de inicialización (entradas). Por ejemplo, considere un proceso cuya función sea la de mostrar el estado de un archivo en la pantalla de un terminal. A ese proceso le proporcionaríamos como entrada el nombre del archivo y el proceso ejecutaría las apropiadas instrucciones y llamadas al sistema para obtener y mostrar en el terminal la información deseada. Cuando el proceso termina, el sistema operativo reclama todos los recursos reutilizables.

Hagamos hincapié en que un programa por sí solo no es un proceso; un programa es una entidad *pasiva*, tal como los contenidos de un archivo almacenado en disco, mientras que un proceso es una entidad *activa*. Un proceso de una sola hebra tiene un **contador de programa** que especifica la siguiente instrucción que hay que ejecutar, (las hebras se verán en el Capítulo 4). La ejecución de un proceso así debe ser secuencial: la CPU ejecuta una instrucción del proceso después de otra, hasta completarlo. Además, en cualquier instante, se estará ejecutando como mucho una instrucción en nombre del proceso. Por tanto, aunque pueda haber dos procesos asociados con el mismo programa, se considerarían no obstante como dos secuencias de ejecución separadas. Un proceso multihebra tiene múltiples contadores de programa, apuntado cada uno de ellos a la siguiente instrucción que haya que ejecutar para una hebra determinada.

Un proceso es una unidad de trabajo en un sistema. Cada sistema consta de una colección de procesos, siendo algunos de ellos procesos del sistema operativo (aquellos que ejecutan código del sistema) y el resto procesos de usuario (aquellos que ejecutan código del usuario). Todos estos procesos pueden, potencialmente, ejecutarse de forma concurrente, por ejemplo multiplexando la CPU cuando sólo se disponga de una.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de procesos:

- Crear y borrar los procesos de usuario y del sistema.
- Suspender y reanudar los procesos.
- Proporcionar mecanismos para la sincronización de procesos.
- Proporcionar mecanismos para la comunicación entre procesos.
- Proporcionar mecanismos para el tratamiento de los interbloqueos.

En los Capítulos 3 a 6 se estudian las técnicas de gestión de procesos.

1.7 Gestión de memoria

Como se ha visto en la Sección 1.2.2, la memoria principal es fundamental en la operación de un sistema informático moderno. La memoria principal es una matriz de palabras o bytes cuyo tama-

ño se encuentra en el rango de cientos de miles a miles de millones de posiciones distintas. Cada palabra o byte tiene su propia dirección. La memoria principal es un repositorio de datos rápidamente accesibles, compartida por la CPU y los dispositivos de E/S. El procesador central lee las instrucciones de la memoria principal durante el ciclo de extracción de instrucciones y lee y escribe datos en la memoria principal durante el ciclo de extracción de datos (en una arquitectura Von Neumann). La memoria principal es, generalmente, el único dispositivo de almacenamiento de gran tamaño al que la CPU puede dirigirse y acceder directamente. Por ejemplo, para que la CPU procese datos de un disco, dichos datos deben transferirse en primer lugar a la memoria principal mediante llamadas de E/S generadas por la CPU. Del mismo modo, las instrucciones deben estar en memoria para que la CPU las ejecute.

Para que un programa pueda ejecutarse, debe estar asignado a direcciones absolutas y cargado en memoria. Mientras el programa se está ejecutando, accede a las instrucciones y a los datos de la memoria generando dichas direcciones absolutas. Finalmente, el programa termina, su espacio de memoria se declara disponible y el siguiente programa puede ser cargado y ejecutado.

Para mejorar tanto la utilización de la CPU como la velocidad de respuesta de la computadora frente a los usuarios, las computadoras de propósito general pueden mantener varios programas en memoria, lo que crea la necesidad de mecanismos de gestión de la memoria. Se utilizan muchos esquemas diferentes de gestión de la memoria. Estos esquemas utilizan enfoques distintos y la efectividad de cualquier algoritmo dado depende de la situación. En la selección de un esquema de gestión de memoria para un sistema específico, debemos tener en cuenta muchos factores, especialmente relativos al diseño *hardware* del sistema. Cada algoritmo requiere su propio soporte hardware.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de memoria:

- Controlar qué partes de la memoria están actualmente en uso y por parte de quién.
- Decidir qué datos y procesos (o partes de procesos) añadir o extraer de la memoria.
- Asignar y liberar la asignación de espacio de memoria según sea necesario.

En los Capítulos 8 y 9 se estudian las técnicas de gestión de la memoria.

1.8 Gestión de almacenamiento

Para que el sistema informático sea cómodo para los usuarios, el sistema operativo proporciona una vista lógica y uniforme del sistema de almacenamiento de la información. El sistema operativo abstrae las propiedades físicas de los dispositivos de almacenamiento y define una unidad de almacenamiento lógico, el **archivo**. El sistema operativo asigna los archivos a los soportes físicos y accede a dichos archivos a través de los dispositivos de almacenamiento.

1.8.1 Gestión del sistema de archivos

La gestión de archivos es uno de los componentes más visibles de un sistema operativo. Las computadoras pueden almacenar la información en diferentes tipos de medios físicos. Los discos magnéticos, discos ópticos y cintas magnéticas son los más habituales. Cada uno de estos medios tiene sus propias características y organización física. Cada medio se controla mediante un dispositivo, tal como una unidad de disco o una unidad de cinta, que también tiene sus propias características distintivas. Estas propiedades incluyen la velocidad de acceso, la capacidad, la velocidad de transferencia de datos y el método de acceso (secuencial o aleatorio).

Un archivo es una colección de información relacionada definida por su creador. Comúnmente, los archivos representan programas (tanto en formato fuente como objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos, alfanuméricos o binarios. Los archivos pueden tener un formato libre (como, por ejemplo, los archivos de texto) o un formato rígido, como por ejemplo una serie de campos fijos. Evidentemente, el concepto de archivo es extremadamente general.

El sistema operativo implementa el abstracto concepto de archivo gestionando los medios de almacenamiento masivos, como las cintas y discos, y los dispositivos que los controlan. Asimismo, los archivos normalmente se organizan en directorios para hacer más fácil su uso. Por último, cuando varios usuarios tienen acceso a los archivos, puede ser deseable controlar quién y en qué forma (por ejemplo, lectura, escritura o modificación) accede a los archivos.

El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de archivos:

- Creación y borrado de archivos.
- Creación y borrado de directorios para organizar los archivos.
- Soporte de primitivas para manipular archivos y directorios.
- Asignación de archivos a los dispositivos de almacenamiento secundario.
- Copia de seguridad de los archivos en medios de almacenamiento estables (no volátiles).

Las técnicas de gestión de archivos se tratan en los Capítulos 10 y 11.

1.8.2 Gestión del almacenamiento masivo

Como ya hemos visto, dado que la memoria principal es demasiado pequeña para acomodar todos los datos y programas, y puesto que los datos que guarda se pierden al desconectar la alimentación, el sistema informático debe proporcionar un almacenamiento secundario como respaldo de la memoria principal. La mayoría de los sistemas informáticos modernos usan discos como principal medio de almacenamiento en línea, tanto para los programas como para los datos. La mayor parte de los programas, incluyendo compiladores, ensambladores o procesadores de texto, se almacenan en un disco hasta que se cargan en memoria, y luego usan el disco como origen y destino de su procesamiento. Por tanto, la apropiada gestión del almacenamiento en disco tiene una importancia crucial en un sistema informático. El sistema operativo es responsable de las siguientes actividades en lo que se refiere a la gestión de disco:

- Gestión del espacio libre.
- Asignación del espacio de almacenamiento.
- Planificación del disco.

Dado que el almacenamiento secundario se usa con frecuencia, debe emplearse de forma eficiente. La velocidad final de operación de una computadora puede depender de las velocidades del subsistema de disco y de los algoritmos que manipulan dicho subsistema.

Sin embargo, hay muchos usos para otros sistemas de almacenamiento más lentos y más baratos (y que, en ocasiones, proporcionan una mayor capacidad) que el almacenamiento secundario. La realización de copias de seguridad de los datos del disco, el almacenamiento de datos raramente utilizados y el almacenamiento definitivo a largo plazo son algunos ejemplos.

Las unidades de cinta magnética y sus cintas y las unidades de CD y DVD y sus discos son dispositivos típicos de **almacenamiento terciario**. Los soportes físicos (cintas y discos ópticos) varían entre los formatos **WORM** (write-once, read-many-times; escritura una vez, lectura muchas veces) y **RW** (read-write, lectura-escritura).

El almacenamiento terciario no es crucial para el rendimiento del sistema, aunque también es necesario gestionarlo. Algunos sistemas operativos realizan esta tarea, mientras que otros dejan el control del almacenamiento terciario a los programas de aplicación. Algunas de las funciones que dichos sistemas operativos pueden proporcionar son el montaje y desmontaje de medios en los dispositivos, la asignación y liberación de dispositivos para su uso exclusivo por los procesos, y la migración de datos del almacenamiento secundario al terciario.

Las técnicas para la gestión del almacenamiento secundario y terciario se estudian en el Capítulo 12.

1.8.3 Almacenamiento en caché

El **almacenamiento en caché** es una técnica importante en los sistemas informáticos. Normalmente, la información se mantiene en algún sistema de almacenamiento, como por ejemplo la memoria principal. Cuando se usa, esa información se copia de forma temporal en un sistema de almacenamiento más rápido, la caché. Cuando necesitamos una información particular, primero comprobamos si está en la caché. Si lo está, usamos directamente dicha información de la caché; en caso contrario, utilizamos la información original, colocando una copia en la caché bajo la suposición de que pronto la necesitaremos nuevamente.

Además, los registros programables internos, como los registros de índice, proporcionan una caché de alta velocidad para la memoria principal. El programador (o compilador) implementa los algoritmos de asignación de recursos y de reemplazamiento de registros para decidir qué información mantener en los registros y cuál en la memoria principal. También hay disponibles cachés que se implementan totalmente mediante hardware. Por ejemplo, la mayoría de los sistemas disponen de una caché de instrucciones para almacenar las siguientes instrucciones en espera de ser ejecutadas. Sin esta caché, la CPU tendría que esperar varios ciclos mientras las instrucciones son extraídas de la memoria principal. Por razones similares, la mayoría de los sistemas disponen de una o más cachés de datos de alta velocidad en la jerarquía de memoria. En este libro no vamos a ocuparnos de estas cachés implementadas totalmente mediante hardware, ya que quedan fuera del control del sistema operativo.

Dado que las cachés tienen un tamaño limitado, la **gestión de la caché** es un problema de diseño importante. La selección cuidadosa del tamaño de la caché y de una adecuada política de reemplazamiento puede dar como un resultado un incremento enorme del rendimiento. Consulte la Figura 1.9 para ver una comparativa de las prestaciones de almacenamiento en las estaciones de trabajo grandes y pequeños servidores; dicha comparativa ilustra perfectamente la necesidad de usar el almacenamiento en caché. En el Capítulo 9 se exponen varios algoritmos de reemplazamiento para cachés controladas por software.

La memoria principal puede verse como una caché rápida para el almacenamiento secundario, ya que los datos en almacenamiento secundario deben copiarse en la memoria principal para poder ser utilizados, y los datos deben encontrarse en la memoria principal antes de ser pasados al almacenamiento secundario cuando llega el momento de guardarlos. Los datos del sistema de archivos, que residen permanentemente en el almacenamiento secundario, pueden aparecer en varios niveles de la jerarquía de almacenamiento. En el nivel superior, el sistema operativo puede mantener una caché de datos del sistema de archivos en la memoria principal. También pueden utilizarse discos RAM (también conocidos como **discos de estado sólido**) para almacenamiento de alta velocidad, accediendo a dichos discos a través de la interfaz del sistema de archivos. La mayor parte del almacenamiento secundario se hace en discos magnéticos. Los datos almacenados en disco magnético, a su vez, se copian a menudo en cintas magnéticas o discos extraíbles con el fin de protegerlos frente a pérdidas de datos en caso de que un disco duro falle. Algunos sistemas realizan automáticamente un archivado definitivo de los datos correspondientes a los archivos antiguos, transfiriéndolos desde el almacenamiento secundario a un almacenamiento terciario, como por ejemplo un servidor de cintas, con el fin de reducir costes de almacenamiento (véase el Capítulo 12).

El movimiento de información entre niveles de una jerarquía de almacenamiento puede ser explícito o implícito, dependiendo del diseño hardware y del software del sistema operativo que controle dicha funcionalidad. Por ejemplo, la transferencia de datos de la caché a la CPU y los registros es, normalmente, una función hardware en la que no interviene el sistema operativo. Por el contrario, la transferencia de datos de disco a memoria normalmente es controlada por el sistema operativo.

En una estructura de almacenamiento jerárquica, los mismos datos pueden aparecer en diferentes niveles del sistema de almacenamiento. Por ejemplo, suponga que un entero A que hay que incrementar en 1 se encuentra almacenado en el archivo B, el cual reside en un disco magnético. La operación de incremento ejecuta primero una operación de E/S para copiar el bloque de disco en el que reside A a la memoria principal. A continuación se copia A en la caché y en un registro

Nivel	1	2	3	4
Nombre	registros	caché	memoria principal	almacenamiento en disco
Tamaño típico	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Tecnología de implementación	memoria custom con múltiples puertos, CMOS on-chip u off-chip	SRAM-CMOS	CMOS-DRAM	disco magnético
Tiempo de acceso (ns)	0,25 – 0,5	0,5 – 25	80 – 250	5.000.000
Ancho de banda (MB/s)	20.000 – 100.000	5000 – 10.000	1000 – 5000	20 – 150
Gestionado por	compilador	hardware	sistema operativo	sistema operativo
Copiado en	caché	memoria principal	disco	CD o cinta

Figura 1.9. Prestaciones de los distintos niveles de almacenamiento.

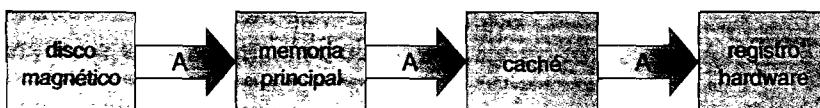


Figura 1.10 Migración de un entero A del disco a un registro.

interno. Por tanto, la copia de A aparece en varios lugares: en el disco magnético, en la memoria principal, en la caché y en un registro interno (véase la Figura 1.10). Una vez que se hace el incremento en el registro interno, el valor de A es distinto en los diversos sistemas de almacenamiento. El valor de A sólo será el mismo después de que su nuevo valor se escriba desde el registro interno al disco magnético.

En un entorno informático donde sólo se ejecuta un proceso cada vez, este proceso no plantea ninguna dificultad, ya que un acceso al entero A siempre se realizará a la copia situada en el nivel más alto de la jerarquía. Sin embargo, en un entorno multitarea, en el que la CPU comunica entre varios procesos, hay que tener un extremo cuidado para asegurar que, si varios procesos desean acceder a A, cada uno de ellos obtenga el valor más recientemente actualizado de A.

La situación se hace más complicada en un entorno multiprocesador donde, además de mantener registros internos, cada una de las CPU también contiene una caché local. En un entorno de este tipo, una copia de A puede encontrarse simultáneamente en varias cachés. Dado que las diversas CPU pueden ejecutar instrucciones concurrentemente, debemos asegurarnos de que una actualización del valor de A en una caché se refleje inmediatamente en las restantes cachés en las que reside A. Esta situación se denomina *coherencia de caché* y, normalmente, se trata de un problema de hardware, que se gestiona por debajo del nivel del sistema operativo.

En un entorno distribuido, la situación se hace incluso más compleja. En este tipo de entorno, varias copias (o réplicas) del mismo archivo pueden estar en diferentes computadoras distribuidas geográficamente. Dado que se puede acceder de forma concurrente a dichas réplicas y actualizarlas, algunos sistemas distribuidos garantizan que, cuando una réplica se actualiza en un sitio, todas las demás réplicas se actualizan lo más rápidamente posible. Existen varias formas de proporcionar esta garantía, como se verá en el Capítulo 17.

1.8.4 Sistemas de E/S

Uno de los propósitos de un sistema operativo es ocultar al usuario las peculiaridades de los dispositivos hardware específicos. Por ejemplo, en UNIX, las peculiaridades de los dispositivos de E/S se ocultan a la mayor parte del propio sistema operativo mediante el **subsistema de E/S**. El subsistema de E/S consta de varios componentes:

- Un componente de gestión de memoria que incluye almacenamiento en búfer, gestión de caché y gestión de colas.
- Una interfaz general para controladores de dispositivo.

- Controladores para dispositivos hardware específicos.

Sólo el controlador del dispositivo conoce las peculiaridades del dispositivo específico al que está asignado.

En la Sección 1.2.3 se expone cómo se usan las rutinas de tratamiento de interrupciones y los controladores de dispositivo en la construcción de subsistemas de E/S eficientes. En el Capítulo 13 se aborda el modo en que el subsistema de E/S interactúa con los otros componentes del sistema, gestiona los dispositivos, transfiere datos y detecta que las operaciones de E/S han concluido.

1.9 Protección y seguridad

Si un sistema informático tiene múltiples usuarios y permite la ejecución concurrente de múltiples procesos, entonces el acceso a los datos debe regularse. Para dicho propósito, se emplean mecanismos que aseguren que sólo puedan utilizar los recursos (archivos, segmentos de memoria, CPU y otros) aquellos procesos que hayan obtenido la apropiada autorización del sistema operativo. Por ejemplo, el hardware de direccionamiento de memoria asegura que un proceso sólo se pueda ejecutar dentro de su propio espacio de memoria; el temporizador asegura que ningún proceso pueda obtener el control de la CPU sin después ceder el control; los usuarios no pueden acceder a los registros de control, por lo que la integridad de los diversos dispositivos periféricos está protegida, etc.

Por tanto, **protección** es cualquier mecanismo que controle el acceso de procesos y usuarios a los recursos definidos por un sistema informático. Este mecanismo debe proporcionar los medios para la especificación de los controles que hay que imponer y para la aplicación de dichos controles.

Los mecanismos de protección pueden mejorar la fiabilidad, permitiendo detectar errores latentes en las interfaces entre los subsistemas componentes. La detección temprana de los errores de interfaz a menudo puede evitar la contaminación de un subsistema que funciona perfectamente por parte de otro subsistema que funcione mal. Un recurso desprotegido no puede defenderse contra el uso (o mal uso) de un usuario no autorizado o incompetente. Un sistema orientado a la protección proporciona un medio para distinguir entre un uso autorizado y no autorizado, como se explica en el Capítulo 14.

Un sistema puede tener la protección adecuada pero estar expuesto a fallos y permitir accesos inapropiados. Considere un usuario al que le han robado su información de autenticación (los medios de identificarse ante el sistema); sus datos podrían ser copiados o borrados, incluso aunque esté funcionando la protección de archivos y de memoria. Es responsabilidad de los mecanismos de **seguridad** defender al sistema frente a ataques internos y externos. Tales ataques abarcan un enorme rango, en el que se incluyen los virus y gusanos, los ataques de denegación de servicio (que usan todos los recursos del sistema y mantienen a los usuarios legítimos fuera del sistema), el robo de identidad y el robo de servicio (el uso no autorizado de un sistema). La prevención de algunos de estos ataques se considera una función del sistema operativo en algunos sistemas, mientras que en otros se deja a la política de prevención o a algún software adicional. Debido a la creciente alarma en lo que se refiere a incidentes de seguridad, las características de seguridad del sistema operativo constituyen un área de investigación e implementación en rápido crecimiento. Los temas relativos a la seguridad se estudian en el Capítulo 15.

La protección y la seguridad requieren que el sistema pueda distinguir a todos sus usuarios. La mayoría de los sistemas operativos mantienen una lista con los nombres de usuario y sus **identificadores de usuario** (ID) asociados. En la jerga de Windows NT, esto se denomina **ID de seguridad (SID, security ID)**. Estos ID numéricos son únicos, uno por usuario. Cuando un usuario inicia una sesión en el sistema, la fase de autenticación determina el ID correspondiente a dicho usuario. Ese ID de usuario estará asociado con todos los procesos y hebras del usuario. Cuando un ID necesita poder ser leído por los usuarios, se utiliza la lista de nombres de usuario para traducir el ID al nombre correspondiente.

En algunas circunstancias, es deseable diferenciar entre conjuntos de usuarios en lugar de entre usuarios individuales. Por ejemplo, el propietario de un archivo en un sistema UNIX puede ejecu-

tar todas las operaciones sobre dicho archivo, mientras que a un conjunto seleccionado de usuarios podría permitírselle sólo leer el archivo. Para conseguir esto, es necesario definir un nombre de grupo y especificar los usuarios que pertenezcan a dicho grupo. La funcionalidad de grupo se puede implementar manteniendo en el sistema una lista de nombres de grupo e **identificadores de grupo**. Un usuario puede pertenecer a uno o más grupos, dependiendo de las decisiones de diseño del sistema operativo. Los ID de grupo del usuario también se incluyen en todos los procesos y hebras asociados.

Durante el uso normal de un sistema, el ID de usuario y el ID de grupo de un usuario son suficientes. Sin embargo, en ocasiones, un usuario necesita **escalar sus privilegios** para obtener permisos adicionales para una actividad. Por ejemplo, el usuario puede necesitar acceder a un dispositivo que está restringido. Los sistemas operativos proporcionan varios métodos para el escalado de privilegios. Por ejemplo, en UNIX, el atributo `setuid` en un programa hace que dicho programa se ejecute con el ID de usuario del propietario del archivo, en lugar de con el ID del usuario actual. El proceso se ejecuta con este **UID efectivo** hasta que se desactivan los privilegios adicionales o se termina el proceso. Veamos un ejemplo de cómo se hace esto en Solaris 10: el usuario pbg podría tener un ID de usuario igual a 101 y un ID de grupo igual a 14, los cuales se asignarían mediante `/etc/passwd:pbg:x:101:14::/export/home/pbg:/usr/bin/bash`.

1.10 Sistemas distribuidos

Un sistema distribuido es una colección de computadoras físicamente separadas y posiblemente heterogéneas que están conectadas en red para proporcionar a los usuarios acceso a los diversos recursos que el sistema mantiene. Acceder a un recurso compartido incrementa la velocidad de cálculo, la funcionalidad, la disponibilidad de los datos y la fiabilidad. Algunos sistemas operativos generalizan el acceso a red como una forma de acceso a archivo, manteniendo los detalles de la conexión de red en el controlador de dispositivo de la interfaz de red. Otros sistemas operativos invocan específicamente una serie de funciones de red. Generalmente, los sistemas contienen una mezcla de los dos modos, como por ejemplo FTP y NFS. Los protocolos que forman un sistema distribuido pueden afectar enormemente a la popularidad y utilidad de dicho sistema.

En términos sencillos, una **red** es una vía de comunicación entre dos o más sistemas. La funcionalidad de los sistemas distribuidos depende de la red. Las redes varían según el protocolo que usen, las distancias entre los nodos y el medio de transporte. TCP/IP es el protocolo de red más común, aunque el uso de ATM y otros protocolos está bastante extendido. Asimismo, los protocolos soportados varían de unos sistemas operativos a otros. La mayoría de los sistemas operativos soportan TCP/IP, incluidos los sistemas operativos Windows y UNIX. Algunos sistemas soportan protocolos propietarios para ajustarse a sus necesidades. En un sistema operativo, un protocolo de red simplemente necesita un dispositivo de interfaz (por ejemplo, un adaptador de red), con un controlador de dispositivo que lo gestione y un software para tratar los datos. Estos conceptos se explican a lo largo del libro.

Las redes se caracterizan en función de las distancias entre sus nodos. Una **red de área local** (LAN) conecta una serie de computadoras que se encuentran en una misma habitación, planta o edificio. Normalmente, una **red de área extendida** (WAN) conecta varios edificios, ciudades o países; una multinacional puede disponer de una red WAN para conectar sus oficinas en todo el mundo. Estas redes pueden ejecutar uno o varios protocolos y la continua aparición de nuevas tecnologías está dando lugar a nuevas formas de redes. Por ejemplo, una **red de área metropolitana** (MAN, metropolitan-area network) puede conectar diversos edificios de una ciudad. Los dispositivos BlueTooth y 802.11 utilizan tecnología inalámbrica para comunicarse a distancias de unos pocos metros, creando en esencia lo que se denomina una **red de área pequeña**, como la que puede haber en cualquier hogar.

Los soportes físicos o medios que se utilizan para implementar las redes son igualmente variados. Entre ellos se incluyen el cobre, la fibra óptica y las transmisiones inalámbricas entre satélites, antenas de microondas y aparatos de radio. Cuando se conectan los dispositivos informáticos a teléfonos móviles, se puede crear una red. También se puede emplear incluso comunicación por infrarrojos de corto alcance para establecer una red. A nivel rudimentario, siempre que las com-

putadoras se comuniquen, estarán usando o creando una red. Todas estas redes también varían en cuanto a su rendimiento y su fiabilidad.

Algunos sistemas operativos han llevado el concepto de redes y sistemas distribuidos más allá de la noción de proporcionar conectividad de red. Un **sistema operativo de red** es un sistema operativo que proporciona funcionalidades como la compartición de archivos a través de la red y que incluye un esquema de comunicación que permite a diferentes procesos, en diferentes computadoras, intercambiar mensajes. Una computadora que ejecuta un sistema operativo de red actúa autónomamente respecto de las restantes computadoras de la red, aunque es consciente de la red y puede comunicarse con los demás equipos conectados en red. Un sistema operativo distribuido proporciona un entorno menos autónomo. Los diferentes sistemas operativos se comunican de modo que se crea la ilusión de que un único sistema operativo controla la red.

En los Capítulos 16 a 18 veremos las redes de computadoras y los sistemas distribuidos.

1.11 Sistemas de propósito general

La exposición ha estado enfocada hasta ahora sobre los sistemas informáticos de propósito general con los que todos estamos familiarizados. Sin embargo, existen diferentes clases de sistemas informáticos cuyas funciones son más limitadas y cuyo objetivo es tratar con dominios de procesamiento limitados.

1.11.1 Sistemas embebidos en tiempo real

Las computadoras embebidas son las computadoras predominantes hoy en día. Estos dispositivos se encuentran por todas partes, desde los motores de automóviles y los robots para fabricación, hasta los magnetoscopios y los hornos de microondas. Estos sistemas suelen tener tareas muy específicas. Los sistemas en los que operan usualmente son primitivos, por lo que los sistemas operativos proporcionan funcionalidades limitadas. Usualmente, disponen de una interfaz de usuario muy limitada o no disponen de ella en absoluto, prefiriendo invertir su tiempo en monitorizar y gestionar dispositivos hardware, como por ejemplo motores de automóvil y brazos robóticos.

Estos sistemas embebidos varían considerablemente. Algunos son computadoras de propósito general que ejecutan sistemas operativos estándar, como UNIX, con aplicaciones de propósito especial para implementar la funcionalidad. Otros son sistemas hardware con sistemas operativos embebidos de propósito especial que sólo proporcionan la funcionalidad deseada. Otros son dispositivos hardware con circuitos integrados específicos de la aplicación (ASIC, application specific integrated circuit), que realizan sus tareas sin ningún sistema operativo.

El uso de sistemas embebidos continúa expandiéndose. La potencia de estos dispositivos, tanto si trabajan como unidades autónomas como si se conectan a redes o a la Web, es seguro que también continuará incrementándose. Incluso ahora, pueden informatizarse casas enteras, de modo que una computadora central (una computadora de propósito general o un sistema embebido) puede controlar la calefacción y la luz, los sistemas de alarma e incluso la cafetera. El acceso web puede permitir a alguien llamar a su casa para poner a calentar el café antes de llegar. Algún día, la nevera llamará al supermercado cuando falte leche.

Los sistemas embebidos casi siempre ejecutan **sistemas operativos en tiempo real**. Un sistema en tiempo real se usa cuando se han establecido rígidos requisitos de tiempo en la operación de un procesador o del flujo de datos; por ello, este tipo de sistema a menudo se usa como dispositivo de control en una aplicación dedicada. Una serie de sensores proporcionan los datos a la computadora. La computadora debe analizar los datos y, posiblemente, ajustar los controles con el fin de modificar los datos de entrada de los sensores. Los sistemas que controlan experimentos científicos, los de imágenes médicas, los de control industrial y ciertos sistemas de visualización son sistemas en tiempo real. Algunos sistemas de inyección de gasolina para motores de automóvil, algunas controladoras de electrodomésticos y algunos sistemas de armamento son también sistemas en tiempo real.

Un sistema en tiempo real tiene restricciones fijas y bien definidas. El procesamiento *tiene que* hacerse dentro de las restricciones definidas o el sistema fallará. Por ejemplo, no sirve de nada instruir a un brazo de robot para que se pare *después* de haber golpeado el coche que estaba construyendo. Un sistema en tiempo real funciona correctamente sólo si proporciona el resultado correcto dentro de sus restricciones de tiempo. Este tipo de sistema contrasta con los sistemas de tiempo compartido, en los que es deseable (aunque no obligatorio) que el sistema responda rápidamente, y también contrasta con los sistemas de procesamiento por lotes, que no tienen ninguna restricción de tiempo en absoluto.

En el Capítulo 19, veremos los sistemas embebidos en tiempo real con más detalle. En el Capítulo 5, presentaremos la facilidad de planificación necesaria para implementar la funcionalidad de tiempo real en un sistema operativo. En el Capítulo 9 se describe el diseño de la gestión de memoria para sistemas en tiempo real. Por último, en el Capítulo 22, describiremos los componentes de tiempo real del sistema operativo Windows XP.

1.11.2 Sistemas multimedia

La mayor parte de los sistemas operativos están diseñados para gestionar datos convencionales, como archivos de texto, programas, documentos de procesadores de textos y hojas de cálculo. Sin embargo, una tendencia reciente en la tecnología informática es la incorporación de **datos multimedia** en los sistemas. Los datos multimedia abarcan tanto archivos de audio y vídeo, como archivos convencionales. Estos datos difieren de los convencionales en que los datos multimedia (como por ejemplo los fotogramas de una secuencia de vídeo) deben suministrarse cumpliendo ciertas restricciones de tiempo (por ejemplo, 30 imágenes por segundo).

La palabra multimedia describe un amplio rango de aplicaciones que hoy en día son de uso popular. Incluye los archivos de audio (por ejemplo MP3), las películas de DVD, la videoconferencia y las secuencias de vídeo con anuncios de películas o noticias que los usuarios descargan a través de Internet. Las aplicaciones multimedia también pueden incluir webcasts en directo (multidifusión a través de la World Wide Web) de conferencias o eventos deportivos, e incluso cámaras web que permiten a un observador que esté en Manhattan ver a los clientes de un café en París. Las aplicaciones multimedia no tienen por qué ser sólo audio o vídeo, sino que a menudo son una combinación de ambos tipos de datos. Por ejemplo, una película puede tener pistas de audio y de vídeo separadas. Además, las aplicaciones multimedia no están limitadas a los PC de escritorio, ya que de manera creciente se están dirigiendo a dispositivos más pequeños, como los PDA y teléfonos móviles. Por ejemplo, un corredor de bolsa puede tener en su PDA en tiempo real y por vía inalámbrica las cotizaciones de bolsa.

En el Capítulo 20, exploramos la demanda de aplicaciones multimedia, analizando en qué difieren los datos multimedia de los datos convencionales y cómo la naturaleza de estos datos afecta al diseño de los sistemas operativos que dan soporte a los requisitos de los sistemas multimedia.

1.11.3 Sistemas de mano

Los **sistemas de mano** incluyen los PDA (personal digital assistant, asistente digital personal), tales como los Palm y Pocket-PC, y los teléfonos móviles, muchos de los cuales usan sistemas operativos embebidos de propósito especial. Los desarrolladores de aplicaciones y sistemas de mano se enfrentan a muchos retos, la mayoría de ellos debidos al tamaño limitado de dichos dispositivos. Por ejemplo, un PDA tiene una altura aproximada de 13 cm y un ancho de 8 cm, y pesa menos de 200 gramos. Debido a su tamaño, la mayoría de los dispositivos de mano tienen muy poca memoria, procesadores lentos y pantallas de visualización pequeñas. Veamos cada una de estas limitaciones.

La cantidad de memoria física en un sistema de mano depende del dispositivo, pero normalmente se encuentra entre 512 KB y 128 MB (compare estos números con un típico PC o una estación de trabajo, que puede tener varios gigabytes de memoria). Como resultado, el sistema operativo y las aplicaciones deben gestionar la memoria de forma muy eficiente. Esto incluye

devolver toda la memoria asignada al gestor de memoria cuando ya no se esté usando. En el Capítulo 9 exploraremos la memoria virtual, que permite a los desarrolladores escribir programas que se comportan como si el sistema tuviera más memoria que la físicamente disponible. Actualmente, no muchos dispositivos de mano usan las técnicas de memoria virtual, por los que los desarrolladores de programas deben trabajar dentro de los confines de la limitada memoria física.

Un segundo problema que afecta a los desarrolladores de dispositivos de mano es la velocidad del procesador usado en los dispositivos. Los procesadores de la mayor parte de los dispositivos de mano funcionan a una fracción de la velocidad de un procesador típico para PC. Los procesadores requieren mayor cantidad de energía cuanto más rápidos son. Para incluir un procesador más rápido en un dispositivo de mano sería necesaria una batería mayor, que ocuparía más espacio o tendría que ser reemplazada (o recargada) con mayor frecuencia. La mayoría de los dispositivos de mano usan procesadores más pequeños y lentos, que consumen menos energía. Por tanto, las aplicaciones y el sistema operativo tienen que diseñarse para no imponer una excesiva carga al procesador.

El último problema al que se enfrentan los diseñadores de programas para dispositivos de mano es la E/S. La falta de espacio físico limita los métodos de entrada a pequeños teclados, sistemas de reconocimiento de escritura manual o pequeños teclados basados en pantalla. Las pequeñas pantallas de visualización limitan asimismo las opciones de salida. Mientras que un monitor de un PC doméstico puede medir hasta 30 pulgadas, la pantalla de un dispositivo de mano a menudo no es más que un cuadrado de 3 pulgadas. Tareas tan familiares como leer un correo electrónico o navegar por diversas páginas web se tienen que condensar en pantallas muy pequeñas. Un método para mostrar el contenido de una página web es el **recorte web**, que consiste en que sólo se suministra y se muestra en el dispositivo de mano un pequeño subconjunto de una página web.

Algunos dispositivos de mano utilizan tecnología inalámbrica, como BlueTooth o 802.11, permitiendo el acceso remoto al correo electrónico y la exploración web. Los teléfonos móviles con conectividad a Internet caen dentro de esta categoría. Sin embargo, para los PDA que no disponen de acceso inalámbrico, descargar datos requiere normalmente que el usuario descargue primero los datos en un PC o en una estación de trabajo y luego transfiera los datos al PDA. Algunos PDA permiten que los datos se copien directamente de un dispositivo a otro usando un enlace de infrarrojos.

Generalmente, las limitaciones en la funcionalidad de los PDA se equilibran con su portabilidad y su carácter práctico. Su uso continúa incrementándose, a medida que hay disponibles más conexiones de red y otras opciones, como cámaras digitales y reproductores MP3, que incrementan su utilidad.

1.12 Entornos informáticos

Hasta ahora, hemos hecho una introducción a la organización de los sistemas informáticos y los principales componentes de los sistemas operativos. Vamos a concluir con una breve introducción sobre cómo se usan los sistemas operativos en una variedad de entornos informáticos.

1.12.1 Sistema informático tradicional

A medida que la informática madura, las líneas que separan muchos de los entornos informáticos tradicionales se difuminan. Considere el “típico entorno de oficina”. Hace unos pocos años este entorno consistía en equipos PC conectados mediante una red, con servidores que proporcionaban servicios de archivos y de impresión. El acceso remoto era difícil y la portabilidad se conseguía mediante el uso de computadoras portátiles. También los terminales conectados a *mainframes* predominaban en muchas empresas, con algunas opciones de acceso remoto y portabilidad.

La tendencia actual se dirige a proporcionar más formas de acceso a estos entornos informáticos. Las tecnologías web están extendiendo los límites de la informática tradicional. Las empresas

establecen **portales**, que proporcionan acceso web a sus servidores internos. Las **computadoras de red** son, esencialmente, terminales que implementan la noción de informática basada en la Web. Las computadoras de mano pueden sincronizarse con los PC, para hacer un uso más portable de la información de la empresa. Los PDA de mano también pueden conectarse a **redes inalámbricas** para usar el portal web de la empresa (así como una multitud de otros recursos web).

En los hogares, la mayoría de los usuarios disponían de una sola computadora con una lenta conexión por módem con la oficina, con Internet o con ambos. Actualmente, las velocidades de conexión de red que antes tenían un precio prohibitivo son ahora relativamente baratas y proporcionan a los usuarios domésticos un mejor acceso a una mayor cantidad de datos. Estas conexiones de datos rápidas están permitiendo a las computadoras domésticas servir páginas web y funcionar en redes que incluyen impresoras, clientes tipo PC y servidores. En algunos hogares se dispone incluso de **cortafuegos** (o servidores de seguridad) para proteger sus redes frente a posibles brechas de seguridad. Estos cortafuegos eran extremadamente caros hace unos años y hace una década ni siquiera existían.

En la segunda mitad del siglo pasado, los recursos informáticos eran escasos (¡y antes, inexistentes!). Durante bastante tiempo, los sistemas estuvieron separados en dos categorías: de procesamiento por lotes e interactivos. Los sistemas de procesamiento por lotes procesaban trabajos masivos, con una entrada predeterminada (procedente de archivos u otros orígenes de datos). Los sistemas interactivos esperaban la introducción de datos por parte del usuario. Para optimizar el uso de los recursos informáticos, varios usuarios compartían el tiempo en estos sistemas. Los sistemas de tiempo compartido empleaban un temporizador y algoritmos de planificación para ejecutar rápidamente una serie de procesos por turnos en la CPU, proporcionando a cada usuario una parte de los recursos.

Hoy en día, los sistemas tradicionales de tiempo compartido no son habituales. Las mismas técnicas de planificación se usan todavía en estaciones de trabajo y servidores, aunque frecuentemente los procesos son todos propiedad del mismo usuario (o del usuario y del sistema operativo). Los procesos de usuario y los procesos del sistema que proporcionan servicios al usuario son gestionados de manera que cada uno tenga derecho frecuentemente a una parte del tiempo. Por ejemplo, considere las ventanas que se muestran mientras un usuario está trabajando en un PC y el hecho de que puede realizar varias tareas al mismo tiempo.

1.12.2 Sistema cliente-servidor

A medida que los PC se han hecho más rápidos, potentes y baratos, los diseñadores han ido abandonando la arquitectura de sistemas centralizada. Los terminales conectados a sistemas centralizados están siendo sustituidos por los PC. Igualmente, la funcionalidad de interfaz de usuario que antes era gestionada directamente por los sistemas centralizados ahora está siendo gestionada de forma cada vez más frecuente en los PC. Como resultado, muchos sistemas actuales actúan como **sistemas servidor** para satisfacer las solicitudes generadas por los **sistemas cliente**. Esta forma de sistema distribuido especializado, denominada **sistema cliente-servidor**, tiene la estructura general descrita en la Figura 1.11.

Los sistemas servidor pueden clasificarse de forma muy general en servidores de cálculo y servidores de archivos.

- El **sistema servidor de cálculo** proporciona una interfaz a la que un cliente puede enviar una solicitud para realizar una acción, como por ejemplo leer datos; en res puesta, el servi-

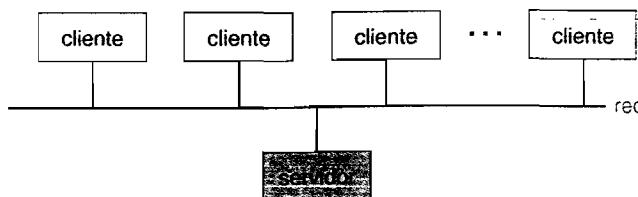


Figura 1.11 Estructura general de un sistema cliente-servidor.

dor ejecuta la acción y devuelve los resultados al cliente. Un servidor que ejecuta una base de datos y responde a las solicitudes de datos del cliente es un ejemplo de sistema de este tipo.

- El **sistema servidor de archivos** proporciona una interfaz de sistema de archivos mediante la que los clientes pueden crear, actualizar, leer y eliminar archivos. Un ejemplo de sistema así es un servidor web que suministra archivos a los clientes que ejecutan exploradores web.

1.12.3 Sistema entre iguales

Otra estructura de sistema distribuido es el modelo de sistema entre iguales (peer-to-peer o P2P). En este modelo, los clientes y servidores no se diferencian entre sí; en su lugar, todos los nodos del sistema se consideran iguales y cada uno puede actuar como cliente o como servidor dependiendo de si solicita o proporciona un servicio. Los sistemas entre iguales ofrecen una ventaja sobre los sistemas cliente-servidor tradicionales. En un sistema cliente-servidor, el servidor es un cuello de botella, pero en un sistema entre iguales, varios nodos distribuidos a través de la red pueden proporcionar los servicios.

Para participar en un sistema entre iguales, un nodo debe en primer lugar unirse a la red de iguales. Una vez que un nodo se ha unido a la red, puede comenzar a proporcionar servicios y solicitar servicios de, otros nodos de la red. La determinación de qué servicios hay disponibles es algo que se puede llevar a cabo de una de dos formas generales:

- Cuando un nodo se une a una red, registra su servicio ante un servicio de búsqueda centralizado existente en la red. Cualquier nodo que desee un servicio concreto, contacta primero con ese servicio de búsqueda centralizado para determinar qué nodo suministra el servicio deseado. El resto de la comunicación tiene lugar entre el cliente y el proveedor del servicio.
- Un nodo que actúe como cliente primero debe descubrir qué nodo proporciona el servicio deseado, mediante la multidifusión de una solicitud de servicio a todos los restantes nodos de la red. El nodo (o nodos) que proporcionen dicho servicio responden al nodo que efectúa la solicitud. Para soportar este método, debe proporcionarse un *protocolo de descubrimiento* que permita a los nodos descubrir los servicios proporcionados por los demás nodos de la red.

Las redes entre iguales han ganado popularidad al final de los años 90, con varios servicios de compartición de archivos como Napster y Gnutella, que permiten a una serie de nodos intercambiar archivos entre sí. El sistema Napster usa un método similar al primer tipo descrito anteriormente: un servidor centralizado mantiene un índice de todos los archivos almacenados en los nodos de la red Napster y el intercambio real de archivos tiene lugar entre esos nodos. El sistema Gnutella emplea una técnica similar a la del segundo tipo: cada cliente difunde las solicitudes de archivos a los demás nodos del sistema y los nodos que pueden servir la solicitud responden directamente al cliente. El futuro del intercambio de archivos permanece incierto, ya que muchos de los archivos tienen derechos de propiedad intelectual (los archivos de música, por ejemplo) y hay leyes que legislan la distribución de este tipo de material. En cualquier caso, la tecnología entre iguales desempeñará sin duda un papel en el futuro de muchos servicios, como los mecanismos de búsqueda, el intercambio de archivos y el correo electrónico.

1.12.4 Sistema basado en la Web

La Web está empezando a resultar omnipresente, proporcionando un mayor acceso mediante una más amplia variedad de dispositivos de lo que hubiéramos podido soñar hace unos pocos años. Los PC todavía son los dispositivos de acceso predominantes, aunque las estaciones de trabajo, los PDA de mano e incluso los teléfonos móviles se emplean ahora también para proporcionar acceso a Internet.

La informática basada en la Web ha incrementado el interés por los sistemas de interconexión por red. Dispositivos que anteriormente no estaban conectados en red ahora incluyen acceso p

cable o inalámbrico. Los dispositivos que sí estaban conectados en red ahora disponen de una conectividad de red más rápida, gracias a las mejoras en la tecnología de redes, a la optimización del código de implementación de red o a ambas cosas.

La implementación de sistemas basados en la Web ha hecho surgir nuevas categorías de dispositivos, tales como los **mecanismos de equilibrado de carga**, que distribuyen las conexiones de red entre una batería de servidores similares. Sistemas operativos como Windows 95, que actuaba como cliente web, han evolucionado a Linux o Windows XP, que pueden actuar como servidores web y como clientes. En general, la Web ha incrementado la complejidad de muchos dispositivos, ya que los usuarios de los mismos exigen poder conectarlos a la Web.

1.13 Resumen

Un sistema operativo es un software que gestiona el hardware de la computadora y proporciona un entorno para ejecutar los programas de aplicación. Quizá el aspecto más visible de un sistema operativo sea la interfaz que el sistema informático proporciona al usuario.

Para que una computadora haga su trabajo de ejecutar programas, los programas deben encontrarse en la memoria principal. La memoria principal es la única área de almacenamiento de gran tamaño a la que el procesador puede acceder directamente. Es una matriz de palabras o bytes, con un tamaño que va de millones a miles de millones de posiciones distintas. Cada palabra de la memoria tiene su propia dirección. Normalmente, la memoria principal es un dispositivo de almacenamiento volátil que pierde su contenido cuando se desconecta o desaparece la alimentación. La mayoría de los sistemas informáticos proporcionan un almacenamiento secundario como extensión de la memoria principal. El almacenamiento secundario proporciona una forma de almacenamiento no volátil, que es capaz de mantener enormes cantidades de datos de forma permanente. El dispositivo de almacenamiento secundario más común es el disco magnético, que proporciona un sistema de almacenamiento para programas y datos.

La amplia variedad de sistemas de almacenamiento en un sistema informático puede organizarse en una jerarquía, en función de su velocidad y su coste. Los niveles superiores son más caros, pero más rápidos. A medida que se desciende por la jerarquía, el coste por bit generalmente disminuye, mientras que el tiempo de acceso por regla general aumenta.

Existen varias estrategias diferentes para diseñar un sistema informático. Los sistemas monoprocesador sólo disponen de un procesador, mientras que los sistemas multiprocesador tienen dos o más procesadores que comparten la memoria física y los dispositivos periféricos. El diseño multiprocesador más común es el multiprocesamiento simétrico (o SMP), donde todos los procesadores se consideran iguales y operan independientemente unos de otros. Los sistemas conectados en cluster constituyen una forma especializada de sistema multiprocesador y constan de múltiples computadoras conectadas mediante una red de área local.

Para un mejor uso de la CPU, los sistemas operativos modernos emplean multiprogramación, la cual permite tener en memoria a la vez varios trabajos, asegurando por tanto que la CPU tenga siempre un trabajo que ejecutar. Los sistemas de tiempo compartido son una extensión de la multiprogramación, en la que los algoritmos de planificación de la CPU comutan rápidamente entre varios trabajos, proporcionando la ilusión de que cada trabajo está ejecutándose de forma concurrente.

El sistema operativo debe asegurar la correcta operación del sistema informático. Para impedir que los programas de usuario interfieran con el apropiado funcionamiento del sistema, el hardware soporta dos modos de trabajo: modo usuario y modo *kernel*. Diversas instrucciones, como las instrucciones de E/S y las instrucciones de espera, son instrucciones privilegiadas y sólo se pueden ejecutar en el modo *kernel*. La memoria en la que el sistema operativo reside debe protegerse frente a modificaciones por parte del usuario. Un temporizador impide los bucles infinitos. Estas características (modo dual, instrucciones privilegiadas, protección de memoria e interrupciones del temporizador) son los bloques básicos que emplea el sistema operativo para conseguir un correcto funcionamiento.

Un proceso (o trabajo) es la unidad fundamental de trabajo en un sistema operativo. La gestión de procesos incluye la creación y borrado de procesos y proporciona mecanismos para que los

procesos se comuniquen y sincronicen entre sí. Un sistema operativo gestiona la memoria haciendo un seguimiento de qué partes de la misma están siendo usadas y por quién. El sistema operativo también es responsable de la asignación dinámica y liberación del espacio de memoria. El sistema operativo también gestiona el espacio de almacenamiento, lo que incluye proporcionar sistemas de archivos para representar archivos y directorios y gestionar el espacio en los dispositivos de almacenamiento masivo.

Los sistemas operativos también deben ocuparse de la protección y seguridad del propio sistema operativo y de los usuarios. El concepto de protección incluye los mecanismos que controlan el acceso de los procesos o usuarios a los recursos que el sistema informático pone a su disposición. Las medidas de seguridad son responsables de defender al sistema informático de los ataques externos e internos.

Los sistemas distribuidos permiten a los usuarios compartir los recursos disponibles en una serie de hosts dispersos geográficamente, conectados a través de una red de computadoras. Los servicios pueden ser proporcionados según el modelo cliente-servidor o el modelo entre iguales. En un sistema en cluster, las múltiples máquinas pueden realizar cálculos sobre los datos que residen en sistemas de almacenamiento compartidos y los cálculos pueden continuar incluso cuando algún subconjunto de los miembros del cluster falle.

Las LAN y WAN son los dos tipos básicos de redes. Las redes LAN permiten que un conjunto de procesadores distribuidos en un área geográfica pequeña se comuniquen, mientras que las WAN permiten que se comuniquen diversos procesadores distribuidos en un área más grande. Las redes LAN son típicamente más rápidas que las WAN.

Existen diversos tipos de sistemas informáticos que sirven a propósitos específicos. Entre estos se incluyen los sistemas operativos en tiempo real diseñados para entornos embebidos, tales como los dispositivos de consumo, automóviles y equipos robóticos. Los sistemas operativos en tiempo real tienen restricciones de tiempo fijas y bien definidas. El procesamiento *tiene que* realizarse dentro de las restricciones definidas, o el sistema fallará. Los sistemas multimedia implican el suministro de datos multimedia y, a menudo, tienen requisitos especiales para visualizar o reproducir audio, vídeo o flujos sincronizados de audio y vídeo.

Recientemente, la influencia de Internet y la World Wide Web ha llevado al desarrollo de sistemas operativos modernos que integran exploradores web y software de red y comunicaciones.

Ejercicios

- 1.1 En un entorno de multiprogramación y tiempo compartido, varios usuarios comparten el sistema simultáneamente. Esta situación puede dar lugar a varios problemas de seguridad.
 - a. ¿Cuáles son dos de dichos problemas?
 - b. ¿Podemos asegurar el mismo grado de seguridad en un sistema de tiempo compartido que en un sistema dedicado? Explique su respuesta.
- 1.2 El problema de la utilización de recursos se manifiesta de diferentes maneras en los diferentes tipos de sistema operativo. Enumere qué recursos deben gestionarse de forma especial en las siguientes configuraciones:
 - a. Sistemas *mainframe* y minicomputadoras
 - b. Estaciones de trabajo conectadas a servidores
 - c. Computadoras de mano
- 1.3 ¿Bajo qué circunstancias sería mejor para un usuario utilizar un sistema de tiempo compartido en lugar de un PC o una estación de trabajo monousuario?
- 1.4 ¿A cuál de las funcionalidades que se enumeran a continuación tiene que dar soporte un sistema operativo, en las dos configuraciones siguientes: (a) una computadora de mano y (b) un sistema en tiempo real?
 - a. Programación por lotes

- b. Memoria virtual
 - c. Tiempo compartido
- 1.5 Describa las diferencias entre multiprocesamiento simétrico y asimétrico. Indique tres ventajas y una desventaja de los sistemas con múltiples procesadores.
- 1.6 ¿En qué se diferencian los sistemas en cluster de los sistemas multiprocesador? ¿Qué se requiere para que dos máquinas que pertenecen a un cluster cooperen para proporcionar un servicio de muy alta disponibilidad?
- 1.7 Indique las diferencias entre los sistemas distribuidos basados en los modelos cliente -servidor y entre iguales.
- 1.8 Considere un sistema en cluster que consta de dos nodos que ejecutan una base de datos. Describa dos formas en las que el software del cluster puede gestionar el acceso a los datos almacenados en el disco. Explique las ventajas y desventajas de cada forma.
- 1.9 ¿En qué se diferencian las computadoras de red de las computadoras personales tradicionales? Describa algunos escenarios de uso en los que sea ventajoso el uso de computadoras de red.
- 1.10 ¿Cuál es el propósito de las interrupciones? ¿Cuáles son las diferencias entre una excepción y una interrupción? ¿Pueden generarse excepciones intencionadamente mediante un programa de usuario? En caso afirmativo, ¿con qué propósito?
- 1.11 El acceso directo a memoria se usa en dispositivos de E/S de alta velocidad para evitar aumentar la carga de procesamiento de la CPU.
 - a. ¿Cómo interactúa la CPU con el dispositivo para coordinar la transferencia?
 - b. ¿Cómo sabe la CPU que las operaciones de memoria se han completado?
 - c. La CPU puede ejecutar otros programas mientras la controladora de DMA está transfiriendo datos. ¿Interfiere este proceso con la ejecución de los programas de usuario? En caso afirmativo, describa las formas de interferencia que se puedan producir.
- 1.12 Algunos sistemas informáticos no proporcionan un modo privilegiado de operación en su hardware. ¿Es posible construir un sistema operativo seguro para estos sistemas informáticos? Justifique su respuesta.
- 1.13 Proporcione dos razones por las que las cachés son útiles. ¿Qué problemas resuelven? ¿Qué problemas causan? Si una caché puede ser tan grande como el dispositivo para el que se utiliza (por ejemplo, una caché tan grande como un disco) ¿por qué no hacerla así de grande y eliminar el dispositivo?
- 1.14 Explique, con ejemplos, cómo se manifiesta el problema de mantener la coherencia de los datos en caché en los siguientes entornos de procesamiento:
 - a. Sistemas de un solo procesador
 - b. Sistemas multiprocesador
 - c. Sistemas distribuidos
- 1.15 Describa un mecanismo de protección de memoria que evite que un programa modifique la memoria asociada con otros programas.
- 1.16 ¿Qué configuración de red se adapta mejor a los entornos siguientes?
 - a. Un piso en una ciudad dormitorio
 - b. Un campus universitario
 - c. Una región
 - d. Una nación

1.17 Defina las propiedades esenciales de los siguientes tipos de sistemas operativos:

- a. Procesamiento por lotes
- b. Interactivo
- c. Tiempo compartido
- d. Tiempo real
- e. Red
- f. Paralelo
- g. Distribuido
- h. En cluster
- i. De mano

1.18 ¿Cuáles son las deficiencias inherentes de las computadoras de mano?

Notas bibliográficas

Brookshear [2003] proporciona una introducción general a la Informática.

Puede encontrar una introducción al sistema operativo Linux en Bovet y Cesati [2002]. Solomon y Russinovich [2000] proporcionan una introducción a Microsoft Windows y considerables detalles técnicos sobre los componentes e interioridades del sistema. Mauro y McDougall [2001] cubren el sistema operativo Solaris. Puede encontrar detalles sobre Mac OS X en <http://www.apple.com/macosx>.

Los sistemas entre iguales se cubren en Parameswaran et al. [2001], Gong [2002], Ripeanu et al. [2002], Agre[2003], Balakrishnan et al. [2003] y Loo [2003]. Para ver detalles sobre los sistemas de partición de archivos en las redes entre iguales, consulte Lee [2003]. En Buyya [1999] podrá encontrar una buena exposición sobre los sistemas en cluster. Los avances recientes sobre los sistemas en cluster se describen en Ahmed [2000]. Un buen tratado sobre los problemas relacionados con el soporte de sistemas operativos para sistemas distribuidos es el que puede encontrarse en Tanenbaum y Van Renesse[1985].

Hay muchos libros de texto generales que cubren los sistemas operativos, incluyendo Stallings [2000b], Nutt [2004] y Tanenbaum [2001].

Hamacher [2002] describe la organización de las computadoras. Hennessy y Paterson [2002] cubren los sistemas de E/S y buses, y la arquitectura de sistemas en general.

Las memorias caché, incluyendo las memorias asociativas, se describen y analizan en Smith [1982]. Dicho documento también incluye una extensa bibliografía sobre el tema.

Podrá encontrar una exposición sobre la tecnología de discos magnéticos en Freedman [1983] y Harker et al. [1981]. Los discos ópticos se cubren en Kenville [1982], Fujitani [1984], O'Leary y Kitts [1985], Gait [1988] y Olsen y Kenley [1989]. Para ver información sobre disquetes, puede consultar Pechura y Schoeffler [1983] y Sarisky [1983]. Chi [1982] y Hoagland [1985] ofrecen explicaciones generales sobre la tecnología de almacenamiento masivo.

Kurose y Rose [2005], Tanenbaum[2003], Peterson y Davie [1996] y Halsall [1992] proporcionan una introducción general a las redes de computadoras. Fortier [1989] presenta una exposición detallada del hardware y software de red.

Wolf[2003] expone los desarrollos recientes en el campo de los sistemas embebidos. Puede encontrar temas relacionados con los dispositivos de mano en Myers y Beig [2003] y Di Pietro y Mancini [2003].

Estructuras de sistemas operativos

Un sistema operativo proporciona el entorno en el que se ejecutan los programas. Internamente, los sistemas operativos varían mucho en su composición, dado que su organización puede analizarse aplicando múltiples criterios diferentes. El diseño de un nuevo sistema operativo es una tarea de gran envergadura, siendo fundamental que los objetivos del sistema estén bien definidos antes de comenzar el diseño. Estos objetivos establecen las bases para elegir entre diversos algoritmos y estrategias.

Podemos ver un sistema operativo desde varios puntos de vista. Uno de ellos se centra en los servicios que el sistema proporciona; otro, en la interfaz disponible para los usuarios y programadores; un tercero, en sus componentes y sus interconexiones. En este capítulo, exploraremos estos tres aspectos de los sistemas operativos, considerando los puntos de vista de los usuarios, programadores y diseñadores de sistemas operativos. Tendremos en cuenta qué servicios proporciona un sistema operativo, cómo los proporciona y las diferentes metodologías para diseñar tales sistemas. Por último, describiremos cómo se crean los sistemas operativos y cómo puede una computadora iniciar su sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Describir los servicios que un sistema operativo proporciona a los usuarios, a los procesos y a otros sistemas.
- Exponer las diversas formas de estructurar un sistema operativo.
- Explicar cómo se instalan, personalizan y arrancan los sistemas operativos.

2.1 Servicios del sistema operativo

Un sistema operativo proporciona un entorno para la ejecución de programas. El sistema presta ciertos servicios a los programas y a los usuarios de dichos programas. Por supuesto, los servicios específicos que se suministran difieren de un sistema operativo a otro, pero podemos identificar perfectamente una serie de clases comunes. Estos servicios del sistema operativo se proporcionan para comodidad del programador, con el fin de facilitar la tarea de desarrollo.

Un cierto conjunto de servicios del sistema operativo proporciona funciones que resultan útiles al usuario:

- **Interfaz de usuario.** Casi todos los sistemas operativos disponen de una **interfaz de usuario** (UI, user interface), que puede tomar diferentes formas. Uno de los tipos existentes es la **interfaz de línea de comandos** (CLI, command-line interface), que usa comandos de textos y algún tipo de método para introducirlos, es decir, un programa que permite introducir y editar los comandos. Otro tipo destacable es la **interfaz de proceso por lotes**, en la que los

comandos y las directivas para controlar dichos comandos se introducen en archivos, y luego dichos archivos se ejecutan. Habitualmente, se utiliza una **interfaz gráfica de usuario** (GUI, graphical user interface); en este caso, la interfaz es un sistema de ventanas, con un dispositivo señalador para dirigir la E/S, para elegir opciones en los menús y para realizar otras selecciones, y con un teclado para introducir texto. Algunos sistemas proporcionan dos o tres de estas variantes.

- **Ejecución de programas.** El sistema tiene que poder cargar un programa en memoria y ejecutar dicho programa. Todo programa debe poder terminar su ejecución, de forma normal o anormal (indicando un error).
- **Operaciones de E/S.** Un programa en ejecución puede necesitar llevar a cabo operaciones de E/S, dirigidas a un archivo o a un dispositivo de E/S. Para ciertos dispositivos específicos, puede ser deseable disponer de funciones especiales, tales como grabar en una unidad de CD o DVD o borrar una pantalla de TRC (tubo de rayos catódicos). Por cuestiones de eficiencia y protección, los usuarios no pueden, normalmente, controlar de modo directo los dispositivos de E/S; por tanto, el sistema operativo debe proporcionar medios para realizar la E/S.
- **Manipulación del sistema de archivos.** El sistema de archivos tiene una importancia especial. Obviamente, los programas necesitan leer y escribir en archivos y directorios. También necesitan crearlos y borrarlos usando su nombre, realizar búsquedas en un determinado archivo o presentar la información contenida en un archivo. Por último, algunos programas incluyen mecanismos de gestión de permisos para conceder o denegar el acceso a los archivos o directorios basándose en quién sea el propietario del archivo.
- **Comunicaciones.** Hay muchas circunstancias en las que un proceso necesita intercambiar información con otro. Dicha comunicación puede tener lugar entre procesos que estén ejecutándose en la misma computadora o entre procesos que se ejecuten en computadoras diferentes conectadas a través de una red. Las comunicaciones se pueden implementar utilizando *memoria compartida* o mediante *paso de mensajes*, procedimiento éste en el que el sistema operativo transfiere paquetes de información entre unos procesos y otros.
- **Detección de errores.** El sistema operativo necesita detectar constantemente los posibles errores. Estos errores pueden producirse en el hardware del procesador y de memoria (como, por ejemplo, un error de memoria o un fallo de la alimentación) en un dispositivo de E/S (como un error de paridad en una cinta, un fallo de conexión en una red o un problema de falta papel en la impresora) o en los programas de usuario (como, por ejemplo, un desbordamiento aritmético, un intento de acceso a una posición de memoria ilegal o un uso excesivo del tiempo de CPU). Para cada tipo de error, el sistema operativo debe llevar a cabo la acción apropiada para asegurar un funcionamiento correcto y coherente. Las facilidades de depuración pueden mejorar en gran medida la capacidad de los usuarios y programadores para utilizar el sistema de manera eficiente.

Hay disponible otro conjunto de funciones del sistema de operativo que no están pensadas para ayudar al usuario, sino para garantizar la eficiencia del propio sistema. Los sistemas con múltiples usuarios pueden ser más eficientes cuando se comparten los recursos del equipo entre los distintos usuarios:

- **Asignación de recursos.** Cuando hay varios usuarios, o hay varios trabajos ejecutándose al mismo tiempo, deben asignarse a cada uno de ellos los recursos necesarios. El sistema operativo gestiona muchos tipos diferentes de recursos; algunos (como los ciclos de CPU, la memoria principal y el espacio de almacenamiento de archivos) pueden disponer de código software especial que gestione su asignación, mientras que otros (como los dispositivos de E/S) pueden tener código que gestione de forma mucho más general su solicitud y liberación. Por ejemplo, para poder determinar cuál es el mejor modo de usar la CPU, el sistema operativo dispone de rutinas de planificación de la CPU que tienen en cuenta la velocidad del procesador, los trabajos que tienen que ejecutarse, el número de registros disponi-

bles y otros factores. También puede haber rutinas para asignar impresoras, modems, unidades de almacenamiento USB y otros dispositivos periféricos.

- **Responsabilidad.** Normalmente conviene hacer un seguimiento de qué usuarios emplean qué clase de recursos de la computadora y en qué cantidad. Este registro se puede usar para propósitos contables (con el fin de poder facturar el gasto correspondiente a los usuarios) o simplemente para acumular estadísticas de uso. Estas estadísticas pueden ser una herramienta valiosa para aquellos investigadores que deseen reconfigurar el sistema con el fin de mejorar los servicios informáticos.
- **Protección y seguridad.** Los propietarios de la información almacenada en un sistema de computadoras en red o multiusuario necesitan a menudo poder controlar el uso de dicha información. Cuando se ejecutan de forma concurrente varios procesos distintos, no debe ser posible que un proceso interfiera con los demás procesos o con el propio sistema operativo. La protección implica asegurar que todos los accesos a los recursos del sistema estén controlados. También es importante garantizar la seguridad del sistema frente a posibles intrusos; dicha seguridad comienza por requerir que cada usuario se autentique ante el sistema, usualmente mediante una contraseña, para obtener acceso a los recursos del mismo. Esto se extiende a la defensa de los dispositivos de E/S, entre los que se incluyen modems y adaptadores de red, frente a intentos de acceso ilegales y el registro de dichas conexiones con el fin de detectar intrusiones. Si hay que proteger y asegurar un sistema, las protecciones deben implementarse en todas partes del mismo: una cadena es tan fuerte como su eslabón más débil.

2.2 Interfaz de usuario del sistema operativo

Existen dos métodos fundamentales para que los usuarios interactúen con el sistema operativo. Una técnica consiste en proporcionar una interfaz de línea de comandos o **intérprete de comandos**, que permita a los usuarios introducir directamente comandos que el sistema operativo pueda ejecutar. El segundo método permite que el usuario interactúe con el sistema operativo a través de una interfaz gráfica de usuario o GUI.

2.2.1 Intérprete de comandos

Algunos sistemas operativos incluyen el intérprete de comandos en el *kernel*. Otros, como Windows XP y UNIX, tratan al intérprete de comandos como un programa especial que se ejecuta cuando se inicia un trabajo o cuando un usuario inicia una sesión (en los sistemas interactivos). En los sistemas que disponen de varios intérpretes de comandos entre los que elegir, los intérpretes se conocen como **shells**. Por ejemplo, en los sistemas UNIX y Linux, hay disponibles varias shells diferentes entre las que un usuario puede elegir, incluyendo la *shell Bourne*, la *shell C*, la *shell Bourne-Again*, la *shell Korn*, etc. La mayoría de las *shells* proporcionan funcionalidades similares, existiendo sólo algunas diferencias menores, casi todos los usuarios seleccionan una *shell* u otra basándose en sus preferencias personales.

La función principal del intérprete de comandos es obtener y ejecutar el siguiente comando especificado por el usuario. Muchos de los comandos que se proporcionan en este nivel se utilizan para manipular archivos: creación, borrado, listado, impresión, copia, ejecución, etc.; las *shells* de MS-DOS y UNIX operan de esta forma. Estos comandos pueden implementarse de dos formas generales.

Uno de los métodos consiste en que el propio intérprete de comandos contiene el código que el comando tiene que ejecutar. Por ejemplo, un comando para borrar un archivo puede hacer que el intérprete de comandos salte a una sección de su código que configura los parámetros necesarios y realiza las apropiadas llamadas al sistema. En este caso, el número de comandos que puede proporcionarse determina el tamaño del intérprete de comandos, dado que cada comando requiere su propio código de implementación.

Un método alternativo, utilizado por UNIX y otros sistemas operativos, implementa la mayoría de los comandos a través de una serie de programas del sistema. En este caso, el intérprete de comandos no “entiende” el comando, sino que simplemente lo usa para identificar el archivo que hay que cargar en memoria y ejecutar. Por tanto, el comando UNIX para borrar un archivo

```
rm file.txt
```

buscaría un archivo llamado `rm`, cargaría el archivo en memoria y lo ejecutaría, pasándole el parámetro `file.txt`. La función asociada con el comando `rm` queda definida completamente mediante el código contenido en el archivo `rm`. De esta forma, los programadores pueden añadir comandos al sistema fácilmente, creando nuevos archivos con los nombres apropiados. El programa intérprete de comandos, que puede ser pequeño, no tiene que modificarse en función de los nuevos comandos que se añadan.

2.2.2 Interfaces gráficas de usuario

Una segunda estrategia para interactuar con el sistema operativo es a través de una interfaz gráfica de usuario (GUI) suficientemente amigable. En lugar de tener que introducir comandos directamente a través de la línea de comandos, una GUI permite a los usuarios emplear un sistema de ventanas y menús controlable mediante el ratón. Una GUI proporciona una especie de **escritorio** en el que el usuario mueve el ratón para colocar su puntero sobre imágenes, o **iconos**, que se muestran en la pantalla (el escritorio) y que representan programas, archivos, directorios y funciones del sistema. Dependiendo de la ubicación del puntero, pulsar el botón del ratón puede invocar un programa, seleccionar un archivo o directorio (conocido como **carpeta**) o desplegar un menú que contenga comandos ejecutables.

Las primeras interfaces gráficas de usuario aparecieron debido, en parte, a las investigaciones realizadas en el departamento de investigación de Xerox Parc a principios de los años 70. La primera GUI se incorporó en la computadora Xerox Alto en 1973. Sin embargo, las interfaces gráficas sólo se popularizaron con la introducción de las computadoras Apple Macintosh en los años 80. La interfaz de usuario del sistema operativo de Macintosh (Mac OS) ha sufrido diversos cambios a lo largo de los años, siendo el más significativo la adopción de la interfaz *Aqua* en Mac OS X. La primera versión de Microsoft Windows (versión 1.0) estaba basada en una interfaz GUI que permitía interactuar con el sistema operativo MS-DOS. Las distintas versiones de los sistemas Windows proceden de esa versión inicial, a la que se le han aplicado cambios cosméticos en cuanto a su apariencia y diversas mejoras de funcionalidad, incluyendo el Explorador de Windows.

Tradicionalmente, en los sistemas UNIX han predominado las interfaces de línea de comandos, aunque hay disponibles varias interfaces GUI, incluyendo el entorno de escritorio CDE (Common Desktop Environment) y los sistemas X-Windows, que son muy habituales en las versiones comerciales de UNIX, como Solaris o el sistema AIX de IBM. Sin embargo, también es necesario resaltar el desarrollo de diversas interfaces de tipo GUI en diferentes proyectos de **código fuente abierto**, como por ejemplo el entorno de escritorio KDE (K Desktop Environment) y el entorno GNOME, que forma parte del proyecto GNU. Ambos entornos de escritorio, KDE y GNOME, se ejecutan sobre Linux y otros varios sistemas UNIX, y están disponibles con licencias de código fuente abierto, lo que quiere decir que su código fuente es del dominio público.

La decisión de usar una interfaz de línea de comandos o GUI es, principalmente, una opción personal. Por regla general, muchos usuarios de UNIX prefieren una interfaz de línea de comandos, ya que a menudo les proporciona interfaces de tipo *shell* más potentes. Por otro lado, la mayor parte de los usuarios de Windows se decantan por el uso del entorno GUI de Windows y casi nunca emplean la interfaz de tipo *shell* MS-DOS. Por el contrario, los diversos cambios experimentados por los sistemas operativos de Macintosh proporcionan un interesante caso de estudio: históricamente, Mac OS no proporcionaba una interfaz de línea de comandos, siendo obligatorio que los usuarios interactuaran con el sistema operativo a través de la interfaz GUI; sin embargo, con el lanzamiento de Mac OS X (que está parcialmente basado en un *kernel* UNIX), el sistema operativo proporciona ahora tanto la nueva interfaz *Aqua*, como una interfaz de línea de comandos.

La interfaz de usuario puede variar de un sistema a otro e incluso de un usuario a otro dentro de un sistema; por esto, la interfaz de usuario se suele, normalmente, eliminar de la propia estruc-

tura del sistema. El diseño de una interfaz de usuario útil y amigable no es, por tanto, una función directa del sistema operativo. En este libro, vamos a concentrarnos en los problemas fundamentales de proporcionar un servicio adecuado a los programas de usuario. Desde el punto de vista del sistema operativo, no diferenciaremos entre programas de usuario y programas del sistema.

2.3 Llamadas al sistema

Las **Llamadas al sistema** proporcionan una interfaz con la que poder invocar los servicios que el sistema operativo ofrece. Estas llamadas, generalmente, están disponibles como rutinas escritas en C y C++, aunque determinadas tareas de bajo nivel, como por ejemplo aquéllas en las que se tiene que acceder directamente al hardware, pueden necesitar escribirse con instrucciones de lenguaje ensamblador.

Antes de ver cómo pone un sistema operativo a nuestra disposición las llamadas al sistema, vamos a ver un ejemplo para ilustrar cómo se usan esas llamadas: suponga que deseamos escribir un programa sencillo para leer datos de un archivo y copiarlos en otro archivo. El primer dato de entrada que el programa va a necesitar son los nombres de los dos archivos, el de entrada y el de salida. Estos nombres pueden especificarse de muchas maneras, dependiendo del diseño del sistema operativo; un método consiste en que el programa pida al usuario que introduzca los nombres de los dos archivos. En un sistema interactivo, este método requerirá una secuencia de llamadas al sistema: primero hay que escribir un mensaje en el indicativo de comandos de la pantalla y luego leer del teclado los caracteres que especifican los dos archivos. En los sistemas basados en iconos y en el uso de un ratón, habitualmente se suele presentar un menú de nombres de archivo en una ventana. El usuario puede entonces usar el ratón para seleccionar el nombre del archivo de origen y puede abrirse otra ventana para especificar el nombre del archivo de destino. Esta secuencia requiere realizar numerosas llamadas al sistema de E/S.

Una vez que se han obtenido los nombres de los dos archivos, el programa debe abrir el archivo de entrada y crear el archivo de salida. Cada una de estas operaciones requiere otra llamada al sistema. Asimismo, para cada operación, existen posibles condiciones de error. Cuando el programa intenta abrir el archivo de entrada, puede encontrarse con que no existe ningún archivo con ese nombre o que está protegido contra accesos. En estos casos, el programa debe escribir un mensaje en la consola (otra secuencia de llamadas al sistema) y terminar de forma anormal (otra llamada al sistema). Si el archivo de entrada existe, entonces se debe crear un nuevo archivo de salida. En este caso, podemos encontrarnos con que ya existe un archivo de salida con el mismo nombre; esta situación puede hacer que el programa termine (una llamada al sistema) o podemos borrar el archivo existente (otra llamada al sistema) y crear otro (otra llamada más). En un sistema interactivo, otra posibilidad sería preguntar al usuario (a través de una secuencia de llamadas al sistema para mostrar mensajes en el indicativo de comandos y leer las respuestas desde el terminal) si desea reemplazar el archivo existente o terminar el programa.

Una vez que ambos archivos están definidos, hay que ejecutar un bucle que lea del archivo de entrada (una llamada al sistema) y escriba en el archivo de salida (otra llamada al sistema). Cada lectura y escritura debe devolver información de estado relativa a las distintas condiciones posibles de error. En la entrada, el programa puede encontrarse con que ha llegado al final del archivo o con que se ha producido un fallo de hardware en la lectura (como por ejemplo, un error de paridad). En la operación de escritura pueden producirse varios errores, dependiendo del dispositivo de salida (espacio de disco insuficiente, la impresora no tiene papel, etc.).

Finalmente, después de que se ha copiado el archivo completo, el programa cierra ambos archivos (otra llamada al sistema), escribe un mensaje en la consola o ventana (más llamadas al sistema) y, por último, termina normalmente (la última llamada al sistema). Como puede ver, incluso los programas más sencillos pueden hacer un uso intensivo del sistema operativo. Frecuentemente, los sistemas ejecutan miles de llamadas al sistema por segundo. Esta secuencia de llamadas al sistema se muestra en la Figura 2.1.

Sin embargo, la mayoría de los programadores no ven este nivel de detalle. Normalmente, los desarrolladores de aplicaciones diseñan sus programas utilizando una **API** (application program-

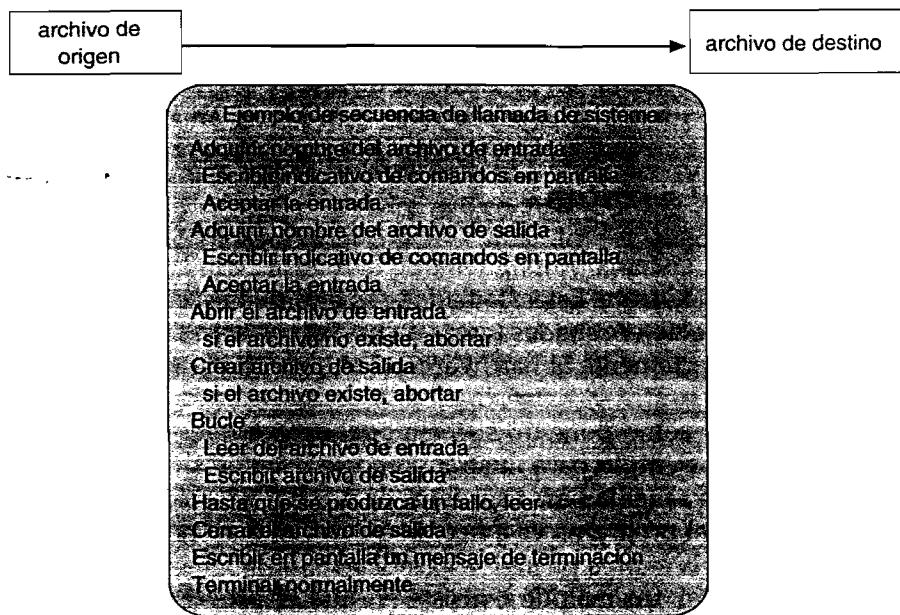


Figura 2.1 Ejemplo de utilización de las llamadas al sistema.

ming interface, interfaz de programación de aplicaciones). La API especifica un conjunto de funciones que el programador de aplicaciones puede usar, indicándose los parámetros que hay que pasar a cada función y los valores de retorno que el programador debe esperar. Tres de las API disponibles para programadores de aplicaciones son la API Win32 para sistemas Windows, la API POSIX para sistemas basados en POSIX (que incluye prácticamente todas las versiones de UNIX, Linux y Mac OS X) y la API Java para diseñar programas que se ejecuten sobre una máquina virtual de Java.

Observe que los nombres de las llamadas al sistema que usamos a lo largo del texto son ejemplos genéricos. Cada sistema operativo tiene sus propios nombres de llamadas al sistema.

Entre bastidores, las funciones que conforman una API invocan, habitualmente, a las llamadas al sistema por cuenta del programador de la aplicación. Por ejemplo, la función `CreateProcess()` de Win32 (que, como su propio nombre indica, se emplea para crear un nuevo proceso) lo que hace, realmente, es invocar la llamada al sistema `NTCreateProcess()` del *kernel* de Windows. ¿Por qué un programador de aplicaciones preferiría usar una API en lugar de invocar las propias llamadas al sistema? Existen varias razones para que sea así. Una ventaja de programar usando una API está relacionada con la portabilidad: un programador de aplicaciones diseña un programa usando una API cuando quiere poder compilar y ejecutar su programa en cualquier sistema que soporte la misma API (aunque, en la práctica, a menudo existen diferencias de arquitectura que hacen que esto sea más difícil de lo que parece). Además, a menudo resulta más difícil trabajar con las propias llamadas al sistema y exige un grado mayor de detalle que usar las API que los programadores de aplicaciones tienen a su disposición. De todos modos, existe una fuerte correlación entre invocar una función de la API y su llamada al sistema asociada disponible en el *kernel*. De hecho, muchas de las API Win32 y POSIX son similares a las llamadas al sistema nativas proporcionadas por los sistemas operativos UNIX, Linux y Windows.

El sistema de soporte en tiempo de ejecución (un conjunto de funciones de biblioteca que suele incluirse con los compiladores) de la mayoría de los lenguajes de programación proporciona una **interfaz de llamadas al sistema** que sirve como enlace con las llamadas al sistema disponibles en el sistema operativo. La interfaz de llamadas al sistema intercepta las llamadas a función dentro de las API e invoca la llamada al sistema necesaria. Habitualmente, cada llamada al sistema tiene asociado un número y la interfaz de llamadas al sistema mantiene una tabla indexada según dichos números. Usando esa tabla, la interfaz de llamadas al sistema invoca la llamada necesaria del *kernel* del sistema operativo y devuelve el estado de la ejecución de la llamada al sistema y los posibles valores de retorno.

EJEMPLO DE API ESTÁNDAR

Como ejemplo de API estándar, considere la función `ReadFile()` de la API Win32, una función que lee datos de un archivo. En la Figura 2.2 se muestra la API correspondiente a esta función.

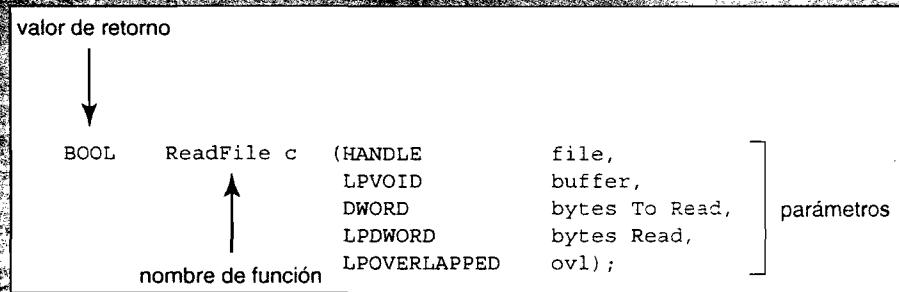


Figura 2.2 API para la función `Readfile()`.

Los parámetros de `Readfile()` son los siguientes:

- `HANDLE file` — archivo que se va a leer.
- `LPVOID buffer` — búfer del que se leerán y en el que se escribirán los datos.
- `DWORD bytes To Read` — número de bytes que se van a leer del búfer.
- `LPDWORD bytes Read` — número de bytes leídos durante la última lectura.
- `LPOVERLAPPED ov1` — indica si se está usando E/S solapada.

Quien realiza la llamada no tiene por qué saber nada acerca de cómo se implementa dicha llamada al sistema o qué es lo que ocurre durante la ejecución. Tan sólo necesita ajustarse a lo que la API especifica y entender lo que hará el sistema operativo como resultado de la ejecución de dicha llamada al sistema. Por tanto, la API oculta al programador la mayor parte de los detalles de la interfaz del sistema operativo, los cuales son gestionados por la biblioteca de soporte en tiempo de ejecución. Las relaciones entre una API, la interfaz de llamadas al sistema y el sistema operativo se muestran en la Figura 2.3, que ilustra cómo gestiona el sistema operativo una aplicación de usuario invocando la llamada al sistema `open()`.

Las llamadas al sistema se llevan a cabo de diferentes formas, dependiendo de la computadora que se utilice. A menudo, se requiere más información que simplemente la identidad de la llamada al sistema deseada. El tipo exacto y la cantidad de información necesaria varían según el sistema operativo y la llamada concreta que se efectúe. Por ejemplo, para obtener un dato de entrada, podemos tener que especificar el archivo o dispositivo que hay que utilizar como origen, así como la dirección y la longitud del búfer de memoria en el que debe almacenarse dicho dato de entrada. Por supuesto, el dispositivo o archivo y la longitud pueden estar implícitos en la llamada.

Para pasar parámetros al sistema operativo se emplean tres métodos generales. El más sencillo de ellos consiste en pasar los parámetros en una serie de *registros*. Sin embargo, en algunos casos, puede haber más parámetros que registros disponibles. En estos casos, generalmente, los parámetros se almacenan en un *bloque* o tabla, en memoria, y la dirección del bloque se pasa como parámetro en un registro (Figura 2.4). Éste es el método que utilizan Linux y Solaris. El programa también puede colocar, o *insertar*, los parámetros en la *pila* y el sistema operativo se encargará de *extraer* de la pila esos parámetros. Algunos sistemas operativos prefieren el método del bloque o el de la pila, porque no limitan el número o la longitud de los parámetros que se quieren pasar.

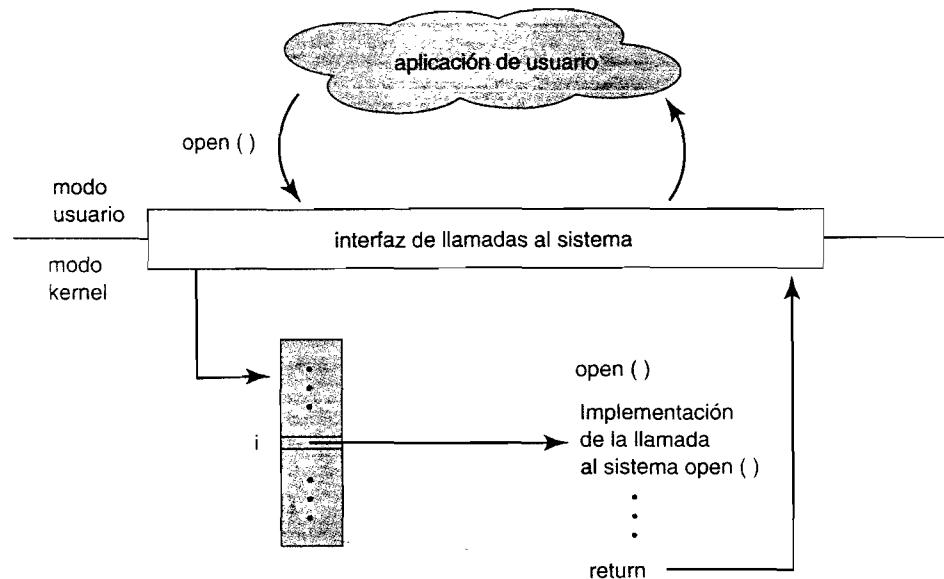


Figura 2.3 Gestión de la invocación de la llamada al sistema `open()` por parte de una aplicación de usuario.

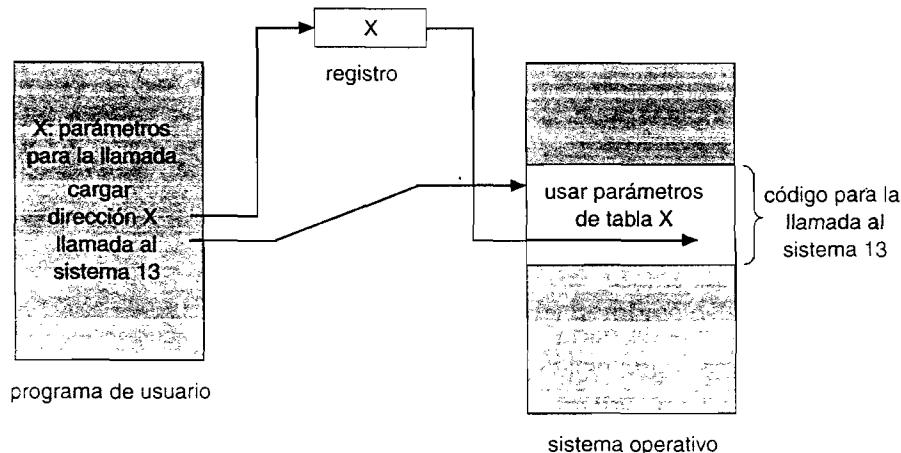


Figura 2.4 Paso de parámetros como tabla.

2.4 Tipos de llamadas al sistema

Las llamadas al sistema pueden agruparse de forma muy general en cinco categorías principales: **control de procesos**, **manipulación de archivos**, **manipulación de dispositivos**, **mantenimiento de información** y **comunicaciones**. En las Secciones 2.4.1 a 2.4.5, expondremos brevemente los tipos de llamadas al sistema que un sistema operativo puede proporcionar. La mayor parte de estas llamadas al sistema soportan, o son soportadas por, conceptos y funciones que se explican en capítulos posteriores. La Figura 2.5 resume los tipos de llamadas al sistema que normalmente proporciona un sistema operativo.

2.4.1 Control de procesos

Un programa en ejecución necesita poder interrumpir dicha ejecución bien de forma normal (`exit`) o de forma anormal (`abort`). Si se hace una llamada al sistema para terminar de forma anormal el programa actualmente en ejecución, o si el programa tiene un problema y da lugar a una excepción de error, en ocasiones se produce un volcado de memoria y se genera un mensaje de erro-

- Control de procesos
 - terminar, abortar
 - cargar, ejecutar
 - crear procesos, terminar procesos
 - obtener atributos del proceso, definir atributos del proceso
 - esperar para obtener tiempo
 - esperar suceso, señalizar suceso
 - asignar y liberar memoria
- Administración de archivos
 - crear archivos, borrar archivos
 - abrir, cerrar
 - leer, escribir, reposicionar
 - obtener atributos de archivo, definir atributos de archivo
- Administración de dispositivos
 - solicitar dispositivo, liberar dispositivo
 - leer, escribir, reposicionar
 - obtener atributos de dispositivo, definir atributos de dispositivo
 - conectar y desconectar dispositivos lógicamente
- Mantenimiento de información
 - obtener la hora o la fecha, definir la hora o la fecha
 - obtener datos del sistema, establecer datos del sistema
 - obtener los atributos de procesos, archivos o dispositivos
 - establecer los atributos de procesos, archivos o dispositivos
- Comunicaciones
 - crear, eliminar conexiones de comunicación
 - enviar, recibir mensajes
 - transferir información de estado
 - conectar y desconectar dispositivos remotos

Figura 2.5 Tipos de llamadas al sistema.

La información de volcado se escribe en disco y un **depurador** (un programa del sistema diseñado para ayudar al programador a encontrar y corregir errores) puede examinar esa información de volcado con el fin de determinar la causa del problema. En cualquier caso, tanto en las circunstancias normales como en las anormales, el sistema operativo debe transferir el control al intérprete de comandos que realizó la invocación del programa; el intérprete leerá entonces el siguiente comando. En un sistema interactivo, el intérprete de comandos simplemente continuará con el siguiente comando, dándose por supuesto que el usuario ejecutará un comando adecuado para responder a cualquier error. En un sistema GUI, una ventana emergente alertará al usuario del error y le pedirá que indique lo que hay que hacer. En un sistema de procesamiento por lotes, el intérprete de comandos usualmente dará por terminado todo el trabajo de procesamiento y con-

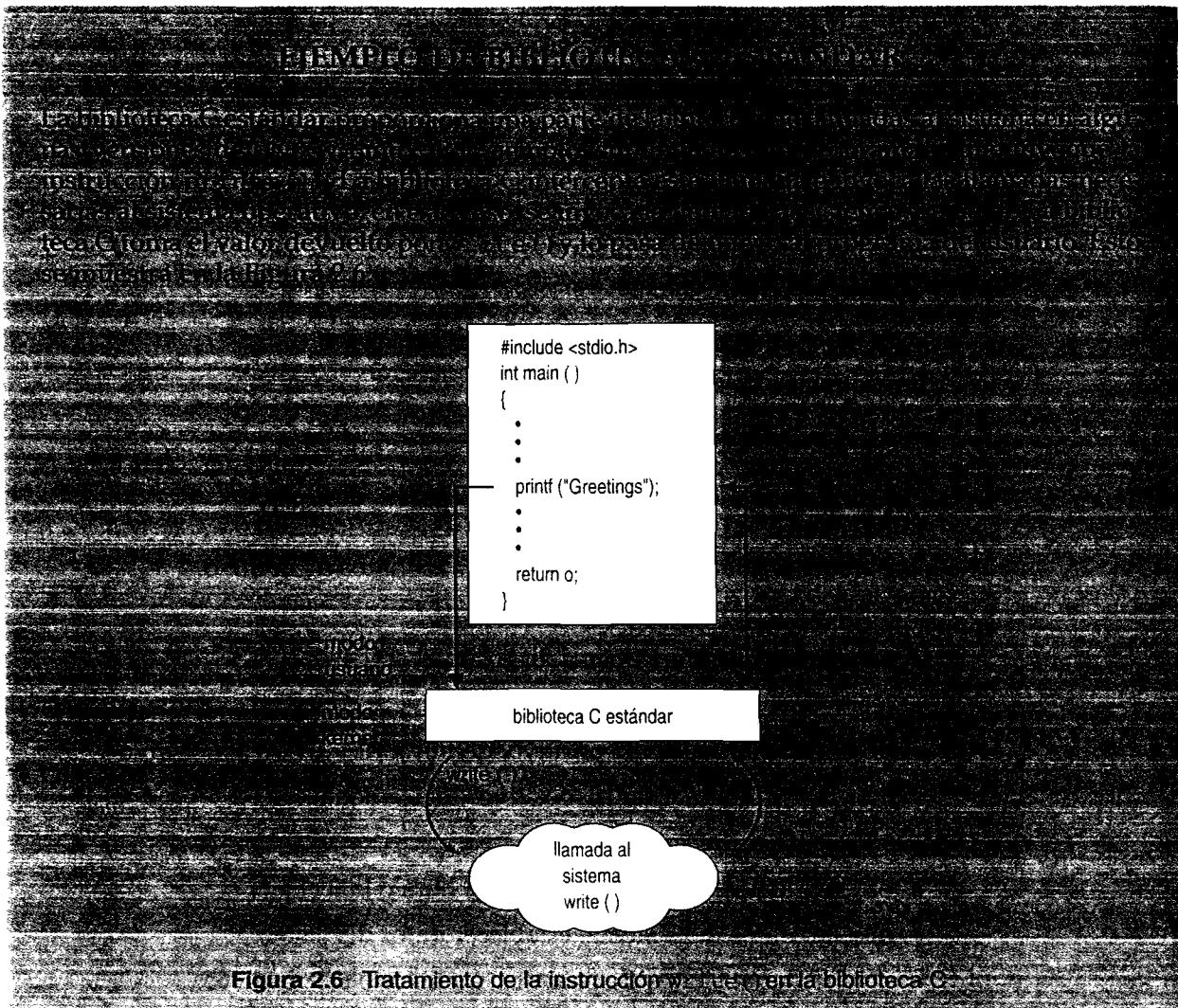


Figura 2.6 Tratamiento de la instrucción write en la biblioteca C

tinuará con el siguiente trabajo. Algunos sistemas permiten utilizar tarjetas de control para indicar acciones especiales de recuperación en caso de que se produzcan errores. Una **tarjeta de control** es un concepto extraído de los sistemas de procesamiento por lotes: se trata de un comando que permite gestionar la ejecución de un proceso. Si el programa descubre un error en sus datos de entrada y decide terminar anormalmente, también puede definir un nivel de error; cuanto más severo sea el error experimentado, mayor nivel tendrá ese parámetro de error. Con este sistema, podemos combinar entonces la terminación normal y la anormal, definiendo una terminación normal como un error de nivel 0. El intérprete de comandos o el siguiente programa pueden usar el nivel de error para determinar automáticamente la siguiente acción que hay que llevar a cabo.

Un proceso o trabajo que ejecute un programa puede querer cargar (load) y ejecutar (execute) otro programa. Esta característica permite al intérprete de comandos ejecutar un programa cuando se le solicite mediante, por ejemplo, un comando de usuario, el clic del ratón o un comando de procesamiento por lotes. Una cuestión interesante es dónde devolver el control una vez que termine el programa invocado. Esta cuestión está relacionada con el problema de si el programa que realiza la invocación se pierde, se guarda o se le permite continuar la ejecución concurrentemente con el nuevo programa.

Si el control vuelve al programa anterior cuando el nuevo programa termina, tenemos que guardar la imagen de memoria del programa existente; de este modo puede crearse un mecanismo para que un programa llame a otro. Si ambos programas continúan concurrentemente, habremos creado un nuevo trabajo o proceso que será necesario controlar mediante los mecanismos de

multiprogramación. Por supuesto, existe una llamada al sistema específica para este propósito, `create process` o `submit job`.

Si creamos un nuevo trabajo o proceso, o incluso un conjunto de trabajos o procesos, también debemos poder controlar su ejecución. Este control requiere la capacidad de determinar y restablecer los atributos de un trabajo o proceso, incluyendo la prioridad del trabajo, el tiempo máximo de ejecución permitido, etc. (`get process attributes` y `set process attributes`). También puede que necesitemos terminar un trabajo o proceso que hayamos creado (`terminate process`) si comprobamos que es incorrecto o decidimos que ya no es necesario.

Una vez creados nuevos trabajos o procesos, es posible que tengamos que esperar a que terminen de ejecutarse. Podemos esperar una determinada cantidad de tiempo (`wait time`) o, más probablemente, esperaremos a que se produzca un suceso específico (`wait event`). Los trabajos o procesos deben indicar cuándo se produce un suceso (`signal event`). En el Capítulo 6 se explican en detalle las llamadas al sistema de este tipo, las cuales se ocupan de la coordinación de procesos concurrentes.

Existe otro conjunto de llamadas al sistema que resulta útil a la hora de depurar un programa. Muchos sistemas proporcionan llamadas al sistema para volcar la memoria (`dump`); esta funcionalidad resulta muy útil en las tareas de depuración. Aunque no todos los sistemas lo permitan, mediante un programa de traza (`trace`) genera una lista de todas las instrucciones según se ejecutan. Incluso hay microprocesadores que proporcionan un modo de la CPU, conocido como *modo paso a paso*, en el que la CPU ejecuta una excepción después de cada instrucción. Normalmente, se usa un depurador para atrapar esa excepción.

Muchos sistemas operativos proporcionan un perfil de tiempo de los programas para indicar la cantidad de tiempo que el programa invierte en una determinada instrucción o conjunto de instrucciones. La generación de perfiles de tiempos requiere disponer de una funcionalidad de traza o de una serie de interrupciones periódicas del temporizador. Cada vez que se produce una interrupción del temporizador, se registra el valor del contador de programa. Con interrupciones del temporizador suficientemente frecuentes, puede obtenerse una imagen estadística del tiempo invertido en las distintas partes del programa.

Son tantas las facetas y variaciones en el control de procesos y trabajos que a continuación vamos a ver dos ejemplos para clarificar estos conceptos; uno de ellos emplea un sistema monotarea y el otro, un sistema multitarea. El sistema operativo MS-DOS es un ejemplo de sistema monotarea. Tiene un intérprete de comandos que se invoca cuando se enciende la computadora [Figura 2.7(a)]. Dado que MS-DOS es un sistema que sólo puede ejecutar una tarea cada vez, utiliza un método muy simple para ejecutar un programa y no crea ningún nuevo proceso: carga el programa en memoria, escribiendo sobre buena parte del propio sistema, con el fin de proporcionar al programa la mayor cantidad posible de memoria [Figura 2.7(b)]. A continuación, establece-

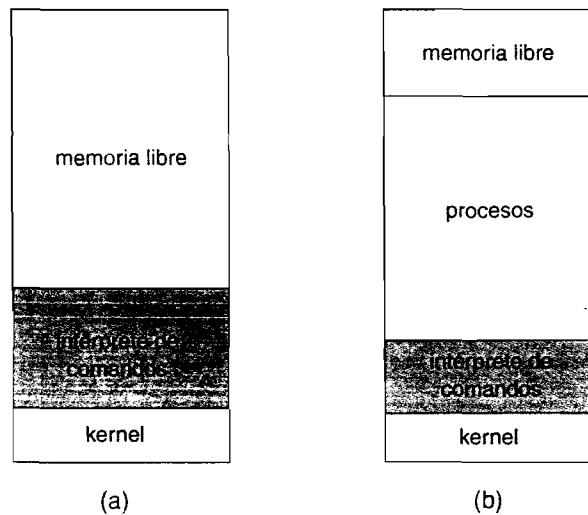


Figura 2.7 Ejecución de un programa en MS-DOS. (a) Al inicio del sistema. (b) Ejecución de un programa.

el puntero de instrucción en la primera instrucción del programa y el programa se ejecuta. Eventualmente, se producirá un error que dé lugar a una excepción o, si no se produce ningún error, el programa ejecutará una llamada al sistema para terminar la ejecución. En ambos casos, el código de error se guarda en la memoria del sistema para su uso posterior. Después de esta secuencia de operaciones, la pequeña parte del intérprete de comandos que no se ha visto sobrescrita reanuda la ejecución. La primera tarea consiste en volver a cargar el resto del intérprete de comandos del disco; luego, el intérprete de comandos pone a disposición del usuario o del siguiente programa el código de error anterior.

FreeBSD (derivado de Berkeley UNIX) es un ejemplo de sistema multitarea. Cuando un usuario inicia la sesión en el sistema, se ejecuta la *shell* elegida por el usuario. Esta *shell* es similar a la *shell* de MS-DOS, en cuanto que acepta comandos y ejecuta los programas que el usuario solicita. Sin embargo, dado que FreeBSD es un sistema multitarea, el intérprete de comandos puede seguir ejecutándose mientras que se ejecuta otro programa (Figura 2.8). Para iniciar un nuevo proceso, la *shell* ejecuta la llamada `fork()` al sistema. Luego, el programa seleccionado se carga en memoria mediante una llamada `exec()` al sistema y el programa se ejecuta. Dependiendo de la forma en que se haya ejecutado el comando, la *shell* espera a que el proceso termine o ejecuta el proceso “en segundo plano”. En este último caso, la *shell* solicita inmediatamente otro comando. Cuando un proceso se ejecuta en segundo plano, no puede recibir entradas directamente desde el teclado, ya que la *shell* está usando ese recurso. Por tanto, la E/S se hace a través de archivos o de una interfaz GUI. Mientras tanto, el usuario es libre de pedir a la *shell* que ejecute otros programas, que monitorice el progreso del proceso en ejecución, que cambie la prioridad de dicho programa, etc. Cuando el proceso concluye, ejecuta una llamada `exit()` al sistema para terminar, devolviendo al proceso que lo invocó un código de estado igual 0 (en caso de ejecución satisfactoria) o un código de error distinto de cero. Este código de estado o código de error queda entonces disponible para la *shell* o para otros programas. En el Capítulo 3 se explican los procesos, con un programa de ejemplo que usa las llamadas al sistema `fork()` y `exec()`.

2.4.2 Administración de archivos

En los Capítulos 10 y 11 analizaremos en detalle el sistema de archivos. De todos modos, podemos aquí identificar diversas llamadas comunes al sistema que están relacionadas con la gestión de archivos.

En primer lugar, necesitamos poder crear (`create`) y borrar (`delete`) archivos. Ambas llamadas al sistema requieren que se proporcione el nombre del archivo y quizás algunos de los atributos del mismo. Una vez que el archivo se ha creado, necesitamos abrirlo (`open`) y utilizarlo. También tenemos que poder leerlo (`read`), escribir (`write`) en él, o reposicionarnos (`reposition`), es decir, volver a un punto anterior o saltar al final del archivo, por ejemplo. Por último, tenemos que poder cerrar (`close`) el archivo, indicando que ya no deseamos usarlo.

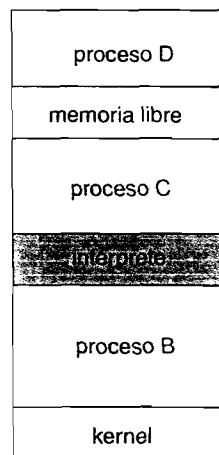


Figura 2.8 FreeBSD ejecutando múltiples programas.

FACILIDAD DE TRAZADO DINÁMICO DE SOLARIS 10.

Hacer que los sistemas operativos sean más fáciles de comprender, de depurar y de optimizar constituye una de las tareas más activas en el campo de la investigación e implementación de sistemas operativos. Por ejemplo, Solaris 10 incluye la funcionalidad de trazado dinámico dtrace. Esta funcionalidad añade dinámicamente una serie de "sondas" al sistema que se esté ejecutando: dichas sondas pueden consultarse mediante el lenguaje de programación D y proporcionan una asombrosa cantidad de información sobre el kernel, el estado del sistema y las actividades de los procesos. Por ejemplo, la Figura 2.9 muestra las actividades de una aplicación cuando se ejecuta una llamada al sistema (ioctl) y muestra también las llamadas funcionales que tienen lugar dentro del kernel para ejecutar la llamada al sistema. Las líneas que terminan en "U" se ejecutan en modo usuario, y las líneas que terminan en "K" en modo kernel.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued          U
  0   -> _XEventsQueued        U
  0   -> _X11TransBytesReadable U
  0   <- _X11TransBytesReadable U
  0   -> _X11TransSocketBytesReadable U
  0   <- _X11TransSocketBytesReadable U
  0   -> ioctl                 U
  0     -> ioctl               K
  0     -> getf                K
  0       -> set_active_fd      K
  0       <- set_active_fd      K
  0     <- getf                K
  0     -> get_udatamodel      K
  0     <- get_udatamodel      K
  ...
  0     -> releaseef          K
  0       -> clear_active_fd    K
  0       <- clear_active_fd    K
  0       -> cv_broadcast       K
  0       <- cv_broadcast       K
  0       <- releaseef          K
  0     <- ioctl               K
  0     -> ioctl               U
  0   <- XEventsQueued         U
  0 <- XEventsQueued          U
```

Figura 2.9. Monitorización de una llamada al sistema dentro del kernel mediante dtrace, en Solaris 10.

Otros sistemas operativos están empezando a incluir diversas herramientas de rendimiento y de traza, animados por los proyectos de investigación realizados en distintas instituciones, incluyendo el proyecto Paradyn.

Necesitamos también poder hacer estas operaciones con directorios, si disponemos de una estructura de directorios para organizar los archivos en el sistema de archivos. Además, para cualquier archivo o directorio, necesitamos poder determinar los valores de diversos atributos y, quizás, cambiarlos si fuera necesario. Los atributos de archivo incluyen el nombre del archivo, el tipo de archivo, los códigos de protección, la información de las cuentas de usuario, etc. Al menos

son necesarias dos llamadas al sistema (`get file attribute` y `set file attribute`) para cumplir esta función. Algunos sistemas operativos proporcionan muchas más llamadas, como por ejemplo llamadas para mover (`move`) y copiar (`copy`) archivos. Otros proporcionan una API que realiza dichas operaciones usando código software y otras llamadas al sistema, y otros incluso suministran programas del sistema para llevar a cabo dichas tareas. Si los programas del sistema son invocables por otros programas, entonces cada uno de ellos puede ser considerado una API por los demás programas del sistema.

2.4.3 Administración de dispositivos

Un proceso puede necesitar varios recursos para ejecutarse: memoria principal, unidades de disco, acceso a archivos, etc. Si los recursos están disponibles, pueden ser concedidos y el control puede devolverse al proceso de usuario. En caso contrario, el proceso tendrá que esperar hasta que haya suficientes recursos disponibles.

Puede pensarse en los distintos recursos controlados por el sistema operativo como si fueran dispositivos. Algunos de esos dispositivos son dispositivos físicos (por ejemplo, cintas), mientras que en otros puede pensarse como en dispositivos virtuales o abstractos (por ejemplo, archivos). Si hay varios usuarios en el sistema, éste puede requerirnos primero que solicitemos (`request`) el dispositivo, con el fin de asegurarnos el uso exclusivo del mismo. Una vez que hayamos terminado con el dispositivo, lo liberaremos (`release`). Estas funciones son similares a las llamadas al sistema para abrir (`open`) y cerrar (`close`) archivos. Otros sistemas operativos permiten un acceso no administrado a los dispositivos. El riesgo es, entonces, la potencial contienda por el uso de los dispositivos, con el consiguiente riesgo de que se produzcan interbloqueos; este tema se describe en el Capítulo 7.

Una vez solicitado el dispositivo (y una vez que se nos haya asignado), podemos leer (`read`), escribir, (`write`) y reposicionar (`reposition`) el dispositivo, al igual que con los archivos. De hecho, la similitud entre los dispositivos de E/S y los archivos es tan grande que muchos sistemas operativos, incluyendo UNIX, mezclan ambos en una estructura combinada de archivo-dispositivo. Algunas veces, los dispositivos de E/S se identifican mediante nombres de archivo, ubicaciones de directorio o atributos de archivo especiales.

La interfaz de usuario también puede hacer que los archivos y dispositivos parezcan similares, incluso aunque las llamadas al sistema subyacentes no lo sean. Éste es otro ejemplo de las muchas decisiones de diseño que hay que tomar en la creación de un sistema operativo y de una interfaz de usuario.

2.4.4 Mantenimiento de información

Muchas llamadas al sistema existen simplemente con el propósito de transferir información entre el programa de usuario y el sistema operativo. Por ejemplo, la mayoría de los sistemas ofrecen una llamada al sistema para devolver la hora (`time`) y la fecha (`date`) actuales. Otras llamadas al sistema pueden devolver información sobre el sistema, como por ejemplo el número actual de usuarios, el número de versión del sistema operativo, la cantidad de memoria libre o de espacio en disco, etc.

Además, el sistema operativo mantiene información sobre todos sus procesos, y se usan llamadas al sistema para acceder a esa información. Generalmente, también se usan llamadas para configurar la información de los procesos (`get process attributes` y `set process attributes`). En la Sección 3.1.3 veremos qué información se mantiene habitualmente acerca de los procesos.

2.4.5 Comunicaciones

Existen dos modelos comunes de comunicación interprocesos: el modelo de paso de mensajes y el modelo de memoria compartida. En el **modelo de paso de mensajes**, los procesos que se comunican intercambian mensajes entre sí para transferirse información. Los mensajes se pueden inter-

cambiar entre los procesos directa o indirectamente a través de un buzón de correo común. Antes de que la comunicación tenga lugar, debe abrirse una conexión. Debe conocerse de antemano el nombre del otro comunicador, ya sea otro proceso del mismo sistema o un proceso de otra computadora que esté conectada a través de la red de comunicaciones. Cada computadora de una red tiene un *nombre de host*, por el que habitualmente se la conoce. Un *host* también tiene un identificador de red, como por ejemplo una dirección IP. De forma similar, cada proceso tiene un *nombre de proceso*, y este nombre se traduce en un identificador mediante el cual el sistema operativo puede hacer referencia al proceso. Las llamadas al sistema `get hostid` y `get processid` realizan esta traducción. Los identificadores se pueden pasar entonces a las llamadas de propósito general `open` y `close` proporcionadas por el sistema de archivos o a las llamadas específicas al sistema `open connection` y `close connection`, dependiendo del modelo de comunicación del sistema. Usualmente, el proceso receptor debe conceder permiso para que la comunicación tenga lugar, con una llamada de aceptación de la conexión (`accept connection`). La mayoría de los procesos que reciben conexiones son de propósito especial; dichos procesos especiales se denominan *demonios* y son programas del sistema que se suministran específicamente para dicho propósito. Los procesos demonio ejecutan una llamada `wait for connection` y despiertan cuando se establece una conexión. El origen de la comunicación, denominado *cliente*, y el demonio receptor, denominado *servidor*, intercambian entonces mensajes usando las llamadas al sistema para leer (`read message`) y escribir (`write message`) mensajes. La llamada para cerrar la conexión (`close connection`) termina la comunicación.

En el **modelo de memoria compartida**, los procesos usan las llamadas al sistema `shared memory create` y `shared memory attach` para crear y obtener acceso a regiones de la memoria que son propiedad de otros procesos. Recuerde que, normalmente, el sistema operativo intenta evitar que un proceso acceda a la memoria de otro proceso. La memoria compartida requiere que dos o más procesos acuerden eliminar esta restricción; entonces pueden intercambiar información leyendo y escribiendo datos en áreas de la memoria compartida. La forma de los datos y su ubicación son determinadas por parte de los procesos y no están bajo el control del sistema operativo. Los procesos también son responsables de asegurar que no escriban simultáneamente en las mismas posiciones. Tales mecanismos se estudian en el Capítulo 6. En el Capítulo 4, veremos una variante del esquema de procesos (lo que se denomina “hebras de ejecución”) en la que se comparte la memoria de manera predeterminada.

Los dos modelos mencionados son habituales en los sistemas operativos y la mayoría de los sistemas implementan ambos. El modelo de paso de mensajes resulta útil para intercambiar cantidades pequeñas de datos, dado que no hay posibilidad de que se produzcan conflictos; también es más fácil de implementar que el modelo de memoria compartida para la comunicación interprocesos. La memoria compartida permite efectuar la comunicación con una velocidad máxima y con la mayor comodidad, dado que puede realizarse a velocidades de memoria cuando tiene lugar dentro de una misma computadora. Sin embargo, plantea problemas en lo relativo a la protección y sincronización entre los procesos que comparten la memoria.

2.5 Programas del sistema

Otro aspecto fundamental de los sistemas modernos es la colección de programas del sistema. Recuerde la Figura 1.1, que describía la jerarquía lógica de una computadora: en el nivel inferior está el hardware; a continuación se encuentra el sistema operativo, luego los programas del sistema y, finalmente, los programas de aplicaciones. Los programas del sistema proporcionan un cómodo entorno para desarrollar y ejecutar programas. Algunos de ellos son, simplemente, interfaces de usuario para las llamadas al sistema; otros son considerablemente más complejos. Pueden dividirse en las siguientes categorías:

- **Administración de archivos.** Estos programas crean, borran, copian, cambian de nombre, imprimen, vuelcan, listan y, de forma general, manipulan archivos y directorios.
- **Información de estado.** Algunos programas simplemente solicitan al sistema la fecha, la hora, la cantidad de memoria o de espacio de disco disponible, el número de usuarios o

información de estado similar. Otros son más complejos y proporcionan información detallada sobre el rendimiento, los inicios de sesión y los mecanismos de depuración. Normalmente, estos programas formatean los datos de salida y los envían al terminal o a otros dispositivos o archivos de salida, o presentan esos datos en una ventana de la interfaz GUI. Algunos sistemas también soportan un **registro**, que se usa para almacenar y recuperar información de configuración.

- **Modificación de archivos.** Puede disponerse de varios editores de texto para crear y modificar el contenido de los archivos almacenados en el disco o en otros dispositivos de almacenamiento. También puede haber comandos especiales para explorar el contenido de los archivos en busca de un determinado dato o para realizar cambios en el texto.
- **Soporte de lenguajes de programación.** Con frecuencia, con el sistema operativo se proporcionan al usuario compiladores, ensambladores, depuradores e intérpretes para los lenguajes de programación habituales, como por ejemplo, C, C++, Java, Visual Basic y PERL.
- **Carga y ejecución de programas.** Una vez que el programa se ha ensamblado o compilado, debe cargarse en memoria para poder ejecutarlo. El sistema puede proporcionar cargadores absolutos, cargadores reubicables, editores de montaje y cargadores de sustitución. También son necesarios sistemas de depuración para lenguajes de alto nivel o para lenguaje máquina.
- **Comunicaciones.** Estos programas proporcionan los mecanismos para crear conexiones virtuales entre procesos, usuarios y computadoras. Permiten a los usuarios enviar mensajes a las pantallas de otros, explorar páginas web, enviar mensajes de correo electrónico, iniciar una sesión de forma remota o transferir archivos de una máquina a otra.

Además de con los programas del sistema, la mayoría de los sistemas operativos se suministran con programas de utilidad para resolver problemas comunes o realizar operaciones frecuentes. Tales programas son, por ejemplo, exploradores web, procesadores y editores de texto, hojas de cálculo, sistemas de bases de datos, compiladores, paquetes gráficos y de análisis estadístico y juegos. Estos programas se conocen como **utilidades del sistema** o **programas de aplicación**.

Lo que ven del sistema operativo la mayoría de los usuarios está definido por los programas del sistema y de aplicación, en lugar de por las propias llamadas al sistema. Consideremos, por ejemplo, los PC: cuando su computadora ejecuta el sistema operativo Mac OS X, un usuario puede ver la GUI, controlable mediante un ratón y caracterizada por una interfaz de ventanas. Alternativamente (o incluso en una de las ventanas) el usuario puede disponer de una *shell* de UNIX que puede usar como línea de comandos. Ambos tipos de interfaz usan el mismo conjunto de llamadas al sistema, pero las llamadas parecen diferentes y actúan de forma diferente.

2.6 Diseño e implementación del sistema operativo

En esta sección veremos los problemas a los que nos enfrentamos al diseñar e implementar un sistema operativo. Por supuesto, no existen soluciones completas y únicas a tales problemas, pero si podemos indicar una serie de métodos que han demostrado con el tiempo ser adecuados.

2.6.1 Objetivos del diseño

El primer problema al diseñar un sistema es el de definir los objetivos y especificaciones. En el nivel más alto, el diseño del sistema se verá afectado por la elección del hardware y el tipo de sistema: de procesamiento por lotes, de tiempo compartido, monousuario, multiusuario, distribuido, en tiempo real o de propósito general.

Más allá de este nivel superior de diseño, puede ser complicado especificar los requisitos. Sin embargo, éstos se pueden dividir en dos grupos básicos: objetivos del *usuario* y objetivos del *sistema*.

Los usuarios desean ciertas propiedades obvias en un sistema: el sistema debe ser cómodo de utilizar, fácil de aprender y de usar, fiable, seguro y rápido. Por supuesto, estas especificaciones no son particularmente útiles para el diseño del sistema, ya que no existe un acuerdo general sobre cómo llevarlas a la práctica.

Un conjunto de requisitos similar puede ser definido por aquellas personas que tienen que diseñar, crear, mantener y operar el sistema. El sistema debería ser fácil de diseñar, implementar y mantener; debería ser flexible, fiable, libre de errores y eficiente. De nuevo, estos requisitos son vagos y pueden interpretarse de diversas formas.

En resumen, no existe una solución única para el problema de definir los requisitos de un sistema operativo. El amplio rango de sistemas que existen muestra que los diferentes requisitos pueden dar lugar a una gran variedad de soluciones para diferentes entornos. Por ejemplo, los requisitos para VxWorks, un sistema operativo en tiempo real para sistemas integrados, tienen que ser sustancialmente diferentes de los requisitos de MVS, un sistema operativo multiacceso y multiusuario para los *mainframes* de IBM.

Especificar y diseñar un sistema operativo es una tarea extremadamente creativa. Aunque ningún libro de texto puede decirle cómo hacerlo, se ha desarrollado una serie de principios generales en el campo de la **ingeniería del software**, algunos de los cuales vamos a ver ahora.

2.6.2 Mecanismos y políticas

Un principio importante es el de separar las **políticas** de los **mecanismos**. Los mecanismos determinan *cómo* hacer algo; las políticas determinan *qué* hacer. Por ejemplo, el temporizador (véase la Sección 1.5.2) es un mecanismo para asegurar la protección de la CPU, pero la decisión de cuáles deben ser los datos de temporización para un usuario concreto es una decisión de política.

La separación de políticas y mecanismos es importante por cuestiones de flexibilidad. Las políticas probablemente cambien de un sitio a otro o con el paso del tiempo. En el caso peor, cada cambio en una política requerirá un cambio en el mecanismo subyacente; sería, por tanto, deseable un mecanismo general insensible a los cambios de política. Un cambio de política requeriría entonces la redefinición de sólo determinados parámetros del sistema. Por ejemplo, considere un mecanismo para dar prioridad a ciertos tipos de programas: si el mecanismo está apropiadamente separado de la política, puede utilizarse para dar soporte a una decisión política que establezca que los programas que hacen un uso intensivo de la E/S tengan prioridad sobre los que hacen un uso intensivo de la CPU, o para dar soporte a la política contraria.

Los sistemas operativos basados en *microkernel* (Sección 2.7.3) llevan al extremo la separación de mecanismos y políticas, implementando un conjunto básico de bloques componentes primitivos. Estos bloques son prácticamente independientes de las políticas concretas, permitiendo que se añadan políticas y mecanismos más avanzados a través de módulos del *kernel* creados por el usuario o a través de los propios programas de usuario. Por ejemplo, considere la historia de UNIX: al principio, disponía de un planificador de tiempo compartido, mientras que en la versión más reciente de Solaris la planificación se controla mediante una serie de tablas cargables. Dependiendo de la tabla cargada actualmente, el sistema puede ser de tiempo compartido, de procesamiento por lotes, de tiempo real, de compartición equitativa, o cualquier combinación de los mecanismos anteriores. Utilizar un mecanismo de planificación de propósito general permite hacer muchos cambios de política con un único comando, `load-new-table`. En el otro extremo se encuentra un sistema como Windows, en el que tanto mecanismos como políticas se codifican en el sistema para forzar un estilo y aspecto globales. Todas las aplicaciones tienen interfaces similares, dado que la propia interfaz está definida en el *kernel* y en las bibliotecas del sistema. El sistema operativo Mac OS X presenta una funcionalidad similar.

Las decisiones sobre políticas son importantes para la asignación de recursos. Cuando es necesario decidir si un recurso se asigna o no, se debe tomar una decisión política. Cuando la pregunta es *cómo* en lugar de *qué*, es un mecanismo lo que hay que determinar.

2.6.3 Implementación

Una vez que se ha diseñado el sistema operativo, debe implementarse. Tradicionalmente, los sistemas operativos tenían que escribirse en lenguaje ensamblador. Sin embargo, ahora se escriben en lenguajes de alto nivel como C o C++.

El primer sistema que no fue escrito en lenguaje ensamblador fue probablemente el MCP (Master Control Program) para las computadoras Burroughs; MCP fue escrito en una variante de ALGOL. MULTICS, desarrollado en el MIT, fue escrito principalmente en PL/1. Los sistemas operativos Linux y Windows XP están escritos en su mayor parte en C, aunque hay algunas pequeñas secciones de código ensamblador para controladores de dispositivos y para guardar y restaurar el estado de registros.

Las ventajas de usar un lenguaje de alto nivel, o al menos un lenguaje de implementación de sistemas, para implementar sistemas operativos son las mismas que las que se obtiene cuando el lenguaje se usa para programar aplicaciones: el código puede escribirse más rápido, es más compacto y más fácil de entender y depurar. Además, cada mejora en la tecnología de compiladores permitirá mejorar el código generado para el sistema operativo completo, mediante una simple recompilación. Por último, un sistema operativo es más fácil de *portar* (trasladar a algún otro hardware) si está escrito en un lenguaje de alto nivel. Por ejemplo, MS-DOS se escribió en el lenguaje ensamblador 8088 de Intel; en consecuencia, está disponible sólo para la familia Intel de procesadores. Por contraste, el sistema operativo Linux está escrito principalmente en C y está disponible para una serie de CPU diferentes, incluyendo Intel 80X86, Motorola 680X0, SPARC y MIPS RX000.

Las únicas posibles desventajas de implementar un sistema operativo en un lenguaje de alto nivel se reducen a los requisitos de velocidad y de espacio de almacenamiento. Sin embargo, éste no es un problema importante en los sistemas de hoy en día. Aunque un experto programador en lenguaje ensamblador puede generar rutinas eficientes de pequeño tamaño, si lo que queremos es desarrollar programas grandes, un compilador moderno puede realizar análisis complejos y aplicar optimizaciones avanzadas que produzcan un código excelente. Los procesadores modernos tienen una *pipeline* profunda y múltiples unidades funcionales que pueden gestionar dependencias complejas, las cuales pueden desbordar la limitada capacidad de la mente humana para controlar los detalles.

Al igual que sucede con otros sistemas, las principales mejoras de rendimiento en los sistemas operativos son, muy probablemente, el resultado de utilizar mejores estructuras de datos y mejores algoritmos, más que de usar un código optimizado en lenguaje ensamblador. Además, aunque los sistemas operativos tienen un gran tamaño, sólo una pequeña parte del código resulta crítica para conseguir un alto rendimiento; el gestor de memoria y el planificador de la CPU son probablemente las rutinas más críticas. Después de escribir el sistema y de que éste esté funcionando correctamente, pueden identificarse las rutinas que constituyan un cuello de botella y reemplazarse por equivalentes en lenguaje ensamblador.

Para identificar los cuellos de botella, debemos poder monitorizar el rendimiento del sistema. Debe añadirse código para calcular y visualizar medidas del comportamiento del sistema. Hay diversas plataformas en las que el sistema operativo realiza esta tarea, generando trazas que proporcionan información sobre el comportamiento del sistema. Todos los sucesos interesantes se registran, junto con la hora y los parámetros importantes, y se escriben en un archivo. Después, un programa de análisis puede procesar el archivo de registro para determinar el rendimiento del sistema e identificar los cuellos de botella y las ineficiencias. Estas mismas trazas pueden proporcionarse como entrada para una simulación que trate de verificar si resulta adecuado introducir determinadas mejoras. Las trazas también pueden ayudar a los desarrolladores a encontrar errores en el comportamiento del sistema operativo.

2.7 Estructura del sistema operativo

La ingeniería de un sistema tan grande y complejo como un sistema operativo moderno debe hacerse cuidadosamente para que el sistema funcione apropiadamente y pueda modificarse con facilidad. Un método habitual consiste en dividir la tarea en componentes más pequeños, en lugar

de tener un sistema monolítico. Cada uno de estos módulos debe ser una parte bien definida del sistema, con entradas, salidas y funciones cuidadosamente especificadas. Ya hemos visto brevemente en el Capítulo 1 cuáles son los componentes más comunes de los sistemas operativos. En esta sección, veremos cómo estos componentes se interconectan y funden en un *kernel*.

2.7.1 Estructura simple

Muchos sistemas comerciales no tienen una estructura bien definida. Frecuentemente, tales sistemas operativos comienzan siendo sistemas pequeños, simples y limitados y luego crecen más allá de su ámbito original; MS-DOS es un ejemplo de un sistema así. Originalmente, fue diseñado e implementado por unas pocas personas que no tenían ni idea de que iba a terminar siendo tan popular. Fue escrito para proporcionar la máxima funcionalidad en el menor espacio posible, por lo que no fue dividido en módulos de forma cuidadosa. La Figura 2.10 muestra su estructura.

En MS-DOS, las interfaces y niveles de funcionalidad no están separados. Por ejemplo, los programas de aplicación pueden acceder a las rutinas básicas de E/S para escribir directamente en la pantalla y las unidades de disco. Tal libertad hace que MS-DOS sea vulnerable a programas erróneos (o maliciosos), lo que hace que el sistema completo falle cuando los programas de usuario fallan. Como el 8088 de Intel para el que fue escrito no proporciona un modo dual ni protección hardware, los diseñadores de MS-DOS no tuvieron más opción que dejar accesible el hardware base.

Otro ejemplo de estructuración limitada es el sistema operativo UNIX original. UNIX es otro sistema que inicialmente estaba limitado por la funcionalidad hardware. Consta de dos partes separadas: el *kernel* y los programas del sistema. El *kernel* se divide en una serie de interfaces y controladores de dispositivo, que se han ido añadiendo y ampliando a lo largo de los años, a medida que UNIX ha ido evolucionando. Podemos ver el tradicional sistema operativo UNIX como una estructura de niveles, ilustrada en la Figura 2.11. Todo lo que está por debajo de la interfaz de llamadas al sistema y por encima del hardware físico es el *kernel*. El *kernel* proporciona el sistema de archivos, los mecanismos de planificación de la CPU, la funcionalidad de gestión de memoria y otras funciones del sistema operativo, a través de las llamadas al sistema. En resumen, es una enorme cantidad de funcionalidad que se combina en un sólo nivel. Esta estructura monolítica era difícil de implementar y de mantener.

2.7.2 Estructura en niveles

Con el soporte hardware apropiado, los sistemas operativos puede dividirse en partes más pequeñas y más adecuadas que lo que permitían los sistemas originales MS-DOS o UNIX. El sistema operativo puede entonces mantener un control mucho mayor sobre la computadora y sobre las

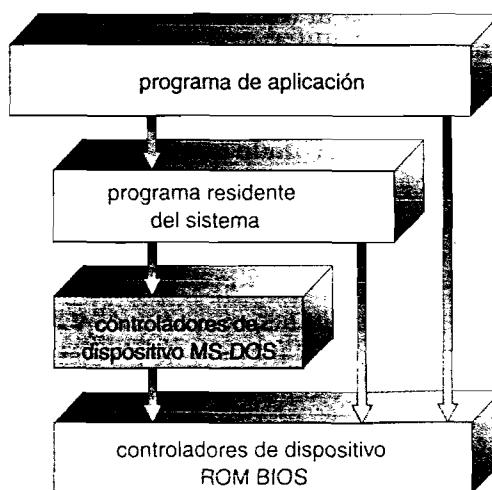


Figura 2.10 Estructura de niveles de MS-DOS.

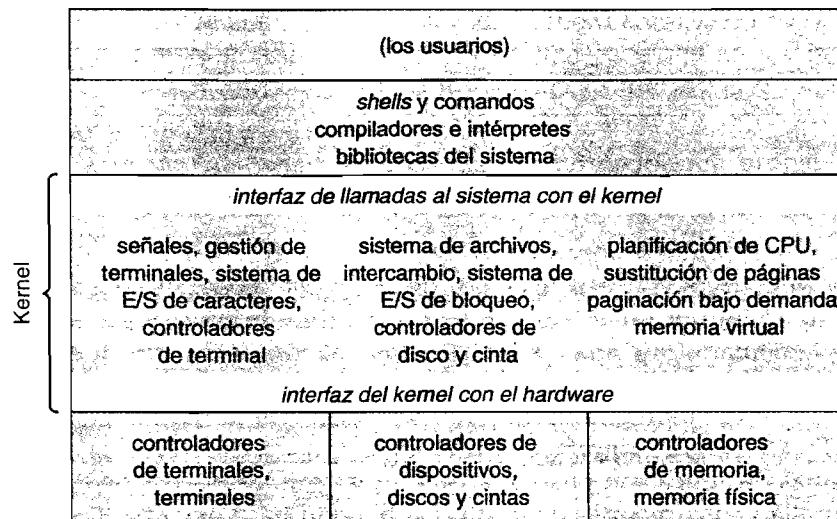


Figura 2.11 Estructura del sistema UNIX.

aplicaciones que hacen uso de dicha computadora. Los implementadores tienen más libertad para cambiar el funcionamiento interno del sistema y crear sistemas operativos modulares. Con el método de diseño arriba-abajo, se determinan las características y la funcionalidad globales y se separan en componentes. La ocultación de los detalles a ojos de los niveles superiores también es importante, dado que deja libres a los programadores para implementar las rutinas de bajo nivel como prefieran, siempre que la interfaz externa de la rutina permanezca invariable y la propia rutina realice la tarea anunciada.

Un sistema puede hacerse modular de muchas formas. Un posible método es mediante una **estructura en niveles**, en el que el sistema operativo se divide en una serie de capas (niveles). El nivel inferior (nivel 0) es el hardware; el nivel superior (nivel N) es la interfaz de usuario. Esta estructura de niveles se ilustra en la Figura 2.12. Un nivel de un sistema operativo es una implementación de un objeto abstracto formado por una serie de datos y por las operaciones que permiten manipular dichos datos. Un nivel de un sistema operativo típico (por ejemplo, el nivel M) consta de estructuras de datos y de un conjunto de rutinas que los niveles superiores pueden invocar. A su vez, el nivel M puede invocar operaciones sobre los niveles inferiores.

La principal ventaja del método de niveles es la simplicidad de construcción y depuración. Los niveles se seleccionan de modo que cada uno usa funciones (operaciones) y servicios de los niveles inferiores. Este método simplifica la depuración y la verificación del sistema. El primer nivel puede depurarse sin afectar al resto del sistema, dado que, por definición, sólo usa el hardware básico (que se supone correcto) para implementar sus funciones. Una vez que el primer nivel se ha depurado, puede suponerse correcto su funcionamiento mientras se depura el segundo nivel, etc. Si se encuentra un error durante la depuración de un determinado nivel, el error tendrá que estar localizado en dicho nivel, dado que los niveles inferiores a él ya se han depurado. Por tanto, el diseño e implementación del sistema se simplifican.

Cada nivel se implementa utilizando sólo las operaciones proporcionadas por los niveles inferiores. Un nivel no necesita saber cómo se implementan dichas operaciones; sólo necesita saber qué hacen esas operaciones. Por tanto, cada nivel oculta a los niveles superiores la existencia de determinadas estructuras de datos, operaciones y hardware.

La principal dificultad con el método de niveles es la de definir apropiadamente los diferentes niveles. Dado que un nivel sólo puede usar los servicios de los niveles inferiores, es necesario realizar una planificación cuidadosa. Por ejemplo, el controlador de dispositivo para almacenamiento de reserva (espacio en disco usado por los algoritmos de memoria virtual) debe estar en un nivel inferior que las rutinas de gestión de memoria, dado que la gestión de memoria requiere la capacidad de usar el almacenamiento de reserva.

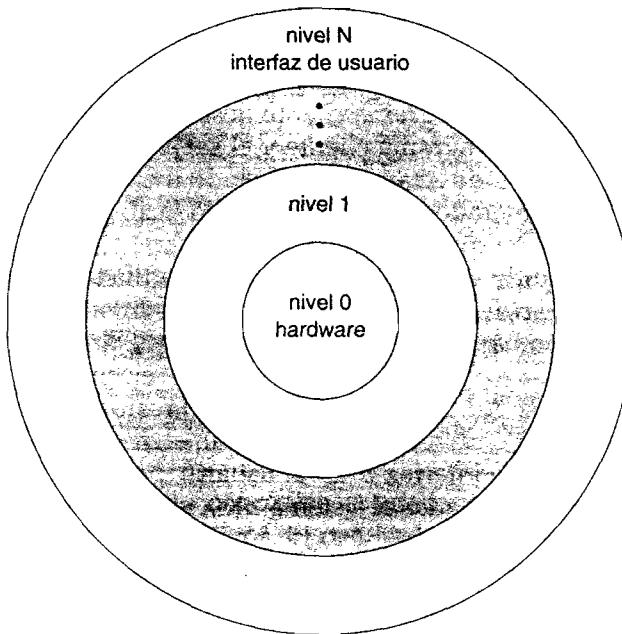


Figura 2.12 Un sistema operativo estructurado en niveles.

Otros requisitos pueden no ser tan obvios. Normalmente, el controlador de almacenamiento de reserva estará por encima del planificador de la CPU, dado que el controlador puede tener que esperar a que se realicen determinadas operaciones de E/S y la CPU puede asignarse a otra tarea durante este tiempo. Sin embargo, en un sistema de gran envergadura, el planificador de la CPU puede tener más información sobre todos los procesos activos de la que cabe en memoria. Por tanto, esta información puede tener que cargarse y descargarse de memoria, requiriendo que el controlador de almacenamiento de reserva esté por debajo del planificador de la CPU.

Un último problema con las implementaciones por niveles es que tienden a ser menos eficientes que otros tipos de implementación. Por ejemplo, cuando un programa de usuario ejecuta una operación de E/S, realiza una llamada al sistema que será capturada por el nivel de E/S, el cual llamará al nivel de gestión de memoria, el cual a su vez llamará al nivel de planificación de la CPU, que pasará a continuación la llamada al hardware. En cada nivel, se pueden modificar los parámetros, puede ser necesario pasar datos, etc. Cada nivel añade así una carga de trabajo adicional a la llamada al sistema; el resultado neto es una llamada al sistema que tarda más en ejecutarse que en un sistema sin niveles.

Estas limitaciones han hecho surgir en los últimos años una cierta reacción contra los sistemas basados en niveles. En los diseños más recientes, se utiliza un menor número de niveles, con más funcionalidad por cada nivel, lo que proporciona muchas de las ventajas del código modular, a la vez que se evitan los problemas más difíciles relacionados con la definición e interacción de los niveles.

2.7.3 *Microkernels*

Ya hemos visto que, a medida que UNIX se expandía, el *kernel* se hizo grande y difícil de gestionar. A mediados de los años 80, los investigadores de la universidad de Carnegie Mellon desarrollaron un sistema operativo denominado **Mach** que modularizaba el *kernel* usando lo que se denomina *microkernel*. Este método estructura el sistema operativo eliminando todos los componentes no esenciales del *kernel* e implementándolos como programas del sistema y de nivel de usuario; el resultado es un *kernel* más pequeño. No hay consenso en lo que se refiere a qué servicios deberían permanecer en el *kernel* y cuáles deberían implementarse en el espacio de usuario. Sin embargo, normalmente los *microkernels* proporcionan una gestión de la memoria y de los procesos mínima, además de un mecanismo de comunicaciones.

La función principal del *microkernel* es proporcionar un mecanismo de comunicaciones entre el programa cliente y los distintos servicios que se ejecutan también en el espacio de usuario. La comunicación se proporciona mediante *paso de mensajes*, método que se ha descrito en la Sección 2.4.5. Por ejemplo, si el programa cliente desea acceder a un archivo, debe interactuar con el servidor de archivos. El programa cliente y el servicio nunca interactúan directamente, sino que se comunican de forma indirecta intercambiando mensajes con el *microkernel*.

Otra ventaja del método de *microkernel* es la facilidad para ampliar el sistema operativo. Todos los servicios nuevos se añaden al espacio de usuario y, en consecuencia, no requieren que se modifique el *kernel*. Cuando surge la necesidad de modificar el *kernel*, los cambios tienden a ser pocos, porque el *microkernel* es un *kernel* muy pequeño. El sistema operativo resultante es más fácil de portar de un diseño hardware a otro. El *microkernel* también proporciona más seguridad y fiabilidad, dado que la mayor parte de los servicios se ejecutan como procesos de usuario, en lugar de como procesos del *kernel*. Si un servicio falla, el resto del sistema operativo no se ve afectado.

Varios sistemas operativos actuales utilizan el método de *microkernel*. Tru64 UNIX (antes Digital UNIX) proporciona una interfaz UNIX al usuario, pero se implementa con un *kernel* Mach. El *kernel* Mach transforma las llamadas al sistema UNIX en mensajes dirigidos a los servicios apropiados de nivel de usuario.

Otro ejemplo es QNX. QNX es un sistema operativo en tiempo real que se basa también en un diseño de *microkernel*. El *microkernel* de QNX proporciona servicios para paso de mensajes y planificación de procesos. También gestiona las comunicaciones de red de bajo nivel y las interrupciones hardware. Los restantes servicios de QNX son proporcionados por procesos estándar que se ejecutan fuera del *kernel*, en modo usuario.

Lamentablemente, los *microkernels* pueden tener un rendimiento peor que otras soluciones, debido a la carga de procesamiento adicional impuesta por las funciones del sistema. Consideremos la historia de Windows NT: la primera versión tenía una organización de *microkernel* con niveles. Sin embargo, esta versión proporcionaba un rendimiento muy bajo, comparado con el de Windows 95. La versión Windows NT 4.0 solucionó parcialmente el problema del rendimiento, pasando diversos niveles del espacio de usuario al espacio del *kernel* e integrándolos más estrechamente. Para cuando se diseñó Windows XP, la arquitectura del sistema operativo era más de tipo monolítico que basada en *microkernel*.

2.7.4 Módulos

Quizá la mejor metodología actual para diseñar sistemas operativos es la que usa las técnicas de programación orientada a objetos para crear un *kernel* modular. En este caso, el *kernel* dispone de un conjunto de componentes fundamentales y enlaza dinámicamente los servicios adicionales, bien durante el arranque o en tiempo de ejecución. Tal estrategia utiliza módulos que se cargan dinámicamente y resulta habitual en las implementaciones modernas de UNIX, como Solaris, Linux y Mac OS X. Por ejemplo, la estructura del sistema operativo Solaris, mostrada en la Figura 2.13, está organizada alrededor de un *kernel central* con siete tipos de módulos de *kernel* cargables:

1. Clases de planificación
2. Sistemas de archivos
3. Llamadas al sistema cargables
4. Formatos ejecutables
5. Módulos STREAMS
6. Módulos misceláneos
7. Controladores de bus y de dispositivos

Un diseño así permite al *kernel* proporcionar servicios básicos y también permite implementar ciertas características dinámicamente. Por ejemplo, se pueden añadir al *kernel* controladores de

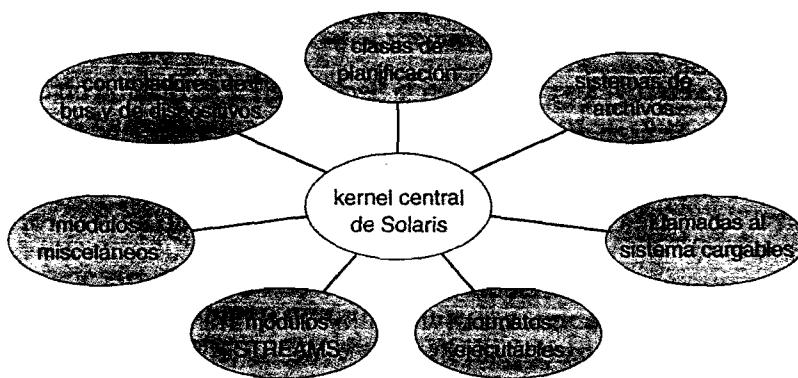


Figura 2.13 Módulos cargables de Solaris.

bus y de dispositivos para hardware específico y puede agregarse como módulo cargable el soporte para diferentes sistemas de archivos. El resultado global es similar a un sistema de niveles, en el sentido de que cada sección del *kernel* tiene interfaces bien definidas y protegidas, pero es más flexible que un sistema de niveles, porque cualquier módulo puede llamar a cualquier otro módulo. Además, el método es similar a la utilización de un *microkernel*, ya que el módulo principal sólo dispone de las funciones esenciales y de los conocimientos sobre cómo cargar y comunicarse con otros módulos; sin embargo, es más eficiente que un *microkernel*, ya que los módulos no necesitan invocar un mecanismo de paso de mensajes para comunicarse.

El sistema operativo Mac OS X de las computadoras Apple Macintosh utiliza una estructura híbrida. Mac OS X (también conocido como *Darwin*) estructura el sistema operativo usando una técnica por niveles en la que uno de esos niveles es el *microkernel* Mach. En la Figura 2.14 se muestra la estructura de Mac OS X.

Los niveles superiores incluyen los entornos de aplicación y un conjunto de servicios que proporcionan una interfaz gráfica a las aplicaciones. Por debajo de estos niveles se encuentra el entorno del *kernel*, que consta fundamentalmente del *microkernel* Mach y el *kernel* BSD. Mach proporciona la gestión de memoria, el soporte para llamadas a procedimientos remotos (RPC, remote procedure call) y facilidades para la comunicación interprocesos (IPC, interprocess communication), incluyendo un mecanismo de paso de mensajes, así como mecanismos de planificación de hebras de ejecución. El módulo BSD proporciona una interfaz de línea de comandos BSD, soporte para red y sistemas de archivos y una implementación de las API de POSIX, incluyendo Pthreads. Además de Mach y BSD, el entorno del *kernel* proporciona un kit de E/S para el desarrollo de controladores de dispositivo y módulos dinámicamente cargables (que Mac OS X denomina **extensões del kernel**). Como se muestra en la figura, las aplicaciones y los servicios comunes pueden usar directamente las facilidades de Mach o BSD.

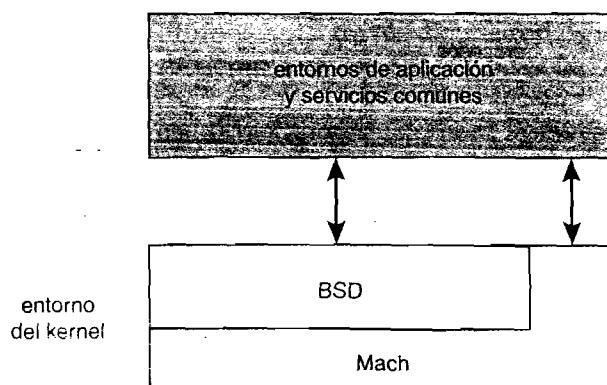


Figura 2.14 Estructura de Mac OS X.

2.8 Máquinas virtuales

La estructura en niveles descrita en la Sección 2.7.2 se plasma en el es llevado a su conclusión lógica con el concepto de **máquina virtual**. La idea fundamental que subyace a una máquina virtual es la de abstraer el hardware de la computadora (la CPU, la memoria, las unidades de disco, las tarjetas de interfaz de red, etc.), forjando varios entornos de ejecución diferentes, creando así la ilusión de que cada entorno de ejecución está operando en su propia computadora privada.

Con los mecanismos de planificación de la CPU (Capítulo 5) y las técnicas de memoria virtual (Capítulo 9), un sistema operativo puede crear la ilusión de que un proceso tiene su propio procesador con su propia memoria (virtual). Normalmente, un proceso utiliza características adicionales, tales como llamadas al sistema y un sistema de archivos, que el hardware básico no proporciona. El método de máquina virtual no proporciona ninguna de estas funcionalidades adicionales, sino que proporciona una interfaz que es *idéntica* al hardware básico subyacente. Cada proceso dispone de una copia (virtual) de la computadora subyacente (Figura 2.15).

Existen varias razones para crear una máquina virtual, estando todas ellas fundamentalmente relacionadas con el poder compartir el mismo hardware y, a pesar de ello, operar con entornos de ejecución diferentes (es decir, diferentes sistemas operativos) de forma concurrente. Exploraremos las ventajas de las máquinas virtuales más detalladamente en la Sección 2.8.2. A lo largo de esta sección, vamos a ver el sistema operativo VM para los sistemas IBM, que constituye un útil caso de estudio; además, IBM fue una de las empresas pioneras en este área.

Una de las principales dificultades del método de máquina virtual son los sistemas de disco. Supongamos que la máquina física dispone de tres unidades de disco, pero desea dar soporte a siete máquinas virtuales. Claramente, no se puede asignar una unidad de disco a cada máquina virtual, dado que el propio software de la máquina virtual necesitará un importante espacio en disco para proporcionar la memoria virtual y los mecanismos de gestión de colas. La solución consiste en proporcionar discos virtuales, denominados *minidiscos* en el sistema operativo VM de IBM, que son idénticos en todo, excepto en el tamaño. El sistema implementa cada minidisco asignando tantas pistas de los discos físicos como necesite el minidisco. Obviamente, la suma de los tamaños de todos los minidiscos debe ser menor que el espacio de disco físicamente disponible.

Cuando los usuarios disponen de sus propias máquinas virtuales, pueden ejecutar cualquiera de los sistemas operativos o paquetes software disponibles en la máquina subyacente. En los sistemas VM de IBM, los usuarios ejecutan normalmente CMS, un sistema operativo interactivo monousuario. El software de la máquina virtual se ocupa de multiprogramar las múltiples máquinas virtuales sobre una única máquina física, no preocupándose de ningún aspecto relativo al

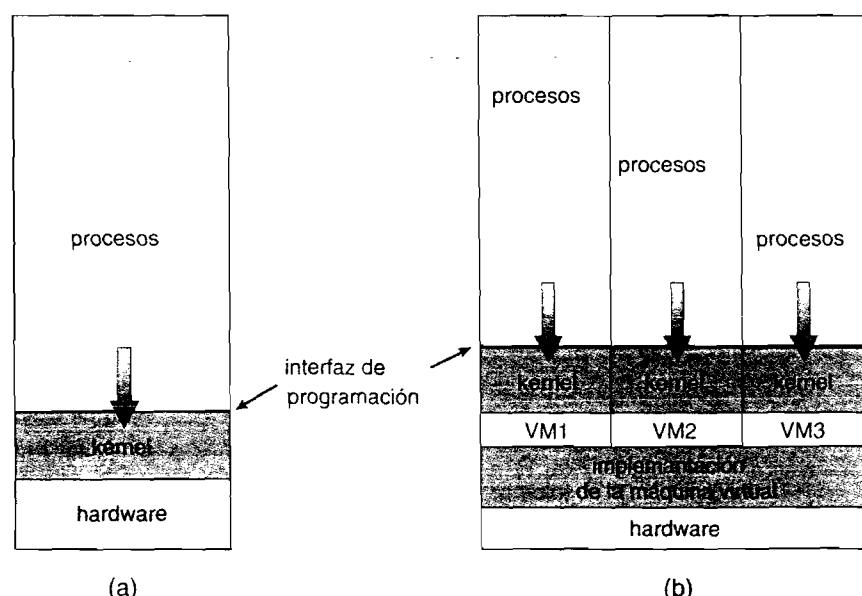


Figura 2.15 Modelos de sistemas. (a) Máquina no virtual. (b) Máquina virtual.

software de soporte al usuario. Esta arquitectura proporciona una forma muy útil de dividir el problema de diseño de un sistema interactivo multiusuario en dos partes más pequeñas.

2.8.1 Implementación

Aunque el concepto de máquina virtual es muy útil, resulta difícil de implementar. Es preciso realizar un duro trabajo para proporcionar un duplicado exacto de la máquina subyacente. Recuerde que la máquina subyacente tiene dos modos: modo usuario y modo *kernel*. El software de la máquina virtual puede ejecutarse en modo *kernel*, dado que es el sistema operativo; la máquina virtual en sí puede ejecutarse sólo en modo usuario. Sin embargo, al igual que la máquina física tiene dos modos, también tiene que tenerlos la máquina virtual. En consecuencia, hay que tener un modo usuario virtual y un modo *kernel* virtual, ejecutándose ambos en modo usuario físico. Las acciones que dan lugar a la transferencia del modo usuario al modo *kernel* en una máquina real (tal como una llamada al sistema o un intento de ejecutar una instrucción privilegiada) también tienen que hacer que se pase del modo usuario virtual al modo *kernel* virtual en una máquina virtual.

Tal transferencia puede conseguirse del modo siguiente. Cuando se hace una llamada al sistema por parte de un programa que se esté ejecutando en una máquina virtual en modo usuario virtual, se produce una transferencia al monitor de la máquina virtual en la máquina real. Cuando el monitor de la máquina virtual obtiene el control, puede cambiar el contenido de los registros y del contador de programa para que la máquina virtual simule el efecto de la llamada al sistema. A continuación, puede reiniciar la máquina virtual, que ahora se encontrará en modo *kernel* virtual.

Por supuesto, la principal diferencia es el tiempo. Mientras que la E/S real puede tardar 100 milisegundos, la E/S virtual puede llevar menos tiempo (puesto que se pone en cola) o más tiempo (puesto que es interpretada). Además, la CPU se multiprograma entre muchas máquinas virtuales, ralentizando aún más las máquinas virtuales de manera impredecible. En el caso extremo, puede ser necesario simular todas las instrucciones para proporcionar una verdadera máquina virtual. VM funciona en máquinas IBM porque las instrucciones normales de las máquinas virtuales pueden ejecutarse directamente por hardware. Sólo las instrucciones privilegiadas (necesarias fundamentalmente para operaciones de E/S) deben simularse y, por tanto, se ejecutan más lentamente.

2.8.2 Beneficios

El concepto de máquina virtual presenta varias ventajas. Observe que, en este tipo de entorno, existe una protección completa de los diversos recursos del sistema. Cada máquina virtual está completamente aislada de las demás, por lo que no existen problemas de protección. Sin embargo, no es posible la compartición directa de recursos. Se han implementados dos métodos para permitir dicha compartición. En primer lugar, es posible compartir un minidisco y, por tanto, compartir los archivos. Este esquema se basa en el concepto de disco físico compartido, pero se implementa por software. En segundo lugar, es posible definir una red de máquinas virtuales, pudiendo cada una de ellas enviar información a través de una red de comunicaciones virtual. De nuevo, la red se modela siguiendo el ejemplo de las redes físicas de comunicaciones, aunque se implementa por software.

Un sistema de máquina virtual es un medio perfecto para la investigación y el desarrollo de sistemas operativos. Normalmente, modificar un sistema operativo es una tarea complicada: los sistemas operativos son programas grandes y complejos, y es difícil asegurar que un cambio en una parte no causará errores complicados en alguna otra parte. La potencia del sistema operativo hace que su modificación sea especialmente peligrosa. Dado que el sistema operativo se ejecuta en modo *kernel*, un cambio erróneo en un puntero podría dar lugar a un error que destruyera el sistema de archivos completo. Por tanto, es necesario probar cuidadosamente todos los cambios realizados en el sistema operativo.

Sin embargo, el sistema operativo opera y controla la máquina completa. Por tanto, el sistema actual debe detenerse y quedar fuera de uso mientras que se realizan cambios y se prueban. Este

período de tiempo habitualmente se denomina *tiempo de desarrollo del sistema*. Dado que el sistema deja de estar disponible para los usuarios, a menudo el tiempo de desarrollo del sistema se planifica para las noches o los fines de semana, cuando la carga del sistema es menor.

Una máquina virtual puede eliminar gran parte de este problema. Los programadores de sistemas emplean su propia máquina virtual y el desarrollo del sistema se hace en la máquina virtual, en lugar de en la máquina física; rara vez se necesita interrumpir la operación normal del sistema para acometer las tareas de desarrollo.

2.8.3 Ejemplos

A pesar de las ventajas de las máquinas virtuales, en los años posteriores a su desarrollo recibieron poca atención. Sin embargo, actualmente las máquinas virtuales se están poniendo de nuevo de moda como medio para solucionar problemas de compatibilidad entre sistemas. En esta sección, exploraremos dos populares máquinas virtuales actuales: VMware y la máquina virtual Java. Como veremos, normalmente estas máquinas virtuales operan por encima de un sistema operativo de cualquiera de los tipos que se han visto con anterioridad. Por tanto, los distintos métodos de diseño de sistemas operativos (en niveles, basado en *microkernel*, modular y máquina virtual) no son mutuamente excluyentes.

2.8.3.1 VMware

VMware es una popular aplicación comercial que abstrae el hardware 80x86 de Intel, creando una serie de máquinas virtuales aisladas. VMware se ejecuta como una aplicación sobre un sistema operativo *host*, tal como Windows o Linux, y permite al sistema *host* ejecutar de forma concurrente varios **sistemas operativos huésped** diferentes como máquinas virtuales independientes.

Considere el siguiente escenario: un desarrollador ha diseñado una aplicación y desea probarla en Linux, FreeBSD, Windows NT y Windows XP. Una opción es conseguir cuatro computadoras diferentes, ejecutando cada una de ellas una copia de uno de los sistemas operativos. Una alternativa sería instalar primero Linux en una computadora y probar la aplicación, instalar después FreeBSD y probar la aplicación y así sucesivamente. Esta opción permite emplear la misma computadora física, pero lleva mucho tiempo, dado que es necesario instalar un sistema operativo para cada prueba. Estas pruebas podrían llevarse a cabo *de forma concurrente* sobre la misma computadora física usando VMware. En este caso, el programador podría probar la aplicación en un sistema operativo *host* y tres sistemas operativos huésped, ejecutando cada sistema como una máquina virtual diferente.

La arquitectura de un sistema así se muestra en la Figura 2.16. En este escenario, Linux se ejecuta como el sistema operativo *host*; FreeBSD, Windows NT y Windows XP se ejecutan como sistemas operativos huésped. El nivel de virtualización es el corazón de VMware, ya que abstrae el hardware físico, creando máquinas virtuales aisladas que se ejecutan como sistemas operativos huésped. Cada máquina virtual tiene su propia CPU, memoria, unidades de disco, interfaces de red, etc., virtuales.

2.8.3.2 Máquina virtual Java

Java es un popular lenguaje de programación orientado a objetos introducido por Sun Microsystems en 1995. Además de una especificación de lenguaje y una amplia biblioteca de interfaces de programación de aplicaciones, Java también proporciona una especificación para una máquina virtual Java, JVM (Java virtual machine).

Los objetos Java se especifican mediante clases, utilizando la estructura `class`; cada programa Java consta de una o más clases. Para cada clase Java, el compilador genera un archivo de salida (`.class`) en **código intermedio (bytecode)** que es neutral con respecto a la arquitectura y se ejecutará sobre cualquier implementación de la JVM.

La JVM es una especificación de una computadora abstracta. Consta de un **cargador de clases** y de un intérprete de Java que ejecuta el código intermedio arquitectónicamente neutro, como se

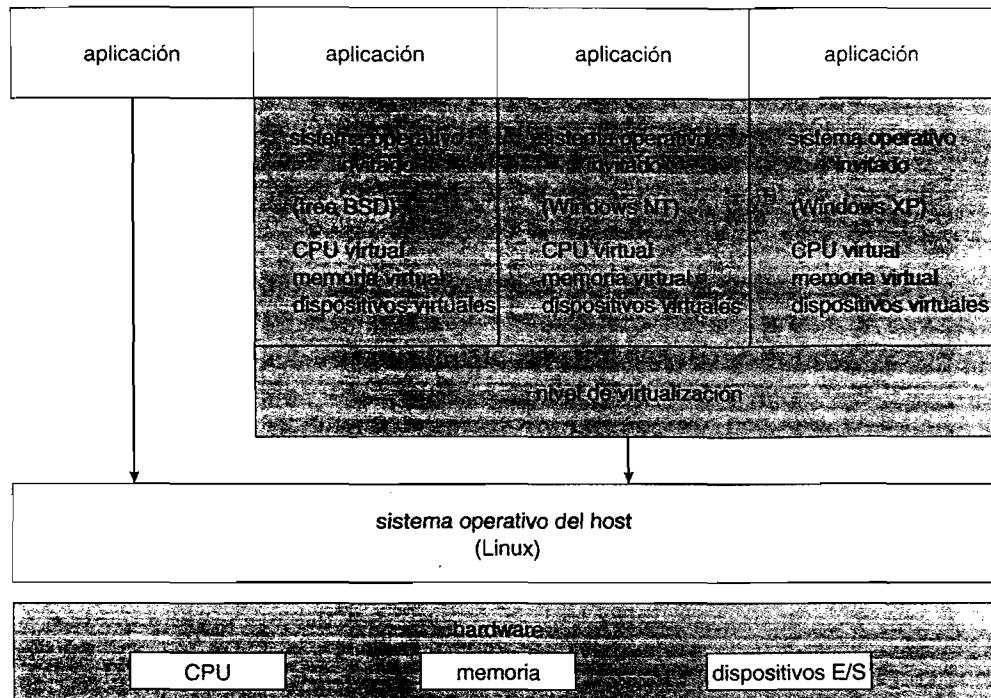


Figura 2.16 Arquitectura de VMware.

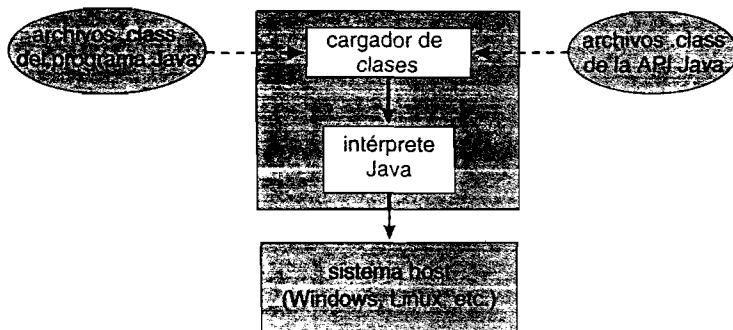


Figura 2.17 Máquina virtual Java.

muestra en la Figura 2.17. El cargador de clases carga los archivos .class compilados correspondientes tanto al programa Java como a la API Java, para ejecutarlos mediante el intérprete de Java. Después de cargar una clase, el verificador comprueba que el archivo .class es un código intermedio Java válido y que no desborda la pila ni por arriba ni por abajo. También verifica que el código intermedio no realice operaciones aritméticas con los punteros que proporcionen acceso ilegal a la memoria. Si la clase pasa la verificación, el intérprete de Java la ejecuta. La JVM también gestiona automáticamente la memoria, llevando a cabo las tareas de **recolección de memoria**, que consisten en reclamar la memoria de los objetos que ya no estén siendo usados, para devolverla al sistema. Buena parte de la investigación actual se centra en el desarrollo de algoritmos de recolección de memoria que permitan aumentar la velocidad de los programas Java ejecutados en la máquina virtual.

La JVM puede implementarse por software encima de un sistema operativo *host*, como Windows, Linux o Mac OS X, o bien puede implementarse como parte de un explorador web. Alternativamente, la JVM puede implementarse por hardware en un chip específicamente diseñado para ejecutar programas Java. Si la JVM se implementa por software, el intérprete de Java interpreta las operaciones en código intermedio una por una. Una técnica software más rápida consiste

.NET FRAMEWORK

.NET Framework es una recopilación de tecnologías, incluyendo un conjunto de bibliotecas de clases y un entorno de ejecución, que proporcionan conjuntamente una plataforma para el desarrollo software. Esta plataforma permite escribir programas destinados a ejecutarse sobre .NET Framework, en lugar de sobre una arquitectura específica. Un programa escrito para .NET Framework no necesita preocuparse sobre las especificidades del hardware o del sistema operativo sobre el que se ejecutara; por tanto, cualquier arquitectura que implemente .NET podrá ejecutar adecuadamente el programa. Esto se debe a que el entorno de ejecución abstracta esos detalles y proporciona una máquina virtual que actúa como intermediario entre el programa en ejecución y la arquitectura subyacente.

En el corazón de .NET Framework se encuentra el entorno CLR (Common Language Runtime). El CLR es la implementación de la máquina virtual .NET. Proporciona un entorno para ejecutar programas escritos en cualquiera de los lenguajes admitidos por .NET Framework. Los programas escritos en lenguajes como C# y VB.NET se compilan en un lenguaje intermedio independiente de la arquitectura denominado MS-IL (Microsoft Intermediate Language, lenguaje intermedio de Microsoft). Estos archivos compilados, denominados "ensamblados", incluyen instrucciones y metadatos MS-IL y utilizan una extensión de archivo EXE o DLL. Durante la ejecución de un programa, el CLR carga los ensamblados en lo que se conoce como dominio de aplicación. A medida que el programa en ejecución solicita instrucciones, el CLR convierte las instrucciones MS-IL contenidas en los ensamblados en código nativo que es específico de la arquitectura subyacente, usando un mecanismo de compilación *just-in-time*. Una vez que las instrucciones se han convertido a código nativo, se conservarán y continuarán ejecutándose como código nativo de la CPU. La arquitectura del entorno CLR para .NET Framework se muestra en la Figura 2.18.

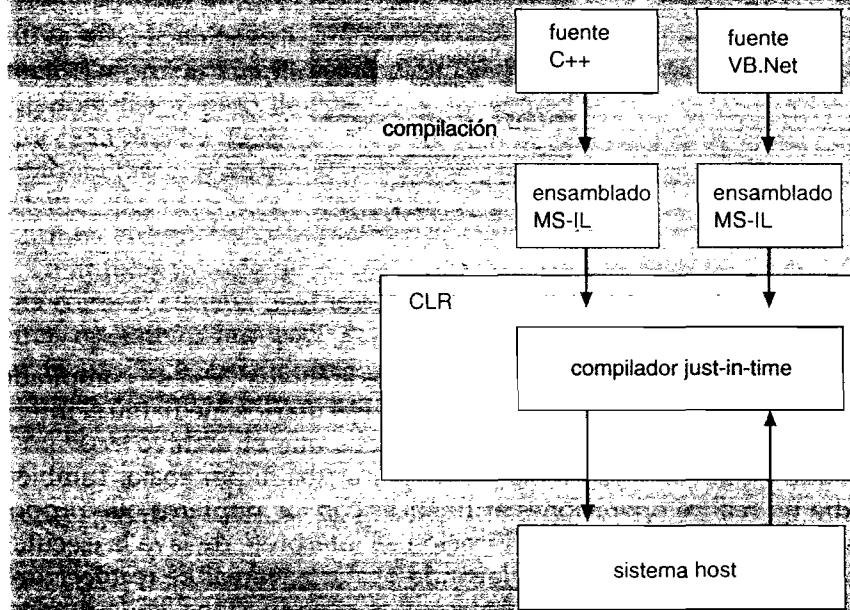


Figura 2.18 Arquitectura de CLR para .NET Framework.

en emplear un compilador *just-in-time*. En este caso, la primera vez que se invoca un método Java, el código intermedio correspondiente al método se convierte a lenguaje máquina nativo del sistema *host*. Estas operaciones se almacenan en caché, con el fin de que las siguientes invocaciones del método se realicen usando las instrucciones en código máquina nativo y las operaciones

en código intermedio no tengan que interpretarse de nuevo. Una técnica que es potencialmente incluso más rápida es la de ejecutar la JVM por hardware, usando un chip Java especial que ejecute las operaciones en código intermedio Java como código nativo, obviando así la necesidad de un intérprete Java o un compilador *just-in-time*.

2.9 Generación de sistemas operativos

Es posible diseñar, codificar e implementar un sistema operativo específicamente para una máquina concreta en una instalación determinada. Sin embargo, lo más habitual es que los sistemas operativos se diseñen para ejecutarse en cualquier clase de máquina y en diversas instalaciones, con una amplia variedad de configuraciones de periféricos. El sistema debe entonces configurarse o generarse para cada computadora en concreto, un proceso que en ocasiones se conoce como **generación del sistema** (SYSGEN, system generation).

Normalmente, el sistema operativo se distribuye en discos o en CD-ROM. Para generar un sistema, se emplea un programa especial. El programa SYSGEN lee un archivo determinado o pide al operador del sistema información sobre la configuración específica del hardware; o bien prueba directamente el hardware para determinar qué componentes se encuentran instalados. Hay que determinar los siguientes tipos de información:

- ¿Qué CPU se va a usar? ¿Qué opciones están instaladas (conjuntos ampliados de instrucciones, aritmética en punto flotante, etc.)? En sistemas con múltiples CPU, debe describirse cada una de ellas.
- ¿Qué cantidad de memoria hay disponible? Algunos sistemas determinarán este valor por sí mismos haciendo referencia a una posición de memoria tras otra, hasta generar un fallo de “dirección ilegal”. Este procedimiento define el final de las direcciones legales y, por tanto, la cantidad de memoria disponible.
- ¿Qué dispositivos se encuentran instalados? El sistema necesitará saber cómo direccionar cada dispositivo (el número de dispositivo), el número de interrupción del dispositivo, el tipo y modelo de dispositivo y cualquier otra característica relevante del dispositivo.
- ¿Qué opciones del sistema operativo se desean o qué valores de parámetros se van a usar? Estas opciones o valores deben incluir cuántos búferes se van a usar y de qué tamaño, qué tipo de algoritmo de planificación de CPU se desea, cuál es el máximo número de procesos que se va a soportar, etc.

Una vez que se ha determinado esta información, puede utilizarse de varias formas. Por un lado, un administrador de sistemas puede usarla para modificar una copia del código fuente del sistema de operativo y, a continuación, compilar el sistema operativo completo. Las declaraciones de datos, valores de inicialización y constantes, junto con los mecanismos de compilación condicional, permiten generar una versión objeto de salida para el sistema operativo que estará adaptada al sistema descrito.

En un nivel ligeramente menos personalizado, la descripción del sistema puede dar lugar a la creación de una serie de tablas y a la selección de módulos de una biblioteca precompilada. Estos módulos se montan para formar el sistema operativo final. El proceso de selección permite que la biblioteca contenga los controladores de dispositivo para todos los dispositivos de E/S soportados, pero sólo se montan con el sistema operativo los que son necesarios. Dado que el sistema no se recompila, la generación del sistema es más rápida, pero el sistema resultante puede ser demasiado general.

En el otro extremo, es posible construir un sistema que esté completamente controlado por tablas. Todo el código forma siempre parte del sistema y la selección se produce en tiempo de ejecución, en lugar de en tiempo de compilación o de montaje. La generación del sistema implica simplemente la creación de las tablas apropiadas que describan el sistema.

Las principales diferencias entre estos métodos son el tamaño y la generalidad del sistema final, y la facilidad de modificación cuando se producen cambios en la configuración del hardware. Tenga en cuenta, por ejemplo, el coste de modificar el sistema para dar soporte a un terminal

gráfico u otra unidad de disco recién adquiridos. Por supuesto, dicho coste variará en función de la frecuencia (o no frecuencia) de dichos cambios.

2.10 Arranque del sistema

Después de haber generado un sistema operativo, debe ponerse a disposición del hardware para su uso. Pero, ¿sabe el hardware dónde está el *kernel* o cómo cargarlo? El procedimiento de inicialización de una computadora mediante la carga del *kernel* se conoce como **arranque** del sistema. En la mayoría de los sistemas informáticos, una pequeña parte del código, conocida como **programa de arranque** o **cargador de arranque**, se encarga de localizar el *kernel*, lo carga en la memoria principal e inicia su ejecución. Algunos sistemas informáticos, como los PC, usan un proceso de dos pasos en que un sencillo cargador de arranque extrae del disco un programa de arranque más complejo, el cual a su vez carga el *kernel*.

Cuando una CPU recibe un suceso de reinicialización (por ejemplo, cuando se enciende o reinicia), el registro de instrucción se carga con una posición de memoria predefinida y la ejecución se inicia allí. En dicha posición se encuentra el programa inicial de arranque. Este programa se encuentra en **memoria de sólo lectura** (ROM, read-only memory), dado que la RAM se encuentra en un estado desconocido cuando se produce el arranque del sistema. La ROM sí resulta adecuada, ya que no necesita inicialización y no puede verse infectada por un virus informático.

El programa de arranque puede realizar diversas tareas. Normalmente, una de ellas consiste en ejecutar una serie de diagnósticos para determinar el estado de la máquina. Si se pasan las pruebas de diagnóstico satisfactoriamente, el programa puede continuar con la secuencia de arranque. También puede inicializar todos los aspectos del sistema, desde los registros de la CPU hasta los controladores de dispositivo y los contenidos de la memoria principal. Antes o después, se terminará por iniciar el sistema operativo.

Algunos sistemas, como los teléfonos móviles, los PDA y las consolas de juegos, almacenan todo el sistema operativo en ROM. El almacenamiento del sistema operativo en ROM resulta adecuado para sistemas operativos pequeños, hardware auxiliar sencillo y dispositivos que operen en entornos agresivos. Un problema con este método es que cambiar el código de arranque requiere cambiar los chips de la ROM. Algunos sistemas resuelven este problema usando una EPROM (erasable programmable read-only memory), que es una memoria de sólo lectura excepto cuando se le proporciona explícitamente un comando para hacer que se pueda escribir en ella. Todas las formas de ROM se conocen también como **firmware**, dado que tiene características intermedias entre las del hardware y las del software. Un problema general con el firmware es que ejecutar código en él es más lento que ejecutarlo en RAM. Algunos sistemas almacenan el sistema operativo en firmware y lo copian en RAM para conseguir una ejecución más rápida. Un último problema con el firmware es que es relativamente caro, por lo que normalmente sólo está disponible en pequeñas cantidades dentro de un sistema.

En los sistemas operativos de gran envergadura, incluyendo los de propósito general como Windows, Mac OS X y UNIX, o en los sistemas que cambian frecuentemente, el cargador de arranque se almacena en firmware y el sistema operativo en disco. En este caso, el programa de arranque ejecuta los diagnósticos y tiene un pequeño fragmento de código que puede leer un solo bloque que se encuentra en una posición fija (por ejemplo, el bloque cero) del disco, cargarlo en memoria y ejecutar el código que hay en dicho **bloque de arranque**. El programa almacenado en el bloque de arranque puede ser lo suficientemente complejo como para cargar el sistema operativo completo en memoria e iniciar su ejecución. Normalmente, se trata de un código simple (que cabe en un solo bloque de disco) y que únicamente conoce la dirección del disco y la longitud de resto del programa de arranque. Todo el programa de arranque escrito en disco y el propio sistema operativo pueden cambiarse fácilmente escribiendo nuevas versiones en disco. Un disco que tiene una partición de arranque (consulte la Sección 12.5.1) se denomina **disco de arranque** o **disco del sistema**.

Una vez que se ha cargado el programa de arranque completo, puede explorar el sistema de archivos para localizar el *kernel* del sistema operativo, cargarlo en memoria e iniciar su ejecución. Sólo en esta situación se dice que el sistema está en **ejecución**.

2.11 Resumen

Los sistemas operativos proporcionan una serie de servicios. En el nivel más bajo, las llamadas al sistema permiten que un programa en ejecución haga solicitudes directamente al sistema operativo. En un nivel superior, el intérprete de comandos o *shell* proporciona un mecanismo para que el usuario ejecute una solicitud sin escribir un programa. Los comandos pueden proceder de archivos de procesamiento por lotes o directamente de un terminal, cuando se está en modo interactivo o de tiempo compartido. Normalmente, se proporcionan programas del sistema para satisfacer muchas de las solicitudes más habituales de los usuarios.

Los tipos de solicitudes varían de acuerdo con el nivel. El nivel de gestión de las llamadas al sistema debe proporcionar funciones básicas, como las de control de procesos y de manipulación de archivos y dispositivos. Las solicitudes de nivel superior, satisfechas por el intérprete de comandos o los programas del sistema, se traducen a una secuencia de llamadas al sistema. Los servicios del sistema se pueden clasificar en varias categorías: control de programas, solicitudes de estado y solicitudes de E/S. Los errores de programa pueden considerarse como solicitudes implícitas de servicio.

Una vez que se han definido los servicios del sistema, se puede desarrollar la estructura del sistema. Son necesarias varias tablas para describir la información que define el estado del sistema informático y el de los trabajos que el sistema esté ejecutando.

El diseño de un sistema operativo nuevo es una tarea de gran envergadura. Es fundamental que los objetivos del sistema estén bien definidos antes de comenzar el diseño. El tipo de sistema deseado dictará las opciones que se elijan, entre los distintos algoritmos y estrategias necesarios.

Dado que un sistema operativo tiene una gran complejidad, la modularidad es importante. Dos técnicas adecuadas son diseñar el sistema como una secuencia de niveles o usando un *microkernel*. El concepto de máquina virtual se basa en una arquitectura en niveles y trata tanto al *kernel* del sistema operativo como al hardware como si fueran hardware. Incluso es posible cargar otros sistemas operativos por encima de esta máquina virtual.

A lo largo de todo el ciclo de diseño del sistema operativo debemos ser cuidadosos a la hora de separar las decisiones de política de los detalles de implementación (mecanismos). Esta separación permite conseguir la máxima flexibilidad si las decisiones de política se cambian con posterioridad.

Hoy en día, los sistemas operativos se escriben casi siempre en un lenguaje de implementación de sistemas o en un lenguaje de alto nivel. Este hecho facilita las tareas de implementación, mantenimiento y portabilidad. Para crear un sistema operativo para una determinada configuración de máquina, debemos llevar a cabo la generación del sistema.

Para que un sistema informático empiece a funcionar, la CPU debe inicializarse e iniciar la ejecución del programa de arranque implementado en firmware. El programa de arranque puede ejecutar directamente el sistema operativo si éste también está en el firmware, o puede completar una secuencia en la que progresivamente se cargan programas más inteligentes desde el firmware y el disco, hasta que el propio sistema operativo se carga en memoria y se ejecuta.

Ejercicios

- 2.1 Los servicios y funciones proporcionados por un sistema operativo pueden dividirse en dos categorías principales. Describa brevemente las dos categorías y explique en qué se diferencian.
- 2.2 Enumere cinco servicios proporcionados por un sistema operativo que estén diseñados para hacer que el uso del sistema informático sea más cómodo para el usuario. ¿En qué casos sería imposible que los programas de usuario proporcionaran estos servicios? Explique su respuesta.
- 2.3 Describa tres métodos generales para pasar parámetros al sistema operativo.

- 2.4 Describa cómo se puede obtener un perfil estadístico de la cantidad de tiempo invertido por un programa en la ejecución de las diferentes secciones de código. Explique la importancia de obtener tal perfil estadístico.
- 2.5 ¿Cuáles son las cinco principales actividades de un sistema operativo en lo que se refiere a la administración de archivos?
- 2.6 ¿Cuáles son las ventajas y desventajas de usar la misma interfaz de llamadas al sistema tanto para la manipulación de archivos como de dispositivos?
- 2.7 ¿Cuál es el propósito del intérprete de comandos? ¿Por qué está normalmente separado del *kernel*? ¿Sería posible que el usuario desarrollara un nuevo intérprete de comandos utilizando la interfaz de llamadas al sistema proporcionada por el sistema operativo?
- 2.8 ¿Cuáles son los dos modelos de comunicación interprocesos? ¿Cuáles son las ventajas y desventajas de ambos métodos?
- 2.9 ¿Por qué es deseable separar los mecanismos de las políticas?
- 2.10 ¿Por qué Java proporciona la capacidad de llamar desde un programa Java a métodos nativos que estén escritos en, por ejemplo, C o C++? Proporcione un ejemplo de una situación en la que sea útil emplear un método nativo.
- 2.11 En ocasiones, es difícil definir un modelo en niveles si dos componentes del sistema operativo dependen el uno del otro. Describa un escenario en el no esté claro cómo separar en niveles dos componentes del sistema que requieran un estrecho acoplamiento de su respectiva funcionalidad.
- 2.12 ¿Cuál es la principal ventaja de usar un *microkernel* en el diseño de sistemas? ¿Cómo interactúan los programas de usuario y los servicios del sistema en una arquitectura basada en *microkernel*? ¿Cuáles son las desventajas de usar la arquitectura de *microkernel*?
- 2.13 ¿En qué se asemejan la arquitectura de *kernel* modular y la arquitectura en niveles? ¿En qué se diferencian?
- 2.14 ¿Cuál es la principal ventaja, para un diseñador de sistemas operativos, de usar una arquitectura de máquina virtual? ¿Cuál es la principal ventaja para el usuario?
- 2.15 ¿Por qué es útil un compilador just-in-time para ejecutar programas Java?
- 2.16 ¿Cuál es la relación entre un sistema operativo huésped y un sistema operativo *host* en un sistema como VMware? ¿Qué factores hay que tener en cuenta al seleccionar el sistema operativo *host*?
- 2.17 El sistema operativo experimental Synthesis dispone de un ensamblador incorporado en el *kernel*. Para optimizar el rendimiento de las llamadas al sistema, el *kernel* ensambla las rutinas dentro del espacio del *kernel* para minimizar la ruta de ejecución que debe seguir la llamada al sistema dentro del *kernel*. Este método es la antítesis del método por niveles, en el que la ruta a través del *kernel* se complica para poder construir más fácilmente el sistema operativo. Explique las ventajas e inconvenientes del método de Synthesis para el diseño del *kernel* y la optimización del rendimiento del sistema.
- 2.18 En la Sección 2.3 hemos descrito un programa que copia el contenido de un archivo en otro archivo de destino. Este programa pide en primer lugar al usuario que introduzca el nombre de los archivos de origen y de destino. Escriba dicho programa usando la API Win32 o la API POSIX. Asegúrese de incluir todas las comprobaciones de error necesarias, incluyendo asegurarse de que el archivo de origen existe. Una vez que haya diseñado y probado correctamente el programa, ejecútelo empleando una utilidad que permita trazar las llamadas al sistema, si es que su sistema soporta dicha funcionalidad. Los sistemas Linux proporcionan la utilidad *ptrace* y los sistemas Solaris ofrecen los comandos *truss* o *dtrace*. En Mac OS X, la facilidad *ktrace* proporciona una funcionalidad similar.

Proyecto: adición de una llamada al sistema al kernel de Linux

En este proyecto, estudiaremos la interfaz de llamadas al sistema proporcionada por el sistema operativo Linux y veremos cómo se comunican los programas de usuario con el *kernel* del sistema operativo a través de esta interfaz. Nuestra tarea consiste en incorporar una nueva llamada al sistema dentro del *kernel*, expandiendo la funcionalidad del sistema operativo.

Introducción

Una llamada a procedimiento en modo usuario se realiza pasando argumentos al procedimiento invocado, bien a través de la pila o a través de registros, guardando el estado actual y el valor del contador de programa, y saltando al principio del código correspondiente al procedimiento invocado. El proceso continúa teniendo los mismos privilegios que antes.

Los programas de usuario ven las llamadas al sistema como llamadas a procedimientos, pero estas llamadas dan lugar a un cambio en los privilegios y en el contexto de ejecución. En Linux sobre una arquitectura 386 de Intel, una llamada al sistema se realiza almacenando el número de llamada al sistema en el registro EAX, almacenando los argumentos para la llamada al sistema en otros registros hardware y ejecutando una excepción (que es la instrucción de ensamblador INT 0x80). Despues de ejecutar la excepción, se utiliza el número de llamada al sistema como índice para una tabla de punteros de código, con el fin de obtener la dirección de comienzo del código de tratamiento que implementa la llamada al sistema. El proceso salta luego a esta dirección y los privilegios del proceso se intercambian del modo usuario al modo *kernel*. Con los privilegios ampliados, el proceso puede ahora ejecutar código del *kernel* que puede incluir instrucciones privilegiadas, las cuales no se pueden ejecutar en modo usuario. El código del *kernel* puede entonces llevar a cabo los servicios solicitados, como por ejemplo interactuar con dispositivos de E/S, realizar la gestión de procesos y otras actividades que no pueden llevarse a cabo en modo usuario.

Los números de las llamadas al sistema para las versiones recientes del *kernel* de Linux se enumeran en /usr/src/linux-2.x/include/asm-i386/unistd.h. Por ejemplo, __NR_close, que corresponde a la llamada al sistema close(), la cual se invoca para cerrar un descriptor de archivo, tiene el valor 6. La lista de punteros a los descriptores de las llamadas al sistema se almacena normalmente en el archivo /usr/src/linux-2.x/arch/i386/kernel_entry.S bajo la cabecera ENTRY(sys\call\table). Observe que sys_close está almacenada en la entrada numerada como 6 en la tabla, para ser coherente con el número de llamada al sistema definido en el archivo unistd.h. La palabra clave .long indica que la entrada ocupará el mismo número de bytes que un valor de datos de tipo long.

Construcción de un nuevo kernel

Antes de añadir al *kernel* una llamada al sistema, debe familiarizarse con la tarea de construir el binario de un *kernel* a partir de su código fuente y reiniciar la máquina con el nuevo *kernel* creado. Esta actividad comprende las siguientes tareas, siendo algunas de ellas dependientes de la instalación concreta del sistema operativo Linux de la que se disponga:

- Obtener el código fuente del *kernel* de la distribución de Linux. Si el paquete de código fuente ha sido previamente instalado en su máquina, los archivos correspondientes se encontrarán en /usr/src/linux o /usr/src/linux-2.x (donde el sufijo corresponde al numero de versión del *kernel*). Si el paquete no ha sido instalado, puede descargarlo del proveedor de su distribución de Linux o en <http://www.kernel.org>.
- Aprenda a configurar, compilar e instalar el binario del *kernel*. Esta operación variará entre las diferentes distribuciones del *kernel*, aunque algunos comandos típicos para la creación del *kernel* (después de situarse en el directorio donde se almacena el código fuente del *kernel*) son:
 - make xconfig
 - make dep

- make bzImage
- Añada una nueva entrada al conjunto de *kernels* de arranque soportados por el sistema. El sistema operativo Linux usa normalmente utilidades como lilo y grub para mantener una lista de kernels de arranque, de entre los cuales el usuario puede elegir durante el proceso de arranque de la máquina. Si su sistema soporta lilo, añada una entrada como la siguiente a lilo.conf:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

donde /boot/bzImage.mykernel es la imagen del *kernel* y mykernel es la etiqueta asociada al nuevo *kernel*, que nos permite seleccionarlo durante el proceso de arranque. Realizando este paso, tendremos la opción de arrancar un nuevo *kernel* o el *kernel* no modificado, por si acaso el *kernel* recién creado no funciona correctamente.

Ampliación del código fuente del *kernel*

Ahora puede experimentar añadiendo un nuevo archivo al conjunto de archivos fuente utilizados para compilar el *kernel*. Normalmente, el código fuente se almacena en el directorio /usr/src/linux-2.x/kernel, aunque dicha ubicación puede ser distinta en su distribución Linux. Tenemos dos opciones para añadir la llamada al sistema. La primera consiste en añadir la llamada al sistema a un archivo fuente existente en ese directorio. La segunda opción consiste en crear un nuevo archivo en el directorio fuente y modificar /usr/src/linux-2.x/kernel/Makefile para incluir el archivo recién creado en el proceso de compilación. La ventaja de la primera opción es que, modificando un archivo existente que ya forma parte del proceso de compilación, Makefile no requiere modificación.

Adición al *kernel* de una llamada al sistema

Ahora que ya está familiarizado con las distintas tareas básicas requeridas para la creación y arranque de *kernels* de Linux, puede empezar el proceso de añadir al *kernel* de Linux una nueva llamada al sistema. En este proyecto, la llamada al sistema tendrá una funcionalidad limitada: simplemente hará la transición de modo usuario a modo *kernel*, presentará un mensaje que se registrará junto con los mensajes del *kernel* y volverá al modo usuario. Llamaremos a esta llamada al sistema *helloworld*. De todos modos, aunque el ejemplo tenga una funcionalidad limitada, ilustra el mecanismo de las llamadas al sistema y arroja luz sobre la interacción entre los programas de usuario y el *kernel*.

- Cree un nuevo archivo denominado helloworld.c para definir su llamada al sistema. Incluya los archivos de cabecera linux/linkage.h y linux/kernel.h. Añada el siguiente código al archivo:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
    printk(KERN_EMERG "hello world!");
    return 1;
}
```

Esto crea un llamada al sistema con el nombre sys_helloworld. Si elige añadir esta llamada al sistema a un archivo existente en el directorio fuente, todo lo que tiene que hacer es añadir la función sys_helloworld() al archivo que elija. asmlinkage es un remanen-

te de los días en que Linux usaba código C++ y C, y se emplea para indicar que el código está escrito en C. La función `printf()` se usa para escribir mensajes en un archivo de registro del *kernel* y, por tanto, sólo puede llamarse desde el *kernel*. Los mensajes del *kernel* especificados en el parámetro `printf()` se registran en el archivo `/var/log/kernel/warnings`. El prototipo de función para la llamada `printf()` está definido en `/usr/include/linux/kernel.h`.

- Defina un nuevo número de llamada al sistema para `__NR_helloworld` en `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Los programas de usuario pueden emplear este número para identificar la nueva llamada al sistema que hemos añadido. También debe asegurarse de incrementar el valor de `__NR_syscalls`, que también se almacena en el mismo archivo. Esta constante indica el número de llamadas al sistema actualmente definidas en el *kernel*.
- Añada una entrada `.long sys_helloworld` a la tabla `sys_call_table` definida en el archivo `/usr/src/linux-2.x/arch/i386/kernel/entry.S`. Como se ha explicado anteriormente, el número de llamada al sistema se usa para indexar esta tabla, con el fin de poder localizar la posición del código de tratamiento de la llamada al sistema que se invoque.
- Añada su archivo `helloworld.c` a `Makefile` (si ha creado un nuevo archivo para su llamada al sistema). Guarde una copia de la imagen binaria de su antiguo *kernel* (por si acaso tiene problemas con el nuevo). Ahora puede crear el nuevo *kernel*, cambiarlo de nombre para diferenciarlo del *kernel* no modificado y añadir una entrada a los archivos de configuración del cargador (como por ejemplo `lilo.conf`). Después de completar estos pasos, puede arrancar el antiguo *kernel* o el nuevo, que contendrá la nueva llamada al sistema.

Uso de la llamada al sistema desde un programa de usuario

Cuando arranque con el nuevo *kernel*, la nueva llamada al sistema estará habilitada; ahora simplemente es cuestión de invocarla desde un programa de usuario. Normalmente, la biblioteca C estándar soporta una interfaz para llamadas al sistema definida para el sistema operativo Linux. Como la nueva llamada al sistema no está montada con la biblioteca estándar C, invocar la llamada al sistema requerirá una cierta intervención manual.

Como se ha comentado anteriormente, una llamada al sistema se invoca almacenando el valor apropiado en un registro hardware y ejecutando una instrucción de excepción. Lamentablemente, éstas son operaciones de bajo nivel que no pueden ser realizadas usando instrucciones en lenguaje C, requiriéndose, en su lugar, instrucciones de ensamblador. Afortunadamente, Linux proporciona macros para instanciar funciones envoltorio que contienen las instrucciones de ensamblador apropiadas. Por ejemplo, el siguiente programa C usa la macro `_syscall0()` para invocar la nueva llamada al sistema:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
    helloworld();
}
```

- La macro `_syscall0` toma dos argumentos. El primero especifica el tipo del valor devuelto por la llamada del sistema, mientras que el segundo argumento es el nombre de la llamada al sistema. El nombre se usa para identificar el número de llamada al sistema, que se

almacena en el registro hardware antes de que se ejecute la excepción. Si la llamada al sistema requiriera argumentos, entonces podría usarse una macro diferente (tal como `_syscall10`, donde el sufijo indica el número de argumentos) para instanciar el código ensamblador requerido para realizar la llamada al sistema.

- Compile y ejecute el programa con el *kernel* recién creado. En el archivo de registro del *kernel* `/var/log/kernel/warnings` deberá aparecer un mensaje “hello world!” para indicar que la llamada al sistema se ha ejecutado.

Como paso siguiente, considere expandir la funcionalidad de su llamada al sistema. ¿Cómo pasaría un valor entero o una cadena de caracteres a la llamada al sistema y lo escribiría en el archivo del registro del *kernel*? ¿Cuáles son las implicaciones de pasar punteros a datos almacenados en el espacio de direcciones del programa de usuario, por contraposición a pasar simplemente un valor entero desde el programa de usuario al *kernel* usando registros hardware?

Notas bibliográficas

Dijkstra [1968] recomienda el modelo de niveles para el diseño de sistemas operativos. Brinch-Hansen [1970] fue uno de los primeros defensores de construir un sistema operativo como un *kernel* (o núcleo) sobre el que pueden construirse sistemas más completos.

Las herramientas del sistema y el trazado dinámico se describen en Tamches y Miller [1999]. DTrace se expone en Cantrill et al. [2004]. Cheung y Loong [1995] exploran diferentes temas sobre la estructura de los sistemas operativos, desde los *microkernels* hasta los sistemas extensibles.

MS-DOS, versión 3.1, se describe en Microsoft [1986]. Windows NT y Windows 2000 se describen en Solomon [1998] y Solomon y Russinovich [2000]. BSD UNIX se describe en Mckusick et al. [1996]. Bovet y Cesati [2002] cubren en detalle el *kernel* de Linux. Varios sistemas UNIX, incluido Mach, se tratan en detalle en Vahalia [1996]. Mac OS X se presenta en <http://www.apple.com/macosx>. El sistema operativo experimental Synthesis se expone en Masalin y Pu [1989]. Solaris se describe de forma completa en Mauro y McDougall [2001].

El primer sistema operativo que proporcionó una máquina virtual fue el CP 67 en un IBM 360/67. El sistema operativo IBM VM/370 comercialmente disponible era un derivado de CP 67. Detalles relativos a Mach, un sistema operativo basado en *microkernel*, pueden encontrarse en Young et al. [1987]. Kaashoek et al [1997] presenta detalles sobre los sistemas operativos con exo-kernel, donde la arquitectura separa los problemas de administración de los de protección, proporcionando así al software que no sea de confianza la capacidad de ejercer control sobre los recursos hardware y software.

Las especificaciones del lenguaje Java y de la máquina virtual Java se presentan en Gosling et al. [1996] y Lindholm y Yellin [1999], respectivamente. El funcionamiento interno de la máquina virtual Java se describe de forma completa en Venners [1998]. Golm et al [2002] destaca el sistema operativo JX; Back et al. [2000] cubre varios problemas del diseño de los sistemas operativos Java. Hay disponible más información sobre Java en la web <http://www.javasoftware.com>. Puede encontrar detalles sobre la implementación de VMware en Sugerman et al. [2001].

Parte Dos

Gestión de procesos

Puede pensarse en un *proceso* como en un programa en ejecución. Un proceso necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos de E/S para llevar a cabo su tarea. Estos recursos se asignan al proceso en el momento de crearlo o en el de ejecutarlo.

En la mayoría de los sistemas, la unidad de trabajo son los procesos. Los sistemas constan de una colección de procesos: los procesos del sistema operativo ejecutan código del sistema y los procesos de usuario ejecutan código de usuario. Todos estos procesos pueden ejecutarse de forma concurrente.

Aunque tradicionalmente los procesos se ejecutaban utilizando una sola *hebra* de control, ahora la mayoría de los sistemas operativos modernos permiten ejecutar procesos compuestos por múltiples hebras.

El sistema operativo es responsable de las actividades relacionadas con la gestión de procesos y hebras: la creación y eliminación de procesos del sistema y de usuario; la planificación de los procesos y la provisión de mecanismos para la sincronización, la comunicación y el tratamiento de interbloqueos en los procesos.

Procesos

Los primeros sistemas informáticos sólo permitían que se ejecutara un programa a la vez. Este programa tenía el control completo del sistema y tenía acceso a todos los recursos del mismo. Por el contrario, los sistemas informáticos actuales permiten que se carguen en memoria múltiples programas y se ejecuten concurrentemente. Esta evolución requiere un mayor control y aislamiento de los distintos programas y estas necesidades dieron lugar al concepto de **proceso**, que es un programa en ejecución. Un proceso es la unidad de trabajo en los sistemas modernos de tiempo compartido.

Cuanto más complejo es el sistema operativo, más se espera que haga en nombre de sus usuarios. Aunque su principal cometido es ejecutar programas de usuario, también tiene que ocuparse de diversas tareas del sistema que, por uno u otro motivo, no están incluidas dentro del *kernel*. Por tanto, un sistema está formado por una colección de procesos: procesos del sistema operativo que ejecutan código del sistema y procesos de usuario que ejecutan código de usuario. Potencialmente, todos estos procesos pueden ejecutarse concurrentemente, multiplexando la CPU (o las distintas CPU) entre ellos. Cambiando la asignación de la CPU entre los distintos procesos, el sistema operativo puede incrementar la productividad de la computadora.

OBJETIVOS DEL CAPÍTULO

- Presentar el concepto de proceso (un programa en ejecución), en el que se basa todo el funcionamiento de un sistema informático.
- Describir los diversos mecanismos relacionados con los procesos, incluyendo los de planificación, creación y finalización de procesos, y los mecanismos de comunicación.
- Describir los mecanismos de comunicación en los sistemas cliente-servidor.

3.1 Concepto de proceso

Una pregunta que surge cuando se estudian los sistemas operativos es cómo llamar a las diversas actividades de la CPU. Los sistemas de procesamiento por lotes ejecutan *trabajos*, mientras que un sistema de tiempo compartido tiene *programas de usuario* o *tareas*. Incluso en un sistema monousouario, como Microsoft Windows, el usuario puede ejecutar varios programas al mismo tiempo: un procesador de textos, un explorador web y un programa de correo electrónico. Incluso aunque el usuario pueda ejecutar sólo un programa cada vez, el sistema operativo puede tener que dar soporte a sus propias actividades internas programadas, como los mecanismos de gestión de la memoria. En muchos aspectos, todas estas actividades son similares, por lo que a todas ellas las denominamos *procesos*.

En este texto, los términos *trabajo* y *proceso* se usan indistintamente. Aunque personalmente preferimos el término *proceso*, gran parte de la teoría y terminología de los sistemas operativos se

desarrolló durante una época en que la principal actividad de los sistemas operativos era el procesamiento de trabajos por lotes. Podría resultar confuso, por tanto, evitar la utilización de aquellos términos comúnmente aceptados que incluyen la palabra *trabajo* (como por ejemplo *planificación de trabajos*) simplemente porque el término *proceso* haya sustituido a *trabajo*.

3.1.1 El proceso

Informalmente, como hemos indicado antes, un proceso es un programa en ejecución. Hay que resaltar que un proceso es algo más que el código de un programa (al que en ocasiones se denomina **sección de texto**). Además del código, un proceso incluye también la actividad actual, que queda representada por el valor del **contador de programa** y por los contenidos de los registros del procesador. Generalmente, un proceso incluye también la **pila** del proceso, que contiene datos temporales (como los parámetros de las funciones, las direcciones de retorno y las variables locales), y una **sección de datos**, que contiene las variables globales. El proceso puede incluir, asimismo, un **cúmulo de memoria**, que es la memoria que se asigna dinámicamente al proceso en tiempo de ejecución. En la Figura 3.1 se muestra la estructura de un proceso en memoria.

Insistamos en que un programa, por sí mismo, no es un proceso; un programa es una entidad *pasiva*, un archivo que contiene una lista de instrucciones almacenadas en disco (a menudo denominado **archivo ejecutable**), mientras que un proceso es una entidad *activa*, con un contador de programa que especifica la siguiente instrucción que hay que ejecutar y un conjunto de recursos asociados. Un programa se convierte en un proceso cuando se carga en memoria un archivo ejecutable. Dos técnicas habituales para cargar archivos ejecutables son: hacer doble clic sobre un ícono que represente el archivo ejecutable e introducir el nombre del archivo ejecutable en la línea de comandos (como por ejemplo, prog.exe o a.out.)

Aunque puede haber dos procesos asociados con el mismo programa, esos procesos se consideran dos secuencias de ejecución separadas. Por ejemplo, varios usuarios pueden estar ejecutando copias diferentes del programa de correo, o el mismo usuario puede invocar muchas copias del explorador web. Cada una de estas copias es un proceso distinto y, aunque las secciones de texto sean equivalentes, las secciones de datos, del cúmulo (*heap*) de memoria y de la pila variarán de unos procesos a otros. También es habitual que un proceso cree muchos otros procesos a medida que se ejecuta. En la Sección 3.4 se explican estas cuestiones.

3.1.2 Estado del proceso

A medida que se ejecuta un proceso, el proceso va cambiando de **estado**. El estado de un proceso se define, en parte, según la actividad actual de dicho proceso. Cada proceso puede estar en uno de los estados siguientes:

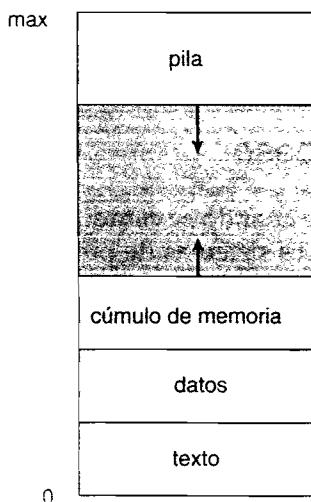


Figura 3.1 Proceso en memoria.

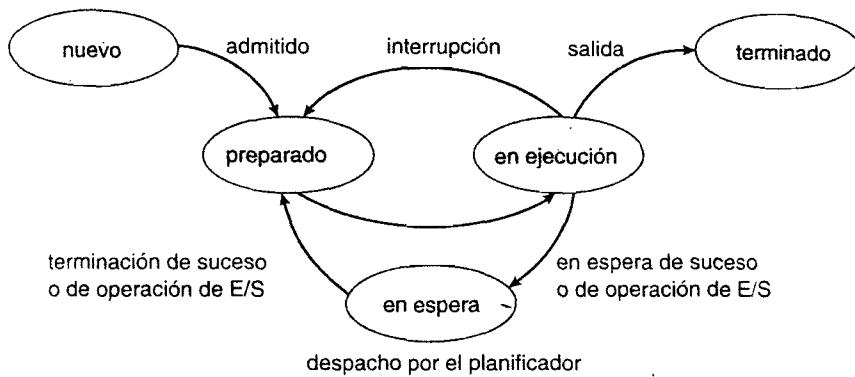


Figura 3.2 Diagrama de estados de un proceso.

- **Nuevo.** El proceso está siendo creado.
- **En ejecución.** Se están ejecutando las instrucciones.
- **En espera.** El proceso está esperando a que se produzca un suceso (como la terminación de una operación de E/S o la recepción de una señal).
- **Preparado.** El proceso está a la espera de que le asignen a un procesador.
- **Terminado.** Ha terminado la ejecución del proceso.

Estos nombres son arbitrarios y varían de un sistema operativo a otro. Sin embargo, los estados que representan se encuentran en todos los sistemas. Determinados sistemas operativos definen los estados de los procesos de forma más específica. Es importante darse cuenta de que sólo puede haber un proceso *ejecutándose* en cualquier procesador en cada instante concreto. Sin embargo, puede haber muchos procesos *preparados* y *en espera*. En la Figura 3.2 se muestra el diagrama de estados de un proceso genérico.

3.1.3 Bloque de control de proceso

Cada proceso se representa en el sistema operativo mediante un **bloque de control de proceso** (PCB, process control block), también denominado *bloque de control de tarea* (véase la Figura 3.3). Un bloque de control de proceso contiene muchos elementos de información asociados con un proceso específico, entre los que se incluyen:

- **Estado del proceso.** El estado puede ser: nuevo, preparado, en ejecución, en espera, detenido, etc.
- **Contador de programa.** El contador indica la dirección de la siguiente instrucción que va a ejecutar dicho proceso.
- **Registros de la CPU.** Los registros varían en cuanto a número y tipo, dependiendo de la arquitectura de la computadora. Incluyen los acumuladores, registros de índice, punteros de pila y registros de propósito general, además de toda la información de los indicadores de estado. Esta información de estado debe guardarse junto con el contador de programa cuando se produce una interrupción, para que luego el proceso pueda continuar ejecutándose correctamente (Figura 3.4).
- **Información de planificación de la CPU.** Esta información incluye la prioridad del proceso, los punteros a las colas de planificación y cualesquiera otros parámetros de planificación que se requieran. El Capítulo 5 describe los mecanismos de planificación de procesos.
- **Información de gestión de memoria.** Incluye información acerca del valor de los registros base y límite, las tablas de páginas o las tablas de segmentos, dependiendo del mecanismo de gestión de memoria utilizado por el sistema operativo (Capítulo 8).

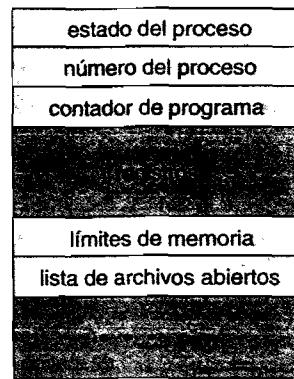


Figura 3.3 Bloque de control de proceso (PCB).

- **Información contable.** Esta información incluye la cantidad de CPU y de tiempo real empleados, los límites de tiempo asignados, los números de cuenta, el número de trabajo o de proceso, etc.
- **Información del estado de E/S.** Esta información incluye la lista de los dispositivos de E/S asignados al proceso, una lista de los archivos abiertos, etc.

En resumen, el PCB sirve simplemente como repositorio de cualquier información que pueda variar de un proceso a otro.

3.1.4 Hebras

El modelo de proceso que hemos visto hasta ahora implicaba que un proceso es un programa que tiene una sola **hebra** de ejecución. Por ejemplo, cuando un proceso está ejecutando un procesador de textos, se ejecuta una sola hebra de instrucciones. Esta única hebra de control permite al proceso realizar sólo una tarea cada vez. Por ejemplo, el usuario no puede escribir simultáneamente

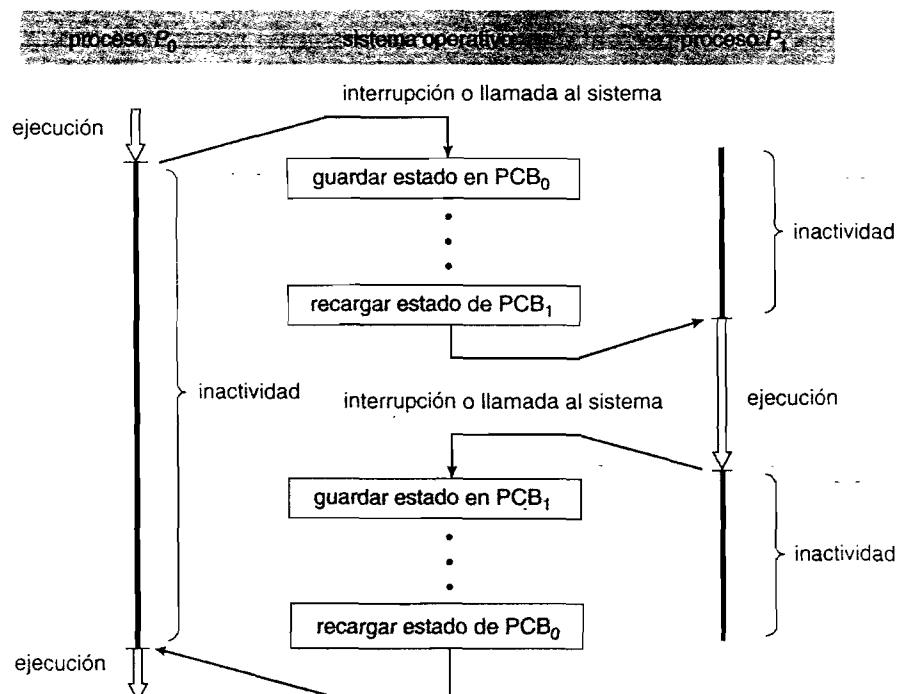


Figura 3.4 Diagrama que muestra la comutación de la CPU de un proceso a otro.

cesos selecciona un proceso disponible (posiblemente de entre un conjunto de varios procesos disponibles) para ejecutar el programa en la CPU. En los sistemas de un solo procesador, nunca habrá más de un proceso en ejecución: si hay más procesos, tendrán que esperar hasta que la CPU esté libre y se pueda asignar a otro proceso.

3.2.1 Colas de planificación

A medida que los procesos entran en el sistema, se colocan en una **cola de trabajos** que contiene todos los procesos del sistema. Los procesos que residen en la memoria principal y están preparados y en espera de ejecutarse se mantienen en una lista denominada **cola de procesos preparados**. Generalmente, esta cola se almacena en forma de lista enlazada. La cabecera de la cola de procesos preparados contiene punteros al primer y último bloques de control de procesos (PCB) de la lista. Cada PCB incluye un campo de puntero que apunta al siguiente PCB de la cola de procesos preparados.

El sistema también incluye otras colas. Cuando se asigna la CPU a un proceso, éste se ejecuta durante un rato y finalmente termina, es interrumpido o espera a que se produzca un determinado suceso, como la terminación de una solicitud de E/S. Suponga que el proceso hace una solicitud de E/S a un dispositivo compartido, como por ejemplo un disco. Dado que hay muchos procesos en el sistema, el disco puede estar ocupado con la solicitud de E/S de algún otro proceso. Por tanto, nuestro proceso puede tener que esperar para poder acceder al disco. La lista de procesos en espera de un determinado dispositivo de E/S se denomina **cola del dispositivo**. Cada dispositivo tiene su propia cola (Figura 3.6).

Una representación que habitualmente se emplea para explicar la planificación de procesos es el **diagrama de colas**, como el mostrado en la Figura 3.7, donde cada rectángulo representa una cola. Hay dos tipos de colas: la cola de procesos preparados y un conjunto de colas de dispositivo. Los círculos representan los recursos que dan servicio a las colas y las flechas indican el flujo de procesos en el sistema.

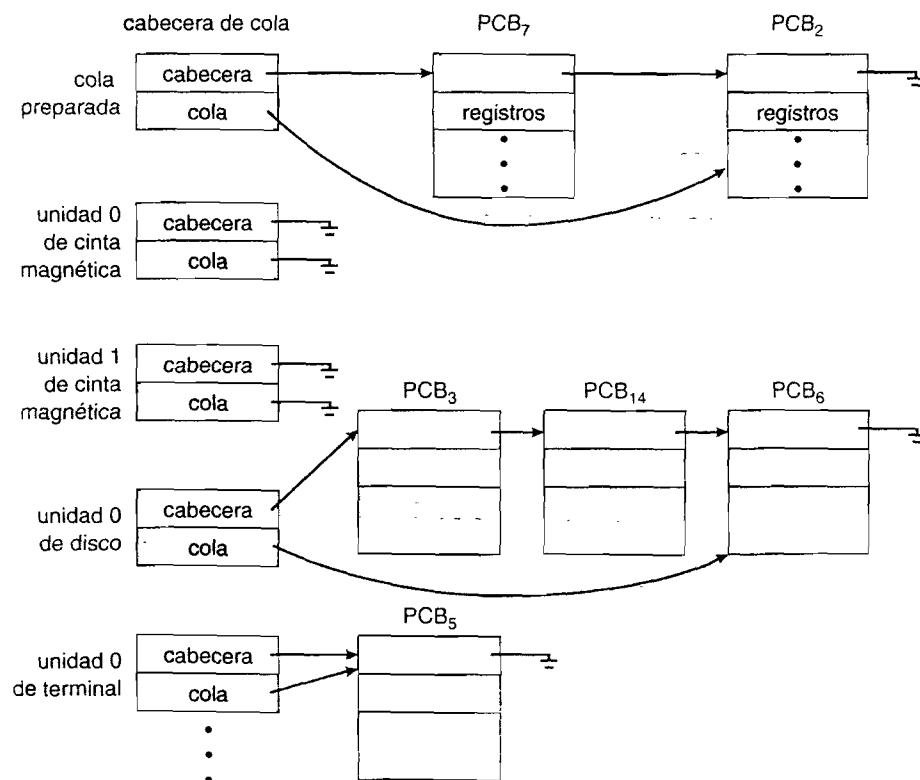


Figura 3.6 Cola de procesos preparados y diversas colas de dispositivos de E/S.

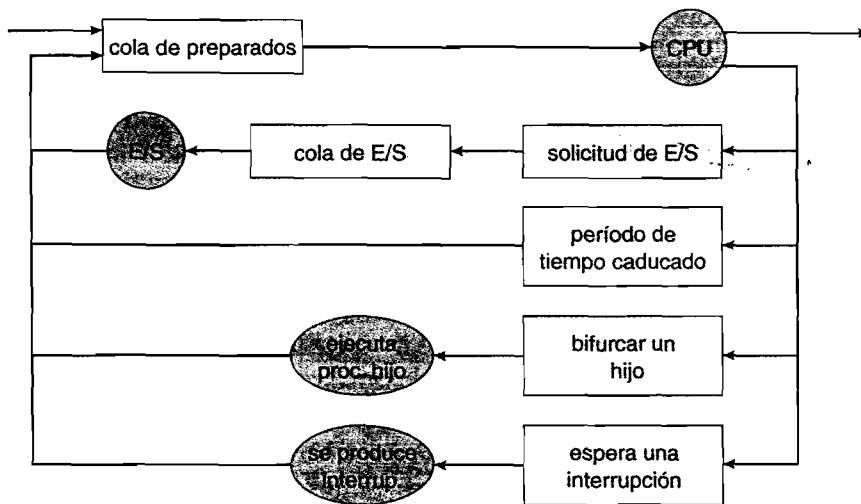


Figura 3.7 Diagrama de colas para la planificación de procesos.

Cada proceso nuevo se coloca inicialmente en la cola de procesos preparados, donde espera hasta que es seleccionado para ejecución, es decir, hasta que es **despachado**. Una vez que se asigna la CPU al proceso y éste comienza a ejecutarse, se puede producir uno de los sucesos siguientes:

- El proceso podría ejecutar una solicitud de E/S y ser colocado, como consecuencia, en una cola de E/S.
- El proceso podría crear un nuevo subprocesso y esperar a que éste termine.
- El proceso podría ser desalojado de la CPU como resultado de una interrupción y puesto de nuevo en la cola de procesos preparados.

En los dos primeros casos, el proceso terminará, antes o después, por cambiar del estado de espera al estado preparado y será colocado de nuevo en la cola de procesos preparados. Los procesos siguen este ciclo hasta que termina su ejecución, momento en el que se elimina el proceso de todas las colas y se desasignan su PCB y sus recursos.

3.2.2 Planificadores

Durante su tiempo de vida, los procesos se mueven entre las diversas colas de planificación. El sistema operativo, como parte de la tarea de planificación, debe seleccionar de alguna manera los procesos que se encuentran en estas colas. El proceso de selección se realiza mediante un **planificador** apropiado.

A menudo, en un sistema de procesamiento por lotes, se envían más procesos de los que pueden ser ejecutados de forma inmediata. Estos procesos se guardan en cola en un dispositivo de almacenamiento masivo (normalmente, un disco), donde se mantienen para su posterior ejecución. El **planificador a largo plazo** o **planificador de trabajos** selecciona procesos de esta cola y los carga en memoria para su ejecución. El **planificador a corto plazo** o **planificador de la CPU** selecciona de entre los procesos que ya están preparados para ser ejecutados y asigna la CPU a uno de ellos.

La principal diferencia entre estos dos planificadores se encuentra en la frecuencia de ejecución. El planificador a corto plazo debe seleccionar un nuevo proceso para la CPU frecuentemente. Un proceso puede ejecutarse sólo durante unos pocos milisegundos antes de tener que esperar por una solicitud de E/S. Normalmente, el planificador a corto plazo se ejecuta al menos una vez cada 100 milisegundos. Debido al poco tiempo que hay entre ejecuciones, el planificador a corto plazo debe ser rápido. Si tarda 10 milisegundos en decidir ejecutar un proceso durante 100 milisegundos, entonces el $10/(100 + 10) = 9$ por ciento del tiempo de CPU se está usando (perdiendo) simplemente para planificar el trabajo.

El planificador a largo plazo se ejecuta mucho menos frecuentemente; pueden pasar minutos entre la creación de un nuevo proceso y el siguiente. El planificador a largo plazo controla el **grado de multiprogramación** (el número de procesos en memoria). Si el grado de multiprogramación es estable, entonces la tasa promedio de creación de procesos debe ser igual a la tasa promedio de salida de procesos del sistema. Por tanto, el planificador a largo plazo puede tener que invocarse sólo cuando un proceso abandona el sistema. Puesto que el intervalo entre ejecuciones es más largo, el planificador a largo plazo puede permitirse emplear más tiempo en decidir qué proceso debe seleccionarse para ser ejecutado.

Es importante que el planificador a largo plazo haga una elección cuidadosa. En general, la mayoría de los procesos pueden describirse como limitados por la E/S o limitados por la CPU. Un **proceso limitado por E/S** es aquel que invierte la mayor parte de su tiempo en operaciones de E/S en lugar de en realizar cálculos. Por el contrario, un **proceso limitado por la CPU** genera solicitudes de E/S con poca frecuencia, usando la mayor parte de su tiempo en realizar cálculos. Es importante que el planificador a largo plazo seleccione una adecuada **mezcla de procesos**, equilibrando los procesos limitados por E/S y los procesos limitados por la CPU. Si todos los procesos son limitados por la E/S, la cola de procesos preparados casi siempre estará vacía y el planificador a corto plazo tendrá poco que hacer. Si todos los procesos son limitados por la CPU, la cola de espera de E/S casi siempre estará vacía, los dispositivos apenas se usarán, y de nuevo el sistema se desequilibrará. Para obtener un mejor rendimiento, el sistema dispondrá entonces de una combinación equilibrada de procesos limitados por la CPU y de procesos limitados por E/S.

En algunos sistemas, el planificador a largo plazo puede no existir o ser mínimo. Por ejemplo, los sistemas de tiempo compartido, tales como UNIX y los sistemas Microsoft Windows, a menudo no disponen de planificador a largo plazo, sino que simplemente ponen todos los procesos nuevos en memoria para que los gestione el planificador a corto plazo. La estabilidad de estos sistemas depende bien de una limitación física (tal como el número de terminales disponibles), bien de la propia naturaleza autoajustable de las personas que utilizan el sistema. Si el rendimiento descende a niveles inaceptables en un sistema multiusuario, algunos usuarios simplemente lo abandonarán.

Algunos sistemas operativos, como los sistemas de tiempo compartido, pueden introducir un nivel intermedio adicional de planificación; en la Figura 3.8 se muestra este planificador. La idea clave subyacente a un planificador a medio plazo es que, en ocasiones, puede ser ventajoso eliminar procesos de la memoria (con lo que dejan de contender por la CPU) y reducir así el grado de multiprogramación. Después, el proceso puede volver a cargarse en memoria, continuando su ejecución en el punto en que se interrumpió. Este esquema se denomina **intercambio**. El planificador a medio plazo descarga y luego vuelve a cargar el proceso. El intercambio puede ser necesario para mejorar la mezcla de procesos o porque un cambio en los requisitos de memoria haya hecho que se sobrepase la memoria disponible, requiriendo que se libere memoria. En el Capítulo 8 se estudian los mecanismos de intercambio.

3.2.3 Cambio de contexto

Como se ha mencionado en la Sección 1.2.1, las interrupciones hacen que el sistema operativo obligue a la CPU a abandonar su tarea actual, para ejecutar una rutina del *kernel*. Estos sucesos se producen con frecuencia en los sistemas de propósito general. Cuando se produce una interrupción el sistema tiene que guardar el **contexto** actual del proceso que se está ejecutando en la CPU, de modo que pueda restaurar dicho contexto cuando su procesamiento concluya, suspendiendo el proceso y reanudándolo después. El contexto se almacena en el PCB del proceso e incluye el valor de los registros de la CPU, el estado del proceso (véase la Figura 3.2) y la información de gestión de memoria. Es decir, realizamos una **salvaguarda del estado** actual de la CPU, en modo *kernel* (en modo usuario, y una **restauración del estado** para reanudar las operaciones).

La commutación de la CPU a otro proceso requiere una salvaguarda del estado del proceso actual y una restauración del estado de otro proceso diferente. Esta tarea se conoce como **cambio de contexto**. Cuando se produce un cambio de contexto, el *kernel* guarda el contexto del proceso antiguo en su PCB y carga el contexto almacenado del nuevo proceso que se ha decidido ejecutar.

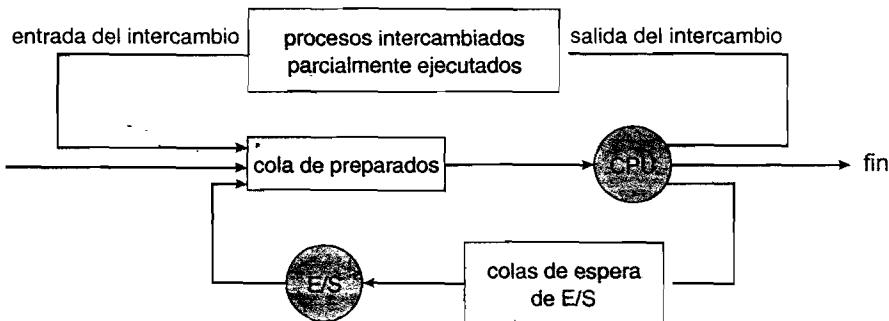


Figura 3.8 Adición de mecanismos de planificación a medio plazo en el diagrama de colas.

El tiempo dedicado al cambio de contexto es tiempo desperdiciado, dado que el sistema no realiza ningún trabajo útil durante la comutación. La velocidad del cambio de contexto varía de una máquina a otra, dependiendo de la velocidad de memoria, del número de registros que tengan que copiarse y de la existencia de instrucciones especiales (como por ejemplo, una instrucción para cargar o almacenar todos los registros). Las velocidades típicas son del orden de unos pocos milisegundos.

El tiempo empleado en los cambios de contexto depende fundamentalmente del soporte hardware. Por ejemplo, algunos procesadores (como Ultra\SPARC de Sun) proporcionan múltiples conjuntos de registros. En este caso, un cambio de contexto simplemente requiere cambiar el puntero al conjunto actual de registros. Por supuesto, si hay más procesos activos que conjuntos de registros, el sistema recurrirá a copiar los datos de los registros en y desde memoria, al igual que antes. También, cuanto más complejo es el sistema operativo, más trabajo debe realizar durante un cambio de contexto. Como veremos en el Capítulo 8, las técnicas avanzadas de gestión de memoria pueden requerir que con cada contexto se intercambien datos adicionales. Por ejemplo, el espacio de direcciones del proceso actual debe preservarse en el momento de preparar para su uso el espacio de la siguiente tarea. Cómo se conserva el espacio de memoria y qué cantidad de trabajo es necesario para conservarlo depende del método de gestión de memoria utilizado por el sistema operativo.

3.3 Operaciones sobre los procesos

En la mayoría de los sistemas, los procesos pueden ejecutarse de forma concurrente y pueden crearse y eliminarse dinámicamente. Por tanto, estos sistemas deben proporcionar un mecanismo para la creación y terminación de procesos. En esta sección, vamos a ocuparnos de los mecanismos implicados en la creación de procesos y los ilustraremos analizando el caso de los sistemas UNIX y Windows.

3.3.1 Creación de procesos

Un proceso puede crear otros varios procesos nuevos mientras se ejecuta; para ello se utiliza una llamada al sistema específica para la creación de procesos. El proceso creador se denomina proceso **padre** y los nuevos procesos son los hijos de dicho proceso. Cada uno de estos procesos nuevos puede a su vez crear otros procesos, dando lugar a un **árbol de procesos**.

La mayoría de los sistemas operativos (incluyendo UNIX y la familia Windows de sistemas operativos) identifican los procesos mediante un **identificador de proceso** único o **pid** (process identifier), que normalmente es un número entero. La Figura 3.9 ilustra un árbol de procesos típico en el sistema operativo Solaris, indicando el nombre de cada proceso y su pid. En Solaris, el proceso situado en la parte superior del árbol es el proceso `sched`, con el pid 0. El proceso `sched` crea varios procesos hijo, incluyendo `pageout` y `fsflush`. Estos procesos son responsables de la gestión de memoria y de los sistemas de archivos. El proceso `sched` también crea el proceso `init`, que sirve como proceso padre raíz para todos los procesos de usuario. En la Figura 3.9

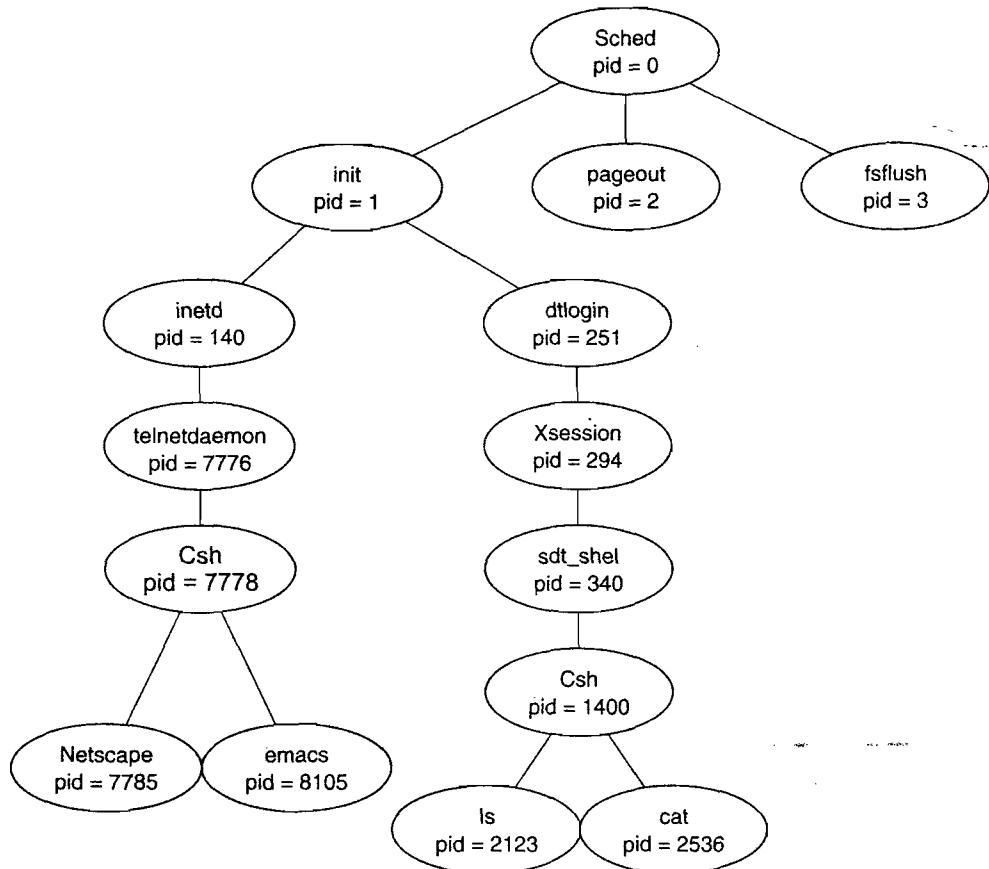


Figura 3.9 Árbol de procesos en un sistema Solaris típico.

vemos dos hijos de `init`: `inetd` y `dtlogin`. El proceso `inetd` es responsable de los servicios de red, como `telnet` y `ftp`; el proceso `dtlogin` es el proceso que representa una pantalla de inicio de sesión de usuario. Cuando un usuario inicia una sesión, `dtlogin` crea una sesión de X-Windows (`Xsession`), que a su vez crea el proceso `sdt_shel`. Por debajo de `sdt_shel`, se crea una *shell* de línea de comandos de usuario, *C-shell* o *csh*. Es en esta interfaz de línea de comandos donde el usuario invoca los distintos procesos hijo, tal como los comandos `ls` y `cat`. También vemos un proceso `csh` con el pid 7778, que representa a un usuario que ha iniciado una sesión en el sistema a través de `telnet`. Este usuario ha iniciado el explorador `Netscape` (pid 7785) y el editor `emacs` (pid 8105).

En UNIX, puede obtenerse un listado de los procesos usando el comando `ps`. Por ejemplo, el comando `ps -el` proporciona información completa sobre todos los procesos que están activos actualmente en el sistema. Resulta fácil construir un árbol de procesos similar al que se muestra en la Figura 3.9, trazando recursivamente los procesos padre hasta llegar al proceso `init`.

En general, un proceso necesitará ciertos recursos (tiempo de CPU, memoria, archivos, dispositivos de E/S) para llevar a cabo sus tareas. Cuando un proceso crea un subproceso, dicho subproceso puede obtener sus recursos directamente del sistema operativo o puede estar restringido a un subconjunto de los recursos del proceso padre. El padre puede tener que repartir sus recursos entre sus hijos, o puede compartir algunos recursos (como la memoria o los archivos) con algunos de sus hijos. Restringir un proceso hijo a un subconjunto de los recursos del padre evita que un proceso pueda sobrecargar el sistema creando demasiados subprocesos.

Además de los diversos recursos físicos y lógicos que un proceso obtiene en el momento de su creación, el proceso padre puede pasar datos de inicialización (entrada) al proceso hijo. Por ejemplo, considere un proceso cuya función sea mostrar los contenidos de un archivo, por ejemplo `img.jpg`, en la pantalla de un terminal. Al crearse, obtendrá como entrada de su proceso padre el

nombre del archivo *img.jpg* y empleará dicho nombre de archivo, lo abrirá y mostrará el contenido. También puede recibir el nombre del dispositivo de salida. Algunos sistemas operativos pasan recursos a los procesos hijo. En un sistema así, el proceso nuevo puede obtener como entrada dos archivos abiertos, *img.jpg* y el dispositivo terminal, y simplemente transferir los datos entre ellos.

Cuando un proceso crea otro proceso nuevo, existen dos posibilidades en términos de ejecución:

1. El padre continúa ejecutándose concurrentemente con su hijo.
2. El padre espera hasta que alguno o todos los hijos han terminado de ejecutarse.

También existen dos posibilidades en función del espacio de direcciones del nuevo proceso:

1. El proceso hijo es un duplicado del proceso padre (usa el mismo programa y los mismos datos que el padre).
2. El proceso hijo carga un nuevo programa.

Para ilustrar estas diferencias, consideremos en primer lugar el sistema operativo UNIX. En UNIX, como hemos visto, cada proceso se identifica mediante su identificador de proceso, que es un entero único. Puede crearse un proceso nuevo mediante la llamada al sistema `fork()`. El nuevo proceso consta de una copia del espacio de direcciones del proceso original. Este mecanismo permite al proceso padre comunicarse fácilmente con su proceso hijo. Ambos procesos (padre e hijo) continúan la ejecución en la instrucción que sigue a `fork()`, con una diferencia: el código de retorno para `fork()` es cero en el caso del proceso nuevo (hijo), mientras que al padre se le devuelve el identificador de proceso (distinto de cero) del hijo.

Normalmente, uno de los dos procesos utiliza la llamada al sistema `exec()` después de una llamada al sistema `fork()`, con el fin de sustituir el espacio de memoria del proceso con un nuevo programa. La llamada al sistema `exec()` carga un archivo binario en memoria (destruyendo la imagen en memoria del programa que contiene la llamada al sistema `exec()`) e inicia su ejecución. De esta manera, los dos procesos pueden comunicarse y seguir luego caminos separados. El padre puede crear más hijos, o, si no tiene nada que hacer mientras se ejecuta el hijo, puede ejecutar una llamada al sistema `wait()` para auto-excluirse de la cola de procesos preparados hasta que el proceso hijo se complete.

El programa C mostrado en la Figura 3.10 ilustra las llamadas al sistema descritas, para un sistema UNIX. Ahora tenemos dos procesos diferentes ejecutando una copia del mismo programa. El valor `pid` del proceso hijo es cero; el del padre es un valor entero mayor que cero. El proceso hijo sustituye su espacio de direcciones mediante el comando `/bin/ls` de UNIX (utilizado para obtener un listado de directorios) usando la llamada al sistema `execlp()` (`execlp()` es una versión de la llamada al sistema `exec()`). El padre espera a que el proceso hijo se complete, usando para ello la llamada al sistema `wait()`. Cuando el proceso hijo termina (invocando implícita o explícitamente `exit()`), el proceso padre reanuda su ejecución después de la llamada a `wait()`, terminando su ejecución mediante la llamada al sistema `exit()`. Esta secuencia se ilustra en la Figura 3.11.

Como ejemplo alternativo, consideremos ahora la creación de procesos en Windows. Los procesos se crean en la API de Win32 mediante la función `CreateProcess()`, que es similar a `fork()` en el sentido de que un padre crea un nuevo proceso hijo. Sin embargo, mientras que con `fork()` el proceso hijo hereda el espacio de direcciones de su padre, `CreateProcess()` requiere cargar un programa específico en el espacio de direcciones del proceso hijo durante su creación. Además, mientras que a `fork()` no se le pasa ningún parámetro, `CreateProcess()` necesita al menos diez parámetros distintos.

El programa C mostrado en la Figura 3.12 ilustra la función `CreateProcess()`, la cual crea un proceso hijo que carga la aplicación `mspaint.exe`. Hemos optado por muchos de los valores predeterminados de los diez parámetros pasados a `CreateProcess()`. Animamos, a los lectores interesados en profundizar en los detalles sobre la creación y gestión de procesos en la API de Win32, a que consulten las notas bibliográficas incluidas al final del capítulo.

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* bifurca un proceso hijo */
pid = fork();

if (pid < 0) /* se produce un error */

    fprintf(stderr, "Fork Failed");
    exit(-1);
}
else if (pid == 0) /* proceso hijo */
    execlp("/bin/ls/", "ls", NULL);
}
else /* proceso padre*/
    /* el padre espera a que el proceso hijo se complete */
    wait(NULL);
    printf("Hijo completado")
}
}

```

Figura 3.10 Programa C que bifurca un proceso distinto.

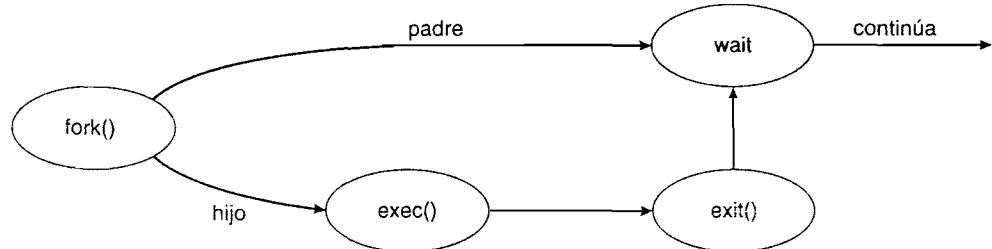


Figura 3.11 Creación de un proceso.

Los dos parámetros pasados a `CreateProcess()` son instancias de las estructuras `STARTUPINFO` y `PROCESS_INFORMATION`. `STARTUPINFO` especifica muchas propiedades del proceso nuevo, como el tamaño y la apariencia de la ventana y gestiona los archivos de entrada y de salida estándar. La estructura `PROCESS_INFORMATION` contiene un descriptor y los identificadores de los procesos recientemente creados y su hebra. Invocamos la función `ZeroMemory()` para asignar memoria a cada una de estas estructuras antes de continuar con `CreateProcess()`.

Los dos primeros parámetros pasados a `CreateProcess()` son el nombre de la aplicación y los parámetros de la línea de comandos. Si el nombre de aplicación es `NULL` (en cuyo caso estamos), el parámetro de la línea de comandos especifica la aplicación que hay que cargar. En este caso, cargamos la aplicación `mspaint.exe` de Microsoft Windows. Además de estos dos parámetros iniciales, usamos los parámetros predeterminados para heredar los descriptores de procesos y hebras, y no especificamos ningún indicador de creación. También usamos el bloque de entorno existente del padre y su directorio de inicio. Por último, proporcionamos dos punteros a las estructuras `PROCESS_INFORMATION` y `STARTUPINFO` creadas al principio del programa. En Figura 3.10, el proceso padre espera a que el hijo se complete invocando la llamada al sistema `wait()`. El equivalente en Win 32 es `WaitForSingleObject()`, a la que se pasa un descriptor.

```

#include <stdio.h>
#include <windows.h>

int main (VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

// asignar memoria
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

// crear proceso hijo
if (!CreateProcess(NULL, // utilizar línea de comandos
    "C:\WINDOWS\system32\mspaint.exe", // línea de comandos
    NULL, // no hereda descriptor del proceso
    NULL, // no hereda descriptor de la hebra
    FALSE, // inhabilitar herencia del descriptor
    0, // no crear indicadores
    NULL, // usar bloque de entorno del padre
    NULL, // usar directorio existente del padre
    &si,
    &pi))
{
    fprintf(stderr, "Fallo en la creación del proceso");
    return -1;
}
// el padre espera hasta que el hijo termina
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Hijo completado");

// cerrar descriptores
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}

```

Figura 3.12 Creación de un proceso separado usando la API de Win32.

del proceso hijo, `pi.hProcess`, cuya ejecución queremos esperar a que se complete. Una vez que el proceso hijo termina, se devuelve el control desde la función `WaitForSingleObject()` del proceso padre.

3.3.2 Terminación de procesos

- Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo elimine usando la llamada al sistema `exit()`. En este momento, el proceso puede devolver un valor de estado (normalmente, un entero) a su proceso padre (a través de la llamada al sistema `wait()`). El sistema operativo libera la asignación de todos los recursos del proceso, incluyendo las memorias física y virtual, los archivos abiertos y los búferes de E/S.

La terminación puede producirse también en otras circunstancias. Un proceso puede causar la terminación de otro proceso a través de la adecuada llamada al sistema (por ejemplo, `TerminateProcess()` en Win32). Normalmente, dicha llamada al sistema sólo puede ser invocada por el padre del proceso que se va a terminar. En caso contrario, los usuarios podrían terminar arbitrariamente los trabajos de otros usuarios. Observe que un padre necesita conocer las

identidades de sus hijos. Por tanto, cuando un proceso crea un proceso nuevo, se pasa al padre la identidad del proceso que se acaba de crear.

Un padre puede terminar la ejecución de uno de sus hijos por diversas razones, como por ejemplo, las siguientes:

- El proceso hijo ha excedido el uso de algunos de los recursos que se le han asignado. Para determinar si tal cosa ha ocurrido, el padre debe disponer de un mecanismo para inspeccionar el estado de sus hijos.
- La tarea asignada al proceso hijo ya no es necesaria.
- El padre abandona el sistema, y el sistema operativo no permite que un proceso hijo continúe si su padre ya ha terminado.

Algunos sistemas, incluyendo VMS, no permiten que un hijo siga existiendo si su proceso padre se ha completado. En tales sistemas, si un proceso termina (sea normal o anormalmente), entonces todos sus hijos también deben terminarse. Este fenómeno, conocido como **terminación en cascada**, normalmente lo inicia el sistema operativo.

Para ilustrar la ejecución y terminación de procesos, considere que, en UNIX, podemos terminar un proceso usando la llamada al sistema `exit()`; su proceso padre puede esperar a la terminación del proceso hijo usando la llamada al sistema `wait()`. La llamada al sistema `wait()` devuelve el identificador de un proceso hijo completado, con el fin de que el padre pueda saber cuál de sus muchos hijos ha terminado. Sin embargo, si el proceso padre se ha completado, a todos sus procesos hijo se les asigna el proceso `init` como su nuevo parente. Por tanto, los hijos todavía tienen un parente al que proporcionar su estado y sus estadísticas de ejecución.

3.4 Comunicación interprocesos

Los procesos que se ejecutan concurrentemente pueden ser procesos independientes o procesos cooperativos. Un proceso es **independiente** si no puede afectar o verse afectado por los restantes procesos que se ejecutan en el sistema. Cualquier proceso que no comparte datos con ningún otro proceso es un proceso independiente. Un proceso es **cooperativo** si puede afectar o verse afectado por los demás procesos que se ejecutan en el sistema. Evidentemente, cualquier proceso que comparte datos con otros procesos es un proceso cooperativo.

Hay varias razones para proporcionar un entorno que permita la cooperación entre procesos:

- **Compartir información.** Dado que varios usuarios pueden estar interesados en la misma información (por ejemplo, un archivo compartido), debemos proporcionar un entorno que permita el acceso concurrente a dicha información.
- **Acelerar los cálculos.** Si deseamos que una determinada tarea se ejecute rápidamente, debemos dividirla en subtareas, ejecutándose cada una de ellas en paralelo con las demás. Observe que tal aceleración sólo se puede conseguir si la computadora tiene múltiples elementos de procesamiento, como por ejemplo varias CPU o varios canales de E/S.
- **Modularidad.** Podemos querer construir el sistema de forma modular, dividiendo las funciones del sistema en diferentes procesos o hebras, como se ha explicado en el Capítulo 2.
- **Conveniencia.** Incluso un solo usuario puede querer trabajar en muchas tareas al mismo tiempo. Por ejemplo, un usuario puede estar editando, imprimiendo y compilando en paralelo.

La cooperación entre procesos requiere mecanismos de **comunicación interprocesos** (IPC, interprocess communication) que les permitan intercambiar datos e información. Existen dos modelos fundamentales de comunicación interprocesos: (1) **memoria compartida** y (2) **paso de mensajes**. En el modelo de memoria compartida, se establece una región de la memoria para que sea compartida por los procesos cooperativos. De este modo, los procesos pueden intercambiar información leyendo y escribiendo datos en la zona compartida. En el modelo de paso de mensa-

jes, la comunicación tiene lugar mediante el intercambio de mensajes entre los procesos cooperativos. En la Figura 3.13 se comparan los dos modelos de comunicación.

Los dos modelos que acabamos de presentar son bastante comunes en los distintos sistemas operativos y muchos sistemas implementan ambos. El paso de mensajes resulta útil para intercambiar pequeñas cantidades de datos, ya que no existe la necesidad de evitar conflictos. El paso de mensajes también es más fácil de implementar que el modelo de memoria compartida como mecanismo de comunicación entre computadoras. La memoria compartida permite una velocidad máxima y una mejor comunicación, ya que puede realizarse a velocidades de memoria cuando se hace en una misma computadora. La memoria compartida es más rápida que el paso de mensajes, ya que este último método se implementa normalmente usando llamadas al sistema y, por tanto, requiere que intervenga el *kernel*, lo que consume más tiempo. Por el contrario, en los sistemas de memoria compartida, las llamadas al sistema sólo son necesarias para establecer las zonas de memoria compartida. Una vez establecida la memoria compartida, todos los accesos se tratan como accesos a memoria rutinarios y no se precisa la ayuda del *kernel*. En el resto de esta sección, nos ocupamos en detalle de cada uno de estos modelos de comunicación IPC.

3.4.1 Sistemas de memoria compartida

La comunicación interprocesos que emplea memoria compartida requiere que los procesos que se estén comunicando establezcan una región de memoria compartida. Normalmente, una región de memoria compartida reside en el espacio de direcciones del proceso que crea el segmento de memoria compartida. Otros procesos que deseen comunicarse usando este segmento de memoria compartida deben conectarse a su espacio de direcciones. Recuerde que, habitualmente, el sistema operativo intenta evitar que un proceso acceda a la memoria de otro proceso. La memoria compartida requiere que dos o más procesos acuerden eliminar esta restricción. Entonces podrán intercambiar información leyendo y escribiendo datos en las áreas compartidas. El formato de los datos y su ubicación están determinados por estos procesos, y no se encuentran bajo el control del sistema operativo. Los procesos también son responsables de verificar que no escriben en la misma posición simultáneamente.

Para ilustrar el concepto de procesos cooperativos, consideremos el problema del productor-consumidor, el cual es un paradigma comúnmente utilizado para los procesos cooperativos. Un proceso **productor** genera información que consume un proceso **consumidor**. Por ejemplo, un compilador puede generar código ensamblado, que consume un ensamblador. El ensamblador, a su vez, puede generar módulos objeto, que consume el cargador. El problema del productor-consumidor también proporciona una metáfora muy útil para el paradigma cliente-servidor.

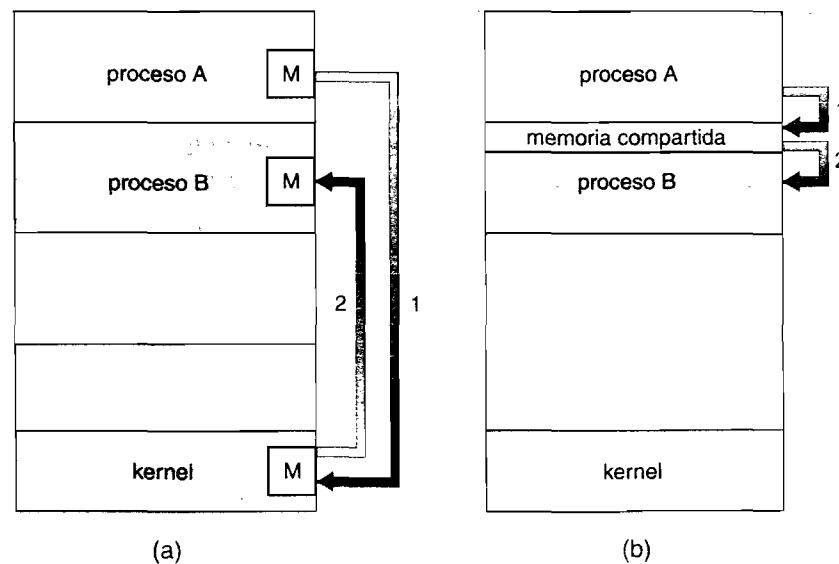


Figura 3.13 Modelos de comunicación. (a) Paso de mensajes. (b) Memoria compartida.

Generalmente, pensamos en un servidor como en un productor y en un cliente como en un consumidor. Por ejemplo, un servidor web produce (es decir, proporciona) archivos HTML e imágenes, que consume (es decir, lee) el explorador web cliente que solicita el recurso.

Una solución para el problema del productor-consumidor es utilizar mecanismos de memoria compartida. Para permitir que los procesos productor y consumidor se ejecuten de forma concurrente, debemos tener disponible un búfer de elementos que pueda llenar el productor y vaciar el consumidor. Este búfer residirá en una región de memoria que será compartida por ambos procesos, consumidor y productor. Un productor puede generar un elemento mientras que el consumidor consume otro. El productor y el consumidor deben estar sincronizados, de modo que el consumidor no intente consumir un elemento que todavía no haya sido producido.

Pueden emplearse dos tipos de búferes. El sistema de **búfer no limitado** no pone límites al tamaño de esa memoria compartida. El consumidor puede tener que esperar para obtener elementos nuevos, pero el productor siempre puede generar nuevos elementos. El sistema de **búfer limitado** establece un tamaño de búfer fijo. En este caso, el consumidor tiene que esperar si el búfer está vacío y el productor tiene que esperar si el búfer está lleno.

Veamos más en detalle cómo puede emplearse un búfer limitado para permitir que los procesos comparten la memoria. Las siguientes variables residen en una zona de la memoria compartida por los procesos consumidor y productor:

```
#define BUFFER_SIZE 10

typedef struct {
    .
    .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

El búfer compartido se implementa como una matriz circular con dos punteros lógicos: *in* y *out*. La variable *in* apunta a la siguiente posición libre en el búfer; *out* apunta a la primera posición ocupada del búfer. El búfer está vacío cuando *in* == *out*; el búfer está lleno cuando ((*in* + 1) % *BUFFER_SIZE*) == *out*.

El código para los procesos productor y consumidor se muestra en las Figuras 3.14 y 3.15, respectivamente. El proceso productor tiene una variable local, *nextProduced*, en la que se almacena el elemento nuevo que se va a generar. El proceso consumidor tiene una variable local, *nextConsumed*, en la que se almacena el elemento que se va a consumir.

Este esquema permite tener como máximo *BUFFER_SIZE* - 1 elementos en el búfer al mismo tiempo. Dejamos como ejercicio para el lector proporcionar una solución en la que *BUFFER_SIZE* elementos puedan estar en el búfer al mismo tiempo. En la Sección 3.5.1 se ilustra la API de POSIX para los sistemas de memoria compartida.

Un problema del que no se ocupa este ejemplo es la situación en la que tanto el proceso productor como el consumidor intentan acceder al búfer compartido de forma concurrente. En el Capítulo 6 veremos cómo puede implementarse la sincronización entre procesos cooperativos de forma efectiva en un entorno de memoria compartida.

```
item nextProduced;
.

while (true) {
    /* produce e inserta un elemento en nextProduced*/
    while ((in+1) % BUFFER_SIZE) == out)
        ; /*no hacer nada*/
    buffer[in]=nextProduced;
    in = (in+1) % BUFFER_SIZE;
}
```

Figura 3.14 El proceso productor.

```

item nextConsumed;

while (true) {
    while (in == out)
        ; /*no hacer nada*/

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume el elemento almacenado en nextConsumed */
}

```

Figura 3.15 El proceso consumidor.

3.4.2 Sistemas de paso de mensajes

En la Sección 3.4.1 hemos mostrado cómo pueden comunicarse procesos cooperativos en un entorno de memoria compartida. El esquema requiere que dichos procesos compartan una zona de la memoria y que el programador de la aplicación escriba explícitamente el código para acceder y manipular la memoria compartida. Otra forma de conseguir el mismo efecto es que el sistema operativo proporcione los medios para que los procesos cooperativos se comuniquen entre sí a través de una facilidad de paso de mensajes.

El paso de mensajes proporciona un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin compartir el mismo espacio de direcciones, y es especialmente útil en un entorno distribuido, en el que los procesos que se comunican pueden residir en diferentes computadoras conectadas en red. Por ejemplo, un programa de **chat** utilizado en la World Wide Web podría diseñarse de modo que los participantes en la conversación se comunicaran entre sí intercambiando mensajes.

Una facilidad de paso de mensajes proporciona al menos dos operaciones: envío de mensajes (**send**) y recepción de mensajes (**receive**). Los mensajes enviados por un proceso pueden tener un tamaño fijo o variable. Si sólo se pueden enviar mensajes de tamaño fijo, la implementación en el nivel de sistema es directa. Sin embargo, esta restricción hace que la tarea de programación sea más complicada. Por el contrario, los mensajes de tamaño variable requieren una implementación más compleja en el nivel de sistema, pero la tarea de programación es más sencilla. Éste es un tipo de compromiso que se encuentra muy habitualmente en el diseño de sistemas operativos.

Si los procesos *P* y *Q* desean comunicarse, tienen que enviarse mensajes entre sí; debe existir un **enlace de comunicaciones** entre ellos. Este enlace se puede implementar de diferentes formas. No vamos a ocuparnos aquí de la implementación física del enlace (memoria compartida, bus hardware o red), que se verá en el Capítulo 16, sino de su implementación lógica. Existen varios métodos para implementar lógicamente un enlace y las operaciones de envío y recepción:

- Comunicación directa o indirecta.
- Comunicación síncrona o asíncrona.
- Almacenamiento en búfer explícito o automático.

Veamos ahora los problemas relacionados con cada una de estas funcionalidades.

3.4.2.1 Nombrado

Los procesos que se van a comunicar deben disponer de un modo de referenciarse entre sí. Pueden usar comunicación directa o indirecta.

En el caso de la **comunicación directa**, cada proceso que desea establecer una comunicación debe nombrar de forma explícita al receptor o transmisor de la comunicación. En este esquema, las primitivas **send()** y **receive()** se definen del siguiente modo:

- **send(*P*, mensaje)**— Envía un mensaje al proceso *P*.

- `receive(Q, mensaje)`— Recibe un mensaje del proceso Q.

Un enlace de comunicaciones, según este esquema, tiene las siguientes propiedades:

- Los enlaces se establecen de forma automática entre cada par de procesos que quieran comunicarse. Los procesos sólo tienen que conocer la identidad del otro para comunicarse.
- Cada enlace se asocia con exactamente dos procesos.
- Entre cada par de procesos existe exactamente un enlace.

Este esquema presenta *simetría* en lo que se refiere al direccionamiento, es decir, tanto el proceso transmisor como el proceso receptor deben nombrar al otro para comunicarse. Existe una variante de este esquema que emplea *asimetría* en el direccionamiento. En este caso, sólo el transmisor nombra al receptor; el receptor no tiene que nombrar al transmisor. En este esquema, las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(P, mensaje)`— Envía un mensaje al proceso P
- `receive(id, mensaje)`— Recibe un mensaje de cualquier proceso; a la variable `id` se le asigna el nombre del proceso con el que se ha llevado a cabo la comunicación.

La desventaja de estos dos esquemas (simétrico y asimétrico) es la limitada modularidad de las definiciones de procesos resultantes. Cambiar el identificador de un proceso puede requerir que se modifiquen todas las restantes definiciones de procesos. Deben localizarse todas las referencias al identificador antiguo, para poder sustituirlas por el nuevo identificador. En general, cualquier técnica de **precodificación**, en la que los identificadores deban establecerse explícitamente, es menos deseable que las técnicas basadas en la indirección, como se describe a continuación.

Con el modelo de comunicación indirecta, los mensajes se envían y reciben en **buzones de correo o puertos**. Un buzón de correo puede verse de forma abstracta como un objeto en el que los procesos pueden colocar mensajes y del que pueden eliminar mensajes. Cada buzón de correo tiene asociada una identificación única. Por ejemplo, las colas de mensajes de POSIX usan un valor entero para identificar cada buzón de correo. En este esquema, un proceso puede comunicarse con otros procesos a través de una serie de buzones de correo diferentes. Sin embargo, dos procesos sólo se pueden comunicar si tienen un buzón de correo compartido. Las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(A, mensaje)`— Envía un mensaje al buzón de correo A.
- `receive(A, mensaje)`— Recibe un mensaje del buzón de correo A.

En este esquema, un enlace de comunicaciones tiene las siguientes propiedades:

- Puede establecerse un enlace entre un par de procesos sólo si ambos tienen un buzón de correo compartido.
- Un enlace puede asociarse con más de dos procesos.
- Entre cada par de procesos en comunicación, puede haber una serie de enlaces diferentes, correspondiendo cada enlace a un buzón de correo.

Ahora supongamos que los procesos P_1 , P_2 y P_3 comparten el buzón de correo A. El proceso P_1 envía un mensaje a A, mientras que los procesos P_2 y P_3 ejecutan una instrucción `receive()` de A. ¿Qué procesos recibirán el mensaje enviado por P_1 ? La respuesta depende de cuál de los métodos siguientes elijamos:

- Permitir que cada enlace esté asociado como máximo con dos procesos.
- Permitir que sólo un proceso, como máximo, ejecute una operación de recepción en cada momento.
- Permitir que el sistema seleccione arbitrariamente qué proceso recibirá el mensaje (es decir, P_2 o P_3 , pero no ambos). El sistema también puede definir un algoritmo para seleccionar qué

proceso recibirá el mensaje (por ejemplo, que los procesos reciban por turnos los mensajes). El sistema puede identificar al receptor ante el transmisor.

Un buzón de correo puede ser propiedad de un proceso o del sistema operativo. Si es propiedad de un proceso, es decir, si el buzón de correo forma parte del espacio de direcciones del proceso, entonces podemos diferenciar entre el propietario (aquel que sólo recibe mensajes a través de este buzón) y el usuario (aquel que sólo puede enviar mensajes a dicho buzón de correo). Puesto que cada buzón de correo tiene un único propietario, no puede haber confusión acerca de quién recibirá un mensaje enviado a ese buzón de correo. Cuando un proceso que posee un buzón de correo termina, dicho buzón desaparece. A cualquier proceso que con posterioridad envíe un mensaje a ese buzón debe notificársele que dicho buzón ya no existe.

Por el contrario, un buzón de correo que sea propiedad del sistema operativo tiene existencia propia: es independiente y no está asociado a ningún proceso concreto. El sistema operativo debe proporcionar un mecanismo que permita a un proceso hacer lo siguiente:

- Crear un buzón de correo nuevo.
- Enviar y recibir mensajes a través del buzón de correo.
- Eliminar un buzón de correo.

Por omisión, el proceso que crea un buzón de correo nuevo es el propietario del mismo. Inicialmente, el propietario es el único proceso que puede recibir mensajes a través de este buzón. Sin embargo, la propiedad y el privilegio de recepción se pueden pasar a otros procesos mediante las apropiadas llamadas al sistema. Por supuesto, esta medida puede dar como resultado que existan múltiples receptores para cada buzón de correo.

3.4.2.2 Sincronización

La comunicación entre procesos tiene lugar a través de llamadas a las primitivas `send()` y `receive()`. Existen diferentes opciones de diseño para implementar cada primitiva. El paso de mensajes puede ser **con bloqueo** o **sin bloqueo**, mecanismos también conocidos como **síncrono** y **asíncrono**.

- **Envío con bloqueo.** El proceso que envía se bloquea hasta que el proceso receptor o el buzón de correo reciben el mensaje.
- **Envío sin bloqueo.** El proceso transmisor envía el mensaje y continúa operando.
- **Recepción con bloqueo.** El receptor se bloquea hasta que hay un mensaje disponible.
- **Recepción sin bloqueo.** El receptor extrae un mensaje válido o un mensaje nulo.

Son posibles diferentes combinaciones de las operaciones `send()` y `receive()`. Cuando ambas operaciones se realizan con bloqueo, tenemos lo que se denomina un **rendezvous** entre el transmisor y el receptor. La solución al problema del productor-consumidor es trivial cuando se usan instrucciones `send()` y `receive()` con bloqueo. El productor simplemente invoca la llamada `send()` con bloqueo y espera hasta que el mensaje se entrega al receptor o al buzón de correo. Por otro lado, cuando el consumidor invoca la llamada `receive()`, se bloquea hasta que hay un mensaje disponible.

Observe que los conceptos de síncrono y asíncrono se usan con frecuencia en los algoritmos de E/S en los sistemas operativos, como veremos a lo largo del texto.

3.4.2.3 Almacenamiento en búfer

Sea la comunicación directa o indirecta, los mensajes intercambiados por los procesos que se están comunicando residen en una cola temporal. Básicamente, tales colas se pueden implementar de tres maneras:

- **Capacidad cero.** La cola tiene una longitud máxima de cero; por tanto, no puede haber ningún mensaje esperando en el enlace. En este caso, el transmisor debe bloquearse hasta que el receptor reciba el mensaje.
- **Capacidad limitada.** La cola tiene una longitud finita n ; por tanto, puede haber en ella n mensajes como máximo. Si la cola no está llena cuando se envía un mensaje, el mensaje se introduce en la cola (se copia el mensaje o se almacena un puntero al mismo), y el transmisor puede continuar la ejecución sin esperar. Sin embargo, la capacidad del enlace es finita. Si el enlace está lleno, el transmisor debe bloquearse hasta que haya espacio disponible en la cola.
- **Capacidad ilimitada.** La longitud de la cola es potencialmente infinita; por tanto, puede haber cualquier cantidad de mensajes esperando en ella. El transmisor nunca se bloquea.

En ocasiones, se dice que el caso de capacidad cero es un sistema de mensajes sin almacenamiento en búfer; los otros casos se conocen como sistemas con almacenamiento en búfer automático.

3.5 Ejemplos de sistemas IPC

En esta sección vamos a estudiar tres sistemas IPC diferentes. En primer lugar, analizaremos la API de POSIX para el modelo de memoria compartida, así como el modelo de paso de mensajes en el sistema operativo Mach. Concluiremos con Windows XP, que usa de una forma interesante el modelo de memoria compartida como mecanismo para proporcionar ciertos mecanismos de paso de mensajes.

3.5.1 Un ejemplo: memoria compartida en POSIX

Para los sistemas POSIX hay disponibles varios mecanismos IPC, incluyendo los de memoria compartida y de paso de mensajes. Veamos primero la API de POSIX para memoria compartida.

En primer lugar, un proceso tiene que crear un segmento de memoria compartida usando la llamada al sistema `shmget()`. `shmget()` se deriva de Shared Memory GET (obtención de datos a través de memoria compartida). El siguiente ejemplo ilustra el uso de `shmget()`.

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

El primer parámetro especifica la clave (o identificador) del segmento de memoria compartida. Si se define como `IPC_PRIVATE`, se crea un nuevo segmento de memoria compartida. El segundo parámetro especifica el tamaño (en bytes) del segmento. Por último, el tercer parámetro identifica el modo, que indica cómo se va a usar el segmento de memoria compartida: para leer, para escribir o para ambas operaciones. Al establecer el modo como `S_IRUSR | S_IWUSR`, estamos indicando que el propietario puede leer o escribir en el segmento de memoria compartida. Una llamada a `shmget()` que se ejecute con éxito devolverá un identificador entero para el segmento. Otros procesos que deseen utilizar esa región de la memoria compartida deberán especificar este identificador.

Los procesos que deseen acceder a un segmento de memoria compartida deben asociarlo a su espacio de direcciones usando la llamada al sistema `shmat()` [Shared Memory ATTach]. La llamada a `shmat()` espera también tres parámetros. El primero es el identificador entero del segmento de memoria compartida al que se va a conectar, el segundo es la ubicación de un puntero en memoria, que indica dónde se asociará la memoria compartida; si pasamos el valor `NULL`, el sistema operativo selecciona la ubicación en nombre del usuario. El tercer parámetro especifica un indicador que permite que la región de memoria compartida se conecte en modo de sólo lectura o sólo escritura. Pasando un parámetro de valor 0, permitimos tanto lecturas como escrituras en la memoria compartida.

El tercer parámetro especifica un indicador de modo. Si se define, el indicador de modo permite a la región de memoria compartida conectarse en modo de sólo lectura; si se define como 0, permite tanto lecturas como escrituras en dicha región. Para asociar una región de memoria compartida usando `shmat()`, podemos hacer como sigue:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

Si se ejecuta correctamente, `shmat()` devuelve un puntero a la posición inicial de memoria a la que se ha asociado la región de memoria compartida.

Una vez que la región de memoria compartida se ha asociado al espacio de direcciones de un proceso, éste puede acceder a la memoria compartida como en un acceso de memoria normal, usando el puntero devuelto por `shmat()`. En este ejemplo, `shmat()` devuelve un puntero a una cadena de caracteres. Por tanto, podríamos escribir en la región de memoria compartida como sigue:

```
sprintf(shared_memory, "Escribir en memoria compartida");
```

Los otros procesos que comparten este segmento podrán ver las actualizaciones hechas en el segmento de memoria compartida.

Habitualmente, un proceso que usa un segmento de memoria compartida existente asocia primero la región de memoria compartida a su espacio de direcciones y luego accede (y posiblemente actualiza) dicha región. Cuando un proceso ya no necesita acceder al segmento de memoria compartida, desconecta el segmento de su espacio de direcciones. Para desconectar una región de memoria compartida, el proceso puede pasar el puntero de la región de memoria compartida a la llamada al sistema `shmctl()`, de la forma siguiente:

```
shmctl(shared_memory);
```

Por último, un segmento de memoria compartida puede eliminarse del sistema mediante la llamada al sistema `shmctl()`, a la cual se pasa el identificador del segmento compartido junto con el indicador `IPC_RMID`.

El programa mostrado en la Figura 3.16 ilustra la API de memoria compartida de POSIX explicada anteriormente. Este programa crea un segmento de memoria compartida de 4.096 bytes. Una vez que la región de memoria compartida se ha conectado, el proceso escribe el mensaje ¡Hola! en la memoria compartida. Después presenta a la salida el contenido de la memoria actualizada, y desconecta y elimina la región de memoria compartida. Al final del capítulo se proporcionan más ejercicios que usan la API de memoria compartida de POSIX.

3.5.2 Un ejemplo: Mach

Como ejemplo de sistema operativo basado en mensajes, vamos a considerar a continuación el sistema operativo Mach, desarrollado en la Universidad Carnegie Mellon. En el Capítulo 2, hemos presentado Mach como parte del sistema operativo Mac OS X. El *kernel* de Mach permite la creación y destrucción de múltiples tareas, que son similares a los procesos, pero tienen múltiples hebras de control. La mayor parte de las comunicaciones en Mach, incluyendo la mayoría de las llamadas al sistema y toda la comunicación inter-tareas, se realiza mediante *mensajes*. Los mensajes se envían y se reciben mediante buzones de correo, que en Mach se denominan *puertos*.

Incluso las llamadas al sistema se hacen mediante mensajes. Cuando se crea una tarea, también se crean dos buzones de correo especiales: el buzón de correo del *kernel* (Kernel) y el de notificaciones (Notify). El *kernel* utiliza el buzón Kernel para comunicarse con la tarea y envía las notificaciones de sucesos al puerto Notify. Sólo son necesarias tres llamadas al sistema para la transferencia de mensajes. La llamada `msg_send()` envía un mensaje a un buzón de correo. Un mensaje se recibe mediante `msg_receive()`. Finalmente, las llamadas a procedimientos remotos (RPC) se ejecutan mediante `msg_rpc()`, que envía un mensaje y espera a recibir como contestación exactamente un mensaje. De esta forma, las llamadas RPC modelan una llamada típica a procedimiento, pero pueden trabajar entre sistemas distintos (de ahí el calificativo de *remoto*).

La llamada al sistema `port_allocate()` crea un buzón de correo nuevo y asigna espacio para su cola de mensajes. El tamaño máximo de la cola de mensajes es, de manera predeterminada, de ocho mensajes. La tarea que crea el buzón es la propietaria de dicho buzón. El propietario también puede recibir mensajes del buzón de correo. Sólo una tarea cada vez puede poseer o recibir de un buzón de correo, aunque estos derechos pueden enviarse a otras tareas si se desea.

```

#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main() ... .
{
    /* el identificador para el segmento de memoria compartida */
    int segment_id=;
    /* un puntero al segmento de memoria compartida */
    char* shared_memory;
    /* el tamaño (en bytes) del segmento de memoria compartida */
    const int size = 4096;
    /* asignar un segmento de memoria compartida */
    segment_id = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* asociar el segmento de memoria compartida */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* escribir un mensaje en el segmento de memoria compartida */
    sprintf(shared_memory, ";Hola!");

    /* enviar a la salida la cadena de caracteres de la memoria
     * compartida */
    printf("**%s\n", shared_memory);

    /* desconectar el segmento de memoria compartida */
    shmdt (shared_memory);

    /* eliminar el segmento de memoria compartida */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}

```

Figura 3.16 Programa C que ilustra la API de memoria compartida de POSIX.

Inicialmente, el buzón de correo tiene una cola de mensajes vacía. A medida que llegan mensajes al buzón, éstos se copian en el mismo. Todos los mensajes tienen la misma prioridad. Mach garantiza que los múltiples mensajes de un mismo emisor se coloquen en la cola utilizando un algoritmo FIFO (first-in, first-out; primero en entrar, primero en salir), aunque el orden no se garantiza de forma absoluta. Por ejemplo, los mensajes procedentes de dos emisores distintos pueden ponerse en cola en cualquier orden.

Los mensajes en sí constan de una cabecera de longitud fija, seguida de unos datos de longitud variable. La cabecera indica la longitud del mensaje e incluye dos nombres de buzón de correo. Uno de ellos es el del buzón de correo al que se está enviando el mensaje. Habitualmente, la hebra emisora espera una respuesta, por lo que a la hebra receptora se le pasa el nombre del buzón del emisor; la hebra emisora puede usar ese buzón como una “dirección de retorno”.

La parte variable de un mensaje es una lista de elementos de datos con tipo. Cada entrada de la lista tiene un tipo, un tamaño y un valor. El tipo de los objetos especificados en el mensaje es importante, ya que pueden enviarse en los mensajes objetos definidos por el sistema operativo (como, por ejemplo, derechos de propiedad o de acceso de recepción, estados de tareas y segmentos de memoria).

Las operaciones de envío y recepción son flexibles. Por ejemplo, cuando se envía un mensaje a un buzón de correo, éste puede estar lleno. Si no está lleno, el mensaje se copia en el buzón y la hebra emisora continúa. Si el buzón está lleno, la hebra emisora tiene cuatro opciones:

1. Esperar indefinidamente hasta que haya espacio en el buzón.

2. Esperar como máximo n milisegundos.
3. No esperar nada y volver inmediatamente.
4. Almacenar el mensaje temporalmente en caché. Puede proporcionarse al sistema operativo un mensaje para que lo guarde, incluso aunque el buzón al que se estaba enviando esté lleno. Cuando el mensaje pueda introducirse en el buzón, el sistema enviará un mensaje de vuelta al emisor; en un instante determinado, para una determinada hebra emisora, sólo puede haber un mensaje pendiente de este tipo dirigido a un buzón lleno,

La última opción está pensada para las tareas de servidor, como por ejemplo un controlador de impresora. Después de terminar una solicitud, tales tareas pueden necesitar enviar una única respuesta a la tarea que solicitó el servicio, pero también deben continuar con otras solicitudes de servicio, incluso aunque el buzón de respuesta de un cliente esté lleno.

La operación de recepción debe especificar el buzón o el conjunto de buzones desde el se van a recibir los mensajes. Un **conjunto de buzones de correo** es una colección de buzones declarados por la tarea, que pueden agruparse y tratarse como un único buzón de correo, en lo que a la tarea respecta. Las hebras de una tarea pueden recibir sólo de un buzón de correo o de un conjunto de buzones para el que la tarea haya recibido autorización de acceso. Una llamada al sistema `port_status()` devuelve el número de mensajes que hay en un determinado buzón. La operación de recepción puede intentar recibir de (1) cualquier buzón del conjunto de buzones o (2) un buzón de correo específico (nominado). Si no hay ningún mensaje esperando a ser recibido, la hebra de recepción puede esperar como máximo n milisegundos o no esperar nada.

El sistema Mach fue especialmente diseñado para sistemas distribuidos, los cuales se estudian en los Capítulos 16 a 18, pero Mach también es adecuado para sistemas de un solo procesador, como demuestra su inclusión en el sistema Mac OS X. El principal problema con los sistemas de mensajes ha sido generalmente el pobre rendimiento, debido a la doble copia de mensajes: el mensaje se copia primero del emisor al buzón de correo y luego desde el buzón al receptor. El sistema de mensajes de Mach intenta evitar las operaciones de doble copia usando técnicas de gestión de memoria virtual (Capítulo 9). En esencia, Mach asigna el espacio de direcciones que contiene el mensaje del emisor al espacio de direcciones del receptor; el propio mensaje nunca se copia realmente. Esta técnica de gestión de mensajes proporciona un mayor rendimiento, pero sólo funciona para mensajes intercambiados dentro del sistema. El sistema operativo Mach se estudia en un capítulo adicional disponible en el sitio web del libro.

3.5.3 Un ejemplo: Windows XP

El sistema operativo Windows XP es un ejemplo de un diseño moderno que emplea la modularidad para incrementar la funcionalidad y disminuir el tiempo necesario para implementar nuevas características. Windows XP proporciona soporte para varios entornos operativos, o *subsistemas*, con los que los programas de aplicación se comunican usando un mecanismo de paso de mensajes. Los programas de aplicación se pueden considerar clientes del servidor de subsistemas de Windows XP.

La facilidad de paso de mensajes en Windows XP se denomina **llamada a procedimiento local** (LPC, local procedure call). En Windows XP, la llamada LPC establece la comunicación entre dos procesos de la misma máquina. Es similar al mecanismo estándar RPC, cuyo uso está muy extendido, pero está optimizado para Windows XP y es específico del mismo. Como Mach, Windows XP usa un objeto puerto para establecer y mantener una conexión entre dos procesos. Cada cliente que llama a un subsistema necesita un canal de comunicación, que se proporciona mediante un objeto puerto y que nunca se hereda. Windows XP usa dos tipos de puertos: puertos de conexión y puertos de comunicación. Realmente son iguales, pero reciben nombres diferentes según cómo se utilicen. Los puertos de conexión se denominan *objetos* y son visibles para todos los procesos; proporcionan a las aplicaciones una forma de establecer los canales de comunicación (Capítulo 22). La comunicación funciona del modo siguiente:

- El cliente abre un descriptor del objeto puerto de conexión del subsistema.

- El cliente envía una solicitud de conexión.
- El servidor crea dos puertos de comunicación privados y devuelve el descriptor de uno de ellos al cliente.
- El cliente y el servidor usan el descriptor del puerto correspondiente para enviar mensajes o realizar retrollamadas y esperar las respuestas.

Windows XP usa dos tipos de técnicas de paso de mensajes a través del puerto que el cliente especifique al establecer el canal. La más sencilla, que se usa para mensajes pequeños, usa la cola de mensajes del puerto como almacenamiento intermedio y copia el mensaje de un proceso a otro. Con este método, se pueden enviar mensajes de hasta 256 bytes.

Si un cliente necesita enviar un mensaje más grande, pasa el mensaje a través de un **objeto sección**, que configura una región de memoria compartida. El cliente tiene que decidir, cuando configura el canal, si va a tener que enviar o no un mensaje largo. Si el cliente determina que va a enviar mensajes largos, pide que se cree un objeto sección. Del mismo modo, si el servidor decide que la respuesta va a ser larga, crea un objeto sección. Para que el objeto sección pueda utilizarse, se envía un mensaje corto que contenga un puntero e información sobre el tamaño del objeto sección. Este método es más complicado que el primero, pero evita la copia de datos. En ambos casos, puede emplearse un mecanismo de retrollamada si el cliente o el servidor no pueden responder inmediatamente a una solicitud. El mecanismo de retrollamada les permite hacer un tratamiento asíncrono de los mensajes. La estructura de las llamadas a procedimientos locales en Windows XP se muestra en la Figura 3.17.

Es importante observar que la facilidad LPC de Windows XP no forma parte de la API de Win32 y, por tanto, no es visible para el programador de aplicaciones. En su lugar, las aplicaciones que usan la API de Win32 invocan las llamadas a procedimiento remoto estándar. Cuando la llamada RPC se invoca sobre un proceso que resida en el mismo sistema, dicha llamada se gestiona indirectamente a través de una llamada a procedimiento local. Las llamadas a procedimiento local también se usan en algunas otras funciones que forman parte de la API de Win32.

3.6 Comunicación en los sistemas cliente-servidor

En la Sección 3.4 hemos descrito cómo pueden comunicarse los procesos utilizando las técnicas de memoria compartida y de paso de mensajes. Estas técnicas también pueden emplearse en los sistemas cliente-servidor (Sección 1.12.2) para establecer comunicaciones. En esta sección, exploraremos otras tres estrategias de comunicación en los sistemas cliente-servidor: *sockets*, llamadas a procedimientos remotos (RPC) e invocación de métodos remotos de Java (RMI, remote method invocation).

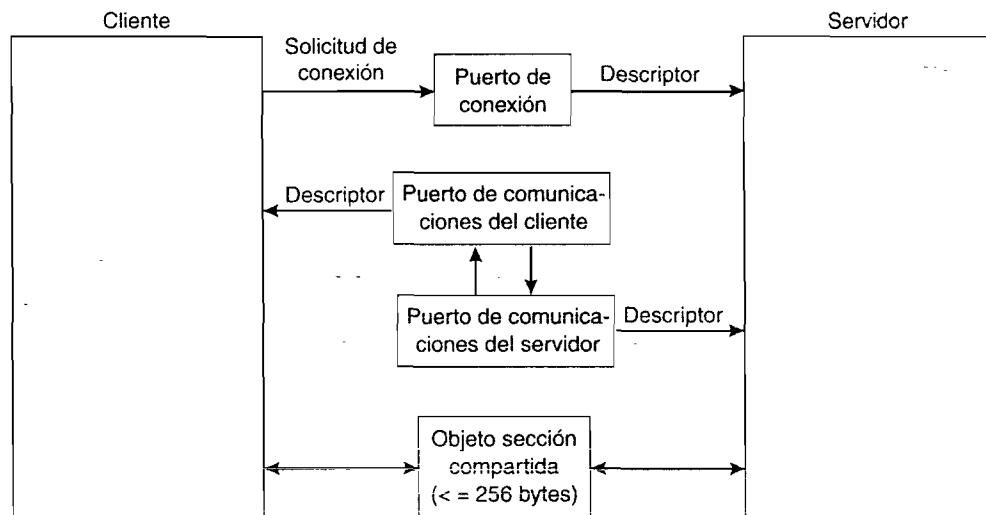


Figura 3.17 Llamadas a procedimiento locales en Windows XP.

3.6.1 Sockets

Un **socket** se define como un punto terminal de una comunicación. Una pareja de procesos que se comunican a través de una red emplea una pareja de *sockets*, uno para cada proceso. Cada *socket* se identifica mediante una dirección IP concatenada con un número de puerto. En general, los *sockets* usan una arquitectura cliente-servidor: el servidor espera a que entren solicitudes del cliente, poniéndose a la escucha en un determinado puerto. Una vez que se recibe una solicitud, el servidor acepta una conexión del *socket* cliente y la conexión queda establecida. Los servidores que implementan servicios específicos (como telnet, ftp y http) se ponen a la escucha en puertos bien conocidos (un servidor telnet escucha en el puerto 23, un servidor ftp escucha en el puerto 21 y un servidor web o http escucha en el puerto 80). Todos los puertos por debajo de 1024 se consideran *bien conocidos* y podemos emplearlos para implementar servicios estándar.

Cuando un proceso cliente inicia una solicitud de conexión, la computadora *host* le asigna un puerto. Este puerto es un número arbitrario mayor que 1024. Por ejemplo, si un cliente en un *host* X con la dirección IP 146.86.5.20 desea establecer una conexión con un servidor web (que está escuchando en el puerto 80) en la dirección 161.25.19.8, puede que al *host* X se le asigne el puerto 1625. La conexión constará de una pareja de *sockets*: (146.86.5.20:1625) en el *host* X y (161.25.19.8:80) en el servidor web. Esta situación se ilustra en la Figura 3.18. Los paquetes que viajan entre los *hosts* se suministran al proceso apropiado, según el número de puerto de destino.

Todas las conexiones deben poderse diferenciar. Por tanto, si otro proceso del *host* X desea establecer otra conexión con el mismo servidor web, deberá asignarse a ese proceso un número de puerto mayor que 1023 y distinto de 1625. De este modo, se garantiza que todas las conexiones dispongan de una pareja distintiva de *sockets*.

Aunque la mayor parte de los ejemplos de programas de este texto usan C, ilustraremos los *sockets* con Java, ya que este lenguaje proporciona una interfaz mucho más fácil para los *sockets* y dispone de una biblioteca muy rica en lo que se refiere a utilidades de red. Aquellos lectores que estén interesados en la programación de *sockets* en C o C++ pueden consultar las notas bibliográficas incluidas al final del capítulo.

Java proporciona tres tipos diferentes de *sockets*. Los *sockets TCP orientados a conexión* se implementan con la clase *Socket*. Los *sockets UDP sin conexión* usan la clase *DatagramSocket*. Por último, la clase *MulticastSocket* (utilizada para multidifusión) es una subclase de *DatagramSocket*. Un *socket* multidifusión permite enviar datos a varios receptores.

Nuestro ejemplo describe un servidor de datos que usa *sockets TCP orientados a conexión*. La operación permite a los clientes solicitar la fecha y la hora actuales al servidor. El servidor escucha en el puerto 6013, aunque el puerto podría usar cualquier número arbitrario mayor que 1024. Cuando se recibe una conexión, el servidor devuelve la fecha y la hora al cliente.

En la Figura 3.19 se muestra el servidor horario. El servidor crea un *ServerSocket* que especifica que se pondrá a la escucha en el puerto 6013. El servidor comienza entonces a escuchar en

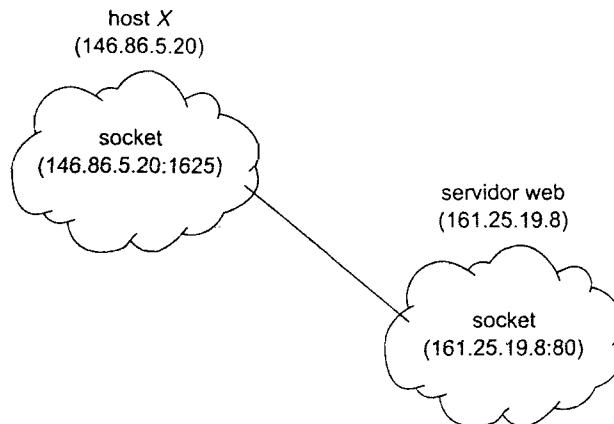


Figura 3.18 Comunicación usando sockets.

el puerto con el método `accept()`. El servidor se bloquea en el método `accept()` esperando a que un cliente solicite una conexión. Cuando se recibe una solicitud, `accept()` devuelve un *socket* que el servidor puede usar para comunicarse con el cliente.

Veamos los detalles de cómo se comunica el servidor con el *socket*. En primer lugar, el servidor establece un objeto `PrintWriter` que se usará para comunicarse con el cliente. Un objeto `PrintWriter` permite al servidor escribir en el *socket* usando como métodos de salida las rutinas `print()` y `println()`. El proceso de servidor envía la fecha al cliente llamando al método `println()`. Una vez que ha escrito la fecha en el *socket*, el servidor cierra el *socket* de conexión con el cliente y continúa escuchando para detectar más solicitudes.

Un cliente se comunica con el servidor creando un *socket* y conectándose al puerto en el que el servidor está escuchando. Podemos implementar tal cliente con el programa Java que se muestra en la Figura 3.20. El cliente crea un *socket* y solicita una conexión con el servidor de la dirección IP 127.0.0.1 a través del puerto 6013. Una vez establecida la conexión, el cliente puede leer en el *socket* usando instrucciones de E/S normales. Después de recibir los datos del servidor, el cliente cierra el *socket* y sale. La dirección IP 127.0.0.1 es una dirección IP especial conocida como **dirección de bucle**. Cuando una computadora hace referencia a la dirección IP 127.0.0.1, se está haciendo referencia a sí misma. Este mecanismo permite a un cliente y un servidor de un mismo *host* comunicarse usando el protocolo TCP/IP. La dirección IP 127.0.0.1 puede reemplazarse por la dirección IP de otro *host* que ejecute el servidor horario. En lugar de una dirección IP, también puede utilizarse un nombre de *host* real, como *www.westminstercollege.edu*.

```

import java.net.*;
import java.io.*;

public class DateServer

{
    public static void main(String[] args) {
        try {

            ServerSocket sock = new ServerSocket(6013);

            // escuchar para detectar conexiones
            while (true) {
                Socket client = sock.accept();

                PrintWriter pout = new
                PrintWriter(client.getOutputStream(), true);

                // escribir la fecha en el socket
                pout.println(new java.util.Date().toString());

                // cerrar el socket y reanudar
                // la escucha para detectar conexiones
                client.close();

            }
        }
    }
}

```

Figura 3.19 Servidor horario.

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            //establece la conexión con el socket del servidor
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader(new InputStreamReader(in));

            // lee la fecha en el socket
            String line;
            while ( (line=bin.readLine()) !=null)
                System.out.println(line);

            // cierra la conexión del socket
            sock.close();

        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

Figura 3.20 Cliente horario.

La comunicación a través de *sockets*, aunque habitual y eficiente, se considera una forma de bajo nivel de comunicación entre procesos distribuidos. Una razón es que los *sockets* sólo permiten que se intercambie un flujo no estructurado de bytes entre las hebras en comunicación. Es responsabilidad de la aplicación cliente o servidor imponer una estructura a los datos. En las dos secciones siguientes veremos dos métodos de comunicación de mayor nivel: las llamadas a procedimiento remoto (RPC) y la invocación de métodos remotos (RMI).

3.6.2 Llamadas a procedimientos remotos

Una de las formas más comunes de prestar servicios remotos es el uso de las llamadas a procedimiento remoto (RPC), que hemos explicado brevemente en la Sección 3.5.2. Las RPC se diseñaron como un método para abstraer los mecanismos de llamada a procedimientos, con el fin de utilizarlos entre sistemas conectados en red. Son similares en muchos aspectos al mecanismo IPC descrito en la Sección 3.4 y, normalmente, se implementan por encima de dicho mecanismo. Sin embargo, dado que vamos a tratar con un entorno en el que los procesos se ejecutan en sistemas separados, debemos emplear un esquema de comunicación basado en mensajes para proporcionar el servicio remoto. Al contrario que con la facilidad IPC, los mensajes intercambiados en la comunicación mediante RPC están bien estructurados y, por tanto, no son simples paquetes de datos. Cada mensaje se dirige a un demonio RPC que escucha en un puerto del sistema remoto, y cada uno contiene un identificador de la función que se va a ejecutar y los parámetros que hay que pasar a dicha función. La función se ejecuta entonces de la forma solicitada y se devuelven los posibles datos de salida a quien haya efectuado la solicitud usando un mensaje diferente.

Un *puerto* es simplemente un número incluido al principio del paquete de mensaje. Aunque un sistema normalmente sólo tiene una dirección de red, puede tener muchos puertos en esa dirección para diferenciar los distintos servicios de red que soporta. Si un proceso remoto necesita un

servicio, envía un mensaje al puerto apropiado. Por ejemplo, si un sistema deseara permitir a otros sistemas que pudieran ver la lista de sus usuarios actuales, podría definir un demonio para el servicio RPC asociado a un puerto, como por ejemplo, el puerto 3027. Cualquier sistema remoto podría obtener la información necesaria (es decir, la lista de los usuarios actuales) enviando un mensaje RPC al puerto 3027 del servidor; los datos se recibirían en un mensaje de respuesta.

La semántica de las llamadas RPC permite a un cliente invocar un procedimiento de un modo remoto del mismo modo que invocaría un procedimiento local. El sistema RPC oculta los detalles que permiten que tenga lugar la comunicación, proporcionando un *stub* en el lado del cliente. Normalmente, existe un *stub* diferente para cada procedimiento remoto. Cuando el cliente invoca un procedimiento remoto, el sistema RPC llama al *stub* apropiado, pasándole los parámetros que hay que proporcionar al procedimiento remoto. Este *stub* localiza el puerto en el servidor y envuelve los parámetros. Envolver los parámetros quiere decir empaquetarlos en un formato que permite su transmisión a través de la red (en inglés, el término utilizado para el proceso de envolver los parámetros es *marshalling*). El *stub* transmite un mensaje al servidor usando el método de paso de mensajes. Un *stub* similar en el lado del servidor recibe este mensaje e invoca al procedimiento en el servidor. Si es necesario, los valores de retorno se pasan de nuevo al cliente usando la misma técnica.

Una cuestión de la que hay que ocuparse es de las diferencias en la representación de los datos entre las máquinas cliente y servidor. Considere las distintas formas de representar los enteros de 32 bits. Algunos sistemas (conocidos como *big-endian*) usan la dirección de memoria superior para almacenar el byte más significativo, mientras que otros sistemas (conocidos como *little-endian*) almacenan el byte menos significativo en la dirección de memoria superior. Para resolver diferencias como ésta, muchos sistemas RPC definen una representación de datos independiente de la máquina. Una representación de este tipo se denomina **representación de datos externa** (*external data representation*). En el lado del cliente, envolver los parámetros implica convertir los datos dependientes de la máquina en una representación externa antes de enviar los datos al servidor. En el lado del servidor, los datos XDR se desenvuelven y convierten a la representación dependiente de la máquina utilizada en el servidor.

Otra cuestión importante es la semántica de una llamada. Mientras que las llamadas a procedimientos locales fallan en circunstancias extremas, las llamadas RPC pueden fallar, o ser duplicadas y ejecutadas más de una vez, como resultado de errores habituales de red. Una forma de abordar este problema es que el sistema operativo garantice que se actúe en respuesta a los mensajes *exactamente una vez*, en lugar de *como máximo una vez*. La mayoría de las llamadas a procedimientos locales presentan la característica de ejecutarse “exactamente una vez”, aunque esta característica es más difícil de implementar.

En primer lugar, considere el caso de “como máximo una vez”. Esta semántica puede garantizarse asociando una marca temporal a cada mensaje. El servidor debe mantener un historial de todas las marcas temporales de los mensajes que ya ha procesado o un historial lo suficientemente largo como para asegurar que se detecten los mensajes repetidos. Los mensajes entrantes tengan una marca temporal que ya esté en el historial se ignoran. El cliente puede entonces enviar un mensaje una o más veces y estar seguro de que sólo se ejecutará una vez. En la Sección 18.4 se estudia la generación de estas marcas temporales.

En el caso de “exactamente una vez”, necesitamos eliminar el riesgo de que el servidor no reciba la solicitud. Para ello, el servidor debe implementar el protocolo de “como máximo una vez” descrito anteriormente, pero también tiene que confirmar al cliente que ha recibido y ejecutado la llamada RPC. Estos mensajes de confirmación (ACK, acknowledge) son comunes en las redes. El cliente debe reenviar cada llamada RPC periódicamente hasta recibir la confirmación de la llamada.

Otro tema importante es el que se refiere a la comunicación entre un servidor y un cliente. Las llamadas a procedimiento estándar se realiza algún tipo de asociación de variables durante el montaje, la carga o la ejecución (Capítulo 8), de manera que cada nombre de llamada a procedimiento es reemplazado por la dirección de memoria de la llamada al procedimiento. El esquema de RPC requiere una asociación similar de los puertos del cliente y el servidor, pero ¿cómo puede conocer un cliente el número de puerto del servidor? Ningún sistema dispone de información completa sobre el otro, ya que no comparten la memoria.

Existen dos métodos distintos. Primero, la información de asociación puede estar predeterminada en forma de direcciones fijas de puerto. En tiempo de compilación, una llamada a procedimiento remoto tiene un número de puerto fijo asociado a ella. Una vez que se ha compilado un programa, el servidor no puede cambiar el número de puerto del servicio solicitado. La segunda posibilidad es realizar la asociación de forma dinámica mediante un mecanismo de negociación. Normalmente, los sistemas operativos proporcionan un demonio de *rendezvous* (también denominado **matchmaker**) en un puerto RPC fijo. El cliente envía entonces un mensaje que contiene el nombre de la llamada RPC al demonio de *rendezvous*, solicitando la dirección de puerto de la llamada RPC que necesita ejecutar. El demonio devuelve el número de puerto y las llamadas a procedimientos remotos pueden enviarse a dicho puerto hasta que el proceso termine (o el servidor falle). Este método impone la carga de trabajo adicional correspondiente a la solicitud inicial, pero es más flexible que el primer método. La Figura 3.21 muestra un ejemplo de este tipo de interacción.

El esquema RPC resulta muy útil en la implementación de sistemas de archivos distribuidos (Capítulo 17). Un sistema de este tipo puede implementarse como un conjunto de demonios y clientes RPC. Los mensajes se dirigen al puerto del sistema de archivos distribuido del servidor en

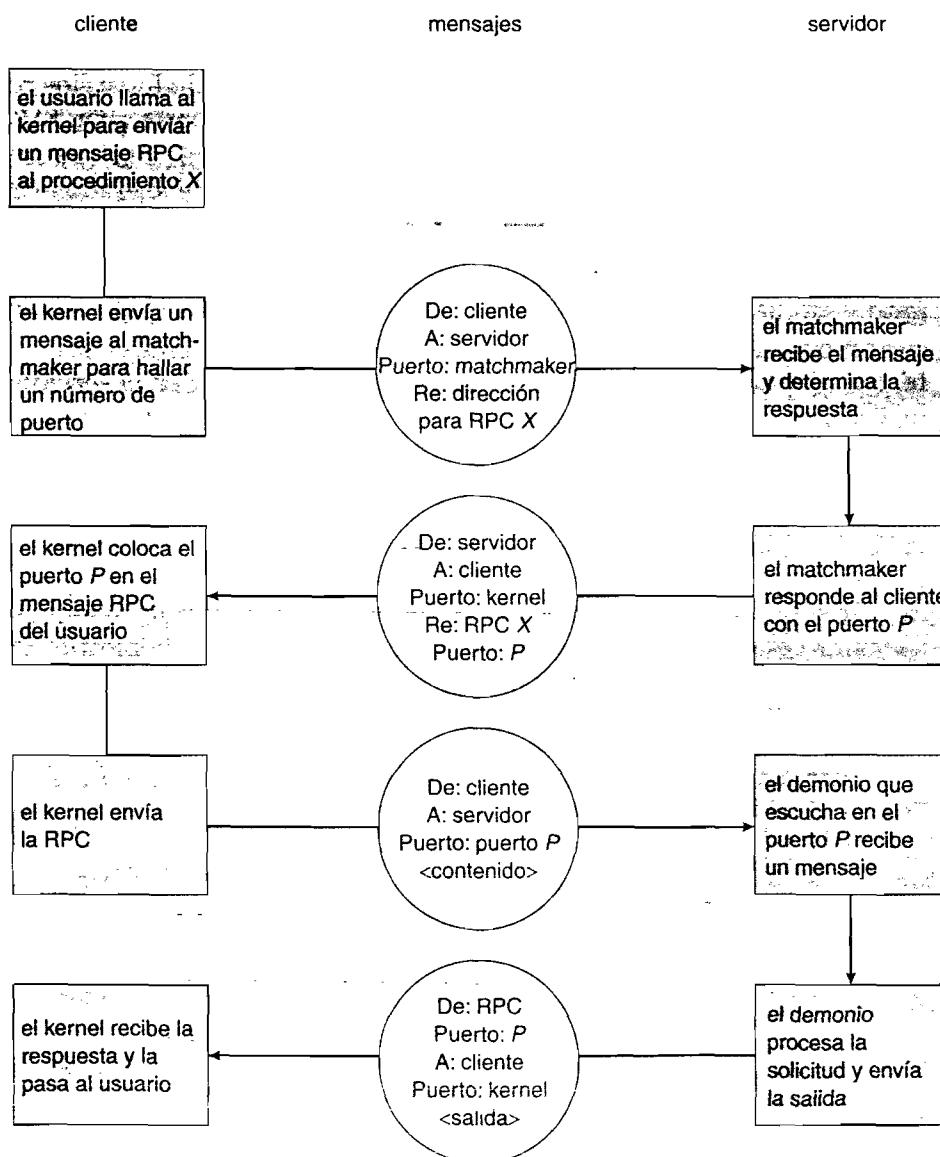


Figura 3.21 Ejecución de una llamada a procedimiento remoto (RPC).

el que deba realizarse una operación sobre un archivo; el mensaje contiene la operación de disco que se desea realizar. La operación de disco puede ser de lectura (read), escritura (write), cambio de nombre (rename), borrado (delete) o estado (status), las cuales se corresponden con las usuales llamadas al sistema relacionadas con archivos. El mensaje de respuesta contiene cualquier dato resultante de dicha llamada, que es ejecutada por el demonio del sistema de archivos distribuido por cuenta del cliente. Por ejemplo, un mensaje puede contener una solicitud para transferir un archivo completo a un cliente o limitarse a una simple solicitud de un bloque. En el último caso, pueden ser necesarias varias solicitudes de dicho tipo si se va a transferir un archivo completo.

3.6.3 Invocación de métodos remotos

La **invocación de métodos remotos** (RMI, remote method invocation) es una funcionalidad Java similar a las llamadas a procedimientos remotos. RMI permite a una hebra invocar un método sobre un objeto remoto. Los objetos se consideran remotos si residen en una máquina virtual Java diferente. Por tanto, el objeto remoto puede estar en una JVM diferente en la misma computadora o en un *host* remoto conectado a través de una red. Esta situación se ilustra en la Figura 3.22.

Los sistemas RMI y RPC difieren en dos aspectos fundamentales. En primer lugar, el mecanismo RPC soporta la programación procedimental, por lo que sólo se puede llamar a *procedimientos* o *funciones* remotas. Por el contrario, el mecanismo RMI se basa en objetos: permite la invocación de *métodos* correspondientes a objetos remotos. En segundo lugar, los parámetros para los procedimientos remotos en RPC son estructuras de datos ordinarias; con RMI, es posible pasar objetos como parámetros a los métodos remotos. Permitiendo a un programa Java invocar métodos sobre objetos remotos, RMI hace posible que los usuarios desarrollen aplicaciones Java distribuidas a través de una red.

Para hacer que los métodos remotos sean transparentes tanto para el cliente como para el servidor, RMI implementa el objeto remoto utilizando *stubs* y esqueletos. Un *stub* es un *proxy* para el objeto remoto y reside en el cliente. Cuando un cliente invoca un método remoto, se llama al *stub* correspondiente al objeto remoto. Este *stub* del lado del cliente es responsable de crear un **paquete**, que consta del nombre del método que se va a invocar en el servidor y de los parámetros del método, debidamente envueltos. El *stub* envía entonces ese paquete al servidor, donde el esqueleto correspondiente al objeto remoto lo recibe. El **esqueleto** es responsable de desenvolver los parámetros y de invocar el método deseado en el servidor. El esqueleto envuelve entonces el valor de retorno (o la excepción, si existe) en un paquete y lo devuelve al cliente. El *stub* desenvuelve el valor de retorno y se lo pasa al cliente.

Veamos más en detalle cómo opera este proceso. Suponga que un cliente desea invocar un método en un servidor de objetos remotos, y que ese método tiene la firma `algunMetodo(Object, Object)`, devolviendo un valor booleano. El cliente ejecuta la instrucción:

```
boolean val = servidor.algunMetodo(A, B);
```

La llamada a `algunMetodo()` con los parámetros A y B invoca al *stub* para al objeto remoto. El *stub* envuelve los parámetros A y B y el nombre del método que va invocarse en el servidor, y

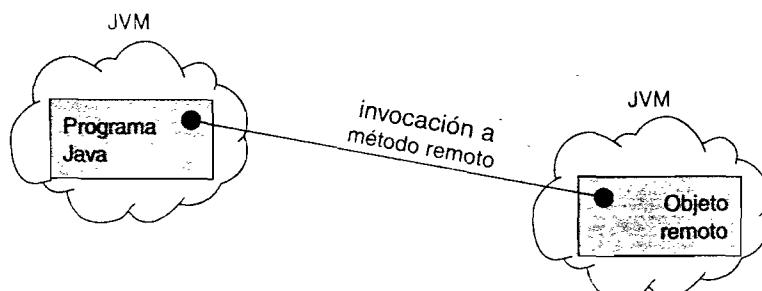


Figura 3.22 Invocación de métodos remotos.

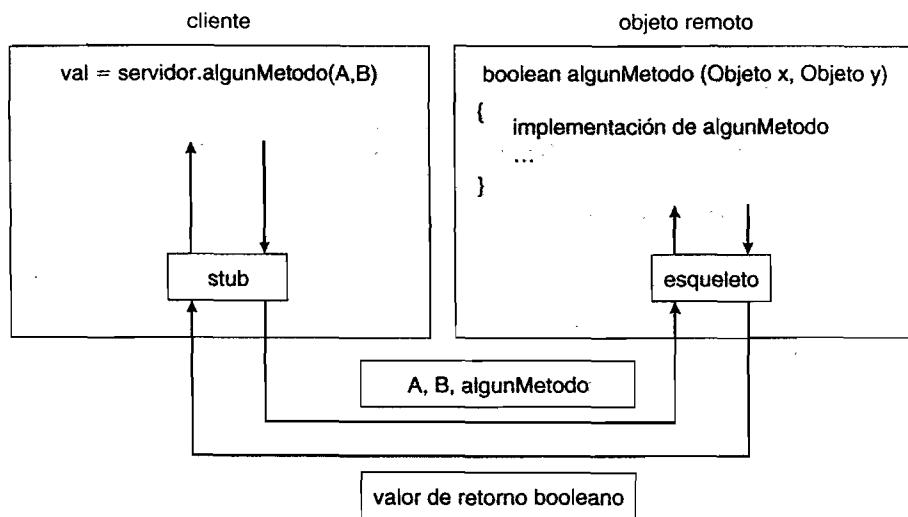


Figura 3.23 Envoltura de parámetros.

envía ese envoltorio al servidor. El esqueleto situado en el servidor desenvuelve los parámetros e invoca al método `algunMetodo()`. La implementación real de `algunMetodo()` reside en el servidor. Una vez que el método se ha completado, el esqueleto envuelve el valor booleano devuelto por `algunMetodo()` y envía de vuelta ese valor al cliente. El *stub* desenvuelve ese valor de retorno y se lo pasa al cliente. El proceso se muestra en la Figura 3.23.

Afortunadamente, el nivel de abstracción que RMI proporciona hace que los *stubs* y esqueletos sean transparentes, permitiendo a los desarrolladores Java escribir programas que invoquen métodos distribuidos del mismo modo que invocarían métodos locales. Sin embargo, es fundamental comprender unas pocas reglas sobre el comportamiento del paso de parámetros:

- Si los parámetros envueltos son objetos **locales**, es decir, no remotos, se pasan mediante copia, empleando una técnica conocida como **serialización de objetos**. Sin embargo, si los parámetros también son objetos remotos, se pasan por referencia. En nuestro ejemplo, si A es un objeto local y B es un objeto remoto, A se serializa y se pasa por copia y B se pasa por referencia. Esto, a su vez, permite al servidor invocar métodos sobre B de forma remota.
- Si se van a pasar objetos locales como parámetros a objetos remotos, esos objetos locales deben implementar la interfaz `java.io.Serializable`. Muchos objetos básicos de la API de Java implementan la interfaz `Serializable`, lo que permite utilizarlos con el mecanismo RMI. La serialización de objetos permite escribir el estado de un objeto en forma de flujo de bytes.

3.7 Resumen

Un proceso es un programa en ejecución. Cuando un proceso se ejecuta, cambia de estado. El estado de un proceso se define en función de la actividad actual del mismo. Cada proceso puede estar en uno de los siguientes estados: nuevo, preparado, en ejecución, en espera o terminado. Cada proceso se representa en el sistema operativo mediante su propio bloque de control de proceso (PCB).

Un proceso, cuando no se está ejecutando, se encuentra en alguna cola en espera. Existen dos clases principales de colas en un sistema operativo: colas de solicitudes de E/S y cola de procesos preparados. Esta última contiene todos los procesos que están preparados para ejecutarse y están esperando a que se les asigne la CPU. Cada proceso se representa mediante un bloque PCB y los PCB se pueden enlazar para formar una cola de procesos preparados. La planificación a largo plazo (trabajos) es la selección de los procesos a los que se permitirá contender por la CPU.

Normalmente, la planificación a largo plazo se ve extremadamente influenciada por las consideraciones de asignación de recursos, especialmente por la gestión de memoria. La planificación a corto plazo (CPU) es la selección de un proceso de la cola de procesos preparados.

Los sistemas operativos deben proporcionar un mecanismo para que los procesos padre creen nuevos procesos hijo. El padre puede esperar a que sus hijos terminen antes de continuar, o el padre y los hijos pueden ejecutarse de forma concurrente. Existen varias razones para permitir la ejecución concurrente: compartición de información, aceleración de los cálculos, modularidad y comodidad.

Los procesos que se ejecutan en el sistema operativo pueden ser procesos independientes o procesos cooperativos. Los procesos cooperativos requieren un mecanismo de comunicación interprocesos para comunicarse entre sí. Fundamentalmente, la comunicación se consigue a través de dos esquemas: memoria compartida y paso de mensajes. El método de memoria compartida requiere que los procesos que se van a comunicar comparten algunas variables; los procesos deben intercambiar información a través del uso de estas variables compartidas. En un sistema de memoria compartida, el proporcionar mecanismos de comunicación es responsabilidad de los programadores de la aplicación; el sistema operativo sólo tiene que proporcionar la memoria compartida. El método de paso de mensajes permite a los procesos intercambiar mensajes; la responsabilidad de proporcionar mecanismos de comunicación corresponde, en este caso, al propio sistema operativo. Estos esquemas no son mutuamente exclusivos y se pueden emplear simultáneamente dentro de un mismo sistema operativo.

La comunicación en los sistemas cliente-servidor puede utilizar (1) *sockets*, (2) llamadas a procedimientos remotos (RPC) o (3) invocación de métodos remotos (RMI) de Java. Un *socket* se define como un punto terminal para una comunicación. Cada conexión entre un par de aplicaciones consta de una pareja de *sockets*, uno en cada extremo del canal de comunicación. Las llamadas RPC constituyen otra forma de comunicación distribuida; una llamada RPC se produce cuando un proceso (o hebra) llama a un procedimiento de una aplicación remota. El mecanismo RMI es la versión Java de RPC. Este mecanismo de invocación de métodos remotos permite a una hebra invocar un método sobre un objeto remoto, del mismo modo que invocaría un método sobre un objeto local. La principal diferencia entre RPC y RMI es que en el primero de esos dos mecanismos se pasan los datos al procedimiento remoto usando una estructura de datos ordinaria, mientras que la invocación de métodos remotos permite pasar objetos en las llamadas a los métodos remotos.

Ejercicios

- 3.1 Describa las diferencias entre la planificación a corto plazo, la planificación a medio plazo y la planificación a largo plazo.
- 3.2 Describa las acciones tomadas por un *kernel* para el cambio de contexto entre procesos.
- 3.3 Considere el mecanismo de las llamadas RPC. Describa las consecuencias no deseables que se producirían si no se forzara la semántica “como máximo una vez” o “exactamente una vez”. Describa los posibles usos de un mecanismo que no presente ninguna de estas garantías.
- 3.4 Usando el programa mostrado en la Figura 3.24, explique cuál será la salida en la Línea A.
- 3.5 ¿Cuáles son las ventajas e inconvenientes en cada uno de los casos siguientes? Considere tanto el punto de vista del sistema como el del programador.
 - a. Comunicación síncrona y asíncrona.
 - b. Almacenamiento en búfer automático y explícito.
 - c. Envío por copia y envío por referencia.
 - d. Mensajes de tamaño fijo y de tamaño variable.
- 3.6 La secuencia de Fibonacci es la serie de números 0, 1, 1, 2, 3, 5, 8, ... Formalmente, se expresa como sigue:

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

    pid = fork();

    if (pid == 0) /* proceso hijo*/
        value +=15;
    }
    else if (pid > 0) /* proceso padre */
        wait(NULL);
        printf ("PARENT: value = %d", value); /* LÍNEA A */
        exit(0);
    }
}

```

Figura 3.24 Programa C

$$\begin{aligned}
 fib_0 &\equiv 0 \\
 fib_1 &\equiv 1 \\
 fib_n &= fib_{n-1} + fib_{n-2}
 \end{aligned}$$

Escriba un programa C, usando la llamada al sistema `fork()`, que genere la secuencia de Fibonacci en el proceso hijo. El límite de la secuencia se proporcionará a través de la línea de comandos. Por ejemplo, si se especifica 5, el proceso hijo proporcionará los primeros cinco números de la secuencia de Fibonacci como salida. Dado que los procesos padre e hijo disponen de sus propias copias de los datos, será necesario que el hijo presente la secuencia de salida. El padre tendrá que invocar la función `wait()` para esperar a que el proceso hijo se complete antes de salir del programa. Realice las comprobaciones de errores necesarias para asegurar que no se pase un número negativo a través de la línea de comandos.

- 3.7 Repita el ejercicio anterior, pero esta vez utilizando la función `CreateProcess()` de la API Win32. En este caso, tendrá que especificar un programa diferente para invocarlo con `CreateProcess()`. Dicho programa se ejecutará como proceso hijo y dará como salida la secuencia de Fibonacci. Realice las comprobaciones de errores necesarias para asegurar que no se pase un número negativo a través de la línea de comandos.
- 3.8 Modifique el servidor horario mostrado en la Figura 3.19 de modo que suministre mensajes de la suerte aleatorios en lugar de la fecha actual. Debe permitir que esos mensajes de la suerte contengan múltiples líneas. Puede utilizarse el cliente horario mostrado en la Figura 3.20 para leer los mensajes multilínea devueltos por el servidor de la suerte.
- 3.9 Un **servidor de eco** es un servidor que devuelve el eco de todo lo que recibe de un cliente. Por ejemplo, si un cliente envía al servidor la cadena `;Hola!`, el servidor responderá con los mismos datos que ha recibido del cliente, es decir, `;Hola!`

Escriba un servidor de eco usando la API de red Java descrita en la Sección 3.6.1. Este servidor esperará a que un cliente establezca una conexión, usando para ello el método `accept()`. Cuando se reciba una conexión de cliente, el servidor entrará en un bucle, ejecutando los pasos siguientes:

- Leer los datos del *socket* y ponerlos en un búfer.
- Escribir de nuevo el contenido del búfer para devolverlo al cliente.

El servidor saldrá del bucle sólo cuando haya determinado que el cliente ha cerrado la conexión.

El servidor horario mostrado en la Figura 3.19 usa la clase `java.io.BufferedReader`. La clase `BufferedReader` amplía la clase `java.io.Reader`, que se usa para leer flujos de caracteres. Sin embargo, el servidor de eco no puede estar seguro de que que se reciba de los clientes sean caracteres; también puede recibir datos binarios. La clase `java.io.InputStream` procesa datos en el nivel de byte en lugar de en el nivel de carácter; por tanto, este servidor de eco debe usar un objeto que amplíe `java.io.InputStream`. El método `read()` en la clase `java.io.InputStream` devuelve - cuando el cliente ha cerrado su extremo del *socket*.

- 3.10** En el Ejercicio 3.6, el proceso hijo proporcionaba como salida la secuencia de Fibonacci dado que el padre y el hijo disponían de sus propias copias de los datos. Otro método para diseñar este programa consiste en establecer un segmento de memoria compartida entre los procesos padre e hijo. Esta técnica permite al hijo escribir el contenido de la secuencia de Fibonacci en el segmento de memoria compartida, para que el padre proporcione como salida la secuencia cuando el hijo termine de ejecutarse. Dado que la memoria se comparte, cualquier cambio que el hijo hace en la memoria compartida se refleja también en el proceso padre.

Este programa se estructurará usando la memoria compartida POSIX que se ha descrito en la Sección 3.5.1. En primer lugar, el programa requiere crear la estructura de datos para el segmento de memoria compartida; la mejor forma de hacer esto es usando una estructura (`struct`). Esta estructura de datos contendrá dos elementos: (1) una matriz de tamaño fijo `MAX_SEQUENCE`, que contendrá los valores de Fibonacci y (2) el tamaño de la secuencia que el proceso hijo debe generar, `sequence_size`, donde `sequence_size ≤ MAX_SEQUENCE`. Estos elementos se pueden representar en una estructura como sigue:

```
#define MAX_SEQUENCE 10

typedef struct {
    long fib_sequence[MAX_SEQUENCE];
    int sequence_size;
} shared_data;
```

El proceso padre realizará los pasos siguientes:

- Aceptar el parámetro pasado a través de la línea de comandos y realizar la comprobación de errores que asegure que el parámetro es $\leq \text{MAX_SEQUENCE}$.
- Crear un segmento de memoria compartida de tamaño `shared_data`.
- Asociar el segmento de memoria compartida a su espacio de direcciones.
- Asignar a `sequence_size` un valor igual al parámetro de la línea de comandos.
- Bifurcar el proceso hijo e invocar la llamada al sistema `wait()` para esperar a que el proceso hijo concluya.
- Llevar a la salida la secuencia de Fibonacci contenida en el segmento de memoria compartida.
- Desasociar y eliminar el segmento de memoria compartida.

Dado que el proceso hijo es una copia del padre, la región de memoria compartida se asociará también al espacio de direcciones del hijo. El hijo escribirá entonces la secuencia de Fibonacci en la memoria compartida y, finalmente, desasociará el segmento.

Un problema que surge con los procesos cooperativos es el relativo a la sincronización. En este ejercicio, los procesos padre e hijo deben estar sincronizados, de modo que el padre no lleve a la salida la secuencia de Fibonacci hasta que el proceso hijo haya terminado de generar la secuencia. Estos dos procesos se sincronizarán usando la llamada al sistema `wait()`; el proceso padre invocará dicha llamada al sistema, la cual hará que quede en espera hasta que el proceso hijo termine.

- 3.11** La mayor parte de los sistemas UNIX y Linux proporcionan el comando `ipcs`. Este comando proporciona el estado de diversos mecanismos de comunicación interprocesos de POSIX, incluyendo los segmentos de memoria compartida. Gran parte de la información que proporciona este comando procede de la estructura de datos `struct shmid_ds`, que está disponible en el archivo `/usr/include/sys/shm.h`. Algunos de los campos de esta estructura son:

- `int shm_segsz`— tamaño del segmento de memoria compartida.
- `short shm_nattch`— número de asociaciones al segmento de memoria compartida.
- `struct ipc_perm shm_perm`— estructura de permisos del segmento de memoria compartida.

La estructura de datos `struct ipc_perm`, disponible en el archivo `/usr/include/sys/ipc.h`, contiene los campos:

- `unsigned short uid`— identificador del usuario del segmento de memoria compartida.
- `unsigned short mode`— modos de permisos.
- `key_t key` (en sistemas Linux, `__key`)— identificador clave especificado por el usuario

Los modos de permiso se establecen según se haya definido el segmento de memoria compartida en la llamada al sistema `shmget()`. Los permisos se identifican de acuerdo con la siguiente tabla:

modo	significado
0400	Permiso de lectura del propietario.
0200	Permiso de escritura del propietario.
0040	Permiso de lectura de grupo.
0020	Permiso de escritura de grupo.
0004	Permiso de lectura de todos.
0002	Permiso de escritura de todos

Se puede acceder a los permisos usando el operador `AND` bit a bit `&`. Por ejemplo, si la expresión `mode & 0400` se evalúa como verdadera, se concede permiso de lectura al propietario del segmento de memoria compartida.

Los segmentos de memoria compartida pueden identificarse mediante una clave especificada por el usuario o mediante un valor entero devuelto por la llamada al sistema `shmget()`, que representa el identificador entero del segmento de memoria compartida recién creado. La estructura `shm_ds` para un determinado identificador entero de segmento puede obtenerse mediante la siguiente llamada al sistema:

```
/* identificador del segmento de memoria compartida */
int segment_id;
shm_ds shmbuffer;

shmctl(segment_id, IPC_STAT, &shmbuffer);
```

Si se ejecuta con éxito, `shmctl()` devuelve 0; en caso contrario, devuelve -1.

Escriba un programa C al que se le pase el identificador de un segmento de memoria compartida. Este programa invocará la función `shmctl()` para obtener su estructura `shm_ds`. Luego proporcionará como salida los siguientes valores del segmento de memoria compartida especificado:

- ID del segmento
- Clave
- Modo
- UID del propietario
- Tamaño
- Número de asociaciones

Proyecto: shell de UNIX y función histrial

Este proyecto consiste en modificar un programa C utilizado como interfaz *shell*, que acepta comandos de usuario y luego ejecuta cada comando como un proceso diferente. Una interfaz *shell* proporciona al usuario un indicativo de comandos en el que el usuario puede introducir los comandos que deseé ejecutar. El siguiente ejemplo muestra el indicativo de comandos `sh>`, en el que se ha introducido el comando de usuario `cat prog.c`. Este comando muestra el archivo `prog.c` en el terminal usando el comando `cat` de UNIX.

```
sh> cat prog.c
```

Una técnica para implementar una interfaz *shell* consiste en que primero el proceso padre lea lo que el usuario escribe en la línea de comandos (por ejemplo, `cat prog.c`), y luego cree un proceso hijo separado que ejecute el comando. A menos que se indique lo contrario, el proceso padre espera a que el hijo termine antes de continuar. Esto es similar en cuanto a funcionalidad al esquema mostrado en la Figura 3.11. Sin embargo, normalmente, las *shell* de UNIX también permiten que el proceso hijo se ejecute en segundo plano o concurrentemente, especificando el símbolo & al final del comando. Reescribiendo el comando anterior como:

```
sh> cat prog.c &
```

los procesos padre e hijo se ejecutarán de forma concurrente.

El proceso hijo se crea usando la llamada al sistema `fork()` y el comando de usuario se ejecuta utilizando una de las llamadas al sistema de la familia `exec*` (como se describe en la Sección 3.3.1).

Shell simple

En la Figura 3.25 se incluye un programa en C que proporciona las operaciones básicas de una *shell* de línea de comandos. Este programa se compone de dos funciones: `main()` y `setup()`. La función `setup()` lee el siguiente comando del usuario (que puede constar de hasta 80 caracteres), lo analiza sintácticamente y lo descompone en identificadores separados que se usan para llenar el vector de argumentos para el comando que se va a ejecutar. (Si el comando se va a ejecutar en segundo plano, terminará con '&' y `setup()` actualizará el parámetro `background` de modo que la función `main()` pueda operar de acuerdo con ello. Este programa se termina cuando el usuario introduce `<Control-D>` y `setup()` invoca, como consecuencia, `exit()`.

La función `main()` presenta el indicativo de comandos `COMMAND->` y luego invoca `setup()`, que espera a que el usuario escriba un comando. Los contenidos del comando introducido por el usuario se cargan en la matriz `args`. Por ejemplo, si el usuario escribe `ls -1` en el indicativo `COMMAND->`, se asigna a `args[0]` la cadena `ls` y se asigna a `args[1]` el valor `-1`. Por “cadena”, queremos decir una variable de cadena de caracteres de estilo C, con carácter de terminación nulo.

```

#include <stdio.h>
#include <unistd.h>

#define MAX_LINE 80

/** setup() lee la siguiente línea de comandos, y la separa en
distintos identificadores, usando los espacios en blanco como delimitado-
res. setup() modifica el parámetro args para que almacene punteros a las
cadenas de caracteres con terminación nula que constituyen los identifica-
dores de la línea de comandos de usuario más reciente, así como un
puntero NULL que indica el final de la lista de argumentos; ese puntero
nulo se incluye después de los punteros de cadena de caracteres
asignados a args */

void setup(char inputBuffer[], char *args[], int *background)
{
    /** el código fuente completo está disponible en línea */
}

int main(void)
{
char inputBuffer[MAX_LINE]; /* búfer para almacenar los comandos
introducidos */
int background; /* igual a 1 si el comando termina con '&' */
char *args[MAX_LINE/2 + 1]; /* argumentos de la línea de comandos */

while (1) {
    background = 0;
    printf(" COMMAND->");
    /* setup() llama a exit() cuando se pulsa Control-D */
    setup(inputBuffer, args, &background);

    /** los pasos son;
    (1) bifurcar un proceso hijo usando fork()
    (2) el proceso hijo invocará execvp()
    (3) si background == 1, el padre esperará;
    en caso contrario, invocará de nuevo la función setup() */
}
}

```

Figura 3.25 Diseño de una *shell* simple.

Este proyecto está organizado en dos partes: (1) crear el proceso hijo y ejecutar el comando en este proceso y (2) modificar la *shell* para disponer de una función histórica.

Creación de un proceso hijo

La primera parte de este proyecto consiste en modificar la función `main()` indicada en la Figura 3.25, de manera que al volver de la función `setup()`, se genere un proceso hijo y ejecute el comando especificado por el usuario.

Como hemos dicho anteriormente, la función `setup()` carga los contenidos de la matriz `args` con el comando especificado por el usuario. Esta matriz `args` se pasa a la función `execvp()`, que dispone de la siguiente interfaz:

```
execvp(char *command, char *params[]);
```

donde `command` representa el comando que se va a ejecutar y `params` almacena los parámetros del comando. En este proyecto, la función `execvp()` debe invocarse como `execv(args[0], args);` hay que asegurarse de comprobar el valor de `background` para determinar si el proceso padre debe esperar a que termine el proceso hijo o no.

Creación de una función histórica

La siguiente tarea consiste en modificar el programa de la Figura 3.25 para que proporcione una función *histórica* que permita al usuario acceder a los, como máximo, 10 últimos comandos que haya introducido. Estos comandos se numerarán comenzando por 1 y se incrementarán hasta sobrepasar incluso 10; por ejemplo, si el usuario ha introducido 35 comandos, los 10 últimos comandos serán los numerados desde 26 hasta 35. Esta función histórica se implementará utilizando unas cuantas técnicas diferentes.

En primer lugar, el usuario podrá obtener una lista de estos comandos cuando pulse `<Control><C>`, que es la señal `SIGINT`. Los sistemas UNIX emplean señales para notificar a un proceso que se ha producido un determinado suceso. Las señales pueden ser síncronas o asíncronas, dependiendo del origen y de la razón por la que se haya señalado el suceso. Una vez que se ha generado una señal debido a que ha ocurrido un determinado suceso (por ejemplo, una división por cero, un acceso a memoria ilegal, una entrada `<Control><C>` del usuario, etc.), la señal se suministra a un proceso, donde será tratada. El proceso que recibe una señal puede tratar mediante una de las siguientes técnicas:

- ignorar la señal,
- usar la rutina de tratamiento de la señal predeterminada, o
- proporcionar una función específica de tratamiento de la señal.

Las señales pueden tratarse configurando en primer lugar determinados campos de la estructura C `struct sigaction` y pasando luego esa estructura a la función `sigaction()`. Las señales se definen en el archivo `/usr/include/sys/signal.h`. Por ejemplo, la señal `SIGHUP` representa la señal para terminar un programa con la secuencia de control `<Control><C>`. La rutina predeterminada de tratamiento de señal para `SIGINT` consiste en terminar el programa.

Alternativamente, un programa puede definir su propia función de tratamiento de la señal configurando el campo `sa_handler` de `struct sigaction` con el nombre de la función que tratará la señal y luego invocando la función `sigaction()`, pasándola (1) la señal para la que está definiendo la rutina de tratamiento y (2) un puntero a `struct sigaction`.

En la Figura 3.26 mostramos un programa en C que usa la función `handle_SIGINT()` para tratar la señal `SIGINT`. Esta función escribe el mensaje “Capturado Control C” y luego invoca la función `exit()` para terminar el programa. Debemos usar la función `write()` para escribir salida en lugar de la más habitual `printf()`, ya que la primera es segura con respecto a las señales, lo que quiere decir que se la puede llamar desde dentro de una función de tratamiento de señal; `printf()` no ofrece dichas garantías. Este programa ejecutará el bucle `while(1)` hasta que el usuario introduzca la secuencia `<Control><C>`. Cuando esto ocurre, se invoca la función de tratamiento de señal `handle_SIGINT()`.

La función de tratamiento de señal se debe declarar antes de `main()` y, puesto que el control puede ser transferido a esta función en cualquier momento, no puede pasarse ningún parámetro a esta función. Por tanto, cualquier dato del programa al que tenga que acceder deberá declararse globalmente, es decir, al principio del archivo fuente, antes de la declaración de funciones. Antes de volver de la función de tratamiento de señal, debe ejecutarse de nuevo el indicativo de comandos.

Si el usuario introduce `<Control><C>`, la rutina de tratamiento de señal proporcionará como salida una lista de los 10 últimos comandos. Con esta lista, el usuario puede ejecutar cualquiera de los 10 comandos anteriores escribiendo `r x` donde ‘x’ es la primera letra de dicho comando. Si hay más de un comando que comienza con ‘x’, se ejecuta el más reciente. También, el usuario debe poder ejecutar de nuevo el comando más reciente escribiendo simplemente ‘r’. Podrán

```

#include <signal.h>
#include <unistd.h>
#include <stdio.h>

#define BUFFER_SIZE 50
char buffer[BUFFER_SIZE];

/* función de tratamiento de señal */
void handle_SIGINT()
{
    write(STDOUT_FILENO,buffer,strlen(buffer));

    exit(0);
}

int main(int argc, char *argv[])
{
    /* configura el descriptor de señal */
    struct sigaction handler;
    handler.sa_handler = handle_SIGINT;
    sigaction(SIGINT, &handler, NULL);

    /* genera el mensaje de salida */
    strcpy(buffer, "Captura Control C\n");

    /* bucle hasta recibir <Control><C> */
    while (1)
        ;

    return 0;
}

```

Figura 3.26 Programa de tratamiento de señal.

asumir que la 'r' estará separada de la primera letra por sólo un espacio y que la letra irá seguida de '\n'. Asimismo, si se desea ejecutar el comando más reciente, 'r' irá inmediatamente seguida del carácter \n.

Cualquier comando que se ejecute de esta forma deberá enviarse como eco a la pantalla del usuario y el comando deberá incluirse en el búfer de historial como comando más reciente. (r x no se incluye en el historial; lo que se incluye es el comando al que realmente representa).

Si el usuario intenta utilizar esta función historial para ejecutar un comando y se detecta que éste es *erróneo*, debe proporcionarse un mensaje de error al usuario y no añadirse el comando a la lista de historial, y no debe llamarse a la función execvp(). (Estaría bien poder detectar los comandos incorrectamente definidos que se entreguen a execvp(), que parecen válidos y no lo son, y no incluirlos en el historial tampoco, pero esto queda fuera de las capacidades de este programa de shell simple). También debe modificarse setup() de modo que devuelva un entero que indique si se ha creado con éxito una lista args válida o no, y la función main() debe actualizarse de acuerdo con ello.

Notas bibliográficas

La comunicación interprocesos en el sistema RC 4000 se explica en Brinch Hansen [1970]. Schlichting y Schneider [1982] abordan la primitivas de paso de mensajes asíncronas. La funcionalidad IPC implementada en el nivel de usuario se describe en Bershad et al [1990].

Los detalles sobre la comunicación interprocesos en sistemas UNIX se exponen en Gray [1990]. Barrera [1991] y Vahalia [1996] describen la comunicación interprocesos en el sistema Mac OS X. Solomon y Russinovich [2000] y Stevens [1999] abordan la comunicación interprocesos en Windows 2000 y UNIX, respectivamente.

La implementación de llamadas RPC se explica en Birrell y Nelson [1984]. Un diseño de un mecanismo de llamadas RPC se describe en Shrivastava y Panzieri [1982], y Tay y Ananda [1990] presentan una introducción a las llamadas RPC. Stankovic [1982] y Staunstrup [1982] presentan los mecanismos de comunicación mediante llamadas a procedimientos y paso de mensajes. Gross [2002] trata en detalle la invocación de métodos remotos (RMI). Calvert [2001] cubre la programación de *sockets* en Java.

Hebras

El modelo de proceso presentado en el Capítulo 3 suponía que un proceso era un programa en ejecución con una sola hebra de control. Ahora, la mayoría de los sistemas operativos modernos proporcionan características que permiten que un proceso tenga múltiples hebras de control. Este capítulo presenta diversos conceptos asociados con los sistemas informáticos multihebra, incluyendo una exposición sobre las API de las bibliotecas de hebras de Pthreads, Win32 y Java. Veremos muchos temas relacionados con la programación multihebra y veremos cómo afecta esta característica al diseño de sistemas operativos. Por último, estudiaremos cómo permiten los sistemas operativos Windows XP y Linux el uso de hebras en el nivel de *kernel*.

OBJETIVOS DEL CAPÍTULO

- Presentar el concepto de hebra, una unidad fundamental de utilización de la CPU que conforma los fundamentos de los sistemas informáticos multihebra.
- Explicar las API de las bibliotecas de hebras de Pthreads, Win32 y Java.

4.1 Introducción

Una hebra es una unidad básica de utilización de la CPU; comprende un ID de hebra, un contador de programa, un conjunto de registros y una pila. Comparte con otras hebras que pertenecen al mismo proceso la sección de código, la sección de datos y otros recursos del sistema operativo, como los archivos abiertos y las señales. Un proceso tradicional (o **proceso pesado**) tiene una sola hebra de control. Si un proceso tiene, por el contrario, múltiples hebras de control, puede realizar más de una tarea a la vez. La Figura 4.1 ilustra la diferencia entre un proceso tradicional **monohebra** y un proceso **multihebra**.

4.1.1 Motivación

Muchos paquetes de software que se ejecutan en los PC modernos de escritorio son multihebra. Normalmente, una aplicación se implementa como un proceso propio con varias hebras de control. Por ejemplo, un explorador web puede tener una hebra para mostrar imágenes o texto mientras que otra hebra recupera datos de la red. Un procesador de textos puede tener una hebra para mostrar gráficos, otra hebra para responder a las pulsaciones de teclado del usuario y una tercera hebra para el corrector ortográfico y gramatical que se ejecuta en segundo plano.

En determinadas situaciones, una misma aplicación puede tener que realizar varias tareas similares. Por ejemplo, un servidor web acepta solicitudes de los clientes que piden páginas web, imágenes, sonido, etc. Un servidor web sometido a una gran carga puede tener varios (quizás, miles) de clientes accediendo de forma concurrente a él. Si el servidor web funcionara como un proceso tradicional de una sola hebra, sólo podría dar servicio a un cliente cada vez y la cantidad

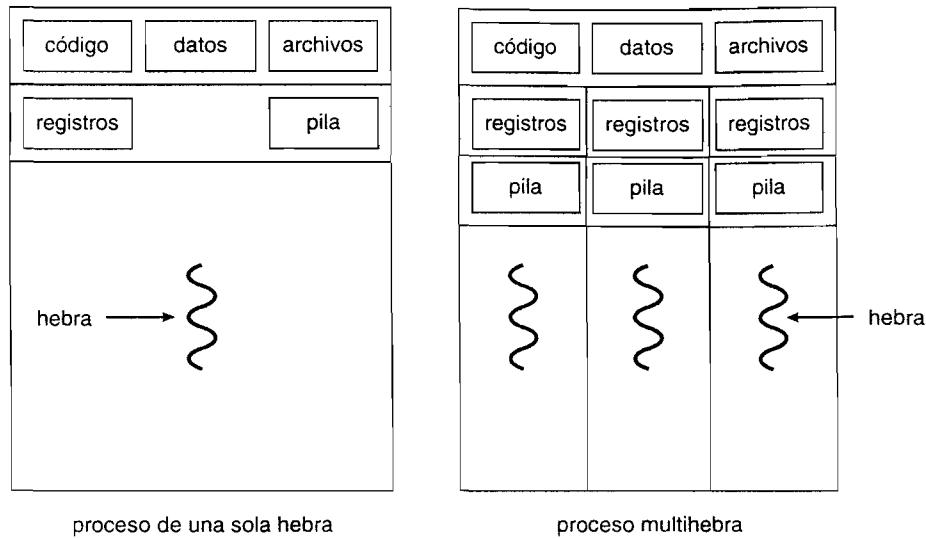


Figura 4.1 Procesos monohebra y multihebra.

de tiempo que un cliente podría tener que esperar para que su solicitud fuera servida podría ser enorme.

Una solución es que el servidor funcione como un solo proceso de aceptación de solicitudes. Cuando el servidor recibe una solicitud, crea otro proceso para dar servicio a dicha solicitud. De hecho, este método de creación de procesos era habitual antes de que las hebras se popularizaran. Como hemos visto en el capítulo anterior, la creación de procesos lleva tiempo y hace un uso intensivo de los recursos. Si el nuevo proceso va a realizar las mismas tareas que los procesos existentes, ¿por qué realizar todo ese trabajo adicional? Generalmente, es más eficiente usar un proceso que contenga múltiples hebras. Según este método, lo que se hace es dividir en múltiples hebras el proceso servidor web. El servidor crea una hebra específica para escuchar las solicitudes de cliente y cuando llega una solicitud, en lugar de crear otro proceso, el servidor crea otra hebra para dar servicio a la solicitud.

Las hebras también juegan un papel importante en los sistemas de llamada a procedimientos remotos (RPC). Recuerde del Capítulo 3 que las RPC permiten la comunicación entre procesos proporcionando un mecanismo de comunicación similar a las llamadas a funciones o procedimientos ordinarias. Normalmente, los servidores RPC son multihebra. Cuando un servidor recibe un mensaje, sirve el mensaje usando una hebra específica. Esto permite al servidor dar servicio a varias solicitudes concurrentes. Los sistemas RMI de Java trabajan de forma similar.

Por último, ahora muchos *kernel* de sistemas operativos son multihebra; hay varias hebras operando en el *kernel* y cada hebra realiza una tarea específica, tal como gestionar dispositivos o tratar interrupciones. Por ejemplo, Solaris crea un conjunto de hebras en el *kernel* específicamente para el tratamiento de interrupciones; Linux utiliza una hebra del *kernel* para gestionar la cantidad de memoria libre en el sistema.

4.1.2 Ventajas

Las ventajas de la programación multihebra pueden dividirse en cuatro categorías principales:

- Capacidad de respuesta.** El uso de múltiples hebras en una aplicación interactiva permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado o realizando una operación muy larga, lo que incrementa la capacidad de respuesta al usuario. Por ejemplo, un explorador web multihebra permite la interacción del usuario a través de una hebra mientras que en otra hebra se está cargando una imagen.
- Compartición de recursos.** Por omisión, las hebras comparten la memoria y los recursos del proceso al que pertenecen. La ventaja de compartir el código y los datos es que permite que

una aplicación tenga varias hebras de actividad diferentes dentro del mismo espacio de direcciones.

3. **Economía.** La asignación de memoria y recursos para la creación de procesos es costosa. Dado que las hebras comparten recursos del proceso al que pertenecen, es más económico crear y realizar cambios de contexto entre unas y otras hebras. Puede ser difícil determinar empíricamente la diferencia en la carga adicional de trabajo administrativo pero, en general, se consume mucho más tiempo en crear y gestionar los procesos que las hebras. Por ejemplo, en Solaris, crear un proceso es treinta veces lento que crear una hebra, y el cambio de contexto es aproximadamente cinco veces más lento.
4. **Utilización sobre arquitecturas multiprocesador.** Las ventajas de usar configuraciones multihebra pueden verse incrementadas significativamente en una arquitectura multiprocesador, donde las hebras pueden ejecutarse en paralelo en los diferentes procesadores. Un proceso monohebra sólo se puede ejecutar en una CPU, independientemente de cuántas haya disponibles. Los mecanismos multihebra en una máquina con varias CPU incrementan el grado de concurrencia.

4.2 Modelos multihebra

Hasta ahora, nuestra exposición se ha ocupado de las hebras en sentido genérico. Sin embargo, desde el punto de vista práctico, el soporte para hebras puede proporcionarse en el nivel de usuario (para las **hebras de usuario**) o por parte del *kernel* (para las **hebras del kernel**). El soporte para las hebras de usuario se proporciona por encima del *kernel* y las hebras se gestionan sin soporte del mismo, mientras que el sistema operativo soporta y gestiona directamente las hebras del *kernel*. Casi todos los sistemas operativos actuales, incluyendo Windows XP, Linux, Mac OS X, Solaris y Tru64 UNIX (antes Digital UNIX) soportan las hebras de *kernel*.

En último término, debe existir una relación entre las hebras de usuario y las del *kernel*; en esta sección, vamos a ver tres formas de establecer esta relación.

4.2.1 Modelo muchos-a-uno

El modelo muchos-a-uno (Figura 4.2) asigna múltiples hebras del nivel de usuario a una hebra del *kernel*. La gestión de hebras se hace mediante la biblioteca de hebras en el espacio de usuario, por lo que resulta eficiente, pero el proceso completo se bloquea si una hebra realiza una llamada bloqueante al sistema. También, dado que sólo una hebra puede acceder al *kernel* cada vez, no podrán

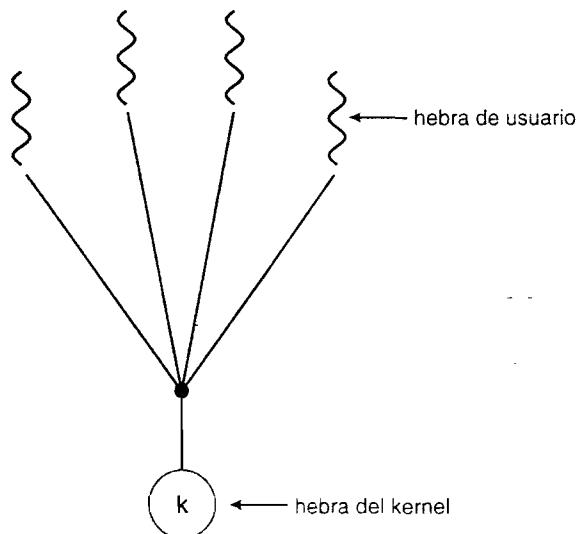


Figura 4.2 Modelo muchos-a-uno.

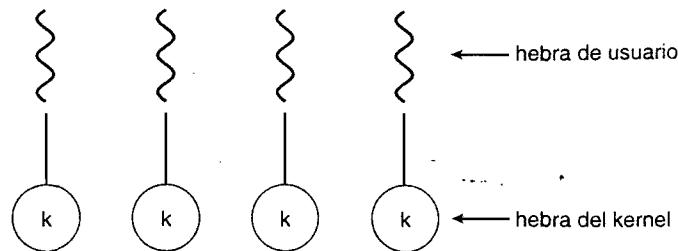


Figura 4.3 Modelo uno-a-uno.

ejecutarse varias hebras en paralelo sobre múltiples procesadores. El sistema de hebras **Green**, una biblioteca de hebras disponible en Solaris, usa este modelo, así como **GNU Portable Threads**.

4.2.2 Modelo uno-a-uno

El modelo uno-a-uno (Figura 4.3) asigna cada hebra de usuario a una hebra del *kernel*. Proporciona una mayor concurrencia que el modelo muchos-a-uno, permitiendo que se ejecute otra hebra mientras una hebra hace una llamada bloqueante al sistema; también permite que se ejecuten múltiples hebras en paralelo sobre varios procesadores. El único inconveniente de este modelo es que crear una hebra de usuario requiere crear la correspondiente hebra del *kernel*. Dado que la carga de trabajo administrativa para la creación de hebras del *kernel* puede repercutir en el rendimiento de una aplicación, la mayoría de las implementaciones de este modelo restringen el número de hebras soportadas por el sistema. Linux, junto con la familia de sistemas operativos Windows (incluyendo Windows 95, 98, NT, 200 y XP), implementan el modelo uno-a-uno.

4.2.3 Modelo muchos-a-muchos

El modelo muchos-a-muchos (Figura 4.4) multiplexa muchas hebras de usuario sobre un número menor o igual de hebras del *kernel*. El número de hebras del *kernel* puede ser específico de una determinada aplicación o de una determinada máquina (pueden asignarse más hebras del *kernel* a una aplicación en un sistema multiprocesador que en uno de un solo procesador). Mientras que el modelo muchos-a-uno permite al desarrollador crear tantas hebras de usuario como desee, no se consigue una concurrencia real, ya que el *kernel* sólo puede planificar la ejecución de una hebra cada vez. El modelo uno-a-uno permite una mayor concurrencia, pero el desarrollador debe tener

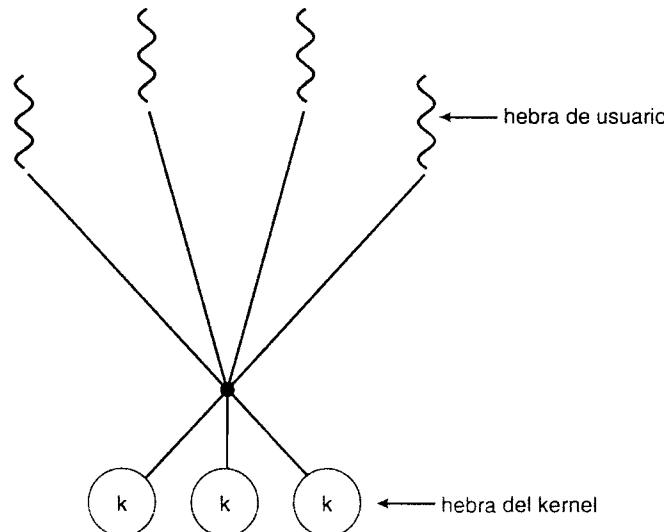


Figura 4.4 Modelo muchos-a-muchos.

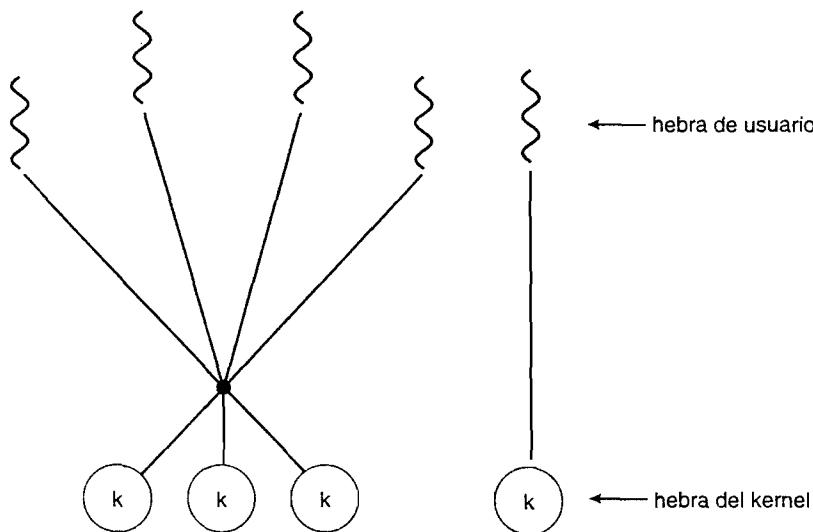


Figura 4.5 Modelo de dos niveles.

cuidado de no crear demasiadas hebras dentro de una aplicación (y, en algunos casos, el número de hebras que pueda crear estará limitado). El modelo muchos-a-muchos no sufre ninguno de estos inconvenientes. Los desarrolladores pueden crear tantas hebras de usuario como sean necesarias y las correspondientes hebras del *kernel* pueden ejecutarse en paralelo en un multiprocesador. Asimismo, cuando una hebra realiza una llamada bloqueante al sistema, el *kernel* puede planificar otra hebra para su ejecución.

Una popular variación del modelo muchos-a-muchos multiplexa muchas hebras del nivel de usuario sobre un número menor o igual de hebras del *kernel*, pero también permite acoplar una hebra de usuario a una hebra del *kernel*. Algunos sistemas operativos como IRIX, HP-UX y Tru64 UNIX emplean esta variante, que algunas veces se denomina *modelo de dos niveles* (Figura 4.5). El sistema operativo Solaris permitía el uso del modelo de dos niveles en las versiones anteriores a Solaris 9. Sin embargo, a partir de Solaris 9, este sistema emplea el modelo uno-a-uno.

4.3 Bibliotecas de hebras

Una **biblioteca de hebras** proporciona al programador una API para crear y gestionar hebras. Existen dos formas principales de implementar una biblioteca de hebras. El primer método consiste en proporcionar una biblioteca enteramente en el espacio de usuario, sin ningún soporte del *kernel*. Todas las estructuras de datos y el código de la biblioteca se encuentran en el espacio de usuario. Esto significa que invocar a una función de la biblioteca es como realizar una llamada a una función local en el espacio de usuario y no una llamada al sistema.

El segundo método consiste en implementar una biblioteca en el nivel del *kernel*, soportada directamente por el sistema operativo. En este caso, el código y las estructuras de datos de la biblioteca se encuentran en el espacio del *kernel*. Invocar una función en la API de la biblioteca normalmente da lugar a que se produzca una llamada al sistema dirigida al *kernel*.

Las tres principales bibliotecas de hebras actualmente en uso son: (1) POSIX Pthreads, (2) Win32 y (3) Java. Pthreads, la extensión de hebras del estándar POSIX, puede proporcionarse como biblioteca del nivel de usuario o del nivel de *kernel*. La biblioteca de hebras de Win32 es una biblioteca del nivel de *kernel* disponible en los sistemas Windows. La API de hebras Java permite crear y gestionar directamente hebras en los programas Java. Sin embargo, puesto que en la mayoría de los casos la JVM se ejecuta por encima del sistema operativo del *host*, la API de hebras Java se implementa habitualmente usando una biblioteca de hebras disponible en el sistema *host*. Esto significa que, normalmente, en los sistemas Windows, las hebras Java se implementan usando la API de Win32, mientras que en los sistemas Linux se suelen implementar empleando Pthreads.

En el resto de esta sección, vamos a describir la creación de hebras usando estas tres bibliotecas. Mediante un ejemplo ilustrativo, vamos a diseñar un programa multihebra que calcule el sumatorio de un entero no negativo en una hebra específica empleando la muy conocida función sumatorio:

$$sum = \sum_{i=0}^N i$$

Por ejemplo, si N fuera igual a 5, esta función representaría el sumatorio desde 0 hasta 5, que es 15. Cada uno de los tres programas se ejecutará especificando el límite superior del sumatorio a través de la línea de comandos; por tanto, si el usuario escribe 8, a la salida se obtendrá la suma de los valores enteros comprendidos entre 0 y 8.

4.3.1 Pthreads

Pthreads se basa en el estándar POSIX (IEEE 1003.1c) que define una API para la creación y sincronización de hebras. Se trata de una *especificación* para el comportamiento de las hebras, no de una *implementación*. Los diseñadores de sistemas operativos pueden implementar la especificación de la forma que deseen. Hay muchos sistemas que implementan la especificación Pthreads, incluyendo Solaris, Linux; Mac OS X y Tru64 UNIX. También hay disponibles implementaciones de libre distribución para diversos sistemas operativos Windows.

El programa C mostrado en la Figura 4.6 ilustra la API básica de Pthreads mediante un ejemplo de creación de un programa multihebra que calcula el sumatorio de un entero no negativo en una hebra específica. En un programa Pthreads, las diferentes hebras inician su ejecución en una función específica. En la Figura 4.6, se trata de la función `runner()`. Cuando este programa se inicia, da comienzo una sola hebra de control en `main()`. Después de algunas inicializaciones, `main()` crea una segunda hebra que comienza en la función `runner()`. Ambas hebras comparten la variable global `sum`.

Analicemos más despacio este programa. Todos los programas Pthreads deben incluir el archivo de cabecera `pthread.h`. La instrucción `pthread_t tid` declara el identificador de la hebra que se va a crear. Cada hebra tiene un conjunto de atributos, que incluye el tamaño de la pila y la información de planificación. La declaración `pthread_attr_t attr` representa los atributos de la hebra; establecemos los atributos en la llamada a la función `pthread_attr_init(&attr)`. Dado que no definimos explícitamente ningún atributo, se usan los atributos predeterminados. En el Capítulo 5, veremos algunos de los atributos de planificación proporcionados por la API de Pthreads. Para crear otra hebra se usa la llamada a la función `pthread_create()`. Además de pasar el identificador de la hebra y los atributos de la misma, también pasamos el nombre de la función en la que la nueva hebra comenzará la ejecución, que en este caso es la función `runner()`. Por último, pasamos el parámetro entero que se proporcionó en la línea de comandos, `argv[1]`.

En este punto, el programa tiene dos hebras: la hebra inicial (o padre) en `main()` y la hebra del sumatorio (o hijo) que realiza la operación de suma en la función `runner()`. Después de crear la hebra sumatorio, la hebra padre esperará a que se complete llamando a la función `pthread_join`. La hebra sumatorio se completará cuando llame a la función `pthread_exit()`. Una vez que la hebra sumatorio ha vuelto, la hebra padre presenta a la salida el valor de la variable compartida `sum`.

4.3.2 Hebras de Win32

La técnica de creación de hebras usando la biblioteca de hebras de Win32 es similar en varios aspectos a la técnica empleada en Pthreads. Vamos a ilustrar la API para hebras Win32 mediante el programa C de la Figura 4.7. Observe que tenemos que incluir el archivo de cabecera `windows.h` cuando se usa la API de Win32.

Al igual que la versión de Pthreads de la Figura 4.6, los datos compartidos por las diferentes hebras, en este caso, `sum`, se declaran globalmente (el tipo de datos `DWORD` es un entero de 32 bits

```

#include <pthread.h>
#include <stdio.h>

int sum /*las hebras comparten esta variable*/
void *runner(void *param); /* la hebra */

int main(int argc, char *argv[])
{
    pthread_t tid; /* el identificador de la hebra */
    pthread_attr_t attr; /* conjunto de atributos de la hebra */

    if (argc != 2) {
        fprintf (stderr, "uso: a.out <valor entero>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf (stderr, "%d debe ser >= 0\n", atoi(argv[1]));
        return -1;
    }

    /* obtener los atributos predeterminados */
    pthread_attr_init(&attr);
    /* crear la hebra */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* esperar a que la hebra termine */
    pthread_join(tid,NULL);

    printf("sum = %d\n", sum);
}

/* La hebra inicia su ejecución en esta función */
void *runner(void *param)
{
    int i, upper = attoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}

```

Figura 4.6 Programa C multihebra usando la API de Pthreads.

sin signo). También definimos la función Summation(), que se ejecuta en una hebra distinta. A esta función se le pasa un puntero a void, que se define en Win32 como LPVOID. La hebra que realiza esta función asigna a la variable global sum el valor del sumatorio desde 0 hasta el parámetro pasado a Summation().

En la API de Win32 las hebras se crean usando la función CreateThread() y, como en Pthreads, a esta función se le pasa un conjunto de atributos para la hebra. Estos atributos incluyen información de seguridad, el tamaño de la pila y un indicador que puede configurarse para indicar si la hebra se va a iniciar en un estado suspendido. En este programa, usamos los valores predeterminados para estos atributos (los cuales inicialmente no establecen que la hebra esté en estado suspendido, sino que cumple los requisitos necesarios para ser ejecutada por el planifica-

```

#include <windows.h>
#include <stdio.h>
DWORD Sum; /* dato compartido por las hebras */
/* la hebra se ejecuta en esta función separada */

DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    /* realiza algunas comprobaciones básicas de errores */
    if (argc != 1) {
        fprintf(stderr, "Se requiere un parámetro entero\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "Se requiere un entero >= 0\n");
        return -1;
    }

    // crear la hebra
    ThreadHandle = CreateThread(
        NULL, // atributos de seguridad predeterminados
        0, // tamaño predeterminado de la pila
        Summation, // función de la hebra
        &Param, // parámetro para la función de la hebra
        0, // indicadores de creación predeterminados
        &ThreadId); // devuelve el identificador de la hebra

    if (ThreadHandle != NULL){
        // ahora esperar a que la hebra termine
        WaitForSingleObject(ThreadHandle, INFINITE);

        // cerrar el descriptor de la hebra
        CloseHandle(ThreadHandle);

        printf("sum = %d\n", Sum);
    }
}

```

Figura 4.7 Programa C multihebra usando la API de Win32.

dor de la CPU). Una vez que se ha creado la hebra sumatorio, el padre debe esperar a que se complete antes de proporcionar a la salida el valor de sum, ya que la hebra sumatorio es quien asigna este valor. Recuerde que en el programa para Pthreads (Figura 4.6), la hebra padre esperaba a la hebra sumatorio usando la instrucción `pthread_join()`. Hacemos lo equivalente en la API de

Win32 usando la función `WaitForSingleObject()`, que hace que la hebra creadora quede bloqueada hasta que la hebra sumatorio haya terminado. (En el Capítulo 6 veremos más en detalle los objetos de sincronización).

4.3.3 Hebras Java

Las hebras son el modelo fundamental de ejecución de programas en un programa Java, y el lenguaje Java y su API proporcionan un rico conjunto de características para la creación y gestión de hebras. Todos los programas Java tienen al menos una hebra de control; incluso un sencillo programa Java que sólo conste de un método `main()` se ejecuta como hebra en la máquina virtual Java.

Existen dos técnicas para crear hebras en un programa Java. Un método consiste en crear una nueva clase que se derive de la clase `Thread` y sustituir apropiadamente su método `run()`. Una técnica alternativa, y más frecuentemente utilizada, consiste en definir una clase que implemente la interfaz `Runnable`. Esta interfaz se define como sigue:

```
public interface Runnable
{
    public abstract void run();
}
```

Cuando una clase implementa la interfaz `Runnable`, debe definir un método `run()`. El código que implementa el método `run()` es el que se ejecuta como una hebra separada.

La Figura 4.8 muestra la versión Java de un programa multihebra que calcula el sumatorio de un entero no negativo. La clase `Summation` implementa la interfaz `Runnable`. La creación de la hebra se lleva a cabo creando una instancia de la clase `Thread` y pasando al constructor a un objeto `Runnable`.

La creación de un objeto `Thread` no crea específicamente la nueva hebra, sino que es el método `start()` el que realmente crea la hebra. La llamada al método `start()` para el nuevo objeto hace dos cosas:

1. Asigna la memoria e inicializa una nueva hebra en la JVM.
2. Llama al método `run()`, haciendo que la hebra cumpla los requisitos necesarios para ser ejecutada por la JVM. Observe que nunca llamamos al método `run()` directamente. En su lugar, llamamos al método `start()`, y éste llamada al método `run()` por cuenta nuestra.

Cuando se ejecuta el programa del sumatorio, la JVM crea dos hebras. La primera es la hebra padre, que inicia su ejecución en el método `main()`. La segunda hebra se crea cuando se invoca el método `start()` del objeto `Thread`. Esta hebra hijo comienza la ejecución en el método `run()` de la clase `Summation`. Después de proporcionar como salida el valor del sumatorio, esta hebra termina cuando sale de su método `run()`.

La compartición de los datos entre las hebras resulta sencilla en Win32 y Pthreads, ya que los datos compartidos simplemente se declaran de modo `global`. Como un lenguaje orientado a objetos puro, Java no tiene el concepto de datos globales; si una o más hebras comparten datos en un programa Java, la compartición se realiza pasando por referencia el objeto compartido a las hebras apropiadas. En el programa Java mostrado en la Figura 4.8, la hebra principal y la hebra sumatorio comparten la instancia de la clase `Sum`. Este objeto, compartido se referencia a través de los métodos `getSum()` y `setSum()` apropiados. (El lector puede estar preguntando por qué no usamos un objeto `Integer` en lugar de diseñar una nueva clase `sum`. La razón es que la clase `Integer` es **inmutable**, es decir, una vez que se ha especificado su valor, no se puede cambiar.)

Recuerde que las hebras padre de las bibliotecas Pthreads y Win32 usan, respectivamente, `pthread_join()` y `WaitForSingleObject()`, para esperar a que las hebras sumatorio terminen antes de continuar. El método `join()` de Java proporciona una funcionalidad similar. Observe que `join()` puede generar una excepción `InterruptedException`, que podemos elegir ignorar.

```

class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run(){
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum +=i;
        sumValue.setSum(sum);
    }
}

public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + "debe ser >= 0.");
            else {
                // crea el objeto que hay que compartir
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("La suma de "+upper+" es "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Uso: Summation <valor entero>"); }
}

```

Figura 4.8 Programa Java para el sumatorio de un entero no negativo.

La JVM y el sistema operativo del host

Normalmente, la JVM se implementa por encima del sistema operativo del *host* (véase la Figura 2.17). Esta configuración permite a la JVM ocuparse de los detalles de implementación del sistema operativo subyacente y proporcionar un entorno abstracto y coherente que permite a los programas Java operar sobre cualquier plataforma que soporte una JVM. La especificación de la JVM no indica cómo asignar las hebras Java al sistema operativo subyacente, dejándose dicha decisión a la implementación concreta de la JVM. Por ejemplo, el sistema operativo Windows XP usa el modelo uno-a-uno; por tanto, cada hebra Java de una máquina virtual que se ejecute sobre un sistema así se mapea sobre una hebra del *kernel*. En los sistemas operativos que usan el modelo muchos-a-muchos (tal como Tru64 UNIX), cada hebra Java se asigna de acuerdo con el modelo muchos-a-muchos. Solaris implementaba inicialmente la JVM usando el modelo muchos-a-uno (la biblioteca de hebras Green mencionada anteriormente); pero las versiones posteriores de la JVM se implementaron usando el modelo muchos-a-muchos. A partir de Solaris 9, las hebras de Java se asignan usando el modelo uno-a-uno. Además, puede existir una relación entre la biblioteca de hebras Java y la biblioteca de hebras disponible en el sistema operativo del *host*. Por ejemplo, las implementaciones de la JVM para la familia Windows de sistemas operativos pueden usar la API de Win32 al crear hebras Java; los sistemas Linux y Solaris pueden emplear la API de Pthreads.

4.4 Consideraciones sobre las hebras

En esta sección vamos a ver algunas de las cuestiones que hay que tener en cuenta en los programas multihebra.

4.4.1 Las llamadas al sistema `fork()` y `exec()`

En el Capítulo 3 hemos descrito cómo se usa la llamada al sistema `fork()` para crear un proceso duplicado e independiente. La semántica de las llamadas al sistema `fork()` y `exec()`, cambia en los programas multihebra.

Si una hebra de un programa llama a `fork()`, ¿duplica todas las hebras el nuevo proceso o es un proceso de una sola hebra? Algunos sistemas UNIX han decidido disponer de dos versiones de `fork()`, una que duplica todas las hebras y otra que sólo duplica la hebra que invocó la llamada al sistema `fork()`.

La llamada al sistema `exec()`, típicamente, funciona de la misma forma que se ha descrito en el Capítulo 3. Es decir, si una hebra invoca la llamada al sistema `exec()`, el programa especificado en el parámetro de `exec()` reemplazará al proceso completo, incluyendo todas las hebras.

Cuál de las dos versiones de `fork()` se use depende de la aplicación. Si se llama a `exec()` inmediatamente después de `fork()`, entonces resulta innecesario duplicar todas las hebras, ya que el programa especificado en los parámetros de `exec()` reemplazará al proceso existente. En este caso, lo apropiado es duplicar sólo la hebra que hace la llamada. Sin embargo, si el nuevo proceso no llama a `exec()` después de `fork()`, el nuevo proceso deberá duplicar todas las hebras.

4.4.2 Cancelación

La **cancelación de una hebra** es la acción de terminar una hebra antes de que se haya completado. Por ejemplo, si varias hebras están realizando una búsqueda de forma concurrente en una base de datos y una hebra devuelve el resultado, las restantes hebras deben ser canceladas. Puede producirse otra situación de este tipo cuando un usuario pulsa un botón en un explorador web que detiene la descarga de una página web. A menudo, las páginas web se cargan usando varias

hebras, cargándose cada imagen en una hebra diferente. Cuando un usuario pulsa el botón `stop` del navegador, todas las hebras de carga de la página se cancelan.

Una hebra que vaya a ser cancelada se denomina a menudo **hebra objetivo**. La cancelación de una hebra objetivo puede ocurrir en dos escenarios diferentes:

1. **Cancelación asíncrona.** Una determinada hebra hace que termine inmediatamente la hebra objetivo.
2. **Cancelación diferida.** La hebra objetivo comprueba periódicamente si debe terminar, lo que la proporciona una oportunidad de terminar por sí misma de una forma ordenada.

La dificultad de la cancelación se produce en aquellas situaciones en las que se han asignado recursos a una hebra cancelada o cuando una hebra se cancela en mitad de una actualización de datos que tuviera compartidos con otras hebras. Estos problemas son especialmente graves cuando se utiliza el mecanismo de cancelación asíncrona. A menudo, el sistema operativo reclamará los recursos asignados a una hebra cancelada, pero no reclamará todos los recursos. Por tanto cancelar una hebra de forma asíncrona puede hacer que no se libere un recurso del sistema que sea necesario para otras cosas.

Por el contrario, con la cancelación diferida, una hebra indica que otra hebra objetivo va a ser cancelada, pero la cancelación sólo se produce después de que la hebra objetivo haya comprobado el valor de un indicador para determinar si debe cancelarse o no. Esto permite que esa comprobación de si debe cancelarse sea realizada por la hebra objetivo en algún punto en que la cancelación se pueda hacer de forma segura. Pthreads denomina a dichos momentos **puntos de cancelación**.

4.4.3 Tratamiento de señales

Una **señal** se usa en los sistemas UNIX para notificar a un proceso que se ha producido un determinado suceso. Una señal puede recibirse síncrona o asíncronamente, dependiendo del origen y de la razón por la que el suceso deba ser señalizado. Todas las señales, sean síncronas o asíncronas, siguen el mismo patrón:

1. Una señal se genera debido a que se produce un determinado suceso.
2. La señal generada se suministra a un proceso.
3. Una vez suministrada, la señal debe ser tratada.

Como ejemplos de señales síncronas podemos citar los accesos ilegales a memoria y la división por 0. Si un programa en ejecución realiza una de estas acciones, se genera una señal. Las señales síncronas se proporcionan al mismo proceso que realizó la operación que causó la señal (ésa es la razón por la que se consideran síncronas).

Cuando una señal se genera por un suceso externo a un proceso en ejecución, dicho proceso recibe la señal en modo asíncrono. Como ejemplos de tales señales podemos citar la terminación de un proceso mediante la pulsación de teclas específicas, como `<control><C>`, y el fin de cuenta de un temporizador. Normalmente, las señales asíncronas se envían a otro proceso.

Cada señal puede ser *tratada* por una de dos posibles rutinas de tratamiento:

1. Una rutina de tratamiento de señal predeterminada.
2. Una rutina de tratamiento de señal definida por el usuario.

Cada señal tiene una **rutina de tratamiento de señal predeterminada** que el *kernel* ejecuta cuando trata dicha señal. Esta acción predeterminada puede ser sustituida por una **rutina de tratamiento de señal definida por el usuario** a la que se llama para tratar la señal. Las señales pueden tratarse de diferentes formas: algunas señales (tales como la de cambio en el tamaño de una ventana) simplemente pueden ignorarse; otras (como un acceso ilegal a memoria) pueden tratarse mediante la terminación del programa.

El tratamiento de señales en programas de una sola hebra resulta sencillo; las señales siempre se suministran a un proceso. Sin embargo, suministrar las señales en los programas multihébra es más complicado, ya que un proceso puede tener varias hebras. ¿A quién, entonces, debe suministrarse la señal?

En general, hay disponibles las siguientes opciones:

1. Suministrar la señal a la hebra a la que sea aplicable la señal.
2. Suministrar la señal a todas las hebras del proceso.
3. Suministrar la señal a determinadas hebras del proceso.
4. Asignar una hebra específica para recibir todas las señales del proceso.

El método para suministrar una señal depende del tipo de señal generada. Por ejemplo, las señales síncronas tienen que suministrarse a la hebra que causó la señal y no a las restantes hebras del proceso. Sin embargo, la situación con las señales asíncronas no es tan clara. Algunas señales asíncronas, como una señal de terminación de un proceso (por ejemplo, <control><C>) deberían enviarse a todas las hebras.

La mayoría de las versiones multihébra de UNIX permiten que las hebras especifiquen qué señales aceptarán y cuáles bloquearán. Por tanto, en algunos casos, una señal asíncrona puede suministrarse sólo a aquellas hebras que no la estén bloqueando. Sin embargo, dado que las señales necesitan ser tratadas sólo una vez, cada señal se suministra normalmente sólo a la primera hebra que no la esté bloqueando. La función estándar UNIX para suministrar una señal es `kill(aid_t aid, int signal)`; aquí, especificamos el proceso `aid` al que se va a suministrar una señal determinada. Sin embargo, Pthreads de POSIX también proporciona la función `pthread_kill(pthread_t tid, int signal)`, que permite que una señal sea suministrada a una hebra especificada (`tid`).

Aunque Windows no proporciona explícitamente soporte para señales, puede emularse usando las llamadas asíncronas a procedimientos (APC, asynchronous procedure call). La facilidad APC permite a una hebra de usuario especificar una función que será llamada cuando la hebra de usuario reciba una notificación de un suceso particular. Como indica su nombre, una llamada APC es el equivalente de una señal asíncrona en UNIX. Sin embargo, mientras que UNIX tiene que enfrentarse con cómo tratar las señales en un entorno multihébra, la facilidad APC es más sencilla, ya que cada APC se suministra a una hebra concreta en lugar de a un proceso.

4.4.4 Conjuntos compartidos de hebras

En la Sección 4.1 hemos mencionado la funcionalidad multihébra de un servidor web. En esta situación, cuando el servidor recibe una solicitud, crea una hebra nueva para dar servicio a la solicitud. Aunque crear una nueva hebra es, indudablemente, mucho menos costoso que crear un proceso nuevo, los servidores multihébra no están exentos de problemas potenciales. El primero concierne a la cantidad de tiempo necesario para crear la hebra antes de dar servicio a la solicitud, junto con el hecho de que esta hebra se descartará una vez que haya completado su trabajo. El segundo tema es más problemático: si permitimos que todas las solicitudes concurrentes sean servidas mediante una nueva hebra, no ponemos límite al número de hebras activas de forma concurrente en el sistema. Un número ilimitado de hebras podría agotar determinados recursos del sistema, como el tiempo de CPU o la memoria. Una solución para este problema consiste en usar lo que se denomina un **conjunto compartido de hebras**.

La idea general subyacente a los conjuntos de hebras es la de crear una serie de hebras al principio del proceso y colocarlas en un *conjunto compartido*, donde las hebras quedan a la espera de que se les asigne un trabajo. Cuando un servidor recibe una solicitud, despierta a una hebra del conjunto, si hay una disponible, y le pasa la solicitud para que la hebra se encargue de darla servicio. Una vez que la hebra completa su tarea, vuelve al conjunto compartido y espera hasta que haya más trabajo. Si el conjunto no tiene ninguna hebra disponible, el servidor espera hasta que quede una libre.

Los conjuntos compartidos de hebras ofrecen las siguientes ventajas:

1. Dar servicio a una solicitud con una hebra existente normalmente es más rápido que esperar a crear una hebra.
2. Un conjunto de hebras limita el número de hebras existentes en cualquier instante dado. Esto es particularmente importante en aquellos sistemas que no puedan soportar un gran número de hebras concurrentes.

El número de hebras del conjunto puede definirse heurísticamente basándose en factores como el número de procesadores del sistema, la cantidad de memoria física y el número esperado de solicitudes concurrentes de los clientes. Las arquitecturas de conjuntos de hebras más complejas pueden ajustar dinámicamente el número de hebras del conjunto de acuerdo con los patrones de uso. Tales arquitecturas proporcionan la importante ventaja de tener un conjunto compartido más pequeño, que consume menos memoria, cuando la carga del sistema es baja.

La API de Win32 proporciona varias funciones relacionadas con los conjuntos de hebras. El uso de la API del conjunto compartido de hebras es similar a la creación de una hebra con la función `Thread Create()`, como se describe en la Sección 4.3.2. El código mostrado a continuación define una función que hay que ejecutar como una hebra independiente:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /**
     * esta función se ejecuta como una hebra independiente.
     */
}
```

Se pasa un puntero a `PoolFunction()` a una de las funciones de la API del conjunto de hebras, y una hebra del conjunto se encarga de ejecutar esta función. Una de esas funciones de la API del conjunto de hebras es `QueueUserWorkItem()`, a la que se pasan tres parámetros:

- `LPTHREAD_START_ROUTINE Function`—un puntero a la función que se va a ejecutar como una hebra independiente.
- `PVOID Param`—el parámetro pasado a `Function`
- `ULONG Flags`—una serie de indicadores que regulan cómo debe el conjunto de hebras crear la hebra y gestionar su ejecución.

Un ejemplo de invocación sería:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

Esto hace que una hebra del conjunto de hebras invoque `PoolFunction()` por cuenta nuestra; en este caso, no pasamos ningún parámetro a `PoolFunction()`. Dado que especificamos 0 como indicador, no proporcionamos al conjunto de hebras ninguna instrucción especial para la creación de la hebra.

Otras funciones de la API del conjunto de hebras de Win32 son las utilidades que invocan funciones a intervalos periódicos o cuando se completa una solicitud de E/S asíncrona. El paquete `java.util.concurrent` de Java 1.5 proporciona también una funcionalidad de conjunto compartido de hebras.

4.4.5 Datos específicos de una hebra

Las hebras que pertenecen a un mismo proceso comparten los datos del proceso. Realmente, esta compartición de datos constituye una de las ventajas de la programación multihebra. Sin embargo, en algunas circunstancias, cada hebra puede necesitar su propia copia de ciertos datos. Llamaremos a dichos datos, **datos específicos de la hebra**. Por ejemplo, en un sistema de procesamiento de transacciones, podemos dar servicio a cada transacción mediante una hebra distinta. Además, puede asignarse a cada transacción un identificador único. Para asociar cada hebra con su identificador único, podemos emplear los datos específicos de la hebra. La mayor parte de las bibliotecas de hebras, incluyendo Win32 y Pthreads, proporcionan ciertos mecanismos de

soporte para los datos específicos de las hebras. Java también proporciona soporte para este tipo de datos.

4.4.6 Activaciones del planificador

Una última cuestión que hay que considerar en los programas multihebra concierne a los mecanismos de comunicación entre el *kernel* y la biblioteca de hebras, que pueden ser necesarios para los modelos muchos-a-muchos y de dos niveles vistos en la Sección 4.2.3. Tal coordinación permite que el número de hebras del *kernel* se ajuste de forma dinámica, con el fin de obtener el mejor rendimiento posible.

Muchos sistemas que implementan el modelo muchos-a-muchos o el modelo de dos niveles colocan una estructura de datos intermedia entre las hebras de usuario y del *kernel*. Esta estructura de datos, conocida normalmente como el nombre de proceso ligero o LWP (lightweight process) se muestra en la Figura 4.9. A la biblioteca de hebras de usuario, el proceso ligero le parece un *procesador virtual* en el que la aplicación puede hacer que se ejecute una hebra de usuario. Cada LWP se asocia a una hebra del *kernel*, y es esta hebra del *kernel* la que el sistema operativo ejecuta en los procesadores físicos. Si una hebra del *kernel* se bloquea (por ejemplo, mientras espera a que se complete una operación de E/S), el proceso ligero LWP se bloquea también. Por último, la hebra de usuario asociada al LWP también se bloquea.

Una aplicación puede requerir un número cualquiera de procesos ligeros para ejecutarse de forma eficiente. Considere una aplicación limitada por CPU que se ejecute en un solo procesador. En este escenario, sólo una hebra puede ejecutarse cada vez, por lo que un proceso ligero es suficiente. Sin embargo, una aplicación que haga un uso intensivo de las operaciones de E/S puede requerir ejecutar múltiples procesos LWP. Normalmente, se necesita un proceso ligero por cada llamada concurrente al sistema que sea de tipo bloqueante. Por ejemplo, suponga que se producen simultáneamente cinco lecturas de archivo diferentes; se necesitarán cinco procesos ligeros, ya que todos podrían tener que esperar a que se complete una operación de E/S en el *kernel*. Si un proceso tiene sólo cuatro LWP, entonces la quinta solicitud tendrá que esperar a que uno de los procesos LWP vuelva del *kernel*.

Un posible esquema de comunicación entre la biblioteca de hebras de usuario y el *kernel* es el que se conoce con el nombre de **activación del planificador**, que funciona como sigue: el *kernel* proporciona a la aplicación un conjunto de procesadores virtuales (LWP) y la aplicación puede planificar la ejecución de las hebras de usuario en uno de los procesadores virtuales disponibles. Además, el *kernel* debe informar a la aplicación sobre determinados sucesos; este procedimiento se conoce como el nombre de **suprallamada (upcall)**. Las suprallamadas son tratadas por la biblioteca de hebras mediante una **rutina de tratamiento de suprallamada**, y estas rutinas deben ejecutarse sobre un procesador virtual. Uno de los sucesos que disparan una suprallamada se produce cuando una hebra de la aplicación esté a punto de bloquearse. En este escenario, el *kernel* realiza una suprallamada a la aplicación para informarla de que una hebra se va a bloquear y para identificar la hebra concreta de que se trata. El *kernel* asigna entonces un procesador virtual nuevo a la

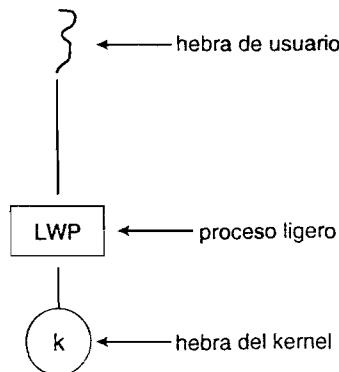


Figura 4.9 Proceso ligero (LWP).

aplicación. La aplicación ejecuta la rutina de tratamiento de la suprallamada sobre el nuevo procesador virtual, que guarda el estado de la hebra bloqueante, y libera el procesador virtual en que se está ejecutando dicha hebra. La rutina de tratamiento de la suprallamada planifica entonces la ejecución de otra hebra que reúna los requisitos para ejecutarse en el nuevo procesador virtual. Cuando se produce el suceso que la hebra bloqueante estaba esperando, el *kernel* hace otra suprallamada a la biblioteca de hebras para informar de que la hebra anteriormente bloqueada ahora puede ejecutarse. La rutina de tratamiento de la suprallamada correspondiente a este suceso también necesita un procesador virtual, y el *kernel* puede asignar un nuevo procesador o suspender temporalmente una de las hebras de usuario y ejecutar la rutina de tratamiento de suprallamada en su procesador virtual. Después de marcar la hebra desbloqueada como disponible para ejecutarse, la aplicación elige una de las hebras que esté preparada para ejecutarse y la ejecuta en un procesador virtual disponible.

4.5 Ejemplos de sistemas operativos

En esta sección exploraremos cómo se implementan las hebras en los sistemas Windows XP y Linux.

4.5.1 Hebras de Windows XP

Windows XP implementa la API Win32. Esta API es la principal interfaz de programación de aplicaciones de la familia de sistemas operativos Microsoft (Windows 95, 98, NT, 2000 y XP). Ciertamente, gran parte de lo que se ha explicado en esta sección se aplica a esta familia de sistemas operativos.

Una aplicación de Windows XP se ejecuta como un proceso independiente, y cada proceso puede contener una o más hebras. La API de Win32 para la creación de hebras se explica en la Sección 4.3.2. Windows XP utiliza el modelo uno-a-uno descrito en la Sección 4.2.2, donde cada hebra de nivel de usuario se asigna a una hebra del *kernel*. Sin embargo, Windows XP también proporciona soporte para una biblioteca de **fibras**, que proporciona la funcionalidad del modelo muchos-a-muchos (Sección 4.2.3). Con la biblioteca de hebras, cualquier hebra perteneciente a un proceso puede acceder al espacio de direcciones de dicho proceso.

Los componentes generales de una hebra son:

- Un identificador de hebra que identifique de forma única la hebra.
- Un conjunto de registros que represente el estado del procesador.
- Una pila de usuario, empleada cuando la hebra está ejecutándose en modo usuario, y una pila del *kernel*, empleada cuando la hebra se esté ejecutando en modo *kernel*.
- Un área de almacenamiento privada usada por las distintas bibliotecas de tiempo de ejecución y bibliotecas de enlace dinámico (DLL).

El conjunto de registros, las pilas y el área de almacenamiento privado constituyen el **contenedor** de la hebra. Las principales estructuras de datos de una hebra son:

- ETHREAD—bloque de hebra ejecutiva
- KTHREAD—bloque de hebra del *kernel*
- TEB—bloque de entorno de la hebra

Los componentes clave de ETHREAD incluyen un puntero al proceso al que pertenece la hebra y la dirección de la rutina en la que la hebra inicia su ejecución. ETHREAD también contiene un puntero al correspondiente bloque KTHREAD.

KTHREAD incluye información de planificación y sincronización de la hebra. Además, KTHREAD incluye la pila del *kernel* (utilizada cuando la hebra se está ejecutando en modo *kernel*) y un puntero al bloque TEB.

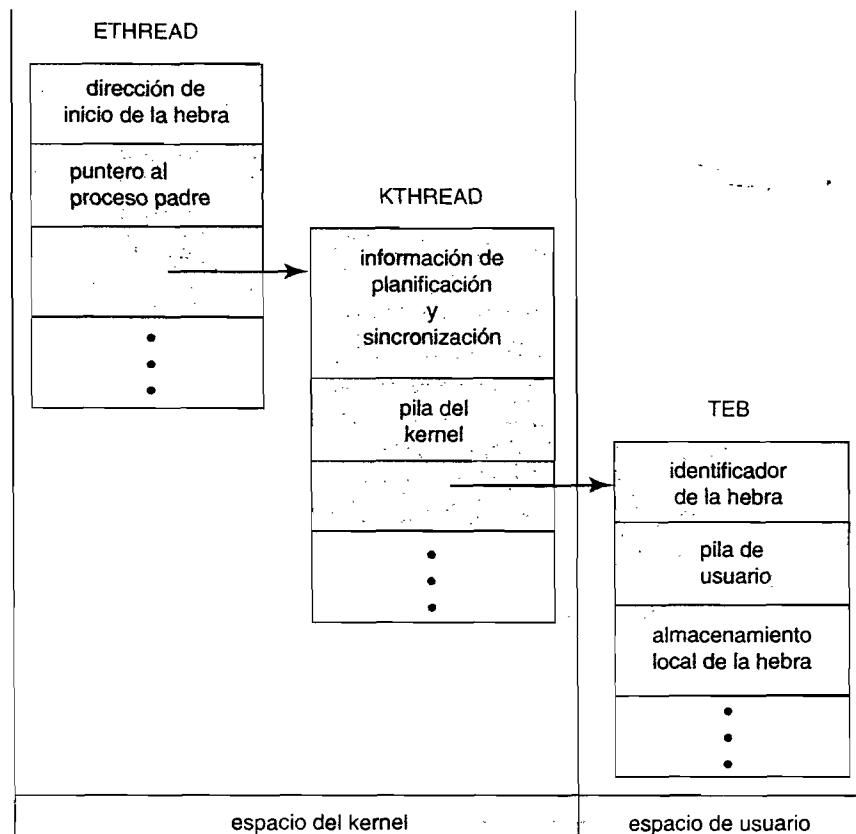


Figura 4.10 Estructuras de datos de una hebra de Windows XP.

ETHREAD y KTHREAD están completamente incluidos en el espacio del *kernel*; esto significa que sólo el *kernel* puede acceder a ellos. El bloque TEB es una estructura de datos del espacio de usuario a la que se accede cuando la hebra está ejecutándose en modo usuario. Entre otros campos, TEB contiene el identificador de la hebra, una pila del modo usuario y una matriz para los datos específicos de la hebra (lo que en la terminología de Windows XP se denomina **almacenamiento local de la hebra**). La estructura de una hebra de Windows XP se ilustra en la Figura 4.10.

4.5.2 Hebras de Linux

Linux proporciona la llamada al sistema `fork()` con la funcionalidad tradicional de duplicar un proceso, como se ha descrito en el Capítulo 3. Linux también proporciona la capacidad de crear hebras usando la llamada al sistema `clone()`. Sin embargo, Linux no diferencia entre procesos y hebras. De hecho, generalmente, Linux utiliza el término *tarea* en lugar de *proceso* o *hebra*, para hacer referencia a un flujo de control dentro de un programa. Cuando se invoca `clone()`, se pasa un conjunto de indicadores, que determina el nivel de compartición entre las tareas padre e hijo. Algunos de estos indicadores son los siguientes:

indicador	significado
<code>CLONE_FS</code>	Se comparte información del sistema de archivos.
<code>CLONE_VM</code>	Se comparte el mismo espacio de memoria.
<code>CLONE_SIGHAND</code>	Se comparten los descriptores de señal.
<code>CLONE_FILES</code>	Se comparte el conjunto de archivos abiertos.

Por ejemplo, si se pasan a `clone()` los indicadores `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` y `CLONE_FILES`, las tareas padre e hijo compartirán la misma información del sistema de archivos (como por ejemplo, el directorio actual de trabajo), el mismo espacio de memoria, las mismas rutinas de tratamiento de señal y el mismo conjunto de archivos abiertos. Usar `clone()` de este modo es equivalente a crear una hebra como se ha descrito en este capítulo, dado que la tarea padre comparte la mayor parte de sus recursos con sus tareas hijo. Sin embargo, si no se define ninguno de estos indicadores al invocar `clone()`, no habrá compartición, teniendo entonces una funcionalidad similar a la proporcionada por la llamada al sistema `fork()`.

El nivel variable de compartición es posible por la forma en que se representa una tarea en el *kernel* de Linux. Existe una estructura de datos del *kernel* específica (concretamente, `struct task_struct`) para cada tarea del sistema. Esta estructura, en lugar de almacenar datos para la tarea, contiene punteros a otras estructuras en las que se almacenan dichos datos, como por ejemplo estructuras de datos que representan la lista de archivos abiertos, información sobre el tratamiento de señales y la memoria virtual. Cuando se invoca `fork()`, se crea una nueva tarea junto con una *copia* de todas las estructuras de datos asociadas del proceso padre. También se crea una tarea nueva cuando se realiza la llamada al sistema `clone()`; sin embargo, en lugar de copiar todas las estructuras de datos, la nueva tarea *apunta* a las estructuras de datos de la tarea padre en función del conjunto de indicadores pasados a `clone()`.

4.6 Resumen

Una hebra es un flujo de control dentro de un proceso. Un proceso multihebra contiene varios flujos de control diferentes dentro del mismo espacio de direcciones. Las ventajas de los mecanismos multihebra son que proporcionan una mayor capacidad de respuesta al usuario, la compartición de recursos dentro del proceso, una mayor economía y la capacidad de aprovechar las ventajas de las arquitecturas multiprocesador.

Las hebras de nivel de usuario son hebras visibles para el programador y desconocidas para el *kernel*. El *kernel* del sistema operativo soporta y gestiona las hebras del nivel de *kernel*. En general las hebras de usuario se crean y gestionan más rápidamente que las del *kernel*, ya que no es necesaria la intervención del *kernel*. Existen tres modelos diferentes que permiten relacionar las hebras de usuario y las hebras del *kernel*: el modelo muchos-a-uno asigna muchas hebras de usuario a una sola hebra del *kernel*; el modelo uno-a-uno asigna a cada hebra de usuario la correspondiente hebra del *kernel*; el modelo muchos-a-muchos multiplexa muchas hebras de usuario sobre un número menor o igual de hebras del *kernel*.

La mayoría de los sistemas operativos modernos proporcionan soporte para hebras en el *kernel*; entre ellos se encuentran Windows 98, NT, 2000 y XP, así como Solaris y Linux.

Las bibliotecas de hebras proporcionan al programador de la aplicación una API para crear y gestionar hebras. Las tres bibliotecas de hebras principales de uso común son: Pthreads de POSIX, las hebras de Win32 para los sistemas Windows y las hebras de Java.

Los programas multihebra plantean muchos retos a los programadores, entre los que se incluye la semántica de las llamadas al sistema `fork()` y `exec()`. Otras cuestiones relevantes son la cancelación de hebras, el tratamiento de señales y los datos específicos de las hebras.

Ejercicios

- 4.1 Proporcione dos ejemplos de programación en los que los mecanismos multihebra *no* proporcionen un mejor rendimiento que una solución monohebra.
- 4.2 Describa las acciones que toma una biblioteca de hebras para cambiar el contexto entre hebras de nivel de usuario.
- 4.3 ¿Bajo qué circunstancias una solución multihebra que usa múltiples hebras del *kernel* proporciona un mejor rendimiento que una solución de una sola hebra sobre un sistema monocore?

- 4.4** ¿Cuáles de los siguientes componentes del estado de un programa se comparten entre las hebras de un proceso multihebra?
- Valores de los registros
 - Cúmulo de memoria
 - Variables globales
 - Memoria de pila
- 4.5** ¿Puede una solución multihebra que utilice múltiples hebras de usuario conseguir un mejor rendimiento en un sistema multiprocesador que en un sistema de un solo procesador?
- 4.6** Como se ha descrito en la Sección 4.5.2, Linux no diferencia entre procesos y hebras. En su lugar, Linux trata del mismo modo a ambos, permitiendo que una tarea se asemeje más a un proceso o a una hebra, en función del conjunto de indicadores que se pasen a la llamada al sistema `clone()`. Sin embargo, muchos sistemas operativos, como Windows XP y Solaris, tratan las hebras y los procesos de forma diferente. Normalmente, dichos sistemas usan una notación en la que la estructura de datos para un proceso contiene punteros a las distintas hebras pertenecientes al proceso. Compare estos dos métodos para el modelado de procesos y hebras dentro del *kernel*.
- 4.7** El programa mostrado en la Figura 4.11 usa la API de Pthreads. ¿Cuál sería la salida del programa en la LÍNEA C y en la LÍNEA P?

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* la hebra */

int main(int argc, char *argv[])
{
    int pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork();

    if (pid == 0) { /* proceso hijo */
        pthread_attr_init(&attr);
        pthread_create(&tid, &attr, runner, NULL);
        pthread_join(tid, NULL);
        printf ("HIJO: valor = %d", value); /* LÍNEA C*/
        return -1;
    }
    else if (pid > 0) /* proceso padre */
        wait(NULL);
        printf ("PADRE: valor = %d", value); /* LÍNEA P*/
    }
}

void *runner(void *param){
    value = 5;
    pthread_exit(0);
}
```

Figura 4.11 Programa en C para el Ejercicio 4.7.

- 4.8 Considere un sistema multiprocesador y un programa multihebra escrito usando el modo muchos-a-muchos. Suponga que el número de hebras de usuario en el programa es mayor que el número de procesadores del sistema. Explique el impacto sobre el rendimiento en los siguientes escenarios:
- El número de hebras del *kernel* asignadas al programa es menor que el número de procesadores.
 - El número de hebras del *kernel* asignadas al programa es igual que el número de procesadores.
 - El número de hebras del *kernel* asignadas al programa es mayor que el número de procesadores, pero menor que el número de hebras de usuario.
- 4.9 Escriba un programa multihebra Java, Pthreads o Win32 que genere como salida los sucesivos números primos. Este programa debe funcionar como sigue: el usuario ejecutará el programa e introducirá un número en la línea de comandos. Entonces, el programa creará una hebra nueva que dará como salida todos los números primos menores o iguales que el número especificado por el usuario.
- 4.10 Modifique el servidor horario basado en sockets (Figura 3.19) del Capítulo 3, de modo que el servidor dé servicio a cada solicitud de un cliente a través de una hebra distinta.
- 4.11 La secuencia de Fibonacci es la serie de números 0, 1, 1, 2, 3, 5, 8,... Formalmente se expresa como sigue:

$$\begin{aligned}fib_0 &= 0 \\fib_1 &= 1 \\fib_n &= fib_{n-1} + fib_{n-2}\end{aligned}$$

Escriba un programa multihebra que genere la serie de Fibonacci usando la biblioteca de hebras Java, Pthreads o Win32. Este programa funcionará del siguiente modo: el usuario especificará en la línea de comandos la cantidad de números de la secuencia de Fibonacci que el programa debe generar. El programa entonces creará una nueva hebra que generará los números de Fibonacci, colocando la secuencia en variables compartidas por todas las hebras (probablemente, una matriz será la estructura de datos más adecuada). Cuando la hebra termine de ejecutarse, la hebra padre proporcionará a la salida la secuencia generada por la hebra hijo. Dado que la hebra padre no puede comenzar a facilitar como salida la secuencia de Fibonacci hasta que la hebra hijo termine, hará falta que la hebra padre espere a que la hebra hijo concluya, mediante las técnicas descritas en la Sección 4.3.

- 4.12 El Ejercicio 3.9 del Capítulo 3 especifica el diseño de un servidor de eco usando la API de hebras Java. Sin embargo, este servidor es de una sola hebra, lo que significa que no puede responder a clientes de eco concurrentes hasta que el cliente actual concluya. Modifique la solución del ejercicio 3.9 de modo que el servidor de eco dé servicio a cada cliente mediante una hebra distinta.

Proyecto: multiplicación de matrices

Dadas dos matrices, A y B , donde A es una matriz con M filas y K columnas y la matriz B es una matriz con K filas y N columnas, la **matriz producto** de A por B es C , la cual tiene M filas y N columnas. La entrada de la matriz C en la fila i , columna j ($C_{i,j}$) es la suma de los productos de los elementos de la fila i de la matriz A y la columna j de la matriz B . Es decir,

$$C_{i,j} = \sum_{n=1}^K A_{i,n} \times B_{n,j}$$

Por ejemplo, si A fuera una matriz de 3 por 2 y B fuera una matriz de 2 por 3, el elemento $C_{3,1}$ sería la suma de $A_{3,1} \times B_{1,1}$ y $A_{3,2} \times B_{2,1}$

En este proyecto queremos calcular cada elemento $C_{i,j}$ mediante una hebra *de trabajo* diferente. Esto implica crear $M \times N$ hebras de trabajo. La hebra principal, o padre, inicializará las matrices A y B y asignará memoria suficiente para la matriz C , la cual almacenará el producto de las matrices A y B . Estas matrices se declararán como datos globales, con el fin de que cada hebra de trabajo tenga acceso a A , B y C .

Las matrices A y B pueden inicializarse de forma estática del siguiente modo:

```
#define M 3
#define K 2
#define N 3

int A [M] [K] = { {1,4}, {2,5}, {3,6} };
int B [K] [N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Alternativamente, pueden llenarse leyendo los valores de un archivo.

Paso de parámetros a cada hebra

La hebra padre creará $M \times N$ hebras de trabajo, pasando a cada hebra los valores de la fila i y la columna j que se van a usar para calcular la matriz producto. Esto requiere pasar dos parámetros a cada hebra. La forma más sencilla con Pthreads y Win32 es crear una estructura de datos usando struct. Los miembros de esta estructura son i y j , y la estructura es:

```
/* estructura para pasar datos a las hebras */

struct v
{
    int i; /* fila */
    int j; /* columna */
};
```

Los programas de Pthreads y Win32 crearán las hebras de trabajo usando una estrategia similar a la que se muestra a continuación:

```
/* Tenemos que crear M * N hebras de trabajo */

for (i = 0; i < M, i++)
    for (j = 0; j < N; j++ ) {
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data->i = i;
        data->j = j;
        /* Ahora creamos la hebra pasándola data como parámetro */
    }
}
```

El puntero data se pasará a la función de Pthreads `pthread_create()` o a la función de Win32 `CreateThread()`, que a su vez lo pasará como parámetro a la función que se va a ejecutar como una hebra independiente.

La compartición de datos entre hebras Java es diferente de la compartición entre hebras de Pthreads o Win32. Un método consiste en que la hebra principal cree e inicialice las matrices A , B y C . Esta hebra principal creará a continuación las hebras de trabajo, pasando las tres matrices, junto con la fila i y la columna j , al constructor de cada hebra de trabajo. Por tanto, el esquema de una hebra de trabajo será como sigue:

```
public class WorkerThread implements Runnable
```

```

{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

    public WorkerThread(int row, int col, int[][] A,
                        int[][] B, int[][] C) {
        this.row = row;
        this.col = col;
        this.A = A;
        this.B = B;
        this.C = C;
    }

    public void run() {
        /* calcular la matriz producto en C[row] [col] */
    }
}

```

Esperar a que se completen las hebras

Una vez que todas las hebras de trabajo se han completado, la hebra principal proporcionará como salida el producto contenido en la matriz *C*. Esto requiere que la hebra principal espere a que todas las hebras de trabajo terminen antes de poder poner en la salida el valor de la matriz producto. Se pueden utilizar varias estrategias para hacer que una hebra espere a que las demás terminen. La Sección 4.3 describe cómo esperar a que una hebra hijo se complete usando las bibliotecas de hebras de Pthreads, Win32 y Java. Win32 proporciona la función `WaitForSingleObject()`, mientras que Pthreads y Java usan, respectivamente, `pthread_join()` y `join()`. Sin embargo, en los ejemplos de programación de esa sección, la hebra padre espera a que una sola hebra hijo termine; completar el presente ejercicio requiere, por el contrario, esperar a que concluyan múltiples hebras.

En la Sección 4.3.2 hemos descrito la función `WaitForSingleObject()`, que se emplea para esperar a que una sola hebra termine. Sin embargo, la API de Win32 también proporciona la función `WaitForMultipleObjects()`, que se usa para esperar a que se completen múltiples hebras. A `WaitForMultipleObjects()` hay que pasarle cuatro parámetros:

1. El número de objetos por los que hay que esperar
2. Un puntero a la matriz de objetos.
3. Un indicador que muestre si todos los objetos han sido señalizados.
4. Tiempo de espera (o `INFINITE`).

Por ejemplo, si `THandles` es una matriz de objetos `HANDLE` (descriptores) de hebra de tamaño *N*, la hebra padre puede esperar a que todas las hebras hijo se completen con la siguiente instrucción:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

Una estrategia sencilla para esperar a que varias hebras terminen usando la función `pthread_join()` de Pthreads o `join()` de Java consiste en incluir la operación en un bucle `for`. Por ejemplo, podríamos esperar a que se completaran diez hebras usando el código de Pthreads de la Figura 4.12. El código equivalente usando Java se muestra en la Figura 4.13.

```
#define NUM_THREADS 10

/* una matriz de hebras que se van a esperar */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Figura 4.12 Código Pthread para esperar diez hebras.

```
final static int NUM_THREADS 10;

/* una matriz de hebras que se van a esperar */
Thread[] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
    try {
        workers[i].join();
    }catch (InterruptedException ie) {}
}
```

Figura 4.13 Código Java para esperar diez hebras.

Notas bibliográficas

Las cuestiones sobre el rendimiento de las hebras se estudian en Anderson et al. [1989], quien continuó su trabajo en Anderson et al. [1991] evaluando el rendimiento de las hebras de nivel de usuario con soporte del *kernel*. Bershad et al. [1990] describe la combinación de los mecanismos de hebras con las llamadas RPC. Engelschall [2000] expone una técnica para dar soporte a las hebras de nivel de usuario. Un análisis sobre el tamaño óptimo de los conjuntos compartidos de hebras es el que puede encontrarse en Ling et al. [2000]. Las activaciones del planificador se presentaron por primera vez en Anderson et al. [1991] y Williams [2002] aborda las activaciones del planificador en el sistema FreeBSD. Otros mecanismos mediante los que la biblioteca de hebras de usuario y el *kernel* pueden cooperar entre sí se tratan en Marsh et al. [1991], Govindan y Anderson [1991], Draves et al. [1991] y Black [1990]. Zabatta y Young [1998] comparan las hebras de Windows NT y Solaris en un multiprocesador simétrico. Pinilla y Gill [2003] comparan el rendimiento de las hebras Java en los sistemas Linux, Windows y Solaris.

Vahalia [1996] cubre el tema de las hebras en varias versiones de UNIX. Mauro y McDougall [2001] describen desarrollos recientes en el uso de hebras del *kernel* de Solaris. Solomon y Russinovich [2000] abordan el uso de hebras en Windows 2000. Bovet y Cesati [2002] explican cómo gestiona Linux el mecanismo de hebras.

En Lewis y Berg [1998] y Butenhof [1997] se proporciona información sobre programación con Pthreads. La información sobre programación de hebras en Solaris puede encontrarse en Sun Microsystems [1995]. Oaks y Wong [1999], Lewis y Berg [2000] y Holub [2000] abordan las soluciones multihebra en Java. Beveridge y Wiener [1997] y Cohen y Woodring [1997] describen las soluciones multihebra usando Win32.

Planificación de la CPU

Los mecanismos de planificación de la CPU son la base de los sistemas operativos multiprogramados. Mediante la conmutación de la CPU entre distintos procesos, el sistema operativo puede hacer que la computadora sea más productiva. En este capítulo, vamos a presentar los conceptos básicos sobre la planificación de la CPU y varios de los algoritmos utilizados para este fin. También consideraremos el problema de seleccionar el mejor algoritmo para un sistema particular.

En el Capítulo 4, hemos presentado el concepto de hebras en el modelo de procesos. En los sistemas operativos que las soportan, son las hebras del nivel del *kernel*, y no los procesos, las que de hecho son planificadas por el sistema operativo. Sin embargo, los términos **planificación de procesos** y **planificación de hebras** se usan indistintamente. En este capítulo, utilizaremos el término *planificación de procesos* cuando hablamos de los conceptos generales sobre planificación y *planificación de hebras* cuando hagamos referencia a conceptos específicos de las hebras.

OBJETIVOS DEL CAPÍTULO

- Presentar los mecanismos de planificación de la CPU, que constituyen los cimientos de los sistemas operativos multiprogramados.
- Describir los distintos algoritmos para la planificación de la CPU.
- Exponer los criterios de evaluación utilizados para seleccionar un algoritmo de planificación de la CPU para un determinado sistema.

5.1 Conceptos básicos

En un sistema de un único procesador, sólo puede ejecutarse un proceso cada vez; cualquier otro proceso tendrá que esperar hasta que la CPU quede libre y pueda volver a planificarse. El objetivo de la multiprogramación es tener continuamente varios procesos en ejecución, con el fin de maximizar el uso de la CPU. La idea es bastante simple: un proceso se ejecuta hasta que tenga que esperar, normalmente porque es necesario completar alguna solicitud de E/S. En un sistema informático simple, la CPU permanece entonces inactiva y todo el tiempo de espera se desperdicia; no se realiza ningún trabajo útil. Con la multiprogramación, se intenta usar ese tiempo de forma productiva. En este caso, se mantienen varios procesos en memoria a la vez. Cuando un proceso tiene que esperar, el sistema operativo retira el uso de la CPU a ese proceso y se lo cede a otro proceso. Este patrón se repite continuamente y cada vez que un proceso tiene que esperar, otro proceso puede hacer uso de la CPU.

Este tipo de planificación es una función fundamental del sistema operativo; casi todos los recursos de la computadora se planifican antes de usarlos. Por supuesto, la CPU es uno de los principales recursos de la computadora, así que su correcta planificación resulta crucial en el diseño del sistema operativo.

5.1.1 Ciclo de ráfagas de CPU y de E/S

La adecuada planificación de la CPU depende de una propiedad observada de los procesos: la ejecución de un proceso consta de un **ciclo** de ejecución en la CPU, seguido de una espera de E/S; los procesos alternan entre estos dos estados. La ejecución del proceso comienza con una **ráfaga de CPU**. Ésta va seguida de una **ráfaga de E/S**, a la cual sigue otra ráfaga de CPU, luego otra ráfaga de E/S, etc. Finalmente, la ráfaga final de CPU concluye con una solicitud al sistema para terminar la ejecución (Figura 5.1).

La duración de las ráfagas de CPU se ha medido exhaustivamente en la práctica. Aunque varían enormemente de un proceso a otro y de una computadora a otra, tienden a presentar una curva de frecuencia similar a la mostrada en la Figura 5.2. Generalmente, la curva es de tipo exponencial o hiperexponencial, con un gran número de ráfagas de CPU cortas y un número menor de ráfagas de CPU largas. Normalmente, un programa limitado por E/S presenta muchas ráfagas de CPU cortas. Esta distribución puede ser importante en la selección de un algoritmo apropiado para la planificación de la CPU.

5.1.2 Planificador de la CPU

Cuando la CPU queda inactiva, el sistema operativo debe seleccionar uno de los procesos que se encuentran en la cola de procesos preparados para ejecución. El **planificador a corto plazo** (o planificador de la CPU) lleva a cabo esa selección del proceso. El planificador elige uno de los procesos que están en memoria preparados para ejecutarse y asigna la CPU a dicho proceso.

Observe que la cola de procesos preparados no necesariamente tiene que ser una cola FIFO (first-in, first-out). Como veremos al considerar los distintos algoritmos de planificación, una cola de procesos preparados puede implementarse como una cola FIFO, una cola prioritaria, un árbol o simplemente una lista enlazada no ordenada. Sin embargo, conceptualmente, todos los proce-

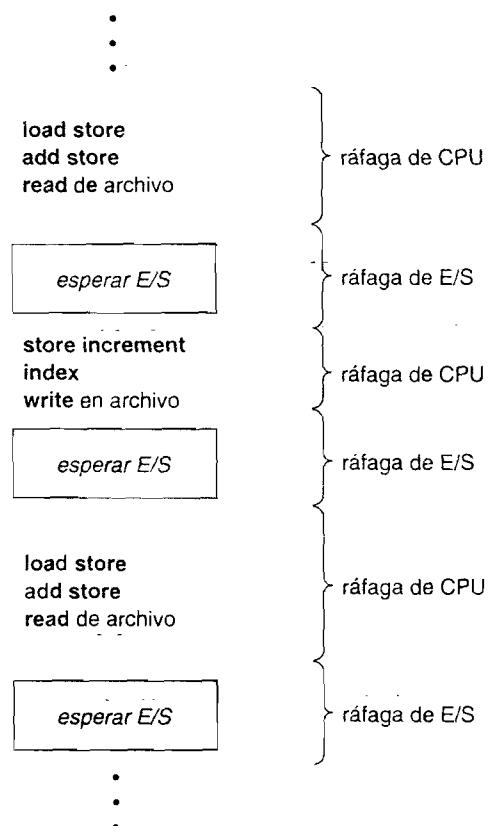


Figura 5.1 Secuencia alternante de ráfagas de CPU y de E/S

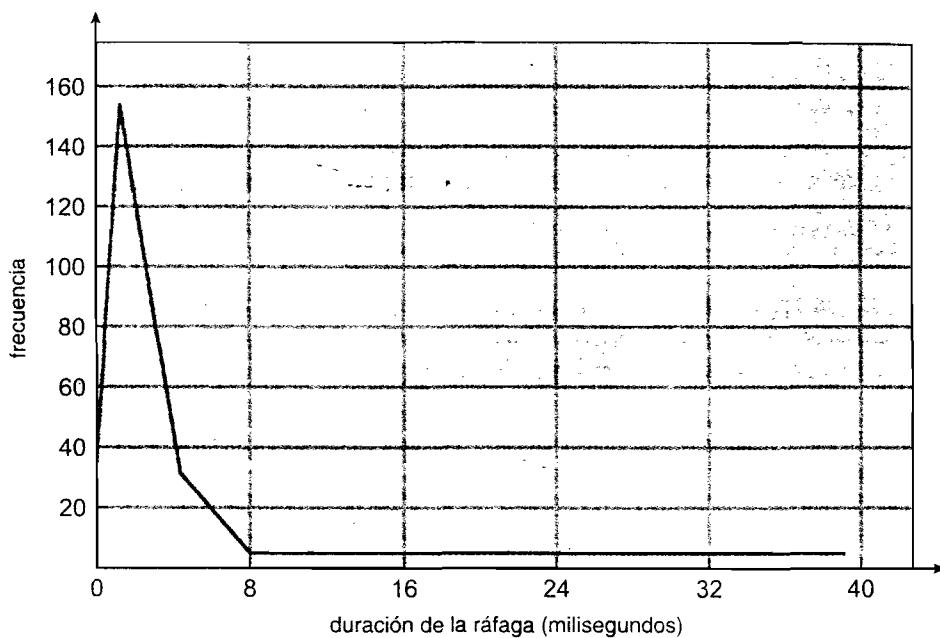


Figura 5.2 Histograma de duración de las ráfagas de CPU.

sos que se encuentran en la cola de procesos preparados se ponen en fila esperando la oportunidad de ejecutarse en la CPU. Los registros que se almacenan en las colas son, generalmente, bloques de control de proceso (PCB) que describen los procesos en cuestión.

5.1.3 Planificación apropiativa

Puede ser necesario tomar decisiones sobre planificación de la CPU en las siguientes cuatro circunstancias:

1. Cuando un proceso cambia del estado de ejecución al estado de espera (por ejemplo, como resultado de una solicitud de E/S o de una invocación de `wait` para esperar a que termine uno de los procesos hijo).
2. Cuando un proceso cambia del estado de ejecución al estado preparado (por ejemplo, cuando se produce una interrupción).
3. Cuando un proceso cambia del estado de espera al estado preparado (por ejemplo, al completarse una operación de E/S).
4. Cuando un proceso termina.

En las situaciones 1 y 4, no hay más que una opción en términos de planificación: debe seleccionarse un nuevo proceso para su ejecución (si hay algún proceso en la cola de procesos preparados). Sin embargo, en las situaciones 2 y 3 sí que existe la opción de planificar un nuevo proceso o no.

Cuando las decisiones de planificación sólo tienen lugar en las circunstancias 1 y 4, decimos que el esquema de planificación es **sin desalojo** o **cooperativo**; en caso contrario, se trata de un esquema **apropiativo**. En la planificación sin desalojo, una vez que se ha asignado la CPU a un proceso, el proceso se mantiene en la CPU hasta que ésta es liberada bien por la terminación del proceso o bien por la conmutación al estado de espera. Este método de planificación era el utilizado por Microsoft Windows 3.x; Windows 95 introdujo la planificación apropiativa y todas las versiones siguientes de los sistemas operativos Windows han usado dicho tipo de planificación. El sistema operativo Mac OS X para Macintosh utiliza la planificación apropiativa; las versiones anteriores del sistema operativo de Macintosh se basaban en la planificación cooperativa. La planificación cooperativa es el único método que se puede emplear en determinadas plataformas

hardware, dado que no requiere el hardware especial (por ejemplo, un temporizador) necesario para la planificación apropiativa.

Lamentablemente, la planificación apropiativa tiene un coste asociado con el acceso a los datos compartidos. Considere el caso de dos procesos que comparten datos y suponga que, mientras que uno está actualizando los datos, resulta desalojado con el fin de que el segundo proceso pueda ejecutarse. El segundo proceso podría intentar entonces leer los datos, que se encuentran en un estado incoherente. En tales situaciones, son necesarios nuevos mecanismos para coordinar el acceso a los datos compartidos; veremos este tema en el Capítulo 6.

La técnica de desalojo también afecta al diseño del *kernel* del sistema operativo. Durante el procesamiento de una llamada al sistema, el *kernel* puede estar ocupado realizando una actividad en nombre de un proceso. Tales actividades pueden implicar cambios importantes en los datos del *kernel* (por ejemplo, en las colas de E/S). ¿Qué ocurre si el proceso se desaloja en mitad de estos cambios y el *kernel* (o el controlador del dispositivo) necesita leer o modificar la misma estructura? El resultado será un auténtico caos. Ciertos sistemas operativos, incluyendo la mayor parte de las versiones de UNIX, resuelven este problema esperando a que se complete la llamada al sistema o a que se transfiera un bloque de E/S antes de hacer un cambio de contexto. Esta solución permite obtener una estructura del *kernel* simple, ya que el *kernel* no desalojará ningún proceso mientras que las estructuras de datos del *kernel* se encuentren en un estado incoherente. Lamentablemente, este modelo de ejecución del *kernel* no resulta adecuado para permitir la realización de cálculos en tiempo real y el multiprocesamiento. Estos problemas y sus soluciones se describen en las Secciones 5.4 y 19.5.

Dado que, por definición, las interrupciones pueden producirse en cualquier momento, y puesto que no siempre pueden ser ignoradas por el *kernel*, las secciones de código afectadas por las interrupciones deben ser resguardadas de un posible uso simultáneo. Sin embargo, el sistema operativo tiene que aceptar interrupciones casi continuamente, ya que de otra manera podrían perderse valores de entrada o sobreescribirse los valores de salida; por esto, para que no puedan acceder de forma concurrente varios procesos a estas secciones de código, lo que se hace es desactivar las interrupciones al principio de cada sección y volver a activarlas al final. Es importante observar que no son muy numerosas las secciones de código que desactivan las interrupciones y que, normalmente, esas secciones contienen pocas instrucciones.

5.1.4 Despachador

Otro componente implicado en la función de planificación de la CPU es el **despachador**. El despachador es el módulo que proporciona el control de la CPU a los procesos seleccionados por el planificador a corto plazo. Esta función implica lo siguiente:

- Cambio de contexto.
- Cambio al modo usuario.
- Salto a la posición correcta dentro del programa de usuario para reiniciar dicho programa.

El despachador debe ser lo más rápido posible, ya que se invoca en cada conmutación de proceso. El tiempo que tarda el despachador en detener un proceso e iniciar la ejecución de otro se conoce como **latencia de despacho**.

5.2 Criterios de planificación

Los diferentes algoritmos de planificación de la CPU tienen distintas propiedades, y la elección de un algoritmo en particular puede favorecer una clase de procesos sobre otros. A la hora de decidir qué algoritmo utilizar en una situación particular, debemos considerar las propiedades de los diversos algoritmos.

Se han sugerido muchos criterios para comparar los distintos algoritmos de planificación. Las características que se usen para realizar la comparación pueden afectar enormemente a la determinación de cuál es el mejor algoritmo. Los criterios son los siguientes:

- **Utilización de la CPU.** Deseamos mantener la CPU tan ocupada como sea posible. Conceptualmente, la utilización de la CPU se define en el rango comprendido entre el 0 y el 100 por cien. En un sistema real, debe variar entre el 40 por ciento (para un sistema ligeramente cargado) y el 90 por ciento (para un sistema intensamente utilizado).
- **Tasa de procesamiento.** Si la CPU está ocupada ejecutando procesos, entonces se estará llevando a cabo algún tipo de trabajo. Una medida de esa cantidad de trabajo es el número de procesos que se completan por unidad de tiempo, y dicha medida se denomina *tasa de procesamiento*. Para procesos de larga duración, este valor puede ser de un proceso por hora; para transacciones cortas, puede ser de 10 procesos por segundo.
- **Tiempo de ejecución.** Desde el punto de vista de un proceso individual, el criterio importante es cuánto tarda en ejecutarse dicho proceso. El intervalo que va desde el instante en que se ordena la ejecución de un proceso hasta el instante en que se completa es el *tiempo de ejecución*. Ese tiempo de ejecución es la suma de los períodos que el proceso invierte en esperar para cargarse en memoria, esperar en la cola de procesos preparados, ejecutarse en la CPU y realizar las operaciones de E/S.
- **Tiempo de espera.** El algoritmo de planificación de la CPU no afecta a la cantidad de tiempo durante la que un proceso se ejecuta o hace una operación de E/S; afecta sólo al período de tiempo que un proceso invierte en esperar en la cola de procesos preparados. El *tiempo de espera* es la suma de los períodos invertidos en esperar en la cola de procesos preparados.
- **Tiempo de respuesta.** En un sistema interactivo, el tiempo de ejecución puede no ser el mejor criterio. A menudo, un proceso puede generar parte de la salida con relativa rapidez y puede continuar calculando nuevos resultados mientras que los resultados previos se envían a la salida para ponerlos a disposición del usuario. Por tanto, otra medida es el tiempo transcurrido desde que se envía una solicitud hasta que se produce la primera respuesta. Esta medida, denominada *tiempo de respuesta*, es el tiempo que el proceso tarda en empezar a responder, no el tiempo que tarda en enviar a la salida toda la información de respuesta. Generalmente, el tiempo de respuesta está limitado por la velocidad del dispositivo de salida.

El objetivo consiste en maximizar la utilización de la CPU y la tasa de procesamiento, y minimizar el tiempo de ejecución, el tiempo de espera y el tiempo de respuesta. En la mayoría de los casos, lo que se hace es optimizar algún tipo de valor promedio. Sin embargo, en determinadas circunstancias, resulta deseable optimizar los valores máximo y mínimo en lugar del promedio. Por ejemplo, para garantizar que todos los usuarios tengan un buen servicio, podemos tratar de minimizar el tiempo de respuesta máximo.

Diversos investigadores han sugerido que, para los sistemas interactivos (como, por ejemplo, los sistemas de tiempo compartido), es más importante minimizar la *varianza* del tiempo de respuesta que minimizar el tiempo medio de respuesta. Un sistema con un tiempo de respuesta razonable y *predecible* puede considerarse más deseable que un sistema que sea más rápido por término medio, pero que resulte altamente variable. Sin embargo, no se han llevado a cabo muchas investigaciones sobre algoritmos de planificación de la CPU que minimicen la varianza.

A medida que estudiemos en la sección siguiente los diversos algoritmos de planificación de la CPU, trataremos de ilustrar su funcionamiento. Para poder ilustrar ese funcionamiento de forma precisa, sería necesario utilizar muchos procesos, compuesto cada uno de una secuencia de varios cientos de ráfagas de CPU y de E/S. No obstante, con el fin de simplificar, en nuestros ejemplos sólo vamos a considerar una ráfaga de CPU (en milisegundos) por cada proceso. La medida de comparación que hemos empleado es el tiempo medio de espera. En la Sección 5.7 se presenta un mecanismo de evaluación más elaborado.

5.3 Algoritmos de planificación

La cuestión de la planificación de la CPU aborda el problema de decidir a qué proceso de la cola de procesos preparados debe asignársele la CPU. Existen muchos algoritmos de planificación de la CPU; en esta sección, describiremos algunos de ellos.

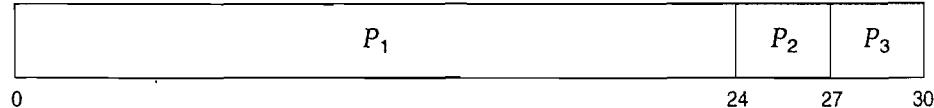
5.3.1 Planificación FCFS

El algoritmo más simple de planificación de la CPU es, con mucho, el algoritmo FCFS (first-come first-served; primero en llegar, primero en ser servido). Con este esquema, se asigna primero la CPU al proceso que primero la solicite. La implementación de la política FCFS se gestiona fácilmente con una cola FIFO. Cuando un proceso entra en la cola de procesos preparados, su PCB se coloca al final de la cola. Cuando la CPU queda libre, se asigna al proceso que esté al principio de la cola y ese proceso que pasa a ejecutarse se elimina de la cola. El código del algoritmo de planificación FCFS es simple de escribir y fácil de comprender.

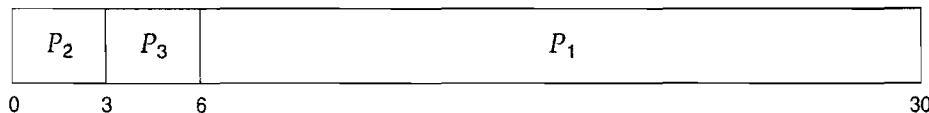
Sin embargo, el tiempo medio de espera con el algoritmo FCFS es a menudo bastante largo. Suponga que el siguiente conjunto de procesos llega en el instante 0, estando la duración de la ráfaga de CPU especificada en milisegundos:

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si los procesos llegan en el orden P_1 , P_2 , P_3 , y se sirven según el orden FCFS, obtendremos el resultado mostrado en el siguiente **diagrama de Gantt**:



El tiempo de espera es de 10 milisegundos para el proceso P_1 , de 24 milisegundos para el proceso P_2 y de 27 milisegundos para el proceso P_3 . Por tanto, el tiempo medio de espera es de $(0 + 24 + 27)/3 = 17$ milisegundos. Sin embargo, si los procesos llegan en el orden P_2 , P_3 , P_1 , los resultados serán los mostrados en el siguiente diagrama de Gantt:



Ahora el tiempo de espera es de $(6 + 0 + 3)/3 = 3$ milisegundos. Esta reducción es sustancial. Por tanto, el tiempo medio de espera con una política FCFS no es, generalmente, mínimo y puede variar significativamente si la duración de las ráfagas de CPU de los procesos es muy variable.

Además, considere el comportamiento del algoritmo de planificación FCFS en un caso dinámico. Suponga que tenemos un proceso limitado por CPU y muchos procesos limitados por E/S. A medida que los procesos fluyen por el sistema, puede llegarse al siguiente escenario: el proceso limitado por CPU obtendrá y mantendrá la CPU; durante ese tiempo, los demás procesos terminarán sus operaciones de E/S y pasarán a la cola de procesos preparados, esperando a acceder a la CPU. Mientras que los procesos esperan en la cola de procesos preparados, los dispositivos de E/S están inactivos. Finalmente, el proceso limitado por CPU termina su ráfaga de CPU y pasa a esperar por un dispositivo de E/S. Todos los procesos limitados por E/S, que tienen ráfagas de CPU cortas, se ejecutan rápidamente y vuelven a las colas de E/S. En este punto, la CPU permanecerá inactiva. El proceso limitado por CPU volverá en algún momento a la cola de procesos preparados

y se le asignará la CPU. De nuevo, todos los procesos limitados por E/S terminarán esperando en la cola de procesos preparados hasta que el proceso limitado por CPU haya terminado. Se produce lo que se denomina un **efecto convoy** a medida que todos los procesos restantes esperan a que ese único proceso de larga duración deje de usar la CPU. Este efecto da lugar a una utilización de la CPU y de los dispositivos menor que la que se conseguiría si se permitiera a los procesos más cortos ejecutarse primero.

El algoritmo de planificación FCFS es cooperativo. Una vez que la CPU ha sido asignada a un proceso, dicho proceso conserva la CPU hasta que la libera, bien porque termina su ejecución o porque realiza una solicitud de E/S. El algoritmo FCFS resulta, por tanto, especialmente problemático en los sistemas de tiempo compartido, donde es importante que cada usuario obtenga una cuota de la CPU a intervalos regulares. Sería desastroso permitir que un proceso mantuviera la CPU durante un largo período de tiempo.

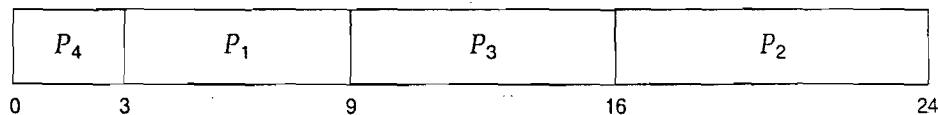
5.3.2 Planificación SJF

Otro método de planificación de la CPU es el algoritmo de planificación con selección del trabajo más corto (SJF, shortest-job-first). Este algoritmo asocia con cada proceso la duración de la siguiente ráfaga de CPU del proceso. Cuando la CPU está disponible, se asigna al proceso que tiene la siguiente ráfaga de CPU más corta. Si las siguientes ráfagas de CPU de dos procesos son iguales, se usa la planificación FCFS para romper el empate. Observe que un término más apropiado para este método de planificación sería el de *algoritmo de la siguiente ráfaga de CPU más corta*, ya que la planificación depende de la duración de la siguiente ráfaga de CPU de un proceso, en lugar de depender de su duración total. Usamos el término SJF porque casi todo el mundo y gran parte de los libros de texto emplean este término para referirse a este tipo de planificación.

Como ejemplo de planificación SJF, considere el siguiente conjunto de procesos, estando especificada la duración de la ráfaga de CPU en milisegundos:

Proceso	Tiempo de ráfaga
P_1	6
P_2	8
P_3	7
P_4	3

Usando la planificación SJF, planificaríamos estos procesos de acuerdo con el siguiente diagrama de Gantt:



El tiempo de espera es de 3 milisegundos para el proceso P_1 , de 16 milisegundos para el proceso P_2 , de 9 milisegundos para P_3 y de 0 milisegundos para P_4 . Por tanto, el tiempo medio de espera es de $(3 + 16 + 9 + 0)/4 = 7$ milisegundos. Por comparación, si estuviéramos usando el esquema de planificación FCFS, el tiempo medio de espera sería de 10,25 milisegundos.

El algoritmo de planificación SJF es probablemente *óptimo*, en el sentido de que proporciona el tiempo medio de espera mínimo para un conjunto de procesos dado. Anteponer un proceso corto a uno largo disminuye el tiempo de espera del proceso corto en mayor medida de lo que incrementa el tiempo de espera del proceso largo. Consecuentemente, el tiempo *medio* de espera disminuye.

La dificultad real del algoritmo SJF es conocer la duración de la siguiente solicitud de CPU. En una planificación a largo plazo de trabajos en un sistema de procesamiento por lotes, podemos usar como duración el límite de tiempo del proceso que el usuario especifique en el momento de enviar el trabajo. Con este mecanismo, los usuarios están motivados para estimar el límite de tiempo del proceso de forma precisa, dado que un valor menor puede significar una respuesta más

rápida. (Un valor demasiado bajo producirá un error de límite de tiempo excedido y será necesario reenviar el proceso.) La planificación SJF se usa frecuentemente como mecanismo de planificación a largo plazo.

Aunque el algoritmo SJF es óptimo, no se puede implementar en el nivel de la planificación de la CPU a corto plazo, ya que no hay forma de conocer la duración de la siguiente ráfaga de CPU. Un método consiste en intentar aproximar la planificación SJF: podemos no *conocer* la duración de la siguiente ráfaga de CPU, pero podemos *predecir* su valor, por el procedimiento de confiar en que la siguiente ráfaga de CPU será similar en duración a las anteriores. De este modo, calculando una aproximación de la duración de la siguiente ráfaga de CPU, podemos tomar el proceso que tenga la ráfaga de CPU predicha más corta.

Generalmente, la siguiente ráfaga de CPU se predice como la media exponencial de las duraciones medidas de las anteriores ráfagas de CPU. Sea t_n la duración de la n -ésima ráfaga de CPU y sea τ_{n+1} el valor predicho para la siguiente ráfaga de CPU. Entonces, para α , $0 \leq \alpha \leq 1$, se define

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

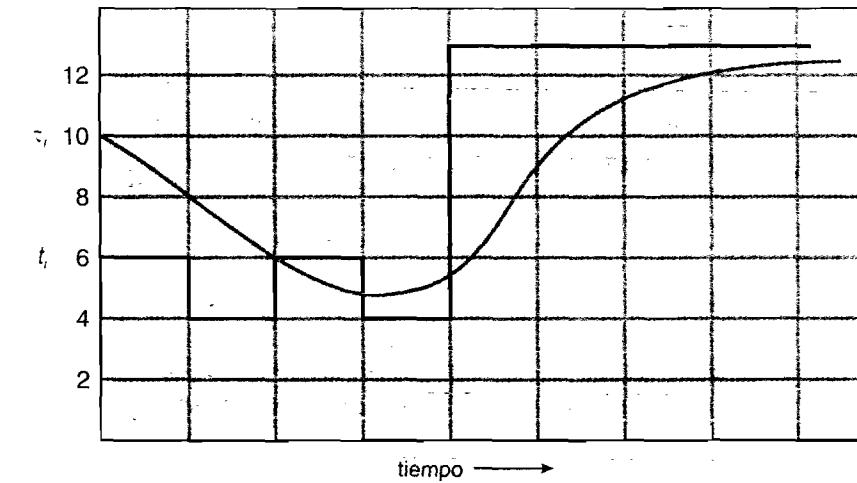
Esta fórmula define un **promedio exponencial**. El valor de t_n contiene la información más reciente; τ_n almacena el historial pasado. El parámetro α controla el peso relativo del historial reciente y pasado de nuestra predicción. Si $\alpha = 0$, entonces $\tau_{n+1} = \tau_n$, y el historial reciente no tiene ningún efecto (se supone que las condiciones actuales van a ser transitorias); si $\alpha = 1$, entonces $\tau_{n+1} = t_n$ y sólo la ráfaga de CPU más reciente importa (se supone que el historial es obsoleto y, por tanto, irrelevante). Frecuentemente, $\alpha = \frac{1}{2}$, en cuyo caso, el historial reciente y pasado tienen el mismo peso. El valor inicial τ_0 puede definirse como una constante o como un promedio global para todo el sistema. La Figura 5.3 muestra un promedio exponencial con $\alpha = \frac{1}{2}$ y $\tau_0 = 10$.

Para comprender el comportamiento del promedio exponencial, podemos desarrollar la fórmula para τ_{n+1} sustituyendo el valor de τ_n y de los términos sucesivos, obteniendo

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n-1} \tau_0$$

Dado que tanto α como $1 - \alpha$ son menores o iguales a 1, cada término sucesivo tiene menor peso que su predecesor.

El algoritmo SJF puede ser cooperativo o apropiativo. La necesidad de elegir surge cuando un proceso llega a la cola de procesos preparados mientras que un proceso anterior está todavía en



ráfaga de CPU (t_n)	6	4	6	4	13	13	13	...	
"invitado" (τ_n)	10	8	6	6	5	9	11	12	...

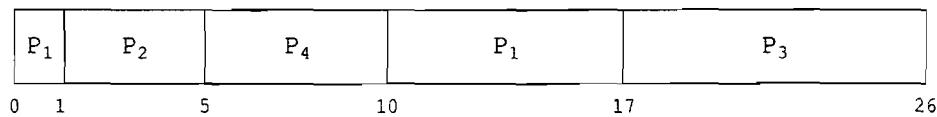
Figura 5.3 Predicción de la duración de la siguiente ráfaga de CPU.

ejecución. La siguiente ráfaga de CPU del proceso que acaba de llegar puede ser más corta que lo que quede del proceso actualmente en ejecución. Un algoritmo SJF apropiativo detendrá el proceso actualmente en ejecución, mientras que un algoritmo sin desalojo permitirá que dicho proceso termine su ráfaga de CPU. La planificación SJF apropiativa a veces se denomina **planificación con selección del proceso con tiempo restante más corto**.

Como ejemplo, considere los cuatro procesos siguientes, estando especificada la duración de la ráfaga de CPU en milisegundos:

Proceso	Tiempo de llegada	Tiempo de ráfaga
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Si los procesos llegan a la cola de procesos preparados en los instantes que se muestran y necesitan los tiempos de ráfaga indicados, entonces la planificación SJF apropiativa es la que se muestra en el siguiente diagrama de Gantt:



El proceso P_1 se inicia en el instante 0, dado que es el único proceso que hay en la cola. El proceso P_2 llega en el instante 1. El tiempo que le queda al proceso P_1 (7 milisegundos) es mayor que el tiempo requerido por el proceso P_2 (4 milisegundos), por lo que el proceso P_1 se desaloja y se planifica el proceso P_2 . El tiempo medio de espera en este ejemplo es de $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)) / 4 = 26 / 4 = 6,5$ milisegundos. La planificación SJF cooperativa proporcionaría un tiempo medio de espera de 7,75 milisegundos.

5.3.3 Planificación por prioridades

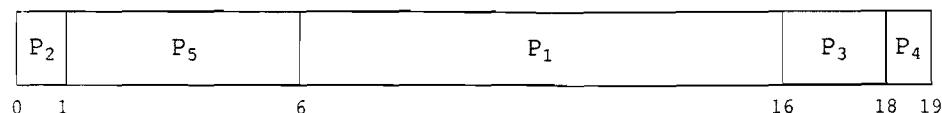
El algoritmo SJF es un caso especial del **algoritmo de planificación por prioridades** general. A cada proceso se le asocia una prioridad y la CPU se asigna al proceso que tenga la prioridad más alta. Los procesos con la misma prioridad se planifican en orden FCFS. Un algoritmo SJF es simplemente un algoritmo por prioridades donde la prioridad (p) es el inverso de la siguiente ráfaga de CPU (predicha). Cuanto más larga sea la ráfaga de CPU, menor será la prioridad y viceversa.

Observe que al hablar de planificación pensamos en términos de *alta* prioridad y *baja* prioridad. Generalmente, las prioridades se indican mediante un rango de números fijo, como por ejemplo de 0 a 7, o de 0 a 4095. Sin embargo, no existe un acuerdo general sobre si 0 es la prioridad más alta o la más baja. Algunos sistemas usan los números bajos para representar una prioridad baja; otros, emplean números bajos para especificar una prioridad alta; esta diferencia puede llevar a confusión. En este texto, vamos a asumir que los números bajos representan una alta prioridad.

Por ejemplo, considere el siguiente conjunto de procesos y suponga que llegan en el instante 0 en este orden: P_1, P_2, \dots, P_5 , estando la duración de las ráfagas de CPU especificada en milisegundos:

Proceso	Tiempo de ráfaga	Prioridad
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando la planificación por prioridades, vamos a planificar estos procesos de acuerdo con el siguiente diagrama de Gantt:



El tiempo medio de espera es de 8,2 milisegundos.

Las prioridades pueden definirse interna o externamente. Las prioridades definidas internamente utilizan algún valor mensurable para calcular la prioridad de un proceso. Por ejemplo, para calcular las prioridades se han usado en diversos sistemas magnitudes tales como los límites de tiempo, los requisitos de memoria, el número de archivos abiertos y la relación entre la ráfaga de E/S promedio y la ráfaga de CPU promedio. Las prioridades definidas externamente se establecen en función de criterios externos al sistema operativo, como por ejemplo la importancia del proceso, el coste monetario de uso de la computadora, el departamento que patrocina el trabajo y otros factores, a menudo de carácter político.

La planificación por prioridades puede ser apropiativa o cooperativa. Cuando un proceso llega a la cola de procesos preparados, su prioridad se compara con la prioridad del proceso actualmente en ejecución. Un algoritmo de planificación por prioridades apropiativo, expulsará de la CPU al proceso actual si la prioridad del proceso que acaba de llegar es mayor. Un algoritmo de planificación por prioridades cooperativo simplemente pondrá el nuevo proceso al principio de la cola de procesos preparados.

Un problema importante de los algoritmos de planificación por prioridades es el **bloqueo indefinido** o la **muerte por inanición**. Un proceso que está preparado para ejecutarse pero está esperando a acceder a la CPU puede considerarse bloqueado; un algoritmo de planificación por prioridades puede dejar a algunos procesos de baja prioridad esperando indefinidamente. En un sistema informático con una carga de trabajo grande, un flujo estable de procesos de alta prioridad puede impedir que un proceso de baja prioridad llegue a la CPU. Generalmente, ocurrirá una de dos cosas: o bien el proceso se ejecutará finalmente (a las 2 de la madrugada del domingo, cuando el sistema finalmente tenga una menor carga de trabajo) o bien el sistema informático terminará fallando y se perderán todos los procesos con baja prioridad no terminados. Se rumorea que, en 1973, en el MIT, cuando apagaron el IBM 7094, encontraron un proceso de baja prioridad que había sido enviado en 1967 y todavía no había sido ejecutado.

Una solución al problema del bloqueo indefinido de los procesos de baja prioridad consiste en aplicar mecanismos de **envejecimiento**. Esta técnica consiste en aumentar gradualmente la prioridad de los procesos que estén esperando en el sistema durante mucho tiempo. Por ejemplo, si el rango de prioridades va desde 127 (baja) a 0 (alta), podríamos restar 1 a la prioridad de un proceso en espera cada 15 minutos. Finalmente, incluso un proceso con una prioridad inicial de 127 llegaría a tener la prioridad más alta del sistema y podría ejecutarse. De hecho, un proceso con prioridad 127 no tardaría más de 32 horas en convertirse en un proceso con prioridad 0.

5.3.4 Planificación por turnos

El algoritmo de planificación por turnos (RR, round robin) está diseñado especialmente para los sistemas de tiempo compartido. Es similar a la planificación FCFS, pero se añade la técnica de desalojo para conmutar entre procesos. En este tipo de sistema se define una pequeña unidad de tiempo, denominada **cuanto de tiempo**, o franja temporal. Generalmente, el cuanto de tiempo se encuentra en el rango comprendido entre 10 y 100 milisegundos. La cola de procesos preparados se trata como una cola circular. El planificador de la CPU recorre la cola de procesos preparados, asignando la CPU a cada proceso durante un intervalo de tiempo de hasta 1 cuanto de tiempo.

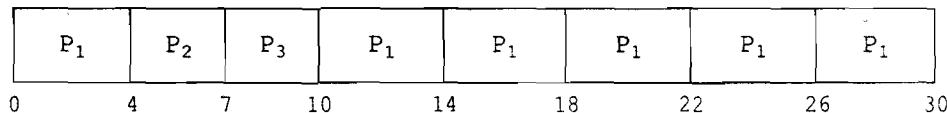
Para implementar la planificación por turnos, mantenemos la cola de procesos preparados como una cola FIFO de procesos. Los procesos nuevos se añaden al final de la cola de procesos preparados. El planificador de la CPU toma el primer proceso de la cola de procesos preparados, configura un temporizador para que interrumpa pasado 1 cuanto de tiempo y despacha el proceso.

Puede ocurrir una de dos cosas. El proceso puede tener una ráfaga de CPU cuya duración sea menor que 1 cuento de tiempo; en este caso, el propio proceso liberará voluntariamente la CPU. El planificador continuará entonces con el siguiente proceso de la cola de procesos preparados. En caso contrario, si la ráfaga de CPU del proceso actualmente en ejecución tiene una duración mayor que 1 cuento de tiempo, se producirá un fin de cuenta del temporizador y éste enviará una interrupción al sistema operativo; entonces se ejecutará un cambio de contexto y el proceso se colocará al final de la cola de procesos preparados. El planificador de la CPU seleccionará a continuación el siguiente proceso de la cola de procesos preparados.

El tiempo medio de espera en los sistemas por turnos es, con frecuencia, largo. Considere el siguiente conjunto de procesos que llegan en el instante 0, estando especificada la duración de las ráfagas de CPU en milisegundos:

Proceso	Tiempo de ráfaga
P_1	24
P_2	3
P_3	3

Si usamos un cuento de tiempo de 4 milisegundos, entonces el proceso P_1 obtiene los 4 primeros milisegundos. Dado que necesita otros 20 milisegundos, es desalojado después del primer cuento de tiempo, y la CPU se concede al siguiente proceso de la cola, el proceso P_2 . Puesto que el proceso P_2 no necesita 4 milisegundos, termina antes de que caduque su cuento de tiempo. Entonces la CPU se proporciona al siguiente proceso, el proceso P_3 . Una vez que cada proceso ha recibido 1 cuento de tiempo, la CPU es devuelta al proceso P_1 para un cuento de tiempo adicional. La planificación por turnos resultante es:



El tiempo medio de espera es de $17/3 = 5,66$ milisegundos.

En el algoritmo de planificación por turnos, a ningún proceso se le asigna la CPU por más de 1 cuento de tiempo en cada turno (a menos que sea el único proceso ejecutable). Si una ráfaga de CPU de un proceso excede 1 cuento de tiempo, dicho proceso se *desaloja* y se coloca de nuevo en la cola de procesos preparados. El algoritmo de planificación por turnos incluye, por tanto, un mecanismo de desalojo.

Si hay n procesos en la cola de procesos preparados y el cuento de tiempo es q , cada proceso obtiene $1/n$ del tiempo de CPU en partes de como máximo q unidades de tiempo. Cada proceso no tiene que esperar más de $(n - 1) \times q$ unidades de tiempo hasta obtener su siguiente cuento de tiempo. Por ejemplo, con cinco procesos y un cuento de tiempo de 20 milisegundos, cada proceso obtendrá 20 milisegundos cada 100 milisegundos.

El rendimiento del algoritmo de planificación por turnos depende enormemente del tamaño del cuento de tiempo. Por un lado, si el cuento de tiempo es extremadamente largo, la planificación por turnos es igual que la FCFS. Si el cuento de tiempo es muy pequeño (por ejemplo, 1 milisegundo), el método por turnos se denomina **compartición del procesador** y (en teoría) crea la apariencia de que cada uno de los n procesos tiene su propio procesador ejecutándose a $1/n$ de la velocidad del procesador real. Este método se usaba en las máquinas de Control Data Corporation (CDC) para implementar diez procesadores periféricos con sólo un conjunto de hardware y diez conjuntos de registros. El hardware ejecuta una instrucción para un conjunto de registros y luego pasa al siguiente; este ciclo se repite, dando lugar en la práctica a diez procesadores lentos, en lugar de uno rápido. Realmente, dado que el procesador era mucho más rápido que la memoria y cada instrucción hacía referencia a memoria, los procesadores no eran mucho más lentos que lo que hubieran sido diez procesadores reales.

En software, también necesitamos considerar el efecto del cambio de contexto en el rendimiento de la planificación por turnos. Suponga que sólo tenemos un proceso de 10 unidades de tiem-

po. Si el cuento tiene 12 unidades de tiempo, el proceso termina en menos de 1 cuento de tiempo, sin requerir ninguna carga de trabajo adicional de cambio de contexto. Sin embargo, si el cuento de tiempo dura 6 unidades, el proceso requerirá 2 cuantos, requiriendo un cambio de contexto. Si el cuento de tiempo es de 1 unidad, entonces se producirán nueve cambios de contexto, ralentizando correspondientemente la ejecución del proceso (Figura 5.4).

Por tanto, conviene que el cuento de tiempo sea grande con respecto al tiempo requerido por un cambio de contexto. Si el tiempo de cambio de contexto es, aproximadamente, el 10 por ciento del cuento de tiempo, entonces un 10 por ciento del tiempo de CPU se invertirá en cambios de contexto. En la práctica, los sistemas más modernos emplean cuantos de tiempo en el rango de 10 a 100 milisegundos y el tiempo requerido para un cambio de contexto es, normalmente, menor que 10 microsegundos; por tanto, el tiempo de cambio de contexto es una fracción pequeña del cuento de tiempo.

El tiempo de ejecución también depende del valor del cuento de tiempo. Como podemos ver en la Figura 5.5, el tiempo medio de ejecución de un conjunto de procesos no necesariamente mejora cuando se incrementa el cuento de tiempo. En general, el tiempo medio de ejecución puede mejorarse si la mayor parte de los procesos termina su siguiente ráfaga de CPU en un solo cuento de tiempo. Por ejemplo, dados tres procesos con una duración, cada uno de ellos, de 10 unidades de tiempo y un cuento igual a 1 unidad de tiempo, el tiempo medio de ejecución será de 29 unidades. Sin embargo, si el cuento de tiempo es 10, el tiempo medio de ejecución cae a 20. Si se añade el tiempo de cambio de contexto, el tiempo medio de ejecución aumenta al disminuir el cuento de tiempo, dado que serán necesarios más cambios de contexto.

Aunque el cuento de tiempo debe ser grande comparado con el tiempo de cambio de contexto, no debe ser demasiado grande. Si el cuento de tiempo es demasiado largo, la planificación por turnos degenera en el método FCFS. Una regla práctica es que el 80 por ciento de las ráfagas de CPU deben ser más cortas que el cuento de tiempo.

5.3.5 Planificación mediante colas multinivel

Otra clase de algoritmos de planificación es la que se ha desarrollado para aquellas situaciones en las que los procesos pueden clasificarse fácilmente en grupos diferentes. Por ejemplo, una clasificación habitual consiste en diferenciar entre procesos **de primer plano** (interactivos) y procesos de **segundo plano** (por lotes). Estos dos tipos de procesos tienen requisitos diferentes de tiempo de respuesta y, por tanto, pueden tener distintas necesidades de planificación. Además, los procesos de primer plano pueden tener prioridad (definida externamente) sobre los procesos de segundo plano.

Un **algoritmo de planificación mediante colas multinivel** divide la cola de procesos preparados en varias colas distintas (Figura 5.6). Los procesos se asignan permanentemente a una cola, generalmente en función de alguna propiedad del proceso, como por ejemplo el tamaño de memoria, la prioridad del proceso o el tipo de proceso. Cada cola tiene su propio algoritmo de planificación. Por ejemplo, pueden emplearse colas distintas para los procesos de primer plano y de

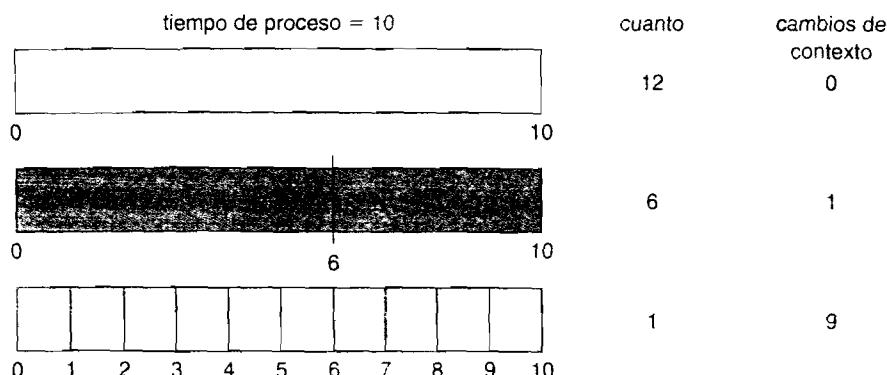


Figura 5.4 La forma en que un cuento de tiempo muy pequeño incrementa los cambios de contexto.

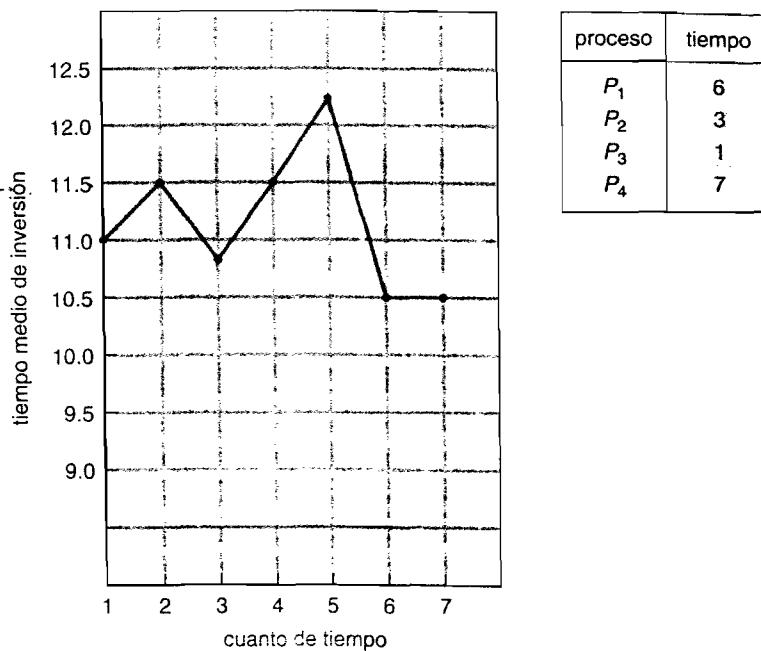


Figura 5.5 La forma en que varía el tiempo de ejecución con el cuento de tiempo.

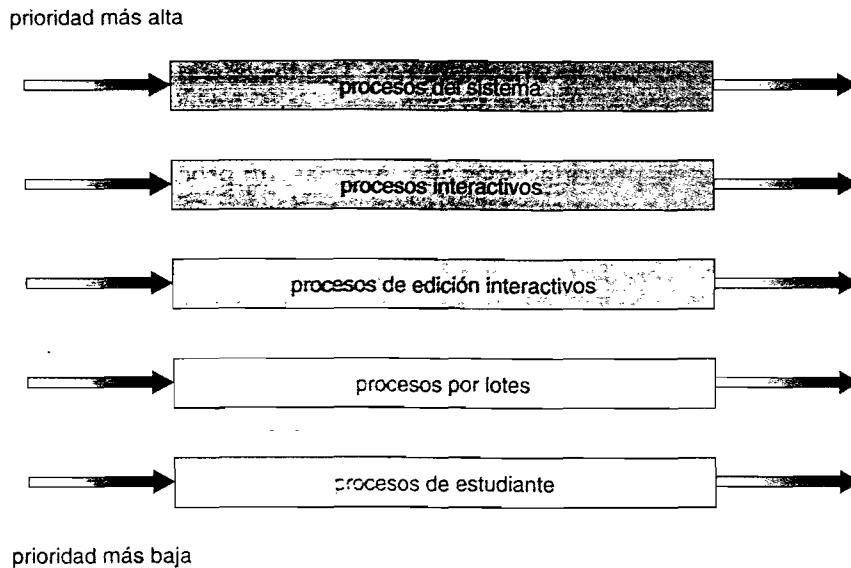


Figura 5.6 Planificación de colas multinivel.

segundo plano. La cola de primer plano puede planificarse mediante un algoritmo por turnos, mientras que para la cola de segundo plano puede emplearse un algoritmo FCFS.

Además, debe definirse una planificación entre las colas, la cual suele implementarse como una planificación apropiativa y prioridad fija. Por ejemplo, la cola de procesos de primer plano puede tener prioridad absoluta sobre la cola de procesos de segundo plano.

Veamos un ejemplo de algoritmo de planificación mediante colas multinivel con las cinco colas que se enumeran a continuación, según su orden de prioridad:

1. Procesos del sistema.
2. Procesos interactivos.

3. Procesos de edición interactivos.
4. Procesos por lotes.
5. Procesos de estudiantes.

Cada cola tiene prioridad absoluta sobre las colas de prioridad más baja. Por ejemplo, ningún proceso de la cola por lotes podrá ejecutarse hasta que se hayan vaciado completamente las colas de los procesos del sistema, los procesos interactivos y los procesos de edición interactivos. Si un proceso de edición interactivo llega a la cola de procesos preparados mientras se está ejecutando un proceso por lotes, el proceso por lotes será desalojado.

Otra posibilidad consiste en repartir el tiempo entre las colas. En este caso, cada cola obtiene una cierta porción del tiempo de CPU, con la que puede entonces planificar sus distintos procesos. Por ejemplo, en el caso de las colas de procesos de primer plano y segundo plano, la cola de primer plano puede disponer del 80 por ciento del tiempo de CPU para planificar por turnos sus procesos, mientras que la cola de procesos de segundo plano recibe el 20 por ciento del tiempo de CPU para gestionar sus procesos mediante el método FCFS.

5.3.6 Planificación mediante colas multinivel realimentadas

Normalmente, cuando se usa el algoritmo de planificación mediante colas multinivel, los procesos se asignan de forma permanente a una cola cuando entran en el sistema. Por ejemplo, si hay colas diferentes para los procesos de primer y segundo plano, los procesos no se mueven de una cola a otra, dado que no pueden cambiar su naturaleza de proceso de primer o segundo plano. Esta configuración presenta la ventaja de una baja carga de trabajo de planificación, pero resulta poco flexible.

Por el contrario, el **algoritmo de planificación mediante colas multinivel realimentadas** permite mover un proceso de una cola a otra. La idea es separar los procesos en función de las características de sus ráfagas de CPU. Si un proceso utiliza demasiado tiempo de CPU, se pasa a una cola de prioridad más baja. Este esquema deja los procesos limitados por E/S y los procesos interactivos en las colas de prioridad más alta. Además, un proceso que esté esperando demasiado tiempo en una cola de baja prioridad puede pasarse a una cola de prioridad más alta. Este mecanismo de envejecimiento evita el bloqueo indefinido.

Por ejemplo, considere un planificador de colas multinivel realimentadas con tres colas, numeradas de 0 a 2 (Figura 5.7). En primer lugar, el planificador ejecuta todos los procesos de la cola 0. Sólo cuando la cola 0 esté vacía ejecutará procesos de la cola 1. De forma similar, los procesos de la cola 2 solo se ejecutarán si las colas 0 y 1 están vacías. Un proceso que llegue a la cola 1 desalojará a un proceso de la cola 2 y ese proceso de la cola 1 será, a su vez, desalojado por un proceso que llegue a la cola 0.

Un proceso que entre en la cola de procesos preparados se coloca en la cola 0 y a cada uno de los procesos de esa cola se le proporciona un cuento de tiempo de 8 milisegundos. Si el proceso no termina en ese tiempo, se pasa al final de la cola 1. Si la cola 0 está vacía, al proceso que se encuentra al principio de la cola 1 se le asigna un cuento de 16 milisegundos. Si no se completa en ese tiempo, se lo desaloja y se lo incluye en la cola 2. Los procesos de la cola 2 se ejecutan basándose en una planificación FCFS, pero sólo cuando las colas 0 y 1 están vacías.

Este algoritmo de planificación proporciona la prioridad más alta a todo proceso que tenga una ráfaga de CPU de 8 milisegundos o menos. Tales procesos acceden rápidamente a la CPU, concluyen su ráfaga de CPU y pasan a su siguiente ráfaga de E/S. Los procesos que necesitan más de 8 milisegundos y menos de 24 milisegundos también son servidos rápidamente, aunque con una prioridad más baja que los procesos más cortos. Los procesos largos terminan yendo automáticamente a la cola 2 y se sirven, siguiendo el orden FCFS, con los ciclos de CPU no utilizados por las colas 0 y 1.

En general, un planificador mediante colas multinivel realimentadas se define mediante los parámetros siguientes:

- El número de colas.

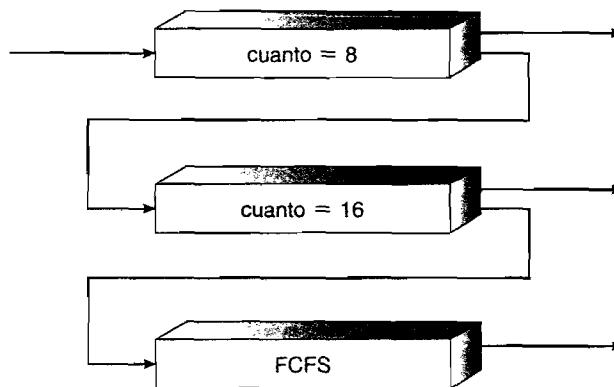


Figura 5.7 Colas multinivel realimentadas.

- El algoritmo de planificación de cada cola.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más alta.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más baja.
- El método usado para determinar en qué cola se introducirá un proceso cuando haya que darle servicio.

La definición del planificador mediante colas multinivel realimentadas le convierte en el algoritmo de planificación de la CPU más general. Puede configurarse este algoritmo para adaptarlo a cualquier sistema específico que se quiera diseñar. Lamentablemente, también es el algoritmo más complejo, puesto que definir el mejor planificador requiere disponer de algún mecanismo para seleccionar los valores de todos los parámetros.

5.4 Planificación de sistemas multiprocesador

Hasta el momento, nuestra exposición se ha centrado en los problemas de planificación de la CPU en un sistema con un solo procesador. Si hay disponibles múltiples CPU, se puede **compartir la carga**; sin embargo, el problema de la planificación se hace más complejo. Se han probado muchas posibilidades; y como ya hemos visto para el caso de la planificación de la CPU con un único procesador, no existe una solución única. Aquí vamos a presentar diversas cuestiones acerca de la planificación multiprocesador. Vamos a concentrarnos en los sistemas en los que los procesadores son idénticos, **homogéneos** en cuanto a su funcionalidad; en este tipo de sistemas, podemos usar cualquiera de los procesadores disponibles para ejecutar cualquier proceso de la cola. (Sin embargo, observe que incluso con múltiples procesadores homogéneos, en ocasiones hay limitaciones que afectan a la planificación; considere un sistema con un dispositivo de E/S conectado a un bus privado de un procesador. Los procesos que deseen emplear ese dispositivo deberán planificarse para ser ejecutados en dicho procesador.)

5.4.1 Métodos de planificación en los sistemas multiprocesador

Un método para planificar las CPU en un sistema multiprocesador consiste en que todas las decisiones sobre la planificación, el procesamiento de E/S y otras actividades del sistema sean gestionadas por un mismo procesador, el servidor maestro. Los demás procesadores sólo ejecutan código de usuario. Este **multiprocesamiento asimétrico** resulta simple, porque sólo hay un procesador que accede a las estructuras de datos del sistema, reduciendo la necesidad de compartir datos.

Un segundo método utiliza el **multiprocesamiento simétrico (SMP)**, en el que cada uno de los procesadores se auto-planifica. Todos los procesos pueden estar en una cola común de procesos

preparados, o cada procesador puede tener su propia cola privada de procesos preparados. Independientemente de esto, la planificación se lleva a cabo haciendo que el planificador de cada procesador examine la cola de procesos preparados y seleccione un proceso para ejecutarlo. Como veremos en el Capítulo 6, si tenemos varios procesadores intentando acceder a una estructura de datos común para actualizarla, el planificador tiene que programarse cuidadosamente: tenemos que asegurar que dos procesadores no elegirán el mismo proceso y que no se perderán procesos de la cola. Prácticamente todos los sistemas operativos modernos soportan el multiprocesamiento simétrico, incluyendo Windows XP, Windows 2000, Solaris, Linux y Mac OS X.

En el resto de esta sección, analizaremos diversas cuestiones relacionadas con los sistemas SMP.

5.4.2 Afinidad al procesador

Considere lo que ocurre con la memoria caché cuando un proceso se ha estado ejecutando en un procesador específico: los datos a los que el proceso ha accedido más recientemente se almacenan en la caché del procesador y, como resultado, los sucesivos accesos a memoria por parte del proceso suelen satisfacerse sin más que consultar la memoria caché. Ahora considere lo que ocurre si el proceso migra a otro procesador: los contenidos de la memoria caché del procesador de origen deben invalidarse y la caché del procesador de destino debe ser rellenada. Debido al alto coste de invalidar y llenar las memorias caché, la mayoría de los sistemas SMP intentan evitar la migración de procesos de un procesador a otro, y en su lugar intentan mantener en ejecución cada proceso en el mismo procesador. Esto se conoce con el nombre de **afinidad al procesador**, lo que significa que un proceso tiene una afinidad hacia el procesador en que está ejecutándose actualmente.

La afinidad al procesador toma varias formas. Cuando un sistema operativo tiene la política de intentar mantener un proceso en ejecución en el mismo procesador, pero no está garantizado que lo haga, nos encontramos ante una situación conocida como **afinidad suave**. En este caso, es posible que un proceso migre entre procesadores. Algunos sistemas, como Linux, también proporcionan llamadas al sistema que soportan la **afinidad dura**, la cual permite a un proceso especificar que no debe migrar a otros procesadores.

5.4.3 Equilibrado de carga

En los sistemas SMP, es importante mantener la carga de trabajo equilibrada entre todos los procesadores, para aprovechar por completo las ventajas de disponer de más de un procesador. Si no se hiciera así, podría darse el caso de que uno o más procesadores permanecieran inactivos mientras otros procesadores tuvieran cargas de trabajo altas y una lista de procesos esperando a acceder a la CPU. Los mecanismos de **equilibrado de carga** intentan distribuir equitativamente la carga de trabajo entre todos los procesadores del sistema SMP. Es importante observar que el equilibrado de carga, normalmente, sólo es necesario en aquellos sistemas en los que cada procesador tiene su propia cola privada de procesos preparados para ejecutarse. En los sistemas con una cola de ejecución común, el equilibrado de carga es a menudo innecesario, ya que una vez que un procesador pasa a estar inactivo, inmediatamente extrae un proceso ejecutable de la cola de ejecución común. No obstante, también es importante destacar que en la mayoría de los sistemas operativos actuales que soportan SMP, cada procesador tiene una cola privada de procesos preparados.

Existen dos métodos generales para equilibrar la carga: **migración comandada** (push migration) y **migración solicitada** (pull migration). Con la migración comandada, una tarea específica comprueba periódicamente la carga en cada procesador y, si encuentra un desequilibrio, distribuye equitativamente la carga moviendo (o cargando) procesos de los procesadores sobrecargados en los menos ocupados o inactivos. La migración solicitada se produce cuando un procesador inactivo extrae de un procesador ocupado alguna tarea que esté en espera. Las migraciones comandadas y solicitadas no tienen por qué ser mutuamente excluyentes y, de hecho, a menudo se implementan en paralelo en los sistemas de equilibrado de carga. Por ejemplo, el planificador de Linux (descrito en la Sección 5.6.3) y el planificador ULE disponible en los sistemas FreeBSD implementan ambas técnicas. Linux ejecuta sus algoritmos de equilibrado de carga cada 200 milisegundos.

segundos (migración comandada) o cuando la cola de ejecución de un procesador está vacía (migración solicitada).

Es interesante comentar que el equilibrado de carga a menudo contrarresta los beneficios de la afinidad al procesador, vista en la Sección 5.4.2. Es decir, la ventaja de mantener un proceso ejecutándose en el mismo procesador es que el proceso se aprovecha de que sus datos se encuentran en la memoria caché de dicho procesador. Al migrar procesos de un procesador a otro, anulamos esta ventaja. Como es habitual en la ingeniería de sistemas, no existe una regla absoluta para determinar cuál es la mejor política; por tanto, en algunos sistemas, un procesador inactivo siempre extrae un proceso de un procesador que no esté inactivo mientras que, en otros sistemas, los procesos migran sólo si el desequilibrio excede determinado umbral.

5.4.4 Mecanismos multihebra simétricos

Los sistemas SMP permiten que varias hebras se ejecuten de forma concurrente, ya que proporcionan varios procesadores físicos. Una estrategia alternativa consiste en proporcionar varios procesadores *lógicos*, en lugar de *físicos*. Esta estrategia se conoce con el nombre de mecanismo multihebra simétrico (SMT, symmetric multithreading), aunque también se denomina **tecnología hiperhebra** (hyperthreading) en los procesadores Intel.

La idea que subyace al mecanismo SMT es la de crear varios procesadores lógicos sobre un mismo procesador físico, presentando una vista de varios procesadores lógicos al sistema operativo, incluso en los sistemas con un solo procesador físico. Cada procesador lógico tiene su propio **estado de la arquitectura**, que incluye los registros de propósito general y los registros de estado de la máquina. Además, cada procesador lógico es responsable de su propio tratamiento de interrupciones, lo que significa que las interrupciones son proporcionadas, y gestionadas, por los procesadores lógicos en lugar de por los físicos. Por lo demás, cada procesador lógico comparte los recursos de su procesador físico, como la memoria caché y los buses. La Figura 5.8 ilustra una arquitectura SMT típica con dos procesadores físicos, albergando cada uno dos procesadores lógicos. Desde la perspectiva del sistema operativo, en este sistema hay disponibles cuatro procesadores para realizar el trabajo.

Es importante recalcar que SMT es una funcionalidad proporcionada por hardware y no por software. Es decir, el hardware tiene que proporcionar la representación del estado de la arquitectura de cada procesador lógico, así como el tratamiento de interrupciones. Los sistemas operativos no tienen necesariamente que diseñarse de forma diferente para ejecutarse en un sistema SMT; sin embargo, se pueden obtener ciertas ventajas si el sistema operativo es consciente de que se está ejecutando en un sistema de este tipo. Por ejemplo, considere un sistema con dos procesadores físicos, ambos en estado de inactividad. En primer lugar, el planificador debe intentar planificar hebras distintas en cada procesador físico en lugar de en procesadores lógicos distintos del mismo procesador físico; en caso contrario, ambos procesadores lógicos de un procesador físico podrían estar ocupados mientras que el otro procesador físico permanecería inactivo.

5.5 Planificación de hebras

En el Capítulo 4, hemos presentado el papel de las hebras en el modelo de procesos, diferenciando entre *hebras de nivel de usuario* y de *nivel de kernel*. En los sistemas operativos que permiten su

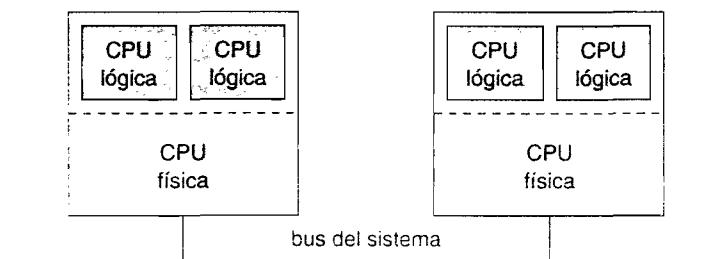


Figura 5.8 Una arquitectura SMT típica.

uso, el sistema operativo planifica hebras del nivel del *kernel*, no procesos. Las hebras del nivel de usuario son gestionadas por una biblioteca de hebras, y el *kernel* no es consciente de ellas. Para ejecutarse en una CPU, las hebras de usuario deben ser asignadas a una hebra de nivel de *kernel* asociada, aunque esta asignación puede ser indirecta y puede emplear un proceso ligero (LWP). En esta sección, vamos a explorar las cuestiones de planificación relativas a las hebras de nivel de usuario y de nivel del *kernel* y ofreceremos ejemplos concretos de planificación en Pthreads.

5.5.1 Ámbito de contienda

Una de las diferencias entre las hebras de nivel de usuario y de nivel del *kernel* radica en la forma en que se planifican unas y otras. En los sistemas que implementan los modelos muchos-a-uno (Sección 4.2.1) y muchos-a-muchos (Sección 4.2.3), la biblioteca de hebras planifica las hebras de nivel de usuario para que se ejecuten sobre un proceso LWP disponible, lo cual es un esquema conocido con el nombre de **ámbito de contienda del proceso** (PCS, process-contention scope), dado que la competición por la CPU tiene lugar entre hebras que pertenecen al mismo proceso. Cuando decimos que la biblioteca de hebras *planifica* las hebras de usuario sobre los procesos ligeros disponibles, no queremos decir que la hebra se ejecute realmente en una CPU; esto requeriría que el sistema operativo planificara la hebra del *kernel* en una CPU física. Para decidir qué hebra del *kernel* planificar en una CPU, el *kernel* usa el **ámbito de contienda del sistema** (SCS); la competición por la CPU con la planificación SCS tiene lugar entre todas las hebras del sistema. Los sistemas que usan el modelo uno-a-uno (tal como Windows XP, Solaris 9 y Linux) planifican las hebras sólo con SCS.

Normalmente, la planificación PCS se lleva a cabo de acuerdo con la prioridad: el planificador selecciona para su ejecución la hebra ejecutable con la prioridad más alta. Las prioridades de las hebras de nivel de usuario son establecidas por el programador y no se ajustan mediante la biblioteca de hebras, aunque algunas bibliotecas de hebras permiten que el programador cambie la prioridad de una hebra. Es importante destacar que PCS normalmente desalojará la hebra actualmente en ejecución en favor de una hebra de prioridad más alta; sin embargo, no se garantiza un reparto equitativo de cuantos de tiempo (Sección 5.3.4) entre las hebras de igual prioridad.

5.5.2 Planificación en Pthread

En la Sección 4.3.1 se ha proporcionando un programa de ejemplo de Pthread en POSIX, junto con una introducción a la creación de hebras con Pthread. Ahora, vamos a analizar la API de Pthread de POSIX que permite especificar el ámbito de contienda del proceso (PCS) o el ámbito de contienda del sistema (SCS) durante la creación de hebras. Pthreads utiliza los siguientes valores para definir el ámbito de contienda:

- PTHREAD_SCOPE_PROCESS planifica las hebras usando la planificación PCS.
- PTHREAD_SCOPE_SYSTEM planifica las hebras usando la planificación SCS.

En los sistemas que implementan el modelo muchos-a-muchos (Sección 4.2.3), la política PTHREAD_SCOPE_PROCESS planifica las hebras de nivel usuario sobre los procesos ligeros disponibles. El número de procesos LWP se mantiene mediante la biblioteca de hebras, quizás utilizando activaciones del planificador (Sección 4.4.6). La política de planificación PTHREAD_SCOPE_SYSTEM creará y asociará un proceso LWP a cada hebra de nivel de usuario en los sistemas muchos-a-muchos, lo que equivale en la práctica a asignar las hebras usando el modelo uno-a-uno (Sección 4.2.2).

La API de Pthread proporciona las dos funciones siguientes para consultar y definir la política de ámbito de contienda:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`

El primer parámetro para ambas funciones contiene un puntero al conjunto de atributos de la hebra. En el segundo parámetro de `pthread_attr_setscope` se pasa el valor PTHREAD_

SCOPE_SYSTEM o PTHREAD_SCOPE_PROCESS, lo que indica cómo se define el ámbito de contienda. En el caso de `pthread_attr_getscope()`, este segundo parámetro contiene un puntero a un valor entero (`int`) al que se asignará el valor actual del ámbito de la contienda. Si se produce un error, cada una de estas funciones devuelve valores distintos de cero.

En la Figura 5.9 se presenta un programa Pthread que primero determina el ámbito de contienda existente y luego lo define como PTHREAD_SCOPE_PROCESS. A continuación crea cinco hebras independientes que se ejecutarán usando la política de planificación SCS. Observe que algunos sistemas sólo permiten determinados valores para el ámbito de contienda. Por ejemplo, los sistemas Linux y Mac OS X sólo permiten PTHREAD_SCOPE_SYSTEM.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* obtener los atributos predeterminados */
    pthread_attr_init(&attr);

    /* primero consultar el ámbito actual */
    if (pthread_attr_getscope(&attr, &scope) !=0)
        fprintf(stderr, "Imposible obtener ámbito de planificación\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Valor de ámbito ilegal.\n");
    }

    /* definir el algoritmo de planificación como PCS o SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* crear las hebras */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* ahora esperar a que termine cada hebra */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Cada hebra iniciará su ejecución en esta función */
void *runner(void *param)
{
    /* realizar alguna tarea ... */

    pthread_exit(0);
}
```

Figura 5.9 API de planificación de Pthread.

5.6 Ejemplos de sistemas operativos

A continuación vamos a describir las políticas de planificación de los sistemas operativos Solaris, Windows XP y Linux. Es importante recordar que lo que vamos a describir en el caso de Solaris y Linux es la planificación de las hebras del *kernel*. Recuerde que Linux no diferencia entre procesos y hebras; por tanto, utilizamos el término *tarea* al hablar del planificador de Linux.

5.6.1 Ejemplo: planificación en Solaris

Solaris usa una planificación de hebras basada en prioridades, definiendo cuatro clases para planificación. Estas clases son, por orden de prioridad:

1. Tiempo real
2. Sistema
3. Tiempo compartido
4. Interactiva

Dentro de cada clase hay diferentes prioridades y diferentes algoritmos de planificación. Los mecanismos de planificación en Solaris se ilustran en la Figura 5.10.

La clase de planificación predeterminada para un proceso es la de tiempo compartido. La política de planificación para tiempo compartido modifica dinámicamente las prioridades y asigna cuantos de tiempo de diferente duración usando colas multinivel realimentadas. De manera predeterminada, existe una relación inversa entre las prioridades y los cuantos de tiempo. Cuanto más alta sea la prioridad, más pequeño será el cuanto de tiempo; y cuanto menor sea la prioridad,

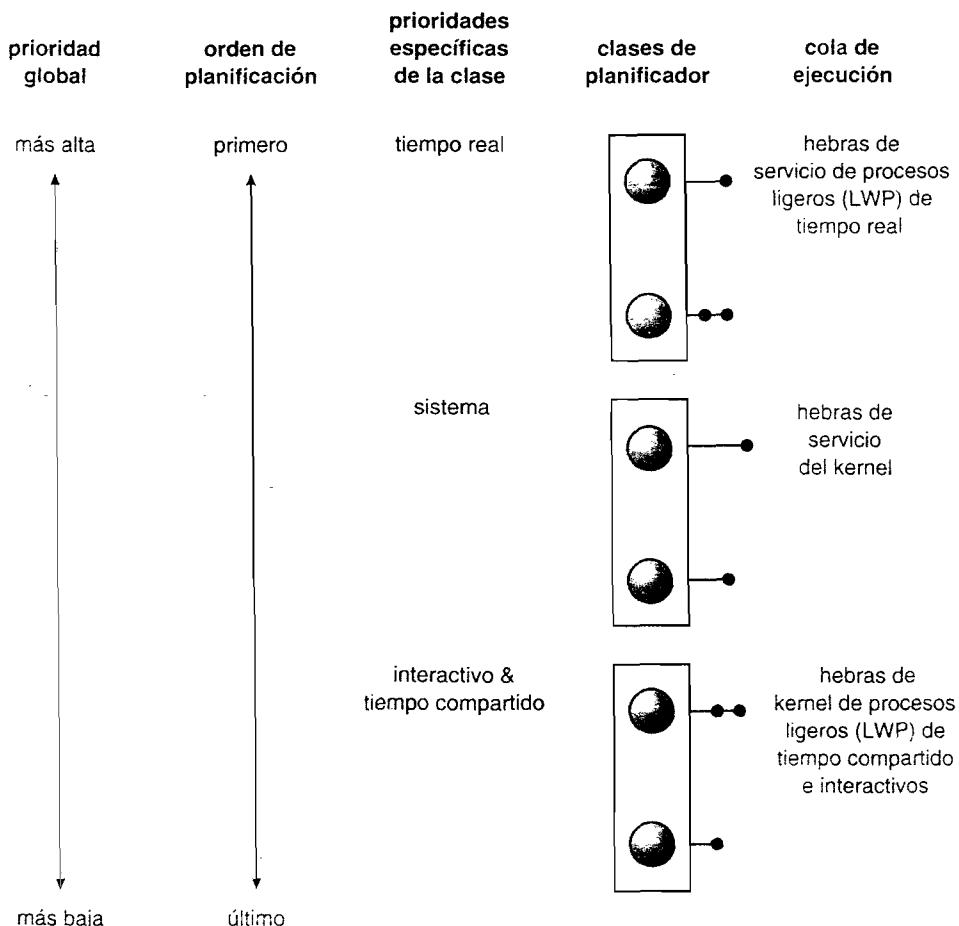


Figura 5.10 Planificación en Solaris.

más larga será la franja. Los procesos interactivos suelen tener la prioridad más alta; los procesos limitados por la CPU tienen la prioridad más baja. Esta política de planificación proporciona un buen tiempo de respuesta para los procesos interactivos y una buena tasa de procesamiento para los procesos limitados por la CPU. La clase interactiva usa la misma política de planificación que la clase de tiempo compartido, pero proporciona a las aplicaciones con interfaz de ventanas una prioridad más alta, para aumentar el rendimiento.

La Figura 5.11 muestra la tabla de despacho para la planificación de hebras interactivas y de tiempo compartido. Estas dos clases de planificación incluyen 60 niveles de prioridad pero, para abreviar, solo se muestran unos cuantos. La tabla de despacho mostrada en la Figura 5.11 contiene los siguientes campos:

- **Prioridad.** La prioridad, dependiente de la clase, para las clases de tiempo compartido e interactiva. Un número alto indica una mayor prioridad.
- **Cuento de tiempo.** El cuento de tiempo para la prioridad asociada. Ilustra la relación inversa entre prioridades y cuantos de tiempo: la prioridad más baja (prioridad 0) tiene el cuento de tiempo más largo (200 milisegundos), mientras que la prioridad más alta (prioridad 59) tiene el cuento de tiempo más bajo (20 milisegundos).
- **Caducidad del cuento de tiempo.** La nueva prioridad que se asignará a una hebra que haya consumido su cuento de tiempo completo sin bloquearse. Tales hebras se considera que hacen un uso intensivo de la CPU. Como se muestra en la tabla, la prioridad de estas hebras se reduce.
- **Retorno del estado dormido.** La prioridad que se asigna a una hebra cuando sale del estado dormido (como, por ejemplo, cuando la hebra estaba esperando para realizar una operación de E/S). Como ilustra la tabla, cuando el dispositivo de E/S está disponible para una hebra en espera, la prioridad de ésta se aumenta a entre 50 y 59, con el fin de soportar la política de planificación consistente en proporcionar un buen tiempo de respuesta para los procesos interactivos.

Solaris 9 introduce dos nuevas clases de planificación: **de prioridad fija** y **de cuota equitativa**. Las hebras de prioridad fija tienen el mismo rango de prioridades que las de tiempo compartido;

prioridad	cuento de tiempo	cuanto de tiempo caducado	retorno del estado durmiente
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Figura 5.11 Tabla de despacho de Solaris para hebras interactivas y de tiempo compartido.

sin embargo, sus prioridades no se ajustan dinámicamente. La clase de cuota equitativa usa cuotas de CPU en lugar de prioridades a la hora de tomar decisiones de planificación. Las cuotas de CPU indican el derecho a utilizar los recursos de CPU disponibles y se asignan a un conjunto de procesos (a cada uno de esos conjuntos se le denomina **proyecto**).

Solaris usa la clase sistema para ejecutar procesos del *kernel*, como el planificador y el demonio de paginación. Una vez establecida, la prioridad de un proceso del sistema no cambia. La clase sistema se reserva para uso del *kernel* (los procesos de usuario que se ejecutan en modo *kernel* no se incluyen en la clase sistema).

A las hebras pertenecientes a la clase de tiempo real se les asigna la prioridad más alta. Esta asignación permite que un proceso en tiempo real tenga una respuesta asegurada del sistema dentro de un período limitado de tiempo. Un proceso en tiempo real se ejecutará antes que los procesos de cualquier otra clase. Sin embargo, en general, son pocos los procesos pertenecientes a la clase de tiempo real.

Cada clase de planificación incluye un conjunto de prioridades. Sin embargo, el planificador convierte las prioridades específicas de una clase en prioridades globales y selecciona la hebra que tenga la prioridad global más alta para ejecutarla. La hebra seleccionada se ejecuta en la CPU hasta que (1) se bloquea, (2) consume su cuanto de tiempo o (3) es desalojada por una hebra de prioridad más alta. Si existen múltiples hebras con la misma prioridad, el planificador usa una cola y selecciona las hebras por turnos. Como ya hemos dicho anteriormente, Solaris ha usado tradicionalmente el modelo muchos-a-muchos (Sección 4.2.3), pero con Solaris 9 se cambió al modelo uno-a-uno (Sección 4.2.2).

5.6.2 Ejemplo: planificación en Windows XP

Windows XP planifica las hebras utilizando un algoritmo de planificación apropiativo basado en prioridades. El planificador de Windows XP asegura que siempre se ejecute la hebra de prioridad más alta. La parte del *kernel* de Windows XP que gestiona la planificación se denomina *despachador*. Una hebra seleccionada para ejecutarse por el despachador se ejecutará hasta que sea desalojada por una hebra de prioridad más alta, hasta que termine, hasta que su cuanto de tiempo concluya o hasta que invoque una llamada bloqueante al sistema, como por ejemplo para una operación de E/S. Si una hebra en tiempo real de prioridad más alta pasa al estado preparado mientras se está ejecutando una hebra de prioridad más baja, esta última será desalojada. Este desalojo proporciona a la hebra de tiempo real un acceso preferencial a la CPU en el momento en que la hebra necesite dicho acceso.

El despachador usa un esquema de prioridades de 32 niveles para determinar el orden de ejecución de las hebras. Las prioridades se dividen en dos clases: la clase **variable** contiene hebras cuyas prioridades van de 1 a 15, mientras que la clase **de tiempo real** contiene hebras con prioridades comprendidas en el rango de 16 a 31. Existe también una hebra que se ejecuta con prioridad 0 y que se emplea para la gestión de memoria. El despachador usa una cola distinta para cada prioridad de planificación y recorre el conjunto de colas desde la más alta a la más baja, hasta que encuentra una hebra que esté preparada para ejecutarse. Si no encuentra una hebra preparada, el despachador ejecuta una hebra especial denominada **hebra inactiva**.

Existe una relación entre las prioridades numéricas del *kernel* de Windows XP y la API Win32. La API Win32 identifica varias clases de prioridades a las que un proceso puede pertenecer. Entre ellas se incluyen:

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- IDLE_PRIORITY_CLASS

Las prioridades de todas las clases, excepto REALTIME_PRIORITY_CLASS, son del tipo variable, lo que significa que la prioridad de una hebra que pertenezca a una de esas clases puede cambiar.

Dentro de cada clase de prioridad hay una prioridad relativa. Los valores para la prioridad relativa son:

- TIME_CRITICAL
- HIGHEST
- ABOVE_NORMAL
- NORMAL
- BELOW_NORMAL
- LOWEST
- IDLE

La prioridad de cada hebra se basa en la clase de prioridad a la que pertenece y en su prioridad relativa dentro de dicha clase. Esta relación se muestra en la Figura 5.12. Los valores de las clases de prioridad se muestran en la fila superior; la columna de la izquierda contiene los valores de las prioridades relativas. Por ejemplo, si la prioridad relativa de una hebra de la clase ABOVE_NORMAL_PRIORITY_CLASS es NORMAL, la prioridad numérica de dicha hebra será 10.

Además, cada hebra tiene una prioridad base que representa un valor dentro del rango de prioridades de la clase a la que pertenece la hebra. De manera predeterminada, la prioridad base es el valor de la prioridad relativa NORMAL para la clase especificada. Las prioridades base para cada clase de prioridad son:

- REALTIME_PRIORITY_CLASS-24
- HIGH_PRIORITY_CLASS-13
- ABOVE_NORMAL_PRIORITY_CLASS-10
- NORMAL_PRIORITY_CLASS-8
- BELOW_NORMAL_PRIORITY_CLASS-6
- IDLE_PRIORITY_CLASS-4

Normalmente, los procesos son miembros de la clase NORMAL_PRIORITY_CLASS. Un proceso pertenecerá a esta clase a menos que el padre del proceso pertenezca a la clase IDLE_PRIORITY_CLASS o que se haya especificado otra clase en el momento de crear el proceso. La prioridad inicial de una hebra es normalmente la prioridad base del proceso al que pertenece la hebra.

Cuando se excede el cuento de tiempo de una hebra, dicha hebra se interrumpe; si la hebra pertenece a la clase de prioridad variable, se reduce su prioridad. No obstante, la prioridad nunca dis-

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Figura 5.12 Prioridades en Windows XP.

minuye por debajo de la prioridad base. Disminuir la prioridad de la hebra tiende a limitar el consumo de CPU por parte de las hebras que realicen cálculos intensivos. Cuando una hebra de prioridad variable sale de un estado de espera, el despachador incrementa su prioridad. Dicho incremento dependerá de lo que la hebra estuviera esperando; por ejemplo, la prioridad de una hebra que estuviera esperando por una operación de E/S de teclado se aumentará de forma significativa, mientras que la de una hebra que estuviera esperando por una operación de disco se incrementará de forma moderada. Esta estrategia suele proporcionar buenos tiempos de respuesta a las hebras interactivas que utilicen el ratón y una interfaz de ventanas. También permite a las hebras limitadas por E/S mantener ocupados a los dispositivos de E/S, al mismo tiempo que las hebras que realizan cálculos intensivos emplean en segundo plano los ciclos de CPU libres. Esta estrategia se usa en varios sistemas operativos de tiempo compartido, incluyendo UNIX. Además, se aumenta la prioridad de la ventana con la que esté interactuando actualmente el usuario, para mejorar el tiempo de respuesta.

Cuando un usuario está ejecutando un programa interactivo, el sistema necesita proporcionar un rendimiento especialmente bueno a dicho proceso. Por esta razón, Windows XP tiene una regla de planificación especial para los procesos de la clase NORMAL_PRIORITY_CLASS. Windows XP diferencia entre el *proceso de primer plano* que está actualmente seleccionado en la pantalla y los *procesos de segundo plano* que no están actualmente seleccionados. Cuando un proceso pasa a primer plano, Windows XP multiplica el cuento de planificación por un cierto factor, normalmente igual a 3. Este incremento proporciona al proceso en primer plano tres veces más tiempo para ejecutarse, antes de que se produzca un desalojo debido a la compartición de tiempo.

5.6.3 Ejemplo: planificación en Linux

Antes de la versión 2.5, el *kernel* de Linux ejecutaba una variante del algoritmo tradicional de planificación de UNIX. Los dos problemas que tiene el planificador tradicional de UNIX son, por un lado, que no proporciona el adecuado soporte para sistemas SMP y por otro que no puede escalarse bien al aumentar el número de tareas en el sistema. Con la versión 2.5, se optimizó el planificador y ahora el *kernel* proporciona un algoritmo de planificación que se ejecuta a velocidad constante [con tasa de proporcionalidad $O(1)$], independientemente del número de tareas del sistema. El nuevo planificador también proporciona un mejor soporte para sistemas SMP, incluyendo mecanismos de afinidad al procesador y equilibrado de carga, así como un reparto equitativo de recursos y soporte para tareas interactivas.

El planificador de Linux es un algoritmo basado en prioridades y apropiativo, con dos rangos distintos de prioridades: un rango de **tiempo** real de 0 a 99 y un valor **normal** (nice) en el rango comprendido entre 100 y 140. Estos dos rangos se asignan a un esquema de prioridades global, en el que los valores numéricamente más bajos indican las prioridades más altas.

A diferencia de otros muchos sistemas, incluyendo Solaris (Sección 5.6.1) y Windows XP (Sección 5.6.2), Linux asigna a las tareas de prioridad más alta cuantos de tiempo más largos y a las tareas de prioridad más baja cuantos de tiempo más cortos. La relación entre la prioridad y la duración del cuento de tiempo se muestra en la Figura 5.13.

Una tarea ejecutable se considera elegible para ejecutarse en la CPU cuando todavía le quede tiempo de su cuento de tiempo. Cuando una tarea ha agotado su cuento de tiempo, se considera caducada y no puede volver a ejecutarse hasta que las otras tareas hayan agotado sus respectivos cuantos de tiempo. El *kernel* mantiene una lista de todas las tareas ejecutables en una estructura de datos denominada **cola de ejecución** (*runqueue*). Debido al soporte para sistemas SMP, cada procesador mantiene su propia cola de ejecución y la planifica de forma independiente. Cada cola de ejecución contiene dos matrices de prioridades, denominadas matrices **activa** y **caducada**. La matriz activa contiene todas las tareas que todavía disponen de tiempo en su cuento de tiempo, mientras que la matriz caducada contiene todas las tareas caducadas. Cada una de estas matrices de prioridades contiene una lista de tareas indexada en función de la prioridad (Figura 5.14). El planificador elige la tarea de la matriz activa con la prioridad más alta para su ejecución en la CPU. En las máquinas multiprocesador, esto significa que cada procesador selecciona la tarea de prioridad más alta de su propia cola de ejecución. Cuando todas las tareas han agotado sus cuantos

prioridad numérica	prioridad relativa	cuanto de tiempo
0	más alta	200 ms
•		
•		
•		
99		
100		
•		
•		
140	más baja	10 ms

Figura 5.13 Relación entre la prioridad y la duración del cuanto de tiempo.

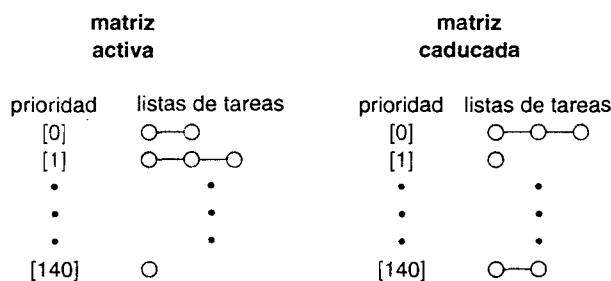


Figura 5.14 Lista de tareas indexadas en función de la prioridad.

de tiempo (es decir, cuando la matriz activa está vacía), las dos matrices de prioridades se intercambian: la matriz caducada pasa a ser la matriz activa, y viceversa.

Linux implementa la planificación en tiempo real tal como se define en POSIX.1b, lo cual se corresponde con el mecanismo descrito en la Sección 5.5.2. A las tareas en tiempo real se les asignan prioridades estáticas; las restantes tareas tienen prioridades dinámicas que se basan en sus valores *nice*, más o menos 5. La interactividad de una tarea determina si el valor 5 tiene que sumarse o restarse del valor *nice*. El grado de interactividad de una tarea se determina por el tiempo que ha estado durmiendo mientras esperaba para realizar una operación de E/S. Las tareas más interactivas suelen permanecer más tiempo en el estado dormido y, por tanto, lo más probable es que usen ajustes próximos a -5, ya que el planificador favorece las tareas interactivas. Inversamente, las tareas que pasan menos tiempo en estado durmiente suelen estar limitadas por la CPU y, por tanto, se les asignará una prioridad menor.

El recálculo de la prioridad dinámica de una tarea se produce cuando la tarea ha agotado su cuanto de tiempo y se pasa a la matriz de caducadas. Por tanto, cuando se intercambian las dos matrices, a todas las tareas que hay en la nueva matriz activa se les habrán asignado nuevas prioridades y los correspondientes cuantos de tiempo.

5.7 Evaluación de algoritmos

¿Cómo seleccionar un algoritmo de planificación de la CPU para un sistema en particular? Como hemos visto en la Sección 5.3, existen muchos algoritmos de planificación distintos, cada uno con sus propios parámetros; por tanto, seleccionar un algoritmo puede resultar complicado.

El primer problema consiste en definir los criterios que se van a emplear para seleccionar un algoritmo. Como hemos visto en la Sección 5.2, los criterios se definen a menudo en términos de utilización de la CPU, del tiempo de respuesta o de la tasa de procesamiento; para seleccionar un algoritmo, primero tenemos que definir la importancia relativa de estas medidas. Nuestros criterios pueden incluir varias medidas distintas, como por ejemplo:

- Maximizar la utilización de la CPU bajo la restricción de que el tiempo máximo de respuesta sea igual a 1 segundo.
- Maximizar la tasa de procesamiento de modo que el tiempo de ejecución sea (como promedio) linealmente proporcional al tiempo total de ejecución.

Una vez definidos los criterios de selección, podemos evaluar los algoritmos que estemos considerando. A continuación se describen los distintos métodos de evaluación que podemos utilizar.

5.7.1 Modelado determinista

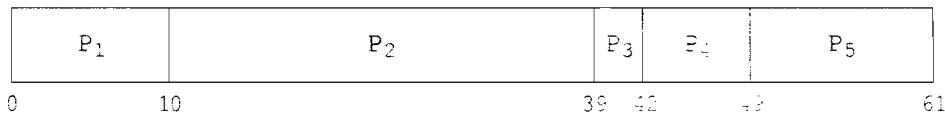
Un método importante de evaluación es la **evaluación analítica**. Este tipo de evaluación utiliza el algoritmo especificado y la carga de trabajo del sistema para generar una fórmula o número que evalúe el rendimiento del algoritmo para dicha carga de trabajo.

Uno de los tipos de evaluación analítica es el **modelado determinista**. Este método toma una carga de trabajo predeterminada concreta y define el rendimiento de cada algoritmo para dicha carga de trabajo. Por ejemplo, suponga que tenemos la carga de trabajo mostrada a continuación. Los cinco procesos llegan en el instante 0 en el orden indicado, con la duración de las ráfagas de CPU especificada en milisegundos:

Proceso	Tiempo de ráfaga
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

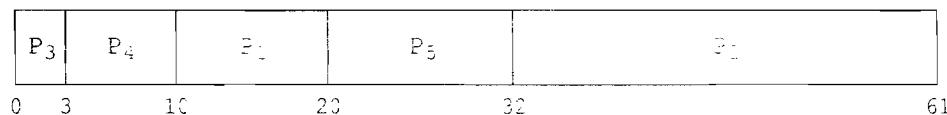
Considere los algoritmos de planificación FCFS, SJF y por turnos (cuanto = 10 milisegundos) para este conjunto de procesos. ¿Qué algoritmo proporcionará el tiempo medio de espera mínimo?

Para el algoritmo FCFS, ejecutariamos los procesos del siguiente modo:



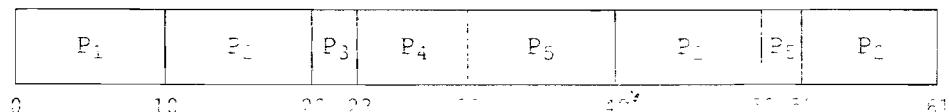
El tiempo de espera es de 0 milisegundos para P_1 , 10 milisegundos para P_2 , 39 milisegundos para P_3 , 42 milisegundos para P_4 y 49 milisegundos para el proceso P_5 . Por tanto, el tiempo medio de espera es de $(0 + 10 + 39 + 42 + 49)/5 = 28$ milisegundos.

Con la planificación SJF cooperativa, ejecutamos los procesos del modo siguiente:



El tiempo de espera es de 10 milisegundos para el proceso P_1 , 32 milisegundos para el proceso P_2 , 0 milisegundos para el proceso P_3 , 3 milisegundos para el proceso P_4 y 20 milisegundos para el proceso P_5 . Por tanto, el tiempo medio de espera es de $(10 + 32 + 0 + 3 + 20)/5 = 13$ milisegundos.

Con el algoritmo de planificación por turnos, ejecutamos los procesos como sigue:



El tiempo de espera es de 0 milisegundos para el proceso P_1 , 32 milisegundos para el proceso P_2 , 20 milisegundos para el proceso P_3 , 23 milisegundos para el proceso P_4 y 40 milisegundos para el proceso P_5 . Por tanto, el tiempo medio de espera es de $(0 + 32 + 20 + 23 + 40)/5 = 23$ milisegundos.

Vemos que, *en este caso*, el tiempo medio de espera obtenido con el algoritmo SJF es menos que la mitad del obtenido con el algoritmo de planificación FCFS; el algoritmo de planificación por turnos nos proporciona un valor intermedio.

El modelado determinista es simple y rápido. Nos proporciona números exactos, permitiendo así comparar los algoritmos. Sin embargo, requiere que se proporcionen números exactos como entrada y sus respuestas sólo se aplican a dichos casos. El modelado determinista se utiliza principalmente para la descripción de los algoritmos de planificación y para proporcionar los correspondientes ejemplos. En los casos en que estemos ejecutando el mismo programa una y otra vez y podamos medir los requisitos de procesamiento de forma exacta, podemos usar el modelado determinista para seleccionar un algoritmo de planificación. Además, utilizando un conjunto de ejemplos, el modelado determinista puede indicar una serie de tendencias, que pueden ser analizadas y demostradas por separado. Por ejemplo, podemos demostrar que, para el entorno descrito (en el que todos los procesos y sus tiempos están disponibles en el instante 0), la política SJF siempre dará como resultado el tiempo de espera mínimo.

5.7.2 Modelos de colas

En muchos sistemas, los procesos que se ejecutan varían de un día a otro, por lo que no existe ningún conjunto estático de procesos (o tiempos) que se pueda emplear en el modelado determinista. Sin embargo, lo que sí es posible determinar es la distribución de las ráfagas de CPU y de E/S. Estas distribuciones pueden medirse y luego aproximarse, o simplemente estimarse. El resultado es una fórmula matemática que describe la probabilidad de aparición de una ráfaga de CPU concreta. Habitualmente, esta distribución es exponencial y se describe mediante su media. De forma similar, podemos describir la distribución de los tiempos de llegada de los procesos al sistema. A partir de estas dos distribuciones, resulta posible calcular la tasa media de procesamiento, la utilización media, el tiempo medio de espera, etc. para la mayoría de los algoritmos.

El sistema informático se describe, dentro de este modelo, como una red de servidores. Cada servidor dispone de una cola de procesos en espera. La CPU es un servidor con su cola de procesos preparados, al igual que el sistema de E/S con sus colas de dispositivos. Conociendo las tasas de llegada y el tiempo de servicio, podemos calcular la utilización, la longitud media de las colas, el tiempo medio de espera, etc. Este área de investigación se denomina **análisis de redes de colas**.

Por ejemplo, sea n la longitud media de la cola (excluyendo el proceso al que se está prestando servicio en ese momento), sea W el tiempo medio de espera en la cola y sea λ la tasa media de llegada de nuevos procesos a la cola (por ejemplo, tres procesos por segundo). Podemos estimar que, durante el tiempo W en que está esperando un proceso, llegarán a la cola $\lambda \times W$ nuevos procesos. Si el sistema está operando en régimen permanente, entonces el número de procesos que abandonan la cola debe ser igual al número de procesos que llegan. Por tanto:

$$n = \lambda \times W$$

Esta ecuación, conocida como **fórmula de Little**, resulta especialmente útil, ya que es válida para cualquier algoritmo de planificación y para cualquier distribución de las llegadas.

Podemos emplear la fórmula de Little para calcular una de las tres variables si conocemos dos de ellas. Por ejemplo, si sabemos que llegan 7 procesos por segundo (valor medio) y que normalmente hay 14 procesos en la cola, entonces podemos calcular el tiempo medio de espera por proceso, obteniendo que es de 2 segundos.

El análisis de colas puede resultar útil para comparar los distintos algoritmos de planificación, aunque también tiene sus limitaciones. Por el momento, las clases de algoritmos y de distribuciones que pueden incluirse en el análisis son bastante limitadas. Los análisis matemáticos necesarios para las distribuciones y algoritmos complejos pueden resultar enormemente difíciles. Por ello, las distribuciones de llegada y de servicio se suelen definir de forma matemáticamente tratable, pero

poco realista. Generalmente, también es necesario hacer una serie de suposiciones independientes, que pueden no ser excesivamente precisas. Debido a estas dificultades, a menudo los modelos de colas sólo representan una aproximación a los sistemas reales, y la precisión de los resultados obtenidos puede ser cuestionable.

5.7.3 Simulaciones

Para obtener una evaluación más precisa de los algoritmos de planificación, podemos usar **simulaciones**. Ejecutar las simulaciones requiere programar un modelo del sistema informático. Los componentes principales del sistema se representan mediante estructuras de datos software. El simulador tiene una variable que representa una señal de reloj y, cuando el valor de esta variable se incrementa, el simulador modifica el estado del sistema para reflejar las actividades de los dispositivos, de los procesos y del planificador. A medida que se ejecuta la simulación, las estadísticas que indican el rendimiento del algoritmo se recopilan y se presentan en la salida.

Los datos para controlar la simulación pueden generarse de varias formas. El método más común utiliza un generador de números aleatorios, que se programa para generar procesos, tiempos de ráfaga de CPU, llegadas, salidas, etc., de acuerdo con una serie de distribuciones de probabilidad. Las distribuciones pueden definirse matemáticamente (uniforme, exponencial, de Poisson) o empíricamente. Si hay que definir empíricamente una distribución, se toman medidas del sistema real que se esté estudiando. Los resultados de las medidas definen la distribución de probabilidad de los sucesos en el sistema real; esta distribución puede entonces utilizarse para controlar la simulación.

Sin embargo, una simulación controlada mediante una distribución de probabilidad puede ser imprecisa, debido a las relaciones entre sucesos sucesivos dentro del sistema real. La distribución de frecuencias sólo indica cuántas veces se produce cada suceso; no indica nada acerca del orden en que los sucesos tienen lugar. Para corregir este problema, podemos usar lo que se denominan **cintas de traza**. Para crear una cinta de traza se monitoriza el sistema real y se registra una secuencia de sucesos reales (Figura 5.15). Luego, esta secuencia se emplea para controlar la simulación. Las cintas de traza proporcionan una forma excelente de comparar dos algoritmos cuando se emplea exactamente el mismo conjunto de entradas reales. Este método permite obtener resultados precisos para los datos de entrada considerados.

Las simulaciones pueden resultar caras, ya que requieren muchas horas de tiempo de computadora. Una simulación más detallada proporciona resultados más precisos, pero también requie-

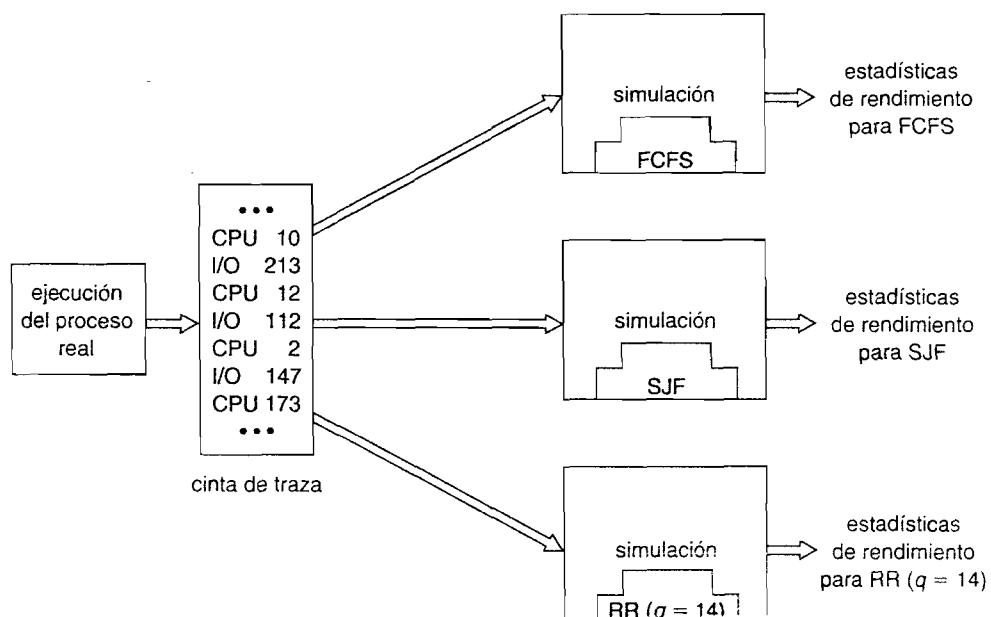


Figura 5.15 Evaluación de planificadores de CPU mediante simulación.

re más tiempo de cálculo. Además, las cintas de traza pueden requerir una gran cantidad de espacio de almacenamiento. Por último, el diseño, codificación y depuración del simulador pueden ser tareas de gran complejidad.

5.7.4 Implementación

Incluso las simulaciones tienen una precisión limitada. La única forma completamente precisa de evaluar un algoritmo de planificación es codificándolo, incluyéndolo en el sistema operativo y viendo cómo funciona. Este método introduce el algoritmo real en el sistema para su evaluación bajo condiciones de operación reales.

La principal dificultad de este método es su alto coste. No sólo se incurre en los costes de codificar el algoritmo y modificar el sistema para que lo soporte (junto con sus estructuras de datos requeridas) sino que también hay que tener en cuenta la reacción de los usuarios a los cambios constantes en el sistema operativo. La mayoría de los usuarios no están interesados en crear un sistema operativo mejor; simplemente desean ejecutar sus procesos y utilizar los resultados obtenidos. Los cambios continuos en el sistema operativo no ayudan a los usuarios a realizar su trabajo.

Otra dificultad es que el entorno en el que se use el algoritmo está sujeto a cambios. El entorno cambiará no sólo de la forma usual, a medida que se escriban nuevos programas y los tipos de problemas cambien, sino también como consecuencia del propio rendimiento del planificador. Si se da prioridad a los procesos cortos, entonces los usuarios pueden dividir los procesos largos en conjuntos de procesos más cortos. Si se asigna una mayor prioridad a los procesos interactivos que a los no interactivos, entonces los usuarios pueden decidir utilizar procesos interactivos.

Por ejemplo, un grupo de investigadores diseñó un sistema que clasificaba los procesos interactivos y no interactivos de forma automática, según la cantidad de operaciones de E/S realizadas a través del terminal. Si un proceso no leía ninguna entrada ni escribía ninguna salida en el terminal en un intervalo de 1 segundo, el proceso se clasificaba como no interactivo y se pasaba a la cola de prioridad más baja. En respuesta a esta política, un programador modificó sus programas para escribir un carácter arbitrario en el terminal a intervalos regulares de menos de 1 segundo. El sistema concedía a esos programas una prioridad alta, incluso aunque la salida presentada a través del terminal no tenía ningún sentido.

Los algoritmos de planificación más flexibles son aquéllos que pueden ser modificados por los administradores del sistema o por los usuarios, de modo que puedan ser ajustados para una aplicación específica o para un determinado conjunto de aplicaciones. Por ejemplo, una estación de trabajo que ejecute aplicaciones gráficas de gama alta puede tener necesidades de planificación diferentes que un servidor web o un servidor de archivos. Algunos sistemas operativos, en particular distintas versiones de UNIX, permiten que el administrador del sistema ajuste los parámetros de planificación para cada configuración concreta del sistema. Por ejemplo, Solaris proporciona el comando `dispadmin` para que el administrador del sistema modifique los parámetros de las clases de planificación descritas en la Sección 5.6.1.

Otro método consiste en emplear interfaces de programación de aplicaciones que permitan modificar la prioridad de un proceso o de una hebra; las API de Java, POSIX y Windows proporcionan dichas funciones. La desventaja de este método es que, a menudo, ajustar el rendimiento de un sistema o aplicación concretos no permite mejorar el rendimiento en otras situaciones más generales.

5.8 Resumen

La planificación de la CPU es la tarea de seleccionar un proceso en espera de la cola de procesos preparados y asignarle la CPU. El despachador asigna la CPU al proceso seleccionado.

La planificación FCFS (first-come, first-served; primero en llegar, primero en ser servido) es el algoritmo de planificación más sencillo, pero puede dar lugar a que los procesos de corta duración tengan que esperar a que se ejecuten otros procesos de duración mucho más grande. Probablemente, el algoritmo de planificación SJF (shortest-job-first, primero el trabajo más corto)

es el óptimo, proporcionando el tiempo medio de espera más corto. Sin embargo, la implementación del mecanismo de planificación SJF es complicada, ya que resulta difícil predecir la duración de la siguiente ráfaga de CPU. El algoritmo SJF es un caso especial del algoritmo de planificación general mediante prioridades, que simplemente asigna la CPU al proceso con prioridad más alta. Tanto la planificación por prioridades como la planificación SFJ presentan el problema de que los procesos pueden sufrir bloqueos indefinidos. El envejecimiento es una técnica que trata, precisamente, de evitar los bloqueos indefinidos.

La planificación por turnos es más apropiada para los sistemas de tiempo compartido (interactivos). La planificación por turnos asigna la CPU al primer proceso de la cola de procesos preparados durante q unidades de tiempo, donde q es el cuarto de tiempo. Después de q unidades de tiempo, si el proceso no ha cedido la CPU, es desalojado y se coloca al final de la cola de procesos preparados. El problema principal es la selección del tamaño del cuarto de tiempo. Si es demasiado largo, la planificación por turnos degenera en una planificación FCFS; si el cuarto de tiempo es demasiado corto, la carga de trabajo adicional asociada a las tareas de planificación (debido a los cambios de contexto) se hace excesiva.

El algoritmo FCFS es cooperativo; el algoritmo de planificación por turnos es apropiativo. Los algoritmos de planificación por prioridades y SFJ pueden ser apropiativo o sin desalojo (cooperativos).

Los algoritmos de colas multinivel permiten utilizar diferentes algoritmos para las diferentes clases de procesos. El modelo más común incluye una cola de procesos interactivos de primer plano que usa la planificación por turnos y una cola de procesos por lotes de segundo plano que usa la planificación FCFS. Las colas multinivel realimentadas permiten pasar los procesos de una cola a otra.

Muchos sistemas informáticos actuales soportan múltiples procesadores y permiten que cada procesador se auto-planifique de forma independiente. Normalmente, cada procesador mantiene su propia cola privada de procesos (o hebras), disponibles para ejecutarse. Entre los problemas relativos a la planificación de sistemas multiprocesador se encuentran los mecanismos de afinidad al procesador y de equilibrado de carga.

Los sistemas operativos que soportan hebras en el nivel del *kernel* deben planificar hebras, no procesos, para que se ejecuten en los procesadores disponibles; éste es el caso de Solaris y Windows XP. Ambos sistemas planifican las hebras usando algoritmos de planificación basados en prioridades y apropiativos, incluyendo soporte para hebras en tiempo real. El planificador de procesos de Linux usa un algoritmo basado en prioridades, también con soporte para tiempo real. Los algoritmos de planificación para estos tres sistemas operativos normalmente favorecen los procesos interactivos frente a los procesos por lotes y los procesos limitados por la CPU.

La amplia variedad de algoritmos de planificación obliga a emplear métodos para elegir entre ellos. Los métodos analíticos usan el análisis matemático para determinar el rendimiento de un algoritmo. Los métodos de simulación determinan el rendimiento imitando el algoritmo de planificación sobre una muestra “representativa” de procesos y calculando el rendimiento resultante. Sin embargo, la simulación sólo puede, como mucho, proporcionar una aproximación al verdadero rendimiento del sistema real; la única técnica plenamente fiable para evaluar un algoritmo de planificación es implementar el algoritmo en un sistema real y monitorizar su rendimiento en un entorno real.

Ejercicios

- 5.1 ¿Por qué es importante para el planificador diferenciar entre programas limitados por E/S y programas limitados por la CPU?
- 5.2 Explique cómo entran en conflicto en determinadas configuraciones los siguientes pares de criterios de planificación:
 - a. Utilización de la CPU y tiempo de respuesta.
 - b. Tiempo medio de procesamiento y tiempo máximo de espera.

- c. Utilización de los dispositivos de E/S y utilización de la CPU.
- 5.3 Considere la fórmula de la media exponencial utilizada para predecir la duración de la siguiente ráfaga de CPU. ¿Cuáles son las implicaciones de asignar los siguientes valores a los parámetros utilizados por el algoritmo?
- $\alpha = 0$ y $\tau_0 = 100$ milisegundos.
 - $\alpha = 0,99$ y $\tau_0 = 10$ milisegundos.
- 5.4 Considere el siguiente conjunto de procesos, estando la duración de las ráfagas de CPU especificada en milisegundos:
- | Proceso | Tiempo de ráfaga | Prioridad |
|---------|------------------|-----------|
| P_1 | 10 | 3 |
| P_2 | 1 | 1 |
| P_3 | 2 | 3 |
| P_4 | 1 | 4 |
| P_5 | 5 | 2 |
- Se supone que los procesos llegan en el orden P_1, P_2, P_3, P_4, P_5 en el instante 0.
- Dibuje cuatro diagramas de Gantt para ilustrar la ejecución de estos procesos, usando los siguientes algoritmos de planificación: FCFS, SJF, planificación por prioridades sin desalojo (un número de prioridad bajo indica una prioridad alta) y planificación por turnos (cuanto de tiempo = 1).
 - ¿Cuál es el tiempo de ejecución de cada proceso para cada algoritmo de planificación mencionado en el apartado a?
 - ¿Cuál es el tiempo de espera de cada proceso para cada algoritmo de planificación mencionado en el apartado a?
 - ¿Cuál de los algoritmos del apartado a permite obtener el tiempo medio de espera mínimo (teniendo en cuenta todos los procesos)?
- 5.5 ¿Cuáles de los siguientes algoritmos de planificación pueden dar lugar a bloqueos indefinidos?
- FCFS
 - SJF
 - planificación por turnos
 - planificación por prioridades
- 5.6 Considere una variante del algoritmo de planificación por turnos en la que las entradas en la cola de procesos preparados son punteros a los bloques PCB.
- ¿Cuál sería el efecto de colocar en la cola de procesos preparados dos punteros que hicieran referencia al mismo proceso?
 - ¿Cuáles son las dos principales ventajas y los dos principales inconvenientes de este esquema?
 - ¿Cómo modificaría el algoritmo por turnos básico para conseguir el mismo efecto sin usar punteros duplicados?
- 5.7 Considere un sistema que ejecuta diez tareas limitadas por E/S y una tarea limitada por la CPU. Suponga que las tareas limitadas por E/S ejecutan una operación de E/S por cada milisegundo de tiempo de CPU y que cada operación de E/S tarda 10 milisegundos en completarse. Suponga también que el tiempo de cambio de contexto es de 0,1 milisegundos y que

todos los procesos son tareas de larga duración. ¿Cuál es el grado de utilización de la CPU para un planificador por turnos cuando:

- a. el cuento de tiempo es de 1 milisegundo?
 - b. el cuento de tiempo es de 10 milisegundos?
- 5.8 Considere un sistema que implementa una planificación por colas multinivel. ¿Qué estrategia puede utilizar una computadora para maximizar la cantidad de tiempo de CPU asignada al proceso del usuario?
- 5.9 Considere un algoritmo de planificación por prioridades y apropiativo, en el que las prioridades cambien de forma dinámica. Los números de prioridad más altos indican una mayor prioridad. Cuando un proceso está esperando por la CPU (en la cola de procesos preparados, pero no en ejecución), su prioridad cambia con una velocidad de cambio α ; cuando se está ejecutando, su prioridad cambia con una velocidad de cambio β . Cuando entran en la cola de procesos preparados, a todos los procesos se les asigna una prioridad 0. Los parámetros α y β pueden seleccionarse para obtener muchos algoritmos de planificación diferentes.
- a. ¿Cuál es el algoritmo que resulta de $\beta > \alpha > 0$?
 - b. ¿Cuál es el algoritmo que resulta de $\alpha < \beta < 0$?
- 5.10 Explique las diferencias con respecto al grado en que los siguientes algoritmos de planificación favorecen a los procesos más cortos:
- a. FCFS
 - b. planificación por turnos
 - c. planificación mediante colas multinivel realimentadas
- 5.11 Usando el algoritmo de planificación de Windows XP, ¿cuál es la prioridad numérica de una hebra en los casos siguientes?
- a. Una hebra de la clase REALTIME_PRIORITY_CLASS con una prioridad relativa HIGHEST.
 - b. Una hebra de la clase NORMAL_PRIORITY_CLASS con una prioridad relativa NORMAL.
 - c. Una hebra de la clase HIGH_PRIORITY_CLASS con una prioridad relativa ABOVE_NORMAL.
- 5.12 Considere el algoritmo de planificación del sistema operativo Solaris para las hebras de tiempo compartido.
- a. ¿Cuál es el cuento de tiempo (en milisegundos) para una hebra con prioridad 10?
 - b. Suponga que una hebra con una prioridad de 35 ha utilizado su cuento de tiempo completo sin bloquearse. ¿Qué nueva prioridad asignará el planificador a esta hebra?
 - c. Suponga que una hebra con una prioridad de 35 se bloquea en espera de una operación de E/S antes de consumir su cuento de tiempo. ¿Qué nueva prioridad asignará el planificador a esta hebra?
- 5.13 El planificador tradicional de UNIX fuerza una relación inversa entre los números de prioridad y las prioridades: cuanto mayor es el número, menor es la prioridad. El planificador recalcula las prioridades de los procesos una vez por segundo usando la siguiente función:
- $$\text{Prioridad} = (\text{uso reciente de la CPU} / 2) + \text{prioridad base}$$
- donde prioridad base = 60 y *uso reciente de la CPU* hace referencia a un valor que indica con qué frecuencia ha empleado un proceso la CPU desde que se calcularon las prioridades por última vez.
- Suponga que el uso reciente de la CPU para el proceso P_1 es de 40, para el proceso P_2 de 18 y para el proceso P_3 de 10. ¿Cuáles serán las nuevas prioridades para estos tres procesos

cuando éstas se vuelvan a calcular? Teniendo esto en cuenta, ¿incrementará o disminuirá el planificador tradicional de UNIX la prioridad relativa de un proceso limitado por la CPU?

Notas bibliográficas

Las colas realimentadas se implementaron originalmente en el sistema CTSS descrito en Corbató et al. [1962]. Este sistema de planificación mediante colas realimentadas se analiza en Schrage [1967]. El algoritmo de planificación mediante prioridades y apropiativo del Ejercicio 5.9 fue sugerido por Kleinrock [1975].

Anderson et al. [1989], Lewis y Berg [1998] y Philbin et al. [1996] se ocupan de la planificación de hebras. La planificación para sistemas multiprocesador se aborda en Tucker y Gupta [1989], Zahorjan y McCann [1990], Feitelson y Rudolph [1990], Leutenegger y Vernon [1990], Blumofe y Leiserson [1994], Polychronopoulos y Kuck [1987] y Lucco [1992]. Las técnicas de planificación que tienen en cuenta la información relativa a los tiempos de ejecución anteriores de los procesos fueron descritas en Fisher [1981], Hall et al. [1996] y Lowney et al. [1993].

Las técnicas de planificación para sistemas en tiempo real se estudian en Liu y Layland [1973], Abbot [1984], Jensen et al. [1985], Hong et al. [1989] y Khanna et al. [1992]. Zhao [1989] compiló una edición especial de *Operating System Review* dedicada los sistemas operativos en tiempo real.

Los planificadores de cuota equitativa se cubren en Henry [1984], Woodside [1986] y Kay y Lauder [1988].

Las políticas de planificación utilizadas en el sistema operativo UNIX V se describen en Bach [1987]; las correspondientes políticas para UNIX BSD 4.4 se presentan en Mckusick et al. [1996]; y para el sistema operativo Mach se describen en Black [1990]. Bovet y Cesati [2002] analizan el tema de la planificación en Linux. La planificación en Solaris se describe en Mauro y McDougall [2001]. Solomon [1998] y Solomon y Russinovich [2000] abordan la planificación en Windows NT y Windows 2000, respectivamente. Butenhof [1997] y Lewis y Berg [1998] describen la planificación en los sistemas Pthreads.

Sincronización de procesos

Un proceso cooperativo es aquel que puede afectar o verse afectado por otros procesos que estén ejecutándose en el sistema. Los procesos cooperativos pueden compartir directamente un espacio de direcciones lógico (es decir, tanto código como datos) o compartir los datos sólo a través de archivos o mensajes. El primer caso se consigue mediante el uso de procesos ligeros o hebras, los cuales se han estudiado en el Capítulo 4. El acceso concurrente a datos compartidos puede dar lugar a incoherencia de los datos y en este capítulo vamos a ver varios mecanismos para asegurar la ejecución ordenada de procesos cooperativos que comparten un espacio de direcciones lógico, de modo que se mantenga la coherencia de los datos.

OBJETIVOS DEL CAPÍTULO

- Presentar el problema de las secciones críticas, cuyas soluciones pueden utilizarse para asegurar la coherencia de los datos compartidos.
- Presentar soluciones tanto software como hardware para el problema de las secciones críticas.
- Presentar el concepto de transacción atómica y describir los mecanismos para garantizar la atomicidad.

6.1 Fundamentos

En el Capítulo 3 hemos desarrollado un modelo de sistema formado por procesos o hebras secuenciales cooperativas, los cuales se ejecutan de manera asíncrona y posiblemente compartiendo datos. Ilustramos este modelo con el problema del productor-consumidor, que es representativo de los sistemas operativos. Específicamente, en la Sección 3.4.1 hemos descrito cómo podría utilizarse un búfer limitado para permitir a los procesos compartir la memoria.

Consideremos de nuevo el búfer limitado. Como ya apuntamos, nuestra solución permite que haya como máximo BUFFER_SIZE -1 elementos en el búfer al mismo tiempo. Suponga que deseamos modificar el algoritmo para remediar esta deficiencia. Una posibilidad consiste en añadir una variable entera counter, inicializada con el valor 0. counter se incrementa cada vez que se añade un nuevo elemento al búfer y se decrementa cada vez que se elimina un elemento del búfer. El código para el proceso productor se puede modificar del siguiente modo:

```
while (true)
{
    /* produce un elemento en nextProduced */
    while (counter == BUFFER_SIZE)
        ; /* no hacer nada */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
```

```

        counter++;
    }
}

```

El código para el proceso consumidor se puede modificar del modo siguiente

```

while (true)
{
    while (counter == 0)
        ; /* no hacer nada */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume el elemento que hay en nextConsumed */
}

```

Aunque las rutinas del productor y del consumidor son correctas por separado, no pueden funcionar correctamente cuando se ejecutan de forma concurrente. Por ejemplo, suponga que el valor de la variable `counter` es actualmente 5 y que los procesos productor y consumidor ejecutan las instrucciones “`counter++`” y “`counter--`” de forma concurrente. Después de la ejecución de estas dos instrucciones, el valor de la variable `counter` puede ser 4, 5 o 6. El único resultado correcto es, no obstante, `counter == 5`, valor que se genera correctamente si los procesos consumidor y productor se ejecutan por separado.

Podemos demostrar que el valor de `counter` puede ser incorrecto del modo siguiente. Observe que la instrucción “`counter++`” se puede implementar en lenguaje máquina, en un procesador típico, como sigue:

```

registro1 = counter
registro1 = registro1 + 1
counter = registro1

```

donde `registro1` es un registro local de la CPU. De forma similar, la instrucción “`counter--`” se implementa como sigue:

```

registro2 = counter
registro2 = registro2 + 1
counter = registro2

```

donde, de nuevo, `registro2` es un registro local de la CPU. Incluso aunque `registro1` y `registro2` puedan ser el mismo registro físico (por ejemplo, un acumulador), recuerde que los contenidos de este registro serán guardados y restaurados por la rutina de tratamiento de interrupciones (Sección 1.2.3).

La ejecución concurrente de “`counter++`” y “`counter--`” es equivalente a una ejecución secuencial donde las instrucciones de menor nivel indicadas anteriormente se intercalan en un cierto orden arbitrario (pero el orden dentro de cada instrucción de alto nivel se conserva). Una intercalación de este tipo sería:

T_0 :	<i>productor</i>	execute	$registro_1 = \text{counter}$	$[\text{registro}_1 = 5]$
T_1 :	<i>productor</i>	execute	$registro_1 = registro_1 + 1$	$[\text{registro}_1 = 6]$
T_2 :	<i>consumidor</i>	execute	$registro_2 = \text{counter}$	$[\text{registro}_2 = 5]$
T_3 :	<i>consumidor</i>	execute	$registro_2 = registro_2 + 1$	$[\text{registro}_2 = 6]$
T_4 :	<i>productor</i>	execute	$\text{counter} = registro_1$	$[\text{counter} = 6]$
T_5 :	<i>productor</i>	execute	$\text{counter} = registro_2$	$[\text{counter} = 4]$

Observe que hemos llegado al estado incorrecto “`counter == 4`”, lo que indica que cuatro posiciones del búfer están llenas, cuando, de hecho, son cinco las posiciones llenas. Si invirtiéramos el orden de las instrucciones T_4 y T_5 , llegaríamos al estado incorrecto “`counter == 6`”.

Llegaríamos a este estado incorrecto porque hemos permitido que ambos procesos manipulen la variable `counter` de forma concurrente. Una situación como ésta, donde varios procesos manipulan y acceden a los mismos datos concurrentemente y el resultado de la ejecución depende del orden concreto en que se produzcan los accesos, se conoce como **condición de carrera**. Para protegerse frente a este tipo de condiciones, necesitamos garantizar que sólo un proceso cada vez pueda manipular la variable `counter`. Para conseguir tal garantía, tenemos que sincronizar de alguna manera los procesos.

Las situaciones como la que acabamos de describir se producen frecuentemente en los sistemas operativos, cuando las diferentes partes del sistema manipulan los recursos. Claramente, necesitamos que los cambios resultantes no interfieran entre sí. Debido a la importancia de este problema, gran parte de este capítulo se dedica a la sincronización y coordinación de procesos.

6.2 El problema de la sección crítica

Considere un sistema que consta de n procesos $\{P_0, P_1, \dots, P_{n-1}\}$. Cada proceso tiene un segmento de código, llamado **sección crítica**, en el que el proceso puede modificar variables comunes, actualizar una tabla, escribir en un archivo, etc. La característica importante del sistema es que, cuando un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica. Es decir, dos procesos no pueden ejecutar su sección crítica al mismo tiempo. El *problema de la sección crítica* consiste en diseñar un protocolo que los procesos puedan usar para cooperar de esta forma. Cada proceso debe solicitar permiso para entrar en su sección crítica; la sección de código que implementa esta solicitud es la **sección de entrada**. La sección crítica puede ir seguida de una **sección de salida**. El código restante se encuentra en la **sección restante**. La estructura general de un proceso típico P_i es la que se muestra en la Figura 6.1. La sección de entrada y la sección de salida se han indicado en recuadros para resaltar estos importantes segmentos de código.

Cualquier solución al problema de la sección crítica deberá satisfacer los tres requisitos siguientes:

1. **Exclusión mutua.** Si el proceso P_i está ejecutándose en su sección crítica, los demás procesos no pueden estar ejecutando sus secciones críticas.
2. **Progreso.** Si ningún proceso está ejecutando su sección crítica y algunos procesos desean entrar en sus correspondientes secciones críticas, sólo aquellos procesos que no estén ejecutando sus secciones restantes pueden participar en la decisión de cuál será el siguiente que entre en su sección crítica, y esta selección no se puede posponer indefinidamente.
3. **Espera limitada.** Existe un límite en el número de veces que se permite que otros procesos entren en sus secciones críticas después de que un proceso haya hecho una solicitud para entrar en su sección crítica y antes de que la misma haya sido concedida.

Estamos suponiendo que cada proceso se ejecuta a una velocidad distinta de cero. Sin embargo, no podemos hacer ninguna suposición sobre la **velocidad relativa** de los n procesos.

En un instante de tiempo determinado, pueden estar activos muchos procesos en modo *kernel* en el sistema operativo. Como resultado, el código que implementa el sistema operativo (el *código*

```

do {
    sección de entrada
    sección crítica
    sección de salida
    sección restante
} while (TRUE);

```

Figura 6.1 Estructura general de un proceso típico P_i .

go del kernel) está sujeto a varias posibles condiciones de carrera. Considere por ejemplo una estructura de datos del *kernel* que mantenga una lista de todos los archivos abiertos en el sistema. Esta lista debe modificarse cuando se abre un nuevo archivo o se cierra un archivo abierto (añadiendo el archivo a la lista o eliminándolo de la misma). Si dos procesos abrieran archivos simultáneamente, las actualizaciones de la lista podrían llevar a una condición de carrera. Otras estructuras de datos del *kernel* propensas a posibles condiciones de carrera son aquéllas empleadas para controlar la asignación de memoria, las listas de procesos y para gestionar las interrupciones. Es responsabilidad de los desarrolladores del *kernel* asegurar que el sistema operativo esté libre de tales condiciones de carrera.

Se usan dos métodos generales para gestionar las secciones críticas en los sistemas operativos: (1) los *kernels apropiativos* y (2) los *kernels no apropiativos*. Un *kernel* apropiativo permite que un proceso sea desalojado mientras se está ejecutando en modo *kernel*. Un *kernel* no apropiativo no permite que un proceso que se esté ejecutando en modo *kernel* sea desalojado; el proceso en modo *kernel* se ejecutará hasta que salga de dicho modo, hasta que se bloquee o hasta que ceda voluntariamente el control de la CPU. Obviamente, un *kernel* no apropiativo está esencialmente libre de condiciones de carrera en lo que respecta a las estructuras de datos del *kernel*, ya que sólo hay un proceso activo en el *kernel* en cada momento. No podemos decir lo mismo acerca de los *kernels* apropiativos, por lo que deben ser diseñados cuidadosamente para asegurar que los datos compartidos del *kernel* no se vean afectados por posibles condiciones de carrera. Los *kernel* apropiativos son especialmente difíciles de diseñar en arquitecturas SMP, dado que en estos entornos es posible que dos procesos se ejecuten simultáneamente en modo *kernel* en procesadores diferentes.

¿Por qué entonces sería preferible un *kernel* apropiativo a uno no apropiativo? Un *kernel* apropiativo es más adecuado para la programación en tiempo real, ya que permite a un proceso en tiempo real desalojar a un proceso que se esté ejecutando actualmente en el *kernel*. Además, un *kernel* apropiativo puede tener una mejor capacidad de respuesta, ya que existe menos riesgo de que un proceso en modo *kernel* se ejecute durante un período de tiempo arbitrariamente largo antes de ceder el procesador a los procesos que estén a la espera. Por supuesto, este efecto puede minimizarse diseñando código de *kernel* que no presente este tipo de comportamiento.

Windows XP y Windows 2000 son *kernels* no apropiativos, al igual que el *kernel* tradicional de UNIX. Antes de Linux 2.6, el *kernel* de Linux también era no apropiativo. Sin embargo, con la versión del *kernel* 2.6, Linux cambió al modelo apropiativo. Varias versiones comerciales de UNIX usan un *kernel* apropiativo, incluyendo Solaris e IRIX.

6.3 Solución de Peterson

A continuación presentamos una solución clásica basada en software al problema de la sección crítica, conocida con el nombre de **solución de Peterson**. Debido a la forma en que las arquitecturas informáticas modernas ejecutan las instrucciones básicas en lenguaje máquina, como *load* y *store*, no hay garantías de que la solución de Peterson funcione correctamente en tales arquitecturas. Sin embargo, presentamos esta solución porque proporciona una buena descripción algorítmica de la resolución del problema de la sección crítica e ilustra algunas de las complejidades asociadas al diseño de software que satisfaga los requisitos de exclusión mutua, progreso y tiempo de espera limitado.

La solución de Peterson se restringe a dos procesos que van alternando la ejecución de sus secciones críticas y de sus secciones restantes. Los procesos se numeran como P_0 y P_1 . Por conveniencia, cuando hablemos de P_i , usaremos P_j para referirnos al otro proceso; es decir, j es igual a $1 - i$.

La solución de Peterson requiere que los dos procesos comparten dos elementos de datos:

```
int turn;
boolean flag[2];
```

La variable *turn* indica qué proceso va a entrar en su sección crítica. Es decir, si *turn* == i , entonces el proceso P_i puede ejecutar su sección crítica. La matriz *flag* se usa para indicar si un proceso está preparado para entrar en su sección crítica. Por ejemplo, si *flag*[i] es verdadera, este

valor indica que P_i está preparado para entrar en su sección crítica. Habiendo explicado estas estructuras de datos, ya estamos preparados para describir el algoritmo mostrado en la Figura 6.2.

Para entrar en la sección crítica, el proceso P_i primero asigna el valor verdadero a `flag[i]` y luego asigna a `turn` el valor j , confirmando así que, si el otro proceso desea entrar en la sección crítica, puede hacerlo. Si ambos procesos intentan entrar al mismo tiempo, a la variable `turn` se le asignarán los valores tanto i como j aproximadamente al mismo tiempo. Sólo una de estas asignaciones permanecerá; la otra tendrá lugar, pero será sobreescrita inmediatamente. El valor que adopte `turn` decidirá cuál de los dos procesos podrá entrar primero en su sección crítica.

Ahora vamos a demostrar que esta solución es correcta. Necesitamos demostrar que:

1. La exclusión mutua se conserva.
2. El requisito de progreso se satisface.
3. El requisito de espera limitada se cumple.

Para probar la primera propiedad, observamos que P_i entra en su sección crítica sólo si `flag[j] == false` o `turn == i`. Observe también que, si ambos procesos pudieran estar ejecutando sus secciones críticas al mismo tiempo, entonces `flag[0] == flag[1] == true`. Estas dos observaciones implican que P_0 y P_1 no podrían haber terminado de ejecutar sus instrucciones `while` aproximadamente al mismo tiempo, dado que el valor de `turn` puede ser 0 o 1, pero no ambos. Por tanto, uno de los procesos, por ejemplo P_j , debe haber terminado de ejecutar la instrucción `while`, mientras que P_i tendrá que ejecutar al menos una instrucción adicional "`turn == j`". Sin embargo, dado que, en dicho instante, `flag[j] == true` y `turn == j`, y esta condición se cumplirá mientras que j se encuentre en su sección crítica, el resultado es que la exclusión mutua se conserva.

Para probar la segunda y tercera propiedades, observamos que sólo se puede impedir que un proceso P_i entre en la sección crítica si el proceso se atasca en el bucle `while` con la condición `flag[j] == true` y `turn == j`; este bucle es el único posible. Si P_j no está preparado para entrar en la sección crítica, entonces `flag[j] == false`, y P_i puede entrar en su sección crítica. Si P_j ha definido `flag[j]` como `true` y también está ejecutando su instrucción `while`, entonces `turn == i` o `turn == j`. Si `turn == i`, entonces P_i entrará en la sección crítica. Si `turn == j`, entonces P_j entrará en la sección crítica. Sin embargo, una vez que P_j salga de su sección crítica, asignará de nuevo a `flag[j]` el valor `false`, permitiendo que P_i entre en su sección crítica. Si P_j asigna de nuevo a `flag[j]` el valor `true`, también debe asignar el valor i a `turn`. Por tanto, dado que P_j no cambia el valor de la variable `turn` mientras está ejecutando la instrucción `while`, P_i entrará en la sección crítica (progreso) después de como máximo una entrada de P_j (espera limitada).

6.4 Hardware de sincronización

Acabamos de describir una solución software al problema de la sección crítica. En general, podemos afirmar que cualquier solución al problema de la sección crítica requiere una herramienta

```

do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
} sección crítica
flag[i] = FALSE;
} sección restante
} while (TRUE);

```

Figura 6.2 La estructura del proceso P_i en la solución de Peterson.

```

do {
    adquirir cerrojo
    sección crítica
    liberar cerrojo
    sección restante
} while (TRUE);

```

Figura 6.3 Solución al problema de la sección crítica usando cerros.

muy simple, un **cerrojo**. Las condiciones de carrera se evitan requiriendo que las regiones críticas se protejan mediante cerros. Es decir, un proceso debe adquirir un cerrojo antes de entrar en una sección crítica y liberarlo cuando salga de la misma. Esto se ilustra en la Figura 6.3.

A continuación, vamos a explorar varias soluciones más al problema de la sección crítica, usando técnicas que van desde el soporte hardware a las API software que los programadores de aplicaciones tienen a su disposición. Todas estas soluciones se basan en la premisa del bloqueo; sin embargo, como veremos, el diseño de tales bloqueos (cerros) puede ser bastante complejo.

El soporte hardware puede facilitar cualquier tarea de programación y mejorar la eficiencia del sistema. En esta sección, vamos a presentar algunas instrucciones hardware sencillas que están disponibles en muchos sistemas y mostraremos cómo se pueden usar de forma efectiva en la resolución del problema de la sección crítica.

El problema de la sección crítica podría resolverse de forma simple en un entorno de un solo procesador si pudiéramos impedir que se produjeran interrupciones mientras se está modificando una variable compartida. De este modo, podríamos asegurar que la secuencia actual de instrucciones se ejecute por orden, sin posibilidad de desalojo. Ninguna otra instrucción se ejecutará, por lo que no se producirán modificaciones inesperadas de la variable compartida. Éste es el método que emplean los *kernels* no apropiativos.

Lamentablemente, esta solución no resulta tan adecuada en un entorno multiprocesador. Desactivar las interrupciones en un sistema multiprocesador puede consumir mucho tiempo, ya que hay que pasar el mensaje a todos los procesadores. Este paso de mensajes retarda la entrada en cada sección crítica y la eficiencia del sistema disminuye. También hay que tener en cuenta el efecto del reloj del sistema, en el caso de que el reloj se actualice mediante interrupciones.

Por tanto, muchos sistemas informáticos modernos proporcionan instrucciones hardware especiales que nos permiten consultar y modificar el contenido de una palabra o intercambiar los contenidos de dos palabras **atómicamente**, es decir, como una unidad de trabajo ininterrumpible. Podemos usar estas instrucciones especiales para resolver el problema de la sección crítica de una forma relativamente simple. En lugar de estudiar una instrucción específica de una determinada máquina, vamos a abstraer los principales conceptos que subyacen a este tipo de instrucciones.

La instrucción `TestAndSet` para leer y modificar atómicamente una variable puede definirse como se muestra en la Figura 6.4. La característica importante es que esta instrucción se ejecuta atómicamente. Por tanto, si dos instrucciones `TestAndSet` se ejecutan simultáneamente (cada una en una CPU diferente), se ejecutarán secuencialmente en un orden arbitrario. Si la máquina soporta la instrucción `TestAndSet`, entonces podemos implementar la exclusión mutua declarando una variable booleana `lock` inicializada con el valor `false`. En la Figura 6.5 se muestra la estructura del proceso P_i .

```

boolean TestAndSet (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}

```

Figura 6.4 Definición de la instrucción `TestAndSet`.

```

do {
    while (TestAndSetLock (&lock))
        ;      // no hacer nada

        // sección crítica
    lock = FALSE;

        // sección restante
} while (TRUE);

```

Figura 6.5 Implementación de la exclusión mutua con TestAndSet().

La instrucción Swap() para intercambiar el valor de dos variables, a diferencia de la instrucción TestAndSet() opera sobre los contenidos de dos palabras; se define como se muestra en la Figura 6.6. Como la instrucción TestAndSet(), se ejecuta atómicamente. Si la máquina soporta la instrucción Swap(), entonces la exclusión mutua se proporciona como sigue: se declara una variable global booleana lock y se inicializa como false. Además, cada proceso tiene una variable local booleana key. La estructura del proceso P_i se muestra en la Figura 6.7.

Aunque estos algoritmos satisfacen el requisito de exclusión mutua, no satisfacen el requisito de espera limitada. En la Figura 6.8 presentamos otro algoritmo usando la instrucción TestAndSet() que satisface todos los requisitos del problema de la sección crítica. Las estructuras de datos comunes son

```

boolean waiting[n];
boolean lock;

```

Estas estructuras de datos se inicializan con el valor false. Para probar que el requisito de exclusión mutua se cumple, observamos que el proceso P_i puede entrar en su sección crítica sólo si $waiting[i] == \text{false}$ o $key == \text{false}$. El valor de key puede ser false sólo si se ejecuta TestAndSet(); el primer proceso que ejecute TestAndSet() comprobará que $key == \text{false}$ y todos los demás tendrán que esperar. La variable $waiting[i]$ puede tomar el valor false sólo si otro proceso sale de su sección crítica; sólo se asigna el valor false a una única variable $waiting[i]$, manteniendo el requisito de exclusión mutua.

Para probar que se cumple el requisito de progreso, observamos que los argumentos relativos a la exclusión mutua también se aplican aquí, dado que un proceso que sale de la sección crítica configura lock como false o $waiting[j]$ como false. En ambos casos, se permite que un proceso en espera entre en su sección crítica para continuar.

Para probar que se cumple el requisito de tiempo de espera limitado, observamos que, cuando un proceso deja su sección crítica, explora la matriz $waiting$ en el orden cíclico ($i + 1, i + 2, \dots, n - 1, 0, \dots, i - 1$) y selecciona el primer proceso (según este orden) que se encuentre en la sección de entrada ($waiting[j] == \text{true}$) como siguiente proceso que debe entrar en la sección crítica. Cualquier proceso que quiera entrar en su sección crítica podrá hacerlo en, como máximo, $n - 1$ turnos.

Lamentablemente para los diseñadores de hardware, la implementación de instrucciones TestAndSet en los sistemas multiprocesador no es una tarea trivial. Tales implementaciones se tratan en los libros dedicados a arquitecturas informáticas.

```

void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

```

Figura 6.6 La definición de la instrucción Swap().

```

do {
    key = TRUE;
    while (key == TRUE)
        Swap(&lock, &key);

        // sección crítica

    lock = FALSE;
    // sección restante
}while (TRUE);

```

Figura 6.7 Implementación de la exclusión mutua con la instrucción Swap().

```

do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;

    // sección crítica

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;

    // sección restante
}while (TRUE);

```

Figura 6.8 Exclusión mutua con tiempo de espera limitada, utilizando la instrucción TestAndSet().

6.5 Semáforos

Las diversas soluciones hardware al problema de la sección crítica, basadas en las instrucciones TestAndSet() y Swap() y presentadas en la Sección 6.4, son complicadas de utilizar por los programadores de aplicaciones. Para superar esta dificultad, podemos usar una herramienta de sincronización denominada **semáforo**.

Un semáforo *S* es una variable entera a la que, dejando aparte la inicialización, sólo se accede mediante dos operaciones atómicas estándar: wait() y signal(). Originalmente, la operación wait() se denominaba P (del término holandés *proberen*, probar); mientras que signal() se denominaba originalmente V (*verhogen*, incrementar). La definición de wait() es la que sigue:

```

wait(S) {
    while S <= 0
        ; - no-op
    S--;
}

```

La definición de signal() es:

```
signal(S) {
    S++;
}
```

Todas las modificaciones del valor entero del semáforo en las operaciones `wait()` y `signal()` deben ejecutarse de forma indivisible. Es decir, cuando un proceso modifica el valor del semáforo, ningún otro proceso puede modificar simultáneamente el valor de dicho semáforo. Además, en el caso de `wait()`, la prueba del valor entero de `S` ($S \leq 0$), y su posible modificación (`S --`) también se deben ejecutar sin interrupción. Veremos cómo pueden implementarse estas operaciones en la Sección 6.5.2, pero antes, vamos a ver cómo pueden utilizarse los semáforos.

6.5.1 Utilización

Los sistemas operativos diferencian a menudo entre semáforos contadores y semáforos binarios. El valor de un **semáforo contador** puede variar en un dominio no restringido, mientras que el valor de un **semáforo binario** sólo puede ser 0 o 1. En algunos sistemas, los semáforos binarios se conocen como **cerrojos mutex**, ya que son cerrojos que proporcionan exclusión mutua.

Podemos usar semáforos binarios para abordar el problema de la sección crítica en el caso de múltiples procesos. Los n procesos comparten un semáforo, `mutex`, inicializado con el valor 1. Cada proceso P_i se organiza como se muestra en la Figura 6.9.

Los semáforos contadores se pueden usar para controlar el acceso a un determinado recurso formado por un número finito de instancias. El semáforo se inicializa con el número de recursos disponibles. Cada proceso que desee usar un recurso ejecuta una operación `wait()` en el semáforo (decrementando la cuenta). Cuando un proceso libera un recurso, ejecuta una operación `signal()` (incrementando la cuenta). Cuando la cuenta del semáforo llega a 0, todos los recursos estarán en uso. Después, los procesos que deseen usar un recurso se bloquearán hasta que la cuenta sea mayor que 0.

También podemos usar los semáforos para resolver diversos problemas de sincronización. Por ejemplo, considere dos procesos que se estén ejecutando de forma concurrente: P_1 con una instrucción S_1 y P_2 con una instrucción S_2 . Suponga que necesitamos que S_2 se ejecute sólo después de que S_1 se haya completado. Podemos implementar este esquema dejando que P_1 y P_2 compartan un semáforo común `synch`, inicializado con el valor 0, e insertando las instrucciones:

```
S1;
signal(synch);
```

en el proceso P_1 , y las instrucciones

```
wait(synch);
S2;
```

en el proceso P_2 . Dado que `synch` se inicializa con el valor 0, P_2 ejecutará S_2 sólo después de que P_1 haya invocado `signal(synch)`, instrucción que sigue a la ejecución de S_1 .

```
do
    waiting(mutex);
    // sección crítica
    signal(mutex);
    // sección restante
}while (TRUE);
```

Figura 6.9 Implementación de la exclusión mutua con semáforos.

6.5.2 Implementación

La principal desventaja de la definición de semáforo dada aquí es que requiere una **espera activa**. Mientras un proceso está en su sección crítica, cualquier otro proceso que intente entrar en su sección crítica debe ejecutar continuamente un bucle en el código de entrada. Este bucle continuo plantea claramente un problema en un sistema real de multiprogramación, donde una sola CPU se comparte entre muchos procesos. La espera activa desperdicia ciclos de CPU que algunos otros procesos podrían usar de forma productiva. Este tipo de semáforo también se denomina **cerrojo mediante bucle sin fin** (*spinlock*), ya que el proceso “permanece en un bucle sin fin” en espera de adquirir el cerrojo. (Los cerrojos mediante bucle sin fin tienen una ventaja y es que no requieren ningún cambio de contexto cuando un proceso tiene que esperar para adquirir un cerrojo. Los cambios de contexto pueden llevar un tiempo considerable. Por tanto, cuando se espera que los cerrojos se mantengan durante un período de tiempo corto, los cerrojos mediante bucle sin fin pueden resultar útiles; por eso se emplean a menudo en los sistemas multiprocesador, donde una hebra puede “ejecutar un bucle” sobre un procesador mientras otra hebra ejecuta su sección crítica en otro procesador).

Para salvar la necesidad de la espera activa, podemos modificar la definición de las operaciones de semáforo `wait()` y `signal()`. Cuando un proceso ejecuta la operación `wait()` y determina que el valor del semáforo no es positivo, tiene que esperar. Sin embargo, en lugar de entrar en una espera activa, el proceso puede *bloquearse* a sí mismo. La operación de bloqueo coloca al proceso en una cola de espera asociada con el semáforo y el estado del proceso pasa al estado de espera. A continuación, el control se transfiere al planificador de la CPU, que selecciona otro proceso para su ejecución.

Un proceso bloqueado, que está esperando en un semáforo *S*, debe reiniciarse cuando algún otro proceso ejecuta una operación `signal()`. El proceso se reinicia mediante una operación `wakeup()`, que cambia al proceso del estado de espera al estado de preparado. El proceso se coloca en la cola de procesos preparados. (La CPU puede o no conmutar del proceso en ejecución al proceso que se acaba de pasar al estado de preparado, dependiendo del algoritmo de planificación de la CPU.)

Para implementar semáforos usando esta definición, definimos un semáforo como una estructura “C”:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Cada semáforo tiene un valor (`value`) entero y una lista de procesos `list`. Cuando un proceso tiene que esperar en un semáforo, se añade a la lista de procesos. Una operación `signal()` elimina un proceso de la lista de procesos en espera y lo despierta.

La operación de semáforo `wait()` ahora se puede definir del siguiente modo:

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        añadir este proceso a S->list;
        block();
    }
}
```

La operación de semáforo `signal()` ahora puede definirse así:

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        eliminar un proceso P de S->list;
```

```

        wakeup P;
    }
}

```

La operación `block()` suspende al proceso que la ha invocado. La operación `wakeup()` reanuda la ejecución de un proceso bloqueado `P`. Estas dos operaciones las proporciona el sistema operativo como llamadas al sistema básicas.

Observe que, aunque bajo la definición clásica de semáforos con espera activa, el valor del semáforo nunca es negativo, esta implementación sí que puede tener valores negativos de semáforo. Si el valor del semáforo es negativo, su módulo es el número de procesos en espera en dicho semáforo. Este hecho resulta de conmutar el orden de las operaciones de decremento y de prueba en la implementación de la operación `wait()`.

La lista de procesos en espera puede implementarse fácilmente mediante un campo de enlace en cada bloque de control de proceso (PCB). Cada semáforo contiene un valor entero y un puntero a la lista de bloques PCB. Una forma de añadir y eliminar procesos de la lista de manera que se garantice un tiempo de espera limitado consiste en usar una cola FIFO, donde el semáforo contenga punteros a ambos extremos de la cola. Sin embargo, en general, la lista puede utilizar *cualquier* estrategia de gestión de la cola. El uso correcto de los semáforos no depende de la estrategia concreta de gestión de la cola que se emplee para las listas de los semáforos.

El aspecto crítico de los semáforos es que se deben ejecutar atómicamente: tenemos que garantizar que dos procesos no puedan ejecutar al mismo tiempo sendas operaciones `wait()` y `signal()` sobre el mismo semáforo. Se trata de un problema de sección crítica. En un entorno de un solo procesador (es decir, en el que sólo existe una CPU), podemos solucionar el problema de forma sencilla inhibiendo las interrupciones durante el tiempo en que se ejecutan las operaciones `wait()` y `signal()`. Este esquema funciona adecuadamente en un entorno de un solo procesador porque, una vez que se inhiben las interrupciones, las instrucciones de los diferentes procesos no pueden intercalarse: sólo se ejecuta el proceso actual hasta que se reactivan las interrupciones y el planificador puede tomar de nuevo el control.

En un entorno multiprocesador, hay que deshabilitar las interrupciones en cada procesador; si no se hace así, las instrucciones de los diferentes procesos (que estén ejecutándose sobre diferentes procesadores) pueden intercalarse de forma arbitraria. Deshabilitar las interrupciones en todos los procesadores puede ser una tarea compleja y, además, puede disminuir seriamente el rendimiento. Por tanto, los sistemas SMP deben proporcionar técnicas alternativas de bloqueo, como por ejemplo cerrojos mediante bucle sin fin, para asegurar que las operaciones `wait()` y `signal()` se ejecuten atómicamente.

Es importante recalcar que no hemos eliminado por completo la espera activa con esta definición de las operaciones `wait()` y `signal()`; en lugar de ello, hemos eliminado la espera activa de la sección de entrada y la hemos llevado a las secciones críticas de los programas de aplicación. Además, hemos limitado la espera activa a las secciones críticas de las operaciones `wait()` y `signal()`, y estas secciones son cortas (si se codifican apropiadamente, no deberían tener más de unas diez instrucciones). Por tanto, la sección crítica casi nunca está ocupada y raras veces se produce este tipo de espera; y, si se produce, sólo dura un tiempo corto. En los programas de aplicación, la situación es completamente diferente, porque sus secciones críticas pueden ser largas (minutos o incluso horas) o pueden estar casi siempre ocupadas. En tales casos, la espera activa resulta extremadamente ineficiente.

6.5.3 Interbloqueos e inanición

La implementación de un semáforo con una cola de espera puede dar lugar a una situación en la que dos o más procesos estén esperando indefinidamente a que se produzca un suceso que sólo puede producirse como consecuencia de las operaciones efectuadas por otro de los procesos en espera. El suceso en cuestión es la ejecución de una operación `signal()`. Cuando se llega a un estado así, se dice que estos procesos se han **interbloqueado**.

Para ilustrar el concepto, consideremos un sistema que consta de dos procesos, P_0 y P_1 , con acceso cada uno de ellos a dos semáforos, S y Q , configurados con el valor 1:

P_0	P_1
wait(S)	wait(Q);
wait(Q)	wait(S);
.	.
.	.
signal(S)	signal(Q);
signal(Q)	signal(S);

Suponga que P_0 ejecuta `wait(S)` y luego P_1 ejecuta `wait(Q)`. Cuando P_0 ejecuta `wait(Q)` tiene que esperar hasta que P_1 ejecute `signal(Q)`. De forma similar, cuando P_1 ejecuta `wait(S)` tiene que esperar hasta que P_0 ejecute `signal(S)`. Dado que estas operaciones `signal()` no pueden ejecutarse, P_0 y P_1 se interbloquean.

Decimos que un conjunto de procesos está en un estado de interbloqueo cuando todos los procesos del conjunto están esperando un suceso que sólo puede producirse como consecuencia de las acciones de otro proceso del conjunto. Los sucesos que más nos interesan aquí son los de *adquisición y liberación de recursos*, pero también hay otros tipos de sucesos que pueden dar lugar a interbloqueos, como veremos en el Capítulo 7. En ese capítulo describiremos varios mecanismos para tratar los problemas de interbloqueo.

Otro problema relacionado con los interbloqueos es el del **bloqueo indefinido** o muerte programada, una situación en la que algunos procesos esperan de forma indefinida dentro del semáforo. El bloqueo indefinido puede producirse si añadimos y eliminamos los procesos a la lista asociada con el semáforo utilizando un esquema LIFO (last-in, first-out).

6.5 Problemas clásicos de sincronización

En esta sección, vamos a presentar una serie de problemas de sincronización como ejemplos de una amplia clase de problemas de control de concurrencia. Estos problemas se utilizan para probar casi todos los nuevos esquemas de sincronización propuestos. En nuestras soluciones a los problemas, emplearemos semáforos para la sincronización.

6.5.1 Problema del búfer limitado

El problema del *buffer limitado* se ha presentado en la Sección 6.1; habitualmente se utiliza para ilustrar la potencia de las primitivas de sincronización. Vamos a presentar una estructura general de este esquema, sin comprometernos con ninguna implementación particular; en los ejercicios final del capítulo se proporciona un proyecto de programación relacionado con este problema.

Suponga que la cola consta de n búferes, siendo cada uno capaz de almacenar un elemento. El semáforo `mutex` proporciona exclusión mutua para los accesos al conjunto de búferes y se inicia

```

do {
    .
    .
    // produce un elemento en nextp
    .
    .
    wait(empty);
    wait(mutex);
    .
    .
    // añadir nextp al búfer
    .
    .
    signal(mutex);
    signal(full);
}while (TRUE);

```

Figura 6.10 Estructura del proceso productor.

```

do {
    wait(full);
    wait(mutex);

    . . .

    // eliminar un elemento del búfer a nextc
    . . .

    signal(mutex);
    signal(empty);

    . . .

    // consume el elemento en nextc
    . . .

}while (TRUE);

```

Figura 6.11 Estructura del proceso consumidor.

liza con el valor 1. Los semáforos `empty` y `full` cuentan el número de búferes vacíos y llenos. El semáforo `empty` se inicializa con el valor n ; el semáforo `full` se inicializa con el valor 0.

El código para el proceso productor se muestra en la Figura 6.10; el código para el proceso consumidor se presenta en la Figura 6.11. Observe la simetría entre el productor y el consumidor: podemos interpretar este código como un productor que genera los búferes llenos para el consumidor o como un consumidor que genera búferes vacíos para el productor.

6.6.2 El problema de los lectores-escritores

Imagine una base de datos que debe ser compartida por varios procesos concurrentes. Algunos de estos procesos pueden simplemente querer leer la base de datos, mientras que otros pueden desear actualizarla (es decir, leer y escribir). Diferenciamos entre estos dos tipos de procesos denominando a los primeros **lectores** y a los otros **escritores**. Obviamente, si dos lectores acceden simultáneamente a los datos compartidos, esto no dará lugar a ningún problema. Sin embargo, si un escritor y algún otro proceso (sea lector o escritor) acceden simultáneamente a la base de datos, el caos está asegurado.

Para asegurar que estos problemas no afloren, requerimos que los procesos escritores tengan acceso exclusivo a la base de datos compartida. Este problema de sincronización se denomina *problema de los lectores y escritores*. Desde que se lo estableció originalmente, este problema se ha utilizado para probar casi todas las nuevas primitivas de sincronización. El problema de los lectores y escritores presenta diversas variantes, todas las cuales utilizan prioridades. La más sencilla, conocida como *primer* problema de los lectores-escritores, requiere que ningún lector se mantenga en espera a menos que un escritor haya obtenido ya permiso para utilizar el objeto compartido. En otras palabras, ningún lector debe esperar a que otros lectores terminen simplemente porque un proceso escritor esté esperando. El *segundo* problema de los lectores-escritores requiere por el contrario que, una vez que un escritor está preparado, dicho escritor realice la escritura tan pronto como sea posible. Es decir, si un escritor está esperando para acceder al objeto, ningún proceso lector nuevo puede iniciar la lectura.

Una solución a cualquiera de estos problemas puede dar como resultado un problema de inanición. En el primer caso, los escritores pueden bloquearse indefinidamente; en el segundo caso, son los lectores los que pueden morir de inanición. Por esta razón, se han propuesto otras variantes del problema. En esta sección, vamos a presentar una solución sin bloqueos indefinidos para el primer problema de los lectores-escritores; consulte las notas bibliográficas incluidas al final del capítulo para ver referencias que describen soluciones sin bloqueos indefinidos para el segundo problema de los lectores-escritores.

En la solución del primer problema de los lectores-escritores, los procesos lectores comparten las siguientes estructuras de datos:

```

semaphore mutex, wr;
int readcount;

```

Los semáforos `mutex` y `wrt` se inicializan con el valor 1, mientras que `readcount` se inicializa con el valor 0. El semáforo `wrt` es común a los procesos lectores y escritores. El semáforo `mutex` se usa para asegurar la exclusión mutua mientras se actualiza la variable `readcount`. La variable `readcount` almacena el número de procesos que están leyendo actualmente el objeto. El semáforo `wrt` funciona como un semáforo de exclusión mutua para los escritores. También lo utilizan el primer o el último lector que entran o salen de la sección crítica. Los lectores que entran o salgan mientras otros procesos lectores estén en sus secciones críticas no lo utilizan.

El código para un proceso escritor se muestra en la Figura 6.12; el código para un proceso lector se muestra en la Figura 6.13. Observe que, si un escritor está en la sección crítica y n lectores están esperando, entonces un proceso lector estará en la cola de `wrt` y $n - 1$ lectores estarán en la cola de `mutex`. Observe también que, cuando un escritor ejecuta `signal(wrt)`, podemos reanudar la ejecución de todos los procesos lectores en espera o de un único proceso escritor en espera, la decisión le corresponderá al planificador.

En algunos sistemas, el problema de los procesos lectores y escritores y sus soluciones se han generalizado para proporcionar bloqueos **lector-escritor**. Adquirir un bloqueo lector-escritor requiere especificar el modo del bloqueo: acceso de *lectura* o de *escritura*. Cuando un proceso sólo desee leer los datos compartidos, solicitará un cerrojo lector-escritor en modo lectura. Un proceso que desee modificar los datos compartidos deberá solicitar el cerrojo en modo escritura. Se permite que múltiples procesos adquieran de forma concurrente un cerrojo lector-escritor en modo lectura, pero sólo un proceso puede adquirir el cerrojo en modo escritura, ya que se requiere acceso exclusivo por parte de los procesos escritores.

Los cerrojos lector-escritor resultan especialmente útiles en las situaciones siguientes:

- En aquellas aplicaciones en las que sea fácil identificar qué procesos sólo desean leer los datos compartidos y qué hebras sólo quieren escribir en los datos compartidos.

```
do {
    wait(wrt);
    .
    .
    // se realiza la escritura
    .
    .
    signal(wrt);
}while (TRUE);
```

Figura 6.12 Estructura de un proceso escritor.

```
do {
    .
    .
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);

    .
    .
    // se realiza la lectura
    .
    .
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);

}while (TRUE);
```

Figura 6.13 Estructura de un proceso lector.

- En aquellas aplicaciones que tengan más procesos lectores que escritores. Esto se debe a que, generalmente, establecer los bloqueos lector-escritor requiere más carga de trabajo que los bloqueos de exclusión mutua o los semáforos, y la carga de trabajo de configurar un cerrojo lector-escritor se compensa mediante el incremento en el grado de concurrencia que se obtiene al permitir el acceso por parte de múltiples lectores.

6.6.3 Problema de la cena de los filósofos

Considere cinco filósofos que gastan sus vidas en pensar y comer. Los filósofos comparten una mesa redonda con cinco sillas, una para cada filósofo. En el centro de la mesa hay una fuente de arroz y la mesa se ha puesto con sólo cinco palillos (Figura 6.14). Cuando un filósofo piensa, no se relaciona con sus colegas. De vez en cuando, un filósofo siente hambre y trata de tomar los palillos más próximos a él (los palillos que se encuentran entre él y sus vecinos de la izquierda y la derecha). Un filósofo sólo puede coger un palillo cada vez. Obviamente, no puede coger un palillo que haya cogido antes un vecino de mesa. Cuando un filósofo hambriento ha conseguido dos palillos, come sin soltar sus palillos. Cuando termina de comer, los coloca de nuevo sobre la mesa y vuelve a pensar.

El *problema de la cena de los filósofos* se considera un problema clásico de sincronización, no por su importancia práctica ni porque los informáticos tengan aversión a los filósofos, sino porque es un ejemplo de una amplia clase de problemas de control de concurrencia. Es una representación sencilla de la necesidad de repartir varios recursos entre varios procesos de una forma que no se produzcan interbloqueos ni bloqueos indefinidos.

Una solución sencilla consiste en representar cada palillo mediante un semáforo. Un filósofo intenta hacerse con un palillo ejecutando una operación `wait()` en dicho semáforo y libera sus palillos ejecutando la operación `signal()` en los semáforos adecuados. Por tanto, los datos compartidos son

```
semaphore palillo[5];
```

donde todos los elementos de `palillo` se inicializan con el valor 1. La estructura del filósofo *i* se muestra en la Figura 6.15.

Aunque esta solución garantiza que dos vecinos de mesa no coman nunca simultáneamente, debe rechazarse, porque puede crear interbloqueos. Supongamos que los cinco filósofos sienten hambre a la vez y cada uno toma el palillo situado a su izquierda. Ahora, todos los elementos de `palillo` tendrán el valor 0. Cuando los filósofos intenten coger el palillo de la derecha, tendrán que esperar eternamente.

A continuación se enumeran varias posibles soluciones para este problema de interbloqueo. En la Sección 6.7 presentaremos una solución para el problema de la cena de los filósofos que garantiza que no existan interbloqueos.

- Permitir que como máximo haya cuatro filósofos sentados a la mesa simultáneamente.

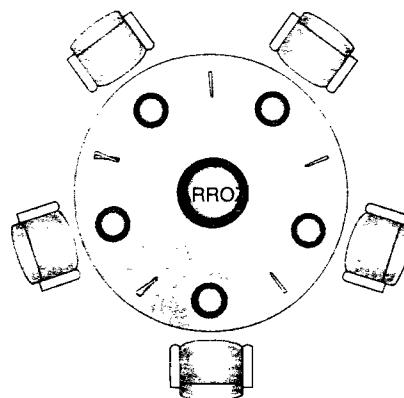


Figura 6.14 La cena de los filósofos.

```

do {
    wait(palillo[i]);
    wait(palillo [(i+1)% 5]);

    . . .
    // comer

    signal(palillo[i]);
    signal(palillo [(i+1)% 5]);

    . . .
    // pensar

}while (TRUE);

```

Figura 6.15 Estructura del filósofo *i*.

- Permitir a cada filósofo coger sus palillos sólo si ambos palillos están disponibles (para ello, deberá coger los palillos dentro de una sección crítica).
- Utilizar una solución asimétrica, es decir, un filósofo impar coge primero el palillo de su izquierda y luego el que está a su derecha, mientras que un filósofo par coge primero el palillo de su derecha y luego el de la izquierda.

Por último, toda solución satisfactoria al problema de la cena de los filósofos debe proteger de la posibilidad de que uno de los filósofos muera por inanición. Una solución libre de interbloqueos no necesariamente elimina la posibilidad de muerte por inanición.

6.7 Monitores

Aunque los semáforos proporcionan un mecanismo adecuado y efectivo para el proceso de sincronización, un uso incorrecto de los mismos puede dar lugar a errores de temporización que son difíciles de detectar, dado que estos errores sólo ocurren si tienen lugar algunas secuencias de ejecución concretas y estas secuencias no siempre se producen.

Hemos visto un ejemplo de dichos errores en el uso de contadores en la solución del problema productor-consumidor (Sección 6.1). En ese ejemplo, el problema de temporización se producía raras veces, e incluso entonces el valor del contador parecía ser razonable: lo que pasaba es que difería en 1 del valor correcto. Pero aunque el valor pareciera correcto, no era aceptable y es por esta razón que se introdujeron los semáforos.

Lamentablemente, estos errores de temporización pueden producirse también cuando se emplean semáforos. Para ilustrar cómo, revisemos la solución con semáforos para el problema de la sección crítica. Todos los procesos comparten una variable de semáforo `mutex`, que se inicializa con el valor 1. Cada proceso debe ejecutar una operación `wait(mutex)` antes de entrar en la sección crítica y una operación `signal(mutex)` después de la misma. Si esta secuencia no se lleva a cabo, dos procesos podrían estar dentro de sus secciones críticas al mismo tiempo. Examinemos los problemas a los que esto da lugar. Observe que estos problemas surgirán incluso aunque sólo sea un único proceso el que no se comporte de la forma adecuada; dicha situación puede deberse a un error de programación no intencionado o a que un cierto programador no tenga muchas ganas de cooperar.

- Suponga que un proceso intercambia el orden en el que se ejecutan las operaciones `wait()` y `signal()`, dando lugar a la siguiente secuencia de ejecución:

```

signal(mutex) ;
. . .
sección crítica
. . .
wait(mutex) ;

```

En esta situación, varios procesos pueden estar ejecutando sus secciones críticas simultáneamente, violando el requisito de exclusión mutua. Observe que este error sólo puede descubrirse si varios procesos están activos simultáneamente en sus secciones críticas y que esta situación no siempre se produce.

- Suponga que un proceso reemplaza `signal(mutex)` por `wait(mutex)`. Es decir, ejecuta

```

    wait(mutex);
    . . .
    sección crítica
    . . .
    wait(mutex);

```

En este caso, se producirá un interbloqueo.

- Suponga que un proceso omite la operación `wait(mutex)`, la operación `signal(mutex)`, o ambas. En este caso, se violará la exclusión mutua o se producirá un interbloqueo.

Estos ejemplos ilustran los distintos tipos de error que se pueden generar fácilmente cuando los programadores emplean incorrectamente los semáforos para solucionar el problema de la sección crítica. Pueden surgir problemas similares en los otros modelos de sincronización que hemos presentado en la Sección 6.6.

Para abordar tales errores, los investigadores han desarrollado estructuras de lenguaje de alto nivel. En esta sección, vamos a describir una estructura fundamental de sincronización de alto nivel, el tipo monitor.

6.7.1 Utilización

Un tipo, o un tipo abstracto de datos, agrupa una serie de datos privados con un conjunto de métodos públicos que se utilizan para operar sobre dichos datos. Un tipo monitor tiene un conjunto de operaciones definidas por el programador que gozan de la característica de exclusión mutua dentro del monitor. El tipo monitor también contiene la declaración de una serie de variables cuyos valores definen el estado de una instancia de dicho tipo, junto con los cuerpos de los procedimientos o funciones que operan sobre dichas variables. En la Figura 6.16 se muestra la sintaxis de un monitor. La representación de un tipo monitor no puede ser utilizada directamente por los diversos procesos. Así, un procedimiento definido dentro de un monitor sólo puede acceder a las variables declaradas localmente dentro del monitor y a sus parámetros formales. De forma similar, a las variables locales de un monitor sólo pueden acceder los procedimientos locales.

La estructura del monitor asegura que sólo un proceso esté activo cada vez dentro del monitor. En consecuencia, el programador no tiene que codificar explícitamente esta restricción de sincronización (Figura 6.17). Sin embargo, la estructura de monitor, como se ha definido hasta ahora, no es lo suficientemente potente como para modelar algunos esquemas de sincronización. Para ello, necesitamos definir mecanismos de sincronización adicionales. Estos mecanismos los proporciona la estructura `condition`. Un programador que necesite escribir un esquema de sincronización a medida puede definir una o más variables de tipo `condition`:

```
condition x, y;
```

Las únicas operaciones que se pueden invocar en una variable de condición son `wait()` y `signal()`. La operación

```
x.wait();
```

indica que el proceso que invoca esta operación queda suspendido hasta que otro proceso invoque la operación

```
x.signal();
```

La operación `x.signal()` hace que se reanude exactamente uno de los procesos suspendidos. Si no había ningún proceso suspendido, entonces la operación `signal()` no tiene efecto, es decir,

```

monitor nombre del monitor
{
    // declaraciones de variables compartidas
    procedimiento P1 ( . . . ) {
        .
        .
    }
    procedimiento P2 ( . . . ) {
        .
        .
    }

    .
    .
    .

    procedimiento Pn ( . . . ) {
        .
        .
    }
    código de inicialización ( . . . ) {
        .
        .
    }
}

```

Figura 6.16 Sintaxis de un monitor.

el estado de *x* será el mismo que si la operación nunca se hubiera ejecutado (Figura 6.18). Compare esta operación con la operación *signal()* asociada con los semáforos, que siempre afectaba al estado del semáforo.

Suponga ahora que, cuando un proceso invoca la operación *x.signal()*, hay un proceso *Q* en estado suspendido asociado con la condición *x*. Evidentemente, si se permite al proceso suspendido *Q* reanudar su ejecución, el proceso *P* que ha efectuado la señalización deberá esperar; en caso contrario, *P* y *Q* se activarían simultáneamente dentro del monitor. Sin embargo, observe que conceptualmente ambos procesos pueden continuar con su ejecución. Existen dos posibilidades:

1. **Señalar y esperar.** *P* espera hasta que *Q* salga del monitor o espere a que se produzca otra condición.

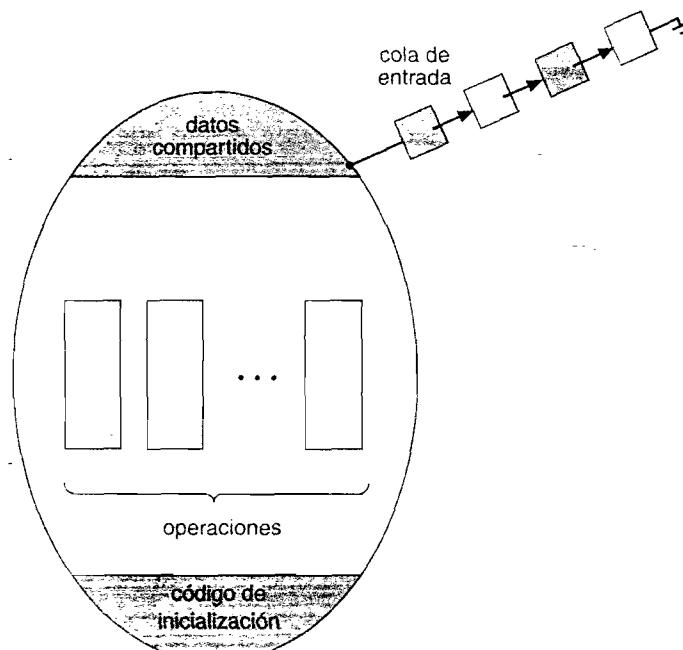


Figura 6.17 Vista esquemática de un monitor.

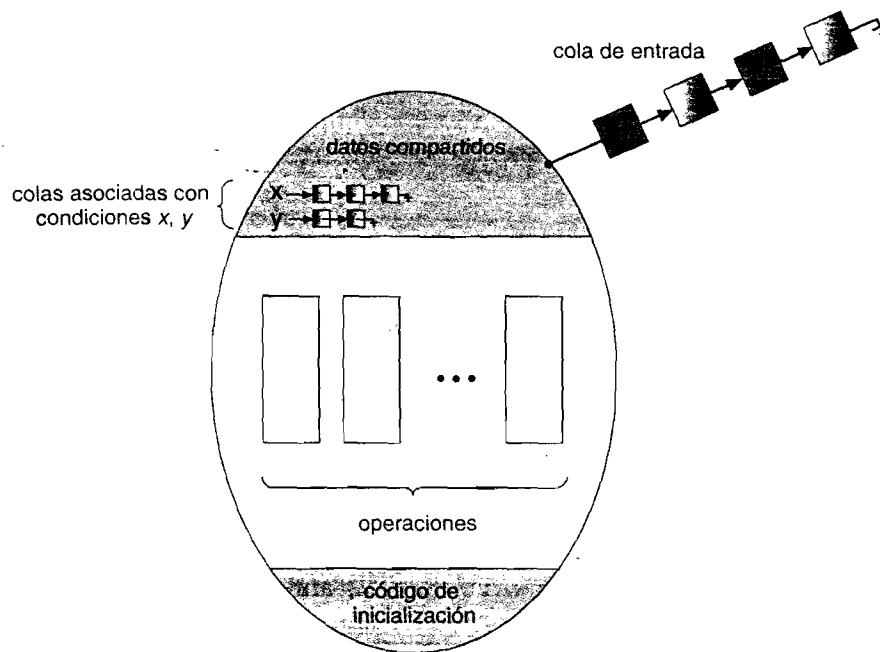


Figura 6.18 Monitor con variables de condición.

2. **Señalar y continuar.** Q espera hasta que P salga del monitor o espere a que se produzca otra condición.

Hay argumentos razonables en favor de adoptar cualquiera de estas opciones. Por un lado, puesto que P ya estaba ejecutándose en el monitor, el método de *señalar y continuar* parece el más razonable. Por otro lado, si permitimos que la hebra P continúe, para cuando se reanude la ejecución de Q , es posible que ya no se cumpla la condición lógica por la que Q estaba esperando. En el lenguaje Pascal Concurrente se adoptó un compromiso entre estas dos opciones: cuando la hebra P ejecuta la operación signal, sale inmediatamente del monitor. Por tanto, la ejecución de Q se reanuda de forma inmediata.

6.7.2 Solución al problema de la cena de los filósofos usando monitores

Vamos a ilustrar ahora el concepto de monitor presentando una solución libre de interbloqueos al problema de la cena de los filósofos. Esta solución impone la restricción de que un filósofo puede coger sus palillos sólo si ambos están disponibles. Para codificar esta solución, necesitamos diferenciar entre tres estados en los que puede hallarse un filósofo. Con este objetivo, introducimos la siguiente estructura de datos:

```
enum {pensar, hambre, comer} state[5];
```

El filósofo i puede configurar la variable $state[i] = \text{comer}$ sólo si sus dos vecinos de mesa no están comiendo: $(state[(i+4) \% 5] \neq \text{comer})$ y $(state[(i+1) \% 5] \neq \text{comer})$.

También tenemos que declarar

```
condition self[5];
```

donde el filósofo i tiene que esperar cuando tiene hambre pero no puede conseguir los palillos que necesita.

Ahora estamos en condiciones de describir nuestra solución al problema de la cena de los filósofos. La distribución de los palillos se controla mediante el monitor dp , cuya definición se muestra en la Figura 6.19. Cada filósofo, antes de empezar a comer, debe invocar la operación `pickup()`. Ésta puede dar lugar a la suspensión del proceso filósofo. Después de completar con éxito esta operación, el filósofo puede comer. A continuación, el filósofo invoca la operación

`putdown()`. Por tanto, el filósofo i tiene que invocar las operaciones `pickup()` y `putdown()` en la siguiente secuencia:

```
dp.pickup(i);
...
comer
...
dp.putdown(i);
```

Es fácil demostrar que esta solución asegura que nunca dos vecinos de mesa estarán comiendo simultáneamente y que no se producirán interbloqueos. Observe, sin embargo, que es posible que un filósofo se muera de hambre. No vamos a proporcionar aquí una solución para este problema; lo dejamos como ejercicio para el lector.

6.7.3 Implementación de un monitor utilizando semáforos

Consideremos ahora una posible implementación del mecanismo de monitor utilizando semáforos. Para cada monitor se proporciona un semáforo `mutex` inicializado con el valor 1. Un proceso debe ejecutar la operación `wait(mutex)` antes de entrar en el monitor y tiene que ejecutar la operación `signal(mutex)` después de salir del monitor.

```
monitor dp
{
    enum {PENSAR, HAMBRE, COMER} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = HAMBRE;
        test(i);
        if (state[i] != COMER)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = PENSAR;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != COMER) &&
            (state[i] == HAMBRE) &&
            (state[(i + 1) % 5] != COMER)) {
            state[i] = COMER;
            self[i].signal();
        }
    }
}

initialization code() {
    for (int i = 0; i < 5; i++)
        state[i] = PENSAR;
}
```

Figura 6.19 Solución empleando monitores para el problema de la cena de los filósofos.

Dado que un proceso que efectúe una operación de señalización debe esperar hasta que el proceso reanudado salga del monitor o quede en espera, se introduce un semáforo adicional, `next`, inicializado a 0, en el que los procesos que efectúen una señalización pueden quedarse suspendidos. También se proporciona una variable entera `next_count` para contar el número de procesos suspendidos en `next`. Así, cada procedimiento externo `F` se reemplaza por

```
wait(mutex);
...
cuerpo de F
...
if (next_count > 0)
    signal(next);
else
    signal(mutex);
```

La exclusión mutua dentro del monitor está asegurada.

Ahora podemos ver cómo se implementan las variables de condición. Para cada condición `x`, introducimos un semáforo `x_sem` y una variable entera `x_count`, ambos inicializados a 0. La operación `x.wait()` se puede implementar ahora como sigue

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

La operación `x.signal()` se puede implementar de la siguiente manera

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Esta implementación es aplicable a las definiciones de monitor dadas por Hoare y por Brinch-Hansen. Sin embargo, en algunos casos, la generalidad de la implementación es innecesaria y puede conseguirse mejorar significativamente la eficiencia. Dejamos este problema para el lector como Ejercicio 6.17.

6.7.4 Reanudación de procesos dentro de un monitor

Volvamos ahora al tema del orden de reanudación de los procesos dentro de un monitor. Si hay varios procesos suspendidos en la condición `x` y algún proceso ejecuta una operación `x.signal()`, ¿cómo determinamos cuál de los procesos en estado suspendido será el siguiente en reanudarse? Una solución sencilla consiste en usar el orden FCFS, de modo que el proceso que lleve más tiempo en espera se reanude en primer lugar. Sin embargo, en muchas circunstancias, un esquema de planificación tan simple no resulta adecuado. Puede utilizarse en este caso la estructura de **espera condicional**, que tiene la siguiente forma

```
x.wait(c);
```

donde `c` es una expresión entera que se evalúa cuando se ejecuta la operación `wait()`. El valor de `c`, que se denomina **número de prioridad**, se almacena entonces junto con el nombre del proceso suspendido. Cuando se ejecuta `x.signal()`, se reanuda el proceso que tenga asociado el número de prioridad más bajo.

Para ilustrar este nuevo mecanismo, considere el monitor `ResourceAllocator` mostrado en la Figura 6.20, que controla la asignación de un recurso entre varios procesos competidores. Cada proceso, al solicitar una asignación de este recurso, especifica el tiempo máximo durante el cual pretende usar dicho recurso. El monitor asigna el recurso al proceso cuya solicitud especifique un tiempo más corto. Un proceso que necesite acceder al recurso en cuestión deberá seguir el siguiente orden de secuencia:

```
R.acquire(t);
...
acceso al recurso;
...
R.release();
```

donde `R` es una instancia del tipo `ResourceAllocator`.

Lamentablemente, el concepto de monitor no puede garantizar que la secuencia de acceso anterior sea respetada. En concreto, pueden producirse los siguientes problemas:

- Un proceso podría acceder a un recurso sin haber obtenido primero el permiso de acceso al recurso.
- Un proceso podría no liberar nunca un recurso una vez que le hubiese sido concedido el permiso de acceso al recurso.
- Un proceso podría intentar liberar un recurso que nunca solicitó.
- Un proceso podría solicitar el mismo recurso dos veces (sin liberar primero el recurso).

Estos mismos problemas existían con el uso de semáforos y son de naturaleza similar a los que nos animaron a desarrollar las estructuras de monitor. Anteriormente, teníamos que preocuparnos por el correcto uso de los semáforos; ahora, debemos preocuparnos por el correcto uso de las operaciones de alto nivel definidas por el programador, para las que no podemos esperar ninguna ayuda por parte del compilador.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization code() {
        busy = FALSE;
    }
}
```

Figura 6.20 Un monitor para asignar un único recurso.

Una posible solución a este problema consiste en incluir las operaciones de acceso al recurso dentro del monitor ResourceAllocator. Sin embargo, el uso de esta solución significará que la planificación se realice de acuerdo con el algoritmo de planificación del monitor, en lugar de con el algoritmo que hayamos codificado.

Para asegurar que los procesos respeten las secuencias apropiadas, debemos inspeccionar todos los programas que usen el monitor ResourceAllocator y el recurso gestionado. Tenemos que comprobar dos condiciones para poder establecer la corrección de este sistema. En primer lugar, los procesos de usuario siempre deben realizar sus llamadas en el monitor en la secuencia correcta. En segundo lugar, tenemos que asegurarnos de que no haya ningún proceso no cooperativo que ignore simplemente el mecanismo de exclusión mutua proporcionado por el monitor e intente acceder directamente al recurso compartido sin utilizar los protocolos de acceso. Sólo si estas dos condiciones se cumplen, podemos garantizar que no se produzcan errores dependientes de la temporización y que el algoritmo de planificación no falle.

Aunque este tipo de inspección resulta posible en un sistema pequeño y estático, no es razonable para un sistema grande o dinámico. Este problema de control de acceso sólo puede resolverse mediante mecanismos adicionales que describiremos en el Capítulo 14.

Muchos lenguajes de programación han incorporado la idea de monitor descrita en esta sección, incluyendo Pascal Concurrente, Mesa, C# y Java. Otros lenguajes, como Erlang, proporcionan un cierto soporte de concurrencia usando un mecanismo similar.

MONITORES JAVA

Java proporciona un mecanismo de concurrencia de tipo monitor para la sincronización de hebras. Todo objeto de Java tiene asociado un único cerrojo. Cuando se declara un método como synchronized, la llamada al método requiere adquirir el bloqueo sobre el cerrojo. Declaramos un método como synchronized (sincronizado) incluyendo la palabra clave synchronized en la definición del método. Por ejemplo, el código siguiente define el método safeMethod() como synchronized:

```
public class SimpleClass {
    ...
    public synchronized void safeMethod() {
        /* Implementación de safeMethod() */
    }
}
```

A continuación creamos una instancia de SimpleClass como sigue:

```
SimpleClass sc = new SimpleClass();
```

Invocar el método sc.safeMethod() requiere adquirir el cerrojo sobre la instancia sc. Si el cerrojo ya lo posee otra hebra, la hebra que llama al método synchronized se bloquea y se coloca en el conjunto de entrada del cerrojo del objeto. El conjunto de entrada representa el conjunto de hebras que esperan a que el cerrojo esté disponible. Si el bloqueo está disponible cuando se llama al método synchronized, la hebra llamante pasa a ser la propietaria del cerrojo del objeto y puede entrar en el método. El cerrojo se libera cuando la hebra sale del método; entonces se selecciona una hebra del conjunto de entrada como nueva propietaria del cerrojo.

Java también proporciona métodos wait() y notify(), similares en funcionalidad a las instrucciones wait() y signal() en un monitor. La versión 1.5 de la máquina virtual Java proporciona una API para soporte de semáforos, variables de condición y bloqueos mútex (entre otros mecanismos de concurrencia) en el paquete java.util.concurrent.

6.8 Ejemplos de sincronización

A continuación se describen los mecanismos de sincronización proporcionados por los sistemas operativos Solaris, Windows XP y Linux, así como por la API Pthreads. Hemos elegido estos tres sistemas porque proporcionan buenos ejemplos de los diferentes métodos de sincronización del *kernel* y hemos incluido la API de Pthreads porque su uso está muy extendido entre los desarrolladores de UNIX y Linux para la creación y sincronización de hebras. Como veremos en esta sección, los métodos de sincronización disponibles en estos diferentes sistemas varían de forma sutil y significativa.

6.8.1 Sincronización en Solaris

Para controlar el acceso a las secciones críticas, Solaris proporciona mítex adaptativos, variables de condición, semáforos, bloqueos o cerrojos lector-escritor y colas de bloqueos (*turnstiles*). Solaris implementa los semáforos y las variables de condición como se ha explicado en las Secciones 6.5 y 6.7. En esta sección, vamos a describir los mítex adaptativos, los bloqueos lector-escritor y las colas de bloqueos.

Un **mítex adaptativo** protege el acceso a todos los elementos de datos críticos. En un sistema multiprocesador, un mítex adaptativo se inicia como un semáforo estándar, implementado como un cerrojo de bucle sin fin (*spinlock*). Si los datos están bloqueados, lo que quiere decir que ya están en uso, el mítex adaptativo hace una de dos cosas: si el cerrojo es mantenido por una hebra que se está ejecutando concurrentemente en otra CPU, la hebra actual ejecuta un bucle sin fin mientras espera a que el bloqueo esté disponible, dado que la hebra que mantiene el cerrojo probablemente vaya a terminar pronto; si la hebra que mantiene el cerrojo no está actualmente en estado de ejecución, la hebra actual se bloquea, pasando a estado durmiente hasta que la liberación del cerrojo la despierta. Se la pone en estado durmiente para que no ejecute un bucle sin fin mientras espera, dado que el cerrojo probablemente no se libere pronto. Los cerrojos mantenidos por hebras durmientes caen, probablemente, dentro de esta categoría. En un sistema monoprocesador, la hebra que mantiene el cerrojo nunca estará ejecutándose cuando otra hebra compruebe el cerrojo, ya que sólo puede haber una hebra ejecutándose cada vez. Por tanto, en este tipo de sistema, las hebras siempre pasan a estado durmiente, en lugar de entrar en un bucle sin fin, cuando encuentran un cerrojo.

Solaris utiliza el método del mítex adaptativo sólo para proteger aquellos datos a los que se accede mediante segmentos de código cortos. Es decir, se usa un mítex sólo si se va a mantener el bloqueo durante un máximo de unos cientos de instrucciones. Si el segmento de código es más largo, la solución basada en bucle sin fin será extremadamente ineficiente. Para estos segmentos de código largos, se usan las variables de condición y los semáforos. Si el cerrojo deseado ya está en uso, la hebra ejecuta una operación de espera y pasa al estado durmiente. Cuando otra hebra libere el cerrojo, ejecutará una operación `signal` dirigida a la siguiente hebra durmiente de la cola. El coste adicional de poner una hebra en estado durmiente y despertarla y de los cambios de contexto asociados es menor que el coste de malgastar varios cientos de instrucciones esperando en un bucle sin fin.

Los bloqueos lector-escritor se usan para proteger aquellos datos a los que se accede frecuentemente, pero que habitualmente se usan en modo de sólo lectura. En estas circunstancias, los bloqueos lector-escritor son más eficientes que los semáforos, dado que múltiples hebras pueden leer los datos de forma concurrente, mientras que los semáforos siempre serializan el acceso a los datos. Los bloqueos lector-escritor son relativamente caros de implementar, por lo que se usan sólo en secciones de código largas.

Solaris utiliza colas de bloqueos para ordenar la lista de hebras en espera de adquirir un mítex adaptativo o un cerrojo lector-escritor. Una **cola de bloqueos** (*turnstile*) es una estructura de cola que contiene las hebras que están a la espera de adquirir un cierto cerrojo. Por ejemplo, si una hebra posee actualmente el cerrojo para un objeto sincronizado, todas las restantes hebras que intenten adquirir el cerrojo se bloquearán y entrarán en la cola de dicho cerrojo. Cuando el cerrojo se libera, el *kernel* selecciona una hebra de la cola de bloqueo como nueva propietaria del cerrojo.

jo. Cada objeto sincronizado que tenga al menos una hebra esperando a adquirir el cerrojo del objeto requiere una cola de bloqueo propia. Sin embargo, en lugar de asociar una cola de bloqueo con cada objeto sincronizado, Solaris proporciona a cada hebra de *kernel* su propia cola de bloqueos. Dado que una hebra sólo puede estar bloqueada en un objeto cada vez, esta solución es más eficiente que tener una cola de bloqueo para cada objeto.

La cola de bloqueo correspondiente a la primera hebra que quede bloqueada por un objeto sincronizado se convierte en la cola de bloqueo del propio objeto. Las hebras subsiguientes se añadirán a esta cola de bloqueo. Cuando la hebra inicial libere finalmente el cerrojo, obtendrá una nueva cola de bloqueo de una lista de colas de bloqueo libres que mantiene el *kernel*. Para impedir una **inversión de prioridad**, las colas de bloqueo se organizan de acuerdo con un **protocolo de herencia de prioridad** (Sección 19.5). Esto significa que, si una hebra de baja prioridad mantiene un cerrojo que una hebra de prioridad más alta necesita, la hebra con la prioridad más baja heredará temporalmente la prioridad de la hebra con prioridad más alta. Después de liberar el cerrojo, la hebra volverá a su prioridad original.

Observe que los mecanismos de bloqueo utilizados por el *kernel* se implementan también para las hebras de nivel de usuario, de modo que los mismos tipos de cerrojo están disponibles fuera y dentro del *kernel*. Una diferencia de implementación fundamental es el protocolo de herencia de prioridad. Las rutinas de bloqueo del *kernel* se ajustan a los métodos de herencia de prioridades del *kernel* utilizados por el planificador, como se describe en la Sección 19.5; los mecanismos de bloqueo de hebras de nivel de usuario no proporcionan esta funcionalidad.

Para optimizar el rendimiento de Solaris, los desarrolladores han depurado y ajustado los métodos de bloqueo. Puesto que los cerros se usan frecuentemente, y como generalmente se emplean para funciones cruciales del *kernel*, optimizar su implementación y uso permite obtener importantes mejoras de rendimiento.

6.8.2 Sincronización en Windows XP

El sistema operativo Windows XP es un *kernel* multihebra que proporciona soporte para aplicaciones en tiempo real y múltiples procesadores. Cuando el *kernel* de Windows XP accede a un recurso global en un sistema monoprocesador, enmascara temporalmente las interrupciones de todas las rutinas de tratamiento de interrupción que puedan también acceder al recurso global. En un sistema multiprocesador, Windows XP protege el acceso a los recursos globales utilizando bloques basados en bucles sin fin. Al igual que en Solaris, el *kernel* usa los bucles sin fin sólo para proteger segmentos de código cortos. Además, por razones de eficiencia, el *kernel* asegura que una hebra nunca será desalojada mientras mantenga un cerrojo basado en bucle sin fin.

Para la sincronización de hebras fuera del *kernel*, Windows XP proporciona **objetos despachadores**. Utilizando un objeto despachador, las hebras se sincronizan empleando diferentes mecanismos, incluyendo mútex, semáforos, sucesos y temporizadores. El sistema protege los datos compartidos requiriendo que una hebra adquiera la propiedad de un mútex para acceder a los datos y libere dicha propiedad cuando termine. Los semáforos se comportan como se ha descrito en la Sección 6.5. Los **sucesos** son similares a las variables de condición; es decir, pueden notificar a una hebra en espera que una condición deseada se ha producido. Por último, los temporizadores se emplean para notificar a una (o más de una) hebra que ha transcurrido un determinado período de tiempo.

Los objetos despachadores pueden estar en estado señalizado o en estado no señalizado. Un **estado señalizado** indica que el objeto está disponible y que una hebra no se bloqueará cuando adquiera el objeto. Un **estado no señalizado** indica que el objeto no está disponible y que una hebra se bloqueará cuando intente adquirir el objeto. En la Figura 6.21 se ilustran las transiciones entre estados de un objeto despachador para un cerrojo mútex.

Existe una relación entre el estado de un objeto despachador y el estado de una hebra. Cuando una hebra se bloquea en un objeto despachador no señalizado, su estado cambia de preparado a en espera, y la hebra se pone en la cola de espera de dicho objeto. Cuando el estado del objeto despachador pasa a señalizado, el *kernel* comprueba si hay hebras en espera para ese objeto. En caso afirmativo, el *kernel* pasa una hebra, o posiblemente más de una hebra, del estado de espera al

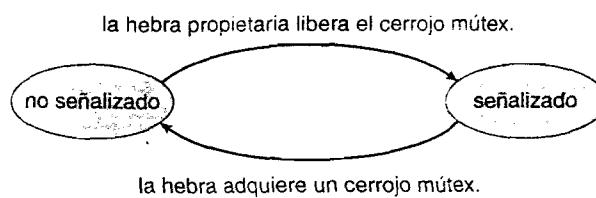


Figura 6.21 Objeto despachador para un mutex.

estado preparado, en el que pueden reanudar su ejecución. La cantidad de hebras que selecciona el *kernel* en la cola de espera dependerá del tipo de objeto despachador. El *kernel* sólo seleccionará una hebra de la cola de espera de un mutex, ya que un objeto mutex sólo puede ser poseído por una hebra. Con un objeto suceso, el *kernel* seleccionará todas las hebras que estén esperando a que se produzca dicho suceso.

Podemos usar un cerrojo mutex como ilustración de los estados de los objetos despachadores y las hebras. Si una hebra intenta adquirir un objeto despachador mutex que se encuentre en un estado no señalizado, dicha hebra se pondrá en estado suspendido y se colocará en la cola de espera del objeto mutex. Cuando el mutex pase al estado señalizado (porque otra hebra haya liberado el bloqueo sobre el mutex), la hebra en espera al principio de la cola pasará del estado de espera al de preparado y adquirirá el cerrojo mutex.

Al final del capítulo se proporciona un proyecto de programación que utiliza bloqueos mutex y semáforos con la API Win32.

6.8.3 Sincronización en Linux

Anteriormente a la versión 2.6, Linux era un *kernel* no apropiativo, lo que significa que un proceso que se ejecutase en modo *kernel* no podía ser desalojado, incluso aunque un proceso de mayor prioridad pasara a estar disponible para ejecución. Sin embargo, ahora el *kernel* de Linux es completamente un *kernel* apropiativo, de modo que una tarea puede ser desalojada aún cuando esté ejecutándose en el *kernel*.

El *kernel* de Linux proporciona bloqueos mediante bucles sin fin y semáforos (así como versiones lector-escritor de estos dos bloqueos) para establecer bloqueos en el *kernel*. En una máquina SMP, el mecanismo fundamental de bloqueo es el que se basa en el uso de bucles sin fin, y el *kernel* se diseña de modo que dicho tipo de bloqueo se mantenga sólo durante períodos de tiempo cortos. En las máquinas con un solo procesador, los bloqueos mediante bucle sin fin no resultan apropiados y se reemplazan por un mecanismo de activación y desactivación de la función de apropiación en el *kernel*. Es decir, en las máquinas monoprocesador, en lugar de mantener un bloqueo mediante un bucle sin fin, el *kernel* desactiva el mecanismo de apropiación, y el proceso de liberación del bloqueo se sustituye por una reactivación del mecanismo de apropiación. Esto se resume del siguiente modo:

un solo procesador	múltiples procesadores
Desactivar kernel apropiativo.	Adquirir bloqueo mediante bucle sin fin.
Activar kernel apropiativo.	Liberar bloqueo mediante bucle sin fin.

Linux utiliza un método interesante para activar y desactivar los mecanismos de desalojo del *kernel*: proporciona dos llamadas al sistema, `preempt_disable()` y `preempt_enable()`, para desactivar y activar los mecanismos de apropiación. Además, el *kernel* no es desalojable si hay una tarea en modo *kernel* que esté manteniendo un bloqueo. Para imponer esta característica, cada tarea del sistema tiene una estructura `thread_info` que contiene un contador, `preempt_count`, para indicar el número de bloqueos que dicha tarea está manteniendo. Cuando se adquiere un cerrojo, `preempt_count` se incrementa, y se decrementa cuando el bloqueado es

liberado. Si el valor de `preempt_count` para la tarea que está actualmente en ejecución es mayor que cero, no resultará seguro desalojar el *kernel*, ya que esa tarea mantiene un cerrojo. Si el contador es cero, el *kernel* puede ser interrumpido de forma segura (suponiendo que no haya llamadas pendientes a `preempt_disable()`).

Los bloqueos mediante bucle sin fin (así como el mecanismo de activación y desactivación del desalojo del *kernel*) se usan en el *kernel* sólo cuando el bloqueo (o la desactivación del desalojo) se mantiene durante un período de tiempo corto. Cuando sea necesario mantener un bloqueo durante un período de tiempo largo, resulta más apropiado utilizar semáforos.

6.8.4 Sincronización en Pthreads

La API de Pthreads proporciona cerrosos mútex, variables de condición y cerrosos de lectura-escritura para la sincronización de hebras. Esta API está disponible para los programadores y no forma parte de ningún *kernel*. Los cerrosos mútex representan la técnica fundamental de sincronización utilizada en Pthreads. Un cerrojo mútex se usa para proteger las secciones críticas de código, es decir, una hebra adquiere el cerrojo antes de entrar en una sección crítica y lo libera al salir de la misma. Las variables de condición en Pthreads se comportan como se ha descrito en la Sección 6.7. Los cerrosos de lectura-escritura se comportan de forma similar al mecanismo de bloqueo descrito en la Sección 6.6.2. Muchos sistemas que implementan Pthreads también proporcionan semáforos, aunque no forman parte del estándar Pthreads, sino que pertenecen a la extensión SEM de POSIX. Otras extensiones de la API de Pthreads incluyen cerrosos mediante bucle sin fin, aunque no todas las extensiones pueden portarse de una implementación a otra. Al final del capítulo se proporciona un proyecto de programación que usa cerrosos mútex y semáforos de Pthreads.

6.9 Transacciones atómicas

La exclusión mutua de sección críticas asegura que éstas se ejecuten atómicamente. Es decir, si dos secciones críticas se ejecutan de forma concurrente, el resultado es equivalente a su ejecución secuencial en algún orden desconocido. Aunque esta propiedad resulta útil en numerosos dominios de aplicación, en muchos casos desearíamos asegurarnos de que una sección crítica forme una sola unidad lógica de trabajo, que se ejecute por completo o que no se ejecute en absoluto. Un ejemplo sería una transferencia de fondos, en la que en una cuenta bancaria se anota un adeudo y en otra un abono. Evidentemente, es esencial para la coherencia de los datos que el adeudo y el abono se realicen ambos, o no se realice ninguno.

El problema de la coherencia de los datos, junto con el del almacenamiento y la recuperación de los mismos, tiene una gran importancia en los **sistemas de bases de datos**. Recientemente, ha resurgido el interés por el uso de las técnicas de los sistemas de bases de datos en los sistemas operativos. Los sistemas operativos pueden verse como manipuladores de datos; por tanto, pueden beneficiarse de las técnicas y modelos avanzados disponibles en las investigaciones sobre bases de datos. Por ejemplo, muchas de las técnicas *ad hoc* utilizadas en los sistemas operativos para gestionar los archivos podrían ser más flexibles y potentes si en su lugar se emplearan métodos más formales extraídos del campo de las bases de datos. En las Secciones 6.9.2 a 6.9.4 describimos algunas de estas técnicas de bases de datos y explicamos cómo pueden utilizarse en los sistemas operativos. No obstante, en primer lugar, vamos a ocuparnos del tema general de la atomicidad de transacciones. En esta propiedad se basan las técnicas utilizadas en las bases de datos.

6.9.1 Modelo del sistema

Una colección de instrucciones (u operaciones) que realiza una sola función lógica se denomina **transacción**. Una cuestión importante en el procesamiento de transacciones es la conservación de la atomicidad, incluso en el caso de que se produzcan fallos dentro del sistema informático.

Podemos pensar en una transacción como en una unidad de programa que accede a (y quizás actualiza) elementos de datos que residen en un disco, dentro de algunos determinados archivos. Desde nuestro punto de vista, una transacción es simplemente una secuencia de operaciones de

lectura (read) y escritura (write) que se termina bien con una operación de confirmación (commit) o una operación de cancelación (abort). Una operación commit significa que la transacción ha terminado su ejecución con éxito, mientras que una operación abort indica que la transacción no ha podido terminar su ejecución de la forma normal, porque se ha producido un error lógico o un fallo del sistema. Si una transacción terminada ha completado su ejecución con éxito, se ~~confirma~~; en caso contrario, se ~~aborta~~.

Dado que una transacción abortada puede haber modificado los datos a los que ha accedido, el estado de estos datos puede no ser el mismo que el que habrían tenido si la transacción se hubiera ejecutado atómicamente. Para poder garantizar la atomicidad, las transacciones abortadas no deben tener ningún efecto sobre el estado de los datos que ya haya modificado. Por tanto, el estado de los datos a los que haya accedido una transacción abortada deben restaurarse al estado en que estaban antes de que la transacción comenzara a ejecutarse. Decimos entonces que tal transacción ha sido ~~anulada~~. Es responsabilidad del sistema garantizar esta propiedad.

Para determinar cómo debe el sistema asegurar la atomicidad, necesitamos identificar en primer lugar las propiedades de los dispositivos utilizados para almacenar los datos a los que las transacciones acceden. Podemos clasificar los diversos tipos de medios de almacenamiento atendiendo a su velocidad, su capacidad y su resistencia a los fallos.

- **Almacenamiento volátil.** La información que reside en los dispositivos de almacenamiento volátiles normalmente no sobrevive a los fallos catastróficos del sistema. Como ejemplos de este tipo de medios de almacenamiento podemos citar la memoria principal y la memoria caché. El acceso a medios volátiles es extremadamente rápido, debido a la velocidad de acceso de la propia memoria y porque se puede acceder directamente a cualquier elemento de datos que se encuentre almacenado en dicho medio.
- **Almacenamiento no volátil.** La información que reside en los medios de almacenamiento no volátiles normalmente sobrevive a los fallos catastróficos del sistema. Como ejemplos de estos medios de almacenamiento podemos citar los discos y las cintas magnéticas. Los discos son más fiables que la memoria principal, pero menos que las cintas magnéticas. Sin embargo, tanto los discos como las cintas están sujetos a fallos, que pueden dar lugar a pérdidas de información. Actualmente, el almacenamiento no volátil es más lento que el volátil en varios órdenes de magnitud, porque los dispositivos de disco y de cinta son electromecánicos y requieren un movimiento físico para acceder a los datos.
- **Almacenamiento estable.** La información que reside en los medios de almacenamiento estables *nunca* se pierde (*nunca* no debe tomarse en sentido absoluto, ya que en teoría esos absolutos no pueden garantizarse). Para tratar de aproximarnos a las características de este tipo de almacenamiento, necesitamos replicar la información en varias cachés de almacenamiento no volátiles (normalmente, discos) con modos de fallo independientes y actualizar la información de una forma controlada (Sección 12.8).

Aquí sólo vamos a ocuparnos de asegurar la atomicidad de las transacciones en un entorno en el que los fallos provoquen una pérdida de la información contenida en los medios de almacenamiento volátil.

6.9.2 Recuperación basada en registro

Una forma de asegurar la atomicidad consiste en grabar en un dispositivo de almacenamiento estable la información que describa todas las modificaciones realizadas por la transacción en los distintos datos a los que haya accedido. El método más extendido para conseguir esta forma de protección es el **registro de escritura anticipada** (*write-ahead logging*). En este caso, el sistema mantiene en un medio de almacenamiento estable una estructura de datos denominada **registro**. Cada entrada del registro describe una única operación de escritura de la transacción y consta de los campos siguientes:

- **Nombre de la transacción.** El nombre único de la transacción que realizó la operación de escritura (write).

- **Nombre del elemento de datos.** El nombre único del elemento de datos escrito.
- **Valor antiguo.** El valor del elemento de datos antes de ejecutarse la operación de escritura.
- **Nuevo valor.** El valor que el elemento de datos tendrá después de la operación de escritura.

Existen otras entradas de registro especiales que permiten registrar los sucesos significativos producidos durante el procesamiento de la transacción, como por ejemplo el inicio de una transacción y la confirmación o cancelación de una transacción.

Antes de que una transacción i inicie su ejecución, se escribe la entrada $\langle T_i \text{ starts} \rangle$ en el registro. Durante su ejecución, cualquier operación `write` de T_i va *precedida* por la escritura de la correspondiente entrada nueva en el registro. Cuando T_i se confirma, se escribe la entrada $\langle T_i \text{ commits} \rangle$ en el registro.

Dado que la información contenida en el registro se utiliza para restaurar el estado de los elementos de datos a los que las distintas transacciones hayan accedido, no podemos permitir que la actualización de un elemento de datos tenga lugar antes de que se escriba la correspondiente entrada del registro en un medio de almacenamiento estable. Por tanto, es necesario que, antes de ejecutar una operación `write(X)`, las entradas del registro correspondientes a X se escriban en un medio de almacenamiento estable.

Observe el impacto sobre el rendimiento que este sistema tiene: son necesarias dos escrituras físicas por cada escritura lógica solicitada. También se necesita más almacenamiento, para los propios datos y para guardar los cambios en el registro. Pero en aquellos casos en los que los datos sean extremadamente importantes y sea necesario disponer de mecanismos rápidos de recuperación de fallos, la funcionalidad adicional que se obtiene bien merece el precio que hay que pagar por ella.

Usando el registro, el sistema puede corregir cualquier fallo que no dé lugar a perdida de información en un medio de almacenamiento no volátil. El algoritmo de recuperación utiliza dos procedimientos:

- `undo(T_i)`, que restaura el valor de todos los datos actualizados por la transacción T_i a sus antiguos valores.
- `redo(T_i)`, que asigna los nuevos valores a todos los datos actualizados por la transacción T_i .

En el registro pueden consultarse el conjunto de datos actualizados por T_i y sus respectivos valores antiguos y nuevos.

Las operaciones `undo` y `redo` deben ser idempotentes (es decir, múltiples ejecuciones deben dar el mismo resultado que una sola ejecución), para garantizar un comportamiento correcto incluso aunque se produzca un fallo durante el proceso de recuperación.

Si una transacción T_i se aborta, podemos restaurar el estado de los datos que se hayan actualizado simplemente ejecutando `undo(T_i)`. Si se produce un fallo del sistema, restauraremos el estado de todos los datos actualizados consultando el registro para determinar qué transacciones tienen que ser rehechas y cuáles deben deshacerse. Esta clasificación de las transacciones se hace del siguiente modo:

- La transacción T_i tiene que deshacerse si el registro contiene la entrada $\langle T_i \text{ starts} \rangle$ pero no contiene la entrada $\langle T_i \text{ commits} \rangle$.
- La transacción T_i tiene que rehacerse si el registro contiene tanto la entrada $\langle T_i \text{ starts} \rangle$ como la entrada $\langle T_i \text{ commits} \rangle$.

6.9.3 Puntos de comprobación

Cuando se produce un fallo del sistema, debemos consultar el registro para determinar aquellas transacciones que necesitan ser rehechas y aquellas que tienen que deshacerse. En principio, necesitamos revisar el registro completo para identificar todas las transacciones. Este método tiene, fundamentalmente, dos inconvenientes:

1. El proceso de búsqueda consume mucho tiempo.
2. La mayor parte de las transacciones que, de acuerdo con nuestro algoritmo, va a haber rehacer ya habrán actualizado los datos que el registro indica que hay que modificar. Aunque rehacer las modificaciones de los datos no causará ningún daño (gracias a la característica de idempotencia), sí que hará que el proceso de recuperación tarde más tiempo.

Para reducir esta carga de trabajo, introducimos el concepto de **puntos de comprobación**. Durante la ejecución, el sistema se encarga de mantener el registro de escritura anticipada. Además, el sistema realiza periódicamente puntos de comprobación, lo que requiere la siguiente secuencia de acciones:

1. Enviar todas las entradas del registro que actualmente se encuentren en un medio de almacenamiento volátil (normalmente en la memoria principal) a un medio de almacenamiento estable.
2. Enviar todos los datos modificados que residan en el medio de almacenamiento volátil a medio de almacenamiento estable.
3. Enviar una entrada de registro <checkpoint> al almacenamiento estable.

La presencia de una entrada <checkpoint> en el registro permite al sistema simplificar el procedimiento de recuperación. Considere una transacción T_i que se haya confirmado antes de un punto de comprobación. La entrada < T_i commits> aparecerá en el registro antes que la entrada <checkpoint>. Cualquier modificación realizada por T_i deberá haberse escrito en un medio de almacenamiento estable antes del punto de comprobación o como parte del propio punto de comprobación. Por tanto, en el momento de la recuperación, no hay necesidad de ejecutar una operación de rehacer (redo) sobre T_i .

Esta observación nos permite mejorar nuestro anterior algoritmo de recuperación. Después de producirse un fallo, la rutina de recuperación examina el registro para determinar la transacción T_i más reciente que haya iniciado su ejecución antes de que tuviera lugar el punto de comprobación. Dicha transacción se localiza buscando hacia atrás en el registro la primera entrada <checkpoint>, y localizando a continuación la siguiente entrada < T_i start>.

Una vez que se ha identificado la transacción T_i , las operaciones redo y undo se aplican sólo a la transacción T_i y a todas las transacciones T_j que hayan comenzado a ejecutarse después de T_i . Denominaremos a este conjunto de transacciones T . Podemos ignorar el resto del registro. Las operaciones de recuperación necesarias se llevan a cabo del siguiente modo:

- Para todas las transacciones T_k de T en las que aparezca la entrada < T_k commits> en el registro, ejecutar redo(T_k).
- Para todas las transacciones T_k de T que no tengan la entrada < T_k commits> en el registro, ejecutar undo(T_k).

6.9.4 Transacciones atómicas concurrentes

Hemos estado considerando un entorno en el que sólo puede ejecutarse una transacción cada vez. Ahora vamos a ver el caso en el que haya activas simultáneamente varias transacciones. Dado que cada transacción es atómica, la ejecución concurrente de transacciones debe ser equivalente al caso en que las transacciones se ejecuten en serie siguiendo un orden arbitrario. Esta propiedad, denominada **serialización**, se puede mantener simplemente ejecutando cada transacción dentro de una sección crítica. Es decir, todas las transacciones comparten un semáforo *mutex* común, inicializado con el valor 1. Cuando una transacción empieza a ejecutarse, su primera acción consiste en ejecutar la operación *wait* (*mutex*).

Aunque este esquema asegura la atomicidad de todas las transacciones que se estén ejecutando de forma concurrente, es bastante restrictivo. Como veremos, en muchos casos podemos permitir que las transacciones solapen sus operaciones, siempre que se mantenga la serialización. Para garantizar la serialización se utilizan distintos algoritmos de control de concurrencia, los cuales se describen a continuación.

6.9.4.1 Serialización

Considere un sistema con dos elementos de datos, A y B , que dos transacciones T_0 y T_1 leen y escriben. Suponga que estas dos transacciones se ejecutan atómicamente, primero T_0 seguida de T_1 . En la Figura 6.22 se representa esta secuencia de ejecución, la cual se denomina **planificación**. En la planificación 1 de la figura, la secuencia de instrucciones sigue un orden cronológico de arriba a abajo, estando las instrucciones de T_0 en la columna izquierda y las de T_1 en la columna de la derecha.

Una planificación en la que cada transacción se ejecuta atómicamente se denomina **planificación serie**. Una planificación serie consta de una secuencia de instrucciones correspondientes a varias transacciones, apareciendo juntas todas las instrucciones que pertenecen a una determinada transacción. Por tanto, para un conjunto de n transacciones, existen $n!$ planificaciones serie válidas diferentes. Cada planificación serie es correcta, ya que es equivalente a la ejecución atómica de las distintas transacciones participantes en un orden arbitrario.

Si permitimos que las dos transacciones solapen su ejecución, entonces la planificación resultante ya no será serie. Una **planificación no serie** no necesariamente implica una ejecución incorrecta, es decir, una ejecución que no sea equivalente a otra representada por una planificación serie. Para ver que esto es así, necesitamos definir el concepto de **operaciones conflictivas**.

Considere una planificación S en la que hay dos operaciones consecutivas O_i y O_j de las transacciones T_i y T_j , respectivamente. Decimos que O_i y O_j son *conflictivas* si acceden al mismo elemento de datos y al menos una de ellas es una operación de escritura. Para ilustrar el concepto de operaciones conflictivas, consideremos la planificación no serie 2 mostrada en la Figura 6.23. La operación `write(A)` de T_0 está en conflicto con la operación `read(A)` de T_1 . Sin embargo, la operación `write(A)` de T_1 no está en conflicto con la operación `read(B)` de T_0 , dado que cada operación accede a un elemento de datos diferente.

Sean O_i y O_j operaciones consecutivas de una planificación S . Si O_i y O_j son operaciones de transacciones diferentes y no están en conflicto, entonces podemos intercambiar el orden de O_i y

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figura 6.22 Planificación 1: una planificación serie en la que T_0 va seguida de T_1 .

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figura 6.23 Planificación 2: planificación serializable concurrente.

O_i para generar una nueva planificación S' . Cabe esperar que S y S' sean equivalentes, ya que todas las operaciones aparecen en el mismo orden en ambas planificaciones, excepto O_i y O_j , cuya orden no importa.

Podemos ilustrar la idea del intercambio considerando de nuevo la planificación 2 mostrada en la Figura 6.23. Como la operación `write(A)` de T_1 no entra en conflicto con la operación `read(B)` de T_0 , podemos intercambiar estas operaciones para generar una planificación equivalente. Independientemente del estado inicial del sistema, ambas planificaciones dan lugar al mismo estado final. Continuando con este procedimiento de intercambio de operaciones no conflictivas, tenemos:

- Intercambio de la operación `read(B)` de T_0 con la operación `read(A)` de T_1 .
- Intercambio de la operación `write(B)` de T_0 con la operación `write(A)` de T_1 .
- Intercambio de la operación `write(B)` de T_0 con la operación `read(A)` de T_1 .

El resultado final de estos intercambios es la planificación 1 de la Figura 6.22, que es una planificación serie. Por tanto, hemos demostrado que la planificación 2 es equivalente a una planificación serie. Este resultado implica que, independientemente del estado inicial del sistema, la planificación 2 producirá el mismo estado final que una planificación serie.

Si una planificación S se puede transformar en una planificación serie S' mediante un conjunto de intercambios de operaciones no conflictivas, decimos que la planificación S es **serializable con respecto a los conflictos**. Por tanto, la planificación 2 es serializable con respecto a los conflictos, dado que se puede transformar en la planificación serie 1.

6.9.4.2 Protocolo de bloqueo

Una forma de asegurar la serialización es asociando con cada elemento de datos un cerrojo y requiriendo que cada transacción siga un **protocolo de bloqueo** que gobierne el modo en que se adquieren y liberan los cerrojos. Hay disponibles varios modos para bloquear los elementos de datos. En esta sección, vamos a centrarnos en dos modos:

- **Compartido.** Si una transacción T_i ha obtenido un cerrojo en modo compartido (indicado por `S`) sobre un elemento de datos Q , entonces T_i puede leer ese elemento pero no puede escribir en él.
- **Exclusivo.** Si una transacción T_i ha obtenido un cerrojo en modo exclusivo (indicado por `X`) sobre un elemento de datos Q , entonces T_i puede leer y escribir en Q .

Es necesario que toda transacción solicite un bloqueo en el modo apropiado sobre el elemento de datos Q , dependiendo del tipo de operaciones que vaya a realizar sobre Q .

Para acceder al elemento de datos Q , la transacción T_i primero tiene que bloquear Q en el modo apropiado. Si Q no está bloqueado, el cerrojo se concede, y T_i puede acceder a dichos datos. Sin embargo, si el elemento de datos Q está bloqueado por alguna otra transacción, T_i puede tener que esperar. En concreto, supongamos que T_i solicita un cerrojo exclusivo sobre Q . En este caso, T_i tendrá que esperar hasta que el bloqueo sobre Q sea liberado. Si T_i solicita un cerrojo compartido sobre Q , entonces T_i tendrá que esperar si Q está bloqueado en modo exclusivo; en caso contrario, puede obtener el cerrojo y acceder a Q . Observe que este esquema es bastante similar al algoritmo de procesos lectores-escritores visto en la Sección 6.6.2.

Una transacción puede desbloquear un elemento de datos que hubiera bloqueado en algún momento anterior. Sin embargo, tiene que mantener un bloqueo sobre el elemento de datos mientras que esté accediendo a él. Además, no siempre es deseable para una transacción desbloquear un elemento de datos de forma inmediata después de su último acceso a dicho elemento de datos, ya que es posible que no pudiera asegurarse la serialización.

Un protocolo que asegura la serialización es el **protocolo de bloqueo en dos fases**. Este protocolo requiere que cada transacción ejecute las solicitudes de bloqueo y desbloqueo en dos fases:

- **Fase de crecimiento.** Durante esta fase, una transacción puede obtener cerrojos, pero no puede liberar ningún cerrojo.

- **Fase de contracción.** Durante esta fase, una transacción puede liberar cerros, pero no puede obtener ningún cerrojo nuevo.

Inicialmente, una transacción se encuentra en la fase de crecimiento y la transacción adquiere los cerros según los necesite. Una vez que la transacción libera un cerrojo, entra en la fase de contracción y no puede ejecutar más solicitudes de bloqueo.

El protocolo de bloqueo en dos fases asegura la serialización con respecto a los conflictos (Ejercicio 6.25); sin embargo, no asegura que no se produzcan interbloqueos. Además, es posible que, para un determinado conjunto de transacciones, existan planificaciones serializables con respecto a los conflictos que no puedan obtenerse mediante el uso del protocolo de bloqueo en dos fases. Sin embargo, para conseguir un rendimiento mejor que con el bloqueo en dos fases, necesitamos disponer de información adicional sobre las transacciones o imponer alguna estructura u ordenación al conjunto de datos.

6.9.4.3 Protocolos basados en marcas temporales

En los protocolos de bloqueo descritos anteriormente, el orden seguido por las parejas de transacciones conflictivas se determinaba en tiempo de ejecución, según el primer bloqueo que ambas transacciones solicitaran y que implicara modos incompatibles. Otro método para determinar el orden de serialización consiste en seleccionar un orden de antemano. El método más común para ello consiste en usar un esquema de ordenación basado en **marcas temporales**.

Con cada transacción T_i del sistema asociamos una marca temporal fija y exclusiva, indicada por $TS(T_i)$. El sistema asigna esta marca temporal antes de que la transacción T_i inicie su ejecución. Si una transacción T_i tiene asignada la marca temporal $TS(T_i)$ y después entra en el sistema una transacción T_j , entonces $TS(T_i) < TS(T_j)$. Hay disponibles dos métodos sencillos para implementar este esquema:

- Usar el valor del reloj del sistema como marca temporal; es decir, la marca temporal de una transacción es igual al valor que tenga el reloj cuando la transacción entra en el sistema. Este método no funcionará para las transacciones que se produzcan en sistemas diferentes o en procesadores que no comparten un reloj.
- Emplear un contador lógico como marca temporal; es decir, la marca temporal de una transacción es igual al valor del contador cuando la transacción entra en el sistema. El contador se incrementa después de asignar cada nueva marca temporal.

Las marcas temporales de las transacciones determinan el orden de serialización. Por tanto, si $TS(T_i) < TS(T_j)$, entonces el sistema debe asegurar que la planificación generada sea equivalente a la planificación serie en la que la transacción T_i aparece antes que la transacción T_j .

Para implementar este esquema, asociamos dos valores de marca temporal con cada elemento de datos Q :

- **W-timestamp(Q)** indica la marca temporal de mayor valor de cualquier transacción que ejecute $\text{write}(Q)$ con éxito.
- **R-timestamp(Q)** indica la marca temporal de mayor valor de cualquier transacción que ejecute $\text{read}(Q)$ con éxito.

Estas marcas temporales se actualizan cada vez que se ejecuta una nueva instrucción $\text{read}(Q)$ o $\text{write}(Q)$.

El protocolo de ordenación mediante marcas temporales asegura que cualesquiera operaciones read y write en conflicto se ejecuten siguiendo el orden de marca temporal. Este protocolo funciona del modo siguiente:

- Suponga que la transacción T_i ejecuta la instrucción $\text{read}(Q)$:
 - Si $TS(T_i) < W\text{-timestamp}()$ entonces T_i necesita leer un valor de Q que ya ha sido sobreescrito. Por tanto, la operación read es rechazada y T_i se anula.

T_2	T_3
read(B)	read(B) write(B)
read(A)	read(A) write(A)

Figura 6.24 Planificación 3: posible planificación utilizando el protocolo de marcas temporales.

- Si $TS(T_i) \geq W\text{-timestamp}()$ entonces la operación `read` se ejecuta y a $R\text{-timestamp}(Q)$ se asigna el máximo de $R\text{-timestamp}(Q)$ y $TS(T_i)$.
- Suponga que la transacción T_i ejecuta la instrucción `write(Q)`:
 - Si $TS(T_i) < R\text{-timestamp}()$, entonces el valor de Q que T_i está generando era necesario anteriormente y T_i supuso que este valor nunca se generaría. Por tanto, la operación de escritura se rechaza y la transacción T_i se anula.
 - Si $TS(T_i) < W\text{-timestamp}()$, entonces T_i está intentando escribir un valor obsoleto de Q . Por tanto, la operación de escritura se rechaza y T_i se anula.
 - En cualquier otro caso, se ejecuta la operación de escritura `write`.

A una transacción T_i que se anula como resultado de la ejecución de una operación de lectura o de escritura se le asigna una nueva marca temporal y se reinicia.

Para ilustrar este protocolo, considere la planificación 3 mostrada en la Figura 6.24, que incluye las transacciones T_2 y T_3 . Suponemos que a cada transacción se le asigna una marca temporal inmediatamente antes de su primera instrucción. Por tanto, en la planificación 3, $TS(T_2) < TS(T_3)$, y la planificación es posible usando el protocolo de marcas temporales.

Esta ejecución también puede generarse utilizando el protocolo de bloqueo en dos fases. Sin embargo, algunas planificaciones son posibles con el protocolo de bloqueo en dos fases pero no con el protocolo de marcas temporales, y viceversa.

El protocolo de marcas temporales asegura la serialización con respecto a los conflictos. Esta característica se sigue del hecho de que las operaciones conflictivas se procesan siguiendo el orden de las marcas temporales. El protocolo también asegura que no se produzcan interbloqueos, dado que ninguna transacción tiene nunca que esperar.

6.10 Resumen

Dada una colección de procesos secuenciales cooperativos que comparten datos, es necesario proporcionar mecanismos de exclusión mutua. Una solución consiste en garantizar que una sección crítica de código sólo sea utilizada por un proceso o hebra cada vez. Existen diferentes algoritmos para resolver el problema de la sección crítica, suponiendo que sólo haya disponibles bloqueos del almacenamiento.

La principal desventaja de estas soluciones codificadas por el programador es que todas ellas requieren una espera activa. Los semáforos permiten resolver este problema. Los semáforos se pueden emplear para solucionar varios problemas de sincronización y se pueden implementar de forma eficiente, especialmente si se dispone de soporte hardware para ejecutar las operaciones atómicamente.

Clásicamente, se han definido diversos problemas de sincronización (como el problema del búfer limitado, el problema de los procesos lectores-escritores, y el problema de la cena de los filósofos) que son importantes principalmente como ejemplos de una amplia clase de problemas de control de concurrencia. Estos problemas clásicos se utilizan para probar casi todos los nuevos esquemas de sincronización propuestos.

El sistema operativo debe proporcionar los medios de protección frente a los errores de temporización. Se han propuesto diversas estructuras para afrontar estos problemas. Los monitores proporcionan mecanismos de sincronización para compartir tipos abstractos de datos. Las variables de condición proporcionan un método mediante el que un procedimiento de un monitor puede bloquear su ejecución hasta recibir la señal de que puede continuar.

Los sistemas operativos también proporcionan soporte para la sincronización. Por ejemplo, Solaris, Windows XP y Linux proporcionan mecanismos como semáforos, mútex, bloqueos mediante bucles sin fin y variables de condición para controlar el acceso a datos compartidos. La API de Pthreads proporciona soporte para bloqueos mútex y variables de condición.

Una transacción es una unidad de programa que se debe ejecutar atómicamente; es decir, todas las operaciones asociadas con ella se ejecutan hasta completarse, o no se ejecuta ninguna de las operaciones. Para asegurar la atomicidad a pesar de los fallos del sistema, podemos usar un registro de escritura anticipada. Todas las actualizaciones se escriben en el registro, que se almacena en un medio de almacenamiento estable. Si se produce un fallo catastrófico del sistema, la información contenida en el registro se usa para restaurar el estado de los elementos de datos actualizados, lo que se consigue a través de las operaciones de deshacer (undo) y rehacer (redo). Para disminuir el trabajo de buscar en el registro después de haberse producido un fallo del sistema, podemos usar un mecanismo de puntos de comprobación.

Para asegurar la serialización cuando se solapa la ejecución de varias transacciones, debemos emplear un esquema de control de concurrencia. Hay diversos esquemas de control de concurrencia que aseguran la serialización retardando una operación o cancelando la transacción que ejecutó la operación. Los métodos más habitualmente utilizados son los protocolos de bloqueo y los esquemas de ordenación mediante marcas temporales.

Ejercicios

- 6.1 Dekker desarrolló la primera solución software correcta para el problema de la sección crítica para dos procesos. Los dos procesos, P_0 y P_1 , comparten las variables siguientes:

```
boolean flag[2]; /* inicialmente falso */
int turn;
```

La estructura del proceso P_i ($i == 0$ o 1) se muestra en la Figura 6.25; el otro proceso es P_j ($j == 1$ o 0). Demostrar que el algoritmo satisface los tres requisitos del problema de la sección crítica.

```
do {
    flag[i] = TRUE;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; // no hacer nada
            flag[i] = TRUE;
        }
    }
    // sección crítica

    turn = j;
    flag[i] = FALSE;

    // sección restante
}while (TRUE);
```

Figura 6.25 Estructura del proceso P_i en el algoritmo de Dekker.

- 6.2 Eisenberg y McGuire presentaron la primera solución software correcta para el problema de la sección crítica para n procesos, con un límite máximo de espera de $n - 1$ turnos. Los procesos comparten las siguientes variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

Todos los elementos de `flag` tienen inicialmente el valor `idle`; el valor inicial de `turn` es irrelevante (entre 0 y $n-1$). La estructura del proceso P_i se muestra en la Figura 6.26.

Demostrar que el algoritmo satisface los tres requisitos del problema de la sección crítica.

- 6.3 ¿Cuál es el significado del término *espera activa*? ¿Qué otros tipos de esperas existen en un sistema operativo? ¿Puede evitarse de alguna manera la espera activa? Explique su respuesta.
- 6.4 Explique por qué los bloqueos mediante bucle sin fin no son apropiados para sistemas monoprocesador, aunque se usen con frecuencia en los sistemas multiprocesador.

```
do {
    while (TRUE) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else
                j = (j + 1) % n;
        }

        flag[i] = in_cs;
        j = 0;

        while ((j < n) && (j == i || flag[j] != in_cs))
            j++;

        if ((j >= n) && (turn == i && flag[turn] == idle))
            break;
    }

    // sección crítica

    j = (turn - 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    // sección restante
}while (TRUE);
```

Figura 6.26 Estructura del proceso P_i en el algoritmo de Eisenberg y McGuire.

- 6.5 Explique por qué la implementación de primitivas de sincronización desactivando las interrupciones no resulta apropiada en un sistema monoprocesador si hay que usar las primitivas de sincronización en programas de nivel de usuario.
- 6.6 Explique por qué las interrupciones no son apropiadas para implementar primitivas de sincronización en los sistemas multiprocesador.
- 6.7 Describa cómo se puede utilizar la instrucción `swap()` para proporcionar un mecanismo de exclusión mutua que satisfaga el requisito de espera limitada.
- 6.8 Los servidores pueden diseñarse de modo que limiten el número de conexiones abiertas. Por ejemplo, un servidor puede autorizar sólo N conexiones simultáneas de tipo `socket`. En cuanto se establezcan las N conexiones, el servidor ya no aceptará otra conexión entrante hasta que una conexión existente se libere. Explique cómo puede utilizar un servidor los semáforos para limitar el número de conexiones concurrentes.
- 6.9 Demuestre que si las operaciones `wait()` y `signal()` de un semáforo no se ejecutan atómicamente, entonces se viola el principio de exclusión mutua.
- 6.10 Demuestre cómo implementar las operaciones `wait()` y `signal()` de un semáforo en entornos multiprocesador usando la instrucción `TestAndSet()`. La solución debe utilizar de modo mínimo los mecanismos de espera activa.
- 6.11 **El problema del barbero dormilón.** Una barbería tiene una sala de espera con n sillas y una sala para afeitado con una silla de barbero. Si no hay clientes a los que atender, el barbero se va a dormir. Si entra un cliente en la barbería y todas las sillas están ocupadas, entonces el cliente se va. Si el barbero está ocupado, pero hay sillas disponibles, el cliente se sienta en una de las sillas libres. Si el barbero está durmiendo, el cliente le despierta. Escriba un programa para coordinar al barbero y a los clientes.
- 6.12 Demuestre que los monitores y semáforos son equivalentes, en cuanto a que se pueden emplear para implementar los mismos tipos de problemas de sincronización.
- 6.13 Escriba un monitor de búfer limitado en el que los búferes (porciones) estén incluidos en el propio monitor,
- 6.14 La exclusión mutua estricta en un monitor hace que el monitor de búfer limitado del Ejercicio 6.13 sea más adecuado para porciones pequeñas.
- Explique por qué es cierto esto.
 - Diseñe un nuevo esquema que sea adecuado para porciones de mayor tamaño.
- 6.15 Indique los compromisos existentes entre una distribución equitativa de las operaciones y la tasa de procesamiento en el problema de los procesos lectores-escritores. Proponga un método para resolver este problema sin que pueda producirse el fenómeno de la inanición.
- 6.16 ¿En qué se diferencia la operación `signal()` asociada a los monitores de la correspondiente operación definida para los semáforos?
- 6.17 Suponga que la operación `signal()` sólo puede aparecer como última instrucción de un procedimiento de monitor. Sugiera cómo se puede simplificar la implementación descrita en la Sección 6.7.
- 6.18 Considere un sistema que consta de los procesos P_1, P_2, \dots, P_n , cada uno de los cuales tiene un número de prioridad único. Escriba un monitor que asigne tres impresoras idénticas a estos procesos usando los números de prioridad para decidir el orden de asignación.
- 6.19 Un archivo va a ser compartido por varios procesos, teniendo cada uno de ellos asociado un número de identificación único. Varios procesos pueden acceder simultáneamente al archivo, estando sujetos a la siguiente restricción: la suma de todos los identificadores asociados con los procesos que acceden al archivo tiene que ser menor que n . Escriba un monitor para coordinar el acceso al archivo.

- 6.20 Cuando se ejecuta una operación `signal` sobre una condición dentro de un monitor, el proceso que realiza la señalización puede continuar con su ejecución o transferir el control al proceso al que se dirige la señalización. ¿Cómo variaría la solución del ejercicio anterior con estas dos diferentes formas de realizar la señalización?
- 6.21 Suponga que reemplazamos las operaciones `wait()` y `signal()` de un monitor por una estructura `await(B)`, donde `B` es una expresión booleana general que hace que el proceso que ejecuta la instrucción espere hasta que `B` tome el valor `true`.
- Escriba un monitor utilizando este esquema para implementar el problema de los procesos lectores-escritores.
 - Explique por qué esta estructura no puede, en general, implementarse de manera eficiente.
 - ¿Qué restricciones hay que incluir en la instrucción `await` para poder implementarla de forma eficiente? (Consejo: restrinja la generalidad de `B`; consulte Kessels [1997])
- 6.22 Escriba un monitor que implemente un *reloj alarma* que permita al programa llamante retardarse un número específico de unidades de tiempo (*ticks*). Puede suponer que existe un reloj hardware real que invoca un procedimiento `tick` del monitor a intervalos regulares.
- 6.23 ¿Por qué Solaris, Linux y Windows 2000 utilizan bloqueos mediante bucle sin fin como mecanismo de sincronización sólo en los sistemas multiprocesador y no en los sistemas de un solo procesador?
- 6.24 En los sistemas basados en registro que proporcionan soporte para transacciones, la actualización de elementos de datos no se puede realizar antes de que las entradas correspondientes se hayan registrado. ¿Por qué es necesaria esta restricción?
- 6.25 Demuestre que son posibles algunas planificaciones con el protocolo de bloqueo en dos fases pero no con el protocolo de ordenación mediante marcas temporales, y viceversa.
- 6.26 ¿Cuáles son las implicaciones de asignar una nueva marca temporal a una transacción que se ha anulado? ¿Cómo procesa el sistema las transacciones que se han ejecutado después de una transacción anulada pero que tienen marcas temporales más pequeñas que la nueva marca temporal de la transacción anulada?
- 6.27 Suponga que existe un número finito de recursos de un mismo tipo que hay que gestionar. Los procesos pueden pedir una cantidad de estos recursos y, una vez que terminen con ellos, devolverlos. Por ejemplo, muchos paquetes comerciales de software operan usando un número fijo de licencias, que especifica el número de aplicaciones que se pueden ejecutar de forma concurrente. Cuando se inicia la aplicación, el contador de licencias se decremente. Si todas las licencias están en uso, las solicitudes para iniciar la aplicación se deniegan. Tales solicitudes sólo serán concedidas cuando el usuario de una de las licencias termine la aplicación y devuelva la licencia.

El siguiente segmento de programa se usa para gestionar un número finito de instancias de un recurso disponible. El número máximo de recursos y el número de recursos disponibles se declaran como sigue:

```
#define MAX_RESOURCES 5
int recursos_disponibles = MAX_RESOURCES;
```

Cuando un proceso desea obtener una serie de recursos, invoca la función `decrease_count()`:

```
/* decrementa recursos_disponibles en count recursos */
/* devuelve 0 si hay suficientes recursos disponibles, */
/* en caso contrario, devuelve -1 */
int decrease_count(int count) {
    if(recursos_disponibles < count)
```

```

        return -1;
    else {
        recursos_disponibles -= count;

        return 0;
    }
}

```

Cuando un proceso desea devolver una serie de recursos, llama a la función `increase_count()`:

```

/* incrementa recursos_disponibles en count */
int increase_count(int count) {
    recursos_disponibles += count;

    return 0;
}

```

El segmento de programa anterior da lugar a una condición de carrera. Haga lo siguiente:

- Identifique los datos implicados en la condición de carrera.
 - Identifique la ubicación (o ubicaciones) en el código donde se produce la condición de carrera.
 - Utilice un semáforo para resolver la condición de carrera.
- 6.28** La función `decrease_count()` del ejercicio anterior actualmente devuelve 0 si hay suficientes recursos disponibles y -1 en caso contrario. Esto lleva a un estilo de programación complicado para un proceso que deseé obtener una serie de recursos:

```

while (decrease_count(count) == -1)
;

```

Reescriba el segmento de código del gestor de recursos usando un monitor y variables de condición, de modo que la función `decrease_count()` suspenda el proceso hasta que haya disponibles suficientes recursos. Esto permitirá a un proceso invocar la función `decrease_count()` así:

```

decrease_count(count);

```

El proceso sólo volverá de esta llamada a función cuando haya suficientes recursos disponibles.

Proyecto: problema del productor-consumidor

En la Sección 6.6.1 hemos presentado una solución basada en semáforos para el problema de los procesos productor-consumidor usando un búfer limitado. Este proyecto va a diseñar una solución software para el problema del búfer limitado usando los procesos productor-consumidor mostrados en las figuras 6.10 y 6.11. La solución presentada en la Sección 6.6.1 utiliza tres semáforos: `empty` y `full`, que contabilizan el número de posiciones vacías y llenas en el búfer, y `mutex`, que es un semáforo binario (o de exclusión mutua) que protege la propia inserción o eliminación de elementos en el búfer. En este proyecto, se utilizarán para `empty` y `full` semáforos contadores estándar y, en lugar de un semáforo binario, se empleará un cerrojo mútex para representar a `mutex`. El productor y el consumidor, que se ejecutan como hebras diferentes, introducirán y eliminarán elementos en un búfer que se sincroniza mediante estas estructuras `empty`, `full` y `mutex`. Puede resolver este problema usando la API Pthreads o Win32.

El búfer

Internamente, el búfer consta de un array de tamaño fijo de tipo `buffer_item` (que se define usando `typedef`). El array de `buffer_item` objetos se manipulará como una cola circular. La definición de `buffer_item`, junto con el tamaño del búfer, puede almacenarse en el archivo de cabecera como sigue:

```
/* buffer.h */
typedef int buffer_item;
#define BUFFER_SIZE 5
```

El búfer se manipulará mediante dos funciones, `insert_item()` y `remove_item()`, a las que llaman los procesos productor y consumidor, respectivamente. Un esqueleto de estas funciones es el siguiente:

```
#include <buffer.h>

/* el búfer */
buffer_item buffer[BUFFER_SIZE];

int insert_item(buffer_item item) {
    /* insertar el elemento en el búfer
       devuelve 0 si se ejecuta con éxito, en caso contrario
       devuelve -1 indicando una condición de error */
}

int remove_item(buffer_item *item) {
    /* eliminar un objeto del búfer
       colocándolo en la variable item
       devuelve 0 si se ejecuta con éxito, en caso contrario
       devuelve -1 indicando una condición de error */
}
```

Las funciones `insert_item()` y `remove_item()` sincronizarán al productor y al consumidor usando los algoritmos descritos en las Figuras 6.10 y 6.11. El búfer también requerirá una función de inicialización que inicialice el objeto de exclusión mutua `mutex` junto con los semáforos `empty` y `full`.

La función `main()` inicializará el búfer y creará las hebras consumidora y productora. Una vez creadas las hebras consumidora y productora, la función `main()` dormirá durante un período de tiempo y, al despertar, terminará la aplicación. A la función `main()` se le pasan tres parámetros a través de la línea de comandos:

1. Cuánto tiempo dormirá antes de terminar.
2. El número de hebras productoras.
3. El número de hebras consumidoras.

Un esqueleto para esta función sería

```
#include <buffer.h>

int main(int argc, char *argv[]) {

    /* 1. Obtener argumentos de la línea de comandos argv[1], argv[2],
       /*      argv[3] */
       /* 2. Inicializar búfer */
       /* 3. Crear hebra(s) productora */
       /* 4. Crear hebra(s) consumidora */
```

```

    /* 5. Dormir */
    /* 6. Salir */
}

```

Hebras productora y consumidora

La hebra productora alternará entre dormir durante un período de tiempo aleatorio e insertar un entero aleatorio en el búfer. Los números aleatorios se generarán usando la función `rand()`, que genera enteros aleatorios entre 0 y `RAND_MAX`. El consumidor también dormirá durante un período de tiempo aleatorio y, al despertar, intentará eliminar un elemento del búfer. Un esquema de las hebras consumidora y productora sería el siguiente:

```

#include <stdlib.h> /* requerido para rand() */
#include <buffer.h>

void *producer(void *param) {
    buffer_item rand;

    while (TRUE) {
        /* dormir durante un período de tiempo aleatorio */
        sleep(...);
        /* generar un número aleatorio */
        rand = rand();
        printf("productor ha producido %f\n", rand);
        if (insert_item(rand))
            fprintf("informar de condición de error");
    }
}

void *consumer(void *param) {
    buffer_item rand;

    while (TRUE) {
        /* dormir durante un período de tiempo aleatorio */
        sleep(...);
        if (remove_item(&rand))
            fprintf("informar de condición de error ");
        else
            printf("consumidor ha consumido %f\n", rand);
    }
}

```

En las siguientes secciones vamos a ver detalles específicos de las implementaciones Pthreads y Win32.

Creación de hebras en Pthreads

En el Capítulo 4 se cubren los detalles sobre la creación de hebras usando la API de Pthreads. Consulte dicho capítulo para ver las instrucciones específicas para crear el productor y el consumidor usando Pthreads.

Bloqueos mútex de Pthreads

El siguiente ejemplo de código ilustra cómo se pueden usar en la API de Pthreads los bloqueos mútex para proteger una sección crítica:

```
#include <pthread.h>
pthread_mutex_t mutex;
```

```

/* crear el cerrojo mutex */
pthread_mutex_init(&mutex,NULL);

/* adquirir el cerrojo mutex */
pthread_mutex_lock(&mutex);

. . .

/* *** sección crítica ***

/* liberar el cerrojo mutex */
pthread_mutex_unlock(&mutex);

```

Pthreads utiliza el tipo de datos `pthread_mutex_t` para los bloqueos mútex. Un mútex se crea con la función `pthread_mutex_init(&mutex,NULL)`, siendo el primer parámetro un puntero al mútex. Pasando `NULL` como segundo parámetro, inicializamos el mútex con sus atributos predeterminados. El mútex se adquiere y libera con las funciones `pthread_mutex_lock()` y `pthread_mutex_unlock()`. Si el cerrojo mútex no está disponible cuando se invoca `pthread_mutex_lock()`, la hebra llamante se bloquea hasta que el propietario invoca la función `pthread_mutex_unlock()`. Todas las funciones mútex devuelven el valor `0` si se ejecutan correctamente; si se produce un error, estas funciones devuelven un código de error distinto de `0`.

Semáforos de Pthreads

Pthreads proporciona dos tipos de semáforos, nominados y no nominados. En este proyecto, usaremos semáforos no nominados. El código siguiente ilustra cómo se crea un semáforo:

```

#include <semaphore.h>
sem_t sem;

/* Crear el semáforo e inicializarlo en 5 */
sem_init(&sem, 0, 5);

```

La función `sem_init()` crea e inicializa un semáforo. A esta función se le pasan tres parámetros: (1) un puntero al semáforo, (2) un indicador que señala el nivel de compartición y (3) el valor inicial del semáforo. En este ejemplo, pasando el indicador `0`, estamos señalando que este semáforo sólo puede ser compartido entre hebras que pertenezcan al mismo proceso que creó el semáforo. Un valor distinto de cero permitiría que otros procesos accedieran también al semáforo. En este ejemplo hemos inicializado el semáforo con el valor `5`.

En la Sección 6.5 hemos descrito las operaciones clásicas `wait()` y `signal()` de los semáforos. En Pthreads, las operaciones `wait()` y `signal()` se denominan, respectivamente, `sem_wait()` y `sem_post()`. El ejemplo de código siguiente crea un semáforo binario mutex con un valor inicial de `1` e ilustra su uso para proteger una sección crítica:

```

#include <semaphore.h>
sem_t sem_mutex;

/* crear el semáforo */
sem_init(&sem_mutex, 0, 1);

/* adquirir el semáforo */
sem_wait(&sem_mutex);

. . .

/* *** sección crítica ***

/* liberar el semáforo */
sem_post(&sem_mutex);

```

Win32

Los detalles sobre la creación de hebras usando la API de Win32 están disponibles en el Capítulo 4. Consulte dicho capítulo para ver las instrucciones específicas.

Bloqueos mútex de Win32

Los bloqueos mútex son un tipo de objeto despachador, como se ha descrito en la Sección 6.8.2. A continuación se muestra cómo crear un cerrojo mútex usando la función `CreateMutex()`.

```
#include <windows.h>

HANDLE Mutex;
Mutex = CreateMutex(NULL, FALSE, NULL);
```

El primer parámetro hace referencia a un atributo de seguridad del cerrojo mútex. Estableciendo este parámetro como `NULL`, impediremos que cualquier hijo del proceso que crea el cerrojo mútex herede el descriptor del mútex. El segundo parámetro indica si el creador del mútex es el propietario inicial del cerrojo mútex. Pasar el valor `FALSE` indica que la hebra que crea el mútex no es el propietario inicial del mismo; en breve veremos cómo adquirir los bloqueos mútex. Por último, el tercer parámetro permite dar un nombre al mútex. Sin embargo, dado que hemos especificado el valor `NULL`, no damos un nombre al mútex en este caso. Si se ejecuta con éxito, `CreateMutex()` devuelve un descriptor `HANDLE` del cerrojo mútex; en caso contrario, devuelve `NULL`.

En la Sección 6.8.2 hemos clasificado los objetos despachadores como *señalizados* y *no señalizados*. Un objeto señalizado estará disponible para su adquisición; una vez que se adquiere un objeto despachador (por ejemplo, un cerrojo mútex), el objeto pasa al estado no señalizado. Cuando el objeto se libera, vuelve al estado señalizado.

Los cerrojos mútex se adquieren invocando la función `WaitForSingleObject()`, pasando a la función el descriptor `HANDLE` del cerrojo y un indicador para especificar cuánto tiempo se va a esperar. El siguiente código muestra cómo se puede adquirir el cerrojo mútex creado anteriormente:

```
WaitForSingleObject(Mutex, INFINITE);
```

El parámetro `INFINITE` indica que se esperará durante un período de tiempo infinito a que el cerrojo esté disponible. Se pueden usar otros valores que permiten que la hebra llameante deje de esperar si el cerrojo no pasa a estar disponible dentro de una franja de tiempo especificada. Si el cerrojo se encuentra en estado señalizado, `WaitForSingleObject()` vuelve inmediatamente y el cerrojo pasa a estado no señalizado. Un cerrojo se libera (pasa al estado señalizado) invocando `ReleaseMutex` como sigue:

```
ReleaseMutex(Mutex);
```

Semáforos de Win32

Los semáforos en la API de Win32 también son objetos despachadores y por tanto utilizan el mismo mecanismo de señalización que los bloqueos mútex. Los semáforos se crean de la forma siguiente:

```
#include <windows.h>

HANDLE Sem;
Sem = CreateSemaphore(NULL, 1, 5, NULL);
```

El primer y el último parámetros especifican un atributo de seguridad y el nombre del semáforo, de forma similar a como se ha descrito para los cerrojos mútex. El segundo y tercer parámetros especifican el valor inicial y el valor máximo del semáforo. En este caso, el valor inicial del semáforo es 1 y su valor máximo es 5. Si se ejecuta satisfactoriamente, `CreateSemaphore()` devuelve un descriptor, `HANDLE`, del cerrojo mútex; en caso contrario, devuelve `NULL`.

Los semáforos se adquieren usando la misma función `WaitForSingleObject()` que los bloques mútex. El semáforo `Sem` creado en este ejemplo se adquiere con la instrucción:

```
WaitForSingleObject(Semaphore, INFINITE);
```

Si el valor del semáforo es > 0 , el semáforo está en estado señalizado y por tanto será adquirido por la hebra llamante. En caso contrario, la hebra llamante se bloquea indefinidamente, ya que hemos especificado `INFINITE`, hasta que el semáforo pase al estado señalizado.

El equivalente de la operación `signal()` en los semáforos de Win32 es la función `ReleaseSemaphore()`. Se pasan tres parámetros a esta función: (1) el descriptor (`HANDLE`) del semáforo, (2) la cantidad en que se incrementa el valor del semáforo y (3) un puntero al valor anterior del semáforo. Podemos incrementar `Sem` en 1 con la siguiente instrucción:

```
ReleaseSemaphore(Sem, 1, NULL);
```

Tanto `ReleaseSemaphore()` como `ReleaseMutex()` devuelven 0 si se ejecutan con éxito; en caso contrario, devuelven un valor distinto de cero.

Notas bibliográficas

Los algoritmos de exclusión mutua fueron tratados por primera vez en los clásicos documentos de Dijkstra [1965]. Los algoritmos de Dekker (Ejercicio 6.1), la primera solución software correcta al problema de exclusión mutua de dos procesos, fue desarrollada por el matemático alemán T. Dekker. Este algoritmo también fue estudiado en Dijkstra [1965]. Una solución más sencilla al problema de exclusión mutua de dos procesos ha sido presentada por Peterson [1981] (Figura 6.2).

Dijkstra [1965b] presentó la primera solución al problema de la exclusión mutua para n procesos. Sin embargo, esta solución no establecía un límite superior para la cantidad de tiempo que un proceso tiene que esperar antes de que pueda entrar en la sección crítica. Knuth [1966] presentó el primer algoritmo con un límite; este límite era 2^n turnos. Una mejora del algoritmo de Knuth hecha por Debruijn [1967] redujo el tiempo de espera a n^2 turnos, después de lo cual Eisenberg [1972] (Ejercicio 6.4) consiguió reducir el tiempo al límite mínimo de $n-1$ turnos. Otro algoritmo que también requiere $n-1$ turnos pero más sencillo de programar y comprender, es el algoritmo de la panadería, que fue desarrollado por Lamport [1974]. Burns [1978] desarrolló el algoritmo de solución hardware que satisface el requisito de tiempo de espera limitado.

Explicaciones generales sobre el problema de exclusión mutua se proporcionan en Lamport [1986] y Lamport [1991]. Puede ver una colección de algoritmos para exclusión mutua en Raynal [1986].

El concepto de semáforo fue introducido por Dijkstra [1965a]. Patil [1971] examinó la cuestión de si los semáforos podrían resolver todos los posibles problemas de sincronización. Parnas [1975] estudió algunos de los defectos de los argumentos de Patil. Kosaraju [1973] continuó con el trabajo de Patil, enunciando un problema que no puede resolverse mediante las operaciones `wait()` y `signal()`. Lipton [1974] se ocupa de las limitaciones de distintas primitivas de sincronización.

Los problemas clásicos de coordinación de procesos que hemos descrito son paradigmas de una amplia clase de problemas de control de concurrencia. El problema del búfer limitado, el problema de la cena de los filósofos y el problema del barbero dormilón (Ejercicio 6.11) fueron sugeridos por Dijkstra [1965a] y Dijkstra [1971]. El problema de los lectores-escritores fue sugerido por Courtois [1971]. En Lamport [1977] se trata el tema de las lecturas y escrituras concurrentes. El problema de sincronización de procesos independientes se aborda en Lamport [1976].

El concepto de la región crítica fue sugerido por Hoare [1972] y Brinchhansen [1972]. El concepto de monitor fue desarrollado por Brinchhansen [1973]. Hoare [1974] proporciona una descripción completa de monitor. Kessels [1977] propuso una extensión del concepto de monitor para permitir la señalización automática. En Ben-Ari [1990] y Birrell [1989] se ofrece información general sobre la programación concurrente.

La optimización del rendimiento y el bloqueo de primitivas se han tratado en muchos trabajos, como Lamport [1987], Mellor-Crummey y Scott [1991] y Anderson [1990]. El uso de objetos compartidos que no requiere utilizar secciones críticas se ha estudiado en Herlihy [1993], Bershad

[1993] y Kopetz y Reisinger [1993]. En trabajos como Culler et al. [1998], Goodman et al. [1989], Barnes [1993] y Herlihy y Moss [1993] se han descrito nuevas instrucciones hardware y su utilización en la implementación de primitivas de sincronización.

Algunos detalles sobre los mecanismos de bloqueo utilizados en Solaris se presentan en Mauro y McDougall [2001]. Observe que los mecanismos de bloqueo utilizados por el *kernel* se implementan también para las hebras del nivel de usuario, por lo que los tipos de bloqueos están disponibles dentro y fuera del *kernel*. En Solomon y Russinovich puede encontrar detalles sobre la sincronización en Windows 2000.

El esquema de registro de escritura anticipada fue presentado por primera vez en System R de Gray et al. [1981]. El concepto de serialización fue formulado por Eswaran et al. [1976] en relación con su trabajo sobre control de concurrencia en System R. Eswaran et al. [1976] se ocupa del protocolo de bloqueo en dos fases. El esquema de control de concurrencia basado en marcas temporales se estudia en Reed [1983]. Una exposición sobre diversos algoritmos de control de concurrencia basados en marcas temporales es la presentada por Bernstein y Goodman [1980].

Interbloqueos

En un entorno de multiprogramación, varios procesos pueden competir por un número finito de recursos. Un proceso solicita recursos y, si los recursos no están disponibles en ese momento, el proceso pasa al estado de espera. Es posible que, algunas veces, un proceso en espera no pueda nunca cambiar de estado, porque los recursos que ha solicitado estén ocupados por otros procesos que a su vez estén esperando otros recursos. Cuando se produce una situación como ésta, se dice que ha ocurrido un **interbloqueo**. Hemos hablado brevemente de esta situación en el Capítulo 6, al estudiar el tema de los semáforos.

Quizá la mejor forma de ilustrar un interbloqueo es recurriendo a una ley aprobada a principios del siglo XX en Kansas, que decía: "Cuando dos trenes se aproximen a la vez a un cruce, ambos deben detenerse por completo y ninguno arrancará hasta que el otro haya salido".

En este capítulo, vamos a describir los métodos que un sistema operativo puede emplear para impedir o solventar los interbloqueos. La mayoría de los sistemas operativos actuales no proporcionan facilidades para la prevención de interbloqueos, aunque probablemente se añadan pronto dichos mecanismos. Los problemas de interbloqueo cada vez van a ser más habituales dadas las tendencias actuales: el gran número de procesos, el uso de programas multihebra, la existencia de muchos más recursos dentro de un sistema y la preferencia por servidores de archivos y bases de datos con transacciones de larga duración, en sustitución de los sistemas de procesamiento por lotes.

OBJETIVOS DEL CAPÍTULO

- Describir los interbloqueos, que impiden que un conjunto de procesos concurrentes completen sus tareas.
- Presentar una serie de métodos para prevenir o evitar los interbloqueos en un sistema informático.

7.1 Modelo de sistema

Un sistema consta de un número finito de recursos, que se distribuyen entre una serie de procesos en competición. Los recursos se dividen en varios tipos, constando cada uno de ellos de un cierto número de instancias. El espacio de memoria, los ciclos de CPU, los archivos y dispositivos de E/S (como impresoras y unidades de DVD) son ejemplos de tipos de recursos. Si un sistema tiene dos CPU, entonces el tipo de recurso *CPU* tiene dos instancias. De forma similar, el tipo de recurso *impresora* puede tener cinco instancias distintas.

Si un proceso solicita una instancia de un tipo de recurso, la asignación de *cualquier* instancia del tipo satisfará la solicitud. Si no es así, quiere decir que las instancias no son idénticas y, por tanto, los tipos de recursos no se han definido apropiadamente. Por ejemplo, un sistema puede

disponer de dos impresoras. Puede establecerse que estas dos impresoras pertenezcan a un mismo tipo de recurso si no importa qué impresora concreta imprime cada documento de salida. Sin embargo, si una impresora se encuentra en el noveno piso y otra en el sótano, el personal del noveno piso puede no considerar equivalentes ambas impresoras y puede ser necesario, por tanto, definir una clase de recurso distinta para cada impresora.

Un proceso debe solicitar cada recurso antes de utilizarlo y debe liberarlo después de usarlo. Un proceso puede solicitar tantos recursos como necesite para llevar a cabo las tareas que tiene asignadas. Obviamente, el número de recursos solicitados no puede exceder el total de recursos disponibles en el sistema: en otras palabras, un proceso no puede solicitar tres impresoras si el sistema sólo dispone de dos.

En modo de operación normal, un proceso puede emplear un recurso sólo siguiendo este secuencia:

1. **Solicitud.** Si la solicitud no puede ser concedida inmediatamente (por ejemplo, si el recurso está siendo utilizado por otro proceso), entonces el proceso solicitante tendrá que esperar hasta que pueda adquirir el recurso.
2. **Uso.** El proceso puede operar sobre el recurso (por ejemplo, si el recurso es una impresora, el proceso puede imprimir en ella).
3. **Liberación.** El proceso libera el recurso.

La solicitud y liberación de los recursos son llamadas al sistema; como se ha explicado en el Capítulo 2. Como ejemplos de llamadas al sistema tendríamos la solicitud y liberación de dispositivos [`request()` y `release()`], la apertura y cierre de archivos [`open()` y `close()`] y la asignación y liberación de memoria [`allocate()` y `free()`]. La solicitud y liberación de los recursos que el sistema operativo no gestiona puede hacerse mediante las operaciones `wait()` y `signal()` de los semáforos, o a través de la adquisición y liberación de un cerrojo mútex. Cada vez que un proceso o una hebra emplea un recurso gestionado por el *kernel*, el sistema operativo comprueba que el proceso ha solicitado el recurso y que éste ha sido asignado a dicho proceso. Una tabla del sistema registra si cada recurso está libre o ha sido asignado; para cada recurso asignado, la tabla también registra el proceso al que está asignado actualmente. Si un proceso solicita un recurso que en ese momento está asignado a otro proceso, puede añadirse a la cola de procesos en espera para ese recurso.

Un conjunto de procesos estará en un estado de interbloqueo cuando todos los procesos del conjunto estén esperando a que se produzca un suceso que sólo puede producirse como resultado de la actividad de otro proceso del conjunto. Los sucesos con los que fundamentalmente vamos a trabajar aquí son los de adquisición y liberación de recursos. El recurso puede ser un recurso físico (por ejemplo, una impresora, una unidad de cinta, espacio de memoria y ciclos de CPU) o un recurso lógico (por ejemplo, archivos, semáforos y monitores). Sin embargo, también otros tipos de sucesos pueden dar lugar a interbloqueos (por ejemplo, las facilidades IPC descritas en el Capítulo 3).

Para ilustrar el estado de interbloqueo, considere un sistema con tres unidades regrabables de CD. Suponga que cada proceso está usando una de estas unidades. Si ahora cada proceso solicita otra unidad, los tres procesos entrarán en estado de interbloqueo. Cada uno de ellos estará esperando a que se produzca un suceso, “la liberación de la unidad regrabable de CD”, que sólo puede producirse como resultado de una operación efectuada por uno de los otros procesos en espera. Este ejemplo ilustra un interbloqueo relativo a un único tipo de recurso.

Los interbloqueos también pueden implicar tipos de recursos diferentes. Por ejemplo, considere un sistema con una impresora y una unidad de DVD. Suponga que el proceso P_i está usando la unidad de DVD y el proceso P_j la impresora. Si P_i solicita la impresora y P_j solicita la unidad de DVD, se producirá un interbloqueo.

Los programadores que desarrollen aplicaciones multihebra deben prestar especial atención a este tipo de problemas. Los programas multihebra son buenos candidatos para los interbloqueos, porque las distintas hebras pueden competir por la utilización de los recursos compartidos.

7.2 Caracterización de los interbloqueos

En un interbloqueo, los procesos nunca terminan de ejecutarse y los recursos del sistema están ocupados, lo que impide que se inicien otros trabajos. Antes de pasar a ver los distintos métodos para abordar el problema de los interbloqueos, veamos en detalle sus características.

```

INTERBLOQUEO CON CERROJOS MUTEX

Véamos como ocurre el producirse un interbloqueo en un programa multihilo de Pthreads
con dos hilos mutex. La función main crea dos hilos, una hebra que intenta desbloquear el cerrojo mutex y la otra que lo bloquea. La primera hebra intenta desbloquear el cerrojo mutex y la segunda lo bloquea. Si una hebra intenta desbloquear el cerrojo mutex, la llamada a pthread_mutex_lock () bloquera la hebra hasta que el propietario del mutex libera el mismo con pthread_mutex_unlock () . Los cerrojos mutex se crean como se muestra en el siguiente ejemplo de código.

//Este es el código fuente de la aplicación. Los hilos se crean de la siguiente forma:
//Hilo 1: pthread_mutex_t *primer mutex;
//Hilo 2: pthread_mutex_t *segundo mutex;

//Código de la función main()
int main()
{
    //Crea dos hilos que realizan diferentes tareas
    //Hilo 1: pthread_t h1;
    //Hilo 2: pthread_t h2;

    //Crea el primer mutex
    pthread_mutex_init(&primer mutex, NULL);

    //Crea el segundo mutex
    pthread_mutex_init(&segundo mutex, NULL);

    //Inicia el hilo 1
    pthread_create(&h1, NULL, &realizar_tarea, &primer mutex);

    //Inicia el hilo 2
    pthread_create(&h2, NULL, &realizar_tarea, &segundo mutex);

    //Espera a que los hilos terminen
    pthread_join(h1, NULL);
    pthread_join(h2, NULL);

    //Libera el primer mutex
    pthread_mutex_destroy(&primer mutex);

    //Libera el segundo mutex
    pthread_mutex_destroy(&segundo mutex);

    //Termina el trabajo
    return 0;
}

//Realiza una tarea
void *realizar_tarea(void *arg)
{
    pthread_mutex_t *mutex = (pthread_mutex_t *)arg;

    //Hilo 1: Desbloquea el mutex
    pthread_mutex_unlock(mutex);

    //Hilo 2: Bloquea el mutex
    pthread_mutex_lock(mutex);

    //Realiza una tarea
    sleep(1);
}

//Función de salida
void salir()
{
    //Crea el primer mutex
    pthread_mutex_init(&primer mutex, NULL);

    //Crea el segundo mutex
    pthread_mutex_init(&segundo mutex, NULL);

    //Inicia el hilo 1
    pthread_create(&h1, NULL, &realizar_tarea, &primer mutex);

    //Inicia el hilo 2
    pthread_create(&h2, NULL, &realizar_tarea, &segundo mutex);

    //Espera a que los hilos terminen
    pthread_join(h1, NULL);
    pthread_join(h2, NULL);

    //Libera el primer mutex
    pthread_mutex_destroy(&primer mutex);

    //Libera el segundo mutex
    pthread_mutex_destroy(&segundo mutex);

    //Termina el trabajo
    exit(0);
}

```

Figura 7.1. Ejemplo de interbloqueo.

Continúa

INTERBLOQUEO CON CERROJOS MÚTEX (Cont.)

En este ejemplo, hebra uno intenta adquirir los cerrojos mutex en el orden (1) primer mutex, (2) segundo mutex mientras que hebra dos intenta adquirir en el orden (1) segundo mutex, (2) primer mutex. Se puede producir un interbloqueo si hebra uno adquiere primer mutex mientras hebra dos adquiere segundo mutex.

Observe que, incluso aunque se pueda producir un interbloqueo, éste no ocurrirá si hebra uno puede adquirir y liberar los bloques mutex para primer mutex y segundo mutex antes de que hebra dos intente adquirir los cerrojos. Este ejemplo ilustra uno de los principales problemas en el tratamiento de los interbloqueos: resulta difícil identificar y comprobar la existencia de interbloqueos que solo pueden ocurrir bajo determinadas circunstancias.

7.2.1 Condiciones necesarias

Una situación de interbloqueo puede surgir si se dan simultáneamente las cuatro condiciones siguientes en un sistema:

1. **Exclusión mutua.** Al menos un recurso debe estar en modo no compartido; es decir, sólo un proceso puede usarlo cada vez. Si otro proceso solicita el recurso, el proceso solicitante tendrá que esperar hasta que el recurso sea liberado.
2. **Retención y espera.** Un proceso debe estar reteniendo al menos un recurso y esperando para adquirir otros recursos adicionales que actualmente estén retenidos por otros procesos.
3. **Sin desalojo.** Los recursos no pueden ser desalojados; es decir, un recurso sólo puede ser liberado voluntariamente por el proceso que le retiene, después de que dicho proceso haya completado su tarea.
4. **Espera circular.** Debe existir un conjunto $\{P_0, P_1, \dots, P_n\}$ de procesos en espera, tal que P_0 esté esperando a un recurso retenido por P_1 , P_1 esté esperando a un recurso retenido por P_2 , ..., P_{n-1} esté esperando a un recurso retenido por P_n , y P_n esté esperando a un recurso retenido por P_0 .

Es sistemas en que deben darse estas cuatro condiciones para que se produzca un interbloqueo. La condición de espera circular implica necesariamente la condición de retención y espera, por lo que las cuatro condiciones no son completamente independientes. Sin embargo, veremos en la sección 7.4 que resulta útil considerar cada condición por separado.

7.2.2 Grafo de asignación de recursos

Los interbloqueos pueden definirse de forma más precisa mediante un grafo dirigido, que se llama **grafo de asignación de recursos del sistema**. Este grafo consta de un conjunto de vértices V y de un conjunto de aristas E . El conjunto de vértices V se divide en dos tipos diferentes de nodos: $P = \{P_1, P_2, \dots, P_n\}$, el conjunto formado por todos los procesos activos del sistema, y $R = \{R_1, R_2, \dots, R_m\}$, el conjunto formado por todos los tipos de recursos del sistema.

Una arista dirigida desde el proceso P_i al tipo de recurso R_j se indica mediante $P_i \rightarrow R_j$ y significa que el proceso P_i ha solicitado una instancia del tipo de recurso R_j y que actualmente está esperando dicho recurso. Una arista dirigida desde el tipo de recurso R_j al proceso P_i se indica mediante $R_j \rightarrow P_i$ y quiere decir que se ha asignado una instancia del tipo de recurso R_j al proceso P_i . Una arista dirigida $P_i \rightarrow R_j$ se denomina **arista de solicitud**, mientras que una arista dirigida $R_j \rightarrow P_i$ se denomina **arista de asignación**.

Gráficamente, representamos cada proceso P_i con un círculo y cada tipo de recurso R_j con un rectángulo. Puesto que el tipo de recurso R_j puede tener más de una instancia, representamos cada

instancia mediante un punto dentro del rectángulo. Observe que una arista de solicitud sólo apunta a un rectángulo R_j , mientras que una arista de asignación también debe asociarse con uno de los puntos contenidos en el rectángulo.

Cuando el proceso P_i solicita una instancia del tipo de recurso R_j , se inserta una arista de solicitud en el grafo de asignación de recursos. Cuando esta solicitud se concede, la arista de solicitud se transforma *instantáneamente* en una arista de asignación. Cuando el proceso ya no necesita acceder al recurso, éste se libera y la arista de asignación se borra.

La Figura 7.2 muestra el grafo de asignación de recursos que ilustra la situación siguiente:

- Conjuntos P , R y E :

- $P = \{P_1, P_2, P_3\}$
- $R = \{R_1, R_2, R_3, R_4\}$
- $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$

- Instancias de recursos

- Una instancia del tipo de recurso R_1
- Dos instancias del tipo de recurso R_2
- Una instancia del tipo de recurso R_3
- Tres instancias del tipo de recurso R_4

- Estados de los procesos

- El proceso P_1 retiene una instancia del tipo de recurso R_2 y está esperando una instancia del recurso R_1 .
- El proceso P_2 retiene una instancia del tipo de recurso R_1 y está esperando una instancia del recurso R_3 .
- El proceso P_3 retiene una instancia del recurso R_3 .

Dada la definición de un grafo de asignación de recursos, podemos demostrar que, si el grafo no contiene ningún ciclo, entonces ningún proceso del sistema está interbloqueado. Si el grafo contiene un ciclo, entonces puede existir un interbloqueo.

Si cada tipo de recurso tiene exactamente una instancia, entonces la existencia de un ciclo implica necesariamente que se ha producido un interbloqueo. Además, aunque haya tipos de recursos con más de una instancia, si el ciclo implica sólo a un determinado conjunto de tipos de recursos, cada uno de ellos con una sola instancia, entonces existirá un interbloqueo: cada uno de los procesos implicados en el ciclo estará interbloqueado. En este caso, la presencia de un ciclo en el grafo es condición necesaria y suficiente para la existencia de interbloqueo.

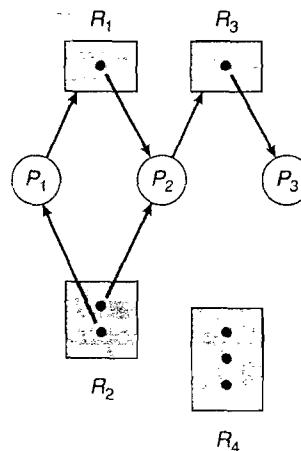


Figura 7.2 Grafo de asignación de recursos.

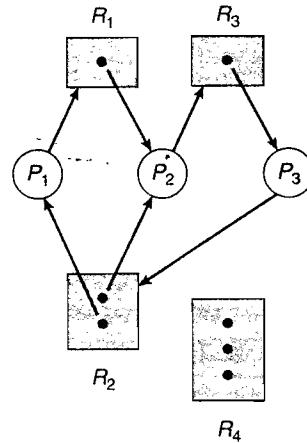


Figura 7.3 Grafo de asignación de recursos con un interbloqueo.

Si cada tipo de recurso tiene varias instancias, entonces la existencia de un ciclo no necesariamente implica que se haya producido un interbloqueo. En este caso, la existencia de un ciclo en el grafo es condición necesaria, pero no suficiente, para la existencia de interbloqueo.

Para ilustrar este concepto, volvamos al grafo de asignación de recursos de la Figura 7.2. Supongamos que el proceso P_3 solicita una instancia del tipo de recurso R_2 . Dado que actualmente no hay disponible ninguna instancia, se añade al grafo una arista de solicitud $P_3 \rightarrow R_2$ (Figura 7.3). En esta situación, en el sistema existen dos ciclos como mínimo:

$$\begin{aligned} P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

Los procesos P_1 , P_2 y P_3 se interbloquean. El proceso P_2 está esperando para acceder al recurso R_3 , que está retenido por el proceso P_3 . El proceso P_3 está esperando a que P_1 o P_2 liberen el recurso R_2 . Además, el proceso P_1 está esperando a que el proceso P_2 libere el recurso R_1 .

Ahora consideremos el grafo de asignación de recursos de la Figura 7.4. En este ejemplo, tenemos también un ciclo

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

Sin embargo, no existe ningún interbloqueo. Observe que el proceso P_4 puede liberar su instancia del tipo de recurso R_2 . Dicho recurso se puede asignar a P_3 , rompiendo así el ciclo.

En resumen, si un grafo de asignación de recursos no tiene un ciclo, entonces el sistema *no* está en estado de interbloqueo. Si existe un ciclo, entonces el sistema puede o no estar en esta-

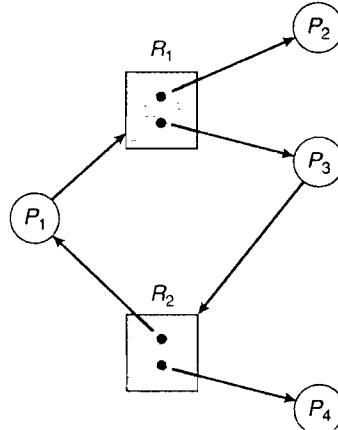


Figura 7.4 Grafo de asignación de recursos con un ciclo pero sin interbloqueo.

do de interbloqueo. Esta observación es importante a la hora de tratar el problema de los interbloqueos.

7.3 Métodos para tratar los interbloqueos

En general, podemos abordar el problema de los interbloqueos de una de tres formas:

- Podemos emplear un protocolo para impedir o evitar los interbloqueos, asegurando que el sistema *nunca* entre en estado de interbloqueo.
- Podemos permitir que el sistema entre en estado de interbloqueo, detectarlo y realizar una recuperación.
- Podemos ignorar el problema y actuar como si nunca se produjeran interbloqueos en el sistema.

La tercera solución es la que usan la mayoría de los sistemas operativos, incluyendo UNIX y Windows; entonces, es problema del desarrollador de aplicaciones el escribir programas que resuelvan posibles interbloqueos.

A continuación, vamos a exponer brevemente cada uno de los tres métodos mencionados; más adelante, en las Secciones 7.4 a 7.7, presentaremos algoritmos detallados. Sin embargo, antes de continuar, debemos mencionar que algunos investigadores han argumentado que ninguno de los métodos básicos resulta apropiado, por sí solo, para abordar el espectro completo de problemas relativos a la asignación de recursos en los sistemas operativos. No obstante, estos métodos básicos se pueden combinar, permitiéndonos seleccionar un método óptimo para cada clase de recurso existente en el sistema.

Para garantizar que nunca se produzcan interbloqueos, el sistema puede emplear un esquema de prevención de interbloqueos o un esquema de evasión de interbloqueos. La **prevención de interbloqueos** proporciona un conjunto de métodos para asegurar que al menos una de las condiciones necesarias (Sección 7.2.1) no pueda cumplirse. Estos métodos evitan los interbloqueos restringiendo el modo en que se pueden realizar las solicitudes. Veremos estos métodos en la Sección 7.4.

La **evasión de interbloqueos** requiere que se proporcione de antemano al sistema operativo información adicional sobre qué recursos solicitará y utilizará un proceso durante su tiempo de vida. Con estos datos adicionales, se puede decidir, para cada solicitud, si el proceso tiene que esperar o no. Para decidir si la solicitud actual puede satisfacerse o debe retardarse, el sistema necesita considerar qué recursos hay disponibles en ese momento, qué recursos están asignados a cada proceso y las futuras solicitudes y liberaciones de cada proceso. Explicaremos estos esquemas en la Sección 7.5.

Si un sistema no emplea un algoritmo de prevención de interbloqueos ni un algoritmo de evasión, entonces puede producirse una situación de interbloqueo. En este tipo de entorno, el sistema puede proporcionar un algoritmo que examine el estado del mismo para determinar si se ha producido un interbloqueo y otro algoritmo para recuperarse de dicho interbloqueo (si realmente se ha producido). Veremos estas cuestiones en las Secciones 7.6 y 7.7.

Si un sistema no garantiza que nunca se producirá un interbloqueo, ni proporciona un mecanismo para la detección y recuperación de interbloqueos, entonces puede darse la situación de que el sistema esté en estado de interbloqueo y no tenga forma ni siquiera de saberlo. En este caso, el interbloqueo no detectado dará lugar a un deterioro del rendimiento del sistema, ya que habrá recursos retenidos por procesos que no pueden ejecutarse y, a medida que se realicen nuevas solicitudes de recursos, cada vez más procesos entrarán en estado de interbloqueo. Finalmente, el sistema dejará de funcionar y tendrá que reiniciarse manualmente.

Aunque este método no parece un sistema viable para abordar el problema del interbloqueo, se usa en la mayoría de los sistemas operativos, como hemos señalado anteriormente. En muchos sistemas, los interbloqueos se producen de forma bastante infrecuente (por ejemplo, una vez al año); por tanto, este método es más barato que los métodos de prevención, de evasión o de detección y recuperación, que deben utilizarse constantemente. Asimismo, en algunas circunstancias,

un sistema puede estar congelado pero no interbloqueado; por ejemplo, esta situación ~~pueder~~ darse si un proceso en tiempo real que se ejecuta con la prioridad más alta (o cualquier ~~procesos~~ que se ejecute con un planificador sin desalojo) nunca devuelve el control al sistema operativo. El sistema debe disponer de métodos de recuperación manual para tales condiciones y puede, ~~solo~~, simplemente, usar esas mismas técnicas para recuperarse de los interbloqueos.

7.4 Prevención de interbloqueos

Como se ha dicho en la Sección 7.2.1, para que se produzca un interbloqueo deben cumplirse las cuatro condiciones necesarias. Asegurando que una de estas cuatro condiciones no se cumpla, podemos *prevenir* la aparición de interbloqueos. Vamos a abordar este método examinando cada una de las cuatro condiciones por separado.

7.4.1 Exclusión mutua

La condición de exclusión mutua se aplica a los recursos que no pueden ser compartidos. Por ejemplo, varios procesos no pueden compartir simultáneamente una impresora. Por el contrario, los recursos que sí pueden compartirse no requieren acceso mutuamente excluyente y, por tanto, no pueden verse implicados en un interbloqueo. Los archivos de sólo lectura son un buen ejemplo de recurso que puede compartirse. Si varios procesos intentan abrir un archivo de sólo lectura al mismo tiempo, puede concedérseles acceso al archivo de forma simultánea. Un proceso no necesita esperar nunca para acceder a un recurso compatible. Sin embargo, en general, no podemos evitar los interbloqueos negando la condición de exclusión mutua, ya que algunos recursos son intrínsecamente no compatibles.

7.4.2 Retención y espera

Para asegurar que la condición de retención y espera nunca se produzca en el sistema, debemos garantizar que, cuando un proceso solicite un recurso, el proceso no esté reteniendo ningún otro recurso. Un posible protocolo consiste en exigir que cada proceso solicite todos sus recursos (y que esos recursos se le asignen) antes de comenzar su ejecución. Podemos implementar este mecanismo requiriendo que las llamadas al sistema que solicitan los recursos para un proceso precedan a todas las demás llamadas al sistema.

Una posible alternativa sería un protocolo que permitiera a un proceso solicitar recursos sólo cuando no tenga ninguno retenido. Un proceso puede solicitar algunos recursos y utilizarlos. Sin embargo, antes de solicitar cualquier recurso adicional, tiene que liberar todos los recursos que tenga asignados actualmente.

Para ilustrar la diferencia entre estos dos protocolos, consideremos un proceso que copia datos de una unidad de DVD en un archivo en disco, ordena el archivo y luego imprime los resultados en una impresora. Si hay que solicitar todos los recursos antes de iniciar la ejecución, entonces el proceso tendrá que solicitar inicialmente la unidad de DVD, el archivo de disco y la impresora. El proceso retendrá la impresora durante todo el tiempo que dure la ejecución, incluso aunque sólo la necesite al final.

El segundo método permite al proceso solicitar inicialmente sólo la unidad de DVD y el archivo de disco. El proceso realiza la copia de la unidad de DVD al disco y luego libera ambos recursos. El proceso tiene entonces que solicitar de nuevo el archivo de disco y la impresora. Después de copiar el archivo de disco en la impresora, libera estos dos recursos y termina.

Ambos protocolos tienen dos desventajas importantes. Primero, la tasa de utilización de los recursos puede ser baja, dado que los recursos pueden asignarse pero no utilizarse durante un período largo de tiempo. Por ejemplo, en este caso, podemos liberar la unidad de DVD y el archivo de disco y luego solicitar otra vez el archivo de disco y la impresora, sólo si podemos estar seguros de que nuestros datos van a permanecer en el archivo. Si no podemos asegurarlo, entonces tendrímos que solicitar todos los recursos desde el principio con ambos protocolos.

La segunda desventaja es que puede producirse el fenómeno de inanición: un proceso que necesite varios recursos muy solicitados puede tener que esperar de forma indefinida, debido a que al menos uno de los recursos que necesita está siempre asignado a algún otro proceso.

7.4.3 Sin desalojo

La tercera condición necesaria para la aparición de interbloqueos es que los recursos que ya están asignados no sean desalojados. Para impedir que se cumpla esta condición, podemos usar el protocolo siguiente: si un proceso está reteniendo varios recursos y solicita otro recurso que no se le puede asignar de forma inmediata (es decir, el proceso tiene que esperar), entonces todos los recursos actualmente retenidos se desalojan. En otras palabras, estos recursos se liberan implícitamente. Los recursos desalojados se añaden a la lista de recursos que el proceso está esperando. El proceso sólo se reiniciará cuando pueda recuperar sus antiguos recursos, junto con los nuevos que está solicitando.

Alternativamente, si un proceso solicita varios recursos, primero comprobaremos si están disponibles. Si lo están, los asignaremos. Si no lo están, comprobaremos si están asignados a algún otro proceso que esté esperando recursos adicionales. Si es así, desalojaremos los recursos deseados del proceso en espera y los asignaremos al proceso que los ha solicitado. Si los recursos no están ni disponibles ni retenidos por un proceso en espera, el proceso solicitante debe esperar. Mientras espera, pueden desalojarse algunos de sus recursos, pero sólo si otro proceso los solicita. Un proceso sólo puede reiniciarse cuando se le asignan los nuevos recursos que ha solicitado y recupera cualquier recurso que haya sido desalojado mientras esperaba.

A menudo, este protocolo se aplica a tipos de recursos cuyo estado puede guardarse fácilmente y restaurarse más tarde, como los registros de la CPU y el espacio de memoria. Generalmente, no se puede aplicar a recursos como impresoras y unidades de cinta.

7.4.4 Espera circular

La cuarta y última condición para la aparición de interbloqueos es la condición de espera circular. Una forma de garantizar que esta condición nunca se produzca es imponer una ordenación total de todos los tipos de recursos y requerir que cada proceso solicite sus recursos en un orden creciente de enumeración.

Para ilustrarlo, sea $R = \{R_1, R_2, \dots, R_m\}$ el conjunto de tipos de recursos. Asignamos a cada tipo de recurso un número entero único, que nos permitirá comparar dos recursos y determinar si uno precede al otro dentro de nuestra ordenación. Formalmente, definimos una función uno-a-uno $F: R \rightarrow N$, donde N es el conjunto de los números naturales. Por ejemplo, si el conjunto de tipos de recursos R incluye unidades de cinta, unidades de disco e impresoras, entonces la función F se define como sigue:

$$\begin{aligned} F(\text{unidad de cinta}) &= 1 \\ F(\text{unidad de disco}) &= 5 \\ F(\text{impresora}) &= 12 \end{aligned}$$

Podemos considerar el siguiente protocolo para impedir los interbloqueos: cada proceso puede solicitar los recursos sólo en orden creciente de enumeración. Es decir, un proceso puede solicitar inicialmente cualquier número de instancias de un cierto tipo de recurso, por ejemplo R_i ; a continuación, el proceso puede solicitar instancias del tipo de recursos R_j si y sólo si $F(R_i) > F(R_j)$. Si se necesitan varias instancias del mismo tipo de recurso, se ejecuta una única solicitud para todos ellos. Por ejemplo, usando la función definida anteriormente, un proceso que desee utilizar la unidad de cinta y la impresora al mismo tiempo debe solicitar en primer lugar la unidad de cinta y luego la impresora. Alternativamente, podemos requerir que, si un proceso solicita una instancia del tipo de recurso R_i , tiene que liberar primero cualquier recurso R_j tal que $F(R_j) \geq F(R_i)$.

Si se usan estos dos protocolos, entonces la condición de espera circular no puede llegar a cumplirse. Podemos demostrar este hecho suponiendo que existe una espera circular (demostración por reducción al absurdo). Sea el conjunto de los procesos implicados en la espera circular $\{P_0, P_1, \dots, P_n\}$

$\dots, P_n\}$, donde P_i espera para acceder al recurso R_i , que está retenido por el proceso P_{i+1} (en los procesos se utiliza la aritmética de módulos, por lo que P_n espera un recurso R_n retenido por P_0). Entonces, dado que el proceso P_{i+1} está reteniendo el recurso R_i mientras solicita el recurso R_{i+1} , tiene que cumplir que $F(R_i) < F(R_{i+1})$ para todo i . Esta condición quiere decir que $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. Por transitividad, $F(R_0) < F(R_0)$, lo que es imposible. Por tanto, no puede existir una espera circular.

Podemos implementar este esquema en los programas de aplicación definiendo una ordenación entre todos los objetos de sincronización del sistema. Todas las solicitudes de objetos de sincronización se deben hacer en orden creciente. Por ejemplo, si el orden de bloqueo en el programa Pthread mostrado en la Figura 7.1 era

$$\begin{aligned}F(\text{primer_mutex}) &= 1 \\F(\text{segundo_mutex}) &= 5\end{aligned}$$

entonces `hebra_dos` no puede solicitar los bloqueos sin respetar el orden establecido.

Tenga presente que el definir una ordenación jerárquica no impide, por sí sólo, la aparición de interbloqueos. Es responsabilidad de los desarrolladores de aplicaciones escribir programas que respeten esa ordenación. Observe también que la función F debería definirse de acuerdo con el orden normal de utilización de los recursos en un sistema. Por ejemplo, puesto que normalmente la unidad de cinta se necesita antes que la impresora, sería razonable definir $F(\text{unidad de cinta}) < F(\text{impresora})$.

Aunque garantizar que los recursos se adquieran en el orden apropiado es responsabilidad de los desarrolladores de aplicaciones, puede utilizarse una solución software para verificar que los bloqueos se adquieran en el orden adecuado, y proporcionar las apropiadas advertencias cuando no sea así y puedan producirse interbloqueos. Uno de los verificadores existentes del orden de bloqueo, que funciona en las versiones BSD de UNIX (como por ejemplo FreeBSD), es `witness`. `Witness` utiliza bloqueos de exclusión mutua para proteger las secciones críticas, como se ha descrito en el Capítulo 6; opera manteniendo de forma dinámica el orden de los bloqueos en el sistema. Utilicemos como ejemplo el programa mostrado en la Figura 7.1. Suponga que la `hebra_uno` es la primera en adquirir los bloqueos y lo hace en el orden (1) `primer_mutex`, (2) `segundo_mutex`. `Witness` registra la relación que define que hay que adquirir `primer_mutex` antes que `segundo_mutex`. Si después `hebra_dos` adquiere los bloqueos sin respetar el orden, `witness` genera un mensaje de advertencia, que envía a la consola del sistema.

7.5 Evasión de interbloqueos

Como se ha explicado en la Sección 7.4, los algoritmos de prevención de interbloqueos impiden los interbloqueos restringiendo el modo en que pueden hacerse las solicitudes. Esas restricciones aseguran que al menos una de las condiciones necesarias para que haya interbloqueo no se produzca y, por tanto, que no puedan aparecer interbloqueos. Sin embargo, esta técnica de preventión de interbloqueos tiene algunos posibles efectos colaterales, como son una baja tasa de utilización de los dispositivos y un menor rendimiento del sistema.

Un método alternativo para evitar los interbloqueos consiste en requerir información adicional sobre cómo van a ser solicitados los recursos. Por ejemplo, en un sistema que disponga de una unidad de cinta y de una impresora, el sistema necesita saber que el proceso P va a requerir primero la unidad de cinta y después la impresora antes de liberar ambos recursos, mientras que el proceso Q va a requerir primero la impresora y luego la unidad de cinta. Conociendo exactamente la secuencia completa de solicitudes y liberaciones de cada proceso, el sistema puede decidir, para cada solicitud, si el proceso debe esperar o no, con el fin de evitar un posible interbloqueo futuro. Cada solicitud requiere que, para tomar la correspondiente decisión, el sistema considere los recursos actualmente disponibles, los recursos actualmente asignados a cada proceso y las solicitudes y liberaciones futuras de cada proceso.

Los diversos algoritmos que usan este método difieren en la cantidad y el tipo de la información requerida. El modelo más simple y útil requiere que cada proceso declare el *número máximo* de recursos de cada tipo que puede necesitar. Proporcionando esta información de antemano, es

possible construir un algoritmo que garantice que el sistema nunca entrará en estado de interbloqueo. Ese algoritmo es el que define el método concreto de evasión de interbloqueos. El algoritmo de evasión de interbloqueos examina dinámicamente el estado de asignación de cada recurso con el fin de asegurar que nunca se produzca una condición de espera circular. El *estado* de asignación del recurso está definido por el número de recursos disponibles y asignados y la demanda máxima de los procesos. En las siguientes secciones, vamos a estudiar dos algoritmos de evasión de interbloqueos.

7.5.1 Estado seguro

Un estado es *seguro* si el sistema puede asignar recursos a cada proceso (hasta su máximo) en determinado orden sin que eso produzca un interbloqueo. Más formalmente, un sistema está en estado seguro sólo si existe lo que se denomina una **secuencia segura**. Una secuencia de procesos $\langle P_1, P_2, \dots, P_n \rangle$ es una secuencia segura para el estado de asignación actual si, para cada P_i , las solicitudes de recursos que P_i pueda todavía hacer pueden ser satisfechas mediante los recursos actualmente disponibles, junto con los recursos retenidos por todos los P_j , con $j < i$. En esta situación, si los recursos que P_i necesita no están inmediatamente disponibles, entonces P_i puede esperar hasta que todos los P_j hayan terminado. Cuando esto ocurra, P_i puede obtener todos los recursos que necesite, completar las tareas que tenga asignadas, devolver sus recursos asignados y terminar. Cuando P_i termina, P_{i+1} puede obtener los recursos que necesita, etc. Si tal secuencia no existe, entonces se dice que el estado del sistema es **inseguro**.

Un estado seguro implica que no puede producirse interbloqueo. A la inversa, todo estado de interbloqueo es inseguro. Sin embargo, no todos los estados inseguros representan un interbloqueo (Figura 7.5). Un estado inseguro *puede* llevar a que aparezca un interbloqueo. Siempre y cuando el estado sea seguro, el sistema operativo podrá evitar los estados inseguros (y de interbloqueo). En un estado inseguro, el sistema operativo no puede impedir que los procesos soliciten recursos de tal modo que se produzca un interbloqueo: el comportamiento de los procesos es el que controla los estados inseguros.

Con el fin de ilustrar este tema, consideremos un sistema con 12 unidades de cinta magnética y tres procesos: P_0 , P_1 y P_2 . El proceso P_0 requiere 10 unidades de cinta, el proceso P_1 puede necesitar como mucho 4 unidades de cinta y el proceso P_2 puede necesitar hasta 9 unidades de cinta. Suponga que, en el instante t_0 , el proceso P_0 está reteniendo 5 unidades de cinta, el proceso P_1 está usando 2 unidades de cinta y el proceso P_2 está reteniendo 2 unidades de cinta (por tanto, quedan 3 unidades de cinta libres).

	Necesidades máximas	Necesidades actuales
P_0	10	5
P_1	4	2
P_2	9	2

En el instante t_0 , el sistema está en estado seguro. La secuencia $\langle P_1, P_0, P_2 \rangle$ satisface la condición de seguridad. Al proceso P_1 se le pueden asignar inmediatamente todas sus unidades de cinta, después de lo cual el proceso terminará por devolverlas (el sistema tendrá entonces 5 unidades de cinta disponibles); a continuación, el proceso P_0 puede tomar todas sus unidades de cinta y devolverlas cuando termine (el sistema tendrá entonces 10 unidades de cinta disponibles); y, por último, el proceso P_2 puede obtener todas sus unidades de cinta y devolverlas (el sistema tendrá entonces 12 unidades de cinta disponibles).

Un sistema puede pasar de un estado seguro a un estado inseguro. Suponga que, en el instante t_1 , el proceso P_2 hace una solicitud y se le asigna una unidad de cinta más. El estado del sistema dejará de ser seguro. En este momento, sólo pueden asignarse todas sus unidades de cinta al proceso P_1 ; cuando las devuelva, el sistema tendrá sólo 4 unidades de cinta disponibles. Dado que el proceso P_0 tiene asignadas cinco unidades pero tiene un máximo de 10, puede solicitar 5 unidades de cinta más; como no están disponibles, el proceso P_0 debe esperar. De forma similar, el proceso P_2 puede solicitar 6 unidades de cinta más y tener que esperar, dando lugar a un interblo-

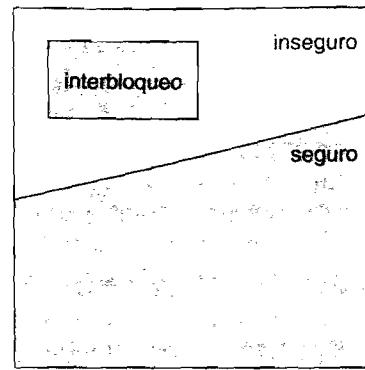


Figura 7.5 Espacio de los estados seguro, inseguro y de interbloqueo.

queo. El error se ha cometido al aprobar la solicitud del proceso P_2 de usar una unidad más. Si hubiéramos hecho esperar a P_2 hasta que los demás procesos hubieran terminado y liberado sus recursos, habríamos evitado el interbloqueo.

Conocido el concepto de estado seguro, podemos definir algoritmos para evitar los interbloqueos que aseguren que en el sistema nunca se producirá un interbloqueo. La idea consiste simplemente en garantizar que el sistema siempre se encuentre en estado seguro. Inicialmente, el sistema se encuentra en dicho estado. Cuando un proceso solicita un recurso que está disponible en ese momento, el sistema debe decidir si el recurso puede asignarse de forma inmediata o el proceso debe esperar. La solicitud se concede sólo si la asignación deja al sistema en estado seguro.

Con este esquema, si un proceso solicita un recurso que está disponible, es posible que tenga que esperar. Por tanto, la tasa de utilización del recurso puede ser menor que si no se utilizara este algoritmo.

7.5.2 Grafo de asignación de recursos

Si tenemos un sistema de asignación de recursos con sólo una instancia de cada tipo de recurso, puede utilizarse una variante del grafo de asignación de recursos definido en la Sección 7.2.2 para evitar los interbloqueos. Además de las aristas de solicitud y de asignación ya descritas, vamos a introducir un nuevo tipo de arista, denominada **arista de declaración**. Una arista de declaración $P_i \rightarrow R_j$ indica que el proceso P_i puede solicitar el recurso R_j en algún instante futuro. Esta arista es similar a una arista de solicitud en lo que respecta a la dirección, pero se representa en el grafo mediante una línea de trazos. Cuando el proceso P_i solicita el recurso R_j , la arista de declaración $P_i \rightarrow R_j$ se convierte en una arista de solicitud. De forma similar, cuando un proceso P_i libera un recurso R_j , la arista de asignación $R_j \rightarrow P_i$ se reconvierte en una arista de declaración $P_i \rightarrow R_j$. Observe que los recursos deben declararse de antemano al sistema, es decir, antes de que el proceso P_i inicie su ejecución, todas sus aristas de declaración deben estar indicadas en su grafo de asignación de recursos. Podemos suavizar esta condición permitiendo que se pueda añadir al grafo una arista de declaración $P_i \rightarrow R_j$ sólo si todas las aristas asociadas con el proceso P_i son aristas de declaración.

Supongamos que el proceso P_i solicita el recurso R_j . La solicitud se puede conceder sólo si la conversión de la arista de solicitud $P_i \rightarrow R_j$ en una arista de asignación $R_j \rightarrow P_i$ no da lugar a la formación de un ciclo en el grafo de asignación de recursos. Observe que realizamos la comprobación de si el estado es seguro utilizando un algoritmo de detección de ciclos. Los algoritmos de detección de ciclos en este grafo requieren del orden de n^2 operaciones, donde n es el número de procesos del sistema.

Si no se crea un ciclo, entonces la asignación del recurso dejará al sistema en un estado seguro. Si se encuentra un ciclo, entonces la asignación colocaría al sistema en un estado inseguro. Por tanto, el proceso P_i tendrá que esperar para que su solicitud sea satisfecha.

Para ilustrar este algoritmo, consideraremos el grafo de asignación de recursos de la Figura 7.6. Supongamos que P_2 solicita el recurso R_2 . Aunque R_2 actualmente está libre, no podemos asignar-

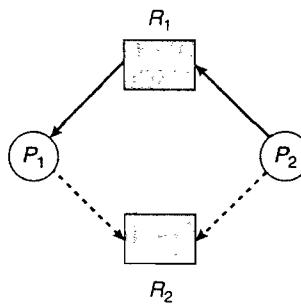


Figura 7.6 Grafo de asignación de recursos para evitar los interbloqueos.

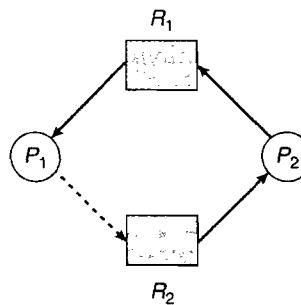


Figura 7.7 Un estado inseguro en un grafo de asignación de recursos.

lo a P_2 , ya que esta acción crearía un ciclo en el grafo (Figura 7.7). La existencia de un ciclo indica que el sistema se encuentra en un estado inseguro. Si P_1 solicita R_2 , y P_2 solicita R_1 , entonces se producirá un interbloqueo.

7.5.3 Algoritmo del banquero

El algoritmo del grafo de asignación de recursos no es aplicable a los sistemas de asignación de recursos con múltiples instancias de cada tipo de recurso. El algoritmo para evitar interbloqueos que vamos a describir a continuación es aplicable a dicho tipo de sistema, pero es menos eficiente que el método basado en el grafo de asignación de recursos. Habitualmente, este algoritmo se conoce con el nombre de *algoritmo del banquero*; se eligió este nombre porque el algoritmo podría utilizarse en los sistemas bancarios para garantizar que el banco nunca asigne sus fondos disponibles de tal forma que no pueda satisfacer las necesidades de todos sus clientes.

Cuando entra en el sistema un proceso nuevo, debe declarar el número máximo de instancias de cada tipo de recurso que puede necesitar. Este número no puede exceder el número total de recursos del sistema. Cuando un usuario solicita un conjunto de recursos, el sistema debe determinar si la asignación de dichos recursos dejará al sistema en un estado seguro. En caso afirmativo, los recursos se asignarán; en caso contrario, el proceso tendrá que esperar hasta que los otros procesos liberen los suficientes recursos.

Deben utilizarse varias estructuras de datos para implementar el algoritmo del banquero. Estas estructuras de datos codifican el estado del sistema de asignación de recursos. Sea n el número de procesos en el sistema y m el número de tipos de recursos. Necesitamos las siguientes estructuras:

- **Available** (disponibles). Un vector de longitud m que indica el número de recursos disponibles de cada tipo. Si $Available[j]$ es igual a k , existen k instancias disponibles del tipo de recurso R_j .
- **Max**. Una matriz $n \times m$ que indica la demanda máxima de cada proceso. Si $Max[i][j]$ es igual a k , entonces el proceso P_i puede solicitar como máximo k instancias del tipo de recurso R_j .

- **Allocation** (asignación). Una matriz $n \times m$ que define el número de recursos de cada uno actualmente asignados a cada proceso. Si $Allocation[i][j]$ es igual a k , entonces el proceso P_i tiene asignadas actualmente k instancias del tipo de recurso R_j .
- **Need** (necesidad). Una matriz $n \times m$ que indica la necesidad restante de recursos de cada uno para completar sus tareas. Si $Need[i][j]$ es igual a k , entonces el proceso P_i necesita k instancias más del tipo de recurso R_j para completar sus tareas. Observe que $Need[i][j]$ es igual a $Max[i][j] - Allocation[i][j]$.

Estas estructuras de datos varían con el tiempo, tanto en tamaño como en valor.

Para simplificar la presentación del algoritmo del banquero, a continuación vamos a definir cierta notación. Sean X e Y sendos vectores de longitud n . Decimos que $X \leq Y$ si y sólo si $X[i] \leq Y[i]$ para todo $i = 1, 2, \dots, n$. Por ejemplo, si $X = (1, 7, 3, 2)$ e $Y = (0, 3, 2, 1)$, entonces $Y \leq X$. $Y < X$ si $Y \leq X$ e $Y \neq X$.

Podemos tratar cada fila de las matrices *Allocation* y *Need* como un vector y denominarlos $Allocation_i$ y $Need_i$. El vector *Allocation_i* especifica los recursos actualmente asignados al proceso P_i , el vector *Need_i* especifica los recursos adicionales que el proceso P_i podría todavía solicitar para completar su tarea.

7.5.3.1 Algoritmo de seguridad

Ahora podemos presentar el algoritmo para averiguar si un sistema se encuentra en estado seguro o no. Este algoritmo puede describirse del siguiente modo:

1. Sean *Work* y *Finish* sendos vectores de longitud m y n , respectivamente. Inicializamos estos vectores de la forma siguiente: *Work* = *Available* y *Finish*[i] = *false* para $i = 0, 1, \dots, n - 1$.
2. Hallar i tal que
 - $Finish[i] == false$
 - $Need_i \leq Work$
- Si no existe i que cumpla estas condiciones, ir al paso 4.
3. $Work = Work + Allocation_i$
 $Finish[i] = true$
 Ir al paso 2.
4. Si $Finish[i] == true$ para todo i , entonces el sistema se encuentra en un estado seguro.

Este algoritmo puede requerir del orden de $m \times n^2$ operaciones para determinar si un estado es seguro.

7.5.3.2 Algoritmo de solicitud de recursos

Ahora vamos a describir el algoritmo que determina si las solicitudes pueden concederse de forma segura.

Sea *Request_i* el vector de solicitud para el proceso P_i . Si $Request_i[j] == k$, entonces el proceso P_i desea k instancias del tipo de recurso R_j . Cuando el proceso P_i hace una solicitud de recursos, se toman las siguientes acciones:

1. Si $Request_i \leq Need_i$, ir al paso 2. En caso contrario, se genera una condición de error, dado que el proceso ha excedido la cantidad máxima de recursos que había declarado.
2. Si $Request_i \leq Available$, ir al paso 3. En caso contrario, P_i tendrá que esperar, dado que los recursos no están disponibles.
3. Hacer como si el sistema hubiera asignado al proceso P_i los recursos solicitados, modificando el estado del modo siguiente:

$$\begin{aligned} Available &= Available - Request_i; \\ Allocation_i &= Allocation_i + Request_i; \\ Need_i &= Need_i - Request_i; \end{aligned}$$

Si el estado que resulta de la asignación de recursos es seguro, la transacción se completa y se asignan los recursos al proceso P_i . Sin embargo, si el nuevo estado resulta ser inseguro, entonces P_i debe esperar a que se le asignen los recursos $Request_i$, y se restaura el antiguo estado de asignación de recursos.

7.5.3.3 Un ejemplo ilustrativo

Por último, para ilustrar el uso del algoritmo del banquero, consideremos un sistema con cinco procesos P_0 a P_4 y tres tipos de recursos A, B y C. El tipo de recurso A tiene 10 instancias, el tipo de recurso B tiene 5 instancias y el tipo de recurso C tiene 7 instancias. Supongamos que, en el instante T_0 , se ha tomado la siguiente instantánea del sistema:

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

El contenido de la matriz *Need* se define como *Max* – *Allocation* y es el siguiente:

	<u>Need</u>
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Podemos afirmar que actualmente el sistema se encuentra en estado seguro. En efecto, la secuencia $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisface los criterios de seguridad. Supongamos ahora que el proceso P_1 solicita una instancia adicional del tipo de recurso A y dos instancias del tipo de recurso C, por lo que $Request_1 = (1, 0, 2)$. Para decidir si esta solicitud se puede conceder inmediatamente, primero comprobamos que $Request_1 \leq Available$, es decir, que $(1, 0, 2) \leq (3, 3, 2)$, lo que es cierto. Suponemos a continuación que esta solicitud se ha satisfecho y llegamos al siguiente nuevo estado:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Debemos determinar si este nuevo estado del sistema es seguro. Para ello, ejecutamos nuestro algoritmo de seguridad y comprobamos que la secuencia $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisface el requisito de seguridad. Por tanto, podemos conceder inmediatamente la solicitud del proceso P_1 .

Sin embargo, puede comprobarse que cuando el sistema se encuentra en este estado, una solicitud de valor $(3, 3, 0)$ por parte de P_4 no se puede conceder, ya que los recursos no están dispo-

nibles. Además, no se puede conceder una solicitud de valor $(0, 2, 0)$ por parte de P_0 , incluso aunque los recursos estén disponibles, ya que el estado resultante es inseguro.

Dejamos como ejercicio de programación para el lector la implementación del algoritmo de un banquero.

7.6 Detección de interbloqueos

Si un sistema no emplea ni algoritmos de prevención ni de evasión de interbloqueos, entonces puede producirse una situación de interbloqueo en el sistema. En este caso, el sistema debe proporcionar:

- Un algoritmo que examine el estado del sistema para determinar si se ha producido un interbloqueo.
- Un algoritmo para recuperarse del interbloqueo.

En la siguiente exposición, vamos a estudiar en detalle estos dos requisitos en lo que concierne tanto a los sistemas con una única instancia de cada tipo de recurso, como con varias instancias de cada tipo de recurso. Sin embargo, hay que resaltar de antemano que los esquemas de detección y recuperación tienen un coste significativo asociado, que incluye no sólo el coste (de tiempo de ejecución) asociado con el mantenimiento de la información necesaria y la ejecución del algoritmo de detección, sino también las potenciales pérdidas inherentes al proceso de recuperación de un interbloqueo.

7.6.1 Una sola instancia de cada tipo de recurso

Si todos los recursos tienen una única instancia, entonces podemos definir un algoritmo de detección de interbloqueos que utilice una variante del grafo de asignación de recursos, denominada grafo de espera. Obtenemos este grafo a partir del grafo de asignación de recursos eliminando los nodos de recursos y colapsando las correspondientes aristas.

De forma más precisa, una arista de P_i a P_j en un grafo de espera implica que el proceso P_i está esperando a que el proceso P_j libere un recurso que P_i necesita. En un grafo de espera existirá una arista $P_i \rightarrow P_j$ si y sólo si el correspondiente grafo de asignación de recursos contiene dos aristas $P_i \rightarrow R_q$ y $R_q \rightarrow P_j$ para algún recurso R_q . A modo de ejemplo, en la Figura 7.8 se presentan un grafo de asignación de recursos y el correspondiente grafo de espera.

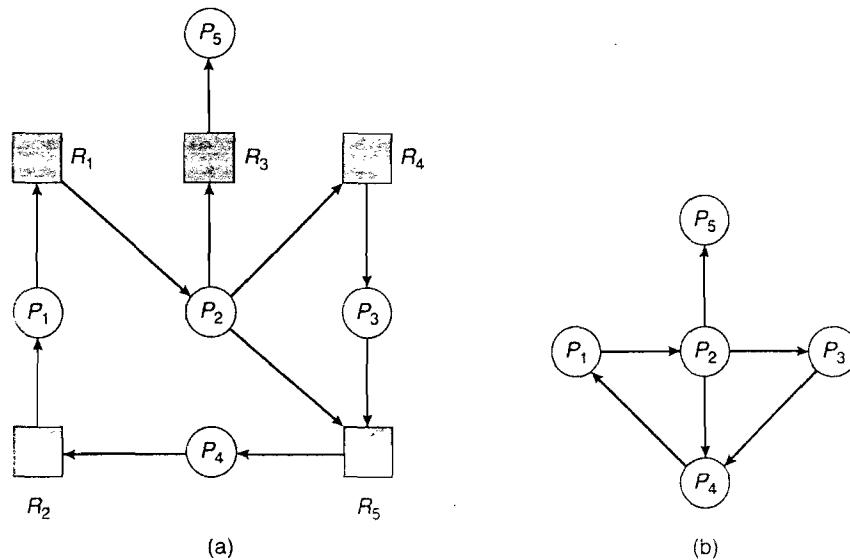


Figura 7.8 (a) Grafo de asignación de recursos. (b) Grafo de espera correspondiente.

Como antes, existirá un interbloqueo en el sistema si y sólo si el grafo de espera contiene un ciclo. Para detectar los interbloqueos, el sistema necesita *mantener* el grafo de espera e *invocar un algoritmo* periódicamente que compruebe si existe un ciclo en el grafo. Un algoritmo para detectar un ciclo en un grafo requiere del orden de n^2 operaciones, donde n es el número de vértices del grafo.

7.6.2 Varias instancias de cada tipo de recurso

El esquema del grafo de espera no es aplicable a los sistemas de asignación de recursos con múltiples instancias de cada tipo de recurso. Veamos ahora un algoritmo de detección de interbloqueos que pueda aplicarse a tales sistemas. El algoritmo emplea varias estructuras de datos variables con el tiempo, que son similares a las utilizadas en el algoritmo del banquero (Sección 7.5.3):

- **Available.** Un vector de longitud m que indica el número de recursos disponibles de cada tipo.
- **Allocation.** Una matriz $n \times m$ que define el número de recursos de cada tipo que están asignados actualmente a cada proceso.
- **Request.** Una matriz $n \times m$ que especifica la solicitud actual efectuada por cada proceso. Si $\text{Request}[i][j]$ es igual a k entonces el proceso P_i está solicitando k instancias más del tipo de recurso R_j .

La relación \leq entre dos vectores se define en la Sección (7.5.3). Para simplificar la notación, de nuevo vamos a tratar las filas de las matrices *Allocation* y *Request* como vectores; los especificaremos con la notación Allocation_i y Request_i . El algoritmo de detección que se describe aquí, simplemente investiga cada posible secuencia de asignación de los procesos que quedan por completarse. Compare este algoritmo con el algoritmo del banquero presentado en la Sección 7.5.3.

1. *Work* y *Finish* son vectores de longitud m y n , respectivamente. Inicialmente, *Work* = *Available*. Para $i = 0, 1, \dots, n - 1$, si $\text{Allocation}_i \neq 0$, entonces *Finish*[i] = *false*; en caso contrario, *Finish*[i] = *true*.
2. Hallar un índice i tal que
 - a. $\text{Finish}[i] == \text{false}$
 - b. $\text{Request}_i \leq \text{Work}$
 Si no existe tal i , ir al paso 4.
3. $\text{Work} = \text{Work} + \text{Allocation}_i$
 $\text{Finish}[i] = \text{true}$
 Ir al paso 2
4. Si $\text{Finish}[i] == \text{false}$, para algún i tal que $0 \leq i < n$, entonces el sistema se encuentra en estado de interbloqueo. Además, si $\text{Finish}[i] == \text{false}$, entonces el proceso P_i está en el interbloqueo.

Este algoritmo requiere del orden de $m \times n^2$ operaciones para detectar si el sistema se encuentra en un estado de interbloqueo.

El lector puede estarse preguntando por qué reclamamos los recursos del proceso P_i (en el paso 3) tan pronto como determinamos que $\text{Request}_i \leq \text{Work}$ (en el paso 2b). Sabemos que P_i actualmente *no* está implicado en un interbloqueo (dado que $\text{Request}_i \leq \text{Work}$). Por tanto, adoptamos una actitud optimista y suponemos que P_i no requerirá más recursos para completar sus tareas y que terminará por devolver al sistema todos los recursos que tenga actualmente asignados. Si nuestra suposición es incorrecta, más adelante se producirá un interbloqueo y dicho interbloqueo se detectará la siguiente vez que se invoque el algoritmo de detección de interbloqueos.

Para ilustrar este algoritmo, consideremos un sistema con cinco procesos P_0 a P_4 y tres tipos de recursos A , B y C . El tipo de recurso A tiene siete instancias, el tipo de recurso B tiene dos instan-

tas y el tipo de recurso C tiene seis instancias. Suponga que en el instante T_0 tenemos el siguiente estado de la asignación de recursos:

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Podemos afirmar que el sistema no se encuentra en estado de interbloqueo. En efecto, si ejecutamos nuestro algoritmo, comprobaremos que la secuencia $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ da como resultado $Finish[i] == true$ para todo i .

Suponga ahora que el proceso P_2 hace una solicitud de una instancia más del tipo de recurso C. La matriz Request se modifica como sigue:

	<u>Request</u>
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Ahora podemos afirmar que el sistema está en interbloqueo. Aunque podemos reclamar los recursos retenidos por el proceso P_0 , el número de recursos disponibles no es suficiente para satisfacer las solicitudes de los restantes procesos. Por tanto, existe un interbloqueo entre los procesos P_1, P_2, P_3 y P_4 .

7.6.3 Utilización del algoritmo de detección

Cuándo debemos invocar el algoritmo de detección? La respuesta depende de dos factores:

1. ¿Con qué frecuencia se producirá probablemente un interbloqueo?
2. ¿Cuántos procesos se verán afectados por el interbloqueo cuando se produzca?

Si los interbloqueos se producen frecuentemente, entonces el algoritmo de detección debe invocarse frecuentemente. Los recursos asignados a los procesos en interbloqueo estarán inactivos hasta que el interbloqueo se elimine. Además, el número de procesos implicados en el ciclo de interbloqueo puede aumentar.

Los interbloqueos se producen sólo cuando algún proceso realiza una solicitud que no se puede conceder de forma inmediata. Esta solicitud puede ser la solicitud final que complete una cadena de procesos en espera. En el caso extremo, podemos invocar el algoritmo de detección de interbloqueos cada vez que una solicitud de asignación no pueda ser concedida inmediatamente. En este caso, podemos no sólo identificar el conjunto de procesos en interbloqueo sino también el proceso concreto que ha “causado” dicho interbloqueo. (En realidad, cada uno de los procesos en interbloqueo es una arista del ciclo que aparece en el grafo de recursos, por lo que todos ellos, conjuntamente, dan lugar al interbloqueo.) Si existen muchos tipos de recursos diferentes, una solicitud puede crear muchos ciclos en el grafo de recursos, siendo completado cada ciclo por la solicitud más reciente y estando “causado” por ese proceso perfectamente identificable.

Por supuesto, si se invoca el algoritmo de detección de interbloqueos para cada solicitud de recursos, esto dará lugar a una considerable sobrecarga en el tiempo de uso del procesador. Una alternativa más barata consiste, simplemente, en invocar el algoritmo a intervalos menos frecuentes, por ejemplo una vez por hora o cuando la utilización de la CPU caiga por debajo del 40 por ciento. (Un interbloqueo puede paralizar el sistema y hacer que la utilización de la CPU disminuya)

ya.) Si el algoritmo de detección se invoca en instantes de tiempo arbitrarios, pueden aparecer muchos ciclos en el grafo de recursos; en este caso, generalmente, no podremos saber cuál de los muchos procesos en interbloqueo ha “causado” el interbloqueo.

7.7 Recuperación de un interbloqueo

Cuando el algoritmo de detección determina que existe un interbloqueo, tenemos varias alternativas. Una posibilidad es informar al operador de que se ha producido un interbloqueo y dejar que lo trate de forma manual. Otra posibilidad es dejar que sea el sistema el que haga la recuperación del interbloqueo de forma automática. Existen dos opciones para romper un interbloqueo. Una de ellas consiste simplemente en interrumpir uno o más procesos para romper la cadena de espera circular. La otra consiste en desalojar algunos recursos de uno o más de los procesos bloqueados.

7.7.1 Terminación de procesos

Para eliminar los interbloqueos interrumpiendo un proceso, se utiliza uno de dos posibles métodos. En ambos métodos, el sistema reclama todos los recursos asignados a los procesos terminados.

- **Interrumpir todos los procesos interbloqueados.** Claramente, este método interrumpirá el ciclo de interbloqueo, pero a un precio muy alto: los procesos en interbloqueo pueden haber consumido un largo período de tiempo y los resultados de estos cálculos parciales deben descartarse y, probablemente, tendrán que repetirse más tarde.
- **Interrumpir un proceso cada vez hasta que el ciclo de interbloqueo se elimine.** Este método requiere una gran cantidad de trabajo adicional, ya que después de haber interrumpido cada proceso hay que invocar un algoritmo de detección de interbloqueos para determinar si todavía hay procesos en interbloqueo.

Interrumpir un proceso puede no ser fácil. Si el proceso estaba actualizando un archivo, su terminación hará que el archivo quede en un estado incorrecto. De forma similar, si el proceso estaba imprimiendo datos en una impresora, el sistema debe reiniciar la impresora para que vuelva a un estado correcto antes de imprimir el siguiente trabajo.

Si se emplea el método de terminación parcial, entonces tenemos que determinar qué proceso o procesos en interbloqueo deben cancelarse. Esta determinación es una decisión de política, similar a las decisiones sobre la planificación de la CPU. La cuestión es básicamente de carácter económico: deberemos interrumpir aquellos procesos cuya terminación tenga un coste mínimo. Lamentablemente, el término *coste mínimo* no es demasiado preciso. Hay muchos factores que influyen en qué procesos seleccionar, entre los que se incluyen:

1. La prioridad del proceso.
2. Durante cuánto tiempo ha estado el proceso ejecutándose y cuánto tiempo de adicional necesita el proceso para completar sus tareas.
3. Cuántos y qué tipo de recursos ha utilizado el proceso (por ejemplo, si los recursos pueden desalojarse fácilmente).
4. Cuántos más recursos necesita el proceso para completarse.
5. Cuántos procesos hará falta terminar.
6. Si se trata de un proceso interactivo o de procesamiento por lotes.

7.7.2 Apropiación de recursos

Para eliminar los interbloqueos utilizando el método de apropiación de recursos, desalojamos de forma sucesiva los recursos de los procesos y asignamos dichos recursos a otros procesos hasta que el ciclo de interbloqueo se interrumpa.

Si se necesita el desalojo para tratar los interbloqueos, entonces debemos abordar tres cuestiones:

- 1. Selección de una víctima.** ¿De qué recursos hay que apropiarse y de qué procesos? Al igual que en la terminación de procesos, es necesario determinar el orden de apropiación de forma que se minimicen los costes. Los factores de coste pueden incluir parámetros como el número de recursos que está reteniendo un proceso en interbloqueo y la cantidad de tiempo que el proceso ha consumido hasta el momento durante su ejecución.
- 2. Anulación.** Si nos apropiamos de un recurso de un proceso, ¿qué debería hacerse con dicho proceso? Evidentemente, no puede continuar su ejecución normal, ya que no dispone de algún recurso que necesita. Debemos devolver el proceso a un estado seguro y reiniciarlo a partir de dicho estado.

En general, dado que es difícil determinar un estado seguro, la solución más sencilla es realizar una operación de anulación completa: interrumpir el proceso y reiniciarlo. Aunque es más efectivo realizar una operación de anulación del proceso sólo hasta donde sea necesario para romper el interbloqueo, este método requiere que el sistema mantenga más información sobre el estado de todos los procesos en ejecución.

- 3. Inanición.** ¿Cómo se puede asegurar que no se produzca la muerte por inanición de un proceso, es decir, cómo podemos garantizar que los recursos no se tomen siempre del mismo proceso?

En un sistema en el que la selección de la víctima se basa fundamentalmente en los factores de coste, puede ocurrir que siempre se elija el mismo proceso como víctima. Como resultado, este proceso nunca completará sus tareas, dando lugar a una situación de inanición que debe abordarse a la hora de implementar cualquier sistema real. Evidentemente, debemos asegurar que cada proceso pueda ser seleccionado como víctima sólo un (pequeño) número finito de veces. La solución más habitual consiste en incluir el número de anulaciones efectuadas dentro del algoritmo de cálculo del coste.

7.8 Resumen

Un estado de interbloqueo se produce cuando dos o más procesos están esperando indefinidamente a que se produzca un suceso que sólo puede producirse como resultado de alguna operación efectuada por otro de los procesos en espera. Existen tres métodos principales para tratar los interbloqueos:

- Utilizar algún protocolo de prevención o evasión de los interbloqueos, asegurando que el sistema nunca entrará en un estado de interbloqueo.
- Permitir que el sistema entre en un estado de interbloqueo, detectarlo y luego recuperarse de él.
- Ignorar el problema y actuar como si los interbloqueos nunca fueran a producirse en el sistema.

La tercera solución es la que aplican la mayor parte de los sistemas operativos, incluyendo UNIX y Windows.

Un interbloqueo puede producirse sólo si se dan simultáneamente en el sistema cuatro condiciones necesarias: exclusión mutua, retención y espera, ausencia de mecanismos de desalojo y espera circular. Para prevenir los interbloqueos, podemos hacer que al menos una de las condiciones necesarias nunca llegue a cumplirse.

Un método de evasión de los interbloqueos que es menos restrictivo que los algoritmos de prevención requiere que el sistema operativo disponga de antemano de información sobre el modo en que cada proceso utilizará los recursos del sistema. Por ejemplo, el algoritmo del banquer requiere tener de antemano información sobre el número máximo de recursos de cada clase que cada proceso puede solicitar. Con esta información, podemos definir un algoritmo para evitar los interbloqueos.

Si un sistema no emplea un protocolo para asegurar que nunca se produzcan interbloqueos, entonces debe utilizarse un esquema de detección y recuperación. Es necesario invocar un algoritmo de detección de interbloqueos para determinar si se ha producido un interbloqueo. En caso afirmativo, el sistema tiene que recuperarse terminando algunos de los procesos interbloqueados o desalojando recursos de algunos de los procesos interbloqueados.

Cuando se emplean técnicas de apropiación para tratar los interbloqueos, deben tenerse en cuenta tres cuestiones: la selección de una víctima, las operaciones de anulación y la muerte por inanición de los procesos. En un sistema que seleccione a las víctimas a las que hay que aplicar las operaciones de anulación fundamentalmente basándose en los costes, puede producirse la muerte por inanición de los procesos y el proceso seleccionado puede no llegar nunca a completar sus tareas.

Por último, algunos investigadores sostienen que ninguno de los métodos básicos resulta apropiado, por sí solo, para cubrir el espectro completo de los problemas de asignación de recursos en los sistemas operativos. Sin embargo, pueden combinarse los distintos métodos básicos con el fin de poder seleccionar un método óptimo para cada clase de recursos de un sistema.

Ejercicios

- 7.1 Considere el interbloqueo entre vehículos mostrado en la Figura 7.9.
 - a. Demuestre que las cuatro condiciones necesarias para que se produzca un interbloqueo se cumplen en este ejemplo.
 - b. Enuncie una regla simple para evitar los interbloqueos en este sistema.
- 7.2 Considere la situación de interbloqueo que podría producirse en el problema de la cena de los filósofos cuando cada uno de ellos toma un palillo cada vez. Explique cómo se cumplen las cuatro condiciones necesarias de interbloqueo en esta situación. Explique cómo podrían evitarse los interbloqueos impidiendo que se cumpla una cualquiera de las cuatro condiciones.
- 7.3 Una posible solución para evitar los interbloqueos es tener un único recurso de orden superior que debe solicitarse antes que cualquier otro recurso. Por ejemplo, si varias hebras intentan acceder a los objetos de sincronización $A \cdots E$, puede producirse un interbloqueo.

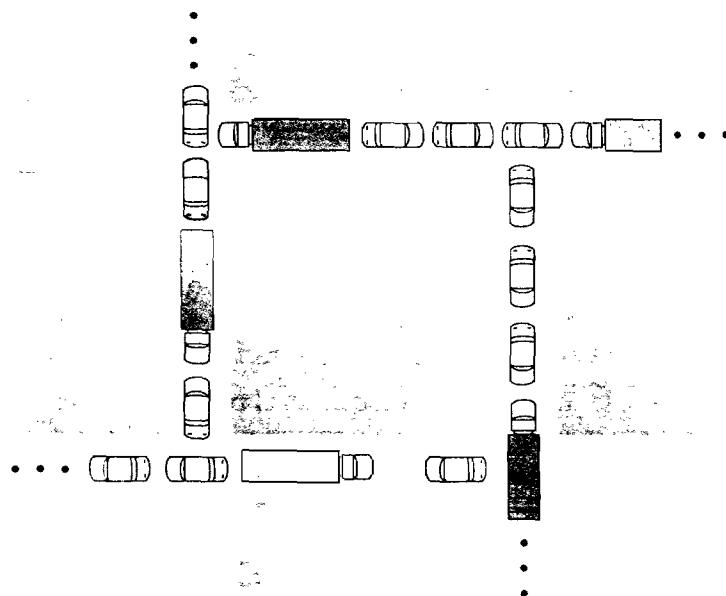


Figura 7.9 Interbloqueo entre vehículos para el Ejercicio 7.1.

(Tales objetos de sincronización pueden ser mítex, semáforos, variables de condición, etc.). Podemos impedir el interbloqueo añadiendo un sexto objeto F . Cuando una hebra quiera adquirir el bloqueo de sincronización de cualquier objeto $A \dots E$, primero deberá adquirir el bloqueo para el objeto F . Esta solución se conoce con el nombre de **contención**: los bloques para los objetos $A \dots E$ están contenidos dentro del bloqueo del objeto F . Compare este esquema con el esquema de espera circular de la Sección 7.4.4.

- 7.4** Compare el esquema de espera circular con los distintos esquemas de evasión de interbloqueos (como por ejemplo, el algoritmo del banquero) en lo que respecta a las cuestiones siguientes:
- Tiempo de ejecución adicional necesario
 - Tasa de procesamiento del sistema
- 7.5** En una computadora real, ni los recursos disponibles ni las demandas de recursos de los procesos son homogéneos durante períodos de tiempo largos (meses): los recursos se agotan o se reemplazan, aparecen y desaparecen procesos nuevos, se compran y añaden al sistema recursos adicionales... Si controlamos los interbloqueos mediante el algoritmo del banquero, ¿cuáles de los siguientes cambios pueden realizarse de forma segura (sin introducir la posibilidad de interbloqueos) y bajo qué circunstancias?
- Aumentar el valor de *Available* (nuevos recursos añadidos).
 - Disminuir el valor de *Available* (recursos eliminados permanentemente del sistema).
 - Aumentar el valor de *Max* para un proceso (el proceso necesita más recursos que los permitidos; puede desear más recursos).
 - Disminuir el valor de *Max* para un proceso (el proceso decide que no necesita tantos recursos).
 - Aumentar el número de procesos.
 - Disminuir el número de procesos.
- 7.6** Considere un sistema que tiene cuatro recursos del mismo tipo, compartidos entre tres procesos; cada proceso necesita como máximo dos recursos. Demostrar que el sistema está libre de interbloqueos.
- 7.7** Considere un sistema que consta de m recursos del mismo tipo, compartidos por n procesos. Los procesos sólo pueden solicitar y liberar los recursos de uno en uno. Demostrar que el sistema está libre de interbloqueos si se cumplen las dos condiciones siguientes:
- La necesidad máxima de cada proceso está comprendida entre 1 y m recursos.
 - La suma de todas las necesidades máximas es menor que $m + n$.
- 7.8** Considere el problema de la cena de los filósofos suponiendo que los palillos se colocan en el centro de la mesa y que cualquier filósofo puede usar dos cualesquier de ellos. Suponga que las solicitudes de palillos se hacen de una en una. Describa una regla simple para determinar si una solicitud concreta podría ser satisfecha sin dar lugar a interbloqueo, dada la asignación actual de palillos a los filósofos.
- 7.9** Considere la misma situación que en el problema anterior y suponga ahora que cada filósofo requiere tres palillos para comer y que cada solicitud de recurso se sigue realizando todavía por separado. Describa algunas reglas simples para determinar si una solicitud concreta podría ser satisfecha sin dar lugar a un interbloqueo, dada la asignación actual de palillos a los filósofos.
- 7.10** Podemos obtener un algoritmo simplificado del banquero para un único tipo de recurso a partir del algoritmo general del banquero, simplemente reduciendo la dimensionalidad de las diversas matrices en 1. Demuestre mediante un ejemplo que el algoritmo del banquero

para múltiples tipos de recursos no se puede implementar con sólo aplicar individualmente a cada tipo de recurso el algoritmo simplificado para un único tipo de recurso.

- 7.11 Considere la siguiente instantánea de un sistema:

	<u>Allocation</u>				<u>Max</u>				<u>Available</u>			
	A	B	C	D	A	B	C	D	A	B	C	D
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

Responda a las siguientes preguntas usando el algoritmo del banquero:

- ¿Cuál es el contenido de la matriz *Need*?
 - ¿Se encuentra el sistema en un estado seguro?
 - Si el proceso P_1 emite una solicitud de valor $(0, 4, 2, 0)$, ¿puede concederse inmediatamente dicha solicitud?
- 7.12 ¿Cuál es la suposición optimista realizada en el algoritmo de detección de interbloqueos? ¿Cómo podría violarse esta suposición?
- 7.13 Escriba un programa multihebra que implemente el algoritmo del banquero presentado en la Sección 7.5.3. Cree n hebras que soliciten y liberen recursos del banco. El banquero concederá la solicitud sólo si deja el sistema en un estado seguro. Puede escribir este programa usando hebras Pthreads o Win32. Es importante que el acceso concurrente a los datos compartidos sea seguro. Puede accederse de forma segura a dichos datos usando bloqueos mútex, disponibles tanto en la API de Pthreads como de Win32. En el proyecto dedicado al “problema del productor-consumidor” del Capítulo 6 se describen los cerrojos mútex disponibles en ambas bibliotecas.
- 7.14 Un puente de un único carril conecta dos pueblos de Estados Unidos denominados North Tunbridge y South Tunbridge. Los granjeros de los dos pueblos usan este puente para suministrar sus productos a la ciudad vecina. El puente puede bloquearse si un granjero de la parte norte y uno de la parte sur acceden al puente al mismo tiempo, porque los granjeros de esos pueblos son tercos y no están dispuestos a dar la vuelta con sus vehículos. Usando semáforos, diseñe un algoritmo que impida el interbloqueo. Inicialmente, no se preocupe del problema de una posible inanición (que se presentaría si los granjeros de uno de los pueblos impidieran a los del otro utilizar el puente).
- 7.15 Modifique la solución del Ejercicio 7.14 de modo que no pueda producirse inanición.

Notas bibliográficas

Dijkstra [1965a] fue uno de los primeros y más influyentes investigadores en el área de los interbloqueos. Holt [1972] fue la primera persona que formalizó el concepto de interbloqueos en términos de un modelo teórico de grafos similar al presentado en este capítulo. En Holt [1972] se cubre el tema de los bloqueos indefinidos. Hyman [1985] proporciona el ejemplo sobre interbloqueos extraído de una ley de Kansas. En Levine [2003] se proporciona un estudio reciente sobre el tratamiento de los interbloqueos.

Diversos algoritmos de prevención se explican en Havender [1968], que diseñó el esquema de ordenación de recursos para el sistema OS/360 de IBM.

El algoritmo del banquero para la evasión de interbloqueos fue desarrollado por Dijkstra [1965a] para el caso de un único tipo de recurso y en Habermann [1969] se extiende a varios tipos de recursos. Los Ejercicios 7.6 y 7.7 son de Holt [1971].

Coffman et al. [1971] presentan el algoritmo de detección de interbloqueos para varias instancias de un tipo de recurso, que se ha descrito en la Sección 7.6.2.

Bach [1987] describe cuántos de los algoritmos del *kernel* tradicional de UNIX tratan los interbloqueos. Las soluciones para los problemas de los interbloqueos en redes se abordan en trabajos tales como Culler et al. [1998] y Rodeheffer y Schroeder [1991].

El verificador del orden de bloqueo de witness se presenta en Baldwin [2002].

Parte Tres

Gestión de memoria

El propósito principal de un sistema informático es ejecutar programas. Estos programas, junto con los datos a los que acceden, deben encontrarse en memoria principal durante la ejecución (al menos parcialmente).

Para aumentar tanto el grado de utilización del procesador como su velocidad de respuesta a los usuarios, la computadora debe ser capaz de mantener varios procesos en memoria. Existen muchos esquemas de gestión de memoria, basados en técnicas diversas, y la efectividad de cada algoritmo depende de cada situación concreta. La selección de un esquema de gestión de memoria para un sistema determinado depende de muchos factores, especialmente, del diseño *hardware* del sistema. Cada uno de los algoritmos existentes requiere su propio soporte hardware.

Memoria principal

En el Capítulo 5, hemos mostrado cómo puede ser compartido el procesador por un conjunto de procesos. Como resultado de la planificación de la CPU, podemos mejorar tanto el grado de utilización del procesador como la velocidad de respuesta a los usuarios de la computadora. Para conseguir este incremento de las prestaciones debemos, sin embargo, ser capaces de mantener varios procesos en memoria; en otras palabras, debemos poder *compartir* la memoria.

En este capítulo, vamos a analizar diversas formas de gestionar la memoria. Como veremos, los algoritmos de gestión de memoria varían, desde técnicas primitivas sin soporte hardware específico a estrategias de paginación y segmentación. Cada una de las técnicas tiene sus propias ventajas y desventajas y la selección de un método de gestión de memoria para un sistema específico depende de muchos factores, y en especial del diseño *hardware* del sistema. Como veremos, muchos algoritmos requieren soporte hardware, aunque los diseños más recientes integran de manera estrecha el hardware y el sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Proporcionar una descripción detallada de las diversas formas de organizar el hardware de memoria.
- Analizar diversas técnicas de gestión de memoria, incluyendo la paginación y la segmentación.
- Proporcionar una descripción detallada del procesador Intel Pentium, que soporta tanto un esquema de segmentación pura como un mecanismo de segmentación con paginación.

8.1 Fundamentos

Como vimos en el Capítulo 1, la memoria es un componente crucial para la operación de un sistema informático moderno. La memoria está compuesta de una gran matriz de palabras o bytes, cada uno con su propia dirección. La CPU extrae instrucciones de la memoria de acuerdo con el valor del contador de programa. Estas instrucciones pueden provocar operaciones adicionales de carga o de almacenamiento en direcciones de memoria específicas.

Un ciclo típico de ejecución de una instrucción procedería en primer lugar, por ejemplo, a extraer una instrucción de la memoria. Dicha instrucción se decodifica y puede hacer que se extraigan de memoria una serie de operandos. Después de haber ejecutado la instrucción con esos operandos, es posible que se almacenen los resultados de nuevo en memoria. La unidad de memoria tan sólo ve un flujo de direcciones de memoria y no sabe cómo se generan esas direcciones (mediante el contador de programa, mediante indexación, indirección, direcciones literales, etc.) ni tampoco para qué se utilizan (instrucciones o datos). Por tanto, podemos ignorar el *cómo* genera el programa las direcciones de memoria; lo único que nos interesa es la secuencia de direcciones de memoria generadas por el programa en ejecución.

Comenzaremos nuestras explicaciones hablando sobre diversas cuestiones relacionadas con diferentes técnicas utilizadas para la gestión de la memoria. Entre estas cuestiones se incluyen una panorámica de los problemas hardware básicos, los mecanismos de acoplamiento de las direcciones simbólicas de memoria a las direcciones físicas reales y los métodos existentes para distinguir entre direcciones lógicas y físicas. Concluiremos con una exposición de los mecanismos para cargar y montar código dinámicamente y hablaremos también de las bibliotecas compartidas.

8.1.1 Hardware básico

La memoria principal y los registros integrados dentro del propio procesador son las únicas áreas de almacenamiento a las que la CPU puede acceder directamente. Hay instrucciones de máquina que toman como argumentos direcciones de memoria, pero no existe ninguna instrucción que acepte direcciones de disco. Por tanto, todas las instrucciones en ejecución y los datos utilizados por esas instrucciones deberán encontrarse almacenados en uno de esos dispositivos de almacenamiento de acceso directo. Si los datos no se encuentran en memoria, deberán llevarse hasta allí antes de que la CPU pueda operar con ellos.

Generalmente, puede accederse a los registros integrados en la CPU en un único ciclo del reloj del procesador. La mayoría de los procesadores pueden decodificar instrucciones y realizar operaciones simples con el contenido de los registros a la velocidad de una o más operaciones por cada tic de reloj. No podemos decir lo mismo de la memoria principal, a la que se accede mediante una transacción del bus de memoria. El acceso a memoria puede requerir muchos ciclos del reloj del procesador para poderse completar, en cuyo caso el procesador necesitará normalmente **detenerse**, ya que no dispondrá de los datos requeridos para completar la instrucción que esté ejecutando. Esta situación es intolerable, debido a la gran frecuencia con la que se accede a la memoria. El remedio consiste en añadir una memoria rápida entre la CPU y la memoria principal. En la Sección 1.8.3 se describe un búfer de memoria utilizado para resolver la diferencia de velocidad; dicho búfer de memoria se denomina **caché**.

No sólo debe preocuparnos la velocidad relativa del acceso a la memoria física, sino que también debemos garantizar una correcta operación que proteja al sistema operativo de los posibles accesos por parte de los procesos de los usuarios y que también proteja a unos procesos de usuario de otros. Esta protección debe ser proporcionada por el hardware y puede implementarse de diversas formas, como veremos a lo largo del capítulo. En esta sección, vamos a esbozar una posible implementación.

Primero tenemos que asegurarnos de que cada proceso disponga de un espacio de memoria separado. Para hacer esto, debemos poder determinar el rango de direcciones legales a las que el proceso pueda acceder y garantizar también que el proceso sólo acceda a esas direcciones legales. Podemos proporcionar esta protección utilizando dos registros, usualmente una base y un límite, como se muestra en la Figura 8.1. El **registro base** almacena la dirección de memoria física legal más pequeña, mientras que el **registro límite** especifica el tamaño del rango. Por ejemplo, si el registro base contiene el valor 300040 y el registro límite es 120900, entonces el programa podrá acceder legalmente a todas las direcciones comprendidas entre 300040 y 420940 (incluyendo los dos extremos).

La protección del espacio de memoria se consigue haciendo que el hardware de la CPU compare *todas* las direcciones generadas en modo usuario con el contenido de esos registros. Cualquier intento, por parte de un programa que se esté ejecutando en modo usuario, de acceder a la memoria del sistema operativo o a la memoria de otros usuarios hará que se produzca una interrupción hacia el sistema operativo, que tratará dicho intento como un error fatal (Figura 8.2). Este esquema evita que un programa de usuario modifique (accidental o deliberadamente) el código y las estructuras de datos del sistema operativo o de otros usuarios.

Los registros base y límite sólo pueden ser cargados por el sistema operativo, que utiliza una instrucción privilegiada especial. Puesto que las instrucciones privilegiadas sólo pueden ser ejecutadas en modo *kernel* y como sólo el sistema operativo se ejecuta en modo *kernel*, únicamente el sistema operativo podrá cargar los registros base y límite. Este esquema permite al sistema operativo modificar el valor de los registros, pero evita que los programas de usuario cambien el contenido de esos registros.

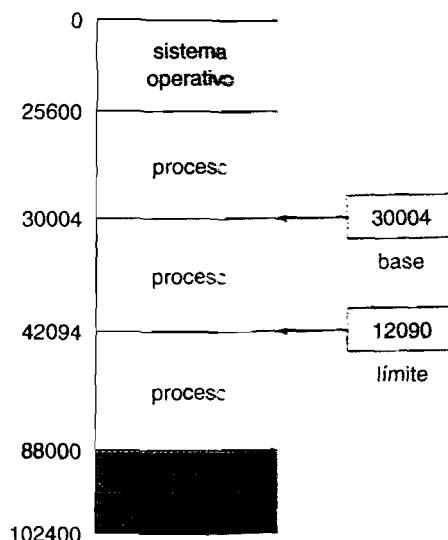


Figura 8.1 Un registro base y un registro límite definen un espacio lógico de direcciones.

El sistema operativo, que se ejecuta en modo privilegiado, tiene acceso no restringido a la memoria tanto del sistema operativo como de los usuarios. Esto permite al sistema operativo cargar los programas de los usuarios en la memoria de los usuarios, volcar dichos programas en caso de error, leer y modificar parámetros de las llamadas al sistema, etc.

8.1.2 Reasignación de direcciones

Usualmente, los programas residen en un disco en forma de archivos ejecutables binarios. Para poder ejecutarse, un programa deberá ser cargado en memoria y colocado dentro de un proceso. Dependiendo del mecanismo de gestión de memoria que se utilice, el proceso puede desplazarse entre disco y memoria durante su ejecución. Los procesos del disco que estén esperando a ser cargados en memoria para su ejecución forman lo que se denomina **cola de entrada**.

El procedimiento normal consiste en seleccionar uno de los procesos de la cola de entrada y cargar dicho proceso en memoria. A medida que se ejecuta el proceso, éste accede a las instrucciones y datos contenidos en la memoria. Eventualmente, el proceso terminará su ejecución y su espacio de memoria será declarado como disponible.

La mayoría de los sistemas permiten que un proceso de usuario resida en cualquier parte de la memoria física. Así, aunque el espacio de direcciones de la computadora comience en 00000, la

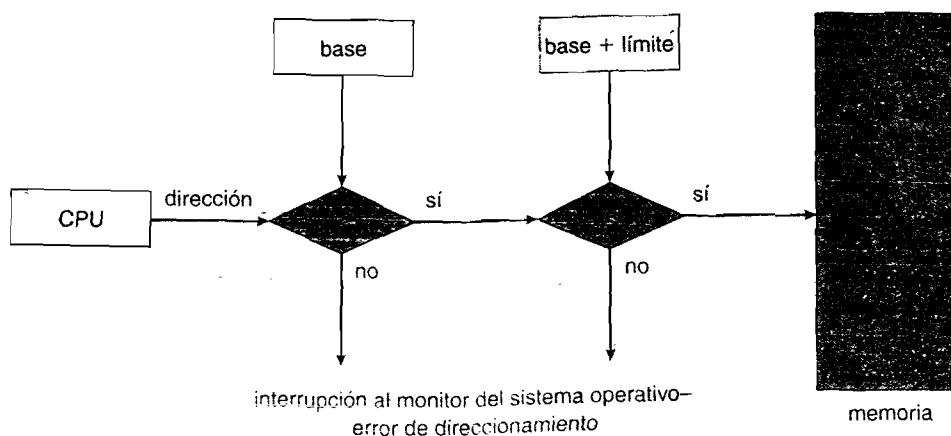


Figura 8.2 Protección hardware de las direcciones, utilizando un registro base y un registro límite.

primera dirección del proceso de usuario no tiene por qué ser 00000. Esta técnica afecta a las direcciones que el programa de usuario puede utilizar. En la mayoría de los casos, el programa de usuario tendrá que recorrer varios pasos (algunos de los cuales son opcionales) antes de ser ejecutado (Figura 8.3). A lo largo de estos pasos, las direcciones pueden representarse de diferentes formas. Las direcciones del programa fuente son generalmente simbólicas (como por ejemplo *saldo*). Normalmente, un compilador se encargará de **reasignar** estas direcciones simbólicas a direcciones reubicables (como por ejemplo, "14 bytes a partir del comienzo de este módulo"). El editor de montaje o cargador se encargará, a su vez, de reasignar las direcciones reubicables a direcciones absolutas (como por ejemplo, 74014). Cada operación de reasignación constituye una relación de un espacio de direcciones a otro.

Clásicamente, la reasignación de las instrucciones y los datos a direcciones de memoria puede realizarse en cualquiera de los pasos:

- **Tiempo de compilación.** Si sabemos en el momento de realizar la compilación dónde va a residir el proceso en memoria, podremos generar **código absoluto**. Por ejemplo, si sabemos que un proceso de usuario va a residir en una zona de memoria que comienza en la ubicación *R*, el código generado por el compilador comenzará en dicha ubicación y se extenderá a partir de ahí. Si la ubicación inicial cambiase en algún instante posterior, entonces sería

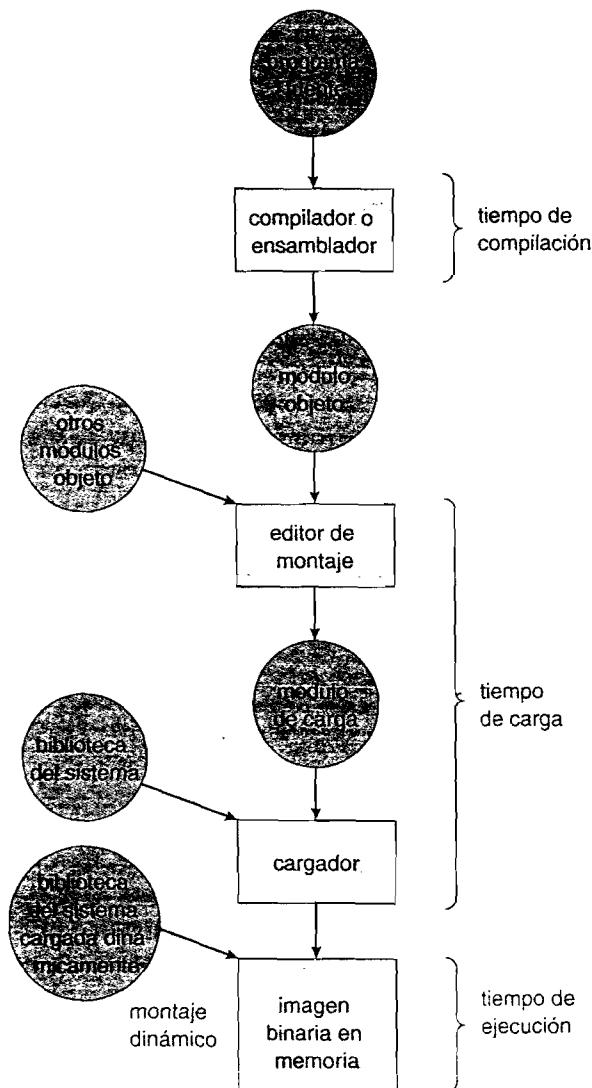


Figura 8.3 Pasos en el procesamiento de un programa de usuario.

necesario recompilar ese código. Los programas en formato .COM de MS-DOS se acoplan en tiempo de compilación.

- **Tiempo de carga.** Si no conocemos en tiempo de compilación dónde va a residir el proceso en memoria, el compilador deberá generar **código reubicable**. En este caso, se retarda la reasignación final hasta el momento de la carga. Si cambia la dirección inicial, tan sólo es necesario volver a cargar el código de usuario para incorporar el valor modificado.
- **Tiempo de ejecución.** Si el proceso puede desplazarse durante su ejecución desde un segmento de memoria a otro, entonces es necesario retardar la reasignación hasta el instante de la ejecución. Para que este esquema pueda funcionar, será preciso disponer de hardware especial, como veremos en la Sección 8.1.3. La mayoría de los sistemas operativos de propósito general utilizan este método.

Buena parte de este capítulo está dedicada a mostrar cómo pueden implementarse estos diversos esquemas de reasignación en un sistema informático de manera efectiva, y a analizar el soporte hardware adecuado para cada uno.

8.1.3 Espacios de direcciones lógico y físico

Una dirección generada por la CPU se denomina comúnmente **dirección lógica**, mientras que una dirección vista por la unidad de memoria (es decir, la que se carga en el **registro de direcciones de memoria** de la memoria) se denomina comúnmente **dirección física**.

Los métodos de reasignación en tiempo de compilación y en tiempo de carga generan direcciones lógicas y físicas idénticas. Sin embargo, el esquema de reasignación de direcciones en tiempo de ejecución hace que las direcciones lógica y física difieran. En este caso, usualmente decimos que la dirección lógica es una **dirección virtual**. A lo largo de este texto, utilizaremos los términos **dirección lógica** y **dirección virtual** de manera intercambiable. El conjunto de todas las direcciones lógicas generadas por un programa es lo que se denomina un **espacio de direcciones lógicas**; el conjunto de todas las direcciones físicas correspondientes a estas direcciones lógicas es un **espacio de direcciones físicas**. Así, en el esquema de reasignación de direcciones en tiempo de ejecución, decimos que los espacios de direcciones lógicas y físicas difieren.

La correspondencia entre direcciones virtuales y físicas en tiempo de ejecución es establecida por un dispositivo hardware que se denomina **unidad de gestión de memoria** (MMU, memory-management unit). Podemos seleccionar entre varios métodos distintos para establecer esta correspondencia, como veremos en las Secciones 8.3 a 8.7. Por el momento, vamos a ilustrar esta operación de asociación mediante un esquema MMU simple, que es una generalización del esquema de registro base descrita en la Sección 8.1. El registro base se denominará ahora **registro de reubicación**. El valor contenido en el registro de reubicación *suma* a todas las direcciones generadas por un proceso de usuario en el momento de enviarlas a memoria (véase la Figura 8.4.). Por ejemplo, si la base se encuentra en la dirección 14000, cualquier intento del usuario de direccionar la posición de memoria cero se reubicará dinámicamente en la dirección 14000; un acceso a la ubicación 346 se convertirá en la ubicación 14346. El sistema operativo MS-DOS que se ejecuta sobre la familia de procesadores Intel 80x86 utiliza cuatro registros de reubicación a la hora de cargar y ejecutar procesos.

El programa de usuario nunca ve las direcciones físicas *reales*. El programa puede crear un puntero a la ubicación 346, almacenarlo en memoria, manipularlo y compararlo con otras direcciones, siempre como el número 346. Sólo cuando se lo utiliza como dirección de memoria (por ejemplo, en una operación de lectura o escritura indirecta) se producirá la reubicación en relación con el registro base. El programa de usuario maneja direcciones *lógicas* y el hardware de conversión (mapeo) de memoria convierte esas direcciones lógicas en direcciones físicas. Esta forma de acoplamiento en tiempo de ejecución ya fue expuesta en la Sección 8.1.2. La ubicación final de una dirección de memoria referenciada no se determina hasta que se realiza esa referencia.

Ahora tenemos dos tipos diferentes de direcciones: direcciones lógicas (en el rango comprendido entre 0 y *max*) y direcciones físicas (en el rango comprendido entre *R* + 0 y *R* + *max* para un valor base igual a *R*). El usuario sólo genera direcciones lógicas y piensa que el proceso se ejecu-

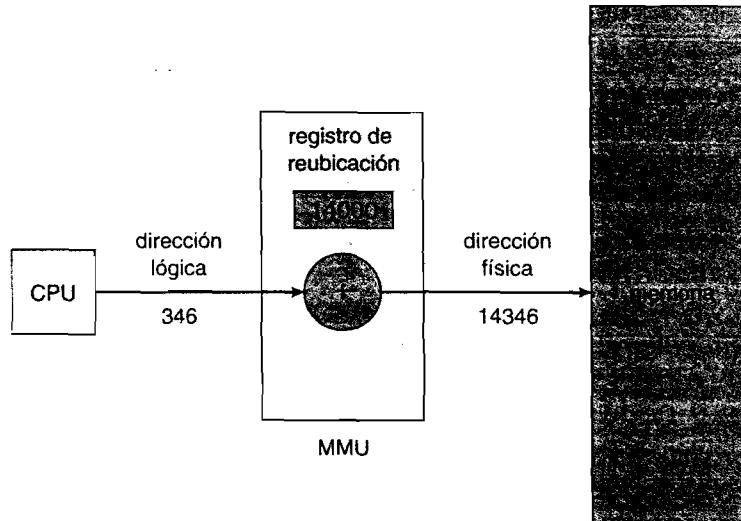


Figura 8.4 Reubicación dinámica mediante un registro de reubicación.

ta en las ubicaciones comprendidas entre 0 y *max*. El programa de usuario suministra direcciones lógicas y estas direcciones lógicas deben ser convertidas en direcciones físicas antes de utilizarlas.

El concepto de un *espacio de direcciones lógicas* que se acopla a un *espacio de direcciones físicas*, separado resulta crucial para una adecuada gestión de la memoria.

8.1.4 Carga dinámica

En las explicaciones que hemos dado hasta el momento, todo el programa y todos los datos de un proceso deben encontrarse en memoria física para que ese proceso pueda ejecutarse. En consecuencia, el tamaño de un proceso está limitado por el tamaño de la memoria física. Para obtener una mejor utilización del espacio de memoria, podemos utilizar un mecanismo de **carga dinámica**. Con la carga dinámica, una rutina no se carga hasta que se la invoca; todas las rutinas se mantienen en disco en un formato de carga reubicable. Según este método, el programa principal se carga en la memoria y se ejecuta. Cuando una rutina necesita llamar a otra rutina, la rutina que realiza la invocación comprueba primero si la otra ya ha sido cargada, si no es así, se invoca el cargador de montaje reubicable para que cargue en memoria la rutina deseada y para que actualice las tablas de direcciones del programa con el fin de reflejar este cambio. Después, se pasa el control a la rutina recién cargada.

La ventaja del mecanismo de carga dinámica es que una rutina no utilizada no se cargará nunca en memoria. Este método resulta particularmente útil cuando se necesitan grandes cantidades de código para gestionar casos que sólo ocurren de manera infrecuente, como por ejemplo rutinas de error. En este caso, aunque el tamaño total del programa pueda ser grande, la porción que se utilice (y que por tanto se cargue) puede ser mucho más pequeña.

El mecanismo de carga dinámica no requiere de ningún soporte especial por parte del sistema operativo. Es responsabilidad de los usuarios diseñar sus programas para poder aprovechar dicho método. Sin embargo, los sistemas operativos pueden ayudar al programador proporcionándole rutinas de biblioteca que implementen el mecanismo de carga dinámica.

8.1.5 Montaje dinámico y bibliotecas compartidas

La Figura 8.3 muestra también **bibliotecas de montaje dinámico**. Algunos sistemas operativos sólo permiten el **montaje estático**, mediante el cual las bibliotecas de lenguaje del sistema se tratan como cualquier otro módulo objeto y son integradas por el cargador dentro de la imagen binaria del programa. El concepto de montaje binario es similar al de carga dinámica, aunque en este caso lo que se pospone hasta el momento de la ejecución es el montaje, en lugar de la carga. Esta

funcionalidad suele emplearse con las bibliotecas del sistema, como por ejemplo las bibliotecas de subrutinas del lenguaje. Utilizando este mecanismo, cada programa de un sistema deberá incluir una copia de su biblioteca de lenguaje (o al menos de las rutinas a las que haga referencia el programa) dentro de la imagen ejecutable. Este requisito hace que se desperdicie tanto espacio de disco como memoria principal.

Con el montaje dinámico, se incluye un *stub* dentro de la imagen binaria para cada referencia a una rutina de biblioteca. El stub es un pequeño fragmento de código que indica cómo localizar la rutina adecuada de biblioteca residente en memoria o cómo cargar la biblioteca si esa rutina no está todavía presente. Cuando se ejecuta el stub, éste comprueba si la rutina necesaria ya se encuentra en memoria; si no es así, el programa carga en memoria la rutina. En cualquiera de los casos, el stub se sustituye así mismo por la dirección de la rutina y ejecuta la rutina. Así, la siguiente vez que se ejecute ese segmento de código concreto, se ejecutará directamente la rutina de biblioteca, sin tener que realizar de nuevo el montaje dinámico. Con este mecanismo, todos los procesos que utilicen una determinada biblioteca de lenguaje sólo necesitan ejecutar una copia del código de la biblioteca.

Esta funcionalidad puede ampliarse a las actualizaciones de las bibliotecas de código (como por ejemplo las destinadas a corregir errores). Puede sustituirse una biblioteca por una nueva versión y todos los programas que hagan referencia a la biblioteca emplearán automáticamente la versión más reciente. Sin el mecanismo de montaje dinámico, sería necesario volver a montar todos esos programas para poder acceder a la nueva biblioteca. Para que los programas no ejecuten accidentalmente versiones nuevas e incompatibles de las bibliotecas, suele incluirse información de versión tanto en el programa como en la biblioteca. Puede haber más de una versión de una biblioteca cargada en memoria y cada programa utilizará su información de versión para decidir qué copia de la biblioteca hay que utilizar. Los cambios de menor entidad retendrán el mismo número de versión, mientras que para los cambios de mayor entidad se incrementará ese número. De este modo, sólo los programas que se compilen con la nueva versión de la biblioteca se verán afectados por los cambios incompatibles incorporados en ella. Otros programas montados antes de que se instalara la nueva biblioteca continuarán utilizando la antigua. Este sistema se conoce también con el nombre de **bibliotecas compartidas**.

A diferencia de la carga dinámica, el montaje dinámico suele requerir algo de ayuda por parte del sistema operativo. Si los procesos de la memoria están protegidos unos de otros, entonces el sistema operativo será la única entidad que pueda comprobar si la rutina necesaria se encuentra dentro del espacio de memoria de otro proceso y será también la única entidad que pueda permitir a múltiples procesos acceder a las mismas direcciones de memoria. Hablaremos más en detalle de este concepto cuando analicemos el mecanismo de paginación en la Sección 8.4.

8.2 Intercambio

Un proceso debe estar en memoria para ser ejecutado. Sin embargo, los procesos pueden ser **intercambiados** temporalmente, sacándolos de la memoria y almacenándolos en un **almacén de respaldo** y volviéndolos a llevar luego a memoria para continuar su ejecución. Por ejemplo, suponga que estamos utilizando un entorno de multiprogramación con un algoritmo de planificación de CPU basado en turnos. Cuando termina un cuento de tiempo, el gestor de memoria comienza a sacar de ésta el proceso que acaba de terminar y a cargar en el espacio de memoria liberado por otro proceso (Figura 8.5). Mientras tanto, el planificador de la CPU asignará un cuento de tiempo a algún otro proceso que ya se encuentre en memoria. Cada vez que un proceso termine su cuento asignado, se intercambiará por otro proceso. Idealmente, el gestor de memoria puede intercambiar los procesos con la suficiente rapidez como para que haya siempre algunos procesos en memoria, listos para ejecutarse, cuando el planificador de la CPU quiera asignar el procesador a otra tarea. Además, el cuento debe ser lo suficientemente grande como para que pueda realizarse una cantidad razonable de cálculos entre una operación de intercambios y la siguiente.

Para los algoritmos de planificación con prioridad se utiliza una variante de esta política de intercambio. Si llega un proceso de mayor prioridad y ese proceso desea ser servido, el gestor de memoria puede descargar el proceso de menor prioridad y, a continuación, cargar y ejecutar

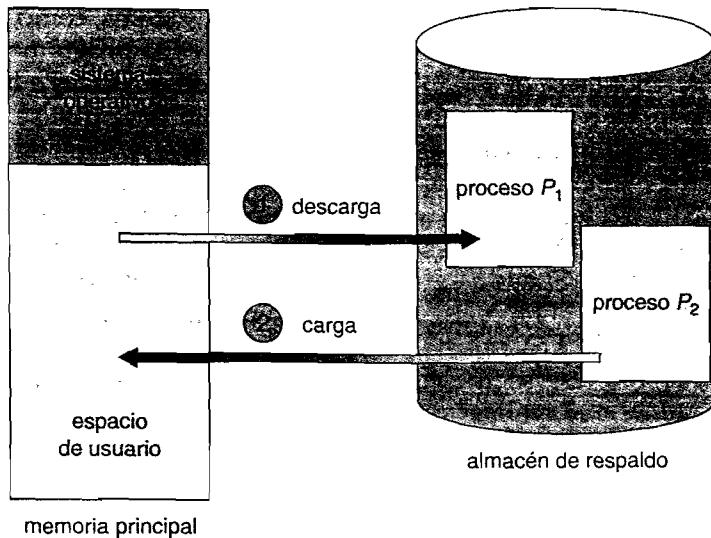


Figura 8.5 Intercambio de dos procesos utilizando un disco como almacén de respaldo.

el que tiene una prioridad mayor. Cuando termine el proceso de mayor prioridad, puede intercambiarse por el proceso de menor prioridad, que podrá entonces continuar su ejecución.

Normalmente, un proceso descargado se volverá a cargar en el mismo espacio de memoria que ocupaba anteriormente. Esta restricción está dictada por el método de reasignación de las direcciones. Si la reasignación se realiza en tiempo de ensamblado o en tiempo de carga, entonces no resulta sencillo mover el proceso a una ubicación diferente. Sin embargo, si se está utilizando reasignación en tiempo de ejecución sí que puede moverse el proceso a un espacio de memoria distinto, porque las direcciones físicas se calculan en tiempo de ejecución.

Los mecanismos de intercambio requieren un almacén de respaldo, que normalmente será un disco suficientemente rápido. El disco debe ser también lo suficientemente grande como para poder albergar copias de todas las imágenes de memoria para todos los usuarios, y debe proporcionar un acceso directo a estas imágenes de memoria. El sistema mantiene una **cola de procesos preparados** que consistirá en todos los procesos cuyas imágenes de memoria se encuentren en el almacén de respaldo o en la memoria y estén listos para ejecutarse. Cada vez que el planificador de la CPU decide ejecutar un proceso, llama al despachador, que mira a ver si el siguiente proceso de la cola se encuentra en memoria. Si no es así, y si no hay ninguna región de memoria libre, el despachador intercambia el proceso deseado por otro proceso que esté actualmente en memoria. A continuación, recarga los registros y transfiere el control al proceso seleccionado.

El tiempo necesario de cambio de contexto en uno de estos sistemas de intercambio es relativamente alto. Para hacernos una idea del tiempo necesario para el cambio de contexto, vamos a suponer que el proceso de usuario tiene 10 MB de tamaño y que el almacén de respaldo es un disco duro estándar con una velocidad de transferencia de 40 MB por segundo. La operación de transferencia del proceso de 10 MB hacia o desde la memoria principal requerirá

$$\begin{aligned} 10000 \text{ KB} / 40000 \text{ KB por segundo} &= 1/4 \text{ segundo} \\ &= 250 \text{ milisegundos.} \end{aligned}$$

Suponiendo que no sea necesario ningún posicionamiento del cabezal y asumiendo una latencia media de 8 milisegundos, el tiempo de intercambio será de 258 milisegundos. Puesto que el intercambio requiere tanto una operación de carga como otra de descarga, el tiempo total necesario será de unos 516 milisegundos.

Para conseguir un uso eficiente de la CPU, es necesario que el tiempo de ejecución de cada proceso sea largo en relación con el tiempo de intercambio. Así, en un algoritmo de planificación de la CPU por turnos, por ejemplo, el cuento de tiempo debe ser sustancialmente mayor que 0,516 segundos.

Observe que la mayor parte del tiempo de intercambio es tiempo de transferencia. El tiempo de transferencia total es directamente proporcional a la *cantidad* de memoria intercambiada. Si tenemos un sistema informático con 512 MB de memoria principal y un sistema operativo residente que ocupa 25 MB, el tamaño máximo de un proceso de usuario será de 487 MB. Sin embargo, muchos procesos de usuario pueden ser mucho más pequeños, como por ejemplo de 10 MB. Un proceso de 10 MB podría intercambiarse en 258 milisegundos, comparado con los 6,4 segundos requeridos para intercambiar 256 MB. Claramente, resultaría útil conocer exactamente cuánta memoria *está utilizando* un proceso de usuario, y no simplemente cuánta *podría estar* utilizando. Si tuviéramos ese dato, sólo necesitaríamos intercambiar lo que estuviera utilizándose realmente, reduciendo así el tiempo de intercambio. Para que este método sea efectivo, el usuario debe mantener informado al sistema acerca de cualquier cambio que se produzca en lo que se refiere a los requisitos de memoria. Así, un proceso con requisitos de memoria dinámicos necesitará ejecutar llamadas al sistema (`request memory` y `release memory`) para informar al sistema operativo de sus cambiantes necesidades de memoria.

El intercambio está restringido también por otros factores. Si queremos intercambiar un proceso, deberemos asegurarnos de que esté completamente inactivo. En este sentido, es necesario prestar una atención especial a todas las operaciones de E/S pendientes. Un proceso puede estar esperando por una operación de E/S en el momento en que queramos intercambiarlo con el fin de liberar memoria. En ese caso, si la E/S está accediendo asíncronamente a la memoria de usuario donde residen los búferes de E/S, el proceso no podrá ser intercambiado. Suponga que la operación de E/S está en cola debido a que el dispositivo está ocupado. Si descargáramos el proceso P_1 y cargáramos el proceso P_2 , la operación de E/S podría entonces intentar utilizar la memoria que ahora pertenece al proceso P_2 . Hay dos soluciones principales a este problema: no descargar nunca un proceso que tenga actividades de E/S pendientes o ejecutar las operaciones de E/S únicamente con búferes del sistema operativo. En este último caso, las transferencias entre los búferes del sistema operativo y la memoria del proceso sólo se realizan después de cargar de nuevo el proceso.

La suposición mencionada anteriormente de que el intercambio requiere pocas (o ninguna) operaciones de posicionamiento de los cabezales de disco requiere una explicación un poco más detallada, pero dejaremos la explicación de esta cuestión para el Capítulo 12, donde hablaremos de la estructura de los almacenamientos secundarios. Generalmente, el espacio de intercambio se asigna como un área de disco separada del sistema de archivos, para que su uso sea lo más rápido posible.

Actualmente, estos mecanismos estándar de intercambio se utilizan en muy pocos sistemas. Dicho mecanismo requiere un tiempo de intercambio muy alto y proporciona un tiempo de ejecución demasiado pequeño como para constituir una solución razonable de gestión de memoria. Sin embargo, lo que sí podemos encontrar en muchos sistemas son versiones modificadas de este mecanismo de intercambio.

En muchas versiones de UNIX, se utiliza una variante del mecanismo de intercambio. El intercambio está normalmente desactivado, pero se activa si se están ejecutando numerosos procesos y si la cantidad de memoria utilizada excede un cierto umbral. Una vez que la carga del sistema se reduce, vuelve a desactivarse el mecanismo de intercambio. Los mecanismos de gestión de memoria en UNIX se describen en detalle en las Secciones 21.7 y A.6.

Las primeras computadoras personales, que carecían de la sofisticación necesaria para implementar métodos de gestión de memoria más avanzados, ejecutaban múltiples procesos de gran tamaño utilizando una versión modificada del mecanismo de intercambio. Un ejemplo sería el sistema operativo Microsoft Windows 3.1, que soportaba la ejecución concurrente de procesos en memoria. Si se cargaba un nuevo proceso y no había la suficiente memoria principal, se descargaba en el disco otro proceso más antiguo. Este sistema operativo, sin embargo, no proporciona un mecanismo de intercambio completo, porque era el usuario, en lugar del planificador, quien decidía cuándo era el momento de descargar un proceso para cargar otro. Cualquier proceso que se hubiera descargado permanecía descargado (y sin ejecutarse) hasta que el usuario lo volviera a seleccionar para ejecución. Las versiones subsiguientes de los sistemas operativos de Microsoft aprovechan las características avanzadas de MMU que ahora incorporan las computadoras perso-

nales. Exploraremos dichas características en la Sección 8.4 y en el Capítulo 9, donde hablaremos de la memoria virtual.

8.3 Asignación de memoria contigua

La memoria principal debe albergar tanto el sistema operativo como los diversos procesos de usuario. Por tanto, necesitamos asignar las distintas partes de la memoria principal de la forma más eficiente posible. Esta sección explica uno de los métodos más comúnmente utilizados, la asignación contigua de memoria.

La memoria está usualmente dividida en dos particiones: una para el sistema operativo residente y otra para los procesos de usuario. Podemos situar el sistema operativo en la zona baja o en la zona alta de la memoria. El principal factor que afecta a esta decisión es la ubicación del vector de interrupciones. Puesto que el vector de interrupciones se encuentra a menudo en la parte baja de la memoria, los programadores tienden a situar también el sistema operativo en dicha zona. Por tanto, en este texto, sólo vamos a analizar la situación en la que el sistema operativo reside en la parte baja de la memoria, aunque las consideraciones aplicables al otro caso resultan similares.

Normalmente, querremos tener varios procesos de usuario residentes en memoria del mismo tiempo. Por tanto, tenemos que considerar cómo asignar la memoria disponible a los procesos que se encuentren en la cola de entrada, esperando a ser cargados en memoria. En este esquema de **asignación contigua de memoria**, cada proceso está contenido en una única sección contigua de memoria.

8.3.1 Mapeo de memoria y protección

Antes de seguir analizando la cuestión de la asignación de memoria, debemos hablar del tema de la conversión (*mapping*, mapeo) de memoria y la protección. Podemos proporcionar estas características utilizando un registro de reubicación, como se explica en la Sección 8.1.3, con un registro límite, como hemos visto en la Sección 8.1.1. El registro de reubicación contiene el valor de la dirección física más pequeña, mientras que el registro límite contiene el rango de las direcciones lógicas (por ejemplo, reubicación = 100040 y límite = 74600). Con los registros de reubicación y de límite, cada dirección lógica debe ser inferior al valor contenido en el registro límite; la MMU convertirá la dirección lógica *dinámicamente* sumándole el valor contenido en el registro de reubicación. Ésta dirección es la que se envía a la memoria (Figura 8.6).

Cuando el planificador de la CPU selecciona un proceso para su ejecución, el despachador carga en los registros de reubicación y de límite los valores correctos, como parte del proceso de cambio de contexto. Puesto que todas las direcciones generadas por la CPU se comparan con estos registros, este mecanismo nos permite proteger tanto al sistema operativo como a los programas

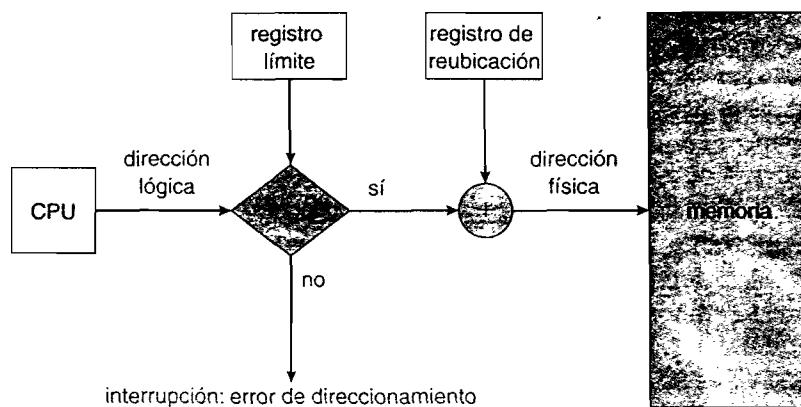


Figura 8.6 Soporte hardware para los registros de reubicación y de límite.

y datos de los otros usuarios de las posibles modificaciones que pudiera realizar este proceso en ejecución.

El esquema basado en registro de reubicación constituye una forma efectiva de permitir que el tamaño del sistema operativo cambie dinámicamente. Esta flexibilidad resulta deseable en muchas situaciones. Por ejemplo, el sistema operativo contiene código y espacio de búfer para los controladores de dispositivo; si un controlador de dispositivo (u otro servicio del sistema operativo) no se utiliza comúnmente, no conviene mantener el código y los datos correspondientes en la memoria, ya que podríamos utilizar dicho espacio para otros propósitos. Este tipo de código se denomina en ocasiones código **transitorio** del sistema operativo, ya que se carga y descarga según sea necesario. Por tanto, la utilización de este tipo de código modifica el tamaño del sistema operativo durante la ejecución del programa.

8.3.2 Asignación de memoria

Ahora estamos listos para volver de nuevo nuestra atención al tema de la asignación de memoria. Uno de los métodos más simples para asignar la memoria consiste en dividirla en varias **particiones** de tamaño fijo. Cada partición puede contener exactamente un proceso, de modo que el grado de multiprogramación estará limitado por el número de particiones disponibles. En este **método de particiones múltiples**, cuando una partición está libre, se selecciona un proceso de la cola de entrada y se lo carga en dicha partición. Cuando el proceso termina, la partición pasa a estar disponible para otro proceso. Este método (denominado MFT) fue usado originalmente por el sistema operativo IBM OS/360, pero ya no se lo utiliza. El método que vamos a describir a continuación es una generalización del esquema de particiones fijas (denominada MVT) y que se utiliza principalmente en entornos de procesamiento por lotes. Muchas de las ideas que aquí se presentan son también aplicables a los entornos de tiempo compartido en los que se utiliza un mecanismo de segmentación pura para la gestión de memoria (Sección 8.6).

En el esquema de particiones fijas, el sistema operativo mantiene una tabla que indica qué partes de la memoria están disponibles y cuáles están ocupadas. Inicialmente, toda la memoria está disponible para los procesos de usuario y se considera como un único bloque de gran tamaño de memoria disponible, al que se denomina **agujero**. Cuando llega un proceso y necesita memoria, busquemos un agujero lo suficientemente grande como para albergar este proceso. Si lo encontramos, sólo se asigna la memoria justa necesaria, manteniendo el resto de la memoria disponible para satisfacer futuras solicitudes.

A medida que los procesos entran en el sistema, se introducen en una cola de entrada. El sistema operativo toma en consideración los requisitos de memoria en cada proceso y la cantidad de memoria disponible a la hora de determinar a qué procesos se le asigna la memoria. Cuando asignamos espacio a un proceso, se carga en memoria y puede comenzar a competir por el uso de la CPU. Cuando un proceso termina, libera su memoria, que el sistema operativo podrá llenar a continuación con otro proceso extraído de la cola de entrada.

En cualquier momento determinado, tendremos una lista de tamaños de bloque disponibles y una cola de entrada de procesos. El sistema operativo puede ordenar la cola de entrada de acuerdo con algún algoritmo de planificación, asignándose memoria a los procesos hasta que finalmente, los requisitos de memoria del siguiente proceso ya no puedan satisfacerse, es decir, hasta que no haya ningún bloque de memoria (o agujero) disponible que sea lo suficientemente grande como para albergar al siguiente proceso. El sistema operativo puede entonces esperar hasta que haya libre un bloque de memoria lo suficientemente grande, o puede examinar el resto de la cola de entrada para ver si pueden satisfacerse los requisitos de memoria de algún otro proceso, que necesite un bloque de memoria menor.

En general, en cualquier momento determinado tendremos un *conjunto* de agujeros de diversos tamaños, dispersos por toda la memoria. Cuando llega un proceso y necesita memoria, el sistema explora ese conjunto en busca de un agujero que sea lo suficientemente grande como para albergar el proceso. Si el agujero es demasiado grande, se lo dividirá en dos partes, asignándose una parte al proceso que acaba de llegar y devolviendo la otra al conjunto de agujeros. Cuando el proceso termina, libera su bloque de memoria, que volverá a colocarse en el conjunto de agujeros.

Si el nuevo agujero es adyacente a otros agujeros, se combinan esos agujeros adyacentes para formar otros de mayor tamaño. En este punto, el sistema puede tener que comprobar si hay procesos esperando a que se les asigne memoria y si esta nueva memoria liberada y recombinada permite satisfacer las demandas de algunos de los procesos en espera.

Este procedimiento constituye un caso concreto del problema general de **asignación dinámica de espacio de almacenamiento**, que se ocupa de cómo satisfacer una solicitud de tamaño n a partir de una lista de agujeros libres. Hay muchas soluciones a este problema, y las estrategias más comúnmente utilizadas para seleccionar un agujero libre entre el conjunto de agujeros disponibles son las de **primer ajuste, mejor ajuste y peor ajuste**.

- **Primer ajuste.** Se asigna el *primer* agujero que sea lo suficientemente grande. La exploración puede comenzar desde el principio del conjunto de agujeros o en el punto en que hubiera terminado la exploración anterior. Podemos detener la exploración en cuanto encontramos un agujero libre que sea lo suficientemente grande.
- **Mejor ajuste.** Se asigna el agujero *más pequeño* que tenga el tamaño suficiente. Debemos explorar la lista completa, a menos que ésta esté ordenada según su tamaño. Esta estrategia hace que se genere el agujero más pequeño posible con la memoria que sobra en el agujero original.
- **Peor ajuste.** Se asigna el agujero *de mayor tamaño*. De nuevo, debemos explorar la lista completa, a menos que ésta esté ordenada por tamaños. Esta estrategia genera el agujero más grande posible con la memoria sobrante del agujero original, lo que puede resultar más útil que el agujero más pequeño generado con la técnica de mejor ajuste.

Las simulaciones muestran que tanto la estrategia de primer ajuste como la de mejor ajuste son mejores que la de peor ajuste en términos del tiempo necesario y de la utilización del espacio de almacenamiento. No está claro cuál de las dos estrategias (la de primer ajuste o la de mejor ajuste) es mejor en términos de utilización del espacio de almacenamiento, pero la estrategia de primer ajuste es, generalmente, más rápida de implementar.

8.3.3 Fragmentación

Tanto la estrategia de primer ajuste como la de mejor ajuste para la asignación de memoria sufren del problema denominado **fragmentación externa**. A medida que se cargan procesos en memoria y se los elimina, el espacio de memoria libre se descompone en una serie de fragmentos de pequeño tamaño. El problema de la fragmentación externa aparece cuando hay un espacio de memoria total suficiente como para satisfacer una solicitud, pero esos espacios disponibles no son contiguos; el espacio de almacenamiento está fragmentado en un gran número de pequeños agujeros. Este problema de fragmentación puede llegar a ser muy grave. En el peor de los casos, podríamos tener un bloque de memoria libre (o desperdiaciada) entre cada dos procesos; si todos estos pequeños fragmentos de memoria estuvieran en un único bloque libre de gran tamaño, podríamos ser capaces de ejecutar varios procesos más.

El que utilicemos el mecanismo de primer ajuste o el de mejor ajuste puede afectar al grado de fragmentación, porque la estrategia de primer ajuste es mejor para algunos sistemas, mientras que para otros resulta más adecuada la de mejor ajuste. Otro factor diferenciador es el extremo de un bloque libre que se asigne: ¿cuál es el fragmento que se deja como agujero restante, el situado en la parte superior o el situado en la parte inferior? Independientemente de qué algoritmo se utilice, la fragmentación externa terminará siendo un problema.

Dependiendo de la cantidad total de espacio de memoria y del tamaño medio de los procesos, esa fragmentación externa puede ser un problema grave o no. El análisis estadístico de la estrategia de primer ajuste revela, por ejemplo, que incluso con algo de optimización, si tenemos N bloques asignados, se perderán otros $0,5 N$ bloques debido a la fragmentación. En otras palabras, un tercio de la memoria puede no ser utilizable. Esta propiedad se conoce con el nombre de **regla del 50 por ciento**.

La fragmentación de memoria puede ser también interna, además de externa. Considere un esquema de asignación de particiones múltiples con un agujero de 18464 bytes. Suponga que el

siguiente proceso solicita 18462 bytes; si asignamos exactamente el bloque solicitado, nos quedará un agujero de 2 bytes. El espacio de memoria adicional requerido para llevar el control de este agujero será sustancialmente mayor que el propio agujero. La técnica general para evitar este problema consiste en descomponer la memoria física en bloques de tamaño fijo y asignar la memoria en unidades basadas en el tamaño de bloque. Con esta técnica, la memoria asignada a un proceso puede ser ligeramente superior a la memoria solicitada. La diferencia entre los dos valores será la **fragmentación interna**, es decir, la memoria que es interna a una partición pero que no está siendo utilizada.

Una solución al problema de la fragmentación externa consiste en la **compactación**. El objetivo es mover el contenido de la memoria con el fin de situar toda la memoria libre de manera contigua, para formar un único bloque de gran tamaño. Sin embargo, la compactación no siempre es posible. Si la reubicación es estática y se lleva a cabo en tiempo de ensamblado o en tiempo de carga, no podemos utilizar el mecanismo de la compactación; la compactación sólo es posible si la reubicación es dinámica y se lleva a cabo en tiempo de ejecución. Si las direcciones se reubican dinámicamente, la reubicación sólo requerirá mover el programa y los datos y luego cambiar el registro base para reflejar la nueva dirección base utilizada. Cuando la compactación es posible, debemos además determinar cuál es su coste. El algoritmo de compactación más simple consiste en mover todos los procesos hacia uno de los extremos de la memoria; de esta forma, todos los agujeros se moverán en la otra dirección, produciendo un único agujero de memoria disponible de gran tamaño. Sin embargo, este esquema puede ser muy caro de implementar.

Otra posible solución al problema de la fragmentación externa consiste en permitir que el espacio de direcciones lógicas de los procesos no sea contiguo, lo que hace que podamos asignar memoria física a un proceso con independencia de dónde esté situada dicha memoria. Hay dos técnicas complementarias que permiten implementar esta solución: la paginación (Sección 8.4) y la segmentación (Sección 8.6). Asimismo, estas técnicas pueden también combinarse (Sección 8.7).

8.4 Paginación

La **paginación** es un esquema de gestión de memoria que permite que el espacio de direcciones físicas de un proceso no sea contiguo. La paginación evita el considerable problema de encajar fragmentos de memoria de tamaño variable en el almacén de respaldo; la mayoría de los esquemas de gestión de memoria utilizados antes de la introducción de la paginación sufrían de este problema, que surgía debido a que, cuando era necesario proceder a la descarga de algunos datos o fragmentos de código que residieran en la memoria principal, tenía que encontrarse el espacio necesario en el almacén de respaldo. El almacén de respaldo también sufre los problemas de fragmentación que hemos mencionado en relación con la memoria principal, pero con la salvedad de que el acceso es mucho más lento, lo que hace que la compactación sea imposible. Debido a sus ventajas con respecto a los métodos anteriores, la mayoría de los sistemas operativos utilizan comúnmente mecanismos de paginación de diversos tipos.

Tradicionalmente, el soporte para la paginación se gestionaba mediante hardware. Sin embargo, algunos diseños recientes implementan los mecanismos de paginación integrando estrechamente el hardware y el sistema operativo, especialmente en los microprocesadores de 64 bytes.

8.4.1 Método básico

El método básico para implementar la paginación implica descomponer la memoria física en una serie de bloques de tamaño fijo denominados **marcos** y descomponer la memoria lógica en bloques del mismo tamaño denominados **páginas**. Cuando hay que ejecutar un proceso, sus páginas se cargan desde el almacén de respaldo en los marcos de memoria disponibles. El almacén de respaldo está dividido en bloques de tamaño fijo que tienen el mismo tamaño que los marcos de memoria.

La Figura 8.7 ilustra el soporte hardware para el mecanismo de paginación. Toda dirección generada por la CPU está dividida en dos partes: un **número de página (p)** y un **desplazamiento de página (d)**. El número de página se utiliza como índice para una tabla de páginas. La tabla de

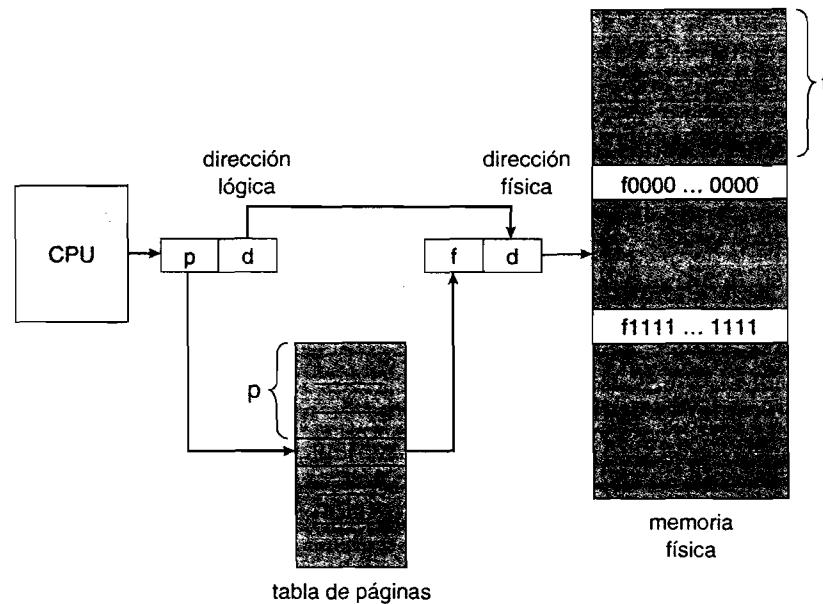


Figura 8.7 Hardware de paginación.

páginas contiene la dirección base de cada página en memoria física; esta dirección base se combina con el desplazamiento de página para definir la dirección de memoria física que se envía a la unidad de memoria. En la Figura 8.8 se muestra el modelo de paginación de la memoria.

El tamaño de página (al igual que el tamaño de marco) está definido por el hardware. El tamaño de la página es, normalmente, una potencia de 2, variando entre 512 bytes y 16 MB por página, dependiendo de la arquitectura de la computadora. La selección de una potencia de 2 como tamaño de página hace que la traducción de una dirección lógica a un número de página y a un desplazamiento de página resulte particularmente fácil. Si el tamaño del espacio de direcciones lógicas es 2^m y el tamaño de página es 2^n unidades de direccionamiento (bytes o palabras), entonces los $m - n$ bits de mayor peso de cada dirección lógica designarán el número de página, mien-

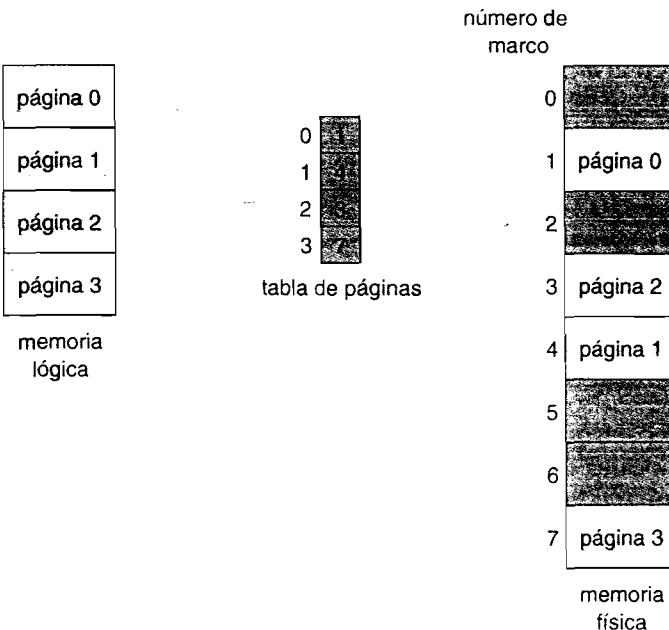


Figura 8.8 Modelo de paginación de la memoria lógica y física.

tras que los n bits de menor peso indicarán el desplazamiento de página. Por tanto, la dirección lógica tiene la estructura siguiente:

número de página	desplazamiento de página
p	d
$m - n$	n

donde p es un índice de la tabla de páginas y d es el desplazamiento dentro de la página.

Como ejemplo concreto (aunque minúsculo), considere la memoria mostrada en la Figura 8.9. Utilizando un tamaño de página de 4 bytes y una memoria física de 32 bytes (8 páginas), podemos ver cómo se hace corresponder la memoria física con la visión de la memoria que tiene el usuario. La dirección lógica 0 representa la página 0, desplazamiento 0. Realizando la indexación en la tabla de páginas, vemos que la página 0 se encuentra en el marco 5. Por tanto, la dirección lógica 0 se hace corresponder con la dirección física 20 (= $(5 \times 4) + 0$). La dirección lógica 3 (página 0, desplazamiento 3) se corresponde con la dirección física 23 (= $(5 \times 4) + 3$). La dirección lógica 4 corresponderá a la página 1, desplazamiento 0; de acuerdo con la tabla de páginas, la página 1 se corresponde con el marco 6. Por tanto, la dirección lógica 4 corresponde a la dirección física 24 (= $(6 \times 4) + 0$). La dirección lógica 13 corresponderá, por el mismo procedimiento, a la dirección física 9.

El lector puede haberse dado cuenta de que el propio esquema de paginación es una forma de reubicación dinámica. Cada dirección lógica es asignada por el hardware de paginación a alguna dirección física. La utilización de la paginación es similar al uso de una tabla de registros base (o de reubicación), uno por cada marco de memoria.

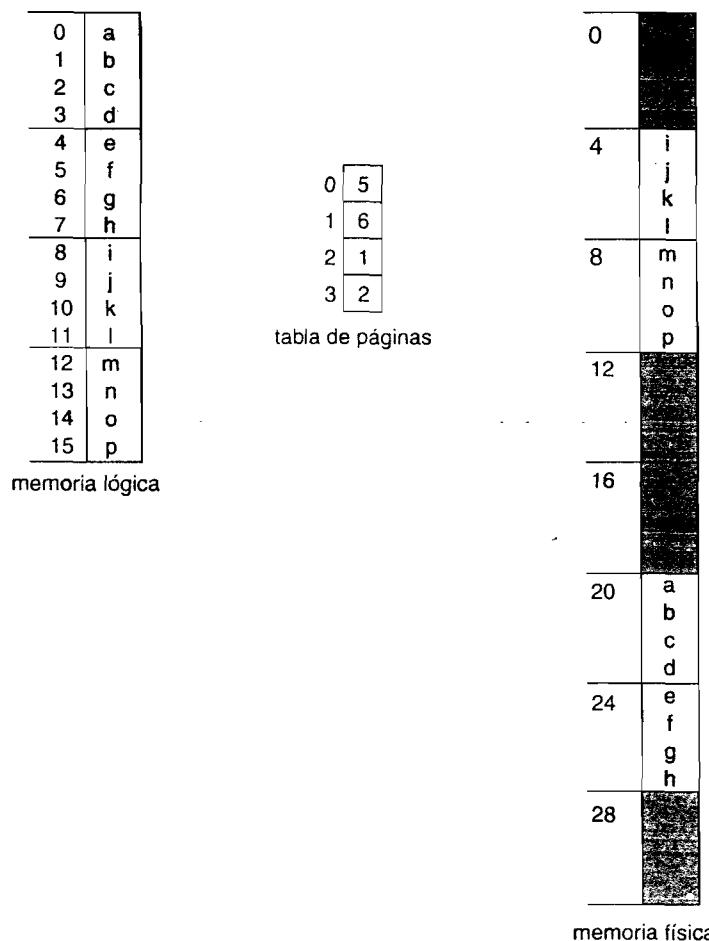


Figura 8.9 Ejemplo de paginación para una memoria de 32 bytes con páginas de 4 bytes.

Cuando usamos un esquema de paginación, no tenemos fragmentación externa: todos los marcos libres podrán ser asignados a un proceso que los necesite. Sin embargo, lo que sí podemos tener es un cierto grado de fragmentación interna, ya que los marcos se asignan como unidades y, si los requisitos de memoria de un proceso no coinciden exactamente con las fronteras de página, el *último* marco asignado puede no estar completamente lleno. Por ejemplo, si el tamaño de página es de 2.048 bytes, un proceso de 72.226 bytes necesitará 35 páginas completas más 1.086 bytes. A ese proceso se le asignarían 36 marcos, lo que daría como resultado una fragmentación interna de $2.048 - 1.086 = 962$ bytes. En el peor de los casos, un proceso podría necesitar n páginas más 1 byte, por lo que se le asignarían $n + 1$ marcos, dando como resultado una fragmentación interna de tamaño prácticamente igual a un marco completo.

Si el tamaño de los procesos es independiente del tamaño de las páginas, podemos esperar que la fragmentación interna sea, como promedio, igual a media página por cada proceso. Esta consideración sugiere que conviene utilizar tamaños de página pequeños. Sin embargo, se necesita dedicar recursos a la gestión de cada una de las entradas de la tabla de páginas, y estos recursos adicionales se reducen a medida que se incrementa el tamaño de la página. Asimismo, las operaciones de E/S de disco son más eficientes cuanto mayor sea el número de datos transferido (Capítulo 12). Generalmente, los tamaños de página utilizados en la práctica han ido creciendo con el tiempo, a medida que los procesos, los conjuntos de datos y la memoria principal han aumentado de tamaño. Hoy en día, las páginas utilizadas se encuentran normalmente entre 4 KB y 8 KB de tamaño y algunos sistemas soportan páginas de tamaño mayor aún. Algunos procesadores y algunos *kernels* soportan incluso tamaños de página múltiples. Por ejemplo, Solaris utiliza tamaños de página de 8 KB y 4 MB, dependiendo de los datos almacenados por las páginas. Los investigadores están actualmente desarrollando mecanismos de soporte con tamaños de página variables, que pueden ajustarse sobre la marcha.

Usualmente, cada entrada de una tabla de páginas tiene 4 bytes de longitud, pero también ese tamaño puede variar. Una entrada de 32 bits puede apuntar a una de 2^{32} marcos de página físicos. Si el tamaño del marco es de 4 KB, entonces un sistema con entradas de 4 bytes podrá direccionar 2^{44} bytes (o 16 TB) de memoria física.

Cuando llega un proceso al sistema para ejecutarlo, se examina su tamaño expresado en páginas. Cada página del proceso necesitará un marco. Por tanto, si el proceso requiere n páginas, deberá haber disponibles al menos n marcos en memoria. Si hay disponibles n marcos, se los asignará al proceso que acaba de llegar. La primera página del proceso se carga en uno de los marcos asignados y se incluye el número de marco en la tabla de páginas para este proceso. La siguiente página se carga en otro marco y su número de marco se coloca en la tabla de páginas, y así sucesivamente (Figura 8.10).

Un aspecto importante de la paginación es la clara separación existente entre la visión de la memoria que tiene el usuario y la memoria física real. El programa de usuario ve la memoria como un único espacio que sólo contiene ese programa. En la práctica, el programa de usuario está disperso por toda la memoria física, lo que también sucede para los restantes programas. La diferencia entre la visión de la memoria que tiene el usuario y la memoria física real se resuelve mediante el hardware de traducción de direcciones. Las direcciones lógicas se traducen a direcciones físicas y esta conversión queda oculta a ojos del usuario, siendo controlado por el sistema operativo. Observe que los procesos de usuario son incapaces, por definición, de acceder a la memoria que no les pertenece. No tienen forma de direccionar la memoria situada fuera de su tabla de páginas y esa tabla incluye únicamente aquellas páginas que sean propiedad del proceso.

Puesto que el sistema operativo está gestionando la memoria física, debe ser consciente de los detalles relativos a la asignación de la memoria física: qué marcos han sido asignados, qué marcos están disponibles, cuál es el número total de marcos, etc. Esta información se suele mantener en una estructura de datos denominada **tabla de marcos**. La tabla de marcos tiene una entrada por cada marco físico que indica si está libre o asignado y, en caso de estar asignado, a qué página de qué proceso o procesos ha sido asignado.

Además, el sistema operativo debe ser consciente de que los procesos de usuario operan en el espacio de usuario y de que todas las direcciones lógicas deben ser convertidas con el fin de generar direcciones físicas. Si un usuario hace una llamada al sistema (por ejemplo, para realizar una

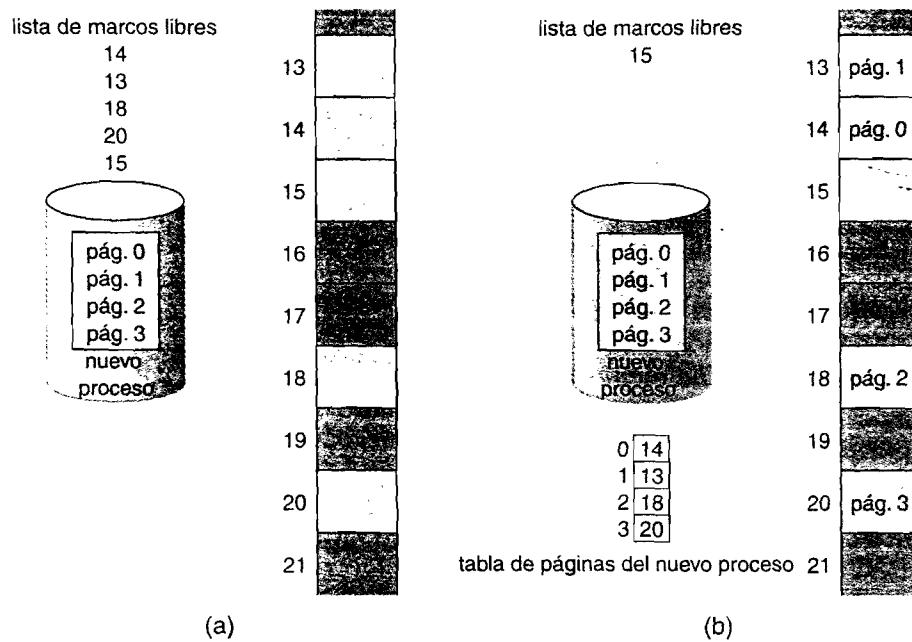


Figura 8.10 Marcos libres (a) antes de la asignación y (b) después de la asignación.

operación de E/S) y proporciona una dirección como parámetro (por ejemplo un búfer), dicha dirección deberá ser convertida para obtener la dirección física correcta. El sistema operativo mantiene una copia de la tabla de páginas de cada proceso, al igual que mantiene una copia del contador de instrucciones y de los contenidos de los registros. Esta copia se utiliza para traducir las direcciones lógicas a direcciones físicas cada vez que el sistema operativo deba convertir manualmente una dirección lógica a una dirección física. El despachador de la CPU utiliza también esa copia para definir la tabla de páginas hardware en el momento de asignar la CPU a un proceso. Los mecanismos de paginación incrementan, por tanto, el tiempo necesario para el cambio de contexto.

8.4.2 Soporte hardware

Cada sistema operativo tiene sus propios métodos para almacenar tablas de páginas. La mayoría de ellos asignan una tabla de páginas para cada proceso, almacenándose un puntero a la tabla de páginas, junto con los otros valores de los registros (como por ejemplo el contador de instrucciones), en el bloque de control del proceso. Cuando se le dice al despachador que inicie un proceso, debe volver a cargar los registros de usuario y definir los valores correctos de la tabla de páginas hardware a partir de la tabla almacenada de páginas de usuario.

La implementación hardware de la tabla de páginas puede hacerse de varias maneras. En el caso más simple, la tabla de páginas se implementa como un conjunto de registros dedicados. Estos registros deben construirse con lógica de muy alta velocidad, para hacer que el proceso de traducción de direcciones basado en la paginación sea muy eficiente. Cada acceso a la memoria debe pasar a través del mapa de paginación, por lo que la eficiencia es una de las consideraciones principales de implementación. El despachador de la CPU recarga estos registros al igual que recarga los registros restantes. Las instrucciones para cargar o modificar los registros de la tabla de páginas son, por supuesto, privilegiadas, de modo que sólo el sistema operativo puede cambiar el mapa de memoria. El sistema operativo PDP-11 de DEC es un ejemplo de este tipo de arquitectura; la dirección está compuesta de 16 bits y el tamaño de página es de 8 KB. La tabla de páginas contiene, por tanto ocho entradas que se almacenan en registros de alta velocidad.

El uso de registros para la tabla de páginas resulta satisfactorio si esta tabla es razonablemente pequeña (por ejemplo, 256 entradas). Sin embargo, la mayoría de las computadoras contemporáneas permiten que la tabla de páginas sea muy grande (por ejemplo, un millón de entradas);

para estas máquinas, el uso de registros de alta velocidad para incrementar la tabla de páginas no resulta factible. En lugar de ello, la tabla de páginas se mantiene en memoria principal, utilizando un **registro base de la tabla de páginas** (PTBR, page-table base register) para apuntar a la tabla de páginas. Para cambiar las tablas de páginas, sólo hace falta cambiar este único registro, reduciéndose así sustancialmente el tiempo de cambio de contexto.

El problema con esta técnica es el tiempo requerido para acceder a una ubicación de memoria del usuario. Si queremos acceder a la ubicación i , primero tenemos que realizar una indexación en la tabla de páginas, utilizando el valor del registro PTBR, desplazado según el número de página, para obtener el número de marco. Esta tarea requiere un acceso a memoria y nos proporciona el número de marco, que se combina con el desplazamiento de página para generar la dirección real. Sólo entonces podremos acceder a la ubicación deseada de memoria. Con este esquema, hacen falta dos accesos de memoria para acceder a un byte (uno para obtener la entrada de la tabla de páginas y otro para obtener el byte). Por tanto, el acceso a memoria se ralentiza dividiéndose a la mitad. En la mayor parte de las circunstancias, este retardo es intolerable; para eso, es mejor utilizar un mecanismo de intercambio de memoria.

La solución estándar a este problema consiste en utilizar una caché hardware especial de pequeño tamaño y con capacidad de acceso rápido, denominada **búfer de consulta de traducción** (TLB, translation look-aside buffer). El búfer TLB es una memoria asociativa de alta velocidad. Cada entrada del búfer TLB está compuesta de dos partes: una clave (o etiqueta) y un valor. Cuando se le presenta un elemento a la memoria asociativa, ese elemento se compara simultáneamente con todas las claves. Si se encuentra el elemento, se devuelve el correspondiente campo de valor. Esta búsqueda se realiza en paralelo de forma muy rápida, pero el hardware necesario es caro. Normalmente, el número de entradas del búfer TLB es pequeño; suele estar comprendido entre 64 y 1024.

El búfer TLB se utiliza con las tablas de página de la forma siguiente: el búfer TLB contiene sólo unas cuantas entradas de la tabla de páginas; cuando la CPU genera una dirección lógica, se presenta el número de página al TLB y si se encuentra ese número de página, su número de marco correspondiente estará inmediatamente disponible y se utilizará para acceder a la memoria. Toda esta operación puede llegar a ser tan sólo un 10 por ciento más lenta que si se utilizara una referencia a memoria no convertida.

Si el número de página no se encuentra en el búfer TLB (lo que se conoce con el nombre de **fallo de TLB**), es necesario realizar una referencia a memoria para consultar la tabla de páginas. Una vez obtenido el número de marco, podemos emplearlo para acceder a la memoria (Figura 8.11). Además, podemos añadir el número de página y el número de marco al TLB, para poder encontrar los correspondientes valores rápidamente en la siguiente referencia que se realice. Si el TLB ya está lleno, el sistema operativo deberá seleccionar una de las entradas para sustituirla. Las políticas de sustitución utilizadas van desde una política aleatoria hasta algoritmos que lo que hacen es sustituir la entrada menos recientemente utilizada (LRU, least recently used). Además, algunos búferes TLB permiten **cablear** las entradas, lo que significa que esas entradas no pueden eliminarse del TLB. Normalmente, las entradas del TLB correspondientes al código del *kernel* están cableadas.

Algunos búferes TLB almacenan **identificadores del espacio de direcciones** (ASID, address-space identifier) en cada entrada TLB. Cada identificador ASID identifica únicamente cada proceso y se utiliza para proporcionar mecanismos de protección del espacio de direcciones correspondiente a ese proceso. Cuando el TLB trata de resolver los números de página virtuales, verifica que el identificador ASID del proceso que actualmente se esté ejecutando se corresponda con el identificador ASID asociado con la página virtual. Si ambos identificadores no coinciden, el intento se trata como un fallo de TLB. Además de proporcionar protección del espacio de direcciones, los identificadores ASID permiten al TLB contener entradas simultáneamente para varios procesos distintos. Si el TLB no soporta la utilización de identificadores ASID distintos, cada vez que se seleccione una nueva tabla de páginas (por ejemplo, en cada cambio de contexto), será necesario **vaciar** (o borrar) el TLB para garantizar que el siguiente proceso que se ejecute no utilice una información incorrecta de traducción de direcciones. Si no se hiciera así, el TLB podría incluir entradas antiguas que contuvieran direcciones virtuales válidas pero que tuvieran asociadas direcciones físicas incorrectas o no válidas que se correspondan con el proceso anterior.

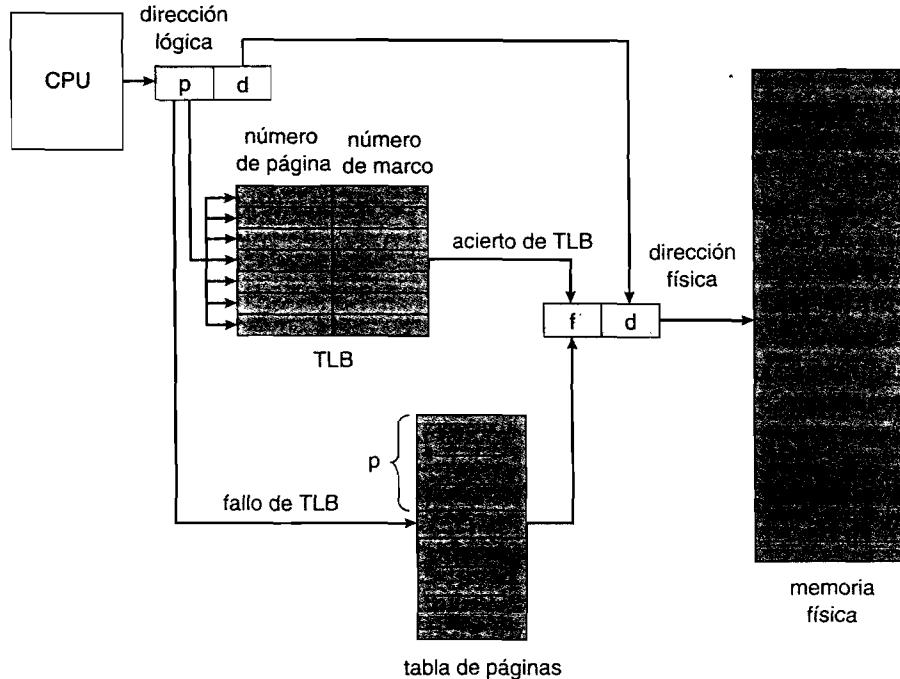


Figura 8.11 Hardware de paginación con TLB.

El porcentaje de veces que se encuentra un número de página concreto en el TLB se denomina **tasa de acierto**. Una tasa de acierto del 80 por ciento significa que hemos encontrado el número de página deseado en el TLB un 80 por ciento de las veces. Si se tarda 20 nanosegundos en consultar el TLB y 100 nanosegundos en acceder a la memoria, entonces un acceso a memoria convertida (mapeada) tardará 120 nanosegundos cuando el número de página se encuentre en el TLB. Si no conseguimos encontrar el número de página en el TLB (20 nanosegundos), entonces será necesario acceder primero a la memoria correspondiente a la tabla de páginas para extraer el número de marco (100 nanosegundos) y luego acceder al byte deseado en la memoria (100 nanosegundos), lo que nos da un total de 220 nanosegundos. Para calcular el **tiempo efectivo de acceso a memoria**, ponderamos cada uno de los casos según su probabilidad:

$$\begin{aligned} \text{tiempo efectivo de acceso} &= 0,80 \times 120 + 0,20 \times 220 \\ &= 140 \text{ nanosegundos.} \end{aligned}$$

En este ejemplo, vemos que se experimenta un aumento del 40 por ciento en el tiempo de acceso a memoria (de 100 a 140 nanosegundos).

Para una tasa de acierto del 98 por ciento, tendremos

$$\begin{aligned} \text{tiempo efectivo de acceso} &= 0,98 \times 120 + 0,02 \times 220 \\ &= 122 \text{ nanosegundos.} \end{aligned}$$

Esta tasa de acierto mejorada genera sólo un 22 por ciento de aumento en el tiempo de acceso. Exploraremos con más detalle el impacto de la tasa de acierto sobre el TLB en el Capítulo 9.

8.4.3 Protección

La protección de memoria en un entorno paginado se consigue mediante una serie de bits de protección asociados con cada marco. Normalmente, estos bits se mantienen en la tabla de páginas.

Uno de los bits puede definir una página como de lectura-escritura o de sólo lectura. Toda referencia a memoria pasa a través de la tabla de páginas con el fin de encontrar el número de marco correcto. Al mismo tiempo que se calcula la dirección física, pueden comprobarse los bits de pro-

tección para verificar que no se esté haciendo ninguna escritura en una página de sólo lectura. Todo intento de escribir una página de sólo lectura provocará una interrupción hardware al sistema operativo (o una violación de protección de memoria).

Podemos ampliar fácilmente este enfoque con el fin de proporcionar un nivel más fino de protección. Podemos diseñar un hardware que proporcione protección de sólo lectura, de lectura y escritura o de sólo ejecución, o podemos permitir cualquier combinación de estos accesos, proporcionando bits de protección separados para cada tipo de acceso. Los intentos ilegales provocarán una interrupción hardware hacia el sistema operativo.

Generalmente, se suele asociar un bit adicional con cada entrada de la tabla de páginas: un **bit válido-inválido**. Cuando se configura este bit como “válido”, la página asociada se encontrará en el espacio de direcciones lógicas del proceso y será, por tanto, una página legal (o válida). Cuando se configura el bit como “inválido”, la página no se encuentra en el espacio de direcciones lógicas del proceso. Las direcciones ilegales se capturan utilizando el bit válido-inválido. El sistema operativo configura este bit para cada página, con el fin de permitir o prohibir el acceso a dicha página.

Suponga, por ejemplo, que en un sistema con un espacio de direcciones de 14 bits (0 a 16383), tenemos un programa que sólo debe utilizar las direcciones comprendidas entre 0 y 10468. Dado un tamaño de página de 2 KB, tendremos la situación mostrada en la Figura 8.12. Las direcciones de las páginas 0, 1, 2, 3, 4 y 5 se convierten normalmente mediante la tabla de páginas, pero cualquier intento de generar una dirección en las páginas 6 o 7, sin embargo, se encontrará con el que el bit válido-inválido está configurado como inválido, por lo que la computadora generará una interrupción hacia el sistema operativo (referencia de página inválida).

Observe que este esquema genera un problema: puesto que el programa sólo se extiende hasta la dirección 10468, todas las referencias que se hagan más allá de dicha dirección serán ilegales. Sin embargo, las referencias a la página 5 están clasificadas como válidas, por lo que los accesos a las direcciones que van hasta la posición 12287 son válidas. Sólo las direcciones comprendidas entre 12288 y 16383 están marcadas como inválidas. Este problema surge como resultado del tamaño de página de 2 KB y refleja el problema de la fragmentación interna que sufren los mecanismos de paginación.

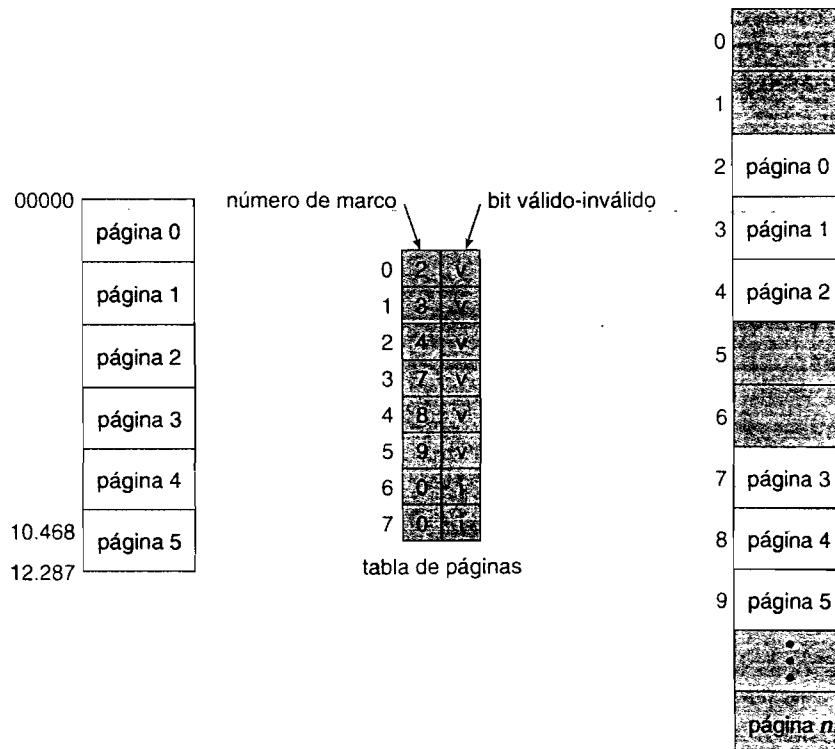


Figura 8.12 Bit válido (v)-inválido (i) en una tabla de páginas.

Los procesos raramente utilizan todo su rango de direcciones. De hecho, muchos procesos sólo emplean una pequeña fracción del espacio de direcciones que tiene disponible. Sería un desperdicio, en estos casos, crear una tabla de páginas con entradas para todas las páginas del rango de direcciones. La mayor parte de esta tabla permanecería sin utilizar, pero ocupando un valioso espacio de memoria. Algunos sistemas proporcionan mecanismos hardware especiales, en la forma de un **registro de longitud de la tabla de páginas** (PTLR, page-table length register), para indicar el tamaño de la tabla de páginas. Este valor se compara con cada dirección lógica para verificar que esa dirección se encuentre dentro del rango de direcciones válidas definidas para el proceso. Si esta comprobación fallara, se produciría una interrupción de error dirigida al sistema operativo.

8.4.4 Páginas compartidas

Una ventaja de la paginación es la posibilidad de *compartir* código común. Esta consideración es particularmente importante en un entorno de tiempo compartido. Considere un sistema que de soporte a 40 usuarios, cada uno de los cuales esté ejecutando un editor de texto. Si el editor de texto está compuesto por 150 KB de código y 50 KB de espacio de datos, necesitaremos 8000 KB para dar soporte a los 40 usuarios. Sin embargo, si se trata de **código reentrante** (o **código puro**), podrá ser compartido, como se muestra en la Figura 8.13. En ella podemos ver un editor de tres páginas (cada página tiene 50 KB de tamaño, utilizándose un tamaño de página tan grande para simplificar la figura) compartido entre los tres procesos. Cada proceso tiene su propia página de datos.

El código reentrant es código que no se auto-modifica; nunca cambia durante la ejecución. De esta forma, dos o más procesos pueden ejecutar el mismo código al mismo tiempo. Cada proceso tendrá su propia copia de los registros y del almacenamiento de datos, para albergar los datos requeridos para la ejecución del proceso. Por supuesto, los datos correspondientes a dos procesos distintos serán diferentes.

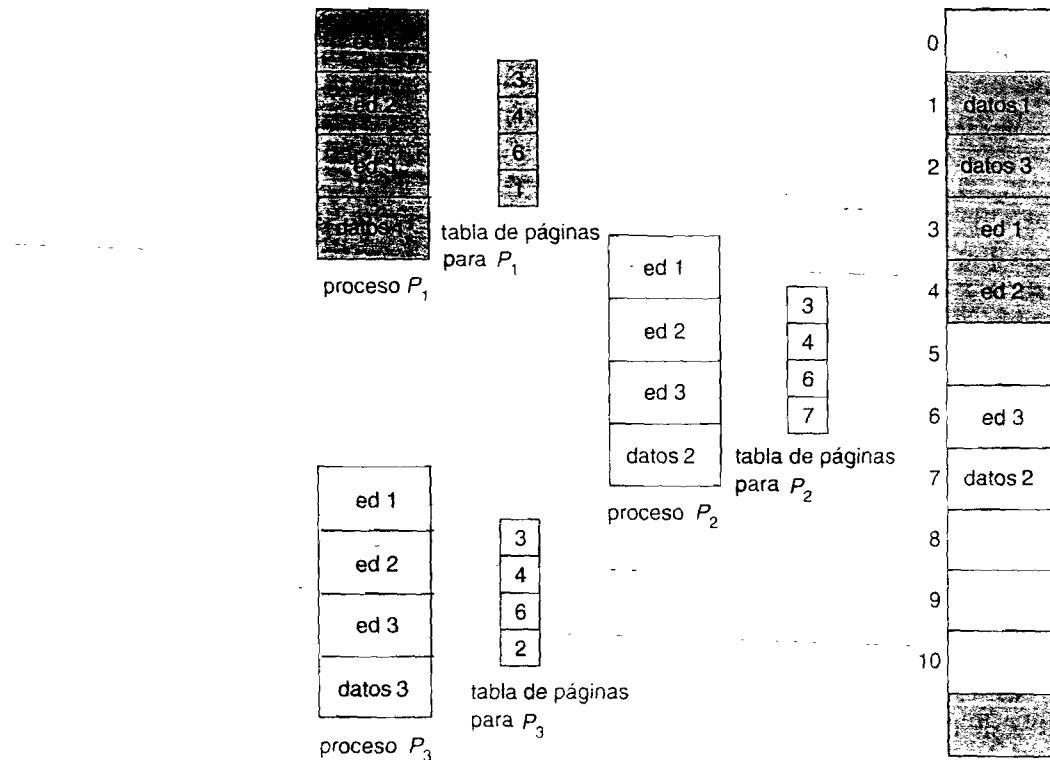


Figura 8.13 Compartición de código en un entorno paginado.

Sólo es necesario mantener en la memoria física una copia del editor. La tabla de páginas de cada usuario se corresponderá con la misma copia física del editor, pero las páginas de datos se harán corresponder con marcos diferentes. Así, para dar soporte a 40 usuarios, sólo necesitaremos una copia del editor (150 KB), más 40 copias del espacio de datos de 50 KB de cada usuario. El espacio total requerido será ahora de 2.150 KB, en lugar de 8.000 KB, lo que representa un ahorro considerable.

También pueden compartirse otros programas que se utilicen a menudo, como por ejemplo compiladores, sistemas de ventanas, bibliotecas de tiempo de ejecución, sistemas de base de datos, etc. Para poder compartirlo, el código debe ser reentrant. El carácter de sólo-lectura del código compartido no debe dejarse a expensas de la corrección de ese código, sino que el propio sistema operativo debe imponer esta propiedad.

La compartición de memoria entre procesos dentro de un sistema es similar a la compartición del espacio de direcciones de una tarea por parte de las hebras de ejecución, descrita en el Capítulo 4. Además, recuerde que en el Capítulo 3 hemos descrito la memoria compartida como un método de comunicación interprocesos. Algunos sistemas operativos implementan la memoria compartida utilizando páginas compartidas.

La organización de la memoria en páginas proporciona numerosos beneficios, además de permitir que varios procesos compartan las mismas páginas físicas. Hablaremos de algunos de los otros beneficios derivados de este esquema en el Capítulo 9.

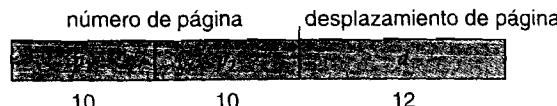
8.5 Estructura de la tabla de páginas

En esta sección, vamos a explorar algunas de las técnicas más comunes para estructurar la tabla de páginas.

8.5.1 Paginación jerárquica

La mayoría de los sistemas informáticos modernos soportan un gran espacio de direcciones lógicas (2^{32} a 2^{64}). En este tipo de entorno, la propia tabla de páginas llega a ser excesivamente grande. Por ejemplo, considere un sistema con un espacio lógico de direcciones de 32 bits. Si el tamaño de página en dicho sistema es de 4 KB (2^{12}), entonces la tabla de páginas puede estar compuesta de hasta un millón de entradas ($2^{32}/2^{12}$). Suponiendo que cada entrada esté compuesta por 4 bytes, cada proceso puede necesitar hasta 4 MB de espacio físico de direcciones sólo para la tabla de páginas. Obviamente, no conviene asignar la tabla de páginas de forma contigua en memoria principal. Una solución simple a este problema consiste en dividir la tabla de páginas en fragmentos más pequeños y podemos llevar a cabo esta división de varias formas distintas.

Una de esas formas consiste en utilizar un algoritmo de paginación de dos niveles, en el que la propia tabla de páginas está también paginada (Figura 8.14). Recuerde nuestro ejemplo de una máquina de 32 bits con un tamaño de página de 4 KB; una dirección lógica estará dividida en un número de página de 20 bits y un desplazamiento de página de 12 bits. Puesto que la tabla de páginas está paginada, el número de páginas se divide a su vez en un número de página de 10 bits y un desplazamiento de página de 10 bits. De este modo, una dirección lógica tendría la estructura siguiente:



donde p_1 es un índice a la tabla de páginas externa y p_2 es el desplazamiento dentro de la página de la tabla de páginas externa. El método de traducción de direcciones para esta arquitectura se muestra en la Figura 8.15. Puesto que la traducción de las direcciones se realiza comenzando por la tabla de páginas externa y continuando luego por la interna, este esquema también se conoce con el nombre de **tabla de páginas con correspondencia (mapeo) directa**.

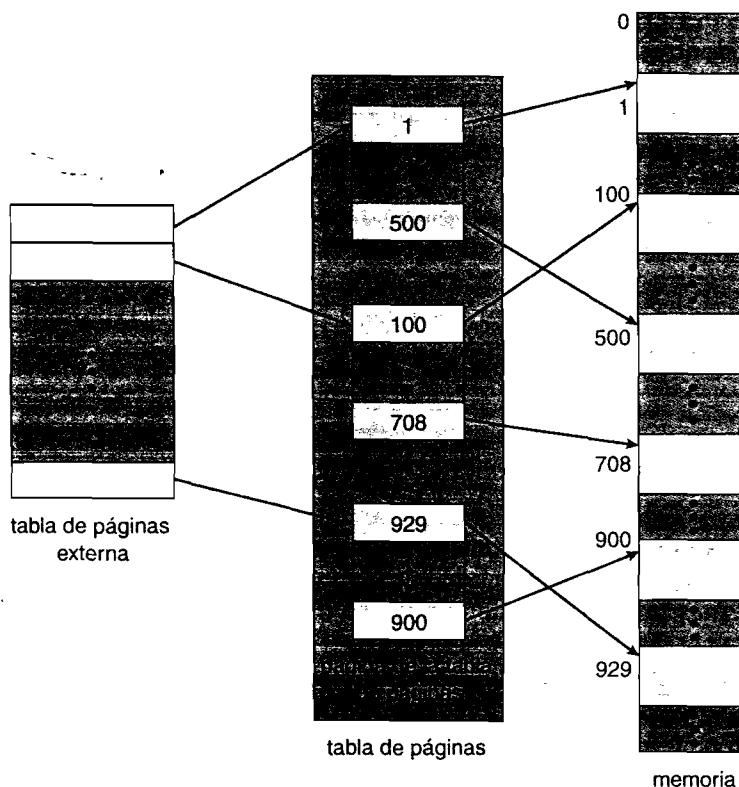


Figura 8.14 Un esquema de tabla de páginas en dos niveles.

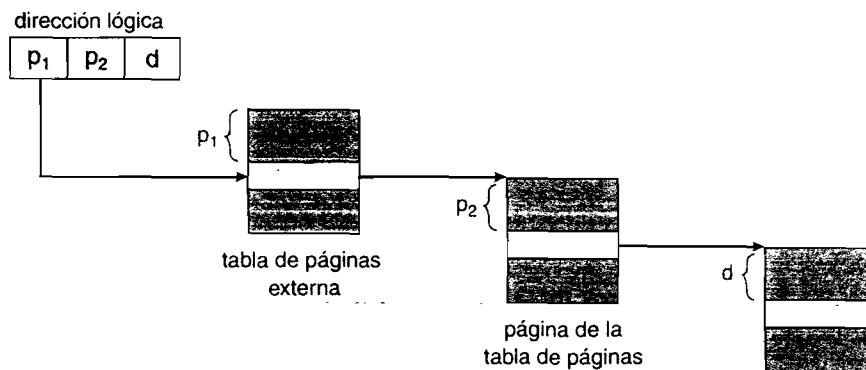
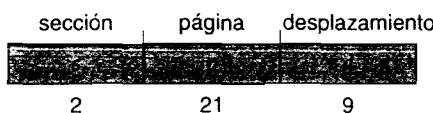


Figura 8.15 Traducción de una dirección para una arquitectura de paginación en dos niveles de 32 bits.

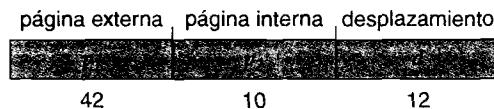
La arquitectura VAX también soporta una variante del mecanismo de paginación en dos niveles. El VAX es una máquina de 32 bits con un tamaño de página de 512 bytes. El espacio lógico de direcciones de un proceso se divide en cuatro secciones iguales, cada una de las cuales está compuesta de 2^{30} bytes. Cada sección representa una parte distinta del espacio lógico de direcciones de un proceso. Los primeros 2 bits de mayor peso de la dirección lógica designan la sección apropiada, los siguientes 21 bits representan el número lógico de página de dicha sección y los 9 bits finales representan un desplazamiento dentro de la página deseada.

Particionando la tabla de páginas de esta manera, el sistema operativo puede dejar particiones sin utilizar hasta que un proceso las necesite. Una dirección en la arquitectura VAX tiene la estructura siguiente:



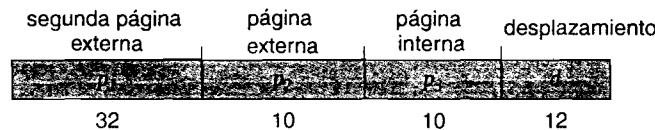
donde s designa el número de la sección, p es un índice a la tabla de páginas y d es el desplazamiento dentro de la página. Aunque se utiliza este esquema, el tamaño de una tabla de páginas de un sólo nivel para un proceso VAX que utilice una sección es de 2^{21} bits * 4 bytes por entrada = 8 MB. Para reducir aún más la utilización de memoria principal, la arquitectura VAX pagina las tablas de páginas de los procesos de usuario.

Para un sistema con un espacio lógico de direcciones de 4 bits, un esquema de paginación en dos niveles ya no resulta apropiado. Para ilustrar este punto, supongamos que el tamaño de página en dicho sistema fuera de 4 KB (2^{12}). En este caso, la tabla de páginas estaría compuesta hasta 2^{52} entradas. Si utilizáramos un esquema de paginación en dos niveles, las tablas de páginas internas podrían tener (como solución más fácil) una página de longitud, es decir, contener 4 entradas de 4 bytes. Las direcciones tendrían la siguiente estructura:



La tabla de páginas externas estará compuesta de 2^{42} entradas, es decir, 2^{44} bytes. La forma obvia para evitar tener una tabla tan grande consiste en dividir la tabla de páginas externa en fragmentos más pequeños. Esta técnica se utiliza también en algunos procesadores de 32 bits, para aumentar la flexibilidad y la eficiencia.

Podemos dividir la tabla de páginas externas de varias formas. Podemos paginar la tabla de páginas externa, obteniendo así un esquema de paginación en tres niveles. Suponga que la tabla de páginas externa está formada por páginas de tamaño estándar (2^{10} entradas o 2^{12} bytes); el espacio de direcciones de 64 bits seguirá siendo inmanejable:



La tabla de páginas externas seguirá teniendo 2^{34} bytes de tamaño.

El siguiente paso sería un esquema de paginación en cuatro niveles, en el que se paginaría también la propia tabla de páginas externas de segundo nivel. La arquitectura SPARC (con direccionamiento de 32 bits) permite el uso de un esquema de paginación en tres niveles, mientras que la arquitectura 68030 de Motorola, también de 32 bits, soporta un esquema de paginación en cuatro niveles.

Para las arquitecturas de 64 bits, las tablas de páginas jerárquicas se consideran, por lo general, inapropiadas. Por ejemplo, la arquitectura UltraSPARC de 64 bits requeriría siete niveles de paginación (un número prohibitivo de accesos a memoria) para traducir cada dirección lógica.

8.5.2 Tablas de páginas hash

Una técnica común para gestionar los espacios de direcciones superiores a 32 bits consiste en utilizar una **tabla hash** de páginas, donde el valor *hash* es el número de página virtual. Cada entrada de la tabla *hash* contiene una lista enlazada de elementos que tienen como valor *hash* una misma ubicación (con el fin de tratar las colisiones). Cada elemento está compuesto de tres campos: (1) el número de página virtual, (2) el valor del marco de página mapeado y (3) un punto del siguiente elemento de la lista enlazada.

El algoritmo funciona de la forma siguiente: al número de página virtual de la dirección virtual se le aplica una función *hash* para obtener un valor que se utiliza como índice para la tabla *hash*. El número de página virtual se compara con el campo 1 del primer elemento de la lista enlazada. Si hay una correspondencia, se utiliza el marco de página correspondiente (campo 2) para formar la dirección física deseada. Si no se produce una correspondencia, se exploran las siguientes entradas de la lista enlazada hasta encontrar el correspondiente número de página virtual. Este esquema se muestra en la Figura 8.16.

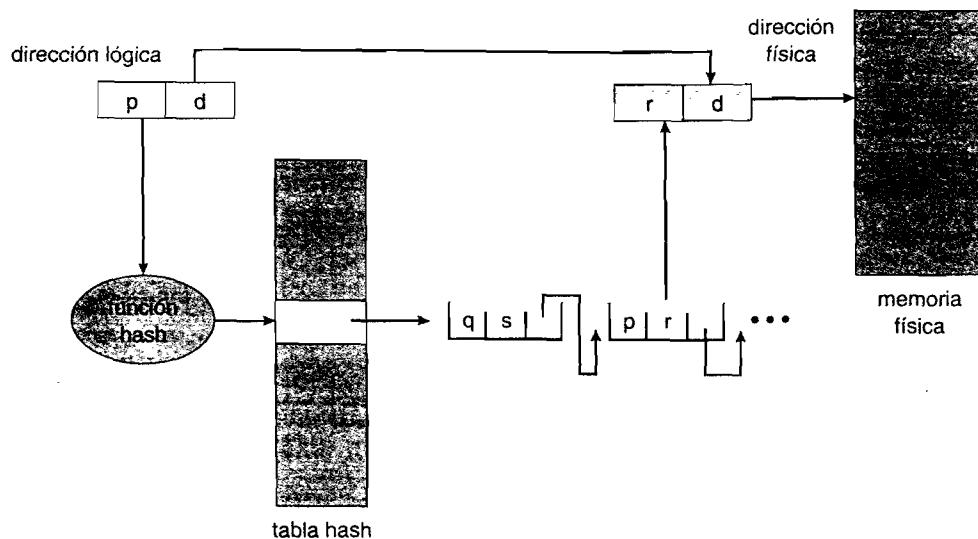


Figura 8.16 Tabla de páginas hash.

También se ha propuesto una variante de este esquema que resulta más adecuada para los espacios de direcciones de 64 bits. Esta variante utiliza **tablas de páginas en clúster**, que son similares a las tablas de páginas *hash*, salvo porque cada entrada de la tabla *hash* apunta a varias páginas (como por ejemplo 16) en lugar de a una sola página. De este modo, una única entrada de la tabla de páginas puede almacenar las correspondencias (mapeos) para múltiples marcos de página física. Las tablas de páginas en *cluster* son particularmente útiles para espacios de direcciones **dispersos** en los que las referencias a memoria son no continuas y están dispersas por todo el espacio de direcciones.

8.5.3 Tablas de páginas invertidas

Usualmente, cada proceso tiene una tabla de páginas asociada. La tabla de páginas incluye una entrada por cada página que el proceso esté utilizando (o un elemento por cada dirección virtual, independientemente de la validez de ésta). Esta representación de la tabla resulta natural, ya que los procesos hacen referencia a las páginas mediante las direcciones virtuales de éstas. El sistema operativo debe entonces traducir cada referencia a una dirección física de memoria. Puesto que la tabla está ordenada según la dirección virtual, el sistema operativo podrá calcular dónde se encuentra en la tabla la entrada de dirección física asociada y podrá utilizar dicho valor directamente. Pero una de las desventajas de este método es que cada tabla de página puede estar compuesta por millones de entradas. Estas tablas pueden ocupar una gran cantidad de memoria física, simplemente para controlar el modo en que se están utilizando otras partes de la memoria física.

Para resolver este problema, podemos utilizar una **tabla de páginas invertida**. Las tablas de páginas invertidas tienen una entrada por cada página real (o marco) de la memoria. Cada entrada está compuesta de la dirección virtual de la página almacenada en dicha ubicación de memoria real, e incluye información acerca del proceso que posee dicha página. Por tanto, en el sistema sólo habrá una única tabla de páginas y esa tabla sólo tendrá una entrada por cada página de memoria física. La Figura 8.17 muestra la operación de una tabla de páginas invertida. Compárela con la Figura 8.7, donde se muestra el funcionamiento de una tabla de páginas estándar. Las tablas de páginas invertidas requieren a menudo que se almacene un identificador del espacio de direcciones (Sección 8.4.2) en cada entrada de la tabla de páginas, ya que la tabla contiene usualmente varios espacios de direcciones distintos que se hacen corresponder con la memoria física. Almacenar el identificador del espacio de direcciones garantiza que cada página lógica de un proceso concreto se relacione con el correspondiente marco de página física. Como ejemplos de sistemas que utilizan las tablas de páginas invertidas podemos citar el PowerPC y la arquitectura UltraSPARC de 64 bits.

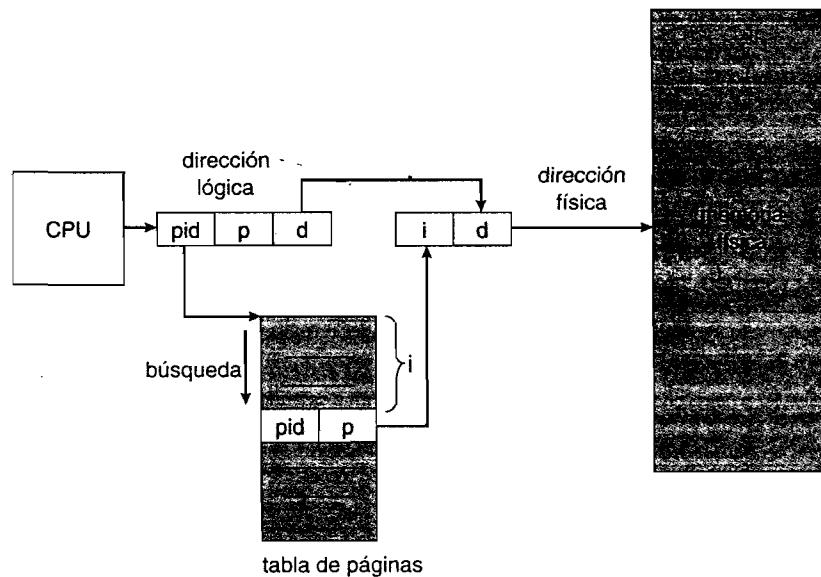


Figura 8.17 Tabla de páginas invertida.

Para ilustrar este método, vamos a describir una versión simplificada de la tabla de páginas invertida utilizada en IBM RT. Cada dirección virtual del sistema está compuesta por una tripleta $\langle \text{id-proceso}, \text{número-página}, \text{desplazamiento} \rangle$.

Cada entrada de la tabla de páginas invertida es una pareja $\langle \text{id-proceso}, \text{número-página} \rangle$, donde el identificador id-proceso asume el papel de identificador del espacio de direcciones. Cuando se produce una referencia a memoria, se presenta al subsistema de memoria una parte de la dirección virtual, compuesta por $\langle \text{id-proceso}, \text{número-página} \rangle$. Entonces, se explora la tabla de páginas invertida en busca de una correspondencia; si se encuentra esa correspondencia (por ejemplo, en la entrada i), se generará la dirección física $\langle i, \text{desplazamiento} \rangle$. Si no se encuentra ninguna correspondencia, querrá decir que se ha realizado un intento de acceso ilegal a una dirección.

Aunque este esquema permite reducir la cantidad de memoria necesaria para almacenar cada tabla de páginas, incrementa la cantidad de tiempo necesario para explorar la tabla cuando se produce una referencia a una página. Puesto que la tabla de páginas invertida está ordenada según la dirección física, pero las búsquedas se producen según las direcciones virtuales, puede que sea necesario explorar toda la tabla hasta encontrar una correspondencia y esta exploración consumiría demasiado tiempo. Para aliviar este problema, se utiliza una tabla hash, como la descrita en la Sección 8.5.2, con el fin de limitar la búsqueda a una sola entrada de la tabla de páginas o (como mucho) a unas pocas entradas. Por supuesto, cada acceso a la tabla hash añade al procedimiento una referencia a memoria, por lo que una referencia a memoria virtual requiere al menos dos lecturas de memoria real: una para la entrada de la tabla hash y otra para la tabla de páginas. Para mejorar la velocidad, recuerde que primero se explora el búfer TLB, antes de consultar la tabla hash.

Los sistemas que utilizan tablas de páginas invertidas tienen dificultades para implementar el concepto de memoria compartida. La memoria compartida se implementa usualmente en forma de múltiples direcciones virtuales (una para cada proceso que comparte la memoria), todas las cuales se hacen corresponder con una misma dirección física. Este método estándar no puede utilizarse con las tablas de páginas invertidas, porque sólo hay una única entrada de página virtual para cada página física y, como consecuencia, una misma página física no puede tener dos (o más) direcciones virtuales compartidas. Una técnica simple para resolver este problema consiste en permitir que la tabla de páginas sólo contenga una única correspondencia de una dirección virtual con la dirección física compartida. Esto significa que las referencias a direcciones virtuales que no estén asociadas darán como resultado un fallo de página.

8.6 Segmentación

Un aspecto importante de la gestión de memoria que se volvió inevitable con los mecanismos de paginación es la separación existente entre la vista que el usuario tiene de la memoria y la memoria física real. Como ya hemos comentado, la vista que el usuario tiene de la memoria no es la misma que la memoria física real, sino que esa vista del usuario se mapea sobre la memoria física. Este mapeo permite la diferenciación entre memoria lógica y memoria física.

8.6.1 Método básico

¿Piensan los usuarios en la memoria en forma de una matriz lineal de bytes, algunos de las cuales contienen instrucciones, mientras que otros contienen datos? La mayoría de nosotros diríamos que no, en lugar de ello, los usuarios prefieren ver la memoria como una colección de segmentos de tamaño variable, sin que exista necesariamente ninguna ordenación entre dichos segmentos (Figura 8.18).

Consideré la forma en que pensamos en un programa en el momento de escribirlo. Tendemos a considerarlo como un programa principal con un conjunto de métodos, procedimientos o funciones y que también puede incluir diversas estructuras de datos (objetos, matrices, pilas, variables, etc.). Para referirnos a cada uno de estos módulos o elementos de datos, utilizamos su nombre. Hablamos acerca de “la pila”, “la biblioteca matemática”, “el programa principal”, etc., sin preocuparnos de las direcciones de memoria que estos elementos puedan apuntar. No nos importa si la pila está almacenada antes o después de la función `Sqrt()`. Cada uno de estos segmentos tiene una longitud variable, que está definida intrínsecamente por el segmento que cumpla ese segmento del programa. Los elementos dentro de un segmento están identificados por su desplazamiento con respecto al inicio del segmento: la primera instrucción del programa, la séptima entrada de la pila, la quinta instrucción de la función `Sqrt()`, etc.

La **segmentación** es un esquema de gestión de memoria que soporta esta visión de la memoria que tienen los usuarios. Un espacio lógico de direcciones es una colección de segmentos y cada segmento tiene un nombre y una longitud. Las direcciones especifican tanto el nombre del segmento como el desplazamiento dentro de ese segmento. El usuario especifica, por tanto, cada dirección proporcionando dos valores: un nombre de segmento y un desplazamiento (compare

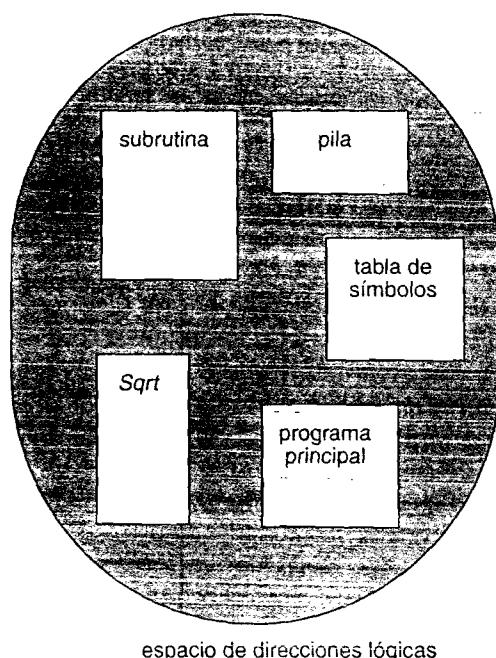


Figura 8.18 Vista de un programa por parte de un usuario.

este esquema con el de paginación, en el que el usuario especificaba una única dirección, que hardware particionaba en un número de páginas y en un desplazamiento, de forma invisible para el programador.

Por simplicidad de implementación, los segmentos están numerados y se hace referencia a ellos mediante un número de segmento, en lugar de utilizar un nombre de segmento. Así, una dirección lógica estará compuesta por una pareja del tipo:

<número-segmento, desplazamiento>.

Normalmente, el programa del usuario se compila y el compilador construye automáticamente los segmentos para reflejar el programa de entrada.

Un compilador C, podría crear segmentos separados para los siguientes elementos:

1. El código.
2. Las variables globales.
3. El cúmulo de memoria a partir del cual se asigna la memoria.
4. Las pilas utilizadas por cada hebra de ejecución.
5. La biblioteca C estándar.

También pueden asignarse segmentos separados a las bibliotecas que se monten en tiempo de compilación. El cargador tomará todos estos segmentos y les asignará los correspondientes números de segmento.

8.6.2 Hardware

Aunque el usuario puede ahora hacer referencia a los objetos del programa utilizando una dirección bidimensional, la memoria física real continúa siendo, por supuesto, una secuencia unidimensional de bytes. Por tanto, deberemos definir una implementación para mapear las direcciones bidimensionales definidas por el usuario sobre las direcciones físicas unidimensionales. Este mapeo se lleva a cabo mediante una tabla de segmentos. Cada entrada de la tabla de segmentos tiene una dirección base del segmento y un límite del segmento. La dirección base del segmento

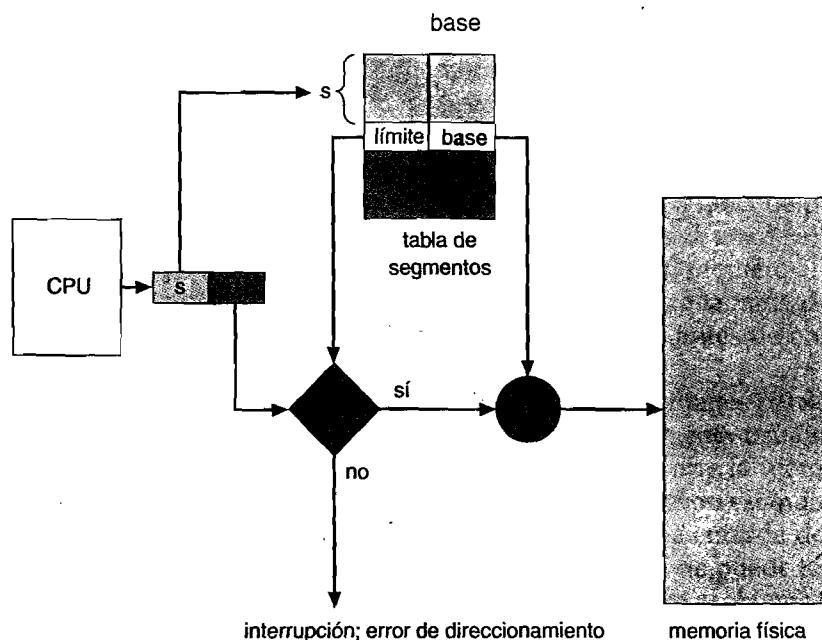


Figura 8.19 Hardware de segmentación.

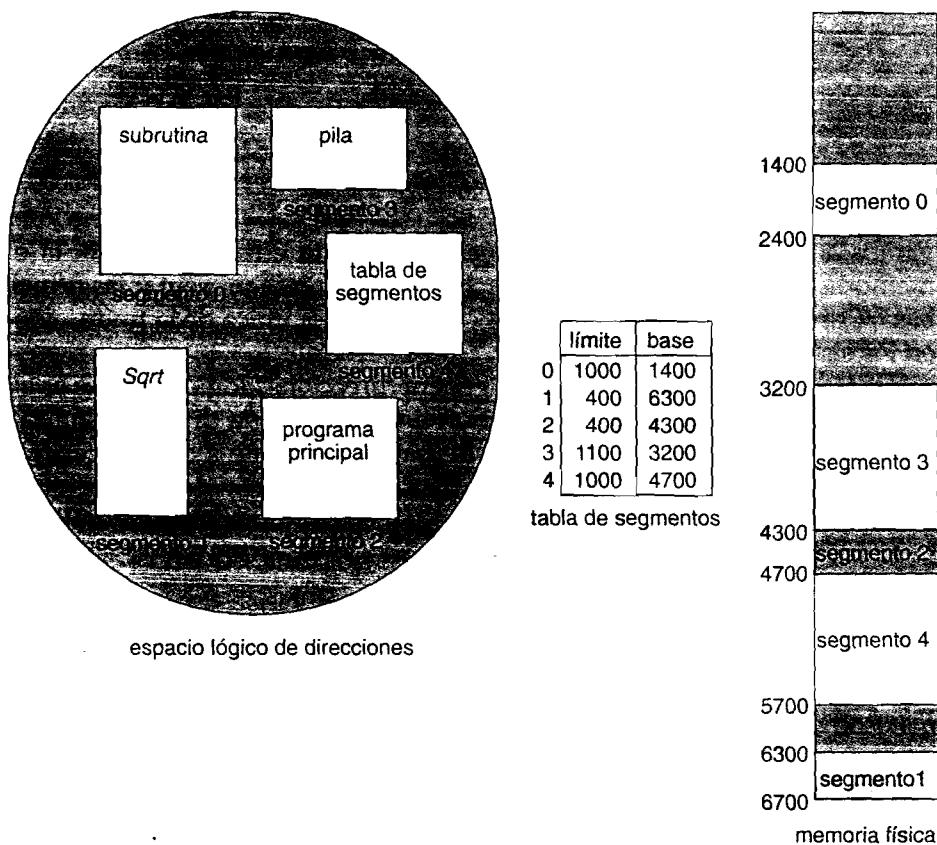


Figura 8.20 Ejemplo de segmentación.

contiene la dirección física inicial del lugar donde el segmento reside dentro de la memoria, mientras que el límite del segmento especifica la longitud de éste.

El uso de una tabla de segmentos se ilustra en la Figura 8.19. Una dirección lógica estará compuesta de dos partes: un número de segmento, s , y un desplazamiento dentro de ese segmento, d . El número de segmento se utiliza como índice para la tabla de segmentos. El desplazamiento d de la dirección lógica debe estar comprendido entre 0 y el límite del segmento; si no lo está, se producirá una interrupción hacia el sistema operativo (intento de direccionamiento lógico más allá del final del segmento). Cuando un desplazamiento es legal, se lo suma a la dirección base del segmento para generar la dirección de memoria física del byte deseado. La tabla de segmentos es, por tanto, esencialmente una matriz de parejas de registros base-límite.

Como ejemplo, considere la situación mostrada en la Figura 8.20. Tenemos cinco segmentos, numerados de 0 a 4. Los segmentos se almacenan en la memoria física de la forma que se muestra. La tabla de segmentos dispone de una tabla separada para cada segmento, en la que se indica la dirección inicial del segmento en la memoria física (la base) y la longitud de ese segmento (el límite). Por ejemplo, el segmento 2 tiene 400 bytes de longitud y comienza en la ubicación 4300. Por tanto, una referencia al byte 53 del segmento 2 se corresponderá con la posición $4300 + 53 = 4353$. Una referencia al segmento 3, byte 852, se corresponderá con la posición 3200 (la base del segmento 3) + 852 = 4052. Una referencia al byte 1222 del segmento 0 provocaría una interrupción hacia el sistema operativo, ya que este segmento sólo tiene 1000 bytes de longitud.

8.7 Ejemplo: Intel Pentium

Tanto la paginación como la segmentación tienen ventajas y desventajas. De hecho, algunas arquitecturas proporcionan ambos mecanismos. En esta sección, vamos a analizar la arquitectura Intel Pentium, que soporta tanto una segmentación pura como una segmentación con paginación. No

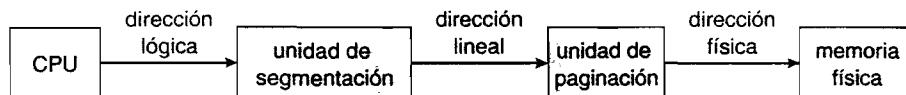


Figura 8.21 Traducción de direcciones lógicas a direcciones físicas en el Pentium.

vamos a proporcionar una descripción completa en este texto de la estructura de gestión de memoria del Pentium, sino que simplemente vamos a presentar las principales ideas en las que se basa. Concluiremos nuestro análisis con una panorámica de la traducción de direcciones en Linux en los sistemas Pentium.

En los sistemas Pentium, la CPU genera direcciones lógicas que se entregan a la unidad de segmentación. Ésta produce una dirección lineal para cada dirección lógica y esa dirección lineal se entrega a continuación a la unidad de paginación, que a su vez genera la dirección física de la memoria principal. Así, las unidades de segmentación y de paginación forman el equivalente de una unidad de gestión de memoria (MMU, memory-management unit). Este esquema se representa en la Figura 8.21.

8.7.1 Mecanismo de segmentación en el Pentium

La arquitectura Pentium permite que un segmento tenga un tamaño de hasta 4 GB y el número máximo de segmentos por cada proceso es de 16 KB. El espacio lógico de direcciones de un proceso está dividido en dos particiones. La primera partición está compuesta de hasta 8 KB segmentos que son privados de ese proceso; la segunda partición está compuesta de hasta 8 KB segmentos, compartidos entre todos los procesos. La información acerca de la primera partición se almacena en la **tabla local de descriptores** (LDT, local descriptor table); la información acerca de la segunda partición se almacena en la **tabla global de descriptores** (GDT, global descriptor table). Cada entrada de la LDT y de la GDT está compuesta por un descriptor de segmento de 8 bytes que incluye información detallada acerca de ese segmento concreto, incluyendo la posición base y el límite del segmento.

La dirección lógica es una pareja (selector, desplazamiento), donde el selector es un número de 16 bits,



en el que *s* designa el número de segmento, *g* indica si el segmento está en la GDT o en la LDT y *p* son dos bits de protección. El desplazamiento es un número de 32 bits que especifica la ubicación del byte (o palabra) dentro del segmento en cuestión.

La máquina tiene seis registros de segmento, lo que permite que un proceso pueda direccionar en cualquier momento concreto hasta seis segmentos distintos. También tiene seis registros de microprograma de 8 bytes para almacenar los correspondientes descriptores de la LDT o de la GDT. Esta caché permite al Pentium tener que leer el descriptor desde la memoria para cada referencia de memoria.

La dirección lineal en el Pentium tiene 32 bits de longitud y está formada de la manera siguiente: el registro de segmento apunta a la entrada apropiada de la LDT o de la GDT; la información de base y de límite acerca del segmento en cuestión se utiliza para generar una dirección lineal. En primer lugar, se utiliza el límite para comprobar la validez de la dirección. Si la dirección no es válida, se genera un fallo de memoria, lo que provoca una interrupción dirigida al sistema operativo. Si es válida, entonces se suma el valor del desplazamiento al valor de la base, lo que da como resultado una dirección lineal de 32 bits; esto se muestra en la Figura 8.22. En la siguiente sección, vamos a analizar cómo transforma la unidad de paginación esta dirección lineal en una dirección física.

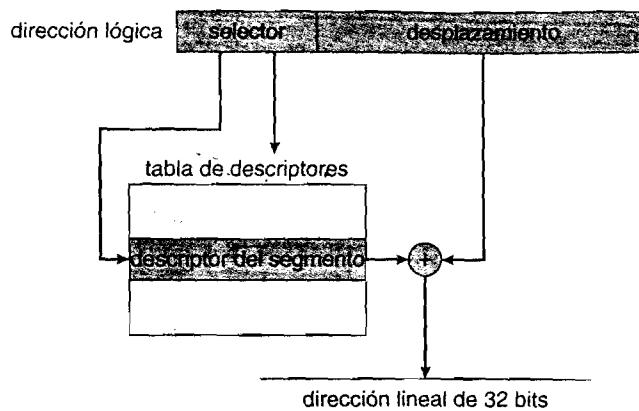
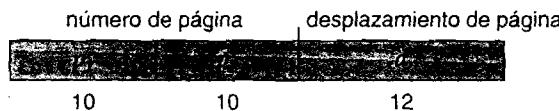


Figura 8.22 Segmentación en el Intel Pentium

8.7.2 Mecanismo de paginación en el Pentium

La arquitectura Pentium permite utilizar un tamaño de página de 4 KB o de 4 MB. Para las páginas de 4 KB, el Pentium utiliza un esquema de paginación en dos niveles en el que la estructura de la dirección lineal de 32 bits es la siguiente:



El esquema de traducción de direcciones para esta arquitectura es similar al que se muestra en la Figura 8.15. En la Figura 8.23 se muestra el mecanismo de traducción de direcciones del Intel Pentium con mayor detalle. Los diez bits de mayor peso hacen referencia a una entrada en la tabla de páginas externa, que en el Pentium se denomina **directorio de páginas** (el registro CR3 apunta al directorio de páginas para el proceso actual). La entrada en el directorio de páginas apunta a una tabla de paginación interna que está indexada según el contenido de los diez bits más internos de la dirección lineal. Finalmente, los bits 0-11 de menor peso indican el desplazamiento dentro de la página de 4 KB a la que se apunta desde la tabla de páginas.

Una de las entradas del directorio de páginas es el indicador Page Size que (si está activado) indica que el tamaño del marco de página es de 4 MB y no de 4 KB, que es el valor normal. Si este indicador está activado, el directorio de páginas apunta directamente al marco de página de 4 MB, puenteando la tabla de páginas interna, en cuyo caso los 22 bits de menor peso de la dirección lineal constituyen el desplazamiento dentro del marco de página de 4 MB.

Para mejorar la eficiencia de utilización de la memoria física, las tablas de páginas del Intel Pentium pueden cargarse y descargarse de disco. En este caso, se utiliza un bit válido-inválido dentro de la entrada del directorio de páginas para indicar si la tabla a la que apunta la entrada se encuentra en memoria o en disco. Si la tabla se encuentra en el disco, el sistema operativo puede usar los otros 31 bits para especificar la ubicación de la tabla dentro del disco; como consecuencia, la tabla puede cargarse en memoria cuando sea necesario.

8.7.3 Linux en los sistemas Pentium

Como ilustración, considere un sistema operativo Linux ejecutándose sobre la arquitectura Intel Pentium. Puesto que Linux está diseñado para ejecutarse sobre diversos procesadores (muchos de los cuales sólo pueden proporcionar un soporte limitado para la segmentación), Linux no depende de los mecanismos de segmentación y los utiliza de forma mínima. En el Pentium, Linux sólo utiliza seis segmentos:

1. Un segmento para el código del kernel.

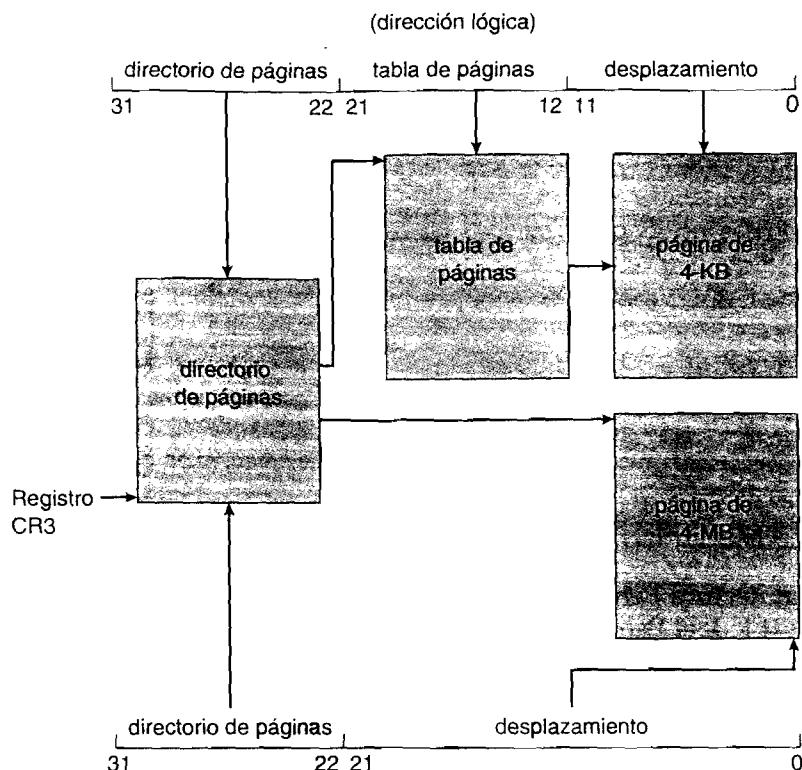


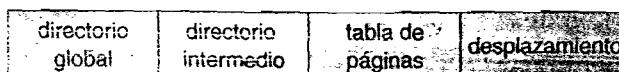
Figura 8.23 Mecanismo de paginación en la arquitectura Pentium.

2. Un segmento para los datos del *kernel*.
3. Un segmento para el código de usuario.
4. Un segmento para los datos de usuario.
5. Un segmento de estado de tareas (TSS, task-state segment).
6. Un segmento LDT predeterminado.

Los segmentos para el código de usuario y los datos de usuario son compartidos por todos los procesos que se ejecutan en modo usuario. Esto es posible porque todos los procesos utilizan el mismo espacio lógico de direcciones y todos los descriptores de segmentos están almacenados en la tabla global de descriptores (GDT). Además, cada proceso tiene su propio segmento de estado de tareas (TSS) y el descriptor de este segmento está almacenado en la GDT. El TSS se utiliza para almacenar el contexto hardware de cada proceso durante los cambios de contexto. El segmento LDT predeterminado está, normalmente, compartido por todos los procesos y no suele utilizarse. Sin embargo, si un proceso requiere su propia LDT, puede crear una y utilizar en lugar de la LDT predeterminada.

Como se indica, cada selector de segmento incluye un campo de dos bits de protección. Por tanto, el Pentium permite cuatro niveles de protección distintos. De estos cuatro niveles, Linux sólo reconoce dos: el modo usuario y el modo *kernel*. Aunque el Pentium emplea un modelo de paginación en dos niveles, Linux está diseñado para ejecutarse sobre diversas plataformas hardware, muchas de las cuales son plataformas en 64 bits en las que no resulta posible utilizar una paginación en dos niveles. Por tanto, Linux ha adoptado una estrategia de paginación en tres niveles que funcionan adecuadamente tanto para arquitecturas de 32 bits como de 64 bits.

La dirección lineal de Linux se descompone en las siguientes cuatro partes:



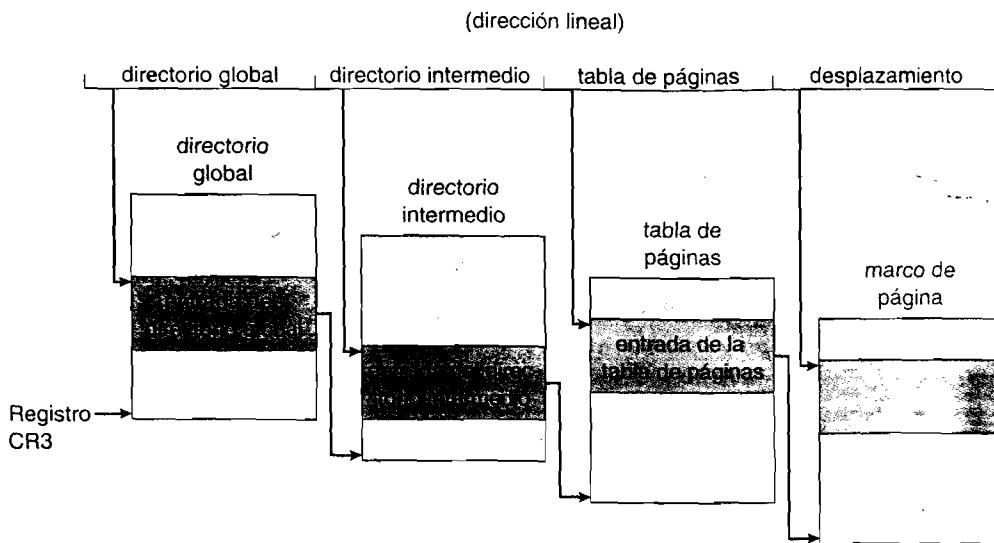


Figura 8.24 Modelo de paginación en Linux.

La Figura 8.24 muestra el modelo de paginación en tres niveles de Linux.

El número de bits de cada parte de la dirección lineal varía de acuerdo con la arquitectura. Sin embargo, como hemos descrito anteriormente en esta sección, la arquitectura Pentium sólo utiliza un modelo de paginación en dos niveles. Entonces, ¿cómo aplica Linux su modelo de tres niveles en el Pentium? En este caso, el tamaño del directorio intermedio es de cero bits, puenteando en la práctica dicho directorio.

Cada tarea en Linux tiene su propio conjunto de tablas de páginas y (al igual que en la Figura 8.23) el registro CR3 apunta al directorio global correspondiente a la tarea que se esté ejecutando actualmente. Durante un cambio de contexto, el valor del registro CR3 se guarda y se restaura en los segmentos TSS de las tareas implicadas en el cambio de contexto.

8.8 Resumen

Los algoritmos de gestión de memoria para los sistemas operativos multiprogramados van desde la técnica simple de los sistemas monousuario hasta los mecanismos de segmentación paginada. La consideración más importante a la hora de determinar el método que hay que utilizar en un sistema completo es el hardware proporcionado. Cada dirección de memoria generada por la CPU puede ser comprobada para verificar su legalidad y debe también, posiblemente, ser mapeada sobre una dirección física. Esas comprobaciones no pueden implementarse (de manera eficiente) por software; por tanto, estamos constreñidos por el hardware disponible.

Los diversos algoritmos de gestión de memoria (asignación contigua, paginación, segmentación y la combinación de los mecanismos de paginación y segmentación) difieren en muchos aspectos. A la hora de comparar las diferentes estrategias de gestión de memoria, utilizamos las siguientes consideraciones:

- **Soporte hardware.** Un simple registro base o una pareja de registros base-límite resulta suficiente para los esquemas de partición simple y múltiple, mientras que la paginación y la segmentación necesitan tablas de mapeo para definir el mapa de direcciones.
- **Rendimiento.** A medida que se incrementa la complejidad del algoritmo de gestión de memoria, el tiempo requerido para mapear una dirección lógica sobre una dirección física también aumenta. Para los sistemas simples, sólo necesitamos realizar operaciones de comparación o de sumas con la dirección lógica, operaciones que son rápidas de realizar. La paginación y la segmentación pueden ser igualmente rápidas si la tabla de mapeo está implementada en registros de alta velocidad. Sin embargo, si la tabla está en memoria, los

accesos a la memoria de usuario pueden degradarse de manera sustancial. Un TLB p...
reducir esa degradación hasta un nivel aceptable.

- **Fragmentación.** Un sistema multiprogramado tendrá, generalmente, un mayor rendimiento y una mayor eficiencia si tiene un alto grado de multiprogramación. Para un conjunto de procesos, sólo podemos incrementar el nivel de multiprogramación haciendo quepan más procesos en memoria. Para poder conseguir esto, necesitamos reducir memoria desperdiciada debido a la fragmentación. Los sistemas con unidades de asignación de tamaño fijo, como los esquemas de partición simple y de paginación, presentan problema de la fragmentación interna. Los sistemas con unidades de asignación de tamaño variable, como los esquemas de partición múltiple y de segmentación, presentan el problema de la fragmentación externa.
- **Reubicación.** Una solución al problema de la fragmentación externa es la compactación. Mecanismo de compactación implica mover un programa en memoria de tal forma que no note el cambio. Este proceso requiere que las direcciones lógicas se reubiquen dinámicamente en tiempo de ejecución. Si las direcciones sólo se reubican en tiempo de carga, podremos compactar el espacio de almacenamiento.
- **Intercambio.** Podemos añadir mecanismos de intercambio a cualquier algoritmo. A intervalos determinados con el sistema operativo, usualmente dictados por las políticas de planificación de la CPU, los procesos se copian desde la memoria principal a un almacenamiento de respaldo y luego se vuelven a copiar de vuelta a la memoria principal. Este esquema permite ejecutar más procesos de los que cabrían en memoria en cualquier instante determinado.
- **Compartición.** Otro medio de incrementar el nivel de multiprogramación consiste en compartir el código y los datos entre diferentes usuarios. La compartición requiere, generalmente, que se utilice un mecanismo de paginación o de segmentación, con el fin de disponer de pequeños paquetes de información (páginas o segmentos) que puedan ser compartidos. La compartición es un modo de ejecutar múltiples procesos con una cantidad limitada de memoria, pero los programas y datos compartidos deben ser diseñados con sumo cuidado.
- **Protección.** Si se proporciona un mecanismo de paginación o segmentación, las diferentes secciones de un programa de usuario pueden declararse como de sólo ejecución, de sólo lectura o de lectura-escritura. Esta restricción es necesaria para el código o los datos compartidos y resulta, generalmente, útil en cualquier caso para proporcionar un mecanismo simple de comprobación en tiempo de ejecución, con el que evitar errores de programación comunes.

Ejercicios

- 8.1 Explique la diferencia entre los conceptos de fragmentación interna y externa.
- 8.2 Considere el siguiente proceso de generación de archivos binarios. Utilizamos un compilador para generar el código objeto de los módulos individuales y luego usamos un editor de montaje para combinar múltiples módulos objetos en un único programa binario. ¿En qué forma el editor de montaje cambia el acoplamiento de las instrucciones y los datos a direcciones de memoria? ¿Qué información debe pasar el compilador al editor de montaje para facilitar las tareas de acoplamiento de memoria de éste?
- 8.3 Dadas cinco particiones de memoria de 100 KB, 500 KB, 200, KB, 300 KB y 600 KB (en este orden), ¿cómo situarían en memoria una serie de procesos de 212 KB, 417 KB, 112 KB y 426 KB (por este orden) con los algoritmos de primer ajuste, mejor ajuste y peor ajuste? ¿Qué algoritmo hace el uso más eficiente de la memoria?
- 8.4 La mayoría de los sistemas permiten a los programas asignar más memoria a su espacio de direcciones durante la ejecución. Como ejemplo de ese tipo de asignación de memoria tenemos los datos asignados en los segmentos de los programas dedicados a cúmulo de memo-

ria. ¿Qué se necesitaría para soportar la asignación dinámica de memoria en los siguientes esquemas?

- a. asignación contigua de memoria
 - b. segmentación pura
 - c. paginación pura
- 8.5 Compare los esquemas de organización de la memoria principal basados en una asignación continua de memoria, en una segmentación pura y en una paginación pura con respecto a las siguientes cuestiones:
- a. fragmentación externa
 - b. fragmentación interna
 - c. capacidad de compartir código entre procesos
- 8.6 En un sistema con paginación, un proceso no puede acceder a una zona de memoria que no sea de su propiedad. ¿Por qué? ¿Cómo podría el sistema operativo permitir el acceso a otras zonas de memoria? ¿Por qué debería o por qué no debería?
- 8.7 Compare el mecanismo de paginación con el de segmentación con respecto a la cantidad de memoria requerida por las estructuras de producción de direcciones para convertir las direcciones virtuales en direcciones físicas.
- 8.8 Los programas binarios están normalmente estructurados en muchos sistemas de la forma siguiente: el código se almacena a partir de una dirección virtual fija de valor pequeño, como por ejemplo 0; el segmento de código está seguido por el segmento de datos que se utiliza para almacenar las variables del programa. Cuando el programa comienza a ejecutarse, la fila se asigna en el otro extremo del espacio virtual de direcciones y se le permite crecer hacia abajo, hacia las direcciones virtuales más pequeñas. ¿Qué importancia tiene esta estructura en los siguientes esquemas?
- a. asignación contigua de memoria
 - b. segmentación pura
 - c. paginación pura
- 8.9 Considere un sistema de paginación en el que la tabla de páginas esté almacenada en memoria.
- a. Si una referencia a memoria tarda en realizarse 200 nanosegundos, ¿cuánto tiempo tardará una referencia a memoria paginada?
 - b. Si añadimos búferes TLB y el 75 por ciento de todas las referencias a las tablas de páginas se encuentran en los búferes TLB, ¿cuál es el tiempo efectivo que tarda una referencia a memoria? (Suponga que la localización a una entrada de la tabla de páginas contenida en los búferes TLB se hace en un tiempo cero, si la entrada ya se encuentra allí.)
- 8.10 ¿Por qué se combinan en ocasiones en un único esquema los mecanismos de segmentación y paginación?
- 8.11 Explique por qué resulta más fácil compartir un módulo reentrantre cuando se utiliza segmentación que cuando se utiliza un mecanismo de paginación pura.
- 8.12 Considere la siguiente tabla de segmento:

Segmento	Base	Longitud	Segmento	Base	Longitud
0	219	600	3	1327	580
1	2300	14	4	1952	96
2	90	100			

¿Cuáles son las direcciones físicas para las siguientes direcciones lógicas?

- a. 0.430
- b. 1.10
- c. 2.500
- d. 3.400
- e. 4.112

8.13 ¿Cuál es el propósito de paginar las tablas de páginas?

- 8.14 Considere el esquema jerárquico de paginación utilizado por la arquitectura VAX. ¿Cuáles operaciones de memoria se realizan cuando un programa de usuario ejecuta una operación de carga en memoria?
- 8.15 Compare el esquema de paginación segmentado con el esquema de tablas *hash* de páginas para la gestión de espacios de direcciones de gran tamaño. ¿Bajo qué circunstancias es preferible un esquema al otro?

8.16 Considere el esquema de traducción de direcciones de Intel que se muestra en la Figura 8.22-

- a. Describa todos los pasos realizados por el Intel Pentium para traducir una dirección lógica a su dirección física equivalente.
- b. ¿Qué ventajas supone para el sistema operativo disponer de un hardware que proporcione ese complicado mecanismo de traducción de memoria?
- c. ¿Tiene una desventaja este sistema de traducción de direcciones? En caso afirmativo, ¿cuáles son esas desventajas? En caso negativo, ¿por qué no todos los fabricantes utilizan este esquema?

Notas bibliográficas

El tema de la asignación dinámica del almacenamiento se analiza en Knuth [1973] (Sección 2.5), donde se muestra a través de resultados de simulación que el algoritmo de primer ajuste es, por regla general, superior al de mejor ajuste. Knuth [1973] explica la regla del 50 por ciento.

El concepto de paginación puede atribuirse a los diseñadores del sistema Atlas, que ha sido descrito por Kilburn et al. [1961] y por Howarth et al. [1961]. El concepto de segmentación fue introducido por primera vez por Dennis [1965]. Los mecanismos de segmentación paginada fueron soportados por vez primera en el GE 645, sobre el que se implementó originalmente MULTICS (Organick [1972] y Daley y Dennis [1967]).

Las tablas de páginas invertidas se explicaban en un artículo de Chang y Mergen [1988] acerca del gestor de almacenamiento del IBM RT.

El tema de la traducción de direcciones por software se trata en Jacob y Mudge [1997].

Hennessy y Patterson [2002] analiza los aspectos hardware de los búferes TLB, de las memorias caché y de las MMU. Talluri et al. [1995] analiza las tablas de páginas para espacios de direcciones de 64 bits. Una serie de técnicas alternativas para garantizar la protección de memoria se proponen y estudian en Wahbe et al. [1993a], Chase et al. [1994], Bershad et al. [1995a] y Thorn [1997]. Dougan et al. [1999] y Jacob y Mudge [2001] analizan diversas técnicas para gestionar los búferes TLB. Fang et al. [2001] incluye una evaluación del soporte necesario para páginas de gran tamaño.

Tanenbaum [2001] presenta el esquema de paginación del Intel 80386. Los sistemas de gestión de memoria para diversas arquitecturas (como el Pentium II, PowerPC y UltraSPARC) han sido descritos por Jacob y Mudge [1998a]. El esquema de segmentación en los esquemas Linux se presenta en Bovet y Cesati [2002].

Memoria virtual

En el Capítulo 8, hemos expuesto diversas estrategias de gestión de memoria utilizadas en los sistemas informáticos. Todas estas estrategias tienen el mismo objetivo: mantener numerosos procesos en memoria simultáneamente, con el fin de permitir la multiprogramación. Sin embargo, esas estrategias tienden a requerir que el proceso completo esté en memoria para poder ejecutarse.

La técnica de memoria virtual es un mecanismo que permite la ejecución de procesos que no se encuentren completamente en la memoria. Una de las principales ventajas de este esquema es que los programas pueden tener un tamaño mayor que la propia memoria física. Además, la memoria virtual abstrae la memoria principal, transformándola conceptualmente en una matriz uniforme y extremadamente grande de posiciones de almacenamiento, separando así la memoria lógica (tal como la ve el usuario) de la memoria física. Esta técnica libera a los programadores de las preocupaciones relativas a las limitaciones del espacio de almacenamiento de memoria. La memoria virtual también permite que los procesos compartan los archivos e implementen mecanismos de memoria compartida. Además, proporciona un mecanismo eficiente para la creación de procesos. Sin embargo, la memoria virtual no resulta fácil de implementar y puede reducir sustancialmente el rendimiento del sistema si se la utiliza sin el debido cuidado. En este capítulo, vamos a analizar los mecanismos de memoria virtual basados en paginación bajo demanda y examinaremos la complejidad y el coste asociados.

OBJETIVOS DEL CAPÍTULO

- Describir las ventajas de un sistema de memoria virtual.
- Explicar los conceptos de paginación bajo demanda, algoritmos de sustitución de páginas y asignación de marcos de páginas.
- Analizar los principios en que se basa el modelo de conjunto de trabajo.

9.1 Fundamentos

Los algoritmos de gestión de memoria esbozados en el Capítulo 8 resultan necesarios debido a un requerimiento básico: las instrucciones que se estén ejecutando deben estar en la memoria física. El primer enfoque para tratar de satisfacer este requisito consiste en colocar el espacio completo de direcciones lógicas dentro de la memoria física. Los mecanismos de carga dinámica pueden ayudar a aliviar esta restricción, pero requieren, por lo general, que el programador tome precauciones especiales y lleve a cabo un trabajo adicional.

El requisito de que las instrucciones deban encontrarse en la memoria física para poderlas ejecutar parece, a la vez, tanto necesario como razonable; pero también es un requisito poco deseable, ya que limita el tamaño de los programas, de manera que éstos no pueden exceder del tamaño

de la propia memoria física. De hecho, un examen de los programas reales nos muestra que en muchos casos, no es necesario tener el programa completo para poderlo ejecutar. Por ejemplo, considere lo siguiente:

- Los programas incluyen a menudo código para tratar las condiciones de error poco usuales. Puesto que estos errores raramente ocurren en la práctica (si es que ocurren alguna vez), este código no se ejecuta prácticamente nunca.
- A las matrices, a las listas y a las tablas se les suele asignar más memoria de la que realmente necesitan. Por ejemplo, puede declararse una matriz como compuesta de 100 por 100 elementos, aunque raramente vaya a tener un tamaño superior a 10 por 10 elementos. Una tabla de símbolos de ensamblador puede tener espacio para 3000 símbolos, aunque un programa típico suela tener menos de 200 símbolos.
- Puede que ciertas opciones y características de un programa se utilicen raramente. Por ejemplo, las rutinas que ejecutan funciones avanzadas de cálculo dentro de determinados programas de hoja de cálculo no suelen ser utilizadas por muchos usuarios.

Incluso en aquellos casos en que se necesite el programa completo, puede suceder que no todo el programa sea necesario al mismo tiempo.

La posibilidad de ejecutar un programa que sólo se encontrara parcialmente en la memoria proporcionaría muchas ventajas:

- Los programas ya no estarían restringidos por la cantidad de memoria física disponible. Los usuarios podrían escribir programas para un espacio de direcciones *virtual* extremadamente grande, simplificándose así la tarea de programación.
- Puesto que cada programa de usuario podría ocupar menos memoria física, se podrían ejecutar más programas al mismo tiempo, con el correspondiente incremento en la tasa de utilización del procesador y en la tasa de procesamiento, y sin incrementar el tiempo de respuesta ni el tiempo de ejecución.
- Se necesitarían menos operaciones de E/S para cargar o intercambiar cada programa de usuario con el fin de almacenarlo en memoria, de manera que cada programa de usuario se ejecutaría más rápido.

Por tanto, ejecutar programas que no se encuentren completamente en memoria proporciona ventajas tanto para el sistema como para el usuario.

La **memoria virtual** incluye la separación de la memoria lógica, tal como la percibe el usuario, con respecto a la memoria física. Esta separación permite proporcionar a los programadores una memoria virtual extremadamente grande, cuando sólo está disponible una memoria física de menor tamaño (Figura 9.1).

La memoria virtual facilita enormemente la tarea de programación, porque el programador ya no tiene por qué preocuparse de la cantidad de memoria física disponible; en lugar de ello, puede concentrarse en el problema que deba resolver mediante su programa.

El **espacio de direcciones virtuales** de un proceso hace referencia a la forma lógica (o virtual) de almacenar un proceso en la memoria. Típicamente, esta forma consiste en que el proceso comienza en una cierta dirección lógica (por ejemplo, la dirección 0) y está almacenado de forma contigua en la memoria, como se muestra en la Figura 9.2. Recuerde del Capítulo 8, sin embargo, que de hecho la memoria física puede estar organizada en marcos de página y que los marcos de página física asignados a un proceso pueden no ser contiguos. Es responsabilidad de la unidad de gestión de memoria (MMU, memory-management unit) establecer la correspondencia entre las páginas lógicas y los marcos de página física de la memoria.

Observe que, en la Figura 9.2, dejamos que el cúmulo crezca hacia arriba en la memoria, ya que se utiliza para la asignación dinámica de memoria. De forma similar, permitimos que la pila crezca hacia abajo en la memoria con las sucesivas llamadas a función. El gran espacio vacío (o hueco) entre el cúmulo y la pila forma parte del espacio de direcciones virtual, pero sólo consumirá páginas físicas reales cuando crezcan el cúmulo o la pila. Los espacios de direcciones virtuales que

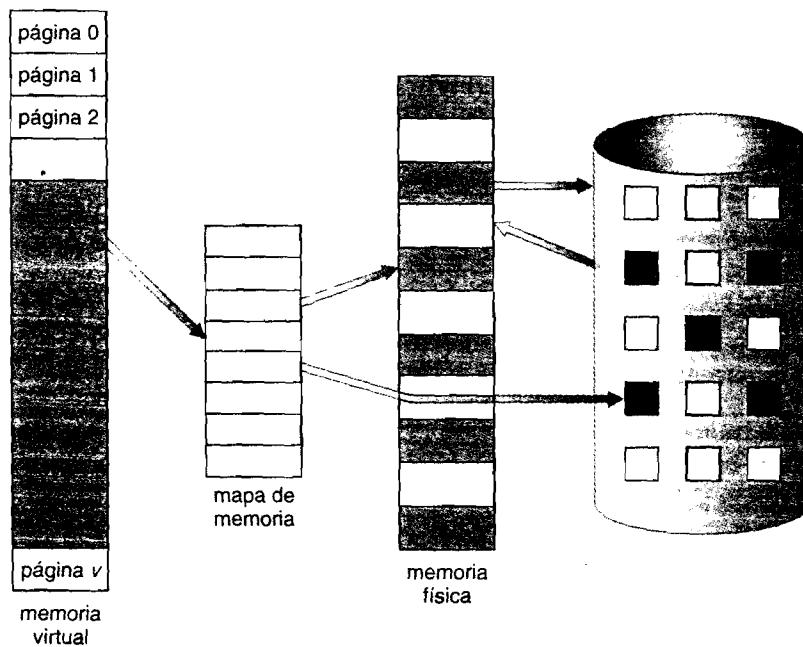


Figura 9.1 Diagrama que muestra una memoria virtual de tamaño mayor que la memoria física.

incluyen este tipo de huecos se denominan espacios de direcciones **dispersos**. Resulta ventajoso utilizar un espacio de direcciones disperso, porque los huecos pueden llenarse a medida que crecen los segmentos de pila o de cúmulo, o también si queremos montar dinámicamente una serie de bibliotecas (o posiblemente otros objetos compartidos) durante la ejecución del programa.

Además de separar la memoria lógica de la memoria física, la memoria virtual también permite que dos o más procesos comparten los archivos y la memoria mediante mecanismos de compartición de páginas (Sección 8.4.4). Esto proporciona las siguientes ventajas:

- Las bibliotecas del sistema pueden ser compartidas por numerosos procesos, mapeando el objeto compartido sobre un espacio de direcciones virtual. Aunque cada proceso considera

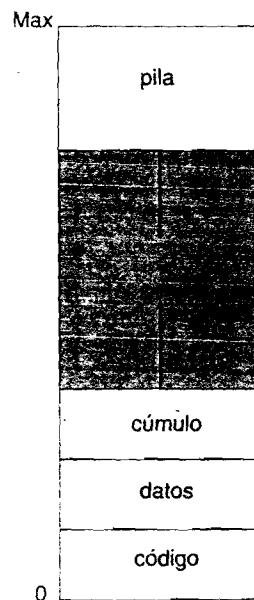


Figura 9.2 Espacio de direcciones virtual.

que las bibliotecas compartidas forman parte de su propio espacio de direcciones virtuales, las páginas reales de memoria física en las que residen las bibliotecas estarán comparten por todos los procesos (Figura 9.3). Normalmente, las bibliotecas se mapean en modo sólo lectura dentro del espacio de cada proceso con el cual se monten.

- De forma similar, la memoria virtual permite a los procesos compartir memoria. Recuerde del Capítulo 3 que dos o más procesos pueden comunicarse utilizando memoria compartida. La memoria virtual permite que un proceso cree una región de memoria que pueda compartir con otro proceso. Los procesos que comparten esta región la consideran parte de su espacio de direcciones virtual, aunque en realidad las páginas físicas reales de la memoria estarán compartidas entre los distintos procesos, de forma similar a como ilustra en la Figura 9.3.
- La memoria virtual puede permitir que se compartan páginas durante la creación de procesos mediante la llamada al sistema `fork()`, acelerando así la tarea de creación de procesos.

Exploraremos estas y otras ventajas de la memoria virtual con más detalle posteriormente en este capítulo; pero antes de ello, veamos cómo puede implementarse la memoria virtual mediante mecanismos de paginación bajo demanda.

9.2 Página bajo demanda

Considere cómo podría cargarse un programa ejecutable desde el disco a la memoria. Una opción consiste en cargar el programa completo en memoria física en el momento de ejecutar el programa. Sin embargo, esta técnica presenta el problema de que puede que no necesitemos inicialmente todo el programa en la memoria. Considere un programa que comience con una lista de acciones disponibles, de entre las cuales el usuario debe seleccionar una. Cargar el programa completo en memoria hace que se cargue el código ejecutable de *todas* las opciones, independientemente de si el usuario selecciona o no una determinada opción. Una estrategia alternativa consiste en cargar inicialmente las páginas únicamente cuando sean necesarias. Esta técnica se denomina **página bajo demanda** y se utiliza comúnmente en los sistemas de memoria virtual. Con la memoria virtual basada en paginación bajo demanda, sólo se cargan las páginas cuando así se solicita durante la ejecución del programa; de este modo, las páginas a las que nunca se acceda no llegarán a cargarse en la memoria física.

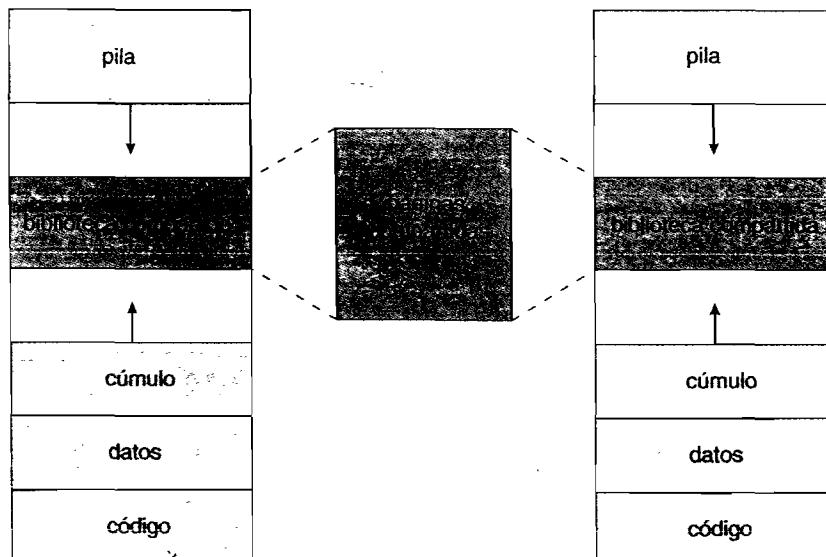


Figura 9.3 Biblioteca compartida mediante memoria virtual.

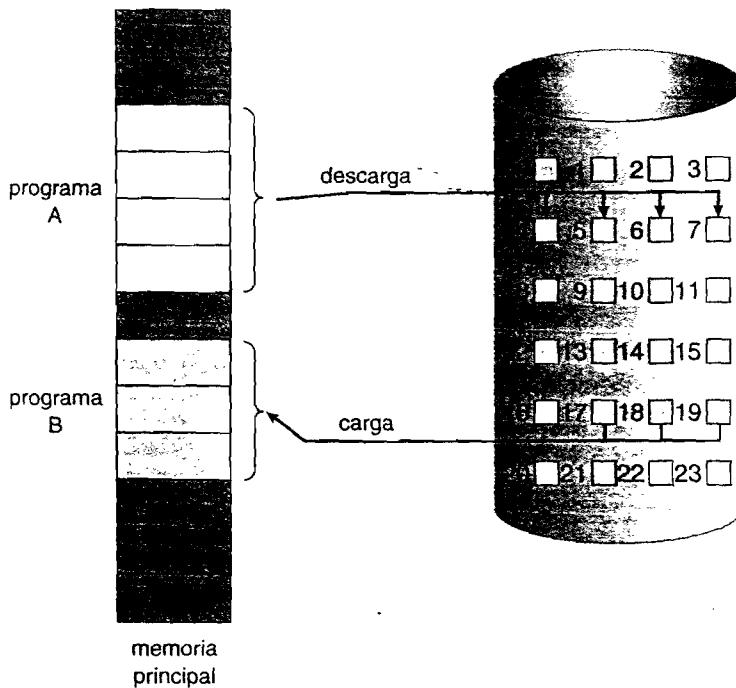


Figura 9.4 Transferencia de una memoria paginada a un espacio contiguo de disco.

Un sistema de paginación bajo demanda es similar a un sistema de paginación con intercambio (Figura 9.4) en el que los procesos residen en memoria secundaria (usualmente en disco). Cuando queremos ejecutar un proceso, realizamos un intercambio para cargarlo en memoria. Sin embargo, en lugar de intercambiar el proceso completo con la memoria, lo que hacemos en este caso es utilizar un **intercambiador perezoso**. El intercambiador perezoso jamás intercambia una página con la memoria a menos que esa página vaya a ser necesaria. Puesto que ahora estamos contemplando los procesos como secuencias de páginas, en lugar de como un único espacio de direcciones contiguas de gran tamaño, la utilización del término *intercambiador* es técnicamente incorrecta. Un intercambiador manipula procesos completos, mientras que un **paginador** sólo se ocupa de las páginas individuales de un proceso. Por tanto, utilizaremos el término *paginador* en lugar de *intercambiador*, cuando hablemos de la paginación bajo demanda.

9.2.1 Conceptos básicos

Cuando hay que cargar un proceso, el paginador realiza una estimación de qué páginas serán utilizadas antes de descargar de nuevo el proceso. En lugar de cargar el proceso completo, el paginador sólo carga en la memoria las páginas necesarias; de este modo, evita cargar en la memoria las páginas que no vayan a ser utilizadas, reduciendo así el tiempo de carga y la cantidad de memoria física necesaria.

Con este esquema, necesitamos algún tipo de soporte hardware para distinguir entre las páginas que se encuentran en memoria y las páginas que residen en el disco. Podemos usar para este propósito el esquema descrito en la Sección 8.5, basado en un bit válido-índice. Sin embargo, esta vez, cuando se configura este bit como “válido”, la página asociada será legal y además se encontrará en memoria. Si el bit se configura como “índice”, querrá decir que o bien la página no es válida (es decir, no se encuentra en el espacio lógico de direcciones del proceso) o es válida pero está actualmente en el disco. La entrada de la tabla de páginas correspondiente a una página que se cargue en memoria se configurará de la forma usual, pero la entrada de la tabla de páginas correspondiente a una página que no se encuentre actualmente en la memoria se marcará simplemente como inválida o contendrá la dirección de la página dentro del disco. Esta situación se ilustra en la Figura 9.5.

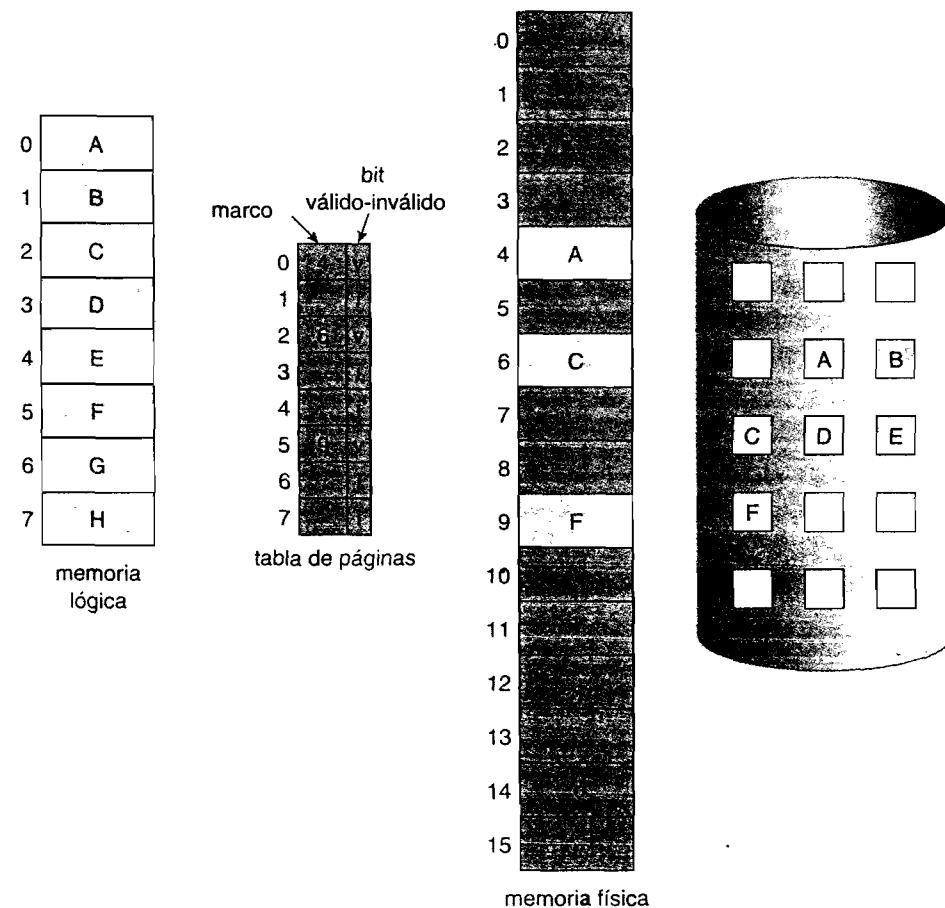


Figura 9.5 Tabla de páginas cuando algunas páginas no se encuentran en memoria principal.

Observe que marcar una página como inválida no tendrá ningún efecto si el proceso no intenta nunca acceder a dicha página. Por tanto, si nuestra estimación inicial es correcta y cargamos en la memoria todas las páginas que sean verdaderamente necesarias (y sólo esas páginas), el proceso se ejecutará exactamente igual que si hubiéramos cargado en memoria todas las páginas. Mientras que el proceso se ejecute y acceda a páginas que sean **residentes en memoria**, la ejecución se llevará a cabo normalmente.

Pero ¿qué sucede si el proceso trata de acceder a una página que no haya sido cargada en memoria? El acceso a una página marcada como inválida provoca una **interrupción de fallo de página**. El hardware de paginación, al traducir la dirección mediante la tabla de páginas, detectará que el bit de página inválida está activado, provocando una interrupción dirigida al sistema operativo. Esta interrupción se produce como resultado de que el sistema operativo no ha cargado anteriormente en memoria la página deseada. El procedimiento para tratar este fallo de página es muy sencillo (Figura 9.6):

1. Comprobamos una tabla interna (que usualmente se mantiene con el bloque del control del proceso) correspondiente a este proceso, para determinar si la referencia era un acceso de memoria válido o inválido.
2. Si la referencia era inválida, terminamos el proceso. Si era válida pero esa página todavía no ha sido cargada, la cargamos en memoria.
3. Buscamos un marco libre (por ejemplo, tomando uno de la lista de marcos libres).
4. Ordenamos una operación de disco para leer la página deseada en el marco recién asignado.

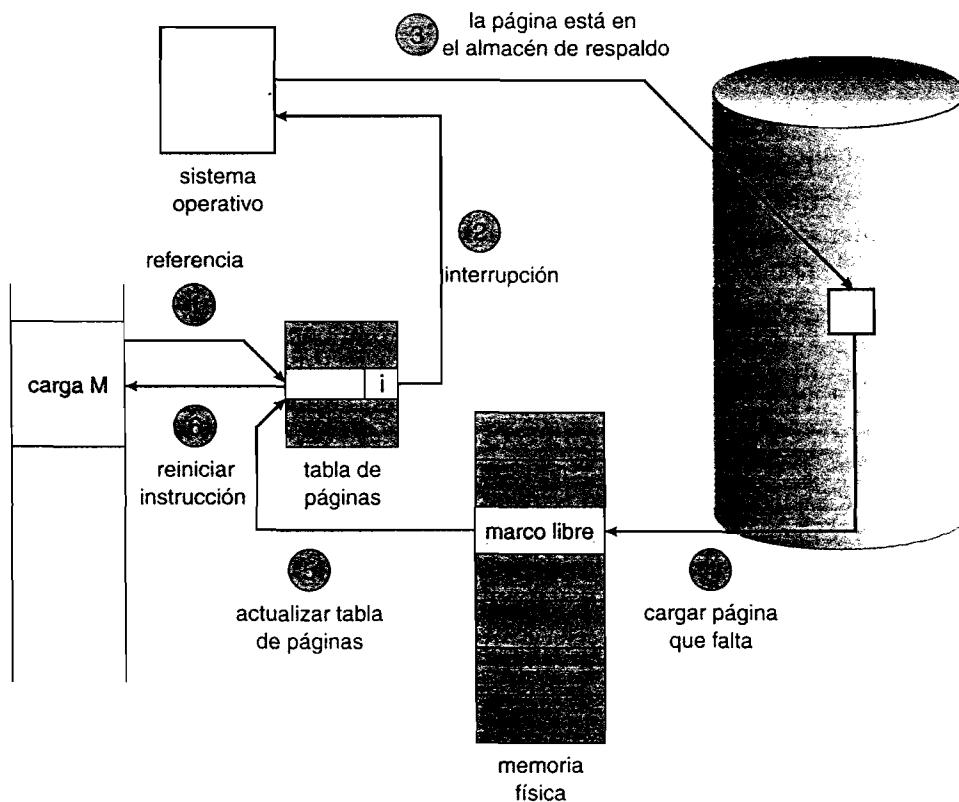


Figura 9.6 Pasos para tratar un fallo de página.

- Una vez completada la lectura de disco, modificamos la tabla interna que se mantiene con los datos del proceso y la tabla de páginas para indicar que dicha página se encuentra ahora en memoria.
- Reiniciamos la instrucción que fue interrumpida. El proceso podrá ahora acceder a la página como si siempre hubiera estado en memoria.

En el caso extremo, podríamos empezar a ejecutar un proceso *sin ninguna* página en memoria. Cuando el sistema operativo hiciera que el puntero de instrucciones apuntara a la primera instrucción del proceso, que se encontraría en una página no residente en memoria, se produciría inmediatamente un fallo de página. Después de cargar dicha página en memoria, el proceso continuaría ejecutándose, generando los fallos de página necesarios hasta que todas las páginas requeridas se encontraran en memoria. A partir de ahí, podría ejecutarse sin ningún fallo de página adicional. Este esquema sería una **paginación bajo demanda pura**: nunca cargar una página en memoria hasta que sea requerida.

En teoría, algunos programas podrían acceder a varias nuevas páginas de memoria con cada ejecución de una instrucción (una página para la instrucción y muchas para los datos), posiblemente generando múltiples fallos de página por cada instrucción. Esta situación provocaría una degradación inaceptable en el rendimiento del sistema. Afortunadamente, el análisis de la ejecución de los procesos muestra que este comportamiento es bastante improbable. Los programas tienden a tener lo que se denomina **localidad de referencia**, que se describe en la Sección 9.6.1, por lo que las prestaciones que pueden obtenerse con el mecanismo de paginación bajo demanda son razonables.

El hardware necesario para soportar la paginación bajo demanda es el mismo que para los mecanismos de paginación e intercambio:

- Una **tabla de páginas**. Esta tabla permite marcar una entrada como inválida mediante un bit válido-inválido o mediante un valor especial de una serie de bits de protección.

- **Memoria secundaria.** Esta memoria almacena aquellas páginas que no están presentes en memoria principal. La memoria secundaria es usualmente un disco de alta velocidad. Se conoce como dispositivo de intercambio y la sección del disco utilizada para este propósito se llama **espacio de intercambio**. La cuestión de la asignación del espacio de intercambio analiza en el Capítulo 12.

Un requisito fundamental para la paginación bajo demanda es la necesidad de poder reiniciar cualquier instrucción después de un fallo de página. Puesto que guardamos el estado (registro de código de condición, contador de instrucciones) del proceso interrumpido en el momento de producirse el fallo de página, debemos poder reiniciar el proceso en *exactamente* el mismo lugar y con el mismo estado, salvo porque ahora la página deseada se encontrará en memoria y será accesible. En la mayoría de los casos, este requisito resulta fácil de satisfacer. Los fallos de página pueden producirse para cualquier referencia a memoria. Si el fallo de página se produce al extraer una instrucción, podemos reiniciar las operaciones volviendo a extraer dicha instrucción. Si el fallo de página se produce mientras estamos leyendo un operando, deberemos extraer y decodificar de nuevo la instrucción y luego volver a leer el operando.

Como ejemplo más desfavorable, considere una instrucción con tres direcciones, como por ejemplo una que realizara la suma del contenido de A con B, colocando el resultado en C. Estos serían los pasos para ejecutar esta instrucción:

1. Extraer y decodificar la instrucción (ADD).
2. Extraer A.
3. Extraer B.
4. Sumar A y B.
5. Almacenar la suma en C.

Si el fallo de página se produce cuando estamos tratando de almacenar el resultado en C (porque C se encuentra en una página que no está actualmente en memoria), tendremos que obtener la página deseada, cargarla, corregir la tabla de páginas y reiniciar la instrucción. Ese reinicio requerirá volver a extraer la instrucción, decodificarla de nuevo, extraer otra vez los dos operandos y luego realizar de nuevo la suma. Sin embargo, el trabajo que hay que repetir no es excesivo (menos de una instrucción completa) y esa repetición sólo es necesaria cuando se produce un fallo de página.

La principal dificultad surge cuando una instrucción puede modificar varias ubicaciones diferentes. Por ejemplo, considere la instrucción MVC (mover carácter) de IBM System 360/370, que puede mover hasta 256 bytes de una ubicación a otra, pudiendo estar solapados los dos rangos de direcciones. Si alguno de los bloques (el de origen o el de destino) atraviesa una frontera de página, podría producirse un fallo de página después de que esa operación de desplazamiento se hubiera completado de manera parcial. Además, si los bloques de origen y de destino se solapan, el bloque de origen puede haber sido modificado, en cuyo caso no podemos simplemente reiniciar la instrucción.

Este problema puede resolverse de dos formas distintas. En una de las soluciones, el microcódigo calcula y trata de acceder a los dos extremos de ambos bloques. Si va a producirse un fallo de página, se producirá en este punto, antes de que se haya modificado nada. Entonces, podemos realizar el desplazamiento (después de cargar páginas según sea necesario) porque sabemos que ya no puede volver a producirse ningún fallo de página, ya que todas las páginas relevantes están en la memoria. La otra solución utiliza registros temporales para almacenar los valores de las ubicaciones sobreescritas. Si se produce un fallo de página, vuelven a escribirse en memoria los antiguos valores antes de que se produzca la interrupción. Esta acción restaura la memoria al estado que tenía antes de iniciar la instrucción, así que la instrucción puede repetirse.

Éste no es, en modo alguno, el único problema relativo a la arquitectura que surge como resultado de añadir el mecanismo de paginación a una arquitectura existente para permitir la paginación bajo demanda. Aunque este ejemplo sí que ilustra muy bien algunas de las dificultades

implicadas. El mecanismo de paginación se añade entre la CPU y la memoria en un sistema informático; debe ser completamente transparente para el proceso de usuario. Debido a ello, todos tendemos a pensar que se pueden añadir mecanismos de paginación a cualquier sistema. Pero, aunque esta suposición es cierta para los entornos de paginación que no son bajo demanda, en los que los fallos de página representan errores fatales, no es cierta cuando un fallo de página sólo significa que es necesario cargar en memoria una página adicional y luego reiniciar el proceso.

9.2.2 Rendimiento de la paginación bajo demanda

La paginación bajo demanda puede afectar significativamente al rendimiento de un sistema informático. Para ver por qué, vamos a calcular el **tiempo de acceso efectivo** a una memoria con paginación bajo demanda. Para la mayoría de los sistemas informáticos, el tiempo de acceso a memoria, que designaremos ma , va de 10 a 200 ns. Mientras no tengamos fallos de página, el tiempo de acceso efectivo será igual al tiempo de acceso a memoria. Sin embargo, si se produce un fallo de página, deberemos primero leer la página relevante desde el disco y luego acceder a la página deseada.

Sea p la probabilidad de que se produzca un fallo de página ($0 \leq p \leq 1$). Cabe esperar que p esté próxima a cero, es decir, que sólo vayan a producirse unos cuantos fallos de página. El **tiempo de acceso efectivo** será entonces

$$\text{tiempo de acceso efectivo} = (1 - p) \times ma + p \times \text{tiempo de fallo de página}.$$

Para calcular el tiempo de acceso efectivo, debemos conocer cuánto tiempo se requiere para dar servicio a un fallo de página. Cada fallo de página hace que se produzca la siguiente secuencia:

1. Interrupción al sistema operativo.
2. Guardar los registros de usuario y el estado del proceso.
3. Determinar que la interrupción se debe a un fallo de página.
4. Comprobar que la referencia de página era legal y determinar la ubicación de la página en el disco.
5. Ordenar una lectura desde el disco para cargar la página en un marco libre:
 - a. Esperar en la cola correspondiente a este dispositivo hasta que se dé servicio a la solicitud de lectura.
 - b. Esperar el tiempo de búsqueda y/o latencia del dispositivo.
 - c. Comenzar la transferencia de la página hacia un marco libre.
6. Mientras estamos esperando, asignar la CPU a algún otro usuario (planificación de la CPU, que será opcional).
7. Recibir una interrupción del subsistema de E/S de disco (E/S completada).
8. Guardar los registros y el estado del proceso para el otro usuario (si se ejecuta el paso 6).
9. Determinar que la interrupción corresponde al disco.
10. Corregir la tabla de páginas y otras tablas para mostrar que la página deseada se encuentra ahora en memoria.
11. Esperar a que se vuelva a asignar la CPU a este proceso.
12. Restaurar los registros de usuario, el estado del proceso y la nueva tabla de páginas y reanudar la ejecución de la instrucción interrumpida.

No todos estos pasos son necesarios en todos los casos. Por ejemplo, estamos suponiendo que, en el paso 6, la CPU se asigna a otro proceso mientras que tiene lugar la E/S. Esta operación permite que los mecanismos de multiprogramación mantengan una alta tasa de utilización de la CPU,

pero requiere un tiempo adicional para reanudar la rutina de servicio del fallo de página después de completarse la transferencia de E/S.

En cualquier caso, nos enfrentamos con tres componentes principales del tiempo de servicio de fallo de página:

1. Servir la interrupción de fallo de página.
2. Leer la página.
3. Reiniciar el proceso.

Las tareas primera y tercera pueden reducirse, codificando cuidadosamente el software, a unos cuantos cientos de instrucciones. Estas tareas pueden requerir entre 1 y 100 microsegundos cada una. Sin embargo, el tiempo de conmutación de página estará cerca, probablemente, de los 8 milisegundos. Un disco duro típico tiene una latencia media de 3 milisegundos, un tiempo de búsqueda de 5 milisegundos y un tiempo de transferencia de 0,05 milisegundos. Por tanto, el tiempo total de paginación es de unos 8 milisegundos, incluyendo los retardos hardware y software. Recuerde también que sólo estamos examinando el tiempo de servicio del dispositivo. Si hay una cola de procesos esperando a que el dispositivo les de servicio (otros procesos que hayan causado fallos de páginas), tendremos que añadir el tiempo de espera en cola para el dispositivo, ya que habrá que esperar a que el dispositivo de paginación esté libre para dar servicio a nuestra solicitud, incrementando todavía más el tiempo necesario para el intercambio.

Si tomamos un tiempo medio de servicio de fallo de página de 8 milisegundos y un tiempo de acceso a memoria de 200 nanosegundos, el tiempo efectivo de acceso en nanosegundos será:

$$\begin{aligned}\text{tiempo efectivo de acceso} &= (1 - p) \times (200) + p(8 \text{ milisegundos}) \\ &= (1 - p) \times 200 + p \times 8.000.000 \\ &= 200 + 7.999.800 \times p.\end{aligned}$$

Podemos ver, entonces, que el tiempo de acceso efectivo es directamente proporcional a la **tasa de fallos de página**. Si sólo un acceso de cada 1000 provoca un fallo de página, el tiempo efectivo de acceso será de 8,2 microsegundos. La computadora se ralentizará, por tanto, según el factor de 40 debido a la paginación bajo demanda. Si queremos que la degradación del rendimiento sea inferior al 10 por ciento, necesitaremos

$$\begin{aligned}220 &> 200 + 7.999.800 \times p, \\ 20 &> 7.999.800 \times p, \\ p &< 0,0000025.\end{aligned}$$

Es decir, para que la reducción de velocidad debida a la paginación sea razonable, sólo podemos permitir que provoque un fallo de página un acceso a memoria de cada 399.990. En resumen, resulta crucial mantener una tasa de fallos de página baja en los sistemas de paginación bajo demanda. Si no se hace así, el tiempo efectivo de acceso se incrementa, ralentizando enormemente la ejecución de los procesos.

Un aspecto adicional de la paginación bajo demanda es la gestión y el uso global del espacio de intercambio. Las operaciones de E/S de disco dirigidas al espacio de intercambio son generalmente más rápidas que las correspondientes al sistema de archivos, porque el espacio de intercambios se asigna en bloques mucho mayores y no se utilizan mecanismos de búsqueda de archivos ni métodos de asignación indirecta (Capítulo 12). El sistema puede, por tanto, conseguir una mayor tasa de transferencia para paginación copiando la imagen completa de un archivo en el espacio de intercambio en el momento de iniciar el proceso y luego realizando una paginación bajo demanda a partir del espacio de intercambio. Otra opción consiste en demandar las páginas inicialmente desde el sistema de archivos, pero ir las escribiendo en el espacio de intercambio a medida que se las sustituye. Esta técnica garantiza que sólo se lean desde el sistema de archivos las páginas necesarias y que todas las operaciones subsiguientes de paginación se lleven a cabo desde el espacio de intercambio.

Algunos sistemas tratan de limitar la cantidad de espacio de intercambio utilizado mediante mecanismos de paginación bajo demanda de archivos binarios. Las páginas demandadas de tales

archivos se cargan directamente desde el sistema de archivos; sin embargo, cuando hace falta sustituir páginas, estos marcos pueden simplemente sobreescribirse (porque nunca se han modificado) y las páginas pueden volver a leerse desde el sistema de archivos en caso necesario. Utilizando esta técnica, el propio sistema de archivos sirve como almacén de respaldo. Sin embargo, seguirá siendo necesario utilizar espacio de intercambio para las páginas que no estén asociadas con un archivo. Estas páginas incluyen la pila y el círculo de cada proceso. Este método parece ser un buen compromiso y es el que se utiliza en varios sistemas, incluyendo Solaris y BSD UNIX.

9.3 Copia durante la escritura

En la Sección 9.2 hemos indicado cómo podría un proceso comenzar rápidamente, cargando únicamente en memoria la página que contuviera la primera instrucción. Sin embargo, la creación de un proceso mediante la llamada al sistema `fork()` puede inicialmente evitar que se tenga que cargar ninguna página, utilizando una técnica similar a la de compartición de páginas que se ha descrito en la Sección 8.4.4. Esta técnica permite la creación rápida de procesos y minimiza el número de nuevas páginas que deben asignarse al proceso recién creado.

Recuerde que la llamada al sistema `fork()` crea un proceso hijo como duplicado de su padre. Tradicionalmente, `fork()` trabajaba creando una copia del espacio de direcciones del padre para el hijo, duplicando las páginas que pertenecen al padre. Sin embargo, considerando que muchos procesos hijos invocan la llamada al sistema `exec()` inmediatamente después de su creación, puede que sea innecesaria la copia del espacio de direcciones del padre. Como alternativa, podemos utilizar una técnica conocida con el nombre de **copia durante la escritura**, que funciona permitiendo que los procesos padre e hijo comparten inicialmente las mismas páginas. Estas páginas compartidas se marcan como páginas de copia durante la escritura, lo que significa que si cualquiera de los procesos escribe en una de las páginas compartidas, se creará una copia de esa página compartida. El proceso de copia durante la escritura se ilustra en las Figuras 9.7 y 9.8, que muestran el contenido de la memoria física antes y después de que el proceso 1 modifique la página C.

Por ejemplo, suponga que el proceso hijo trata de modificar una página que contiene parte de la pila, estando las páginas definidas como de copia durante la escritura. El sistema operativo creará entonces una copia de esta página, y la mapeará sobre el espacio de direcciones del proceso hijo. El proceso hijo modificará entonces su página copiada y no la página que pertenece al proceso padre. Obviamente, cuando se utiliza la técnica de copia durante la escritura, sólo se copian las páginas que sean modificadas por algunos de los procesos; todas las páginas no modificadas podrán ser compartidas por los procesos padre e hijo. Observe también que sólo es necesario marcar como de copia durante la escritura aquellas páginas que puedan ser modificadas. Las páginas que no puedan modificarse (páginas que contengan el código ejecutable) pueden compartirse entre el padre y el hijo. La técnica de copia durante la escritura resulta común en varios sistemas operativos, incluyendo Windows XP, Linux y Solaris.

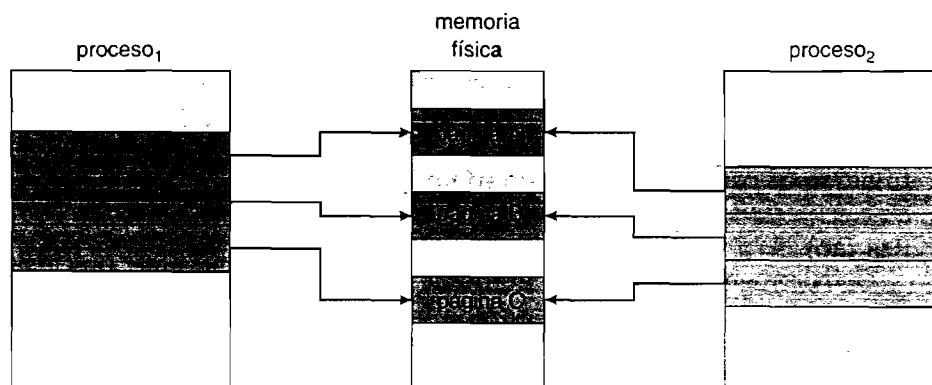


Figura 9.7 Antes de que el proceso 1 modifique la página C.

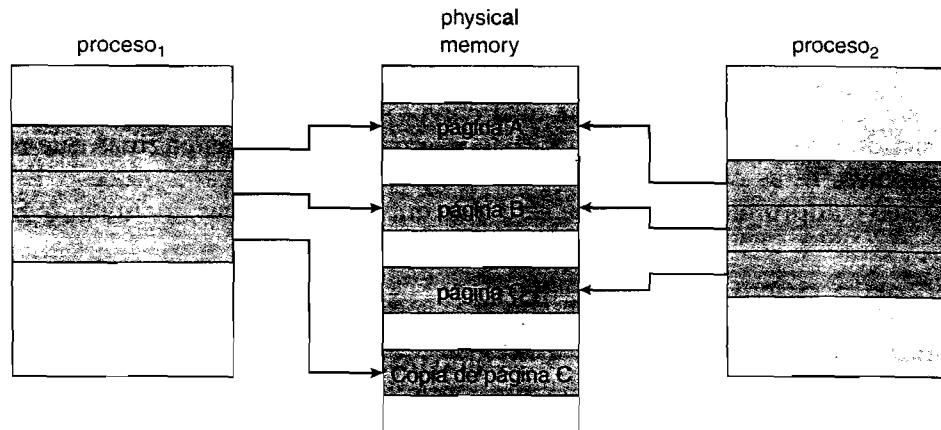


Figura 9.8 Despues de que el proceso 1 modifique la página C.

Cuando se determina que se va a duplicar una página utilizando el mecanismo de copia durante la escritura, es importante fijarse en la ubicación desde la que se asignará la página libre. Muchos sistemas operativos proporcionan un **conjunto compartido** de páginas libres para satisfacer tales solicitudes. Esas páginas libres se suelen asignar cuando la pila o el cùmulo de un proceso deben expandirse o cuando es necesario gestionar páginas de copia durante la escritura. Los sistemas operativos asignan normalmente esas páginas utilizando una tècnica denominada **relleno de ceros bajo demanda**. Las páginas de relleno de cero bajo demanda se llenan de ceros antes de asignarlas, eliminando así el contenido anterior.

Diversas versiones de UNIX (incluyendo Solaris y Linux) proporcionan tambièn una variante de la llamada al sistema `fork()`: se trata de la llamada `vfork()` (que quiere decir `fork para memoria virtual`). `vfork()` opera de forma diferente de `fork()` en las operaciones de copia durante la escritura. Con `vfork()`, el proceso padre se suspende y el proceso hijo utiliza el espacio de direcciones del padre. Puesto que `vfork()` no utiliza el mecanismo de copia durante la escritura, si el proceso hijo modifica cualquiera de las páginas del espacio de direcciones del padre, las páginas modificadas serán visibles para el padre una vez que éste reanude su ejecución. Por tanto, `vfork()` debe utilizarse con precaución, para garantizar que el proceso hijo no modifique el espacio de direcciones del padre. `vfork()` està pensado para usarse cuando el proceso hijo invoca `exec()` inmediatamente después de la creación. Puesto que no se produce ninguna copia de páginas, `vfork()` es un mètodo extremadamente eficiente para la creación de procesos y se utiliza en ocasiones para implementar interfaces *shell* de linea de comandos UNIX.

9.4 Sustitución de páginas

En nuestra explicación anterior acerca de la tasa de fallos de página, hemos supuesto que para cada página sólo se produce como mucho un fallo, la primera vez que se hace referencia a la misma. Sin embargo, esta suposición no es del todo precisa. Si un proceso de diez páginas sólo utiliza en realidad la mitad de ellas, el mecanismo de paginación bajo demanda nos ahorrará las operaciones de E/S necesarias para cargar las cinco páginas que nunca se utilizan. De este modo, podemos incrementar el grado de multiprogramación ejecutando el doble de procesos. Así, si tuviéramos cuarenta marcos, podríamos ejecutar ocho procesos en lugar de los cuatro que podrían ejecutarse si cada uno de los procesos requiriera diez marcos (cinco de los cuales jamás se utilizarían).

Si incrementamos nuestro grado de multiprogramación, estaremos **sobreasignando** la memoria. Si ejecutamos seis procesos, cada uno de los cuales tiene diez páginas de tamaño pero utiliza en realidad únicamente cinco páginas, tendremos una tasa de utilización de la CPU y una tasa de procesamiento más altas, quedándonos diez marcos libres. Es posible, sin embargo, que cada uno de estos procesos, para un determinado conjunto de datos, trate repentinamente de utilizar sus diez páginas, lo que implicaría que hacen falta 60 marcos cuando sólo hay cuarenta disponibles.

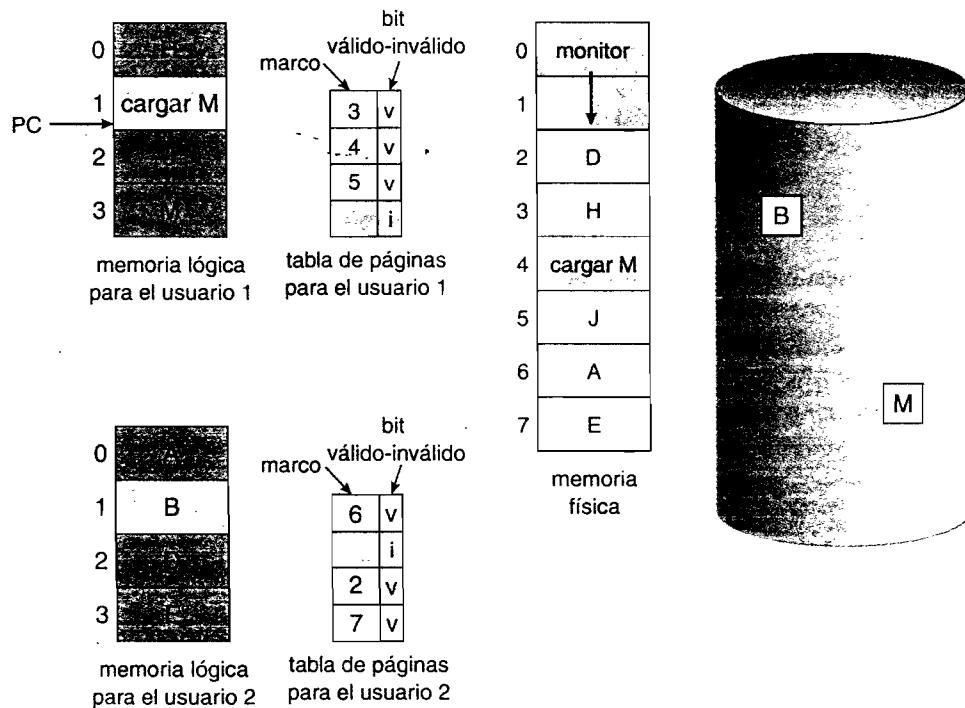


Figura 9.9 La necesidad de la sustitución de páginas.

Además, tenga en cuenta que la memoria del sistema no se utiliza sólo para almacenar páginas de programas. Los búferes de E/S también consumen una cantidad significativa de memoria. Este otro tipo de uso puede incrementar las demandas impuestas a los algoritmos de asignación de memoria. Decidir cuánta memoria asignar a la E/S y cuánta a las páginas de programa representa un considerable desafío. Algunos sistemas asignan un porcentaje fijo de memoria para los búferes de E/S, mientras que otros permiten que los procesos de usuario y el subsistema de E/S compitan por la memoria del sistema.

La sobreasignación de memoria se manifiesta de la forma siguiente. Imagine que, cuando se está ejecutando un proceso de usuario, se produce un fallo de página. El sistema operativo determina dónde reside la página deseada dentro del disco y entonces se encuentra con que no hay *ningún* marco libre en la lista de marcos libres; toda la memoria está siendo utilizada (Figura 9.9).

El sistema operativo dispone de varias posibilidades en este punto. Una de ellas sería terminar el proceso de usuario; sin embargo, la paginación bajo demanda es, precisamente, un intento del sistema operativo para mejorar la tasa de utilización y la tasa de procesamiento del sistema informático. Los usuarios no deben ser conscientes de que sus procesos se están ejecutando en un sistema paginado, es decir, la paginación debería ser lógicamente transparente para el usuario. Por tanto, esta opción no es la mejor de las posibles.

En lugar de ello, el sistema operativo puede descargar un proceso de memoria, liberando todos los marcos correspondientes y reduciendo el nivel de multiprogramación. Esta opción resulta adecuada en algunas circunstancias, y la analizaremos con más detalle en la Sección 9.6. Sin embargo, vamos a ver aquí la solución más comúnmente utilizada: la **sustitución de páginas**.

9.4.1 Sustitución básica de páginas

La sustitución de páginas usa la siguiente técnica. Si no hay ningún marco libre, localizamos uno que no esté siendo actualmente utilizado y lo liberamos. Podemos liberar un marco escribiendo su contenido en el espacio de intercambio y modificando la tabla de páginas (y todas las demás tablas) para indicar que esa página ya no se encuentra en memoria (Figura 9.10). Ahora podremos utilizar el marco liberado para almacenar la página que provocó el fallo de página en el proceso.

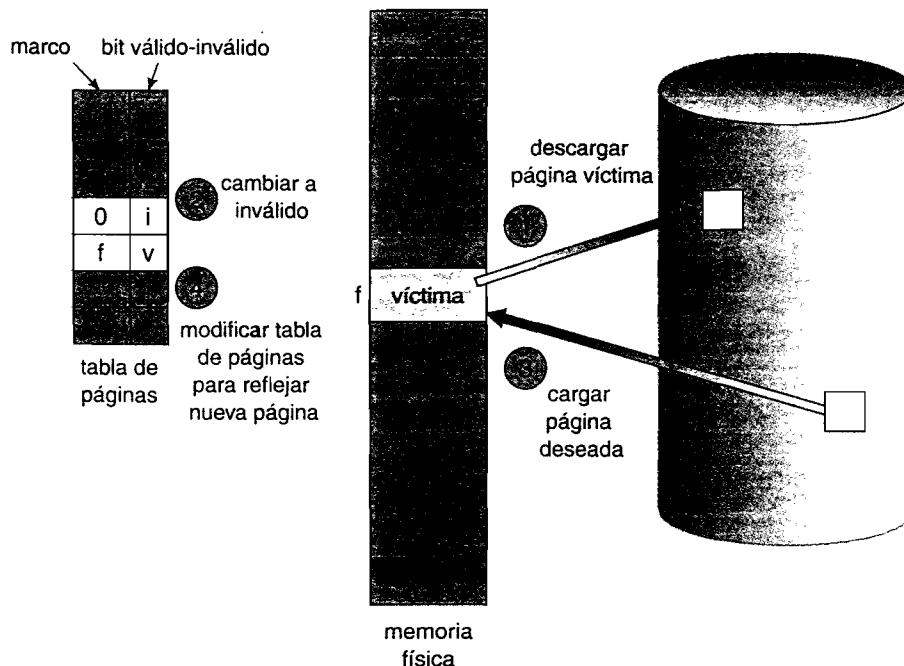


Figura 9.10 Sustitución de páginas.

Modifiquemos la rutina de servicio del fallo de página para incluir este mecanismo de sustitución de páginas:

1. Hallar la ubicación de la página deseada dentro del disco.
2. Localizar un marco libre:
 - a. Si hay un marco libre, utilizarlo.
 - b. Si no hay ningún marco libre, utilizar un algoritmo de sustitución de páginas para seleccionar un **marco víctima**
 - c. Escribir el marco víctima en el disco; cambiar las tablas de páginas y de marcos correspondientemente.
3. Leer la página deseada y cargarla en el marco recién liberado; cambiar las tablas de páginas y de marcos.
4. Reiniciar el proceso de usuario

Observe que, si no hay ningún marco libre, se necesitan *dos* transferencias de páginas (una descarga y una carga). Esta situación duplica, en la práctica, el tiempo de servicio del fallo de página e incrementa correspondientemente el tiempo efectivo de acceso.

Podemos reducir esta carga de trabajo adicional utilizando un **bit de modificación** (o **bit sucio**). Cuando se utiliza este esquema, cada página o marco tiene asociado en el hardware un bit de modificación. El bit de modificación para una página es activado por el hardware cada vez que se escribe en una palabra o byte de la página, para indicar que la página ha sido modificada. Cuando seleccionemos una página para sustitución, examinaremos su bit de modificación. Si el bit está activado, sabremos que la página ha sido modificada desde que se la leyó del disco. En este caso, deberemos escribir dicha página en el disco. Sin embargo, si el bit de modificación no está activado, la página *no* habrá sido modificada desde que fue leída y cargada en memoria. Por tanto, si la copia de la página en el disco no ha sido sobreescrita (por ejemplo, por alguna otra página), no necesitaremos escribir la página de memoria en el disco, puesto que ya se encuentra allí. Esta técnica también se aplica a las páginas de sólo lectura (por ejemplo, las páginas de código binario): dichas páginas no pueden ser modificadas, por lo que se las puede descartar cuando se desee.

Este esquema puede reducir significativamente el término requerido para dar servicio a un fallo de página, ya que reduce a la mitad el tiempo de E/S si la página no ha sido modificada.

El mecanismo de sustitución de páginas resulta fundamental para la paginación bajo demanda. Completa la separación entre la memoria lógica y la memoria física y, con este mecanismo, se puede proporcionar a los programadores una memoria virtual de gran tamaño a partir de una memoria física más pequeña. Sin el mecanismo de paginación bajo demanda, las direcciones de usuario se mapearían sobre direcciones físicas, por lo que los dos conjuntos de direcciones podrían ser distintos, pero todas las páginas de cada proceso deberían cargarse en la memoria física. Por el contrario, con la paginación bajo demanda, el tamaño del espacio lógico de direcciones ya no está restringido por la memoria física. Si tenemos un proceso de usuario de veinte páginas, podemos ejecutarlo en diez marcos de memoria simplemente utilizando la paginación bajo demanda y un algoritmo de sustitución para encontrar un marco libre cada vez que sea necesario. Si hay que sustituir una página que haya sido modificada, su contenido se copiará en el disco. Una referencia posterior a dicha página provocaría un fallo de página y, en ese momento, la página volvería a cargarse en memoria, quizás sustituyendo a alguna otra página en el proceso.

Debemos resolver dos problemas principales a la hora de implementar la paginación bajo demanda: hay que desarrollar un **algoritmo de asignación de marcos** y un **algoritmo de sustitución de páginas**. Si tenemos múltiples procesos en memoria, debemos decidir cuántos marcos asignar a cada proceso. Además, cuando se necesita una sustitución de páginas, debemos seleccionar los marcos que hay que sustituir. El diseño de los algoritmos apropiados para resolver estos problemas es una tarea de gran importancia, porque las operaciones del E/S de disco son muy costosas en términos de rendimiento. Cualquier pequeña mejora en los métodos de paginación bajo demanda proporciona un gran beneficio en términos de rendimiento del sistema.

Existen muchos algoritmos distintos de sustitución de páginas; probablemente, cada sistema operativo tiene su propio esquema de sustitución. ¿Cómo podemos seleccionar un algoritmo de sustitución concreto? En general, intentaremos seleccionar aquel que tenga la tasa de fallos de página más baja.

Podemos evaluar un algoritmo ejecutándolo con una cadena concreta de referencias de memoria y calculando el número de fallos de página. La cadena de referencias de memoria se denomina **cadena de referencia**. Podemos generar cadenas de referencia artificialmente (por ejemplo, utilizando un generador de números aleatorios) o podemos obtener una traza de un sistema determinado y registrar la dirección de cada referencia de memoria. Esta última opción produce un gran número de datos (del orden de un millón de direcciones por segundo). Para reducir el número de datos, podemos emplear dos hechos.

En primer lugar, para un cierto tamaño de página (y el tamaño de página generalmente está fijo por el hardware o por el sistema), tan sólo necesitamos considerar el número de página, más que la dirección completa. En segundo lugar, si tenemos una referencia a la página p , todas las referencias *inmediatamente* siguientes que se hagan a la página p nunca provocarán un fallo de página. La página p estará en la memoria después de la primera referencia, por lo que todas las referencias que la sigan de forma inmediata no provocarán ningún fallo de página.

Por ejemplo, si trazamos un proceso concreto, podríamos registrar la siguiente secuencia de direcciones:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

Suponiendo 100 bytes por página, esta secuencia se reduciría a la siguiente cadena de referencia:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Para determinar el número de fallos de página para una cadena de referencia concreta y un algoritmo de sustitución de páginas determinado, también necesitamos conocer el número de marcos de página disponibles. Obviamente, a medida que se incrementa el número de marcos disponibles, se reduce el número de fallos de página. Para la cadena de referencia considerada ante-

riormente, por ejemplo, si tuviéramos tres o más marcos, sólo se producirían tres fallos de página: uno para la primera referencia a cada página.

Por contraste, si sólo hubiera un marco disponible, tendríamos una sustitución para cada referencia, lo que daría como resultado once fallos de página. En general, podemos esperar obtener una curva como la que se muestra en la Figura 9.11. A medida que se incrementa el número de marcos, el número de fallos de página cae hasta un cierto nivel mínimo. Por supuesto, si se añade memoria física se incrementará el número de marcos.

A continuación vamos a ilustrar diversos algoritmos de sustitución de páginas. Para ello, usaremos la cadena de referencia

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

para una memoria con tres marcos.

9.4.2 Sustitución de páginas FIFO

El algoritmo más simple de sustitución de páginas es un algoritmo de tipo FIFO (first-in, first-out). El algoritmo de sustitución FIFO asocia con cada página el instante en que dicha página fue cargada en memoria. Cuando hace falta sustituir una página, se elige la página más antigua. Observe que no es estrictamente necesario anotar el instante en que se cargó cada página, ya que podríamos simplemente crear una cola FIFO para almacenar todas las páginas en memoria y sustituir la página situada al principio de la cola. Cuando se carga en memoria una página nueva, se la inserta al final de la cola.

Para nuestra cadena de referencia de ejemplo, los tres marcos estarán inicialmente vacíos. Las primeras tres referencias (7, 0, 1) provocan fallos de página y esas páginas se cargan en esos marcos vacíos. La siguiente referencia (2) sustituirá a la página 7, porque la página 7 fue la primera en cargarse. Puesto que 0 es la siguiente referencia y 0 ya está en memoria, no se producirá ningún fallo de página para esta referencia. La primera referencia a 3 da como resultado la sustitución de la página 0, ya que ahora es la primera de la cola. Debido a esta sustitución, la siguiente referencia a 0, provocará un fallo de página. Entonces, la página 1 será sustituida por la página 0. Este proceso continua como se muestra en la Figura 9.12. Cada vez que se produce un fallo de página, mostramos las páginas que se encuentran en los tres marcos disponibles. En total, hay 15 fallos de página.

El algoritmo de sustitución de páginas FIFO es fácil de entender y de programar. Sin embargo, su rendimiento no siempre es bueno. Por un lado, la página sustituida puede ser un modulo

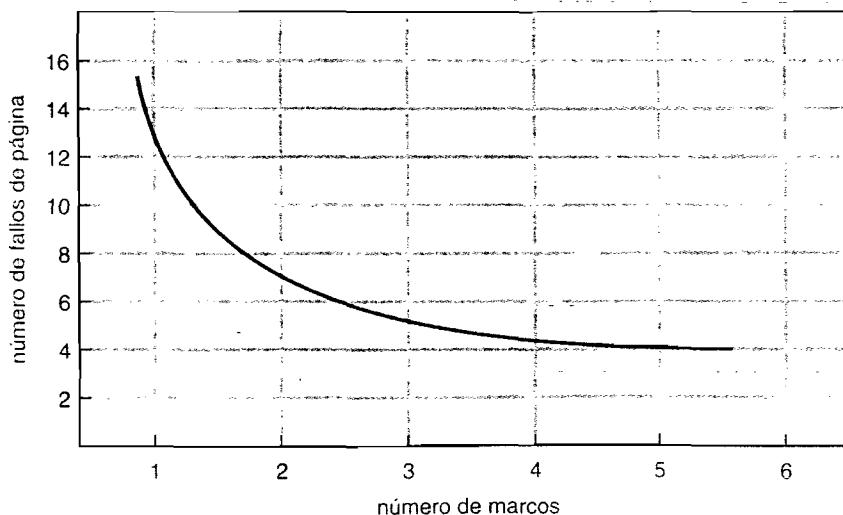


Figura 9.11 Gráfica de fallos de página en función del número de marcos.

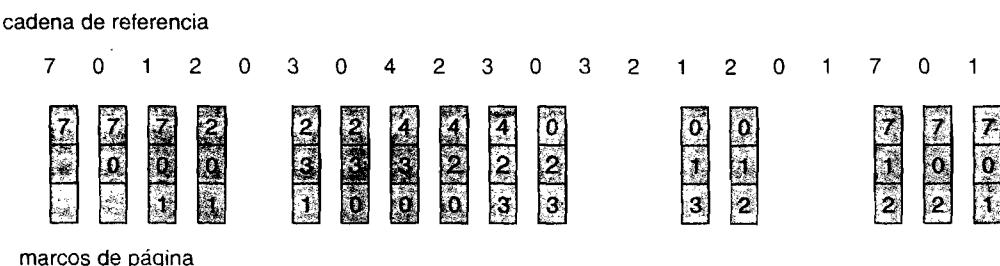


Figura 9.12 Algoritmos de sustitución de páginas FIFO.

de inicialización que hubiera sido empleado hace mucho tiempo y que ya no es necesario, pertambién podría contener una variable muy utilizada que fuera inicializada muy pronto y que se utilice de manera constante.

Observe que, aunque seleccionemos para sustitución una página que se esté utilizando activamente, todo sigue funcionando de forma correcta. Después de sustituir una página activa por otra nueva, se producirá casi inmediatamente un nuevo fallo de página que provocará la recarga de la página activa. Entonces, será necesario sustituir alguna otra página con el fin de volver a cargar en la memoria la página activa. De este modo, una mala elección de la página que hay que sustituir incrementa la tasa de fallos de página y ralentiza la ejecución de los procesos. Aunque, como vemos, eso no implica que la ejecución sea incorrecta.

Para ilustrar los problemas que pueden producirse con los algoritmos de sustitución de páginas FIFO, vamos a considerar la siguiente cadena de referencia:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

La Figura 9.13 muestra la curva de fallos de página para esta cadena de referencia, en función del número de marcos disponibles. Observe que el número de fallos para cuatro marcos (diez) es *superior* que el número de fallos para tres marcos (nueve). Este resultado completamente inesperado se conoce con el nombre de **anomalía de Belady**: para algunos algoritmos de sustitución de páginas, la tasa de fallos de página puede *incrementarse* a medida que se incrementa el número de marcos asignados. En principio, cabría esperar que al asignar más memoria a un proceso se mejorara su rendimiento, pero en algunos trabajos pioneros de investigación, los investigadores observaron que esta suposición no siempre es correcta. La anomalía de Belady se descubrió como consecuencia de estas investigaciones.

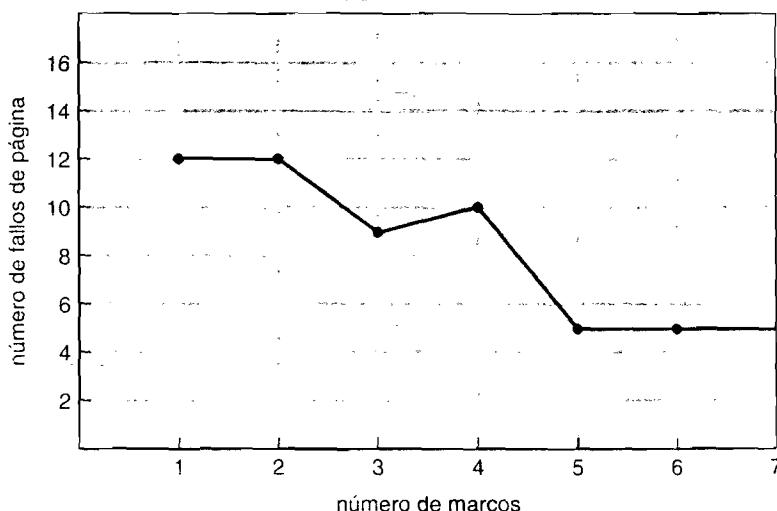


Figura 9.13 Curva de fallos de página para una sustitución FIFO con una determinada cadena de referencia.

9.4.3 Sustitución óptima de páginas

Uno de los resultados del descubrimiento de la anomalía de Belady fue la búsqueda de un ritmo óptimo de sustitución de páginas. Un algoritmo óptimo de sustitución de páginas es aquel que tenga la tasa más baja de fallos de página de entre todos los algoritmos y que nunca esté sujeto a la anomalía de Belady. Resulta que dicho algoritmo existe, habiéndoselo denominado o MIN. Consiste simplemente en lo siguiente:

Sustituir la página que no vaya a ser utilizada durante el período de tiempo más largo.

La utilización de este algoritmo de sustitución de página garantiza la tasa de fallos de página más baja posible para un número fijo de marcos.

Por ejemplo, para nuestra cadena de referencia de ejemplo, el algoritmo óptimo de sustitución de páginas nos daría nueve fallos de página, como se muestra en la Figura 9.14. Las primeras referencias provocan sendos fallos de página que llenarán los tres marcos libres. La referencia a la página 2 sustituirá a la página 7, porque 7 no va a utilizarse hasta la referencia 18, mientras que la página 0 será utilizada en 5 y la página 1 en 14. La referencia a la página 3 sustituye a la página 1, ya que la página 1 será la última de las tres páginas de memoria a la que se volverá a hacer referencia. Con sólo nueve fallos de página, el algoritmo óptimo de sustitución es mucho mejor que un algoritmo FIFO, que dará como resultado quince fallos de página (si ignoramos los primeros tres fallos de página, a los que todos los algoritmos están sujetos, entonces el algoritmo óptimo de sustitución es el doble de bueno que un algoritmo de sustitución FIFO). De hecho, ningún algoritmo de sustitución puede procesar esta cadena de referencia para tres marcos con menos de nueve fallos de página.

Desafortunadamente, el algoritmo óptimo de sustitución de páginas resulta difícil de implementar, porque requiere un conocimiento futuro de la cadena de referencia (nos hemos encontrado con una situación similar al hablar del algoritmo SJF de planificación de la CPU en la Sección 5.3.2). Como resultado, el algoritmo óptimo se utiliza principalmente con propósitos comparativos. Por ejemplo, puede resultar útil saber que, aunque un nuevo algoritmo no sea óptimo, tiene una desviación de menos del 12,3 por ciento con respecto al óptimo en caso peor y el 4,7 por ciento como promedio.

9.4.4 Sustitución de páginas LRU

Si el algoritmo óptimo no resulta factible, quizás sea posible una aproximación a ese algoritmo óptimo. La distinción clave entre los algoritmos FIFO y OPT (dejando a parte el hecho de que uno mira hacia atrás en el tiempo, mientras que el otro mira hacia adelante) es que el algoritmo FIFO utiliza el instante de tiempo en que se cargó una página en memoria, mientras que el algoritmo OPT utiliza el instante en el que hay que utilizar una página. Si utilizamos el pasado reciente como aproximación del futuro próximo, podemos entonces sustituir la página que no haya sido utilizada durante el período más largo de tiempo (Figura 9.15). Esta técnica se conoce como algoritmo LRU (least-recently-used, menos recientemente utilizada).

El algoritmo de sustitución LRU asocia con cada página el instante correspondiente al último uso de dicha página. Cuando hay que sustituir una página, el algoritmo LRU selecciona la página que no haya sido utilizada durante un período de tiempo más largo. Debemos considerar esta estrategia como el algoritmo óptimo de sustitución de páginas si miramos hacia atrás en el tiempo.

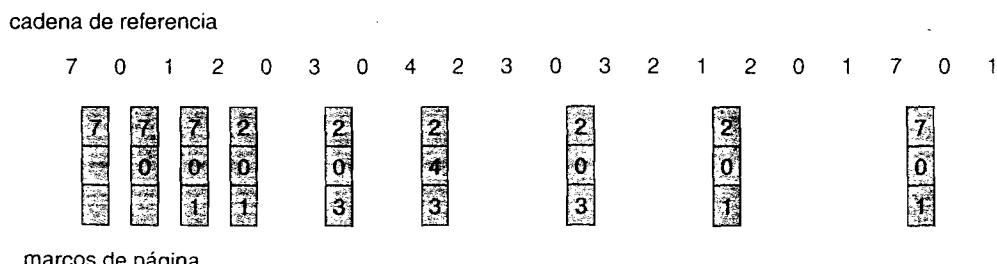


Figura 9.14 Algoritmo óptimo de sustitución de páginas.

po, en lugar de mirar hacia adelante (extrañamente, si denominamos S^R a la inversa de una cadena de referencia S , la tasa de fallos de página para el algoritmo OPT aplicado a S es igual que la tasa de fallos de página para el algoritmo OPT aplicado a S^R . De forma similar, la tasa de fallos de página para el algoritmo LRU aplicado a S es igual que la tasa de fallos de página del algoritmo LRU aplicado a S^R).

El resultado de aplicar el algoritmo de sustitución LRU a nuestra cadena de referencia de ejemplo se muestra en la Figura 9.15. El algoritmo LRU produce 12 fallos de página. Observe que los primeros cinco fallos coinciden con los del algoritmo óptimo de sustitución. Sin embargo, cuando se produce la referencia a la página 4, el algoritmo de sustitución LRU comprueba que, de los tres marcos que hay en la memoria, la página 2 es la que ha sido utilizada menos recientemente. Por tanto, el algoritmo LRU sustituye la página 2, al no ser consciente de que la página 2 va a ser utilizada en breve. Cuando se produce a continuación el fallo de página correspondiente a la página 2, el algoritmo LRU sustituye la página 3, ya que ahora será la menos recientemente utilizada de las tres páginas que hay en la memoria. A pesar de estos problemas, una sustitución LRU con 12 fallos de página es mucho mejor que una sustitución FIFO con 15 fallos de página.

Esta política LRU se utiliza a menudo como algoritmo de sustitución de páginas y se considera que es bastante buena. El principal problema es *cómo* implementar ese mecanismo de sustitución LRU. Un algoritmo LRU de sustitución de páginas puede requerir una considerable asistencia hardware. El problema consiste en determinar un orden para los marcos definido por el instante correspondiente al último uso. Existen dos posibles implementaciones:

- **Contadores.** En el caso más simple, asociamos con cada entrada en la tabla de páginas un campo de tiempo de uso y añadimos a la CPU un reloj lógico o contador. El reloj se incrementa con cada referencia a memoria. Cuando se realiza una referencia a una página, se copia el contenido del registro de reloj en el campo de tiempo de uso de la entrada de la tabla de páginas correspondiente a dicha página. De esta forma, siempre tenemos el “tiempo” de la última referencia a cada página y podremos sustituir la página que tenga el valor temporal menor. Este esquema requiere realizar una búsqueda en la tabla de páginas para localizar la página menos recientemente utilizada y realizar una escritura en memoria (para continuar el campo de tiempo de uso en la tabla de páginas) para cada acceso a memoria. Los tiempos deben también mantenerse apropiadamente cuando se modifiquen las tablas de páginas (debido a las actividades de planificación de la CPU). Asimismo, es necesario tener en cuenta el desbordamiento del reloj.
- **Pila.** Otra técnica para implementar el algoritmo de sustitución LRU consiste en mantener una pila de números de página. Cada vez que se hace referencia a una página, se extrae esa página de la pila y se la coloca en la parte superior. De esta forma, la página más recientemente utilizada se encontrará siempre en la parte superior de la pila y la menos recientemente utilizada en la inferior (Figura 9.16). Puesto que es necesario eliminar entradas de la parte intermedia de la pila, lo mejor para implementar este mecanismo es utilizar una lista doblemente enlazada con un puntero a la cabecera y otro a la cola. Entonces, eliminar una página y colocarla en la parte superior de la pila requiere modificar seis punteros en el caso peor. Cada actualización es algo más cara que con el otro método, pero no hay necesidad de implementar un reloj.

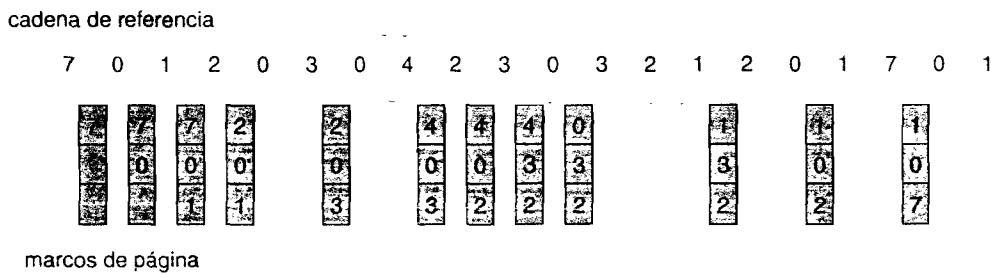


Figura 9.15 Algoritmo LRU de sustitución de páginas.

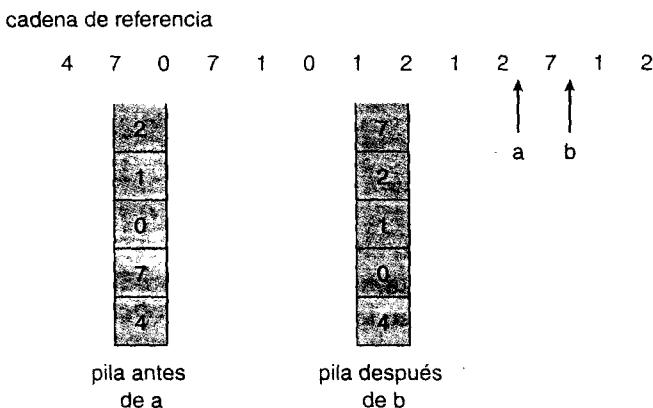


Figura 9.16 Utilización de una pila para registrar las referencias de página más recientes.

dad de buscar la página que hay que sustituir: el puntero de cola siempre apuntará a la parte inferior de la pila, que será la página menos recientemente utilizada. Esta técnica resulta particularmente apropiada para las implementaciones de algoritmos de sustitución LRU realizadas mediante software o microcódigo.

Al igual que el algoritmo óptimo de sustitución, el algoritmo de sustitución LRU no sufre la anomalía de Belady. Ambos algoritmos pertenecen a una clase de algoritmos de sustitución de páginas, denominada **algoritmos de pila**, que jamás pueden exhibir la anomalía de Belady. Un algoritmo de pila es un algoritmo para el que puede demostrarse que el conjunto de páginas en memoria para n marcos es siempre un *subconjunto* del conjunto de páginas que habría en memoria con $n + 1$ marcos. Para el algoritmo de sustitución LRU, el conjunto de páginas en memoria estaría formado por las n páginas más recientemente referenciadas. Si se incrementa el número de marcos, estas n páginas seguirán siendo las más recientemente referenciadas y, por tanto, continuarán estando en la memoria.

Observe que ninguna de las dos implementaciones del algoritmo LRU sería concebible sin disponer de una asistencia hardware más compleja que la que proporcionan los registros TLB estándar. La actualización de los campos de reloj o de la pila debe realizarse para *todas* las referencias de memoria. Si utilizáramos una interrupción para cada referencia con el fin de permitir la actualización por software de dichas estructuras de datos, todas las referencias a memoria se ralentizarían en un factor de al menos diez, ralentizando también según un factor de diez todos los procesos de usuario. Pocos sistemas podrían tolerar este nivel de carga de trabajo adicional debida a la gestión de memoria.

9.4.5 Sustitución de páginas mediante aproximación LRU

Pocos sistemas informáticos proporcionan el suficiente soporte hardware como para implementar un verdadero algoritmo LRU de sustitución de páginas. Algunos sistemas no proporcionan soporte hardware en absoluto, por lo que deben utilizarse otros algoritmos de sustitución de páginas (como por ejemplo el algoritmo FIFO). Sin embargo, dichos sistemas proporcionan algo de ayuda, en la forma de un **bit de referencia**. El bit de referencia para una página es activado por el hardware cada vez que se hace referencia a esa página (cada vez que se lee o escribe cualquier byte de dicha página). Los bits de referencia están asociados con cada entrada de la tabla de páginas.

Inicialmente, todos los bits son desactivados (con el valor 0) por el sistema operativo. A medida que se ejecuta un proceso de usuario, el bit asociado con cada una de las páginas referenciadas es activado (puesto a 1) por el hardware. Después de un cierto tiempo, podemos determinar qué páginas se han utilizado y cuáles no examinando los bits de referencia, aunque no podremos saber el *orden* de utilización de las páginas. Esta información constituye la base para muchos algoritmos de sustitución de páginas que traten de aproximarse al algoritmo de sustitución LRU.

9.4.5.1 Algoritmo de los bits de referencia adicionales

Podemos disponer de información de ordenación adicional registrando los bits de referencia a intervalos regulares. Podemos mantener un byte de 8 bits para cada página en una tabla de memoria. A intervalos regulares (por ejemplo, cada 100 milisegundos), una interrupción de temporización transfiere el control al sistema operativo. El sistema operativo desplaza el bit de referencia de cada página, transfiriéndolo al bit de mayor peso de ese byte de 8 bits, desplazando asimismo los otros bits una posición hacia la derecha descartando el bit de menor peso. Estos registros de desplazamiento de 8 bits contienen el historial de uso de las páginas en los 8 últimos períodos temporales. Por ejemplo, si el registro de desplazamiento contiene 00000000, entonces la página no habrá sido utilizada durante ocho períodos temporales; una página que haya sido utilizada al menos una vez en cada período tendrá un valor de su registro de desplazamiento igual a 11111111. Una página con un valor del registro de historial igual a 11000100 habrá sido utilizada más recientemente que otra con un valor de 01110111. Si interpretamos estos bytes de 8 bits como enteros sin signo, la página con el número más bajo será la menos recientemente utilizada y podremos sustituirla. Observe, sin embargo, que no se garantiza la unicidad de esos números. Por ello, podemos sustituir (descargar) todas las páginas que tengan el valor más pequeño o utilizar el método FIFO para elegir una de esas páginas.

El número de bits de historial puede, por supuesto, variarse y se selecciona (dependiendo del hardware disponible) para hacer que la actualización sea lo más rápida posible. En el caso extremo, ese número puede reducirse a cero, dejando sólo el propio bit de referencia. Este algoritmo se denomina **algoritmo de segunda oportunidad para la sustitución de páginas**.

9.4.5.2 Algoritmo de segunda oportunidad

El algoritmo básico de segunda oportunidad para la sustitución de páginas es un algoritmo de sustitución FIFO. Sin embargo, cuando se selecciona una página, inspeccionamos su bit de referencia. Si el valor es 0, sustituimos dicha página, pero si el bit de referencia tiene el valor 1, damos a esa página una segunda oportunidad y seleccionamos la siguiente página de la FIFO. Cuando una página obtiene una segunda oportunidad, su bit de referencia se borra y el campo que indica su instante de llegada se reconfigura, asignándole el instante actual. Así, una página a la que se le haya dado una segunda oportunidad no será sustituida hasta que todas las demás páginas hayan sido sustituidas (o se les haya dado también una segunda oportunidad). Además, si una página se utiliza de forma lo suficientemente frecuente como para que su bit de referencia permanezca activado, nunca será sustituida.

Una forma de implementar el algoritmo de segunda oportunidad (algunas veces denominado algoritmo del *reloj*) es una cola circular. Con este método, se utiliza un puntero (es decir, una manecilla del reloj) para indicar cuál es la siguiente página que hay que sustituir. Cuando hace falta un marco, el puntero avanza hasta que encuentra una página con un bit de referencia igual a 0. Al ir avanzando, va borrando los bits de referencia (Figura 9.17). Una vez que se encuentra una página víctima, se sustituye la página y la nueva página se inserta en la cola circular en dicha posición. Observe que, en el peor de los casos, cuando todos los bits estén activados, el puntero recorrerá la cola completa, dando a cada una de las páginas una segunda oportunidad y borrando todos los bits de referencia antes de seleccionar la siguiente página que hay que sustituir. El algoritmo de sustitución de segunda oportunidad degenera convirtiéndose en un algoritmo de sustitución FIFO si todos los bits están activados.

9.4.5.3 Algoritmo mejorado de segunda oportunidad

Podemos mejorar el algoritmo de segunda oportunidad considerando el bit de referencia y el bit de modificación (descrito en la Sección 9.4.1) como una pareja ordenada. Con estos dos bits, tenemos los cuatro casos posibles siguientes:

1. (0, 0) no se ha utilizado ni modificado recientemente: es la mejor página para sustitución.

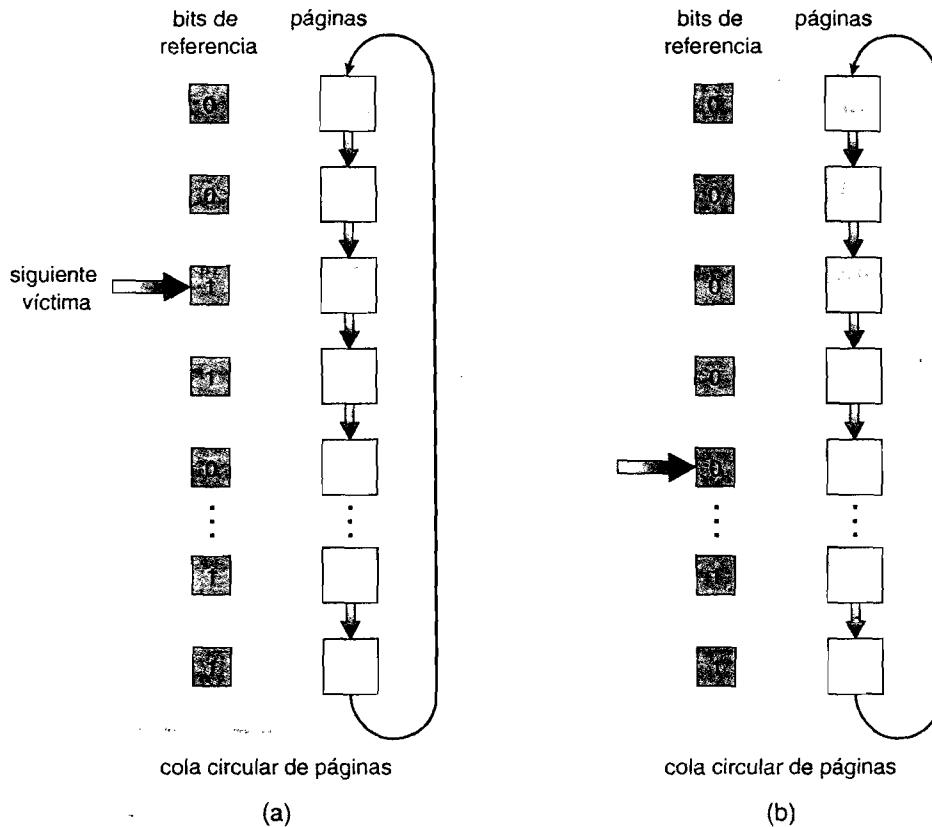


Figura 9.17 Algoritmo de sustitución de páginas de segunda oportunidad (del reloj).

2. (0, 1) no se ha usado recientemente pero sí que se ha modificado: no es tan buena como la anterior, porque será necesario escribir la página antes de sustituirla.
3. (1, 0) se ha utilizado recientemente pero está limpia: probablemente se la vuelva a usar pronto.
4. (1, 1) se la ha utilizado y modificado recientemente: probablemente se la vuelva a usar pronto y además sería necesario escribir la página en disco antes de sustituirla.

Cada página pertenece a una de estas cuatro clases. Cuando hace falta sustituir una página, se utiliza el mismo esquema que en el algoritmo del reloj; pero en lugar de examinar si la página a la que estamos apuntando tiene el bit de referencia puesto a 1, examinamos la clase a la que dicha página pertenece. Entonces, sustituimos la primera página que encontremos en la clase no vacía más baja. Observe que podemos tener que recorrer la cola circular varias veces antes de encontrar una página para sustituir.

La principal diferencia entre este algoritmo y el algoritmo más simple del reloj es que aquí damos preferencia a aquellas páginas que no hayan sido modificadas, con el fin de reducir el número de operaciones de E/S requeridas.

9.4.6 Sustitución de páginas basada en contador

Hay muchos otros algoritmos que pueden utilizarse para la sustitución de páginas. Por ejemplo, podemos mantener un contador del número de referencias que se hayan hecho a cada página y desarrollar los siguientes dos esquemas:

- El algoritmo de sustitución de páginas LFU (least frequently used, menos frecuentemente utilizada) requiere sustituir la página que tenga el valor más pequeño de contador. La razón para esta selección es que las páginas más activamente utilizadas deben tener un

valor grande en el contador de referencias. Sin embargo, puede surgir un problema cuando una página se utiliza con gran frecuencia durante la fase inicial de un proceso y luego ya no se la vuelve a utilizar. Puesto que se la emplea con gran frecuencia, tendrá un valor de contador grande y permanecerá en memoria a pesar de que ya no sea necesaria. Una solución consiste en desplazar una posición a la derecha los contenidos del contador a intervalos regulares, obteniendo así un contador de uso medio con decrecimiento exponencial.

- El algoritmo de sustitución de páginas MFU (**most frequently used, más frecuentemente utilizada**) se basa en el argumento de que la página que tenga el valor de contador más pequeño acaba probablemente de ser cargada en memoria y todavía tiene que ser utilizada.

Como cabría esperar, ni el algoritmo de sustitución LFU ni el MFU se utilizan de forma común. La implementación de estos algoritmos resulta bastante cara y tampoco constituyen buenas aproximaciones al algoritmo de sustitución OPT.

9.4.7 Algoritmos de búfer de páginas

Además de un algoritmo de sustitución de páginas específico, a menudo se utilizan otros procedimientos. Por ejemplo, los sistemas suelen mantener un conjunto compartido de marcos libres. Cuando se produce un fallo de página, se selecciona un marco víctima como antes. Sin embargo, la página deseada se lee en un marco libre extraído de ese conjunto compartido, antes de escribir en disco la víctima. Este procedimiento permite que el proceso se reinicie lo antes de posible, sin tener que esperar a que se descargue la página víctima. Posteriormente, cuando la víctima se descarga por fin, su marco se añade al conjunto compartido de marcos libre.

Una posible ampliación de este concepto consiste en mantener una lista de páginas modificadas. Cada vez que el dispositivo de paginación está inactivo, se selecciona una página modificada y se la escribe en el disco, desactivando a continuación su bit de modificación. Este esquema incrementa la probabilidad de que una página esté limpia en el momento de seleccionarla para sustitución, con lo que no será necesario descargarla.

Otra posible modificación consiste en mantener un conjunto compartido de marcos libres, pero recordando qué página estaba almacenada en cada marco. Puesto que el contenido de un marco no se modifica después de escribir el marco en el disco, la página antigua puede reutilizarse directamente a partir del conjunto compartido de marcos libres en caso de que fuera necesaria y si dicho marco no ha sido todavía reutilizado. En este caso, no sería necesaria ninguna operación de E/S. Cuando se produce un fallo de página, primero comprobamos si la página deseada se encuentra en el conjunto compartido de marcos libres. Si no está allí, deberemos seleccionar un marco libre y cargar en él la página deseada.

Esta técnica se utiliza en el sistema VAX/VMS, junto con un algoritmo de sustitución FIFO. Cuando el algoritmo de sustitución FIFO sustituye por error una página que continúa siendo activamente utilizada, dicha página vuelve a ser rápidamente extraída del conjunto de marcos libres y no hace falta ninguna operación de E/S. El búfer de marcos libre proporciona una protección frente al algoritmo FIFO de sustitución, que es relativamente poco eficiente, pero bastante simple. Este método es necesario porque las primeras versiones de VAX no implementaban correctamente el bit de referencia.

Algunas versiones del sistema UNIX utilizan este método en conjunción con el algoritmo de segunda oportunidad y dicho método puede ser una extensión útil de cualquier algoritmo de sustitución de páginas, para reducir el coste en que se incurre si se selecciona la página víctima incorrecta.

9.4.8 Aplicaciones y sustitución de páginas

En ciertos casos, las aplicaciones que acceden a los datos a través de la memoria virtual del sistema operativo tienen un rendimiento peor que si el sistema operativo no proporcionara ningún mecanismo de búfer. Un ejemplo típico sería una base de datos, que proporciona sus propios mecanismos de gestión de memoria y de búferes de E/S. Las aplicaciones como éstas conocen su

utilización de la memoria y del disco mejor de lo que lo hace un sistema operativo, que implementa algoritmos de propósito general. Si el sistema operativo proporciona mecanismos de búfer E/S y la aplicación también, se utilizará el doble de memoria para las operaciones de E/S.

Otro ejemplo serían los almacenes de datos, en los que frecuentemente se realizan lecturas de disco secuenciales masivas, seguidas de cálculos intensivos y escrituras. Un algoritmo LRU serviría para eliminar las páginas antiguas y a preservar las nuevas, mientras que la aplicación tendría que leer más las páginas antiguas que las nuevas (a medida que comience de nuevo sus lecturas secuenciales). En un caso como éste, el algoritmo MFU sería, de hecho, más eficiente que el LRU.

Debido a tales problemas, algunos sistemas operativos dan a ciertos programas especializada la capacidad de utilizar una partición del disco como si fuera una gran matriz secuencial de bloques lógicos, sin ningún tipo de estructura de datos propia de los sistemas de archivos. Esta matriz se denomina en ocasiones **disco sin formato** y las operaciones de E/S que se efectúan sobre esta matriz se denominan E/S sin formato. La E/S sin formato puentea todos los servicios del sistema de archivos, como por ejemplo la paginación bajo demanda para la E/S de archivos, el bloqueo de archivos, la preextracción, la asignación de espacio, los nombres de archivo y los directorios. Observe que, aunque ciertas aplicaciones son más eficientes a la hora de implementar sus propios servicios de almacenamiento de propósito especial en una partición sin formato, la mayoría de las aplicaciones tienen un mejor rendimiento cuando utilizan los servicios normales del sistema de archivos.

9.5 Asignación de marcos

Vamos a volver ahora nuestra atención a la cuestión de la asignación. ¿Cómo asignamos la cantidad fija de memoria libre existente a los distintos procesos? Si tenemos 93 marcos libres y dos procesos, ¿cuántos marcos asignamos a cada proceso?

El caso más simple es el de los sistemas monousuario con 128 KB de memoria compuesta de páginas de 1 KB de tamaño. Este sistema tendrá 128 marcos. El sistema operativo puede ocupar 35 KB, dejando 93 marcos para el proceso de usuario. Con una paginación bajo demanda pura, los 93 marcos se colocarían inicialmente en la lista de marcos libres. Cuando un proceso de usuario comenzara su ejecución, generaría una secuencia de fallos de página. Los primeros 93 fallos de página obtendrían marcos extraídos de la lista de marcos libres. Una vez que se agotara la lista de marcos libres, se utilizaría un algoritmo de sustitución de páginas para seleccionar una de las 93 páginas en memoria con el fin de sustituirla por la página 94, y así sucesivamente. Cuando el proceso terminara, los 93 marcos volverían a ponerse en la lista de marcos libres.

Hay muchas variantes de esta estrategia simple. Podemos, por ejemplo, obligar al sistema operativo a asignar todo su espacio de búferes y de tablas a partir de la lista de marcos libres. Cuando este espacio no esté siendo utilizado por el sistema operativo, podrá emplearse para soportar las necesidades de paginación del usuario. Asimismo, podemos tratar de reservar en todo momento tres marcos libres dentro de la lista de marcos libres; así, cuando se produzca un fallo de página, siempre habrá un marco libre en el que cargar la página. Mientras que esté teniendo lugar el intercambio de la página, se puede elegir un sustituto, que será posteriormente escrito en disco mientras el proceso de usuario continúa ejecutándose. También son posibles otras variantes, pero la estrategia básica está clara: al proceso de usuario se le asignan todos los marcos libres.

9.5.1 Número mínimo de marcos

Nuestras estrategias para la asignación de marcos están restringidas de varias maneras. No podemos, por ejemplo, asignar un número de marcos superior al número total de marcos disponibles (a menos que existan mecanismos de compartición de páginas). Asimismo, debemos asignar al menos un número mínimo de marcos. En esta sección, vamos a examinar más detalladamente este último requisito.

Una razón para asignar al menos un número mínimo de marcos se refiere al rendimiento. Obviamente, a medida que el número de marcos asignados a un proceso se reduzca, se incremen-

tará la tasa de fallos de páginas, ralentizando la ejecución del proceso. Además, recuerde que, cuando se produce un fallo de página antes de completar la ejecución de una instrucción, dicha instrucción debe reiniciarse. En consecuencia, debemos tener suficientes marcos como para albergar todas las diferentes páginas a las que una misma instrucción pudiera hacer referencia.

Por ejemplo, considere una máquina en la que todas las instrucciones que hagan referencia a memoria tengan sólo una dirección de memoria. En este caso, necesitamos al menos un marco para destrucción y otro marco para la referencia a memoria. Además, si se permite un direccionamiento con un nivel de indirección (por ejemplo una instrucción `load` en la página 16 puede hacer referencia a una dirección en la página 0, que es una referencia indirecta a la página 23), entonces el mecanismo de paginación requerirá que haya al menos tres marcos por cada proceso. Piense en lo que sucedería si un proceso sólo dispusiera de dos marcos.

El número mínimo de marcos está definido por la arquitectura informática. Por ejemplo, la instrucción de desplazamiento en el PDP-11 incluye más de una palabra en algunos modos de direccionamiento, por lo que la propia instrucción puede abarcar dos páginas. Además, cada uno de esos dos operandos puede ser una referencia indirecta, lo que nos da un total de seis marcos. Otro ejemplo sería la instrucción `MVC` del IBM 370. Puesto que la instrucción se refiere a un desplazamiento desde una ubicación de almacenamiento a otra, puede ocupar 6 bytes y abarcar dos páginas. El bloque de caracteres que hay que desplazar y el área al que hay que desplazarlo también pueden abarcar dos páginas cada uno. Esta situación requeriría, por tanto, seis marcos. El caso peor se produce cuando la instrucción `MVC` es el operando de una instrucción `EXECUTE` que atravesie una frontera de página; en este caso, necesitamos ocho marcos.

El escenario de caso peor se produce en aquellas arquitecturas informáticas que permiten múltiples niveles de indirección (por ejemplo, cada palabra de 16 bits podría contener una dirección de 15 bits y un indicador de indirección de 1 bit). En teoría, una simple instrucción de carga podría hacer referencia a una dirección indirecta que hiciera referencia a una dirección indirecta (en otra página) que también hiciera referencia a una dirección indirecta (en otra página más), y así sucesivamente, hasta que todas las páginas de la memoria virtual se vieran afectadas. Por tanto, en el peor de los casos, toda la memoria virtual debería estar en memoria física. Para resolver esta dificultad, debemos imponer un límite en el número de niveles de indirección (por ejemplo, limitar cada instrucción a un máximo de 16 niveles de indirección). Cuando se produce la primera indirección, se asigna a un contador el valor 16; después, el contador se decrementa para cada indirección sucesiva que se encuentre en esta instrucción. Si el contador se decrementa hasta alcanzar el valor 0, se produce una interrupción (nivel excesivo de indirección). Esta limitación reduce el número máximo de referencias a memoria por cada instrucción a 17, requiriendo un número igual de marcos.

Mientras que el número mínimo de marcos por proceso está definido por la arquitectura, el número máximo está definido por la cantidad de memoria física disponible. Entre esos dos casos extremos, seguimos teniendo un grado significativo de posibilidades de elección en lo que respecta a la asignación de marcos.

9.5.2 Algoritmos de asignación

La forma más fácil de repartir m marcos entre n procesos consiste en dar a cada uno un número igual de marcos, m/n . Por ejemplo, si hay 93 marcos y cinco procesos, cada proceso obtendrá 18 marcos. Los tres marcos restantes pueden utilizarse como conjunto compartido de marcos libres. Este sistema se denomina **asignación equitativa**.

Una posible alternativa consiste en darse cuenta de que los diversos procesos necesitarán cantidades diferentes de memoria. Considere un sistema con un tamaño de marco de 1 KB. Si los dos únicos procesos que se ejecutan en un sistema con 62 marcos libres son un pequeño proceso de un estudiante que tiene un tamaño de 10 KB y una base de datos interactiva de 127 KB, no tiene mucho sentido asignar a cada proceso 31 marcos. El proceso del estudiante no necesita más de 10 marcos, por lo que los otros 21 estarán siendo literalmente desperdiciados.

Para resolver este problema, podemos utilizar una **asignación proporcional**, asignando la memoria disponible a cada proceso de acuerdo con el tamaño de éste. Sea s_i el tamaño de la memoria virtual para el proceso p_i , y definamos

$$S = \sum s_i.$$

Entonces, si el número total de marcos disponibles es m , asignaremos a_i marcos al proceso i , donde a_i es, aproximadamente,

$$a_i = s_i / S \times m.$$

Por supuesto, deberemos ajustar cada valor a_i para que sea un entero superior al número mínimo de marcos requeridos por el conjunto de instrucciones específico de la máquina, sin que la suma exceda del valor m .

Utilizando un mecanismo de asignación proporcional, repartiríamos 62 marcos entre dos procesos, uno de 10 páginas y otro de 127 páginas, asignando 4 marcos y 57 marcos, respectivamente a los dos procesos, ya que

$$\begin{aligned} 10/137 \times 62 &\approx 4 \\ 127/137 \times 62 &\approx 57. \end{aligned}$$

De esta forma, ambos procesos compartirán los marcos disponibles de acuerdo con sus "necesidades", en lugar de repartir los marcos equitativamente.

Por supuesto, tanto con la asignación equitativa como con la proporcional, las asignaciones concretas pueden variar de acuerdo con el nivel de multiprogramación. Si se incrementa el nivel de multiprogramación, cada proceso perderá algunos marcos con el fin de proporcionar la memoria necesaria para el nuevo proceso. A la inversa, si se reduce el nivel de multiprogramación, los marcos que hubieran sido asignados al proceso terminado pueden distribuirse entre los procesos restantes.

Observe que, con los mecanismos de asignación equitativo y proporcional, a los procesos de alta prioridad se les trata igual que a los de baja prioridad. Sin embargo, por su propia definición, puede que convenga proporcionar al proceso de alta prioridad más memoria con el fin de acelerar su ejecución, en detrimento de los procesos de baja prioridad. Una solución consiste en utilizar un esquema de asignación proporcional en el que el cociente de marcos dependa no del tamaño relativo de los procesos, sino más bien de las prioridades de los procesos, o bien de una combinación del tamaño de la prioridad.

9.5.3 Asignación global y local

Otro factor importante en la forma de asignar los marcos a los diversos procesos es el mecanismo de sustitución de página. Si hay múltiples procesos compitiendo por los marcos, podemos clasificar los algoritmos de sustitución de páginas en dos categorías amplias: **sustitución global** y **sustitución local**. La sustitución global permite a un proceso seleccionar un marco de sustitución de entre el conjunto de todos los marcos, incluso si dicho marco está asignado actualmente a algún otro proceso; es decir, un proceso puede quitar un marco a otro. El mecanismo de sustitución local requiere, por el contrario, que cada proceso sólo efectúe esa selección entre su propio conjunto de marcos asignado.

Por ejemplo, considere un esquema de asignación en el que permitamos a los procesos de alta prioridad seleccionar marcos de los procesos de baja prioridad para la sustitución. Un proceso podrá seleccionar un marco de sustitución de entre sus propios marcos o de entre los marcos de todos los procesos de menor prioridad. Esta técnica permite que un proceso de alta prioridad incremente su tasa de asignación de marcos a expensas de algún proceso de baja prioridad.

Con una estrategia de sustitución local, el número de marcos asignados a un proceso no se modifica. Con la estrategia de sustitución global, puede que un proceso sólo seleccione marcos asignados a otros procesos, incrementando así su propio número de marcos asignados. Suponiendo que otros procesos no seleccionen sus marcos para sustitución.

Un posible problema con el algoritmo de sustitución global es que un proceso no puede comprobar su propia tasa de fallos de página. El conjunto de páginas en memoria para un proceso dependerá no sólo del comportamiento de paginación de dicho proceso, sino también del comportamiento de paginación de los demás procesos. Por tanto, el mismo proceso puede tener un rendimiento completamente distinto (por ejemplo, requiriendo 0,5 segundos para una ejecución y

10,3 segundos para la siguiente) debido a circunstancias totalmente externas al proceso. Este fenómeno no se produce en el caso de los algoritmos de sustitución local. Con una sustitución local, el conjunto de páginas en memoria de un cierto proceso se verá afectado sólo por el comportamiento de paginación de dicho proceso. Por otro lado, la sustitución local puede resultar perjudicial para un proceso, al no tener a su disposición otras páginas de memoria menos utilizadas. En consecuencia, el mecanismo de sustitución global da como resultado, generalmente, una mayor tasa de procesamiento del sistema y es por tanto el método que más comúnmente se utiliza.

9.6 Sobrepaginación

Si el número de marcos asignados a un proceso de baja prioridad cae por debajo del número mínimo requerido por la arquitectura de la máquina, deberemos suspender la ejecución de dicho proceso y a continuación descargar de memoria todas sus restantes páginas, liberando así todos los marcos que tuviera asignados. Este mecanismo introduce un nivel intermedio de planificación de la CPU, basado en la carga y descarga de páginas.

De hecho, pensemos en un proceso que no disponga de “suficientes” marcos. Si el proceso no tiene el número de marcos que necesita para soportar las páginas que se están usando activamente, generará rápidamente fallos de página. En ese momento, deberá sustituir alguna página; sin embargo, como todas sus páginas se están usando activamente, se verá forzado a sustituir una página que va a volver a ser necesaria enseguida. Como consecuencia, vuelve a generar rápidamente una y otra vez sucesivos fallos de página, sustituyendo páginas que se ve forzado a recargar inmediatamente.

Esta alta tasa de actividad de paginación se denomina **sobrepaginación**. Un proceso entrará en sobrepaginación si invierte más tiempo implementando los mecanismos de paginación que en la propia ejecución del proceso.

9.6.1 Causa de la sobrepaginación

La sobrepaginación provoca graves problemas de rendimiento. Considere el siguiente escenario, que está basado en el comportamiento real de los primeros sistemas de paginación que se utilizaron.

El sistema operativo monitoriza la utilización de la CPU. Si esa tasa de utilización es demasiado baja, se incrementa el grado de multiprogramación introduciendo un nuevo proceso en el sistema. Se utiliza un algoritmo de sustitución global de páginas, que sustituye las páginas sin tomar en consideración a qué proceso pertenecen. Ahora suponga que un proceso entra en una nueva fase de ejecución y necesita más marcos de memoria. El proceso comenzará a generar fallos de página y a quitar marcos a otros procesos. Dichos procesos necesitan, sin embargo, esas páginas por lo que también generan fallos de página, quitando marcos a otros procesos. Los procesos que generan los fallos de páginas deben utilizar los procesos de paginación para cargar y descargar las páginas en memoria y, a medida que se ponen en cola para ser servidos por el dispositivo de paginación, la cola de procesos preparados se vacía. Como los procesos están a la espera en el dispositivo de paginación, la tasa de utilización de la CPU disminuye.

El planificador de la CPU ve que la tasa de utilización de la CPU ha descendido e *incrementa* como resultado el grado de multiprogramación. El nuevo proceso tratará de iniciarse quitando marcos a los procesos que se estuvieran ejecutando, haciendo que se provoquen más fallos de página y que la cola del dispositivo de paginación crezca. Como resultado, la utilización de la CPU cae todavía más y el planificador de la CPU trata de incrementar el grado de multiprogramación todavía en mayor medida. Se ha producido una sobrepaginación y la tasa de procesamiento del sistema desciende vertiginosamente, a la vez que se incrementa enormemente la tasa de fallos de página. Como resultado, también se incrementa el tiempo efectivo de acceso a memoria y no se llegará a realizar ningún trabajo útil, porque los procesos invertirán todo su tiempo en los mecanismos de paginación.

Este fenómeno se ilustra en la Figura 9.18, que muestra la tasa de utilización de la CPU en función del grado de multiprogramación. A medida que se incrementa el grado de multiprograma-

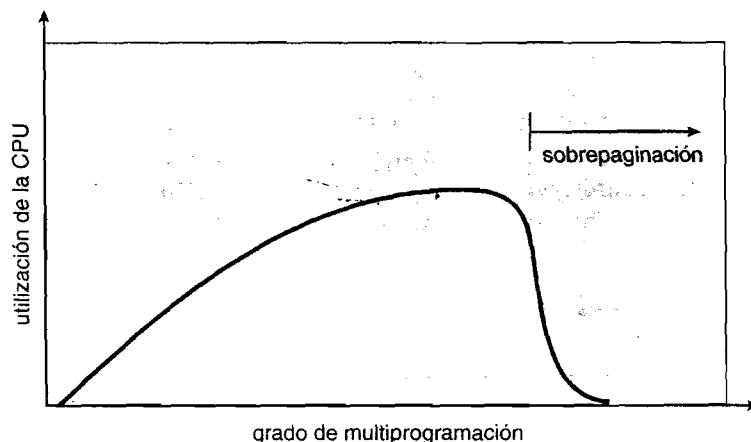


Figura 9.18 Sobrepaginación.

ción, también lo hace la tasa de utilización de la CPU, aunque más lentamente, hasta alcanzar un máximo. Si el grado de multiprogramación se incrementa todavía más, aparece la sobrepaginación y la tasa de utilización de la CPU cae abruptamente. En este punto, para incrementar la tasa de utilización de la CPU y poner fin a la sobrepaginación, es necesario *reducir* el grado de multiprogramación.

Podemos limitar los efectos de la sobrepaginación utilizando un **algoritmo de sustitución local** (o un **algoritmo de sustitución basado en prioridades**). Con la sustitución local, si uno de los procesos entra en sobrepaginación, no puede robar marcos de otro proceso y hacer que éste también entre en sobrepaginación. Sin embargo, esto no resuelve el problema completamente. Si los procesos están en sobrepaginación, se pasarán la mayor parte del tiempo en la cola del dispositivo de paginación. De este modo, el tiempo medio de servicio para un fallo de página se incrementará, debido a que ahora el tiempo medio invertido en la cola del dispositivo de paginación es mayor. Por tanto, el tiempo de acceso efectivo se incrementará incluso para los procesos que no estén en sobrepaginación.

Para prevenir la sobrepaginación, debemos proporcionar a los procesos tantos marcos como necesiten. Pero, ¿cómo sabemos cuántos marcos “necesitan”? Existen varias técnicas distintas. La estrategia basada en conjunto de trabajo (Sección 9.6.2) comienza examinando cuántos marcos está utilizando realmente un proceso; esta técnica define el **modelo de localidad** de ejecución del proceso.

El modelo de localidad afirma que, a medida que un proceso se ejecuta, se va desplazando de una localidad a otra. Una localidad es un conjunto de páginas que se utilizan activamente de forma combinada (Figura 9.19). Todo programa está generalmente compuesto de varias localidades diferentes, que pueden estar solapadas.

Por ejemplo, cuando se invoca una función, ésta define una nueva localidad. En esta localidad, las referencias a memoria se hacen a las instrucciones de la llamada a función, a sus variables locales y a un subconjunto de las variables globales. Cuando salimos de la función, el proceso abandona esta localidad, ya que las variables locales y las instrucciones de la función dejan de ser activamente utilizadas, aunque podemos volver a esta localidad posteriormente.

Por tanto, vemos que las localidades están definidas por la estructura del programa y por sus estructuras de datos. El modelo de localidad afirma que todos los programas exhibirán esta estructura básica de referencias a memoria. Observe que el modelo de localidad es ese principio no iniciado que subyace a las explicaciones sobre las memorias caché que hemos proporcionado hasta el momento en el libro. Si los accesos a los diferentes tipos de datos fueran aleatorios en lugar de seguir determinados patrones, las memorias caché serían inútiles.

Suponga que asignamos a un proceso los suficientes marcos como para acomodar su localidad actual. El proceso generará fallos de página para todas las páginas de su localidad, hasta que todas ellas estén en memoria; a partir de ahí, no volverá a generar fallos de página hasta que cambie de

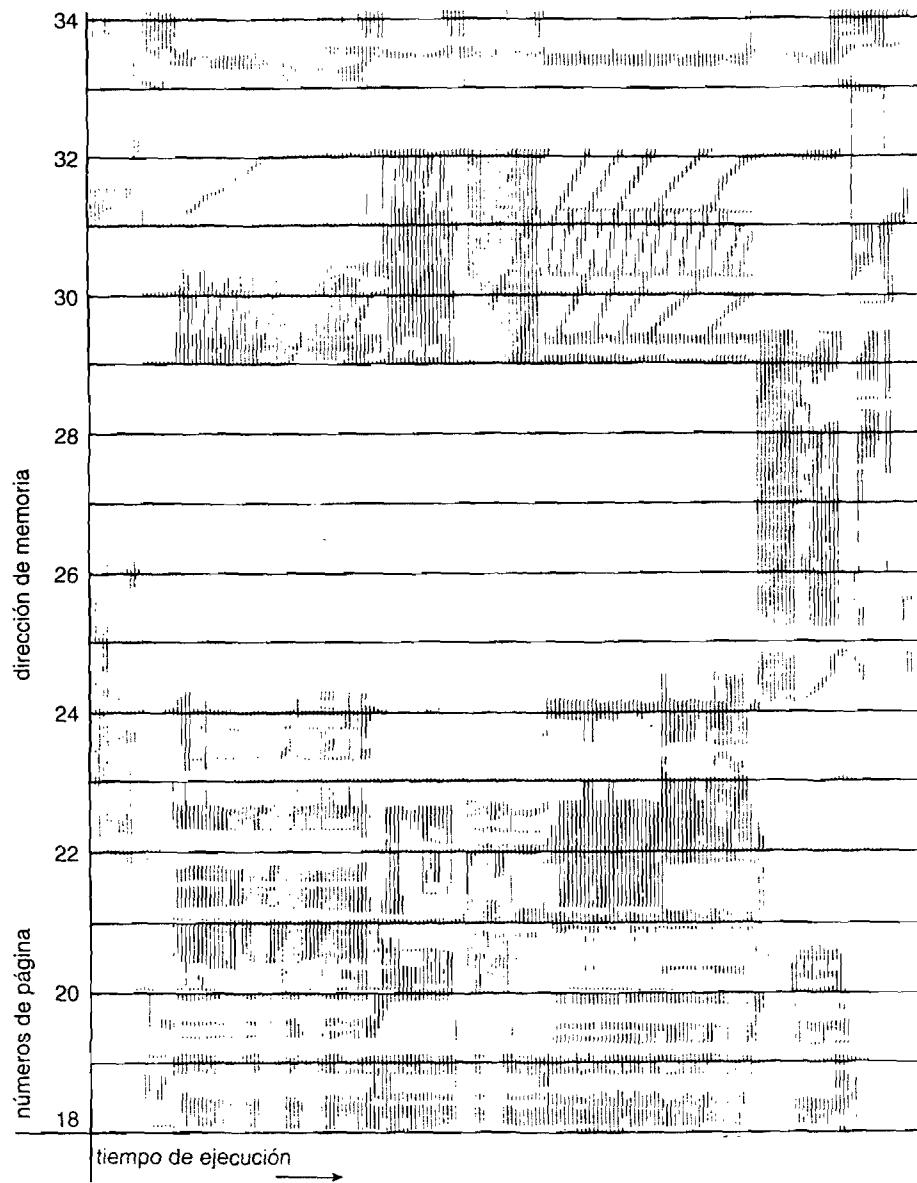


Figura 9.19 Localidad en un patrón de referencias a memoria.

localidad. Si asignamos menos marcos que el tamaño de la localidad actual, el proceso entrará en **sobrepaginación**, ya que no podrá mantener en memoria todas las páginas que esté utilizando activamente.

9.6.2 Modelo del conjunto de trabajo

Como hemos mencionado, el **modelo del conjunto de trabajo** está basado en la suposición de la localidad de ejecución de los programas. Este modelo utiliza un parámetro, Δ , para definir la **ventana del conjunto de trabajo**. La idea consiste en examinar las Δ referencias de página más recientes. El conjunto de páginas en las Δ referencias de páginas más recientes es el **conjunto de trabajo** (Figura 9.20). Si una página está siendo usada de forma activa, se encontrará dentro del conjunto de trabajo. Si ya no está siendo utilizada, será eliminada del conjunto de trabajo Δ unidades de tiempo después de la última referencia que se hiciera a la misma. Por tanto, el conjunto de trabajo es una aproximación de la localidad del programa.

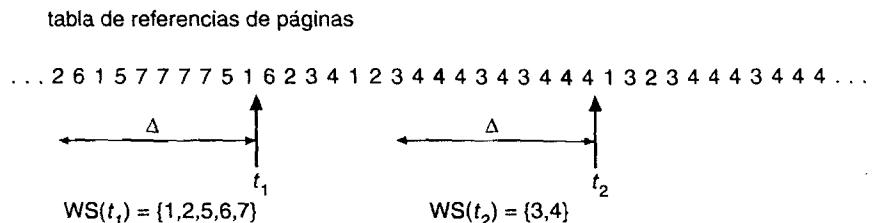


Figura 9.20 Modelo del conjunto de trabajo.

Por ejemplo, dada la secuencia de referencias de memoria mostrada en la Figura 9.20, si $\Delta = 10$ referencias de memoria, entonces el conjunto de trabajo en el instante t_1 será $\{1, 2, 5, 6, 7\}$. En el instante t_2 , el conjunto de trabajo habrá cambiado a $\{3, 4\}$.

La precisión del conjunto de trabajo depende de la selección de Δ . Si Δ es demasiado pequeña, no abarcará la localidad completa; si Δ es demasiado grande, puede que se solapen varias localidades. En el caso extremo, si Δ es infinita, el conjunto de trabajo será el conjunto de páginas utilizadas durante la ejecución del proceso.

La propiedad más importante del conjunto de trabajo es, entonces, su tamaño. Si calculamos el tamaño del conjunto de trabajo, WSS_i , para cada proceso del sistema, podemos considerar que

$$D = \sum WSS_i$$

donde D es la demanda total de marcos. Cada proceso estará utilizando activamente las páginas de su conjunto de trabajo. Así, el proceso i necesita WSS_i marcos. Si la demanda total es superior al número total de marcos disponibles ($D > m$), se producirá una sobrepaginación, porque algunos procesos no dispondrán de los suficientes marcos.

Una vez seleccionada Δ , la utilización del modelo del conjunto de trabajo es muy simple. El sistema operativo monitoriza el conjunto de trabajo de cada proceso y asigna a ese conjunto de trabajo los suficientes marcos como para satisfacer los requisitos de tamaño del conjunto de trabajo. Si hay suficientes marcos adicionales, puede iniciarse otro proceso. Si la suma de los tamaños de los conjuntos de trabajo se incrementa, hasta exceder el número total de marcos disponibles, el sistema operativo seleccionará un proceso para suspenderlo. Las páginas de ese proceso se escribirán (descargarán) y sus marcos se reasignarán a otros procesos. El proceso suspendido puede ser reiniciado posteriormente.

Esta estrategia del conjunto de trabajo impide la sobrepaginación al mismo tiempo que mantiene el grado de multiprogramación con el mayor valor posible. De este modo, se optimiza la tasa de utilización de la CPU.

La dificultad inherente al modelo del conjunto de trabajo es la de controlar cuál es el conjunto de trabajo de cada proceso. La ventana del conjunto de trabajo es una ventana móvil. Con cada referencia a memoria, aparece una nueva referencia en un extremo y la referencia más antigua se pierde por el otro. Una página se encontrará en el conjunto de trabajo si ha sido referenciada en cualquier punto dentro de la ventana del conjunto de trabajo.

Podemos aproximar el modelo del conjunto de trabajo mediante una interrupción de temporización a intervalos fijos y un bit de referencia. Por ejemplo, suponga que Δ es igual a 10000 referencias y que podemos hacer que se produzca una interrupción de temporización cada 5000 referencias. Cada vez que se produzca una interrupción de temporización, copiamos y borramos los valores del bit de referencia de cada página. Así, si se produce un fallo de página, podemos examinar el bit de referencia actual y dos bits que se conservarán en memoria, con el fin de determinar si una página fue utilizada dentro de las últimas 10000 a 15000 referencias. Si fue utilizada, al menos uno de estos bits estará activado. Si no ha sido utilizada, ambos bits estarán desactivados. Las páginas que tengan al menos un bit activado se considerarán como parte del conjunto de trabajo. Observe que este mecanismo no es completamente preciso, porque no podemos decir dónde se produjo una referencia dentro de cada intervalo de 5000 referencias. Podemos reducir la incertidumbre incrementando el número de bits de historial y la frecuencia de las interrupciones (por ejemplo, 10 bits e interrupciones cada 1000 referencias). Sin embargo, el coste para dar servicio a estas interrupciones más frecuentes será correspondientemente mayor.

9.6.3 Frecuencia de fallos de página

El modelo de conjunto de trabajo resulta muy adecuado y el conocimiento de ese conjunto de trabajo puede resultar útil para la prepaginación (Sección 9.9.1), pero parece una forma un tanto torpe de controlar la sobrepaginación. Hay una estrategia más directa que está basada en la **frecuencia de fallos de página** (PFF, page-fault frequency).

El problema específico es cómo prevenir la sobrepaginación. Cuando se entra en sobrepaginación se produce una alta tasa de fallos de página; por tanto, lo que queremos es controlar esa alta tasa. Cuando es demasiado alta, sabemos que el proceso necesita más marcos; a la inversa, si la tasa de fallos de página es demasiado baja, puede que el proceso tenga demasiados marcos asignados. Podemos establecer sendos límites superior e inferior de la tasa deseada de fallos de página (Figura 9.21). Si la tasa real de fallos de página excede del límite superior, asignamos al proceso otro marco, mientras que si esa tasa cae por debajo del límite inferior, eliminamos un marco del proceso. Por tanto, podemos medir y controlar directamente la tasa de fallos de página para evitar la sobrepaginación.

Al igual que con la estrategia del conjunto de trabajo, puede que tengamos que suspender algún proceso. Si la tasa de fallos de página se incrementa y no hay ningún marco libre disponible, deberemos seleccionar algún proceso y suspenderlo. Los marcos liberados se distribuirán entonces entre los procesos que tengan una mayor tasa de fallos de página.

9.7 Archivos mapeados en memoria

Considere una lectura secuencial de un archivo de disco utilizando las llamadas estándar al sistema `open()`, `read()` y `write()`. Cada acceso al archivo requiere una llamada al sistema y un acceso al disco. Alternativamente, podemos utilizar las técnicas de memoria virtual explicadas hasta ahora para tratar la E/S de archivo como si fueran accesos rutinarios a memoria. Esta técnica, conocida con el nombre de **mapeo en memoria** de un archivo, permite asociar lógicamente con el archivo una parte del espacio virtual de direcciones.

9.7.1 Mecanismo básico

El mapeo en memoria de un archivo se lleva a cabo mapeando cada bloque de disco sobre una página (o páginas) de la memoria. El acceso inicial al archivo se produce mediante los mecanismos ordinarios de paginación bajo demanda, provocando un fallo de página. Sin embargo, lo que se hace es leer una parte del archivo equivalente al tamaño de una página, extrayendo los datos del sistema de archivos y depositándolos en una página física (algunos sistemas pueden optar por cargar más de una página de memoria cada vez). Las subsiguientes lecturas y escrituras en el

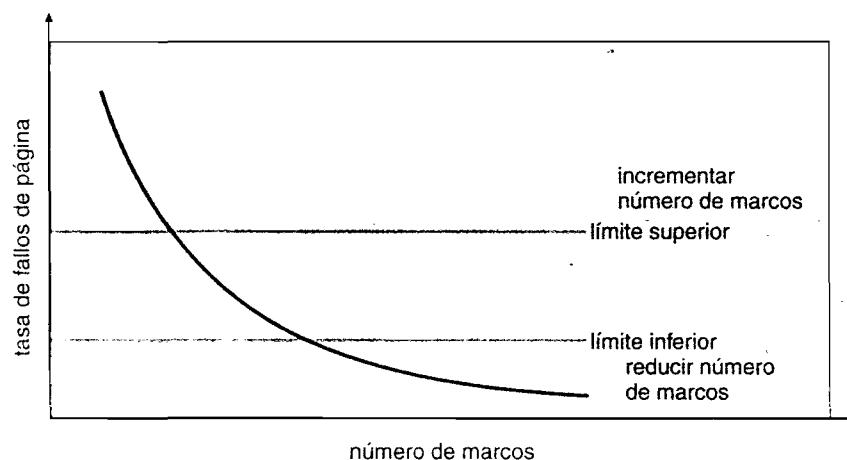


Figura 9.21 Frecuencia de fallos de página.

CONJUNTOS DE TRABAJO Y TASAS DE FALLO DE PÁGINA

Existe una relación entre el consumo de trabajo de un proceso y su tasa de fallos de página. Una estrategia de manejo de memoria que minimiza el consumo de trabajo es la de mover el conjunto de trabajo de un proceso cuando se desplaza por la memoria. Sin embargo, al almacenar el consumo de trabajo de un proceso en función del tiempo, se observa que la tasa de fallos de página del proceso no alternan entre una serie de valles y picos a lo largo del tiempo. Este comportamiento general se ilustra en la figura.

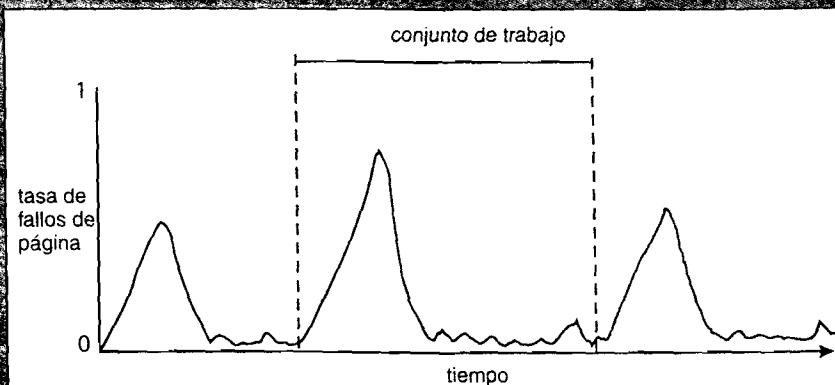


FIGURA 9.22 Tasa de fallos de página en función del tiempo.

Los picos y valles de la tasa de fallos de página se producen cuando una página, bajo demanda, las referencias correspondientes a una nueva ubicación. Entonces, una vez que el conjunto de trabajo de esa nueva localidad se ejecuta, la tasa de fallos de página caerá. Cuando el proceso se desplace a un nuevo conjunto de trabajo, la tasa de fallos de página volverá de nuevo a exhibir un pico, volviendo posteriormente a una tasa menor una vez que el nuevo conjunto de trabajo se haya cargado en memoria. El intervalo de tiempo entre el comienzo de un pico y del siguiente ilustra la transición entre un conjunto de trabajo y otro.

archivo se gestionarán como accesos normales de memoria, simplificando así el acceso de archivo y también su utilización, al permitir que el sistema manipule los archivos a través de la memoria en lugar de incurrir en la carga de trabajo adicional asociada a la utilización de las llamadas al sistema `read()` y `write()`.

Observe que las escrituras en el archivo mapeado en memoria no se transforman necesariamente en escrituras inmediatas (síncronas) en el archivo del disco. Algunos sistemas pueden elegir actualizar el archivo físico cuando el sistema operativo compruebe periódicamente si la página de memoria ha sido modificada. Cuando el archivo se cierre, todos los datos mapeados en memoria se volverán a escribir en el disco y serán eliminados de la memoria virtual del proceso.

Algunos sistemas operativos proporcionan mapeo de memoria únicamente a través de una llamada al sistema específica y utilizan las llamadas al sistema normales para realizar todas las demás operaciones de E/S de archivo. Sin embargo, los otros sistemas prefieren mapear en memoria un archivo independientemente de si ese archivo se ha especificado como archivo mapeado en memoria o no. Tomemos Solaris como ejemplo: si se especifica un archivo como mapeado en memoria (utilizando la llamada al sistema, `mmap()`), Solaris mapeará el archivo en el espacio de direcciones del proceso; si se abre un archivo y se accede a él utilizando las llamadas al sistema normales, como `open()`, `read()` y `write()`, Solaris continuará mapeando en memoria el archivo, pero ese archivo estará mapeado sobre el espacio de direcciones del kernel. Por tanto,

independientemente de cómo se abra el archivo, Solaris trata todas las operaciones de E/S de archivo como operaciones mapeadas en memoria, permitiendo que el acceso de archivo tenga lugar a través del eficiente subsistema de memoria.

Puede dejarse que múltiples procesos mapeen de forma concurrente el mismo archivo, con el fin de permitir la compartición de datos. Las escrituras realizaras por cualquiera de los procesos modificarán los datos contenidos en la memoria virtual y esas modificaciones podrán ser vistas por todos los demás procesos que haya mapeado la misma sección del archivo. Teniendo en cuenta nuestras explicaciones anteriores sobre la memoria virtual, debe quedar claro cómo se implementa la compartición de las secciones mapeadas en memoria: el mapa de memoria virtual de cada proceso que participa en esa compartición apunta a la misma página de la memoria física, es decir, a la página que alberga una copia del bloque de disco. Este esquema de compartición de memoria se ilustra en la Figura 9.23. Las llamadas al sistema para mapeo en memoria pueden también soportar la funcionalidad de copia durante la escritura, permitiendo a los procesos compartir un archivo en modo de sólo lectura, pero disponiendo de sus propias copias de los datos que modifiquen. Para poder coordinar el acceso a los datos compartidos, los procesos implicados pueden utilizar alguno de los mecanismos de exclusión mutua que se describen en el Capítulo 6.

En muchos aspectos, la compartición de archivos mapeados en memoria es similar a los mecanismos de memoria compartida, descritos en la Sección 3.4.1. No todos los sistemas utilizan el mismo mecanismo para ambos tipos de funcionalidad. En los sistemas UNIX y Linux, por ejemplo, el mapeo de memoria se realiza mediante la llamada al sistema `mmap()`, mientras que la memoria compartida se implementa con las llamadas al sistema `shmget()` y `shmat()` compatibles con POSIX (Sección 3.5.1). Sin embargo, en los sistemas Windows NT, 2000 y XP, los mecanismos de memoria compartida se implementan mapeando en memoria archivos. En estos sistemas, los procesos pueden comunicarse utilizando memoria compartida, haciendo que los procesos que se quieren comunicar mapeen en memoria el mismo archivo dentro de su espacio virtual de direcciones. El archivo mapeado en memoria sirve como región de memoria compartida entre los procesos que se tengan que comunicar (Figura 9.24). En la sección siguiente, vamos a ilustrar el soporte existente en la API Win32 para la implementación del mecanismo de memoria compartida utilizando archivos mapeados en memoria.

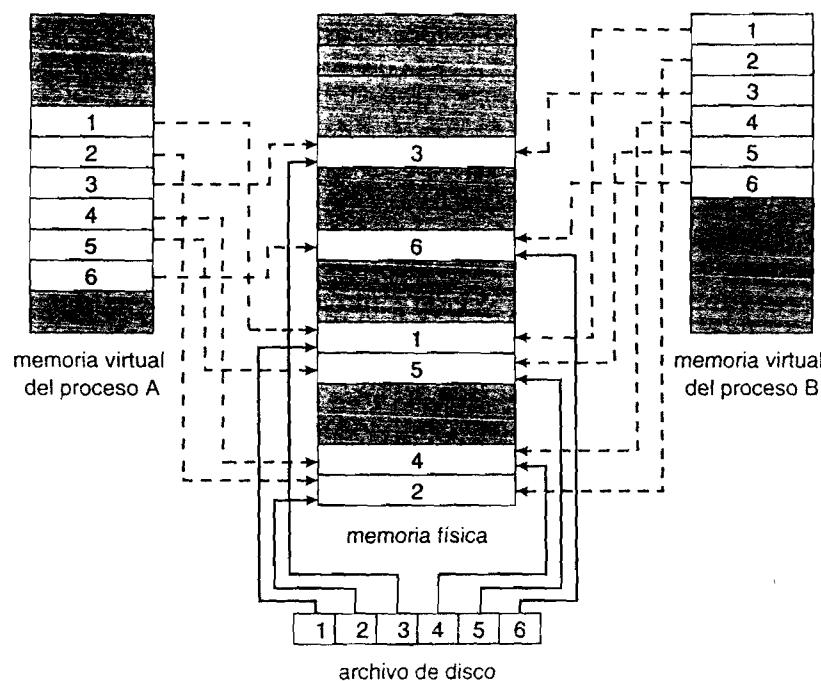


Figura 9.23 Archivos mapeados en memoria.

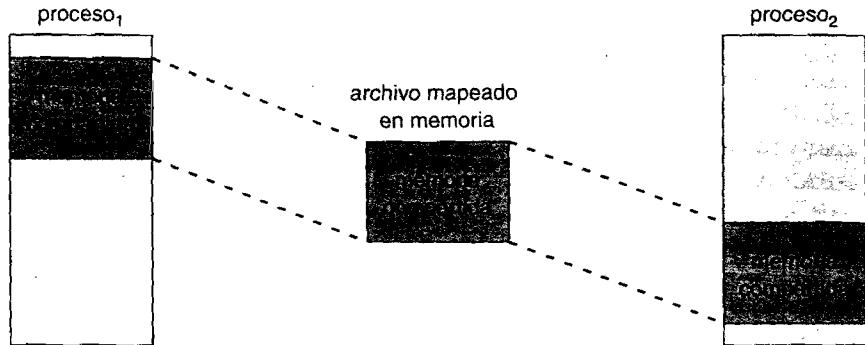


Figura 9.24 Memoria compartida en Windows utilizando E/S mapeada en memoria.

9.7.2 Memoria compartida en la API Win32

El esquema general para crear una región de memoria compartida utilizando archivos mapeados en memoria en la API Win32 implica crear primero un **mapeo de archivo** para el archivo que hay que mapear y luego establecer una *vista* del archivo mapeado dentro del espacio virtual de direcciones de un proceso. Un segundo proceso podrá entonces abrir y crear una vista del archivo mapeado dentro de su propio espacio virtual de direcciones. El archivo mapeado representa el objeto de memoria compartida que permitirá que se produzca la comunicación entre los dos procesos.

A continuación vamos a ilustrar estos pasos con más detalle. En este ejemplo, un proceso productor crea primero un objeto de memoria compartida utilizando las características de mapeado en memoria disponibles en la API Win32. El productor escribe entonces un mensaje en la memoria compartida. Después de eso, un proceso consumidor abre un mapeo al objeto de memoria compartida y lee el mensaje escrito por el productor.

Para establecer un archivo mapeado en memoria, un proceso tiene primero que abrir el archivo que hay que mapear mediante la función `CreateFile()`, que devuelve un descriptor HANDLE al archivo abierto. El proceso crea entonces un mapeo de este descriptor de archivo utilizando la función `CreateFileMapping()`. Una vez establecido el mapeo del archivo, el proceso establece una vista del archivo mapeado en su propio espacio virtual de direcciones mediante la función `MapViewOfFile()`. La vista del archivo mapeado representa la parte del archivo que se está mapeando dentro del espacio virtual de direcciones del proceso (puede mapearse el archivo completo o sólo una parte del mismo). Ilustramos esta secuencia en el programa que se muestra en la Figura 9.25 (en el programa se ha eliminado buena parte del código de comprobación de errores, en aras de la brevedad).

La llamada a `CreateFileMapping()` crea un **objeto nominado de memoria compartida** denominado `SharedObject`. El proceso consumidor se comunicará utilizando este segmento de memoria compartida por el procedimiento de crear un mapeo del mismo objeto nominado. El productor crea entonces una vista del archivo mapeado en memoria en su espacio virtual de direcciones. Pasando a los últimos tres parámetros el valor 0 indica que la vista mapeada debe ser del archivo completo. Podría haber pasado, en su lugar, sendos valores que especificaran un desplazamiento y un tamaño, creando así una vista que contuviera sólo una subsección del archivo. Es importante resaltar que puede que no todo el mapeo se cargue en memoria en el momento de definirlo; en lugar de ello, el archivo mapeado puede cargarse mediante un mecanismo de paginación bajo demanda, trayendo así las páginas a memoria sólo a medida que se acceda a ellas. La función `MapViewOfFile()` devuelve un puntero al objeto de memoria compartida; todos los accesos a esta ubicación de memoria serán, por tanto, accesos al archivo mapeado en memoria. En este caso, el proceso productor escribe el mensaje “Mensaje de memoria compartida” en la memoria compartida.

En la Figura 9.26 se muestra un programa que ilustra cómo establece el proceso consumidor una vista del objeto nominado de memoria compartida. Este programa es algo más simple que el

```

#include <windows.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    HANDLE hFile, hMapFile;
    LPVOID lpMapAddress;

    hFile = CreateFile("temp.txt", // nombre archivo
                       GENERIC_READ | GENERIC_WRITE, // acceso de lectura/escritura
                       0, // sin compartición del archivo
                       NULL, // seguridad predeterminada
                       OPEN_ALWAYS, // abrir archivo nuevo o existente
                       FILE_ATTRIBUTE_NORMAL, // atributos de archivo normales
                       NULL); // sin plantilla de archivo

    hMapFile = CreateFileMapping(hFile, // descriptor de archivo
                                NULL, // seguridad predeterminada
                                PAGE_READWRITE, // acceso de lectura/escritura a las páginas mapeadas
                                0, // mapear archivo completo
                                0,
                                TEXT("SharedObject")); // objeto nominado de memoria compartida

    lpMapAddress = MapViewOfFile(hMapFile, // descriptor de objeto mapeado
                                FILE_MAP_ALL_ACCESS, // acceso de lectura/escritura
                                0, // vista mapeada del archivo completo
                                0,
                                0);

    // escribir en memoria compartida
    sprintf(lpMapAddress, "Mensaje de memoria compartida");

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hFile);
    CloseHandle(hMapFile);
}

```

Figura 9.25 Un productor que escribe en memoria compartida utilizando la API Win32.

que se muestra en la Figura 9.25, ya que lo único que hace falta es que el proceso cree un mapeado del objeto nominado de memoria compartida existente. El proceso consumidor deberá crear también una vista del archivo mapeado, al igual que hizo el proceso productor en el programa de la Figura 9.25. Después, el consumidor lee de la memoria compartida el mensaje “Mensaje de memoria compartida” que fue escrito por el proceso productor.

Finalmente, ambos procesos eliminan la vista del archivo mapeado mediante una llamada a `UnmapViewOfFile()`. Al final de este capítulo, proporcionamos un ejercicio de programación que utiliza la memoria compartida mediante los mecanismos de mapeo de memoria de la API Win32.

9.7.3 E/S mapeada en memoria

En el caso de la E/S, como hemos mencionado en la Sección 1.2.1, cada controlador de E/S incluye registros para almacenar comandos y los datos que hay que transferir. Usualmente, una serie de instrucciones de E/S especiales permiten transferir los datos entre estos registros y la memo-

```

#include <windows.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    HANDLE hMapFile;
    LPVOID lpMapAddress;

    hMapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, // acceso de
                               // lectura/escritura
                               FALSE, // sin herencia
                               TEXT("SharedObject")); // nombre de objeto del archivo mapeado

    lpMapAddress = MapViewOfFile(hMapFile, // descripto del objeto mapeado
                                FILE_MAP_ALL_ACCESS, // acceso de lectura/escritura
                                0, // vista mapeada del archivo completo
                                0,
                                0);

    // leer de la memoria compartida
    printf("Mensaje leído: %s", lpMapAddress);

    UnmapViewOfFile(lpMapAddress);
    CloseHandle(hMapFile);
}

```

Figura 9.26 Un proceso consumidor lee de la memoria compartida utilizando la API Win32.

ria del sistema. Para permitir un acceso más cómodo a los dispositivos de E/S, muchas arquitecturas informáticas proporcionan **E/S mapeada en memoria**. En este caso, se reservan una serie de rangos de direcciones de memoria y se mapean esas direcciones sobre los registros de los dispositivos. Las lecturas y escrituras en estas direcciones de memoria hacen que se transfieran datos hacia y desde los registros de los dispositivos. Este método resulta apropiado para los dispositivos que tengan un rápido tiempo de respuesta, como los controladores de vídeo. En el IBM PC, cada ubicación de la pantalla se mapea sobre una dirección de memoria. De este modo, mostrar texto en la pantalla es casi tan sencillo como escribir el texto en las ubicaciones apropiadas mapeadas en memoria.

La E/S mapeada en memoria también resulta conveniente para otros dispositivos, como los puertos serie y paralelo utilizados para conectar módems e impresoras a una computadora. La CPU transfiere datos a través de estos tipos de dispositivos leyendo y escribiendo unos cuantos registros de los dispositivos, denominados **puertos** de E/S. Para enviar una larga cadena de bytes a través de un puerto serie mapeado en memoria, la CPU escribe un byte de datos en el registro de datos y activa un bit en el registro de control para indicar que el byte está disponible. El dispositivo toma el byte de datos y borra a continuación el bit del registro de control, para indicar que está listo para recibir el siguiente byte. Entonces, la CPU puede transferir ese siguiente byte al dispositivo. Si la CPU utiliza un mecanismo de sondeo para comprobar el estado del bit de control, ejecutando constantemente un bucle para ver si el dispositivo está listo, este método de operaciones se denomina **E/S programada** (PIO, programmed I/O). Si la CPU no sondea el bit de control, sino que recibe una interrupción cuando el dispositivo está listo para el siguiente byte, se dice que la transferencia de datos está **dirigida por interrupciones**.

9.8 Asignación de la memoria del *kernel*

Cuando un proceso que se está ejecutando en modo usuario solicita memoria adicional, las páginas se asignan a partir de la lista de marcos de página libres mantenida por el *kernel*. Esta lista

suele rellenarse utilizando un algoritmo de sustitución de páginas como los que hemos expuesto en la Sección 9.4 y lo más probable es que contenga páginas libres que estarán dispersas por toda la memoria física, como se ha explicado anteriormente. Recuerde también que, si un proceso de usuario solicita un único byte de memoria, se producirá una fragmentación interna, ya que al proceso se le concederá un marco de página completo.

Sin embargo, la memoria del *kernel* suele asignarse a partir de un conjunto compartido de memoria compartida que es distinto de la lista utilizada para satisfacer las necesidades de los procesos normales que se ejecutan en modo usuario. Hay dos razones principales para que esto sea así:

1. El *kernel* solicita memoria para estructuras de datos de tamaños variables, algunas de las cuales tiene un tamaño inferior a una página. Como resultado, el *kernel* debe utilizar la memoria de manera conservadora y tratar de minimizar el desperdicio de memoria debido a la fragmentación. Esto es especialmente importante porque muchos sistemas operativos no aplican el sistema de paginación al código y a los datos del *kernel*.
2. Las páginas asignadas a los procesos en modo usuario no necesariamente tienen que estar en memoria física contigua. Sin embargo, ciertos dispositivos hardware interaccionan directamente con la memoria física (sin tener las ventajas de una interfaz de memoria virtual) y, consecuentemente, pueden requerir memoria que resida en páginas físicas contiguas.

En las siguientes secciones, vamos a examinar dos estrategias para la gestión de la memoria libre asignada a los procesos del *kernel*.

9.8.1 Descomposición binaria

El sistema de descomposición binaria (buddy system) asigna la memoria a partir de un segmento de tamaño fijo compuesto de páginas físicamente contiguas. La memoria se asigna a partir de este segmento mediante un **asignador de potencias de 2**, que satisface las solicitudes en unidades cuyo tamaño es una potencia de 2 (4 KB, 8 KB, 16 KB, etc.). Toda solicitud cuyas unidades no tengan el tamaño apropiado se redondeará hasta la siguiente potencia de 2 más alta. Por ejemplo, si se solicitan 11 KB, esa solicitud se satisfará mediante un segmento de 16 KB. Vamos a explicar a continuación la operación del sistema de descomposición binaria mediante un ejemplo simple.

Supongamos que el tamaño de un segmento de memoria es inicialmente de 256 KB y que el *kernel* solicita 21 KB de memoria. El segmento se dividirá inicialmente en dos *subsegmentos* (que denominaremos A_L y A_R), cada uno de los cuales tendrá un tamaño igual a 128 KB. Uno de estos

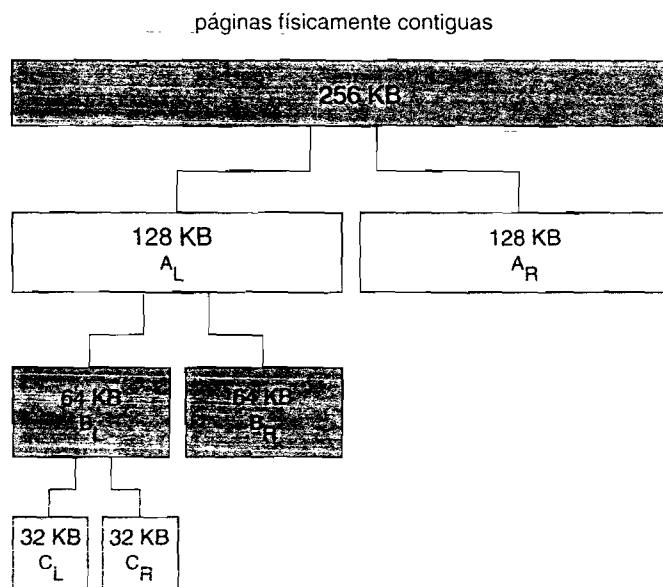


Figura 9.27 Asignación de la descomposición binaria.

subsegmentos se vuelve a subdividir en dos subsegmentos de 64 KB, B_L y B_R . Sin embargo, la siguiente potencia de dos más alta para 21 KB es 32 KB, por lo que B_L o B_R tendrán que subdividirse de nuevo en dos subsegmentos de 32 KB, C_L y C_R . Uno de estos subsegmentos se utilizará para satisfacer la solicitud de 21 KB. Este esquema se ilustra en la Figura 9.27, donde C_L es el segmento asignado a la solicitud de 21 KB.

Una ventaja del sistema de descomposición binaria es la rapidez con la que pueden combinarse subsegmentos adyacentes para formar segmentos de mayor tamaño, utilizando una técnica denominada **consolidación**. En la Figura 9.27, por ejemplo, cuando el *kernel* libere la unidad C_L que fue asignada, el sistema puede consolidar C_L y C_R para formar un segmento de 64 KB. Este segmento, B_L , puede a su vez consolidarse con su compañero, B_R , para formar un segmento de 128 KB. En último término, podemos terminar obteniendo el segmento de 256 KB original.

La desventaja obvia del sistema de descomposición binaria es que el redondeo de la siguiente potencia de 2 más alta producirá, muy probablemente, que aparezca fragmentación dentro de los segmentos asignados. Por ejemplo, una solicitud de 32 KB sólo puede satisfacerse mediante un segmento de 64 KB. De hecho, no podemos garantizar que vaya a desperdiciarse menos del 50 por ciento de la unidad asignada debido a la fragmentación interna. En la sección siguiente, vamos a explorar un esquema de asignación de memoria en el que no se pierde ningún espacio debido a la fragmentación.

9.8.2 Asignación de franjas

Una segunda estrategia para la asignación de la memoria del *kernel* se conoce con el nombre de **asignación de franjas** (slabs). Una **franja** está formada por una o más páginas físicamente contiguas. Una **caché** está compuesta de una o más franjas. Existe una única caché por cada estructura de datos del *kernel* distinta; por ejemplo, hay una caché para la estructura de datos que representa los descriptores de los procesos, otra caché distinta para los objetos de archivo, otra más para los semáforos, etc. Cada caché se rellena de **objetos** que sean instancias de la estructura de datos del *kernel* que esa caché representa. Por ejemplo, la caché que representa los semáforos almacena instancias de objetos semáforo. La caché que representa descriptores de procesos almacena instancias de objetos descriptor de procesos, etc. La relación entre franjas, cachés y objetos se muestra en la Figura 9.28. La figura muestra dos objetos del *kernel* de 3 KB de tamaño y tres objetos de 7 KB. Estos objetos están almacenados en sus respectivas cachés.

El algoritmo de asignación y franjas utiliza cachés para almacenar objetos del *kernel*. Cuando se crea una caché, se asigna a la caché un cierto número de objetos (que están inicialmente marcados como **libres**). El número de objetos de la caché dependerá del tamaño de la franja asociada.

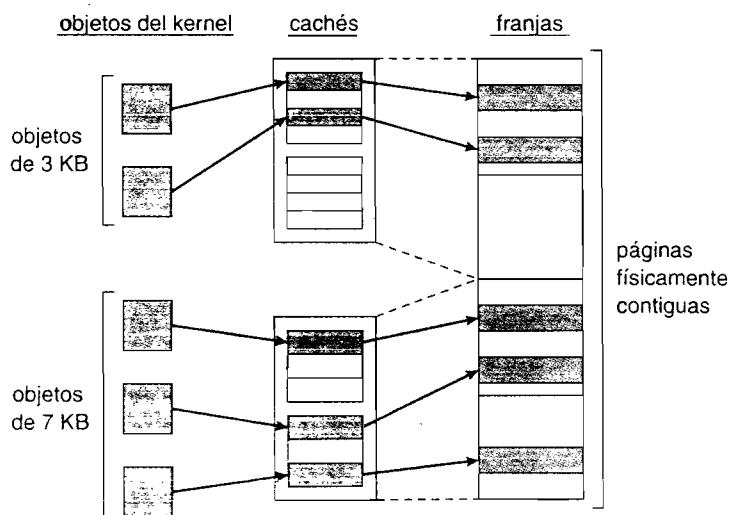


Figura 9.28 Asignación de franjas.

Por ejemplo, una franja de 12 KB (compuesta de tres páginas contiguas de 4 KB) podría almacenar seis objetos de 2 KB. Inicialmente, todos los objetos de la caché se marcan como libres. Cuando hace falta un nuevo objeto para una estructura de datos del *kernel*, el asignador puede asignar cualquier objeto libre de la caché para satisfacer dicha solicitud. El objeto asignado de la caché se marca como **usado**.

Veamos un ejemplo en el que el *kernel* solicita memoria al asignador de franjas para un objeto que representa un descriptor de proceso. En los sistemas Linux, un descriptor de procesos es del tipo `task_struct`, que requiere aproximadamente 1,7 KB de memoria. Cuando el *kernel* de Linux crea una nueva tarea, solicita la memoria necesaria para el objeto `task_struct` de su correspondiente caché. La caché satisfará la solicitud utilizando un objeto `task_struct` que ya haya sido asignado en una franja y esté marcado como libre.

En Linux, una franja puede estar en una de tres posibles estados:

1. **Llena.** Todos los objetos de la franja están marcados como utilizados.
2. **Vacía.** Todos los objetos de la franja están marcados como libres.
3. **Parcial.** La franja contiene tanto objetos usados como libres.

El asignador de franjas trata primero de satisfacer la solicitud con un objeto libre de una franja parcialmente llena. Si no hay ninguna, se asigna un objeto libre de una franja vacía. Si tampoco hay disponibles franjas vacías, se asigna una nueva franja compuesta por páginas físicas contiguas y esa franja se asigna a una caché; la memoria para el objeto se asigna a partir de esta franja.

El asignador de franjas tiene dos ventajas principales:

1. No se pierde memoria debido a la fragmentación. La fragmentación no es un problema porque cada tipo de estructura de datos del *kernel* tiene una caché asociada, y cada caché está compuesta de una o más franjas que están divididas en segmentos cuyo tamaño coincide con el de los objetos que representan. Así, cuando el *kernel* solicita memoria para un objeto, el asignador de franjas devuelve la cantidad exacta de memoria requerida para representar el objeto.
2. Las solicitudes de memoria pueden satisfacerse rápidamente. El esquema de asignación de franjas es, por tanto, particularmente efectivo para gestionar la memoria en aquellas situaciones en que los objetos se asignan y desasignan frecuentemente, como suele ser el caso con las solicitudes del *kernel*. El acto de asignar y liberar memoria puede ser un proceso que consume mucho tiempo; sin embargo, los objetos se crean de antemano y pueden, por tanto, ser asignados rápidamente a partir de la caché. Además, cuando el *kernel* ha terminado de trabajar con un objeto y lo libera, se lo marca como libre y se lo devuelve a la caché, haciendo que esté así inmediatamente disponible para las siguientes solicitudes procedentes del *kernel*.

El asignador de franjas apareció por primera vez en el *kernel* de Solaris 2.4. Debido a su naturaleza de propósito general, este asignador se usa también ahora para ciertas solicitudes de memoria en modo usuario en Solaris. Linux utilizaba originalmente el asignador basado en descomposición binaria; sin embargo, a partir de la versión 2.2, el *kernel* de Linux adoptó el mecanismo de asignación de franjas.

9.9 Otras consideraciones

Las principales decisiones que tenemos que tomar para un sistema de paginación son la selección de un algoritmo de sustitución y de una política de asignación, temas ambos de los que ya hemos hablado en este capítulo. Hay también otras muchas consideraciones, y a continuación vamos a presentar algunas de ellas.

9.9.1 Prepaginación

Una característica obvia del mecanismo de paginación bajo demanda pura es el gran número de fallos de página que se producen en el momento de iniciar un proceso. Esta situación se produce

al tratar de cargar la localidad inicial en memoria. La misma situación puede surgir en otros momentos; por ejemplo, cuando se reinicia un proceso que hubiera sido descargado, todas las páginas se encontrarán en el disco y cada una de ellas deberá ser cargada como resultado del propio fallo de página. La **prepaginación** es un intento de evitar este alto nivel de paginación inicial. La estrategia consiste en cargar en memoria de una sola vez todas las páginas que vayan a ser necesarias. Algunos sistemas operativos, y en especial Solaris, prepagan los marcos de página para los archivos de pequeño tamaño.

Por ejemplo, en un sistema que utilice el modelo del conjunto de trabajo, se mantiene con cada proceso una lista de las páginas de su conjunto de trabajo. Si fuera preciso suspender un proceso (debido a una espera de E/S o a una falta de marcos libres), recordaríamos el conjunto de trabajo para dicho proceso. Cuando el proceso pueda reanudarse (porque la E/S haya finalizado o porque hayan pasado a estar disponibles los suficientes marcos libres), cargaremos automáticamente en memoria su conjunto de trabajo completo antes de reiniciar el proceso.

La prepaginación puede ser muy ventajosa en algunos casos. La cuestión es simplemente, si el coste de utilizar la prepaginación es menor que el coste de dar servicio a los correspondientes fallos de página. Podría darse el caso de que muchas de las páginas que fueran cargadas en memoria por el mecanismo de prepaginación no llegaran a utilizarse.

Suponga que se prepagan s páginas y que una fragmentación α de esas s páginas llega a utilizarse ($0 \leq \alpha \leq 1$). La cuestión es si el coste de los $s^*\alpha$ fallos de página que nos ahorraremos es mayor o menor que el coste de prepagar $s^*(1 - \alpha)$ páginas innecesarias. Si α está próxima a 0 la prepaginación no será aconsejable; si α está próxima a 1, la prepaginación será la solución más adecuada.

9.9.2 Tamaño de página

Los diseñadores de un sistema operativo para una máquina existente raramente pueden elegir el tamaño de página. Sin embargo, cuando se están diseñando nuevas plataformas, es necesario tomar una decisión en lo referente a cuál es el tamaño de página más adecuado. Como cabría esperar, no hay ningún tamaño de página que sea mejor en términos absolutos. En lugar de ello, lo que hay es una serie de factores que favorecen los distintos tamaños. Los tamaños de página son siempre potencias de 2, y suelen ir generalmente de 4096 (2^{12}) a 4.194.304 (2^{22}) bytes.

¿Cómo seleccionamos un tamaño de página? Uno de los aspectos que hay que tener en cuenta es el tamaño de la tabla de páginas. Para un espacio de memoria virtual determinado, al reducir el tamaño de página se incrementa el número de páginas y, por tanto, el tamaño de la tabla de páginas. Para una memoria virtual de 4 MB (2^{22}), por ejemplo, habría 4096 páginas de 1024 bytes pero sólo 512 páginas de 8192 bytes. Puesto que cada proceso activo debe tener su propia copia de la tabla de páginas, conviene utilizar un tamaño de página grande.

Por otro lado, la memoria se aprovecha mejor si las páginas son más pequeñas. Si a un proceso se le asigna memoria comenzando en la ubicación 00000 y continuando hasta que tenga toda la que necesite, lo más probable es que no termine exactamente en una frontera de página. Por tanto, será necesario asignar una parte de la página final (porque las páginas son las unidades de asignación) pero una porción del espacio de esa página no se utilizará (dando lugar al fenómeno de la fragmentación interna). Suponiendo que existe una independencia entre el tamaño de los procesos y el tamaño de las páginas, podemos esperar que, como promedio, la mitad de la página final de cada proceso se desperdicie. Esta pérdida sólo será de 256 bytes para una página de 512 bytes, pero de 4096 bytes para una página de 8192 bytes. Por tanto, para minimizar la fragmentación interna, necesitamos utilizar un tamaño de página pequeño.

Otro problema es el tiempo requerido para leer o escribir una página. El tiempo de E/S está compuesto de los tiempos de búsqueda (posicionamiento de cabezales), latencia y transferencia. El tiempo de transferencia es proporcional a la cantidad de datos transferida (es decir, al tamaño de página), un hecho que parece actuar en favor de los tamaños de página pequeños. Sin embargo, como veremos en la Sección 12.1.1, los tiempos de latencia y de búsqueda normalmente son mucho mayores que el tiempo de transferencia. Para una tasa de transferencia de 2 MB por segundo, sólo hacen falta 0,2 milisegundos para transferir 512 bytes; sin embargo, la latencia puede ser

de 8 milisegundos y el tiempo de búsqueda de 20 milisegundos. Por tanto, del tiempo total de E/S (28,2 milisegundos), sólo un 1 por ciento es atribuible a la propia transferencia de datos. Doblando el tamaño de página, se incrementa el tiempo de E/S a sólo 28,4 milisegundos, lo que quiere decir que hacen falta 28,4 milisegundos para leer una única página de 1024 bytes, pero 56,4 milisegundos para leer esa misma cantidad de datos en forma de dos páginas de 512 bytes cada una. Por tanto, el deseo de minimizar el tiempo de E/S es un argumento en favor de los tamaños de página grandes.

Por otro lado, con un tamaño de página pequeño, la cantidad total de E/S debería reducirse, ya que se mejora la localidad. Un tamaño de página pequeño permite que cada página se ajuste con la localidad de un programa de forma más precisa. Por ejemplo, considere un proceso de 200 KB de tamaño, de los que sólo la mitad (100 KB) se utilice realmente durante una ejecución. Si sólo tenemos una única página grande, deberemos cargar la página completa, lo que significa que habrá que transferir y asignar un total de 200 KB. Si, por el contrario, tuviéramos páginas de sólo 1 byte, podríamos cargar en memoria únicamente los 100 KB realmente utilizados, lo que hace que sólo se transfieran y asignen 100 KB. Con un tamaño de página más pequeño, tenemos una mejor **resolución**, lo que nos permite usar sólo la memoria que sea realmente necesaria. Con un tamaño de página grande, debemos asignar y transferir no sólo lo que necesitamos, sino también cualquier otra cosa que esté en la página, sea necesaria o no. Por tanto, un tamaño de página más pequeño debería reducir la cantidad de operaciones de E/S y la cantidad total de memoria asignada.

Pero, ¿se ha dado cuenta de que, con un tamaño de página de 1 byte, tendríamos un fallo de página por *cada* byte? Un proceso de 200 KB que utilizara sólo la mitad de esa memoria generaría un único fallo de página si el tamaño de página fuera de 200 KB, pero daría lugar a 102.400 fallos de página si el tamaño de página fuera de 1 byte. Cada fallo de página requiere una gran cantidad de trabajo adicional, en forma de procesamiento de la interrupción, almacenamiento de los registros, sustitución de una página, puesta en cola en el dispositivo de paginación y actualización de las tablas. Para minimizar el número de fallos de página, necesitamos disponer de un tamaño de página grande.

También es necesario tener en cuenta otros factores (como por ejemplo la relación entre el tamaño de página y el tamaño de sector en el dispositivo de paginación). El problema no tiene una única respuesta. Como hemos visto, algunos factores (fragmentación interna, localidad) favorecen la utilización de tamaños de página pequeños, mientras que otros (tamaño de la tabla, tiempo de E/S) constituyen argumentos en pro de un tamaño de página grande. Sin embargo, la tendencia histórica es hacia los tamaños de página grandes. De hecho, la primera edición de este libro (1983) utilizaba 4096 bytes como límite superior de los tamaños de página y este valor era el más común en 1990. Sin embargo, los sistemas modernos pueden utilizar ahora tamaños de página mucho mayores, como veremos en la siguiente sección.

9.9.3 Alcance del TLB

En el Capítulo 8 hemos presentado el concepto de **tasa de aciertos** del TLB. Recuerde que esta tasa de aciertos hace referencia al porcentaje de traducciones de direcciones virtuales que se resuelven mediante el TLB, en lugar de recurrir a la tabla de páginas. Claramente, la tasa de aciertos está relacionada con el número de entradas contenidas en el TLB y la forma de incrementar la tasa de aciertos es incrementar ese número de entradas. Sin embargo, esta solución tiene su coste, ya que la memoria asociativa utilizada para construir el TLB es muy cara y consume mucha potencia.

Relacionada con la tasa de aciertos, existe otra métrica similar: el **alcance del TLB**. El alcance del TLB hace referencia a la cantidad de memoria accesible a partir del TLB y es, simplemente, el número de entradas multiplicado por el tamaño de página. Idealmente, en el TLB se almacenará el conjunto de trabajo completo de un proceso. Si no es así, el proceso invertirá una considerable cantidad de tiempo resolviendo referencias a memoria mediante la tabla de páginas, en lugar de mediante el TLB. Si doblamos el número de entradas del TLB, se doblará el alcance del TLB; sin embargo, para algunas aplicaciones que utilizan intensivamente la memoria, puede que esto siga siendo insuficiente para poder almacenar el conjunto de trabajo.

Otra técnica para incrementar el alcance del TLB consiste en incrementar el tamaño de la página o proporcionar tamaños de página múltiples. Si incrementamos el tamaño de página (por ejemplo, de 8 KB a 32 KB), cuadruplicamos el alcance del TLB. Sin embargo, esto puede conducir a una mayor fragmentación en algunas aplicaciones que no requieran un tamaño de página tan grande, de 32 KB. Alternativamente, los sistemas operativos pueden proporcionar varios tamaños de página distintos. Por ejemplo, UltraSPARC permite tamaños de página de 8 KB, 64 KB, 512 KB y 4 MB. De estos tamaños de página disponibles, Solaris utiliza páginas de 8 KB y de 4 MB. Y con un TLB de 64 entradas, el alcance del TLB en Solaris va de 512 KB con páginas de 8 KB a 256 MB con páginas de 4 MB. Para la mayoría de las aplicaciones, el tamaño de página de 8 KB es suficiente, aunque Solaris mapea los primeros 4 MB del código y los datos del *kernel* mediante dos páginas de 4 MB. Solaris también permite que las aplicaciones (por ejemplo, las bases de datos) aprovechen el tamaño de página mayor, de 4 MB.

Proporcionar soporte para tamaños múltiples de página requiere que sea el sistema operativo (y no el hardware) el que gestione el TLB. Por ejemplo, uno de los campos de una entrada del TLB debe indicar el tamaño del marco de página correspondiente a la entrada del TLB. Gestionar el TLB por software y no por hardware tiene un impacto en el rendimiento; sin embargo, el incremento en la tasa de aciertos y en el alcance del TLB compensan suficientemente este coste. De hecho, las tendencias recientes indican que cada vez se emplean más los TLB gestionados por software y que cada vez es más común que los sistemas operativos soporten tamaños múltiples de página. Las arquitecturas UltraSPARC, MIPS y Alpha emplean el mecanismo de TLB gestionado por software. PowerPC y Pentium gestionan el TLB por hardware.

9.9.4 Tablas de páginas invertidas

En la Sección 8.5.3 hemos presentado el concepto de tabla de páginas invertida. El propósito de este tipo de gestión de páginas consiste en reducir la cantidad de memoria física necesaria para controlar la traducción de direcciones virtuales a físicas. Conseguimos este ahorro creando una tabla que dispone de una entrada por cada tabla de memoria física, indexada mediante la pareja <id-proceso, número-página>.

Puesto que mantienen información acerca de qué página de memoria virtual está almacenada en cada marco físico, las tablas de páginas invertidas reducen la cantidad de memoria física necesaria para almacenar esta información. Sin embargo, la tabla de páginas invertida ya no contiene información completa acerca del espacio lógico de direcciones de un proceso y dicha información es necesaria si una página a la que se haga referencia no se encuentra actualmente en memoria; los mecanismos de paginación bajo demanda necesitan esta información para procesar los fallos de página. Para que esta información esté disponible, es necesario guardar una tabla de páginas externa por cada proceso. Cada una de esas tablas es similar a la tabla de páginas tradicional de cada proceso y contiene información sobre la ubicación de cada página virtual.

Pero, ¿no implican estas tablas externas de páginas que estamos restando utilidad a las tablas de páginas invertidas? En realidad no, porque, como esas tablas sólo se consultan cuando se produce un fallo de página, no es necesario que estén disponibles de manera rápida. En lugar de ello, se las puede cargar y descargar de memoria según sea necesario, mediante el mecanismo de paginación. Desafortunadamente, esto implica que un fallo de página puede ahora hacer que el gestor de memoria virtual genere otro fallo de página al cargar la tabla de páginas externa que necesita para localizar la página virtual en el dispositivo de almacenamiento. Este caso especial requiere una cuidadosa gestión dentro del *kernel* y provoca un retardo en el procesamiento de búsqueda de páginas.

9.9.5 Estructura de los programas

La paginación bajo demanda está diseñada para ser transparente para el programa de usuario. En muchos casos, el usuario no será consciente de la naturaleza paginada de la memoria. En otros casos, sin embargo, puede mejorarse el rendimiento del sistema si el usuario (o el compilador) conoce el mecanismo subyacente de paginación bajo demanda.

Veamos un ejemplo sencillo, pero bastante ilustrativo. Suponga que las páginas tienen 128 palabras de tamaño. Considere un programa C cuya función sea inicializar con el valor 0 cada elemento de una matriz de 128 por 128. El siguiente código sería bastante típico:

```
int i, j;
int [128] [128] data;

for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i][j] = 0;
```

Observe que la matriz se almacena por filas, es decir, la matriz se almacena de la forma $\text{data}[0][0]$, $\text{data}[0][1]$, . . . , $\text{data}[0][127]$, $\text{data}[1][0]$, $\text{data}[1][1]$, . . . , $\text{data}[127][127]$. Para páginas de 128 palabras, cada fila ocupará una página. Por tanto, el código anterior almacena un cero en una palabra de cada página, luego en otra palabra de cada página, y así sucesivamente. Si el sistema operativo asigna menos de 128 marcos al programa, la ejecución de éste provocará $128 \times 128 = 16384$ fallos de página. Por el contrario, si cambiamos el código a

```
int i, j;
int [128] [128] data;

for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i][j] = 0;
```

ahora se escribirá el valor cero en todas las palabras de una página antes de comenzar con la página siguiente, reduciendo el número de fallos de página a 128.

Una cuidadosa selección de las estructuras de datos y de las estructuras de programación puede incrementar la localidad de los programas y, por tanto, reducir la tasa de fallos de página y el número de páginas que componen el conjunto de trabajo. Por ejemplo, una pila tiene una buena localidad, ya que siempre se accede a ella por la parte superior. Por contraste, una tabla hash está diseñada para dispersar las referencias, lo que da como resultado una mala localidad. Por supuesto, la localidad de referencia es sólo una de las medidas de la eficiencia de uso de una estructura de datos; otros factores muy importantes son la velocidad de búsqueda, el número total de referencias a memoria y el número total de páginas usadas.

En una etapa posterior, el compilador y el cargador pueden tener un efecto significativo sobre la paginación. Si sepáramos el código y los datos y generamos código reentrant, las páginas de código podrán ser de sólo lectura y no serán, por tanto, modificadas nunca. Las páginas limpias no tienen que ser descargadas a disco para poderlas sustituir. El cargador, por su parte, puede tratar de evitar colocar las rutinas de modo que atraviesen las fronteras de las páginas, manteniendo cada rutina completamente en una página. Las rutinas que se invoquen unas a otras muchas veces pueden empaquetarse en una misma página. Este empaquetamiento es una variante del problema del empaquetamiento de envases, que es un problema tradicional en el campo de la investigación operativa: tratar de empaquetar los segmentos de carga de tamaño variable en las páginas de tamaño fijo de modo que se minimicen las referencias interpáginas. Dicha técnica resulta particularmente útil para los tamaños de página grandes.

La elección del lenguaje de programación también puede afectar a la programación. Por ejemplo, C y C++ utilizan punteros frecuentemente y los punteros tienden a aleatorizar el acceso a la memoria, disminuyendo así, posiblemente, la localidad de un proceso. Algunos estudios han mostrado que los programas orientados a objetos también tienden a tener una baja localidad de referencia.

9.9.6 Interbloqueo de E/S

Cuando se utiliza la paginación bajo demanda, en ocasiones es necesario permitir que ciertas páginas queden **bloqueadas** en memoria. Una de tales situaciones es cuando se realizan operaciones

de E/S hacia o desde la memoria de usuario (virtual). La E/S suele implementarse mediante un procesador de E/S independiente. Por ejemplo, al controlador de un dispositivo de almacenamiento USB generalmente se le proporciona el número de bytes que hay que transferir y una dirección de memoria para el búfer (Figura 9.29). Cuando la transferencia se completa, dicho procesador interrumpe a la CPU.

Debemos asegurarnos de que no se produzca la siguiente secuencia de sucesos: un proceso realiza una solicitud de E/S y es puesto en una cola para dicho dispositivo de E/S; mientras tanto, la CPU se asigna a otros procesos; estos procesos provocan fallos de páginas y uno de ellos, utilizando un algoritmo de sustitución global, sustituye la página que contiene el búfer de memoria del proceso en espera, por lo que dichas páginas se descargan; algún tiempo después, cuando la solicitud de E/S llegue a la parte inicial de la cola del dispositivo, la E/S se realizará utilizando la dirección especificada; sin embargo, dicho marco está siendo ahora utilizado para una página distinta, que pertenece a otro proceso.

Existen dos soluciones comunes a este problema: una solución consiste en no ejecutar nunca las operaciones de E/S sobre la memoria de usuario. En lugar de ello, los datos se copian siempre entre la memoria del sistema y la memoria del usuario y la E/S sólo tiene lugar entre la memoria del sistema y la E/S del dispositivo. Para escribir un bloque en una cinta, primero copiamos el bloque en la memoria del sistema y luego lo escribimos en cinta. Esta operación adicional de copia puede provocar un trabajo adicional inaceptablemente alto.

Otra solución consiste en permitir bloquear páginas en memoria. En este caso, se asocia un bit de bloqueo con cada marco. Si el marco está bloqueado, no podrá ser seleccionado para sustitución. Con este mecanismo, para escribir un bloque en una cinta, bloquearíamos en memoria las páginas que contienen el bloque. El sistema puede entonces continuar de la forma usual, ya que las páginas bloqueadas no podrán ser sustituidas. Cuando la E/S se complete, las páginas se desbloquearán.

Los bits de bloqueo se utilizan en diversas situaciones. Frecuentemente, parte del *kernel* del sistema operativo, o incluso todo él, está bloqueado en memoria, ya que muchos sistemas operativos no pueden tolerar que se produzcan fallos de página provocados por el *kernel*.

Otra utilidad de los bits de bloqueo está relacionada con la sustitución normal de páginas. Considere la siguiente secuencia de sucesos: un proceso de baja prioridad provoca un fallo de página y, después de seleccionar un marco de sustitución, el sistema de paginación lee la página necesaria en memoria; como ahora está listo para continuar, el proceso de baja prioridad entra en

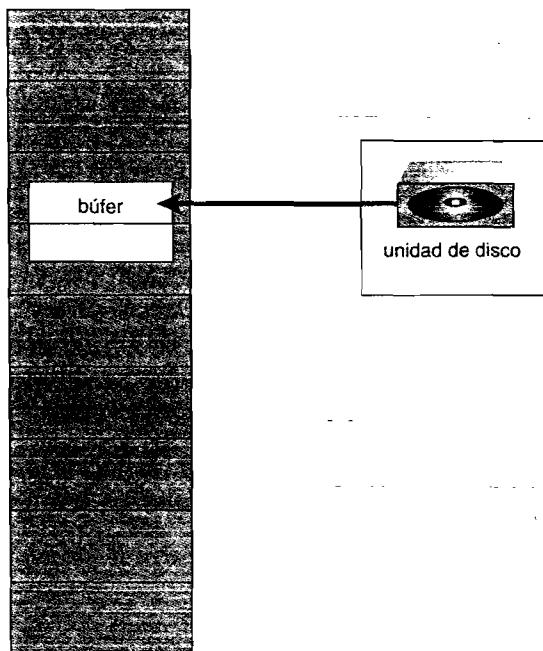


Figura 9.29 La razón por la que los marcos utilizados para E/S deben estar en memoria.

la cola de procesos preparados y espera a que le asignen la CPU pero, como se trata de un proceso de baja prioridad, puede que no sea seleccionado por el planificador de la CPU durante un tiempo considerable; mientras que este proceso de baja prioridad está esperando, otro proceso de alta prioridad genera un fallo de página; al buscar un sustituto, el sistema de paginación ve que hay una página en memoria que no ha sido referenciada ni modificada y que es, precisamente, la página que el proceso de baja prioridad acaba de cargar; esta página parece un candidato perfecto para la sustitución, ya que está limpia y no será necesario escribirla, y aparentemente no ha sido utilizado durante un largo tiempo.

El que el proceso de alta prioridad deba poder sustituir la página del proceso de baja prioridad es una decisión de política. Después de todo, simplemente estamos retardando el proceso de baja prioridad en beneficio del que tiene una prioridad más alta. Sin embargo, estamos desperdiciando el esfuerzo invertido en cargar la página para el proceso de baja prioridad. Si decidimos impedir la sustitución de una página recién cargada hasta que pueda ser usada al menos una vez, podemos utilizar el bit de bloqueo para implementar este mecanismo. Cuando se selecciona una página para sustitución, su bit de bloqueo se activa y permanece activado hasta que el proceso que generó el fallo de página vuelva a ocupar la CPU.

La utilización de un bit de bloqueo puede ser peligrosa: puede que el bit de bloqueo se active y no vuelva a desactivarse nunca. Si esta situación se produjera (debido a un error en el sistema operativo, por ejemplo), el marco bloqueado dejará de ser utilizable. En un sistema monousuario, el abuso de estos bloqueos sólo dañaría al usuario que los efectúe, pero los sistemas multiusuario no pueden confiar tanto en sus usuarios. Por ejemplo, Solaris permite que se proporcionen “consejos” de bloqueo, pero el sistema operativo es libre para descartar esos consejos si el conjunto compartido de marcos libres llega a ser demasiado pequeño o si un proceso determinado requiere que se bloqueen en memoria demasiadas páginas.

9.10 Ejemplos de sistemas operativos

En esta sección, vamos a describir el modo en que Windows XP y Solaris implementan la memoria virtual.

9.10.1 Windows XP

Windows XP implementa la memoria virtual utilizando paginación bajo demanda con *clustering*. El *clustering* gestiona los fallos de página cargando no sólo la página que generó el fallo sino también varias páginas sucesivas. Cuando se crea por primera vez un proceso, se le asigna un valor mínimo y máximo al conjunto de trabajo. El **mínimo del conjunto de trabajo** es el número mínimo de páginas que se garantiza que el proceso tendrá en memoria. Si hay suficiente memoria disponible, puede asignarse a un proceso tantas páginas como indique su **máximo del conjunto de trabajo**. Para la mayoría de las aplicaciones, el valor del mínimo y el máximo del conjunto de trabajo son 50 y 345 páginas, respectivamente (en algunas circunstancias, puede permitirse a un proceso exceder su máximo del conjunto de trabajo). El gestor de memoria virtual mantiene una lista de marcos de página libres, lista con la que hay asociado un valor de umbral que se utiliza para indicar si hay suficiente memoria libre disponible. Si se produce un fallo de página para un proceso que esté por debajo de su máximo del conjunto de trabajo, el gestor de memoria virtual le asignará una página de este lista de páginas libres. Si un proceso se encuentra ya en su máximo del conjunto de trabajo y genera un fallo de página, deberá seleccionar una página para sustitución empleando una política de sustitución de páginas local.

Cuando la cantidad de memoria libre cae por debajo del umbral, el gestor de memoria virtual utiliza una táctica conocida con el nombre de **ajuste automático del conjunto de trabajo** para restaurar el valor por encima del umbral. El ajuste automático del conjunto de trabajo funciona evaluando el número de páginas asignadas a los procesos. Si a un proceso se le han asignado más páginas de las que indica su mínimo del conjunto de trabajo, el gestor de memoria virtual eliminará páginas hasta que el proceso alcance dicho mínimo. A los procesos que estén por debajo de

su mínimo del conjunto de trabajo se les puede asignar páginas de la lista de marcos de páginas libres una vez que haya suficiente memoria libre disponible.

El algoritmo utilizado para determinar qué página hay que eliminar del conjunto de trabajo depende del tipo de procesador. En los sistemas 80x86 monoprocesador, Windows XP utiliza una variable del algoritmo de reloj expuesta en la Sección 9.4.5.2. En los sistemas Alpha y los sistemas x86 multiprocesador, el borrado del bit de referencia puede requerir que se invalide la entrada en el búfer de consultar de traducción de los otros procesadores. En lugar de asumir toda esta carga de trabajo adicional, Windows XP utiliza una variante del algoritmo FIFO expuesto en la Sección 9.4.2.

9.10.2 Solaris

En Solaris, cuando una hebra genera un fallo de página, el *kernel* asigna una página a dicha hebra a partir de la lista de páginas libres que mantiene. Por tanto, es obligatorio que el *kernel* mantenga disponible una cantidad suficiente de memoria libre. Asociado con esta lista de páginas libres hay un parámetro (*lotsfree*) que representa un umbral para el comienzo de la paginación. El parámetro *lotsfree* tiene típicamente un valor igual a 1/64 del tamaño de la memoria física. Cuatro veces por segundo, el *kernel* comprueba si la cantidad de memoria libre es inferior a *lotsfree*. Si el número de páginas libres cae por debajo de *lotsfree*, se inicia un proceso denominado *pageout*. El proceso *pageout* es similar al algoritmo de segunda oportunidad descrito en la Sección 9.4.5.2, salvo porque utiliza dos manecillas a la hora de explorar las páginas, en lugar de la única manecilla que se describe en la Sección 9.4.5.2. El proceso *pageout* funciona de la forma siguiente. La manecilla frontal del reloj explora todas las páginas de la memoria, asignando el valor 0 al bit de referencia. Más adelante, la manecilla posterior del reloj examina el bit de referencia de las páginas que hay en memoria, añadiendo a la lista libre aquellas páginas cuyo bit siga teniendo el valor 0 y escribiendo en disco su contenido, si es que se ha modificado. Solaris mantiene en caché una lista de páginas que han sido “liberadas” pero que todavía no han sido sobreescritas. La lista libre contiene marcos cuyo contenido no es válido. Pueden reclamarse páginas de la caché si se accede a ellas antes de que sean transferidas a la lista libre.

El algoritmo *pageout* utiliza varios parámetros para controlar la tasa de exploración de las páginas (conocida con el nombre de *scanrate*). La tasa de exploración se expresa en páginas por segundo y va desde un valor mínimo (*slowscan*) a un valor máximo (*fastscan*). Cuando la memoria libre cae por debajo de *lotsfree*, la exploración se produce a una velocidad de *slowscan* páginas por segundo y va acelerándose hasta el valor *fastscan*, dependiendo de la cantidad de memoria libre disponible. El valor predeterminado de *slowscan* es de 100 páginas por segundo, mientras que *fastscan* tiene, normalmente, el valor (*páginas físicas totales*)/2 páginas por segundo, con un máximo de 8192 páginas por segundo. Esto se muestra en la Figura 9.30 (donde *fastscan* tiene su valor máximo).

La distancia (en páginas) entre las manecillas del reloj está determinada por un parámetro del sistema (*handspread*). La cantidad de tiempo que transcurre entre el proceso de borrado de un bit por parte de la manecilla frontal y la consulta de su valor por parte de la manecilla posterior dependerá de los valores *scanrate* y *handspread*. Si *scanrate* es 100 páginas por segundo y *handspread* es 1024 páginas, pueden pasar 10 segundos entre el momento en que la manecilla frontal activa un bit y el momento en que la manecilla posterior lo comprueba. Sin embargo, debido a la gran cantidad de trabajo que se suele imponer al sistema de memoria, no resulta raro que *scanrate* tenga un valor de varios miles. Esto quiere decir que la cantidad de tiempo entre el momento que se borra un bit y se consulta su valor puede ser fácilmente de unos cuantos segundos.

Como hemos dicho anteriormente, el proceso *pageout* comprueba la memoria cuatro veces por segundo. Sin embargo, si la memoria libre cae por debajo del valor *desfree* (Figura 9.30), *pageout* se ejecutará 100 veces por segundo con la intención de mantener disponible una cantidad de memoria libre igual al menos a *desfree*. Si el proceso *pageout* es incapaz de mantener una cantidad de memoria libre igual a *desfree* durante 30 segundos, el *kernel* comenzará a descargar procesos, liberando así todas las páginas asignadas a los procesos descargados. En general, el *kernel* seleccionará los procesos que hayan estado inactivos durante largos períodos de tiempo. Si el sistema es



Figura 9.30 Explorador de página de Solaris.

incapaz de mantener una cantidad de memoria libre igual al menos al valor *minfree*, el proceso *pageout* será invocado para cada solicitud de una nueva página.

Las versiones más recientes del *kernel* de Solaris han incluido una serie de mejoras en el algoritmo de paginación. Una de tales mejoras implica reconocer las páginas que pertenecen a bibliotecas compartidas. Las páginas que pertenecen a bibliotecas que están siendo compartidas por varios procesos (incluso si son elegibles para ser reclamadas por el explorador de páginas) serán omitidas durante el proceso de exploración de las páginas. Otra mejora consiste en distinguir entre las páginas que han sido asignadas a los procesos y las páginas asignadas a los archivos normales. Este mecanismo se conoce con el nombre de **paginación con prioridad** y se analiza en la Sección 11.6.2.

9.11 Resumen

Resulta deseable poder ejecutar un proceso cuyo espacio lógico de direcciones sea mayor que el espacio físico de direcciones disponible. La memoria virtual es una técnica que nos permite mapear un espacio lógico de direcciones de gran tamaño sobre una memoria física más pequeña. La memoria virtual nos permite ejecutar procesos extremadamente grandes e incrementar el grado de multiprogramación, aumentando así la tasa de utilización de la CPU. Además, evita que los programadores de aplicaciones tengan que preocuparse acerca de la disponibilidad de memoria. Además, con la memoria virtual, varios procesos pueden compartir las bibliotecas del sistema y la memoria. La memoria virtual también nos permite utilizar un eficiente tipo de mecanismo de creación de procesos conocido con el nombre de copia durante la escritura y que es un mecanismo mediante el que los procesos padre e hijo comparten páginas de la memoria.

La memoria virtual se suele implementar mediante un mecanismo de paginación bajo demanda. En la paginación bajo demanda púra nunca se carga una página en memoria hasta que se haga referencia a esa página. La primera referencia provoca un fallo de página que deberá ser tratado por el sistema operativo. El *kernel* del sistema operativo consulta una tabla interna para determinar dónde está ubicada la página dentro del dispositivo de almacenamiento del respaldo. A continuación, localiza un marco libre y carga en él la página desde ese dispositivo de almacenamiento. La tabla de páginas se actualiza para reflejar este cambio y a continuación se reinicia la instrucción que provocó el fallo de página. Esta técnica permite que un proceso se ejecute aun cuando no se encuentre completa en la memoria principal toda su imagen de memoria. Siempre que la tasa de fallos de páginas sea razonablemente baja, el rendimiento será aceptable.

Podemos utilizar la paginación bajo demanda para reducir el número de marcos asignados a un proceso. Este mecanismo puede incrementar el grado de multiprogramación (permitiendo que haya más procesos disponibles para ejecución en cada momento) y, al menos en teoría, la tasa de utilización de la CPU del sistema. También permite que se ejecuten procesos aunque sus requisitos totales de memoria excedan de la memoria física disponible total. Dichos procesos se ejecutan en memoria virtual.

Si los requisitos totales de memoria sobrepasan la memoria física disponible, puede que sea necesario sustituir páginas de la memoria, con el fin de liberar marcos para cargar las páginas. Se utilizan diversos algoritmos de sustitución de páginas. La sustitución FIFO es fácil de programar pero sufre de la denominada anomalía de Belady. La sustitución óptima de páginas requiere un conocimiento futuro sobre la secuencia de referencias a memoria. La sustitución LRU es una aproximación de la sustitución óptima de páginas, pero puede resultar difícil de implementar. La mayoría de los algoritmos de sustitución de páginas, como el algoritmo de segunda oportunidad, son aproximaciones del mecanismo de sustitución LRU.

Además del algoritmo de sustitución de páginas, hace falta definir una política de asignación de marcos. La asignación puede ser fija, lo que favorece una sustitución local de páginas, o dinámica, lo que favorece una sustitución global. El modelo del conjunto de trabajo presupone que los procesos se ejecutan en localidades. El conjunto de trabajo es el conjunto de páginas contenidas en la localidad actual. Correspondientemente, a cada proceso debe asignársele los suficientes marcos para su conjunto de trabajo actual. Si un proceso no dispone de la suficiente memoria para su conjunto de trabajo, entrará en sobrepaginación. Proporcionar los suficientes marcos a cada proceso para evitar la sobrepaginación puede requerir mecanismos de intercambio y planificación de procesos.

La mayoría de los sistemas operativos proporcionan funciones para mapear en memoria los archivos, permitiendo así tratar la E/S de archivo como si fuera una serie de accesos normales a memoria. La API Win32 implementa la memoria compartida mediante un mecanismo de mapeo en memoria de archivos.

Los procesos del *kernel* suelen requerir que se les asigne memoria utilizando páginas que sean físicamente contiguas. El sistema de descomposición binaria asigna la memoria a los procesos del *kernel* en una serie de unidades cuyo tamaño es una potencia de 2, lo que provoca a menudo que aparezca el fenómeno de la fragmentación. El mecanismo de asignación de franjas asigna las estructuras de datos del *kernel* a una serie de cachés asociadas con franjas, que estén compuestas de una o más páginas físicamente contiguas. Con la asignación de franjas, no se desperdicia memoria debido a la fragmentación y las solicitudes de memoria pueden satisfacerse rápidamente.

Además de requerir que resolvamos los problemas principales, que son la sustitución de páginas y la asignación de marcos, un apropiado diseño de un sistema de paginación requiere que tengamos en cuenta el tamaño de las páginas, la E/S, el bloqueo, la prepaginación, la creación de procesos, la estructura de los programas y otras cuestiones.

Ejercicios

- 9.1 Proporcione un ejemplo que ilustre el problema que existe al reiniciar la instrucción de movimiento en bloques (MVC) en el IBM 360/370 cuando las regiones de origen y de destino se solapan.
- 9.2 Explique el soporte hardware requerido para implementar la paginación bajo demanda.
- 9.3 ¿Qué es la característica de copia durante la escritura y en qué circunstancias es ventajoso usar estas características? ¿Cuál es el soporte hardware requerido para implementar esta característica?
- 9.4 Una cierta computadora proporciona a sus usuarios un espacio de memoria virtual de 2^{32} bytes. La computadora tiene 2^{18} bytes de memoria física. La memoria virtual se implementa mediante paginación y el tamaño de página es de 4096 bytes. Un proceso de usuario

genera la dirección virtual 11123456. Explique cómo calcula el sistema la correspondiente ubicación física. Distinga entre operaciones software y hardware.

- 9.5 Suponga que tenemos una memoria con paginación bajo demanda. La tabla de páginas se almacena en registros. Hacen falta 8 milisegundos para dar servicio a un fallo de página si hay disponible un marco libre o si la página sustituida no ha sido modificada y 20 milisegundos si la página sustituida ha sido modificada. El tiempo de acceso a memoria es de 100 nanosegundos.

Suponga que la página que hay que sustituir ha sido modificada el 70 por ciento de las veces. ¿Cuál es la tasa máxima aceptable de fallos de página para obtener un tiempo efectivo de acceso de no más de 200 nanosegundos?

- 9.6 Suponga que está monitorizando la tasa con la que se mueve la manecilla en el algoritmo del reloj (que indica la página candidata para sustitución). ¿Qué es lo que puede deducir acerca del sistema si observa el siguiente comportamiento?

- La manecilla se mueve rápido
- La manecilla se mueve lentamente.

- 9.7 Indique algunas situaciones en las que el algoritmo de sustitución de las páginas menos frecuentemente utilizadas genere menos fallos de página que el algoritmo de sustitución de las páginas más recientemente utilizadas. Indique también en qué circunstancias se da la relación opuesta.

- 9.8 Indique algunas situaciones en las que la el algoritmo de sustitución de las páginas más frecuentemente utilizadas genere menos fallos de página que el algoritmo de sustitución de las páginas menos recientemente utilizadas. Indique también en qué circunstancias se da la relación inversa.

- 9.9 El sistema VAX/VMS utiliza un algoritmo de sustitución FIFO para las páginas residentes y un conjunto compartido de marcos libres compuesto por páginas recientemente utilizadas. Suponga que el conjunto compartido de marcos libres se gestiona utilizando la política de sustitución menos recientemente utilizadas. Responda las siguientes cuestiones:

- Si se produce un fallo de página y la página no se encuentra en el conjunto compartido de marcos libres, ¿cómo puede generarse espacio libre para la nueva página solicitada?
- Si se produce un fallo de página y la página se encuentra en el conjunto compartido de marcos libres, ¿cómo se activa la página residente y cómo se gestiona el conjunto compartido de marcos libres para hacer sitio para la página solicitada?
- ¿Hacia qué degenera el sistema si el número de páginas residentes se configura con el valor uno?
- ¿Hacia qué degenera el sistema si el número de páginas del conjunto compartido de marcos libres es cero?

- 9.10 Considere un sistema de paginación bajo demanda con las siguientes tasas medidas de utilización:

Uso de la CPU	20%
Paginación de disco	97,7%
Otros dispositivos de E/S	5%

Para cada una de las siguientes afirmaciones, indique si permitirá (o si es probable que lo haga) mejorar la tasa de utilización de la CPU. Razone su respuesta.

- Instalar una CPU más rápida.
- Instalar un disco de paginación de mayor tamaño.

- c. Incrementar el grado de multiprogramación.
 - d. Reducir el grado de multiprogramación.
 - e. Instalar más memoria principal.
 - f. Instalar un disco duro más rápido o múltiples controladoras con múltiples discos duros.
 - g. Añadir un mecanismo de prepaginación a los algoritmos de carga de páginas.
 - h. Incrementar el tamaño de página.
- 9.11** Suponga que una máquina proporciona instrucciones que pueden acceder a ubicaciones de memoria utilizando el esquema de direccionamiento indirecto de un nivel. ¿Cuál es la secuencia de fallos de página en que se incurre cuando todas las páginas de un programa son actualmente no residentes y la primera instrucción del programa es una operación de carga en memoria indirecta. ¿Qué sucede cuando el sistema operativo está utilizando una técnica de asignación de marcos por proceso y sólo hay dos páginas asignadas a este proceso?
- 9.12** Suponga que la política de sustitución (en un sistema paginado) consiste en examinar cada página regularmente y descartar dicha página si no ha sido utilizada desde el último examen. ¿Qué ventajas y qué inconvenientes tendríamos si utilizáramos esta política en lugar de la sustitución LRU o el algoritmo de segunda oportunidad?
- 9.13** Un algoritmo de sustitución de páginas debe minimizar el número de fallos de página. Podemos conseguir esta minimización distribuyendo equitativamente las páginas más utilizadas por toda la memoria, en lugar de hacerlas competir por un pequeño número de marcos de página. Podemos asociar con cada marco de página un contador del número de páginas asociadas con dicho marco. Entonces, para sustituir una página, podemos buscar el marco de página que tenga el valor de contador más pequeño.
- a. Defina un algoritmo de sustitución de páginas utilizando esta idea básica. Específicamente, tenga en cuenta los siguientes problemas:
 - 1. ¿Cuál es el valor inicial de los contadores?
 - 2. ¿Cuándo se incrementan los contadores?
 - 3. ¿Cuándo se decrementan los contadores?
 - 4. ¿Cómo se selecciona la página que hay que sustituir?
 - b. ¿Cuántos fallos de página se producen en su algoritmo para la siguiente cadena de referencia, si se utilizan cuatro marcos de página?
 - 1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.
 - c. ¿Cuál es el número mínimo de fallos de página para una estrategia óptima de sustitución de páginas, para la cadena de referencia de la parte b con cuatro marcos de página?
- 9.14** Considere un sistema de paginación bajo demanda con un disco de paginación que tiene un tiempo medio de acceso y de transferencia de 20 milisegundos. Las direcciones se traducen mediante una tabla de páginas que se conservan en memoria principal, con un tiempo de acceso de un microsegundo por cada acceso a memoria. Por tanto, cada referencia a memoria a través de la tabla de páginas requiere dos accesos. Para mejorar este tiempo, hemos añadido una memoria asociativa que reduce el tiempo de acceso a una sola referencia a memoria si la entrada de la tabla de páginas se encuentra en la memoria asociativa.
- Suponga que el 80 por ciento de los accesos están en la memoria asociativa y que, de los restantes, el 10 por ciento (es decir, el 2 por ciento del total) provocan fallos de página. ¿Cuál es el tiempo efectivo de acceso a memoria?

- 9.15 ¿Qué es lo que provoca la sobrepaginación? ¿Cómo detecta el sistema la sobrepaginación? Una vez que detecta la sobrepaginación, ¿qué puede hacer el sistema para eliminar este problema?
- 9.16 ¿Es posible que un proceso tenga dos conjuntos de trabajo, uno que represente los datos y otro que represente el código? Razona su respuesta.
- 9.17 Considere el parámetro Δ utilizado para definir la ventana del conjunto de trabajo en el modelo del conjunto de trabajo. ¿Cuál es el efecto que tiene asignar a Δ un valor pequeño sobre la frecuencia de fallos de página y el número de procesos activos (no suspendidos) que se ejecuten actualmente en el sistema? ¿Cuál es el efecto si se asigna a Δ un valor muy alto?
- 9.18 Suponga que existe un segmento inicial de 1024 KB a partir del que se asigna la memoria utilizando el esquema de distribución binaria. Utilizando la Figura 9.27 como guía, dibuje el árbol que ilustra cómo se satisfarían las siguientes solicitudes de memoria:
- solicitud de 240 bytes
 - solicitud de 120 bytes
 - solicitud de 60 bytes
 - solicitud de 130 bytes

A continuación, modifique el árbol para reflejar las siguientes operaciones de liberación de memoria. Realice una consolidación siempre que sea posible:

- liberación de 240 bytes
- liberación de 60 bytes
- liberación de 120 bytes

- 9.19 El algoritmo de asignación de franjas utiliza una caché separada para cada tipo de objeto. Suponiendo que haya una caché por cada tipo de objeto, explique por qué este sistema no se escala muy bien cuando se utilizan múltiples procesadores. ¿Qué podría hacerse para resolver este problema de escalabilidad?
- 9.20 Considere un sistema que asigne páginas de diferentes tamaños a los procesos. ¿Cuáles son las ventajas de este tipo de esquema de paginación? ¿Qué modificaciones del sistema de memoria virtual permitirían proporcionar esta funcionalidad?
- 9.21 Escriba un programa que implemente los algoritmos de sustitución de páginas FIFO y LRU presentados en este capítulo. Primero, genere una cadena de referencia de páginas aleatoria en la que los números de página estén comprendidos entre 0 y 9. Aplique la cadena aleatoria de referencias de página a cada algoritmo y registre el número de fallos de página provocados por cada algoritmo. Implemente los algoritmos de sustitución de modo que el número de marcos de página pueda variarse entre 1 y 7. Suponga que se utiliza un sistema de paginación bajo demanda.
- 9.22 Los números de *Catalan* son una secuencia entera C_n que aparece en los problemas de enumeración de árboles. Los primeros números de Catalan para $n = 1, 2, 3, \dots$ son 1, 2, 5, 14, 42, 132, ... Una fórmula para generar C_n es

$$C_n = \frac{1}{(n+1)} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

Diseñe dos programas que se comuniquen mediante memoria compartida utilizando la API Win32, como se esboza en la Sección 9.7.2. El proceso productor generará la secuencia de Catalan y la escribirá en un objeto de memoria compartida. El proceso consumidor leerá entonces la secuencia desde la memoria compartida y la proporcionará como salida.

En este caso, al proceso productor se le pasará un parámetro entero a través de la línea de comandos que especificará la cantidad de números de Catalan que hay que producir; decir, si se introduce 5 en la línea de comandos querrá decir que el proceso productor debe generar los primeros 5 números de Catalan.

Notas bibliográficas

La paginación bajo demanda se utilizó por primera vez en el sistema Atlas, implementado en la computadora MUSE de la Universidad de Manchester hacia 1960 (Kilburn et al. [1961]). Otro sistema pionero de paginación bajo demanda era MULTICS, implementado en el sistema GE 645 (Organick [1972]).

Belady et al. [1969] fueron los primeros investigadores en observar que la estrategia de sustitución FIFO podía producir la anomalía que lleva el nombre de Belady. Mattson et al. [1970] demostraron que los algoritmos de pila no están sujetos a la anomalía de Belady.

El algoritmo óptimo de sustitución fue presentado por Belady [1966]. La demostración de que era óptimo fue proporcionado por Mattson et al. [1970]. El algoritmo óptimo de Belady es para una asignación fija; Prieve y Fabry [1976] presentaron un algoritmo óptimo para aquellas situaciones en las que la asignación puede variar.

El algoritmo mejorado del reloj fue actualizado por Carr y Hennessy [1981]. El modelo de conjunto de trabajo fue desarrollado por Denning [1968]. En Denning [1980] se incluyen diversos análisis relativos al modelo del conjunto de trabajo.

El esquema para monitorización de la tasa de fallos de página fue desarrollado por Wulf [1969], que logró aplicar con éxito esta técnica al sistema informático Burroughs B5500.

Wilson et al. [1995] presentaron varios algoritmos para la asignación dinámica de memoria. Johnstone y Wilson [1998] describieron diversos problemas de fragmentación de memoria. Los asignadores de memoria basados en el mecanismo de distribución binaria fueron descritos en Knowlton [1965], Peterson y Norman [1977] y Purdom, Jr. y Stigler [1970]. Bonwick [1994] analizaba el asignador de franjas, y Bonwick y Adams [2001] ampliaron ese análisis para el caso de múltiples procesadores. Puede encontrar otros algoritmos de ajuste de memoria en Stephenson [1983], Bays [1977] y Brent [1989]. Para ver una panorámica de las estrategias de asignación de memoria, consulte Wilson et al. [1995].

Solomon y Russinovich [2000] describen el modo en que Windows 2000 implementa la memoria virtual. Mauro y McDougall [2001] analizan la memoria virtual en Solaris. Las técnicas de memoria virtual en Linux y BSD se describen en Bovet y Cesati [2002] y McKusick et al. [1996], respectivamente. Ganapathy y Schimmel [1998] y Navarro et al. [2002] analizan el soporte para tamaños múltiples de página en los sistemas operativos. Ortiz [2001] describió la memoria virtual utilizada en un sistema operativo embebido en tiempo real.

Jacob y Mudge [1998b] comparan las implementaciones de memoria virtual en las arquitecturas MIPS, PowerPC y Pentium. Un artículo relacionado (Jacob y Mudge [1998a]) describía el soporte hardware necesario para la implementación de la memoria virtual en seis arquitecturas diferentes, incluyendo UltraSPARC.

Parte Cuatro

Gestión de almacenamiento

Puesto que la memoria principal es, usualmente, demasiado pequeña para que quepan todos los datos y programas de manera permanente, el sistema informático debe proporcionar un almacenamiento secundario que sirva como respaldo de la memoria principal. La mayoría de los sistemas informáticos utilizan los discos como medio principal de almacenamiento en línea de la información (tanto programas como datos). El sistema de archivos proporciona los mecanismos para el almacenamiento en línea de los datos y programas que residen en los discos y para el acceso a esa información. Un archivo es una colección de información relacionada definida por su creador. El sistema operativo mapea los archivos sobre los dispositivos físicos y normalmente se organizan en directorios para facilitar su uso.

Los dispositivos que pueden conectarse a una computadora varían en diversos aspectos: algunos dispositivos transfieren un carácter o un bloque de caracteres cada vez; a algunos dispositivos sólo se puede acceder de forma secuencial, mientras que a otros se accede de manera aleatoria; algunos transfieren los datos de manera síncrona, mientras que otros son asíncronos; algunos dispositivos son dedicados, cuando otros son compartidos; los dispositivos pueden ser de sólo lectura o de lectura-escritura. También varían enormemente en cuanto a su velocidad y, desde diversos puntos de vista, son también el más lento de los componentes principales de la computadora.

Debido a esta amplia variedad de los dispositivos, el sistema operativo necesita proporcionar un amplio rango de funcionalidad a las aplicaciones, para permitir que éstas controlen todos los aspectos de los dispositivos. Uno de los objetivos principales del subsistema de E/S del sistema operativo es el de proporcionar la interfaz más simple posible al resto del sistema. Puesto que los dispositivos constituyen un cuello de botella en lo que a prestaciones se refiere, otro de los objetivos clave es optimizar las operaciones de E/S para conseguir un grado máximo de concurrencia.

Interfaz del sistema de archivos

Para la mayoría de los usuarios, el sistema de archivos es el aspecto más visible de un sistema operativo. Proporciona los mecanismos para el almacenamiento en línea de los datos y programas del propio sistema operativo y para todos los usuarios del sistema informático, así como para el acceso a esos datos y programas. El sistema de archivos está compuesto de dos partes diferenciadas: una colección de *archivos*, cada uno de los cuales almacena una serie de datos relacionados, y una *estructura de directorios*, que organiza todos los archivos del sistema y proporciona información acerca de los mismos. Los sistemas de archivos residen en dispositivos, los cuales analizaremos con detalle en los siguientes capítulos, aunque también los veremos brevemente en este. Dentro de este capítulo, vamos a considerar los diversos aspectos de los archivos y las principales estructuras de directorio. También analizaremos la semántica de compartición de archivos entre múltiples procesos, usuarios y computadoras. Finalmente, veremos qué formas existen para gestionar la *protección de archivos* necesaria cuando tenemos múltiples usuarios y queremos controlar quién puede acceder a los archivos y de qué manera puede accederse a los mismos.

OBJETIVOS DEL CAPÍTULO

- Explicar la función de los sistemas de archivos.
- Describir las interfaces de los sistemas de archivos.
- Analizar los compromisos de diseño de los sistemas de archivos, incluyendo los métodos de acceso, la compartición de archivos, el bloqueo de archivos y las estructuras de directorio.
- Explorar los mecanismos de protección de un sistema de archivos.

10.1 Concepto de archivo

Las computadoras pueden almacenar información en varios soportes de almacenamiento, como discos magnéticos, cintas magnéticas y discos ópticos. Para que el sistema informático sea cómodo de utilizar, el sistema operativo proporciona una vista lógica uniforme para el almacenamiento de la información. El sistema operativo realiza una abstracción de las propiedades físicas de los dispositivos de almacenamiento, con el fin de definir una unidad lógica de almacenamiento, el *archivo*. Los archivos son mapeados por el sistema operativo sobre los dispositivos físicos. Estos dispositivos de almacenamiento son, usualmente, no volátiles; de modo que los contenidos persisten aunque se produzcan fallos de alimentación o reinicios del sistema.

Un archivo es una colección de información relacionada, con un nombre, que se graba en almacenamiento secundario. Desde la perspectiva del usuario, un archivo es la unidad más pequeña de almacenamiento secundario lógico; en otras palabras, no pueden escribirse datos en el almacenamiento secundario a menos que estos se encuentren dentro de un archivo. Comúnmente, los archivos representan programas (tanto en versión fuente como en versión objeto) y datos. Los

archivos de datos pueden ser numéricos, alfabéticos, alfanuméricos o binarios: puede haber archivos de formato libre, como por ejemplo archivos de texto, o los archivos pueden estar formateados de manera rígida. En general, un archivo es una secuencia de bits, bytes, líneas o registros, cuyo significado está definido por el creador y el usuario del archivo. Por tanto, el concepto de archivo es extremadamente general.

La información contenida en un archivo es definida por su creador. En un archivo pueden almacenarse muchos tipos distintos de información: programas puente, programas objeto, programas ejecutables, datos numéricos, texto, registros de nómina, imágenes gráficas, grabaciones sonoras, etc. Un archivo tiene una determinada **estructura** definida que dependerá de su tipo. Un archivo de *texto* es una secuencia de caracteres organizada en líneas (y posiblemente en páginas). Un archivo *fuente* es una secuencia de subrutinas y funciones, cada una de las cuales está a su vez organizada como una serie de declaraciones, seguidas de instrucciones ejecutables. Un archivo *objeto* es una secuencia de bytes organizada en bloques que el programa montador del sistema puede comprender. Un archivo *ejecutable* es una serie de secciones de código que el cargador puede cargar en memoria y ejecutar.

10.1.1 Atributos de archivo

Los archivos tienen un nombre, por comodidad de sus usuarios humanos, y para referirnos a él utilizamos ese nombre. Un nombre de archivo es, usualmente, una cadena de caracteres, como por ejemplo *example.c*. Algunos sistemas diferencian entre letras mayúsculas y minúsculas dentro de los nombres, mientras que otros sistemas no hacen esa distinción. Cuando se asigna un nombre a un archivo, éste pasa a ser independiente del proceso, del usuario e incluso del sistema que lo creó. Por ejemplo, un usuario puede crear el archivo *example.c* y otro usuario puede editar dicho archivo especificando su nombre. El propietario del archivo puede escribir ese archivo en un disquete, enviarlo como parte de un correo electrónico o copiarlo a través de una red, y dicho archivo se podría seguir llamando *example.c* en el sistema de destino.

Los atributos de un archivo varían de un sistema operativo a otro, pero típicamente son los siguientes:

- **Nombre.** El nombre de archivo simbólico es la única información que se mantiene en un formato legible por parte de las personas.
- **Identificador.** Esta etiqueta única, que usualmente es un número, identifica el archivo dentro del sistema de archivos; se trata de la versión no legible por las personas del nombre del archivo.
- **Tipo.** Esta información es necesaria para los sistemas que soporten diferentes tipos de archivos.
- **Ubicación.** Esta información es un puntero a un dispositivo y a la ubicación del archivo dentro de dicho dispositivo.
- **Tamaño.** Este atributo expresa el tamaño actual del archivo (en bytes, palabras o bloques) y, posiblemente, el tamaño máximo permitido.
- **Protección.** Información de control de acceso que determina quién puede leer el archivo, escribir en el archivo, ejecutarlo, etc.
- **Fecha, hora e identificación del usuario.** Esta información puede mantenerse para los sucesos de creación, de última modificación y de último uso del archivo. Estos datos pueden resultar útiles para propósitos de protección, seguridad y monitorización del uso del archivo.

La información acerca de los archivos se almacena en la estructura de directorios, que también reside en el almacenamiento secundario. Típicamente, una entrada de directorio está compuesta del nombre de un archivo y de su identificador único. El identificador, a su vez, permite localizar los demás atributos del archivo. Puede que haga falta más de un kilobyte para almacenar esta información para cada archivo. En los sistemas donde haya múltiples archivos, el archivo del pro-

pio directorio puede llegar a ser del orden de los megabytes. Puesto que los directorios, al igual que los archivos, deben ser no volátiles, hay que almacenarlos en el dispositivo y cargarlos en memoria por partes, según sea necesario.

10.1.2 Operaciones con los archivos

Un archivo es un **tipo abstracto de datos**. Para definir un archivo apropiadamente, debemos considerar las operaciones que pueden realizarse con los archivos. El sistema operativo puede proporcionar llamadas al sistema para crear, escribir, leer, reposicionar, borrar y truncar archivos. Veamos qué es lo que el sistema operativo tiene que hacer para llevar a cabo cada una de estas seis operaciones básicas con los archivos; después, nos resultará sencillo ver cómo pueden implementarse otras operaciones similares, como por ejemplo la de renombrar un archivo.

- **Creación de un archivo.** Para crear un archivo hace falta ejecutar dos pasos. En primer lugar, es necesario encontrar espacio para el archivo dentro del sistema de archivos; veremos la cuestión de la asignación de espacio a los archivos en el Capítulo 11. En segundo lugar, es necesario incluir en el directorio una entrada para el nuevo archivo.
- **Escritura en un archivo.** Para escribir en un archivo, debemos realizar una llamada al sistema que especifique tanto el nombre del archivo como la información que hay que escribir en el archivo. Dado el nombre del archivo, el sistema explora el directorio para hallar la ubicación del archivo. El sistema debe mantener un puntero de *escritura* que haga referencia a la ubicación dentro del archivo en la que debe tener lugar la siguiente escritura. El puntero de escritura debe actualizarse cada vez que se escriba en el archivo nueva información.
- **Lectura de un archivo.** Para leer desde un archivo, utilizamos una llamada al sistema que especifica el nombre del archivo y dónde debe colocarse (dentro de la memoria) el siguiente bloque del archivo. De nuevo, exploramos el directorio para hallar la entrada asociada y el sistema necesitará mantener un puntero de *lectura* que haga referencia a la ubicación dentro del archivo en la que tiene que tener lugar la siguiente lectura. Una vez que la lectura se ha completado, se actualiza el puntero de lectura. Puesto que los procesos, usualmente, están o bien leyendo del archivo o bien escribiendo en él, se puede almacenar la ubicación correspondiente a la operación actual en un **puntero de posición actual del archivo** que será diferente para cada proceso. Las operaciones de lectura y escritura utilizan el mismo puntero, ahorrando así espacio y reduciendo la complejidad del sistema.
- **Reposicionamiento dentro de un archivo.** Se explora el directorio para hallar la correspondiente entrada y se reposiciona el puntero de posición actual dentro de un archivo, asignándole un nuevo valor. El reposicionamiento dentro del archivo no tiene por qué implicar que se realice ninguna operación de E/S. Esta operación de archivo se conoce también con el nombre de *búsqueda* en el archivo.
- **Borrado de un archivo.** Para borrar un archivo, exploramos el directorio en busca del archivo indicado. Habiendo hallado la entrada de directorio asociada, liberamos todo el espacio del archivo, de modo que pueda ser reutilizado por otros archivos, y borramos también la propia entrada del directorio.
- **Truncado de un archivo.** El usuario puede querer borrar el contenido de un archivo, pero manteniendo sus atributos. En lugar de forzar al usuario a borrar el archivo y volverlo a crear, esta función permite que los atributos no se vean modificados (excepto la longitud del archivo), mientras que el archivo se reinicializa asignándole una longitud igual a cero y liberando el espacio que tuviera asignado.

Estas seis operaciones básicas forman el conjunto mínimo de operaciones de archivo requeridas. Otras operaciones comunes son la *adición* de nueva información al final de un archivo existente y el *renombrado* de un archivo existente. Estas operaciones primitivas pueden a su vez combinarse para realizar otras operaciones de archivo. Por ejemplo, podemos crear una *copia* de un archivo, o copiar el archivo en otro dispositivo de E/S, como por ejemplo una impresora o una

pantalla, creando un nuevo archivo y luego leyendo la información del archivo antiguo y ~~escribiéndola en el nuevo~~. También conviene disponer de operaciones que permitan al usuario ~~comparar~~ y modificar los diversos atributos de un archivo. Por ejemplo, podemos disponer operaciones que permitan al usuario determinar el estado de un archivo, como por ejemplo longitud, y configurar atributos de archivo, como por ejemplo el propietario del archivo.

La mayoría de las operaciones de archivo mencionadas implican realizar una búsqueda en directorio para encontrar la entrada asociada con el archivo cuyo nombre se ha especificado. Para evitar estas constantes búsquedas, muchos sistemas requieren que se realice una llamada al sistema `open()` antes de utilizar por primera vez activamente un determinado archivo. El sistema operativo mantiene una pequeña tabla, denominada **tabla de archivos abiertos**, que contiene información acerca de todos los archivos abiertos. Cuando se solicita que se realice una operación con un archivo, el archivo se especifica mediante un índice a esta tabla, con lo que no hace falta realizar ninguna exploración del directorio. Cuando el archivo deja de ser utilizado activamente, será *cerrado* por el proceso y el sistema operativo eliminará la entrada correspondiente de la tabla de archivos abiertos. `create` y `delete` son llamadas al sistema que funcionan con archivos cerrados, en lugar de con archivos abiertos.

Algunos sistemas abren implícitamente los archivos cuando se realiza la primera referencia a los mismos. El archivo se cerrará automáticamente cuando termine el trabajo o programa que lo abrió. Sin embargo, la mayoría de los sistemas requieren que el programador abra el archivo explícitamente con la llamada al sistema `open()`, antes de poder utilizar dicho archivo. La operación `open()` toma un nombre de archivo y explora el directorio, copiando la entrada de directorio correspondiente en la tabla de archivos abiertos. La llamada `open()` pueden también aceptar información acerca del modo de acceso: creación, sólo lectura, lectura-escritura, sólo adición, etc. Este modo se compara con los permisos definidos para el archivo; si el modo solicitado está permitido, se abre el archivo para que lo utilice el proceso. La llamada al sistema `open()` devuelve normalmente un puntero que hace referencia a la entrada correspondiente dentro de la tabla de archivos abiertos. Es este puntero, y no el nombre real del archivo, lo que se utiliza en todas las operaciones de E/S, evitando sucesivas búsquedas y simplificando la interfaz de llamadas al sistema.

La implementación de las operaciones `open()` y `close()` es más complicada en los entornos donde varios procesos puedan abrir un mismo archivo simultáneamente. Esto puede suceder, por ejemplo, en los sistemas donde varias aplicaciones distintas abran a la vez el mismo archivo. Normalmente, el sistema operativo usará dos niveles de tablas internas: una tabla por cada proceso y una tabla global del sistema. La tabla de cada proceso indica todos los archivos que ese proceso ha abierto. En esta tabla se almacena información relativa al uso de cada archivo por parte del proceso. Por ejemplo, es en esa tabla donde se almacena el puntero de archivo actual para cada uno de los archivos; también pueden incluirse en ella otros datos, como los derechos de acceso al archivo y la información de contabilización.

Cada entrada de la tabla de archivos correspondiente a un proceso apunta, a su vez, a una tabla de archivos abiertos que tiene un carácter global para todo el sistema. Esta tabla global del sistema contiene información independiente de los procesos, como por ejemplo la ubicación del archivo dentro del disco, las fechas de acceso y el tamaño del archivo. Una vez que un proceso ha abierto un archivo, se incluye una entrada para dicho archivo en la tabla global del sistema. Cuando otro proceso ejecute una llamada `open()`, se añadirá simplemente una entrada a la tabla de archivos abiertos de ese proceso, apuntando dicha entrada a la entrada correspondiente dentro de la tabla global del sistema. Típicamente, la tabla de archivos abiertos almacena también un *contador de aperturas* asociado con cada archivo, para indicar cuántos procesos han abierto el archivo. Cada llamada a `close()` reduce este *contador de aperturas* y cuando el *contador de aperturas* alcanza el valor cero, querrá decir que el archivo ha dejado de estar en uso y se eliminará de la tabla de archivos abiertos la tabla correspondiente al archivo.

En resumen, con cada archivo abierto se asocian diferentes tipos de datos:

- **Puntero de archivo.** En aquellos sistemas que no incluyen un desplazamiento de archivo como parte de las llamadas a sistema `read()` y `write()`, el sistema deberá registrar la ubicación correspondiente a la última lectura-escritura, utilizando para ello un puntero de

posición actual dentro del archivo. Este puntero es distinto para cada proceso que esté operando con el archivo, y deberá por tanto almacenarse de forma separada de los atributos del archivo almacenados en disco.

- **Contador de aperturas del archivo.** A medida que se cierran archivos, el sistema operativo debe reutilizar las correspondientes entradas de la tabla de archivos abiertos, ya que en caso contrario se quedaría sin espacio en esa tabla. Puesto que un mismo archivo puede haber sido abierto por múltiples procesos, el sistema deberá esperar a que el último de esos procesos cierre el archivo, antes de eliminar la correspondiente entrada de la tabla de archivos abiertos. El contador de aperturas del archivo controla el número de aperturas y cierres y alcanzará el valor cero cuando el último proceso cierre el archivo. Entonces, el sistema podrá eliminar la correspondiente entrada.
- **Ubicación del archivo dentro del disco.** La mayoría de las operaciones de archivo requieren que el sistema modifique datos dentro del archivo. La información necesaria para ubicar el archivo en el disco se almacena en la memoria, para que el sistema no tenga que leer de nuevo esa información desde el disco en cada operación.
- **Derechos de acceso.** Cada proceso abre un determinado archivo en un cierto modo de acceso. Esta información se almacena en la tabla correspondiente a cada proceso, para que el sistema operativo pueda autorizar o denegar las siguientes solicitudes de E/S.

Algunos sistemas operativos proporcionan funciones para bloquear un archivo abierto (o determinadas secciones de un archivo). Los bloqueos de archivo permiten que un proceso bloquee un archivo e impida que otros procesos puedan acceder al mismo. Los bloqueos de archivo resultan útiles para aquellos archivos que son compartidos por varios procesos, como por ejemplo un archivo de registro del sistema que pueda ser consultado y modificado por varios procesos distintos del sistema.

Los bloqueos de archivo proporcionan una funcionalidad similar a los bloqueos lector-escritor, de los que hemos hablado en la Sección 6.6.2. Un **bloqueo compartido** es similar a un bloqueo lector, en el sentido de que varios procesos pueden adquirir dichos bloqueos concurrentemente. Un **bloqueo exclusivo** se comporta como un bloqueo escritor: sólo puede adquirir dicho tipo de bloqueo un proceso cada vez. Es importante observar que no todos los sistemas operativos proporcionan ambos tipos de bloqueo; algunos sistemas sólo proporcionan bloqueo de archivos exclusivo.

Además, los sistemas operativos pueden proporcionar mecanismos de bloqueo de archivos **obligatorios** o **sugeridos**. Si un bloqueo es obligatorio, después de que un proceso adquiera un

BLOQUEO DE ARCHIVOS EN JAVA

En Java, para adquirir un bloqueo, requiere que primero se obtenga el objeto `FileChannel` correspondiente al archivo que hay que bloquear. Para adquirir el bloqueo se utiliza el método `lock()` de `FileChannel`. La API del método `lock()` es:

```
public Future<Lock> lock(long pos, long end, boolean shared)
```

donde `pos`, `n` y `end` son las posiciones inicial y final de la región que se quiere bloquear. Si se asigna a `shared` el valor `true`, se tratará de un bloqueo compartido; por el contrario, si se asigna a `shared` el valor `false`, estaremos adquiriendo el bloqueo en modo exclusivo. El bloqueo se libera invocando el método `release()` del objeto `FutureLock` devuelto por la operación `lock()`.

El programa de la Figura 10.1 ilustra el mecanismo del bloqueo de archivos en Java. Este programa adquiere dos bloqueos sobre el archivo `file.txt`. La primera mitad del archivo se adquiere con un bloqueo exclusivo, mientras que el bloqueo para la segunda mitad es un bloqueo compartido.

Continúa

BLOQUEO DE ARCHIVOS EN JAVA (Cont.)

```

import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // obtener el canal para el archivo
            FileChannel ch = raf.getChannel();

            // esto bloquea la primera mitad del archivo - exclusivo
            exclusiveLock = ch.lock(0, raf.length(), EXCLUSIVE);

            /** Ahora modificar los datos */

            // liberar el bloqueo
            exclusiveLock.release();

            // esto bloquea a la segunda mitad del archivo - compartido
            sharedLock = ch.lock(raf.length()/2, raf.length(), SHARED);

            /** Ahora leer los datos . . . */

            // liberar el bloqueo
            exclusiveLock.release();
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release();
            if (sharedLock != null)
                sharedLock.release();
        }
    }
}

```

Figura 10.1 Ejemplo de bloqueo de archivos en Java.

bloqueo exclusivo, el sistema operativo impedirá a todos los demás procesos que accedan al archivo bloqueado. Por ejemplo, suponga que un proceso adquiere un bloqueo exclusivo sobre el archivo `system.log`. Si tratamos de abrir `system.log` desde otro proceso (por ejemplo, un editor de texto), el sistema operativo impedirá dicho acceso hasta que se libere el bloqueo exclusivo. Esta

prohibición tendrá lugar incluso si el editor de textos se ha escrito de forma que no adquiera explícitamente el bloqueo. Alternativamente, si el bloqueo es del tipo sugerido, el sistema operativo no impedirá que el editor de textos adquiera un acceso a `system.log`. En este caso, será necesario escribir el editor de textos de modo que adquiera manualmente el bloqueo antes de acceder al archivo. En otras palabras, si el esquema de bloqueo es obligatorio, el sistema operativo garantiza la integridad de los bloqueos, mientras que para los bloqueos sugeridos, es responsabilidad de los desarrolladores software garantizar que se adquieran y liberen apropiadamente los distintos bloqueos. Como regla general, los sistemas operativos Windows adoptan un mecanismo de bloqueo obligatorio, mientras que los sistemas UNIX emplean bloqueos sugeridos.

El uso de los bloqueos de archivo requiere que se adopten las mismas precauciones que para la sincronización normal de procesos. Por ejemplo, los programadores que estén desarrollando software para sistemas con bloqueo obligatorio deben tener cuidado en mantener los bloqueos de archivo exclusivos únicamente mientras estén accediendo al archivo; en caso contrario, impedirían que otros procesos accedieran también al archivo. Además, debe tomarse alguna medida para garantizar que dos o más procesos no se interbloqueen al tratar de adquirir bloqueos sobre archivos.

10.1.3 Tipos de archivo

Cuando diseñamos un sistema de archivos (de hecho, cuando diseñamos todo un sistema operativo) siempre debemos considerar si el sistema operativo debe reconocer y soportar el concepto de tipo de archivo. Si un sistema operativo reconoce el tipo de un archivo, podrá operar con ese archivo de formas razonables. Por ejemplo, uno de los errores más comunes es que un usuario trate de imprimir la versión objeto o versión binaria de un programa. Esta operación produce normalmente información sin sentido; sin embargo, sí que se podría imprimir correctamente el contenido de ese archivo si el sistema operativo sabe que el archivo está en formato objeto o formato binario.

Una técnica común para implementar los tipos de archivo consiste en incluir el tipo como parte del nombre de archivo. El nombre se divide en dos partes: un nombre y una *extensión*, usualmente separadas por un carácter de punto (Figura 10.2). De esta forma, el usuario y el sistema operativo pueden determinar, con sólo analizar el nombre, cuál es el tipo de cada archivo. Por ejemplo, la mayoría de los sistemas operativos permiten a los usuarios especificar los nombres de archivo como una secuencia de caracteres seguida de un punto y terminada por una extensión formada por caracteres adicionales. Por ejemplo de nombres de archivo podríamos citar `resume.doc`, `Server.java` y `ReaderThread.c`. El sistema utiliza la extensión para indicar el tipo del archivo y el tipo de operaciones que pueden realizarse con dicho archivo. Por ejemplo, sólo los archivos con extensión `.com`, `.exe` o `.bat` pueden ejecutarse. Los archivos `.com` y `.exe` son dos tipos de archivos ejecutables binarios, mientras que un archivo `.bat` es un **archivo de procesamiento por lotes** que contiene, en formato ASCII, una serie de comandos dirigidos al sistema operativo. MS-DOS sólo reconoce unas pocas extensiones, pero los programas de aplicación también utilizan las extensiones para indicar los tipos de archivo en los que están interesadas. Por ejemplo, los ensambladores esperan que los archivos fuente tengan una extensión `.asm`, y el procesador de textos Microsoft Word espera que sus archivos terminen con una extensión `.doc`. Estas extensiones no son obligatorias, por lo que un usuario podría especificar un archivo sin la extensión (para ahorrarse unas cuantas pulsaciones de tecla) y la aplicación buscará un archivo que tenga el nombre especificado y la extensión necesaria. Puesto que estas extensiones no están soportadas por el sistema operativo, pueden considerarse como “sugerencias” dirigidas a las aplicaciones que operan con las mismas.

Otro ejemplo de la utilidad de los tipos de archivo podemos extraerlo del sistema operativo TOPS-20. Si el usuario trata de ejecutar un programa objeto cuyo archivo fuente ha sido modificado (o editado) desde que se produjo el archivo objeto, el archivo fuente se recompilará automáticamente. Esta función garantiza que el usuario siempre ejecute un archivo objeto actualizado. En caso contrario, el usuario podría perder una cantidad de tiempo considerable ejecutando el antiguo archivo objeto. Para que esta función sea posible, el sistema operativo debe poder diferenciar el archivo fuente del archivo objeto, con el fin de verificar la fecha en que cada archivo fue creado

tipo de archivo	extensión usual	función
ejecutable	exe, com, pif, bat, sh, ...	programas que se ejecutan
script	sh, bat, vbs, ...	programas que se ejecutan
de datos	txt, doc, pdf, ps, ...	información de texto
de imágenes	jpg, gif, bmp, ...	información visual
de video	mpg, avi, mpeg, ...	información de video
de audio	wav, mp3, midi, ...	información auditiva
de procesamiento de lotes	bat, sh, vbs, ...	información de ejecución
de aplicaciones	app, ...	información de ejecución
de texto	txt, doc, pdf, ps, ...	información de texto
de imágenes	jpg, gif, bmp, ...	información visual
de video	mpg, avi, mpeg, ...	información de video
de audio	wav, mp3, midi, ...	información auditiva
de procesamiento de lotes	bat, sh, vbs, ...	información de ejecución
de aplicaciones	app, ...	información de ejecución
de texto	txt, doc, pdf, ps, ...	información de texto
de imágenes	jpg, gif, bmp, ...	información visual
de video	mpg, avi, mpeg, ...	información de video
de audio	wav, mp3, midi, ...	información auditiva
de procesamiento de lotes	bat, sh, vbs, ...	información de ejecución
de aplicaciones	app, ...	información de ejecución

Figura 10.2 Tipos comunes de archivo.

o modificado por última vez, así como determinar el lenguaje del programa puente (para utilizar el compilador correcto).

Considere, también, el sistema operativo Mac OS X. En este sistema, cada archivo tiene un tipo, como por ejemplo *TEXT* (para los archivos de texto) o *APPL* (para las aplicaciones). Cada archivo tiene también un atributo de creador, que contiene el nombre del programa que lo creó. Este atributo es configurado por el sistema operativo durante la llamada *create()*, por lo que su uso es impuesto y soportado por el propio sistema. Por ejemplo, un archivo producido por un procesador de textos tendrá como creador el nombre del procesador de textos. Cuando el usuario abra dicho archivo, haciendo doble clic con el ratón sobre el ícono que lo representa, el procesador de textos se invoca automáticamente y el archivo se carga en memoria, listo para ser editado.

El sistema UNIX utiliza un simple **número mágico** almacenado al principio de algunos archivos para indicar, aproximadamente, el tipo del archivo: programa ejecutable, archivo de procesamiento por lotes (o *script de la shell*), archivo PostScript, etc. No todos los archivos tienen números mágicos, por lo que la funcionalidad del sistema no puede basarse exclusivamente en esta información. UNIX tampoco almacena el nombre del programa que creó el archivo. En UNIX sí que se permiten las sugerencias en forma de extensión del nombre del archivo, pero estas extensiones ni son obligatorias ni el sistema operativo depende de ellas; principalmente, su objetivo es ayudar a los usuarios a determinar el tipo de contenido del archivo. Esas extensiones pueden ser utilizadas o ignoradas por cada aplicación concreta, correspondiendo dicha decisión al programador de la aplicación.

10.1.4 Estructura de los archivos

Los tipos de archivo también pueden usarse para indicar la estructura interna del archivo. Como hemos mencionado en la Sección 10.1.3, los archivos fuente y objeto tienen estructuras que se corresponden con las expectativas de los programas que se van a encargar de leerlos. Además, ciertos archivos deben adaptarse a una estructura requerida, comprensible por parte del sistema operativo. Por ejemplo, el sistema operativo requiere que los archivos ejecutables tengan una

estructura concreta, para poder determinar en qué parte de la memoria cargar el archivo y dónde está ubicada la primera instrucción. Algunos sistemas operativos amplían esta idea y utilizan un conjunto de estructuras de archivos soportadas por el sistema, con una serie de operaciones especiales para manipular los archivos que tengan dichas estructuras. Por ejemplo, el sistema operativo VMS de DEC tiene un sistema de archivos que soporta tres estructuras de archivo definidas.

Este punto nos lleva a una de las desventajas de que el sistema operativo soporte múltiples estructuras de archivo: el tamaño resultante del sistema operativo es excesivo. Si el sistema operativo define cinco tipos distintos de estructuras, necesitará contener el código para soportar esas estructuras de archivo. Además, todo archivo puede necesitar definirse como uno de los tipos de archivo soportados por el sistema operativo. Cuando una aplicación nueva requiera información estructurada en alguna forma que no esté soportada por el sistema operativo, pueden aparecer graves problemas.

Por ejemplo, suponga que un sistema soporta dos tipos de archivo: archivos de texto (compuestos de caracteres ASCII separados por un retorno de carro y un avance de línea) y archivos ejecutables binarios. En estas condiciones, si nosotros (como usuarios) queremos definir un archivo cifrado para evitar que el contenido pueda ser leído por personas no autorizadas, es posible que ninguno de los dos tipos de archivo nos resulte apropiado. El archivo cifrado no está compuesto por líneas de texto ASCII, sino que está formado por bits (aparentemente) aleatorios. Al mismo tiempo, aunque parezca ser un archivo binario, no es ejecutable. Como resultado, es posible que tengamos que ignorar o malutilizar el mecanismo de tipos de archivo del sistema operativo, o renunciar a implementar nuestro esquema de cifrado.

Algunos sistemas operativos imponen (y soportan) un número mínimo de estructuras de archivo. Este enfoque ha sido adoptado en UNIX, en MS-DOS y en otros sistemas. UNIX considera que cada archivo es una secuencia de bytes de 8 bits, sin que el sistema operativo realice ninguna interpretación de dichos bits. Este esquema proporciona una máxima flexibilidad, pero un escaso soporte: cada programa de aplicación debe incluir su propio código para interpretar los archivos de entrada de acuerdo con la estructura apropiada. Sin embargo, todos los sistemas operativos soportan al menos una estructura (la de los archivos ejecutables) para que el sistema pueda cargar y ejecutar programas.

El sistema operativo Macintosh también soporta un número mínimo de estructuras de archivo. Este sistema espera que los archivos contengan dos partes: un **subarchivo de recursos** y un **subarchivo de datos**. El subarchivo de recursos contiene información de interés para el usuario; por ejemplo, almacena las etiquetas de los botones mostrados por el programa. Un usuario extranjero podría re-etiquetar estos botones en su propio idioma y el sistema operativo Macintosh proporciona herramientas para permitir la modificación de la información contenida en el subarchivo de recursos. El subarchivo de datos contiene código de programa o datos, que son los contenidos tradicionales de los archivos. Para realizar la misma tarea en un sistema UNIX o MS-DOS, el programador necesitaría modificar y recomilar el código fuente, a menos que definiera su propio archivo de datos modificable con el usuario. Claramente, resulta útil que el sistema operativo soporte estructuras que vayan a ser utilizadas frecuentemente y que ahorren al programador un esfuerzo sustancial. La existencia de muy pocas estructuras hace que la tarea de programación resulte poco cómoda, mientras que si existen demasiadas el sistema operativo crece sin medida y los programadores pueden llegar a verse confundidos.

10.1.5 Estructura interna de los archivos

Internamente, localizar un determinado desplazamiento dentro de un archivo puede ser complicado para el sistema operativo. Los sistemas de disco suelen tener un tamaño de bloque bien definido, que está determinado por el tamaño de un sector. Todas las operaciones de E/S de disco se realizan en unidades de un bloque (registro físico) y todos los bloques tienen el mismo tamaño. Resulta poco probable que el tamaño del registro físico se corresponda exactamente con la longitud deseada de los registros lógicos. Los registros lógicos pueden incluso variar en longitud. Una solución común a este problema consiste en **empaquetar** varios registros lógicos dentro de los bloques físicos.

Por ejemplo, el sistema operativo UNIX define todos los archivos como simples flujos de bytes. Cada byte es direccionable de manera individual, a partir de su desplazamiento con respecto al principio (o al final) del archivo. En este caso, el tamaño de registro lógico es un byte. El sistema de archivos empaqueta y desempaquetará automáticamente los bytes en los bloques físicos del disco (por ejemplo, 512 bytes por bloque) según sea necesario.

El tamaño del registro lógico, el tamaño del bloque físico y la técnica de empaquetamiento determinan cuántos registros lógicos se almacenarán en cada bloque físico. El empaquetamiento puede ser realizado por el programa de aplicación del usuario o por el sistema operativo.

En cualquiera de los dos casos, podemos considerar el archivo como una secuencia de bloques. Todas las funciones de E/S básicas operan en términos de bloques. La conversión entre registros lógicos y bloques físicos es un problema software relativamente simple.

Puesto que el espacio de discos se asigna siempre en bloques, generalmente se desperdiciará una parte del último bloque de cada archivo. Si cada bloque tiene 512 bytes, por ejemplo, entonces un archivo de 1949 bytes tendría cuatro bloques (2048 bytes); los últimos 99 bytes se desperdiciarían. Ese desperdicio que se produce al tratar de mantener la información almacenada en unidades de bloques (en lugar de bytes) se conoce con el nombre de **fragmentación interna**. Todos los sistemas de archivos sufren el problema de la fragmentación interna; además, cuanto más grande sea el tamaño del bloque, mayor será esa fragmentación interna.

10.2 Métodos de acceso

Los archivos almacenan información. Cuando hace falta utilizarla, es necesario acceder a esta información y leerla en la memoria de la computadora. Puede accederse a la información contenida en el archivo en varias formas distintas. Algunos sistemas sólo proporcionan un método de acceso para los archivos, mientras que otros sistemas, como por ejemplo los de IBM, soportan muchos métodos de acceso y elegir el método adecuado para cada aplicación concreta constituye uno de los principales problemas de diseño.

10.2.1 Acceso secuencial

El método de acceso más simple es el **acceso secuencial**. La información del archivo se procesa por orden, un registro después de otro. Este modo de acceso es, como mucho, el más común; por ejemplo, los editores y compiladores suelen acceder a los archivos de esta forma.

Las lecturas y escrituras constituyen el grueso de las operaciones realizadas con un archivo. Una operación de lectura (*leer siguiente*) lee la siguiente porción del archivo e incrementa automáticamente un puntero de archivo, que controla la ubicación de E/S. De forma similar, la operación de escritura (*escribir siguiente*) añade información al final del archivo y hace que el puntero avance hasta el final de los datos recién escritos (el nuevo final del archivo). Dichos archivos podrán reinicializarse para situar el puntero al principio de los mismos y, en algunos sistemas, puede que un programa sea capaz de saltar hacia adelante o hacia atrás n registros, para algún cierto valor entero n , quizás sólo para $n = 1$. El acceso secuencial, que se ilustra en la Figura 10.3, está basado en un modelo de archivo que se corresponde con las cintas magnéticas y funciona tanto en los dispositivos de acceso secuencial como en los de acceso aleatorio.

10.2.2 Acceso directo

Otro método es el **acceso directo** o **acceso relativo**. Un archivo está compuesto de **registros lógicos** de longitud fija que permiten a los programas leer y escribir registros rápidamente, sin ningún orden concreto. El método de acceso directo se basa en un modelo de archivos que se corresponde con los dispositivos de disco, ya que los discos permiten el acceso aleatorio a cualquier bloque de un archivo. Para el acceso directo, el archivo se considera como una secuencia numerada de bloques o registros. Por tanto, podemos leer el bloque 14, luego leer el bloque 53 y luego escribir el bloque 7. No existe ninguna restricción en cuanto al orden de lectura o escritura en los archivos de acceso directo.

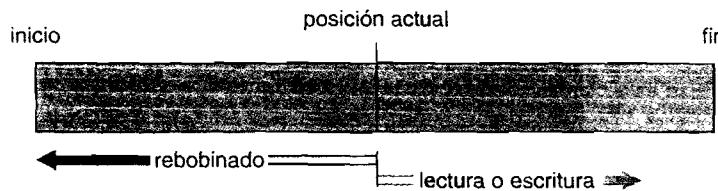


Figura 10.3 Archivo de acceso secuencial.

Los archivos de acceso directo tienen una gran utilidad para el acceso inmediato a grandes cantidades de información; las bases de datos suelen implementarse con archivos de este tipo. Cuando se recibe una consulta relativa a un tema concreto, se calcula qué bloque contiene la respuesta y luego se lee ese bloque directamente para proporcionar la información deseada.

Como ejemplo simple, en un sistema de reserva de billetes de avión, podríamos almacenar toda la información acerca de un vuelo concreto (por ejemplo, el vuelo 713) en el bloque identificado por el número de vuelo. Así, el número de asientos disponibles en el vuelo 713 estará almacenado en el bloque 713 del archivo de reservas. Para almacenar información acerca de un conjunto de mayor tamaño, como por ejemplo un conjunto de personas, podríamos calcular una función hash con los nombres de las personas o realizar una búsqueda en un pequeño índice almacenado en memoria para determinar el bloque que hay que leer o analizar.

En el método de acceso directo, las operaciones de archivo deben modificarse para incluir el número de bloque como parámetro. Así, tendremos operaciones tales como *leer n*, donde *n* es el número de bloque, en lugar de *leer siguiente*, y *escribir n*, en lugar de *escribir siguiente*. Una técnica alternativa consiste en retener las operaciones de *leer siguiente* y *escribir siguiente*, como con el acceso secuencial, y añadir una operación *posicionar archivo en n*, donde *n* minúscula es el número de bloque. Entonces, para realizar una operación *leer n*, ejecutaríamos primero *posicionar en n* y luego *leer siguiente*.

El número de bloque proporcionado por el usuario al sistema operativo es normalmente un **número de bloque relativo**. Un número de bloque relativo es un índice referido al comienzo del archivo. Así, el primer bloque relativo del archivo será el 0, el siguiente será el 1, etc., aun cuando la dirección de disco absoluta real del bloque pueda ser 14703 para el primer bloque y 3192 para el segundo. La utilización de números de bloque relativos permite al sistema operativo decidir dónde hay que colocar el archivo (lo que se denomina el *problema de asignación*, que se analiza en el Capítulo 11) y ayuda a impedir que el usuario acceda a porciones del sistema de archivos que no formen parte de su archivo. Algunos sistemas hacen que los números de bloque relativos comiencen en 0, mientras que otros comienzan en 1.

¿Cómo satisface el sistema una solicitud referida al registro *N* de un archivo? Suponiendo que tengamos una longitud de registro lógico igual a *N*, la solicitud relativa al registro *N* se transforma en una solicitud de E/S donde se piden *L* bytes comenzando en la ubicación *L * (N)* dentro del archivo (suponiendo que el primer registro sea *N = 0*). Puesto que los registros lógicos tienen un tamaño fijo, también resulta fácil leer, escribir o borrar un registro.

No todos los sistemas de archivos soportan tanto el acceso secuencial como el acceso directo a los archivos. Algunos sistemas sólo permiten el acceso secuencial a archivos, mientras que otros sólo soportan el acceso directo. Algunos sistemas requieren que los archivos se definan como secuenciales o directos en el momento de crearlos y a dichos archivos sólo se podrá acceder de una forma que sea coherente con su declaración. Podemos simular fácilmente el acceso secuencial en un archivo de acceso directo simplemente manteniendo una variable *cp* que defina nuestra posición actual, como se ilustra en la Figura 10.4. Sin embargo, simular un archivo de acceso directo sobre un archivo de acceso secuencial es extremadamente poco eficiente y engorroso.

10.2.3 Otros métodos de acceso

Pueden construirse otros métodos de acceso por encima del método de acceso directo. Estos métodos implican, generalmente, la construcción de un índice para el archivo. El **índice**, como los índi-

acceso secuencial	implementación para el acceso directo
reiniciar	$cp = 0;$
leer siguiente	leer $cp;$ $cp = cp + 1;$
escribir siguiente	escribir $cp;$ $cp = cp + 1;$

Figura 10.4 Simulación del acceso secuencial sobre un archivo de acceso directo.

ces de la parte posterior de un libro, contiene punteros a los distintos bloques. Para encontrar un registro dentro del archivo, primero exploramos el índice y luego usamos el puntero para acceder al archivo directamente y para hallar el registro deseado.

Por ejemplo, un archivo con una lista de precios de venta podría incluir los códigos de producto universales (UPC, universal product code) de los elementos, junto con los precios asociados. Cada registro consistirá en un UPC de 10 dígitos y un precio de 6 dígitos, lo que nos da una longitud de registro de 16 bytes. Si nuestro disco tiene 1024 bytes por bloque, podremos almacenar 64 registros en cada bloque. Un archivo con 120000 registros ocuparía unos 2000 bloques (2 millones de bytes). Si mantenemos el archivo almacenado según el código UPC, podemos definir un índice compuesto por el primer valor UPC de cada bloque. Este índice tendría 2000 entradas de 10 dígitos cada una, es decir, 20000 bytes, y podría por tanto almacenarse en memoria. Para hallar el precio correspondiente a un elemento concreto, podemos hacer una búsqueda binaria en el índice y, con esta búsqueda, determinar exactamente qué bloque contiene el registro deseado, después de lo cual accederemos a este bloque. Esta estructura nos permite explorar un archivo de gran tamaño con un número relativamente bajo de operaciones de E/S.

Con los archivos de gran tamaño, el propio archivo del índice puede ser demasiado grande como para almacenarlo en la memoria. Una solución consiste en crear un índice del archivo índice. El archivo de índice principal contendría punteros a los archivos de índice secundarios que a su vez apuntarían a los propios elementos de datos. Por ejemplo, el método de acceso secuencial indexado de IBM (ISAM, indexed sequential-access method) utiliza un pequeño índice maestro que apunta a los bloques de disco de un índice secundario. Los bloques de índice secundario apuntan a los propios índices del archivo. El archivo se almacena ordenado según una determinada clave. Para localizar un elemento concreto, primero hacemos una búsqueda binaria en el índice principal, que nos proporcionará el número de bloque de índice secundario. A continuación, leemos este bloque y utilizamos de nuevo una búsqueda binaria para hallar el bloque que contiene el registro deseado. Finalmente, se realiza una búsqueda secuencial dentro de este bloque. De esta forma, puede localizarse cualquier registro a partir de su clave mediante, como mucho,

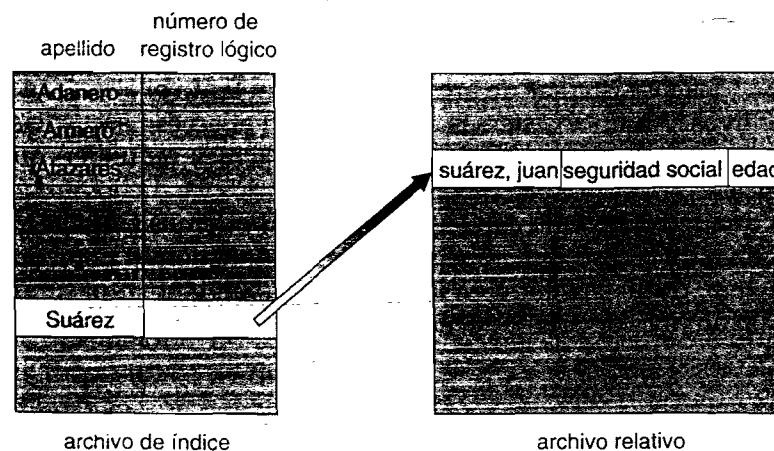


Figura 10.5 Ejemplos de archivo de índice y de archivos relativos.

dos lecturas de acceso directo. La Figura 10.5 muestra una situación similar, implementada mediante los archivos de índice y archivos relativos de VMS.

10.3 Estructura de directorios

Hasta este punto, hemos estado analizando “un sistema de archivos”. En realidad, los sistemas pueden tener cero o más sistemas de archivos y los sistemas de archivos pueden ser de tipos variados. Por ejemplo, un sistema Solaris típico puede tener unos cuantos sistemas de archivos UFS, un sistema de archivos VFS y algunos sistemas de archivos NFS. En el Capítulo 11 se analizan los detalles de la implementación de los sistemas de archivos.

Los sistemas de archivos de las computadoras pueden, por tanto, tener una gran complejidad. Algunos sistemas almacenan millones de archivos en terabytes de disco. Para gestionar todos estos datos, necesitamos organizarlos de alguna manera y esta organización implica el uso de directorios. En esta sección, vamos a analizar el tema de la estructura de directorios. Pero primero, es necesario explicar algunas características básicas de la estructura de almacenamiento.

10.3.1 Estructura de almacenamiento

Un disco (o cualquier dispositivo de almacenamiento que sea lo suficientemente grande) puede utilizarse completamente para un sistema de archivos. Sin embargo, en ciertas ocasiones es deseable colocar múltiples sistemas de archivos en un mismo disco o utilizar partes de un disco para un sistema de archivos y otras partes para otras cosas, como por ejemplo para espacio de intercambio o como espacio de disco sin formato (*raw*). Estas partes se conocen con diversos nombres, como **particiones**, **franjas** o (en el mundo IBM) **minidiscos**. Podemos crear un sistema de archivos en cada una de estas partes del disco. Como veremos en el siguiente capítulo, las partes también pueden combinarse para formar estructuras de mayor tamaño, conocidas con el nombre de **volúmenes**, y también pueden crearse sistemas de archivos en dichos volúmenes. Pero por el momento, en aras de la claridad, utilizaremos el término **volumen** simplemente para referirnos a un espacio de almacenamiento que alberga un sistema de archivos. Cada volumen puede considerarse como si fuera un disco virtual. Los volúmenes pueden también almacenar múltiples sistemas operativos, permitiendo que un sistema se inicie en cualquiera de ellos y ejecute dicho sistema operativo.

Cada volumen que contenga un sistema de archivos debe también contener información acerca de los archivos almacenados en el sistema. Esta información se almacena como entrada en un **directorio de dispositivo** o **tabla de contenidos del volumen**. El directorio del dispositivo (más comúnmente conocido simplemente como **directorio**) almacena información de todos los archivos de dicho volumen, como por ejemplo el nombre, la ubicación, el tamaño y el tipo. La Figura 10.6 muestra una organización típica de un sistema de archivo.

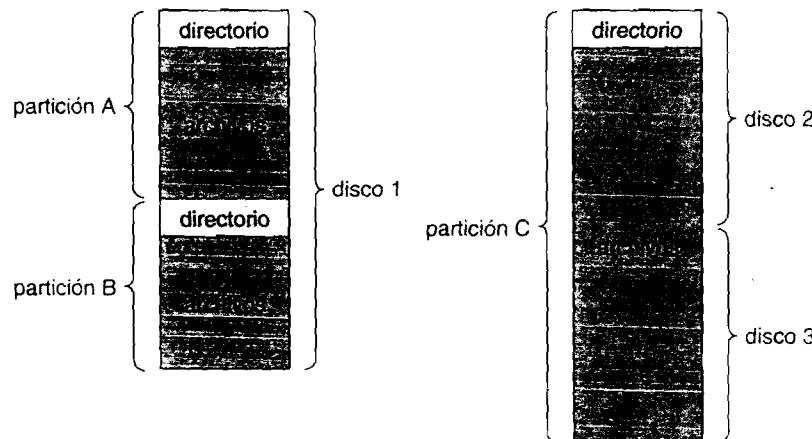


Figura 10.6 Una organización típica de un sistema de archivos.

10.3.2 Introducción a los directorios

El directorio puede considerarse como una tabla de símbolos que traduce los nombres de archivo a sus correspondientes entradas de directorio. Si adoptamos esta visión, es fácil comprender que el propio directorio puede organizarse de muchas formas. Queremos poder insertar entradas, borrar entradas, buscar una entrada conociendo su nombre y enumerar todas las entradas del directorio. En esta sección, vamos a examinar diversos esquemas para definir la estructura lógica del sistema de directorios.

Cuando consideremos una estructura de directorio concreta, tendremos que tener presentes las operaciones que habrá que realizar con el directorio:

- **Búsqueda de un archivo.** Tenemos que poder explorar la estructura de directorio para encontrar la entrada correspondiente a un archivo concreto. Puesto que los archivos tienen nombres simbólicos y los nombres similares pueden indicar que existe una relación entre los archivos, también queremos poder encontrar todos los archivos cuyos nombres se correspondan con un patrón concreto.
- **Crear un archivo.** Es necesario poder crear nuevos archivos y añadirlos al directorio.
- **Borrar un archivo.** Cuando un archivo ya no es necesario, queremos poder eliminarlo del directorio.
- **Listar un directorio.** Tenemos que poder enumerar los archivos contenidos en un directorio y el contenido de la entrada de directorio correspondiente a cada uno de los archivos de la lista.
- **Renombrar un archivo.** Puesto que el nombre de un archivo representa su contenido para los usuarios, debemos poder cambiar el nombre cuando el contenido o el uso del archivo varíen. Renombrar un archivo puede también significar que se modifique su posición dentro de la estructura de directorio.
- **Recorrer el sistema de archivos.** Puede que queramos acceder a todos los directorios y a todos los archivos contenidos dentro de una estructura de directorios. Para conseguir una mayor fiabilidad, resulta conveniente guardar el contenido y la estructura de todo el sistema de archivos a intervalos regulares. A menudo, esto se suele hacer copiando todos los archivos en una cinta magnética. Esta técnica proporciona una copia de seguridad para el caso de que se produzca un fallo del sistema. Además, si un archivo ya ha dejado de utilizarse, el archivo puede copiarse en cinta y puede liberarse el espacio de disco ocupado por ese archivo, con el fin de que lo reutilicen otros archivos.

En las siguientes secciones, vamos a describir los esquemas más comunes que se usan para definir la estructura lógica de un directorio.

10.3.3 Directorio de un único nivel

La estructura de directorio más simple es el directorio de un único nivel. Todos los archivos están contenidos en el mismo directorio, que resulta fácil de mantener y de comprender (Figura 10.7).

Sin embargo, un directorio de un único nivel tiene limitaciones significativas cuando el número de archivos se incrementa o cuando el sistema tiene más de un usuario. Puesto que todos los archivos se encuentran en el mismo directorio, deberán tener nombres distintivos. Si dos usuarios llaman a su archivo de datos *test*, se violará la regla de unicidad de los nombres. Por ejemplo, en una clase de programación, 23 estudiantes denominaron al programa correspondiente a la segunda de las prácticas realizadas *prog2*, mientras que otros 11 lo denominaron *assign2*. Aunque los nombres de archivo se seleccionan, por regla general, de modo que reflejen el contenido del archivo, a menudo están limitados en longitud, lo que complica la tarea de hacer que esos nombres de archivos sean únicos. El sistema operativo MS-DOS sólo permite utilizar nombres de archivo de once caracteres; UNIX, por el contrario, permite hasta 255 caracteres.

Incluso aunque sólo haya un usuario en un directorio de un único nivel, puede que ese usuario tenga dificultades para recordar los nombres de todos los archivos a medida que se incremen-

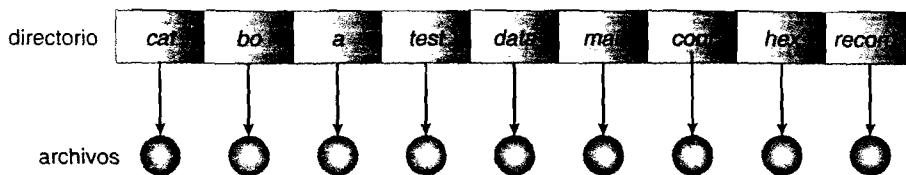


Figura 10.7 Directorio de un único nivel.

ta el número de archivos. No resulta extraño que un usuario tenga cientos de archivos en su sistema informático y un número igual de archivos adicionales en otros sistemas. Controlar todos esos archivos es una tarea extremadamente compleja.

10.3.4 Directorio en dos niveles

Como hemos visto, los directorios de un único nivel conducen a menudo a que exista confusión entre los nombres de archivo de los distintos usuarios. La solución estándar consiste en crear un directorio *separado* para cada usuario.

En la estructura de directorio de dos niveles, cada usuario tiene su propio **directorio de archivos de usuario** (UFD, user file directory). Cada uno de los UFD tiene una estructura similar, pero sólo incluye los archivos de un único usuario. Cuando un trabajo de un usuario se inicia o cuando un usuario se conecta al sistema, se explora el **directorio maestro de archivos** (MFD, master file directory) del sistema. El MFD está indexado por el nombre de usuario o por el número de cuenta y cada una de sus entradas apunta al UFD de dicho usuario (Figura 10.8).

Cuando un usuario hace referencia a un archivo concreto, sólo se explora su propio UFD. Por tanto, cada uno de los usuarios puede tener archivos con el mismo nombre, siempre y cuando los nombres de archivo dentro de cada UFD sean únicos. Para crear un archivo para un usuario, el sistema operativo sólo explora el UFD de ese usuario para comprobar si ya existe otro archivo con el mismo nombre. Para borrar un archivo, el sistema operativo confina su búsqueda al UFD local y no puede, por tanto, borrar accidentalmente un archivo de otro usuario que tenga el mismo nombre.

Los propios directorios de usuario deben poder crearse y borrarse según sea necesario. Para ello se ejecuta un programa especial del sistema, con el nombre de usuario apropiado y la correspondiente información de cuenta. El programa crea un nuevo UFD y añade una entrada para él en el MFD. La ejecución de este programa puede estar restringida a los administradores del sistema. La asignación de espacio de disco para las direcciones de usuario puede gestionarse mediante las técnicas explicadas en el Capítulo 11 para los propios archivos.

Aunque la estructura de directorio en dos niveles resuelve el problema de la colisión de nombres, sigue teniendo ciertas desventajas. Esta estructura aísla efectivamente a un usuario y otro y ese aislamiento es una ventaja cuando los usuarios son completamente independientes, pero puede llegar a ser una desventaja cuando los usuarios *quieren* cooperar en una cierta tarea y poder acceder a los archivos del otro. Algunos sistemas simplemente no permiten que un usuario acceda a los archivos locales de otro usuario.

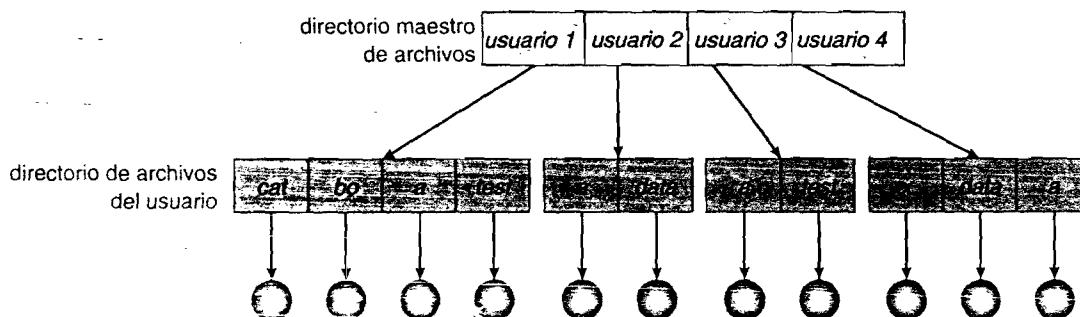


Figura 10.8 Estructura de directorios en dos niveles.

Si queremos permitir ese tipo de acceso, cada usuario debe tener la posibilidad de especificar un archivo que pertenezca al directorio de otro usuario. Para denominar a los archivos de manera única en una estructura de directorio de dos niveles, debemos proporcionar tanto el nombre de usuario como el nombre del archivo. Un directorio en dos niveles puede verse como una estructura de árbol, o de árbol invertido, de altura 2. La raíz del árbol es el MFD y sus descendientes directos son los UFD. Los descendientes de los UFD son los propios archivos, que actúan como hojas del árbol. Al especificar un nombre de usuario y un nombre de archivo, estamos definiendo una ruta dentro del árbol que va desde la raíz (el MFD) hasta una hoja (el archivo especificado). Por tanto, un nombre de usuario y un nombre de archivo definen un *nombre de ruta*. Cada archivo del sistema tiene un nombre de ruta y para designar a un archivo de manera única, el usuario debe conocer el nombre de ruta del archivo deseado.

Por ejemplo, si el usuario A quiere acceder a su propio archivo denominado *test*, simplemente puede referirse a él como *test*. Sin embargo, para acceder al archivo denominado *test* del usuario B (cuyo nombre de entrada de directorio es *userb*), puede que tenga que utilizar la notación */userb/test*. Cada sistema tiene su propia sintaxis para nombrar los archivos contenidos en los directorios que no pertenecen al propio usuario.

Se necesita una sintaxis adicional para especificar el volumen de un archivo. Por ejemplo, en MS-DOS un volumen se especifica mediante una letra seguida de un carácter de dos puntos. Por tanto, una especificación de archivo podría tener el siguiente aspecto *C:/userb/test*. Algunos sistemas van todavía más allá y separan las partes de la especificación correspondientes al volumen, al nombre del directorio y al nombre del archivo. Por ejemplo, en VMS, el archivo *login.com* puede especificarse como *u:[sst.jdeck]login.com;1*, donde *u* es el nombre del volumen, *sst* es el nombre del directorio, *jdeck* es el nombre del subdirectorío y *1* es el número de versión. Otros sistemas simplemente tratan el nombre del volumen como parte del nombre de directorio. El primer nombre que se proporcione será el del volumen y el resto se referirá al directorio y al archivo. Por ejemplo, */u/pbg/test* podría especificar el volumen *u*, el directorio *pbg* y el archivo *test*.

Un caso especial de esta situación es el que se refiere a los archivos del sistema. Los programas proporcionados como parte del sistema (cargadores, ensambladores, compiladores, rutinas de utilidad, bibliotecas, etc.) están generalmente definidos como archivos. Cuando se proporcionan los comandos apropiados al sistema operativo, el cargador carga estos archivos y los ejecuta. Muchos intérpretes de comandos simplemente tratan dicho comando como el nombre de un archivo que hay que cargar y ejecutar. Tal como hemos definido el sistema de directorios, este nombre de archivos se buscaría en el UFD actual. Una solución consiste en copiar los archivos del sistema dentro de cada UFD, pero copiar todos los archivos del sistema implicaría desperdiciar una cantidad enorme de espacio (si los archivos del sistema requieren 5 MB, soportar a 12 usuarios requeriría $5 \times 12 = 60$ MB simplemente para las copias de los archivos del sistema).

La solución estándar consiste en complicar el procedimiento de búsqueda ligeramente. Se define un directorio de usuario especial para contener los archivos del sistema (por ejemplo, el usuario 0). Cada vez que se proporciona un nombre de archivo para cargarlo, el sistema operativo analiza primero el UFD local; si encuentra allí el archivo, lo utiliza, pero si no lo encuentra, el sistema explora automáticamente el directorio de usuario especial que contiene los archivos del sistema. La secuencia de directorios que se exploran cada vez que se proporciona un nombre de archivos se denomina **ruta de búsqueda**. La ruta de búsqueda puede emplearse para contener un número ilimitado de directorios que haya que explorar cada vez que se proporcione un nombre de comando. Este método es el más utilizado en UNIX y MS-DOS. También pueden diseñarse los sistemas de modo que cada usuario tenga su propia ruta de búsqueda.

10.3.5 Directorios con estructura de árbol

Una vez que hemos visto cómo puede contemplarse un directorio en dos niveles como un árbol de dos niveles, la generalización natural consiste en ampliar la estructura de directorios para que sea un árbol de altura arbitraria (Figura 10.9). Esta generalización permite a los usuarios crear sus propios subdirectorios y organizar sus archivos correspondientemente. Los árboles son la estructura de directorio más común. Cada árbol tiene un directorio raíz y todos los archivos del sistema tienen un nombre de ruta distintivo.

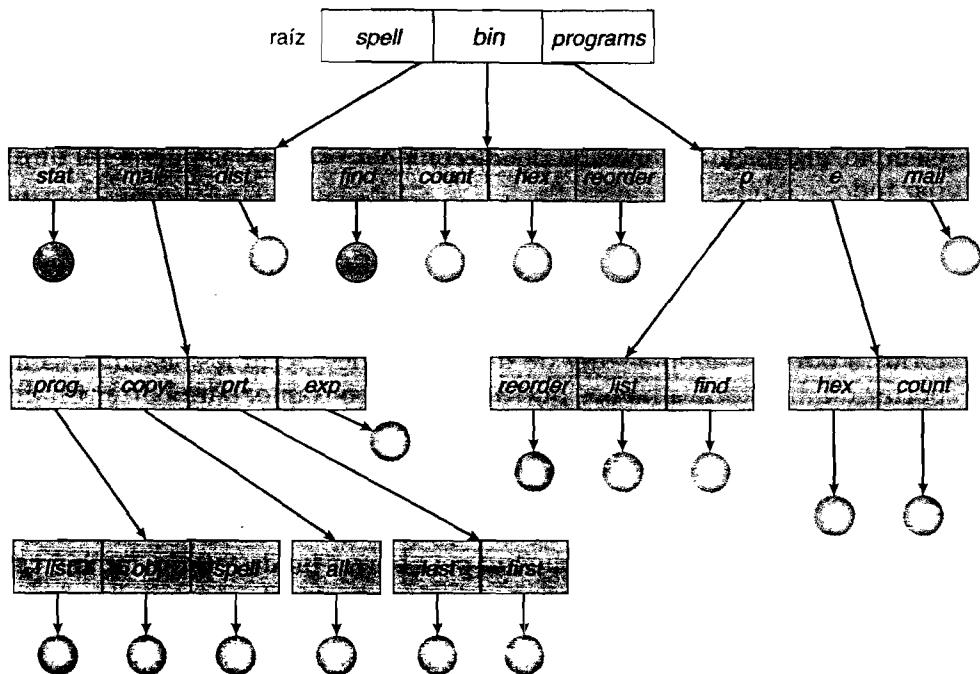


Figura 10.9 Estructura de directorio en forma de árbol.

Cada directorio (o subdirectorío) contiene un conjunto de archivos o subdirectoríos. Un directorio es simplemente otro archivo, pero que se trata de forma especial. Todos los directorios tienen el mismo formato interno. Un bit de cada entrada de directorio define si esa entrada es un archivo (0) o un subdirectorío (1). Se utilizan llamadas al sistema especiales para crear y borrar los directorios.

En el uso normal, cada proceso tiene un directorio actual. Ese **directorio actual** debe contener la mayor parte de los archivos que actualmente interesen al proceso. Cuando se hace referencia a un archivo, se explora el directorio actual. Si se necesita un archivo que no se encuentre en el directorio actual, entonces el usuario deberá normalmente especificar un nombre de ruta o cambiar el directorio actual, para situarse en el directorio donde fue almacenado ese archivo. Para cambiar de directorio, se proporciona una llamada al sistema que toma como parámetro un nombre de directorio y lo utiliza para redefinir el directorio actual. Así, el usuario puede cambiar su directorio actual cada vez que lo deseé. Entre una llamada al sistema `change_directory` (cambiar directorio) y la siguiente, todas las llamadas al sistema `open` analizarán el directorio actual en busca del archivo especificado. Observe que la ruta de búsqueda puede o no contener una entrada especial que signifique “el directorio actual”.

El directorio actual inicial de la shell de inicio de sesión de un usuario se designa en el momento de comenzar el trabajo del usuario o en el momento en que el usuario inicia la sesión. El sistema operativo analiza el archivo de cuentas (o alguna otra ubicación predefinida) con el fin de localizar la entrada correspondiente a este usuario (para propósitos de contabilización). En el archivo de cuentas habrá almacenado un puntero (o el nombre) al directorio inicial del usuario. Este puntero se copia en una variable local de ese usuario que especifica el directorio actual inicial del mismo. A partir de esa shell, pueden arrancarse otros procesos. El directorio actual de cada subprocesso será, usualmente, el directorio actual que su proceso padre tenía en el momento de crear el subprocesso.

Los nombres de ruta pueden ser de dos tipos: *absolutos* y *relativos*. Un **nombre de ruta absoluto** comienza en la raíz y sigue una ruta descendente hasta el archivo especificado, indicando los nombres de los directorios que componen la ruta. Un **nombre de ruta relativo** define una ruta a partir del directorio actual. Por ejemplo, en el sistema de archivos con estructura de árbol de la Figura 10.9, si el directorio actual es `root/spell/mail`, entonces el nombre de ruta relativo `prt/first` hará referencia al mismo archivo que el nombre de ruta absoluto `root/spell/mail/prt/first`.

Permitir a un usuario definir sus propios subdirectorios hace que el usuario pueda imponer una estructura a sus archivos. Esta estructura puede dar como resultado que se utilicen directorios separados para los archivos asociados con diferentes temas (por ejemplo, el subdirectorio que nosotros creamos para almacenar el texto de este libro) o con diferentes tipos de información (por ejemplo, el directorio *programs* puede contener programas puente; el directorio *bin* puede almacenar todos los binarios, etc.).

Una decisión interesante de política dentro de un directorio con estructura de árbol es la que se refiere a qué hacer si se borra un directorio. Si el directorio está vacío, podemos simplemente borrar la entrada correspondiente dentro del directorio que lo contuviera. Sin embargo, suponga que el directorio que hay que borrar no está vacío, sino que contiene varios archivos o subdirectorios. Podemos adoptar una de dos soluciones. Algunos sistemas, como MS-DOS, no permiten borrar un directorio a menos que esté vacío; por tanto, para borrar un directorio, el usuario debe primero borrar todos los archivos contenidos en ese directorio. Si existen subdirectorios, este procedimiento debe aplicarse recursivamente a los mismos, para que también puedan ser borrados. Esta técnica puede dar como resultado que haya que realizar una gran cantidad de trabajo. Otra técnica alternativa, como la adoptada por el comando *rm* de UNIX, consiste en proporcionar una opción: cuando se hace una solicitud para borrar un directorio, también se borran todos los archivos y subdirectorios de dicho directorio. Las dos técnicas son fáciles de implementar y la decisión entre una y otra es cuestión de la política que se adopte. La última de las dos políticas mencionadas es más cómoda, pero también es más peligrosa, porque puede eliminarse una estructura de directorios completa con un único comando. Si se ejecutara por error ese comando, puede que fuera necesario restaurar un gran número de archivos y directorios (suponiendo que exista una copia de seguridad).

Con un sistema de directorios con estructura de árbol, podemos permitir que los usuarios accedan a los archivos de otros usuarios, además de acceder a los suyos propios. Por ejemplo, el usuario B puede acceder a un archivo del usuario A especificando su nombre de ruta; el usuario B puede especificar un nombre de ruta relativo o absoluto. Alternativamente, el usuario B puede cambiar el subdirectorio actual para situarse en el directorio del usuario A y acceder al archivo simplemente proporcionando su nombre.

Una ruta a un archivo en un directorio con estructura de árbol puede ser más larga que las rutas típicas de los directorios en dos niveles. Para permitir a los usuarios acceder a los programas sin tener que recordar esos largos nombres de ruta, el sistema operativo Macintosh automatizaba la búsqueda de los programas ejecutables. Este sistema mantiene un archivo, denominado *Desktop File*, que contiene los nombres y ubicaciones de todos los programas ejecutables que ha encontrado. Cuando se añade un nuevo disco duro o disquete al sistema, o cuando se accede a la red, el sistema operativo recorre la estructura de directorios en busca de programas ejecutables que pueda haber en el dispositivo y almacena la información pertinente. Este mecanismo soporta la funcionalidad de ejecución mediante doble clic que hemos descrito anteriormente. Un doble clic sobre un archivo hace que se lea su atributo de creador y que se explore el archivo *Desktop File* en busca de una correspondencia. Si se encuentra una correspondencia, se inicia el programa ejecutable apropiado, utilizando como entrada el archivo sobre el que se ha hecho clic. La familia Microsoft Windows de sistemas operativos (95, 98, NT, 2000, XP) mantiene una estructura de directorio en dos niveles ampliada, asignando letras de unidad a los dispositivos y a los volúmenes (Sección 10.4).

10.3.6 Directorios en un grafo acíclico

Considere dos programadores que estén trabajando en un proyecto conjunto. Los archivos asociados con dicho proyecto pueden almacenarse en un subdirectorio, separándolos de otros proyectos y archivos de los dos programadores. Pero, como ambos programadores son igualmente responsables del proyecto, ambos quieren que el subdirectorio se encuentre dentro de su propio directorio. El subdirectorio común debe, por tanto, ser *compartido*. Cada directorio o archivo compartido existirá en el sistema de archivos en dos (o más) lugares simultáneamente.

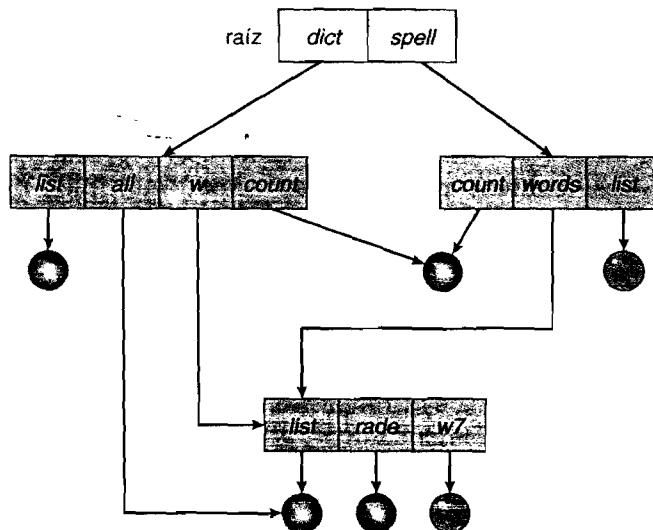


Figura 10.10 Estructura de directorio en grafo acíclico.

Una estructura en árbol prohíbe la compartición de archivos o directorios. Por el contrario, un grafo acíclico (es decir, un grafo sin ningún ciclo) permite que los directorios compartan subdirectorios de archivos (Figura 10.10). El *mismo* archivo o subdirectorio puede estar en dos directorios diferentes. El grafo acíclico es una generalización natural del esquema de directorio con estructura de árbol.

Es importante observar que un archivo (o directorio) compartido no es lo mismo que dos copias del archivo. Con dos copias cada programador puede ver su propia copia en lugar del original, pero si un programador modifica el archivo, esos cambios no se reflejarán en la copia del otro. Con un archivo compartido, sólo existe *un* archivo real, por lo que cualquier cambio realizado por una persona será inmediatamente visible para la otra. La compartición es particularmente importante para los subdirectorios; cada nuevo archivo creado por una persona aparecerá automáticamente en todos los subdirectorios compartidos,

Cuando las personas trabajan en grupo, todos los archivos que quieran compartir pueden colocarse dentro de un directorio. El UFD de cada miembro del grupo contendrá como subdirectorio este directorio de archivos compartidos. Incluso en el caso de un único usuario, la organización de archivos del usuario puede requerir que algún archivo sea colocado en diferentes subdirectorios. Por ejemplo, un programa escrito para un proyecto concreto podría tener que almacenarse tanto en el directorio correspondiente a todos los programas como en el directorio correspondiente a ese trayecto.

Los archivos y subdirectorios compartidos pueden implementarse de diversas formas. Una manera bastante común, ejemplificada por muchos de los sistemas UNIX, consiste en crear una nueva entrada de directorio denominada enlace (*link*). Un enlace es, en la práctica, un puntero a otro archivo o subdirectorio. Por ejemplo, un enlace puede implementarse como un nombre de ruta absoluto o relativo. Cuando se realiza una referencia a un archivo, exploramos el directorio y, si la entrada del directorio está marcada como enlace, entonces se incluye el nombre del archivo real dentro de la información del enlace. Para resolver el enlace, utilizamos el nombre de ruta con el fin de localizar el archivo real. Los enlaces pueden identificarse fácilmente por el formato que tienen en la entrada de directorio (o bien porque tengan un tipo especial en aquellos sistemas que soporten los tipos) y son, en la práctica, punteros indirectos nominados. El sistema operativo ignora estos enlaces a la hora de recorrer los árboles de directorio, con el fin de preservar la estructura acíclica del sistema.

Otra técnica común para la implementación de archivos compartidos consiste simplemente en duplicar toda la información acerca de esos archivos en todos los directorios que comparten esos archivos. Así, ambas entradas serán idénticas. Un enlace es, claramente, diferente de la entrada original de directorio, por lo que ambas entradas no son iguales; sin embargo, las entradas de

directorio duplicadas hacen que el original y la copia sean indistinguibles. El principal problema con las entradas de directorio duplicadas es el de mantener la coherencia cuando se modifica un archivo.

Una estructura de directorio en forma de grafo acíclico es más flexible que una estructura simple en árbol, pero también es más compleja. Es necesario prestar cuidadosa atención a determinados problemas. Los archivos podrán tener ahora múltiples nombres de ruta absoluta. En consecuencia, puede haber nombres de archivo distintos que hagan referencia a un mismo archivo. La situación es similar al problema de los alias en los lenguajes de programación. Si estamos intentando recorrer el sistema de archivos completo (para encontrar un archivo, para acumular estadísticas sobre todos los archivos o para copiar todos los archivos en un dispositivo de copia de seguridad) este problema cobra una gran importancia, ya que no conviene recorrer las estructuras compartidas más de una vez.

Otro problema es el que se refiere al borrado. ¿Cuándo puede desasignarse y reutilizarse el espacio asignado a un archivo compartido? Una posibilidad consiste en eliminar el archivo cuando un usuario cualquiera lo borre, pero esta acción puede dejar punteros colgantes al archivo que ha dejado de existir. El problema puede complicarse si los punteros de archivo restantes contienen direcciones reales de disco y ese espacio se reutiliza subsiguientemente para otros archivos; en ese caso, esos punteros colgantes pueden apuntar a un lugar cualquiera de esos nuevos archivos.

En un sistema en el que la compartición se implemente mediante enlaces simbólicos, la situación es algo más fácil de manejar. El borrado de un enlace no tiene por qué afectar al archivo original, ya que sólo se elimina el enlace. Si lo que se elimina es la propia entrada del archivo, se desasignará el espacio del archivo, dejando que los enlaces cuelguen. Podemos buscar estos enlaces y eliminarlos también, pero a menos que se mantenga con cada archivo una lista de los enlaces asociados esta búsqueda puede ser bastante costosa. Alternativamente, podemos dejar esos enlaces hasta que se produzca un intento de utilizarlos, en cuyo momento podemos determinar que el archivo del nombre indicado por el enlace no existe y que no podemos resolver el nombre del enlace; el acceso se tratará exactamente igual que se haría con cualquier otro nombre legal de archivo (en este caso, el diseñador del sistema debe considerar cuidadosamente qué hacer cuando se borra un archivo y se crea otro archivo del mismo nombre, antes de que se utilice un enlace simbólico al archivo original). En el caso de UNIX, cuando se borra un archivo se dejan los enlaces simbólicos y es responsabilidad del usuario darse cuenta de que el archivo original ya no existe o ha sido sustituido. Microsoft Windows (todas las versiones) utiliza la misma técnica.

Otra técnica de borrado consiste en preservar el archivo hasta que se borren todas las referencias al mismo. Para implementar esta técnica, debemos disponer de algún mecanismo para determinar que se ha borrado la última referencia al archivo. Podríamos mantener una lista de todas las referencias al archivo (entradas de directorio o enlaces simbólicos). Cuando se establece un enlace o una copia de la entrada de directorio, se añade una nueva entrada a la lista de referencias al archivo. Cuando se borra un enlace o una entrada de directorio, eliminamos la correspondiente entrada de la lista. El archivo se borrará cuando se vacíe su lista de referencias al archivo.

El problema con esa técnica es el tamaño variable, y potencialmente grande, de la lista de referencias al archivo. Sin embargo, no es necesario que mantengamos la lista completa, sino que bastaría con mantener sólo un recuerdo del *número* de referencia. Añadir una nueva entrada de directorio o un nuevo enlace hará que se incremente el contador de referencias, mientras que borrar un enlace o una entrada hará que el contador se decremente. Cuando el contador sea 0, podrá borrarse el archivo, ya que no habrá más referencias a él. El sistema operativo UNIX utiliza esta técnica para los enlaces no simbólicos (o enlaces duros), manteniendo un contador de referencias dentro del bloque de información del archivo (o *inode*; véase el Apéndice A.7.2). Prohibiendo en la práctica que existan múltiples referencias a los directorios, podemos mantener una estructura de grafo acíclico.

Para evitar problemas como los que se acaban de describir, algunos sistemas no permiten que existan enlaces o directorios compartidos. Por ejemplo, en MS-DOS, la estructura de directorio es una estructura de árbol en lugar de un grafo acíclico.

10.3.7 Directorio en forma de grafo general

Uno de los problemas más graves que afectan a la utilización de una estructura en forma de grafo acíclico consiste en garantizar que no exista ningún ciclo. Si comenzamos con un directorio en dos niveles y permitimos a los usuarios crear subdirectorios, obtendremos un directorio con estructura de árbol. Es fácil darse cuenta de que la simple adición de nuevos archivos y subdirectorios a una estructura en árbol existente preserva la naturaleza y la estructura de árbol de ese directorio. Sin embargo, si añadimos enlaces a un directorio existente con estructura de árbol, la estructura en árbol se destruye, dando como resultado una estructura en grafo simple (Figura 10.11).

La principal ventaja de un grafo acíclico es la relativa simplicidad de los algoritmos requeridos para recorrer el grafo y para determinar cuándo no existen ya más referencias a un archivo. Necesitamos poder evitar el tener que recorrer las secciones compartidas de un grafo cíclico dos veces, principalmente por razones de rendimiento. Si acabamos de explorar un subdirectorio compartido de gran tamaño en busca de un archivo concreto sin haber llegado a encontrarlo, necesitaremos evitar tener que volver a explorar dicho subdirectorio otra vez, ya que esa segunda búsqueda sería una pérdida de tiempo.

Si permitimos que existan ciclos en el directorio, necesitaremos, de la misma manera evitar tener que buscar cualquier componente dos veces, por razones tanto de corrección como de rendimiento. Un algoritmo mal diseñado podría provocar un bucle infinito que estuviera explorando el ciclo continuamente, sin terminar nunca. Una solución consiste en limitar arbitrariamente el número de directorio a los que se accederá durante una búsqueda.

Un problema similar existe cuando tratamos de determinar si un archivo puede ser borrado. Con las estructuras de directorio en grafo acíclico, un valor de 0 en el contador de referencia significará que ya no hay más referencias al archivo o directorio, y que ese archivo puede ser borrado. Sin embargo, si existen ciclos, el contador de referencias puede no ser 0 incluso aunque ya no sea posible hacer referencia a un directorio de archivo. Esta anomalía resulta de la posibilidad de auto-referencia (o de un ciclo) en la estructura de directorio. En este caso, generalmente necesitaremos utilizar un esquema de recolección de memoria para determinar cuándo se ha borrado la última referencia y que, en consecuencia, puede reasignarse el espacio de disco. La recolección en memoria implica recorrer todo el sistema de archivos, marcando todos aquellos elementos a los que se pueda acceder. Después, una segunda pasada recopila todo aquello que no esté marcado, incluyéndolo en una lista de espacio libre (puede utilizarse un procedimiento similar de marcación para garantizar que todo recorrido o búsqueda visitará cada elemento del sistema de archivos una vez y sólo una vez). Sin embargo, la recolección de memoria para un sistema de archivos basado en disco consume muchísimo tiempo y raramente se la suele utilizar.

La recolección de memoria sólo es necesaria debido a la posible existencia de ciclos dentro del grafo. Por tanto, es mucho más sencillo trabajar con estructuras de grafo acíclico. La principal difi-

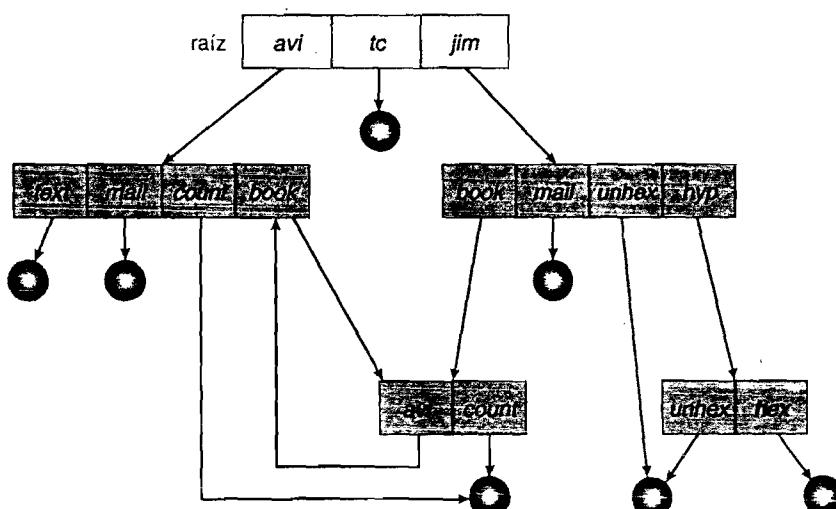


Figura 10.11 Directorio con forma de grafo general.

cultad es la de evitar que aparezcan ciclos a medida que se añaden nuevos enlaces a la estructura. ¿Cómo podemos saber si un nuevo enlace va a completar el ciclo? Existen algoritmos para detectar la existencia de ciclos en los grafos; sin embargo, estos algoritmos son muy costosos desde el punto de vista computacional, especialmente cuando el grafo se encuentra almacenado en disco. Un algoritmo más simple en el caso especial de directorios y enlaces consiste en ignorar los enlaces durante el recorrido de los directorios. De este modo, se evitan los ciclos sin necesidad de efectuar ningún procesamiento adicional.

10.4 Montaje de sistemas de archivos

De la misma forma que un archivo debe *abrirse* antes de utilizarlo, un sistema de archivos debe *montarse* para poder estar disponible para los procesos del sistema. Más específicamente, la estructura de directorios puede estar formada por múltiples volúmenes, que puede montarse para hacer que estén disponibles dentro del espacio de nombres del sistema de archivos.

El proceso de montaje es bastante simple. Al sistema operativo se le proporciona el nombre de dispositivo y el **punto de montaje** que es la ubicación dentro de la estructura de archivos a la que hay que conectar el sistema de archivos que se está montando. Normalmente, el punto de montaje será un directorio vacío. Por ejemplo, en un sistema UNIX, un sistema de archivos que contenga los directorios principales de un usuario puede montarse como */home*; después, para acceder a la estructura de directorios contenida en ese sistema de archivos, podemos anteponer a los nombres de directorio el nombre */home*, como por ejemplo en */home/jane*. Si montáramos ese sistema de archivos bajo el directorio */users* obtendríamos el nombre de ruta */users/jane*, que podríamos utilizar para acceder al mismo directorio.

A continuación, el sistema operativo verifica que el dispositivo contiene un sistema de archivos válido. Para ello, pide al controlador del dispositivo que lea el directorio de dispositivo y verifique que ese directorio tiene el formato esperado. Finalmente, el sistema operativo registra en su estructura de directorios que hay un sistema de archivo montado en el punto de montaje especificado. Este esquema permite al sistema operativo recorrer su estructura de directorios, pasando de un sistema de archivos a otro según sea necesario.

Para ilustrar el montaje de archivos, considere el sistema de archivos mostrado en la Figura 10.12, donde los triángulos representan subárboles de directorios en los que estamos interesados. La Figura 10.12(a) muestra un sistema de archivos existente, mientras que la Figura 10.12(b) muestra un volumen no montado que reside en */device/dsk*. En este punto, sólo puede accederse a los archivos del sistema de archivos existente. La Figura 10.13 muestra los efectos de mostrar en

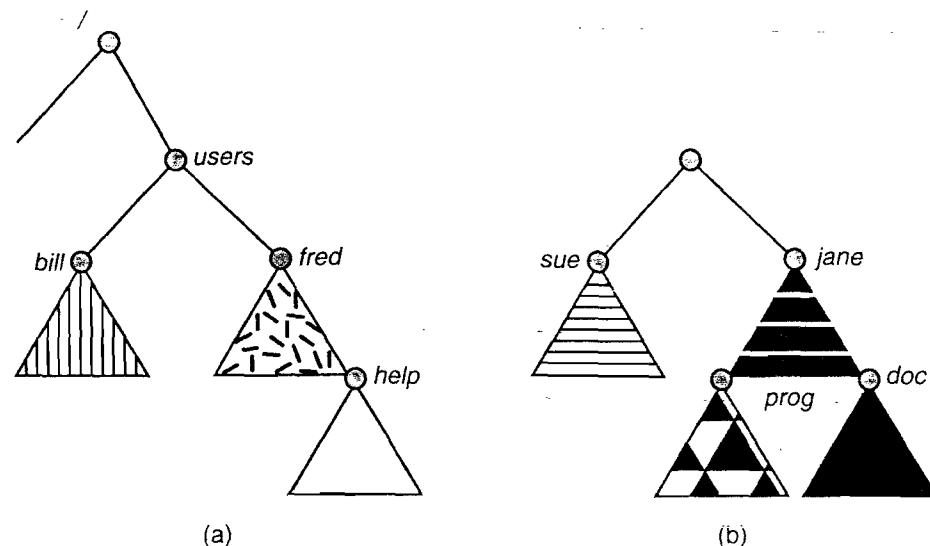


Figura 10.12 Sistema de archivos. (a) Sistema existente. (b) Volumen no montado.

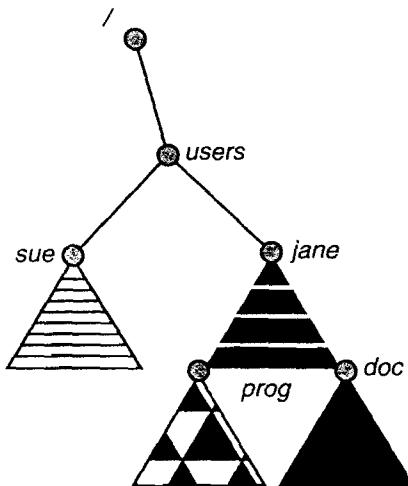


Figura 10.13 Punto de montaje.

/users el volumen que reside en /device/dsk. Si se desmonta el volumen, el sistema de archivos vuelve a la situación mostrada en la Figura 10.12.

Los sistemas imponen una cierta semántica para clarificar la funcionalidad. Por ejemplo, un determinado sistema podría prohibir que se realizara un montaje en un directorio que contuviera archivos, o bien podría hacer que el sistema de archivos montado estuviera disponible en dichos directorios y ocultar los archivos existentes en dicho directorio hasta que el sistema de archivos se desmontara, en cuyo momento se prohíbe el uso de ese sistema de archivos y se permite el acceso a los archivos originales contenidos en el directorio. Otro ejemplo, un determinado sistema podría permitir que se montara repetidamente el mismo sistema de archivos, en diferentes puntos de montaje; o, por el contrario, podría permitir un único montaje por cada sistema de archivos.

Consideremos el ejemplo del sistema operativo Macintosh. Cuando el sistema encuentra un disco por primera vez (los discos duros se localizan en el momento de iniciar el sistema, mientras que los disquetes se detectan cuando se insertan en la unidad), el sistema operativo Macintosh busca un sistema de archivos en el dispositivo. Si encuentra uno, monta automáticamente el sistema de archivos en el nivel raíz, añadiendo a la pantalla un ícono de carpeta cuya etiqueta coincide con el nombre del sistema de archivos (tal como esté almacenado en el directorio del dispositivo). El usuario podrá entonces hacer clic sobre el ícono y mostrar el sistema de archivos recién montado.

La familia Microsoft Windows de sistemas operativos (95, 98, NT, 2000, XP) mantiene una estructura ampliada de directorio en dos niveles, en la que a los dispositivos en los volúmenes se les asignan letras de unidad. Los volúmenes tienen una estructura de directorio en forma de grafo general asociada con la letra de la unidad. La ruta para un archivo específico toma la forma *letra-unidad:\ruta\al\archivo*. Las versiones más recientes de Windows permiten montar un sistema de archivos en cualquier punto del árbol de directorios, al igual que en UNIX. Los sistemas operativos Windows descubren automáticamente todos los dispositivos y montan todos los sistemas de archivo localizados en el momento de iniciar el sistema. En algunos sistemas, como UNIX, los comandos de montaje son explícitos. Un determinado archivo de configuración del sistema contiene una lista de los dispositivos y puntos de montaje para realizar el montaje automático en el momento de iniciar el sistema, pero pueden ejecutarse otras operaciones de montaje manualmente.

Los temas relativos al montaje de sistema de archivos se analizan con más profundidad en la Sección 11.2.2 y en el Apéndice A.7.5.

10.5 Compartición de archivos

En las secciones anteriores, hemos explorado los motivos que existen para compartir archivos y algunas de las dificultades que surgen al permitir a los usuarios compartir archivos. Dicha com-

partición de archivos es muy deseable para aquellos usuarios que quieran colaborar y reducir el esfuerzo requerido para conseguir desarrollar un programa. Por tanto, los sistemas operativos orientados al usuario deben satisfacer la necesidad de compartir archivos a pesar de las dificultades inherentes a este mecanismo.

En esta sección, vamos a examinar algunos aspectos adicionales de la compartición de archivos. Comenzaremos analizando diversas cuestiones generales que surgen en el instante en que se permite a múltiples usuarios compartir los archivos. Una vez que se permite compartir los archivos a diversos usuarios, el desafío principal consiste en ampliar la compartición a múltiples sistemas de archivos, incluyendo sistemas de archivos remotos y vamos a analizar también aquellas dificultades. Finalmente, analizaremos el tema de qué hacer cuando tienen lugar acciones conflictivas en los archivos compartidos; por ejemplo, si hay múltiples usuarios escribiendo en un archivo, ¿debe permitirse que se lleven a cabo todas las escrituras o el sistema operativo debe, por el contrario, proteger las acciones de los usuarios frente a las posibles interferencias con otras secciones similares?

10.5.1 Múltiples usuarios

Cuando un sistema operativo tiene múltiples usuarios, las cuestiones relativas a la compartición de archivos, a la denominación de archivos y a la protección de archivos cobran una gran importancia. Dada una estructura de directorio que permite que los usuarios comparten archivos, el sistema debe adoptar un papel de mediador en lo que a la compartición de archivos respecta. El sistema puede permitir que un usuario acceda a los archivos de otros usuarios de manera predeterminada, o puede por el contrario requerir que los usuarios concedan explícitamente el acceso a sus archivos. Estos son los problemas del control de acceso y de la protección, que trataremos en la Sección 10.6.

Para implementar la compartición y los mecanismos de protección, el sistema debe mantener más atributos de los archivos y de los directorios que los que se necesitan en un sistema monouuario. Aunque históricamente se han adoptado diversos enfoques para satisfacer estos requisitos, la mayoría de los sistemas han terminado por evolucionar para usar el concepto de *propietario* (o *usuario*) de un archivo o directorio y el concepto de *grupo*. El propietario es el usuario que puede cambiar los atributos y conceder el acceso y que dispone del máximo grado de control sobre el archivo. El atributo de grupo define un subconjunto de usuarios que pueden compartir el acceso al archivo. Por ejemplo, el propietario de un archivo en un sistema UNIX puede ejecutar todas las operaciones sobre el archivo, mientras que los miembros del grupo de un archivo sólo pueden ejecutar un subconjunto de esas operaciones, y todos los restantes usuarios podrán ejecutar otro subconjunto de operaciones. El propietario del archivo es quien define exactamente qué operaciones pueden ser ejecutadas por los miembros del grupo y por los restantes usuarios. En la siguiente sección proporcionaremos más detalles sobre los atributos relativos a los permisos.

Los identificadores del propietario y del grupo de un archivo (o directorio) determinado se almacenan junto con los otros atributos del archivo. Cuando un usuario solicita realizar una operación sobre un archivo, se puede comparar el ID del usuario con el atributo de propietario para determinar si el usuario solicitante es el propietario del archivo. De la misma manera, pueden compararse los identificadores de grupo. El resultado indicará qué permisos son aplicables. A continuación, el sistema aplicará dichos permisos a la operación solicitada y la autorizará o denegará.

Muchos sistemas disponen de múltiples sistemas de archivos locales, incluyendo volúmenes compuestos por un único disco o múltiples volúmenes almacenados en múltiples discos conectados. En estos casos, los procesos de comprobación de las identidades y de comprobación de los permisos son bastante sencillos, una vez que los sistemas de archivos hayan sido montados.

10.5.2 Sistemas de archivos remotos

Con el advenimiento de las redes (Capítulo 16) se ha hecho posible la comunicación entre computadoras remotas. La interconexión por red permite la compartición de una serie de recursos que

son distribuidos por un campus o incluso por todo el mundo. Uno de los recursos más obvios para compartir son los datos existentes, en la forma de archivos.

Con la evolución de las tecnologías de red y de archivos, los métodos de compartición de archivos remotos han ido cambiando. El primer método que se implementó implicaba la transferencia manual de archivos entre dos máquinas mediante programas como `ftp`. El segundo método principal utiliza un **sistema de archivos distribuido** (DFS, distributed file system) en el que los directorios remotos son visibles desde una máquina local. En cierta forma, el tercer método, la **World Wide Web**, es una reversión al primero de los métodos indicados. Se necesita un explorador para poder acceder a los archivos remotos y se utilizan operaciones independientes (esencialmente un envoltorio para `ftp`) para transferir los archivos.

`ftp` se utiliza tanto para el acceso anónimo como para el acceso autenticado. El **acceso anónimo** permite al usuario transferir archivos sin disponer de una cuenta en el sistema remoto. La World Wide Web utiliza casi exclusivamente un mecanismo de intercambio anónimo de archivos. DFS requiere una integración mucho más estrecha entre la máquina que está accediendo a los archivos remotos y la máquina que suministra los archivos. Esta integración añade un alto grado de complejidad, que vamos a describir en esta sección.

10.5.2.1 El modelo cliente-servidor

Los sistemas de archivos remotos permiten a una computadora montar uno o más sistemas de archivos desde una o más máquinas remotas. En este caso, la máquina que contiene los archivos es el *servidor*, mientras que la máquina que trata de acceder a los archivos es el *cliente*. La relación cliente-servidor resulta muy común en las máquinas conectadas por red. Generalmente, el servidor declara ante los clientes que hay un cierto recurso disponible y especifica exactamente de qué recurso se trata (en este caso, qué archivos son) y exactamente para qué clientes está disponible. Un servidor puede dar servicio a múltiples clientes y cada cliente puede utilizar múltiples servidores, dependiendo de los detalles de implementación de cada arquitectura cliente-servidor.

El servidor especifica usualmente los archivos disponibles en el nivel de volumen o de directorio. La identificación de los clientes resulta algo más complicada. Un cliente puede estar especificado por un nombre de red o por otro identificador, como por ejemplo una *dirección IP*, pero estas identidades pueden ser **suplantadas o imitadas**. Como resultado de la suplantación, un cliente no autorizado podría obtener acceso al servidor. Otras soluciones más seguras incluyen una autenticación segura del cliente mediante claves cifradas. Desafortunadamente, los mecanismos de seguridad presentan numerosos desafíos, incluyendo garantizar la compatibilidad del cliente y el servidor (ambos deben usar los mismos algoritmos de cifrado) y la seguridad de los intercambios de claves (las claves interceptadas podrían, de nuevo, permitir un acceso no autorizado). Debido a la dificultad de resolver estos problemas, la solución más común consiste en emplear métodos de autenticación no seguros.

En el caso de UNIX y en sus sistemas de archivos en red (NFS, network file system), la autenticación tiene lugar, de manera predeterminada, mediante la información de conexión de red del cliente. Según este esquema, los identificadores del usuario en el cliente y en el servidor deben corresponderse; si no lo hacen, el servidor no podrá determinar los derechos de acceso a los archivos. Considere el ejemplo de un usuario que tenga un ID igual a 1000 en el cliente e igual a 2000 en el servidor. Una solicitud emitida por el cliente y dirigida al servidor que se refiera a un archivo específico no podrá ser tratada apropiadamente, ya que el servidor trataría de comprobar si el usuario 1000 tiene acceso al archivo, en lugar de basar esa determinación en el *verdadero* ID del usuario, que es igual a 2000. En consecuencia, el acceso se concedería o se denegaría basándose en una información de autenticación incorrecta. El servidor debe confiar en que el cliente le presente el ID de usuario correcto. Observe que los protocolos NFS permiten relaciones muchos a muchos, es decir, muchos servidores pueden proporcionar archivos a muchos clientes. De hecho, una máquina determinada puede ser a la vez un servidor para otros clientes NFS y un cliente de otros servidores NFS.

Una vez que el sistema de archivos remoto haya sido montado, las solicitudes de operaciones con los archivos se envían al usuario por cuenta del usuario a través de la red, utilizando el protocolo DFS. Normalmente, la solicitud de apertura de un archivo se envía junto con el ID del usua-

rio solicitante. El servidor aplica entonces las comprobaciones de acceso normales para determinar si el usuario dispone de credenciales para acceder al archivo del modo solicitado. En consecuencia, la solicitud será concedida o denegada. Si se la concede, se devuelve un descriptor de archivo a la aplicación cliente y esa aplicación puede a partir de ahí realizar lecturas, escrituras y otras operaciones con el archivo. Cuando haya terminado de acceder, el cliente cerrará el archivo. El sistema operativo puede aplicar una semántica similar a la del montaje de un sistema de archivos local o puede utilizar una semántica distinta.

10.5.2.2 Sistemas de información distribuidos

Para hacer que los sistemas cliente-servidor sean más fáciles de gestionar, los **sistemas de información distribuidos**, también conocidos como **servicios de denominación distribuidos**, proporcionan un acceso unificado a la información necesaria para la informática remota. El **sistema de nombres de dominio** (DNS, domain name system) proporciona un servicio de traducción de nombres de host a direcciones de red para toda Internet (incluyendo la World Wide Web). Antes de la adopción general de DNS, se intercambiaban mediante correo electrónico o ftp entre todos los host conectados en red archivos que contenían la misma información. Esta metodología no era escalable. Hablaremos de DNS con más detalle en la Sección 16.5.1.

Otros sistemas de información distribuidos proporcionan un espacio de *nombre de usuario/contraseña/ID de usuario/ID de grupo* para una instalación distribuida. Los sistemas UNIX han empleado a lo largo de los años diversos métodos de distribución de información. Sun Micro-system introdujo las **páginas amarillas** (que después se denominaron **servicio de información de red**, o NIS, network information service) y la mayoría del sector adoptó su uso. Este mecanismo centraliza el almacenamiento de nombres de usuario, nombres de host, información de impresoras y otros datos similares. Desafortunadamente, utilizan métodos de autenticación inseguros, incluyendo el envío sin cifrar (como *texto en claro*) de las contraseñas de usuario y la identificación de los host mediante sus direcciones IP. El sistema NIS+ de Sun es un sustituto de NIS mucho más seguro, pero también mucho más complicado, y no ha sido adoptado de manera general. En el caso del **sistema común de archivos Internet** (CIFS, common internet file system) de Microsoft, se utiliza la información de red en conjunción con la autenticación de usuario (nombre de usuario y contraseña) para crear un **inicio de sesión de red** que el servidor emplea para decidir si puede o no permitir el acceso al sistema de archivos solicitado. Para que esta autenticación sea válida, los nombres de usuario deben coincidir en las distintas máquinas (al igual que con NFS). Microsoft utiliza dos estructuras distribuidas de denominación para proporcionar un único espacio de nombres para los usuarios. La tecnología de denominación más antigua son los **dominios**. La tecnología más moderna, disponible en Windows XP y Windows 2000, es **Active Directory**. Una vez instalada, la funcionalidad de denominación distribuida es utilizada por todos los clientes y por todos los servidores para autenticar a los usuarios.

El sector está evolucionando hacia la utilización del **protocolo ligero de acceso al directorio** (LDAP, ligthweight directory-access protocol) como mecanismo distribuido seguro de denominación. De hecho, Active Directory está basado en LDAP. Sun Microsystems incluye LDAP en su sistema operativo y permite utilizarlo para la autenticación de los usuarios, así como para el acceso global a información del sistema, como por ejemplo la información de disponibilidad de las impresoras. En teoría, una organización podría utilizar un directorio LDAP distribuido para almacenar toda la información sobre usuarios y recursos relativa a las computadoras de las que la organización disponga. El resultado sería un mecanismo de **inicio de sesión unificado seguro** para los usuarios, que introducirían una única vez su información de autenticación para acceder a todas las computadoras de la organización. Esto podría facilitar también las tareas de administración del sistema al combinar en una misma ubicación una serie de informaciones que actualmente están distribuidas entre varios archivos en cada sistema, o entre varios servicios de información distribuidos.

10.5.2.3 Modos de fallo

Los sistemas de archivos locales pueden fallar por diversas razones, incluyendo los fallos del disco donde esté almacenado el sistema de archivos, la corrupción de la estructura de directorios

o de alguna otra información de gestión del disco (colectivamente denominada **metadatos**), los fallos de las controladoras de disco, los fallos de cable y los fallos de las adaptadoras de las máquinas host. Los errores de los usuarios o de los administradores del sistema también pueden hacer que se pierdan archivos o que se borren directorios o volúmenes completos. Muchos de estos fallos pueden hacer que un host experimente un fallo catastrófico y que muestre algún tipo de mensaje de error, requiriéndose la intervención humana para reparar los daños.

Los sistemas de archivos remotos tienen todavía más modos de fallo. Debido a la complejidad de los sistemas de red y a las interacciones requeridas entre las máquinas remotas, hay muchos más problemas que pueden interferir con la adecuada operación de los sistemas de archivos remotos. En el caso de las redes, la red que conecta a dos hosts podría verse interrumpida y dicha interrupción podría ser provocada por un fallo del hardware, por una inadecuada configuración del hardware o por algún problema relativo a la implementación de la interconexión por red. Aunque algunas redes tienen mecanismos de protección incorporados, incluyendo la existencia de múltiples rutas de conexión entre los hosts, otras muchas redes no disponen de esas protecciones. Así, cualquier simple fallo puede interrumpir el flujo de comandos DFS.

Consideré un cliente que esté utilizando un sistema de archivos remoto. Suponga que el cliente tiene una serie de archivos abiertos en el host remoto y que, entre otras actividades, puede estar realizando búsquedas en directorios para abrir archivos, puede estar leyendo o escribiendo datos en los archivos o puede estar cerrando archivos. Ahora piense en qué sucede si se produce un particionamiento de la red, un fallo catastrófico del servidor o incluso una detención programada del servidor. De repente, el sistema de archivos remoto ya no será alcanzable. Este escenario resulta bastante común, así que no sería apropiado que el sistema cliente actúe de la misma forma que si se hubiera perdido un sistema de archivos local. En lugar de ello, el sistema puede terminar todas las operaciones que hubiera iniciado en el servidor ahora inalcanzable o puede retardar las operaciones hasta que pueda volver a alcanzar a dicho servidor. Esta semántica de fallo se define e implementa como parte del protocolo del sistema de archivos remoto. La terminación de todas las operaciones puede dar como resultado que los usuarios pierdan sus datos y la paciencia; por tanto, la mayoría de los protocolos DFS imponen o permiten que se retarden las operaciones del sistema de archivos que se estuvieran realizando en los hosts remotos, en la esperanza de que el host remoto vuelva a estar disponible.

Para implementar este tipo de recuperación de fallos, puede mantenerse algo de **información de estado** tanto en el cliente como en el servidor. Si tanto el servidor como el cliente mantienen información sobre sus actividades actuales y los archivos abiertos, podrán recuperarse transparentemente de un fallo. En aquellas situaciones en las que el servidor sufre un fallo catastrófico pero debe reconocer que tiene sistemas de archivos exportados montados de manera remota, así como archivos abiertos, NFS adopta un enfoque simple, implementando un **DFS sin memoria del estado**. En esencia, se presupone que no puede producirse una solicitud de un cliente para efectuar una lectura o escritura en un archivo a menos que el sistema de archivos haya sido montado remotamente y ese archivo se haya abierto de forma previa. El protocolo NFS transporta toda la información necesaria para localizar el archivo apropiado y realizar la operación solicitada. De forma similar, el protocolo no controla qué clientes han montado los volúmenes exportados, asumiendo de nuevo que si llega una solicitud esta debe ser legítima. Aunque esta técnica sin memoria del estado hace que NFS sea muy resistente a los fallos y bastante fácil de implementar, también hace que sea un protocolo inseguro. Por ejemplo, determinadas solicitudes ilegítimas de lectura o escritura podrían llegar a ser permitidas por un servidor NFS aun cuando la necesaria solicitud de montaje y las correspondientes comprobaciones de permisos no hubieran tenido lugar. Estos problemas se resuelven en el estándar NFS versión 4, que hace que NFS tenga memoria del estado para mejorar su seguridad, sus prestaciones y su funcionalidad.

10.5.3 Semántica de coherencia

La **semántica de coherencia** representa un criterio importante para evaluar todo sistema de archivos que soporte la compartición de archivos. Esta semántica especifica cómo pueden acceder simultáneamente a un archivo los múltiples usuarios del sistema. En particular, especifica cuándo las modificaciones que un usuario realice en los datos serán observables por parte de los otros

usuarios. Esta semántica se implementa, típicamente, como código software dentro del sistema de archivos.

La semántica de coherencia está directamente relacionada con los algoritmos de sincronización de procesos que hemos presentado en el Capítulo 6. Sin embargo, lo normal es que los algoritmos complejos de dicho capítulo no se implementen en el caso de la E/S de archivo, debido a la gran latencia y a las bajas velocidades de transferencia de los discos y de las redes. Por ejemplo, la realización de una transacción atómica en un disco remoto podría implicar que se llevaran a cabo varias operaciones de comunicación por red, varias lecturas y escrituras en disco, o ambas cosas. Los sistemas que tratan de proporcionar dicho conjunto completo de funcionalidades tienden a exhibir unas bajas prestaciones. En el sistema de archivos Andrew se incluye una implementación bastante satisfactoria de este tipo de semántica compleja de compartición.

Para nuestras siguientes explicaciones, vamos a asumir que la serie de accesos a archivo (es decir, lecturas y escrituras) realizados por un usuario con un archivo determinado siempre está encerrada entre sendas operaciones `open()` y `close()`. La serie de acceso comprendida entre las operaciones `open()` y `close()` forma una **sesión de archivo**. Para ilustrar este concepto, vamos a esbozar varios ejemplos significativos de la semántica de coherencia.

10.5.3.1 Semántica de UNIX

El sistema de archivos UNIX (Capítulo 17) utiliza la siguiente semántica de coherencia:

- Las escrituras en un archivo abierto por parte de un usuario son visibles inmediatamente para los otros usuarios que hayan abierto ese archivo.
- Un modo de compartición permite a los usuarios compartir el puntero de ubicación actual dentro del archivo. Así, el incremento de ese puntero por parte del usuario afectará a todos los usuarios que estén compartiendo el archivo. En este caso, el archivo tiene una única imagen en la que se entrelazan todos los accesos, independientemente de su origen.

En la semántica de UNIX, cada archivo está asociado con una única imagen física a la que se accede en forma de recurso exclusivo. La contienda por esta única imagen provoca retardos en los procesos de usuario.

10.5.3.2 Semántica de sesión

El sistema de archivos Andrew (AFS, del que hablaremos en el Capítulo 17), utiliza la siguiente semántica de coherencia:

- Las escrituras en un archivo abierto por parte de un usuario no son visibles inmediatamente para los restantes usuarios que hayan abierto el mismo archivo.
- Una vez que se cierra un archivo, los cambios realizados en él son visibles únicamente en las sesiones que den comienzo posteriormente. Las instancias ya abiertas del archivo no reflejarán dichos cambios.

De acuerdo con esta semántica, un archivo puede estar temporalmente asociado con varias (posiblemente diferentes) imágenes al mismo tiempo. En consecuencia, se permite que múltiples usuarios realicen accesos concurrentes tanto de lectura como de escritura en sus propias imágenes del archivo, sin ningún retardo. No se impone prácticamente ninguna restricción a la planificación de los accesos.

10.5.3.3 Semántica de archivos compartidos inmutables

Una técnica bastante original es la de los **archivos compartidos inmutables**. Una vez un archivo es declarado como *compartido* por su creador, no puede ser modificado. Un archivo inmutable tiene dos propiedades clave. Su nombre no puede reutilizarse y su contenido no puede ser modificado. Por tanto, el hecho de que un archivo sea inmutable significa que su contenido está fijo. La implementación de esta semántica en un sistema distribuido (Capítulo 17) es bastante simple, porque el mecanismo de compartición es disciplinado (de sólo lectura).

10.6 Protección

Cuando se almacena información en un sistema informático, necesitamos protegerla frente a los daños físicos (*fiabilidad*) y frente a los accesos incorrectos (*protección*).

La *fiabilidad* se proporciona, generalmente, mediante copias duplicadas de los archivos. Muchas computadoras tienen programas del sistema que copian automáticamente (o mediante la intervención del operador) los archivos del disco en una cinta a intervalos regulares (una vez al día, o a la semana o al mes), para mantener una copia por si acaso resultara destruido accidentalmente el sistema de archivos. Los sistemas de archivos pueden verse dañados por problemas de hardware (como por ejemplo errores en la lectura o escritura), sobretensiones o caídas de tensión, aterrizajes de cabezales del disco, suciedad, temperaturas extremas y vandalismo. Los archivos también pueden borrarse accidentalmente y los errores en el software del sistema de archivos también pueden provocar la pérdida del contenido de un archivo. La cuestión de la fiabilidad se cubre con mayor detalle en el Capítulo 12.

Podemos proporcionar protección de varias formas. Para un pequeño sistema monousuario, podemos proporcionar protección extrayendo físicamente los disquetes y guardándolos bajo llave en un cajón o en un archivador. En los sistemas multiusuario, sin embargo, se necesita utilizar otros mecanismos.

10.6.1 Tipos de acceso

La necesidad de proteger los archivos es consecuencia directa de la posibilidad de acceder a esos archivos. Los sistemas que no permiten el acceso a los sistemas de otros usuarios no necesitan ninguna protección. Así, podríamos proporcionar una protección completa simplemente prohibiendo el acceso. Alternativamente, podríamos proporcionar un acceso libre sin ninguna protección. Ambas técnicas son demasiado extremas para poder utilizarlas de forma general; lo que necesitamos, en su lugar, es un **acceso controlado**.

Los mecanismos de protección proporcionan un acceso controlado limitando los tipos de accesos a archivo que puedan realizarse. El acceso se permite o se deniega dependiendo de varios factores, uno de los cuales es el tipo de acceso solicitado. Podemos controlar varios tipos de operaciones diferentes:

- **Lectura.** Lectura del archivo.
- **Escritura.** Escritura o reescritura del archivo.
- **Ejecución.** Carga del archivo en memoria y ejecución del mismo.
- **Adición.** Escritura de nueva información al final del archivo.
- **Borrado.** Borrado del archivo y liberación del espacio para su posible reutilización.
- **Listado.** Listado del nombre y atributos del archivo.

Otras operaciones, como el renombrado, el copiado y la edición del archivo también pueden controlarse. Sin embargo, para muchos sistemas, estas funciones de mayor nivel pueden implementarse mediante un programa del sistema que realice llamadas al sistema de menor nivel y la protección se proporciona únicamente en ese nivel inferior. Por ejemplo, la copia de un archivo puede implementarse simplemente mediante una secuencia de solicitudes de lectura. En este caso, un usuario con acceso de lectura también podrá copiar el archivo, imprimirla, etc.

Se han propuesto muchos mecanismos de protección y cada uno tiene sus ventajas y desventajas y es preciso valorar si resulta apropiado para cada aplicación en cuestión. Un pequeño sistema informático que sólo sea utilizado por un pequeño grupo de miembros de un departamento de investigación, por ejemplo, puede no necesitar los mismos tipos de protección que una gran computadora corporativa que se utilice para investigación, finanzas y gestión de personal. Hablaremos de algunas de las técnicas de protección en las siguientes secciones y presentaremos un tratamiento más completo en el Capítulo 14.

10.6.2 Control de acceso

La técnica más común para resolver el problema de la protección consiste en hacer que el acceso dependa de la identidad del usuario. Los diferentes usuarios pueden necesitar diferentes tipos de acceso a un archivo o directorio. El esquema más general para implementar un acceso dependiente de la identidad consiste en asociar con cada archivo y directorio una **lista de control de acceso** (ACL, access-control list) que especifique los nombres de usuario y los tipos de acceso que se permiten para cada uno. Cuando un usuario solicita acceder a un archivo concreto, el sistema operativo comprueba la lista de acceso asociada con dicho archivo; si dicho usuario está incluido en la lista para el tipo de acceso solicitado, se permite el acceso. En caso contrario, se producirá una violación de la protección y se denegará al trabajo de usuario el acceso al archivo.

Esta técnica tiene la ventaja de permitir la implementación de complejas metodologías de acceso. El problema principal con las listas de acceso es su longitud. Si queremos permitir que todo el mundo lea un archivo, deberemos enumerar todos los usuarios que disponen de ese acceso de lectura. Esta técnica tiene dos consecuencias poco deseables:

- Construir dicha lista puede ser una tarea tediosa y poco gratificante, especialmente si no sabemos de antemano la lista de usuarios del sistema.
- La entrada de directorio, que anteriormente tenía un tamaño fijo, ahora tendrá que ser de tamaño variable, lo que requiere mecanismos más complejos de gestión del espacio.

Estos problemas pueden resolverse utilizando una versión condensada de la lista de acceso.

Para condensar la longitud de la lista de control de acceso, muchos sistemas clasifican a los usuarios en tres grupos, en lo que respecta con cada archivo:

- **Propietario.** El usuario que creó el archivo será su propietario.
- **Grupo.** Un conjunto de usuarios que están compartiendo el archivo y necesitan un acceso similar al mismo es un grupo, o grupo de trabajo.
- **Universo.** Todos los demás usuarios del sistema constituyen el universo.

La técnica reciente más común consiste en combinar las listas de control de acceso con el esquema más general (y más fácil de implementar) de control de acceso que acabamos de describir, compuesto por los conceptos de propietario, grupo y universo. Por ejemplo, Solaris 2.6 y las versiones posteriores utilizan estas tres categorías de acceso de manera predeterminada, pero permiten añadir listas de control de acceso a archivos y directorios específicos, cuando se desee disponer de un control de acceso de granularidad más fina.

Como ilustración, considere una persona, Sara, que esté escribiendo un nuevo libro. Sara ha contratado a tres estudiantes (Jaime, David y Julia) para que la ayuden con el proyecto. El texto del libro se almacena en un archivo denominado *book*. La protección asociada con este archivo será la siguiente:

- Sara debe poder invocar todas las operaciones sobre el archivo.
- Jaime, David y Julia sólo deben poder leer y escribir el archivo; no se les debe permitir borrar el archivo.
- Todos los demás usuarios deben poder leer, pero no escribir, el archivo (Sara está interesada en dejar que el texto sea leído por el mayor número de personas posible, para poder recibir sugerencias).

Para conseguir dicha protección, debemos crear un nuevo grupo (por ejemplo, *text*) cuyos miembros serán Jaime, David y Julia. El nombre del grupo, *text*, deberá entonces asociarse con el archivo *book* y los derechos de acceso deberán configurarse de acuerdo con la política que hemos esbozado anteriormente.

Ahora considere el caso de un visitante al que Sara quisiera conceder un acceso temporal al Capítulo 1. El visitante no puede ser añadido al grupo *text* porque eso le daría acceso a todos los capítulos. Asimismo, como cada archivo sólo puede estar en un grupo, no podemos añadir otro

grupo para el Capítulo 1. Con la adición de la funcionalidad basada en listas de control de acceso, el visitante puede añadirse a la lista de control de acceso del Capítulo 1.

Para que este esquema funcione adecuadamente, deben controlarse estrechamente tanto los permisos como las listas de control de acceso. Este control puede llevarse a cabo de varias formas distintas. Por ejemplo, en el sistema UNIX, los grupos sólo pueden ser creados y modificados por el administrador de la instalación (o por cualquier superusuario). Así, el grado de control requerido se consigue gracias a la intervención humana. En el sistema VMS, el propietario del archivo puede crear y modificar esta lista de control de acceso. Hablaremos más en detalle de las listas de acceso en la Sección 14.5.2.

Si utilizamos la clasificación más limitada de protección, sólo necesitamos tres campos para definir los mecanismos de protección. A menudo, cada campo es una colección de bits y cada bit permite o deniega el acceso asociado con el mismo. Por ejemplo, el sistema UNIX define tres campos de 3 bits cada uno: *rwx*, donde *r* controla el acceso de lectura, *w* controla el acceso de escritura y *x* controla la ejecución. Se mantiene un campo distinto para el propietario del archivo, para el grupo al que el archivo pertenece y para todos los demás usuarios. En este esquema, se necesitan nueve bits por archivo para almacenar la información de protección. Así, para nuestro ejemplo, los campos de protección para el archivo *book* serían los siguientes: para el propietario *Sara*, todos los bits estarán activados; para el grupo *text*, estarán activados los bits *r* y *w* y para el universo sólo estará activado el bit *x*.

Una dificultad a la hora de combinar las distintas técnicas es la que surge en relación con la interfaz de usuario. Los usuarios deben poder determinar cuándo están activados los permisos ACL opcionales en un archivo. En el ejemplo de Solaris, un símbolo "+" añade los permisos normales, como el:

```
19 -rw-r--r--+ 1 jim staff 130 May 25 22:13 file1
```

Se utiliza un conjunto separado de comandos, *setfacl* y *getfacl*, para gestionar las listas ACL.

Los usuarios de Windows XP suelen gestionar las listas de control de acceso mediante la GUI. La Figura 10.14 muestra una ventana de permisos de archivo en el sistema de archivos NTFS de Windows XP. En este ejemplo, al usuario "guest" se le deniega específicamente el acceso al archivo *10.tex*.

Otra dificultad es la relativa a la asignación de precedencias cuando los permisos y las listas ACL entran en conflicto. Por ejemplo, si Juan pertenece al grupo de un archivo, que tiene permiso de lectura, pero el archivo tiene una lista ACL que concede a Juan permiso de lectura y escritura, ¿debemos conceder o denegar una solicitud de escritura realizada por Juan? Solaris proporciona a las listas ACL prioridad sobre los permisos (ya que son de granularidad más fina y no se asignan de manera predeterminada). Esto sigue la regla general de que lo más específico debe tener prioridad sobre lo más genérico.

10.6.3 Otras técnicas de protección

Otra técnica para resolver el problema de la protección consiste en asociar una contraseña con cada archivo. Al igual que el acceso al sistema informático suele controlarse mediante una contraseña, el acceso a cada archivo puede controlarse de la misma forma. Si se seleccionan aleatoriamente las contraseñas y se las cambia con frecuencia, este esquema puede ser muy efectivo a la hora de limitar el acceso a un archivo. Sin embargo, el uso de contraseñas tiene unas cuantas desventajas. En primer lugar, el número de contraseñas que un usuario necesita recordar puede llegar a ser muy grande, haciendo que este esquema resulte poco práctico. En segundo lugar, si se utiliza una única contraseña para todos los archivos, una vez que esa contraseña sea descubierta todos los archivos serán accesibles, de modo que la protección se proporciona en forma "todo o nada". Algunos sistemas (por ejemplo TOPS-20) permiten a los usuarios efectuar una contraseña con un subdirectorio, en lugar de con un archivo concreto, para aliviar este problema. El sistema operativo VM/CMS de IBM permite asignar tres contraseñas a cada minidisco: una para lectura, otra para escritura y otra para acceso de multiescritura.

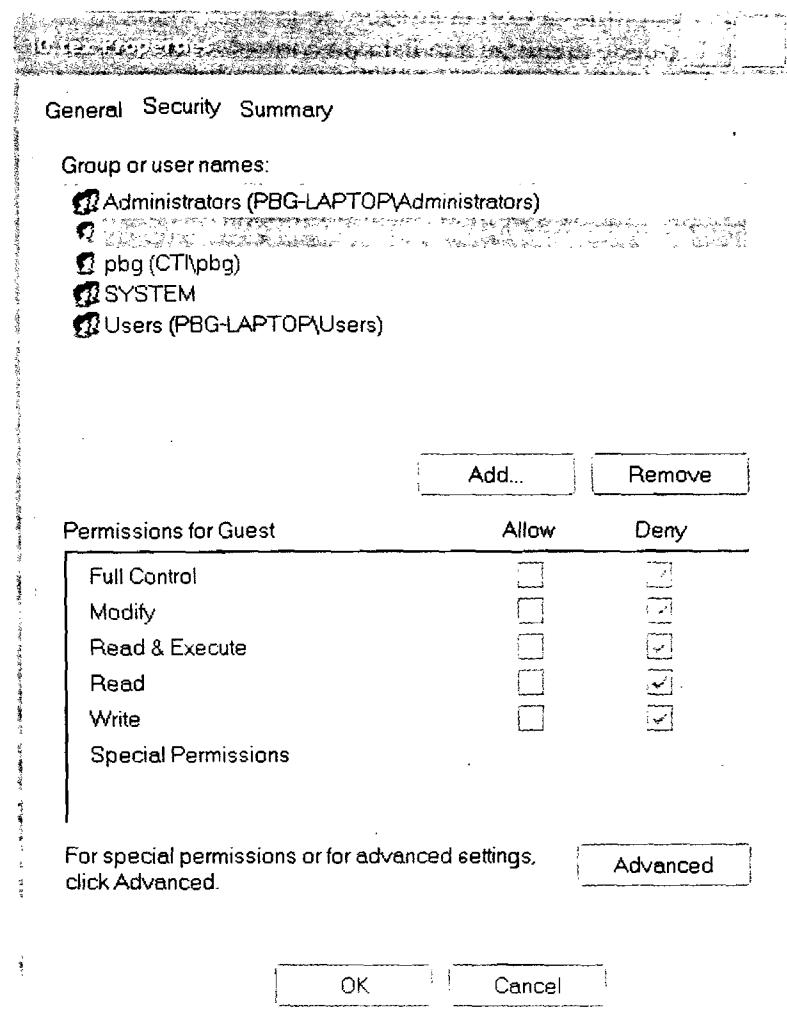
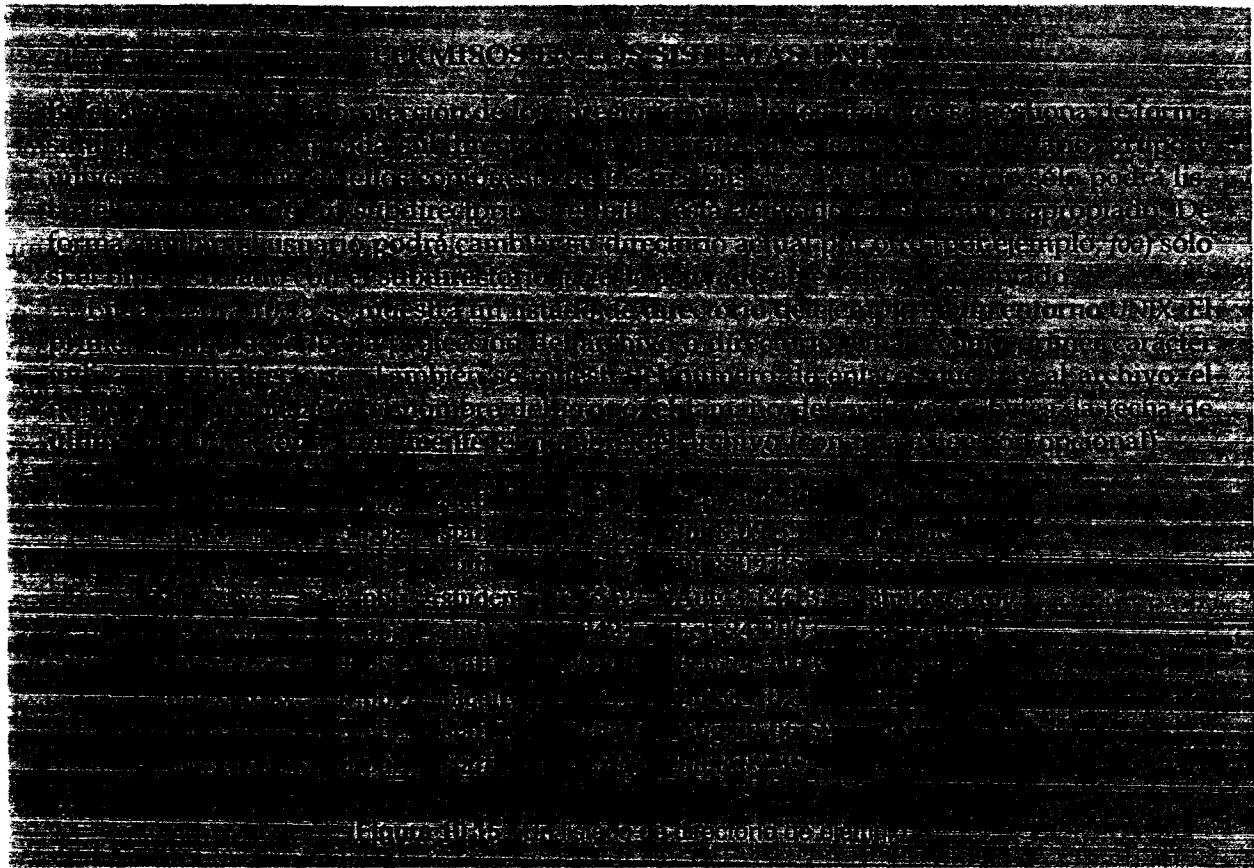


Figura 10.14 Gestión de listas de control de acceso en Windows XP.

Algunos sistemas operativos monousuario (como MS-DOS y las versiones del sistema operativo Macintosh anteriores a Mac OS X) proporcionan pocos mecanismos de protección de archivos. En aquellos casos en que estos sistemas antiguos están siendo conectados a redes en las que son necesarias la compartición y la comunicación de archivos, se están **retroimplementando** mecanismos de protección en ellos. Diseñar una determinada funcionalidad para un nuevo sistema operativo es casi siempre más fácil que añadir una funcionalidad a un sistema operativo existente. Dichas actualizaciones suelen ser menos efectivas y no resultan transparentes.

En una estructura de directorios multinivel, necesitamos proteger no sólo los archivos individuales sino también las colecciones de archivo contenidas en los subdirectorios; en otras palabras, necesitamos proporcionar un mecanismo para la protección de los directorios. Las operaciones de directorio que deben protegerse son algo distintas de las operaciones de archivo. Queremos controlar la protección y borrado de los archivos de un directorio y, además, probablemente queremos controlar si un usuario puede determinar la existencia de un determinado archivo dentro de un directorio. Algunas veces, el conocimiento de la existencia y del nombre de un archivo son significativos por sí mismos, por lo que listar el contenido de un directorio debe ser una operación protegida. De forma similar, si un nombre de ruta hace referencia a un archivo dentro de un directorio, debe permitirse al usuario acceder tanto al directorio como al archivo. En aquellos sistemas en los que los archivos puedan tener numerosos nombres de ruta (como por ejemplo los grafos acíclicos o los grafos generales), un cierto usuario puede tener diferentes derechos de acceso a un archivo concreto, dependiendo del nombre de ruta que utilice.



10.7 Resumen

Un archivo es un tipo abstracto de datos definido e implementado por el sistema operativo. Se trata de una secuencia de registros lógicos donde cada registro puede ser un byte, una línea (de longitud fija o variable) o un elemento de datos más complejo. El sistema operativo puede soportar específicamente diversos tipos de registros o puede dejar dicho soporte a los programas de aplicación.

La principal tarea del sistema operativo consiste en mapear el concepto de archivo lógico sobre los dispositivos de almacenamiento físico, como las cintas magnéticas o discos. Puesto que el tamaño del registro físico del dispositivo puede no coincidir con el tamaño del registro lógico, puede que sea necesario empaquetar los registros lógicos en los registros físicos. De nuevo, esta tarea puede ser soportada por el sistema operativo o dejada para que se encargue de ella el programa de aplicación.

Cada dispositivo de un sistema de archivos mantiene una tabla de contenidos del volumen o un directorio del dispositivo en donde se indica la ubicación de los archivos dentro del dispositivo. Además, resulta útil crear directorios que permitan organizar los archivos. Los directorios de un único nivel en los sistemas monousuario hacen que aparezcan problemas de denominación, ya que cada archivo deberá tener un nombre distintivo. Los directorios en dos niveles resuelven este problema creando un directorio separado por cada usuario; cada usuario tendrá su propio directorio, que contendrá sus propios archivos. El directorio enumera los archivos según su nombre e incluye la información sobre la ubicación del archivo dentro del disco, sobre la longitud del archivo, sobre su tipo, su propietario, la fecha de su creación, la fecha de último uso, etc.

La generalización natural de un directorio en dos niveles son los directorios con estructura de árbol. Un directorio con estructura de árbol permite al usuario crear subdirectorios para organizar los archivos. Las estructuras de directorio en forma de grafo acíclico permiten a los usuarios compartir subdirectorios y archivos, pero complican las tareas de búsqueda y de borrado. Una

estructura de grafo general proporciona una flexibilidad completa en lo que a la compartición de archivos y directorios se refiere, pero en ocasiones requiere mecanismos de recolección de memoria para recuperar el espacio de disco no utilizado.

Los discos están segmentados en uno o más volúmenes, cada uno de los cuales contiene un sistema de archivos o puede dejarse "sin formato". Los sistemas de archivos pueden montarse en las estructuras de nombres del sistema, para que estén disponibles para los usuarios. El esquema de nombres varía de un sistema operativo a otro. Una vez montados, los archivos del volumen estarán disponibles para su uso. Los sistemas de archivos pueden desmontarse para impedir el acceso a los mismos o con propósitos de mantenimiento.

La compartición de archivos depende de la semántica proporcionada por el sistema. Los archivos pueden tener múltiples lectores, múltiples escritores o una serie de límites relativos a la compartición. Los sistemas de archivos distribuidos permiten que los hosts clientes monten volúmenes o directorios de los servidores, siempre y cuando puedan comunicarse entre sí a través de la red. Los sistemas de archivos remotos presentan diversos desafíos en lo que respecta a la fiabilidad, a las prestaciones y a la seguridad. Los sistemas de información distribuidos mantienen información sobre los usuarios, sobre los hosts y sobre el acceso, de modo que los clientes y servidores puedan compartir información de estado para gestionar el uso y el acceso a los archivos.

Puesto que los archivos son el principal mecanismo de almacenamiento de información en la mayoría de los sistemas informáticos, es necesario proporcionar mecanismos de protección de los archivos. El acceso a los archivos puede controlarse separadamente para cada tipo de acceso: lectura, escritura, ejecución, adición, borrado, listado de directorio, etc. La protección de archivos puede proporcionarse mediante contraseñas, mediante listas de acceso o mediante otras técnicas.

Ejercicios

- 10.1 Considere un sistema de archivos en el que pueda borrarse un archivo y en el que pueda reclamarse su correspondiente espacio de disco mientras todavía existen enlaces a dicho archivo. ¿Qué problemas pueden surgir si se crea un nuevo archivo en el mismo área de almacenamiento o con el mismo nombre de ruta absoluto? ¿Cómo pueden evitarse estos problemas?
- 10.2 La tabla de archivos abiertos se utiliza para mantener información acerca de los archivos que están actualmente abiertos. ¿Debe el sistema operativo mantener una tabla separada para cada usuario o simplemente mantener una tabla que contenga referencias a los archivos a los que están accediendo actualmente todos los usuarios? Si dos programas o usuarios distintos están accediendo al mismo archivo, ¿deberían existir entradas separadas en la tabla de archivos abiertos?
- 10.3 ¿Cuáles son las ventajas y desventajas de un sistema que proporcione bloqueos obligatorios en lugar de bloqueos sugeridos, cuyo uso se deja a la discreción del usuario?
- 10.4 ¿Cuáles son las ventajas y desventajas de registrar el nombre del programa creador junto con los atributos del archivo (como hace, el sistema operativo Macintosh)?
- 10.5 Algunos sistemas abren automáticamente un archivo cuando se hace referencia por primera vez al mismo, y cierran el archivo cuando el trabajo correspondiente termina. Explique las ventajas y desventajas de este esquema, comparado con el esquema más tradicional en el que el usuario tiene que abrir y cerrar el archivo explícitamente.
- 10.6 Si el sistema operativo supiera que una cierta aplicación va a acceder a los datos de un archivo de forma secuencial, ¿cómo podría aprovechar esta información para aumentar las prestaciones?
- 10.7 Proporcione un ejemplo de aplicación que podría beneficiarse de que el sistema operativo proporcionara soporte para el acceso aleatorio a archivos indexados.

- 10.8 Explique las ventajas y desventajas de soportar enlaces a archivos que crucen los puntos de montaje (es decir, el enlace hace referencia a un archivo que está almacenado en un volumen distinto).
- 10.9 Algunos sistemas proporcionan mecanismos de compartición de archivos manteniendo una única copia del archivo; otros sistemas mantienen varias copias, una para cada uno de los usuarios que están compartiendo el archivo. Explique las ventajas relativas de cada una de estas técnicas.
- 10.10 Explique las ventajas y desventajas de asociar con los sistemas de archivos remotos (almacenados en servidores de archivos) una semántica de fallo distinta de la que se asocia con los sistemas de archivos locales.
- 10.11 ¿Cuáles son las implicaciones de soportar la semántica de coherencia de UNIX para el acceso compartido a aquellos archivos que estén almacenados en sistemas de archivos remotos?

Notas bibliográficas

Un análisis general de los sistemas de archivos es el que se incluye en Grosshans [1986]. Golden y Pechura [1986] describieron la estructura de los sistemas de archivos para microcomputadoras. Los sistemas de base de datos y sus estructuras de archivos se describen en detalle en Silberschatz et al. [2001].

La primera implementación de una estructura de directorio multinivel es la del sistema MULTICS (Organick [1972]). La mayoría de los sistemas operativos implementan ahora estructuras de directorio multinivel. Entre ellos se incluye Linux (Bovet y Cesati [2002]), Mac OS X (<http://www.apple.com/macosx/>), Solaris (Mauro y McDougall [2001]) y todas las versiones de Windows, incluyendo Windows 2000 (Solomon y Russinovich [2000]).

El sistema de archivos en red (NFS), diseñado por Sun Microsystems, permite distribuir las estructuras de directorio entre una serie de sistemas informáticos conectados por red. NFS se describe en detalle en el Capítulo 17, mientras que NFS versión 4 se describe en RFC3505 (<http://www.ietf.org/rfc/rfc3505.txt>).

DNS fue propuesto por primera vez por Su [1982] y ha sufrido varias revisiones desde entonces, habiendo Mockapetris [1987] añadido varias funcionalidades principales. Eastlake [1999] ha propuesto una serie de extensiones de seguridad para que DNS pueda almacenar claves de seguridad.

LDAP, también conocido como X.509, es un subconjunto derivado del protocolo de directorio distribuido X.500. Fue definido por Yeong et al. [1995] y ha sido implementado en muchos sistemas operativos.

Hay diversas investigaciones interesantes en marcha en el área de las interfaces en los sistemas de archivos y, en particular, sobre las cuestiones relativas a los atributos y a la denominación de los archivos. Por ejemplo, el sistema operativo Plan 9 de Bell Laboratories (Lucent Technology) hace que todos los objetos parezcan sistemas de archivos. Así, para mostrar una lista de los procesos del sistema, el usuario simplemente tiene que listar el contenido del directorio `/proc`. De forma similar, para mostrar la hora del día, el usuario sólo necesita escribir el nombre de archivo `/dev/time`.

Implementación de sistemas de archivos

Como hemos visto en el Capítulo 10, el sistema de archivos proporciona el mecanismo para el almacenamiento en línea y para el acceso a los contenidos de los archivos, incluyendo datos y programas. El sistema de archivos reside permanentemente en *almacenamiento secundario*, que está diseñado para albergar de manera permanente una gran cantidad de datos. Este capítulo se ocupa principalmente de los temas relativos al almacenamiento de archivos y al acceso a archivos en el medio más común de almacenamiento secundario, que es el disco. Exploraremos una serie de formas para estructurar el uso de los archivos, para asignar el espacio del disco, para recuperar el espacio liberado, para controlar la ubicación de los datos y para realizar la interfaz entre otras partes del sistema operativo y el almacenamiento secundario. A lo largo de todo el capítulo, prestaremos especial atención a las cuestiones de rendimiento.

OBJETIVOS DEL CAPÍTULO

- Describir los detalles de implementación de sistemas de archivos locales y estructuras de directorio.
- Describir la implementación de sistemas de archivos remotos.
- Explicar los algoritmos de asignación de bloques y de control de bloques libres, así como los compromisos inherentes a ellos.

11.1 Estructura de un sistema de archivos

Los discos constituyen el principal tipo de almacenamiento secundario para mantener sistemas de archivos. Tienen dos características que los convierten en un medio conveniente para el almacenamiento de múltiples archivos:

1. El disco puede ser reescrito de manera directa; es posible leer un bloque del disco, modificar el bloque y volverlo a escribir en el mismo lugar.
2. Con un disco, se puede acceder directamente a cualquier bloque de información que contenga. Por tanto, resulta simple acceder a cualquier archivo de forma secuencial aleatoria y el comutar de un archivo a otro sólo requiere mover los cabezales de lectura-escritura y esperar a que el disco termine de rotar.

Analizaremos la estructura de los discos con más detalle en el Capítulo 12.

En lugar de transferir un byte cada vez, las transferencias de E/S entre la memoria y el disco se realizan en unidades de *bloques*, para mejorar la eficiencia de E/S. Cada bloque tiene uno o más sectores. Dependiendo de la unidad de disco, los sectores varían entre 32 bytes y 4096 bytes; usualmente su tamaño es de 512 bytes.

Para proporcionar un acceso eficiente y cómodo al disco, el sistema operativo impone uno o más **sistemas de archivos**, con los que los datos pueden almacenarse, localizarse y extraerse fácilmente. Un sistema de archivos acarrea dos problemas de diseño bastante diferentes. El primer problema es definir qué aspecto debe tener el sistema de archivos para el usuario. Esta tarea implica definir un archivo y sus atributos, las operaciones permitidas sobre los archivos y la estructura de directorio utilizada para organizar los archivos. El segundo problema es crear algoritmos y estructuras de datos que permitan mapear el sistema lógico de archivos sobre los dispositivos físicos de almacenamiento secundario.

El propio sistema de archivos está compuesto, generalmente, de muchos niveles diferentes. La estructura que se muestra en la Figura 11.1 es un ejemplo de diseño en niveles; cada nivel del diseño utiliza las funciones de los niveles inferiores para crear nuevas funciones que serán, a su vez, utilizadas por los niveles superiores a ese.

El nivel más bajo, el *control de E/S*, está compuesto por **controladores de dispositivo** y rutinas de tratamiento de interrupción, que se encargan de transferir la información entre la memoria principal y el sistema de disco. Un controlador de dispositivo puede considerarse una especie de traductor: su entrada está compuesta por comandos de alto nivel tales como "extraer bloque 123"; su salida consta de instrucciones de bajo nivel específicas del hardware que son utilizadas por la controladora hardware, que es quien establece la interfaz del dispositivo de E/S con el resto del sistema. El controlador de dispositivo escribe usualmente una serie de patrones de bits específicos en ubicaciones especiales de la memoria de la controladora de E/S, para decir a la controladora en qué ubicación del dispositivo debe operar y que acción debe llevar a cabo. Los detalles acerca de los controladores de dispositivo de la infraestructura de E/S se cubren en el Capítulo 13.

El **sistema básico de archivos** sólo necesita enviar comandos genéricos al controlador de dispositivo apropiado, con el fin de leer y escribir bloques físicos en el disco. Cada bloque físico se identifica mediante su dirección de disco numérica (por ejemplo, unidad 1, cilindro 73, pista 2, sector 10).

El **módulo de organización de archivos** tiene conocimiento acerca de los archivos y de sus bloques lógicos, así como de sus bloques físicos. Conociendo el tipo de asignación de archivos utilizado y la ubicación del archivo, el módulo de organización de archivos puede traducir las direcciones lógicas de bloque a direcciones físicas de bloque, que serán las que envíe al sistema básico de archivos para que realice las necesarias transferencias. Los bloques lógicos de cada archivo están numerados desde 0 (o 1) a N . Puesto que los bloques físicos que contienen los datos no suelen corresponderse con los números lógicos de bloque, es necesaria una tarea de traducción para localizar cada bloque. El módulo de organización de archivos incluye también el gestor de espacio libre, que controla los bloques no asignados y proporciona dichos bloques al módulo de organización de archivos cuando así se solicita.

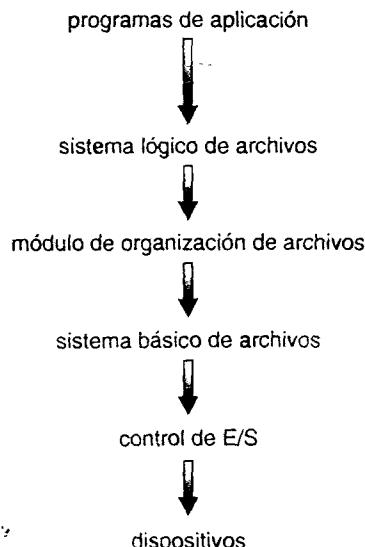


Figura 11.1 Sistema de archivos en niveles.

Finalmente, el **sistema lógico de archivos** gestiona la información de metadatos. Los metadatos incluyen toda la estructura del sistema de archivos, excepto los propios *datos* (es decir, el propio contenido de los archivos). El sistema lógico de archivos gestiona la estructura de directorio para proporcionar al módulo de organización de archivos la información que éste necesita, a partir de un nombre de archivo simbólico. Este nivel mantiene la estructura de los archivos mediante bloques de control de archivo. Un **bloque de control de archivo** (FCB, file-control block) contiene información acerca del archivo, incluyendo su propietario, los permisos y la ubicación del contenido del archivo. El sistema lógico de archivos también es responsable de las tareas de protección y seguridad, que ya hemos analizado en el Capítulo 10 y que analizaremos con más profundidad en el Capítulo 14.

Cuando se utiliza una estructura de niveles para la implementación de un sistema de archivos, se minimiza la duplicación de código. El código correspondiente al control de E/S y, en ocasiones, al sistema básico de archivos puede ser utilizado por múltiples sistemas de archivos. Cada sistema de archivos puede tener entonces su propio sistema lógico de archivos y su propio módulo de organización de archivos.

Hoy en día se utilizan muchos sistemas de archivos distintos. La mayoría de los sistemas operativos soportan más de un sistema. Por ejemplo, la mayoría de los CD-ROM están escritos en el formato ISO 9660, que es un formato estándar adoptado por consenso de los fabricantes de CD-ROM. Además de los sistemas de archivos para soportes extraíbles, cada sistema operativo tiene un (o más de un) sistema de archivos basado en disco. UNIX utiliza el **sistema de archivos UNIX** (UFS, UNIX file system), que está basado en el sistema FFS (Fast File System) de Berkeley. Windows NT, 2000 y XP soportan los formatos de sistema de archivos de disco FAT, FAT32 y NTFS (Windows NT File System), además de formatos de sistemas de archivos para CD-ROM, DVD y disquete. Aunque Linux soporta más de cuarenta sistemas de archivos distintos, el sistema de archivos estándar en Linux se denomina **sistema de archivos extendido**, siendo las versiones más comunes ext2 y ext3. También existen sistemas de archivos distribuidos, en los que un sistema de archivos que reside en un servidor puede montarse en uno o más clientes.

11.2 Implementación de sistemas de archivos

Como hemos descrito en la Sección 10.1.2, los sistemas operativos implementan llamadas al sistema `open()` y `close()` para que los procesos soliciten acceder al contenido de los archivos. En esta sección, vamos a profundizar en las estructuras y operaciones utilizadas para implementar las acciones relativas a los sistemas de archivos.

11.2.1 Introducción

Se utilizan varias estructuras en disco y en memoria para implementar un sistema de archivos. Estas estructuras varían dependiendo del sistema operativo y del sistema de archivos, aunque hay algunos principios básicos que son de aplicación general.

En el disco, el sistema de archivos puede contener información acerca de cómo iniciar un sistema operativo que esté almacenado allí, acerca del número total de bloques, del número y la ubicación de los bloques libres, de la estructura de directorios y de los archivos individuales. En el resto del capítulo detallaremos muchas de estas estructuras, pero vamos a describirlas aquí brevemente:

- Un **bloque de control de arranque** (por cada volumen) puede contener la información que el sistema necesita para iniciar un sistema operativo a partir de dicho volumen. Si el disco no contiene un sistema operativo, este bloque puede estar vacío. Normalmente, es el primer bloque del volumen. En UFS, se denomina **bloque de inicio**. En NTFS, se denomina **sector de arranque de la partición**.
- Un **bloque de control de volumen** (por cada volumen) contiene detalles acerca del volumen (o partición), tales como el número de bloques que hay en la partición, el tamaño de los bloques, el número de bloques libres y los punteros de bloques libres, así como un contador de

bloques de información FCB libres y punteros FCB. En UFS, esto se denomina **superbloque**, en NFTS, esta información se almacena en la **tabla maestra de archivos**.

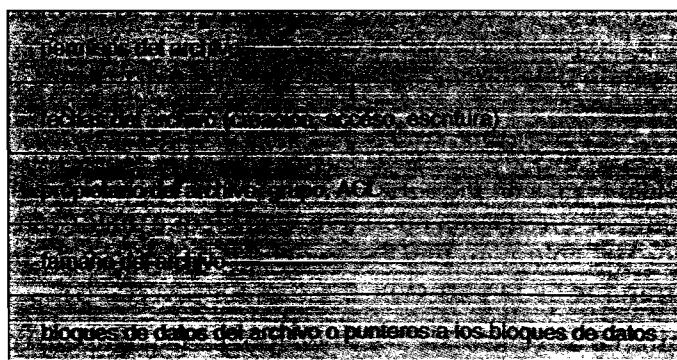
- Se utiliza una estructura de directorios por cada sistema de archivos para realizar los archivos. En UFS, esto incluye los nombres de los archivos y los nombres de **inodo** asociados. En NFTS, esta información se almacena en la **tabla maestra de archivos**.
- Un bloque FTB por cada archivo contiene numerosos detalles acerca del archivo, incluyendo los permisos correspondientes, el propietario, el tamaño y la ubicación de los bloques de datos. En UFS, esta estructura se denomina inodo. En NFTS, esta información se almacena dentro de la tabla maestra de archivos, que utiliza una estructura de base de datos relacional, con una fila por cada archivo.

La información almacenada en memoria se utiliza tanto para la gestión de un sistema de archivos como para la mejora del rendimiento mediante mecanismos de caché. Los datos se cargan en el momento del montaje y se descartan cuando el dispositivo se desmonta. Las estructuras existentes pueden incluir las que a continuación se describen:

- Una **tabla de montaje** en memoria contiene información acerca de cada volumen montado.
- Una **caché** de la estructura de directorios en memoria almacena la información relativa a los directorios a los que se ha accedido recientemente (para los directorios en los que hay que montar los volúmenes, puede contener un puntero a la tabla del volumen).
- La **tabla global de archivos abiertos** contiene una copia del FCB de cada archivo abierto, además de otras informaciones.
- La **tabla de archivos abiertos de cada proceso** contiene un puntero a la entrada apropiada de la entrada global de archivos abiertos, así como otras informaciones adicionales.

Para crear un nuevo archivo, un programa de aplicación llama al sistema lógico de archivos. El sistema lógico de archivos conoce el formato de las estructuras de directorio y, para crear un nuevo archivo, asigna un nuevo FCB (alternativamente, si la implementación del sistema de archivos crea todos los bloques FCB en el momento de creación del sistema de archivos, se asignará un FCB a partir del conjunto de bloques FCB libres). El sistema lee entonces el directorio apropiado en la memoria, lo actualiza con el nuevo nombre de archivo y el nuevo FCB y lo vuelve a escribir en el disco. En la Figura 11.2 se muestra un FCB típico.

Algunos sistemas operativos, incluyendo UNIX, tratan a los directorios exactamente igual que a los archivos, salvo porque se tratará de un archivo cuyo campo de tipo indicará que es un directorio. Otros sistemas operativos, incluyendo Windows NT, poseen llamadas al sistema diferentes para los archivos y los directorios, y tratan a los directorios como entidades distintas de los archivos. Independientemente de estas consideraciones estructurales de mayor nivel, el sistema lógico de archivos puede llamar al módulo de organización de archivos para mapear las operaciones de E/S de directorio sobre los números de bloque del disco, que se pasarán a su vez al sistema básico de archivos del sistema de control de E/S.



Ahora que hemos creado un archivo, podemos utilizarlo para E/S. Pero primero, sin embargo, es necesario *abrirlo*. La llamada `open()` pasa un nombre de archivo al sistema de archivos. La llamada al sistema `open()` primero busca en la tabla global de archivos abiertos para ver si el archivo está siendo ya utilizado por otro proceso. En caso afirmativo, se crea una entrada en la tabla de archivos abiertos del proceso que apunte a la tabla global de archivos abiertos existente. Este algoritmo puede ahorrarnos una gran cantidad de trabajo. Cuando un archivo está abierto, se busca en la estructura de directorios para encontrar el nombre de archivo indicado. Usualmente, se almacenan parte de la estructura de directorio en una caché de memoria para acelerar las operaciones de directorio. Una vez encontrado el archivo, el FCB se copia en la tabla global de archivos abiertos existente en la memoria. Esta tabla no sólo almacena el FCB, sino que también controla el número de procesos que han abierto el archivo.

A continuación, se crea una entrada en la tabla de archivos abiertos del proceso, con un puntero a la entrada de la tabla global de archivos abiertos y algunos otros campos de información. Estos otros campos de información pueden incluir un puntero a la ubicación actual dentro del archivo (para la siguiente operación `read()` o `write()`) y el modo de acceso en que se ha abierto el archivo. La llamada `open()` devuelve un puntero a la entrada apropiada de la tabla de archivos abiertos del proceso; todas las operaciones de archivo se realizan a partir de ahí mediante este puntero. El nombre del archivo puede no formar parte de la tabla de archivos abiertos, ya que el sistema no lo utiliza para nada una vez que se ha localizado el FCB apropiado en el disco. Sin embargo, se lo puede almacenar en caché, para ahorrar tiempo en las subsiguientes aperturas del mismo archivo. El nombre que se proporciona a esas entradas varía de unos sistemas a otros. En los sistemas UNIX se utiliza el término inglés **file descriptor**, mientras que Windows prefiere el término **file handle**; en castellano, utilizamos el término **descriptor de archivo**. En consecuencia, hasta que se cierre el archivo, todas las operaciones con el archivo se llevan a cabo mediante el descriptor de archivo contenido en la tabla de archivos abierto.

Cuando un proceso cierra el archivo, se elimina la entrada de la tabla del proceso y se decrementa el contador de aperturas existente en la tabla global. Cuando todos los usuarios que hayan abierto el archivo lo cierren, los metadatos actualizados se copian en la estructura de directorio residente en el disco y se elimina la entrada de la tabla global de archivos abiertos.

Algunos sistemas complican este esquema todavía más, utilizando el sistema de archivos como interfaz para otros aspectos del sistema, como por ejemplo la comunicación por red. Como ilustración, en UFS, la tabla global de archivos abiertos almacena los inodos y otras informaciones para los archivos y directorios, pero también alberga información similar para las conexiones de red y para los dispositivos. De esta forma, puede usarse un mismo mecanismo para múltiples propósitos.

No deben infravalorarse los aspectos de almacenamiento en caché de las estructuras de los sistemas de archivos. La mayoría de los sistemas mantienen en la memoria toda la información acerca de un archivo abierto, salvo los propios bloques de datos. El sistema BSD UNIX resulta bastante típico en su uso de cachés para tratar de ahorrar operaciones de E/S de disco. Su tasa media de aciertos de caché, que es del 85 por ciento, muestra que merece la pena implementar estas técnicas. El sistema BSD UNIX se describe en detalle en el Apéndice A.

La Figura 11.3 resume las estructuras de operaciones utilizadas en la implementación de un sistema de archivos.

11.2.2 Particiones y montaje

La disposición de la información en un disco puede tener múltiples variantes, dependiendo del sistema operativo. Un disco puede dividirse en múltiples particiones, o bien un volumen puede abarcar múltiples particiones en múltiples discos. Analizaremos aquí el primero de esos dos tipos de disposición, mientras que del segundo, que puede en realidad considerarse una forma de tecnología RAID, se analizará en la Sección 12.7.

Cada partición puede ser una partición “en bruto” (sin formato), que no contenga ningún sistema de archivos, o una partición “procesada”, que sí contenga un sistema de archivos. Los discos sin formato se utilizan en aquellos casos en que no resulte apropiado emplear un sistema de

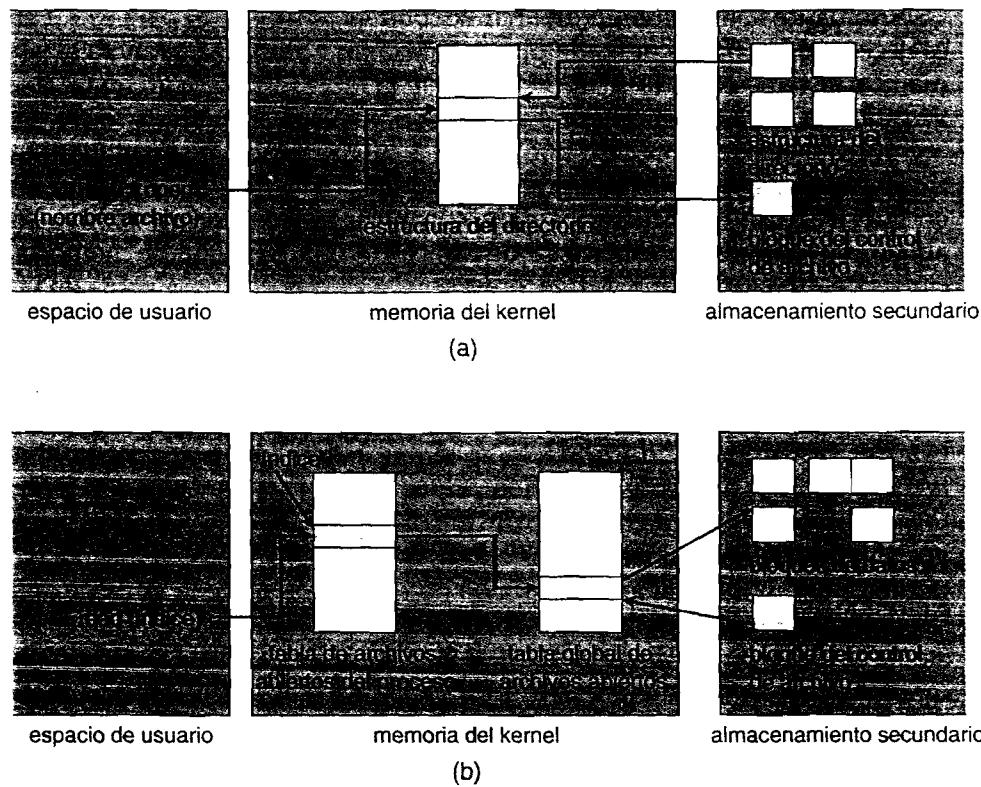


Figura 11.3 Estructuras en memoria de un sistema de archivos. (a) Apertura de un archivo.
(b) Lectura de un archivo.

archivos. Por ejemplo, el espacio de intercambio en UNIX puede utilizar una partición sin formato, ya que ese espacio de intercambio usa su propio formato en el disco y no emplea un sistema de archivos. De la misma forma, algunas bases de datos utilizan discos sin formato y formatean los datos de la forma que mejor se adapte a sus necesidades. Los discos sin formato también pueden almacenar la información que necesitan los sistemas de discos RAID, como los mapas de bits que indican qué bloques están ubicados en espejo y cuáles han sido modificados y necesitan ser duplicados. De forma similar, los discos sin formato pueden contener una base de datos en miniatura que almacene la información de configuración RAID, como por ejemplo la información que indique qué discos son miembro de cada conjunto RAID. El uso de discos sin formato se analiza con más detalle en la Sección 12.5.1.

La información de arranque puede almacenarse en una partición independiente. De nuevo, esa partición tiene su propio formato, porque en el momento del inicio el sistema no ha cargado aún los controladores de dispositivo para los sistemas de archivos y no puede, por tanto, interpretar el formato de ningún sistema de archivos. En lugar de ello, la información de arranque es, usualmente, una serie secuencial de bloques, que se cargan como una imagen binaria en la memoria. La ejecución de esa imagen binaria comienza en una ubicación predefinida, como por ejemplo el primer byte. Esta imagen de arranque puede contener más información, a parte de las instrucciones relativas a cómo iniciar un sistema operativo específico. Por ejemplo, en un PC y en otros sistemas, la máquina puede ser de **arranque dual**, pudiendo instalarse a múltiples sistemas operativos en dichos tipos de sistemas. ¿Cómo sabe el sistema cuál de los sistemas operativos arrancar? Podemos almacenar en el espacio de arranque un cargador de arranque que sea capaz de comprender múltiples sistemas de archivos y múltiples sistemas operativos; una vez cargado, ese cargador de arranque puede iniciar uno de los sistemas operativos disponibles en el disco. El disco puede tener múltiples particiones, cada una de las cuales contendrá un tipo diferente de sistema de archivos y un sistema operativo distinto.

La **partición raíz**, que contiene el *kernel* del sistema operativo y, en ocasiones, otros archivos del sistema, se monta en el momento del arranque. Otros volúmenes pueden montarse automáti-

camente durante el arranque o pueden ser montados manualmente en algún momento posterior, dependiendo del sistema operativo. Como parte de una operación de montaje, el sistema operativo verifica que el dispositivo contenga un sistema de archivos válido. Para realizar esa verificación, el sistema operativo pide al controlador del dispositivo que lea el directorio del dispositivo y verifique que éste directorio tiene el formato esperado. Si el formato no es válido, será necesario comprobar y posiblemente corregir la coherencia de la partición, con o sin intervención del usuario. Finalmente, el sistema operativo registrará en su **tabla de montaje** de la memoria que se ha montado un sistema de archivos, indicando el tipo de sistema de archivos de que se trata. Los detalles de esta función dependerán del sistema operativo. Los sistemas Microsoft Windows montan cada volumen en un espacio de nombres separado, que se designa mediante una letra y un carácter de dos puntos. Por ejemplo, para registrar que un sistema de archivos está montado en F:, el sistema operativo coloca un puntero al sistema de archivos dentro de un campo de la estructura de dispositivo correspondiente a F:. Cuando un proceso especifica dicha letra de unidad, el sistema operativo localiza el puntero apropiado al sistema de archivos y recorre las estructuras de directorio de dicho dispositivo hasta encontrar el archivo o directorio especificado. Las versiones más recientes de Windows pueden montar el sistema de archivos en cualquier punto dentro de la estructura de directorios existentes.

En UNIX, los sistemas de archivos pueden montarse en cualquier directorio. El montaje se implementa configurando un indicador dentro de la copia en memoria del inodo correspondiente a dicho directorio. Ese indicador atestigua que el directorio es un punto de montaje. Otro campo apunta a una entrada dentro de la tabla de montaje, que indica qué dispositivo está montado allí. La entrada de la tabla de montaje contiene un puntero al superbloque del sistema de archivos existente en dicho dispositivo. Este esquema permite que el sistema operativo recorra su estructura de directorios, conmutando transparentemente entre sistemas de archivos de diferentes tipos.

11.2.3 Sistemas de archivos virtuales

Queda claro, a partir de las explicaciones de la sección anterior, que los sistemas operativos modernos deben soportar concurrentemente múltiples tipos de sistemas de archivo. Pero, ¿cómo permite el sistema operativo integrar múltiples tipos de sistemas de archivos en una estructura de directorio? ¿Y cómo pueden los usuarios moverse transparentemente entre los distintos tipos de sistemas de archivos a medida que recorren el espacio de los sistemas de archivos. Vamos a analizar a continuación algunos de estos detalles de implementación.

Un método obvio, pero subóptimo, de implementar múltiples tipos de sistemas de archivos consiste en escribir rutinas diferentes de acceso a directorios y a archivos para cada uno de los tipos existentes. Sin embargo, en lugar de usar esta solución, la mayoría de los sistemas operativos, incluyendo UNIX, utilizan técnicas de orientación a objetos para simplificar, organizar y modularizar la implementación. El uso de estos métodos permite implementar tipos muy distintos de sistemas de archivos dentro de una misma estructura, incluyendo sistemas de archivos de red, como por ejemplo NFS. Los usuarios pueden acceder a archivos que estén contenidos dentro de múltiples sistemas de archivos en el disco local, o incluso en sistemas de archivos disponibles a través de la red.

Se utilizan estructuras de datos y procedimientos para aislar la funcionalidad básica de llamadas al sistema de los detalles de implementación. Así, la implementación del sistema de archivos está compuesta de tres niveles principales, como se muestra esquemáticamente en la Figura 11.4. El primer nivel es la interfaz del sistema de archivos, basada en las llamadas `open()`, `read()`, `write()` y `close()` y en los descriptores de archivos.

El segundo nivel se denomina **sistema de archivos virtual** (VFS, virtual file system); este nivel cumple dos importantes funciones:

1. Separa las operaciones genéricas del sistema de archivos con respecto a su implementación, definiendo una interfaz VFS muy clara. Pueden coexistir diversas implementaciones de la interfaz VFS en la misma máquina, permitiendo un acceso transparente a diferentes tipos de sistemas de archivos montados de modo local.

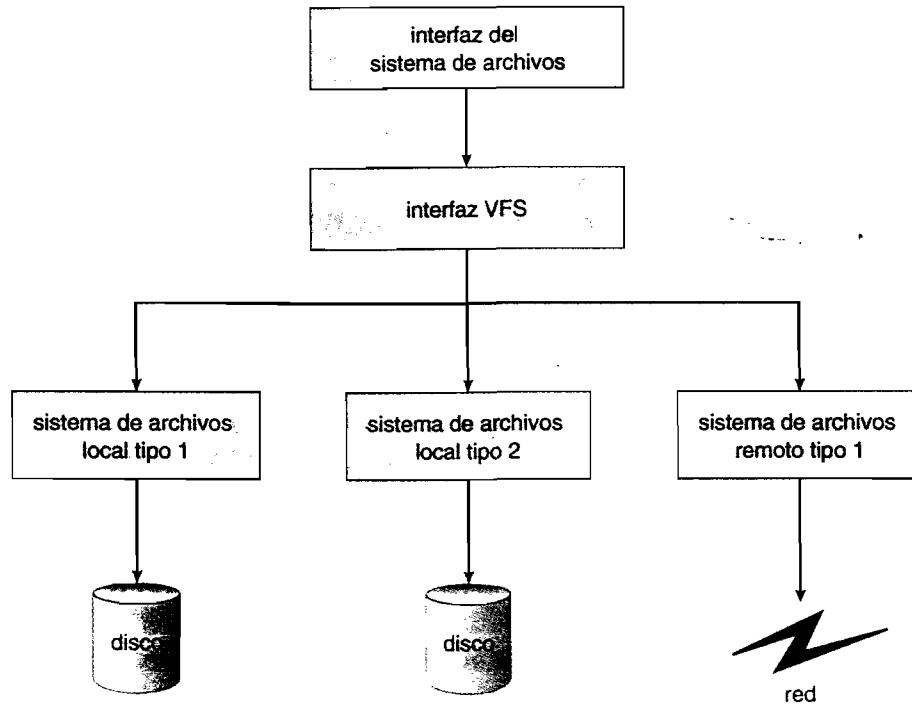


Figura 11.4 Vista esquemática de un sistema de archivos virtual.

2. El VFS proporciona un mecanismo para representar de forma coherente los archivos a través de una red. El VFS está basado en una estructura de representación de archivos denominada **vnode**, que contiene un designador numérico para los archivos únicamente identificables dentro de la red (los inodos de UNIX sólo son únicos dentro de un único sistema de archivos). Esta unicidad en el nivel de red es obligatoria para poder soportar los sistemas de archivos de red. El *kernel* mantiene una estructura de vnode para cada nodo activo (archivo o directorio).

Así, el VFS distingue entre los archivos locales y los remotos, y los archivos locales se distinguen entre sí de acuerdo con el tipo de sistema de archivos.

El VFS activa operaciones específicas del sistema de archivos para poder satisfacer las solicitudes locales de acuerdo con el tipo de sistema de archivos correspondiente, y también se encarga de invocar los procedimientos del protocolo NFS para satisfacer las solicitudes remotas. Se construyen descriptores de archivo a partir de los vnodes relevantes y esos descriptores se pasan como argumentos a dichos procedimientos. El nivel que implementa el tipo de sistema de archivos o el protocolo del sistema de archivos remoto constituye el tercer nivel de la arquitectura.

Examinemos brevemente la arquitectura VFS en Linux. Los cuatro principales tipos de objeto definidos por el VFS de Linux son:

- El **objeto inodo**, que representa un archivo individual.
- El **objeto archivo**, que representa un archivo abierto.
- El **objeto superbloque**, que representa un sistema de archivos completo.
- El **objeto entrada de directorio**, que representa una entrada de directorio individual.

Para cada uno de estos cuatro tipos de objeto, el VFS define un conjunto de operaciones que hay que implementar. Cada objeto de uno de estos tipos contiene un puntero a una tabla de funciones. La tabla de funciones indica las direcciones de las funciones reales que implementan las operaciones definidas para ese objeto concreto. Por ejemplo, una API abreviada para algunas de las operaciones para los objetos archivo incluiría:

- int open(. . .) — Abre un archivo.

- `ssize_t read(...)`—Lee de un archivo.
- `ssize_t write(...)`—Escribe en un archivo.
- `int mmap(...)`—Mapea en memoria un archivo.

Cada implementación del objeto archivo para un tipo de archivo específico necesita implementar cada una de las funciones especificadas en la definición del objeto archivo (la definición completa del objeto archivo se especifica en la estructura `file_operations`, que está ubicada en el archivo `/usr/include/linux/fs.h`).

Así, el nivel de software VFS puede realizar una operación sobre uno de estos objetos invocando la función apropiada a partir de la tabla de funciones del objeto, sin tener que conocer de antemano exactamente el tipo de objeto con el que está tratando. El VFS no conoce, ni tampoco le importa, si un inodo representa un archivo de disco, un archivo de directorio o un archivo remoto. La función apropiada para la operación `read()` de dicho archivo se encontrará siempre en el mismo lugar dentro de su tabla de funciones, y el nivel de software VFS invocará dicha función sin preocuparse de cómo se lean en la práctica los datos.

11.3 Implementación de directorios

La selección de los algoritmos de asignación de directorios y gestión de directorios afecta significativamente a la eficiencia, las prestaciones y la fiabilidad del sistema de archivos. En esta sección, vamos a ver los compromisos existentes a la hora de seleccionar uno de estos algoritmos.

11.3.1 Lista lineal

El método más simple para implementar un directorio consiste en utilizar una lista lineal de nombres de archivos, con punteros a los bloques de datos. Este método es simple de programar, pero requiere mucho tiempo de ejecución. Para crear un nuevo archivo, debemos primero explorar el directorio para asegurarnos de que no haya ningún archivo existente con el mismo nombre. Después, añadiremos una nueva entrada al final del directorio. Para borrar un archivo, exploraremos el directorio en busca del archivo especificado y liberaremos el espacio asignado al mismo. Para reutilizar la entrada del directorio, podemos hacer varias cosas: podemos marcar la entrada como no utilizada (asignándola un nombre especial, como por ejemplo un nombre completamente en blanco o utilizando un bit usado-libre para cada entrada), o podemos insertarla en una lista de entradas libres de directorio. Una tercera alternativa consiste en copiar la última entrada del directorio en la ubicación que ha quedado libre y reducir la longitud del directorio. También puede utilizarse una lista enlazada para reducir el tiempo requerido para borrar un archivo.

La principal desventaja de una lista lineal de entradas del directorio es que, para localizar un archivo, se requiere realizar una búsqueda lineal. La información de directorio se utiliza frecuentemente y los usuarios notarán inmediatamente que el acceso a esa información es muy lento. De hecho, muchos sistemas operativos implementan una caché software para almacenar la información de directorio más recientemente utilizada. Cada acierto de caché evita la necesidad de volver a ver constantemente la información del disco. Una lista ordenada permite efectuar una búsqueda ordinaria y reduce el tiempo medio de búsqueda; sin embargo, el requisito de mantener la lista ordenada puede complicar los procesos de creación y borrado de archivos, ya que puede que tengamos que mover cantidades sustanciales de información de directorio para poder mantener el directorio ordenado. En este caso, podría sernos de ayuda utilizar una estructura de datos en árbol más sofisticada, como por ejemplo un árbol-B. Una ventaja de la lista ordenada es que puede generarse un listado ordenado del directorio sin ejecutar un paso separado de ordenación.

11.3.2 Tabla hash

Otro tipo de estructura de datos utilizado para los directorios de archivos son las tablas *hash*. Con este método, se almacenan las entradas de directorio en una lista lineal, pero también se utiliza

una estructura de datos *hash*. La tabla *hash* toma un valor calculado a partir del nombre del archivo y devuelve un puntero a la ubicación de dicho nombre de archivo dentro de la lista lineal. Por tanto, puede reducir enormemente el tiempo de búsqueda en el directorio. La inserción y el borrado son también bastante sencillas, aunque es necesario tener en cuenta la posible aparición de colisiones, que son aquellas situaciones en las que dos nombres de archivo proporcionan, al aplicar la función *hash*, la misma ubicación dentro de la lista.

Las principales dificultades asociadas con las tablas *hash* son que su tamaño es, por regla general, fijo y que la función *hash* depende de dicho tamaño. Por ejemplo, suponga que implementamos una tabla *hash* de prueba lineal que tenga 64 entradas. La función *hash* convertirá los nombres de archivo a enteros comprendidos en el rango de 0 a 63, utilizando por ejemplo el resto de una división por 64. Si después tratamos de crear otro archivo más, el número 65, deberemos agrandar la tabla *hash* del directorio, por ejemplo a 128 entradas. Como resultado, necesitaremos una nueva función *hash* que mapee los nombres de archivo al rango comprendido entre 0 y 127, y deberemos reorganizar las entradas de directorio existentes para reflejar los nuevos valores proporcionados por la función *hash*.

Alternativamente, podemos usar una tabla *hash* con desbordamiento encadenada. Cada entrada *hash* puede ser una lista enlazada en lugar de un valor individual y podemos resolver las colisiones añadiendo la nueva entrada a esa lista enlazada. Este mecanismo puede ralentizar algo las búsquedas, ya que la búsqueda de un nombre requerirá recorrer una lista enlazada de todas las entradas de la tabla que se mapeen sobre el mismo valor *hash*. De todos modos, este método será normalmente mucho más rápido que una búsqueda lineal a través de todo el directorio.

11.4 Métodos de asignación

El acceso directo inherente a la naturaleza de los discos nos proporciona una gran flexibilidad a la hora de implementar los archivos. En casi todos los casos, habrá múltiples archivos almacenados en el mismo disco. El principal problema es cómo asignar el espacio a esos archivos de modo que el espacio de disco se utilice eficazmente y que se pueda acceder a los archivos de forma rápida. Hay tres métodos principales de asignación del espacio de disco que se utilizan de manera común: asignación contigua, enlazada e indexada. Cada método tiene sus ventajas y desventajas. Algunos sistemas (como el sistema RDOS para la línea NOVA de computadoras de Data General) soportan los tres métodos de asignación, pero lo más común es que cada sistema utilice un único método para todos los archivos correspondientes a un tipo de sistema de archivos específico.

11.4.1 Asignación contigua

La **asignación contigua** requiere que cada archivo ocupe un conjunto de bloques contiguos en el disco. Las direcciones de disco definen una ordenación lineal del disco. Con esta ordenación, suponiendo que sólo haya una tarea accediendo al disco, acceder al bloque $b + 1$ después del bloque b normalmente no requiere ningún movimiento de cabezal. Cuando hace falta un movimiento de cabezal (para pasar del último sector de un cilindro al primer sector del siguiente cilindro), el cabezal sólo necesitará moverse de una pista a la siguiente, por tanto, el número de reposicionamientos del cabezal del disco requeridos para acceder a los archivos que están asignados de modo contiguo es mínimo, al igual que lo es el tiempo de búsqueda cada vez que hace falta reposicionar los cabezales. El sistema operativo VM/CMS de IBM utiliza una asignación contigua, debiendo precisamente a las altas prestaciones que proporciona.

La asignación contigua de un archivo está definida por la asignación de disco del primer bloque y por la longitud del archivo (en unidades de bloques). Si el archivo tienen n bloques de longitud y comienza en la ubicación b , entonces ocupará los bloques $b, b + 1, b + 2, \dots, b + n - 1$. La entrada de directorio de cada archivo indicará la dirección del bloque de inicio y la longitud del área asignada al archivo (Figura 11.5).

Acceder a un archivo que haya sido asignado de manera contigua resulta sencillo. Para el acceso secuencial, el sistema de archivos recuerda la dirección de disco del último bloque al que se haya hecho referencia y leerá el siguiente bloque cuando sea necesario. Para el acceso directo al

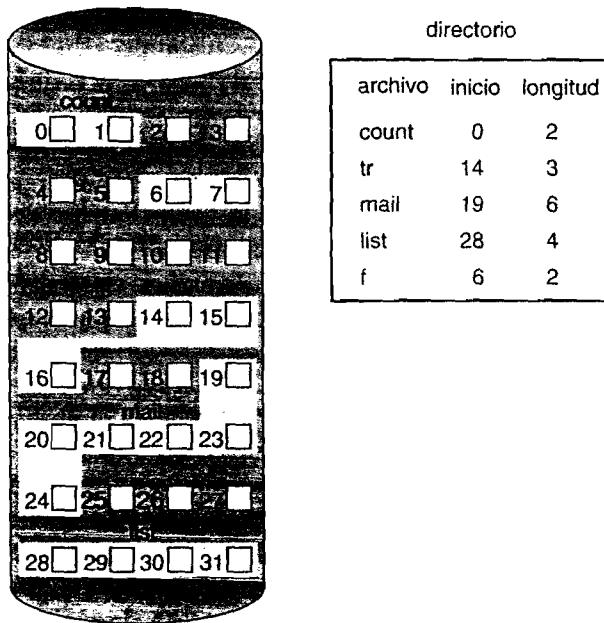


Figura 11.5 Asignación contigua del espacio de disco.

bloque i de un archivo que comience en el bloque b , podemos acceder inmediatamente al bloque $b + i$. Así, la asignación contigua permite soportar tanto el acceso directo como el secuencial.

Sin embargo, la asignación contigua también tiene sus problemas. Una de las dificultades estriba en encontrar espacio para un nuevo archivo. El sistema elegido para gestionar el espacio libre determinará como se lleve a cabo esta tarea. Esos sistemas de gestión se analizan en la Sección 11.5. Puede utilizarse cualquier sistema de gestión que se desee, aunque algunos son más lentos que otros.

El problema de la asignación contigua puede verse como un caso concreto del problema general de **asignación dinámica del almacenamiento** que hemos analizado en la Sección 8.3 y que se refiere a cómo satisfacer una solicitud de tamaño n a partir de una lista de huecos libres. Las estrategias más comunes para seleccionar un hueco libre a partir del conjunto de huecos disponibles son las de primer ajuste y mejor ajuste. Las simulaciones muestran que tanto la técnica de primer ajuste como la de mejor ajuste son más eficientes que la de peor ajuste tanto en términos de tiempo como en términos de utilización del almacenamiento. Las técnicas de primer ajuste y de mejor ajuste son muy similares en lo que se refiere a la utilización del espacio de almacenamiento, pero la técnica de primer ajuste es generalmente más rápida.

Todos estos algoritmos sufren el problema de la **fragmentación externa**. A medida que se asignan y borran archivos, el espacio libre del disco se descompone en pequeños fragmentos. Siempre que el espacio libre se descompone en fragmentos distintos aparece el problema de la fragmentación externa y este problema llegará a ser significativo cuando el fragmento contiguo más grande sea insuficiente para satisfacer una solicitud; el espacio de almacenamiento estará fragmentado en una serie de huecos, ninguno de los cuales será lo suficientemente grande como para almacenar los datos. Dependiendo de la cantidad total del espacio de almacenamiento en disco y del tamaño medio de los archivos, la fragmentación externa puede ser un problema grave o no demasiado importante.

Algunos sistemas antiguos para PC utilizaban un mecanismo de asignación contigua en los disquetes. Para impedir la pérdida de cantidades significativas de espacio de disco debido a la fragmentación externa, el usuario debía ejecutar una rutina de reempaquetamiento que copiara todo el sistema de archivos en otro disquete o en una cinta. Entonces, el disquete original se liberaba completamente, creando un único espacio libre contiguo de gran tamaño. La rutina copiaba entonces de nuevo los archivos en el disquete, asignando espacio contiguo a partir de este único hueco de gran tamaño. Este esquema **compacta** de manera efectiva todo el espacio libre en un

único espacio contiguo, resolviendo el problema de la fragmentación. El coste de este procedimiento de compactación es el tiempo. Ese tiempo adicional es particularmente grande para los discos duros de gran tamaño que utilicen mecanismos de asignación contigua, donde la compactación de todo el espacio puede requerir horas y puede ser necesario llevarla a cabo semanalmente. Algunos sistemas requieren que esta función se ejecute **fuera de línea**, teniendo el sistema de archivos desmontado. Durante este **tiempo de detención**, generalmente no puede permitirse continuar con la operación normal del sistema, por lo que dichas técnicas de compactación se evitan por todos los medios en las máquinas utilizadas en los sistemas de producción. La mayoría de los sistemas modernos que necesitan mecanismos de desfragmentación pueden realizar esa tarea **en línea** durante la operación normal del sistema, aunque la influencia sobre el rendimiento puede ser sustancial.

Otro problema de los mecanismos de asignación contigua es determinar cuánto espacio se necesita para un archivo. En el momento de crear el archivo, será necesario determinar la cantidad total de espacio que vamos a necesitar y asignar esa cantidad total. ¿Cómo sabe el creador (programa o persona) del archivo qué tamaño va a tener éste? En algunos casos, esta determinación puede ser bastante simple (por ejemplo, cuando se copia un archivo existente); sin embargo, en general, el tamaño de un archivo de salida puede ser difícil de estimar. Si asignamos demasiado poco espacio a un archivo, podemos encontrarnos con que el archivo no puede ampliarse. Especialmente con las estrategias de asignación de mejor ajuste, puede que el espacio situado a ambos lados del archivo ya esté siendo utilizado, por tanto, no podemos ampliar el tamaño del archivo sin desplazarlo a otro lugar. Entonces, existen dos posibilidades. La primera es terminar el programa de usuario, emitiendo el apropiado mensaje de error. El usuario deberá entonces asignar más espacio y volver a ejecutar el programa. Estas ejecuciones repetidas pueden ser muy costosas; para prevenirlas, el usuario sobreestimará normalmente la cantidad de espacio necesaria, lo que dará como resultado un desperdicio de espacio considerable. La otra posibilidad consiste en localizar un hueco de mayor tamaño, copiar el contenido del archivo al nuevo espacio y liberar el espacio anterior. Esta serie de acciones puede repetirse siempre y cuando exista suficiente espacio, aunque pueden consumir bastante tiempo. Sin embargo, con este método nunca será necesario informar explícitamente al usuario acerca de lo que está sucediendo; el sistema continuará operando a pesar del problema, aunque lo hará de forma cada vez más lenta.

Incluso si la cantidad total de espacio necesario para un archivo se conoce de antemano, el mecanismo de preasignación puede ser poco eficiente. A un archivo que crezca lentamente a lo largo de un período muy dilatado (meses o años) será necesario asignarle suficiente espacio para su tamaño final, aun cuando buena parte de ese espacio vaya a estar sin utilizar durante un largo tiempo. Por tanto, el archivo tendrá un alto grado de fragmentación interna.

Para minimizar estos problemas, algunos sistemas operativos utilizan un esquema modificado de asignación contigua, mediante el que se asigna inicialmente un bloque contiguo de espacio y, posteriormente, si dicho espacio resulta no ser lo suficientemente grande, se añade otro área de espacio contiguo, denominado **extensión**. La ubicación de los bloques de un archivo se registra entonces mediante una dirección y un número de bloques, junto con un enlace al primer bloque de la siguiente extensión. En algunos sistemas, el propietario del archivo puede establecer el tamaño de la extensión, pero este modo de configuración puede hacer que disminuya la eficiencia si el propietario realiza una estimación incorrecta. La fragmentación interna puede continuar siendo un problema si las extensiones son demasiado grandes y la fragmentación externa puede llegar a ser un problema a medida que se asignen y desasignen extensiones de tamaño variable. El sistema de archivos comercial Veritas utiliza las extensiones para optimizar el funcionamiento; se trata de un sustituto de altas prestaciones para el sistema UFS estándar de UNIX.

11.4.2 Asignación enlazada

El mecanismo de **asignación enlazada** resuelve todos los problemas de la asignación contigua. Con la asignación enlazada, cada archivo es una lista enlazada de bloques de disco, pudiendo estar dichos bloques dispersos por todo el disco. El directorio contiene un puntero al primer y al último bloque de cada archivo. Por ejemplo, un archivo de cinco bloques podría comenzar en el

bloque 9 y continuar en el bloque 16, luego en el 1, después en el bloque 10 y, finalmente, en el bloque 25 (Figura 11.6). Cada bloque contiene un puntero al bloque siguiente. Dichos punteros no están a disposición del usuario. Con este mecanismo, si cada bloque tiene 512 bytes de tamaño y una dirección de disco (el puntero) requiere 4 bytes, el usuario verá bloques de 508 bytes.

Para crear un nuevo archivo, simplemente creamos una nueva entrada en el directorio. Con el mecanismo de asignación enlazada, cada entrada de directorio tiene un puntero al primer bloque de disco del archivo. Este puntero se inicializa con el valor *nulo* (el valor de puntero que indica el fin de la lista) para representar un archivo vacío. El campo de tamaño también se configura con el valor 0. Una escritura en el archivo hará que el sistema de gestión de espacio libre localice un bloque libre y la información se escribirá en este nuevo bloque y será enlazada al final del archivo. Para leer un archivo, simplemente leemos los bloques siguiendo los punteros de un bloque a otro. No hay ninguna fragmentación externa con asignación enlazada y puede utilizarse cualquiera de los bloques de la lista de bloques libres para satisfacer una solicitud. No es necesario declarar el tamaño del archivo en el momento de crearlo y el archivo puede continuar creciendo mientras existan bloques libres. En consecuencia, nunca es necesario compactar el espacio de disco.

Sin embargo, el mecanismo de asignación enlazada también tiene sus ventajas. El principal problema es que sólo se lo puede utilizar de manera efectiva para los archivos de acceso secuencial. Para encontrar el bloque *i*-ésimo de un archivo, podemos comenzar al principio de dicho archivo y seguir los punteros hasta llegar al bloque *i*-ésimo. Cada acceso a un puntero requiere una lectura de disco y algunos de ellos requerirán un reposicionamiento de cabezales. En consecuencia, resulta muy poco eficiente tratar de soportar una funcionalidad de acceso directo para los archivos de asignación enlazada.

Otra desventaja es el espacio requerido para los punteros. Si un puntero ocupa 4 bytes de un bloque de 512 bytes, el 0,758 por ciento del espacio del disco se estará utilizando para los punteros, en lugar de para almacenar información útil. Cada archivo requerirá un poco más de espacio de lo que requeriría con otros mecanismos.

La solución usual para este problema consiste en consolidar los bloques en grupos, denominados *clusters*, y asignar *clusters* en lugar de bloques. Por ejemplo, el sistema de archivos puede definir un *cluster* como cuatro bloques y operar con el disco sólo en unidades de *cluster*. Los punteros utilizarán entonces un porcentaje mucho más pequeño del espacio de disco del archivo. Este método permite que el mapeo entre bloques lógicos y físicos siga siendo simple, pero mejora la tasa de transferencia del disco (porque hacen falta menos reposicionamientos de los cabezales) y reduce el espacio necesario para la asignación de bloques y para la gestión de la lista de bloques libres. El coste asociado con esta técnica es un incremento en el grado de fragmentación interna,

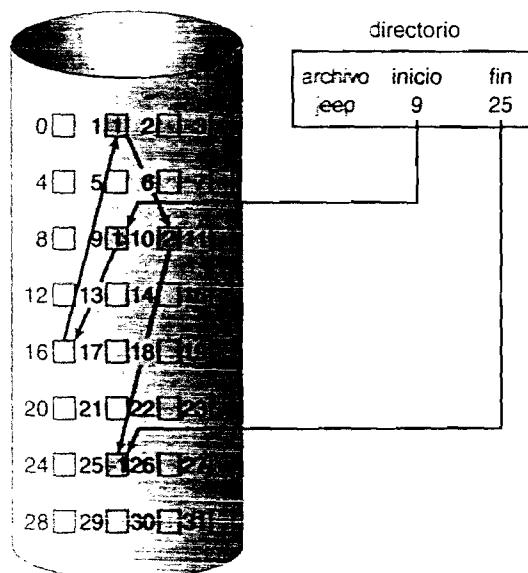


Figura 11.6 Asignación enlazada del espacio de disco.

porque ahora se desperdiciará más espacio cuando un *cluster* esté parcialmente lleno de lo que se desperdiciaba cuando sólo era un bloque lo que estaba parcialmente lleno. Los *clusters* pueden también utilizarse para mejorar el tiempo de acceso a disco para muchos otros algoritmos, así que se los utiliza en la mayoría de los sistemas de archivos.

Otro problema de la asignación enlazada es la fiabilidad. Recuerde que los archivos están enlazados mediante punteros que están dispersos por todo el disco; considere ahora lo que sucedería si uno de esos punteros se pierde o resulta dañado. Un error en el software del sistema operativo o un fallo del hardware del disco podrían hacer que se interpretara incorrectamente un puntero; este error, a su vez, podría hacer que un enlace apuntara a la lista de bloques libres o a otro archivo. Una solución parcial a este problema consiste en utilizar listas de elementos enlazadas, mientras que otra consiste en almacenar el nombre del archivo y el número de bloque relativo en cada bloque; sin embargo, estos esquemas requieren desperdiciar todavía más espacio en cada archivo.

Una variante importante del mecanismo de asignación enlazada es la que se basa en el uso de una **tabla de asignación de archivos** (FAT, file-allocation table). Este método simple, pero eficiente, de asignación del espacio del disco se utiliza en los sistemas operativos MS-DOS y OS/2. Una sección del disco al principio de cada volumen se reserva para almacenar esa tabla, que tiene una entrada por cada bloque del disco y está indexada según el número de bloque. La FAT se utiliza de forma bastante similar a una lista enlazada. Cada entrada de directorio contiene el número de bloque del primer bloque del archivo. La entrada de la tabla indexada según ese número de bloque contiene el número de bloque del siguiente bloque del archivo. Esta cadena continua hasta el último bloque, que tiene un valor especial de fin de archivo como entrada de la tabla. Los bloques no utilizados se indican mediante un valor de tabla igual a 0. Para asignar un nuevo bloque a un archivo, basta con encontrar la primera entrada de la tabla con valor 0 y sustituir el anterior valor de fin de archivo por la dirección del nuevo bloque. Después, el 0 se sustituye por el valor de fin de archivo. Un ejemplo ilustrativo sería la estructura FAT mostrada en la Figura 11.7, para un archivo compuesto por los bloques de disco 217, 618 y 339.

El esquema de asignación FAT puede provocar un número significativo de reposicionamiento de los cabezales del disco, a menos que la tabla FAT se almacene en caché. El cabezal del disco debe moverse al principio del volumen para leer la FAT y encontrar la ubicación del bloque en cuestión, después de lo cual tendrá que moverse a la ubicación del propio bloque. En el caso peor, ambos movimientos tendrá lugar para cada uno de los bloques. Una de las ventajas de este esquema es que se mejora el tiempo de acceso aleatorio, porque el cabezal del disco puede encontrar la ubicación de cualquier bloque leyendo la información de la FAT.

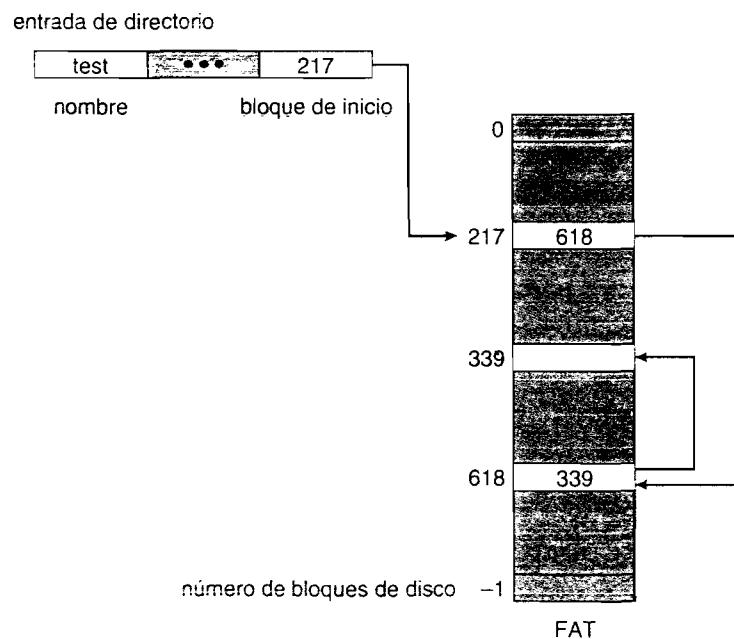


Figura 11.7 Tabla de asignación de archivos.

11.4.3 Asignación indexada

El método de la asignación enlazada resuelve los problemas de fragmentación externa y de declaración del tamaño que presentaba el método de la asignación contigua. Sin embargo, si no se utiliza una FAT, la asignación enlazada no puede soportar un acceso directo eficiente, ya que los punteros a los bloques están dispersos junto con los propios bloques por todo el disco y deben extraerse secuencialmente. El mecanismo de **asignación indexada** resuelve este problema agrupando todos los punteros en una única ubicación: el **bloque de índice**.

Cada archivo tiene su propio bloque de índice, que es una matriz de direcciones de bloques de disco. La entrada i -ésima del bloque de índice apunta al bloque i -ésimo del archivo. El directorio contiene la dirección del bloque de índice (Figura 11.8). Para localizar y leer el bloque i -ésimo, utilizamos el puntero contenido en la entrada i -ésima del bloque de índice. Este esquema es similar al esquema de paginación descrito en la Sección 8.4.

Cuando se crea el archivo, se asigna el valor *nulo* a todos los punteros del bloque de índice. Cuando se escribe por primera vez en el bloque i -ésimo, se solicita un bloque al gestor de espacio libre y su dirección se almacena en la entrada i -ésima del bloque de índice.

El mecanismo de asignación indexada soporta el acceso directo, sin sufrir el problema de la fragmentación externa, ya que puede utilizarse cualquier bloque del disco para satisfacer una solicitud de más espacio. Sin embargo, el mecanismo de asignación indexada sí que sufre el problema del desperdicio de espacio. El espacio adicional requerido para almacenar los punteros del bloque de índice es, generalmente, mayor que el que se requiere en el caso de la asignación enlazada. Considere un caso común en el que tenemos un archivo de sólo uno o dos bloques. Con el mecanismo de asignación enlazada, sólo perderemos el espacio de un puntero por cada bloque, mientras que con el mecanismo de asignación indexada, es preciso asignar un bloque de índice completo, incluso si sólo uno o dos punteros de este bloque van a tener un valor distinto del valor *nulo*.

Este punto nos plantea la cuestión de cuál debe ser el tamaño del bloque de índice. Cada archivo debe tener un bloque de índice, así que ese bloque deberá ser lo más pequeño posible. Sin embargo, si el bloque de índice es demasiado pequeño, no podrá almacenar suficientes punteros para un archivo de gran tamaño y será necesario implementar un mecanismo para resolver este problema. Entre los mecanismos que pueden utilizarse para ello se encuentran los siguientes:

- **Esquema enlazado.** Cada bloque de índice ocupa normalmente un bloque de disco. Por tanto, puede leerse y escribirse directamente. Para que puedan existir archivos de gran

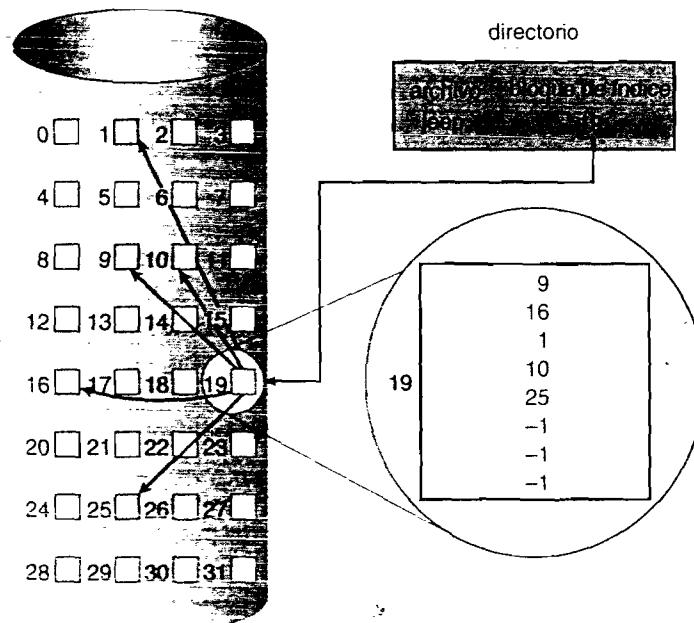


Figura 11.8 Asignación indexada del espacio de disco.

tamaño, podemos enlazar varios bloques de índice. Por ejemplo, un bloque de índice puede contener una pequeña cabecera que indique un nombre del archivo y un conjunto formado por las primeras 100 direcciones de bloque del disco. La siguiente dirección (la última palabra del bloque de índice) será el valor *nulo* (para un archivo de pequeño tamaño) o un puntero a otro bloque de índice (para un archivo de gran tamaño).

- **Índice multinivel.** Una variante de la representación enlazada consiste en utilizar un bloque de índice de primer nivel para apuntar a un conjunto de bloques de índice de segundo nivel, que a su vez apuntarán a los bloques del archivo. Para acceder a un bloque, el sistema operativo utiliza el índice de primer nivel para encontrar un bloque de índice de segundo nivel y luego emplea dicho bloque para hallar el bloque de datos deseado. Esta técnica podría ampliarse hasta un tercer o cuarto nivel, dependiendo del tamaño máximo de archivo que se desee. Con bloques de 4096 bytes, podríamos almacenar 1024 bytes en un bloque de índice; dos niveles de índice nos permitirían direccionar 1.048.576 bloques de datos, lo que equivale a un tamaño de archivo de hasta 4 GB.
- **Esquema combinado.** Otra alternativa, utilizada con el sistema UFS, consiste en mantener, por ejemplo, los primeros 15 punteros del bloque de índice en el nodo del archivo. Los primeros 12 de estos punteros hacen referencia a **bloques directos**, es decir, contienen la dirección de una serie de bloques que almacenan los datos del archivo. De este modo, los datos para los archivos de pequeño tamaño (de no más de 12 bloques) no necesitan un bloque de índice separado. Si el tamaño de bloque es de 4 KB, podrá accederse directamente a 48 KB de datos. Los siguientes tres punteros hacen referencia a **bloques indirectos**. El primero apunta a un **bloque indirecto de un nivel**, que es un bloque de índice que no contiene datos sino las direcciones de otros bloques que almacenan los datos. El segundo puntero hace referencia a un **bloque doblemente indirecto**, que contiene la dirección de un bloque que almacena las direcciones de otras series de bloques que contienen punteros a los propios bloques de datos. El último puntero contiene la dirección de un **bloque triplemente indirecto**. Con este método, el número de bloques que pueden asignarse a un archivo excede de la cantidad de espacio direccionable por los punteros de archivo de 4 bytes que se utilizan en muchos sistemas operativos. Un puntero de archivo de 32 bits sólo permite direccionar 2^{32} bytes, es decir, 4 GB. Muchas implementaciones de UNIX, incluyendo Solaris y AIX de IBM, soportan ahora punteros de archivo de hasta 64 bits. Los punteros de este tamaño permiten que los archivos y los sistemas de archivos tengan un tamaño del orden de los terabytes. En la Figura 11.9 se muestra un inodo de UNIX.

Los esquemas de asignación indexados sufren de los mismos problemas de rendimiento que el mecanismo de asignación enlazada. Específicamente, los bloques de índice pueden almacenarse en memoria caché, pero los bloques de datos pueden estar distribuidos por todo un volumen.

11.4.4 Prestaciones

Los métodos de asignación que hemos expuesto varían en lo que respecta a su eficiencia de almacenamiento y a los tiempos de acceso a los bloques de datos. Ambos criterios son importantes a la hora de seleccionar el método o métodos apropiados para implementarlos en un sistema operativo. Antes de seleccionar un método de asignación, necesitamos determinar cómo se utilizará el sistema: un sistema en el que el acceso sea preferentemente secuencial no debe utilizar el mismo método que otro donde el acceso sea fundamentalmente aleatorio.

Para cualquier tipo de acceso, el mecanismo de asignación contigua sólo requiere un acceso para extraer un bloque de disco. Puesto que podemos mantener fácilmente la dirección del archivo en memoria, podemos calcular inmediatamente la dirección de disco del bloque *i*-ésimo (o del siguiente bloque) y leerlo directamente.

En el mecanismo de asignación enlazada, también podemos mantener en memoria la dirección del siguiente bloque y leerlo directamente. Este método resulta adecuado para el acceso secuencial, pero para el acceso directo, un acceso al bloque *i*-ésimo puede requerir *i* lecturas de disco.

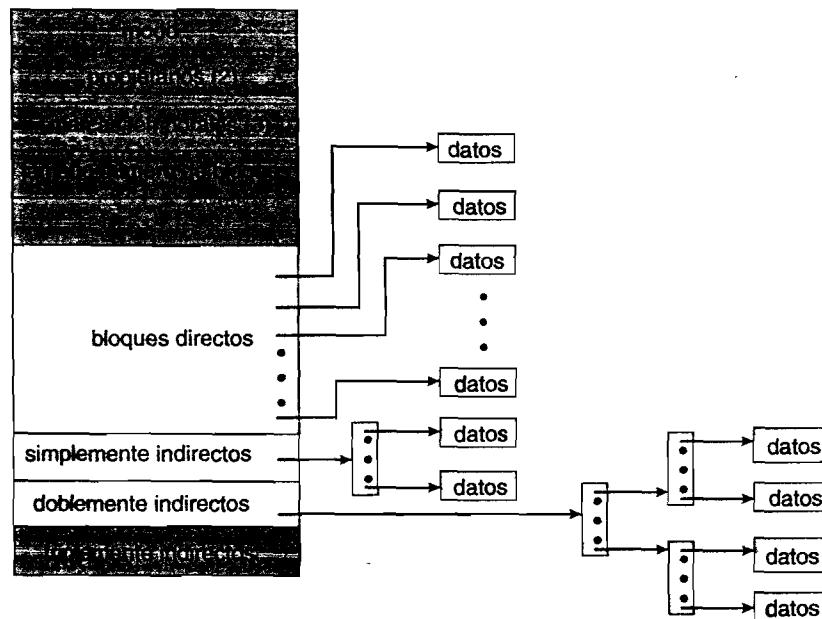


Figura 11.9 Un inodo de UNIX.

Este problema muestra por qué no debe utilizarse el mecanismo de asignación enlazada para las aplicaciones que requieran acceso directo.

Como resultado, algunos sistemas soportan los archivos de acceso directo utilizando un mecanismo de asignación contigua y emplean el mecanismo de asignación enlazada para el acceso secuencial. Para estos sistemas, debe declararse el tipo de acceso que va a realizarse en el momento de crear el archivo. Los archivos creados para acceso secuencial estarán enlazados y no podrán utilizarse con acceso directo. Los archivos creados para acceso directo serán contiguos y podrán soportar tanto el acceso directo como el acceso secuencial, pero será necesario declarar su longitud máxima en el momento de crear el archivo. En este caso, el sistema operativo deberá disponer de las estructuras de datos y los algoritmos apropiados para soportar *ambos* métodos de asignación. Los archivos pueden convertirse de un tipo a otro creando un nuevo archivo del tipo deseado, en el que se copiará el contenido del archivo antiguo. El archivo antiguo puede entonces borrarse, después de lo cual se renombrará el nuevo archivo.

El esquema de asignación indexada es más complejo. Si el bloque de índice ya está en memoria, puede realizarse el acceso directamente. Sin embargo, mantener en memoria el bloque de índice requiere un espacio considerable. Si este espacio de memoria no está disponible, podemos tener que leer el primer el bloque de índice y luego el bloque de datos deseado. Para un índice en dos niveles, puede que sean necesarias dos lecturas de bloques de índice. Para un archivo extremadamente grande, acceder a un bloque situado cerca del final del archivo podría requerir leer todos los bloques de índice antes de leer el bloque de datos necesario. Así, las prestaciones del mecanismo de asignación indexada dependerán de la estructura del índice, del tamaño del archivo y de la posición del bloque deseado.

Algunos sistemas combinan la asignación contigua con la asignación indexada, utilizando una asignación contigua para los archivos de pequeño tamaño (de hasta tres o cuatro bloques) y comutando automáticamente a un mecanismo de asignación indexada si el tamaño del archivo crece. Puesto que la mayoría de los archivos son pequeños y la asignación contigua resulta eficiente para los pequeños archivos, las prestaciones medias pueden ser bastante buenas.

Por ejemplo, la versión del sistema operativo UNIX de Sun Microsystems fue modificada en 1991 para mejorar el rendimiento del algoritmo de asignación del sistema de archivos. Las medidas de rendimiento indicaban que la tasa máxima de transferencia de disco en una estación de trabajo típica (una SPARCstation1 de 12 MIPS) ocupaba un 50 por ciento de la CPU y permitía obtener un ancho de banda de disco de sólo 1,5 MB por segundo. Para mejorar las prestaciones, Sun rea-

lizó una serie de cambios para asignar el espacio en *clusters* de 56 KB siempre que fuera posible (56 KB era el tamaño máximo de una transferencia DMA en los sistemas Sun en aquel momento). Este esquema de asignación reduce la fragmentación externa y, por tanto, los tiempos de búsqueda y de latencia. Además, se optimizaron las rutinas de lectura de disco para leer estos *clusters* de gran tamaño. La estructura de los inodos se dejó sin modificar. Como resultado de estos cambios y de la introducción de los mecanismos de lectura anticipada y de liberación postpuesta (de los que se habla en la Sección 11.6.2), pasó a utilizarse un 25 por ciento menos de tiempo de CPU y la tasa de transferencia mejoró sustancialmente.

En los sistemas reales se utilizan muchas otras técnicas de optimización. Dada la disparidad entre la velocidad de la CPU y la velocidad del disco, resulta razonable añadir miles de instrucciones adicionales al sistema operativo simplemente para ahorrarse unos pocos movimientos de los cabezales del disco. Además, esta disparidad se va acrecentando con el tiempo, hasta el punto de que se podría pensar en utilizar cientos de miles de instrucciones para optimizar los movimientos de los cabezales.

11.5 Gestión del espacio libre

Puesto que el espacio del disco está limitado, necesitamos reutilizar el espacio de los archivos borrados para los nuevos archivos, siempre que sea posible (los discos ópticos de una sola escritura sólo permiten una escritura en cada sector, por lo que dicha reutilización no es físicamente posible). Para controlar el espacio libre del disco, el sistema mantiene una **lista de espacio libre**. La lista de espacio libre indica todos los bloques de disco *libres*, aquellos que no están asignados a ningún archivo o directorio. Para crear un archivo, exploramos la lista de espacio libre hasta obtener la cantidad de espacio requerida y asignamos ese espacio al nuevo archivo. A continuación, este espacio se elimina de la lista de espacio libre. Cuando se borra un archivo, su espacio de disco se añade a la lista de espacio libre. La lista de espacio libre, a pesar de su nombre, puede no estar implementada con una estructura de lista, como vamos a ver a continuación.

11.5.1 Vector de bits

Recientemente, la lista de espacio libre se implementa como un **mapa de bits** o **vector de bits**. Cada bloque está representado por 1 bit. Si el bloque está libre, el bit será igual a 1; si el bloque está asignado, el bit será 0.

Por ejemplo, considere un disco en el que los bloques 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 y 27 están libres y el resto de los bloques están asignados. El mapa de bits de espacio libre sería

0011100111110001100000011100000 ...

La principal ventaja de este enfoque es su relativa simplicidad y la eficiencia que permite a la hora de localizar el primer bloque libre o *n* bloques libres consecutivos en el disco. De hecho, muchas computadoras suministran instrucciones de manipulación de bits que pueden utilizarse de manera efectiva para dicho propósito. Por ejemplo, la familia Intel a partir del 80386 y la familia Motorola a partir del 68020 (procesadores incorporados en los sistemas PC y Macintosh, respectivamente) tienen instrucciones que devuelven el desplazamiento dentro de una palabra del primer bit con el valor 1. Una técnica para localizar el primer bloque libre en un sistema que utilice un lector de bits para asignar el espacio de disco consiste en comprobar secuencialmente cada palabra del mapa de bits para ver si dicho valor es distinto de 0, ya que una palabra con valor 0 tendrá todos los bits iguales a 0 y representará un conjunto de bloques asignado. Cuando se encuentra la primera palabra distinta de 0 se la explora en busca del primer bit 1, que se corresponderá con la ubicación del primer bloque libre. El cálculo del número de bloque será:

$$(número\ de\ bits\ por\ palabra) \times (número\ de\ palabras\ de\ valor\ 0) \\ + \text{desplazamiento\ de\ primer\ bit\ 1}$$

De nuevo, podemos ver cómo determinadas características hardware imponen la funcionalidad software. Desafortunadamente, los vectores de bit son ineficientes a menos que se mantenga el vector completo en la memoria principal (y se lo escriba en disco ocasionalmente para propósitos de recuperación). Mantener ese vector en la memoria principal es posible para los discos de pequeño tamaño, pero no necesariamente para los discos de tamaño más grande. Un disco de 1,3 GB con bloques de 512 bytes necesitaría un mapa de bits de más de 332 KB para controlar todos los bloques libres, aunque si agrupamos todos los bloques en *clusters* de cuatro se reduce este número a unos 83 KB por disco. Un disco de 40 GB con bloques de 1 KB requeriría más de 5 MB para almacenar el mapa de bits.

11.5.2 Lista enlazada

Otra técnica para la gestión del espacio libre consiste en enlazar todos los bloques de disco libres, manteniendo un puntero al primer bloque libre en ubicación especial del disco y almacenándolo en la memoria caché. Este primer bloque contendrá un puntero al siguiente bloque libre del disco, etc. En nuestro ejemplo anterior (Sección 11.5.1), mantendríamos un puntero al bloque 2 como primer bloque libre. El bloque 2 contendría un puntero al bloque 3, que a su vez apuntaría al bloque 4, que apuntaría al bloque 5, el cual apuntaría al bloque 8, etc. (Figura 11.10). Sin embargo, este esquema es poco eficiente; para recorrer la lista, debemos leer cada bloque, lo que requiere un tiempo sustancial de E/S. Afortunadamente, no resulta frecuente tener que recorrer la lista de espacio libre. Usualmente, el sistema operativo simplemente necesita un bloque libre para poder asignar dicho bloque a un archivo, por lo que se utiliza el primer bloque de la lista de espacio libre. El método FAT incorpora el control de los bloques libres dentro de la estructura de datos utilizada para el algoritmo de asignación; en este caso, no hace falta utilizar ningún método separado.

11.5.3 Agrupamiento

Una modificación de la técnica basada en una lista de espacio libre consiste en almacenar las direcciones de n bloques libres en el primer bloque libre. Los primeros $n - 1$ de estos bloques estarán realmente libres. El último bloque contendrá las direcciones de otros n bloques libres, etc. De este modo, podrán encontrarse rápidamente las direcciones de un gran número de bloques libres, a diferencia del caso en que se utilice la técnica estándar de lista enlazada.

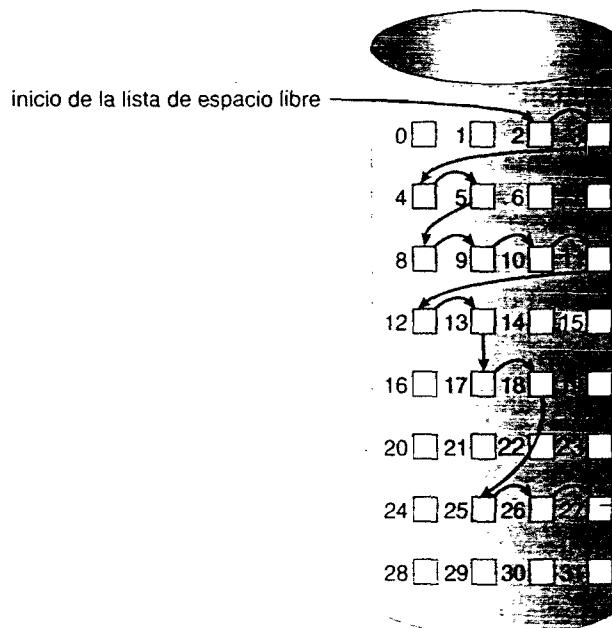


Figura 11.10 Lista de espacio libre enlazada en el disco.

11.5.4 Recuento

Otra técnica consiste en aprovechar el hecho de que, generalmente, puede asignarse o liberarse simultáneamente varios bloques contiguos, particularmente cuando se asigna el espacio mediante el algoritmo de asignación contigua o mediante un mecanismo de agrupación en *cluster*. Así, en lugar de mantener una lista de n direcciones de bloques de disco libres, podemos mantener la dirección del primer bloque libre y el número n de bloques libres contiguos que siguen a ese primer bloque. Cada entrada en la lista de espacio libre estará entonces compuesta por una dirección de disco y un contador. Aunque cada entrada requiere más espacio de lo que haría falta con una simple dirección de disco, la lista global será más corta, siempre y cuando el valor de ese contador sea generalmente superior a 1.

11.6 Eficiencia y prestaciones

Ahora que hemos expuesto diversas opciones de asignación de bloques y de gestión de directorios, podemos analizar con más detalle el efecto que tienen sobre las prestaciones y el uso eficiente del disco. Los discos tienden a representar uno de los principales cuellos de botella en el rendimiento de un sistema, ya que son el más lento de los principales componentes de un sistema informático. En esta sección, vamos a presentar diversas técnicas utilizadas para mejorar la eficiencia y las prestaciones del almacenamiento secundario.

11.6.1 Eficiencia

El uso eficiente del espacio de disco depende en gran medida de los algoritmos de asignación de disco y de directorio que se utilicen. Por ejemplo, los inodos de UNIX están preasignados en cada volumen. Incluso un disco "vacío" pierde un porcentaje de su espacio debido al almacenamiento de los nodos. Sin embargo, preasignando los inodos y distribuyéndolos por el volumen, mejoramos las prestaciones del sistema de archivos. Esta mejora de las prestaciones se debe a los algoritmos de asignación y de gestión del espacio libre de UNIX, que tratan de mantener los bloques de datos de un archivo cerca del bloque de inodo de ese archivo, con el fin de reducir el tiempo de búsqueda.

Como ejemplo adicional, volvemos a considerar el esquema de agrupación en *cluster* expuesto en la Sección 11.4, que ayuda a mejorar la velocidad de búsqueda y de transferencia de los datos de los archivos, a cambio de aumentar el grado de fragmentación interna. Para reducir esta fragmentación, el sistema BSD UNIX hace que varíe el tamaño de *cluster* a medida que crece el archivo. Los *clusters* de gran tamaño se utilizan siempre que vayan a poder llenarse, mientras que los *clusters* de pequeño tamaño se utilizan para los pequeños archivos y para el último *cluster* de cada archivo. Este sistema se describe en el Apéndice A.

También debemos considerar los tipos de datos que normalmente se almacenan en la entrada de directorio (o inodo) de un archivo. Normalmente, suele escribirse una "fecha de última modificación" para suministrar información al usuario y para determinar si es necesario realizar una copia de seguridad del archivo. Algunos sistemas mantienen también una "fecha de último acceso", con el fin de que el usuario pueda determinar cuándo se leyó el archivo por última vez. El resultado de mantener esta información es que, cada vez que se lee el archivo, es necesario escribir en uno de los campos de la estructura de directorio. Esto significa que se necesita leer el bloque en memoria, cambiar una sección y volver a escribir el bloque en el disco, ya que las operaciones de disco sólo tienen lugar en unidades de bloque (o de *cluster*). Por tanto, cada vez que se abre un archivo para lectura, es necesario también leer y escribir su entrada de directorio. Este requisito puede resultar bastante poco eficiente para los archivos a los que se acceda frecuentemente, así que debemos comparar las ventajas que proporciona con el impacto que tiene sobre el rendimiento a la hora de diseñar un sistema de archivos. Generalmente, tenemos que considerar *todos* los elementos de datos asociados con un archivo para ver el efecto que tienen sobre la eficiencia y sobre las prestaciones.

Como ejemplo, considere el impacto que tiene sobre la eficiencia el tamaño de los punteros que se utilicen para acceder a los datos. La mayoría de los sistemas utilizan punteros de 16 o de 32 bits en todo el sistema operativo. Estos tamaños de puntero limitan la longitud de un archivo a 2^{16} (64 KB) o 2^{32} bytes (4 GB). Algunos sistemas implementan punteros de 64 bits para incrementar este límite a 2^{64} bytes, que es ciertamente un número muy grande. Sin embargo, los punteros de 64 bits necesitan más espacio de almacenamiento y hacen, a su vez, que los métodos de asignación y de gestión del espacio libre (listas enlazadas, índices, etc.) utilicen también más espacio de disco.

Una de las dificultades a la hora de seleccionar un tamaño de puntero (o, de hecho, cualquier tamaño de asignación fijo dentro del sistema operativo) es la clarificación de los efectos de los cambios en la tecnología. Considere, por ejemplo, que el IBM XT PC tenía un disco duro de 10 MB y un sistema de archivos MS-DOS que sólo podía soportar 32 MB (cada entrada de la tabla FAT era de 12 bits y apuntaba a un *cluster* de 8 KB). A medida que se fueron incrementando las capacidades de disco, los discos de mayor tamaño tenían que dividirse en particiones de 32 MB, porque el sistema de archivos no podía controlar los bloques situados más allá de 32 MB. A medida que fueron haciéndose comunes los discos duros con capacidades de más de 100 MB, hubo que modificar las estructuras de datos y algoritmos de gestión del disco en MS-DOS para permitir la utilización de sistemas de archivos de mayor tamaño (se expandió cada entrada de la tabla FAT a 16 bits y posteriormente a 32 bits). Las decisiones iniciales relativas al sistema de archivos se tomaron por razones de eficiencia; sin embargo, con el lanzamiento de MS-DOS versión 4, millones de usuarios de computadoras sufrieron las incomodidades derivadas de tener que comutar al nuevo sistema de archivos de mayor tamaño. El sistema de archivos ZFS de Sun utiliza punteros de 128 bits, que en teoría nunca llegarán a necesitar ser ampliados (la masa mínima de un dispositivo capaz de almacenar 2^{128} bytes utilizando almacenamiento de nivel atómico sería de unos 272 billones de kilogramos).

Como ejemplo adicional, considere la evolución del sistema operativo Solaris de Sun. Originalmente, muchas estructuras de datos tenían longitud fija, que se asignaba durante el inicio del sistema. Estas estructuras incluían la tabla de procesos y la tabla de archivos abiertos. Cuando la tabla de procesos se llenaba, ya no podían crearse más procesos; cuando la tabla de archivos abiertos se llenaba, ya no podían abrirse más archivos. En estos casos, el sistema dejaba de poder proporcionar servicio a los usuarios. Los tamaños de las tablas sólo pueden incrementarse recompiñando el *kernel* y reiniciando el sistema. Desde el lanzamiento de Solaris 2, casi todas las estructuras del *kernel* se asignan dinámicamente, eliminando estos límites artificiales impuestos a las prestaciones del sistema. Por supuesto, los algoritmos que manipulan esta tabla son más complicados y el sistema operativo es un poco más lento, porque debe asignar y desasignar dinámicamente las entradas de las tablas; pero dicho precio es el que normalmente hay que pagar para obtener una funcionalidad de carácter más general.

11.6.2 Prestaciones

Incluso después de haber seleccionado los algoritmos básicos del sistema de archivos, podemos mejorar las prestaciones de diversas maneras. Como veremos en el Capítulo 13, la mayoría de las controladoras de disco incluyen una memoria local que actúa como **caché** incorporada en la tarjeta y que es lo suficientemente grande como para almacenar pistas completas al mismo tiempo. Una vez que se realiza una búsqueda, se lee la correspondiente pista en la caché de disco, comenzando por el sector situado bajo el cabezal de disco (lo que reduce el tiempo del ejemplo). La controladora de disco transfiere entonces al sistema operativo los sectores solicitados. Una vez que los bloques se transfieren desde la controladora de disco a la memoria principal, el sistema operativo puede almacenar allí los bloques en caché.

Algunos sistemas mantienen una sección separada de memoria principal, reservándola para **caché de búfer** en la que se almacenan los bloques bajo la suposición de que en breve volverán a ser utilizados de nuevo. Otros sistemas almacenan en caché los datos de los archivos utilizando una **caché de páginas**. La caché de páginas utiliza técnicas de memoria virtual para almacenar en caché los datos de los archivos en forma de páginas, en lugar de como bloques del sistema de archivos. El almacenamiento en caché de los datos de los archivos mediante direcciones virtuales

es mucho más eficiente que el almacenamiento en caché mediante bloques de disco físicos, ya que los accesos se producen a la memoria virtual en lugar de al sistema de archivos. Varios sistemas (incluyendo Solaris, Linux y Windows NT, 2000 y XP) utilizan una caché de páginas para almacenar en memoria tanto páginas de los procesos como datos de los archivos. Este mecanismo se conoce con el nombre de **memoria virtual unificada**.

Algunas versiones de UNIX y Linux proporcionan una **caché de búfer unificada**. Para ilustrar los beneficios de una caché de búfer unificada considere las dos alternativas existentes para abrir un archivo y acceder a su contenido. Una técnica consiste en utilizar el mapeo de memoria (Sección 9.7), mientras que la segunda es utilizar las llamadas estándar al sistema `read()` y `write()`. Sin una caché de búfer unificada, tendrímos una situación similar a la de la Figura 11.11. Aquí, las llamadas al sistema `read()` y `write()` pasan a través de la caché de búfer. Sin embargo, la llamada de mapeo de memoria requiere la utilización de dos cachés: la caché de páginas y la caché de búfer. Un mapeo de memoria se lleva a cabo leyendo bloques de disco del sistema de archivos y almacenándolos en la caché de búfer. Puesto que el sistema de memoria virtual no se comunica con la caché de búfer, el contenido del archivo en la caché de búfer debe copiarse en la caché de páginas. Esta situación se conoce con el nombre de **doble caché** y requiere almacenar en caché dos veces los datos del sistema de archivos. No sólo se desperdicia así memoria, sino que también se desperdicia un número significativo de ciclos de CPU y de E/S, debido a las transferencias extra de datos dentro de la memoria del sistema. Además, las posibles incoherencias entre las dos memorias caché pueden provocar la corrupción de los archivos. Por contraste, cuando se proporciona una caché de búfer unificada, tanto el mapeo de memoria como las llamadas al sistema `read()` y `write()` utilizan la misma caché de páginas. Esto tiene la ventaja de evitar el mecanismo de doble caché y también permite que el sistema de memoria virtual gestione los datos del sistema de archivos. La caché de búfer unificada se ilustra en la Figura 11.12.

Independientemente de si estamos almacenando en caché bloques de disco o páginas (o ambos), el algoritmo LRU (Sección 9.4.4) parece un algoritmo de propósito general bastante razonable para la sustitución de bloques o páginas. Sin embargo, la evolución de los algoritmos de caché de páginas de Solaris nos revela la dificultad inherente a la selección de un algoritmo. Solaris permite que los procesos y la caché de páginas compartan la memoria no utilizada. Las versiones anteriores de Solaris 2.5.1 no hacían ninguna distinción entre la asignación de páginas a un proceso y la asignación de páginas a la caché de páginas. Como resultado, un sistema que realizara muchas operaciones de E/S utilizaba la mayor parte de la memoria disponible para almacenar las páginas en caché. Debido a las altas tasas de E/S, el explorador de páginas (Sección 9.10.2) reclamaba páginas de los procesos (en lugar de reclamar las de las caché de páginas, cuando la memoria libre comenzaba a escasear). Solaris 2.6 y Solaris 7 implementaron un mecanismo

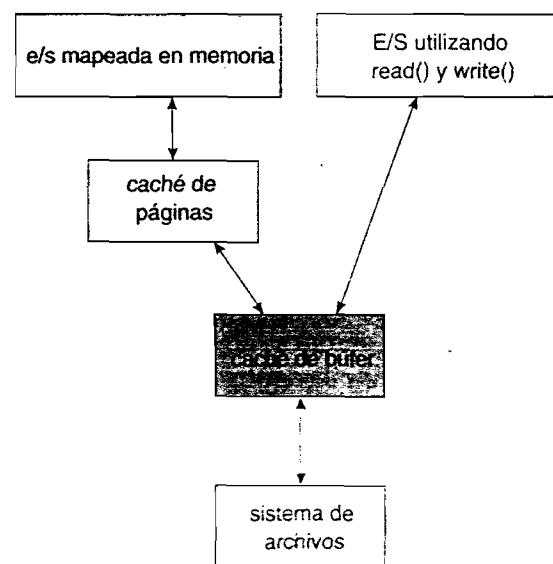


Figura 11.11 E/S sin una caché de búfer unificada.

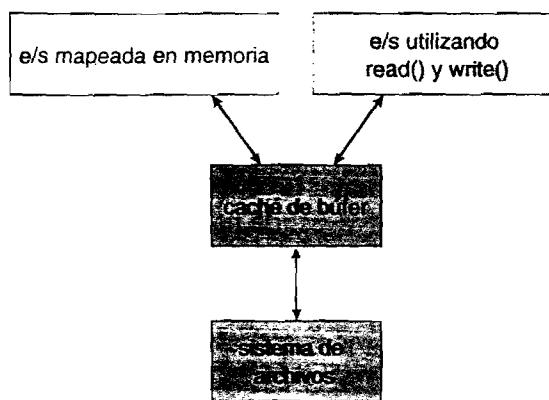


Figura 11.12 E/S utilizando una caché de búfer unificada.

opcional de *paginación con prioridad*, en el que el explorador de páginas daba prioridad a las páginas de los procesos frente a la caché de páginas. Solaris 8 aplicaba un límite fijo a las páginas de los procesos y a la caché de páginas del sistema de archivos, evitando que los procesos y la caché de páginas se expulsaran mutuamente de la memoria. Solaris 9 y 10 volvieron a cambiar los algoritmos para maximizar el uso de memoria y minimizar el fenómeno de la sobrepaginación. Este ejemplo práctico nos muestra la complejidad inherente a los intentos de optimizar el rendimiento y regular la optimización de la memoria caché.

Hay otras cuestiones que pueden afectar al rendimiento de E/S, como por ejemplo el hecho de si las escrituras en el sistema de archivos se producen en modo síncrono o asíncrono. Las **escrituras síncronas** se producen en el orden en el que el subsistema de disco las recibe, y esas escrituras no se almacenan en búfer. Por tanto, la rutina que realiza la invocación debe esperar a que los datos alcancen la unidad de disco, antes de continuar con sus tareas. Las **escrituras asíncronas** son las que se llevan a cabo la mayor parte de las veces. En una escritura asíncrona, los datos se almacenan en la caché y el control se devuelve a la rutina que hubiera hecho la invocación. Las escrituras de metadatos, entre otras, pueden ser síncronas. Los sistemas operativos incluyen frecuentemente un indicador en la llamada al sistema `open` para permitir que un proceso solicite que las escrituras se realicen síncronamente. Por ejemplo, las bases de datos utilizan estas características para las transacciones atómicas, con el fin de garantizar que los datos alcancen el dispositivo de almacenamiento estable en el orden requerido.

Algunos sistemas optimizan su caché de páginas utilizando diferentes algoritmos de sustitución, dependiendo del tipo de acceso del archivo. Para los archivos que se lean o escriban secuencialmente, sus páginas no se sustituirán en orden LRU, porque las páginas más recientemente utilizadas serán las que se utilicen más tarde, o quizás no lleguen a utilizarse nunca. En lugar de ello, el acceso secuencial puede optimizarse mediante ciertas técnicas conocidas con el nombre de liberación retardada y lectura anticipada. La **liberación retardada** elimina una página del búfer en cuanto se solicita la página siguiente; las páginas anteriores no volverán, probablemente, a utilizarse de nuevo, así que representan un desperdicio de espacio de búfer. Con el mecanismo de **lectura anticipada**, cuando se solicita una página, se leen y almacenan en caché tanto esa página como varias páginas sucesivas. Dichas páginas es bastante probable que se soliciten después de procesar la página actual. Al extraer los datos del disco en una única transferencia y almacenarlos en caché, ahorraremos una considerable cantidad de tiempo. Podríamos pensar que una caché de pista de disco en la controladora elimina la necesidad de utilizar un mecanismo de lectura anticipada en un sistema multiprogramado. Sin embargo, debido a la alta latencia y al coste de procesamiento adicional implicado en la realización de muchas pequeñas transferencias desde la caché de la pista del disco a la memoria principal, sigue siendo beneficioso realizar una lectura anticipada.

La caché de páginas, el sistema de archivos y los controladores de disco interactúan entre sí de forma bastante interesante. Cuando se escriben datos en un archivo del disco, las páginas se almacenan temporalmente en la caché y el controlador de disco ordena su cola de salida de acuerdo con la dirección de disco. Estas dos opciones permiten al controlador de disco minimizar los repon-

sicionamientos de los cabezales del disco y escribir los datos en los instantes óptimos, de acuerdo con la rotación del disco. A menos que se requieran escrituras síncronas, un proceso que quiere escribir en el disco simplemente escribirá en la caché y el sistema escribirá asíncronamente los datos en el disco cuando sea conveniente. El proceso de usuario verá así que las escrituras se producen muy rápido. Cuando se leen datos de un archivo de disco, el sistema de E/S de bloque realiza una cierta lectura anticipada; sin embargo, las escrituras son bastante más asíncronas que las lecturas. Así, las operaciones de salida hacia disco a través del sistema de archivos son a menudo más rápidas que la entrada cuando se realizan transferencias de grandes cantidades de información, un hecho que choca con lo que intuitivamente cabría esperar.

11.7 Recuperación

Los archivos y directorios se mantienen tanto en memoria principal como en disco, y debe tenerse cuidado para que los fallos del sistema no provoquen una pérdida de datos o una incoherencia en los mismos. Vamos a analizar estas cuestiones en las secciones siguientes.

11.7.1 Comprobación de coherencia

Como hemos explicado en la Sección 11.3, parte de la información de directorios se almacena en la memoria principal (o en caché) para acelerar el acceso. La información de directorios en la memoria principal está, generalmente, más actualizada que la correspondiente información en el disco, porque la información de directorios almacenada en caché no se escribe necesariamente en el disco nada más producirse la actualización.

Consideremos, entonces, el posible ejemplo de un fallo de la computadora. El contenido de la caché y de los búferes, así como de las operaciones de E/S que se estuvieran realizando en ese momento, pueden perderse, y con él se perderán los cambios realizados en los directorios correspondientes a los archivos abiertos. Dicho suceso puede dejar el sistema de archivos en un estado incoherente. El estado real de algunos archivos no será el que se describe en la estructura de directorios. Con frecuencia, suele ejecutarse un programa especial durante el reinicio para comprobar las posibles incoherencias del disco y corregirlas.

El **comprobador de coherencia** (un programa del sistema tal como `fsck` en UNIX o `chkdsk` en MS-DOS), compara los datos de la estructura de directorios con los bloques de datos del disco y trata de corregir todas las incoherencias que detecte. Los algoritmos de asignación y de gestión del espacio libre dictan los tipos de problemas que el comprobador puede tratar de detectar y dictan también el grado de éxito que el comprobador puede tener en esta tarea. Por ejemplo, si se utiliza un sistema de asignación enlazada y existe un enlace entre cada bloque y el siguiente, puede reconstruirse el archivo completo a partir de los bloques de datos y volver a crear la estructura de directorios. Por el contrario, la pérdida de una entrada de directorio en un sistema de asignación indexada puede ser desastrosa, porque los bloques de datos no tienen ningún conocimiento acerca de los demás bloques de datos del archivo. Por esta razón, UNIX almacena en caché las entradas de directorio para las lecturas, pero todas las escrituras de datos que provoquen algún cambio en la asignación de espacio o en algún otro tipo de metadato se realizan síncronamente, antes de escribir los correspondientes bloques de datos. Por supuesto, también pueden aparecer problemas si se interrumpe una escritura síncrona debido a un fallo catastrófico.

11.7.2 Copia de seguridad y restauración

Los discos magnéticos fallan en ocasiones y es necesario tener cuidado para garantizar que los datos perdidos debido a esos fallos no se pierdan para siempre. Con este fin, pueden utilizarse programas del sistema para realizar una **copia de seguridad** de los datos del disco en otro dispositivo de almacenamiento, como por ejemplo un disquete, una cinta magnética, un disco óptico o incluso otro disco duro. La recuperación de la pérdida de un archivo individual o de un disco completo puede ser entonces, simplemente, una cuestión de **restaurar** los datos a partir de la copia de seguridad.

Para minimizar la cantidad de datos que haya que copiar, podemos utilizar la información contenida en la entrada de directorio de cada archivo. Por ejemplo, si el programa de copia de seguridad sabe cuándo se realizó la última copia de seguridad de un archivo y la fecha de última modificación del archivo contenida en el directorio indica que el archivo no ha cambiado desde esa fecha, no será necesario volver a copiar el archivo. Así, un plan típico de copia de seguridad podría ser el siguiente:

- **Día 1.** Copiar en el soporte de copia de seguridad todos los archivos del disco. Esto se denomina **copia de seguridad completa**.
 - **Día 2.** Copiar en otro soporte físico todos los archivos que se hayan modificado desde el día 1. Esta es una **copia de seguridad incremental**.
 - **Día 3.** Copiar en otro soporte físico todos los archivos que se hayan modificado desde el día 2.
-
- **Día N.** Copiar en otro soporte físico todos los archivos que se hayan modificado desde el día $N - 1$. Después, volver al día 1.

Podemos escribir las copias de seguridad correspondientes al nuevo ciclo sobre el conjunto anterior de soportes físicos o en un nuevo conjunto de soportes de copia de seguridad. De esta forma, podemos restaurar un disco completo comenzando la restauración con la copia de seguridad completa y continuando con cada una de las copias de seguridad incrementales. Por supuesto, cuanto mayor sea el valor de N , más cintas o discos habrá que leer para efectuar una restauración completa. Una ventaja adicional de este ciclo de copia de seguridad es que podemos restaurar cualquier archivo que haya sido borrado accidentalmente durante ese ciclo, extrayendo el archivo borrado de la copia de seguridad del día anterior. La longitud del ciclo será un compromiso entre la cantidad de soportes físicos de copia de seguridad requeridos y el número de días pasados a partir de los cuales podamos realizar una restauración. Para reducir el número de cintas que haya que leer para efectuar una restauración, una opción consiste en realizar una copia de seguridad completa y luego copiar cada día todos los archivos que hayan cambiado desde la última copia de seguridad completa. De esta forma, puede realizarse la restauración utilizando sólo la copia de seguridad incremental más reciente y la copia de seguridad completa, no necesitándose ninguna otra copia de seguridad incremental. El compromiso inherente a este sistema es que el número de archivos modificado se incrementa a diario, por lo que cada copia de seguridad incremental sucesiva contiene más archivos y requiere más espacio en el soporte de copia de seguridad.

Es posible que un usuario se dé cuenta de que un archivo concreto falta o está corrompido mucho después de que ese daño se haya producido. Por esta razón, conviene planificar la realización de copias de seguridad completas de tiempo en tiempo, las cuales se guardarán “para siempre”. Resulta conveniente almacenar estas copias de seguridad permanentes en algún lugar alejado de las copias de seguridad normales, con el fin de proteger los datos frente a desastres físicos, como por ejemplo un incendio que destruya la computadora y todas sus copias de seguridad. Asimismo, si el ciclo de copia de seguridad implica una reutilización de los soportes físicos, debemos tomar las precauciones para no reutilizar esos soportes físicos demasiadas veces: si el soporte físico se desgasta, puede que no sea posible restaurar ningún dato a partir de las copias de seguridad.

11.8 Sistemas de archivos con estructura de registro

Los investigadores del campo de la informática suelen encontrarse con que los algoritmos y tecnologías originalmente usados en un área resultan igualmente útiles en otras áreas distintas. Éste es el caso de los algoritmos de recuperación basados en registro que se utilizan en las bases de

datos y que hemos descrito en la Sección 6.9.2. Estos algoritmos de registro se han aplicado con éxito al problema de la comprobación de coherencia. Las implementaciones resultantes se conocen con el nombre de sistemas de archivos **orientados a transacciones y basados en registro** (**o sistemas de archivos basados en diario**).

Recuerde que un fallo catastrófico del sistema puede hacer que aparezcan incoherencias entre las estructuras de datos del sistema de archivos residente en el disco, como por ejemplo las estructuras de directorios, los punteros de bloques libres y los punteros de bloques FCB libres. Antes de que se utilizaran técnicas basadas en registro en los sistemas operativos, los cambios realizados en estas estructuras se efectuaban de manera directa. Una operación típica, como la de creación de un archivo, podía implicar muchos cambios estructurales dentro del sistema de archivos del disco. Por ejemplo, se modificaban estructuras de directorios, se asignaban bloques FCB, se asignaban bloques de datos y se reducían los contadores de bloques libres para todos estos tipos de bloques. Estos cambios pueden verse interrumpidos por un fallo catastrófico y, como resultado, pueden aparecer incoherencias entre las distintas estructuras. Por ejemplo, el contador de bloques FCB libres podría indicar que un cierto FCB ha sido asignado y, sin embargo, la estructura de directorios podría no apuntar a ese FCB. Si no hubiera una fase de comprobación de coherencia, dicho FCB dejaría de ser utilizable.

Aunque podemos dejar que las estructuras se descompongan y repararlas durante las tareas de recuperación, este enfoque tiene varios problemas. Uno de ellos es que la incoherencia puede ser de carácter irreparable; el comprobador de coherencia puede no ser capaz de recuperar las estructuras, lo que provocaría la pérdida de archivos o incluso de directorios completos. La comprobación de coherencia puede requerir la intervención humana para resolver los conflictos y es posible que no haya ninguna persona para intervenir en el momento preciso. De este modo, el sistema puede dejar de estar disponible hasta que una persona le diga lo que tiene que hacer. Las comprobaciones de coherencia también tardan mucho tiempo y consumen recursos del sistema. Para comprobar varios terabytes de datos pueden hacer falta horas.

La solución a este problema consiste en aplicar técnicas de recuperación basadas en registro para las actualizaciones de los metadatos del sistema de archivos. Tanto NTFS como el sistema de archivos Veritas utilizan este método, que también es una adición opcional a UFS en Solaris 7 y versiones posteriores. De hecho, esta técnica está empezando a resultar común en muchos sistemas operativos.

Fundamentalmente, todos los cambios de los metadatos se escriben secuencialmente en un registro. Cada conjunto de operaciones necesario para realizar una tarea específica es una **transacción**. Una vez que los cambios se han escrito en este registro, se consideran confirmados y la llamada al sistema puede volver al proceso de usuario, permitiéndolo continuar con su ejecución. Mientras tanto, estas entradas de registro se aplican en las estructuras reales del sistema de archivos. A medida que se realizan los cambios, se actualiza un puntero para indicar qué acciones se han completado y cuáles no se han completado todavía. Una vez que se ha completado una transacción confirmada, se la elimina del archivo de registro, que es en la práctica un búfer circular. Un **búfer circular** escribe hasta llegar al final de su espacio asignado y luego continúa por el principio, sobreescritiendo los valores antiguos durante el proceso. Obviamente, no queremos que el búfer escriba sobre los datos que no hayan sido todavía guardados, por lo que se implementan los mecanismos necesarios para evitar que eso suceda. El registro puede estar en una sección separada del sistema de archivos o incluso en una sección separada del disco. Lo más eficiente, aunque también es más complejo, es que el registro utilice cabezales separados de lectura y escritura, reduciendo así la contienda por el uso de los cabezales y los tiempos de búsqueda.

Si el sistema sufre un fallo catastrófico, el archivo de registro contendrá cero o más transacciones. Todas las transacciones que contenga no habrán sido aplicadas al sistema de archivos, aun cuando hubieran sido confirmadas por el sistema operativo, por lo que será necesario aplicarlas. Las transacciones situadas después del puntero pueden ejecutarse hasta que el trabajo se complete, de modo que las estructuras del sistema de archivos sigan siendo coherentes. El único problema que puede aparecer es cuando una transacción fue abortada, es decir, cuando la transacción no llegó a confirmarse antes de que el sistema sufriera el fallo catastrófico. Todos los cambios correspondientes a dicha transacción que hubieran sido aplicados al sistema de archivos deberán

deshacerse, preservando de nuevo la coherencia del sistema de archivos. Este proceso de recuperación es lo único que se necesita después de un fallo catastrófico, eliminando los problemas relacionados con el proceso de comprobación de coherencia.

Un beneficio adicional de la utilización de un mecanismo de registro de las actualizaciones de los metadatos del disco es que dichas actualizaciones tienen lugar mucho más rápidamente que cuando se las aplica directamente a las estructuras de datos residentes en el disco. La razón de esta mejora de la eficiencia es que la E/S secuencial es más rápida que la E/S aleatoria. Las costosas escrituras síncronas de metadatos aleatorios se transforman en escrituras secuenciales síncronas mucho menos costosas, que se llevan a cabo sobre el área de registro del sistema de archivos basado en registros. Dichos cambios, a su vez, se aplican asíncronamente mediante escrituras aleatorias en las estructuras apropiadas. El resultado global es una significativa ganancia en la velocidad de las operaciones relativas a los metadatos, como por ejemplo la creación y borrado de archivos.

11.9 NFS

Los sistemas de archivos en red son de uso bastante común hoy en día. Normalmente, están integrados con la estructura global de directorios y con la interfaz del sistema cliente. NFS es un buen ejemplo de sistema de archivos en red cliente-servidor ampliamente utilizado y bien implementado. Aquí, vamos a utilizarlo como ejemplo para analizar los detalles de implementación de los sistemas de archivos en red.

NFS es tanto una implementación como una especificación de un sistema software para el acceso a archivos remotos a través de redes LAN (o incluso redes WAN). NFS es parte de ONC+, que la mayoría de los fabricantes de sistemas UNIX y algunos sistemas operativos para PC soportan. La implementación descrita aquí forma parte del sistema operativo Solaris, que es una versión modificada de UNIX SVR4 que se ejecuta sobre estaciones de trabajo Sun y otras plataformas hardware. Utiliza el protocolo TCP o UDP/IP (dependiendo de las redes de interconexión). La especificación e implementación están enlazadas en nuestra descripción de NFS; cuando necesitemos proporcionar algún detalle, haremos referencia a la implementación de Sun, mientras que cuando la descripción sea general, se aplicará también a la especificación.

11.9.1 Introducción

NFS considera un conjunto de estaciones de trabajo interconectadas como si fueran un conjunto de máquinas independientes con sistemas de archivos independientes. El objetivo es permitir un cierto grado de compartición entre estos sistemas de archivos (cuando se realicen solicitudes explícitas) de forma completamente transparente. La compartición se basa en relaciones cliente-servidor. Una máquina puede ser, y a menudo lo es, tanto un cliente como un servidor. La compartición es posible entre cualquier pareja de máquinas. Para garantizar la independencia con respecto a las máquinas, la compartición de un sistema de archivos remoto sólo afecta a la máquina cliente, y no a ninguna otra máquina.

Para que pueda accederse a un directorio remoto de forma transparente desde una máquina concreta (por ejemplo, desde M_1), un cliente en dicha máquina deberá ejecutar primero una operación de montaje. La semántica de esta operación implica montar un directorio remoto sobre un directorio de un sistema de archivos local. Una vez completada la operación de montaje, el directorio montado parecerá un subárbol normal del sistema de archivos local, que sustituirá al subárbol descendiente del directorio local. El directorio local se convertirá en el nombre de la raíz del directorio recién montado. La especificación del directorio remoto como argumento para la operación de montaje no se realiza de forma transparente, sino que es necesario proporcionar la ubicación (o nombre de host) del directorio remoto. Sin embargo, a partir de ahí, los usuarios de la máquina M_1 podrán acceder a los archivos del directorio remoto de forma totalmente transparente.

Para ilustrar el montaje de archivos, considere el sistema de archivos mostrado en la Figura 11.13, donde los triángulos representan los subárboles de directorios que nos interesan. La figura muestra tres sistemas de archivos independientes en las máquinas denominadas U , S_1 y S_2 . En

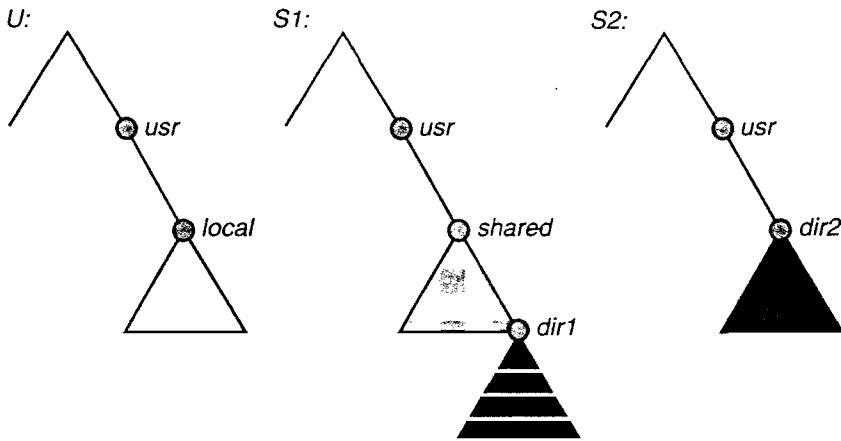


Figura 11.13 Tres sistemas de archivos independientes.

este punto, sólo puede accederse en cada máquina a los archivos locales. En la Figura 11.14(a) se muestran los efectos de montar $S1:/usr/shared$ sobre $U:/usr/local$. Esta figura muestra la vista que los usuarios de U tendrían de su sistema de archivos. Observe que, después de completar el montaje, podrán acceder a cualquier archivo del directorio $dir1$ utilizando el prefijo $/usr/local/dir1$. El directorio original $/usr/local$ de dicha máquina ya no será visible.

Respetando el mecanismo de acreditación de derechos de acceso, cualquier sistema de archivos o cualquier directorio de un sistema de archivos puede montarse remotamente sobre cualquier directorio local. Las estaciones de trabajo sin disco pueden incluso montar su propia raíz a partir de servidores.

En algunas implementaciones NFS también se permiten los montajes en cascada. En otras palabras, puede montarse un sistema de archivos sobre otro sistema de archivos que, en lugar de ser local, haya sido montado remotamente. Cada máquina sólo se verá afectada por aquellos montajes que ella misma haya invocado. Montar un sistema de archivos remoto no proporciona al cliente acceso a otros sistemas de archivos que hubieran podido ser montados sobre el sistema de archivos anterior. Por tanto, el mecanismo de montaje no posee la propiedad de transitividad.

Continuando con nuestro ejemplo anterior, en la Figura 11.14(b) se ilustra una serie de montajes en cascada. La figura muestra el resultado de montar $S2:/usr/dir2$ sobre $U:/usr/local/dir1$, que ya había sido remotamente montado a partir de $S1$. Los usuarios podrán acce-

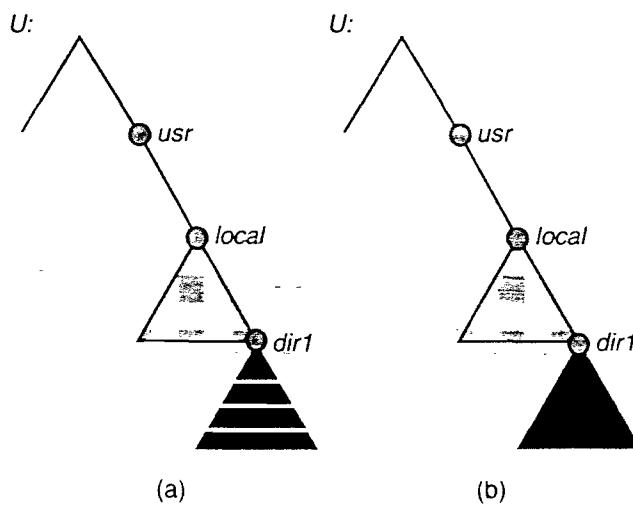


Figura 11.14 Montaje en NFS. (a) Montajes. (b) Montajes en cascada.

der a los archivos contenidos en `dir1` en `U` utilizando el prefijo `/usr/local/dir1`. Si se monta un sistema de archivos compartidos sobre los directorios de inicio de un usuario en todas las máquinas de una red, el usuario podrá iniciar una sesión en cualquier estación de trabajo y obtener su entorno inicial. Esta técnica permite la **movilidad de los usuarios**.

Uno de los objetivos de diseño de NFS era operar en un entorno heterogéneo de diversas máquinas, diversos sistemas operativos y diversas arquitecturas de red. La especificación NFS es independiente de todos estos soportes y permite, por tanto, que se realicen otras implementaciones. Esta independencia se consigue utilizando primitivas RPC construidas sobre un protocolo de representación de datos externos (XDR, external data representation) utilizado entre dos interfaces independientes de la implementación. Por tanto, si el sistema está compuesto por máquinas y sistemas de archivos heterogéneos con una interfaz adecuada a NFS, pueden montarse sistemas de archivos de tipos diferentes tanto local como remotamente. La especificación NFS distingue entre los servicios proporcionados por un mecanismo de montaje y los propios servicios de acceso a archivos remotos. En consecuencia, se han especificado dos protocolos diferentes para estos servicios: un protocolo de montaje y un protocolo para los accesos a archivos remotos, que es el **protocolo NFS**. Los protocolos están especificados como conjuntos de llamadas RPC. Estas llamadas a RPC son los bloques básicos que se utilizan para implementar un acceso transparente a archivos remotos.

11.9.2 El protocolo de montaje

El **protocolo de montaje** establece la conexión lógica inicial entre un **servidor** y un **cliente**. En la implementación de Sun, cada máquina tiene un proceso servidor, fuera del `kernel`, que implementa las funciones del protocolo.

Una implementación de montaje incluye el nombre del directorio remoto que hay que montar y el nombre de la máquina servidora donde está almacenado. La solicitud de montaje se mapea sobre la correspondiente llamada RPC y se reenvía al servidor de montaje que se esté ejecutando sobre la máquina servidora especificada. El servidor mantiene una **lista de exportación** que especifica los sistemas de archivos locales que está exportando para montaje, junto con los nombres de las máquinas que están autorizada a montarlos (en Solaris, esta lista es `/etc/dfs/dfstab`, que sólo puede ser editada por un superusuario). La especificación puede también incluir derechos de acceso, como por ejemplo de sólo lectura. Para simplificar el mantenimiento de las listas de exportación y de las tablas de montaje, puede utilizarse un esquema distribuido de denominación para albergar esta información y hacer que esté disponible para los clientes apropiados.

Recuerde que cualquier directorio dentro de un sistema de archivos exportados puede ser montado remotamente por una máquina convenientemente acreditada. Las unidades componentes del sistema son esos directorios. Cuando el servidor recibe una solicitud de montaje que esté conforme con su lista de exportación, devolverá al cliente un descriptor de archivo que sirve como clave para ulteriores accesos a los archivos contenidos en el sistema de archivos montado. El descriptor de archivo contiene toda la información que el servidor necesita para distinguir uno de los archivos individuales que estén almacenados en él. En términos de UNIX, el descriptor de archivo está compuesto por un identificador del sistema de archivos y un número de modo que identifica el directorio montado exacto dentro del sistema de archivos exportado.

El servidor también mantiene una lista de las máquinas clientes y de los directorios actualmente montados correspondientes a cada una. Esta lista se utiliza principalmente para propósitos administrativos, como por ejemplo para notificar a todos los clientes que el servidor va a ser detenido. El estado del servidor sólo puede verse afectado por el protocolo de montaje mediante la adición y borrado de entradas a dicha lista.

Usualmente, cada sistema tiene una preconfiguración estática de montaje que se establece durante el arranque del sistema (`/etc/vfstab` en Solaris); sin embargo, esta configuración puede ser modificada. Además del propio procedimiento de montaje, el protocolo de montaje incluye varios otros procedimientos, como uno para desmontar y otro para devolver la lista de exportación.

11.9.3 El protocolo NFS

El protocolo NFS proporciona un conjunto de llamadas RPC para operaciones remotas con archivos. Los procedimientos definidos soportan las siguientes operaciones:

- Búsqueda de un archivo en un directorio.
- Lectura de un conjunto de entradas de directorio.
- Manipulación de enlaces y directorios.
- Accesos a los atributos de un archivo.
- Lectura y escritura de archivos.

Estos procedimientos sólo pueden invocarse después de que se haya establecido un descriptor de archivo para el directorio remotamente montado.

La omisión de las operaciones `open()` y `close()` es intencionada. Una característica importante de los servidores NFS es que *no tienen memoria del estado*. Los servidores no mantienen información acerca de sus clientes entre un acceso y otro. En el lado del servidor no existe nada similar a las estructuras de archivos o a las tablas de archivos abiertos de UNIX. En consecuencia, cada solicitud debe proporcionar un conjunto completo de argumentos, incluyendo un identificador único de archivo y un desplazamiento absoluto dentro del archivo para realizar las operaciones apropiadas. El diseño resultante es bastante robusto, ya que no hace falta ninguna medida especial para recuperar un servidor después de un fallo catastrófico. Para que esto sea así, las operaciones con los archivos deben ser idempotentes. Toda solicitud NFS tiene un número de secuencia, que permite al servidor determinar si una solicitud está duplicada o si le falta alguna solicitud.

Parece que el mantener la lista de clientes que hemos mencionado anteriormente viole la naturaleza del servidor, carente de memoria. Sin embargo, esta lista no es esencial para la operación correcta del cliente o el servidor y no necesita, por tanto, ser restaurada después de un fallo catastrófico del servidor. En consecuencia, puede incluir datos incoherentes y sólo se trata como una sugerencia.

Una implicación adicional de la filosofía basada en servidores carentes del estado que además es resultado del carácter síncrono de las llamadas RPC es que los datos modificados, incluyendo los bloques de bidirección y de estado, deben confirmarse en el disco del servidor antes de devolver los resultados al cliente. Es decir, un cliente puede almacenar en caché los bloques que haya que escribir, pero cuando se los envía al servidor, asumirá que esos bloques han alcanzado los discos del servidor. El servidor debe escribir todos los datos NFS de manera síncrona. De este modo, un fallo catastrófico de un servidor, seguido de la correspondiente recuperación, será invisible para los clientes; todos los bloques que el servidor gestiona por cuenta del cliente estarán intactos. El consiguiente impacto en las prestaciones puede ser bastante grande, porque se pierden las ventajas del almacenamiento en caché. Las prestaciones pueden incrementarse utilizando un almacenamiento que tenga su propia caché no volátil (usualmente, una memoria alimentada por batería). La controladora de disco confirma la escritura de disco una vez que los datos se han escrito en la caché no volátil. En esencia, el host ve una escritura síncrona muy rápida. Estos bloques permanecerán intactos incluso después de un fallo catastrófico del sistema y se escriben periódicamente desde este almacenamiento estable al disco.

Se garantiza que cada llamada a procedimiento de escritura NFS es atómica y que no se entremezclará con otras llamadas de escritura sobre el mismo archivo. Sin embargo, el protocolo NFS no proporciona mecanismos de control de concurrencia. Una llamada al sistema `write()` puede descomponerse en varias escrituras RFC, porque cada llamada de lectura o escritura NFS puede contener hasta 8 KB de datos y los paquetes UDP están limitados a 1500 bytes. Como resultado, pueden entremezclarse los datos de dos usuarios que estén escribiendo sobre el mismo archivo remoto. Para evitar esto, y teniendo en cuenta que la gestión de bloqueos tiene una memoria inherente del estado, debe utilizarse un servicio externo NFS para proporcionar el bloqueo (y así se hace en Solaris). Se aconseja a los usuarios que coordinen el acceso a los archivos compartidos utilizando mecanismos externos a NFS.

NFS se encuadra en el sistema operativo mediante un VFS. Como ilustración de la arquitectura, veamos cómo se gestionaría una operación sobre un archivo remoto que ya esté abierto (sigue el ejemplo en la Figura 11.15). El cliente inicia la operación mediante una llamada normal al sistema. El nivel del sistema operativo mapea esta llamada sobre una operación VFS relativa al vnode apropiado. A continuación se realiza una llamada RPC al nivel de servicio NFS situado en el servidor remoto. Esta llamada se reinyecta en el nivel VFS del sistema remoto, que determinará que se trata de una llamada local e invocará la operación apropiada del sistema de archivos. Esta ruta se sigue a la inversa para devolver el resultado. Una ventaja de esta arquitectura es que el cliente y el servidor son idénticos; por tanto, una máquina puede actuar como cliente, como servidor o como ambos. El propio servicio en cada servidor se lleva a cabo mediante hebras del *kernel*.

11.9.4 Traducción de nombres de ruta

La traducción de nombres de ruta en NFS implica analizar sintácticamente un nombre de ruta tal como /usr/local/dir1/file.txt y descomponerlo en entradas de directorio separadas, es decir, extraer sus diferentes componentes: (1) usr, (2) local y (3) dir1. La traducción de nombres de ruta se realiza descomponiendo la ruta en los nombres componentes y realizando una llamada lookup NFS separada para cada pareja formada por un nombre de componente y por un vnode de directorio. Una vez que se cruza un punto de montaje, toda búsqueda de un componente genera una RPC separada dirigida al servidor. Este caro esquema de recorrido de nombres de ruta es absolutamente necesario, ya que la disposición del espacio lógico de nombres de cada cliente es diferente y está dictada por los montajes que el cliente haya realizado. Sería mucho más eficiente entregar a un servidor un nombre de ruta y recibir como respuesta un vnode de destino una vez que se encontrara un punto de montaje. Sin embargo, en cualquier punto podemos encontrarnos con otro punto de montaje para ese cliente concreto, punto de montaje del que no será consciente el servidor, que carece de memoria del estado.

Para que la búsqueda sea rápida, una caché de búsqueda de nombres de directorio en el lado del cliente almacena los vnodes de los nombres de directorio remotos. Esta caché acelera las referencias a archivos que tengan el mismo nombre de ruta inicial. La caché de directorio se descarta cuando los atributos devueltos por el servidor no se corresponden con los atributos del vnode almacenado en la caché.

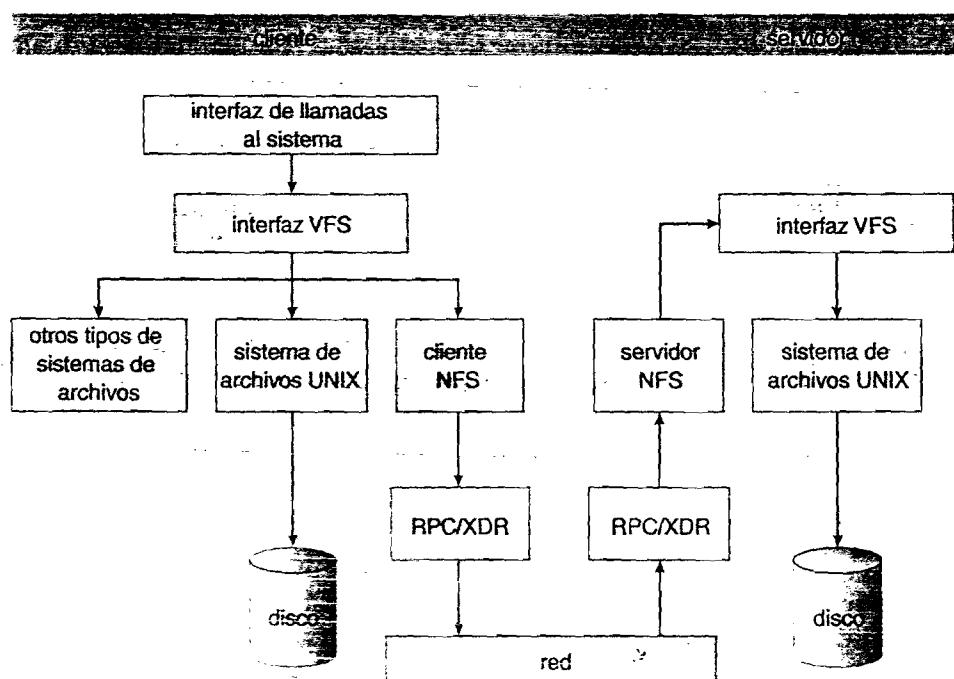


Figura 11.15. Vista esquemática de la arquitectura NFS.

Recuerde que montar un sistema de archivos remoto sobre otro sistema de archivos remoto ya montado (un montaje en cascada) es una operación permitida en algunas implementaciones de NFS. Sin embargo, un servidor no puede actuar como intermediario entre un cliente y otro servidor. En lugar de ello, el cliente debe establecer una conexión directa cliente-servidor con ese segundo servidor, montando directamente el directorio deseado. Cuando un cliente tiene un montaje en cascada, puede involucrar a más de un servidor en el recorrido de un nombre de ruta. Sin embargo, cada búsqueda de componentes se realiza entre el cliente original y un servidor. Por tanto, cuando un cliente realiza una búsqueda en un directorio en el que el servidor ha montado un sistema de archivos, el cliente verá el directorio subyacente, en lugar del directorio montado.

11.9.5 Operaciones remotas

Con la excepción de la apertura y el cierre de archivos, existe prácticamente una correspondencia uno a uno entre las llamadas al sistema normales de UNIX que se utilizan para las operaciones de archivos y las llamadas RPC del protocolo NFS. De este modo, una operación con un archivo remoto puede traducirse directamente a la RPC correspondiente. Conceptualmente, NFS cumple con el paradigma de los servicios remotos, pero en la práctica, se utilizan técnicas de almacenamiento en búfer y en caché para mejorar las prestaciones. No existe una correspondencia directa entre una operación remota y una llamada RPC. En lugar de ello, los bloques de archivos y los atributos de los archivos son extraídos por las llamadas RPC y se almacenan en caché localmente. Las operaciones remotas subsiguientes utilizarán los datos de la caché, sujetos a las relevantes restricciones de coherencia.

Existen dos cachés: la caché de atributos de archivo (información del inodo) y la caché de bloques de archivo. Cuando se abre un archivo, el *kernel* consulta al servidor remoto para determinar si debe extraer o re-validar los atributos almacenados en caché. Los bloques de archivo almacenados en caché sólo se utilizan si los correspondientes atributos almacenados en la caché están actualizados. La caché de atributos se actualiza cada vez que llegan nuevos atributos desde el servidor. Los atributos almacenados en caché se descartan, de forma predeterminada, después de 60 segundos. Se utilizan técnicas tanto de lectura anticipada como de escritura diferida entre el servidor y el cliente. Los clientes no liberan los bloques de escritura diferida hasta que el servidor confirme que los datos se han escrito en disco. A diferencia del sistema utilizado en el sistema de archivos distribuido Sprite, los datos de escritura diferida se retienen incluso cuando un archivo se abra de manera concurrente en varios nodos conflictivos. Por tanto, la semántica UNIX descrita en la Sección 10.5.3.1 no se preserva.

La optimización de un sistema para conseguir el máximo rendimiento hace que sea difícil caracterizar la semántica de coherencia de NFS. Los nuevos archivos creados en una máquina pueden no ser visibles en otras partes de la red durante 30 segundos. Además, las escrituras realizadas en un archivo en un determinado nodo pueden o no ser visibles en otros nodos de la red que hayan abierto dicho archivo para lectura. Los procesos que abran un archivo sólo observarán aquellos cambios que ya hubieran sido volcados en el servidor antes de la apertura. Por tanto, NFS no proporciona ni una emulación estricta de la semántica de UNIX ni tampoco de la semántica de sesión de Andrew (Sección 10.5.3.2). A pesar de estas desventajas, la utilidad y las buenas prestaciones de este mecanismo hacen que sea el sistema distribuido multifabricante más ampliamente utilizado hoy en día.

11.10 Ejemplo: el sistema de archivos WAFL

La E/S de disco tiene un impacto enorme sobre las prestaciones del sistema. Como resultado, el diseño y la implementación de sistemas de archivos requieren una gran dosis de atención por parte de los diseñadores del sistema. Algunos sistemas de archivos son de propósito general, en el sentido de que pueden proporcionar una funcionalidad y unas prestaciones razonables para una amplia variedad de tamaños de archivos, tipos de archivos y cargas de E/S. Otros sistemas están optimizados para tareas específicas, en un intento de proporcionar unas mejores prestaciones que los sistemas de archivos de propósito general en esas áreas concretas. El sistema de archi-

vos WAFL de Network Appliance es un ejemplo de este tipo de optimización. WAFL (que quiere decir *write-anywhere file layout*, disposición de archivo para escritura ubicua) es un sistema de archivos potente y elegante optimizado para las escrituras aleatorias.

WAFL se utiliza exclusivamente en los servidores de archivos de red fabricados por Network Appliance y está, por tanto, pensado para usarlo como sistema de archivos distribuido. Puede proporcionar archivos a los clientes mediante los protocolos NFS, CIFS, *ftp* y *http*, aunque fue diseñado sólo para NFS y CIFS. Cuando muchos clientes utilizan estos protocolos para hablar con un servidor de archivos, el servidor puede verse sometido a una demanda muy grande de lecturas aleatorias y a una demanda todavía mayor de escrituras aleatorias. Los protocolos NFS y CIFS almacenan en caché los datos de las operaciones de lectura, por lo que las escrituras son la preocupación principal de los fabricantes de servidores de archivos.

WAFL se utiliza en servidores de archivos que incluyan una caché NVRAM para las escrituras. Los diseñadores de WAFL aprovecharon el hecho de que el sistema se ejecutaba en una arquitectura específica, con el fin de optimizar el sistema de archivos para la E/S aleatoria, empleando una caché de almacenamiento estable para las escrituras. La facilidad de uso es uno de las principales directrices de diseño de WAFL, porque fue diseñado para usarse como dispositivo autónomo. Sus creadores también diseñaron el sistema para incluir una funcionalidad de toma de instantáneas que crea múltiples copias de sólo lectura del sistema de archivos en diferentes instantes de tiempo, como veremos más adelante.

El sistema de archivos es similar al sistema Fast File System de Berkeley, con varias modificaciones. Está basado en bloques y utiliza inodos para describir los archivos. Cada inodo contiene 16 punteros a bloques (o bloques indirectos) que pertenecen al archivo descrito por el inodo. Cada sistema de archivos tiene un inodo raíz. Todos los metadatos se guardan en archivos: todos los inodos están en un archivo, el mapa de bloques libres se encuentra en otro y el mapa de inodos libres en un tercer bloque, como se muestra en la Figura 11.16. Puesto que se trata de archivos estándar, los bloques de datos no están limitados en cuanto a su ubicación y pueden colocarse en cualquier lugar. Si se expande un sistema de archivos mediante la adición de discos, el sistema de archivos expande automáticamente la longitud de estos archivos de metadatos.

Por tanto, un sistema de archivos WAFL es un árbol de bloques cuya raíz es el inodo raíz. Para tomar una instantánea, WAFL crea un inodo raíz duplicado. Todas las actualizaciones de archivos o metadatos subsiguientes se realizan en bloques nuevos, en lugar de sobreescribiendo los bloques existentes. El nuevo inodo raíz apunta a los metadatos y a los datos modificados como resultado de estas escrituras. Mientras tanto, el antiguo inodo raíz sigue apuntando a los bloques antiguos, que no habrán sido actualizados. Por tanto, proporciona acceso al sistema de archivos tal como era en el momento de tomar la instantánea y proporciona esta funcionalidad consumiendo muy poco espacio de disco. En esencia, el espacio de disco adicional ocupado por una instantánea está compuesto simplemente por los bloques que hayan sido modificados desde que fue tomada la instantánea.

Un cambio importante con respecto a los sistemas de archivos más estándar es que el mapa de bloques libres tiene más de un bit por bloque. Se trata de un mapa de bits en el que se activa un bit por cada instantánea que esté utilizando el bloque. Cuando se borren todas las instantáneas que hayan estado utilizando el bloque, el mapa de bits correspondiente a dicho bloque estará for-

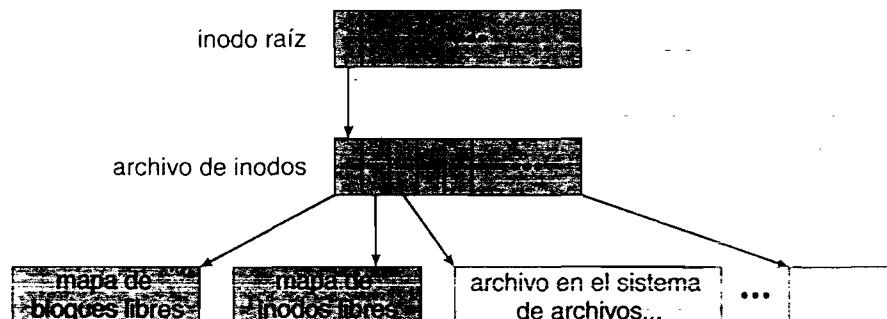


Figura 11.16 Disposición de archivos en WAFL.

las propiedades lógicas de los archivos. Los niveles intermedios mapean los conceptos de archivos lógicos sobre las propiedades de los dispositivos físicos.

Cualquier tipo de sistema de archivos puede tener diferentes estructuras y algoritmos. Un nivel VFS permite que los niveles superiores traten con cada tipo de sistema de archivos de manera uniforme. Incluso pueden integrarse sistemas de archivos remotos dentro de la estructura de directorios del sistema y se podrá operar con esos sistemas de archivos mediante llamadas al sistema estándar, a través de la interfaz VFS.

Para asignar espacio del disco a los diversos archivos pueden utilizarse tres mecanismos: asignación contigua, enlazada o indexada. El mecanismo de asignación contigua puede sufrir del problema de la fragmentación externa. El acceso directo es muy poco eficiente con el mecanismo de asignación enlazada. El mecanismo de asignación indexada puede requerir un gasto adicional considerable, ya que hace falta disponer de un bloque de índice. Estos algoritmos pueden optimizarse de muchas formas. El espacio contiguo puede agrandarse utilizando extensiones, con el fin de aumentar la flexibilidad y reducir el grado de fragmentación externa. La asignación indexada puede llevarse a cabo en *clusters* formados por múltiples bloques, con el fin de incrementar la tasa de transferencia y para reducir el número de entradas de índice necesarias. La indexación mediante *clusters* de gran tamaño es similar al mecanismo de asignación contigua con extensiones.

Los métodos de asignación del espacio libre también influyen sobre la eficiencia de la utilización del espacio de disco, sobre las prestaciones del sistema de archivos y sobre la fiabilidad del almacenamiento secundario. Los métodos usados incluyen los vectores de bits y las vistas enlazadas. Entre las posibles optimizaciones están el agrupamiento, la utilización de contadores y la tabla FAT, que sitúa la lista enlazada en un área de disco contigua.

Las rutinas de gestión de directorios deben tener en cuenta las consideraciones de eficiencia, prestaciones y fiabilidad. Uno de los métodos comúnmente utilizados son las tablas *hash*, ya que son rápidas y eficientes. Desafortunadamente, si la tabla sufre algún daño o el sistema tiene un fallo catastrófico, pueden aparecer incoherencias entre la información de directorios y el contenido del disco; puede utilizarse una comprobación de coherencia para reparar esos daños. Las herramientas de copia de seguridad del sistema operativo permiten copiar los datos del disco en una cinta, con lo que el usuario podrá recuperarse de la pérdida de datos o incluso de un disco completo debido a fallos del hardware, a errores del sistema operativo o a errores de los usuarios.

Los sistemas de archivos en red, como NFS, utilizan metodologías clientes-servidor para que los usuarios puedan acceder a archivos y directorios situados en máquinas remotas como si éstos se encontraran en sistemas de archivos locales. Las llamadas al sistema realizadas en el cliente se traducen a protocolos de red y se vuelven a traducir en operaciones del sistema de archivos dentro del servidor. La interconexión en red y el acceso simultáneo por parte de múltiples clientes constituyen serios desafíos en lo que respecta a la coherencia de los datos y a las prestaciones del sistema.

Debido al papel fundamental que los sistemas de archivos juegan en la operación del sistema, sus prestaciones y su fiabilidad resultan cruciales. Determinadas técnicas, como las estructuras de registro y el almacenamiento en caché, ayudan a mejorar las prestaciones, mientras que las escrituras de registro y las tecnologías RAID mejoran la fiabilidad. El sistema de archivos WAFL es un ejemplo de optimización del rendimiento con el fin de ajustarse a una carga de E/S específica.

Ejercicios

- 11.1 Considere un sistema de archivos que utilice un esquema modificado de asignación contigua con soporte para extensiones. Cada archivo será una colección de extensiones, correspondiendo cada extensión a un conjunto contiguo de bloques. Una cuestión clave en tales sistemas es el grado de variabilidad en el tamaño de las extensiones. ¿Cuáles son las ventajas y desventajas de los siguientes esquemas?
- Todas las extensiones tienen el mismo tamaño y ese tamaño está predeterminado.
 - Las extensiones pueden tener cualquier tamaño y se asignan dinámicamente.

- c. Las extensiones pueden tener unos cuantos tamaños fijos, y estos tamaños están pre-determinados.
- 11.2 ¿Cuáles son las ventajas de la variante de la asignación enlazada que utiliza una tabla FAT para encadenar los bloques de un archivo?
- 11.3 Considere un sistema en el que el espacio libre se controla mediante una lista de espacio libre.
- Suponga que se pierde el puntero a la lista de espacio libre. ¿Puede el sistema reconstruir la lista de espacio libre? Razone su respuesta.
 - Considere un sistema de archivos similar al utilizado por UNIX, con asignación indexada. ¿Cuántas operaciones de E/S de disco pueden requerirse para leer el contenido de un pequeño archivo local situado en /a/b/c? Suponga que ninguno de los bloques del disco está actualmente almacenado en la caché.
 - Sugiera un esquema para garantizar que el puntero nunca se pierda como resultado de un fallo de memoria.
- 11.4 Algunos sistemas de archivos permiten asignar el espacio de almacenamiento en disco con diferentes niveles de granularidad. Por ejemplo, un sistema de archivos podría asignar 4 KB de espacio de disco como un único bloque de 4 KB o como ocho bloques de 512 bytes. ¿Cómo podríamos aprovechar esta flexibilidad para mejorar las prestaciones? ¿Qué modificaciones habría que hacer en el esquema de gestión del espacio libre para soportar esta característica?
- 11.5 Explique por qué las optimizaciones de rendimiento en los sistemas de archivos puede provocar dificultades a la hora de mantener la coherencia de los sistemas en caso de fallos catastróficos de la computadora.
- 11.6 Considere un sistema de archivos en un disco que tenga tamaños de bloque tanto lógico como físico de 512 bytes. Suponga que la información acerca de cada archivo ya se encuentra en memoria. Para cada una de las tres estrategias de asignación (contigua, enlazada e indexada), responda a las siguientes cuestiones:
- ¿Cómo puede realizarse el mapeo de direcciones lógicas sobre las direcciones físicas en este sistema? (Para la asignación indexada, suponga que un archivo tiene siempre menos de 512 bloques de longitud).
 - Si nos encontramos actualmente en el bloque lógico 10 (el último bloque al que se ha accedido es el bloque 10) y queremos acceder al bloque lógico 4, ¿cuántos bloques físicos habrá que leer del disco?
- 11.7 La fragmentación en un dispositivo de almacenamiento puede eliminarse recompactando la información. Los dispositivos de disco típicos no tienen registros de reubicación ni registros base (como los que se usan cuando hay que compactar la memoria), de modo que ¿cómo podemos reubicar los archivos? Proporcione tres razones por las que la recompactación y reubicación de archivos no suelen emplearse.
- 11.8 ¿En qué situaciones sería más útil emplear la memoria como disco RAM que como caché de disco?
- 11.9 Considere la siguiente extensión de un protocolo de acceso a archivos remotos: cada cliente mantiene una caché de nombres donde se almacenan las traducciones entre los nombres de archivo y los descriptores de archivos correspondientes. ¿Qué cuestiones deberíamos tener en cuenta a la hora de implementar la caché de nombres?
- 11.10 Explique por qué el mecanismo de registro de las actualizaciones de los metadatos garantiza la recuperación de un sistema de archivos después de que el sistema de archivos sufra un fallo catastrófico.
- 11.11 Considere el siguiente esquema de realización de copias de seguridad:

- **Día 1.** Copia en un soporte de copia de seguridad de todos los archivos del disco.
- **Día 2.** Copia en otro soporte de todos los archivos modificados desde el día 1.
- **Día 3.** Copia en otro soporte de todos los archivos modificados desde el día 1.

Este mecanismo difiere de la planificación proporcionada en la Sección 11.7.2, al hacer que todas las copias de seguridad subsiguientes copien todos los archivos modificados desde la primera copia de seguridad completa. ¿Cuáles son las ventajas de este sistema sobre el expuesto en la Sección 11.7.2? ¿Cuáles son las desventajas? ¿Son más fáciles o más difíciles las operaciones de restauración? Razoné su respuesta.

Notas bibliográficas

El sistema FAT de MS-DOS se explica en Norton y Wilton [1988] y la descripción correspondiente a OS/2 puede encontrarse en Jacobucci [1988]. Estos sistemas operativos usan la CPU Intel 8086 (Intel [1985b], Intel [1985a], Intel [1986], Intel [1990]). Los métodos de asignación utilizados por IBM se describen en Deitel [1990]. Los detalles internos relativos al sistema BSD UNIX se cubren en detalle en McKusick et al. [1996]. McVoy y Kleiman [1991] presenta las optimizaciones de estos métodos que se han realizado en Solaris.

La asignación de archivos de disco basada en el sistema de descomposición binaria se analiza en Koch [1987]. Un esquema de organización de archivos que garantiza la extracción en un único acceso se presenta en Larson y Kajla [1984]. Las organizaciones de archivos con estructura de registro para la mejora de las prestaciones y de la coherencia se exponen en Rosenblum y Ousterhout [1991], Seltzer et al. [1993] y Seltzer et al. [1995].

Los mecanismos de caché de disco se tratan en McKeon [1985] y Smith [1985]. Los mecanismos de caché en el sistema operativo experimental Sprite se describen en Nelson et al. [1988]. Una serie interesante de análisis generales relativos a las tecnologías de almacenamiento masivo son los de Chi [1982] y Hoagland [1985]. Folk y Zoellick [1987] analizan la gama de estructuras de archivos existentes. Silvers [2000] explica la implementación de la caché de páginas en el sistema operativo NetBSD.

El sistema de archivos de red (NFS) se expone en Sandberg et al. [1985], Sandberg [1987], Sun [1990] y Callaghan [2000]. Las características de las cargas de trabajo en los sistemas de archivos distribuidos fueron estudiadas por Baker et al. [1991]. Ousterhout [1991] analiza el papel de la información de estado distribuida en los sistemas de archivos en red. Los diseños con estructura de registro para los sistemas de archivos en red fueron propuestos en Hartman y Ousterhout [1995] y Thekkath et al. [1997]. NFS y el sistema de archivos UNIX (UFS) se describen en Vahalia [1996] y Mauro y McDougall [2001]. El sistema de archivos Windows NT, NTFS, se explica en Solomon [1998]. El sistema de archivos Ext2 utilizado en Linux se describe en Bovet y Cesati [2002] y el sistema de archivos WAFL en Hitz et al. [1995].

Estructura de almacenamiento masivo

El sistema de archivos puede considerarse, desde el punto de vista lógico, como compuesto de tres partes. En el Capítulo 10, hemos visto la interfaz del usuario y el programador con el sistema de archivos. En el Capítulo 11, hemos descrito las estructuras de datos y algoritmos internos utilizados por el sistema operativo para implementar esta interfaz. En este capítulo, vamos a analizar el nivel inferior del sistema de archivos: las estructuras de almacenamiento secundario y terciario. Primero vamos a describir la estructura física de los discos magnéticos y las cintas magnéticas. Después, describiremos los algoritmos de planificación de disco que se encargan de organizar el orden de las operaciones de E/S de disco con el fin de mejorar el rendimiento. A continuación, analizaremos el formato de los discos y la gestión de los bloques de inicio, de los bloques dañados y del espacio de intercambio. Después examinaremos la estructura de almacenamiento secundario, tratando tanto la fiabilidad de los discos como la implementación de los mecanismos de almacenamiento estable. Concluiremos con una breve descripción de los dispositivos de almacenamiento terciario y de los problemas que surgen cuando un sistema operativo utiliza almacenamiento terciario.

OBJETIVOS DEL CAPÍTULO

- Describir la estructura física de los dispositivos de almacenamiento secundario y terciario y los efectos resultantes sobre la utilización de estos dispositivos.
- Explicar las características de los dispositivos de almacenamiento masivo, en lo que respecta a las prestaciones.
- Analizar los servicios que el sistema operativo proporciona para el almacenamiento masivo, incluyendo RAID y HSM.

12.1 Panorámica de la estructura de almacenamiento masivo

En esta sección, vamos a presentar una panorámica general de la estructura física de los dispositivos de almacenamiento secundario y terciario.

12.1.1 Discos magnéticos

Los **discos magnéticos** proporcionan la parte principal del almacenamiento secundario en los modernos sistemas informáticos. Desde el punto de vista conceptual, los discos son relativamente simples (Figura 12.1). Cada **plato** tiene una forma circular plana, como un CD. Los diámetros comunes de los platos van de 1,8 a 5,25 pulgadas. Las dos superficies de cada plato están recubiertas de un material magnético. La información se almacena grabándola magnéticamente sobre los platos.

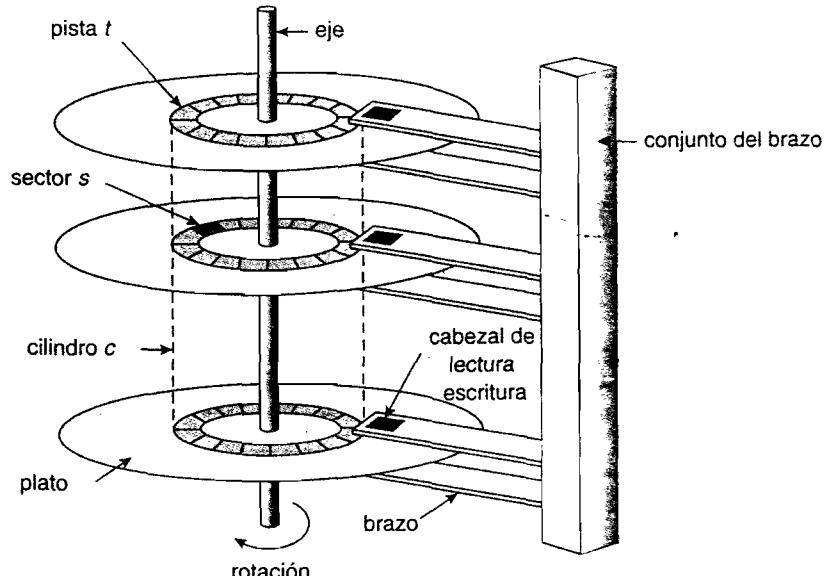


Figura 12.1 Mecanismo de cabezal móvil del disco.

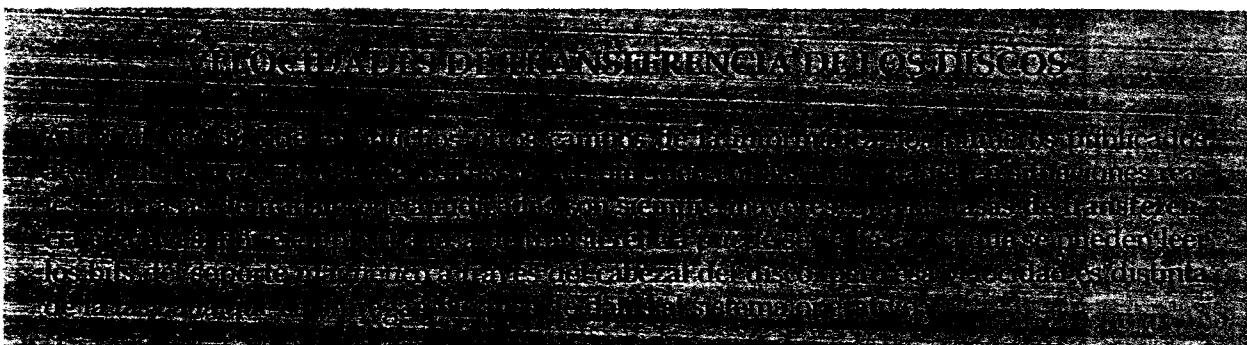
Un cabezal de lectura-escritura “vuela” justo por encima de cada una de las superficies de cada plato. Los cabezales están conectados a un **brazo del disco** que mueve todos los cabezales como una sola unidad. La superficie de cada plato está dividida desde el punto de vista lógico en **pistas** circulares, que a su vez se subdividen en **sectores**. El conjunto de las pistas que están situadas en una determinada posición del brazo forman un **cilindro**. Puede haber miles de cilindros concéntricos en una unidad de disco y cada pista puede contener cientos de sectores. La capacidad de almacenamiento de las unidades de disco comunes se mide en gigabytes.

Cuando se está usando el disco, un motor hace que gire a gran velocidad. La mayoría de los motores rotan entre 60 y 200 veces por segundo. La velocidad de un disco puede considerarse compuesta por dos partes diferenciadas: la **velocidad de transferencia** es la velocidad con la que los datos fluyen entre la unidad de disco y la computadora. El **tiempo de posicionamiento**, a veces denominado **tiempo de acceso aleatorio**, está compuesto del tiempo necesario para mover el brazo del disco hasta el cilindro deseado, denominado **tiempo de búsqueda**, y el tiempo requerido para que el sector deseado rote hasta pasar por debajo del cabezal del disco, denominado **latencia rotacional**. Los discos típicos pueden transferir varios megabytes de datos por segundo, y tienen tiempos de búsqueda y latencias rotacionales de varios milisegundos.

Puesto que el cabezal del disco vuela sobre un colchón de aire extremadamente fino (que se mide en micras), existe el peligro de que el cabezal entre en contacto con la superficie del disco. Aunque los platos del disco están recubiertos de una fina capa protectora, a veces el cabezal puede llegar a dañar la superficie magnética; este accidente se denomina **aterrizaje de cabezales**. Normalmente, los aterrizajes de cabezales no pueden repararse, siendo necesario sustituir el disco completo.

Un disco puede ser **extraíble**, lo que permite montar diferentes discos según sea necesario. Los discos magnéticos extraíbles constan generalmente de un solo plato, albergado en una carcasa plástica para impedir que resulte dañado mientras no se encuentre dentro de la unidad de disco. Los **disquetes** son discos magnéticos extraíbles de bajo coste que tienen una carcasa de plástico blando que contiene un plato flexible. El cabezal de una unidad de disquete se apoya generalmente de forma directa sobre la superficie del disco, por lo que la unidad está diseñada para rotar más lentamente que una unidad de disco duro, con el fin de reducir el desgaste de la superficie. La capacidad de almacenamiento de un disquete es típicamente de sólo 1,44 MB. Hay disponibles discos extraíbles que funcionan de forma bastante similar a los discos duros normales y cuyas capacidades se miden en gigabytes.

Para conectar una unidad de disco a una computadora, se utiliza un conjunto de cables denominado **bus de E/S**. Hay disponibles varios tipos de buses, incluyendo **EIDE** (enhanced integra-



ted drive electronics), ATA (advanced technology attachment), ATA serie (SATA), USB (universal serial bus), fiber channel (FC) y SCSI. Las transferencias de datos en un bus son realizadas por procesadores electrónicos especiales denominados **controladoras**. La **controladora host** es la controladora situada en el extremo del bus correspondiente a la computadora; además, en cada unidad de disco se integra una **controladora de disco**. Para realizar una operación de E/S de disco, la computadora coloca un comando en la controladora *host*, normalmente utilizando puertos de E/S mapeados en memoria, como se describe en la Sección 9.7.3. La controladora *host* envía entonces el comando mediante una serie de mensajes a la controladora de disco y ésta hace operar el hardware de la unidad de disco para ejecutar el comando. Las controladoras de disco suelen disponer de una caché integrada. Las transferencias de datos en la unidad de disco tienen lugar entre la caché y la superficie del disco y la transferencia de datos hacia el *host*, a velocidades electrónicas muy altas, se produce entre la caché y la controladora *host*.

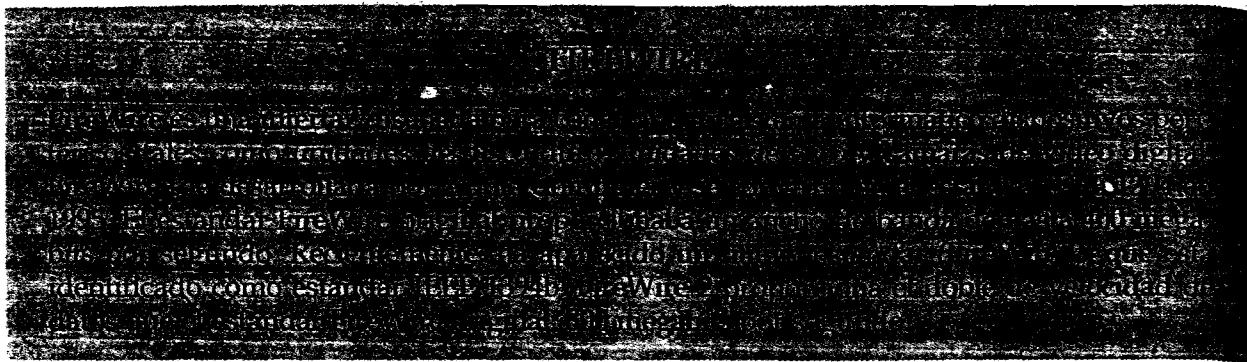
12.1.2 Cintas magnéticas

La **cinta magnética** se solía utilizar hace tiempo como soporte de almacenamiento secundario. Aunque es relativamente permanente y puede albergar grandes cantidades de datos, su tiempo de acceso es lento comparado con el de la memoria principal y el disco magnético. Además, el acceso aleatorio a cinta magnética es unas 1000 veces más lento que el acceso aleatorio a un disco magnético, por lo que las cintas no resultan muy útiles como almacenamiento secundario. Las cintas se utilizan principalmente para copias de seguridad, para el almacenamiento de datos que se utilicen de forma infrecuente y como medio para transferir información de un sistema a otro.

Las cintas se insertan en una unidad de cinta y se bobinan o rebobinan para hacerlas pasar por debajo de un cabezal de lectura-escritura. Para desplazarse al punto correcto de la cinta pueden hacer falta minutos, pero una vez que hemos posicionado la cinta, las unidades de cinta pueden escribir datos a velocidades comparables a las de las unidades de disco. Las capacidades de las cintas varían enormemente, dependiendo del tipo concreto de unidad de cinta. Normalmente, pueden almacenar entre 20 GB y 200 GB. Algunas unidades de cinta tienen mecanismos de compresión integrados que permiten doblar con creces la capacidad de almacenamiento efectiva. Las cintas y sus unidades suelen clasificarse según su anchura, incluyendo las cintas de 4, 8 y 19 milímetros y de 1/4 y 1/2 pulgada. Algunas se denominan de acuerdo con la tecnología empleada, como por ejemplo LTO-2 y SDLT. El almacenamiento en cinta se describe con más detalle en la Sección 12.9.

12.2 Estructura de un disco

Las unidades de disco modernas se direccionan como grandes matrices unidimensionales de **bloques lógicos**, en las que el bloque lógico es la unidad más pequeña de transferencia. El tamaño de un bloque lógico es usualmente de 512 bytes, aunque algunos discos pueden formatearse a bajo nivel para tener un tamaño de bloque lógico distinto, como por ejemplo 1024 bytes; esta opción se describe en la Sección 12.5.1. La matriz unidimensional de bloques lógicos se mapea sobre los sectores del disco secuencialmente. El sector 0 es el primer sector de la primera pista del cilindro



más externo y el mapeo continua por orden a lo largo de dicha pista, después a lo largo del resto de las pistas de dicho cilindro y luego a lo largo del resto de los cilindros, desde el más externo al más interno.

Utilizando este mapeo, podemos (al menos en teoría) convertir un número de bloque lógico en una dirección de disco a la antigua usanza, compuesta por un número de cilindro, el número de pista dentro de dicho cilindro y el número de sector dentro de dicha pista. Sin embargo, en la práctica, resulta difícil realizar esta traducción, por dos razones distintas. En primer lugar, la mayoría de los discos tienen algunos sectores defectuosos, pero el proceso de mapeo oculta este hecho utilizando en su lugar sectores libres adicionales situados en algún otro lugar del disco. En segundo lugar, el número de sectores por pista no es constante en algunas unidades de disco.

Examinemos con más detalle esta segunda razón. En los soportes físicos que utilizan una **velocidad lineal constante** (CLV, constant linear velocity), la densidad de bits por cada pista es uniforme. Cuanto más lejos esté una pista del centro del disco, mayor será su longitud, por lo que podrá albergar más sectores. A medida que vamos pasando de los más externos a los más internos, el número de sectores por pista se reduce. Las pistas en la zona más externa suelen tener un 40 por ciento más de sectores que las pistas situadas en la zona más interna. La unidad de disco incrementa su velocidad de rotación a medida que el cabezal se mueve de las pistas externas a las internas, con el fin de que el cabezal transfiera siempre la misma cantidad de datos por segundo. Este método se utiliza en las unidades de CD-ROM y DVD-ROM. Alternativamente, la velocidad de rotación del disco puede permanecer constante y la densidad de bits se reduce al pasar de las pistas internas a las externas, con el fin de mantener constante la velocidad de transferencia de datos. Este método se utiliza en los discos duros y se conoce como **velocidad angular constante** (CAV, constant angular velocity).

El número de sectores por pista se ha ido incrementando a medida que mejoraba la tecnología de los discos, y la zona exterior de un disco suele tener varios cientos de sectores por pista. De forma similar, también el número de cilindros por disco se ha ido incrementando y los discos de mayor tamaño tienen decenas de miles de cilindros.

12.3 Conexión de un disco

Las computadoras acceden a los datos almacenados en disco de dos formas distintas. Una manera es mediante puertos de E/S (o **almacenamiento conectado al host**); esta solución resulta bastante común en los sistemas de pequeño tamaño. La otra forma es mediante un *host* remoto en un sistema de archivos distribuido; esto se denomina **almacenamiento conectado a la red**.

12.3.1 Almacenamiento conectado al host

El almacenamiento conectado al *host* es aquel al que se accede a través de puertos de E/S locales. Estos puertos utilizan diversas tecnologías. El PC típico de sobremesa emplea una arquitectura de bus de E/S denominada IDE o ATA. Esta arquitectura soporta un máximo de dos unidades por cada bus de E/S. Un protocolo similar y más reciente que tiene un cableado simplificado es SATA.

Los servidores y estaciones de trabajo de gama alta utilizan generalmente arquitecturas de E/S más sofisticadas, como SCSI y fiber channel (FC).

SCSI es una arquitectura de bus. Su soporte físico es, usualmente, un cable de cinta que tiene un gran número de conductores (normalmente 50 o 68). El protocolo SCSI soporta un máximo de 16 dispositivos conectados al bus. Generalmente, los dispositivos incluyen una tarjeta controladora en el *host* (el iniciador SCSI) y hasta 15 dispositivos de almacenamiento (los destinos SCSI). Un destino SCSI bastante común es un disco SCSI, pero el protocolo proporciona la capacidad de dirigir hasta 8 unidades lógicas en cada destino SCSI. Una utilización típica del mecanismo de direccionamiento de unidades lógicas consiste en dirigir comandos a los componentes de una matriz RAID o a los componentes de una biblioteca de soportes extraíbles [como un intercambiador (jukebox) de CD que envíe comandos al mecanismo de cambios de soportes o a una de las unidades].

FC es una arquitectura serie de alta velocidad que puede operar sobre fibra óptica o sobre un cable de cobre de cuatro hilos. Tiene dos variantes: una de ellas es una estructura conmutada de gran tamaño que tiene un espacio de direcciones de 24 bits; la expectativa es que esta variante sea la que domine en el futuro y esta tecnología forma la base de las redes de áreas de almacenamiento (SAN, storage-area network), de las que hablaremos en la Sección 12.3.3. Debido al gran espacio de direcciones y a la naturaleza conmutada de las comunicaciones, pueden conectarse múltiples *hosts* y dispositivos de almacenamiento a la estructura de comunicaciones, lo que proporciona una gran flexibilidad en las comunicaciones de E/S. La otra variante de FC es un bucle arbitrado (FC-AL, arbitrated loop) que puede direccionar 126 dispositivos (unidades y controladoras).

Hay una amplia variedad de dispositivos de almacenamiento que resultan adecuados para utilizarlos como almacenamiento conectado al *host*. Entre ellos podemos citar las unidades de disco duro, las matrices RAID y las unidades de CD, DVD y cinta. Los comandos de E/S que inician las transferencias de datos a un dispositivo de almacenamiento conectado al *host* son las lecturas y escrituras de bloques de datos lógicos dirigidas hacia unidades de almacenamiento específicamente identificadas (como por ejemplo ID de bus, ID SCSI y unidad lógica de destino).

12.3.2 Almacenamiento conectado a la red

Un dispositivo de almacenamiento conectado a la red (NAS, network-attached storage) es un sistema de almacenamiento de propósito especial al que se accede de forma remota a través de una red de datos (Figura 12.2). Los clientes acceden al almacenamiento conectado a la red a través de una interfaz de llamadas a procedimientos remotos, como por ejemplo NFS para los sistemas UNIX o CIFS para las máquinas Windows. Las llamadas a procedimientos remotos (RPC, remote procedure call), se realizan mediante los protocolos TCP o UDP sobre una red IP, usualmente la misma red de área local (LAN, local area network) que transporta todo el tráfico de datos hacia los clientes. La unidad de almacenamiento conectado a la red se implementa usualmente como una matriz RAID con un software que proporciona la interfaz RPC. Lo más fácil es considerar NAS simplemente como otro protocolo más de acceso al almacenamiento. Por ejemplo, en lugar de utilizar un controlador de dispositivo SCSI y los protocolos SCSI para acceder al almacenamiento, un sistema que utilice NAS emplearía RCP sobre TCP/IP.

El almacenamiento conectado a la red proporciona una forma conveniente para que todas las computadoras de una LAN compartan un conjunto de dispositivos de almacenamiento con la misma facilidad de denominación y de acceso que si se tratara de dispositivos de almacenamiento locales conectados al host. Sin embargo, este sistema tiende a ser menos eficiente y proporciona menos velocidad que algunas opciones de almacenamiento de conexión directa.

El protocolo más reciente de almacenamiento conectado a la red es iSCSI. En esencia, utiliza el protocolo de red IP para transportar el protocolo SCSI. Así, pueden utilizarse redes en lugar de cables SCSI como mecanismo de interconexión entre los *hosts* y los dispositivos de almacenamiento. Como resultado, los *hosts* pueden tratar esos dispositivos como si estuvieran conectados directamente, pero esos dispositivos pueden estar situados a una gran distancia del *host*.

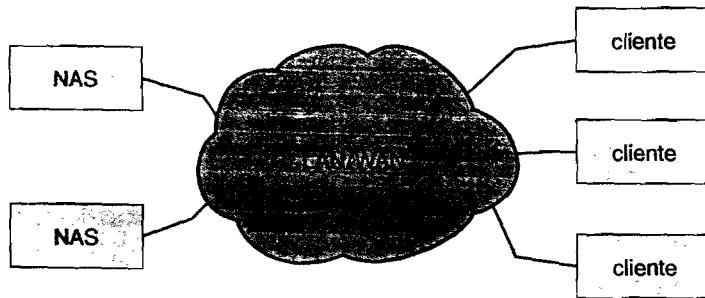


Figura 12.2 Almacenamiento conectado a la red.

12.3.3 Redes de área de almacenamiento

Una de las desventajas de los sistemas de almacenamiento conectados a la red es que las operaciones de E/S para acceso a los dispositivos consumen ancho de banda de la red de datos, incrementando así la latencia de las comunicaciones a través de red. Este problema puede ser particularmente grave en las instalaciones cliente-servidor de gran envergadura, ya que las comunicaciones entre servidores y clientes compiten por el ancho de banda con las comunicaciones existentes entre los servidores y los dispositivos de almacenamiento.

Una red de área de almacenamiento (SAN, storage-area network) es una red privada (que utiliza protocolos de almacenamiento en lugar de protocolos de red) que conecta los servidores con las unidades de almacenamiento, como se muestra en la Figura 12.3. La principal ventaja de una SAN radica en su flexibilidad, ya que pueden conectarse múltiples *host* y múltiples matrices de almacenamiento a la misma SAN y los recursos de almacenamiento pueden asignarse de forma dinámica a los *hosts*. Un conmutador de la SAN permite o prohíbe el acceso de los *hosts* a los dispositivos de almacenamiento. Como ejemplo, si un *host* se está quedando sin espacio de disco, puede configurarse la SAN para asignar más espacio de almacenamiento a dicho *host*. Las redes SAN consiguen que los *clusters* de servidores puedan compartir los mismos dispositivos de almacenamiento y permiten que las matrices de almacenamiento incluyan múltiples conexiones directas a los *hosts*. Las redes SAN tienen normalmente más puertos (y además más baratos) que las matrices de almacenamiento. El mecanismo de interconexión más utilizado en las redes SAN es FC.

Una alternativa emergente es una arquitectura de bus de propósito especial denominada InfiniBand, que proporciona soporte hardware y software para redes de conexión de alta velocidad entre servidores y unidades de almacenamiento.

12.4 Planificación de disco

Una de las responsabilidades del sistema operativo es la utilizar de manera eficiente el hardware disponible. Para las unidades de disco, cumplir con esta responsabilidad implica tener un tiempo de acceso rápido y un gran ancho de banda de disco. El tiempo de acceso tiene dos componentes principales (véase también la Sección 12.1.1). El **tiempo de búsqueda** es el tiempo requerido para que el brazo del disco mueva los cabezales hasta el cilindro que contiene el sector deseado. La **latencia rotacional** es el tiempo adicional necesario para que el disco rote y sitúe el sector deseado bajo el cabezal del disco. El **ancho de banda** del disco es el número total de bytes transferidos, dividido entre el tiempo total transcurrido entre la primera solicitud de servicio y la terminación de la última transferencia. Podemos mejorar tanto el tiempo de acceso como el ancho de banda planificando en el orden adecuado el servicio de las distintas solicitudes de E/S de disco.

Cada vez que un proceso necesita realizar una operación de E/S hacia o desde el disco, ejecuta una llamada al sistema operativo. La solicitud especifica varios elementos de información:

- Si esta operación es una entrada o una salida.
- Cuál es la dirección de disco para la transferencia.

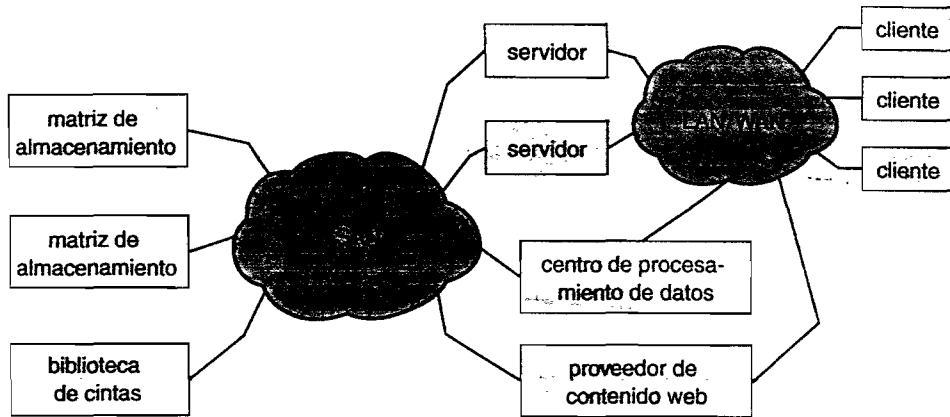


Figura 12.3 Red de área de almacenamiento.

- Cuál es la dirección de memoria para la transferencia.
- Cuál es el número de sectores que hay que transferir.

Si están disponibles la unidad de disco y la controladora deseadas, puede darse servicio a la solicitud de manera inmediata. Si la unidad y la controladora están ocupadas, las nuevas solicitudes de servicio se colocarán en la cola de solicitudes pendientes para dicha unidad. Para un sistema de multiprogramación con múltiples procesos, la cola del disco puede tener a menudo varias solicitudes pendientes. Así, cuando se completa una solicitud, el sistema operativo selecciona cuál es la solicitud pendiente a la que hay que dar servicio a continuación. ¿Cómo realiza esta elección el sistema operativo? Pueden usarse varios algoritmos de planificación de disco, de los cuales vamos a hablar a continuación.

12.4.1 Planificación FCFS

La forma más simple de planificación de disco es, por supuesto, el algoritmo FCFS (first-come, first-served; primero en llegar, primero en ser servido). Este algoritmo es intrínsecamente equitativo, pero generalmente no proporciona el tipo de servicio más rápido. Considere, por ejemplo, una cola de disco en la que hubiera una serie de solicitudes de E/S dirigidas a ciertos bloques en los cilindros

98, 183, 37, 122, 14, 124, 65, 67,

por ese orden. Si el cabezal del disco está inicialmente en el cilindro 53, se moverá primero del 53 al 98, luego al 183, al 37, al 122, al 14, al 124, al 65 y finalmente al 67, lo que da una cantidad total de movimiento de los cabezales equivalente a 640 cilindros. Esta planificación se ilustra en la Figura 12.4.

La amplísima oscilación desde el cilindro 122 al 14 y luego al 124 ilustra perfectamente el problema existente con esta planificación. Si se pudiera satisfacer de forma sucesiva las solicitudes relativas a los cilindros 37 y 14, antes o después de las solicitudes correspondientes a los cilindros 122 y 124, la cantidad total de movimiento de los cabezales podría reducirse sustancialmente y la velocidad mejoraría de forma correspondiente.

12.4.2 Planificación SSTF

Parece razonable satisfacer todas las solicitudes que estén próximas a la situación actual del cabezal antes de desplazar el cabezal hasta una posición muy alejada para dar servicio a otras solicitudes. Esta suposición es la base para el **algoritmo del tiempo de búsqueda más corto** (SSTF, shortest-seek-time-first). El algoritmo SSTF selecciona la solicitud que tenga el tiempo de búsqueda mínimo con respecto a la posición actual del cabezal. Puesto que el tiempo de búsqueda se

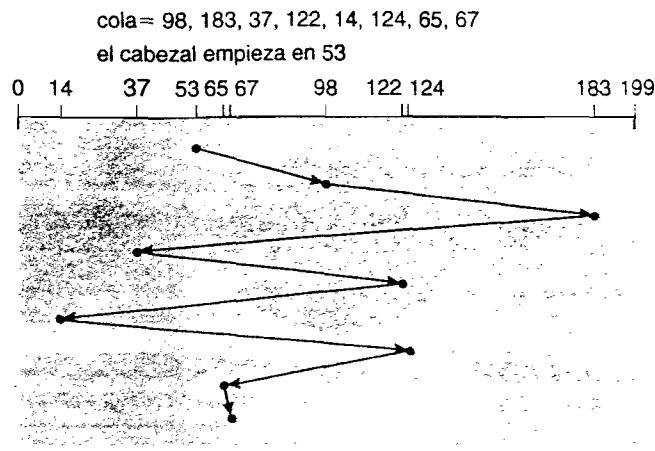


Figura 12.4 Planificación FCFS del disco.

incrementa a medida que lo hace el número de cilindros recorridos por el cabezal, SSTF selecciona la solicitud pendiente que esté más próxima a la posición actual del cabezal.

Para nuestra cola de solicitudes de ejemplo, la solicitud más próxima a la posición inicial del cabezal (53) es la correspondiente al cilindro 65. Una vez situados en el cilindro 65, la solicitud siguiente más próxima es la del cilindro 67. A partir de ahí, la solicitud del cilindro 37 está más próxima que la del 98, por lo que se da servicio primero a la solicitud correspondiente al 37. Continuando con este proceso, se da servicio a las solicitudes correspondientes a los cilindros 14, 98, 122, 124 y finalmente 183 (Figura 12.5). Este método de planificación proporciona un movimiento total de los cabezales de sólo 236 cilindros, es decir, un poco más de la tercera parte de la distancia que el algoritmo de planificación FCFS necesitaba para dar servicio a esta cola de solicitudes. Este algoritmo proporciona una mejora sustancial de las prestaciones.

La planificación SSTF es, esencialmente, un tipo de planificación SJF (shortest-job-first, primero el trabajo más corto) y, al igual que la planificación SJF, puede provocar la muerte por inanición de algunas solicitudes. Recuerde que las solicitudes pueden llegar en cualquier momento. Suponga que tenemos dos solicitudes en la cola, referidas a los cilindros 14 y 186 y que, mientras se está dando servicio a la solicitud correspondiente al cilindro 14, llega una nueva solicitud próxima a ese cilindro. Esta nueva solicitud será la que sea atendida a continuación, haciendo que la solicitud correspondiente al 186 tenga que esperar. Mientras que esta nueva solicitud está siendo servida, puede llegar otra solicitud próxima al cilindro 14 y, en teoría, podría seguir llegando un flujo continuo de solicitudes muy próximas, haciendo que la solicitud correspondiente al cilindro

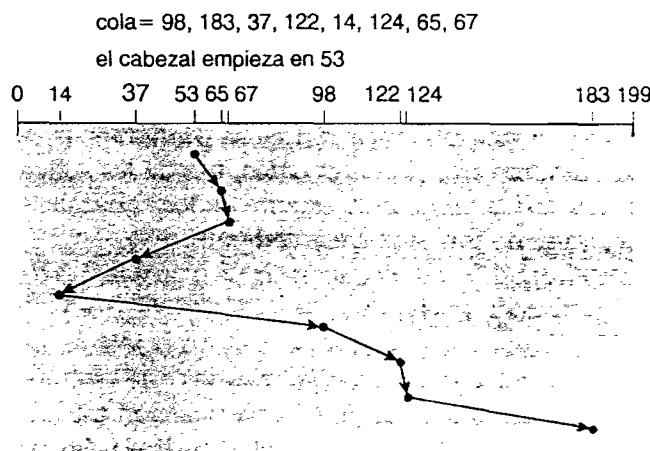


Figura 12.5 Planificación SSTF de disco.

186 tuviera que esperar indefinidamente. Este escenario es tanto más probable cuanto más larga sea la cola de solicitudes pendiente.

Aunque el algoritmo SSTF representa una mejora sustancial con respecto al algoritmo FCFS, no es un algoritmo óptimo. En nuestro ejemplo, podríamos conseguir mejores resultados moviendo el cabezal desde el cilindro 53 al 37, aun cuando este último no sea el más próximo, y luego al cilindro 14, antes de dar la vuelta y prestar servicio a las solicitudes correspondientes a los cilindros 65, 67, 98, 122, 124 y 183. Esta estrategia reduce el movimiento total de los cabezales a 208 cilindros.

12.4.3 Planificación SCAN

En el **algoritmo SCAN** (o algoritmo de exploración), el grafo del disco comienza en uno de los extremos del disco y se mueve hacia el otro extremo, dando servicio a las solicitudes a medida que pasa por cada cilindro, hasta llegar al otro extremo del disco. En ese otro extremo, se invierte la dirección de movimiento del cabezal y se continua dando servicio a las solicitudes. El cabezal está continuamente explorando el disco en una y otra dirección. El algoritmo SCAN se denomina también en ocasiones **algoritmo del ascensor** ya que el brazo del disco se comporta como un ascensor dentro de un edificio, primero dando servicio a todas las solicitudes de subida y luego invirtiendo su movimiento para dar servicio a las solicitudes de bajada.

Volvamos a nuestro ejemplo para ilustrar el procedimiento. Antes de aplicar el algoritmo SCAN para planificar las solicitudes relativas a los cilindros 98, 183, 37, 122, 14, 124, 65 y 67, necesitamos conocer la dirección del movimiento del cabezal, además de la posición actual del mismo (53). Si el brazo del disco se está moviendo hacia el cilindro 0, el cabezal dará servicio a las solicitudes correspondientes a los cilindros 37 y 14. Al llegar al cilindro 0, el brazo invertirá su movimiento y se desplazará hacia el otro extremo del disco, dando servicio a las solicitudes correspondientes a los cilindros 65, 67, 98, 122, 124 y 183 (Figura 12.6). Si llega una solicitud a la cola y esa solicitud corresponde a un cilindro situado justo delante del cabezal, dicha solicitud será servida casi inmediatamente; por el contrario, una solicitud relativa a un cilindro situado justo detrás del cabezal tendrá que esperar hasta que el brazo alcance el extremo del disco, invierta su dirección y vuelva atrás.

Suponiendo que existe una distribución uniforme de solicitudes de los cilindros, vamos a considerar la densidad de solicitudes cuando el cabezal alcanza uno de los extremos e invierte su dirección. En dicho punto, habrá un número relativamente bajo de solicitudes situadas inmediatamente delante del cabezal, ya que a esos cilindros se les ha prestado servicio recientemente. La mayor densidad de solicitudes corresponderá al otro extremo del disco. Dichas solicitudes también habrán estado esperando el tiempo más largo, por lo que ¿por qué no ir allí en primer lugar? Esa es la idea en que se basa el siguiente algoritmo.

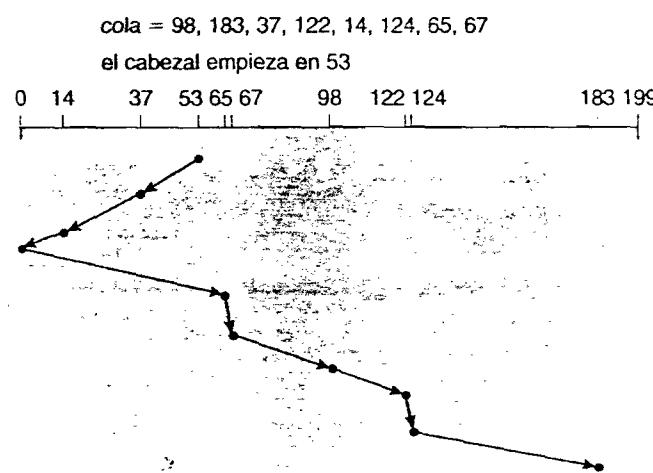


Figura 12.6 Planificación SCAN del disco.

12.4.4 Algoritmo C-SCAN

La **planificación SCAN circular** (C-SCAN) es una variante de SCAN diseñada para proporcionar un tiempo de espera más uniforme. Al igual que SCAN, C-SCAN mueve el cabezal de un extremo del disco al otro, prestando servicio a las solicitudes a lo largo de ese trayecto. Sin embargo, cuando el cabezal alcanza el otro extremo, vuelve inmediatamente al principio del disco, sin dar servicio a ninguna solicitud en el viaje de vuelta (Figura 12.7). El algoritmo de planificación C-SCAN trata esencialmente a los cilindros como si fueran una lista circular que se plegara sobre sí misma conectando el cilindro final con el primer cilindro.

12.4.5 Planificación LOOK

Tal como los hemos descrito, tanto el algoritmo SCAN como el C-SCAN mueven el brazo del disco a través de la anchura completa del disco; sin embargo, en la práctica, ninguno de los dos algoritmos se suele implementar de esta manera. Lo más común es que el brazo sólo vaya hasta el cilindro correspondiente a la solicitud final en cada dirección. Entonces, invierte su dirección inmediatamente, sin llegar hasta el extremo del disco. Las versiones de SCAN y de C-SCAN que siguen este patrón se denominan **planificación LOOK** y **C-LOOK**, ya que estos algoritmos *miran* (*look*) si una hay una solicitud antes de continuar moviéndose en una determinada dirección (Figura 12.8).

12.4.6 Selección de un algoritmo de planificación de disco

Dado que existen tantos algoritmos de planificación de disco distintos, ¿cómo podemos elegir el mejor de ellos? SSTF resulta bastante común y tiene un atractivo natural, porque mejora la velocidad que puede obtenerse con FCFS. SCAN y C-SCAN tienen un mejor comportamiento en aquellos sistemas donde el disco está sometido a una intensa carga, porque es menos probable que provoquen problemas de muerte por inanición. Para cualquier lista concreta de solicitudes, podemos definir un orden óptimo de extracción de los datos, pero los cálculos necesarios para encontrar una planificación óptima pueden no justificar el ahorro que se obtenga con respecto a SSTF o SCAN. De todos modos, con cualquier algoritmo de planificación, las prestaciones dependen en gran medida del número de solicitudes y del tipo de éstas. Por ejemplo, suponga que la cola suele tener tan solo una solicitud pendiente. Entonces, todos los algoritmos de planificación tendrán el mismo comportamiento porque sólo tienen una elección a la hora de decidir a dónde mover el cabezal del disco: todos se comportarán igual que el algoritmo FCFS.

Las solicitudes de servicio de disco pueden verse enormemente influenciadas por el método de asignación de archivos. Un programa que esté leyendo un archivo con asignación contigua generará varias solicitudes que están próximas entre sí en el disco, lo que da como resultado un movi-

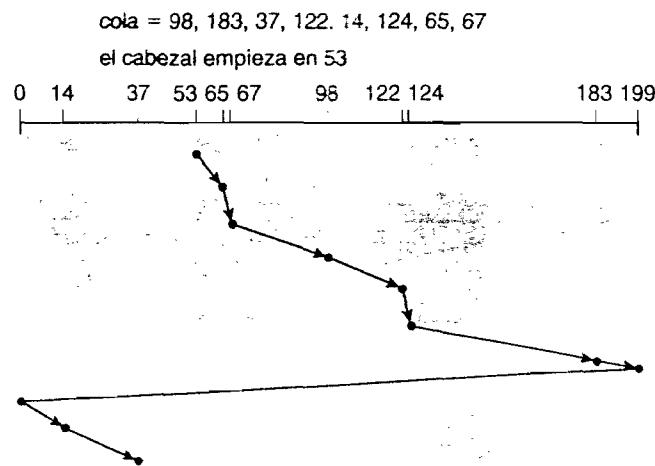


Figura 12.7 Planificación C-SCAN del disco.

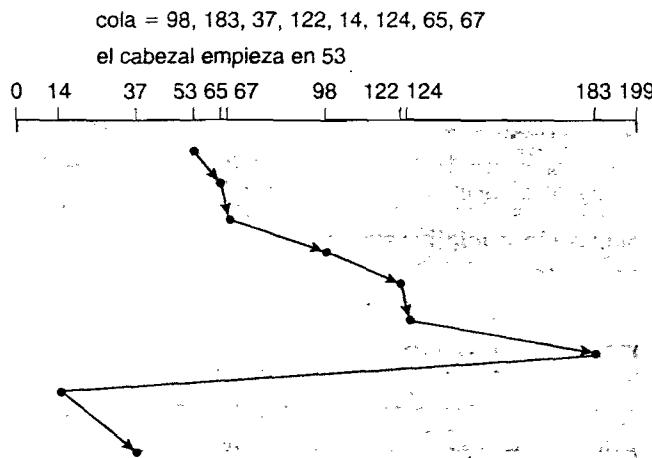


Figura 12.8 Planificación C-LOOK del disco.

miento de cabezales limitado. Por el contrario, un archivo enlazado o indexado puede incluir bloques que estén muy dispersos por el disco, lo que provoca un movimiento mucho mayor de los cabezales.

La ubicación de los directorios y de los bloques de índice también es importante. Puesto que todo archivo debe abrirse para ser utilizado y abrir un archivo requiere explorar la estructura de directorios, será necesario acceder a los directorios frecuentemente. Suponga que una entrada de directorios se encuentra en el primer cilindro y que los datos de un archivo se encuentran en el cilindro final. En este caso, el cabezal del disco tiene que atravesar toda la anchura del disco. Si la entrada de directorio estuviera en el cilindro intermedio, el cabezal tendría que moverse, como mucho, la mitad de la anchura. Si se almacenan en caché los directorios y los bloques de índice, dentro de la memoria principal, también puede reducirse el movimiento del brazo del disco, particularmente para las solicitudes de lectura.

Debido a estas complejidades, el algoritmo de planificación del disco debe escribirse como módulo independiente del sistema operativo, para poderlo sustituir por otro algoritmo distinto en caso necesario. Tanto SSTF como LOOK son una elección razonable como algoritmo predeterminado.

Los algoritmos de planificación descritos aquí sólo tienen en cuenta las instancias de búsqueda. Para los discos modernos, la latencia rotacional puede ser casi tan grande como el tiempo medio de búsqueda. Sin embargo, resulta difícil para el sistema operativo realizar una planificación para optimizar la latencia rotacional, porque los discos modernos no revelan la ubicación física de los bloques lógicos. Los fabricantes de disco tratan de aliviar este problema implementando algoritmos de planificación del disco en el hardware controlador integrado dentro de la unidad de disco. Si el sistema operativo envía un lote de solicitudes a la controladora, ésta las puede poner en cola y luego planificar su servicio para mejorar tanto el tiempo de búsqueda como la latencia rotacional.

Si la velocidad de E/S fuera la única consideración importante, el sistema operativo sólo tendría que ceder la responsabilidad de la planificación del disco al hardware del disco. En la práctica, sin embargo, el sistema operativo puede tener otras restricciones que afecten al orden de servicio de las solicitudes. Por ejemplo, la paginación bajo demanda puede tener prioridad sobre las operaciones de E/S de las aplicaciones, y las escrituras son más urgentes que las lecturas si la caché se está quedando sin páginas libres. Asimismo, puede ser deseable garantizar el orden de un conjunto de escrituras en disco para hacer que el sistema de archivos sea robusto en caso de fallos catastróficos del sistema. Considere lo que sucedería si el sistema operativo asignara una página de disco a un archivo y la aplicación escribiera datos en dicha página antes de que el sistema operativo tuviera la posibilidad de volcar el inodo modificado y la lista de espacio libre en el disco. Para tener en cuenta estos requisitos, el sistema operativo puede decidir realizar su

propia planificación de disco y entregar las solicitudes a la controladora de disco una a una, para que no se forme una cola, en algunos tipos de E/S.

12.5 Gestión del disco

El sistema operativo es responsable también de varios otros aspectos de la gestión del disco. Aquí vamos a hablar de la inicialización del disco, del arranque desde el disco y de la recuperación de bloques defectuosos.

12.5.1 Formateo del disco

Un disco magnético nuevo es como una pizarra en blanco: se trata simplemente de una placa de material magnético para grabación. Antes de poder almacenar datos en el disco, es necesario dividir éste en sectores que la controladora de disco pueda leer y escribir. Este proceso se denomina **formateo de bajo nivel** o **formateo físico**. El formateo de bajo nivel llena el disco con una estructura de datos especial para cada sector. La estructura de datos para un sector consta típicamente de una cabecera, un área de datos (que inicialmente tiene 512 bytes de tamaño) y una cola. La cabecera y la cola contienen información utilizada por la controladora de disco, como el número de sector y un **código de corrección de errores** (ECC, error-correcting code). Cuando la controladora escribe un sector de datos durante la E/S normal, el ECC se actualiza con un valor que se calcula a partir de todos los bytes contenidos en el área de datos. Cuando se lee el sector, se recalculará el ECC y se lo compara con el valor almacenado. Si los valores almacenado y calculado son diferentes, esa diferencia indica que el área de datos del sector se ha corrompido y que el sector de disco puede ser defectuoso (Sección 12.5.3). El ECC es un código de *corrección* de errores porque contiene suficiente información como para que, en caso de que sólo unos pocos bits de datos se hayan corrompido, la controladora pueda identificar qué bits han sido modificados y calcular cuáles deberían ser sus valores correctos. En este caso, la controladora informará de que se ha producido un **error blando** recuperable. La controladora realiza automáticamente el procesamiento ECC cada vez que se lee o escribe un sector.

La mayoría de los discos duros se formatean a bajo nivel en la fábrica como parte del proceso de fabricación. Este formateo permite al fabricante probar el disco e inicializar el mapeo entre números de bloque lógicos y sectores libres de defectos en el disco. En muchos discos duros, cuando se ordena a la controladora de disco realizar un formateo a bajo nivel del disco, también se la puede decir cuántos bytes de datos debe dejar entre la cabecera y la cola de cada sector. Normalmente, es posible elegir entre unos cuantos tamaños, como por ejemplo 256, 512 y 1024 bytes. Formatear un disco con un tamaño de sector más grande implica que en cada pista cabrán menos sectores, pero también que se escribirán menos cabeceras y colas en cada pista y que habrá más espacio disponible para los datos de usuario. De todos modos, algunos sistemas operativos sólo pueden admitir un tamaño de sector igual a 512 bytes.

Para utilizar un disco para almacenar archivos, el sistema operativo sigue necesitando poder grabar sus propias estructuras de datos en el disco y para ello sigue un proceso en dos pasos. El primer paso consiste en **particionar** el disco en uno o más grupos de cilindros. El sistema operativo puede tratar cada partición como si fuera un disco distinto. Por ejemplo, una partición puede albergar una copia del código ejecutable del sistema operativo, mientras que otra puede almacenar los archivos de usuario. Después del particionamiento, el segundo paso es el **formateo lógico** (o creación de un sistema de archivos). En este paso, el sistema operativo almacena las estructuras de datos iniciales del sistema de archivos en el disco. Estas estructuras de datos pueden incluir mapas de espacio libre y asignado (una tabla FAT o una serie de inodos) y un directorio inicial vacío.

Para aumentar la eficiencia, la mayoría de los sistemas de archivos agrupan los bloques en una serie de fragmentos de mayor tamaño, que frecuentemente se denominan *clusters*. La E/S de disco se realiza mediante los bloques, pero la E/S del sistema de archivos se realiza mediante *clusters*, garantizando así que la E/S tenga un acceso con características más secuenciales y menos aleatorias.

Algunos sistemas operativos proporcionan a algunos programas especiales la capacidad de utilizar una partición de disco como si fuera una gran matriz secuencial de bloques lógicos, sin ninguna estructura de datos correspondiente a ningún sistema de archivos. Esta matriz se denomina en ocasiones disco en bruto o disco sin formato y la E/S dirigida a esta matriz se denomina E/S sin formato. Por ejemplo, algunos sistemas de base de datos prefieren la E/S sin formato, porque les permite controlar la ubicación exacta en el disco donde se almacenará cada registro de la base de datos. La E/S sin formato puentea todos los servicios del sistema de archivos, como la caché de búfer, el bloqueo de archivos, la pre-extracción, la asignación de espacio, los nombres de archivo y los directorios. Podemos hacer que ciertas aplicaciones sean más eficientes permitiéndoles implementar sus propios servicios de almacenamiento de propósito especial sobre una partición sin formato, pero la mayoría de las aplicaciones tendrán unas mejores prestaciones cuando utilicen los servicios normales del sistema de archivos.

12.5.2 Bloque de arranque

Para que una computadora comience a operar (por ejemplo, cuando se la enciende o cuando se la reinicia) debe tener un programa inicial que ejecutar. Este programa inicial de *arranque* tiende a ser muy simple. Se encarga de inicializar todos los aspectos del sistema, desde los registros de la CPU hasta las controladoras de dispositivo y el contenido de la memoria principal, y luego arranca el sistema operativo. Para llevar a cabo su tarea, el programa de arranque localiza el *kernel* del sistema operativo en el disco, carga dicho *kernel* en memoria y salta hasta una dirección inicial con el fin de comenzar la ejecución del sistema operativo.

Para la mayoría de las computadoras, el programa de arranque está almacenado en **memoria de sólo lectura** (ROM, read-only memory). Esta ubicación resulta muy adecuada, porque la ROM no necesita ser inicializada y se encuentra en una dirección fija en la que el procesador puede comenzar la ejecución al encenderlo o reiniciarlo. Además, puesto que la ROM es de sólo lectura, no puede verse afectada por virus informáticos. El problema es que cambiar este código de arranque requiere cambiar los chips hardware de la ROM. Por esta razón, la mayoría de los sistemas almacenan un programa cargador de arranque de muy pequeño tamaño en la ROM de arranque, cuya única tarea consiste en cargar un programa de arranque completo desde el disco. El programa de arranque completo puede cambiarse fácilmente. Cada nueva versión puede simplemente escribirse en el disco. El programa cargador completo se almacena en los “bloques de arranque”, en una ubicación fija del disco. Un disco que tenga una partición de arranque se denomina **disco de arranque** o **disco del sistema**.

El código de la ROM de arranque ordena a la controladora de disco que lea los bloques de arranque en memoria (todavía no hay ningún controlador de dispositivo cargado en este punto) y luego comienza a ejecutar dicho código. El programa de arranque completo es más sofisticado que el cargador de arranque almacenado en la ROM; el programa de arranque completo es capaz de cargar todo el sistema operativo desde una ubicación no fija dentro del disco y luego iniciar la ejecución del sistema operativo. A pesar de ello, el código del programa de arranque completo puede ser pequeño.

Consideremos como ejemplo el proceso de arranque en Windows 2000. El sistema Windows 2000 coloca el código de arranque en el primer sector del disco duro (el cual denomina **registro de arranque maestro**, o MBR, master-boot record). Además, Windows 2000 permite dividir un disco duro en una o más particiones. Una de las particiones, identificada como la **partición de arranque**, contiene el sistema operativo y los controladores de dispositivo. El arranque comienza en un sistema Windows 2000 ejecutando un código residente en la memoria ROM del sistema. Este código instruye al sistema para que lea el código de arranque del MBR. Además de contener el código de arranque, el MBR contiene una tabla que numera las particiones del disco duro, junto con un indicador que marca desde cuál partición hay que arrancar el sistema. Esto se ilustra en la Figura 12.9. Una vez que el sistema identifica la partición de arranque, lee el primer sector de dicha partición (que se denomina **sector de arranque**) y continúa con el resto del proceso de arranque, que incluye cargar los diversos subsistemas y servicios del sistema.

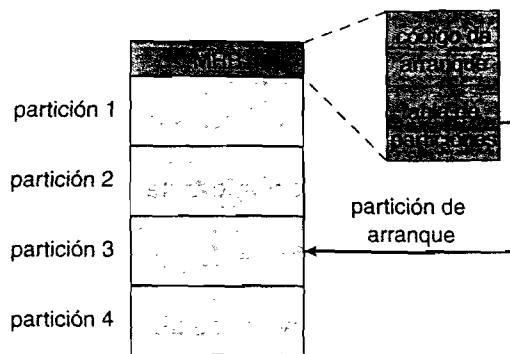


Figura 12.9 Arranque desde disco en Windows 2000.

12.5.3 Bloques defectuosos

Puesto que los discos tienen partes móviles y tolerancias muy pequeñas (recuerde que el cabezal del disco vuela justo por encima de la superficie del disco) son bastante propensos a los fallos. Algunas veces, el fallo que se produce es completo, en cuyo caso será necesario cambiar el disco y restaurar su contenido en el nuevo disco a partir de un soporte de copia de seguridad. Lo más frecuente es que uno o más sectores pasen a ser defectuosos. La mayoría de los discos vienen incluso de fábrica con una serie de **bloques defectuosos**. Dependiendo del disco y de la controladora que se utilice, estos bloques se gestionan de diversas formas.

En los discos más simples, como por ejemplo algunos discos con controladoras IDE, los bloques defectuosos se gestionan de forma manual. Por ejemplo, el comando `format` de MS-DOS realiza el formateo lógico y, como parte de dicho proceso, analiza el disco para localizar los bloques defectuosos. Si `format` encuentra un bloque defectuoso, escribe un valor especial en la entrada correspondiente de la FAT para informar a las rutinas de asignación de que no deben utilizar dicho bloque. Si los bloques pasan a ser defectuosos durante la operación normal, es necesario ejecutar manualmente un programa especial (como por ejemplo `chkdsk`) para buscar los bloques defectuosos y prohibir su utilización, de la misma forma que antes. Los datos que estuvieran almacenados en los bloques defectuosos usualmente se pierden.

Los discos más sofisticados, como los discos SCSI utilizados en las computadoras PC de gama alta y en la mayoría de las estaciones de trabajo y servidores, tienen un mecanismo más inteligente de recuperación de bloques defectuosos. La controladora mantiene una lista de bloques defectuosos en el disco. La lista se inicializa durante el formateo de bajo nivel en la fábrica y se actualiza a todo lo largo de la vida del disco. El formateo a bajo nivel también reserva una serie de sectores adicionales que no son visibles para el sistema operativo. Se puede ordenar a la controladora que sustituya lógicamente cada vector defectuoso por uno de los sectores adicionales reservados. Este esquema se denomina **reserva de sectores** o **sustitución de sectores**.

Una transacción típica relativa a un sector defectuoso sería la siguiente:

- El sistema operativo trata de leer el bloque lógico 87.
- La controladora calcula el ECC y comprueba que el sector es defectuoso, por lo que informa de este hecho al sistema operativo.
- La siguiente vez que se reinicie el sistema, se ejecuta un comando especial para informar a la controladora SCSI de que debe sustituir el sector defectuoso por uno de los sectores reservados.
- A partir de ahí, siempre que el sistema solicite el bloque lógico 87, la solicitud será traducida por la controladora, que sustituirá dicha dirección por la del sector sustituto.

Este tipo de redirección efectuado por la controladora podría invalidar las optimizaciones realizadas por el algoritmo de planificación del disco del sistema operativo. Por esta razón, la mayo-

ría de los discos se formatean de modo que se proporcionen unos cuantos sectores reservados en cada cilindro y también un cilindro reservado. Cuando se remapea un bloque defectuoso, la controladora utiliza un sector reservado del mismo cilindro, siempre que sea posible.

Como alternativa al mecanismo de reserva de sectores, algunas controladoras ofrecen la posibilidad de sustituir un bloque defectuoso por el procedimiento de **deslizamiento de sectores**. He aquí un ejemplo: suponga que el bloque lógico 17 pasara a estar defectuoso y que el primer sector reservado disponible estuviera situado después del sector 202. Entonces, el procedimiento de deslizamiento de sectores remapea todos los sectores comprendidos entre el 17 y el 202, moviéndolos una posición. En otras palabras, el sector 202 se copia en el sector reservado, luego el sector 201 se copia en el sector 202, el sector 200, en el 201, etc., hasta que el sector 18 se copia en el sector 19. Deslizar los sectores de esta forma libera el espacio del sector 18, por lo que se puede mapear el sector 17 sobre el espacio que aquel ocupaba.

La sustitución de un bloque defectuoso no es, generalmente, del todo automática, porque los datos del bloque defectuoso suelen perderse. Diversos errores blandos podrían hacer que se desencadenara un proceso en el que se realizará una copia de los datos del bloque y dicho bloque se marcará como no válido, empleándose a continuación los procedimientos de sustitución o deslizamiento de bloques. Sin embargo, los **errores duros** no recuperables provocan la pérdida de datos. En ese caso, será necesario reparar el archivo que estuviera utilizando dicho bloque (por ejemplo, realizando una restauración a partir de una cinta de seguridad) y ese proceso requiere una intervención manual.

12.6 Gestión del espacio de intercambio

El procedimiento de intercambio ya ha sido presentado en la Sección 8.2, donde hemos hablado de los procedimientos para mover procesos completos entre el disco y la memoria principal. El intercambio, en dicho tipo de entorno, tiene lugar cuando la cantidad de memoria física se reduce hasta un punto críticamente bajo y los procesos (que se seleccionan usualmente porque son los menos activos) se mueven desde la memoria al espacio de intercambio con el fin de aumentar la memoria libre disponible. En la práctica, muy pocos sistemas operativos modernos implementan los procedimientos de intercambio de esta manera. En lugar de ello, los sistemas combinan ahora los procedimientos de intercambio con las técnicas de memoria virtual (Capítulo 9) e intercambian páginas, y no necesariamente procesos completos. De hecho, algunos sistemas utilizan ahora los términos *intercambio* y *paginación* como sinónimos, reflejando la convergencia de estos dos conceptos.

La **gestión del espacio de intercambio** es otra tarea de bajo nivel del sistema operativo. La memoria virtual utiliza el espacio de disco como una extensión de la memoria principal. Puesto que el acceso a disco es mucho más lento que el acceso a memoria, la utilización del espacio de intercambio reduce significativamente las prestaciones del sistema. El objetivo principal del diseño y la implementación del espacio de intercambio es proporcionar la mejor tasa de transferencia posible para el sistema de memoria virtual. En esta sección, veremos cómo se utiliza el espacio de intercambio, dónde se lo sitúa dentro del disco y cómo se gestiona ese espacio de intercambio.

12.6.1 Utilización del espacio de intercambio

El espacio de intercambio se utiliza de diversas formas en los distintos sistemas operativos, dependiendo de los algoritmos de gestión de memoria que se empleen. Por ejemplo, los sistemas que implementan el mecanismo de intercambio pueden utilizar el espacio de intercambio para guardar la imagen completa de un proceso, incluyendo el código y los segmentos de datos. Los sistemas de paginación pueden, simplemente, almacenar las páginas que hayan sido expulsadas de la memoria principal. La cantidad de espacio de intercambio necesaria en un sistema puede, por tanto, variar dependiendo de la cantidad de memoria física, de la cantidad de memoria virtual a la que ese espacio de intercambio esté prestando respaldo y de la forma en que se utilice la memoria virtual. El tamaño del espacio de intercambio puede ir desde unos pocos megabytes de memoria de disco hasta varios gigabytes.

Observe que puede ser resultar más seguro sobreestimar la cantidad de espacio de intercambio requerido, en lugar de subestimarla, porque si un sistema se queda sin espacio de intercambio puede verse forzado a abortar procesos o puede incluso sufrir un fallo catastrófico. La sobreestimación hace que se desperdicie un espacio de disco que podría, de otro modo, utilizarse para los archivos, pero no provoca ningún otro daño. Algunos sistemas recomiendan cuál es la cantidad de espacio que hay que reservar para el espacio de intercambio. Solaris, por ejemplo, sugiere que se reserve un espacio de intercambio igual a la cantidad en la que la memoria virtual exceda a la memoria física paginable. Históricamente, Linux sugiere que se reservara un espacio de intercambio igual al doble de la memoria física disponible, aunque la mayoría de los sistemas Linux utilizan ahora un espacio de intercambio considerablemente inferior. De hecho, existe un gran debate actualmente en la comunidad Linux sobre si debe o no reservarse ningún espacio de intercambio.

Algunos sistemas operativos, incluido Linux, permiten utilizar múltiples espacios de intercambio. Estos espacios de intercambio se colocan usualmente en discos separados, para que la carga impuesta al sistema de E/S por los mecanismos de paginación e intercambio pueda distribuirse entre los distintos dispositivos de E/S del sistema.

12.6.2 Ubicación del espacio de intercambio

El espacio de intercambio puede residir en uno de dos lugares: puede construirse a partir del sistema de archivos normal o puede residir en una partición de disco separada. Si el espacio de intercambio es simplemente un archivo de gran tamaño dentro del sistema de archivos, pueden usarse las rutinas normales del sistema de archivos para crearlo, nombrarlo y asignarle el espacio. Esta técnica, aunque resulta fácil de implementar, también es poco eficiente. La navegación por la estructura del directorio y por las estructuras de datos de asignación de espacio en el disco consume mucho tiempo y requiere, potencialmente, accesos adicionales al disco. El fenómeno de la fragmentación externa puede incrementar enormemente el tiempo requerido para el intercambio, forzando a realizar múltiples búsquedas durante la lectura o escritura de la imagen de un proceso. Podemos mejorar el rendimiento almacenando en caché la información de ubicación de los bloques dentro de la memoria física y utilizando herramientas especiales para asignar bloques físicamente contiguos para el archivo de intercambio, pero nos seguirá quedando el coste asociado a recorrer las estructuras de datos del sistema de archivos.

Alternativamente, puede crearse un espacio de intercambio en una partición sin formato separada, ya que en este espacio no se almacena ningún sistema de archivos ni estructura de directorio. En lugar de ello, se utiliza un gestor de almacenamiento independiente para el espacio de intercambio, con el fin de asignar y desasignar los bloques de la partición sin formato. Este gestor utiliza algoritmos optimizados para obtener la mayor velocidad (en lugar de la mayor eficiencia de almacenamiento) porque al espacio de intercambio se accede de forma mucho más frecuente que a los sistemas de archivos (cuando se utiliza ese espacio de intercambio). La fragmentación interna puede incrementarse, pero este compromiso resulta aceptable porque la vida de los datos dentro del espacio de intercambio es, generalmente, mucho más corta que la de los archivos residentes en el sistema de archivos. El espacio de intercambio se reinicializa en el momento del arranque de la máquina, por lo que cualquier fragmentación que exista es de corta duración. Esta técnica crea una cantidad fija de espacio de intercambio durante el particionamiento del disco. Para añadir más espacio de intercambio, es necesario volver a particionar el disco (lo que implica desplazar las otras particiones de sistemas de archivo o destruirlas y restaurarlas a partir de una copia de seguridad) o añadir otro espacio de intercambio en algún otro lugar.

Algunos sistemas operativos son flexibles y pueden realizar el intercambio tanto en particiones sin formato como en el espacio del sistema de archivos. Un ejemplo de este tipo de sistema sería Linux: la política y la implementación están separadas, lo que permite al administrador de la máquina decidir qué tipo de intercambio utilizar. El compromiso que hay que adoptar es entre la comodidad de asignación y de gestión dentro del sistema de archivos y las mayores prestaciones que pueden obtenerse al realizar el intercambio en particiones sin formato.

12.6.3 Gestión del espacio de intercambio: un ejemplo

Podemos ilustrar cómo se utiliza el espacio de intercambio siguiendo la evolución de los mecanismos de intercambio y de paginación en diversos sistemas UNIX. El *kernel* UNIX tradicional comenzó como una implementación del intercambio que copiaba procesos completos entre regiones contiguas del disco y la memoria. UNIX evolucionó posteriormente para adoptar una combinación de intercambio y paginación, a medida que se fue incorporando hardware de paginación en las máquinas.

En Solaris 1 (SunOS), los diseñadores cambiaron los métodos estándar de UNIX para mejorar la eficiencia y reflejar los cambios sufridos por la tecnología. Cuando se ejecuta un proceso, las páginas de segmentos de texto que contienen código se cargan desde el sistema de archivos, se accede a ellas en la memoria principal y se las elimina cuando el mecanismo de paginación las selecciona para descarga. Resulta más eficiente volver a leer una página del sistema de archivos que escribirla en el espacio de intercambio y luego volver a leerla de allí. El espacio de intercambio sólo se utiliza como almacenamiento de respaldo para las páginas de memoria anónima, que incluyen la memoria asignada a la pila, al círculo y a los datos no inicializados de un proceso.

En las versiones posteriores de Solaris se hicieron más cambios. El cambio principal es que Solaris ahora asigna el espacio de intercambio únicamente cuando se descarga una página de la memoria física, en lugar de en el momento en que se crea por primera vez la página de memoria virtual. Este esquema proporciona unas mayores prestaciones en las computadoras modernas, que tienen más memoria física que los sistemas antiguos y tienden a utilizar en menor medida los mecanismos de paginación.

Linux es similar a Solaris, en el sentido de que el espacio de intercambio sólo se utiliza para la memoria anónima o para las regiones de memoria compartidas por varios procesos. Linux permite establecer una o más áreas de intercambio. Un área de intercambio puede estar en un archivo de intercambio dentro de un sistema de archivos normal o en una partición de intercambio sin formato. Cada área de intercambio está compuesta por una serie de ranuras de página de 4 KB, que se utilizan para albergar las páginas descargadas por el mecanismo de intercambio. Asociado con cada área de intercambio hay un mapa de intercambio, que es una matriz de contadores enteros, cada uno de los cuales corresponde a una ranura de página dentro del área de intercambio. Si el valor de un contador es 0, la correspondiente ranura de página está disponible. Los valores superiores a 0 indican que la ranura de página está ocupada por una página intercambiada. El valor del contador indica el número de mapeos que existen sobre la página intercambiada; por ejemplo, un valor de 3 indica que la página intercambiada está mapeada sobre tres procesos diferentes (lo que puede suceder si la página intercambiada almacena una región de memoria compartida por tres procesos). En la Figura 12.10 se ilustra las estructuras de datos del intercambio en los sistemas Linux.

12.7 Estructuras RAID

Las unidades de disco han ido evolucionando para hacerse más pequeñas y más baratas, por lo que ahora es económicamente factible conectar muchos discos a un mismo sistema informático.

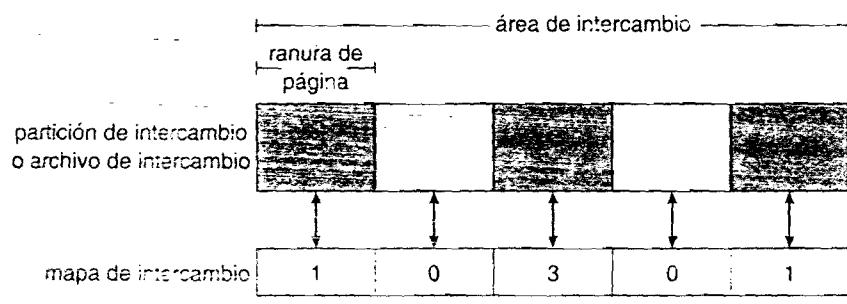


Figura 12.10 Estructuras de datos para los mecanismos de intercambio en los sistemas Linux.

Disponer de un gran número de discos en un sistema permite mejorar la velocidad con la que los datos pueden leerse o escribirse, siempre y cuando los discos se operen en paralelo. Además, esta configuración también puede permitir mejorar la fiabilidad del almacenamiento de los datos, ya que puede almacenarse información redundante en ese conjunto de múltiples discos. De este modo, el fallo de uno de los discos no conduce a la pérdida de datos. Existen diversas técnicas de organización de discos, colectivamente denominadas **matrices redundantes de discos de bajo coste** (RAID, redundant arrays of inexpensive disks), que se utilizan comúnmente para resolver estas cuestiones de velocidad y de fiabilidad.

En el pasado, las matrices RAID compuestas de discos pequeños y baratos se contemplaban como una alternativa económica a los discos de gran tamaño, que eran bastante caros; hoy en día las matrices RAID se utilizan debido a su mayor fiabilidad y a su mayor velocidad de transferencia de datos, más que por razones económicas. Por esta razón, la I de RAID ahora quiere decir "independientes" en lugar de "inexpensive" (de bajo coste).

12.7.1 Mejora de la fiabilidad a través de la redundancia

Consideremos primero la fiabilidad de las matrices RAID. La probabilidad de que falle un disco dentro de un conjunto de N discos es mucho mayor que la probabilidad de que falle un disco específico. Suponga que el **tiempo medio entre fallos** de un único disco es de 100.000 horas. Entonces, el tiempo medio entre fallos de un disco cualquiera dentro de una matriz de 100 discos será $100.000/100 = 1000$ horas, es decir 41,66 días, que hay que reconocer que no es un tiempo muy largo. Si sólo almacenamos una copia de los datos, cada fallo de un disco provocará la pérdida de una cantidad significativa de datos, y dicha alta tasa de pérdida de datos es completamente inaceptable.

La solución al problema de la fiabilidad consiste en introducir **redundancia**. Con este sistema, almacenamos información adicional que no es normalmente necesaria pero que puede utilizarse en caso de que falle el disco, para reconstruir la información perdida. Así, incluso si un disco falla, los datos no se pierden.

La técnica más simple, pero la más cara, para introducir redundancia consiste en duplicar cada uno de los discos. Esta técnica se denomina **duplicación en espejo** (mirroring). Así, cada disco lógico estará compuesto por dos discos físicos y toda escritura se llevará a cabo en ambos discos a la vez. Si uno de los discos falla, pueden leerse los datos del otro disco. Los datos sólo se perderán si el segundo disco fallara antes de que el primer disco estropeado se reemplace.

El tiempo medio entre fallos (donde un *fallo* es la pérdida de datos) de un volumen duplicado en espejo (compuesto de dos discos duplicados) depende de dos factores. Uno es el tiempo medio entre fallos de los discos individuales y el otro es el **tiempo medio de reparación**, que es el tiempo que se tarda (como promedio) en sustituir un disco fallido y restaurar en él los datos. Suponga

12.7.2 INTRODUCCIÓN DE MATRICES RAID

El almacenamiento RAID puede organizarse de varias formas. Por ejemplo, un sistema que instala los discos conectados directamente a sus buses. En este caso, el sistema opera sin el software del sistema que implementa la funcionalidad RAID. Alternativamente, una centralizada, en la que tienen que estar todos los discos conectados a un controlador RAID que administra los discos. Por hardware, usualmente también puede organizarse mediante una memoria interna o matriz RAID. Una matriz RAID es una memoria interna que tiene su propia configuración, su propia memoria cache (usualmente) y sus discos están conectados a través de una o más controladoras estándar ATA SCSI o FC. Esta configuración es independiente de los sistemas con protección RAID a cualquier sistema operativo y a cualquier software que no disponga de funcionalidad RAID. Incluso se la usa en sistemas que si que disponen de capas de software RAID debido a su simplicidad y a su flexibilidad.

que los fallos de los dos discos son **independientes**, es decir, que el fallo de uno de los discos no está conectado con el fallo del otro. Entonces, si el tiempo medio entre fallos de un único disco es de 100.000 horas y el tiempo medio de reparación es de 10 horas, el **tiempo medio de pérdida de datos** de un sistema con duplicación de discos en espejo será $100.000^2 / (2 * 10) = 500 * 10^6$ horas, o 57.000 años.

Hay que ser conscientes de que la suposición de la independencia entre los fallos de disco no es válida. Los fallos de alimentación y los desastres naturales, como los terremotos, incendios e inundaciones, pueden provocar que resulten dañados ambos discos al mismo tiempo. Asimismo, los defectos de fabricación en un lote de discos pueden provocar fallos correlacionados. A medida que los discos envejecen, la probabilidad de fallo crece, incrementando la posibilidad de que falle un segundo disco mientras que el primero se está reparando. Pero, a pesar de todas estas consideraciones, los sistemas de discos con duplicación en espejo ofrecen una fiabilidad mucho más alta que los sistemas de un único disco.

Los fallos de alimentación son uno de los principales motivos de preocupación, ya que suceden con una frecuencia mucho mayor que los desastres naturales. Incluso con la duplicación en espejo de los discos, si se están produciendo sendas escrituras en el mismo bloque en ambos discos y la alimentación falla antes de que ambos bloques estén completamente escritos, los dos bloques pueden quedar en un estado incoherente. Una solución a este problema, consiste en escribir primero una copia y luego la siguiente, de modo que una de las dos copias sea siempre coherente. Otra solución consiste en añadir una **RAM no volátil** (NVRAM) a la matriz RAID como memoria caché. Esta caché de escritura diferida está protegida frente a la pérdida de datos durante los fallos de alimentación, por lo que la escritura puede considerarse completa en dicho punto, siempre cuando la NVRAM tenga algún tipo de mecanismo de protección y corrección de errores, como por ejemplo FC o la duplicación en espejo.

12.7.2 Mejoras en las prestaciones a través del paralelismo

Ahora, consideraremos el modo en que el acceso paralelo a múltiples discos mejora las prestaciones. Con la multiplicación en espejo de los discos, la velocidad a la que pueden gestionarse las solicitudes de lectura se dobla, ya que esas solicitudes pueden enviarse a cualquiera de los dos discos (siempre y cuando los dos discos de una pareja estén funcionando correctamente, como casi siempre sucede). La tasa de transferencia de cada lectura es igual que en un sistema de un único disco, pero el número de lecturas por unidad de tiempo se duplica.

Con múltiples discos, también (o en lugar de ello) podemos mejorar la velocidad de transferencia dividiendo los datos entre los distintos discos. En su forma más simple, el mecanismo de **distribución en bandas de los datos** (data striping) consiste en dividir los bits de cada byte entre múltiples discos; dicha distribución se denomina **distribución en bandas de nivel bit**. Por ejemplo, si tenemos una matriz de ocho discos, podemos escribir el bit i de cada byte en el disco i . La matriz de ocho discos puede tratarse como un único disco con sectores que son ocho veces más grandes de lo normal y que, todavía más importante, tienen una velocidad de acceso ocho veces mayor. Con este tipo de organización, todos los discos participan en cada acceso (de lectura o escritura) porque el número de accesos que pueden procesarse por segundo es más o menos el mismo que con un único disco, pero cada acceso puede leer ocho veces más datos en el mismo tiempo que si se utiliza un único disco.

La distribución en banda del nivel de bit puede generalizarse para incluir un número de discos que sea un múltiplo o un divisor de ocho. Por ejemplo, si utilizamos una matriz de cuatro discos, los bits i y $4 + i$ de cada byte van al disco i . Además, la distribución en bandas no tiene por qué realizarse en el nivel de bit. Por ejemplo, con el mecanismo de **distribución en bandas de nivel de bloque**, los bloques de cada archivo se dividen entre múltiples discos; con n discos, el bloque i de un archivo va al disco $(i \bmod n) + 1$. También son posibles otros niveles de distribución en bandas, distribuyendo por ejemplo los bytes de un sector en los sectores de un bloque. El sistema más común es la distribución en banda de nivel de bloque.

El paralelismo en un sistema de discos, tal como puede conseguirse mediante la distribución en bandas, tiene dos objetivos principales:

1. Incrementar la tasa de transferencia de múltiples accesos de pequeña envergadura (es decir, accesos de página), distribuyendo la carga entre los distintos discos.
2. Reducir el tiempo de respuesta de los accesos de gran volumen.

12.7.3 Niveles RAID

La duplicación en espejo proporciona una alta fiabilidad, pero resulta muy cara. La distribución en bandas proporciona una alta velocidad de transferencia de datos, pero no mejora la fiabilidad. Se han propuesto numerosos esquemas para proporcionar redundancia a un menor coste utilizando la idea de distribución en bandas combinada con bits de "paridad" (de los cuales hablaremos a continuación). Estos esquemas tienen diferentes tipos de compromisos entre el coste y las prestaciones y se clasifican de acuerdo con una serie de niveles denominados **niveles RAID**. Vamos a describir aquí los diversos niveles existentes; la Figura 12.11 muestra esos niveles en forma de ilustración (en la figura, *P* indica bits de corrección de errores y *C* indica una segunda copia de los datos). En todos los casos mostrados en la figura, estamos almacenando una cantidad de datos equivalente a la capacidad total de cuatro discos y los discos adicionales se utilizan para almacenar información redundante con vistas a la recuperación de fallos.

- **RAID nivel 0.** El mecanismo RAID nivel 0 hace referencia a matrices de discos con distribución de bandas en el nivel de bloques, pero sin ninguna redundancia (no tiene ni mecanismos de duplicación en espejo ni bits de paridad), como se muestra en la Figura 12.11(a).
- **RAID nivel 1.** RAID nivel 1 hace referencia a la duplicación en espejo de los discos. La Figura 12.11(b) muestra una organización con duplicación en espejo.
- **RAID nivel 2.** RAID nivel 2 también se conoce como **organización con códigos de corrección de errores (ECC) de tipo memoria**. En los sistemas de memoria hace ya tiempo que se detectan ciertos errores utilizando bits de paridad. Cada byte de un sistema de memoria puede tener un bit de paridad asociado que registra si el número de bits del byte con valor igual a 1 es par (paridad = 0) o impar (paridad = 1). Si uno de los bits del byte resulta dañado (porque un 1 se transforma en un 0 o un 0 se transforma en un 1), la paridad del byte cambia y, por tanto, no se corresponderá con el valor de paridad almacenado. De forma similar, si resulta dañado el bit de paridad almacenado, no se corresponderá con el valor de paridad calculado. De este modo, el sistema de memoria puede detectar todos los errores de un único bit. Los esquemas de corrección de errores almacenan dos o más bits adicionales y pueden reconstruir los datos si resulta dañado un único bit. La idea de los códigos ECC puede usarse directamente en las matrices de discos distribuyendo los bytes entre los distintos discos. Por ejemplo, el primer bit de cada byte puede almacenarse en el disco 1, el segundo bit en el disco 2, y así sucesivamente hasta el octavo bit, que se almacenará en el disco 8; los bits de corrección de errores se almacenan en discos adicionales. Este esquema se ilustra en la Figura 12.11(c), en donde los discos etiquetados con la letra *P* almacenan los bits de corrección de errores. Si uno de los discos falla, los restantes bits del byte y los bits de corrección de errores asociados pueden leerse en otros discos y utilizarse para reconstruir los datos dañados. Observe que el mecanismo RAID nivel 2 sólo requiere tres discos adicionales para cuatro discos de datos, a diferencia de RAID nivel 1, que requiere cuatro discos adicionales.
- **RAID nivel 3.** El esquema de organización RAID nivel 3, u **organización de paridad con entrelazado de bits**, representa una mejora con respecto al nivel 2, porque tiene en cuenta el hecho de que, a diferencia de los sistemas de memoria, las controladoras de disco pueden detectar si un sector ha sido leído correctamente, por lo que puede utilizarse un único bit de paridad tanto para detección como para corrección de errores. La idea es la siguiente: si uno de los sectores está dañado, sabemos exactamente qué sector es, y podemos determinar si algún bit del sector es un 1 o un 0 calculando la paridad de los bits correspondientes de los sectores almacenados en los otros discos. Si la paridad de los bits restantes es igual a la paridad almacenada, el bit que falta será un 0; en caso contrario, será un 1. RAID nivel 3 ofrece



Figura 12.11 Niveles RAID.

el mismo nivel de protección que el nivel 2, pero resulta más barato en cuanto al número de discos adicionales requeridos (sólo requiere un disco adicional) por lo que el nivel 2 nunca se utiliza en la práctica. Este esquema se ilustra en la Figura 12.11(d).

RAID nivel 3 tiene dos ventajas sobre el nivel 1. En primer lugar, se reduce la cantidad de almacenamiento necesario, ya que sólo hace falta un único disco de paridad para varios discos normales, mientras que en el nivel 1 se necesita un disco espejo por cada disco de datos. En segundo lugar, puesto que las lecturas y escrituras de un byte se distribuyen entre múltiples discos mediante un mecanismo de distribución de los datos con N vías, la tasa de transferencia para leer o escribir un único bloque es N veces más rápida que con RAID nivel 1. En el lado negativo, RAID nivel 3 soporta menos operaciones de E/S por segundo, ya que todos los discos tienen que participar en todas las solicitudes de E/S.

Un problema adicional de prestaciones con RAID 3 (al igual que con todos los niveles RAID que utilizan mecanismos de paridad) es el gasto de procesamiento adicional necesario para calcular y escribir los valores de paridad. Este procesamiento adicional hace que las escrituras sean significativamente más lentas que con las matrices RAID que no utilizan bits de paridad. Para compensar esta disminución del rendimiento, muchas matrices de almacenamiento RAID incluyen una controladora hardware con circuitería dedicada de cálculo paridad. Dicha controladora asume los cálculos de paridad, liberando a la CPU de dicha tarea. La matriz tiene también una caché NVRAM para almacenar los bloques mientras se

realizan las operaciones de escritura y para actuar como búfer de escritura para la transferencia de

datos entre la controladora y los discos físicos. Esta combinación puede hacer que las soluciones RAID con paridad sean casi tan rápidas como las que no utilizan paridad. De hecho, una matriz RAID con paridad que incluya una memoria caché puede ser más rápida que una matriz RAID sin paridad que no incluya esa memoria caché.

- **RAID nivel 4.** El esquema de organización RAID nivel 4, u **organización de paridad con entrelazado de bloques**, utiliza una distribución en bandas de nivel de bloque, como en RAID 0, y además mantiene un bloque de paridad en un disco separado, con información de paridad para los bloques correspondientes de los otros N discos. Este esquema se ilustra en la Figura 12.11(e). Si uno de los discos falla, puede usarse el bloque de paridad con los bloques correspondientes de los otros discos, para restaurar los bloques del disco fallido.

Una lectura de un bloque sólo accede a un disco, permitiendo que los otros discos procesen otras solicitudes. Así, la tasa de transferencia de datos para cada acceso es más lenta, pero pueden realizarse múltiples accesos de lectura en paralelo, lo que proporciona una mayor tasa de E/S global. Las tasas de transferencia para las lecturas de grandes volúmenes de datos son altas, ya que pueden leerse todos los discos en paralelo; las escrituras de grandes volúmenes de información también tienen altas tasas de transferencia, ya que los datos y la paridad pueden escribirse en paralelo.

Las escrituras independientes de pequeños volúmenes de datos no pueden realizarse en paralelo. Una escritura por parte del sistema operativo de un conjunto de datos cuyo tamaño sea inferior a un bloque requerirá leer el bloque, modificarlo con los nuevos datos y volverlo a escribir. También será necesario actualizar el bloque de paridad. Esto se conoce con el nombre de **ciclo de lectura-modificación-escritura**. Así, una única escritura requiere cuatro accesos de disco. Dos para leer los dos bloques antiguos y dos para escribir los dos nuevos bloques.

WAFL (Capítulo 11) utiliza RAID nivel 4 porque este nivel RAID permite añadir discos a una matriz RAID de forma completamente transparente. Si los discos añadidos se inicializan con bloques que contengan valores todos iguales a cero, entonces el valor de paridad no se modifica y el conjunto RAID seguirá siendo correcto.

- **RAID nivel 5.** El esquema de organización RAID nivel 5, o **paridad distribuida con entrelazado de bloques**, difiere del nivel 4 en que se distribuyen los datos y la información de paridad entre los $N + 1$ discos, en lugar de almacenar los datos en N discos y la información de paridad en un disco. Para cada bloque, uno de los discos almacena la información de paridad y los otros almacenan los datos. Por ejemplo, con una matriz de cinco discos, la paridad del n -ésimo bloque se almacena en el disco $(n \bmod 5) + 1$; los bloques n -ésimos de los otros cuatro discos almacenan los datos reales correspondientes a dicho bloque. Esta configuración se ilustra en la Figura 12.11(f), donde los valores de paridad P están distribuidos entre todos los discos. Un bloque de paridad no puede almacenar información de paridad para bloques situados en el mismo disco, porque un fallo del disco provocaría una pérdida de los datos, además de perderse la información de paridad, por lo que dicha pérdida no sería recuperable. Distribuyendo la información de paridad entre todos los discos del conjunto, RAID 5 evita el uso potencial excesivo de un único disco de paridad que puede tener lugar en RAID 4. RAID 5 es el sistema RAID con paridad que más comúnmente se usa.
- **RAID nivel 6.** El esquema de organización RAID nivel 6, también denominado **esquema de redundancia P + Q**, es bastante similar a RAID nivel 5, pero almacena información redundante adicional para proteger los datos frente a múltiples fallos de disco. En lugar de emplearse información de paridad, se utilizan códigos de corrección de errores, tales como los **códigos Reed-Solomon**. En el esquema mostrado en la Figura 12.11(g), se almacenan 2 bits de datos redundantes por cada 4 bits de datos (comparado con un bit de paridad en el nivel 5) y el sistema puede tolerar que fallen dos de los discos.
- **RAID nivel 0 + 1.** El esquema de organización RAID nivel 0 + 1 hace referencia a una combinación de los niveles RAID 0 y 1. RAID 0 proporciona la mejora en las prestaciones, mientras que RAID 1 proporciona la fiabilidad. Generalmente, este nivel proporciona un mejor

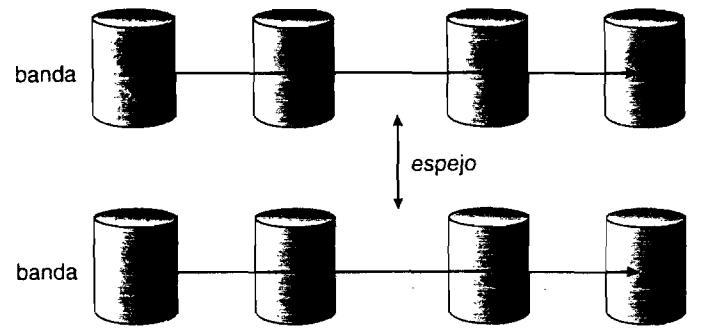
rendimiento que RAID 5. Resulta común en aquellos entornos en los que son importantes tanto las prestaciones como la fiabilidad. Desafortunadamente, este esquema dobla al número de discos necesarios para el almacenamiento, al igual que RAID 1, por lo que también es más caro. En RAID 0 + 1, se distribuye en bandas un conjunto de discos y luego ese conjunto se duplica en espejo en otro conjunto equivalente.

Otra opción RAID que también está empezando a estar disponible comercialmente es RAID nivel 1 + 0, en el que los discos se duplican en parejas y luego se distribuyen en bandas los datos entre las parejas de discos resultantes. Esta organización RAID tiene algunas ventajas teóricas sobre RAID 0 + 1. Por ejemplo, si falla un único disco en RAID 0 + 1, la banda completa deja de estar accesible, siendo accesible únicamente la otra banda. Con un fallo en RAID 1 + 0, ese único disco dejará de estar disponible, pero su pareja duplicada en espejo seguirá estando operativa, al igual que el resto de los discos (Figura 12.12).

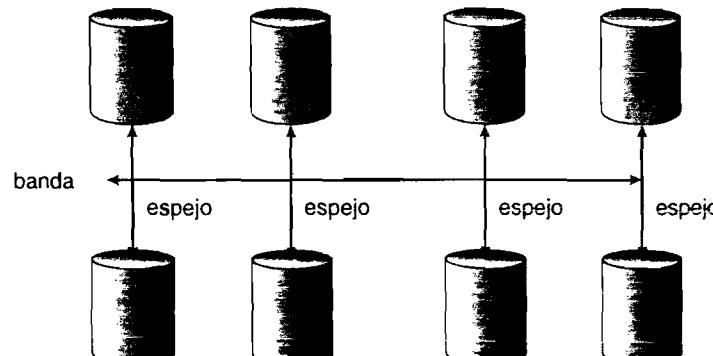
Se han propuesto numerosas variantes a los esquemas RAID básicos que hemos descrito. Como resultado, puede existir cierta confusión acerca de las definiciones exactas de los diferentes niveles RAID.

La implementación de RAID es otra de las áreas donde existen variantes. Considere los siguientes niveles en los que pueden implementarse los mecanismos RAID:

- El software de gestión de volúmenes puede implementar RAID dentro del *kernel* o en el nivel del software del sistema. En este caso, el hardware de almacenamiento puede proporcionar una serie de características mínimas y continuar formando parte de una solución RAID completa. Las soluciones RAID con paridad son bastante lentas cuando se las implementa por software, por lo que normalmente se utiliza RAID 0, 1 o 0 + 1.
- Puede implementarse el mecanismo RAID en el hardware adaptador de bus del *host* (HBA, *host bus-adapter*). Sólo los discos que estén directamente conectados al HBA podrán formar parte de un conjunto RAID determinado. Esta solución tiene un coste bajo, pero no resulta muy flexible.



a) RAID 0 + 1 con un único fallo de disco.



b) RAID 1 - 0 con un único fallo de disco.

Figura 12.12 RAID 0 + 1 y 1 + 0.

- Puede implementarse el mecanismo RAID en el hardware de la matriz de almacenamiento. Dicha matriz puede crear conjuntos RAID de varios niveles y puede incluso dividir esos conjuntos en volúmenes más pequeños, que serán los que se presenten en el sistema operativo. El sistema operativo sólo necesitará implementar el sistema de archivos en cada uno de los volúmenes. Las matrices pueden tener disponibles múltiples conexiones o pueden formar parte de una SAN, permitiendo que múltiples *host* aprovechen la funcionalidad de la matriz de almacenamiento.
- El mecanismo RAID puede implementarse en el nivel de interconexión SAN mediante dispositivos de visualización de disco. En este caso, se inserta un dispositivo entre los *host* y el almacenamiento. Dicho dispositivo acepta comandos de los servidores y gestiona el acceso a los dispositivos de almacenamiento. Por ejemplo, puede proporcionar una duplicación en espejo escribiendo cada bloque en dos dispositivos de almacenamiento separados.

También pueden implementarse en cada uno de estos niveles otras características, como las instantáneas y los mecanismos de replicación. La replicación implica la duplicación automática de las escrituras en nodos separados, para proporcionar redundancia y para recuperación de desastres. La replicación puede ser síncrona o asíncrona. En la replicación síncrona, cada bloque debe escribirse local y remotamente antes de que la escritura se considere completa, mientras que en la replicación asíncrona se agrupan las escrituras y se escribe periódicamente. La replicación asíncrona puede provocar la pérdida de datos en caso de que el nodo primario falle, pero se trata de un mecanismo más rápido y que no tiene limitaciones de distancia.

La implementación de estas características difiere dependiendo del nivel en que se implemente el mecanismo RAID. Por ejemplo, si se implementa el mecanismo RAID por software, cada *host* puede tener que implementar y gestionar su propio esquema de replicación. Sin embargo, si la replicación se implementa en la matriz de almacenamiento o en la interconexión SAN, entonces independientemente de cuáles sean las funcionalidades del sistema operativo del *host*, podrán replicarse los datos de los *hosts*.

Otro aspecto importante de la mayoría de las implementaciones RAID es la utilización de un disco o discos de reserva en caliente. Los discos de reserva en caliente no se utilizan para almacenar datos, sino que están configurados para entrar en acción como sustitutos en caso de que falle algún otro de los discos. Por ejemplo, puede utilizarse un disco de reserva en caliente para reconstruir una pareja de replicación en espejo en caso de que uno de los discos de la pareja falle. De esta forma, puede restablecerse automáticamente el nivel RAID, sin esperar a que sea sustituido el disco fallido. Si se asigna más de un disco de reserva en caliente, podrá repararse más de un fallo sin necesidad de intervención humana.

12.7.4 Selección de un nivel RAID

Dadas las múltiples opciones existentes, ¿cómo eligen un nivel RAID los diseñadores de sistemas? Una de las consideraciones principales es la velocidad de reconstrucción. En caso de que falle un disco, el tiempo necesario para reconstruir sus datos puede ser significativo y variará dependiendo del nivel RAID utilizado. La reconstrucción es más sencilla en RAID nivel 1, ya que pueden copiarse los datos de otro disco; para los otros niveles, necesitamos acceder a los demás discos de la matriz para reconstruir los datos de un disco fallido. La velocidad de reconstrucción en un sistema RAID puede ser un factor importante si se necesita un suministro continuo de datos, como es el caso en los sistemas de bases de datos interactivas o de altas prestaciones. Además, la velocidad de reconstrucción influye sobre el tiempo medio entre fallos. El tiempo de reconstrucción puede ser de horas en las soluciones RAID 5, cuando haga falta reconstruir grandes conjuntos de discos.

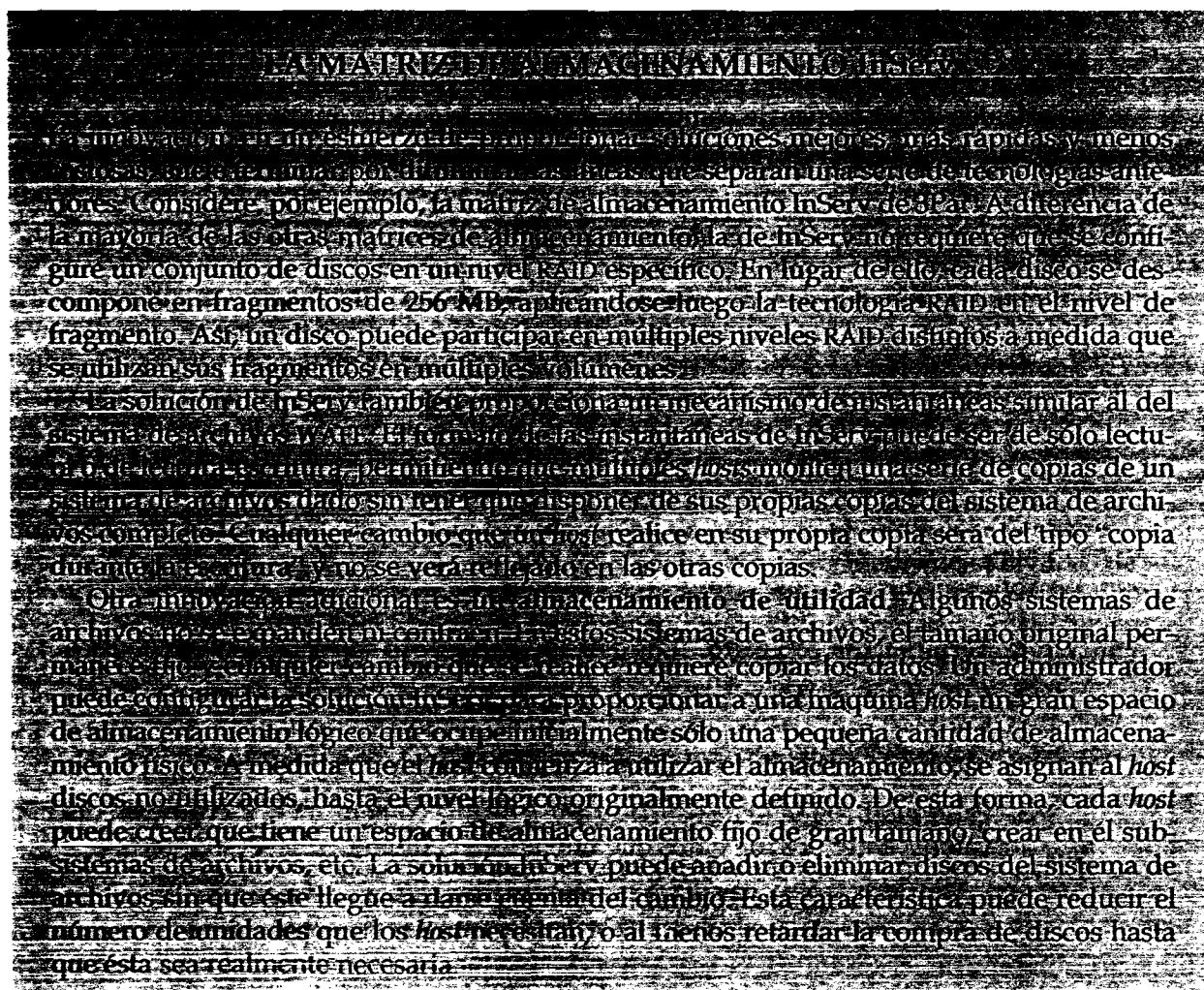
RAID nivel 0 se utiliza en las aplicaciones de altas prestaciones cuando el problema de la pérdida de datos no es crítico. RAID nivel 1 resulta bastante popular para aquellas aplicaciones que requieren una alta fiabilidad con una capacidad de recuperación rápida. RAID 0 + 1 y 1 + 0 se utilizan cuando son importantes tanto las prestaciones como la fiabilidad, como por ejemplo en las bases de datos de pequeño tamaño. Debido al gran espacio adicional necesario en las soluciones de tipo RAID 1, suele preferirse utilizar RAID nivel 5 para almacenar grandes volúmenes de datos.

El nivel 6 no está soportado actualmente en muchas implementaciones RAID, pero ofrece en general una mayor fiabilidad que el nivel 5.

Los diseñadores de sistemas RAID y los administradores de dispositivos de almacenamiento tienen que tomar también otras decisiones. Por ejemplo, ¿cuántos discos debe haber en un determinado conjunto RAID? ¿Cuántos bits deben estar protegidos por cada bit de paridad? Cuantos más discos haya en una matriz, mayores serán las velocidades de transferencia de datos, pero el sistema será también más caro. Cuantos más bits estén protegidos por cada bit de paridad, el espacio adicional necesario requerido para los bits de paridad es menor, pero la probabilidad de que falle un segundo disco antes de que el primer disco fallido sea reparado será mayor y eso provocará una mayor tasa de pérdida de datos.

12.7.5 Extensiones

Los conceptos de RAID se han generalizado a otros dispositivos de almacenamiento, incluyendo las matrices de cintas, e incluso a la difusión de datos en sistemas inalámbricos. Cuando se aplican a las matrices de cintas, las estructuras RAID son capaces de recuperar los datos incluso si una de las cintas de la matriz resulta dañada. Cuando se aplican a la difusión de datos, cada bloque de datos se divide en unidades de corta longitud y se difunde junto con una unidad de paridad; si no se recibe una de las unidades por alguna razón, podrá ser reconstruida a partir de las otras unidades. Comúnmente, los robots de las unidades de cinta que contienen múltiples cintas dividirán los datos en bandas entre todas las unidades, para incrementar la tasa de transferencia y reducir el tiempo requerido para la realización de copias de seguridad.



12.7.6 Problemas de las soluciones RAID

Desafortunadamente, las soluciones RAID no garantizan siempre que los datos estén disponibles para el sistema operativo y sus usuarios. Por ejemplo, podría haber un puntero erróneo a un archivo, o una serie de punteros dentro de la estructura de archivos podría corromperse. Igualmente, las escrituras incompletas, si no se realiza una recuperación adecuada, podrían provocar la corrupción de los datos. Del mismo modo, algún otro proceso podría escribir accidentalmente sobre las estructuras de un sistema de archivos. Las soluciones RAID protegen frente a los errores de los soportes físicos, pero no frente a otros errores hardware o software; y como el abanico de posibles errores hardware y software es tan amplio, también es muy amplia la gama de peligros potenciales de pérdida de datos en un sistema.

El sistema de archivos ZFS de Solaris adopta un innovador enfoque para tratar de resolver estos problemas. Este sistema mantiene una serie de sumas de comprobación internas para todos los bloques, incluyendo tanto los datos como los metadatos. La funcionalidad añadida estriba en la ubicación de las sumas de comprobación; dichas sumas no se almacenan con el bloque correspondiente, sino con el puntero que hace referencia a dicho bloque. Considere un inodo con una serie de punteros a sus datos; dentro del inodo se almacenará la suma de comprobación de cada bloque de datos. Si se produce un problema con los datos, la suma de comprobación será incorrecta y el sistema de archivos podrá detectar esa circunstancia. Si los datos están duplicados en espejo y existe un bloque con una suma de comprobación correcta y otro bloque con una suma incorrecta, ZFS actualizará automáticamente el bloque erróneo con los datos contenidos en el bloque correcto. De forma similar, la entrada de directorio que apunta al inodo tiene una suma de comprobación para el inodo. Cualquier problema que se produzca en el inodo será detectado en el momento de acceder al directorio. Estas sumas de comprobación se emplean en todas las estructuras de ZFS, proporcionando un nivel de coherencia, de detección de errores y de corrección de errores mucho más alto del que puede obtenerse con los conjuntos de discos RAID o los sistemas de archivos estándar. El espacio y la capacidad de procesamiento adicionales requeridos por los cálculos de las sumas de comprobación y los ciclos adicionales de lectura-modificación-escritura de los bloques no son perceptibles, porque el funcionamiento global de ZFS es muy rápido.

12.8 Implementación de un almacenamiento estable

En el Capítulo 6 hemos presentado el registro de escritura anticipada, que requiere la disponibilidad de un almacenamiento estable. Por definición, la información que reside en un almacenamiento estable no se pierde *nunca*. Para implementar este tipo de almacenamiento, necesitamos replicar la información necesaria en múltiples dispositivos de almacenamiento (usualmente discos) con modos independientes de fallo. Necesitamos coordinar la escritura de las actualizaciones en una forma que garantice que un fallo durante una actualización no dejará todas las copias en un estado incorrecto y que, cuando nos estemos recuperando de un fallo, podamos forzar a todas las copias para que adopten un valor coherente y correcto, incluso si se produce otro fallo durante la recuperación. En esta sección, vamos a ver cómo satisfacer estas necesidades.

Una escritura en disco puede tener uno de tres posibles resultados:

1. **Terminación con éxito.** Los datos se han escrito correctamente en el disco.
2. **Fallo parcial.** Ha ocurrido un fallo en mitad de una transferencia, por lo que sólo se han escrito algunos de los sectores con los nuevos datos, y el sector que estuviera siendo escrito cuando se produjo el fallo puede haber resultado corrompido.
3. **Fallo total.** El fallo ha ocurrido antes de que comenzara la escritura en disco, por lo que los valores anteriores de datos que hubiera en el disco permanecen intactos.

Cuando se produce un fallo durante la escritura de un bloque, el sistema necesita detectarlo e invocar un procedimiento de recuperación para restaurar el bloque a un estado coherente. Para hacer esto, el sistema debe mantener dos bloques físicos por cada bloque lógico. Cada operación de salida se ejecuta de la forma siguiente:

1. Se escribe la información en el primer bloque físico.
2. Cuando la primera escritura se ha completado adecuadamente, se escribe la misma información en el segundo bloque físico.
3. La operación se declara completa sólo después de que la segunda escritura se haya completado con éxito.

Durante la recuperación de un fallo, se examina cada pareja de bloques físicos. Si ambos coinciden y no hay ningún error detectable, no será necesaria ninguna acción adicional. Si uno de los bloques contiene un error detectable, sustituiremos su contenido con el valor del otro bloque. Si ninguno de los bloques contiene un error detectable pero los bloques difieren en cuanto a su contenido, sustituiremos el contenido del primer bloque con el segundo. Este procedimiento de recuperación garantiza que cada escritura de almacenamiento estable tenga éxito de forma completa o, por el contrario, no provoque ningún cambio.

Podemos ampliar este procedimiento fácilmente para utilizar un número arbitrariamente grande de copias de cada bloque de almacenamiento estable. Aunque disponer de un gran número de copias reduce aún más la probabilidad de que se produzca un fallo, usualmente es razonable simular el almacenamiento estable con sólo dos copias. Se garantiza que los datos en almacenamiento estable serán seguros, a menos que un fallo destruya todas las copias.

Puesto que consume mucho tiempo esperar a que las escrituras de disco se completen (E/S sincrona), muchas matrices de almacenamiento añaden una memoria NVRAM como caché. Puesto que la memoria es no volátil (usualmente tiene una alimentación por batería como respaldo de la alimentación normal de la unidad), podemos confiar en que almacene los datos dirigidos a los discos. Por tanto, se la considera parte del almacenamiento estable. Las escrituras a este tipo de memoria son mucho más rápidas que al disco, por lo que se mejoran las prestaciones enormemente.

12.9 Estructura de almacenamiento terciario

¿Compraría usted un magnetoscopio que sólo tuviera una cinta en su interior que no se pudiera sacar ni sustituir? ¿O un reproductor de DVD o CD que sólo tuviera un único disco en su interior, imposible de extraer? ¡Por supuesto que no! Lo normal es utilizar un magnetoscopio o un reproductor de CD con muchas cintas de vídeo o discos ópticos distintos, siendo esas cintas y discos de bajo coste. De la misma forma, en una computadora puede reducirse el coste global utilizando muchos discos de bajo coste con una misma unidad. El bajo coste es la característica definitoria del almacenamiento terciario, del cual vamos a hablar en esta sección.

12.9.1 Dispositivos de almacenamiento terciario

Puesto que el coste es tan importante, en la práctica el almacenamiento terciario se construye con **soportes extraíbles**. Los ejemplos más comunes son los disquetes, las cintas y los CD y DVD de sólo lectura, de una única lectura o reescribibles. Hay disponibles, asimismo, muchos otros tipos de dispositivos de almacenamiento terciarios, incluyendo dispositivos extraíbles que almacenan los datos en memoria flash e interactúan con el sistema informático a través de una interfaz USB.

12.9.1.1 Discos extraíbles

Los discos extraíbles son uno de los tipos de almacenamiento terciario. Los disquetes constituyen un ejemplo de disco magnético extraíble: están formados por un fino disco flexible, recubierto de material magnético y encerrado en una carcasa plástica protectora. Aunque los disquetes comunes pueden almacenar sólo entorno a 1 MB, se utiliza una tecnología similar para discos magnéticos extraíbles que pueden albergar más de 1 GB de datos. Los discos magnéticos extraíbles pueden ser casi tan rápidos como los discos duros, aunque la superficie de grabación tiene un mayor riesgo de resultar dañada por posibles arañazos.

Otro tipo de disco extraíble es el **disco magneto-óptico**. Este tipo de disco almacena los datos en una placa rígida recubierta de material magnético, pero la tecnología de grabación es muy distinta de la de los discos magnéticos. El cabezal magneto-óptico vuela mucho más alejado de la superficie del disco de lo que lo hace el cabezal de un disco magnético, y el material magnético está cubierto por una gruesa protectora de plástico o de cristal. Esta disposición hace que el disco sea mucho más resistente a los aterrizajes de cabezales.

La unidad tiene una bobina que genera un campo magnético; a temperatura ambiente, el campo es demasiado débil para poder magnetizar un bit del disco. Para escribir un bit, el cabezal del disco dirige un rayo láser a la superficie del disco. El láser se dirige a un diminuto punto en el que hay que escribir el bit. El láser calienta este punto, lo que hace que éste sea susceptible al campo magnético. De este modo, ese débil campo magnético puede grabar un bit sobre la superficie.

El cabezal magneto-óptico está demasiado alejado de la superficie del disco como para leer los datos detectando los diminutos campos magnéticos en la forma en que lo hace el cabezal de un disco duro. En lugar de ello, la unidad lee cada bit utilizando una propiedad de los rayos láser denominada **efecto Kerr**. Cuando un rayo láser rebota en un punto magnetizado de la superficie, la polarización del rayo gira en el sentido de las agujas del reloj o en sentido contrario de las agujas del reloj, dependiendo de la orientación del campo magnético. Es esta rotación lo que el cabezal detecta para leer un bit.

Otra categoría de discos extraíbles son los **discos ópticos**. Los discos ópticos no utilizan en absoluto el magnetismo. En lugar de ello, emplean materiales especiales que pueden modificarse mediante rayos láser para tener puntos más oscuros o más brillantes. Un ejemplo de tecnología de disco óptico es el **disco de cambio de fase**, que está recubierto de un material que puede solidificarse en un estado cristalino o amorfo. El estado cristalino es más transparente y, por tanto, un rayo láser será más brillante cuando pase a través del material y rebote en la capa reflectora. Las unidades de cambio de fase utilizan rayos láser con tres diferentes potencias: baja potencia para leer los datos, potencia media para borrar el disco fundiendo y volviendo a solidificar el medio de grabación en un estado cristalino y alta potencia para fundir el medio en un estado amorfo a la hora de escribir en el disco. Los ejemplos más comunes de esta tecnología son los discos CD-RW y DVD-RW regrabables.

Los tipos de discos que acabamos de describir pueden utilizarse una y otra vez, por lo que se denominan **discos de lectura-escritura**. Por contraste, los **discos de una única escritura** (WORM, write-one, read-many) sólo pueden escribirse una vez. Una forma antigua de fabricar un disco WORM consiste en disponer una fina capa de aluminio entre dos placas de plástico o cristal. Para escribir un bit, la unidad utiliza un rayo láser con el fin de hacer un pequeño agujero en el aluminio. Este proceso no puede revertirse. Aunque es posible escribir la información contenida en un disco WORM agujereándolo por todas partes, es virtualmente imposible alterar los datos en el disco, porque los agujeros únicamente pueden añadirse y no eliminarse, y el código ECC asociado con cada sector detectará, probablemente, dichas adiciones. Los discos WORM se consideran duraderos y fiables, porque la capa de metal está encapsulada de forma segura entre las placas protectoras de cristal o de plástico y los campos magnéticos no pueden dañar la grabación. Otra tecnología más reciente de una única escritura realiza la grabación sobre una tinta polimérica orgánica, en lugar de sobre una capa de aluminio. La tinta absorbe la luz láser para formar una serie de marcas. Esta tecnología se utiliza en los discos CD-R y DVD-R grabables.

Los **discos de sólo lectura**, como los discos CD-ROM y DVD-ROM, salen de fábrica con los datos ya grabados. Utilizan una tecnología similar a la de discos WORM (aunque los bits se graban por presión y no quemando la superficie) y son muy duraderos.

La mayoría de los discos extraíbles son más lentos que sus equivalentes no extraíbles. El proceso de escritura requiere más tiempo, y también es más lenta la rotación y (en ocasiones) el tiempo de búsqueda es mayor.

12.9.1.2 Cintas

La cinta magnética es otro tipo de soporte extraíble. Como regla general, una cinta puede almacenar más datos que un cartucho de disco óptico o magnético. Las unidades de cinta o las unidades

de disco tienen velocidades de transferencia similares, pero el acceso aleatorio a una cinta es mucho más lento que una búsqueda de disco, porque requiere una operación de bobinado o rebobinado rápido que necesita decenas de segundos o incluso minutos.

Aunque una unidad de cinta típica es más cara que una unidad de disco normal, el precio de un cartucho de cinta es más bajo que el de un disco magnético de capacidad equivalente, por lo que la cinta es un soporte barato para todas aquellas aplicaciones donde no se necesite un acceso aleatorio rápido. Las cintas se utilizan comúnmente para albergar copias de seguridad de los datos de los discos. También se las emplea en los grandes centros de supercomputación para almacenar los enormes volúmenes de datos utilizados en la investigación científica y por las grandes organizaciones de carácter comercial.

Las instalaciones de cintas de mayor envergadura utilizan normalmente cambiadores robóticos de cintas, que transfieren las cintas entre las unidades de cinta y las ranuras de almacenamiento de una biblioteca de cintas. Estos mecanismos proporcionan a la computadora un acceso automatizado a un conjunto muy grande de cartuchos de cinta.

Una biblioteca robotizada de cintas puede reducir el coste global de almacenamiento de los datos. Un archivo residente en disco que no vaya a ser necesario durante un tiempo puede ser archivado en cinta, donde el coste por gigabyte es inferior; si el archivo se necesita en el futuro, la computadora puede volverlo a cargar en disco para utilizarlo de forma activa. Una biblioteca robotizada de cinta se denomina en ocasiones almacenamiento casi en línea, puesto que constituye una solución intermedia entre las altas prestaciones de los discos magnéticos en línea y el bajo coste de las cintas fuera de línea que estén almacenadas en una serie de estantes dentro de una sala de archivo.

12.9.1.3 Tecnologías futuras

En el futuro, otras tecnologías de almacenamiento llegarán a cobrar una gran importancia. Una prometedora tecnología de almacenamiento, el **almacenamiento holográfico**, utiliza rayos láser para grabar fotografías holográficas sobre soportes especiales. Podemos pensar en un holograma como una matriz tridimensional de píxeles. Cada píxel representa un bit: 0 representa el negro y 1 representa el blanco. Todos los píxeles de un holograma se transfieren mediante un único pulso de luz láser, por lo que la velocidad de transferencia de los datos es extremadamente alta. A medida que continúen los desarrollos, cabe esperar que el almacenamiento holográfico llegue a ser comercialmente viable.

Otra tecnología de almacenamiento para la que se están investigando activamente se basa en los **sistemas mecánicos microelectrónicos** (MEMS, micro-electronic mechanical systems). La idea consiste en aplicar las tecnologías de fabricación utilizadas para producir chips electrónicos a la fabricación de pequeñas máquinas de almacenamiento de datos. Una de las propuestas sugiere fabricar una matriz de 10000 diminutos cabezales de disco, con un centímetro cuadrado de material de almacenamiento magnético suspendido por encima de esa matriz. Cuando se mueve el material de almacenamiento longitudinalmente sobre los cabezales, cada cabezal accede a su propia pista lineal de datos grabados en el material. El material de almacenamiento puede desplazarse lateralmente un poco para permitir a todos los cabezales acceder a su siguiente pista. Aunque todavía queda por ver si esta tecnología puede llegar a tener éxito, podría proporcionar una tecnología de almacenamiento no volátil de datos más rápida que los discos magnéticos y más barata que la memoria DRAM semiconductor.

Independientemente de si el medio de almacenamiento es un disco magnético extraíble, un DVD o una cinta magnética, el sistema operativo necesita proporcionar diversas funcionalidades para poder utilizar los soportes extraíbles para el almacenamiento de datos. Estas funcionalidades se analizan en la Sección 12.9.2.

12.9.2 Soporte del sistema operativo

Dos de los principales cometidos de un sistema operativo son gestionar los dispositivos físicos y presentar una abstracción de máquina virtual a las aplicaciones. En este capítulo, hemos visto que, para los discos duros, el sistema operativo proporciona dos abstracciones. Una es el dispositivo

sin formato, compuesto simplemente por una matriz de bloques de datos. La otra es un sistema de archivos. Para el sistema de archivos en un disco magnético, el sistema operativo pone en cola y planifica las solicitudes entrelazadas emitidas por varias aplicaciones. Ahora, vamos a ver cómo lleva a cabo el sistema operativo su tarea cuando el soporte físico de almacenamiento es de tipo extraíble.

12.9.2.1 Interfaz de aplicación

La mayoría de los sistemas operativos pueden gestionar los discos extraíbles de forma casi exactamente igual que los discos fijos. Cuando se inserta un cartucho en blanco en la unidad (o cuando se monta ese cartucho de almacenamiento), el cartucho debe formatearse, con lo que se escribe en el disco un sistema de archivos vacío. Este sistema de archivos se utiliza igual que los sistemas de archivo residentes en disco duro.

Las cintas suelen manejarse de modo distinto. El sistema operativo, usualmente, presenta las cintas como soportes de almacenamiento sin formato. Las aplicaciones no abren un archivo en la cinta, sino que abren la unidad de cinta completa como un dispositivo sin formato. Usualmente, la unidad de cinta queda entonces reservada para el uso exclusivo por parte de dicha aplicación, hasta que la aplicación termina o hasta que cierra el dispositivo de cinta. Esta exclusividad tiene un gran sentido, porque el acceso aleatorio en una cinta puede requerir decenas de segundos o incluso minutos, por lo que tratar de entrelazar una serie de accesos aleatorios a las cintas procedentes de más de una aplicación provocaría un intenso fenómeno de sobrepaginación.

Cuando se presenta la unidad de cinta como dispositivo sin formato, el sistema operativo no proporciona servicios de sistemas de archivos. La aplicación debe decidir cómo utilizar la matriz de bloques. Por ejemplo, un programa que realice una copia de seguridad en cinta de un disco duro puede almacenar una lista de nombres y tamaños de archivos al principio de la cinta y luego copiar los datos de los archivos en la cinta en el orden correspondiente.

Resulta fácil ver los problemas que pueden surgir debido a esta forma de utilizar las cintas. Puesto que cada aplicación tiene sus propias reglas en cuanto al modo de organizar una cinta, cada cinta llena de datos sólo puede, por regla general, ser utilizada por el programa que la creó. Por ejemplo, incluso si sabemos que una cinta de copia de seguridad contiene una lista de nombres y tamaños de archivos, seguida de los datos de los archivos en el orden en que aparecen en la lista, seguiría siendo difícil poder utilizar la cinta porque ¿cómo están almacenados exactamente los nombres de archivo? ¿Están expresados los tamaños de archivo en formato binario o en formato ASCII? ¿Está cada archivo escrito en un bloque distinto o están todos concatenados en una única cadena de bytes extremadamente larga? Ni siquiera conocemos el tamaño de los bloques en la cinta, porque esta variable es una de las que, generalmente, se pueden elegir de forma separada para cada bloque que se escriba.

Para una unidad de disco, las operaciones básicas son `read()`, `write()` y `seek()`, pero las unidades de cinta tienen un conjunto distinto de operaciones básicas. En lugar de `seek()`, una unidad de cinta utiliza la operación `locate()`. La operación `locate()` en las cintas es más precisa que la operación `seek()` en los discos, porque sitúa la cinta en un bloque lógico específico, en lugar de situarla en una pista completa determinada. Si se utiliza esa operación para posicionarse en el bloque 0 se obtiene el mismo resultado que si se rebobinara la cinta.

En la mayoría de los tipos de unidades de cinta, se puede localizar cualquier bloque que haya sido escrito en la cinta. Sin embargo, en una cinta parcialmente llena no será posible situarse en el espacio vacío existente más allá del área escrita, porque la mayoría de las unidades de cinta no gestionan su espacio físico de la misma forma que las unidades de disco. En una unidad de disco, los sectores tienen un tamaño fijo y debe utilizarse un proceso de formateo para colocar sectores vacíos en sus posiciones finales, antes de poder escribir ningún dato. Sin embargo, la mayoría de las unidades de cinta tienen un tamaño de bloque variable y el tamaño de cada bloque se determina sobre la marcha, en el momento de escribir el bloque. Si se encuentra un área de cinta defectuosa durante la escritura, se salta el área defectuosa y el bloque se escribe de nuevo. Esta operación explica por qué no es posible situarse en el espacio vacío existente más allá del área escrita: las posiciones y los números de los bloques lógicos todavía no han sido determinados para ese área.

La mayoría de las unidades de cinta tienen una operación `read_position()` que devuelve el número de bloque lógico sobre el que está situado el cabezal de la cinta. Muchas unidades de cinta soportan también una operación `space()` de movimiento relativo; por ejemplo, la operación `space(-2)` efectuaría un salto de dos bloques lógicos hacia atrás.

Para la mayoría de los tipos de unidades de cintas, escribir un bloque tiene el efecto colateral de borrar desde un punto de vista lógico todo aquello que esté situado más allá de la posición donde se haya efectuado la escritura. En la práctica, este efecto colateral implica que la mayoría de las unidades de cinta son dispositivos donde sólo puede añadirse información, porque la actualización de un bloque en mitad de la cinta hace que se borre en la práctica toda la información situada detrás de dicho bloque. La unidad de cinta implementa esta operación de adición grabando un marcador de fin de cinta (EOT, end-of-tape) después de cada bloque escrito. La unidad se negará a situarse más allá de la marca EOT, aunque lo que sí se puede es situarse en la propia marca EOT y luego comenzar a escribir. Al hacer esto, se sobreescribe la anterior marca EOT y se coloca una nueva al final de los nuevos bloques recién escritos.

En principio, podemos implementar un sistema de archivos en una cinta, pero muchas de las estructuras de datos y algoritmos de ese sistema de archivos serían distintos de los que se utilizan para los discos, debido a la propiedad de “sólo adición” que tienen las cintas a la hora de grabar información.

19.9.2.2 Denominación de archivos

Otra cuestión que el sistema operativo necesita resolver es cómo nombrar los archivos en los soportes extraíbles. En un disco fijo, el proceso de denominación no es difícil. En un PC, el nombre de archivo está compuesto por una letra de unidad seguido por un nombre de ruta; en UNIX, el nombre de archivo no contiene la letra de unidad, pero la tabla de montaje permite al sistema operativo descubrir en qué unidad se ubica el archivo. Si el disco es extraíble, sin embargo, el saber qué unidad contenía el cartucho en algún momento del pasado no implica que sepamos cómo encontrar el archivo actualmente. Si todo cartucho extraíble del mundo tuviera un número de serie distinto, el nombre de un archivo en un dispositivo extraíble podría prefijarse con el número de serie, pero garantizar que no hubiera dos números de serie normales requeriría que cada uno tuviera unos 12 dígitos de longitud. ¿Quién podría recordar los nombres de esos archivos si tuviera que memorizar un número de serie de 12 dígitos para cada uno?

El problema se vuelve aún más difícil cuando queremos escribir datos en un cartucho extraíble en una computadora y luego usar el cartucho en otra computadora distinta. Si ambas máquinas son del mismo tipo y tiene el mismo tipo de unidad extraíble, la única dificultad es conocer el contenido y la disposición de los datos en el cartucho; pero si las máquinas o las unidades son diferentes, pueden surgir muchos problemas adicionales. Incluso si las unidades son compatibles, las diferentes computadoras pueden almacenar los bytes en orden distinto y pueden utilizar diferente tipo de codificación para los números binarios e incluso para las letras (como por ejemplo ASCII en un PC, por contraposición al código EBCDIC utilizado en las máquinas de tipo *mainframe*).

Los sistemas operativos actuales dejan generalmente el problema del espacio de nombres sin resolver para los medios extraíbles y depende de las aplicaciones y de los usuarios averiguar cómo acceder e interpretar los datos. Afortunadamente, hay unos cuantos tipos de soportes extraíbles que están tan bien estandarizados que todas las computadoras los usan de la misma manera. Uno de los ejemplos es el CD. Un CD de música utiliza un formato universal que puede ser comprendido por cualquier unidad de CD. Los CD de datos sólo están disponibles en unos cuantos formatos diferentes, por lo que resulta usual que las unidades de CD y los controladores de dispositivo de los sistemas operativos se programen para que puedan gestionar todos los formatos comunes. Los formatos DVD también están bien estandarizados.

12.9.2.3 Gestión del almacenamiento jerárquico

Un *intercambiador* de cintas o discos (*jukebox*) robotizado permite a la computadora cambiar el carroza de lectura entre una cinta o en una unidad de disco sin intervención humana. Dos de los usos

principales de esta tecnología son la realización de copias de seguridad y los sistemas de almacenamiento jerárquico. La utilización de un intercambiador para copias de seguridad es muy simple: cuando se llena el cartucho, la computadora instruye al intercambiador para que conmute al cartucho siguiente. Algunos intercambiadores albergan decenas de unidades y miles de cartuchos, con una serie de brazos robotizados que gestionan el movimiento de las cintas hacia las unidades.

Los sistemas de almacenamiento jerárquico amplían el concepto de jerarquía de almacenamiento más allá de la memoria principal del almacenamiento secundario (es decir, los discos magnéticos), para incorporar también un almacenamiento terciario. El almacenamiento terciario se implementa usualmente como un intercambiador de cintas o de discos extraíbles. Este nivel de la jerarquía de almacenamiento tiene una mayor capacidad, es más barato y también es más lento.

Aunque el sistema de memoria virtual puede ampliarse de forma sencilla al almacenamiento terciario, esta ampliación raramente se llevará a cabo en la práctica. La razón es que una extracción de datos de un intercambiador puede requerir decenas de segundos o incluso minutos, y un retardo tan grande resulta intolerable para los mecanismos de paginación bajo demanda y para otros tipos de utilización de la memoria virtual.

La forma usual de incorporar el almacenamiento terciario consiste en ampliar el sistema de archivos. Los archivos de pequeño tamaño y los frecuentemente utilizados permanecen en el disco magnético, mientras que los archivos antiguos y de gran tamaño que no se utilicen de manera activa se archivan en el intercambiador. En algunos sistemas de archivado de archivos, la entrada de directorio correspondiente al archivo se mantiene, pero el contenido del archivo deja de ocupar espacio en el almacenamiento secundario. Si una aplicación trata de abrir el archivo, se suspende la llamada al sistema `open()` hasta que el contenido del archivo pueda leerse desde el almacenamiento terciario. Cuando el contenido vuelve a estar disponible en el disco magnético, la operación `open()` devuelve el control a la aplicación, que procederá a utilizar la copia de los datos residente en el disco.

Hoy en día, los sistemas de gestión del almacenamiento jerárquico (HSM, hierarchical storage management) suelen encontrarse en instalaciones que tengan grandes volúmenes de datos que se utilicen raramente, esporádicamente o periódicamente. Los actuales trabajos de investigación y desarrollo en tecnologías HSM incluyen la extensión de estos conceptos para proporcionar una gestión del ciclo de vida de la información (ILM, information life-cycle management). En esta tecnología, los datos se mueven de disco a cinta y de vuelta a disco, según sea necesario, pero se borran de manera planificada o de acuerdo con una cierta política. Por ejemplo, algunas instalaciones guardan los correos electrónicos durante siete años pero desean asegurarse de que se destruyan al final de ese período de siete años. En dicho momento, los datos podrían estar residiendo en disco, en una cinta HSM y en una cinta de copia de seguridad. La tecnología ILM centraliza el conocimiento acerca de la ubicación de los datos, de modo que puedan aplicarse las políticas adecuadas en todas esas ubicaciones.

12.9.3 Cuestiones de rendimiento

Al igual que sucede con cualquier otro componente del sistema operativo, los tres aspectos más importantes del rendimiento del almacenamiento terciario son la velocidad, la fiabilidad y el coste.

12.9.3.1 Velocidad

La velocidad del almacenamiento terciario puede analizarse desde dos puntos de vista: ancho de banda y latencia. El ancho de banda se mide en bytes por segundo; el **ancho de banda sostenido** es la velocidad media de transferencia de datos durante una transferencia de gran envergadura, es decir, el número de bytes dividido por el tiempo total de transferencia. El **ancho de banda efectivo** mide el promedio para toda la duración de la operación de E/S, incluyendo el tiempo requerido para efectuar las operaciones `seek()` o `locate()`, así como el tiempo de conmutación de cartuchos en un intercambiador. En esencia, el ancho de banda sostenido es la velocidad de trans-

ferencia de datos mientras está transmitiéndose el flujo de datos, mientras que el ancho de banda efectivo es la velocidad global de transferencia de datos proporcionada por la unidad. Cuando se habla de *ancho de banda de una unidad*, generalmente se está haciendo referencia al ancho de banda sostenido.

Para los discos extraíbles, el ancho de banda va desde unos pocos megabytes por segundo para las unidades más lentas, hasta más de 40 MB por segundo para las unidades más rápidas. Las cintas tienen un rango similar de anchos de banda, desde unos cuantos megabytes por segundo a más de 30 MB por segundo.

El segundo aspecto que hay que analizar en lo que se refiere a la velocidad es la **latencia de acceso**. Según esta medida de rendimiento, los discos son mucho más rápidos que las cintas: el almacenamiento de un disco es, esencialmente, bidimensional: todos los bits están simultáneamente disponibles para poder acceder a ellos. En un acceso a disco, simplemente se desplaza el brazo de la unidad hasta el cilindro seleccionado y se espera a que transcurra la latencia rotacional, que puede ser inferior a 5 milisegundos. Por contraste, el almacenamiento en cinta es tridimensional: en cualquier momento dado, sólo una pequeña parte de la cinta es accesible por parte del cabezal, estando la mayoría de los bits enterrados bajo cientos o miles de capas de cinta bobinada en el carrete. Un acceso aleatorio a una cinta requiere bobinar los carretes de la cinta hasta que el bloque seleccionado pase por debajo del cabezal de la cinta, lo que puede requerir decenas o centenares de segundos. Por tanto, podemos decir generalmente que el acceso aleatorio en un cartucho de cinta es más de 1000 veces más lento que el acceso aleatorio en un disco.

Si se está utilizando un intercambiador, la latencia de acceso puede ser significativamente mayor. Para poder cambiar un disco extraíble, la unidad debe dejar de rotar y luego el brazo robotizado debe conmutar los cartuchos de disco, tras lo cual la unidad deberá comenzar a rotar de nuevo el cartucho recién introducido. Esta operación requiere varios segundos, lo que es 100 veces más lento que un acceso aleatorio dentro de un disco. Por tanto, la conmutación de discos en un intercambiador tiene un impacto relativamente alto sobre la velocidad de acceso.

En las cintas, el tiempo requerido por el brazo robotizado es aproximadamente igual que para el caso de un disco. Pero para poder conmutar las cintas, generalmente es necesario rebobinar la cinta anterior antes de poder sacarla de la unidad, y dicha operación puede requerir hasta unos 4 minutos. Asimismo, después de cargar una nueva cinta dentro de la unidad, pueden hacer falta varios segundos para que la unidad calibre su funcionamiento de acuerdo con la cinta y se prepare para las operaciones de E/S. Aunque un intercambiador de cintas no muy rápido puede tener un tiempo de conmutación de cinta de 1 o 2 minutos, este tiempo no es mucho mayor que el tiempo de acceso aleatorio a la propia cinta.

Por tanto, para generalizar, podemos decir que el acceso aleatorio en un intercambiador de discos tiene una latencia de decenas de segundos, mientras que el acceso aleatorio de un intercambiador de cintas tiene una latencia de centenares de segundos; conmutar las cintas requiere mucho tiempo, pero conmutar los discos no requiere tanto. Sin embargo, tenga cuidado con estas generalizaciones: algunos intercambiadores de cintas de gama alta pueden rebobinar, expulsar, cargar una nueva cinta y realizar el avance rápido hasta un elemento aleatorio de datos en menos de 30 segundos.

Si sólo prestamos atención a la velocidad de las unidades de un intercambiador, el ancho de banda y la latencia parecen razonables. Sin embargo, si centramos nuestra atención en los propios cartuchos, nos encontramos con un terrible cuello de botella. Considere en primer lugar el ancho de banda. La relación ancho de banda/capacidad de almacenamiento de una biblioteca robotizada es mucho menos favorable que en el caso de un disco fijo. Para leer todos los datos almacenados en un disco duró de gran tamaño podríamos necesitar en torno a una hora, mientras que para leer todos los datos almacenados en una biblioteca de cintas de gran volumen podríamos necesitar años. La situación en lo que respecta a la latencia de acceso es casi igual de negativa. Como ilustración, si ponemos en cola 100 solicitudes dirigidas a una unidad de disco, el tiempo medio de espera será de entorno a un segundo. Sin embargo, si ponemos en cola 100 solicitudes dirigidas a una biblioteca de cintas, el tiempo media de espera podría ser de más de una hora. El bajo coste del almacenamiento terciario es el resultado de disponer de muchos cartuchos de bajo coste que comparten unas cuantas unidades muy caras; pero el mejor uso que se puede dar a una bibli-

teca de soportes extraíbles es el almacenamiento de datos que se utilicen de manera infrecuente porque la biblioteca sólo puede satisfacer un número relativamente pequeño de solicitudes de E/S por hora.

12.9.3.2 Fiabilidad

Aunque muy a menudo tendemos a pensar que *buen rendimiento* significa *alta velocidad*, otro aspecto importante del rendimiento es la *fiabilidad*. Si tratamos de leer ciertos datos y no podemos hacerlo debido a un fallo de la unidad o del soporte físico, desde el punto de vista práctico el *tiempo* de acceso será infinitamente largo y el ancho de banda será infinitamente pequeño. Por tanto, es importante poder comprender cuál es el grado de fiabilidad de los soportes extraíbles.

Los discos magnéticos extraíbles son algo menos fiables que los discos duros fijos, porque el cartucho tiene mayor probabilidad de verse expuesto a condiciones de entorno dañinas, como por ejemplo el polvo, los grandes cambios de temperatura y de humedad y fuerzas mecánicas debidas a golpes o a que se doble el soporte físico. Los discos ópticos se consideran bastante fiables, porque la tapa que almacena los bits está protegida por otra tapa de plástico o cristal transparente. La fiabilidad de la cinta magnética varía enormemente, dependiendo del tipo de unidad. Algunas unidades de bajo coste desgastan las cintas después de unas cuantas docenas de usos, mientras que otros tipos son lo suficientemente poco agresivos como para permitir millones de reutilizaciones. Por comparación con el cabezal de un disco magnético, el cabezal en una unidad de cinta magnética es uno de los puntos débiles. El cabezal de un disco vuela sobre el soporte físico, pero el cabezal de una cinta está en estrecho contacto con la propia cinta. La acción de frotamiento de la cinta puede terminar por desgastar el cabezal después de unos cuantos miles o decenas de miles de horas.

En resumen, podemos decir que una unidad de disco fijo tiende a ser más fiable que una unidad de cinta o de disco extraíble, mientras que un disco óptico tiende a ser más fiable que una cinta o disco magnético. Pero un disco magnético fijo tiene una debilidad: un aterrizaje de cabezales en un disco duro destruye por regla general los datos, mientras que el fallo de una unidad de cinta o de una unidad de disco óptico no provoca normalmente daños en el cartucho de datos.

12.9.3.3 Coste

El coste de almacenamiento es otro factor importante. He aquí un ejemplo concreto de la forma en que los soportes extraíbles pueden reducir el coste global de almacenamiento: suponga que un disco duro con una capacidad de X GB tiene un precio de 200 dólares. De esta cantidad, 190 dólares corresponden a la carcasa, el motor y la controladora y 10 dólares corresponden a los propios platos magnéticos. El coste de almacenamiento para este disco es de 200 dólares/X GB. Ahora, suponga que podemos fabricar esos mismos platos albergándolos en un cartucho extraíble. Para una unidad y 10 cartuchos, el precio total será 190 + 100 dólares y la capacidad será 10X GB, por lo que el coste de almacenamiento es 29 dólares/X por gigabyte. Incluso aunque sea un poco más caro fabricar un cartucho extraíble, el coste por gigabyte del almacenamiento extraíble puede ser muy inferior que el coste por gigabyte de un disco duro, debido a que el coste de la unidad se reparte entre varios cartuchos extraíbles de bajo coste.

Las Figuras 12.13, 12.14 y 12.15 muestran las tendencias de coste por megabyte para la memoria DRAM, para los discos duros magnéticos y para las unidades de cinta. Los precios de estas gráficas son los precios más bajos que hemos podido localizar analizando los anuncios de diversas revistas informáticas y en la World Wide Web al final de cada año. Estos precios reflejan las tendencias en el mercado informático de consumo en el que se mueven la mayoría de los lectores de estas revistas, donde los precios son bajos por comparación con los mercados de las computadoras de tipo mainframe y minicomputadoras. En el caso de las cintas, el precio es para una unidad con una cinta. El coste global del almacenamiento en cinta se reduce mucho a medida que se compran más cintas para utilizarlas con una misma unidad, porque el precio de una cinta es una pequeña fracción del precio de la unidad. Sin embargo, en una biblioteca de cintas de gran tama-

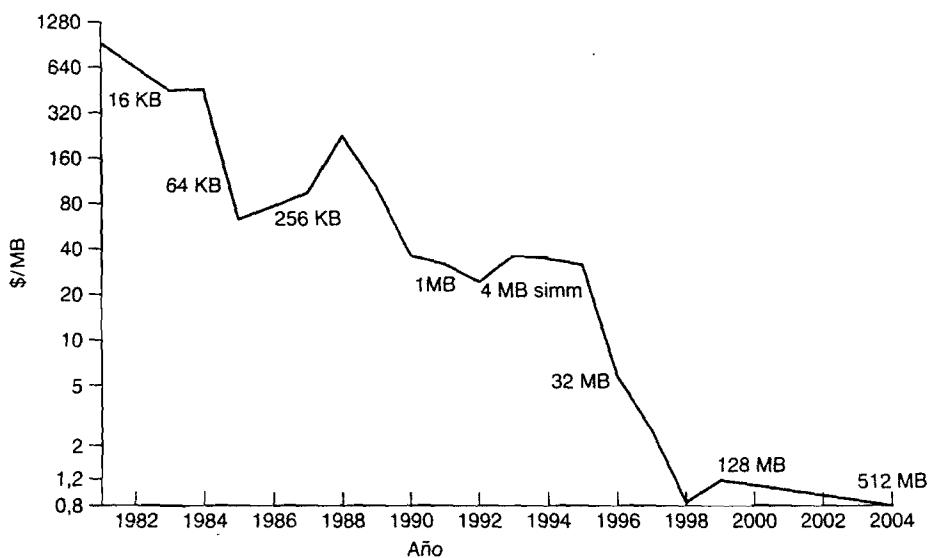


Figura 12.13 Precio por megabyte para las memorias DRAM, entre 1981 y 2004.

ño que contenga miles de cartuchos, el coste de almacenamiento está dominado por el coste de los cartuchos de cinta. En el momento de escribir estas líneas en 2004, el coste por GB de los cartuchos de cinta se situaba aproximadamente en el entorno de 2 dólares.

El coste de la memoria DRAM fluctúa enormemente. En el período comprendido entre 1981 y 2004, podemos ver tres caídas abruptas de precios (en torno a 1981, 1989 y 1996), correspondiendo a los momentos en que un exceso de producción provocó un colapso en el mercado. También podemos ver dos períodos (alrededor de 1987 y 1993) donde la carestía en el mercado provocó significativos aumentos de precio. En el caso de los discos duros, la caída de los precios ha sido mucho más sostenida, aunque parece haberse acelerado desde 1992. Los precios de las unidades de cinta también cayeron de forma sostenida hasta 1997. Desde 1997, el precio por gigabyte de las unidades de cinta de bajo coste ha dejado de caer de forma tan rápida, aunque el precio de la tecnología de cinta de gama media (como por ejemplo DAT/DDS) ha continuado cayendo y ahora se aproxima al de las unidades de bajo coste. No se muestran precios de las unidades de cinta anteriores a 1984 porque, como ya hemos mencionado, las revistas utilizadas para realizar esta

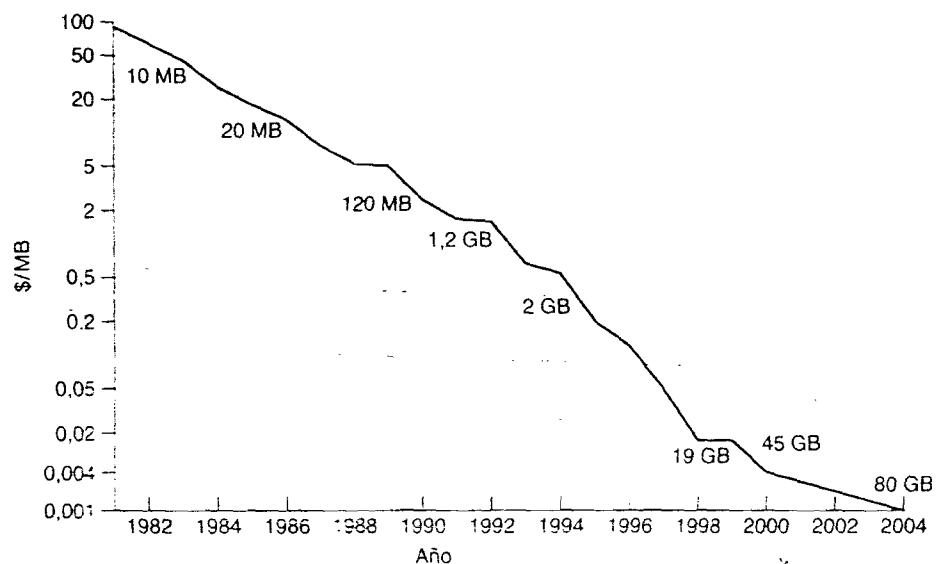


Figura 12.14 Precio por megabyte para los discos duros magnéticos, entre 1981 y 2004.

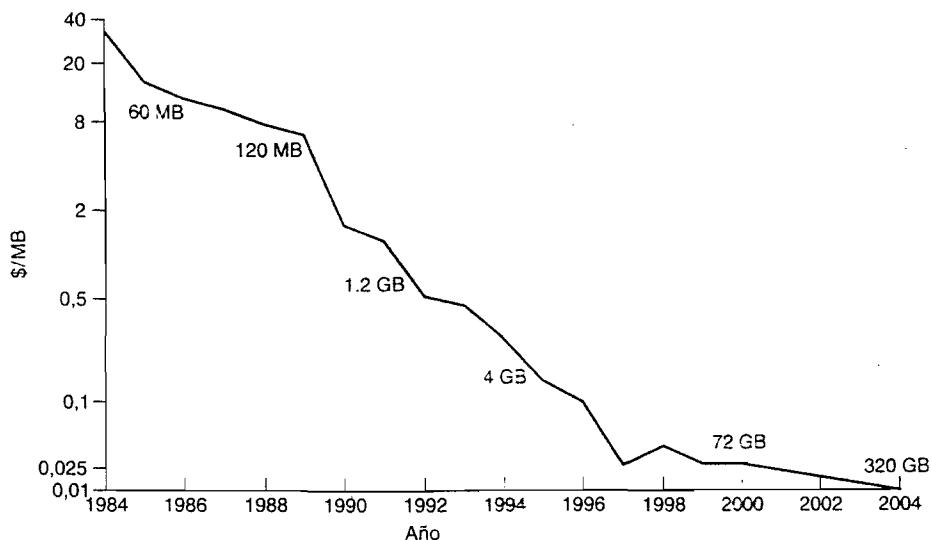


Figura 12.15 Precio por megabyte para una unidad de cinta, entre 1984 y 2004.

comparativa en las tendencias de precios se dirigen al mercado de la informática de consumo y las unidades de cinta no se utilizaban ampliamente en este mercado antes de 1984.

Podemos ver en estas gráficas que el coste del almacenamiento ha caído enormemente en los últimos 20 años. Comparando las gráficas, podemos también ver que el precio del almacenamiento en disco ha bajado en gran medida en relación con el precio de la memoria DRAM y de las cintas.

El precio por megabyte de los discos magnéticos ha mejorado en más de cuatro órdenes de magnitud durante las últimas dos décadas, mientras que la correspondiente mejora para los chips de memoria principal sólo ha sido de tres órdenes de magnitud. La memoria principal hoy en día es más de 100 veces más cara que el almacenamiento en disco.

El precio por megabyte ha caído también mucho más rápidamente para las unidades de disco que para las unidades de cinta. De hecho, el precio por megabyte de una unidad de disco magnético se está aproximando al de un cartucho de cinta (sin tener en cuenta el coste de la propia unidad). En consecuencia, las bibliotecas de cintas de pequeño y medio tamaño tienen un coste de almacenamiento más alto hoy en día que los sistemas de discos con una capacidad equivalente.

La dramática caída en los precios de los discos ha hecho que el almacenamiento terciario llegue a ser casi obsoleto: ya no disponemos de ninguna tecnología de almacenamiento terciario que sea varios órdenes de magnitud más barata que los discos magnéticos. Parece, en consecuencia, que el renacimiento del almacenamiento terciario deberá esperar a que se produzca algún cambio tecnológico revolucionario. Entre tanto, el almacenamiento en cinta verá su uso limitado principalmente a cosas tales como las copias de seguridad de unidades de disco y el archivado definitivo en enormes bibliotecas de cintas que excedan con mucho la capacidad práctica de almacenamiento de las grandes granjas de discos.

12.10 Resumen

Las unidades de disco son los principales dispositivos de E/S de almacenamiento secundario en la mayoría de las computadoras. La mayoría de los dispositivos de almacenamiento secundario son discos magnéticos o cintas magnéticas. Las unidades de disco modernas están estructuradas como una gran matriz unidimensional de bloques lógicos de disco cuyo tamaño es, usualmente, de 512 bytes.

Los discos pueden conectarse a un sistema informático de dos formas distintas: (1) utilizando los puertos de E/S locales de la computadora *host* o (2) utilizando una conexión de red, como por ejemplo redes de área de almacenamiento.

Las solicitudes de E/S de disco son generadas por el sistema de archivos y por el sistema de memoria virtual. Cada solicitud especifica la dirección de disco a la que hay que referenciar, en la forma de un número lógico de bloque. Los algoritmos de planificación de disco pueden mejorar el ancho de banda efectivo, el tiempo medio de respuesta y la varianza del tiempo de respuesta. Algoritmos tales como SSTF, SCAN, C-SCAN, LOOK y C-LOOK están diseñados para llevar a cabo tales optimizaciones, empleando diversas estrategias de ordenación de la cola de solicitudes de disco.

Las prestaciones pueden verse afectadas por el fenómeno de la fragmentación externa. Algunos sistemas disponen de utilidades que permiten explorar el sistema de archivos para identificar los archivos fragmentados, después de lo cual mueven los bloques de un sitio a otro para reducir el grado de fragmentación. Desfragmentar un sistema de archivos muy fragmentado puede mejorar significativamente las prestaciones, aunque el rendimiento del sistema se reducirá mientras esté en marcha el proceso de desfragmentación. Los sistemas de archivos sofisticados, como el sistema Fast File System de UNIX, incorporan muchas estrategias para controlar la fragmentación durante la asignación de espacio, por lo que no es necesario llevar a cabo reorganizaciones del disco.

El sistema operativo gestiona los bloques de disco. En primer lugar, es necesario formatear el disco a bajo nivel para crear los sectores en el hardware sin formato; los nuevos discos normalmente se adquieren ya preformateados. Después, hay que particionar el disco, crear los sistemas de archivos y asignar bloques de arranque para almacenar un programa de arranque del sistema. Finalmente, cuando un bloque se corrompa, el sistema debe tener una forma de marcar dicho bloque como no utilizable o de sustituirlo desde el punto de vista lógico por otro bloque adicional libre.

Puesto que disponer de un espacio de intercambio con un comportamiento eficiente resulta clave para obtener un buen rendimiento, los sistemas suelen puentear el sistema de archivos y utilizar un acceso al disco sin formato para las operaciones de E/S de paginación. Algunos sistemas dedican una partición del disco sin formato al espacio de intercambio, mientras que otros utilizan un archivo dentro del propio sistema de archivos. Finalmente, otros sistemas permiten al usuario o al administrador del sistema que tomen la decisión sobre qué tipo de espacio de intercambio utilizar, ya que proporcionan ambas opciones.

Debido a la cantidad de espacio de almacenamiento requerida en los sistemas de gran tamaño, normalmente se suelen hacer redundantes los discos utilizando algoritmos RAID. Estos algoritmos permiten utilizar más de un disco para una operación determinada y con ellos se puede garantizar una operación continua e incluso una recuperación automática en caso de que se produzca un fallo de disco. Los algoritmos RAID se clasifican en diferentes niveles, proporcionando cada uno de ellos una cierta combinación de mecanismos de fiabilidad y de altas tasas de transferencia.

El esquema de registro con escritura anticipada requiere disponer de un sistema de almacenamiento estable. Para implementar tal tipo de almacenamiento, tenemos que replicar la información necesaria en múltiples dispositivos no volátiles de almacenamiento (usualmente discos) con modos de fallo independientes. También necesitamos actualizar la información de manera controlada para garantizar que podamos recuperar los datos estables después de cualquier fallo que tenga lugar durante la transferencia de datos o la recuperación.

El almacenamiento terciario se construye a partir de unidades de disco o de cinta que utilizan soportes extraíbles. Hay disponibles muchas tecnologías distintas, incluyendo cintas magnéticas, discos magnéticos y magneto-ópticos extraíbles y discos ópticos.

Para los discos extraíbles, el sistema operativo proporciona generalmente los servicios completos de una interfaz de sistema de archivos, incluyendo la gestión del espacio y la planificación de la cola de solicitudes. En muchos sistemas operativos, el nombre de un archivo en un cartucho extraíble es una combinación de un nombre de unidad y de un nombre de archivo dentro de dicha unidad. Este convenio es más simple, pero potencialmente más confuso, que utilizar un nombre que identifique un cartucho específico.

Para las cintas, el sistema operativo suele proporcionar sólo una interfaz sin formato. Muchos sistemas operativos no tienen ningún soporte integrado para sistemas intercambiadores. El soporte para intercambiadores puede proporcionarse mediante un controlador de dispositivo o

mediante una aplicación privilegiada diseñada para la realización de copias de seguridad o para sistemas HSM.

Tres parámetros importantes del rendimiento son el ancho de banda, la latencia y la fiabilidad. En los discos y las cintas nos encontramos con una gama muy amplia de anchos de banda, pero la latencia de acceso aleatorio para una cinta es generalmente mucho mayor que para un disco. La operación de cambio de cartuchos en un intercambiador es también relativamente lenta. Puesto que un intercambiador tiene una baja relación entre unidades y cartuchos, leer una gran parte de los datos en un intercambiador puede requerir un tiempo considerable. Los soportes ópticos, que protegen la capa sensible con un recubrimiento transparente, son generalmente más robustos que los soportes magnéticos, en los cuales el material magnético está más expuesto a daños físicos.

Ejercicios

- 12.1 Ninguno de los algoritmos de planificación de disco, excepto FCFS, es realmente *equitativo* (puede producirse el fenómeno de muerte por inanición).
- Explique por qué es cierta esta afirmación.
 - Describa una forma de modificar algoritmos tales como SCAN para garantizar que sean equitativos.
 - Explique por qué es importante el objetivo de la distribución equitativa en un sistema de tiempo compartido.
 - Proporcione tres o más ejemplos de circunstancias en las que sea importante que el sistema operativo *no sea equitativo* a la hora de satisfacer las solicitudes de E/S.
- 12.2 Suponga que una unidad de disco tiene 5000 cilindros, numerados de 0 a 4999. La unidad está sirviendo actualmente una solicitud en el cilindro 143 y la solicitud anterior correspondió al cilindro 125. La cola de solicitudes pendientes, en orden FIFO, es

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Comenzando desde la posición actual del cabezal, ¿cuál será la distancia total (en cilindros) que el brazo del disco tendrá que moverse para satisfacer todas las solicitudes pendientes para cada uno de los siguientes algoritmos de planificación de disco?

- FCFS
 - SSTF
 - SCAN
 - LOOK
 - C-SCAN
 - C-LOOK
- 12.3 Los principios de física elemental establecen que cuando se somete un objeto a una aceleración constante a , la relación entre la distancia d y el tiempo t está dada por $d = \frac{1}{2}at^2$. Suponga que, durante una búsqueda, el disco del Ejercicio 12.2 acelera el brazo de disco a una velocidad constante durante la primera mitad de la búsqueda y luego frena el brazo de disco a la misma velocidad durante la segunda mitad de la búsqueda. Suponga que el disco puede realizar una búsqueda hasta un cilindro adyacente en 1 milisegundo y una búsqueda recorriendo los 5000 cilindros en 18 milisegundos.
- La distancia de una búsqueda es el número de cilindros que se mueve el cabezal. Explique por qué el tiempo de búsqueda es proporcional a la raíz cuadrada de la distancia de búsqueda.

- b. Escriba una ecuación para el tiempo de búsqueda en función de la distancia de búsqueda. Esta ecuación debe tener la forma $t = x + y\sqrt{L}$, donde t es el tiempo en milisegundos y L es la distancia de búsqueda en cilindros.
 - c. Calcule el tiempo total de búsqueda para cada uno de los algoritmos de planificación del Ejercicio 12.2. Determine qué algoritmo de planificación es el más rápido (el que tiene el tiempo total de búsqueda menor).
 - d. El *porcentaje de aceleración* es el tiempo ahorrado dividido por el tiempo original. ¿Cuál es el porcentaje de aceleración del algoritmo de planificación más rápido, con respecto a FCFS?
- 12.4 Suponga que el disco del Ejercicio 12.3 rota a 7200 RPM.
- a. ¿Cuál es la latencia rotacional media en esta unidad de disco?
 - b. ¿Qué distancia de búsqueda puede recorrerse en el tiempo calculado en el apartado a?
- 12.5 Escriba un programa Java para planificación de disco utilizando los algoritmos de planificación SCAN y C-SCAN.
- 12.6 Compare las prestaciones de los algoritmos de planificación C-SCAN y SCAN, suponiendo una distribución uniforme de las solicitudes. Considere el tiempo medio de respuesta, el tiempo comprendido entre la llegada de una solicitud y la terminación del servicio correspondiente a dicha solicitud, la varianza en el tiempo de respuesta y el ancho de banda efectivo. ¿Cómo dependen las prestaciones de los valores relativos del tiempo de búsqueda y de la latencia rotacional?
- 12.7 Las solicitudes no suelen estar uniformemente distribuidas. Por ejemplo, cabe esperar que un cilindro que contenga la tabla FAT de un sistema de archivos o los inodos tenga una mayor tasa de acceso que un cilindro que sólo contenga archivos. Suponga que sabemos que el 50 por ciento de las solicitudes están dirigidas a un número fijo pequeño de cilindros.
- a. ¿Sería especialmente adecuado para este caso algunos de los algoritmos de planificación expuestos en este capítulo? Explique su respuesta.
 - b. Proponga un algoritmo de planificación de disco que proporcione un rendimiento aun mejor, aprovechando este “punto caliente” del disco.
 - c. Los sistemas de archivos normalmente localizan los bloques de datos mediante una tabla de indirección, como por ejemplo una tabla FAT en DOS o los inodos en UNIX. Describa una o más maneras de aprovechar este mecanismo de indirección para mejorar el rendimiento del disco.
- 12.8 ¿Podría un esquema de organización RAID nivel 1 proporcionar un mejor rendimiento para las solicitudes de lectura que un esquema de organización RAID nivel 0 (con distribución en bandas no redundante de los datos)? En caso afirmativo, ¿cómo?
- 12.9 Considere un esquema de organización RAID nivel 5 compuesto por cinco discos, con la información de paridad para cada conjunto de cuatro bloques en cuatro discos almacenada en el quinto disco. ¿A cuántos bloques hay que acceder para llevar a cabo las siguientes operaciones?
- a. Una escritura de un bloque de datos.
 - b. Una escritura de siete bloques continuos de datos.
- 12.10 Compare la tasa de transferencia que puede conseguirse en un esquema de organización RAID nivel 5 con la que se puede obtener en un esquema de organización RAID nivel 1 para las siguientes operaciones:
- a. Operaciones de lectura en bloques aislados.

- b. Operaciones de lectura en múltiples bloques contiguos.
- 12.11 Compare la velocidad de las operaciones de escritura que puede conseguirse en un esquema de organización RAID nivel 5 con la que puede obtenerse en un esquema de organización RAID nivel 1.
- 12.12 Suponga que dispone de una configuración mixta, compuesta por discos con una organización RAID nivel 1 y discos con una organización RAID nivel 5. Suponga que el sistema tiene flexibilidad a la hora de decidir qué organización de discos utilizar para almacenar cada archivo concreto. ¿Qué archivos habría que almacenar en los discos RAID nivel 1 y cuáles en los discos RAID nivel 5 para optimizar el rendimiento?
- 12.13 ¿Existe alguna forma de implementar un almacenamiento verdaderamente estable? Razone su respuesta.
- 12.14 La fiabilidad de una unidad de disco duro suele describirse en términos de una magnitud denominada *tiempo medio entre fallos* (MTBF, mean time between failures).
- Si un sistema contiene 1000 unidades de disco, cada una de las cuales tiene un MTBF igual a 750.000 horas, ¿cuál de las siguientes afirmaciones describe de forma más aproximada la frecuencia con la que se producirá un fallo de una unidad en esa granja de discos: una por cada 1000 años, una por siglo, una por década, una por año, una por mes, una por semana, una por día, una por hora, una por minuto o una por segundo?
 - Las estadísticas de mortalidad indican que, como promedio, los habitantes de los Estados Unidos tienen una probabilidad de 1 entre 1000 de morir entre los 20 y los 21 años. Calcule el MTBF para las personas de 20 años. Pase este resultado de horas a años. ¿Qué información nos proporciona este valor de MTBF acerca de la esperanza de vida de una persona de 20 años?
 - El fabricante garantiza un MTBF de 1 millón de horas para un cierto modelo de unidad de disco. ¿Qué podemos concluir acerca del número de años durante los cuales está en garantía una de estas unidades?
- 12.15 Explique las ventajas y desventajas relativas de las técnicas de utilización de sectores adicionales y de deslizamiento de sectores.
- 12.16 Explique las razones por las que el sistema operativo podría requerir información precisa acerca del modo en que están almacenados los bloques en el disco. ¿Cómo podría el sistema operativo mejorar la velocidad del sistema de archivos utilizando esta información?
- 12.17 El sistema operativo trata generalmente los discos extraíbles como sistemas de archivos compartidos, pero las unidades de cinta las asigna a una única aplicación cada vez. Indique tres razones que puedan explicar este diferente tratamiento de los discos y las cintas. Describa las características adicionales que necesitaría el sistema operativo para soportar un acceso compartido al sistema de archivos en caso de emplear un intercambiador de cintas. ¿Necesitarían tener alguna propiedad especial las aplicaciones que compartieran el intercambiador de cintas o podrían utilizar los archivos como si éstos residieran en el disco? Razone su respuesta.
- 12.18 ¿Cuáles serán los efectos en términos de coste y de prestaciones si el almacenamiento en cinta tuviera la misma densidad de almacenamiento por unidad de área que el almacenamiento en disco? (La **densidad de almacenamiento** es el número de gigabits por centímetro cuadrado).
- 12.19 Podemos utilizar estimaciones simples para comparar el coste y las prestaciones de un sistema de almacenamiento con una capacidad en terabytes formado enteramente por discos con otro que incorpore almacenamiento terciario. Suponga que cada uno de los discos magnéticos tiene una capacidad de 10 GB, que su coste es de 1000 dólares, que permiten transferir 5 MB por segundo y que tienen una latencia media de acceso de 15 milisegundos.

Suponga que una biblioteca de cintas tiene un coste de 10 dólares por gigabyte, que permite transferir 10 MB por segundo y que tiene una latencia media de acceso de 20 segundos. Calcule el coste total, la tasa máxima total de transferencia de datos y el tiempo medio de espera para un sistema formado exclusivamente por discos. Si tiene que realizar algún tipo de suposición acerca de la carga de trabajo, describa esas suposiciones y justifíquelas. Ahora, suponga que sólo se utiliza frecuentemente el 5 por ciento de los datos, los cuales deberán residir en disco, pero que el otro 95 por ciento se archiva en la biblioteca de cintas. Suponga también que el sistema de discos gestiona el 95 por ciento de las solicitudes y que la biblioteca gestiona el otro 5 por ciento. ¿Cuál será el coste total, la tasa máxima total de transferencia de datos y el tiempo de espera medio para este sistema de almacenamiento jerárquico?

- 12.20** Imagine que ya se hubiera inventado una unidad de almacenamiento holográfica. Suponga que la unidad holográfica costara 10000 dólares y tuviera un tiempo medio de acceso de 40 milisegundos. Suponga también que utilizara un cartucho de 100 dólares del tamaño de un CD; dicho cartucho permitiría almacenar 40000 imágenes y cada una es una imagen cuadrada en blanco y negro con una resolución de 6000×6000 píxeles (cada píxel requiere un bit de almacenamiento). Suponga que la unidad puede leer o escribir una imagen en 1 milisegundo. Responda a las siguientes cuestiones:
- ¿Qué posibles aplicaciones tendría este dispositivo?
 - ¿Cómo afectaría este dispositivo al rendimiento de E/S de un sistema informático?
 - ¿Qué otros tipos de dispositivos de almacenamiento, si es que hay alguno, quedarían obsoletos como resultado de la invención de este dispositivo?
- 12.21** Suponga que un cartucho de disco óptico de 5,25 pulgadas y una sola cara tiene una densidad de almacenamiento de 1 GB por pulgada cuadrada. Suponga que una cinta magnética tiene una densidad de almacenamiento de 20 megabits por pulgada cuadrada y que tiene 1/2 pulgadas de anchura y 1800 pies de longitud. Realice una estimación de las capacidades de almacenamiento de estos dos tipos de cartuchos. Suponga que existiera una cinta óptica que tuviera el mismo tamaño físico que la cinta magnética pero la misma densidad de almacenamiento que el disco óptico. ¿Qué volumen de datos podría almacenar la cinta óptica? ¿Cuál podría ser el precio de mercado de la cinta óptica si la cinta magnética cuesta 25 dólares? (*Nota: 1 pulgada = 2,54 cm; 1 pie = 12 pulgadas.*)
- 12.22** Explique cómo podría mantener el sistema operativo una lista de espacio libre para un sistema de archivos residente en cinta. Suponga que la tecnología de cinta es del tipo de sólo adición y que utiliza marcas EOT y comandos locate, space y read_position como se describe en la Sección 12.9.2.1.

Notas bibliográficas

Un análisis de las matrices redundantes de discos independientes (RAID) se presenta en Patterson et al. [1988] y en la detallada obra de Chen et al. [1994]. Las arquitecturas de sistemas de discos para la informática de altas prestaciones se analiza en Katz et al. [1989]. Las extensiones de los sistemas RAID se presentan en Wilkes et al. [1996] y Yu et al. [2000]. Teorey y Pinkerton [1972] proporciona uno de los primeros análisis comparativos de los algoritmos de planificación de disco; en dicha obra se utilizan simulaciones en las que se modela un disco de forma que el tiempo de búsqueda es lineal con respecto al número de cilindros recorridos; para este disco, LOOK es una buena elección para longitudes de cola inferiores a 140, mientras que C-LOOK resulta adecuado para longitudes de cola por encima de 100. King [1990] describe formas de mejorar el tiempo de búsqueda moviendo el brazo de disco cuando el disco no está realizando ninguna otra tarea. Seitzer et al. [1990] y Jacobson y Wilkes [1991] describen algoritmos de planificación de discos que tienen en cuenta la latencia rotacional, además del tiempo de búsqueda. Los sistemas de optimización de la asignación de memoria se describen en Lumb et

al. [2000]. Worthington et al. [1994] trata el tema de las prestaciones de los discos y demuestra que los mecanismos de gestión de defectos tienen un impacto completamente despreciable sobre esas prestaciones. La cuestión de cómo ubicar los denominados datos calientes para mejorar los tiempos de búsqueda ha sido considerada por Ruemmler y Wilkes [1991] y Akyurek y Salem [1993]. Ruemmler y Wilkes [1994] describe un modelo de rendimiento muy preciso para una unidad de disco moderna. Worthington et al. [1995] indica cómo determinar las propiedades de bajo nivel de los discos, como la estructura de zonas, y el trabajo iniciado en esta obra se amplía en Schindler y Gregory [1999]. Las cuestiones relativas a la gestión de potencia en los discos se analizan en Douglis et al. [1994], Douglis et al. [1995], Greenawalt [1994] y Golding et al. [1995].

El tamaño de las operaciones de E/S y la aleatoriedad de la carga de trabajo tienen una considerable influencia sobre las prestaciones de los discos. Ousterhout et al. [1985] y Ruemmler y Wilkes [1993] proporcionan numerosos datos interesantes sobre las características de las cargas de trabajo, incluyendo que la mayoría de los archivos son de pequeño tamaño, que la mayoría de los archivos de reciente creación se borran poco tiempo después, que la mayoría de los archivos que se abren para lectura se leen de manera secuencial de principio a fin y que la mayoría de las búsquedas son cortas. McKusick et al. [1984] describe el sistema FFS (Fast File System) de Berkeley, que utiliza muchas técnicas sofisticadas para obtener un buen rendimiento con diversos tipos de carga de trabajo. McVoy y Kleiman [1991] exponen mejoras adicionales del sistema FFS básico. Quinlan [1991] describe cómo implementar un sistema de archivos en un almacenamiento WORM con una caché en disco magnético; Richards [1990] analiza la utilización de un sistema de archivos en un almacenamiento secundario. Maher et al. [1994] proporciona una panorámica de la integración de sistemas de archivos distribuidos y un almacenamiento secundario.

El concepto de jerarquía de almacenamiento ha sido estudiado durante más de treinta años. Por ejemplo un artículo de 1970 de Mattson et al. [1970] describe una técnica matemática para predecir el rendimiento de una jerarquía de almacenamiento. Alt [1993] describe la utilización de soportes de almacenamiento extraíbles en un sistema operativo comercial y Miller y Katz [1993] describe las características del acceso al almacenamiento terciario en un entorno de supercomputación. Benjamin [1990] proporciona una panorámica de los requerimientos de almacenamiento masivos del proyecto EOSDIS en la NASA. La gestión y utilización de discos conectados por red y discos programables se analizan en Gibson et al. [1997b], Gibson et al. [1997a], Riedel et al. [1998] y Lee y Thekkath [1996].

La tecnología de almacenamiento holográfica es la materia de un artículo de Psaltis y Mok [1995]; una buena recopilación de artículos sobre este tema publicados desde 1963 es la que ha realizado Sincerbox [1994]. Asthana y Finkelstein [1995] describe varias tecnologías de almacenamiento emergentes, incluyendo el almacenamiento holográfico, la cinta óptica y la tecnología de trampas de electrones. Toigo [2000] proporciona una descripción detallada de la tecnología de disco moderna y de varias tecnologías de almacenamiento potenciales que podrían usarse en el futuro.

Sistemas de E/S

Las dos tareas principales de una computadora son la E/S y el procesamiento. En muchos casos, la tarea principal es la E/S y el procesamiento es meramente incidental. Por ejemplo, cuando leemos una página web o editamos un archivo, nuestro interés inmediato es leer o introducir cierta información, no calcular una respuesta.

El papel del sistema operativo en la E/S de la computadora consiste en gestionar y controlar las operaciones y dispositivos de E/S. Aunque en otros capítulos aparecen temas relacionados, vamos a juntar aquí las diversas piezas para dibujar una imagen completa de la E/S. Primero, describiremos los fundamentos del hardware de E/S, porque la naturaleza de interfaz hardware impone una serie de requisitos a la funcionalidad interna del sistema operativo. A continuación, veremos los servicios de E/S proporcionados por el sistema operativo y el modo de integrar dichos servicios en la interfaz de E/S de las aplicaciones. Después, explicaremos cómo el sistema operativo establece el puente entre la interfaz hardware y la interfaz de las aplicaciones. Analizaremos también el mecanismo STREAMS de UNIX System V, que permite a las aplicaciones ensamblar dinámicamente *pipelines* de código de controladores. Finalmente, veremos los aspectos de rendimiento relativos a la E/S y los principios del diseño de sistemas operativos que mejoran dicho rendimiento.

OBJETIVOS DEL CAPÍTULO

- Analizar la estructura del subsistema de E/S de un sistema operativo.
- Explorar los principios en que se basa el hardware de E/S y los aspectos relativos a su complejidad.
- Proporcionar detalles sobre el rendimiento del hardware y el software de E/S.

13.1 Introducción

El control de los dispositivos conectados a la computadora es una de las principales preocupaciones de los diseñadores de sistemas operativos. Debido a que existe una variedad tan amplia de dispositivos de E/S tanto en lo respecta a sus funciones como en cuanto a su velocidad de operación (considere, por ejemplo, un ratón, un disco duro y una *jukebox* para CD-ROM), se necesitan diversos métodos para controlar esos dispositivos. Dichos métodos forman el *subsistema de E/S* del *kernel*, que aísla al resto del *kernel* de la complejidad asociada con la gestión de los dispositivos de E/S.

En el campo de la tecnología de dispositivos de E/S se experimentan dos tendencias que están en conflicto mutuo. De un lado, podemos ver una creciente estandarización de las interfaces software y hardware; esta tendencia nos ayuda a incorporar generaciones mejoradas de dispositivos

dentro de las computadoras de sistemas operativos existentes. Por otro lado, podemos ver una variedad cada vez más amplia de dispositivos de E/S; algunos nuevos dispositivos son tan distintos de los dispositivos anteriores que constituye todo un desafío incorporarlos en las computadoras y los sistemas operativos. Este desafío se afronta mediante una combinación de técnicas hardware y software. Los elementos básicos del hardware de E/S, como los puertos, buses y controladores de dispositivo, permiten integrar una amplia variedad de dispositivos de E/S. Para encapsular los detalles y las peculiaridades de diferentes tipos de dispositivos, el *kernel* de un sistema operativo se estructura de modo que haga uso de módulos específicos controladores de dispositivos. Los **controladores de dispositivo** presentan al subsistema de E/S una interfaz uniforme de acceso a los dispositivos, de forma parecida a como las llamadas al sistema proporcionan una interfaz estándar entre la aplicación y el sistema operativo.

13.2 Hardware de E/S

Las computadoras interaccionan con una amplia variedad de tipos de dispositivos. La mayoría de esos dispositivos pueden clasificarse en una serie de categorías generales: dispositivos de almacenamiento (discos, cintas), dispositivos de transmisión (tarjetas de red, módems) y dispositivos de interfaz humana (pantalla, teclado, ratón). Otros dispositivos son más especializados, como por ejemplo el volante de un avión de caza militar o de una lanzadera espacial. En este tipo de aeronaves, las personas proporcionan los datos de entrada a la computadora de vuelo mediante una palanca y una serie de pedales controlados con los pies, y la computadora envía una serie de comandos de salida que hacen que los motores muevan los alerones, los timones o las válvulas de combustible. Sin embargo, a pesar de la increíble variedad de dispositivos de E/S existentes, sólo son necesarios unos cuantos conceptos para comprender cómo se conectan los dispositivos y cómo puede el software controlar el hardware asociado.

Los dispositivos se comunican con los sistemas informáticos enviando señales a través de un cable o incluso a través del aire. Cada dispositivo se comunica con la máquina a través de un punto de conexión (o **puerto**), como por ejemplo un puerto serie. Si los dispositivos utilizan un conjunto común de hilos, dicha conexión se denomina **bus**. Un **bus** es un conjunto de hilos, junto con un protocolo rígidamente definido que especifica el conjunto de mensajes que pueden enviarse a través de esos hilos. En términos electrónicos, los mensajes se transmiten mediante patrones de tensiones eléctricas aplicadas a los hilos, con unos requisitos de temporización bien definidos. Cuando el dispositivo A tiene un cable que se inserta en el dispositivo B, y el dispositivo B tiene un cable que se inserta en el dispositivo C, y el dispositivo C se inserta en un puerto de la computadora, este tipo de dispositivo se denomina **conexión en cascada**. Las conexiones en cascada suelen funcionar como un bus.

Los buses se utilizan en multitud de ocasiones dentro de la arquitectura de los sistemas informáticos. En la Figura 13.1 se muestra una estructura típica de un bus de PC. Esta figura muestra un **bus PCI** (el bus común de los sistemas PC) que conecta el subsistema procesador-memoria con los dispositivos de alta velocidad y un **bus de expansión** que conecta los dispositivos relativamente lentos, como el teclado y los puertos serie y paralelo. En la parte superior derecha de la figura, podemos ver cuatro discos conectados a un bus SCSI que se inserta en una controladora SCSI.

Una **controladora** es una colección de componentes electrónicos que permite controlar un puerto, un bus o un dispositivo. Un ejemplo simple de controladora de dispositivo sería la controladora de un puerto serie: se trata de un único chip (o una parte de un chip) dentro de la controladora que controla las señales que se transmiten a través de los hilos de un puerto serie. Por contraste, una controladora de bus SCSI no es tan simple; como el protocolo SCSI es muy complejo, la controladora de bus SCSI se suele implementar mediante una tarjeta de circuitos separada o (**adaptadora host**) que se inserta en la computadora. Normalmente, dicha tarjeta separada contiene un procesador, microcódigo y algo de memoria privada para poder procesar los mensajes del protocolo SCSI. Algunos dispositivos tienen sus propias controladoras integradas. Si examinamos una unidad de disco, podemos ver una tarjeta de circuito en uno de los lados; dicha tarjeta es la controladora de disco e implementa el lado correspondiente al disco para el protocolo utilizado en algún tipo de conexión, como por ejemplo SCSI o ATA. Tiene el microcódigo necesario y un

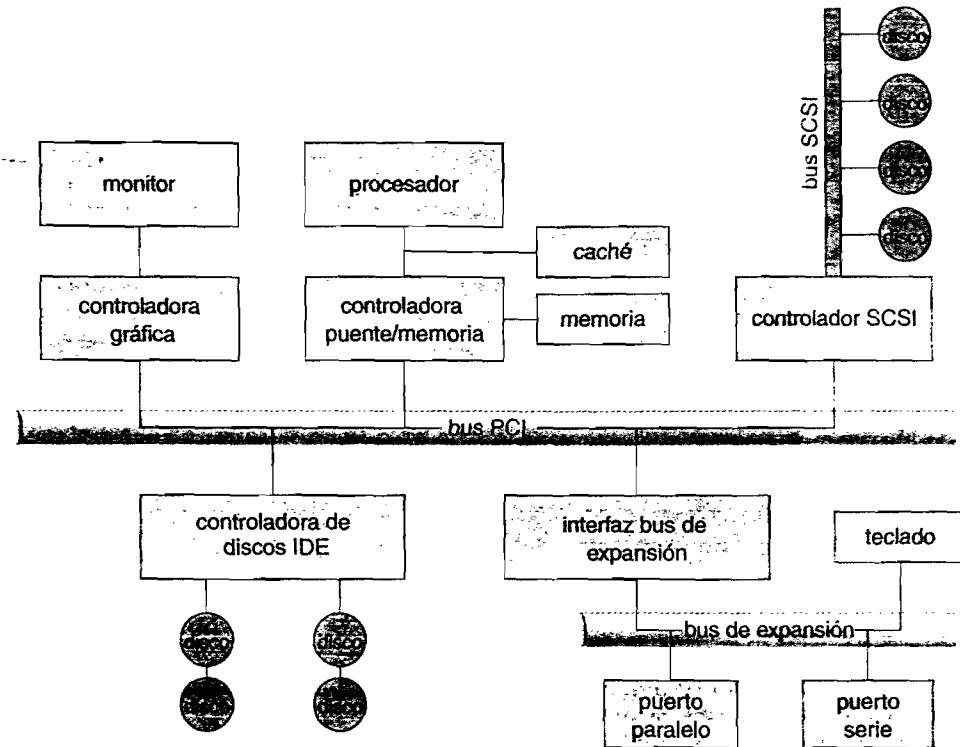


Figura 13.1 Una estructura de buses típica en un PC.

procesador para realizar diversas tareas, como el mapeo de sectores erróneos, la pre-extracción, el almacenamiento en búfer y el almacenamiento en caché.

¿Cómo puede proporcionar comandos y datos el procesador a una controladora para llevar a cabo una transferencia de E/S? La respuesta simple es que la controladora dispone de uno o más registros para los datos y las señales de control. El procesador se comunica con la controladora leyendo y escribiendo patrones de bits en dichos registros. Una forma de llevar a cabo esta comunicación es utilizando instrucciones de E/S especiales que especifican la transferencia de un byte o de una palabra a una dirección de puerto de E/S. La instrucción de E/S configura las líneas de bus para seleccionar el dispositivo apropiado y para leer o escribir bits en un registro del dispositivo. Alternativamente, la controladora de dispositivo puede soportar una E/S mapeada en memoria. En este caso, los registros de control del dispositivo están mapeados en el espacio de direcciones del procesador. La CPU ejecuta las solicitudes utilizando instrucciones estándar de transferencia de datos para leer y escribir los registros de control del dispositivo.

Algunos sistemas utilizan ambas técnicas. Por ejemplo, un PC utiliza instrucciones de E/S para controlar algunos dispositivos y E/S mapeada en memoria para controlar otros. La Figura 13.2 muestra las direcciones usuales de puertos de E/S en un PC. La controladora gráfica tiene puertos de E/S para las operaciones básicas de control, pero también dispone de una gran región mapeada en memoria para almacenar el contenido de la pantalla. El proceso envía la salida a la pantalla escribiendo datos en esa región mapeada en memoria. La controladora genera la imagen de pantalla basándose en el contenido de esa memoria. Esta técnica resulta muy simple de utilizar; además, resulta más rápido escribir millones de bytes en la memoria gráfica que ejecutar millones de instrucciones de E/S. Pero la facilidad de escritura en una controladora de E/S mapeada en memoria se ve compensada por una desventaja: puesto que uno de los tipos más comunes de fallo de software es la escritura mediante un puntero incorrecto en una región de memoria distinta de la que se pretendía, los registros de dispositivos mapeados en memoria son vulnerables a la modificación accidental por parte de los programas. Por supuesto, la memoria protegida nos ayuda a reducir este riesgo.

Un puerto de E/S está compuesto típicamente de cuatro registros, denominados (1) registro de estado, (2) registro de control, (3) registro de entrada de datos y (4) registro de salida de datos.

Rango de direcciones de E/S (hexadecimal)	Dispositivo
0x00000000 - 0x0000000F	Controlador de disco duro
0x00000010 - 0x0000001F	Controlador de memoria RAM
0x00000020 - 0x0000002F	Controlador de video
0x00000030 - 0x0000003F	Controlador de audio
0x00000040 - 0x0000004F	Controlador de red
0x00000050 - 0x0000005F	Controlador de impresora
0x00000060 - 0x0000006F	Controlador de escáner
0x00000070 - 0x0000007F	Controlador de monitor
0x00000080 - 0x0000008F	Controlador de teclado
0x00000090 - 0x0000009F	Controlador de ratón
0x000000A0 - 0x000000AF	Controlador de tarjeta gráfica
0x000000B0 - 0x000000BF	Controlador de altavoces
0x000000C0 - 0x000000CF	Controlador de unidad óptica
0x000000D0 - 0x000000DF	Controlador de tarjeta de red
0x000000E0 - 0x000000EF	Controlador de tarjeta de sonido
0x000000F0 - 0x000000FF	Controlador de tarjeta de video

Figura 13.2 Ubicaciones de los puertos de E/S de los dispositivos en un PC (parcial).

- El *host* lee el registro de **entrada de datos** para obtener la entrada.
- El *host* escribe en el registro de **salida de datos** para enviar la salida.
- El **registro de estado** contiene bits que el *host* puede leer. Estos bits indican estados, como por ejemplo si se ha completado la ejecución del comando actual, si hay disponible un byte para ser leído en el registro de entrada de datos o si se ha producido una condición de error en el dispositivo.
- El **registro de control** puede ser escrito por el *host* para iniciar un comando o para cambiar el modo de un dispositivo. Por ejemplo, un cierto bit del registro de control de un puerto serie permite seleccionar entre comunicación full-dúplex y semi-dúplex, otro bit activa la comprobación de paridad, un tercer bit configura la longitud de palabra para que sea de 7 u 8 bits y otros bits seleccionan algunas de las velocidades soportadas por el puerto serie.

Los registros de datos tienen normalmente entre 1 y 4 bytes de tamaño. Algunas controladoras tienen chips FIFO que permiten almacenar varios bytes de datos de entrada y de salida, para expandir la capacidad de la controladora más allá del tamaño que el registro de datos tenga. Un chip FIFO puede almacenar una pequeña ráfaga de datos hasta que el dispositivo o *host* sea capaz de recibir dichos datos.

13.2.1 Sondeo

El protocolo completo de interacción entre el *host* y una controladora puede ser intrincado, pero la noción básica de *negociación* resulta muy simple. Vamos a explicar el concepto de negociación mediante un ejemplo. Supongamos que se utilizan 2 bits para coordinar la relación productor-consumidor entre la controladora y el *host*. La controladora indica su estado mediante el bit de *ocupado* en el registro de *estado*. La controladora activa el bit de *ocupado* cuando está ocupada trabajando y borra el bit de *ocupado* cuando está lista para aceptar el siguiente comando. El *host* indica sus deseos mediante el bit de *comando preparado* en el registro de *comando*: el *host* activa el bit de *comando preparado* cuando hay disponible un comando para que la controladora lo ejecute. En nuestro ejemplo, el *host* escribe la salida a través de un puerto, coordinándose con la controladora mediante un procedimiento de negociación de la forma siguiente:

1. El *host* lee repetidamente el bit de *ocupado* hasta que dicho bit pasa a cero.
2. El *host* activa el bit de *escritura* en el registro de *comando* y escribe un byte en el registro de *datos de salida*.

3. El *host* activa el bit de *comando preparado*.
4. Cuando la controladora observa que está activado el bit de *comando preparado*, activa el bit de *ocupada*.
5. La controladora lee el registro de comandos y ve el comando de *escritura*. A continuación, lee el registro de *salida de datos* para obtener el byte y lleva a cabo la E/S hacia el dispositivo.
6. La controladora borra el bit de *comando preparado*, borra el bit de *error* en el registro de *estado* para indicar que la E/S de dispositivo ha tenido éxito y borra el bit de *ocupado* para indicar que ha finalizado.

Este bucle se repite para cada byte.

En el paso 1, el *host* se encuentra en un estado de **espera activa o sondeo**: ejecuta un bucle, leyendo una y otra vez el registro de *estado* hasta que el bit de *ocupada* se desactiva. Si la controladora y el dispositivo son de alta velocidad, este método resulta razonable, pero si la espera puede ser larga, quizás convendría más que el *host* comutara a otra tarea. Pero entonces, ¿cómo puede saber el *host* cuando ha pasado a estar inactiva la controladora? En algunos dispositivos, el *host* debe dar servicio al dispositivo rápidamente o se producirá una pérdida de datos. Por ejemplo, cuando llega un flujo de datos a través de un puerto serie o desde el teclado, el pequeño búfer de la controladora se desbordará si el *host* espera demasiado antes de leer los bytes, con lo que podría producirse una pérdida de datos.

En muchas arquitecturas informáticas, para sondear un dispositivo basta con tres ciclos de instrucciones de CPU: *leer* un registro del dispositivo, efectuar una operación de *and lógica* para extraer un bit de estado y *saltar* si ese bit es distinto de cero. Obviamente, la operación básica de sondeo resulta muy eficiente. Pero el sondeo pasa a ser ineficiente cuando se le ejecuta de manera repetida para encontrar únicamente que sólo en raras ocasiones está listo el dispositivo para ser servido, mientras otras tareas útiles de procesamiento que podría haber ejecutado la CPU permanecen sin hacer. En tales casos, puede ser más eficiente que la controladora hardware notifique a la CPU cuándo ha pasado el dispositivo a estar listo para el servicio, en lugar de forzar a la CPU a sondear repetidamente el dispositivo para ver si se ha terminado la operación de E/S. El mecanismo hardware que permite a un dispositivo notificar los eventos a la CPU se denomina **interrupción**.

13.2.2 Interrupciones

El mecanismo básico de interrupción funciona de la forma siguiente. El hardware de la CPU tiene un hilo denominado **línea de solicitud de interrupción** que la CPU comprueba después de ejecutar cada instrucción. Cuando la CPU detecta que una controladora ha activado una señal a través de la línea de solicitud de interrupción, la CPU guarda el estado actual y salta a la **rutina de tratamiento de interrupciones** situada en una dirección fija de la memoria. La rutina de tratamiento de interrupciones determina la causa de la interrupción, lleva a cabo el procesamiento necesario, realiza una restauración del estado y ejecuta una instrucción `return from interrupt` para volver a situar la CPU en el estado de ejecución anterior a que se produjera la interrupción. Decimos que la controladora del dispositivo *genera* una interrupción enviando una determinada señal a través de la línea de solicitud de interrupción, la CPU *atraja* la interrupción y la *despacha* a la rutina de tratamiento de interrupciones y la rutina *borra* la interrupción dando servicio al dispositivo. La Figura 13.3 resume el ciclo de E/S dirigido por interrupción.

El mecanismo básico de interrupción permite a la CPU responder a un suceso asíncrono, como es el caso en que una controladora de dispositivo pasa a estar lista para ser servida. Sin embargo, en los sistemas operativos modernos necesitamos funciones más sofisticadas de tratamiento de interrupciones:

1. Necesitamos poder diferir el tratamiento de una interrupción durante las secciones de procesamiento crítico.
2. Necesitamos una forma eficiente de despachar la interrupción a la rutina de tratamiento de interrupciones apropiada para un cierto dispositivo sin necesidad de sondear primero todos los dispositivos para ver cuál ha generado la interrupción.

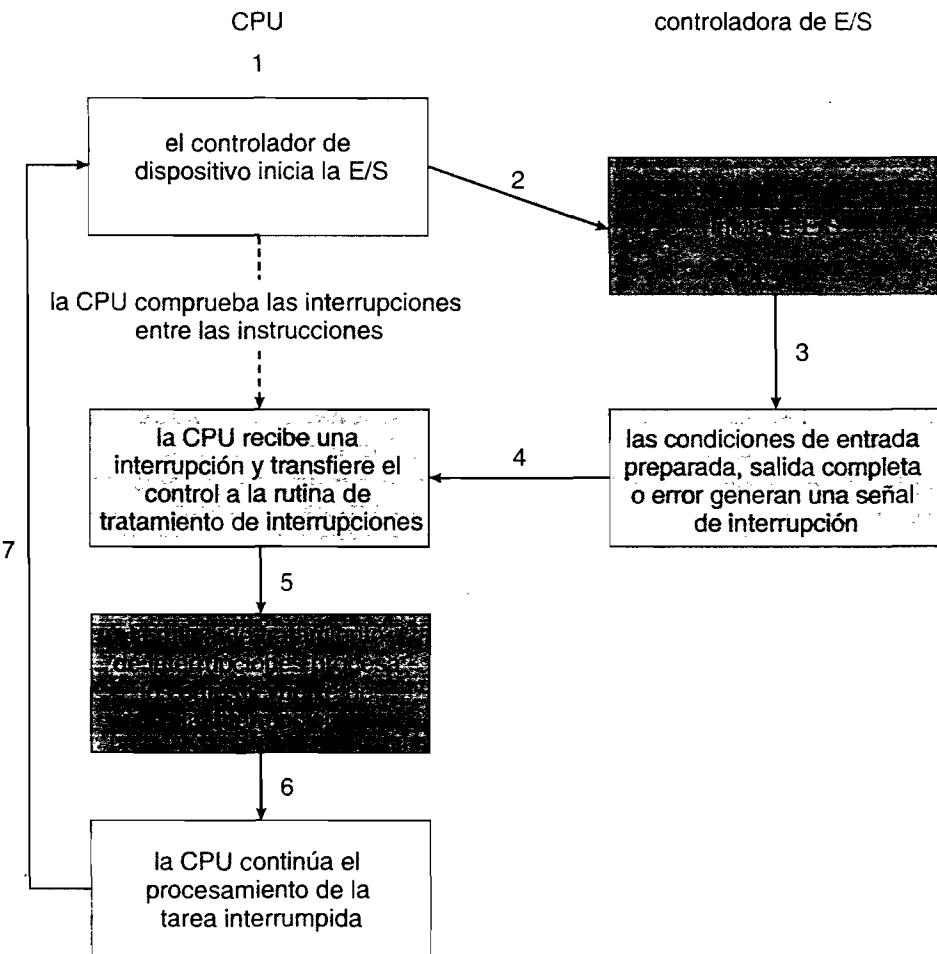


Figura 13.3 Ciclo de E/S dirigido por interrupción.

3. Necesitamos interrupciones multinivel, de modo que el sistema operativo pueda distinguir entre interrupciones de alta y baja prioridad, y pueda responder con el grado de urgencia apropiado.

En el hardware informático moderno, estas tres funcionalidades las proporciona la CPU y el hardware de la **controladora de interrupciones**.

La mayoría de los procesadores tienen dos líneas de solicitud de interrupción. Una de ellas es la **interrupción no enmascarable**, que está reservada para sucesos tales como los errores de memoria no recuperables. La segunda línea de interrupción es **enmascarable**: puede ser desactivada por la CPU antes de la ejecución de secuencias de instrucciones críticas que no deban ser interrumpidas. La interrupción enmascarable es la que las controladoras de dispositivo utilizan para solicitar servicio.

El mecanismo de interrupción acepta una **dirección**, que es el número que selecciona una rutina específica de tratamiento de interrupciones de entre un pequeño conjunto de rutinas disponibles. En la mayoría de las arquitecturas, esta dirección es un desplazamiento dentro de una tabla denominada **vector de interrupciones**. Este vector contiene las direcciones de memoria de las rutinas especializadas de tratamiento de interrupciones. El propósito de un mecanismo de interrupciones vectorizado es reducir la necesidad de que una única rutina de tratamiento de interrupciones tenga que analizar todas las posibles fuentes de interrupción para determinar cuál es la que necesita servicio. En la práctica, sin embargo, las computadoras tienen más dispositivos (y por tanto más rutinas y tratamientos de interrupciones) que direcciones existentes en el vector de interrupción. Una forma común de resolver este problema consiste en utilizar la técnica del **encadenamiento de interrupciones**, en la que cada elemento del vector de interrupciones apunta

a la cabeza de una lista de rutinas de tratamiento de interrupción. Cuando se genera una interrupción, se llama una a una a las rutinas de la lista correspondiente, hasta que se encuentre una rutina que pueda dar servicio a la solicitud. Esta estructura representa un compromiso entre el gasto asociado a disponer de una tabla de interrupciones de gran tamaño y la poca eficiencia que se experimenta a la hora de despachar las interrupciones a una única rutina de tratamiento.

La Figura 13.4 ilustra el diseño del vector de interrupción para el procesador Intel Pentium. Los sucesos numerados de 0 a 31, que son no enmascarables, se utilizan para señalizar diversas condiciones de error. Los sucesos comprendidos entre 32 y 255, que son enmascarables, se utilizan para cosas tales como las interrupciones generadas por los dispositivos.

El mecanismo de interrupciones también implementa un sistema de niveles de prioridad de interrupción. Este mecanismo permite a la CPU diferir el tratamiento de las interrupciones de baja prioridad sin enmascarar todas las interrupciones, y hace posible que una interrupción de alta prioridad desaloje a otra interrupción de prioridad más baja.

Los sistemas operativos modernos interactúan con el mecanismo de interrupciones de varias formas distintas. Durante el arranque, el sistema operativo comprueba los buses hardware para determinar qué dispositivos existen e instalar las rutinas de tratamiento de interrupción correspondientes dentro del vector de interrupciones. Durante la E/S, las diversas controladoras de dispositivo generan interrupciones cuando están listas para ser servidas. Estas interrupciones significan que se ha completado una operación de salida, o que hay disponibles datos de entrada o que se ha detectado un fallo. El mecanismo de interrupciones se utiliza también para gestionar una amplia variedad de excepciones, como la división por cero, el acceso a direcciones de memoria protegidas o no existentes, o el intento de ejecutar una instrucción privilegiada en modo usuario. Los sucesos que generan interrupciones tienen una propiedad en común: son sucesos que inducen a la CPU a ejecutar una rutina urgente y autocontenido.

Los sistemas operativos tienen otros usos adecuados para un mecanismo hardware y software eficiente que guarde una pequeña cantidad de información de estado del procesador y luego invoque una rutina privilegiada dentro del *kernel*. Por ejemplo, muchos sistemas operativos utilizan el

Número de vector	Descripción
0	División por cero
1	Excepción de depuración
2	Interrupción nula
3	Punto de parada
4	Desbordamiento detectado
5	Excepción de rango
6	Código de operación no válido
7	Dispositivo no disponible
8	Reservado
9	Desbordamiento de segmento en coprocesador (reservada)
10	Segmento de estado de tarea no válido
11	Segmento no presente
12	Fallo de pila
13	Excepción general
14	Reservado para Intel (no utilizado)
15	Reservado para Intel (no utilizado)
16	Comprobación de alineación
17	Comprobación de máscara
18-31	Reservadas para Intel (no utilizadas)
32-255	Interrupciones enmascarables

Figura 13.4 Tabla de vectores de sucesos en el procesador Intel Pentium.

mecanismo de interrupciones para la paginación de la memoria virtual. Un fallo de página es una excepción que genera una interrupción. La interrupción suspende el proceso actual y salta a la rutina de procesamiento de fallo de página dentro del *kernel*. Esta rutina de tratamiento guarda el estado del proceso, mueve el proceso a la cola de espera, realiza la gestión de la caché de páginas, planifica una operación de E/S para extraer la página, planifica otro proceso para reanudar la ejecución y luego vuelve de la interrupción.

Otro ejemplo es el de la implementación de las llamadas al sistema. Usualmente, los programas utilizan llamadas a biblioteca para realizar llamadas al sistema. Estas rutinas de biblioteca comprueban los argumentos proporcionados por la aplicación, construyen una estructura de datos para entregar esos argumentos al *kernel* y luego ejecutan una instrucción especial denominada **interrupción software**. Esta instrucción tiene un operando que identifica el servicio del *kernel* deseado. Cuando un proceso ejecuta la interrupción software, el hardware de interrupción guarda el estado del código de usuario, comuta a modo supervisor y realiza el despacho a la rutina del *kernel* que implementa el servicio solicitado. La interrupción software tiene una prioridad de interrupción relativamente baja, comparada con la que se asigna a las interrupciones de los dispositivos: ejecutar una llamada al sistema por cuenta de una aplicación es menos urgente que dar servicio a una controladora de dispositivo antes de que su cola FIFO se desborde, provocando la pérdida de datos.

Las interrupciones también pueden usarse para gestionar el flujo de control dentro del *kernel*. Por ejemplo, considere el procesamiento requerido para completar una lectura de disco. Uno de los pasos es copiar los datos desde el espacio del *kernel* al búfer de usuario; esta operación de copia lleva mucho tiempo pero no es urgente, así que no debe bloquear el tratamiento de otras interrupciones de alta prioridad. Otro paso consiste en iniciar la siguiente E/S pendiente para esa unidad de disco. Este paso tiene una mayor prioridad: si queremos utilizar los discos eficientemente, necesitamos comenzar la siguiente E/S en cuanto se complete la anterior. En consecuencia, hay una *pareja* de rutinas de tratamiento de interrupción que implementan el código del *kernel* necesario para completar una lectura de disco. La rutina de tratamiento de alta prioridad registra el estado de E/S, borra la interrupción del dispositivo, comienza la siguiente E/S pendiente y genera una interrupción de baja prioridad para completar la tarea. Posteriormente, cuando la CPU no esté ocupada con otro trabajo de alta prioridad, se despachará la interrupción de prioridad más baja. La rutina de tratamiento correspondiente completará la E/S de nivel de usuario copiando los datos desde los búferes del *kernel* al espacio de la aplicación y luego invocando al planificador para colocar la aplicación en la cola de procesos preparados.

Las arquitecturas de *kernel* con estructura de hebras están bien adaptadas para implementar múltiples prioridades de interrupción y para imponer la precedencia del tratamiento de interrupciones sobre el procesamiento no urgente correspondiente a las rutinas del *kernel* y de las aplicaciones. Vamos a ilustrar este aspecto mediante el *kernel* de Solaris. En Solaris, las rutinas de tratamiento de interrupciones se ejecutan como hebras del *kernel*, reservándose un rango de prioridades altas para estas hebras. Estas prioridades proporcionan a las rutinas de tratamiento de interrupciones precedencia frente al código de aplicación y frente a las tareas administrativas del *kernel* e implementan las relaciones de prioridad entre las propias rutinas de tratamiento de interrupciones. El mecanismo de prioridades hace que el planificador de hebras de Solaris desaloje a las rutinas de tratamiento de interrupciones de baja prioridad en favor de las de prioridad más alta, y la implementación del mecanismo de hebras permite al hardware multiprocesador ejecutar concurrentemente varias rutinas de tratamiento de interrupciones. La arquitectura de interrupciones de UNIX y de Windows XP se describe en el Apéndice A y en el Capítulo 22, respectivamente.

En resumen, las interrupciones se utilizan extensivamente en los sistemas operativos modernos para gestionar sucesos asíncronos y para saltar a rutinas en modo supervisor dentro del *kernel*. Para que las tareas más urgentes se lleven a cabo primero, las computadoras modernas utilizan un sistema de prioridades de interrupciones. Las controladoras de dispositivo, los fallos hardware y las llamadas al sistema generan interrupciones para provocar la ejecución de rutinas del *kernel*. Dado que las interrupciones se utilizan de forma tan constante para el procesamiento más crítico en términos de tiempo, se necesita que el tratamiento de las interrupciones sea eficiente para obtener un buen rendimiento del sistema.

13.2.3 Acceso directo a memoria

Para un dispositivo que realice transferencias de gran tamaño, como por ejemplo una unidad de disco, parece bastante poco apropiado utilizar un caro procesador de propósito general para comprobar dicho estado y para escribir datos en un registro de una controladora de byte en byte, lo cual es un proceso que se denomina **E/S programada** (PIO, programmed I/O). La mayoría de las computadoras evitan sobrecargar a la CPU principal con las tareas PIO, descargando parte de este trabajo en un procesador de propósito especial denominado controladora de **acceso directo a memoria** (DMA, *direct-memory-access*). Para iniciar una transferencia de DMA, el *host* describe un bloque de comando DMA en la memoria. Este bloque contiene un puntero al origen de una transferencia, un puntero al destino de la transferencia y un contador del número de bytes que hay que transferir. La CPU escribe la dirección de este bloque de comandos en la controladora DMA y luego continúa realizando otras tareas. La controladora de DMA se encarga entonces de operar el bus de memoria directamente, colocando direcciones en el bus para realizar las transferencias sin ayuda de la CPU principal. En las computadoras de tipo PC, uno de los componentes estándar es una controladora de DMA simple, y las tarjetas de E/S con **control maestro del bus** para PC suelen contener su propio hardware de DMA de alta velocidad.

El procedimiento de negociación entre la controladora de DMA y la controladora de dispositivo se realiza mediante un par de hilos denominados DMA-request y DMA-acknowledge. La controladora de dispositivo coloca una señal en el hilo DMA-request cada vez que hay disponible para transferencia una palabra de datos. Esta señal hace que la controladora de DMA tome el control del bus de memoria, coloque la dirección deseada en los hilos de dirección de memoria y coloque una señal en el hilo DMA-acknowledge. Cuando la controladora de dispositivo recibe la señal DMA-acknowledge, transfiere la palabra de datos a memoria y borra la señal DMA-request.

Una vez finalizada la transferencia completa, la controladora de DMA interrumpe a la CPU. Este proceso se ilustra en la Figura 13.5. Cuando la controladora de DMA toma el control del bus de memoria, se impide momentáneamente a la CPU acceder a la memoria principal, aunque podrá seguir accediendo a los elementos de datos almacenados en sus cachés principal y secundaria.

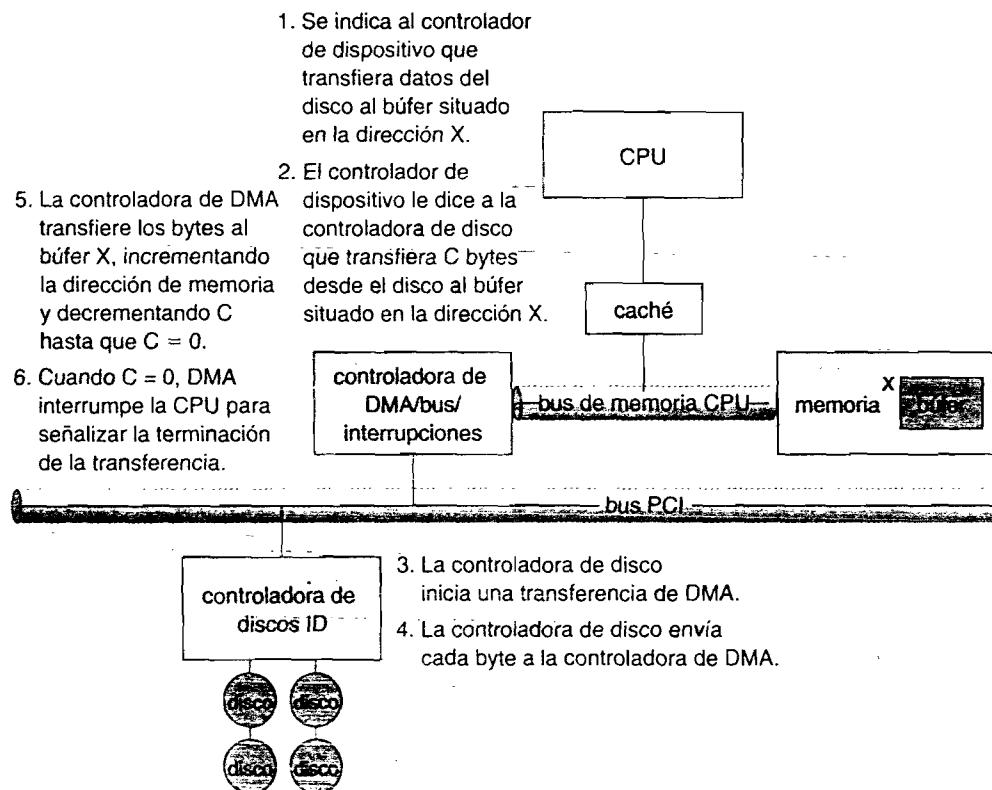


Figura 13.5 Pasos de una transferencia de DMA.

Aunque este proceso de **robo de ciclos** puede ralentizar los cálculos realizados por la CPU, descargar el trabajo de transferencia de datos en una controladora de DMA suele mejorar el rendimiento global del sistema. Algunas arquitecturas de computadora utilizan direcciones de memoria física para las operaciones de DMA, mientras que otras realizan un **acceso directo a memoria virtual** (DVMA, *direct virtual memory access*), utilizando direcciones virtuales que serán traducidas a direcciones físicas. El mecanismo de DVMA puede realizar una transferencia entre dos dispositivos mapeados en memoria sin la intervención de la CPU y sin usar la memoria principal.

En un *kernel* de modo protegido, el sistema operativo impide generalmente que los procesos ejecuten directamente comandos de los dispositivos. Esto protege a los datos frente a las violaciones de los mecanismos de control de acceso y también protege al sistema frente al uso erróneo de las controladoras de dispositivo que pudiera provocar un fallo catastrófico del sistema. En lugar de usar esos mecanismos, el sistema operativo exporta una serie de funciones que un proceso con los suficientes privilegios puede utilizar para acceder a las operaciones de bajo nivel sobre el hardware subyacente. En un *kernel* sin protección de memoria, los procesos pueden acceder directamente a las controladoras de dispositivo. Este acceso directo puede usarse para obtener un mayor rendimiento, ya que puede evitar determinadas comunicaciones dentro del *kernel*, determinados cambios de contexto y el paso por diversas capas del software del *kernel*. Desafortunadamente, este mecanismo afecta a la seguridad y a la estabilidad del sistema, por lo que la tendencia en los sistemas operativos de propósito general es proteger a la memoria y a los dispositivos de modo que el sistema pueda tratar de defenderse frente a las aplicaciones erróneas o maliciosas.

13.2.4 Resumen del hardware de E/S

Aunque los aspectos hardware de las operaciones de E/S son complejos cuando se consideran con el nivel de detalle requerido durante el diseño del hardware electrónico, los conceptos que acabamos de describir son suficientes para comprender muchas de las funciones de E/S de los sistemas operativos. Repasemos los conceptos principales:

- Un bus.
- Una controladora.
- Un puerto de E/S y sus registros.
- El procedimiento de negociación entre el *host* y una controladora de dispositivo.
- La ejecución de este procedimiento mediante un bucle de sondeo o mediante interrupciones.
- La descarga de este trabajo en una controladora de DMA para las transferencias de gran envergadura.

Anteriormente en esta sección, hemos proporcionado un ejemplo básico del procedimiento de negociación que tiene lugar entre una controladora de dispositivo y el *host*. En realidad, la amplia variedad de dispositivos disponibles constituye un problema para los encargados de implementar un sistema operativo. Cada tipo de dispositivo tiene su propio conjunto de capacidades, sus propias definiciones de bits de control y sus propios protocolos para interactuar con el *host*, y todos ellos son diferentes. ¿Cómo podemos diseñar el sistema operativo para poder conectar nuevos dispositivos a la computadora sin reescribir el propio sistema operativo? Y, cuando los dispositivos varían tan ampliamente, ¿cómo puede proporcionar el sistema operativo una interfaz cómoda y uniforme de E/S a todas las aplicaciones? Vamos a tratar a continuación de responder a estas preguntas.

13.3 Interfaz de E/S de las aplicaciones

En esta sección, vamos a hablar de las técnicas de estructuración y de las interfaces del sistema operativo que permiten tratar de una forma estándar y uniforme a los dispositivos de E/S. Explicaremos, por ejemplo, cómo puede una aplicación abrir un archivo en disco sin saber de qué

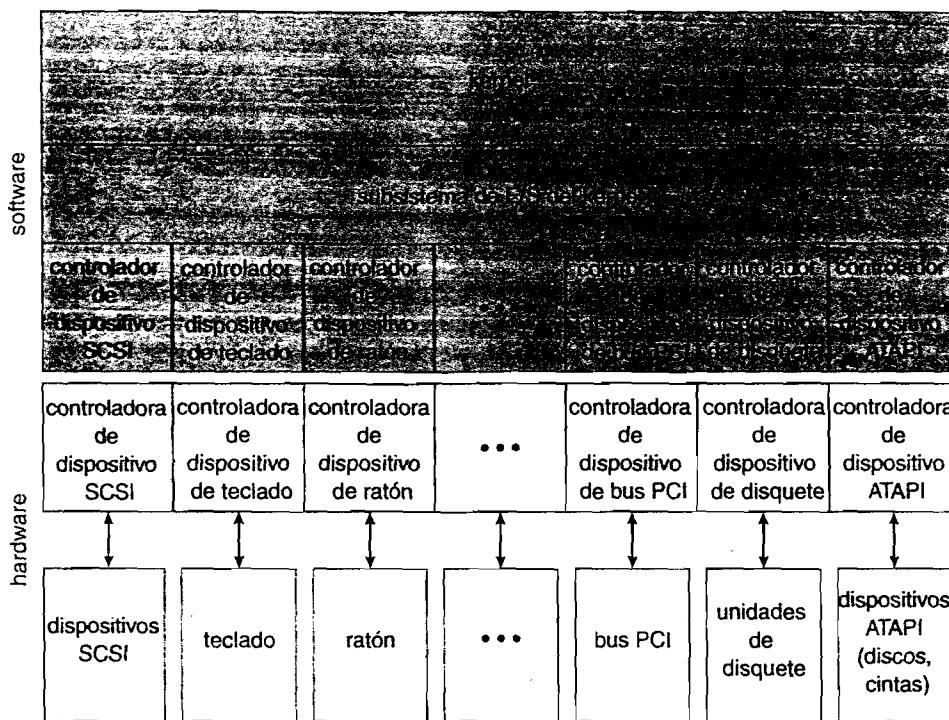


Figura 13.6 Estructura de E/S de un kernel.

tipo de disco se trata, y veremos también cómo pueden añadirse nuevos discos y otros dispositivos a una computadora sin perturbar el sistema operativo.

Al igual que otros problemas complejos de ingeniería del software, la técnica utilizada aquí se basa en los conceptos de abstracción, encapsulación y descomposición del software en niveles. Específicamente, podemos abstraer las diferencias de detalle existentes entre los dispositivos de E/S identificando unos cuantos tipos generales de dispositivo. Para acceder a cada tipo general se utiliza un conjunto estandarizado de funciones, es decir, una **interfaz**. Las diferencias se encapsulan en módulos del *kernel* denominados controladores del dispositivo que están personalizados internamente para cada dispositivo pero exportan una de las interfaces estándar. La Figura 13.6 ilustra cómo se estructuran en niveles de software las partes del *kernel* relacionadas con la E/S.

El propósito de la capa de controladores de dispositivo es ocultar a ojos del subsistema de E/S del *kernel* las diferencias existentes entre las controladoras de dispositivo, de forma similar a como las llamadas al sistema de E/S encapsulan el comportamiento de los dispositivos en unas cuantas clases genéricas que ocultan a ojos de las aplicaciones las diferencias hardware. Hacer el subsistema de E/S independiente del hardware simplifica el trabajo del desarrollador de sistemas operativos y también beneficia a los fabricantes de hardware. Éstos pueden diseñar los nuevos dispositivos para que sean compatibles con una interfaz de controladora *host* existente (como por ejemplo SCSI-2), o pueden escribir controladores de dispositivo para implementar la interfaz del nuevo hardware con una serie de sistemas operativos populares. De este modo, podemos conectar nuevos periféricos a una computadora sin esperar a que el fabricante del sistema operativo desarrolle el correspondiente código de soporte.

Desafortunadamente para los fabricantes de dispositivos hardware, cada tipo de sistema operativo tiene sus propios estándares en cuanto a la interfaz del controlador de dispositivo. Un dispositivo determinado puede comercializarse junto con múltiples controladores de dispositivo, como por ejemplo controladores para MS-DOS, Windows 95/98, Windows NT/2000 y Solaris. Los dispositivos pueden variar desde muchos puntos de vista, como se ilustra en la Figura 13.7.

- **Flujo de caracteres o bloque.** Un dispositivo de flujo de caracteres transfiere los bytes uno a uno, mientras que un dispositivo de bloque transfiere un bloque de bytes como una sola unidad.

aspecto	variación	ejemplo
modo de transferencia de datos	Carácter bloque	terminal disco
método de acceso	secuencial aleatorio	modem CD-ROM
forma de transferencia	síncrona asíncrona	CD teclado
compatición	dedicado compatible	CD teclado
velocidad del dispositivo	latencia tiempo de búsqueda velocidad de transferencia retardo entre operaciones	CD teclado
dirección de E/S	sólo lectura sólo escritura lectura-escritura	CD-ROM controlador gráfica disco

Figura 13.7 Características de los dispositivos de E/S.

- **Acceso secuencial o aleatorio.** Un dispositivo secuencial transfiere los datos en un orden fijo determinado por el dispositivo, mientras que el usuario de un dispositivo de acceso aleatorio puede instruir al dispositivo para que se posicione en cualquiera de las ubicaciones disponibles de almacenamiento de datos.
- **Síncrono o asíncrono.** Un dispositivo síncrono realiza transferencias de datos con tiempos de respuesta predecibles. Un dispositivo asíncrono exhibe unos tiempos de respuesta irregulares o no predecibles.
- **Compatible o dedicado.** Un dispositivo compatible puede ser usado de forma concurrente por varios procesos o hebras; un dispositivo dedicado no puede ser compartido de esta forma.
- **Velocidad de operación.** Las velocidades de los dispositivos van desde unos pocos bytes por segundo a unos cuantos gigabytes por segundo.
- **Lectura-escritura, sólo lectura o sólo escritura.** Algunos dispositivos realizan tanto entrada como salida, pero otros sólo soportan una única dirección de transferencia de los datos.

En lo que respecta al acceso por parte de las aplicaciones, muchas de estas diferencias quedan ocultas gracias al sistema operativo, y los dispositivos se agrupan en unos cuantos tipos convencionales. Los estilos resultantes de acceso al dispositivo han demostrado ser en la práctica muy útiles y de aplicación general. Aunque las llamadas exactas al sistema pueden diferir de unos sistemas operativos a otros, las categorías de dispositivo son bastante estándar. Los principales sistemas de acceso incluyen la E/S de bloque, la E/S de flujo de caracteres, el acceso a archivos mapeados en memoria y los *sockets* de red. Los sistemas operativos también proporcionan llamadas especiales al sistema para acceder a unos cuantos dispositivos adicionales, como por ejemplo un reloj o un contador. Algunos sistemas operativos proporcionan un conjunto de llamadas al sistema para dispositivos de visualización gráfica, de vídeo y de audio.

La mayoría de los sistemas operativos también tienen un *escape* (o *puerta trasera*) que pasa de manera transparente una serie de comandos arbitrarios desde una aplicación a un controlador de dispositivo. En UNIX, esta llamada al sistema es *ioctl()* (que quiere "control de E/S"). La llamada al sistema *ioctl()* permite a las aplicaciones acceder a cualquier funcionalidad que pueda implementarse por algún controlador de dispositivo, sin necesidad de inventar una nueva llamada al sistema. La llamada al sistema *ioctl()* tiene tres argumentos. El primero es un descriptor

de archivo que conecta la aplicación con el controlador haciendo referencia a un dispositivo hardware gestionado por ese controlador. El segundo es un valor entero que selecciona uno de los comandos implementados en el controlador. El tercero es un puntero a una estructura de datos arbitraria en memoria que permite que la aplicación y el controlador se intercambien cualquier dato o cualquier información de control necesaria.

13.3.1 Dispositivos de bloques y de caracteres

La interfaz de **dispositivo de bloques** captura todos los aspectos necesarios para acceder a unidades de disco y a otros dispositivos orientados a bloques. El dispositivo debe comprender comandos tales como `read()` o `write()`; si se trata de un dispositivo de acceso aleatorio, también se espera que disponga de un comando `seek()` para especificar qué bloque hay que transferir a continuación. Las aplicaciones acceden normalmente a este tipo de dispositivos mediante una interfaz de sistema de archivos. Podemos ver que `read()`, `write()` y `seek()` capturan el comportamiento esencial de los dispositivos de almacenamiento de bloques, de modo que las aplicaciones quedan aisladas de las diferencias de bajo nivel que puedan existir entre los dispositivos.

El propio sistema operativo, al igual que algunas aplicaciones como los sistemas de gestión de bases de datos, pueden preferir acceder a un dispositivo de bloques como si fuera una simple matriz lineal de bloques. Este modo de acceso se denomina en ocasiones **E/S sin formato** o **en bruto**. Si la aplicación realiza su propio almacenamiento en búfer, la utilización de un sistema de archivos provocaría un almacenamiento en búfer adicional e innecesario. De la misma forma, si la aplicación proporciona su propio mecanismo de bloqueo de regiones o de bloques de archivos, los servicios de bloqueo del sistema operativo serían redundantes como mínimo y contradictorios en el caso peor. Para evitar estos conflictos, los mecanismos de acceso a dispositivos sin formato pasan el control del dispositivo directamente a la aplicación, dejando que el sistema operativo no interfiera en la tarea. Desafortunadamente, con este mecanismo no se pueden realizar servicios del sistema operativo para ese dispositivo. Una solución de compromiso que cada vez se está adoptando de forma más común consiste en que el sistema operativo incluya un modo de operación con los archivos en el que se desactiven el almacenamiento en búfer y los mecanismos de bloqueo. En el mundo UNIX, este modo se denomina **E/S directa**.

Los mecanismos de acceso a archivos mapeados en memoria pueden implementarse por encima de los controladores de dispositivos de bloques. En lugar de ofrecer operaciones de lectura y escritura, una interfaz de mapeo en memoria proporciona acceso al almacenamiento en disco mediante una matriz de bytes de la memoria principal. La llamada al sistema que mapea un archivo en la memoria devuelve la dirección de memoria virtual que contiene una copia del archivo. Las propias transferencias de datos se analizan únicamente cuando son necesarias para satisfacer el acceso a la imagen en memoria. Puesto que las transferencias se gestionan mediante el mismo mecanismo que se emplea para el acceso a memoria virtual con paginación bajo demanda, la E/S mapeada en memoria resulta muy eficiente. El mapeo en memoria también es cómodo para los programadores, ya que el acceso a un archivo mapeado en memoria es tan simple como leer y escribir en la memoria. Los sistemas operativos que ofrecen memoria virtual utilizan comúnmente la interfaz de mapeo para los servicios del *kernel*. Por ejemplo, para ejecutar el programa, el sistema operativo mapea el archivo ejecutable en la memoria y luego transfiere el control a la dirección de entrada de ese código ejecutable. La interfaz de mapeo también se utiliza comúnmente para que el *kernel* acceda al espacio de intercambio en el disco.

El teclado es un ejemplo de dispositivo al que se accede mediante una interfaz de flujo de **caracteres**. Las llamadas básicas al sistema en esta interfaz permiten a las aplicaciones leer o escribir un carácter, mediante las llamadas `get()` y `put()`. Por encima de esta interfaz, pueden construirse bibliotecas que permitan el acceso de línea en línea, con servicios de almacenamiento en búfer y edición (por ejemplo, cuando un usuario escribe un carácter de retroceso, se elimina del flujo de entrada el carácter anterior). Este estilo de acceso resulta cómodo para dispositivos de entrada tales como los teclados, los ratones y los módems que producen datos de entrada "espontánea" en instantes que no pueden necesariamente ser predichos por la aplicación. Este esti-

lo de acceso también resulta adecuado para dispositivos de salida tales como las impresoras y las tarjetas de sonido, que se adaptan naturalmente al concepto de un flujo lineal de bytes.

13.3.2 Dispositivos de red

Puesto que las características de velocidad y de direccionamiento de la E/S de red difieren significativamente de las de la E/S de disco, la mayoría de los sistemas operativos proporcionan una interfaz de E/S de red que es diferente de la interfaz `read()`-`write()`-`seek()` utilizada para los discos. Una interfaz disponible en muchos sistemas operativos, incluyendo UNIX y Windows NT, es la interfaz de *sockets* de red.

La palabra *socket* en inglés hace referencia a las tomas de electricidad existentes en las paredes de una vivienda: en esas tomas podemos conectar cualquier electrodoméstico. Por analogía, las llamadas al sistema de la interfaz *sockets*, permiten a la aplicación crear un *socket*, conectar el *socket* local a una dirección remota (lo que conecta esta aplicación al *socket* creado por otra aplicación), quedar a la espera de que cualquier aplicación remota se conecte al *socket* local y enviar y recibir paquetes a través de la conexión. Para soportar la implementación de servidores, la interfaz de *sockets* también proporciona una función denominada `select()` que gestiona un conjunto de *sockets*. Una llamada a `select()` devuelve información acerca de qué *sockets* tienen un paquete a la espera de ser recibido y qué *sockets* disponen del espacio para aceptar un paquete que haya que enviar. La utilización de `select()` elimina los mecanismos de sondeo y de espera activa que serían necesarios para la E/S de red en caso de que esta llamada no existiera. Estas funciones encapsulan el comportamiento esencial de las redes, facilitando enormemente la creación de aplicaciones distribuidas que pueden utilizar cualquier hardware de red y cualquier pila de protocolos subyacentes.

También se vienen implementando muchas otras técnicas de comunicación interprocesos y de comunicación por red. Por ejemplo, Windows NT proporciona una interfaz para la tarjeta de interfaz de red y una segunda interfaz para los protocolos de red (Sección C.6). En UNIX, que tiene un largo historial de campo de pruebas para las tecnologías de red, podemos encontrar canalizaciones semi-dúplex, colas FIFO full-dúplex, conexiones STREAMS full-dúplex, colas de mensajes y *sockets*. En el Apéndice A (Sección A. 9) se proporciona información sobre los mecanismos de comunicación por red en UNIX.

13.3.3 Relojes y temporizadores

La mayoría de las computadoras disponen de relojes y temporizadores hardware que proporcionan tres funciones básicas:

- Proporcionar la hora actual.
- Proporcionar el tiempo transcurrido.
- Configurar un temporizador para que provoque la ejecución de la operación *X* en el instante *T*.

El sistema operativo, así como las aplicaciones que tengan restricciones temporales críticas utilizan intensivamente estas funciones. Desafortunadamente, las llamadas al sistema que implementan estas funciones no están estandarizadas entre unos sistemas operativos y otros.

El hardware necesario para medir el tiempo transcurrido y para provocar la ejecución de operaciones se denomina **temporizador de intervalo programable**. Se puede configurar el temporizador para esperar una cierta cantidad de tiempo y luego generar una interrupción, y se lo puede configurar para hacer esto una sola vez o para repetir el proceso de modo que se generen interrupciones periódicas. El planificador utiliza este mecanismo para generar una interrupción que provocará el desalojo de un proceso al final de su correspondiente franja temporal de ejecución. El subsistema de E/S de disco utiliza este mecanismo para invocar la orden de volcar en el disco periódicamente los búferes caché sucios y el subsistema de red lo utiliza para cancelar aquellas operaciones que estén progresando de forma demasiado lenta debido a fallos de red o a la con-

gestión. El sistema operativo puede también proporcionar una interfaz para que los procesos de usuario utilicen temporizadores. El sistema operativo puede soportar más solicitudes de temporización que el número de canales hardware de temporización disponibles, simulando relojes virtuales. Para hacer esto, el *kernel* (o el controlador del dispositivo temporizador) mantiene una lista de las interrupciones deseadas por sus propias rutinas y por las solicitudes de usuario, ordenadas de acuerdo con el instante en que van a producirse, de la más próxima a la más lejana. El *kernel* considera el temporizador para dar servicio a la interrupción más próxima. Cuando el temporizador interrumpe, el *kernel* señala dicho suceso al proceso solicitante y recarga el temporizador con la siguiente interrupción de la secuencia.

En muchas computadoras, la frecuencia de interrupciones generada por el reloj hardware está comprendida entre 18 y 60 tics por segundo. Esta resolución no es muy buena, ya que una computadora moderna puede ejecutar centenares de millones de instrucciones por segundo. La precisión de los disparadores está limitada por esta baja resolución del temporizador, además de por el procesamiento adicional que se requiere para la gestión de los relojes virtuales. Además, si se utilizan los tics del temporizador para mantener el reloj del sistema, dicho reloj puede llegar a experimentar una deriva. En la mayoría de las computadoras, el reloj hardware se construye a partir de un contador de alta frecuencia. En algunas computadoras, puede leerse el valor de este contador en un registro del dispositivo, en cuyo caso el contador puede considerarse un reloj de alta resolución. Aunque este reloj no genera interrupciones, sí que permite realizar mediciones precisas de intervalos de tiempo.

13.3.4 E/S bloqueante y no bloqueante

Otro aspecto de la interfaz de llamadas al sistema es el relativo a la elección entre E/S bloqueante y no bloqueante. Cuando una aplicación ejecuta una llamada al sistema **bloqueante**, se suspende la ejecución de la aplicación. En ese instante, la aplicación se pasa de la cola de ejecución del sistema operativo a una cola de espera. Después de que se complete la llamada al sistema, vuelve a colocarse la aplicación en la cola de ejecución, donde pasará a ser elegible para reanudar su ejecución, en cuyo momento se le entregarán los valores devueltos por la llamada al sistema. Las acciones físicas realizadas por los dispositivos de E/S son generalmente asíncronas, consumiendo una cantidad variable o impredecible de tiempo. De todos modos, la mayoría de los sistemas operativos utilizan llamadas al sistema bloqueantes para la interfaz de las aplicaciones, porque el código de aplicación bloqueante es más fácil de entender que el no bloqueante.

Algunos procesos de nivel de usuario necesitan una E/S **no bloqueante**. Un ejemplo sería una interfaz de usuario que recibe la entrada de teclado y de ratón mientras que está procesando y mostrando datos en la pantalla. Otro ejemplo sería una aplicación de vídeo que leyera imágenes de un archivo en disco mientras que simultáneamente las descomprime y muestra la salida en la pantalla.

Una forma en que el desarrollador de una aplicación puede solapar la ejecución con las operaciones de E/S consiste en escribir una aplicación multihebra. Algunas hebras pueden realizar llamadas al sistema bloqueantes mientras que otras continúan ejecutándose. Los desarrolladores de Solaris utilizaron esta técnica para implementar una biblioteca de nivel de usuario para la E/S asíncrona, liberando a los desarrolladores de aplicaciones de esa tarea. Algunos sistemas operativos proporcionan llamadas al sistema de E/S no bloqueante. Una llamada no bloqueante no detiene la ejecución de la aplicación durante un período de tiempo prolongado. En lugar de ello, la llamada vuelve rápidamente, con un valor de retorno que indica cuántos bytes se han transferido.

Una alternativa a las llamadas al sistema no bloqueantes consiste en utilizar llamadas al sistema asíncronas. Una llamada al sistema asíncrona vuelve inmediatamente, sin esperar a que la operación de E/S se complete. La aplicación continúa entonces ejecutando su código. La terminación de la operación de E/S se comunica a la aplicación en algún instante futuro, configurando alguna variable en el espacio de direcciones de la aplicación o generando una señal o interrupción software o ejecutando una rutina de retrollamada que se ejecuta fuera del flujo lineal de control de la aplicación. La diferencia entre las llamadas al sistema no bloqueantes y asíncronas es que una operación `read()` no bloqueante vuelve inmediatamente con los datos que haya disponibles

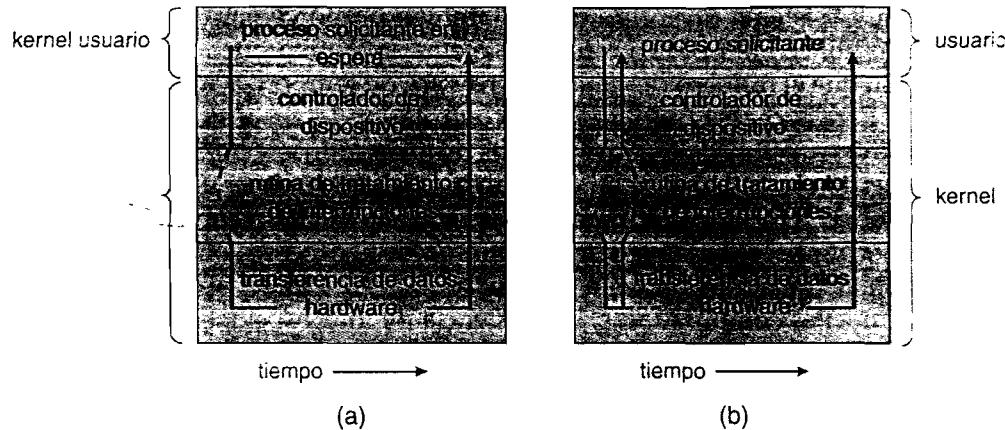


Figura 13.8 Dos métodos de E/S: (a) síncrono y (b) asíncrono.

(el número completo de bytes solicitados, un número menor o ningún byte en absoluto), mientras que una llamada `read()` asíncrona solicita una transferencia que se realizará de modo completo pero que se completará en algún instante futuro. Estos dos métodos de E/S se ilustran en la Figura 13.8.

Un buen ejemplo de comportamiento no bloqueante es la llamada al sistema `select()` para los *sockets* de red. Esta llamada al sistema acepta un argumento que especifica un tiempo de espera máximo: si le asignamos el valor 0, la aplicación puede efectuar un sondeo de la actividad de red sin bloquearse. Pero utilizar `select()` introduce una carga de procesamiento adicional, porque la llamada `select()` sólo comprueba si la E/S es posible. Para realizar una transferencia de datos, hay que ejecutar después de `select()` algún tipo de comando `read()` o `write()`. Una variación de esta técnica, que podemos encontrar en Mach, es la llamada de lectura múltiple bloqueante. Esta llamada especifica las lecturas deseadas para varios dispositivos en una única llamada al sistema y vuelve en cuanto se completa una de las lecturas.

13.4 Subsistema de E/S del kernel

Un *kernel* proporciona muchos servicios relacionados con la E/S. Varios de los servicios (planificación, almacenamiento en búfer, almacenamiento en caché, gestión de colas, reserva de dispositivos y tratamiento de errores) son proporcionados por el subsistema de E/S del *kernel* y se construyen sobre la infraestructura formada por el hardware y los controladores de dispositivo. El subsistema de E/S también es responsable de protegerse a sí mismo de los procesos erróneos y de los usuarios maliciosos.

13.4.1 Planificación de E/S

Planificar un conjunto de solicitudes de E/S significa determinar un orden adecuado en el que ejecutarlas. El orden en el que las aplicaciones realizan las llamadas al sistema no suele ser la mejor elección. La planificación puede mejorar el rendimiento global del sistema, permite compartir el acceso a los dispositivos equitativamente entre los distintos procesos y puede reducir el tiempo de espera medio requerido para que la E/S se complete. He aquí un ejemplo simple para ilustrar estos aspectos: suponga que el brazo de un disco está situado cerca del principio del mismo y que tres aplicaciones distintas ejecutan llamadas de lectura bloqueantes dirigidas a dicho disco. La aplicación 1 solicita un bloque situado cerca del final del disco, la aplicación 2 solicita otro situado cerca del principio y la aplicación 3 solicita un bloque en parte intermedia del disco. El sistema operativo puede reducir la distancia recorrida por el brazo del disco sirviendo a las aplicaciones en el orden 2, 3, 1. Reordenar el servicio de las distintas solicitudes de esta manera es la esencia de los mecanismos de planificación de E/S.

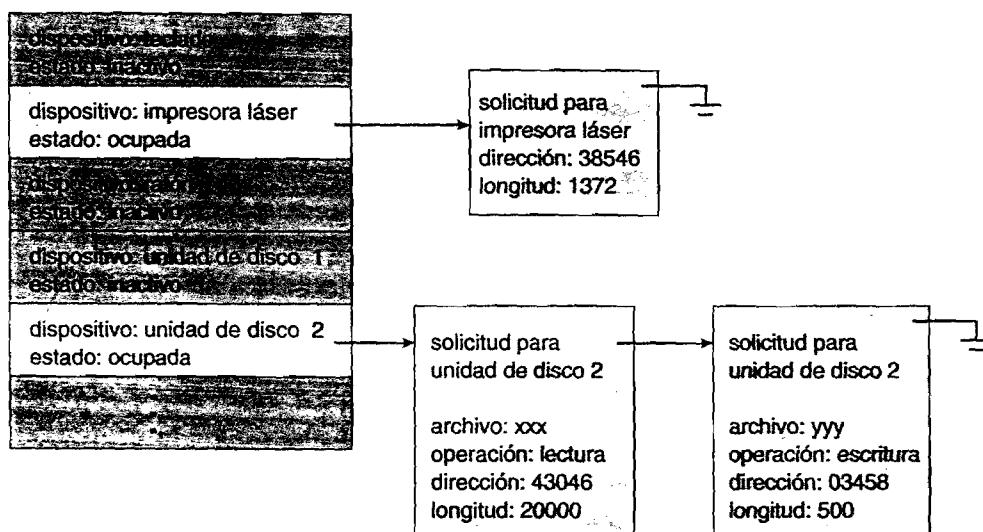


Figura 13.9 Tabla de estado de los dispositivos.

Los desarrolladores de sistemas operativos implementan los mecanismos de planificación manteniendo una cola de espera de solicitudes para cada dispositivo. Cuando una aplicación ejecuta una llamada al sistema de E/S bloqueante, la solicitud se coloca en la cola correspondiente a dicho dispositivo. El planificador de E/S reordena la cola para mejorar la eficiencia global del sistema y el tiempo medio de respuesta experimentado por las aplicaciones. El sistema operativo puede tratar también de ser equitativo, de modo que ninguna aplicación reciba un servicio especialmente pobre, o puede proporcionar un servicio prioritario a las solicitudes más sensibles al retardo. Por ejemplo, las solicitudes procedentes del subsistema de memoria virtual pueden tener prioridad sobre las solicitudes realizadas por las aplicaciones. En la Sección 12.4 se detallan diversos algoritmos de planificación para la E/S de disco.

Cuando un *kernel* soporta mecanismos de E/S asíncrona, debe ser capaz de controlar múltiples solicitudes de E/S al mismo tiempo. Con este fin, el sistema operativo puede asociar la cola de espera con una **tabla de estado del dispositivo**. El *kernel* se encarga de gestionar esta tabla, que contiene una entrada por cada dispositivo de E/S, como se muestra en la Figura 13.9. Cada entrada en la tabla indica el tipo, la dirección y el estado (no funcional, inactivo u ocupado) del dispositivo. Si el dispositivo está ocupado con una solicitud, se almacenarán en la entrada de la tabla correspondiente a dicho dispositivo el tipo de la solicitud y otros parámetros.

Una forma en que el subsistema de E/S mejora la eficiencia en la computadora es planificando las operaciones de E/S. Otra forma es utilizando espacio de almacenamiento en la memoria principal o en disco mediante técnicas denominadas almacenamiento en búfer, almacenamiento en caché y gestión de colas.

13.4.2 Almacenamiento en búfer

Un **búfer** es un área de memoria que almacena datos mientras se están transfiriendo entre dos dispositivos o entre un dispositivo y una aplicación. El almacenamiento en búfer se realiza por tres razones. Una razón es realizar una adaptación de velocidades entre el productor y el consumidor de un flujo de datos. Suponga, por ejemplo, que se está recibiendo un archivo vía módem para su almacenamiento en el disco duro. El módem es unas 1000 veces más lento que el disco duro, por lo que se crea un búfer en memoria principal para acumular los bytes recibidos del módem. Una vez que ha llegado un búfer completo de datos, puede escribirse el búfer en disco en una sola operación. Puesto que la escritura en el disco no es instantánea y el módem sigue necesitando un lugar para almacenar los datos adicionales entrantes, se utilizan los búferes. Después de que el módem llene el primer búfer, se solicita la escritura en disco y el módem comienza entonces a enviar datos al búfer mientras que el primero se escribe en el disco. Para cuando el módem

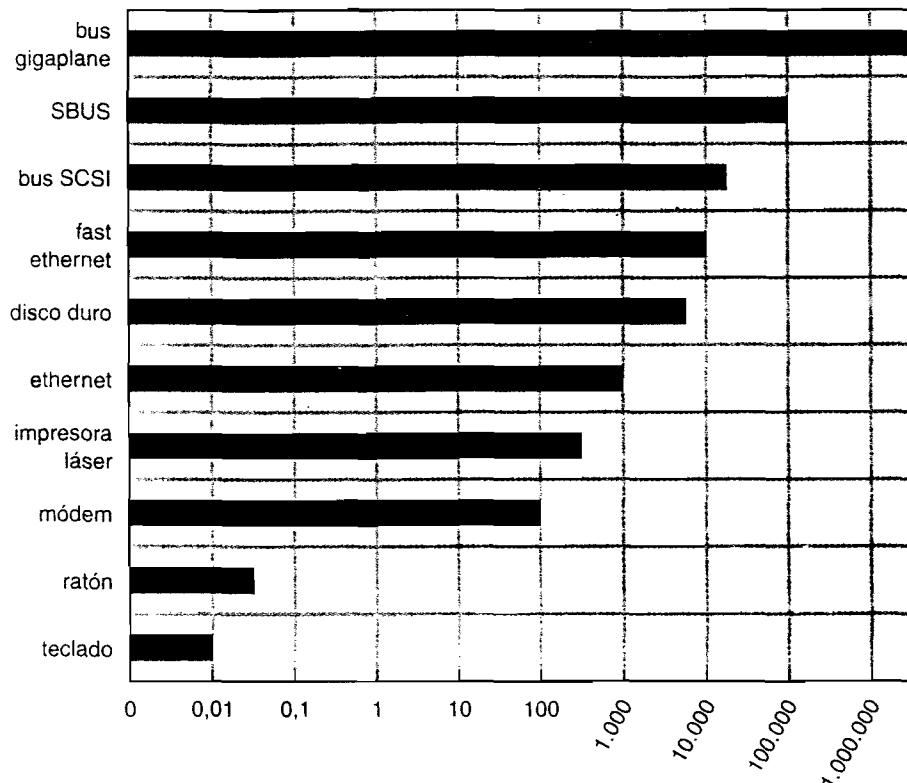


Figura 13.10 Velocidades de transferencia de dispositivos en las máquinas Sun Enterprise 6000 (escala logarítmica).

haya llenado el segundo búfer, la escritura en disco del primero deberá haberse completado, por lo que el módem podrá conmutar de nuevo al primer búfer mientras se escribe en el disco el contenido del segundo. Esta técnica de **doble búfer** desacopla al productor de datos del consumidor de los mismos, relajando así los requisitos de temporización entre ellos. La necesidad de este desacoplamiento se ilustra en la Figura 13.10, donde se indican las enormes diferencias en velocidades de los dispositivos para un hardware típico de una computadora.

Un segundo uso del almacenamiento en búfer consiste en realizar la adaptación entre dispositivos que tengan diferentes tamaños de transferencia de datos. Dichas disparidades son especialmente comunes en la conexión por red de computadoras, en la que se utilizan intensivamente los búferes para la fragmentación y recomposición de mensajes. En el lado emisor, se fragmentan los mensajes de gran tamaño en pequeños paquetes de red. Estos paquetes se envían a través de la red y el lado receptor los coloca en un búfer de recomposición, para formar una imagen de los datos de origen.

Un tercer uso del almacenamiento de un búfer es el de soportar la semántica de copia en la E/S de las aplicaciones. Un ejemplo nos permitirá clarificar cuál es el significado del concepto de "semántica de copia". Suponga que una aplicación dispone de un búfer de datos que desea escribir en disco. La aplicación ejecutará la llamada al sistema `write()`, proporcionando un puntero al búfer y un número entero que especifique el número de bytes que hay que escribir. Después de que vuelve la llamada al sistema, ¿qué sucede si la aplicación cambia el contenido del búfer? Con la **semántica de copia**, se garantiza que la versión de los datos escrita en el disco será la versión existente en el momento en que la aplicación ejecute la llamada al sistema, independientemente de los cambios posteriores que la aplicación pueda realizar en el búfer. Una forma simple en la que el sistema operativo puede garantizar la semántica de copia es que la llamada al sistema `write()` copie los datos de la aplicación en un búfer del *kernel* antes de devolver el control a la aplicación. La escritura en disco se realiza a partir del búfer del *kernel*, de modo que los cambios subsiguientes realizados en el búfer de la aplicación no tendrán ningún efecto. La copia de datos

entre búferes del *kernel* y el espacio de datos de la aplicación resulta bastante común en los sistemas operativos, a pesar de la carga de procesamiento adicional que esta operación introduce, y la razón de que resulte bastante común es que se trate de una semántica muy limpia. Puede obtenerse el mismo efecto de manera más eficiente utilizando de forma inteligente los mecanismos de mapeo en memoria virtual y de protección de páginas mediante funciones de copia durante la escritura.

13.4.3 Almacenamiento en caché

Una **caché** es una región de memoria rápida que alberga copias de ciertos datos. El acceso a la copia almacenada en caché es más eficiente que el acceso al original. Por ejemplo las instrucciones del proceso actualmente en ejecución están almacenadas en el disco, almacenadas en la caché de la memoria física y almacenadas también en las cachés principal y secundaria de la CPU. La diferencia entre un búfer y una caché es que un búfer puede almacenar la única copia existente de un elemento de datos, mientras que una caché, por definición, almacena simplemente en un dispositivo de almacenamiento más rápido una copia de un elemento que reside en algún otro lugar.

El almacenamiento en caché y el almacenamiento en búfer son funciones bien diferenciadas, aunque en ocasiones puede utilizarse una misma región de memoria para ambos propósitos. Por ejemplo, para preservar la semántica de copia y para permitir una eficiente planificación de la E/S de disco, el sistema operativo utiliza búferes en memoria principal para almacenar los datos del disco. Estos búferes también se emplean como caché, para mejorar la eficiencia de E/S para aquellos archivos que estén compartidos por varias aplicaciones o que estén siendo escritos y vuelto a leer de forma rápida. Cuando el *kernel* recibe una solicitud de E/S de archivo, accede primero a la caché de búfer para ver si dicha región del archivo ya está disponible en la memoria principal. En caso afirmativo, podemos editar o diferir una E/S de disco físico. Asimismo, las escrituras en disco se acumulan en la caché del búfer durante varios segundos, con el fin de componer grandes transferencias de datos y permitir así una planificación eficiente de las operaciones de escritura. Esta estrategia de retardar las escrituras para mejorar la eficiencia de E/S se analiza, en el contexto del acceso a archivos remotos, en la Sección 17.3.

13.4.4 Gestión de colas y reserva de dispositivos

Una **cola de dispositivo** es un búfer que almacena la salida dirigida a un dispositivo, como por ejemplo una impresora, que no pueda aceptar flujos de datos entrelazados. Aunque una impresora sólo puede dar servicio a un cierto trabajo cada vez, es posible que varias aplicaciones quieran imprimir sus datos de salida de manera concurrente, sin que la salida de unas aplicaciones se mezcle con la de otras. El sistema operativo resuelve este problema interceptando toda la salida dirigida a la impresora. La salida de cada aplicación se almacena temporalmente en un archivo de disco separado. Cuando una aplicación termina de imprimir, el sistema de gestión de colas pone el correspondiente archivo temporal en la cola de salida de la impresora. El sistema de gestión de colas va copiando los archivos de salida en la impresora de uno en uno. En algunos sistemas operativos, estas colas de impresión se gestionan mediante un proceso demonio del sistema. En otros, se gestiona mediante una hebra interna al *kernel*. En cualquiera de los dos casos el sistema operativo proporciona una interfaz de control que permite a los usuarios y a los administradores del sistema visualizar la cola de solicitudes de impresión, eliminar los trabajos no deseados antes de que lleguen a imprimirse, suspender la impresión mientras se está solventando algún error de impresora, etc.

Algunos dispositivos, como las unidades de cinta y las impresoras, no pueden multiplexar las solicitudes de E/S de múltiples aplicaciones concurrentes. La gestión de colas de impresión es una de las formas en que el sistema operativo puede coordinar estas operaciones concurrentes de salida. Otra forma de tratar con el acceso concurrente a los dispositivos consiste en proporcionar facilidades explícitas de coordinación. Algunos sistemas operativos (incluyendo VMS) proporcionan soporte para el acceso exclusivo a los dispositivos, permitiendo a los procesos asignar un dispositivo inactivo y desasignarlo cuando ya no sea necesario. Otros sistemas operativos imponen un

límite de un sólo descriptor de archivo abierto para tales dispositivos. Muchos sistemas operativos proporcionan funciones que permiten a los procesos coordinar entre sí el acceso exclusivo. Por ejemplo, Windows NT proporciona llamadas al sistema para realizar una espera hasta que un objeto dispositivo pase a estar disponible. También dispone de un parámetro de la llamada al sistema `open()` que declara los tipos de acceso que deben permitirse a otras hebras concurrentes. En estos sistemas, es responsabilidad de las aplicaciones evitar los interbloqueos.

13.4.5 Tratamiento de errores

Un sistema operativo que utilice memoria protegida puede defenderse frente a muchos tipos de errores hardware y de las aplicaciones, de modo que cada pequeño error no provoque un fallo completo del sistema. Los dispositivos y las transferencias de E/S pueden fallar de muchas formas, debido a razones transitorias (como por ejemplo cuando una red se sobrecarga) o a razones “permanentes” (como por ejemplo si falla una controladora de disco). Los sistemas operativos pueden a menudo compensar de manera efectiva los fallos transitorios. Por ejemplo, un fallo de una operación `read()` de disco provoca que se reintente la operación `read()` y un error en una operación `send()` a través de la red provoca una operación de reenvío `resend()`, si el protocolo así lo especifica. Desafortunadamente, si un componente importante experimenta un fallo de carácter permanente, resulta poco probable que el sistema operativo pueda recuperarse.

Como regla general, una llamada de E/S al sistema devolverá un bit de información acerca del estado de la llamada, mediante el que se indicará si ésta ha tenido éxito o no. En el sistema operativo UNIX, se utiliza una variable entera adicional denominada `errno` para devolver un código de error (de entre un centenar de valores posibles) que indica la naturaleza general del fallo (por ejemplo, argumento fuera de rango, puntero erróneo o archivo no abierto). Por contraste, ciertos tipos de hardware pueden proporcionar información de error altamente detallada, aunque muchos de los sistemas operativos actuales no están diseñados para transmitir esta información a la aplicación. Por ejemplo, un fallo en un dispositivo SCSI se documenta por parte del protocolo SCSI en tres niveles de detalle: una **clave de tipo** que identifica la naturaleza general del fallo, como por ejemplo un error hardware o una solicitud ilegal; un **código de tipo adicional** que indica la categoría del fallo, como por ejemplo un parámetro erróneo de comando o un fallo de auto-test; y un **cualificador adicional del código de tipo** que proporciona todavía más detalle, como por ejemplo qué parámetro del comando era erróneo o qué subsistema hardware ha fallado al hacer el auto-test. Además, muchos dispositivos SCSI mantienen páginas internas de información de registro de errores que pueden ser solicitadas por el *host*, aunque raramente se solicitan.

13.4.6 Protección de E/S

Los errores están estrechamente relacionados con las cuestiones de protección. Un proceso de usuario puede intentar accidental o deliberadamente interrumpir la operación normal de un sistema, tratando de ejecutar instrucciones de E/S ilegales. Podemos utilizar varios mecanismos para garantizar que ese tipo de problemas no aparezcan en el sistema.

Para evitar que los usuarios realicen operaciones de E/S ilegales, definimos todas las instrucciones de E/S como instrucciones privilegiadas. Así, los usuarios no pueden ejecutar instrucciones de E/S directamente, sino que tienen que hacerlo a través del sistema operativo. Para llevar a cabo una operación de E/S, el programa de usuario ejecuta una llamada al sistema para solicitar que el sistema operativo realice esa operación de E/S por cuenta suya (Figura 13.11). El sistema operativo, ejecutándose en modo monitor, comprueba que la solicitud es válida y, en caso afirmativo, realiza la E/S solicitada. A continuación, el sistema operativo devuelve el control al usuario.

Además, habrá que utilizar el sistema de protección de memoria para proteger todas las ubicaciones de memoria mapeada y de los puertos de E/S frente a los accesos de los usuarios. Observe que el *kernel* no puede simplemente denegar todos los accesos de los usuarios. La mayoría de los juegos gráficos y del software de edición y reproducción de vídeo necesitan acceso directo a la memoria de la controladora gráfica mapeada en memoria, con el fin de acelerar la velocidad.

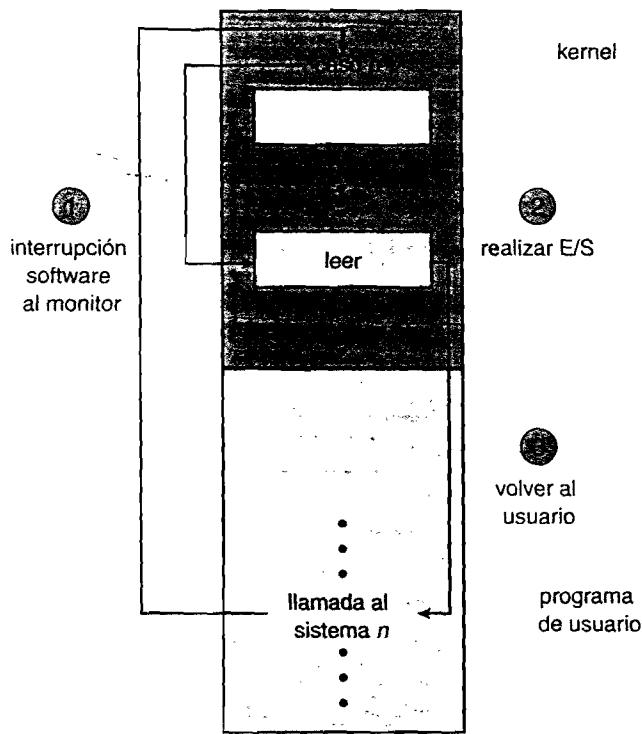


Figura 13.11 Utilización de una llamada al sistema para realizar la E/S.

de los gráficos (por ejemplo). El *kernel* puede en este caso proporcionar un mecanismo de bloqueo para permitir que se asigne a un proceso cada vez una cierta sección de la memoria gráfica (que representará una ventana en la pantalla).

13.4.7 Estructuras de datos del kernel

El *kernel* necesita mantener información de estado acerca del uso de los componentes de E/S. El *kernel* hace esto mediante diversas estructuras de datos internas al *kernel*, como por ejemplo la tabla de archivos abiertos de la Sección 11.1. El *kernel* utiliza muchas estructuras similares para controlar las conexiones de red, las comunicaciones con los dispositivos de tipo carácter y otras actividades de E/S.

UNIX proporciona acceso de sistema de archivos a diversas entidades, como los archivos de usuario, los dispositivos sin formato y los espacios de direcciones de los procesos. Aunque cada una de estas actividades soporta una operación `read()`, la semántica difiere. Por ejemplo, para leer un archivo de usuario, el *kernel* necesita comprobar la caché de búfer antes de decidir si debe realizar una E/S de disco. Para leer un disco sin formato, el *kernel* necesita garantizar que el tamaño de la solicitud sea un múltiplo del tamaño del sector de disco y que esté alineada con una frontera de sector. Para leer la imagen de un proceso, simplemente basta con copiar los datos desde memoria. UNIX encapsula estas diferencias dentro de una estructura uniforme utilizando una técnica de orientación a objetos. El registro de archivos abiertos, mostrado en la Figura 13.12, contiene una tabla de despacho que almacena punteros a las rutinas apropiadas, dependiendo del tipo de archivo.

Algunos sistemas operativos utilizan métodos de orientación a objetos de forma todavía más intensiva. Por ejemplo, Windows NT utiliza una implementación de paso de mensajes para la E/S. Cada solicitud de E/S se convierte en un mensaje que se envía a través del *kernel* al gestor de E/S y luego al controlador de dispositivo, cada uno de los cuales puede modificar el contenido del mensaje. Para las operaciones de salida, el mensaje contiene los datos que hay que escribir. Para las operaciones de entrada, el mensaje contiene un búfer para recibir los datos. La técnica de paso de mensajes puede añadir carga de proceso adicional, por comparación con las técnicas procedi-

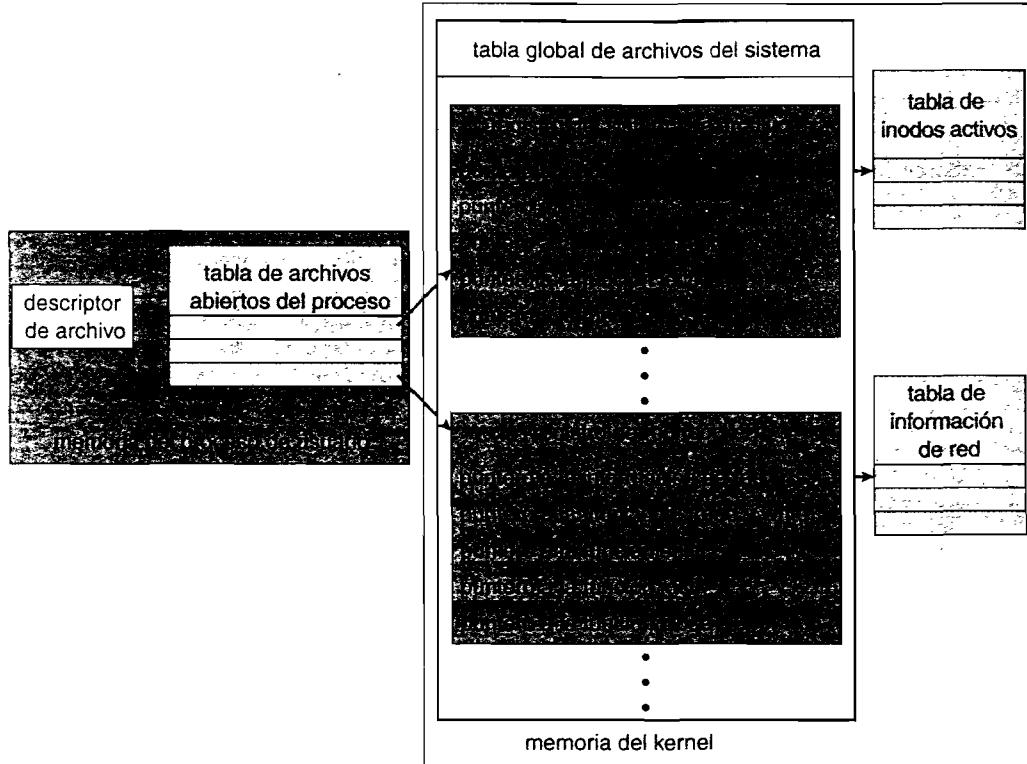


Figura 13.12 Estructura de E/S del *kernel* de UNIX.

mentales que utilizan estructuras de datos compartidas, pero simplifica la estructura y el diseño del sistema de E/S y añade un grado considerable de flexibilidad.

13.4.8 Resumen del subsistema de E/S del *kernel*

En resumen, el subsistema de E/S coordina una amplia colección de servicios disponibles para las aplicaciones y para otras partes del *kernel*. El subsistema de E/S supervisa los siguientes procedimientos:

- Gestión del espacio de nombres para archivos y dispositivos.
- Control de acceso a los archivos y dispositivos.
- Control de operaciones (por ejemplo, un módem no puede ejecutar una operación `seek()`).
- Asignación de espacio en el sistema de archivos.
- Asignación de dispositivos.
- Almacenamiento en búfer.
- Almacenamiento en caché y gestión de colas de impresión.
- Planificación de E/S.
- Monitorización del estado de los dispositivos, tratamiento de errores y recuperación de fallos.
- Configuración e inicialización de controladores de dispositivos.

Los niveles superiores del subsistema de E/S acceden a los dispositivos a través de la interfaz uniforme proporcionada por los controladores de dispositivo.

13.5 Transformación de las solicitudes de E/S en operaciones hardware

Anteriormente, hemos descrito el procedimiento de negociación entre un controlador de dispositivo y una tarjeta controladora de dispositivo, pero no hemos explicado cómo conecta el sistema operativo una solicitud de aplicación con un conjunto de hilos de red o con un sector de disco específico. Vamos a considerar el ejemplo de la lectura de un archivo de disco. La aplicación hace referencia a los datos mediante el nombre del archivo. Dentro de un disco, el sistema de archivos mapea los nombres de archivo mediante una serie de directorios para averiguar la asignación de espacio del archivo. Por ejemplo, en MS-DOS, el nombre se mapea sobre un número que indica una entrada dentro de la tabla de acceso a los archivos y dicha entrada de la tabla nos dice qué bloques de disco están asignados al archivo. En UNIX, el nombre se mapea sobre un número de inodo, y el correspondiente inodo contiene la información de asignación de espacio. ¿Cómo se realiza la conexión entre el nombre del archivo y la controladora de disco (la dirección del puerto hardware o los registros mapeados en memoria de la controladora)? Primero, vamos a considerar el caso de MS-DOS, que es un sistema operativo relativamente simple. La primera parte de un nombre de archivo MS-DOS, situado delante del carácter de dos puntos, es una cadena que identifica un dispositivo hardware específico. Por ejemplo, *c:* es la primera parte de todos los nombres de archivo que hagan referencia al disco duro principal. El hecho de que *c:* represente el disco duro principal está pre-escrito dentro del sistema operativo; *c:* se mapea sobre una dirección de puerto específica a través de una tabla de dispositivos. Debido al separador de los dos puntos, el espacio de nombres de dispositivos está separado del espacio de nombres del sistema de archivos dentro de cada dispositivo. Esta separación hace que resulte sencillo para el sistema operativo asociar una funcionalidad adicional con cada dispositivo. Por ejemplo, resulta fácil invocar los mecanismos de gestión de colas de impresión para los archivos que se escriban en la impresora.

Si, en lugar de ello, el espacio de nombres de los dispositivos está incorporado dentro del espacio de nombres normal del sistema de archivos, como es el caso en UNIX, los servicios de nombres normales del sistema de archivos se proporcionan de manera automática. Si el sistema de archivos proporciona mecanismos de control de propiedad y de control de acceso para todos los nombres de archivo, entonces los dispositivos tendrán propietarios y mecanismos de control de acceso. Puesto que los archivos se almacenan en dispositivos, dicha interfaz proporciona acceso al sistema de E/S en dos niveles: los nombres pueden utilizarse para acceder a los propios dispositivos o para acceder a los archivos almacenados en los dispositivos.

UNIX representa los nombres de los dispositivos dentro del espacio de nombres normal del sistema de archivos. A diferencia de un nombre de archivo MS-DOS, que tiene un separador de dos puntos, un nombre de ruta UNIX no tiene una separación clara de la parte correspondiente al dispositivo. De hecho, ninguna parte concreta del nombre de ruta es el nombre de un dispositivo. UNIX tiene una **tabla de montaje** que asocia prefijos de los nombres de ruta con nombres de dispositivo específicos. Para resolver el nombre de ruta, UNIX busca el nombre en la tabla de montaje para localizar el prefijo correspondiente de mayor longitud; la entrada correspondiente de la tabla de montaje proporcionará el nombre de dispositivo. Este nombre de dispositivo también tiene la forma de un nombre dentro del espacio de nombres del sistema de archivos. Cuando UNIX busca este nombre en las estructuras de directorio del sistema de archivos, lo que encuentra no es un número de inodo, sino un número de dispositivo *<principal, secundario>*. El número de dispositivo principal identifica un controlador de dispositivo al que habrá que llamar para tratar las operaciones de E/S dirigidas a este dispositivo. El número de dispositivo secundario se pasa al controlador de dispositivo como índice para una tabla de dispositivos. La entrada correspondiente de la tabla de dispositivos proporciona la dirección de puerto o la dirección mapeada en memoria del controlador de dispositivo.

Los sistemas operativos modernos obtienen una gran flexibilidad de estas múltiples etapas de tablas de búsqueda dentro de la ruta comprendida entre una solicitud y una controladora física de dispositivo. Los mecanismos que pasan las solicitudes entre las aplicaciones y los controladores son generales, por lo que podemos introducir nuevos dispositivos y controladores en una computadora sin necesidad de recompilar el *kernel*. De hecho, algunos sistemas operativos tienen la capacidad de cargar controladores de dispositivo bajo demanda. En el momento del arranque, el

sistema comprueba los buses hardware para determinar qué dispositivos están presentes y luego carga los controladores necesarios, inmediatamente o cuando sean requeridos por primera vez por una solicitud de E/S.

Vamos a describir ahora el ciclo típico de vida de una solicitud de lectura bloqueante, como se muestra en la Figura 13.13. La figura sugiere que una operación de E/S requiere muchos pasos distintos que consumen, entre todos, un número enorme de ciclos de CPU.

1. Un proceso ejecuta una llamada al sistema bloqueante `read()` dirigida a un descriptor de archivo o a un archivo que se haya abierto anteriormente.

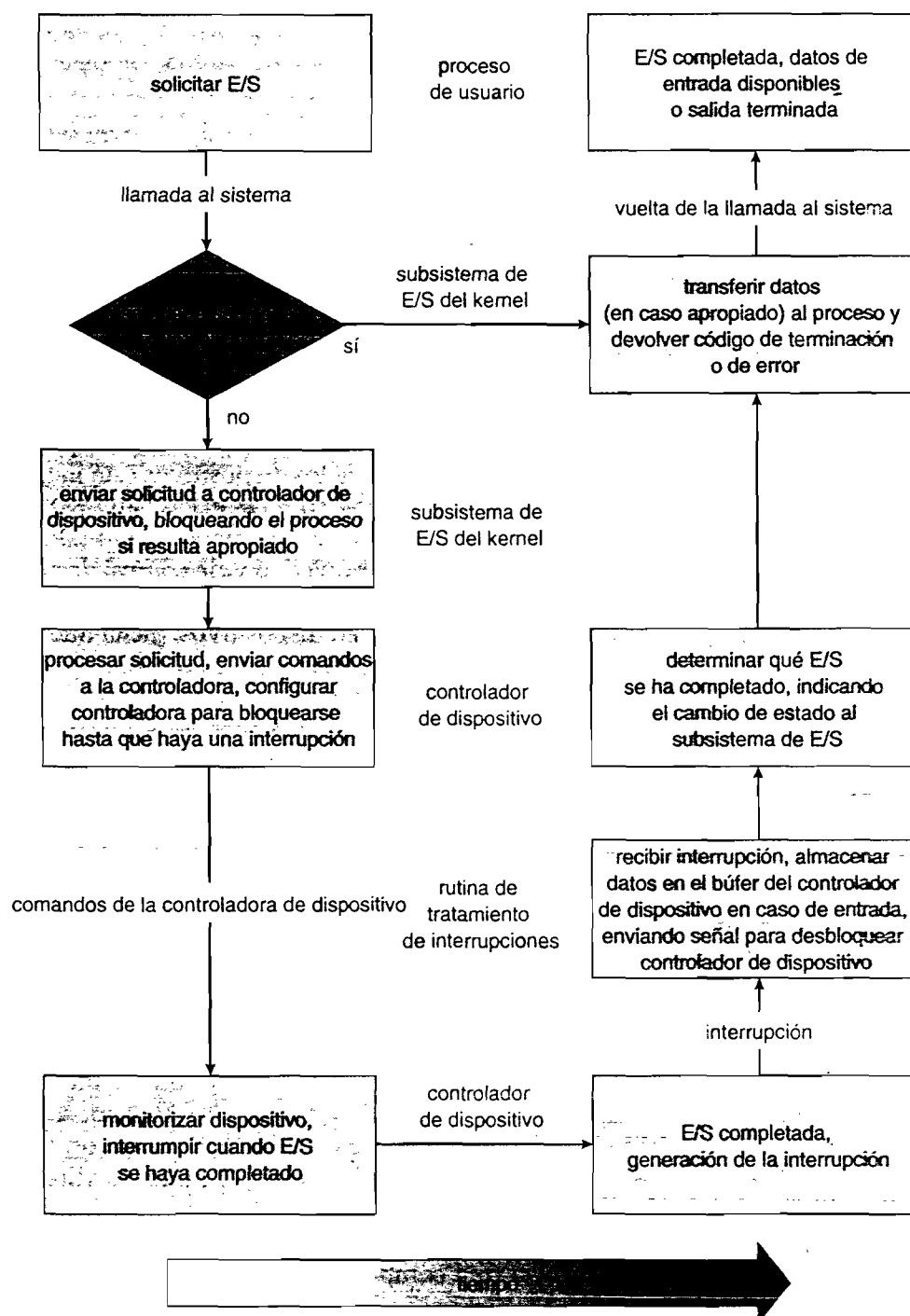


Figura 13.13 Ciclo de vida de una solicitud de E/S.

2. El código de la llamada al sistema en el *kernel* comprueba la corrección de los parámetros. En el caso de una operación de entrada, si los datos ya están disponibles en la caché de búfer, los datos se devuelven en el proceso y se completa la solicitud de E/S.
3. En caso contrario, es necesario realizar una E/S física. El proceso se elimina de la cola de ejecución y se coloca en la cola de espera en el dispositivo, planificándose la solicitud de E/S. Eventualmente, el subsistema de E/S enviará la solicitud al controlador de dispositivo. Dependiendo del sistema operativo, la solicitud se envía mediante una llamada a subrutina o un mensaje interno al *kernel*.
4. El controlador de dispositivo asigna espacio de búfer del *kernel* para recibir los datos y planifica la E/S. Eventualmente, el controlador envía los comandos a la tarjeta controladora de dispositivo escribiendo en los registros de control del dispositivo.
5. La controladora del dispositivo opera el hardware del dispositivo para realizar la transferencia de datos.
6. El controlador puede realizar un sondeo para ver la información de estado y recolectar los datos, o puede haber configurado una transferencia de DMA hacia la memoria del *kernel*. Estamos asumiendo que la transferencia es gestionada por una controladora de DMA, que generará una interrupción cuando la transferencia se complete.
7. La rutina correcta de tratamiento de interrupciones recibirá la interrupción a través de la tabla de vectores de interrupción, almacenará los datos necesarios, efectuará una señalización dirigida al controlador de dispositivo y volverá de la interrupción.
8. El controlador de dispositivo recibe la señal, determina qué solicitud de E/S se ha completado, determina el estado de la solicitud y señaliza al subsistema de E/S del *kernel* que la solicitud se ha completado.
9. El *kernel* transfiere los datos a los códigos de retorno al espacio de direcciones del proceso solicitante y mueve el proceso de la cola de espera a la cola de procesos preparados.
10. Al mover el proceso a la cola de procesos preparados se desbloquea el proceso. Cuando el planificador asigne la CPU al proceso, éste reanudará su ejecución en el punto correspondiente a la terminación de la llamada al sistema.

13.6 Streams

UNIX System V tiene un mecanismo interesante, denominado STREAMS, que permite a una aplicación componer dinámicamente *pipelines* de código controlador de dispositivo. Un *stream* es una conexión full-dúplex entre un controlador de dispositivo y un proceso de nivel de usuario. Consiste de una **cabecera de stream** que efectúa la interfaz con el proceso de usuario, un **extremo de controlador** que controla el dispositivo y cero o más **módulos stream** comprendidos entre ellos. La cabecera de *stream*, el extremo de controlador y cada módulo contienen una pareja de colas: una cola de lectura y una cola de escritura. Se utiliza un mecanismo de paso de mensajes para transferir los datos entre las distintas colas. La estructura STREAMS se muestra en la Figura 13.14.

Los módulos proporcionan la funcionalidad del procesamiento STREAMS; se *insertan* en un *stream* utilizando la llamada al sistema `ioctl()`. Por ejemplo, un proceso puede abrir un dispositivo de puerto serie mediante un *stream* y puede insertar un módulo para gestionar la edición de los datos de entrada. Puesto que los mensajes se intercambian entre las colas situadas en los módulos adyacentes, una cola de un módulo puede desbordar otra cola adyacente. Para evitar que esto ocurra, cada cola puede soportar un mecanismo de **control de flujo**. Sin el control de flujo, una cola aceptará todos los mensajes y los enviará inmediatamente a la cola situada en el módulo adyacente, sin almacenarlos en búfer. Una cola que soporte el mecanismo de control de flujo almacenará en búfer los mensajes y no aceptará mensajes cuando no tenga espacio de búfer suficiente. Este proceso implica intercambiar mensajes de control entre las colas situadas en módulos adyacentes.

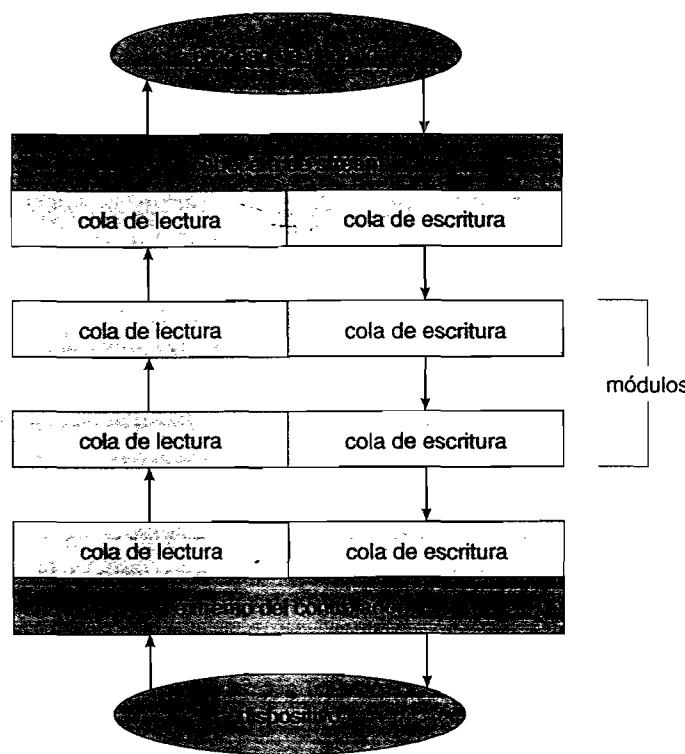


Figura 13.14 La estructura STREAMS.

Un proceso de usuario escribe datos en un dispositivo utilizando la llamada al sistema `write()` o `putmsg()`. La llamada al sistema `write()` escribe datos sin formato en el *stream*, mientras que `putmsg()` permite al proceso de usuario especificar un mensaje. Independientemente de la llamada al sistema utilizada para el proceso de usuario, la cabecera de *stream* copia los datos en un mensaje y entrega éste a la cola correspondiente al siguiente módulo de la secuencia. Esta copia de mensaje continúa hasta que el mensaje se copia en el extremo del controlador y, por tanto, en el dispositivo. De forma similar, el proceso de usuario lee datos de la cabecera del *stream* utilizando las llamadas al sistema `read()` o `getmsg()`. Si se utiliza `read()`, la cabecera de *stream* extrae un mensaje de su cola adyacente y devuelve datos normales (un flujo no estructurado de bytes) al proceso. Si se usa `getmsg()`, se devuelve un mensaje al proceso.

La E/S de tipo STREAMS es asíncrona (o no bloqueante), excepto cuando el proceso de usuario se comunica con la cabecera de *stream*. Cuando escribe en el *stream*, el proceso de usuario se bloquea, asumiendo que la siguiente cola utilice un mecanismo de control de flujo, hasta que haya espacio suficiente para copiar el mensaje. De forma similar, el proceso de usuario se bloqueará al leer del *stream*, hasta que haya datos disponibles.

El extremo del controlador es similar a una cabecera de *stream* o a un módulo, en el sentido de que dispone de una cola de lectura y otra de escritura. Sin embargo, el extremo del controlador debe responder a las interrupciones, como por ejemplo la que se genera cuando hay un paquete listo para ser leído de la red. A diferencia de la cabecera de *stream*, que puede bloquearse si no puede copiar un mensaje en la siguiente cola de la secuencia, el extremo del controlador debe gestionar todos los datos entrantes. Los controladores deben soportar también mecanismos de control de flujo. Sin embargo, si el búfer de un dispositivo está lleno, el dispositivo suele recurrir al procedimiento de eliminar los mensajes entrantes. Considere una tarjeta de red cuyo búfer de entrada esté lleno; la tarjeta de red deberá simplemente eliminar los mensajes adicionales hasta que haya suficiente espacio de búfer como para almacenar los mensajes entrantes.

El beneficio de utilizar streams es que proporciona un marco de trabajo con el que se puede implementar una técnica modular e incremental de escritura de controladores de dispositivos y protocolos de red. Los distintos módulos pueden ser usados por streams diferentes y, por tanto, por diferentes dispositivos. Por ejemplo, un módulo de red puede ser usado tanto por una tarjeta

de red Ethernet como por una tarjeta de red token-ring. Además, en lugar de tratar la E/S de los dispositivos orientados a caracteres como un flujo de bytes no estructurado, STREAMS permite añadir soporte para implementar las fronteras entre mensajes y la información de control que deben intercambiarse los módulos. El soporte para STREAMS está muy extendido entre la mayoría de las variantes de UNIX, y es el método preferido para escribir protocolos y controladores de dispositivo. Por ejemplo, UNIX System V y Solaris implementan el mecanismo de *sockets* utilizando STREAMS.

13.7 Rendimiento

La E/S es uno de los factores que más afectan al rendimiento del sistema. Estas operaciones imponen una intensa demanda a la CPU para ejecutar código de los controladores de dispositivo y para planificar los procesos de forma equitativa y eficiente a medida que se bloquean y desbloquean. Los cambios de contexto resultantes imponen una gran carga de trabajo a la CPU y a sus cachés hardware. Las operaciones de E/S también ponen al descubierto cualquier falta de eficiencia que existe en los mecanismos de tratamiento de interrupciones del *kernel*. Además, la E/S carga al bus de memoria durante la copia de datos entre las controladoras y la memoria física y, de nuevo, durante las copias entre los búferes del *kernel* y el espacio de datos de las aplicaciones. Tratar de satisfacer adecuadamente todas esas demandas es una de las principales preocupaciones de los arquitectos de un sistema informático.

Aunque las computadoras modernas pueden gestionar muchos miles de interrupciones por segundo, el tratamiento de interrupciones es una tarea relativamente cara en términos de procesamiento: cada interrupción hace que el sistema realice un cambio de estado, ejecute la rutina de tratamiento de la interrupción y luego restaure el estado. Las operaciones de E/S programadas pueden ser más eficientes que las E/S dirigidas por interrupciones, si el número de ciclos invertidos en las esperas activas no resulta excesivo. La terminación de una operación de E/S hace, típicamente, que se desbloquee un proceso, lo que provoca la sobrecarga asociada al correspondiente cambio de contexto.

El tráfico de red también puede provocar una alta tasa de cambios de contexto. Considere, por ejemplo, un inicio de sesión remoto en una máquina desde otra. Cada carácter escrito en la máquina local debe transportarse hasta la máquina remota. En la máquina local, el carácter se escribe en el teclado, se genera una interrupción de teclado y el carácter se pasa a través de la rutina de tratamiento de interrupción al controlador de dispositivo, al *kernel* y luego al proceso de usuario. El proceso de usuario ejecuta una llamada de E/S de red al sistema para enviar el carácter a la página remota. El carácter fluye entonces hacia el *kernel* local, a través de los niveles de red que construyen un paquete de red y hacia el controlador del dispositivo de red. El controlador del dispositivo de red transfiere el paquete a la tarjeta controladora de red, que envía el carácter y genera una interrupción. La interrupción pasa de nuevo a través del *kernel* para hacer que se complete la llamada de E/S de red al sistema.

Ahora, el hardware de red del sistema remoto recibe el paquete y se genera una interrupción. El carácter se desempaquetá segú los protocolos de red y se entrega al demonio de red apropiado. El demonio de red identifica qué sesión remota es la implicada y pasa el paquete al subdemonio apropiado para dicha sesión. A lo largo de este flujo, se producen cambios de contexto y cambios de estado (Figura 13.15). Usualmente, el receptor devuelve el carácter como eco al transmisor, y dicha operación duplica la cantidad de trabajo que hay que realizar.

Para eliminar los cambios de contextos necesarios para mover cada carácter entre los demonios del *kernel*, los desarrolladores de Solaris reimplementaron el demonio *telnet* utilizando hebras internas al *kernel*. Sun estima que esta mejora permite incrementar el número máximo de inicios de sesión de red desde unos cuantos centenares a unos cuantos miles en un servidor de gran tamaño.

Otros sistemas utilizan procesadores frontales separados para la E/S de terminales con el fin de reducir la carga de interrupciones en la CPU principal. Por ejemplo, un concentrador de terminales puede multiplexar el tráfico de centenares de terminales remotos con un único puerto en una computadora de gran tamaño. Un canal de E/S es una CPU dedicada, de propósito especial,

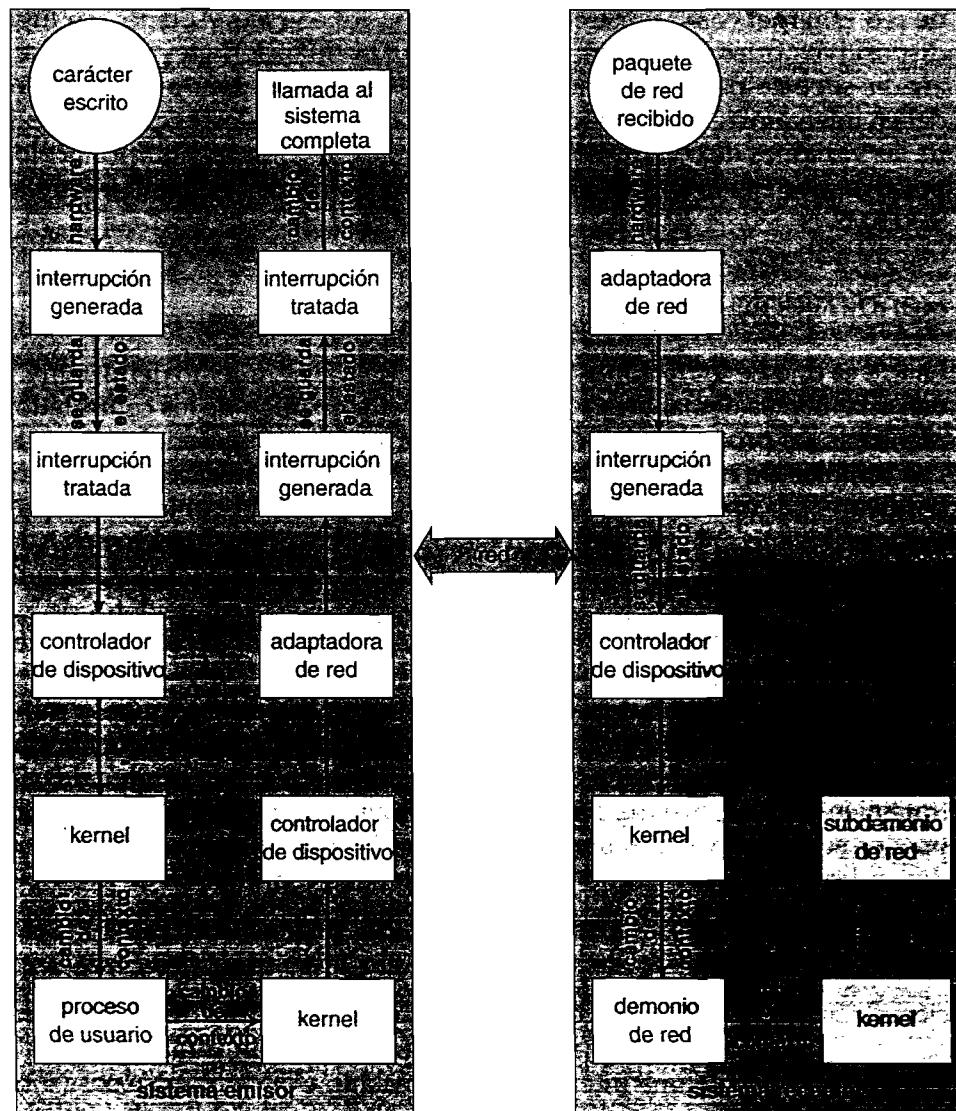


Figura 13.15 Comunicaciones entre computadoras.

que podemos encontrar en las computadoras de tipo mainframe y en otros sistemas de alta gama. La tarea de un canal es descargar el trabajo de E/S de la CPU principal. La idea es que los canales hacen que los datos fluyan de manera suave, mientras que la CPU principal queda libre para procesar los datos. Al igual que las controladoras de dispositivo y las controladoras de DMA que podemos encontrar en las computadoras de menor tamaño, un canal puede procesar programas más generales y sofisticados, de modo que los canales pueden optimizarse para ciertos tipos de cargas de trabajo.

Podemos emplear diversos principios para mejorar la eficiencia de la E/S:

- Reducir el número de cambios de contexto.
- Reducir el número de veces que los datos deben copiarse en memoria mientras pasan desde el dispositivo a la aplicación o viceversa.
- Reducir la frecuencia de las interrupciones utilizando transferencias de gran tamaño, controladoras inteligentes y mecanismos de sondeo (si puede minimizarse la espera activa).
- Incrementar la concurrencia utilizando controladoras preparadas para DMA o canales, con el fin de descargar a la CPU de las operaciones simples de copia de datos.

- Desplazar las primitivas de procesamiento al hardware, para permitir que se ejecuten en las controladoras de dispositivo, de forma concurrente con las operaciones de la CPU y del bus.
- Equilibrar el rendimiento de la CPU, del subsistema de memoria, del bus y de la E/S, porque cualquier sobrecarga en una de esas áreas provocará la aparición de tiempos muertos en las otras.

Los dispositivos varían enormemente en cuanto a complejidad. Por ejemplo, un ratón es simple: los movimientos y los clics de los botones del ratón se convierten en valores numéricos que se pasan desde el hardware, a través del controlador de dispositivo del ratón, hasta la aplicación. Por contraste, la funcionalidad proporcionada por el controlador de dispositivos de disco de Windows NT es muy compleja. No sólo gestiona los discos individuales sino que también implementa matrices RAID (Sección 12.7). Para hacer esto, convierte las solicitudes de lectura o escritura de las aplicaciones en un conjunto coordinado de operaciones de E/S de disco. Además, implementa algoritmos sofisticados de tratamiento de errores y de recuperación y lleva a cabo numerosos pasos para optimizar el rendimiento del disco.

¿Dónde debe implementarse la funcionalidad de E/S, en el hardware del dispositivo, en el controlador del dispositivo o en el software de aplicación? En ocasiones podemos observar la progresión que se ilustra en la Figura 13.16.

- Inicialmente, implementamos los algoritmos de E/S experimentales en el nivel de aplicación, porque el código de aplicación es flexible y los errores de aplicación raramente provocan un fallo catastrófico del sistema. Además, desarrollando el código en el nivel de aplicación, evitamos la necesidad de reiniciar el sistema o recargar controladores de dispositivo después de cada cambio de código. Una implementación de nivel de aplicación puede ser poco eficiente, sin embargo, debido a la carga adicional implicada por los cambios de contexto y debido a que la aplicación no puede aprovechar las estructuras de datos internas al *kernel* y la funcionalidad del *kernel* (como los eficientes mecanismos de intercambio de mensajes internos al *kernel*, los mecanismos multihebra y los mecanismos de bloqueo).
- Cuando un algoritmo de nivel de aplicación ha demostrado su utilidad, podemos reimplementarlo en el *kernel*. Esto puede mejorar el rendimiento, pero el esfuerzo de desarrollo es mucho mayor, porque un *kernel* de un sistema operativo es un sistema software muy complejo y de gran envergadura. Además, las implementaciones internas al *kernel* deben depurarse exhaustivamente para evitar la corrupción de los datos y los fallos catastróficos del sistema.

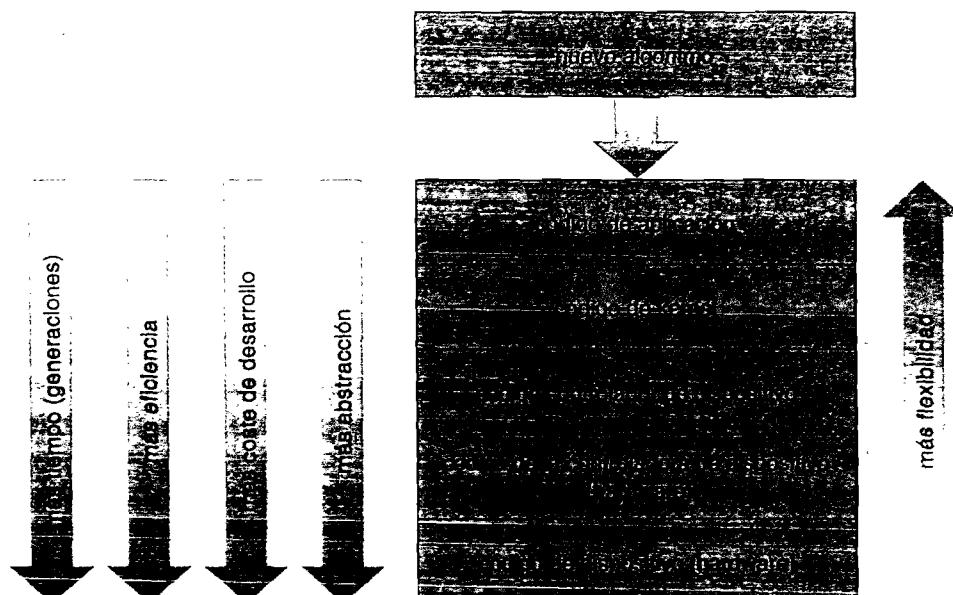


Figura 13.16 Progresión de la funcionalidad de los dispositivos.

- El rendimiento más alto puede obtenerse mediante una implementación especializada en hardware, bien en el dispositivo o en la tarjeta controladora. Las desventajas de una implementación hardware incluyen la dificultad y el coste de realizar mejoras ulteriores o de corregir errores, el tiempo de desarrollo adicional (meses en lugar de días) y la menor flexibilidad. Por ejemplo, una controladora hardware RAID puede no proporcionar ningún medio para que el *kernel* influya en el orden o la ubicación de las lecturas o escrituras de bloques individuales, incluso si el *kernel* tiene información especial acerca de la carga de trabajo que podría permitir al *kernel* mejorar el rendimiento de E/S.

13.8 Resumen

Los elementos hardware básicos implicados en las operaciones de E/S son los buses, las controladoras de dispositivo y los propios dispositivos. La tarea de mover datos entre los dispositivos y la memoria principal es llevada a cabo por la CPU como E/S programada o se descarga en una controladora de DMA. El módulo del *kernel* que controla un dispositivo se denomina controlador de dispositivo. La interfaz de llamadas al sistema que se proporciona a las aplicaciones está diseñada para gestionar varias categorías básicas de hardware, incluyendo dispositivos de bloques, dispositivos orientados a carácter, archivos mapeados en memoria, *sockets* de red y temporizadores de intervalos programables. Las llamadas al sistema bloquean usualmente al proceso que las emite, pero el propio *kernel* y algunas aplicaciones que no pueden quedar inactivas mientras están esperando a que una operación de E/S se complete pueden utilizar llamadas no bloqueantes y llamadas asíncronas.

El subsistema de E/S del *kernel* proporciona numerosos servicios. Entre estos podemos citar la planificación de E/S, el almacenamiento en búfer, el almacenamiento en caché, la gestión de colas de impresión, la reserva de dispositivos y el tratamiento de errores. Otro servicio, el de traducción de nombres, realiza la conexión entre los dispositivos hardware y los nombres de archivos simbólicos usados por las aplicaciones. Este servicio implica varios niveles de mapeo que efectúan la traducción entre los nombres formados por cadenas de caracteres y las direcciones de dispositivo y controladores de dispositivo específicos, así como la traducción de éstos a las direcciones físicas de los puertos de E/S y de las controladoras de bus. Este mapeo puede llevarse a cabo dentro del espacio de nombres del sistema de archivos, como sucede en UNIX, o en un espacio de nombres de dispositivo separado, como sucede en MS-DOS.

STREAMS es una implementación y una metodología para hacer que los controladores sean reutilizables y fáciles de emplear. Con este mecanismo, se pueden apilar los controladores, pasando los datos a través de ellos de forma secuencial y bidireccional para su procesamiento.

Las llamadas de E/S al sistema son costosas en términos de tiempo de procesador, debido a la gran cantidad de niveles de software existentes entre un dispositivo físico y la aplicación. Estos niveles imponen la sobrecarga de los cambios de contexto necesarios para cruzar la frontera de protección del *kernel*, así como la sobrecarga derivada del tratamiento de señales e interrupciones necesario para dar servicio a los dispositivos de E/S y la carga adicional de la CPU y del sistema de memoria que se requiere para copiar datos entre los búferes del *kernel* y el espacio de la aplicación.

Ejercicios

- 13.1 Cuando aparecen múltiples interrupciones de diferentes dispositivos aproximadamente al mismo tiempo, puede utilizarse un esquema de prioridades para determinar el orden en que se debe dar servicio a las distintas interrupciones. Analice las cuestiones que habrá que tomar en consideración a la hora de asignar prioridades a las diferentes interrupciones.
- 13.2 ¿Cuáles son las ventajas y desventajas de soportar un mecanismo de E/S mapeado en memoria para los registros de control de los dispositivos?
- 13.3 Considere los siguientes escenarios de E/S en un PC monousouario:

Milenkovic [1987] analiza la complejidad de los métodos de E/S y de su implementación. El uso y la programación de los diversos mecanismos de comunicación interprocesos y protocolos de red en UNIX se explora en Stevens [1992]. Brain [1996] documenta la interfaz de aplicaciones de Windows NT. La implementación de E/S en el sistema operativo MINIX de ejemplo se describe en Tanenbaum y Woodhull [1997]. Custer [1994] incluye información detallada sobre la implementación de la E/S mediante paso de mensajes en NT.

Para obtener más detalles sobre el tratamiento de la E/S en el nivel hardware y la funcionalidad de mapeo de memoria, las fuentes más adecuadas son los manuales de referencia de los procesadores (Motorola [1993] e Intel [1993]). Hennessy y Patterson [2002] describe las cuestiones relativas a los sistemas multiprocesador y a los problemas de coherencia de caché.. Tanenbaum [1990] describe el diseño de hardware de E/S a bajo nivel y Sargent y Shoemaker [1995] proporcionan una guía de programación para el hardware y software de bajo nivel de un PC. El número de marzo de 1994 de *IEEE Computer* está dedicado al hardware y software avanzado de E/S. Rago [1993] realiza un buen análisis de STREAMS.

Parte Cinco

Protección y seguridad

Los mecanismos de protección controlan el acceso a un sistema limitando los tipos de acceso a archivos permitidos a los usuarios. Además, los mecanismos de protección deben garantizar que sólo los procesos que hayan obtenido la adecuada autorización del sistema operativo puedan operar sobre los segmentos de memoria, la CPU y otros recursos.

La protección se proporciona mediante un mecanismo que controla el acceso de los programas, de los procesos o de los usuarios a los recursos definidos por un sistema informático. Este mecanismo debe proporcionar un medio para especificar los controles que hay que imponer, junto con algún modo de imponerlos.

La seguridad garantiza la autenticación de los usuarios del sistema, con el fin de proteger la integridad de la información almacenada en el mismo (tanto datos como código), así como la de los recursos físicos del sistema informático. El sistema de seguridad impide los accesos no autorizados, la destrucción o manipulación maliciosas de los datos y la introducción accidental de incoherencias.



Protección

Los procesos en un sistema operativo deben protegerse de las actividades realizadas por otros procesos. Para proporcionar dicha protección, podemos utilizar diversos mecanismos para garantizar que sólo los procesos que hayan obtenido la adecuada autorización del sistema operativo puedan operar sobre los archivos, los segmentos de memoria, sobre la CPU y sobre otros recursos del sistema.

El concepto de protección hace referencia a un mecanismo para controlar el acceso de los programas, de los procesos o de los usuarios a los recursos definidos por el sistema informático. Este mecanismo debe proporcionar un medio de especificar los controles que hay que imponer, junto con un modo de imponerlos. Podemos distinguir entre los conceptos de protección y seguridad; este último es una medida de la confianza en que se puedan preservar la integridad de un sistema y de sus datos. La garantía de seguridad es un tema mucho más amplio que la protección, y la analizaremos en el Capítulo 15.

OBJETIVOS DEL CAPÍTULO

- Analizar los objetivos y principios de la protección en un sistema informático moderno.
- Explicar cómo se utilizan los dominios de protección junto con una matriz de acceso para especificar los recursos a los que un proceso puede acceder.
- Examinar los sistemas de protección basados en capacidades y basados en el lenguaje.

14.1 Objetivos de la protección

A medida que los sistemas informáticos se han hecho más sofisticados y a medida que su rango de aplicaciones se ha ido incrementando, también ha crecido la necesidad de proteger la integridad de esos sistemas. La protección se concebía originalmente como algo asociado a los sistemas operativos multiprogramados, de modo que los usuarios que no fueran de confianza pudieran compartir de manera segura un espacio lógico de nombres común, como por ejemplo un directorio de archivos, o compartir un espacio físico de nombres común, como por ejemplo la memoria. Los conceptos modernos de protección han evolucionado para incrementar la fiabilidad de cualquier sistema complejo que haga uso de recursos compartidos.

Necesitamos proporcionar protección por diversas razones. La más obvia es la necesidad de impedir una violación maliciosa e intencionada de una restricción de acceso por parte de un usuario. Sin embargo, tiene una mayor importancia general la necesidad de garantizar que cada componente de programa activo en un sistema utilice los recursos del sistema sólo en ciertas formas

que sean coherentes con las políticas establecidas. Este requerimiento tiene un carácter primordial si se quiere disponer de un sistema fiable.

Los mecanismos de protección pueden mejorar la fiabilidad detectando los errores latentes en las interfaces definidas entre los distintos subsistemas componentes. La detección temprana de errores de interfaz puede a menudo impedir que un subsistema correcto se vea contaminado por otro que no esté funcionando adecuadamente. Un recurso no protegido no puede defenderse frente al uso (o mal uso) por parte de un usuario no autorizado o incompetente. Un sistema orientado a la protección proporcionará medios para distinguir entre el uso autorizado y el no autorizado.

El papel de la protección en un sistema informático es proporcionar un mecanismo para la imposición de las políticas que gobiernen el uso de recursos. Estas políticas pueden establecerse de diversas formas. Algunas están fijas en el diseño de un sistema, mientras que otras se formulan al administrar ese sistema. Existen también otras que son definidas por los usuarios individuales para proteger sus propios archivos y programas. Un sistema de protección deberá tener la flexibilidad suficiente para poder imponer una diversidad de políticas.

Las políticas de uso de recursos pueden variar según la aplicación y también pueden variar a lo largo del tiempo. Por estas razones, la protección no es sólo cuestión del diseñador de un sistema operativo. El programador de aplicaciones necesita utilizar también los mecanismos de protección, para defender de un uso incorrecto los recursos creados y soportados por un subsistema de aplicación. En este capítulo, describiremos los mecanismos de protección que el sistema operativo debe proporcionar para que los diseñadores de aplicaciones puedan usarlos a la hora de diseñar su propio software de protección.

Observe que los *mecanismos* son distintos de las *políticas*. Los mecanismos determinan *cómo* se llevará algo a cabo; las políticas deciden *qué* es lo que hay que hacer. La separación entre políticas y mecanismos resulta importante si queremos tener una cierta flexibilidad. Es probable que las políticas cambien de un lugar a otro o a lo largo del tiempo. En el caso peor, cada cambio de política requeriría un cambio en el mecanismo subyacente; la utilización de mecanismos generales nos permite evitar este tipo de situaciones.

14.2 Principios de la protección

Frecuentemente, podemos utilizar un principio director a lo largo de un proyecto, como pueda ser el diseño de un sistema operativo. Ajustarnos a este principio simplifica las decisiones de diseño y hace que el sistema continúe siendo coherente y fácil de comprender. Uno de los principios directores clave y que ha resistido al paso del tiempo a la hora de proporcionar protección es el **principio del mínimo privilegio**. Este principio dicta que a los programas, a los usuarios, incluso a los sistemas se les concedan únicamente los suficientes privilegios para llevar a cabo sus tareas.

Considere la analogía de un guardia de seguridad que dispusiera de una tarjeta magnética. Si esta tarjeta le permite entrar simplemente en las áreas públicas que está vigilando, un mal uso de esa tarjeta provocará un daño mínimo. Sin embargo, si esa tarjeta permite acceder a todas las áreas, el daño derivado del robo, la pérdida, el mal uso, la copia u otro tipo de actividad comprometida realizada con la tarjeta será mucho mayor.

Un sistema operativo que se ajuste al principio del mínimo privilegio implementará sus características, programas, llamadas al sistema y estructuras de datos de modo que el fallo o el compromiso de un componente provoquen un daño mínimo y no permitan realizar más que un daño mínimo. El desbordamiento de un búfer en un demonio del sistema puede hacer, como por ejemplo, que el demonio falle, pero no debería permitir la ejecución de código contenido en la pila del proceso que permitiera a un usuario remoto obtener privilegios máximos y acceder al sistema completo (como sucede de forma demasiado frecuente hoy en día).

Dicho sistema operativo también proporcionará llamadas al sistema y servicios que permitan escribir aplicaciones con controles de acceso de granularidad fina. Proporcionará mecanismos para activar los privilegios cuando sean necesarios y desactivarlos cuando dejan de serlo. También resulta ventajosa la creación de pistas de auditoría para todos los accesos a funciones privi-

legiadas. La pista de auditoría permite al programador, al administrador del sistema o a los miembros de las fuerzas del orden revisar todas las actividades realizadas en el sistema que estén relacionadas con los mecanismos de protección y de seguridad.

La gestión de los usuarios con el principio del mínimo privilegio implica crear una cuenta separada para cada usuario, con sólo los privilegios que ese usuario necesite. Un operador que necesite montar cintas y realizar copias de seguridad de archivos del sistema tendrá sólo acceso a esos comandos y archivos que necesita para llevar a cabo su tarea. Algunos sistemas implementan mecanismos de control de acceso basado en roles (RBAC, role-based access control) para proporcionar esta funcionalidad.

Las computadoras implementadas dentro de una instalación que se ajuste al principio del mínimo privilegio pueden limitarse a ejecutar servicios específicos, a acceder a máquinas *host* remotas a través de servicios específicos y a hacer esto sólo durante períodos específicos. Normalmente, estas restricciones se implementan activando y desactivando cada servicio y mediante listas de control de acceso, como se describe en las Secciones 10.6.2 y 14.6.

El principio de mínimo privilegio puede ayudar a obtener un entorno informático más seguro. Desafortunadamente, con frecuencia no lo hace. Por ejemplo, Windows 2000 tiene integrado un complejo esquema de protección, a pesar de lo cual tiene muchos agujeros de seguridad. Por comparación, Solaris se considera relativamente seguro, aunque es una variante de UNIX, que históricamente se diseñó teniendo muy poco presentes los mecanismos de protección. Una de las razones para esta diferencia puede ser que Windows 2000 tiene más líneas de código y más servicios que Solaris, por lo que tiene más componentes a los que dotar de seguridad y proteger. Otra razón podría ser que el esquema de protección de Windows 2000 es incompleto o protege los aspectos incorrectos del sistema operativo, dejando otras áreas vulnerables.

14.3 Dominio de protección

Un sistema informático es una colección de procesos y objetos. Por *objetos* queremos definir tanto **objetos hardware** (como la CPU, los segmentos de memoria, las impresoras, los discos y las unidades de cinta) y **objetos software** (como archivos, programas y semáforos). Cada objeto tiene un nombre distintivo que lo diferencia de todos los demás objetos del sistema y sólo se puede acceder a cada objeto mediante operaciones significativas bien definidas. Los objetos son, esencialmente, tipos abstractos de datos.

Las operaciones posibles pueden depender de cada objeto. Por ejemplo, en una CPU lo único que se puede hacer es ejecutar código. En los segmentos de memoria se puede leer y escribir, mientras que en un CD-ROM o un DVD-ROM sólo puede leerse. Las unidades de cinta pueden ser leídas, escritas y rebobinadas. Los archivos de datos pueden crearse, abrirse, leerse, escribirse, cerrarse y borrarse; los archivos de programa pueden leerse, escribirse, ejecutarse y borrarse.

A un proceso sólo se le debe permitir acceder a aquellos recursos para los que tenga autorización. Además, en cualquier instante determinado, un proceso sólo debería poder acceder a aquellos recursos que necesite actualmente para completar su tarea. Este segundo requisito, al que comúnmente se denomina principio de la *necesidad de conocer*, resulta útil a la hora de limitar la cantidad de daño que un proceso erróneo pueda provocar en el sistema. Por ejemplo, cuando el proceso *p* invoca el procedimiento *A()*, el procedimiento sólo debe poder acceder a sus propias variables y a los parámetros formales que se le hayan pasado; no debería poder acceder a todas las variables del proceso *p*. De forma similar, considere el caso en que el proceso *p* invoca un compilador para compilar un archivo concreto. El compilador no debería poder acceder a archivos arbitrariamente, sino que sólo debería acceder a un subconjunto bien definido de los archivos (como por ejemplo el archivo fuente, el archivo de listado, etc.) relacionados con el archivo que hay que compilar. A la inversa, el compilador puede tener archivos privados que utilice para propósitos de contabilización o de optimización y a los que el proceso *p* no debería poder acceder. El principio de la necesidad de conocer es similar al principio del mínimo privilegio del que hemos hablado en la Sección 14.2, en el sentido de que los objetivos de los mecanismos de protección son minimizar los riesgos de las posibles violaciones de seguridad.

14.3.1 Estructura de dominios

Para facilitar este esquema, un proceso opera dentro de un **dominio de protección**, que especifica los recursos a los que el proceso puede acceder. Cada dominio define un conjunto de objetos y los tipos de operaciones que pueden invocarse sobre cada objeto. La capacidad de ejecutar una operación sobre un objeto es un **derecho de acceso**. Un dominio es una colección de derechos de acceso, cada uno de los cuales es una pareja ordenada $\langle \text{nombre-objeto}, \text{conjunto-derechos} \rangle$. Por ejemplo, si el dominio D tiene el derecho de acceso $\langle \text{archivo } F, \{\text{read,write}\} \rangle$, entonces un proceso que se ejecute en el dominio D podrá leer y escribir el archivo F ; sin embargo, no podrá realizar ninguna otra operación sobre ese objeto.

Los dominios no tienen por qué ser disjuntos, sino que pueden compartir derechos de acceso. Por ejemplo, en la Figura 14.1, tenemos tres dominios: D_1 , D_2 y D_3 . El derecho de acceso $\langle O_4, \{\text{print}\} \rangle$ es compartido por D_2 y D_3 , lo que implica que un proceso que se ejecute en cualquiera de estos dos dominios podrá imprimir el objeto O_4 . Observe que un proceso deberá ejecutarse en el dominio D_1 para leer y escribir el objeto O_1 , mientras que sólo los procesos en el dominio D_3 pueden ejecutar el objeto O_1 .

La asociación entre un proceso y un dominio puede ser **estática**, si el conjunto de recursos disponibles para el proceso está fijo durante la vida del proceso, o **dinámica**. Como cabría esperar, establecer dominios dinámicos de protección es más complicado que establecer dominios estáticos de protección.

Si la asociación entre los procesos y los dominios es fija y queremos adherirnos al principio de la necesidad de conocer, deberá haber disponible un mecanismo para cambiar el contenido de un dominio. La razón deriva del hecho de que un proceso puede ejecutarse en dos fases distintas y puede, por ejemplo, necesitar acceso de lectura en una fase y acceso de escritura en la otra. Si un dominio es estático, deberemos definir el dominio para que incluya tanto el acceso de lectura como el de escritura; sin embargo, esta disposición proporciona más derechos de los necesarios en cada una de las dos fases, ya que tenemos acceso de lectura en la fase donde sólo necesitamos acceso de escritura, y viceversa. Por tanto, se violaría el principio de la necesidad de conocer. Debemos permitir que se modifique el contenido de un dominio para que siempre refleje el mínimo necesario de derechos de acceso.

Si la asociación es dinámica, habrá disponible un mecanismo para permitir la **comutación de dominio**, permitiendo al proceso comutar de un dominio a otro. También podemos permitir que se modifique el contenido de un dominio. Si no podemos cambiar el contenido de un dominio, podemos proporcionar el mismo efecto creando un nuevo dominio con el contenido modificado y comutando a este nuevo dominio cuando queramos cambiar el contenido del dominio.

Un dominio puede llevarse a la práctica de diversas formas:

- Cada *usuario* puede ser un dominio. En este caso, el conjunto de objetos a los que se podrá acceder dependerá de la identidad del usuario. La comutación de dominios tiene lugar cuando cambia el usuario, es decir, generalmente cuando un usuario cierre la sesión y otro usuario la inicie.
- Cada *proceso* puede ser un dominio. En este caso, el conjunto de objetos a los que se podrá acceder dependerá de la identidad del proceso. La comutación de dominio tendrá lugar cuando un proceso envíe un mensaje a otro proceso y espere una respuesta.

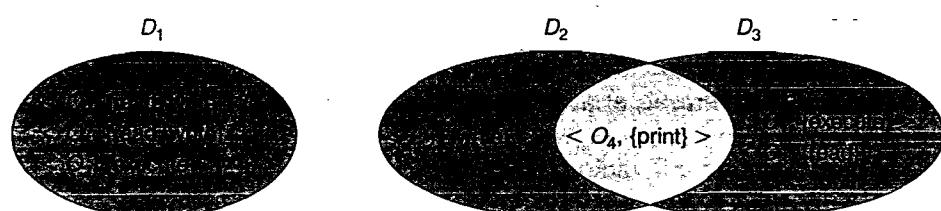


Figura 14.1 Sistema con tres dominios de protección.

- Cada *procedimiento* puede ser un dominio. En este caso, el conjunto de objetos a los que se podrá acceder se corresponderá con las variables locales definidas dentro del procedimiento. La conmutación de dominios sólo se produce cuando se lleva a cabo una llamada a procedimiento.

Hablaremos de la conmutación de dominios con más detalle en la Sección 14.4.

Considere el modelo estándar de modo dual (modo monitor-usuario) de la ejecución de sistemas operativos. Cuando un proceso se ejecuta en modo monitor, puede ejecutar instrucciones privilegiadas y obtener así un control completo del sistema informático. Por contraste, cuando un proceso se ejecuta en modo usuario, sólo puede invocar instrucciones no privilegiadas. En consecuencia, sólo se podrá ejecutar dentro de su espacio de memoria predefinido. Estos dos modos protegen al sistema operativo (que se ejecuta en el dominio monitor) de los procesos de usuario (que se ejecutan en el dominio de usuario). En un sistema operativo multiprogramado, dos dominios de protección son insuficientes, ya que cada usuario quiere también estar protegido frente a los otros usuarios. Por tanto, necesitamos un esquema más elaborado. Vamos a ilustrar dicho tipo de esquemas examinando dos sistemas operativos importantes: UNIX y MULTICS, para ver el modo en que se han implementado en ellos estos conceptos.

14.3.2 Un ejemplo: UNIX

En el sistema operativo UNIX, un dominio está asociado con el usuario. La conmutación de dominio se corresponde con un cambio temporal en la identificación del usuario. Este cambio se lleva a cabo a través del sistema de archivos de la forma siguiente: con cada archivo hay asociada una identificación de propietario y un bit de dominio (conocido como bit *setuid*); cuando el bit *setuid* está *activado* y un usuario ejecuta dicho archivo, el ID de usuario se configura con el valor correspondiente al propietario del archivo; sin embargo, cuando el bit está *desactivado*, el ID de usuario no se modifica. Por ejemplo, cuando un usuario A (es decir, un usuario con *userID* = A) comienza a ejecutar un archivo propiedad de B, cuyo bit de dominio asociado está *desactivado*, el valor *userID* del proceso se configura con el valor A. Cuando el bit *setuid* está *activado*, el valor *userID* se configura de modo que se corresponda con el propietario del archivo: B. Cuando el proceso termina, se da por finalizado este cambio temporal del valor de *userID*.

También se utilizan otros métodos para cambiar los dominios en aquellos sistemas operativos en los que se utilizan las identidades de los usuarios para la definición de dominios, porque casi todos los sistemas necesitan proporcionar dicho tipo de mecanismo. Este mecanismo se utiliza cuando algún recurso que sea privilegiado con carácter general tenga que ser puesto a disposición de la población general de usuarios. Por ejemplo, puede ser deseable permitir a los usuarios acceder a una red sin que tengan que escribir sus propios programas de interconexión por red. En dicho caso, en un sistema UNIX, activaríamos el bit *setuid* de un programa de comunicación por red, haciendo que el ID de usuario cambie cuando se ejecute el programa. El ID de usuario cambiará para corresponderse con el de un usuario que tenga el privilegio de acceder a la red (como por ejemplo *root*, el ID de usuario más potente). Un problema con este método es que si un usuario consigue crear un archivo con el ID de usuario *root* y con el bit *setuid* *activado*, dicho usuario podrá convertirse en *root* y hacer lo que quiera en el sistema. El mecanismo *setuid* se analiza con más detalle en el Apéndice A.

Una alternativa a este método que se utiliza en otros sistemas operativos consiste en situar los programas privilegiados en un directorio especial. El sistema operativo se diseña para modificar el ID de usuario de cualquier programa que se ejecute desde este directorio, bien para asignarle un valor igual al equivalente de *root* o el ID de usuario del propietario del directorio. Esto elimina un problema de seguridad, que es el de los programas *setuid* creados y ocultados (utilizando nombres extraños de archivos o directorios) por los piratas informáticos para su uso posterior. Sin embargo, este método es menos flexible que el que se utiliza en UNIX.

Todavía más restrictivos, y por tanto más protectores, son los sistemas que simplemente no permiten cambiar el ID de usuario. En estos casos, deben utilizarse técnicas especiales para permitir a los usuarios acceder a recursos privilegiados. Por ejemplo, puede iniciarse durante el arranque del sistema un *proceso demonio* y ejecutarse como un ID de usuario especial. Los usuarios

ejecutan entonces un programa independiente, que envía solicitudes a este proceso cada vez que necesiten usar el recurso privilegiado. Este método se utiliza en el sistema operativo TOPS-20.

En cualquiera de estos sistemas, debe tenerse un gran cuidado a la hora de escribir los programas privilegiados. Cualquier descuido puede provocar una total falta de protección en el sistema. Generalmente, estos programas son los primeros en ser atacados por aquellas personas que tratan de irrumpir dentro de un sistema; desafortunadamente, los atacantes tienen éxito en muchas ocasiones. Por ejemplo, la seguridad se ha visto rota en muchos sistemas UNIX debido a la funcionalidad setuid. Hablaremos en detalle de los temas de seguridad en el Capítulo 15.

14.3.3 Un ejemplo: MULTICS

En el sistema MULTICS, los dominios de protección están organizados jerárquicamente en una estructura de anillos concéntricos. Cada anillo se corresponde con un único dominio (Figura 14.2). Los anillos están numerados de 0 a 7. Sean D_i y D_j dos anillos de dominio cualquiera. Si $j < i$, entonces D_i es un subconjunto de D_j , es decir, un proceso que se ejecute en el dominio D_j tiene más privilegios que otro que se ejecute en el dominio D_i . Un proceso que se ejecute en el dominio D_0 será el que tenga más privilegios. Si sólo existen dos anillos, este esquema es equivalente al modo monitor-usuario de ejecución, donde el modo monitor corresponderá a D_0 y el modo usuario corresponderá a D_1 .

MULTICS tiene un espacio de direcciones segmentado; cada segmento es un archivo y cada segmento está asociado con uno de los anillos. La descripción de un segmento incluye una entrada que identifica el número de anillo. Además, incluye tres bits de acceso para controlar la lectura, la escritura y la ejecución. La asociación entre segmentos y anillos es una decisión de política en la que aquí no estamos interesados.

Con cada proceso hay asociado un contador de *número actual de anillo*, que identifica el anillo en el que se está ejecutando el proceso actualmente. Cuando un proceso se está ejecutando en el anillo i , no puede acceder a un segmento asociado con el anillo j ($j < i$), pero sí podrá acceder a un segmento asociado con el anillo k ($k \geq i$). Sin embargo, el tipo de acceso estará restringido de acuerdo con los bits de acceso asociados con ese segmento.

La conmutación de dominios en MULTICS tiene lugar cuando un proceso cruza de un anillo a otro invocando un procedimiento situado en un anillo diferente. Obviamente, esta conmutación debe realizarse de forma controlada; en caso contrario, un proceso podría comenzar a ejecutar código en el anillo 0 y no se proporcionaría ninguna protección. Para permitir una conmutación controlada de dominio, modificamos el campo de anillo del descriptor de segmento con el fin de incluir la siguiente información:

- **Fronteras de acceso.** Una pareja de enteros, $b1$ y $b2$, tal que $b1 \leq b2$.

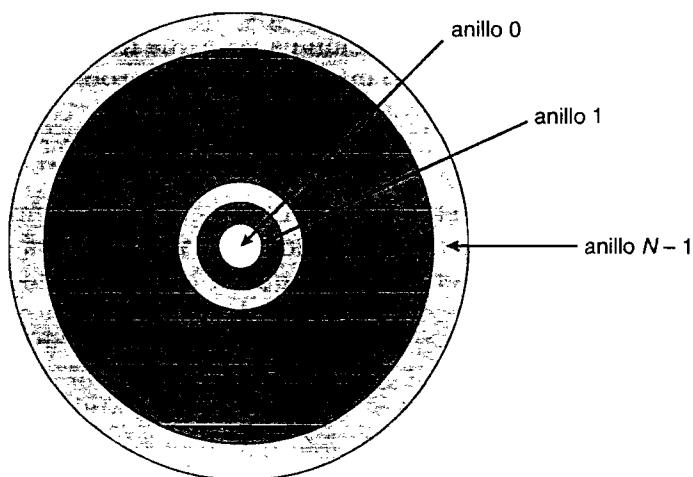


Figura 14.2 Estructura de anillos de MULTICS.

- **Límite.** Un entero b_3 tal que $b_3 > b_2$.
- **Lista de puertas.** Identifica los puntos de entrada (o puertas) en los que pueden invocarse los segmentos.

Si un proceso que está ejecutando en el anillo i invoca un procedimiento (o segmento) con fronteras de acceso (b_1, b_2), entonces la llamada será permitida si $b_1 \leq i \leq b_2$, y el número actual de anillo del proceso seguirá siendo i . En caso contrario, se produce una interrupción software dirigida al sistema operativo y la situación se trata de la forma siguiente:

- Si $i < b_1$, entonces, se permite que tenga lugar la llamada, porque tenemos una transferencia a un anillo (o dominio) con menores privilegios. Sin embargo, si se pasan parámetros que hagan referencia a segmentos situados en un anillo menor (es decir, segmentos no accesibles para el procedimiento invocado), entonces estos segmentos deberán copiarse en un área a la que sí pueda acceder el procedimiento invocado.
- Si $i > b_2$, entonces sólo se permite que se produzca la llamada si b_3 es mayor o igual que i y la llamada ha sido dirigida a uno de los puntos de entrada designados en la lista de puertas. Este esquema permite que procesos con derechos de acceso limitados invoquen procedimientos en los anillos inferiores que tengan más derechos de acceso, pero sólo de una manera cuidadosamente controlada.

La principal desventaja de la estructura en anillos (o jerárquica) es que no nos permite imponer el principio de la necesidad de conocer. En particular, si un objeto debe estar accesible en el dominio D_j pero no en el dominio D_i , entonces debemos tener $j < i$. Pero este requisito significa que todos los segmentos accesibles en D_i serán también accesibles en D_j .

El sistema de protección de MULTICS es, generalmente, más complejo y menos eficiente que los utilizados en los sistemas operativos actuales. Si el mecanismo de protección interfiere con la facilidad de uso del sistema o reduce significativamente el rendimiento del mismo, habrá que ponderar cuidadosamente su uso poniéndolo en relación con el propósito del sistema. Por ejemplo, conviene tener un sistema de protección complejo en una computadora que vaya a ser utilizada por una universidad para procesar las notas de los estudiantes y que también vaya a ser usada por los estudiantes para sus trabajos de curso. Sin embargo, un esquema de protección similar no sería adecuado para una computadora utilizada para cálculo masivo, en la que la velocidad de proceso es el criterio más importante. Convienen separar el mecanismo de la política de protección, para permitir que un mismo sistema tenga una protección compleja o simple posible dependiendo de las necesidades de sus usuarios. Para separar el mecanismo de la política, necesitamos un modelo de protección más general.

14.4 Matriz de acceso

Nuestro modelo de protección puede contemplarse de forma abstracta como una matriz, denominada **matriz de acceso**. Las filas de la matriz de acceso representan dominios y las columnas representan objetos. Cada entrada de la matriz está compuesta de un conjunto de derechos de acceso. Puesto que la columna define los objetos explícitamente, podemos omitir el nombre del objeto del derecho de acceso. La entrada $\text{access}(i,j)$ define el conjunto de operaciones que un proceso que se ejecute en el dominio D_i puede invocar sobre el objeto O_j .

Para ilustrar estos conceptos, vamos a considerar la matriz de acceso mostrada en la Figura 14.3. Hay cuatro dominios y cuatro objetos: tres archivos (F_1, F_2, F_3) y una impresora láser. Un proceso que se ejecute en el dominio D_1 podrá leer los archivos F_1 y F_3 . Un proceso que se ejecute en el dominio D_4 tendrá los mismos privilegios que otro que se ejecute en el dominio D_1 , pero además también podrá escribir en los archivos F_1 y F_3 . Observe que a la impresora láser sólo podrán acceder los procesos que se ejecuten en el dominio D_2 .

El esquema basado en matriz de acceso nos proporciona mecanismos para especificar una diversidad de políticas. El mecanismo consiste en implementar la matriz de acceso y garantizar que las propiedades semánticas que hemos esbozado se cumplan. Más específicamente, debemos

dominio	objeto	F_1	F_2	F_3	impresora

Figura 14.3 Matriz de acceso.

garantizar que un proceso que se ejecute en el dominio D_i sólo pueda acceder a los objetos especificados en la pila i y únicamente según permitan las entradas de la matriz de acceso.

La matriz de acceso puede implementar decisiones de política relativas a la protección. Las decisiones de política implican qué derechos deben incluirse en la entrada (i,j) . También debemos definir el dominio en el que cada proceso se habrá de ejecutar. Esta última política es usualmente definida por el sistema operativo.

Los usuarios normalmente deciden el contenido de las entradas de la matriz de acceso. Cuando un usuario crea un nuevo objeto O_j , se añade la columna O_j a la matriz de acceso, con las apropiadas entradas de inicialización, según determine el creador. El usuario puede decidir introducir algunos derechos en determinadas entradas de la columna j y otros derechos en otras entradas, según sea necesario.

La matriz de acceso proporciona el mecanismo apropiado para definir e implementar un control estricto de la asociación tanto estática como dinámica entre procesos y dominios. Cuando comutamos un proceso de un dominio a otro, ejecutamos una operación (*switch*) sobre un objeto (el dominio). Podemos controlar la comutación de dominios incluyendo los dominios entre los objetos de la matriz de acceso. De forma similar, cuando cambiamos el contenido de la matriz de acceso, realizamos una operación sobre un objeto: la propia matriz de acceso. De nuevo, podemos controlar estos cambios incluyendo la propia matriz de acceso como un objeto. De hecho, puesto que cada entrada de la matriz de acceso puede modificarse individualmente, debemos considerar cada entrada de la matriz de acceso como un objeto que hay que proteger. Ahora, sólo necesitamos considerar las operaciones que sean posibles sobre estos objetos (dominios y matriz de acceso) y decidir cómo queremos que los procesos puedan ejecutar dichas operaciones.

Los procesos deben poder comutar de un dominio a otro. La comutación del dominio D_i al dominio D_j estará permitida si y sólo si el derecho de acceso $\text{switch} \in \text{access}(i,j)$. Por tanto, en la Figura 14.4, un proceso que se ejecute en el dominio D_2 podrá comutar al dominio D_3 o al dominio D_4 . Un proceso en el dominio D_4 podrá comutar al dominio D_1 y un dominio D_1 podrá comutar al dominio D_2 .

Permitir una modificación controlada del contenido de las entradas de la matriz de acceso requiere tres operaciones adicionales: *copy*, *owner* y *control*. Vamos a examinar estas operaciones a continuación.

La capacidad de copiar un derecho de acceso de un dominio (o fila) de la matriz de acceso a otro se denota mediante un asterisco (*) añadido al derecho de acceso. El derecho de *copy* permite copiar el derecho de acceso sólo dentro de la columna (es decir, para el objeto) en la que esté definido el derecho. Por ejemplo, en la Figura 14.5(a), un proceso que se ejecute en el dominio D_2 podrá copiar la operación elegida en cualquier entrada asociada con el archivo F_2 . Por tanto, la matriz de acceso de la Figura 14.5(a) podrá modificarse para obtener la matriz de acceso mostrada en la Figura 14.5(b).

Este esquema tiene dos variantes:

dominio	objeto	F_1	F_2	F_3	Impresor laser	D_1	D_2	D_3	D_4

Figura 14.4 Matriz de acceso de la Figura 14.3 incluyendo los dominios como objetos.

dominio	objeto	F_1	F_2	F_3

(a)

dominio	objeto	F_1	F_2	F_3

(b)

Figura 14.5 Matriz de acceso con derechos de *copia*.

1. Un derecho se copia de $\text{access}(i,j)$ a $\text{access}(k,j)$; y a continuación se elimina de $\text{access}(i,j)$. Esta acción es una *transferencia* de un derecho, en lugar de una copia.
2. La propagación del derecho de *copia* puede estar limitada. Es decir, cuando el derecho R^* se copia de $\text{access}(i,j)$ a $\text{access}(k,j)$, sólo se crea el derecho R (no R^*). Un proceso que se ejecute en el dominio D_k no podrá copiar a su vez el derecho R .

Un sistema puede seleccionar sólo uno de estos tres derechos de *copia*, o puede proporcionar los tres identificándolos como derechos distintos: *copia*, *transferencia* y *copia limitada*.

También necesitamos un mecanismo para permitir la adición de nuevos derechos y la eliminación de algunos derechos. El derecho *owner* controla estas operaciones. Si $\text{access}(i,j)$ incluye el derecho *owner*, entonces un proceso que se ejecute en el dominio D_i podrá añadir y eliminar cualquier derecho en cualquier entrada de la columna j . Por ejemplo, en la Figura 14.6(a), el dominio D_1 es el propietario de F_1 y por tanto puede añadir y borrar cualquier derecho válido en la columna F_1 . De forma similar, el dominio D_2 es el propietario de F_2 y F_3 y puede, por tanto, añadir y eliminar cualquier derecho válido dentro de estas dos columnas. Así, la matriz de acceso de la Figura 14.6(a) puede modificarse para obtener la matriz de acceso mostrada en la Figura 14.6(b).

dominio \ objeto	F_1	F_2	F_3
dominio			

(a)

dominio \ objeto	F_1	F_2	F_3
dominio			

(b)

Figura 14.6 Matriz de acceso con derechos de *proprietario*.

Los derechos de *copia* y el derecho de *proprietario* permiten a un proceso modificar las entradas de una columna. También hace falta un mecanismo para modificar las entradas de una fila. El derecho *control* sólo es aplicable a los objetos dominio. Si $\text{access}(i,j)$ incluye el derecho *control*, entonces un proceso que se ejecute en el dominio D_i puede eliminar cualquier derecho de acceso de la fila j . Por ejemplo, suponga que, en la Figura 14.4, incluimos el derecho *control* en $\text{access}(D_2, D_4)$. Entonces, un proceso que se ejecute en el dominio D_2 podría modificar el dominio D_4 , como se muestra en la Figura 14.7.

Los derechos de *copia* y el derecho de *proprietario* proporcionan un mecanismo para limitar la propagación de derechos de acceso. Sin embargo, no nos proporcionan las herramientas apropiadas para impedir la propagación (o revelación) de información. El problema de garantizar que ninguna información que se almacenara inicialmente en un objeto pueda migrar fuera de su entorno de ejecución se denomina **problema del confinamiento**. Este problema es, en general, irresoluble (véanse las referencias en las Notas bibliográficas).

dominio \ objeto	F_1	F_2	F_3	impres. láser	D_1	D_2	D_3	D_4
dominio								
dominio								
dominio								
dominio								
dominio								
dominio								
dominio								
dominio								

Figura 14.7 Matriz de acceso modificada de la Figura 14.4.

Estas operaciones sobre los dominios y la matriz de acceso no son en sí mismas importantes, pero ilustran la capacidad del modelo de la matriz de acceso para permitir la implementación y control de requisitos de protección dinámicos. Pueden crearse dinámicamente nuevos objetos y nuevos dominios e incluirlos en el modelo de la matriz de acceso. Sin embargo, sólo hemos mostrado que el mecanismo básico existe; los diseñadores del sistema y los usuarios deben tomar las decisiones de política relativas a qué dominios deben poder acceder a qué objetos y en qué manera.

14.5 Implementación de la matriz de acceso

¿Cómo puede implementarse efectivamente la matriz de acceso? En general, la matriz será alguna vez dispersa, es decir, la mayoría de las entradas estarán vacías. Aunque existen técnicas de representación de estructuras de datos para representar matrices dispersas, no resultan particularmente útiles para esta aplicación, debido a la forma en que se utiliza la funcionalidad de protección. En esta sección, vamos a describir primero varios métodos de implementar la matriz de acceso y luego vamos a comparar los distintos métodos.

14.5.1 Tabla global

La implementación más simple de la matriz de acceso es una tabla global compuesta de un conjunto de triplets ordenadas $\langle \text{dominio}, \text{objeto}, \text{conjunto-derechos} \rangle$. Cada vez que se ejecuta una operación M sobre un objeto O_j dentro del dominio D_i , se analiza la tabla global en busca de una triplete $\langle D_i, O_j, R_k \rangle$, con $M \in R_k$. Si se encuentra esta triplete, se permite que la operación continúe; en caso contrario, se genera una condición de excepción (o error).

Esta implementación tiene varias desventajas. La tabla es usualmente muy grande y no puede, por tanto, ser conservada en memoria principal, por lo que hacen falta operaciones adicionales de E/S. A menudo se utilizan técnicas de memoria virtual para gestionar esta tabla. Además, resulta difícil aprovechar las agrupaciones especiales de objetos o dominios. Por ejemplo, si todo el mundo puede leer un objeto concreto, es necesario definir una entrada separada en cada uno de los dominios.

14.5.2 Listas de acceso para los objetos

Cada columna de la matriz de acceso puede implementarse como una lista de acceso de un objeto, como se escribe en la Sección 10.6.2. Obviamente, las entradas vacías pueden descartarse. La lista resultante para cada objeto estará compuesta por una serie de parejas ordenadas $\langle \text{dominio}, \text{conjunto-derechos} \rangle$, que definen todos los dominios que tengan un conjunto de derechos de acceso no vacío para dicho objeto.

Esta técnica puede extenderse fácilmente para definir una lista más un conjunto *predeterminado* de derechos de acceso. Cuando se intenta realizar una operación M sobre un objeto O_j en el dominio D_i , buscamos en la lista de acceso del objeto O_j una entrada $\langle D_i, R_k \rangle$, con $M \in R_k$. Si se encuentra esa entrada, permitimos la operación; en caso contrario, comprobamos el conjunto predeterminado. Si M está en el conjunto predeterminado, permitimos el acceso. Si no lo está, se deniega el acceso y se produce una condición de excepción. Para aumentar la eficiencia, podemos comprobar primero el conjunto predeterminado y luego buscar en la lista de acceso.

14.5.3 Listas de capacidades para los dominios

En lugar de asociar las columnas de la matriz de acceso con los objetos en forma de listas de acceso, podemos asociar cada fila con su dominio. Una lista de capacidades para un dominio es una lista de objetos junto con las operaciones permitidas sobre esos objetos. Cada objeto se suele representar mediante su dirección o nombre físico, denominada **capacidad**. Para ejecutar la operación M sobre el objeto O_j , el proceso ejecuta la operación M especificando la capacidad (o puntero) para

el objeto O_j como parámetro. La simple posesión de la capacidad significa que el acceso está permitido.

La lista de capacidades está asociada con un dominio, pero un proceso que se ejecute en ese dominio no puede nunca acceder directamente a ella. Por el contrario, la lista de capacidades es en sí misma un objeto protegido, mantenido por el sistema operativo y al que el usuario sólo puede acceder indirectamente. El mecanismo de protección basado en capacidades descansa en el hecho de que nunca se permite a las capacidades migrar a ningún espacio de direcciones que sea directamente accesible por parte de un proceso de usuario (donde podrían ser modificadas). Si todas las capacidades son seguras, el objeto al que protegen también estará defendido frente a accesos no autorizados.

Las capacidades se propusieron originalmente como una especie de puntero seguro, para satisfacer la necesidad de protección de los recursos que se preveía que iba a ser necesaria a medida que los sistemas informáticos multiprogramados se generalizaran. La idea de un puntero inherentemente protegido proporciona una base para la protección que puede extenderse hacia el nivel de las aplicaciones.

Para proporcionar una protección inherente, debemos distinguir las capacidades de otros objetos y debemos interpretarlas mediante una máquina abstracta sobre la que se ejecuten los programas de mayor nivel. Las capacidades se suelen distinguir de otros tipos de datos en una de dos formas:

- Cada objeto tiene una **etiqueta** para denotar su tipo, es decir, para indicar si se trata de una capacidad o de un dato accesible. Las propias etiquetas no deben ser directamente accesibles por parte de un programa de aplicación. Puede utilizarse soporte hardware o firmware para imponer esta restricción. Aunque sólo hace falta un bit para distinguir entre capacidades y otros objetos, a menudo se utilizan bits adicionales. Esta extensión permite que el hardware etiquete todos los objetos con su tipo. De este modo, el hardware puede distinguir enteros, números en coma flotante, punteros, valores booleanos, caracteres, instrucciones, capacidades y valores no inicializados, simplemente consultando sus etiquetas.
- Alternativamente, el espacio de direcciones asociado con un programa puede dividirse en dos partes. Una parte es accesible para el programa y contiene las instrucciones y los datos normales del programa. La otra parte, que contiene la lista de capacidades, sólo es accesible por el sistema operativo. Para soportar este mecanismo, resulta útil disponer de un espacio de memoria segmentado (Sección 8.6).

Se han desarrollado diversos sistemas de protección basados en capacidades; describiremos brevemente esos sistemas en la Sección 14.8. El sistema operativo Mach también utiliza una versión del mecanismo de protección basado en capacidades y esa versión se describe en el Apéndice B.

14.5.4 Un mecanismo de bloqueo-clave

El **esquema de bloqueo-clave** es un compromiso entre las listas de acceso y las listas de capacidades. Cada objeto tiene una lista de patrones de bit distintivos, denominados **bloqueos**. De forma similar, cada dominio tiene una lista de patrones de bit distintivos, denominados **claves**. Un proceso que se ejecute dentro de un dominio podrá acceder a un objeto sólo si dicho dominio tiene una clave que se corresponda con uno de los bloques del objeto.

Al igual que con las listas de capacidades, la lista de claves de un dominio debe ser gestionada por el sistema operativo por cuenta del dominio. Los usuarios no están autorizados a examinar o modificar la lista de claves (o de bloques) directamente.

14.5.5 Comparación

Vamos ahora a comparar las diversas técnicas de implementación de las matrices de acceso. La utilización de una tabla global resulta muy simple; sin embargo, la tabla puede ser bastante gran-

de y a menudo no podemos aprovecharnos de las agrupaciones especiales de objetos o dominios. Las listas de acceso se corresponden directamente con las necesidades de los usuarios; cuando un usuario crea un objeto, puede especificar los dominios que pueden acceder, así como las operaciones permitidas. Sin embargo, puesto que la información de derechos de acceso para un dominio concreto no está localizada, resulta difícil determinar el conjunto de derechos de acceso de cada dominio. Además, hay que comprobar cada acceso al objeto, lo que requiere explorar la lista de acceso. En un sistema de gran envergadura con listas de acceso largas, esta búsqueda puede consumir mucho tiempo.

Las listas de capacidades no se corresponden directamente con las necesidades de los usuarios; resultan útiles, sin embargo, para localizar información para un proceso determinado. El proceso que esté intentando realizar el acceso deberá presentar una capacidad para dicho acceso. Entonces, el sistema de protección sólo necesita verificar que la capacidad es válida. La revocación de capacidades, sin embargo, puede resultar poco eficiente (Sección 14.7).

El mecanismo de bloqueo-clave, como hemos mencionado, representa un compromiso entre las listas de acceso y las listas de capacidades. El mecanismo puede ser a la vez efectivo y flexible, dependiendo de la longitud de las claves. Esas claves pueden pasar seguramente de un dominio a otro. Además, los privilegios de acceso pueden revocarse de manera efectiva mediante la simple técnica de cambiar algunos de los bloques asociados con el objeto (Sección 14.7).

La mayoría de los sistemas usan una combinación de listas de acceso y capacidades. Cuando un proceso trata por primera vez de acceder a un objeto, se analiza la lista de acceso. Si se deniega el acceso, se genera una condición de excepción. En caso contrario, se crea una capacidad y se la asocia al proceso. Las referencias adicionales utilizarán la capacidad para demostrar directamente que el acceso está permitido. Después del último acceso, se destruye la capacidad. Esta estrategia se utiliza en el sistema MULTICS y en el sistema CAL.

Como ejemplo del modo en que funciona dicha estrategia, considere un sistema de archivos en el que cada archivo tenga una lista de acceso asociada. Cuando un proceso abre un archivo, se explora la estructura de directorio para encontrar el archivo, se comprueban los permisos de acceso y se asignan los búferes. Toda esta información se registra en una nueva entrada dentro de la tabla de archivos asociada con el proceso. La operación devuelve un índice a esta tabla para el archivo recién abierto. Todas las operaciones sobre el archivo se realizan especificando el índice de la tabla de archivos. La entrada de la tabla de archivos apunta, a su vez, al archivo y a sus búferes. Cuando se cierra el archivo, se borra la entrada de la tabla de archivos. Puesto que la tabla de archivos es mantenida por el sistema operativo, el usuario no puede corromperla accidentalmente. Así, el usuario sólo puede acceder a aquellos archivos que hayan sido abiertos. Puesto que el acceso se comprueba en el momento de abrir el archivo, la protección está garantizada. Esta estrategia se utiliza en el sistema UNIX.

El derecho de acceso *deberá* seguir comprobándose para cada acceso, y la entrada de la tabla de archivos incluye una capacidad únicamente para las operaciones permitidas. Si el archivo se abre para lectura, se coloca una capacidad para acceso de lectura dentro de la entrada de la tabla de archivos. Si se realiza un intento de escribir en el archivo, el sistema identificará esta violación de la protección comparando la operación solicitada con la capacidad incluida en la entrada de la tabla de archivos.

14.6 Control de acceso

En la Sección 10.6.2 hemos descrito cómo pueden usarse los controles de acceso a los archivos dentro de un sistema de archivos. A cada archivo y directorio se le asignan un propietario, un grupo o posiblemente una lista de usuarios y para cada una de estas entidades se asigna una información de control de acceso. Podemos añadir una función similar a otros aspectos de un sistema informático. Un buen ejemplo de esta estrategia es el que podemos encontrar en Solaris 10.

Solaris 10 amplía el sistema de protección disponible en el sistema operativo Sun Microsystems añadiendo explícitamente el principio del mínimo privilegio mediante el **control de acceso basado en roles** (RBAC, role-based access control). Esta funcionalidad gira en torno a los privilegios. Un privilegio es el derecho a ejecutar una llamada al sistema o a usar una opción dentro de dicha

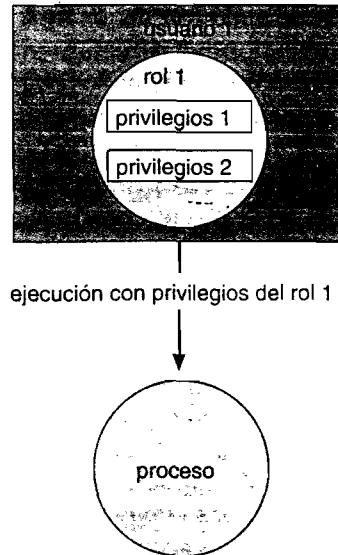


Figura 14.8 Control de acceso basado en roles en Solaris 10.

llamada al sistema (como por ejemplo, abrir un archivo con acceso de escritura). Podemos asignar privilegios a los procesos, limitando éstos a exactamente el tipo de acceso que necesitan para llevar a cabo su tarea. Los privilegios y programas también pueden asignarse a **roles**. A los usuarios se les asignan roles (o los usuarios adoptan roles) basándose en contraseñas asignadas a los roles. De esta forma, un usuario puede adoptar un rol que activa un privilegio, permitiendo al usuario ejecutar un programa para llevar a cabo una tarea específica, como se ilustra en la Figura 14.8. Esta implementación de los privilegios reduce el riesgo de seguridad asociado con los superusuarios y con los programas setuid.

Observe que esta funcionalidad es similar a la matriz de acceso descrita en la Sección 14.4. Exploraremos esta relación con más detalle en los ejercicios del final del capítulo.

14.7 Revocación de derechos de acceso

En un sistema de protección dinámico, puede que necesitemos en ocasiones revocar derechos de acceso a objetos compartidos por diferentes usuarios. En este punto pueden surgir diversas cuestiones acerca de la revocación:

- **Inmediata o diferida.** ¿La revocación tiene lugar inmediatamente o se produce de forma diferida? Si se difiere la revocación, ¿podemos averiguar cuándo tendrá lugar?
- **Selectiva o general.** Cuando se revoca un derecho de acceso a un objeto, ¿afecta a *todos* los usuarios que tienen derecho de acceso a ese objeto o podemos especificar un grupo seleccionado de usuarios cuyos derechos de acceso deben revocarse?
- **Parcial o total.** ¿Podemos revocar un subconjunto de los derechos asociados con un objeto o debemos revocar todos los derechos de acceso al objeto?
- **Temporal o permanente.** ¿Podemos revocar el acceso permanentemente (es decir, el derecho de acceso revocado nunca volverá a estar disponible), o podemos revocar el acceso y luego obtenerlo de nuevo?

Con un esquema basado en lista de acceso, la revocación resulta sencilla. Se analiza la lista de acceso en busca de los derechos de acceso que hay que revocar y se los borra de la lista. La revocación es inmediata y puede ser general o selectiva, total o parcial y permanente o temporal.

Las capacidades, sin embargo, presentan un problema mucho más difícil de cara a la revocación. Puesto que las capacidades están distribuidas por todo el sistema, debemos encontrarlas

antes de poder revocarlas. Entre los esquemas que pueden utilizarse para implementar la revocación en un sistema basado en capacidades podemos citar:

- **Readquisición.** Periódicamente, se borran las capacidades de cada dominio. Si un proceso quiere usar una capacidad, puede que se encuentre con que esa capacidad ha sido borrada. El proceso puede entonces tratar de volver a adquirir la capacidad. Si se ha revocado el acceso, el proceso no será capaz de efectuar esa readquisición.
- **Retropunteros.** Con cada objeto se mantiene una lista de punteros, que hace referencia a todas las capacidades asociadas con ese objeto. Cuando se necesita realizar una revocación, podemos seguir esos punteros, cambiando las capacidades según sea necesario. Este esquema fue adoptado en el sistema MULTICS. Es un sistema muy general, aunque su implementación es costosa.
- **Indirección.** Las capacidades apuntan indirectamente, en lugar de directamente, a los objetos. Cada capacidad apunta a una entrada única dentro de una tabla global, que a su vez apunta al objeto. Implementamos la revocación analizando la tabla global en busca de la entrada deseada y borrándola. Entonces, cuando se intenta realizar un acceso, se verá que la capacidad está apuntando a una entrada ilegal de la tabla. Las entradas de la tabla pueden reutilizarse para otras capacidades sin ninguna dificultad, ya que tanto la capacidad como la entrada de la tabla contienen el nombre distintivo del objeto. El objeto asociado a una capacidad y su entrada de la tabla deben corresponderse. Este esquema fue adoptado en el sistema CAL. El sistema no permite la revocación selectiva.
- **Claves.** Una clave es un patrón distintivo de bits que puede asociarse con una capacidad. Esta clave se define en el momento de crear la capacidad y no puede ser nunca modificada ni inspeccionada por el proceso que posee la capacidad. Con cada objeto hay asociada una clave maestra; esa clave puede definirse o sustituirse mediante la operación `set-key`. Cuando se crea una capacidad, se asocia con esa capacidad el valor actual de la clave maestra. Cuando se ejerce esa capacidad, su clave se compara con la clave maestra. Si las dos claves coinciden, se permite que la operación continúe; en caso contrario, se genera una condición de excepción. El proceso de revocación sustituye la clave maestra con un nuevo valor mediante la operación `set-key`, invalidando todas las capacidades anteriores para este objeto.

Este esquema no permite la revocación selectiva, ya que con cada objeto sólo hay asociada una clave maestra. Si asociamos una lista de claves con cada objeto, entonces podrá implementarse la revocación selectiva. Finalmente, podemos agrupar todas las claves en una tabla global de claves. Una capacidad será válida sólo si su clave se corresponde con alguna de las claves de la tabla global. Implementamos la revocación eliminando de la tabla esa clave que se corresponda. Con este esquema, podemos asociar una clave con varios objetos y puede haber también varias claves asociadas con un único objeto, proporcionando así la máxima flexibilidad.

En los esquemas basados en claves, las operaciones de definición de claves, de inserción de claves en listas y de borrado de claves de las listas no deben estar disponibles para todos los usuarios. En particular, sería razonable permitir que sólo el propietario de un objeto pueda configurar las claves de dicho objeto. Esta elección, sin embargo, es una decisión de política que el sistema de protección puede implementar pero que no debe definir a priori.

14.8 Sistemas basados en capacidades

En esta sección, vamos a repasar dos sistemas de protección basados en capacidades. Estos sistemas varían tanto en lo que se refiere a su complejidad como en el tipo de políticas que pueden implementarse sobre ellos. En ninguno de los dos sistemas se utiliza ampliamente, pero constituyen casos de prueba interesantes para verificar los análisis teóricos relativos a los mecanismos de protección.

14.8.1 Un ejemplo: Hydra

Hydra es un sistema de protección basado en capacidades que proporciona una considerable flexibilidad. El sistema conoce e interpreta un conjunto fijo de posibles derechos de acceso. Estos derechos incluyen formas básicas de acceso tales como el derecho de leer, escribir o ejecutar un segmento de memoria. Además, un usuario (del sistema de protección) puede declarar otros derechos. La interpretación de los derechos definidos por el usuario se lleva a cabo exclusivamente por el programa de usuario, pero el sistema proporciona protección de acceso para el uso de estos derechos, además de para el uso de los derechos definidos por el sistema. Estas características constituyen un avance significativo en la tecnología de protección.

Las operaciones sobre los objetos se definen procedimentalmente. Los procedimientos que implementan tales operaciones son en sí mismos una forma de objeto y se accede a ellos indirectamente a través de capacidades. Los nombres de los procedimientos definidos por el usuario deben ser identificados ante el sistema de protección para que éste pueda gestionar los objetos del tipo definido por el usuario. Cuando se da a conocer a Hydra la definición de un objeto, los nombres de las operaciones que pueden realizarse en este tipo de objetos pasan a ser **derechos auxiliares**. Los derechos auxiliares pueden describirse en una capacidad para una instancia del tipo de objeto. Para que un proceso realice una operación sobre un objeto tipado, la capacidad que tiene para ese objeto deberá contener entre sus derechos auxiliares el nombre de la operación que se está invocando. Esta restricción permite discriminar los derechos de acceso instancia por instancia y proceso por proceso.

Hydra también proporciona un mecanismo de **amplificación de derechos**. Este esquema permite certificar un procedimiento como *de confianza* para que actúe sobre un parámetro formal de un tipo especificado por cuenta de cualquier proceso que disponga de un derecho para ejecutar el procedimiento. Los derechos que mantiene un procedimiento de confianza son independientes de los derechos que tenga el proceso que realiza la invocación y pueden ser mucho más amplios. Sin embargo, dicho procedimiento no debe ser considerado como universalmente de confianza (el procedimiento no está autorizado, por ejemplo, a operar sobre otros tipos de objetos) y ese carácter de confianza no debe extenderse a ningún otro procedimiento o segmento de programa que pueda ser ejecutado por un proceso.

La amplificación permite que los procedimientos de implementación accedan a las variables de representación de un tipo abstracto de datos. Por ejemplo, si un proceso tiene una capacidad sobre un objeto tipado *A*, esta capacidad puede incluir un derecho auxiliar para invocar cierta operación *P* pero no incluiría ninguno de los denominados derechos del *kernel*, como por ejemplo leer, escribir o ejecutar el segmento que representa a *A*. Dicho tipo de capacidad proporciona a un proceso una forma de acceso indirecto (a través de la operación *P*) a la representación de *A*, pero sólo para propósitos específicos.

Cuando un proceso invoca la operación *P* sobre un objeto *A*, sin embargo, la capacidad para acceder a *A* puede ser amplificada al pasar el control al cuerpo de código de *P*. Esta amplificación puede ser necesaria para permitir a *P* el derecho de acceso al segmento de almacenamiento que representa a *A*, para implementar la operación que *P* define sobre el tipo abstracto de datos. El cuerpo de código de *P* puede ser autorizado para leer o escribir directamente en el segmento de *A*, incluso aunque el proceso que realizó la invocación no pueda. Al volver de *P*, se restaura a su estado original no amplificado la capacidad sobre *A*. Este caso representa un ejemplo típico en el que los derechos de acceso que un proceso tiene a un segmento protegido deben cambiar dinámicamente, dependiendo de la tarea que haya que realizar. El ajuste dinámico de los derechos se realiza para garantizar la coherencia de una abstracción definida por el programador. La amplificación de derechos puede enunciarse explícitamente en la declaración de un tipo abstracto para darla a conocer al sistema operativo Hydra.

Cuando un usuario pasa un objeto como argumento a un procedimiento, podemos necesitar garantizar que el procedimiento no pueda modificar el objeto. Podemos implementar esta restricción fácilmente pasando un derecho de acceso que no incluya el derecho de modificación (escritura). Sin embargo, si puede tener lugar una amplificación, es posible que se restablezca el derecho de modificación. De este modo, el requisito de protección definido por el usuario podría

ser puenteado. En general, por su puesto, el usuario puede confiar en que un procedimiento realice su tarea correctamente. De todos modos, esta suposición no siempre es adecuada, debido a los errores hardware o software. Hydra resuelve este problema restringiendo las amplificaciones.

El mecanismo de llamada a procedimientos de Hydra fue diseñado como solución directa al problema de los subsistemas mutuamente sospechosos. Este problema se define de la forma siguiente: suponga que se proporciona un programa que puede ser invocado como servicio por varios usuarios distintos (por ejemplo, una rutina de ordenación, un compilador o un juego). Cuando los usuarios invocan este programa de servicio, asumen el riesgo de que el programa funcione incorrectamente y dañe los datos proporcionados o retenga algunos derechos de acceso a dichos datos para usarlos (sin autorización) posteriormente. De forma similar, el programa de servicio puede tener algunos archivos privados (por ejemplo, para propósitos de contabilización) a los que no debe poder acceder directamente el programa de usuario que realiza la invocación. Hydra proporciona mecanismos para tratar directamente este problema.

Los subsistemas de Hydra se construyen por encima de su *kernel* de protección y pueden requerir protección de sus propios componentes. Un subsistema interactúa con el *kernel* a través de una serie de llamadas a un conjunto de primitivas definidas por el *kernel* que establecen derechos de acceso a los recursos definidos por el subsistema. El diseñador del subsistema puede definir políticas que regulen el uso de estos recursos por parte de los procesos de usuario, pero las políticas se imponen utilizando la protección estándar de acceso permitida por el sistema de capacidades.

Un programador puede hacer un uso directo del sistema de protección después de familiarizarse con sus características mediante el manual de referencia apropiado. Hydra proporciona una amplia biblioteca de procedimientos definidos por el sistema que pueden ser invocados por los programas de usuario. Un usuario del sistema Hydra incorporará explícitamente llamadas a estos procedimientos del sistema en el código de sus programas o utilizará un traductor de programas que disponga de una interfaz con Hydra.

14.8.2 Un ejemplo: sistema CAP de Cambridge

En el sistema CAP de Cambridge se ha adoptado un enfoque distinto para la implementación del mecanismo de protección basado en capacidades. El sistema de capacidades de CAP es más simple y superficialmente menos potente que el de Hydra. Sin embargo, un examen más atento muestra que también puede usarse este sistema para proporcionar una protección segura de los objetos definidos por el usuario. CAP dispone de dos tipos de capacidades. El tipo normal se denomina **capacidad de datos** y puede usarse para proporcionar acceso a los objetos, pero los únicos derechos proporcionados son los derechos normales de lectura, escritura y ejecución de los segmentos de almacenamiento individuales asociados con el objeto. Las capacidades de datos son interpretadas mediante microcódigo incluida en la máquina CAP.

El segundo tipo de capacidades se denomina **capacidad software** y está protegido, pero no interpretado, por el microcódigo de CAP. Para interpretar estas capacidades se utiliza un procedimiento *protegido* (es decir, privilegiado) que puede ser escrito por un programador de aplicaciones como parte de un subsistema. Con cada procedimiento protegido se asocia un tipo particular de amplificación de derechos. Cuando esté ejecutando el cuerpo de código de dicho tipo de procedimiento, un proceso adquirirá temporalmente el derecho a leer o escribir el contenido de una capacidad software. Este tipo específico de amplificación de derechos se corresponde con una implementación de las primitivas *seal* y *unseal* utilizadas para las capacidades. Por supuesto, este privilegio sigue estando sujeto a un mecanismo de verificación de tipos para garantizar que sólo se pasen a uno de estos procedimientos capacidades software para un tipo abstracto especificado. No se asigna una confianza universal a ningún segmento de código distinto del propio microcódigo de la máquina CAP (véanse las referencias en las Notas bibliográficas).

La interpretación de una capacidad software se deja completamente al arbitrio del subsistema, a través de los procedimientos protegidos que contenga. Este esquema permite implementar diversas políticas de protección. Aunque un programador puede definir sus propios procedimientos protegidos (cuálquiero de los cuales puede ser incorrecto), la seguridad del sistema global no

puede verse comprometida. El sistema básico de protección no permitirá que un procedimiento protegido definido por el usuario y no verificado acceda a ningún segmento de almacenamiento (o capacidad) que no pertenezca al entorno de protección en el que respira. La consecuencia más seria de un procedimiento protegido inseguro será, de este modo, un fallo de protección del subsistema del que dicho procedimiento sea responsable.

Los diseñadores del sistema CAP han podido observar que el uso de capacidades software les permitía economizar considerablemente a la hora de formular e implementar políticas de protección adecuadas a los requisitos de los recursos abstractos. Sin embargo, un diseñador de subsistemas que quiera hacer uso de esta funcionalidad no puede simplemente estudiar un manual de referencia, como en el caso de Hydra. En lugar de ello, deberá aprender los principios y técnicas de protección, ya que el sistema no proporciona ninguna biblioteca de procedimientos.

14.9 Protección basada en el lenguaje

El grado de protección que se proporciona en los sistemas informáticos existentes suele conseguirse mediante un *kernel* del sistema operativo, que actúa como agente de seguridad para inspeccionar y validar cada intento de acceder a un recurso protegido. Puesto que una validación de acceso exhaustiva es potencialmente una fuente de gasto adicional considerable de recursos de procesamiento, debemos proporcionar a este mecanismo un soporte hardware para reducir el coste de cada validación o debemos aceptar que el diseñador del sistema llegue a ciertos compromisos en lo que respecta a los objetivos de la protección. Satisfacer todos esos objetivos es difícil si la flexibilidad para implementar las políticas de protección está restringida por los mecanismos de soporte proporcionados o si los entornos de protección deben hacerse más complejos de lo necesario para garantizar una mayor eficiencia de operación.

A medida que ha aumentado la complejidad de los sistemas operativos, y particularmente a medida que éstos han tratado de proporcionar interfaces de usuario de mayor nivel, los objetivos de la protección se han hecho mucho más refinados. Los diseñadores de sistemas de protección han aprovechado al máximo una serie de ideas que tienen su origen en los enlaces de programación y especialmente en los conceptos de tipos abstractos de datos y objetos. Hoy en día, los sistemas de protección no sólo se ocupan de la identidad de un recurso al que se intente acceder, sino también de la naturaleza funcional de dicho acceso. En los sistemas de protección más modernos, la preocupación por la función que se está invocando va más allá de un conjunto de funciones definidas por el sistema, como por ejemplo los métodos estándar de acceso a archivos, para incluir también las funciones que el usuario pueda haber definido.

Las políticas de uso de los recursos también pueden variar, dependiendo de la aplicación, y pueden estar sujetas a cambios a lo largo del tiempo. Por estas razones, la protección ya no puede ser considerada exclusivamente como una preocupación del diseñador del sistema operativo, sino que también debe estar disponible como herramienta para que la use el diseñador de aplicaciones, de modo que los recursos de un subsistema de aplicación puedan estar protegidos frente a modificaciones o frente a la influencia de posibles errores.

14.9.1 Imposición de reglas basadas en compilador

Es aquí donde entran dentro del panorama los lenguajes de programación. Especificar el control de acceso deseado a un recurso compartido con un sistema no es otra cosa que realizar un enunciado declarativo acerca del recurso. Este tipo de enunciado puede integrarse en un lenguaje extendiendo su funcionalidad de definición de tipos. Cuando se declara la protección junto con la definición del tipo de datos, el diseñador de cada subsistema puede especificar sus requisitos de protección, así como su necesidad de utilizar otros recursos dentro de un sistema. Dicha especificación debe proporcionarse directamente a medida que se compone un programa y en el propio lenguaje en el que el programa esté escrito. Esta técnica tiene varias ventajas significativas.

1. Las necesidades de protección simplemente se declaran, en lugar de programarlas como una secuencia de llamadas a procedimientos de un sistema operativo.

2. Los requisitos de protección pueden enunciarse independientemente de la funcionalidad proporcionada por un sistema operativo concreto.
3. El diseñador de un subsistema no tiene por qué proporcionar los medios para imponer el mecanismo de protección.
4. La notación declarativa resulta bastante natural, porque los privilegios de acceso están estrechamente relacionados con el concepto lingüístico de tipo de datos.

La implementación de un lenguaje de programación puede proporcionar diversas técnicas para imponer la protección, pero todas estas técnicas dependen hasta cierto punto del soporte proporcionado por la máquina subyacente y por su sistema operativo. Por ejemplo, suponga que se utiliza un lenguaje para generar un código que deba ejecutarse sobre el sistema CAP de Cambridge. En este sistema, cada referencia al almacenamiento que se realice sobre el hardware subyacente tiene lugar de manera indirecta a través de una capacidad. Esta restricción impide a cualquier proceso acceder a un recurso que esté situado, en un momento dado, fuera de su entorno de protección. Sin embargo, un programa puede imponer restricciones arbitrarias en lo que respecta al modo en que un recurso puede utilizarse durante la ejecución de un segmento de código concreto. Podemos implementar dichas restricciones fácilmente utilizando las capacidades software proporcionadas por CAP. La implementación del lenguaje puede proporcionar procedimientos protegidos estándar para interpretar las capacidades software que lleven a la práctica las políticas de protección que puedan especificarse en el lenguaje. Este esquema pone la especificación de políticas a disposición de los programadores, al mismo tiempo que les libera de tener que implementar la imposición de dichas políticas.

Incluso si un sistema no proporciona un *kernel* de protección tan potente como los de Hydra o CAP, sigue habiendo mecanismos disponibles para implementar las especificaciones de protección dadas en un lenguaje de programación. La distinción principal es que la *seguridad* de esta protección no será tan grande como la que podría obtenerse mediante un *kernel* de protección, porque el mecanismo debe confiar en un conjunto mayor de suposiciones acerca del estado operacional del sistema. Un compilador puede separar las referencias para las que pueda certificar que no se va a producir ninguna violación de protección de aquellas para las que una violación podría ser posible, y puede tratar de manera diferente ambos tipos de referencias. La seguridad proporcionada por este tipo de protección descansa en la suposición de que el *kernel* generado por el compilador no será modificado antes de su ejecución o durante la misma.

Entonces, ¿cuáles son las ventajas relativas de imponer el mecanismo de protección basándose exclusivamente en un *kernel*, por oposición a imponer esos mecanismos principalmente mediante un compilador?

- **Seguridad.** La imposición de los mecanismos de protección mediante un *kernel* proporciona un mayor grado de seguridad del propio sistema de protección, si la comparamos con el procedimiento de generación de código de comprobación de la protección mediante un compilador. En un esquema basado en compilador, la seguridad descansa en la corrección del traductor, en algún mecanismo subyacente de gestión del almacenamiento que proteja los segmentos desde los que se ejecuta el código compilado y, en último término, en la seguridad de los archivos desde los que se carga un programa. Alguna de estas consideraciones se aplica también a un *kernel* de protección con soporte software, pero en menor medida, porque el *kernel* puede residir en segmentos de almacenamiento físico fijos y puede ser cargado exclusivamente desde un archivo designado. Con un sistema de capacidades basado en etiquetas, en el que todos los cálculos de direcciones se realizan por hardware o mediante un micropograma fijo, es posible incluso obtener un grado mayor de seguridad. La protección basada en hardware también es relativamente inmune a las violaciones de protección que puedan tener lugar como resultado de fallos de funcionamiento en el hardware o el software del sistema.
- **Flexibilidad.** Existen límites en lo que respecta a la flexibilidad de un *kernel* de protección a la hora de implementar una política definida por el usuario, aunque el *kernel* puede suministrar la funcionalidad adecuada para que el sistema imponga sus propias políticas. Con

un lenguaje de programación, puede declararse la política de protección y puede imponerse la misma según sea necesario en cada implementación. Si un lenguaje no proporciona la suficiente flexibilidad, puede extenderse o sustituirse con un menor impacto sobre un sistema que esté en funcionamiento que el que se experimentaría si se modificara el *kernel* del sistema operativo.

- **Eficiencia.** La mayor eficiencia se obtiene cuando la imposición de los mecanismos de protección está soportada directamente por el hardware (o el microcódigo). Si se necesita soporte software, la imposición de la protección basada en el lenguaje tiene la ventaja de que los accesos de carácter estático pueden verificarse fuera de línea en tiempo de compilación. Asimismo, puesto que un compilador inteligente puede ajustar el mecanismo de imposición para satisfacer cada necesidad específica, a menudo puede evitarse el gasto fijo adicional de procesamiento implicado por las llamadas al *kernel*.

En resumen, la especificación de los mecanismos de protección en un lenguaje de programación permite describir a alto nivel las políticas de asignación y uso de los recursos. La implementación de un lenguaje puede proporcionar software para la imposición de los mecanismos de protección en aquellos casos en que no haya disponible un mecanismo automático de comprobación con soporte hardware. Además, pueden interpretar las especificaciones de protección para generar llamadas al sistema de protección proporcionado por el hardware y el sistema operativo.

Una forma de hacer que la protección esté disponible para el programa de aplicación es mediante el uso de una capacidad software que pueda utilizarse como objeto del procesamiento. Inherente a este concepto está la idea de que ciertos componentes de programa pueden tener el privilegio de crear o examinar estas capacidades software. Un programa de creación de capacidades sería capaz de ejecutar una operación primitiva que sellara una estructura de datos, haciendo que sus contenidos sean inaccesibles para todos los componentes de programa que no dispongan del privilegio de sellar o eliminar el sello de dicha estructura. Esos programas no privilegiados pueden copiar su estructura de datos o pasar su dirección a otros componentes de programa, pero no pueden acceder a su contenido. La razón de introducir tales capacidades software es incluir un mecanismo de protección dentro del lenguaje de programación. El único problema con este concepto, tal como está propuesto, es que el uso de las operaciones *seal* y *unseal* (definición y eliminación del sello) adoptan una técnica procedural para especificar la protección. La notación no procedural o declarativa parece preferible a la hora de poner a disposición del programador de aplicaciones los mecanismos de protección.

Lo que necesitamos es un mecanismo seguro y dinámico de control de acceso para distribuir entre los procesos de usuario las capacidades relativas a los recursos del sistema. Para contribuir a la fiabilidad global del sistema, el mecanismo de control de acceso debe ser seguro de utilizar. Para ser útil en la práctica, también debe ser razonablemente eficiente. Este requisito ha conducido al desarrollo de una serie de estructuras de lenguaje que permiten al programador declarar diversas restricciones relativas al uso de un recurso específico gestionado (véanse las apropiadas referencias en las Notas bibliográficas).

Estas estructuras proporcionan mecanismos para tres funciones:

1. Distribución de las capacidades de manera segura y eficiente entre los procesos cliente: en particular, los mecanismos garantizan que un proceso de usuario sólo pueda usar el recurso gestionado si se le ha concedido una capacidad sobre dicho recurso.
2. Especificar el tipo de operaciones que un proceso concreto pueda invocar sobre un recurso asignado (por ejemplo, un lector de un archivo sólo debe poder leer el archivo, mientras que un escritor debe poder tanto leer como escribir el archivo): no debe ser necesario conceder el mismo conjunto de derechos a todos los procesos de usuario y debe ser imposible que un proceso pueda ampliar su conjunto de derechos de acceso, excepto con la autorización del mecanismo de control de acceso.
3. Especificar el orden en que un proceso concreto puede invocar las diversas operaciones sobre un recurso (por ejemplo, antes de poder leer un archivo hay que abrirlo). Debe ser posible

asignar a dos procesos restricciones distintas en lo que respecta al orden en que pueden invocar las operaciones relativas al recurso asignado.

La incorporación de conceptos de protección en los lenguajes de programación, como herramienta práctica para el diseño de sistemas, está todavía en su infancia. La protección constituirá probablemente una mayor preocupación para los diseñadores de nuevos sistemas con arquitecturas distribuidas y requisitos cada vez más estrictos en lo que respecta a la seguridad de los datos. Es entonces cuando se reconocerá de manera más generalizada la importancia de notaciones de lenguaje adecuadas mediante las cuales expresar los requisitos de protección.

14.9.2 Protección en Java

Como Java fue diseñado para ejecutarse en un entorno distribuido, la máquina virtual de Java (o JVM) tiene muchos mecanismos de protección integrados. Los programas Java están compuestos por **clases**, cada una de las cuales es una colección de campos de datos y funciones (denominadas **métodos**) que operan sobre esos campos. La JVM carga una clase en respuesta a una solicitud de creación de instancias (u objetos) de dicha clase. Una de las características más novedosas y útiles de Java es su soporte para cargar dinámicamente clases que no sean de confianza a través de una red y para ejecutar clases que desconfían mutuamente unas de otras dentro de una misma JVM.

Debido a estas capacidades de Java, la protección es una de las principales preocupaciones. Las clases que se ejecutan en una misma JVM pueden tener diferentes orígenes y pueden no gozar de un grado igual de confianza. Como resultado, resulta insuficiente garantizar la protección con la granularidad de los procesos JVM. Intuitivamente, el que una solicitud de apertura de un archivo deba ser permitida dependerá, generalmente, de la clase que haya efectuado la solicitud. El sistema operativo carece de este conocimiento.

Por tanto, dichas decisiones de protección se gestionan dentro de la JVM. Cuando la JVM carga una clase, la asigna a un dominio de protección que proporciona los permisos correspondientes a esa clase. El dominio de protección al que se asigna la clase dependerá de la URL desde la que se cargara la clase y de las firmas digitales que contenga el archivo de clases (hablaremos de las firmas digitales en la Sección 15.4.1.3). Un archivo configurable de política determina los permisos concedidos al dominio (y a sus clases). Por ejemplo, las clases cargadas desde un servidor de confianza pueden colocarse en un dominio de protección que nos permita acceder a archivos situados en el directorio principal del usuario, mientras que las clases cargadas desde un servidor que no sea de confianza pueden no tener ningún permiso de acceso a archivos.

Puede ser complicado para la JVM determinar qué clase es responsable de una solicitud de acceso a un recurso protegido. Los accesos se realizan a menudo indirectamente, mediante bibliotecas del sistema u otras clases. Por ejemplo, considere una clase que no esté autorizada para abrir conexiones de red y piense qué sucede si esa clase invoca una biblioteca del sistema para solicitar la carga del contenido de una URL. La JVM debe decidir si abrir o no la conexión de red de acuerdo con esta solicitud. ¿Pero qué clase debe utilizarse para determinar si hay que permitir la conexión, la aplicación o la biblioteca del sistema?

La filosofía adoptada en Java consiste en requerir que la clase de la biblioteca permita explícitamente la conexión de red para cargar la URL solicitada. De forma más general, para poder acceder a un recurso protegido, alguno de los métodos en la secuencia de llamadas que dio como resultado la solicitud deberá permitir explícitamente el privilegio de acceder al recurso. Al hacer esto, dicho método *se hace responsable* de la solicitud y, presumiblemente, realizará también las comprobaciones necesarias para garantizar que dicha solicitud sea segura. Por supuesto, no todos los métodos están autorizados a establecer un privilegio; un método puede establecer un privilegio sólo si su clase se encuentra dentro de un dominio de protección que a su vez está autorizado a ejercer el privilegio.

Esta técnica de implementación se denomina **inspección de la pila**. Cada hebra de la JVM tiene una pila asociada con sus invocaciones actuales a métodos. Cuando no pueda confiar en aquel que realizó la invocación, un método ejecutará una solicitud de acceso dentro de un bloque `doPrivileged` para conseguir el acceso a un recurso protegido directa o indirectamente. `doPrivileged()` es un método estático en la clase `the AccessController` al que se le pasa

una clase con un método `run()` al que haya que invocar. Cuando se entra en el bloque `doPrivileged`, se inserta la correspondiente anotación en la pila de este método para indicar ese hecho. A continuación, se ejecuta el contenido del bloque. Cuando se solicita posteriormente un acceso a un recurso protegido, bien por este método o por otro método al que éste haya llamado, se utiliza una llamada a `checkPermissions()` para invocar la inspección de la pila con el fin de determinar si la solicitud debe autorizarse. La inspección examina el contenido de la pila de la hebra, empezando por la entrada añadida más recientemente y continuando hacia la más antigua. Si se encuentra primero una entrada de la pila que tenga la anotación `doPrivileged()`, entonces `checkPermissions()` volverá inmediatamente y de forma silenciosa, permitiendo el acceso. Si se encuentra primero una entrada de pila para la que el acceso no esté permitido, basándose en el dominio de protección de la clase del método, entonces `checkPermissions()` genera una excepción `AccessControlException`. Si se termina la inspección de la pila sin encontrar ninguno de estos dos tipos de entradas, entonces el que el acceso se permita dependerá de la implementación (por ejemplo, algunas implementaciones de la JVM pueden permitir el acceso mientras que otras pueden denegarlo).

El mecanismo de inspección de la pila se ilustra en la Figura 14.9. Aquí, el método `gui()` de una clase en el dominio de protección correspondiente a las *untrusted applets* (applets no de confianza) realiza dos operaciones, primero una operación `get()` y luego una `open()`. La primera es una invocación del método `get()` de una clase del dominio de protección *URL loader*, al que se permite abrir con `open()` sesiones en sitios situados en el dominio `lucent.com`, en particular en un servidor proxy denominado `proxy.lucent.com` para extraer páginas URL. Por esta razón, la invocación a `get()` de la *applet* no de confianza será autorizada: la llamada a `checkPermissions()` en la biblioteca de red encontrará la entrada de pila correspondiente al método `get()`, que realizó su llamada a `open()` en un bloque `doPrivileged`. Sin embargo, la invocación a `open()` por parte de la *applet* no de confianza generará una excepción, porque la llamada a `checkPermissions()` no encontrará ninguna anotación `doPrivileged` antes de encontrar la entrada de pila correspondiente al método `gui()`.

Por supuesto, para que el mecanismo de inspección de pila funcione, un programa no debe poder modificar las anotaciones de su propia pila ni realizar ningún otro tipo de manipulación de la pila. Esta es una de las diferencias más importantes entre Java y muchos otros lenguajes (incluyendo C++). Un programa Java no puede acceder directamente a la memoria. En lugar de ello, sólo puede manipular un objeto para el que disponga de una referencia. Las referencias no pueden ser imitadas y las manipulaciones sólo se realizan mediante interfaces bien definidas. El respeto de estas reglas se impone mediante una sofisticada colección de comprobaciones que se llevan a cabo en tiempo de carga y en tiempo de ejecución. Como resultado, un objeto no puede manipular su pila de ejecución, porque no puede obtener una referencia a la pila ni a otros componentes del sistema de protección.

De manera más general, las comprobaciones de tiempo de carga y de tiempo de ejecución de Java imponen una **seguridad de los tipos** de las clases Java. La seguridad de los tipos garantiza

dominio de protección:	<code>untrusted applet</code>	<code>URL loader</code>	<code>red</code>
permiso de socket:	<code>get()</code>	<code>lucent.com:80</code>	<code>get()</code>
clase:	<code>get()</code>	<code>doPrivileged()</code>	<code>checkPermission()</code>

Diagrama de la inspección de la pila:

```

graph TD
    subgraph Stack [Pila]
        direction TB
        S1["untrusted applet"] --- S2["URL loader"]
        S2 --- S3["red"]
    end
    S1 --- M1["get()"]
    S2 --- M2["doPrivileged()"]
    S3 --- M3["checkPermission()"]
    M1 --- R1["lucent.com:80"]
    M2 --- R2["open(proxy.lucent.com:80)"]
    M3 --- R3["(e)"]
    style Stack fill:none,stroke:none
    style S1 fill:none,stroke:none
    style S2 fill:none,stroke:none
    style S3 fill:none,stroke:none
    style M1 fill:none,stroke:none
    style M2 fill:none,stroke:none
    style M3 fill:none,stroke:none
    style R1 fill:none,stroke:none
    style R2 fill:none,stroke:none
    style R3 fill:none,stroke:none

```

Figura 14.9 Inspección de la pila.

que las clases no puedan tratar los enteros como punteros, no puedan escribir más allá del final de una matriz ni acceder a la memoria de ninguna otra manera arbitraria. Más bien, un programa sólo puede acceder a un objeto a través de los métodos definidos para dicho objeto mediante su clase. En esto es en lo que se basa la protección de Java, ya que permite a las clases **encapsular** de manera efectiva y proteger sus datos y métodos frente a otras clases cargadas en la misma JVM. Por ejemplo, puede definirse una variable como **private** de modo que sólo pueda acceder a ella la clase que la contiene, o puede definírsela como **protected** para que sólo puedan acceder a ella la clase que la contiene, las subclases de esa clase o las clases contenidas en el mismo paquete. La seguridad de tipos garantiza que puedan imponerse estas restricciones.

14.10 Resumen

Los sistemas informáticos contienen muchos objetos y estos objetos necesitan protegerse frente a posibles malos usos. Los objetos pueden ser hardware (como la memoria, el tiempo de CPU y los dispositivos de E/S) o software (como los archivos, programas y semáforos). Un derecho de acceso es el permiso para realizar una operación sobre un objeto. Un dominio es un conjunto de derechos de acceso. Los procesos se ejecutan en dominios y pueden usar cualquiera de los derechos de acceso del dominio para acceder a los objetos y manipularlos. Durante su ciclo de vida, un proceso puede estar limitado a un dominio de protección o se le puede permitir que conmute de un dominio de protección a otro.

La matriz de acceso es un modelo general de protección que proporciona un mecanismo que no impone ninguna política concreta de protección al sistema ni a sus usuarios. La separación entre política y mecanismo es una importante propiedad de diseño.

La matriz de acceso es una matriz dispersa. Normalmente se la implementa mediante listas de acceso asociadas con cada objeto o mediante listas de capacidades asociadas con cada dominio. Podemos incluir mecanismos de protección dinámicos dentro del modelo de matriz de acceso considerando los dominios y la propia matriz de acceso como objetos. La reagrupación de derechos de acceso en un modelo dinámico de protección es normalmente más fácil de implementar con un esquema de listas de acceso que con uno basado en listas de capacidades.

Los sistemas reales están mucho más limitados que el modelo general y tienden a proporcionar protección exclusivamente para los archivos. UNIX es bastante representativo, y proporciona protección de lectura, escritura y ejecución de manera separada para el propietario, el grupo y el público general de cada archivo. MULTICS utiliza una estructura de anillos además del mecanismo de protección de acceso a archivos. Hydra, el sistema CAP de Cambridge y Mach son sistemas basados en capacidades que amplían la protección a los objetos software definidos por el usuario. Solaris 10 implementa el principio del mínimo privilegio mediante un control de acceso basado en roles, que es una forma de matriz de acceso.

La protección basada en lenguaje proporciona una capacidad de arbitraje de solicitudes y privilegios con una granularidad más fina de la que el sistema operativo es capaz de proporcionar. Por ejemplo, una misma JVM Java puede ejecutar varias hebras, cada una de ellas en una clase de protección diferente. Java aplica a las solicitudes de recursos un mecanismo sofisticado de inspección de la pila y los mecanismos de seguridad de tipos del lenguaje para imponer los mecanismos de protección.

Ejercicios

- 14.1 Considere el esquema de protección en anillos de MULTICS. Si quisieramos implementar las llamadas al sistema de un sistema operativo típico y almacenarlas en un segmento asociado con el anillo 0, ¿cuáles deberían ser los valores almacenados en el campo de anillo del descriptor del segmento? ¿Qué pasa durante una llamada al sistema cuando un proceso que se ejecuta en un anillo de número más alto invoca un procedimiento del anillo 0?
- 14.2 Podría utilizarse la matriz de control de acceso para determinar si un proceso puede conmutar de, por ejemplo, el dominio A al dominio B y disfrutar de los privilegios de acceso

del dominio B. ¿Es esta técnica equivalente a incluir los privilegios de acceso del dominio B en los del dominio A?

- 14.3 Considere un sistema informático en el que los estudiantes sólo puedan ejecutar programas de juegos entre las 10 de la noche y las 6 de la mañana, en el que los profesores puedan ejecutarlos entre las 5 de la tarde y las 8 de la mañana y en el que el personal del centro de cálculo pueda ejecutarlos en cualquier momento. Sugiera un esquema para implementar esta política de manera eficiente.
- 14.4 ¿Qué características hardware se necesitan en un sistema informático para poder manipular de manera eficiente las capacidades? ¿Pueden utilizarse estas características hardware para la protección de memoria?
- 14.5 Explique las fortalezas y debilidades de un sistema que implemente una matriz de acceso utilizando listas de acceso asociadas con los objetos.
- 14.6 Explique las fortalezas y debilidades de un sistema que implemente una matriz de acceso utilizando capacidades asociadas con los dominios.
- 14.7 Explique por qué un sistema basado en capacidades como Hydra proporciona una mayor flexibilidad que el esquema de protección en anillos a la hora de imponer políticas de protección.
- 14.8 Explique la necesidad del mecanismo de amplificación de derechos de Hydra. Compare este mecanismo con las llamadas inter-anillos en un esquema de protección en anillos.
- 14.9 ¿Qué es el principio de la necesidad de conocer? ¿Por qué es importante que un sistema de protección se ajuste a este principio?
- 14.10 Explique cuáles de los siguientes sistemas permiten a los diseñadores de módulos imponer el principio de la necesidad de conocer:
 - a. El esquema de protección en anillos de MULTICS.
 - b. El sistema de capacidades de Hydra.
 - c. El esquema de inspección de pila de la JVM.
- 14.11 Describa cómo se vería afectado el modelo de protección de Java si se permitiera a un programa Java modificar directamente las anotaciones incluidas en su entrada de pila.
- 14.12 ¿En qué sentido son similares el modelo de matriz de acceso y el modelo de control de acceso basado en roles? ¿En qué sentido difieren?
- 14.13 ¿Cómo ayuda a la creación de sistemas de protección el principio del mínimo privilegio?
- 14.14 ¿Cómo pueden los sistemas que implementan el principio del mínimo privilegio tener fallos de protección que conduzcan a violaciones de seguridad?

Notas bibliográficas

El modelo de protección basado en una matriz de acceso de dominios y objetos fue desarrollado por Lampson [1969] y Lampson [1971]. Popek [1974] y Saltzer y Schroeder [1975] proporcionan excelentes panorámicas sobre el tema de la protección. Harrison et al. [1976] utilizó una versión formal de este modelo para poder demostrar matemáticamente las propiedades de un sistema de protección.

El concepto de capacidad evolucionó a partir de las denominadas *palabras de código* de Illiffe y Jodeit, que se implementaron en la computadora de Rice University (Illiffe y Jodeit [1962]). El término *capacidad* fue introducido por Dennis y Horn [1966]. El sistema Hydra fue descrito por Wulf et al. [1981], mientras que el sistema CAP lo fue por Needham y Walker [1977]. Organick [1972] analiza el sistema de protección en anillos de MULTICS.

El tema de la revocación se explora en Redell y Fabry [1974], Cohen y Jefferson [1975] y Ekanadham y Bernstein [1979]. El principio de la separación entre política y mecanismo fue defendido por el diseñador de Hydra (Levin et al. [1975]). El problema de confinamiento fue analizado por primera vez por Lampson [1973] y posteriormente fue examinado por Lipner [1975].

El uso de lenguajes de alto nivel para especificar los controles de acceso fue sugerido por primera vez por Morris [1973], que propuso el uso de las operaciones `seal` y `unseal` que se explica en la Sección 14.9. Kieburz y Silberschatz [1978], Kieburz y Silberschatz [1983] y McGraw y Andrews [1979] propusieron varias estructuras de lenguaje para implementar esquemas generales de gestión dinámica de recursos. Jones y Liskov [1978] analizaron cómo puede incorporarse un esquema estático de control de acceso en un lenguaje de programación que soporte tipos abstractos de datos. El uso de un soporte mínimo por parte del sistema operativo para imponer los mecanismos de protección fue defendido por el proyecto Exokernel (Ganger et al. [2002], Kaashoek et al. [1997]). Las ampliabilidad del código del sistema mediante mecanismos de protección basados en el lenguaje fue analizada en Bershad et al. [1995b]. Otras técnicas para garantizar la protección incluyen la técnica denominada de cajón de arena (Goldberg et al. [1996]) y la de aislamiento de los fallos software (Wahbe et al. [1993b]). Las cuestiones relativas a cómo reducir la sobrecarga de procesamiento asociada con los mecanismos de protección y a cómo permitir el acceso de nivel de usuario a los dispositivos de red fue analizada en McCanne y Jacobson [1993] y Basu et al. [1995].

Puede encontrar análisis más detallados del mecanismo de inspección de pila, incluyendo comparaciones con otras técnicas de implementación de mecanismos de seguridad en Java, en Wallach et al. [1997] y Gong et al. [1997].



Seguridad

La protección, como hemos explicado en el Capítulo 14, es estrictamente un problema *interno*: ¿cómo proporcionamos un acceso controlado a los programas y a los datos almacenados en un sistema informático? La **seguridad**, por otro lado, requiere no sólo un adecuado sistema de protección, sino también tener en cuenta el entorno *externo* dentro del que opera el sistema. Un sistema de protección no será efectivo si la autenticación de los usuarios se ve comprometida o si un programa es ejecutado por un usuario no autorizado.

Los recursos informáticos deben protegerse frente a los accesos no autorizados, frente a la modificación o destrucción maliciosas y frente a la introducción accidental de incoherencias. Entre esos recursos podemos incluir la información almacenada en el sistema (tanto datos como código), así como la CPU, la memoria, los discos, las cintas y los sistemas de interconexión por red que forman la computadora. En este capítulo, comenzaremos examinando las formas en que los recursos pueden ser mal utilizados de forma accidental o premeditada. A continuación, exploraremos uno de los factores principales en los que se basa la seguridad: la criptografía. Finalmente, examinaremos los mecanismos que existen para defenderse de los ataques y detectarlos.

OBJETIVOS DEL CAPÍTULO

- Analizar las amenazas a la seguridad y los tipos de ataques posibles.
- Explicar los fundamentos del cifrado, la autenticación y las funciones *hash*.
- Examinar los usos de la criptografía en el campo de la Informática.
- Describir las diversas contramedidas existentes para los ataques a los sistemas de seguridad.

15.1 El problema de la seguridad

En muchas aplicaciones, merece la pena dedicar un esfuerzo considerable a garantizar la seguridad del sistema informático. Los sistemas comerciales de gran envergadura que contengan datos de nóminas u otros datos financieros constituyen objetivos muy codiciados por los delincuentes. Asimismo, los sistemas que contengan datos relativos a las operaciones de una corporación pueden resultar de interés para los competidores carentes de escrúpulos. Además, la pérdida de tales datos, ya sea accidentalmente o como consecuencia del fraude, puede afectar seriamente a la capacidad de esa corporación para seguir funcionando.

En el Capítulo 14 hemos examinado los mecanismos que el sistema operativo puede proporcionar (con la adecuada ayuda por parte del hardware) para permitir a los usuarios proteger sus recursos, incluyendo los programas y los datos. Estos mecanismos funcionan adecuadamente mientras que los usuarios respeten los usos previstos y el tipo de acceso que se hubiera pensado para esos recursos. Decimos que un sistema es **seguro** si sus recursos se utilizan de la forma pre-

vista y si se accede a ellos tal como se pretendía, en todas las circunstancias. Desafortunadamente, no es posible conseguir una seguridad total; a pesar de ello, debemos prever mecanismos para hacer que los fallos de seguridad constituyan la excepción, en lugar de la norma.

Las violaciones de seguridad o la mala utilización de un sistema pueden clasificarse en dos categorías: intencionadas (maliciosas) o accidentales. Resulta más fácil protegerse frente a una mala utilización accidental que frente a otra maliciosa. Principalmente, los mecanismos de protección forman la base de la defensa frente a posibles accidentes. En la siguiente lista indicamos diversas formas de violaciones de seguridad accidentales y maliciosas. Conviene observar que en nuestras explicaciones acerca de la seguridad utilizamos los términos *intruso* y *cracker* para designar a las personas que tratan de romper los sistemas de seguridad. Asimismo, una **amenaza** es la posibilidad de que exista una violación de seguridad, como por ejemplo el descubrimiento de una vulnerabilidad, mientras que un **ataque** es un intento de romper la seguridad.

- **Ruptura de la confidencialidad.** Este tipo de violación implica la lectura no autorizada de determinados datos (o el robo de información). Típicamente, el objetivo de los intrusos es una ruptura de la confidencialidad. La captura de datos secretos en un sistema o en un flujo de datos, como por ejemplo información relativa a tarjetas de crédito o información personal para fingir una identidad ficticia, puede reportar al intruso un beneficio monetario directo.
- **Ruptura de la integridad.** Este tipo de ataque implica la modificación no autorizada de los datos. Estos ataques pueden, por ejemplo, provocar que se atribuya una cierta responsabilidad a alguien que es inocente o que se modifique el código fuente de una aplicación comercial de cierta importancia.
- **Ruptura de la disponibilidad.** Esta violación de seguridad implica la destrucción no autorizada de datos. Algunos atacantes prefieren causar daño y adquirir un cierto renombre en lugar de obtener beneficios financieros. La sustitución de la página de entrada de un sitio web es un ejemplo común de este tipo de ruptura de la seguridad.
- **Robo de servicio.** Este tipo de violación de seguridad implica el uso no autorizado de recursos. Por ejemplo, un intruso (o un programa de intrusión) puede instalar un demonio en un sistema que actúe como servidor de archivos.
- **Denegación de servicio.** Esta violación de seguridad implica impedir el uso legítimo del sistema. Los ataques de **denegación de servicio** (DOS, denial-of-service) son en ocasiones accidentales. Un ejemplo común es el de un programa gusano que atacó Internet hace ya unos cuantos años y que se transformó en un ataque DOS cuando un error en el propio programa hizo que no se retardara su rápida diseminación. Hablaremos más en detalle de los ataques DOS en la Sección 15.3.3.

Los atacantes utilizan diversos métodos estándar en sus intentos de romper la seguridad de los sistemas. El más común es la **mascarada**, en la que un participante en una comunicación pretende ser otra persona (u otro *host*). Mediante la mascarada, los atacantes rompen la **autenticación**, es decir, la corrección de la identificación; como consecuencia, pueden obtener tipos de acceso que normalmente no les estarían permitidos o escalar sus privilegios, es decir, obtener privilegios de los que normalmente no podrían disfrutar. Otro ataque común consiste en reproducir un intercambio de datos previamente capturado: los **ataques de reproducción** consisten en la repetición maliciosa o fraudulenta de una transmisión de datos válida. En ocasiones, esa reproducción constituye el propio ataque, como por ejemplo cuando se repite una solicitud para transferir dinero, pero frecuentemente la reproducción se realiza en conjunción con una **modificación de mensaje**, que de nuevo suele tener como objetivo aumentar los privilegios. Considere, por ejemplo, los daños que podrían producirse si se sustituyera la información de un usuario legítimo por la de un usuario no autorizado en una solicitud de autenticación. Otro tipo de ataque es el **ataque por interposición (man-in-the-middle)**, en el cual un atacante se introduce dentro del flujo de datos de una comunicación, haciendo pasar por el emisor a ojos del receptor y viceversa. En una comunicación por red, un ataque por interposición puede estar precedido por un **secuestro de**

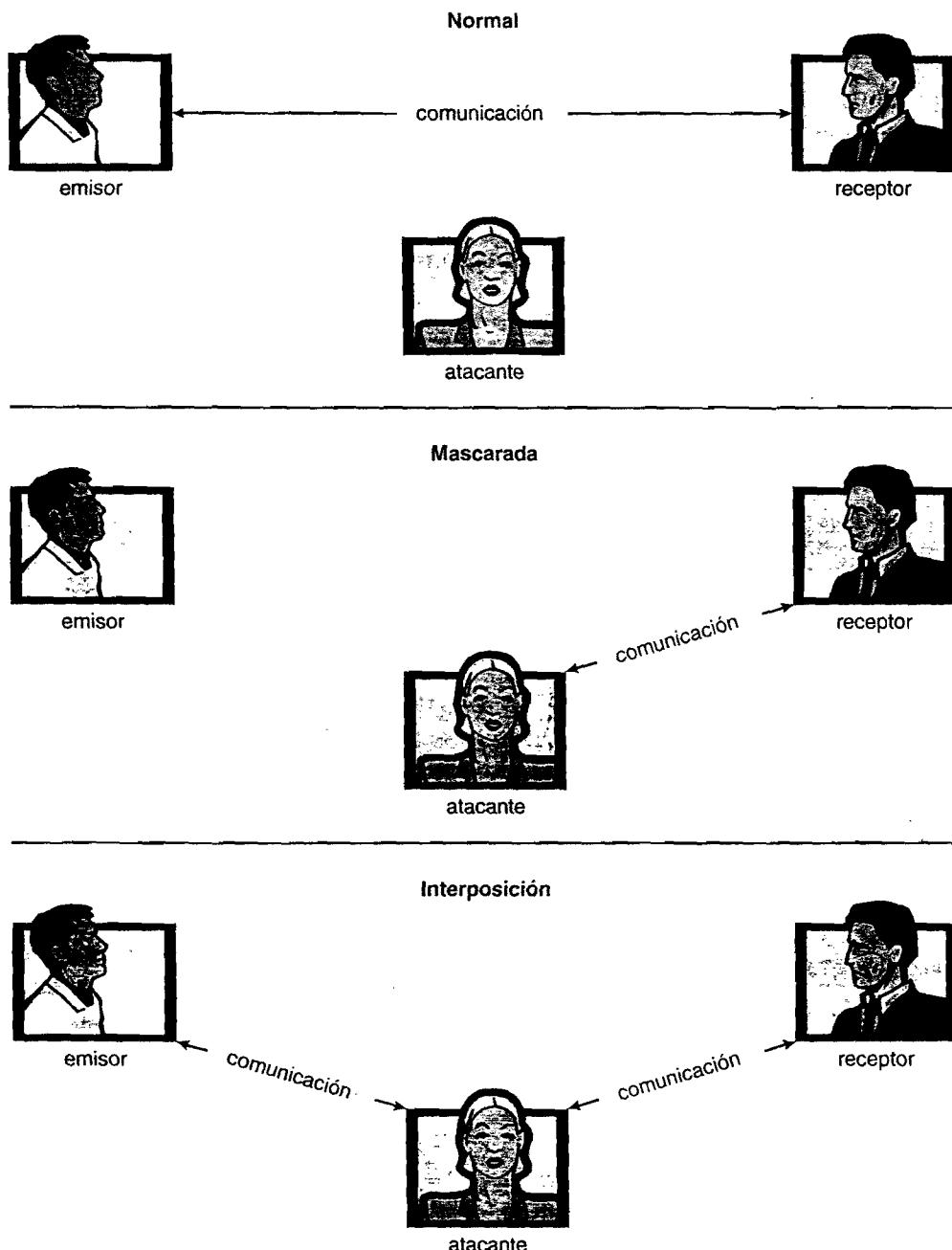


Figura 15.1 Ataques estándar a la seguridad.

sesión (*hijacking*), en el que se intercepta una sesión de comunicación activa. La Figura 15.1 ilustra diversos métodos de ataque.

Como ya hemos comentado, resulta imposible garantizar una protección absoluta del sistema frente a los abusos de carácter malicioso, pero podemos hacer que el coste para aquel que lo intente sea tan alto como para disuadir a la mayoría de los intrusos. En algunos casos, como por ejemplo con los ataques de denegación de servicio, es preferible prevenir el ataque lo suficiente como para detectarlo de modo que puedan adoptarse las necesarias contramedidas.

Para proteger un sistema, debemos adoptar las necesarias medidas de seguridad en cuatro niveles distintos:

1. **Físico.** El nodo o nodos que contengan los sistemas informáticos deben dotarse de medidas de seguridad físicas frente a posibles intrusiones armadas o subrepticias por parte de poten-

ciales intrusos. Hay que dotar de seguridad tanto a las habitaciones donde las máquinas residen como a los terminales o estaciones de trabajo que tengan acceso a dichas máquinas.

2. **Humano.** La autorización de los usuarios debe llevarse a cabo con cuidado, para garantizar que sólo los usuarios apropiados tengan acceso al sistema. Sin embargo, incluso los usuarios autorizados pueden verse “motivados” para permitir que otros usen su acceso (por ejemplo, a cambio de un soborno). También pueden ser engañados para permitir el acceso a otros, mediante técnicas de **ingeniería social**. Uno de los tipos de ataque basado en las técnicas de ingeniería social es el denominado **phishing**; con este tipo de ataque, un correo electrónico o página web de aspecto auténtico llevan a engaño a un usuario para que introduzca información confidencial. Otra técnica comúnmente utilizada es el **análisis de desperdicios**, un término general para describir el intento de recopilar información para poder obtener un acceso no autorizado a la computadora (por ejemplo, examinando el contenido de las papeleras, localizando listines de teléfonos o encontrando notas con contraseñas). Estos problemas de seguridad son cuestiones relacionadas con la gestión y con el personal, más que problemas relativos a los sistemas operativos.
3. **Sistema operativo.** El sistema debe autoprotegerse frente a los posibles fallos de seguridad accidentales o premeditados. Un proceso que esté fuera de control podría llegar a constituir un ataque accidental de denegación de servicio. Asimismo, una cierta consulta a un servicio podría conducir a la revelación de contraseñas o un desbordamiento de la pila podría permitir que se iniciara un proceso no autorizado. La lista de posibles fallos de seguridad es casi infinita.
4. **Red.** Son muchos los datos en los modernos sistemas informáticos que viajan a través de líneas arrendadas privadas, de líneas compartidas como Internet, de conexiones inalámbricas o de líneas de acceso telefónico. La interceptación de estos datos podría ser tan dañina como el acceso a una computadora, y la interrupción de la comunicación podría constituir un ataque remoto de denegación de servicio, disminuyendo la capacidad de uso del sistema y la confianza en el mismo por parte de los usuarios.

Si queremos poder garantizar la seguridad del sistema operativo, es necesario garantizar la seguridad en los primeros dos niveles. Cualquier debilidad en uno de los niveles altos de seguridad (físico o humano) permitirá puentear las medidas de seguridad que son estrictamente de bajo nivel (del nivel del sistema operativo). Así, la frase que afirma que una cadena es tan fuerte como el más débil de sus eslabones es especialmente cierta cuando hablamos de la seguridad de los sistemas. Para poder mantener la seguridad, debemos contemplar todos estos aspectos.

Además, el sistema debe proporcionar mecanismos de protección (Capítulo 14) para permitir la implementación de las características de seguridad. Sin la capacidad de autorizar a los usuarios y procesos, de controlar su acceso y de registrar sus actividades, sería imposible que un sistema operativo implementara medidas de seguridad o se ejecutara de forma segura. Para soportar un esquema global de protección hacen falta mecanismos de protección hardware. Por ejemplo, un sistema donde la memoria no esté protegida no puede nunca estar seguro. Las nuevas características hardware permiten hacer más seguros los sistemas, como veremos más adelante.

Desafortunadamente, casi ninguno de los aspectos relativos a la seguridad resulta sencillo. A medida que los intrusos aprenden a aprovechar las vulnerabilidades de los sistemas de seguridad, se crean y se implantan las correspondientes contramedidas, lo cual hace que los intrusos vayan realizando ataques cada vez más sofisticados. Por ejemplo, algunos incidentes relativos a la seguridad que se han experimentando en los últimos años incluyen el uso de software espía como conducto para la distribución de *spam* a través de sistemas inocentes (hablaremos de esta práctica en la Sección 15.2). Este juego del gato y el ratón continuará, probablemente, en el futuro próximo, haciendo necesario disponer de más herramientas de seguridad para bloquear las técnicas y actividades cada vez más sofisticadas de los piratas informáticos.

En el resto de este capítulo, vamos analizar el tema de la seguridad en los niveles de red y del sistema operativo. La seguridad en los niveles físico y humano, aunque importante, cae fundamentalmente fuera del alcance de este texto. La seguridad dentro del sistema operativo y entre sis-

temas operativos se implementa mediante diversas técnicas, que van desde la utilización de contraseñas para la autenticación a la defensa frente a virus y a la detección de intrusos. Vamos a comenzar explorando la gama de amenazas a la seguridad.

15.2 Amenazas relacionadas con los programas

Los procesos son, junto con el *kernel*, el único medio de realizar un trabajo útil en una computadora. Por tanto, un objetivo común de los piratas informáticos consiste en escribir un programa que cree una brecha de seguridad o que haga que un proceso normal cambie su comportamiento y cree esa brecha de seguridad. De hecho, la mayoría de las brechas de seguridad no relacionadas con programas tienen por objetivo crear una brecha que sí esté basada en un programa. Por ejemplo, aunque resulta útil iniciar una sesión en un sistema sin autorización, normalmente es mucho más útil dejar un demonio de tipo **puerta trasera** que proporcione información o que permita un fácil acceso incluso aunque se bloquee la brecha de seguridad original. En esta sección, vamos a describir algunos métodos comunes mediante los que los programas pueden provocar brechas de seguridad. Hay que resaltar que existe una considerable variación en lo que respecta a los convenios de denominación de los agujeros de seguridad, y que en este texto utilizamos los términos más comunes o descriptivos.

15.2.1 Caballo de Troya

Muchos sistemas tienen mecanismos para permitir que programas escritos por unos usuarios sean ejecutados por otros. Si estos programas se ejecutan en un dominio que proporcione los derechos de acceso del usuario ejecutante, los otros usuarios podrían utilizar inapropiadamente estos derechos. Por ejemplo, un programa editor de texto puede incluir código para buscar ciertas palabras clave en el archivo que hay que editar. Si se encuentra alguna, el archivo completo puede copiarse en un área especial accesible por el creador del editor de textos. Un segmento de código que utilice inapropiadamente su entorno se denomina **caballo de Troya**. Las rutas de búsqueda de gran longitud, como las que aparecen frecuentemente en los sistemas UNIX, agravan el problema de los caballos de Troya. La ruta de búsqueda enumera el conjunto de directorios en el que hay que buscar cuando se proporciona un nombre de programa ambiguo. Lo que se hace es buscar en esa ruta un archivo con dicho nombre y ejecutar el archivo. Todos los directorios de esa ruta de búsqueda deben ser seguros, porque de lo contrario podría introducirse un caballo de Troya en la ruta de ejecución del usuario y ejecutarse accidentalmente.

Por ejemplo, considere el uso del carácter “.” dentro de una ruta de búsqueda. El “.” le dice a la *shell* que incluya el directorio actual en la búsqueda. Así, si un usuario tiene “.” en su ruta de búsqueda, ha configurado el directorio actual para situarse en el directorio de un amigo e introduce el nombre de un comando normal del sistema, dicho comando podría ejecutarse desde el directorio de ese amigo. El programa se ejecutaría dentro del dominio del usuario, permitiendo que ese programa hiciera cualquier cosa que el usuario puede hacer, incluyendo por ejemplo borrar los archivos de usuario.

Una variante de caballo de Troya es un programa que emula el típico programa de inicio de sesión. Un usuario que no esté advertido tratará de iniciar la sesión en un terminal y observará que aparentemente ha escrito mal su contraseña; después, vuelve a intentarlo y esta vez lo hace con éxito. Lo que ha sucedido es que su clave de autenticación y su contraseña han sido robadas por el emulador de inicio de sesión, que fue dejado ejecutándose en el terminal por parte del ladrón. El emulador almacena la contraseña, imprime un mensaje de error de inicio de sesión y sale del programa, después de lo cual se presenta al usuario una verdadera pantalla de inicio de sesión. Este tipo de ataque puede evitarse haciendo que el sistema operativo imprima un mensaje de uso al final de una sesión interactiva, exigiendo al usuario que utilice para iniciar la sesión una secuencia de teclas no capturable, como la combinación control-alt-supr usada en todos los sistemas operativos Windows modernos.

Otra variante del caballo de Troya es el **spyware**. Los programas *spyware* acompañan en ocasiones a ciertos programas que el usuario haya decidido instalar. En la mayoría de los casos, se

incluyen con programas de tipo freeware o shareware, pero en otras ocasiones también se incluyen en software de carácter comercial. El objetivo del *spyware* es descargar anuncios para mostrarlos en el sistema del usuario, crear **ventanas de explorador emergentes** cuando se visiten ciertos sitios o capturar información del sistema del usuario y enviarla a un sitio central. Este último modo es un ejemplo de una categoría general de ataques conocida como **canales encubiertos**, a través de los cuales tienen lugar comunicaciones subrepticias. Como ejemplo, la instalación de un programa aparentemente inocuo en un sistema Windows podría provocar la carga de un demonio *spyware*. El *spyware* podría contactar un sitio central, recibir desde allí un mensaje y una lista de direcciones de destino y entregar el mensaje basura a esos usuarios desde la máquina Windows. Este proceso continuaría hasta que el usuario descubriera la existencia del *spyware*. En muchísimos casos, ese programa *spyware* no llega a ser descubierto. En 2004, las estimaciones indicaban que el 80 por ciento del correo basura se distribuía a través de este método. ¡Este robo de servicio ni siquiera se considera un delito en la mayoría de los países!

El *spyware* es un ejemplo a pequeña escala de un problema a gran escala: la violación del principio del mínimo privilegio. En la mayoría de las circunstancias, un usuario de un sistema operativo no necesita instalar demonios de red. Dichos demonios llegan a instalarse debido a dos errores. En primer lugar, un usuario podría estar operando con más privilegios de los necesarios (por ejemplo, como administrador), permitiendo que los programas que ejecute tengan más acceso al sistema del necesario; este es un caso de error humano, que es una de las vulnerabilidades de seguridad más habituales. En segundo lugar, un sistema operativo puede permitir, de manera predeterminada, más privilegios de los que un usuario normal necesita; este es un ejemplo de decisiones de diseño del sistema operativo inadecuadas. Un sistema operativo (y, en general, cualquier tipo de software) debe ofrecer una seguridad y un control de acceso de granularidad fina pero también debe ser fácil de gestionar y de comprender. Las medidas de seguridad incómodas o inadecuadas están condenadas a ser puenteadas, provocando un debilitamiento global del sistema de seguridad que en teoría deberían implementar.

15.2.2 Puerta trasera

El diseñador de un programa o un sistema puede dejar detrás suyo un agujero en el software que sólo él sea capaz de utilizar. Este tipo de brecha de seguridad (o **puerta trasera**) era el que se ilustraba en la película *Juegos de guerra*. Por ejemplo, el código puede tratar de detectar un ID de usuario o una contraseña específicos y, al detectarlo, evitar los procedimientos de seguridad normales. Se han dado casos de programadores que fueron condenados por estafar a los bancos incluyendo errores de redondeo en su código y haciendo que esas fracciones de céntimo fueran abonadas en sus cuentas. Estos abonos podían llegar a acumular grandes sumas de dinero, considerando el gran número de transacciones que un banco ejecuta.

Podría incluirse una puerta trasera inteligente dentro del propio compilador. El compilador generaría código objeto estándar junto con la puerta trasera, independientemente de qué código puente se estuviera compilando. Esta actividad es particularmente peligrosa, ya que un análisis del código fuente del programa no permitiría revelar ningún problema. Sólo el código fuente del compilador contendría la información necesaria para detectar la puerta trasera.

Las puertas traseras plantean un difícil problema porque, para detectarlas, tenemos que analizar todo el código fuente de todos los componentes de un sistema. Puesto que los sistemas software pueden estar compuestos por millones de líneas de código, este análisis no se lleva a cabo frecuentemente, y en la mayoría de los casos no se lleva a cabo nunca.

15.2.3 Bomba lógica

Considere un programa capaz de iniciar un incidente de seguridad sólo cuando se dan determinadas circunstancias. Sería difícil de detectar porque, en condiciones normales de operación, no existiría ningún agujero de seguridad. Sin embargo, cuando se satisficiera un conjunto predefinido de parámetros, se crearía el agujero de seguridad. Este escenario se conoce con el nombre de **bomba lógica**. Un programador, por ejemplo, podría escribir código para detectar si todavía con-

tinúa contratado por la empresa; en caso de que dicha comprobación fallara, podría crearse un demonio para permitir el acceso remoto, o podría iniciarse un determinado fragmento de código que provocara algún tipo de daño en la instalación.

15.2.4 Desbordamiento de pila y de búfer

El ataque por desbordamiento de pila o de búfer es la forma más común para que un atacante externo al sistema, a través de una conexión de red o de acceso telefónico, obtenga acceso no autorizado al sistema objetivo. Los usuarios autorizados del sistema también pueden utilizar este tipo de ataque para escalar sus privilegios.

Esencialmente, lo que el ataque hace es aprovechar un error de un programa. El error puede deberse a un simple caso de mala práctica de programación, en el que el programador se olvidó de incluir en el código comprobaciones del límite para un determinado caso de entrada. En este caso, el atacante envía más datos de los que el programa está esperando. Utilizando un método de prueba y error, o examinando el código fuente del programa atacado, si es que está disponible, el atacante determina la vulnerabilidad y escribe un programa para hacer lo siguiente:

1. Desbordar un campo de entrada, un argumento de línea de comandos o un búfer de entrada (por ejemplo, en un demonio de red) hasta escribir en la zona correspondiente a la pila.
2. Sobreescribir la dirección actual de retorno de la pila, sustituyéndola por la dirección de los códigos de ataque cargados en el paso 3.
3. Escribir un fragmento simple de código en el siguiente espacio de la pila, que incluye los comandos que el atacante quiera ejecutar, como por ejemplo arrancar un programa *shell*.

El resultado de la ejecución de este programa de ataque será una *shell* raíz o la ejecución de otro comando privilegiado.

Por ejemplo, si un formulario de página web espera que se introduzca un nombre de usuario en un campo, el atacante podría enviar el nombre de usuario más una serie de caracteres adicionales con el fin de desbordar el búfer y alcanzar la pila, más una nueva dirección de retorno que cargará en la pila, más el código que el atacante quiera ejecutar. Cuando la subrutina encargada de leer el búfer vuelve de su ejecución, la dirección de retorno será la correspondiente al código de ataque y ese código se ejecutará.

Analicemos un ataque de desbordamiento de búfer con más detalle. Considere el programa C simple que se muestra en la Figura 15.2. Este programa crea una matriz de caracteres de tamaño `BUFFER_SIZE` y copia el contenido del parámetro proporcionado en la línea de comandos: `argv[1]`. Mientras que el tamaño de este parámetro sea inferior a `BUFFER_SIZE` (necesitamos un byte para almacenar el terminador nulo), este programa funcionará adecuadamente. Pero considere lo que sucede si el parámetro proporcionado en la línea de comandos tiene una longitud mayor que `BUFFER_SIZE`. En este caso, la función `strcpy(1)` empezará a copiar desde `argv[1]` hasta que encuentre un terminador nulo (\0) o hasta que le programa sufra un fallo catastrófico. Así, este programa sufre de un problema potencial de desbordamiento de búfer en el que los datos copiados desbordarán la matriz `buffer`.

Observe que un programador cuidadoso podría haber realizado una comprobación de los límites de tamaño de `argv[1]` utilizando la función `strncpy(1)` en lugar de `strcpy(1)`, sustituyendo la línea “`strcpy(buffer, argv[1]);`” por “`strncpy(buffer, argv[1], sizeof(buffer) - 1);`”. Desafortunadamente, las comprobaciones adecuadas de los límites constituyen la excepción, más que la norma.

Además, la falta de comprobaciones de límites no es la única causa posible del comportamiento del programa de la Figura 15.2. El programa podría haber sido diseñado expresamente para comprometer la integridad del sistema. Vamos a considerar ahora las posibles vulnerabilidades de seguridad derivadas de un desbordamiento de búfer.

Cuando se invoca una función en una arquitectura informática típica, las variables definidas localmente a la función (conocidas en ocasiones con el nombre de **variables automáticas**), los parámetros pasados a la función y la dirección a la que volverá el control cuando la función ter-

```

#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];

    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}

```

Figura 15.2 Programa C con una condición de desbordamiento de búfer.

mine se almacenan en una **entrada de pila**. En la Figura 15.3 se muestra la disposición de una entrada de pila típica. Examinando la entrada de pila de arriba hacia abajo, vemos primero los parámetros pasados a la función, seguidos por las variables automáticas declaradas en la función. Despues se encuentra el **puntero de entrada de pila**, que es la dirección del comienzo de la entrada de pila. Finalmente, tenemos la dirección de retorno, que determina a dónde hay que devolver el control una vez que la función termine. El puntero de entrada de pila debe guardarse en la pila, ya que el puntero de pila puede variar durante la llamada a la función; el puntero de entrada de pila permite el acceso relativo a los parámetros de las variables automáticas.

Dada esta disposición estándar en memoria, un pirata informático podría ejecutar un ataque por desbordamiento de búfer. Su objetivo sería sustituir la dirección de retorno de la entrada de pila de modo que ahora apuntara al segmento de código que contenga el programa atacante.

El programador escribiría primero un pequeño segmento de código como el siguiente:

```

#include <stdio.h>

int main(int argc, char *argv[])
{
    ... execvp('/bin/sh', '/bin \sh', NULL);
    return 0;
}

```

Utilizando la llamada al sistema `execvp()`, este segmento de código crea un proceso *shell*. Si el programa que se está atacando se ejecuta con permisos globales del sistema, esta *shell* recién

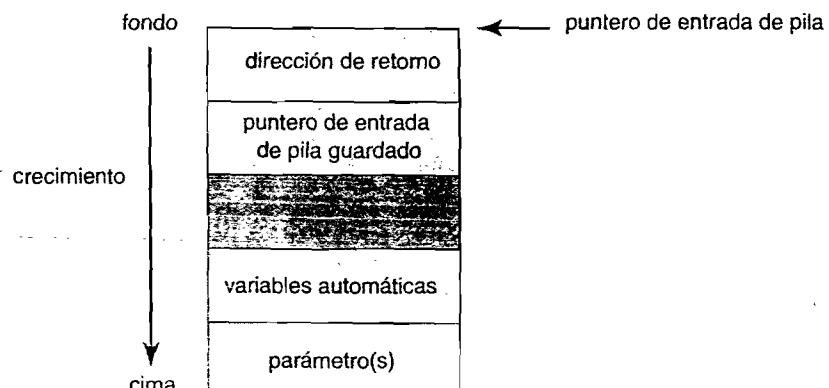


Figura 15.3 Estructura de una entrada de pila típica.

creada obtendría un acceso completo al sistema. Por supuesto, el segmento de código podría hacer cualquier cosa que esté permitida por los privilegios del proceso atacado. Después, se compila este segmento de código para poder modificar las instrucciones en lenguaje ensamblador. La principal modificación consiste en eliminar todas las partes de código innecesarias, reduciendo el tamaño de código para que pueda caber en una entrada de pila. Este fragmento de código ensamblado será ahora una secuencia binaria que constituirá el corazón del ataque.

Consulte de nuevo el programa mostrado en la Figura 15.2. Supongámos que cuando se invoca la función `main()` en dicho programa la entrada de pila tiene el aspecto que se muestra en la Figura 15.4(a). Utilizando un depurador, el programador averigua la dirección de `buffer[0]` dentro de la pila. Dicha dirección es la ubicación del código que el atacante quiere ejecutar, por lo que se añade a la secuencia binaria la cantidad necesaria de instrucciones NO-OP (lo que quiere decir NO-OPeration) para llenar la entrada de pila hasta alcanzar la ubicación de la dirección de retorno; a continuación se añade la nueva dirección de retorno, que será la ubicación de `buffer[0]`. El ataque estará completo cuando el atacante proporcione como entrada al proceso esta secuencia binaria que ha diseñado. El proceso copiará entonces la secuencia binaria desde `argv[1]` a la posición `buffer[0]` en la entrada de pila. Ahora, cuando el control vuelva desde `main()`, en lugar de volver a la ubicación especificada por el antiguo valor de la dirección de retorno, volverá al código de *shell* modificado, que se ejecutará con los derechos de acceso del proceso atacado. La Figura 15.4(b) contiene el código *shell* modificado.

Hay muchas formas de tratar de aprovechar los problemas potenciales de desbordamiento de búfer. En este ejemplo, hemos considerado la posibilidad de que el programa que está siendo atacado (el código mostrado en la Figura 15.2) se estuviera ejecutando con permisos globales del sistema. Sin embargo, el segmento de código que se ejecute una vez modificado el valor de la dirección de retorno podría realizar cualquier tipo de acto malicioso, como borrar archivos, abrir puertos de red para realizar ataques ulteriores, etc.

Este ataque de ejemplo por desbordamiento de búfer revela que hacen falta unos grandes conocimientos y habilidades de programación para poder reconocer el código atacable y diseñar el ataque adecuado. Desafortunadamente, para lanzar ataques de seguridad no hace falta ser un gran programador: un pirata informático podría determinar el error existente en un cierto programa y escribir un módulo de ataque. Cualquiera que tenga unos conocimientos informáticos rudimentarios y que se haga con ese código de ataque (los denominados *script kiddie*, niños script) podrían tratar de lanzar el ataque contra determinados sistemas objetivo.

El ataque por desbordamiento de búfer resulta especialmente pernicioso porque pueden lanzarse ataques de un sistema a otro y ese código de ataque puede ser transmitido a través de los

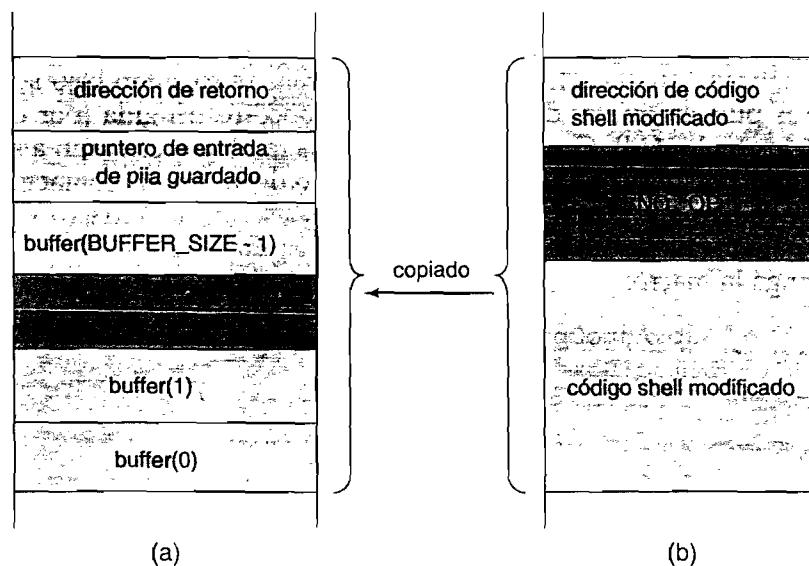


Figura 15.4 Entrada de pila hipotética para la Figura 15.2, (a) antes y (b) después.

canales de comunicaciones permitidos. Dichos ataques pueden realizarse utilizando los protocolos normales que se utilicen para comunicarse con la máquina objetivo, por lo que pueden resultar difíciles de detectar y prevenir. Esos ataques pueden incluso puentejar los mecanismos de seguridad proporcionados por los cortafuegos (Sección 15.7).

Una solución a este problema es que la CPU disponga de una característica que no permita la ejecución de código contenido en una sección de pila de la memoria. Las versiones recientes del chip SPARC de Sun incluyen este tipo de mecanismo de seguridad y las versiones más recientes de Solaris utilizan ese mecanismo. Puede seguirse modificando la dirección de retorno de la rutina desbordada, pero cuando la dirección de retorno apunta a una ubicación dentro de la propia pila y se intenta ejecutar el código almacenado en ella, se genera una excepción y el programa se detiene con un error.

Las versiones recientes de los chips AMD e Intel x86 incluyen la funcionalidad NX para impedir este tipo de ataque. El uso de esta funcionalidad está soportado en varios sistemas operativos para x86, incluyendo Linux y Windows XP SP2. La implementación hardware implica el uso de un nuevo bit dentro de las tablas de páginas de la CPU. Este bit marca la página asociada como no ejecutable, impidiendo leer y ejecutar instrucciones desde ella. A medida que esta característica sea más utilizada, los ataques por desbordamiento de búfer deberían disminuir significativamente.

15.2.5 Virus

Otro tipo de amenaza en forma de programa son los **virus**. Los virus son auto-replicantes y están diseñados para “infectar” otros programas. Pueden causar estragos en un sistema modificando o destruyendo archivos y provocando funcionamientos inadecuados de los programas y fallos catastróficos del sistema. Un virus es el fragmento de código integrado dentro de un programa legítimo. Al igual que la mayoría de los ataques de penetración, los virus son muy específicos de las arquitecturas, de los sistemas operativos y de las aplicaciones. Los virus constituyen un problema especialmente grave para los usuarios de máquina de tipo PC. UNIX y otros sistemas operativos multiusuario no son, generalmente, susceptibles a los virus, porque los programas ejecutables están protegidos contra escritura por el propio sistema operativo. Incluso si un virus llega a infectar a uno de esos programas, sus poderes están usualmente limitados, porque otros aspectos del sistema están también protegidos.

Los virus suelen propagarse a través de correo electrónico, siendo el correo basura el vector más común. También pueden propagarse cuando los usuarios descargan programas infectados desde servicios de compartición de archivos de Internet o cuando intercambian discos infectados.

Otra forma común de transmisión de virus utiliza los archivos Microsoft Office, como por ejemplo los documentos de Microsoft Word. Estos documentos pueden contener *macros* (o programas Visual Basic) que los programas del paquete Office (Word, PowerPoint y Excel) ejecutarán automáticamente. Puesto que estos programas se ejecutan bajo la propia cuenta del usuario, las macros pueden operar de forma bastante poco restringida (por ejemplo, cerrando archivos del usuario a voluntad). Comúnmente, los virus también se envían a sí mismos por correo electrónico a otras personas que se encuentren dentro de la lista de contacto del usuario. He aquí un ejemplo de código que muestra la simplicidad de escribir una macro Visual Basic que un virus podría usar para formatear el disco duro de una computadora Windows en cuanto se abra el archivo que contenga la macro:

```
Sub AutoOpen()
Dim oFS
Set oFS = CreateObject("Scripting.FileSystemObject")
vs = Shell("c:
command.com /k format c:='',vbHide)
End Sub
```

¿Cómo funcionan los virus? Una vez que un virus alcance una máquina objetivo, un programa conocido como **lanzador de virus** inserta el virus en el sistema. El lanzador de virus es usualmente un caballo de Troya, que se ejecuta por otras razones pero cuya principal actividad consiste en

instalar el virus. Una vez instalado, el virus puede hacer una de varias cosas. Existen literalmente miles de virus distintos, pero se los puede clasificar en varias categorías generales. Observe que muchos virus pertenecen a más de una categoría a la vez:

- **Archivo.** Un virus de archivo estándar infecta un sistema insertándose a un archivo y modificando el inicio del programa para que la ejecución salte al código del virus. Después de ejecutarse, el virus devuelve el control al programa para que no pueda detectarse que el virus se ha ejecutado. Los virus de archivo se conocen, en ocasiones, con el nombre de virus parásitos, ya que no dejan ningún archivo completo detrás suyo y permiten que el programa huésped siga funcionando.
- **Arranque.** Un virus de arranque infecta el sector de arranque del sistema, ejecutándose cada vez que el sistema se arranca y antes de que se cargue el sistema operativo. El virus busca otros soportes de arranque (es decir, disquetes) y también los infecta. Este virus también se conoce con el nombre de virus de memoria, porque no aparecen en el sistema de archivos. La Figura 15.5 muestra cómo funciona un virus de arranque.
- **Macro.** La mayoría de los virus están escritos en un lenguaje de bajo nivel, como por ejemplo ensamblador o C. Los virus de macro están escritos en un lenguaje de alto nivel, como Visual Basic. Estos virus se activan cuando se inicia un programa capaz de ejecutar

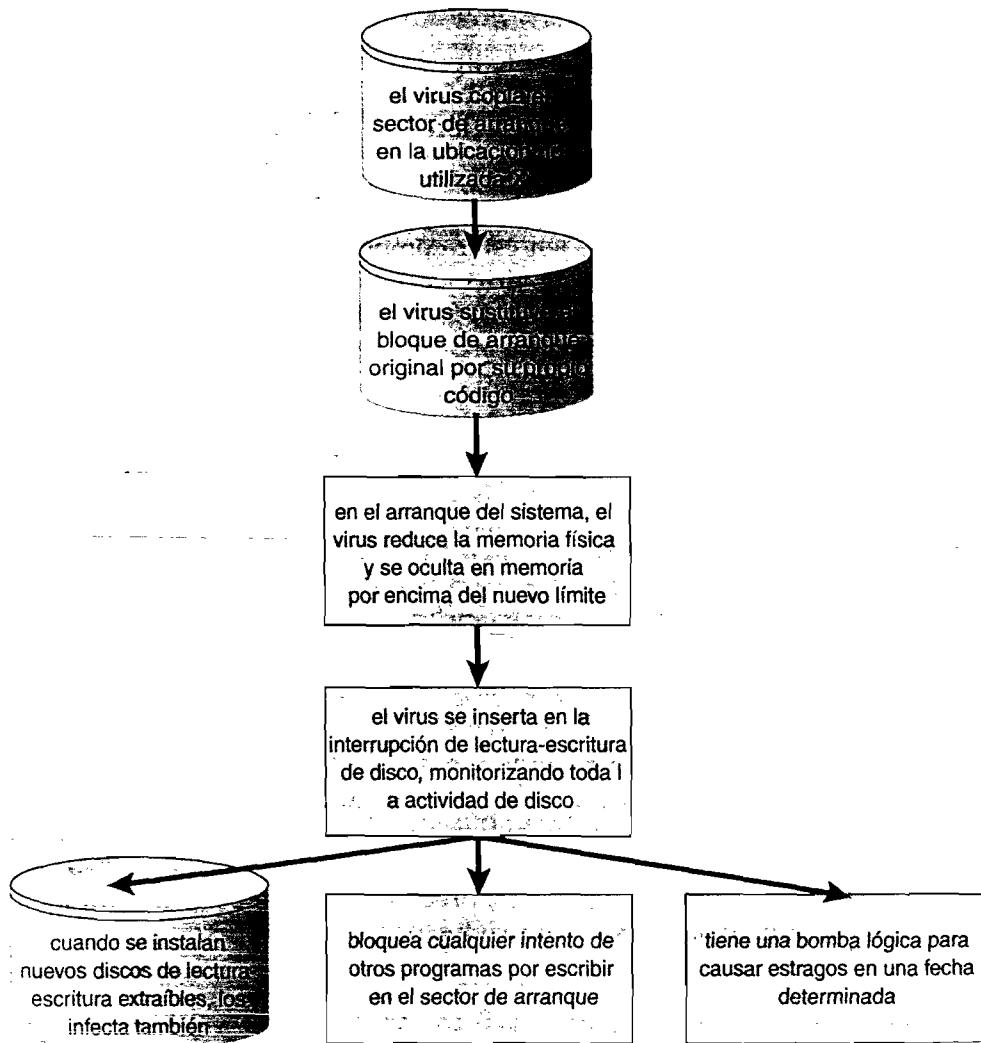


Figura 15.5 Un virus informático que afecta al sector de arranque.

la macro: Por ejemplo, un virus de macro podría estar contenido en un archivo de hoja de cálculo.

- **Código fuente.** Un virus de código fuente busca código fuente y lo modifica para incluir el virus y ayudar a su distribución.
- **Polimórfico.** Este tipo de virus cambia cada vez que se instala, para evitar su detección por parte del software antivirus. Los cambios no afectan a la funcionalidad del virus, sino que sólo modifican la **signatura del virus** es un patrón que puede usarse para identificar un virus, normalmente una serie de bytes que forman parte del código de virus.
- **Cifrado.** Un virus cifrado incluye código de descripción junto con el virus cifrado, de nuevo para evitar la detección. El virus se descifra primero y luego se ejecuta.
- **Encubierto.** Este insidioso virus trata de evitar la detección modificando partes del sistema que podrían ser usadas para detectarlo. Por ejemplo, podría modificar la llamada al sistema `read` para que, si se lee el archivo que el virus ha modificado, se devuelva el código original en lugar del código infectado.
- **Túnel.** Este tipo de virus trata de evitar la detección por los programas antivirus instalándose asimismo en la cadena de rutinas de tratamiento de interrupciones. Otros virus similares se instalan en los controladores de dispositivo.
- **Multiparte.** Los virus de este tipo son capaces de infectar múltiples partes de un sistema, incluyendo los sectores de arranque, la memoria y los archivos. Esto hace que sea difícil detectarlos y evitar su propagación.
- **Acorazado.** Los virus acorazados están codificados de tal manera que resultan difíciles de desentrañar y de comprender por parte de los investigadores que desarrollan programas antivirus. También pueden estar comprimidos para evitar la detección y la desinfección. Además, los lanzadores de virus y otros archivos complejos que forman parte de un determinado virus de este tipo suelen frecuentemente ocultarse utilizando los atributos de archivos o empleando nombres de archivo no visualizables.

Esta amplia variedad de virus es probable que continúe creciendo. De hecho, en 2004 se detectó un nuevo virus de amplia distribución, que aprovechaba tres errores diferentes para operar. El virus comenzó infectando centenares de servidores Windows (incluyendo muchos sitios de confianza) que ejecutaban Microsoft Internet Information Server (IIS). Todo explorador web Microsoft Explorer vulnerable que visitara esos sitios recibía un virus de explorador con cada descarga. El virus de explorador instalaba varios programas de puerta trasera, incluyendo un **registrador de pulsaciones de teclas**, que registraba todo lo que se introdujera a través del teclado (incluyendo contraseñas y números de tarjetas de crédito). También instalaba un demonio para permitir un acceso remoto no limitado a los intrusos y otro que permitía al intruso distribuir correo basura a través de la máquina de escritorio infectada.

Generalmente, los virus son el tipo de ataque de seguridad más dañino; y como son bastante efectivos, continuarán desarrollándose y distribuyéndose. Uno de los más activos debates dentro de la comunidad informática es si la **monocultura**, es decir, esa situación en la que muchos sistemas ejecutan el mismo hardware, el mismo sistema operativo y/o el mismo software de aplicación, están incrementando el grado de amenaza y el nivel de daños provocados por las intrusiones de seguridad. Parte de ese debate es, precisamente, la cuestión de si existe o no en la actualidad una monocultura (compuesta por productos Microsoft).

15.3 Amenazas del sistema y de la red

Las amenazas basadas en programas utilizan típicamente un fallo en los mecanismos de protección de un sistema para atacar a los programas. Por contraste, las amenazas del sistema y de la

red implican el abuso de los servicios y de las conexiones de red. En ocasiones, se utiliza un ataque del sistema y de la red para lanzar un ataque de programa, y viceversa.

Las amenazas del sistema y de la red crean una situación en la que se utilizan inapropiadamente los recursos del sistema operativo y los archivos del usuario. En esta sección vamos a analizar algunos ejemplos de estas amenazas, incluyendo los gusanos, el escaneo de puertos y los ataques por denegación de servicio.

Es importante destacar que las mascaradas y los ataques por reproducción también resultan comunes en las redes que interconectan los sistemas. De hecho, estos ataques son más efectivos y más difíciles de contrarrestar cuando están implicados múltiples sistemas. Por ejemplo, dentro de una computadora, el sistema operativo puede determinar, usualmente, el emisor y el receptor de un mensaje. Incluso si el emisor adopta el ID de alguna otra persona, puede que exista un registro de dicho cambio de ID. Cuando están implicados múltiples sistemas, especialmente sistemas que son controlados por los atacantes, realizar esa labor de traza resulta mucho más difícil.

La generalización de este concepto es que el compartir secretos (para demostrar la identidad y en forma de claves de cifrado) es una necesidad para la autenticación del cifrado, y que esa compartición resulta más sencilla en aquellos entornos (por ejemplo con un único sistema operativo) en los que existan métodos seguros de compartición. Estos métodos incluyen la memoria compartida y los mecanismos de comunicación interprocesos. Los temas de la creación de comunicaciones seguras y de la autenticación se analizan en las Secciones 15.4 y 15.5.

15.3.1 Gusanos

Un **gusano** es un proceso que utiliza un mecanismo de **reproducción** para afectar al rendimiento del sistema. El gusano crea copias de sí mismo, utilizando recursos del sistema y en ocasiones impidiendo operar a todos los demás procesos. En las redes informáticas, los gusanos son particularmente potentes, ya que pueden reproducirse de un sistema a otro y colapsar una red completa. Esto fue lo que sucedió en 1988 con los sistemas UNIX de Internet, provocando millones de dólares de pérdidas en tiempo de detección de los sistemas y en tiempo dedicado por los administradores de los mismos.

Al finalizar la jornada del 2 de noviembre de 1988, Robert Tappan Morris Jr., un estudiante de segundo ciclo de Cornell, lanzó un programa gusano en una o más máquinas *host* conectadas en Internet. Teniendo por objetivo a las estaciones de trabajo Sun 3 de Sun Microsystems y a las computadoras VAX que ejecutaban variantes del UNIX BSD versión 4, el gusano se expandió rápidamente, atravesando grandes distancias; en unas pocas horas después de su lanzamiento, había llegado a consumir los recursos de los sistemas hasta el punto de provocar el colapso de las máquinas infectadas.

Aunque Robert Morris diseñó el programa auto-replicante para su rápida reproducción y distribución, son algunas de las características del entorno de comunicación por red de UNIX las que proporcionaron los medios para la propagación del gusano en los distintos sistemas. Probablemente, Morris eligió para la infección inicial algún *host* Internet que hubiera sido dejado abierto para el acceso por parte de usuarios externos. Desde allí, el programa gusano aprovechó una serie de fallos en las rutinas de seguridad del sistema operativo UNIX y se dedicó a emplear algunas utilidades UNIX que simplificaban la compartición de recursos en las redes de área local, con el fin de obtener acceso no autorizado a miles de otros sitios conectados a la red. Los métodos de ataque de Morris son los que a continuación se esbozan.

El gusano estaba compuesto de dos programas. Un **vector** (también denominado **arranque**) y el programa principal. Denominado *l1.c*, el vector estaba compuesto de 99 líneas de código C compilado y se ejecutaba en cada máquina a la que se accedía. Una vez establecido en el sistema informático objeto del ataque, el vector se conectaba a la máquina donde se había originado y cargaba una copia del gusano original en el sistema *atacado* (Figura 15.6). El programa principal se dedicaba entonces a buscar otras máquinas con las que el sistema recién infectado pudiera conectarse fácilmente. Para realizar estas acciones, Morris aprovechó la utilidad de red UNIX *rsh* para la ejecución remota de tareas. Definiendo archivos especiales que enumeran parejas de nombres *host*-inicio de sesión, los usuarios pueden evitarse introducir una contraseña cada vez que accedan a

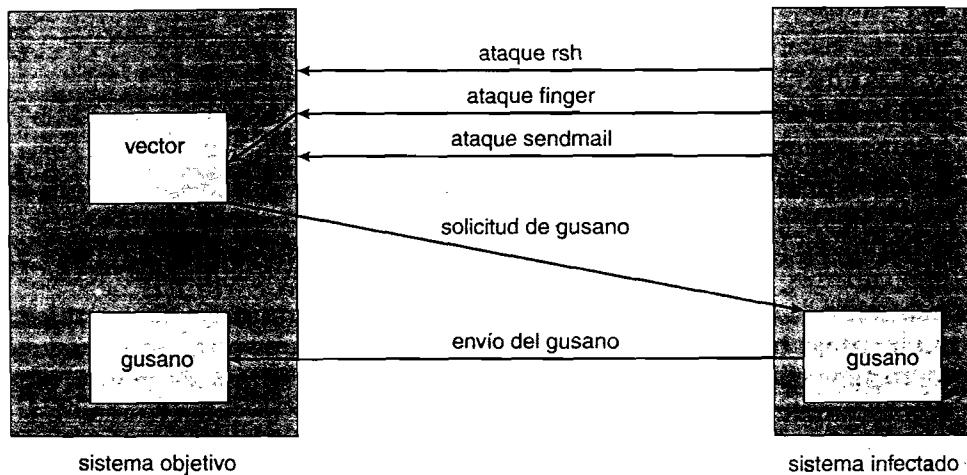


Figura 15.6 El gusano Internet de Morris.

una cuenta remota contenida en esa lista de parejas. El gusano analizaba estos archivos especiales en busca de nombres de sitios que permitieran la ejecución remota sin una contraseña. Allí donde podía establecerse una *shell* remota, se cargaba el programa gusano y comenzaba a ejecutarse.

El ataque a través del mecanismo de acceso remoto era sólo uno de los tres métodos de infección incluidos en el gusano. Los otros dos aprovechaban errores del sistema operativo en los programas finger y sendmail de UNIX.

La utilidad finger funciona como un listín telefónico electrónico; el comando

```
finger nombre-usuario@nombrehost
```

devuelve el nombre real de una persona y sus credenciales de inicio de sesión, junto con algunas otras informaciones que el usuario pueda haber proporcionado, como por ejemplo la dirección del trabajo y del domicilio, el número telefónico, sus intereses de investigación o alguna cita célebre. Finger se ejecuta como un proceso de segundo plano (o demonio) en cada nodo BSD y responde a las consultas a través de Internet. El gusano ejecutaba un ataque de desbordamiento de búfer sobre finger. El programa consultaba finger con una cadena de 536 bytes diseñada para exceder el búfer asignado para la entrada y para sobreescibir la entrada de la pila. En lugar de volver a la rutina principal en la que estuviera antes de la llamada efectuada por Morris, el demonio finger devolvía el control a un procedimiento contenido dentro de la cadena de 536 bytes invasora que ahora residía en la pila. El nuevo procedimiento ejecutaba /bin/sh, que en caso de iniciarse satisfactoriamente proporcionaba al gusano una *shell* remota en la máquina objeto del ataque.

El error que se aprovechaba en sendmail también implicaba la utilización de un proceso demonio para efectuar una entrada de datos maliciosa. sendmail envía, recibe y encamina correo electrónico. El código de depuración contenido en esta utilidad permite a los desarrolladores verificar y mostrar el estado del sistema de correo. La opción de depuración resultaba muy útil para los administradores de sistemas, por lo que normalmente se la dejaba activada. Morris incluyó en su arsenal de ataques una llamada a debug (el programa de depuración) que, en lugar de especificar una dirección de usuario (como se haría normalmente durante las pruebas) contenía un conjunto de comandos que enviaban por correo y ejecutaban una copia del programa vector.

Una vez instalado, el gusano principal realizaba una serie de intentos sistemáticos de descubrir las contraseñas de los usuarios. Comenzaba probando los casos simples de que no se utilice ninguna contraseña o de que las contraseñas estuvieran formadas por combinaciones de los nombres de la cuenta y del usuario. Después, utilizaba comparaciones con un diccionario interno de 432 contraseñas favoritas que muchos usuarios suelen elegir y luego entraba en una base final en la que se probaba cada una de las palabras del diccionario estándar en línea de UNIX como posible contraseña. Este complejo y eficiente algoritmo de averiguación de contraseñas formado por tres pasos permitía al gusano obtener acceso a otras cuentas de usuario del sistema infectado. El

gusano buscaba entonces archivos de datos `rsh` en esas cuentas recién capturadas y los usaba como se ha descrito anteriormente para tener acceso a cuentas de usuario en sistemas remotos.

Con cada nuevo acceso, el programa gusano buscaba copias de sí mismo que ya pudieran existir. Si encontraba una, la nueva copia terminaba su ejecución, excepto en uno de cada siete casos. Si el gusano hubiera terminado su ejecución siempre que se detectaran copias duplicadas, podría haber llegado a no ser detectado nunca. Sin embargo, al permitir que uno de cada siete duplicados continuara ejecutándose (posiblemente para tratar de derrotar los esfuerzos para evitar su diseminación ejecutando falsos gusanos) se creó una infección masiva en los sistemas Sun y VAX de Internet.

Las mismas características de los entornos de red UNIX que activaron la propagación del gusano sirvieron también para detener su avance. La facilidad de las comunicaciones electrónicas, los mecanismos para copiar archivos fuente y archivos binarios en máquinas remotas y el acceso tanto al código fuente como a la experiencia humana permitieron un esfuerzo cooperativo para desarrollar soluciones rápidamente. Al llegar la noche del día siguiente al comienzo de la infección, el 3 de noviembre, una serie de métodos para detener al programa invasor fueron distribuidos a todos los administradores de sistemas a través de Internet. En sólo unos días, ya había disponibles parches software específicos para resolver los fallos de seguridad que el gusano aprovechaba.

¿Por qué lanzó Morris ese gusano? Se ha atribuido su acción tanto a una broma inocente que se le fue de las manos como a un intento criminal deliberado. Teniendo en cuenta la complejidad que requiere iniciar ese ataque, resulta bastante poco probable que el lanzamiento del gusano o su gran distribución no fueran intencionados. El programa gusano llevaba a cabo una serie de pasos muy elaborados para tratar de no dejar huella y para contrarrestar los esfuerzos dirigidos a detener su diseminación. Sin embargo, el programa no contenía ningún código capaz de dañar o destruir los sistemas en que se ejecutaba. El autor tenía, claramente, la experiencia necesaria para haber incluido tales comandos; de hecho, el código de arranque tenía estructuras de datos que podrían haber sido usadas perfectamente para transferir caballos de Troya o virus informáticos. El comportamiento del programa nos permite realizar diversas observaciones de interés, pero no nos proporciona la suficiente información como para deducir los motivos que alentaban al creador de este gusano. Lo que no deja, de todos modos, lugar a la especulación es el resultado legal del caso: los jueces consideraron culpable a Morris y le condenaron a tres años de cárcel, a la realización de 400 horas de servicio a la comunidad y a una multa de 10.000 dólares. Los gastos legales del propio Morris probablemente excedieron de los 100.000 dólares.

Los expertos en seguridad continúan elaborando métodos para reducir o eliminar el riesgo de aparición de gusanos. Sin embargo, un caso más reciente muestra que los gusanos siguen proliferando en Internet y también que, a medida que Internet crece, el daño que incluso los gusanos "inocuos" pueden realizar también crece y puede ser significativo. Este ejemplo tuvo lugar durante agosto de 2003, cuando una serie de personas todavía desconocidas liberaron la quinta versión del gusano "Sobig", más concretamente conocida con el nombre de "W32.Sobig.F@mm". Este ha sido el gusano de más rápida distribución que se ha lanzado hasta la fecha, y que infectó en su momento de máxima actividad a centenares de miles de computadoras y a uno de cada 17 mensajes de correo electrónico que se intercambiaron a través de Internet. El gusano colapsó las bandejas de entrada de correo electrónico, ralentizó las redes de comunicaciones e hizo que se tuviera que dedicar una enorme cantidad de horas a tareas de limpieza.

Sobig.F fue lanzado cargándolo en un grupo de noticias pornográfico a través de una cuenta creada con una tarjeta de crédito robada. El gusano estaba oculto en una fotografía. El virus contenido en el gusano tomaba como objetivo los sistemas Microsoft Windows y utilizaba su propio motor SMTP para enviarse a sí mismo por correo electrónico a todas las direcciones que se encontrarán en el sistema infectado. El gusano utilizaba diversas líneas de tema para evitar su detección, incluyendo "Thank You!", "Your details" y "Re: Approved". También utilizaba una dirección aleatoria extraída del *host* como dirección de origen del correo electrónico, lo que hacía difícil determinar a partir del propio mensaje cuál era la máquina que estaba actuando como origen de la infección. Sobig.F incluía un adjunto para que el lector del correo electrónico pulsara en él, de nuevo con una variedad de nombres. Si se ejecutaba este adjunto, éste almacenaba un programa

denominado WINPPR32.EXE en el directorio Windows predeterminado, junto con un archivo de texto y también modificaba el registro de Windows.

El código incluido en el adjunto también estaba programado para tratar periódicamente de conectarse a uno de un conjunto de veinte servidores y para descargar y ejecutar un programa desde ellos. Afortunadamente, esos servidores fueron desactivados antes de que se pudiera descargar el código. Todavía no se ha podido determinar el contenido del programa que se hubiera descargado de estos servidores. Si dicho código fuera malintencionado, el resultado podría haber sido un considerable daño en un enorme número de máquinas.

15.3.2 Escaneo de puertos

El escaneo de puertos no es un ataque, sino más bien un método para que los piratas informáticos detecten las vulnerabilidades del sistema que puedan ser atacadas. El escaneo de puertos se realiza normalmente de forma automatizada, lo que implica utilizar una herramienta que trate de crear una conexión TCP/IP a un puerto o rango de puertos específicos. Por ejemplo, suponga que existe una vulnerabilidad (o error) conocida en sendmail. Un pirata informático podría ejecutar un escáner de puertos para tratar de conectarse a, por ejemplo, el puerto 25 de un sistema o rango de sistemas concretos. Si la conexión tiene éxito, el pirata (la herramienta) podría tratar de comunicarse con el servicio que responda para determinar si se trata de sendmail y, en caso afirmativo, si la versión es la que tiene ese error conocido.

Ahora, imagínese una herramienta en la que se hubieran codificado todos los errores conocidos de todos los servicios de todos los sistemas operativos. Esa herramienta podría tratar de conectarse a todos los puertos de uno o más sistemas y, para cada servicio que respondiera, podría tratar de utilizar cada uno de los errores conocidos. Frecuentemente, esos errores son desbordamientos de búfer, que permiten la creación de una *shell* de comandos privilegiada en el sistema. A partir de ahí, por supuesto, el atacante podría instalar caballos de Troya, programas de puerta trasera, etc.

No existe ninguna herramienta de ese tipo, pero sí que hay herramientas que ofrecen un subconjunto de dicha funcionalidad. Por ejemplo, nmap (de <http://www.insecure.org/nmap/>) es una utilidad de código abierto muy versátil para la exploración de red y la auditoría de seguridad. Cuando se la apunta a un puerto objetivo, determina qué servicios se están ejecutando, incluyendo los nombres y versiones de las aplicaciones. Puede determinar el sistema operativo *host* y también puede proporcionar información acerca de las defensas, como por ejemplo qué cortafuegos están defendiendo ese objetivo. Esta herramienta no aprovecha ninguno de los errores conocidos.

Nessus (de <http://www.nessus.org/>) realiza una función similar, pero tiene también una base de datos de errores y de los modos de aprovecharlos en un ataque. Permite explorar un conjunto de sistemas, determinar los servicios que se están ejecutando en esos sistemas y tratar de atacar todos los errores apropiados, generando después una serie de informes acerca de los resultados. La herramienta no realiza el paso final de explorar los errores encontrados, pero un pirata informático con los suficientes conocimientos o uno de los denominados “scripts kiddie” sí que podrían llevar a cabo ese ataque.

Puesto que los escaneos de puertos son detectables (véase la Sección 15.6.3), se suelen realizar desde **sistemas zombi**. Dichos sistemas son máquinas independientes y previamente comprometidas que están prestando un servicio normal a sus propietarios al mismo tiempo que son utilizadas inadvertidamente para propósitos inconfesables, incluyendo la realización de ataques por denegación de servicio y la retransmisión de correo basura. Los zombis hacen que resulte particularmente difícil perseguir a los piratas informáticos, ya que resulta muy difícil determinar el origen del ataque y la persona que lo ha iniciado. Esta es una de las muchas razones que aconsejan dotar de seguridad también a los sistemas “no importantes”, y no sólo a los sistemas que contengan información o servicios “valiosos”.

15.3.3 Denegación de servicio

Como hemos mencionado anteriormente, los ataques DOS están dirigidos no a obtener información o a robar recursos, sino a impedir el uso legítimo de un sistema o funcionalidad. La mayoría

de los ataques de denegación de servicio afectan a sistemas en los que el atacante no ha penetrado. De hecho, lanzar un ataque que impida el uso legítimo de un sistema resulta, frecuentemente, más sencillo que irrumpir en una máquina o instalación.

Los ataques de denegación de servicio se realizan generalmente a través de la red. Se los puede clasificar en dos categorías. El primer caso es el de los ataques que consumen tantos recursos de la máquina atacada que prácticamente no puede realizarse con ella ningún trabajo útil. Por ejemplo, al hacer clic en un sitio web podría descargarse una applet Java que consumiera todo el tiempo de CPU disponible o que se dedicara a abrir una serie infinita de ventanas emergentes. El segundo caso de ataque implica hacer caer la red o la instalación; ha habido numerosos ataques de denegación de servicio de este tipo que se han realizado contra algunos sitios web muy conocidos. Este tipo de ataque es el resultado de un abuso de algunas de las funciones fundamentales de TCP/IP. Por ejemplo, si el atacante envía la parte del protocolo que dice "Quiero iniciar una conexión TCP" pero luego no sigue con el mensaje estándar que dice "La conexión está ahora establecida", el resultado puede ser un conjunto de sesiones TCP parcialmente iniciadas. Un número suficiente de estas sesiones puede consumir todos los recursos de red del sistema, incluyendo ulteriores intentos legítimos de establecer una conexión TCP. Dichos ataques, que pueden durar horas o días, han provocado en numerosas ocasiones un fallo parcial o total de los intentos de utilizar la instalación objeto del ataque. Estos ataques suelen detenerse en el nivel de red, hasta que pueden actualizarse los sistemas operativos con el fin de reducir su vulnerabilidad.

Generalmente, es imposible prevenir los ataques de denegación de servicio. Los ataques utilizan los mismos mecanismos que la operación normal. Todavía más difíciles de prevenir y de solucionar son los **ataques distribuidos de denegación de servicio** (DDOS, distributed denial-of-service attacks). Estos ataques se inician desde múltiples sitios a la vez, dirigidos hacia un objetivo común, normalmente por parte de programas zombis.

En ocasiones, un determinado sitio puede no ser ni siquiera consciente de que está siendo atacado. Puede resultar difícil determinar si la ralentización de un sistema se debe simplemente a un pico de utilización del mismo o a un ataque. Considere, por ejemplo, el caso de una campaña publicitaria que tuviera un enorme éxito y que incrementara enormemente el tráfico hacia un determinado sitio web; en ciertas condiciones, podría confundirse esa actividad con un ataque de tipo DDOS.

Hay otros aspectos interesantes de los ataques DOS que conviene resaltar. Por ejemplo, los programadores y los administradores de sistemas deben tratar de comprender a la perfección los algoritmos y las tecnologías que estén implantando. Si un algoritmo de autenticación bloquea una cuenta durante un cierto período de tiempo después de varios intentos incorrectos de inicio de sesión, un atacante podría hacer qué se bloqueara todo el mecanismo de autenticación del sistema simplemente generando una multitud de intentos incorrectos de sesión para todas las cuentas. De forma similar, un cortafuegos que bloqueara automáticamente ciertos tipos de tráfico podría ser inducido a bloquear dicho tráfico en las ocasiones en que no debiera hacerlo. Finalmente, las clases de informática son una conocida fuente de ataques DOS accidentales; considere, a este respecto, los primeros ejercicios de programación en los que los estudiantes aprenden a crear subprocesos o hebras: uno de los errores más comunes implica la creación de una serie infinita de subprocesos, que termina por consumir todos los recursos de la CPU y toda la memoria libre.

15.4 La criptografía como herramienta de seguridad

Existen muchas defensas frente a los ataques informáticos, que abarcan toda la gama que va desde la metodología a la tecnología. La herramienta de carácter más general que está a disposición de los usuarios y de los diseñadores de sistemas es la criptografía. En esta sección, vamos a explicar algunos detalles acerca de la criptografía y de su uso en el campo de la seguridad informática.

En una computadora aislada, el sistema operativo puede determinar de manera fiable quiénes son el emisor y el receptor de todas las comunicaciones interprocesos, ya que el sistema operativo controla todos los canales de comunicaciones de la computadora. En una red de computadoras, la situación es bastante distinta. Una computadora conectada a la red recibe bits desde el exterior, y no tiene ninguna forma inmediata y fiable de determinar qué máquina o aplicación ha

enviado esos bits. De forma similar, la propia computadora envía bits hacia la red sin tener ninguna forma de determinar quién puede llegar a recibirlos.

Comúnmente, se utilizan las direcciones de red para inferir los emisores y receptores potenciales de los mensajes que circulan por la red. Los paquetes de red llegan con una dirección de origen, como por ejemplo una dirección IP. Y cuando una computadora envía un mensaje, indica quién es el receptor pretendido del mismo especificando una dirección de destino. Sin embargo, para aquellas aplicaciones en las que la seguridad tenga importancia, correríamos el riesgo de meternos en problemas si asumíramos que la dirección de origen o de destino de un paquete permiten determinar con fiabilidad quién ha enviado o recibido dicho paquete. Una computadora maliciosa podría enviar un mensaje con una dirección de origen falsificada y, asimismo, otras muchas computadoras distintas de la especificada por la dirección de destino podrían (y normalmente lo hacen) recibir un paquete. Por ejemplo, todos los encaminadores situados en la ruta hacia el destino recibirán también el paquete. ¿Cómo puede, entonces, decidir el sistema operativo si debe conceder una solicitud, cuando no puede confiar en el origen especificado en dicha solicitud? ¿Y cómo se supone que debe proporcionar protección para una solicitud o para un conjunto de datos, cuando no puede determinar quién recibirá la respuesta o el contenido del mensaje que envíe a través de la red?

Generalmente, se considera impracticable construir una red (de cualquier tamaño) en la que se pueda "confiar" en este sentido en las direcciones de origen y destino de los paquetes. Por tanto, la única alternativa es eliminar, de alguna manera, la necesidad de confiar en la red; este es el trabajo de la criptografía. Desde un punto de vista abstracto, la **criptografía** se utiliza para restringir los emisores y/o receptores potenciales de un mensaje. La criptografía moderna se basa en una serie de secretos, denominados **claves**, que se distribuyen selectivamente a las computadoras de una red y se utilizan para procesar mensajes. La criptografía permite al receptor de un mensaje verificar que el mensaje ha sido creado por alguna computadora que posee una cierta clave: esa clave es el *origen* del mensaje. De forma similar, un emisor puede codificar su mensaje de modo que sólo una computadora que disponga de una cierta clave pueda decodificar el mensaje, de manera que esa clave se convierte en el *destino*. Sin embargo, a diferencia de las direcciones de red, las claves están diseñadas de modo que no sea computacionalmente factible calcularlas a partir de los mensajes que se hayan generado con ellas, ni a partir de ninguna otra información pública. Por tanto, las claves proporcionan un medio mucho más fiable de restringir los emisores y receptores de los mensajes. Observe que la criptografía es un campo de estudio completo por derecho propio, con una gran complejidad; aquí, vamos a explorar únicamente los aspectos más importantes de aquellas partes de la criptografía que se relacionan con los sistemas operativos.

15.4.1 Cifrado

Puesto que ayuda a resolver una amplia variedad de problemas de seguridad de las comunicaciones, el cifrado se utiliza frecuentemente en muchos aspectos de la informática moderna. El cifrado es un medio de restringir los posibles receptores de un mensaje. Un algoritmo de cifrado permite al emisor de un mensaje garantizar que sólo pueda leer el mensaje una computadora que posea una cierta clave. El cifrado de mensajes es una práctica muy antigua, por supuesto, y han existido muchos algoritmos de cifrado, anteriores incluso a la época de César. En esta sección, vamos a describir los algoritmos y principios modernos más importantes del cifrado.

La Figura 15.7 muestra un ejemplo de dos usuarios que se comunican de manera segura a través de un canal inseguro. A lo largo de toda la sección haremos referencia a esta figura. Observe que el intercambio de claves puede tener lugar directamente entre las dos partes o a través de una tercera parte de confianza (es decir, una autoridad de certificación) como se explica en la Sección 15.4.1.4.

Un algoritmo de cifrado consta de los siguientes componentes:

- Un conjunto K de claves.
- Un conjunto M de mensajes.
- Un conjunto C de mensajes de texto cifrado.

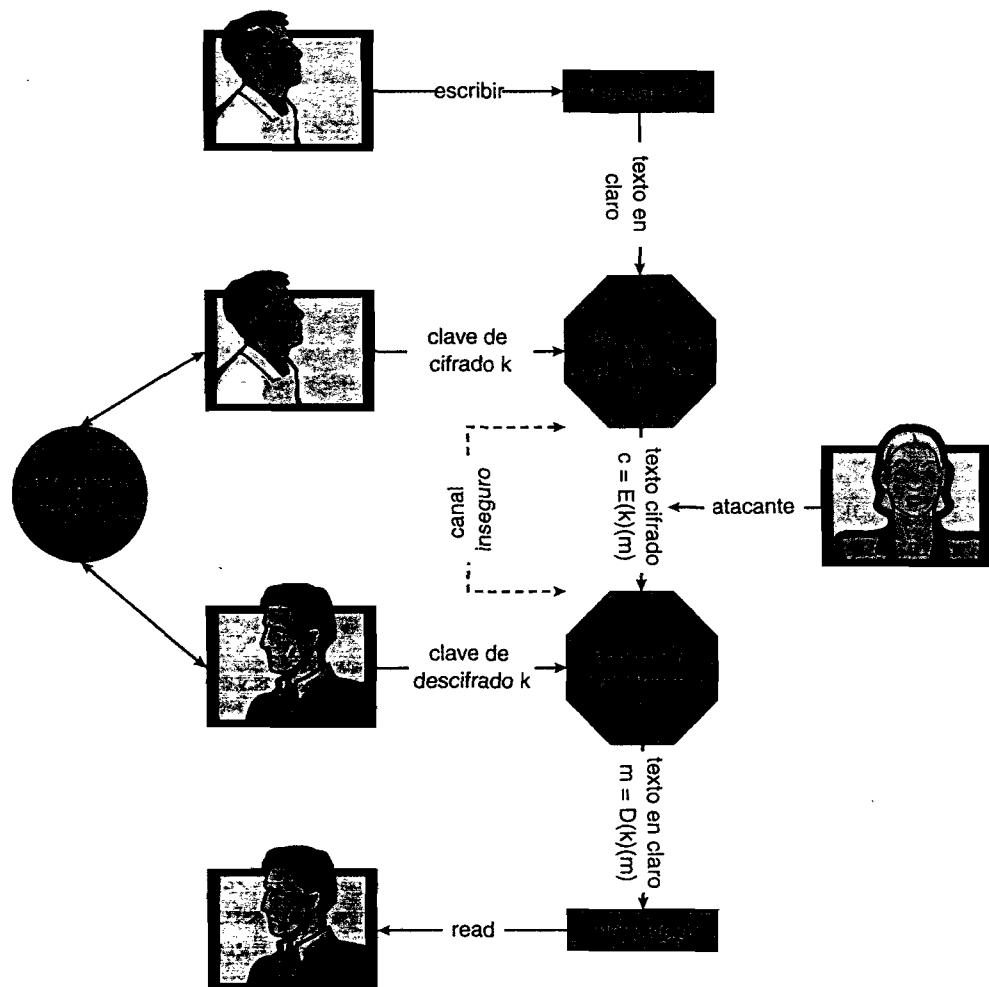


Figura 15.7 Una comunicación segura a través de un medio inseguro.

- Una función $E: K \rightarrow (M \rightarrow C)$. Es decir, para cada $k \in K$, $E(k)$ es una función para generar mensajes de texto cifrado a partir de los mensajes de texto en claro. Tanto E como $E(k)$ para cualquier k deben ser funciones computables de manera eficiente.
- Una función $D: K \rightarrow (C \rightarrow M)$. Es decir, para cada $k \in K$, $D(k)$ es una función para generar mensajes de texto en claro a partir de los mensajes de texto cifrado. Tanto D como $D(k)$ para cualquier k deben ser funciones eficientemente computables.

Un algoritmo de cifrado debe proporcionar esta propiedad esencial: dado un mensaje de texto cifrado $c \in C$, una computadora puede calcular m tal que $E(k)(m) = c$ sólo si posee $D(k)$. Así, una computadora que posea $D(k)$ puede descifrar los mensajes de texto cifrado para obtener los mensajes de texto en claro que se usaron para producirlos, pero una computadora que no posea $D(k)$ no puede descifrar esos mensajes cifrados. Puesto que los mensajes cifrados están generalmente expuestos (por ejemplo, enviándolos a través de la red), es importante que no se pueda deducir $D(k)$ a partir de los mensajes cifrados.

Existen dos tipos principales de algoritmos de cifrado: simétricos y asimétricos. Vamos a analizar ambos tipos de algoritmos en las siguientes secciones.

15.4.1.1 Cifrado simétrico

En un **algoritmo de cifrado simétrico**, se utiliza la misma clave para cifrar y para descifrar, es decir, $E(k)$ puede deducirse a partir de $D(k)$ y viceversa. Por tanto, es necesario proteger el secreto de $E(k)$ con el mismo grado que el de $D(k)$.

En los últimos 20 años, el algoritmo de cifrado simétrico más comúnmente utilizado en los Estados Unidos para aplicaciones civiles ha sido el **estándar DES** (data-encryption standard) adoptado por el NIST (National Institute of Standards and Technology). DES funciona tomando un valor de 64 bits y una clave de 56 bits y realizando una serie de transformaciones. Estas transformaciones están basadas en operaciones de sustitución y permutación, como suele ser generalmente el caso para las transformaciones de cifrado simétricas. Algunas de las transformaciones son de las denominadas **transformaciones de caja negra**, en el sentido de que sus algoritmos están ocultos. De hecho, estas denominadas “cajas S” son información clasificada por el gobierno de los Estados Unidos. Los mensajes de más de 64 bits se descomponen en fragmentos de 64 bits y los bloques que tengan una menor longitud se llenan con el fin de completar el tamaño de bloque requerido. Puesto que DES opera sobre un conjunto de bits simultáneamente, se denomina algoritmo de **cifrado de bloque**. Si se utiliza la misma clave para cifrar una gran cantidad de datos, esa clave comienza a ser vulnerable a los ataques. Consideré, por ejemplo, que un mismo bloque fuente generaría el mismo texto cifrado si se utilizaran la misma clave y el mismo algoritmo de cifrado; en consecuencia, esos fragmentos no sólo se cifran sino que también se hace una operación XOR con el bloque de texto cifrado anterior antes de proceder al cifrado. Este mecanismo se conoce con el nombre de **encadenamiento de bloques cifrados**.

DES se considera ahora inseguro para muchas aplicaciones, porque se puede realizar una exploración exhaustiva de las claves utilizando unos recursos informáticos no excesivamente grandes. Sin embargo, en lugar de abandonar DES, el NIST diseñó una modificación denominada **triple DES**, en la que el algoritmo DES se repite tres veces (dos cifrados y un descifrado) sobre un mismo texto en claro, utilizando dos o tres claves; por ejemplo, $c = E(k_3)(D(k_2)(E(K_1)(m)))$. Cuando se utilizan tres claves, la longitud efectiva de clave es de 168 bits. Hoy en día, el algoritmo triple DES se utiliza muy ampliamente.

En 2001, el NIST adoptó un nuevo algoritmo de cifrado, denominado **AES** (advanced encryption standard), para sustituir a DES. El algoritmo AES es otro algoritmo de cifrado de bloques simétricos. Puede utilizar longitudes de clave de 128, 192 y 256 bits y funciona sobre bloques de 128 bits. Opera realizando entre 10 y 14 rondas de transformaciones sobre una matriz formada a partir de un cierto bloque de datos. Generalmente, el algoritmo es bastante compacto y eficiente.

Hay varios otros algoritmos de cifrado de bloque simétricos que se utilizan hoy en día y que conviene mencionar. El algoritmo **twofish** es rápido, compacto y fácil de implementar. Puede utilizar una longitud variable de clave de hasta 256 bits y opera sobre bloques de 128 bits. **RC5** permite variar la longitud de clave, el número de transformaciones y el tamaño de bloque; puesto que sólo usa operaciones de cálculo básicas, puede ejecutarse sobre una amplia variedad de procesadores.

RC4 es, quizás, el algoritmo de cifrado de flujo más común. Un algoritmo de **cifrado de flujo** está diseñado para cifrar y descifrar un flujo de bytes o bits, en lugar de un bloque. Esto resulta útil cuando la longitud de una comunicación pueda hacer que un algoritmo de cifrado de bloques sea demasiado lento. La clave se introduce en un generador de bits pseudoaleatorio, que es un algoritmo que trata de producir bits aleatorios. La salida del generador, cuando se le alimenta con una clave, es lo que se denomina flujo de clave. Un **flujo de clave** es un conjunto infinito de claves que pueden usarse para el flujo de texto en claro que se proporcione como entrada. RC4 se utiliza para cifrar flujos de datos, como por ejemplo en WEP, el protocolo de LAN inalámbrica. También se lo utiliza en las comunicaciones entre exploradores y servidores web, como se explica más adelante. Desafortunadamente, se ha demostrado que RC4, tal como se lo utiliza en WEP (estándar IEEE 802.11) puede romperse utilizando una cantidad razonable de tiempo de procesamiento. De hecho, el propio RC4 tiene vulnerabilidades inherentes.

15.4.1.2 Cifrado asimétrico

En un **algoritmo de cifrado asimétrico**, las claves de cifrado y descifrado son distintas. Aquí, vamos a describir uno de tales algoritmos, conocido con el nombre **RSA** debido a las iniciales de los nombres de sus inventores (Rivest, Shamir y Adleman). El algoritmo RSA de cifrado es un algoritmo de cifrado de bloque de clave pública y es el algoritmo asimétrico más ampliamente uti

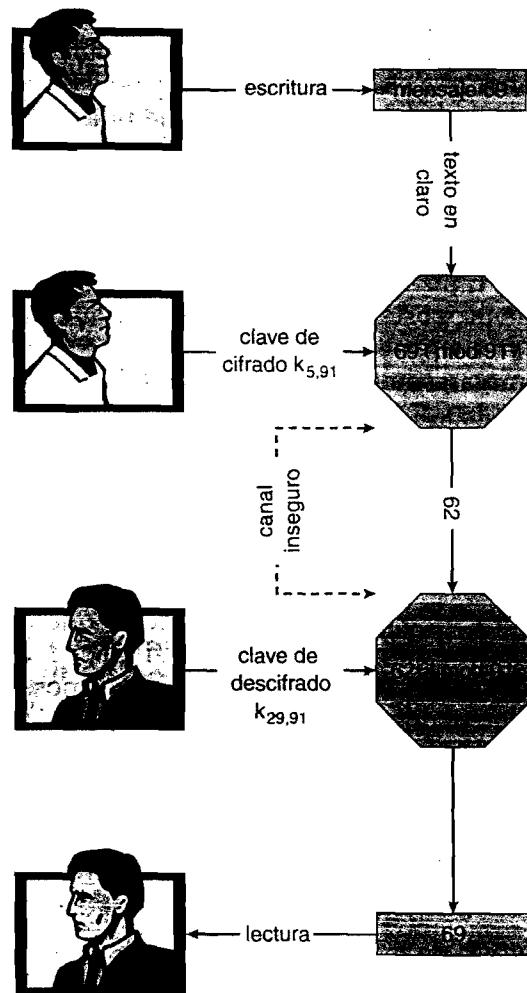


Figura 15.8 Cifrado y descifrado usando criptografía asimétrica RSA.

lizado. Sin embargo, los algoritmos asimétricos basados en curvas elípticas están ganando cada vez más terreno, porque la longitud de clave de dichos algoritmos puede ser más corta para un grado determinado de fortaleza criptográfica.

Resulta imprácticable, desde el punto de vista computacional, deducir $D(k_d, N)$ a partir de $E(k_e, N)$, por lo que no es necesario mantener $E(k_e, N)$ en secreto y dicha clave puede ser ampliamente diseminada; por tanto, $E(k_e, N)$ (o simplemente k_e) es la **clave pública** y $D(k_d, N)$ (o simplemente k_d) es la **clave privada**. N es el producto de dos números primos de gran magnitud p y q aleatoriamente seleccionados (por ejemplo, p y q podrían tener 512 bits cada uno). El algoritmo de cifrado es $E(k_e, N)(m) = m^{k_e} \text{ mod } N$, donde k_e satisface la ecuación $k_e k_d \text{ mod } (p-1)(q-1) = 1$. El algoritmo de descifrado será entonces $D(k_d, N)(c) = c^{k_d} \text{ mod } N$.

En la Figura 15.8 se muestra un ejemplo utilizando valores pequeños. En este ejemplo, hacemos $p = 7$ y $q = 13$. Calculamos entonces $N = 7 * 13 = 91$ y $(p-1)(q-1) = 72$. A continuación, seleccionamos k_e relativamente prima a 72 y < 72, lo que nos da 5. Finalmente, calculamos k_d tal que $k_e k_d \text{ mod } 72 = 1$, lo que nos da 29. Ahora tenemos nuestras claves: la clave pública, $k_e, N = 5, 91$ y la clave privada, $k_d, N = 29, 91$. Al cifrar el mensaje 69 con la clave pública obtenemos el mensaje 62, que puede ser decodificado por el receptor utilizando la clave privada.

El uso de un mecanismo de cifrado asimétrico comienza con la publicación de la clave pública del destino. Para la comunicación bidireccional, el origen debe también publicar su clave pública. Esta "publicación" puede ser tan simple como entregar una copia electrónica de la clave, o puede tratarse de un mecanismo más complejo. La clave privada (o "clave secreta") debe ser guar-

dada celosamente, ya que cualquiera que disponga de esa clave podrá descifrar cualquier mensaje creado a partir de la correspondiente clave pública.

Hay que resaltar que esa diferencia aparentemente pequeña en la utilización de claves entre la criptografía simétrica y asimétrica tiene, en la práctica, una gran importancia. La criptografía asimétrica se basa en funciones matemáticas en lugar de en transformaciones, lo que hace que sea mucho más cara de implementar, en términos de los recursos de computación requeridos. Para una computadora, resulta mucho más rápido codificar y decodificar un texto cifrado empleando los algoritmos simétricos usuales que utilizando algoritmos asimétricos. Entonces, se preguntará el lector, ¿por qué utilizar un algoritmo asimétrico? En realidad, estos algoritmos no se emplean para el cifrado de propósito general de grandes cantidades de datos. Sin embargo, no sólo se utilizan para cifrar pequeñas cantidades de datos sino también para proporcionar autenticación, confidencialidad y mecanismos de distribución de claves, como veremos en las siguientes secciones.

15.4.1.3 Autenticación

Hemos visto que el cifrado ofrece una manera de restringir el conjunto de posibles receptores de un mensaje. El proceso de restringir el conjunto de potenciales emisores de un mensaje se denomina **autenticación**. La autenticación es, por tanto, complementaria al cifrado. De hecho, algunas veces sus funciones se solapan. Tenga en cuenta que un mensaje cifrado también puede demostrar la identidad del emisor; por ejemplo, si $D(k_d, N)(E(k_e, N)(m))$ produce un mensaje válido, entonces sabemos que el creador del mensaje debe poseer k_e . La autenticación también resulta útil para demostrar que un mensaje no ha sido modificado. En esta sección, vamos a analizar la autenticación como restricción de los posibles receptores de un mensaje. Observe que este tipo de autenticación es similar, aunque diferente, a la autenticación de usuarios, de la que hablaremos en la Sección 15.5.

Un algoritmo de autenticación consta de los siguientes componentes:

- Un conjunto K de claves.
- Un conjunto M de mensajes.
- Un conjunto A de autenticadores.
- Una función $S : K \rightarrow (M \rightarrow A)$. Es decir, para cada $k \in K$, $S(k)$ es una función para generar autenticadores a partir de los mensajes. Tanto S como $S(k)$ para cualquier k deben ser funciones computacionalmente eficientes.
- Una función $V : K \rightarrow (M \times A \rightarrow \{\text{true}, \text{false}\})$. Es decir, para cada $k \in K$, $V(k)$ es una función para verificar autenticadores de los mensajes. Tanto V como $V(k)$ para cualquier k deben ser funciones eficientemente computables.

La propiedad crítica que un algoritmo de autenticación debe poseer es esta: para un mensaje m , una computadora puede generar un autenticador $a \in A$ tal que $V(k)(m, a) = \text{true}$ sólo si posee $S(k)$. Así, una computadora que posea $S(k)$ podrá generar autenticadores para los mensajes de modo que cualquier otra computadora que posea $V(k)$ pueda verificarlo. Sin embargo, una computadora que no tenga $S(k)$ no podrá generar autenticadores para los mensajes que puedan verificarse utilizando $V(k)$. Puesto que los autenticadores están generalmente expuestos (por ejemplo, cuando se los envía a través de la red con los propios mensajes), no debe ser factible deducir $S(k)$ a partir de los autenticadores.

Al igual que hay dos algoritmos de cifrado, hay también dos variedades principales de algoritmos de autenticación. El primer paso para comprender estos algoritmos consiste en analizar las funciones *hash*. Una función *hash* crea un pequeño bloque de datos de tamaño fijo, conocido como **resumen de mensaje** o **valor hash**, a partir de un mensaje dado. Las funciones hash operan tomando un mensaje en bloques de n bits y procesando los bloques para generar un valor hash de n bits. H debe ser resistente a las colisiones con respecto a m , es decir, no debe ser factible calcular un $m' \neq m$ tal que $H(m) = H(m')$. En estas condiciones, si $H(m) = H(m')$ sabemos que $m_1 = m_2$, es decir, que el mensaje no ha sido modificado. Entre las funciones más comunes de cálculo de

resúmenes de mensajes podemos citar **MD5**, que produce un valor hash de 128 bits y **SHA-1**, que genera un valor hash de 160 bits.

Los resúmenes de mensajes resultan útiles para detectar los mensajes modificados, pero no sirven como autenticadores. Por ejemplo, podemos enviar $H(m)$ junto con un mensaje; pero si H es conocida, entonces alguien podría modificar m y recalcular $H(m)$ y la modificación del mensaje nos sería detectada. Por tanto; se utiliza un algoritmo de autenticación para tomar el resumen del mensaje y cifrarlo.

El primer tipo de algoritmo de autenticación utiliza cifrado simétrico. En un **código de autenticación de mensajes** (MAC, message-authentication code), se genera una suma de comprobación criptográfica a partir del mensaje utilizando una clave secreta. El conocimiento de $V(k)$ y el conocimiento de $S(k)$ son equivalentes: podemos derivar una función a partir de la otra, por lo que es necesario mantener secreto el valor de k . Un ejemplo simple de código MAC define $S(k)(m) = f(k, H(m))$, donde f es una función de una sola dirección con respecto a su primer argumento es decir, no puede deducirse k a partir de $f(k, H(m))$. Debido a la propiedad de resistencia a las colisiones de la función hash, podemos estar razonablemente seguros de que ningún otro mensaje permita crear el mismo valor MAC. Un algoritmo de verificación apropiado será entonces $V(k)(m, a) \equiv (f(k, m) = a)$. Observe que se necesita k para calcular tanto $S(k)$ como $V(k)$, por lo que cualquiera que sea capaz de calcular esas funciones podrá calcular también la otra.

El segundo tipo principal de algoritmo de autenticación es el **algoritmo de firma digital**, y los autenticadores generados por uno de estos algoritmos se denominan **firmas digitales**. En un algoritmo de firma digital, no resulta computacionalmente factible deducir $S(k_v)$ a partir de $V(k_v)$; en particular, V es una función de una sola dirección. Por tanto, k_v es la clave pública y k_s es la clave privada.

Considere como ejemplo el algoritmo de firma digital RSA. Es similar al algoritmo de cifrado RSA, pero la utilización de las claves se invierte. La firma digital de un mensaje se obtiene calculando $S(k_s)(m) = H(m)^{k_s} \bmod N$. La clave k_s es, de nuevo, una pareja $\langle d, N \rangle$, donde N es el producto de dos números primos de gran magnitud p y q aleatoriamente elegidos. El algoritmo de verificación será entonces $V(k_v)(m, a) \equiv (a^{k_v} \bmod N = H(m))$, donde k_v satisface la ecuación $k_v k_s \bmod (p-1)(q-1) = 1$.

Si el cifrado permite demostrar la identidad del emisor de un mensaje, entonces ¿por qué necesitamos algoritmos de autenticación separados? Existen tres razones principales:

- Generalmente, los algoritmos de autenticación requieren menos cálculos (con la excepción de las firmas digitales RSA). Con grandes cantidades de texto en claro, esto puede representar una enorme diferencia en el uso de recursos y en el tiempo necesario para autenticar un mensaje.
- Un autenticador de un mensaje casi siempre es más corto que el mensaje y su texto cifrado correspondiente. Esto mejora el uso del espacio y reduce el tiempo de transmisión.
- En ocasiones, deseamos disponer de la posibilidad de autenticación, pero no de la confidencialidad. Por ejemplo, una empresa podría proporcionar un parche software y “firmar” dicho parche para demostrar que procede de la empresa y que no ha sido modificado.

La autenticación es un componente de muchos aspectos de la seguridad. Por ejemplo, es la base de los mecanismos de **no repudio**, que proporcionan una demostración de que una determinada entidad ha realizado una cierta acción. Un ejemplo típico de no repudio sería la cumplimentación de formularios electrónicos como alternativa a la firma de contratos en papel. La característica de no repudio asegura que una persona que haya cumplimentado un formulario electrónico no pueda negar que lo ha hecho.

15.4.1.4 Distribución de claves

Sin ninguna duda, una gran parte de la batalla entre criptógrafos (aquellos que inventan los mecanismos de cifrado) y criptoanalistas (aquellos que intentan romperlos) se encuentra en las claves. Con los algoritmos simétricos, ambas partes necesitan la clave y ninguna otra persona debe dis-

poner de ella. La tarea de suministrar la clave simétrica a sus usuarios legítimos constituye un reto importante. En ocasiones, esa distribución se hace **fuerza de banda**, por ejemplo mediante un documento escrito o una conversación. Sin embargo, estos métodos no resultan adecuados para distribución de claves a gran escala y también es necesario considerar el reto de la gestión de esas claves. Supongamos que un usuario desea comunicarse de forma privada con otros N usuarios. Dicho usuario necesitaría N claves y, para mayor seguridad, tendría que cambiar dichas claves con frecuencia.

Éstas son las razones por las que se han hecho esfuerzos para crear algoritmos de clave asimétrica. No sólo las claves se pueden intercambiar en público, sino que un determinado usuario sólo necesita una clave privada, independientemente de con cuántas personas deseé el usuario comunicarse. Queda todavía la cuestión de gestionar una clave pública por interlocutor con el que se desee establecer comunicación, pero puesto que las claves públicas no necesitan ser seguras, puede utilizarse un medio de almacenamiento simple para implementar dicho **anillo de claves**.

Lamentablemente, incluso la distribución de claves públicas requiere que se tenga cierto cuidado. Considere el ataque por interposición mostrado en la Figura 15.9. En este caso, la persona que desea recibir un mensaje cifrado envía su clave pública, pero un atacante también envía su clave pública "mala" (que se corresponde con su clave privada). La persona que desea enviar el mensaje cifrado desconoce esto y utiliza la clave mala para cifrar el mensaje. El atacante entonces lo descifra sin problemas.

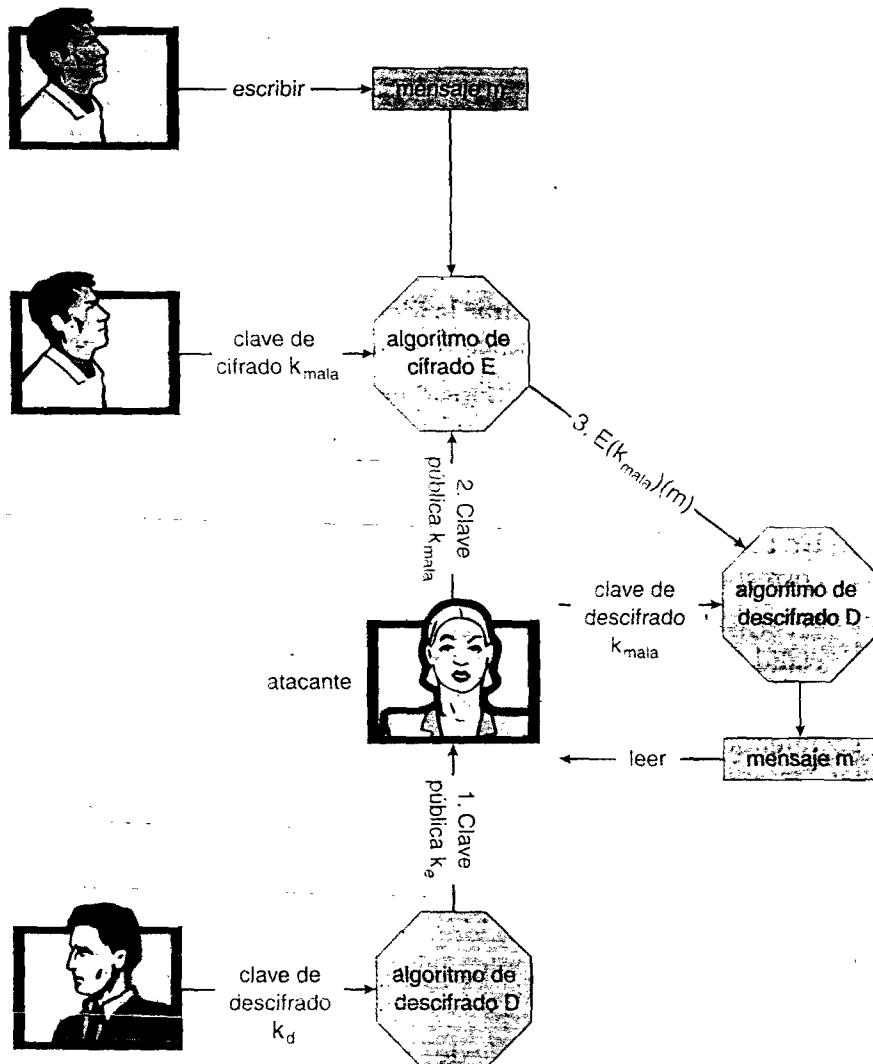


Figura 15.9 Un ataque por interposición en criptografía asimétrica.

El problema se encuentra en la autenticación; lo que necesitamos es demostrar quién (o qué) posee una determinada clave pública. Una forma de resolver este problema consiste en utilizar certificados digitales. Un **certificado digital** es una clave pública firmada digitalmente por un organismo de confianza. El organismo de confianza recibe la prueba de identificación de alguna entidad y certifica que la clave pública pertenece a dicha entidad. Pero, ¿cómo sabemos que el certificador es seguro? Estas **autoridades de certificación** incluyen sus claves públicas en exploradores web (y otros consumidores de certificados) antes de que éstos sean distribuidos. Estas autoridades de certificación pueden entonces avalar a otras autoridades de certificación (firmando digitalmente las claves públicas de estas otras autoridades), y así crear una jerarquía de confianza. Los certificados pueden distribuirse en un formato de certificado digital estándar X.509 que puede ser analizado por la computadora. Este esquema se emplea para conseguir una comunicación web segura, como se explica en la Sección 15.4.3.

15.4.2 Implementación de los mecanismos criptográficos

Usualmente, los protocolos de red se organizan en **niveles**, actuando cada nivel como un cliente del nivel inferior. Es decir, cuando un protocolo genera un mensaje para enviarlo a su correspondiente protocolo en otra máquina, pasa su mensaje al protocolo inferior de la pila de protocolos de red para que éste lo envíe a su correspondiente protocolo en dicha máquina. Por ejemplo, en una red IP, el protocolo TCP (un protocolo de *nivel de transporte*) actúa como un cliente de IP (un protocolo de *nivel de red*): los paquetes TCP se pasan al nivel IP para entregarlos al protocolo TCP que se encuentra en el otro extremo de la conexión TCP. IP encapsula el paquete TCP en un paquete IP, el cual se pasa de forma similar al *nivel de enlace de datos* inferior, para ser transmitido a través de la red a su correspondiente protocolo IP en la computadora de destino. Este protocolo IP entrega entonces el paquete TCP al protocolo TCP de dicha máquina. En conjunto, el **modelo de referencia ISO**, que se ha adoptado casi universalmente como modelo para las redes de datos, define siete niveles de protocolo. En el Capítulo 16 se explica más en detalle el modelo ISO; la Figura 16.6 muestra un diagrama del modelo.

La criptografía puede incluirse en casi cualquier nivel del modelo ISO. Por ejemplo, SSL (Sección 15.4.3) proporciona seguridad en el nivel de transporte. Generalmente, la seguridad del nivel de red se ha estandarizado en **IPSec**, que define formatos de los paquetes IP que permiten la inserción de autenticadores y el cifrado del contenido de los paquetes. Utiliza cifrado simétrico y el protocolo IKE para el intercambio de claves. IPSec está empezando a utilizarse de forma generalizada como base de las **redes privadas virtuales** (VPN, virtual private network), en las que todo el tráfico entre dos puntos extremos IPSec se cifra para construir una red privada utilizando una red que de otra manera sería pública. También se han desarrollado numerosos protocolos para que los utilicen las aplicaciones, aunque en este caso las propias aplicaciones deben programarse de modo que implementen los mecanismos de seguridad.

¿En qué lugar de la pila de protocolos es mejor incluir la protección criptográfica? En general, no hay una respuesta definitiva. Por un lado, cuando se incluyen las protecciones en la parte inferior de la pila, hay un número mayor de protocolos que pueden beneficiarse de ellas. Por ejemplo, puesto que los paquetes IP encapsulan paquetes TCP, el cifrado de paquetes IP (usando por ejemplo IPSec) también oculta el contenido de los paquetes TCP encapsulados. De forma similar, los autenticadores de los paquetes IP detectan la modificación de la información de cabecera TCP contenida en los paquetes.

Por otro lado, la protección en los niveles inferiores de la pila de protocolos puede resultar insuficiente para los protocolos de los niveles superiores. Por ejemplo, un servidor de aplicaciones que se ejecute sobre IPSec debe poder autenticar a las computadoras cliente de las que se reciben solicitudes. Sin embargo, para autenticar un usuario en una computadora cliente, el servidor puede tener que utilizar un protocolo del nivel de aplicación, requiriendo, por ejemplo, que el usuario escriba una contraseña. También hay que considerar el problema del correo electrónico. El correo electrónico suministrado a través del protocolo estándar SMTP se almacena y reenvía, frecuentemente múltiples veces, antes de ser entregado. Cada uno de estos saltos podría llevar a una red segura o a una red no segura. Para que el correo electrónico sea seguro, los mensajes de

correo electrónico tienen que cifrarse de manera que su seguridad sea independiente de los niveles de transporte utilizados para distribuirlos.

15.4.3 Un ejemplo: SSL

SSL 3.0 es un protocolo criptográfico que permite que dos computadoras se comuniquen de forma segura; es decir, cada una de ellas puede establecer limitaciones que garanticen que el transmisor o receptor de los mensajes sea la otra computadora. Es quizás el protocolo criptográfico más comúnmente empleado actualmente en Internet, dado que es el protocolo estándar mediante el que se comunican de forma segura los exploradores web con los servidores web. Hay que hacer notar que SSL fue diseñado por Netscape y que ha evolucionado transformándose en el estándar TLS. En esta exposición, emplearemos SSL para referirnos tanto a SSL como a TLS.

SSL es un protocolo complejo con muchas opciones. Aquí sólo vamos a presentar una de las variantes existentes, de forma muy simplificada y abstracta, con el fin de centrarnos en la utilización de primitivas criptográficas. Lo que vamos a ver es un complejo sistema en el que se usa la criptografía asimétrica de manera que un cliente y un servidor puedan establecer una clave de sesión segura, que puede emplearse para cifrado simétrico de la sesión mantenida por esos dos interlocutores, todo ello evitando los ataques por interposición y por repetición. Para aumentar la fortaleza criptográfica, las claves de sesión se olvidan una vez que la sesión se ha completado. Otra comunicación entre ambos interlocutores requerirá la generación de nuevas claves de sesión.

El protocolo SSL es iniciado por un cliente c para comunicarse de forma segura con un servidor s . Antes de utilizar el protocolo, se supone que el servidor s ha obtenido un certificado, denotado por cert_s , de una autoridad de certificación CA. Este certificado es una estructura que contiene lo siguiente:

- Varios atributos attr_s del servidor, tal como su nombre *distintivo* único y su nombre *común* (DNS).
- La identidad de un algoritmo de cifrado público $E()$ para el servidor.
- La clave pública k_e de ese servidor.
- Un intervalo de validez interval durante el que el certificado debe considerarse válido.
- Una firma digital a para la información anterior, proporcionada por la certificación de autoridad CA, es decir, $a = S(k_{\text{CA}})(\langle \text{attr}_s, E(k_e), \text{interval} \rangle)$

Además, antes de usar el protocolo, se supone que el cliente ha obtenido el algoritmo público de verificación $V(k_{\text{CA}})$ para la autoridad de certificación CA. En el caso de la Web, el explorador del usuario suministrado por su proveedor contendrá los algoritmos de verificación y las claves públicas de determinadas autoridades de certificación. El usuario puede añadir o eliminar autoridades de certificación según desee.

Cuando c se conecta a s , envía un valor aleatorio n_c de 28 bytes al servidor, el cual responde con un valor aleatorio n_s generado por él, más su certificado cert_s . El cliente verifica que $V(k_{\text{CA}})(\langle \text{attr}_s, E(k_e), \text{interval} \rangle, a) = \text{true}$ y que el momento actual está dentro del intervalo de validez interval . Si ambas condiciones se satisfacen, el servidor ha demostrado su identidad. Entonces el cliente genera un **secreto premaestro** pms aleatorio de 46 bytes y envía al servidor $\text{cpms} = E(k_s)(\text{pms})$. El servidor recupera $\text{pms} = D(k_d)(\text{cpms})$. Ahora, tanto el cliente como el servidor están en posesión de n_c , n_s y pms , y cada uno de ellos puede calcular un **secreto maestro** de 48 bytes $\text{ms} = f(n_c, n_s, \text{pms})$, donde f es una función unidireccional resistente a colisiones. Sólo el servidor y el cliente pueden calcular ms , dado que sólo ellos conocen pms . Además, la dependencia de ms con respecto a n_c y n_s asegura que ms es un valor *fresco*; es decir, una clave de sesión que no se ha utilizado en ninguna comunicación anterior. En esta situación, el cliente y el servidor calculan las claves siguientes a partir de ms :

- Una clave de cifrado simétrica k_{cs}^{crypt} para cifrar mensajes del cliente al servidor.
- Una clave de cifrado simétrica k_{sc}^{crypt} para cifrar mensajes del servidor al cliente.

- Una clave de generación MAC k_{cs}^{mac} para generar autenticadores para los mensajes del cliente al servidor.
- Una clave de generación MAC k_{sc}^{mac} para generar autenticadores para los mensajes del servidor al cliente.

Para enviar un mensaje m al servidor, el cliente envía

$$c = E(k_{cs}^{crypt})(\langle m, S(k_{cs}^{mac})(m) \rangle)$$

Al recibir c , el servidor recupera

$$\langle m, a \rangle = D(k_{cs}^{crypt})(c)$$

y acepta m si $V(k_{cs}^{mac})(m, a) = \text{true}$. De forma similar, para enviar un mensaje m al cliente, el servidor envía

$$c = E(k_{sc}^{crypt})(\langle m, S(k_{sc}^{mac})(m) \rangle)$$

y el cliente recupera

$$\langle m, a \rangle = D(k_{sc}^{crypt})(c)$$

y acepta m si $V(k_{sc}^{mac})(m, a) = \text{true}$.

Este protocolo permite al servidor limitar los receptores de sus mensajes al cliente que generó el pms y limitar también a dicho cliente los emisores de los mensajes que acepta. De forma similar, el cliente puede limitar los receptores de los mensajes que envía y el emisor de los mensajes que acepta al interlocutor que conoce $S(k_d)$ (es decir, al interlocutor que puede descifrar $cpms$). En muchas aplicaciones, como por ejemplo las transacciones web, el cliente necesita verificar la identidad del interlocutor que conoce $S(k_d)$. Éste es uno de los propósitos del certificado cert_s ; en particular, el campo `attrs` contiene información que el cliente puede emplear para determinar la identidad (por ejemplo, el nombre de dominio) del servidor con el que se está comunicando. Para aplicaciones en las que el servidor también necesita información acerca del cliente, SSL proporciona una opción mediante la que un cliente puede enviar un certificado al servidor.

Además de su uso en Internet, SSL se emplea actualmente en una amplia variedad de tareas. Por ejemplo, ahora las redes privadas virtuales IPSec tienen un competidor en las redes privadas virtuales SSL. IPSec resulta adecuado para el cifrado de tráfico punto a punto, por ejemplo entre dos oficinas de una empresa. Las VPN SSL son más flexibles pero no tan eficientes, por lo que pueden utilizarse, por ejemplo, para la comunicación entre un empleado concreto que trabaje en modo remoto y su oficina.

15.5 Autenticación de usuario

La exposición anterior sobre autenticación hacía referencia a mensajes y sesiones, pero, ¿qué pasa con los usuarios? Si un sistema no puede autenticar a un usuario, entonces autenticar un mensaje procedente de dicho usuario no sirve de nada. Por tanto, un problema de seguridad importante en los sistemas operativos es el de la **autenticación de usuario**. El sistema de protección depende de la capacidad de identificar los programas y procesos que están actualmente en ejecución, lo que a su vez depende de la capacidad de identificar a cada usuario del sistema. Normalmente, cada usuario se identifica a sí mismo. ¿Cómo podemos determinar si la identidad de un usuario es auténtica? Generalmente, la autenticación de usuario se basa en una o más de tres cuestiones: la posesión de algo (una clave o tarjeta) por parte del usuario, el conocimiento de algo (un identificador de usuario y una contraseña) por parte del usuario y/o un atributo del usuario (huella digital, patrón retinal o firma).

15.5.1 Contraseñas

El método más habitual para autenticar la identidad de un usuario consiste en usar **contraseñas**. Cuando el usuario se identifica a sí mismo mediante un ID de usuario o un nombre de cuenta, se le pide una contraseña. Si la contraseña suministrada por el usuario coincide con la contraseña almacenada en el sistema, el sistema supone que el propietario de la cuenta está accediendo a la misma.

A menudo, en ausencia de esquemas de protección más completos, se usan contraseñas para proteger objetos del sistema informático. Las contraseñas pueden considerarse un caso especial de las claves o de las capacidades. Por ejemplo, una contraseña podría estar asociada con un recurso (por ejemplo, un archivo): si se hace una solicitud para usar el recurso, debe proporcionarse la contraseña; si ésta es correcta, se concede el acceso. Pueden asociarse diferentes contraseñas con distintos derechos de acceso. Por ejemplo, pueden emplearse diferentes contraseñas para leer archivos, añadir información a archivos y actualizar archivos.

En la práctica, la mayoría de los sistemas sólo requieren una contraseña para que el usuario pueda adquirir todos los derechos. Aunque, en teoría, cuantas más contraseñas, mayor será la seguridad, tales sistemas no suelen implementarse debido al clásico compromiso entre seguridad y comodidad. Si la seguridad produce incomodidad, entonces frecuentemente la seguridad se pasa por alto o se evita de alguna manera.

15.5.2 Vulnerabilidades de las contraseñas

Las contraseñas son extremadamente comunes porque son fáciles de comprender y utilizar. Lamentablemente, a menudo pueden adivinarse, ser mostradas por accidente, ser interceptadas o ser ilegalmente transferidas por un usuario autorizado a otro que no lo está, como vamos a ver a continuación.

Existen dos formas habituales de adivinar una contraseña. Una forma consiste en que el intruso (persona o programa) conoce al usuario o tiene información acerca de él. Con demasiada frecuencia, las personas emplean como contraseñas informaciones obvias, como los nombres de sus mascotas o de sus parejas. Otra forma consiste en emplear la fuerza bruta, probando a enumerar todas las posibles combinaciones formadas por caracteres válidos de la contraseña (letras, números y algunos símbolos de puntuación) hasta encontrar la contraseña. Las contraseñas cortas son especialmente vulnerables a este método. Por ejemplo, una contraseña de cuatro dígitos decimales sólo da lugar a 10.000 posibles variaciones. Como promedio, probar 5000 posibilidades nos daría la contraseña correcta. Un programa que probara una contraseña por milisegundo sólo tardaría unos 5 segundos en adivinar una contraseña de cuatro dígitos. Este método de enumeración no tiene tanto éxito en los sistemas que permiten emplear contraseñas más largas, que incluyan tanto letras mayúsculas como minúsculas, además de números y todos los caracteres de puntuación. Por supuesto, los usuarios deben aprovechar la longitud de la contraseña y no deben, por ejemplo, emplear sólo letras minúsculas.

Además de poder ser adivinadas, las contraseñas pueden averiguarse mediante mecanismos de monitorización visual o electrónica. Un intruso puede mirar por encima del hombro del usuario cuando el usuario está iniciando una sesión y aprenderse la contraseña fácilmente mirando el teclado. Alternativamente, cualquiera con acceso a la red en la que reside la computadora puede añadir sin problemas un monitor de red, que le permita ver todos los datos que estén siendo transferidos a través de la red, actividad que se conoce con el nombre de *husmear (sniffing)*, incluyendo los ID de usuario y las contraseñas. El cifrado del flujo de datos que contiene la contraseña resuelve este problema. Sin embargo, incluso en un sistema de este tipo podrían robarse las contraseñas. Por ejemplo, si se emplea un archivo para guardar las contraseñas, podría copiarse para llevar a cabo un análisis externo al sistema. O puede tenerse instalado un caballo de Troya en el sistema, que capture las pulsaciones de tecla antes de ser enviadas a la aplicación.

La exposición de las contraseñas puede ser un problema importante si éstas se escriben en un lugar donde puedan ser leídas o donde puedan perderse. Como veremos, algunos sistemas fuerzan a los usuarios a seleccionar contraseñas difíciles de recordar o muy largas, lo que da lugar a

que el usuario la apunte o la reutilice. Como resultado, tales sistemas proporcionan mucha menos seguridad que los sistemas que permiten a los usuarios elegir contraseñas fáciles.

El último tipo de amenaza relativa a las contraseñas, la transferencia ilegal, es resultado de la propia naturaleza humana. La mayor parte de las instalaciones de computadoras tienen una regla que prohíbe a los usuarios compartir cuentas. En ocasiones, esta regla se implementa por razones contables, pero con frecuencia se impone para mejorar la seguridad. Por ejemplo, supongamos que varios usuarios comparten un mismo ID de usuario y que se produce una brecha de seguridad que afecta a dicho ID de usuario. Es imposible saber quién estaba usando el ID en el momento en que se produjo la brecha e incluso si se trataba de un usuario autorizado. Con un ID para cada usuario, es posible preguntar directamente a cualquier usuario sobre el uso de la cuenta; además, el usuario puede detectar que algo ocurre en la cuenta y detectar la intromisión. Algunas veces, los usuarios incumplen las reglas de compartición de cuentas para ayudar a sus amigos o para esquivar los mecanismos contables, y este comportamiento puede dar lugar a accesos al sistema, posiblemente dañinos, por parte de usuarios no autorizados.

Las contraseñas pueden ser generadas por el sistema o seleccionadas por el usuario. Las contraseñas generadas por el sistema pueden ser difíciles de recordar, por lo que en consecuencia los usuarios las anotarán. Sin embargo, como se ha mencionado anteriormente, a menudo las contraseñas seleccionadas por los usuarios son fáciles de adivinar (el nombre del usuario o su coche favorito). Algunos sistemas comprobarán una contraseña propuesta antes de aceptarla, para ver si resulta fácil su adivinación. En algunos sitios, los administradores comprueban ocasionalmente las contraseñas de usuario e informan al usuario de que su contraseña es fácil de adivinar. Algunos sistemas también controlan la *edad* de las contraseñas, forzando a los usuarios a cambiarlas a intervalos regulares de tiempo (por ejemplo, cada tres meses). Este método no es a toda prueba, ya que los usuarios pueden alternar entre dos contraseñas. La solución, tal y como se ha implementado en algunos sistemas, consiste en registrar un historial de contraseñas para cada usuario. Por ejemplo, el sistema puede guardar las N últimas contraseñas y no permitir que se reutilicen.

Pueden emplearse diversas variantes de estos sencillos esquemas de contraseñas. Por ejemplo, la contraseña puede cambiarse con una mayor frecuencia. En el caso extremo, la contraseña se cambiaría en cada sesión: al final de la sesión, el sistema o el usuario selecciona una nueva contraseña, que se empleará en la sesión siguiente. En este caso, incluso aunque una contraseña se use mal, sólo se empleará una vez. Cuando el usuario legítimo intente utilizar una contraseña no válida en la siguiente sesión, descubrirá que ha habido una violación de la seguridad. Entonces, deberá seguir los pasos necesarios para reparar la brecha de seguridad.

15.5.3 Contraseñas cifradas

Un problema que plantean todos estos métodos es la dificultad de mantener en secreto la contraseña dentro de la computadora. ¿Cómo puede almacenar el sistema una contraseña de forma segura para que el mecanismo de autenticación pueda utilizarla cuando el usuario especifique su contraseña? Los sistemas UNIX utilizan el cifrado para evitar la necesidad de mantener en secreto su lista de contraseñas. Cada usuario tiene una contraseña. El sistema contiene una función que es extremadamente difícil (los diseñadores esperan que imposible) de invertir, pero fácil de calcular. Es decir, dado un valor x , es fácil calcular el valor de la función $f(x)$. Sin embargo, dado un valor de la función $f(x)$, es imposible calcular x . Esta función se emplea para codificar todas las contraseñas y sólo se almacenan las contraseñas codificadas. Cuando un usuario presenta una contraseña, se codifica y se compara con la contraseña codificada que se tiene almacenada. Aunque esta contraseña codificada pueda verse, no puede decodificarse, por lo que no es posible determinar la contraseña real. Por tanto, no es necesario mantener en secreto el archivo de contraseñas. La función $f(x)$ es, normalmente, un algoritmo de cifrado que se ha diseñado y probado de forma rigurosa.

El fallo de este método es que el sistema ya no tiene el control sobre las contraseñas. Aunque las contraseñas se cifren, cualquiera que disponga de una copia del archivo de contraseñas puede

ejecutar rutinas de cifrado rápido, cifrando cada palabra en un diccionario, por ejemplo, y comparando los resultados con las contraseñas almacenadas. Si el usuario ha seleccionado una contraseña que sea una palabra contenida en el diccionario, este sistema permitirá romper la contraseña. En computadoras lo suficientemente rápidas o incluso en *clusters* de computadoras lentes, tal comparación puede llevar sólo unas pocas horas. Además, dado que los sistemas UNIX utilizan un algoritmo de cifrado bien conocido, un *cracker* puede mantener una caché de contraseñas que haya roto previamente. Por estas razones, las nuevas versiones de UNIX almacenan las entradas de contraseñas cifradas en un archivo que sólo puede leer el **superusuario**. Los programas que comparan una contraseña especificada por el usuario con la contraseña almacenada ejecutan *setuid* con la cuenta *root*, de modo que pueden leer este archivo, pero otros usuarios no pueden. También incluyen un *aleatorizador*, o número aleatorio grabado, en el algoritmo de cifrado. El aleatorizador se añade a la contraseña para asegurar que, si dos contraseñas son iguales sin cifrar, darán lugar a contraseñas cifradas diferentes.

Otra debilidad de los métodos de contraseñas de UNIX es que muchos sistemas UNIX sólo tratan los ocho primeros caracteres como significativos. Por tanto, es extremadamente importante para los usuarios aprovecharse del espacio de contraseñas disponible. Para evitar el método de análisis basado en diccionario, algunos sistemas no permiten el uso de palabras del diccionario como contraseñas. Una buena técnica consiste en generar la contraseña usando la primera letra de cada palabra de una frase que sea fácil de recordar, usando tanto letras mayúsculas como minúsculas, además de un número o un signo de puntuación. Por ejemplo, la frase "El nombre de mi madre es Catalina" podría dar la contraseña "Endmm.esC". La contraseña es difícil de romper, pero el usuario puede recordarla rápidamente.

15.5.4 Contraseñas de un solo uso

Para evitar los problemas de la intercepción de contraseñas y las miradas de los fisgones por encima del hombro, un sistema podría usar un conjunto de **contraseñas emparejadas**. Cuando se inicia una sesión, el sistema selecciona aleatoriamente una pareja de contraseñas y presenta una parte de la misma; el usuario debe suministrar la otra parte. En este sistema, el usuario es **desafiado** y debe **responder** con la respuesta correcta a dicho desafío.

Este método se puede generalizar, utilizando un algoritmo como contraseña. Por ejemplo, el algoritmo puede ser una función entera: el sistema selecciona un entero aleatorio y lo presenta al usuario. El usuario aplica la función y responde con el resultado correcto. El sistema también aplica la función. Si los dos resultados se corresponden, se permite el acceso.

Estas contraseñas algorítmicas no son susceptibles de ser reutilizadas; es decir, un usuario puede escribir una contraseña y ninguna entidad que intercepte dicha contraseña podrá reutilizarla. En esta variante, el sistema y el usuario comparten un secreto y ese secreto nunca se transmite a través de un medio que permita su exposición. En su lugar, el secreto se usa como entrada de la función, junto con una semilla compartida. Una **semilla** es un número aleatorio o una secuencia alfanumérica. La semilla es el desafío de autenticación de la computadora. El secreto y la semilla se usan como entrada de la función $f(\text{secreto}, \text{semilla})$. El resultado de esta función se transmite como contraseña a la computadora. Dado que la computadora también conoce el secreto y la semilla, puede realizar el mismo cálculo. Si los resultados se corresponden, el usuario es autenticado. La siguiente vez que el usuario necesite ser autenticado, se generará otra semilla y seguirán los mismos pasos. Esta vez, la contraseña será diferente.

En este sistema de **contraseña de un solo uso**, la contraseña es diferente en cada caso. Cualquiera que capture la contraseña de una sesión e intente reutilizarla en otra sesión no tendrá éxito. Las contraseñas de un solo uso constituyen un método para impedir las autenticaciones erróneas debidas a la exposición de la contraseña.

Los sistemas de contraseña de un solo uso se implementan de varias formas. Las implementaciones comerciales, tales como SecurID, utilizan calculadoras hardware. La mayoría de estas calculadoras tienen forma de tarjeta de crédito, de mochila de claves o de dispositivo USB; disponen de una pantalla y pueden o no tener un teclado. Algunos sistemas utilizan la hora actual como semilla aleatoria; otros requieren que los usuarios introduzcan a través del teclado el número

secreto compartido, también conocido como **número personal de identificación** o PIN (personal identification number). La pantalla muestra entonces la contraseña de un solo uso. La utilización de un generador de contraseñas de un solo uso y de un PIN es una forma de **autenticación de dos factores**. Este tipo de autenticación ofrece una mejor protección que la autenticación de un sólo factor.

Otra variación de las contraseñas de un solo uso es la utilización de un **libros de código**, o **cuaderno de un solo uso**, que es una lista de contraseñas de un sólo uso. En este método, cada contraseña de la lista se usa, por orden, una vez y luego se tacha o se borra. El sistema S/Key comúnmente utilizado emplea una calculadora software o un libro de códigos basado en estos cálculos como fuente de contraseñas de un solo uso. Por supuesto, el usuario tiene que proteger este libro de códigos.

15.5.5 Biométrica

Otra variante del uso de contraseñas en los mecanismos de autenticación implica el uso de medidas biométricas. Los lectores palmares o de manos se usan habitualmente para dotar de seguridad a los accesos físicos, como por ejemplo, el acceso a un centro de datos. Estos lectores establecen la correspondencia entre los parámetros almacenados y los que se obtienen mediante los lectores de manos. Los parámetros pueden incluir un mapa de temperaturas, así como la longitud del dedo, la anchura del mismo y los patrones de las líneas. Estos dispositivos son actualmente demasiado grandes y caros como para ser utilizados en los mecanismos de autenticación de las computadoras normales.

Los lectores de huellas digitales son muy precisos y su relación coste-efectividad es buena, por lo que en el futuro serán de uso común. Estos dispositivos leen los patrones de las líneas del dedo y los convierten en una secuencia de números. Con el tiempo, pueden almacenar un conjunto de secuencias para adaptarse a la localización del dedo sobre la almohadilla de lectura y otros factores. El software puede entonces explorar un dedo situado sobre la almohadilla y comparar sus características con estas secuencias almacenadas, para determinar si el dedo que hay sobre el dispositivo de lectura es el mismo que el que tiene almacenado. Por supuesto, varios usuarios pueden tener almacenados sus perfiles y el escáner puede establecer la diferencia entre ellos. Puede obtenerse un esquema de autenticación de dos factores muy preciso si se requiere una contraseña y un nombre de usuario, además de una exploración de huella digital. Si esta información se cifra mientras está en tránsito, el sistema será muy resistente a los ataques de suplantación y de repetición.

La autenticación mediante **múltiples factores** es todavía mejor. Considere lo fuerte que puede ser un mecanismo de autenticación que emplee un dispositivo USB, que se conectará al sistema, un PIN y un escáner de huella digital. Salvo porque el usuario tiene que colocar su dedo sobre el lector y conectar el dispositivo USB al sistema, este método de autenticación es casi igual de cómodo que el uso de contraseñas normales. Recuerde, sin embargo, que una autenticación fuerte no es suficiente por sí misma para garantizar el ID del usuario. Una sesión autenticada puede estar sujeta a manipulaciones si no está cifrada.

15.6 Implementación de defensas de seguridad

Dado que existe una multitud de amenazas al sistema y a la seguridad de la red, existen muchas soluciones de seguridad. Las soluciones recorren toda la gama que va desde mejorar la educación del usuario hasta los aspectos tecnológicos relacionados con la escritura de software libre de errores. La mayor parte de los profesionales de la seguridad defienden la teoría de la **defensa en profundidad**, la cual establece que es mejor utilizar más niveles de defensa que menos. Por supuesto, esta teoría se aplica a cualquier clase de seguridad. Considere la seguridad de una casa sin un cerrojo en la puerta, con un cerrojo y con un cerrojo y una alarma. En esta sección, vamos a exponer los métodos, herramientas y técnicas más importantes que se pueden utilizar para mejorar la resistencia a las amenazas.

15.6.1 Política de seguridad

El primer paso para mejorar la seguridad de cualquier aspecto de la informática es disponer de una **política de seguridad**. Las políticas son muy variadas pero, generalmente, incluyen una declaración que describe qué es lo que se va a proteger. Por ejemplo, una política puede establecer que debe revisarse el código de todas las aplicaciones a las que se pueda acceder desde el exterior, antes de implementarlas, o que los usuarios no deben compartir sus contraseñas, o que debe ejecutarse un análisis de puertos cada seis meses en todos los puntos de conexión entre una empresa y el exterior. Si no se dispone de una política, es imposible que los usuarios y los administradores sepan lo que está permitido, lo que es necesario y lo que no está permitido. La política es un mapa de carreteras para la seguridad y si un sitio está intentando pasar de una situación poco segura a otra más segura, necesita un mapa para saber cómo llegar.

Una vez que se ha definido la política de seguridad, las personas afectadas por ella deben conocerla bien. Debería ser una guía. La política también debe ser un **documento vivo**, que se revise y actualice periódicamente para asegurar que sigue siendo adecuado y que se cumple.

15.6.2 Evaluación de la vulnerabilidad

¿Cómo podemos determinar si una política de seguridad se ha implementado correctamente? La mejor forma consiste en realizar una evaluación de la vulnerabilidad. Tales evaluaciones pueden cubrir un amplio espectro, desde la ingeniería social a la **evaluación de riesgos** y los análisis de puertos. Por ejemplo, la evaluación de riesgos hace todo lo posible por evaluar los activos de la entidad en cuestión (un programa, un equipo gerencial, un sistema o una instalación) y determina las probabilidades de que un incidente de seguridad afecte a la entidad y disminuya su valor. Cuando se conocen las probabilidades de sufrir una pérdida y el valor de las pérdidas potenciales, puede asignarse un valor al intento de dotar de seguridad a la entidad.

La actividad principal de la mayor parte de las evaluaciones de vulnerabilidad es una **prueba de penetración**, en la que se analiza la entidad para conocer las vulnerabilidades. Dado que este libro se ocupa de los sistemas operativos y del software que se ejecuta sobre ellos, nos vamos a centrar en dichos aspectos.

Normalmente, los análisis de vulnerabilidades se realizan cuando la actividad de la computadora es relativamente baja, con el fin de minimizar su impacto. Cuando resulta apropiado, se realizan sobre sistemas de prueba en lugar de sobre los sistemas de producción, ya que pueden inducir un comportamiento no adecuado de los sistemas objetivo o de los dispositivos de red.

Un análisis de un sistema individual puede comprobar el siguiente conjunto de aspectos del sistema:

- Contraseñas cortas o fáciles de adivinar.
- Programas privilegiados no autorizados, tales como programas *setuid*.
- Programas no autorizados en directorios del sistema.
- Programas de duración inesperadamente larga.
- Protecciones inapropiadas en los directorios del sistema y de los usuarios.
- Protecciones inapropiadas en archivos de datos del sistema, como el archivo de contraseñas, los controladores de dispositivos o el propio *kernel* del sistema operativo.
- Entradas peligrosas en la ruta de búsqueda de un programa (por ejemplo, el caballo de Troya mencionado en la Sección 15.2.1).
- Cambios en los programas del sistema, detectados mediante valores de sumas de comprobación.
- Demonios de red ocultos o inesperados.

Cualquier problema encontrado mediante un análisis de seguridad puede resolverse automáticamente, o bien puede informarse del mismo a los gestores del sistema.

Las computadoras conectadas en red son mucho más susceptibles a los ataques de seguridad que los sistemas autónomos. En lugar de ataques procedentes de un conjunto conocido de puntos de acceso, tales como terminales directamente conectados, nos enfrentamos a ataques de un conjunto grande y desconocido de puntos de acceso, lo cual constituye un problema de seguridad potencialmente grave. En menor grado, los sistemas conectados a líneas telefónicas a través de un módem también están más expuestos.

De hecho, el gobierno de EE. UU. considera que un sistema sólo es tan seguro como lo sea su conexión más distante. Por ejemplo, a un sistema de alto secreto sólo se puede acceder desde dentro de un edificio que también esté considerado como de alto secreto. Los sistemas pierden su categoría de alto secreto si puede producirse cualquier forma de comunicación externa a dicho entorno. Algunas instalaciones del gobierno disponen de precauciones de seguridad extremas. Los conectores que enchufan un terminal a una computadora se guardan en una caja fuerte en la oficina cuando el terminal deja de estar en uso. Para poder acceder al edificio y a su oficina, una persona también debe tener su ID apropiado, debe conocer la combinación de una cerradura física y también debe conocer la información de autenticación para poder acceder a la computadora; esto es un ejemplo de autenticación mediante múltiples factores.

Desafortunadamente para los administradores de sistemas y los profesionales de la seguridad informática, frecuentemente es imposible encerrar bajo llave una máquina en una sala y desactivar todos los accesos remotos. Por ejemplo, actualmente, la red Internet conecta millones de computadoras y se ha convertido en un recurso indispensable y de misión crítica para muchas empresas e individuos. Si consideramos Internet como un club, entonces, como en cualquier club con millones de miembros, hay muchos miembros bienintencionados y otros muchos que son malintencionados. Los miembros malintencionados disponen de muchas herramientas que pueden utilizar para intentar obtener acceso a las computadoras interconectadas, al igual que hizo Morris con su famoso gusano.

Los análisis de vulnerabilidad se pueden aplicar a las redes para abordar algunos de los problemas relacionados con la seguridad de la red. Los análisis buscan en la red puertos que respondan a una solicitud. Si hay activados servicios que no deberían estarlo, puede bloquearse el acceso a los mismos o pueden desactivarse. Luego, los análisis determinan la información acerca de las aplicaciones que están a la escucha en esos puertos e intentan determinar si cada una de esas aplicaciones tiene vulnerabilidades conocidas. Probar dichas vulnerabilidades permitirá determinar si el sistema está mal configurado o si le faltan los parches necesarios.

Por último, consideremos el uso de los analizadores de puertos en las manos de un *cracker*, en lugar de en las manos de alguien que intenta mejorar la seguridad. Estas herramientas pueden ayudar a los *crackers* a encontrar las vulnerabilidades que pueden atacar (afortunadamente, es posible detectar los análisis de puertos mediante mecanismos de detección de anomalías, como veremos a continuación). Es un problema generalizado de la seguridad el que las mismas herramientas puedan emplearse para hacer el bien o el mal. De hecho, algunas personas recomiendan implementar la **seguridad mediante la oscuridad**, defendiendo que no se escriban herramientas para probar la seguridad, de manera que sea más difícil encontrar (y explotar) dichos agujeros de seguridad. Otros creen que este enfoque de la seguridad no es válido, apuntando, por ejemplo, que los *crackers* podrían escribir sus propias herramientas. Parece razonable que la seguridad a través de la oscuridad se considere sólo uno más de los niveles de seguridad, debiendo utilizarse también otros niveles. Por ejemplo, una empresa podría hacer pública su información de configuración de red completa, pero mantener en secreto dicha información haría que fuera más difícil para los intrusos saber qué atacar o determinar qué puede ser detectado. Incluso en este caso, una empresa que suponga que dicha información se mantendrá en secreto estará cometiendo un grave error.

15.6.3 Detección de intrusiones

Dotar de seguridad a los sistemas e instalaciones está íntimamente relacionado con la detección de intrusiones. La **detección de intrusiones**, como su nombre sugiere, consiste en detectar los intentos de intrusión y las intrusiones que hayan tenido éxito en los sistemas informáticos, e ini-

ciar las apropiadas respuestas a dichas intrusiones. La detección de intrusiones conlleva una amplia matriz de técnicas, que varían según diversos ejes. Estos ejes incluyen:

- El instante en que se produce la detección. La detección puede tener lugar en tiempo real (mientras se produce la intrusión) o después de que haya ocurrido.
- Los tipos de informaciones examinadas para detectar la actividad de intrusión. Entre estas informaciones pueden incluirse los comandos de la *shell* del usuario, las llamadas al sistema por parte de los procesos y el contenido o la cabecera de los paquetes de red. Algunas formas de intrusión sólo pueden detectarse correlando la información de varias de estas fuentes.
- El rango de las capacidades de respuesta. Entre las formas simples de respuesta se incluyen el alertar a un administrador de una potencial intrusión o de algún modo detener la actividad potencialmente intrusiva, por ejemplo interrumpiendo un proceso ocupado en una actividad aparentemente intrusiva. En una forma sofisticada de respuesta, un sistema podría desviar de forma transparente la actividad de un intruso a lo que se denomina un **tarro de miel**: un falso recurso que queda expuesto al atacante. Dicho recurso parece real a los ojos del atacante y permite al sistema monitorizar y obtener información acerca del ataque.

Estos grados de libertad en el espacio de diseño para la detección de intrusiones han dado lugar a un amplio rango de soluciones, conocidas como **sistemas de detección de intrusiones** (IDS, intrusion-detection system) y sistemas de prevención de intrusiones (IDP, intrusion-prevention system). Los sistemas IDS activan una alarma cuando se detecta una intrusión, mientras que los sistemas IDP actúan como encaminadores, dejando pasar el tráfico hasta que se detecta una intrusión (momento en el que se bloquea el tráfico).

Pero, ¿qué constituye una intrusión? Definir adecuadamente el concepto de intrusión es bastante complicado, por lo que los sistemas IDS e IDP actuales normalmente se contentan con uno de dos métodos no tan ambiciosos. Con el primero, denominado **detección basada en signatura**, se examinan la entrada del sistema o el tráfico de red en busca de patrones de comportamiento específicos (es decir, **signaturas**) que se sabe que son indicativos de que se está produciendo un ataque. Un ejemplo sencillo de detección basada en signatura consiste en analizar los paquetes de red en busca de la cadena */etc/passwd/* dirigida a un sistema UNIX. Otro ejemplo sería el software de detección de virus, que analiza los archivos ejecutables o los paquetes de red en busca de virus conocidos.

El segundo método, normalmente denominado **detección de anomalías**, intenta mediante varias técnicas detectar comportamientos anómalos en los sistemas informáticos. Por supuesto, no todas las actividades anómalas del sistema indican que se ha producido una intrusión, pero se presupone que a menudo las intrusiones inducen comportamientos anómalos. Un ejemplo de detección de anomalías consistiría en monitorizar las llamadas al sistema de un proceso demonio para detectar si el comportamiento de las llamadas al sistema se desvía de los patrones normales, posiblemente indicando que ha sido explotado un desbordamiento de buffer en el programa demonio para corromper su comportamiento. Otro ejemplo sería la monitorización de los comandos de la *shell* para detectar comandos anómalos de un determinado usuario o detectar un inicio de sesión anormal de un usuario, lo que podría indicar que un atacante ha conseguido obtener acceso a dicha cuenta de usuario.

La detección basada en signatura y la detección de anomalías pueden verse como las dos caras de una misma moneda: la detección basada en signatura intenta caracterizar los comportamientos peligrosos y detectar cuándo se produce uno de ellos, mientras que la detección de anomalías intenta caracterizar los comportamientos normales (o no peligrosos) y detectar cuándo ocurre algo distinto a esos comportamientos.

Sin embargo, estos diferentes métodos dan lugar a sistemas IDS e IDP con muy diferentes propiedades. En particular, la detección de anomalías puede detectar métodos de intrusión desconocidos con anterioridad (lo que se denomina **ataques de día cero**). En contraste, la detección basada en signatura identificará sólo los ataques conocidos que puedan codificarse en un patrón reconocible; por tanto, los nuevos ataques que no se hayan contemplado cuando se generaron las signa-

turas eludirán la detección basada en signatura. Los proveedores de software de detección de virus conocen bien este problema y deben proporcionar nuevas signaturas con mucha frecuencia a medida que se detectan manualmente nuevos virus.

Sin embargo, la detección de anomalías no es necesariamente superior a la detección basada en signatura. Realmente, un desafío importante para los sistemas que emplean la detección de anomalías es establecer un marco de referencia para describir de forma precisa lo que constituye el comportamiento "normal" del sistema. Si el sistema ya ha sido penetrado cuando se establece el marco de referencia, entonces la actividad intrusiva puede ser incluida en el marco "normal". Incluso aunque se defina el marco de referencia del sistema cuando éste está limpio, sin influencia de un comportamiento intrusivo, ese marco de referencia debe proporcionar una imagen completa del comportamiento normal. De otra manera, el número de **falsos positivos** (falsas alarmas), o peor, **falsos negativos** (intrusiones no detectadas), será excesivo.

Para ilustrar el impacto de una tasa de falsas alarmas que fuera simplemente un poco alta, considere una instalación que conste de un centenar de estaciones de trabajo UNIX en las que se han grabado, para poder detectar las intrusiones, registros de los sucesos de seguridad relevantes. Una pequeña instalación tal como ésta podría generar fácilmente un millón de registros de auditoría por día. Sólo uno o dos sucesos podrían merecer la atención de una investigación por parte del administrador. Si suponemos, siendo optimistas, que cada ataque se refleja en diez registros de auditoría, podemos entonces calcular del siguiente modo aproximado la tasa de registros de auditoría que reflejan una actividad verdaderamente intrusiva

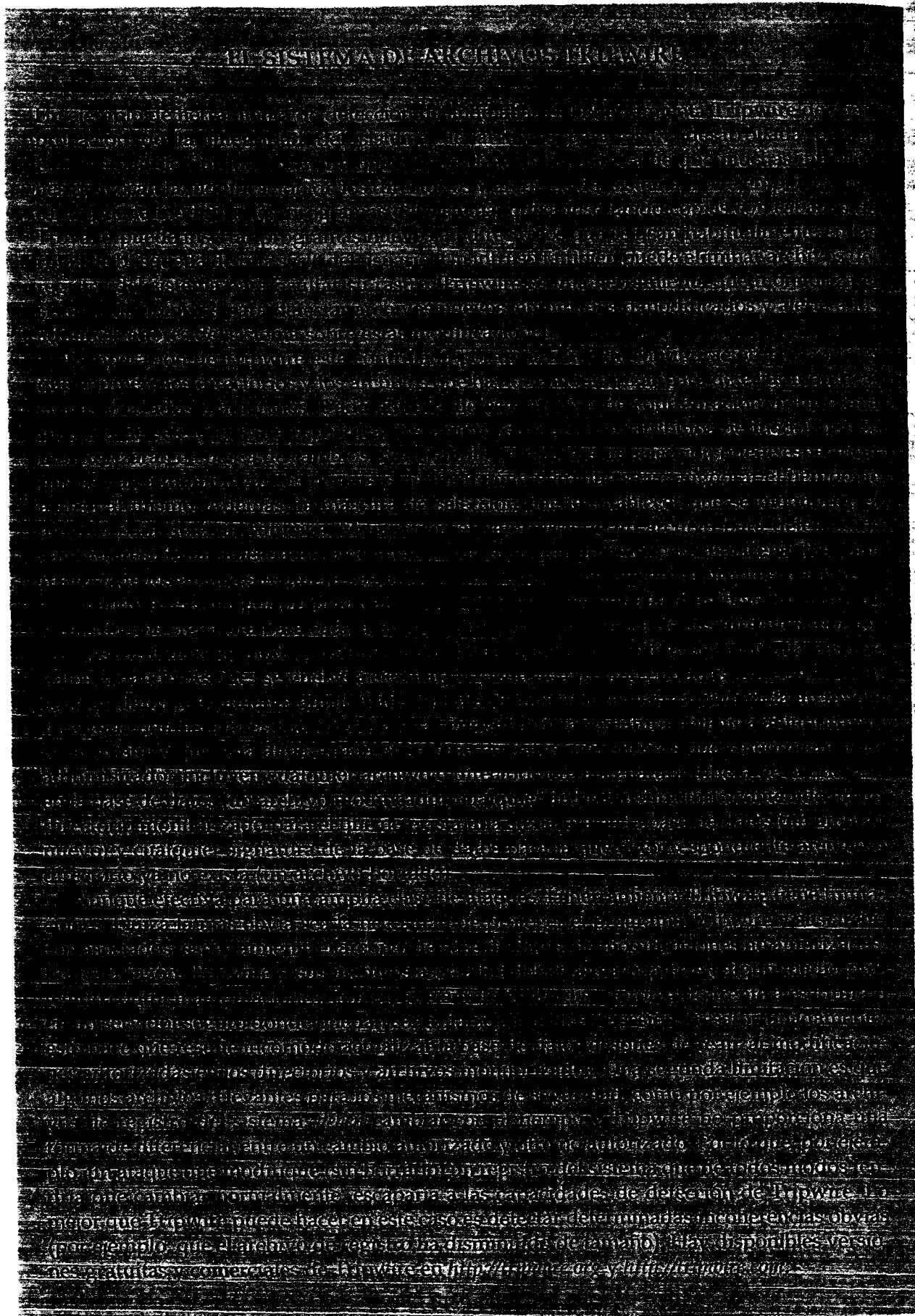
$$\frac{2 \frac{\text{intrusiones}}{\text{día}} \cdot 10 \frac{\text{registros}}{\text{intrusión}}}{10^6 \frac{\text{registros}}{\text{día}}} = 0,00002$$

Interpretando esto como una "probabilidad de aparición de registros intrusivos", podemos designarlos como $P(I)$; es decir, el suceso I es la aparición de un registro que refleja un comportamiento verdaderamente intrusivo. Dado que $P(I) = 0,00002$, también sabemos que $P(\neg I) = 1 - P(I) = 0,99998$. Ahora, vamos a designar mediante A la generación de una alarma por parte de un sistema IDS. Un sistema IDS preciso debería maximizar tanto $P(I|A)$ como $P(\neg I|\neg A)$, es decir, las probabilidades de que una alarma indique una intrusión y de que la falta de alarma indique que no se ha producido una intrusión. Si nos centramos por el momento en $P(I|A)$, podemos calcularla utilizando el **teorema de Bayes**:

$$\begin{aligned} P(I|A) &= \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} \\ &= \frac{0,00002 \cdot P(A|I)}{0,00002 \cdot P(A|I) + 0,99998 \cdot P(A|\neg I)} \end{aligned}$$

Consideremos ahora el impacto de la tasa de falsas alarmas $P(A|\neg I)$ sobre $P(I|A)$. Incluso con una buena tasa de alarmas verdaderas $P(A|I) = 0,8$, una tasa de falsas alarmas aparentemente buena de $P(A|\neg I)$ nos da una $P(I|A) \approx 0,14$. Es decir, ¡tan sólo una de cada siete alarmas indica una intrusión real! En los sistemas en los que un administrador de seguridad tenga que investigar cada una de las alarmas, una alta tasa de falsas alarmas (lo que se denomina efecto "árbol de Navidad") provoca que se pierda demasiado tiempo y hace que el administrador termine rápidamente por ignorar las alarmas.

Este ejemplo ilustra un principio general para los sistemas ids e idp: por cuestiones de usabilidad, estos sistemas deben ofrecer una tasa de falsas alarmas extremadamente baja. Como se ha mencionado, conseguir una tasa de falsas alarmas lo suficientemente baja es un desafío especialmente serio en los sistemas de detección de anomalías, debido a las dificultades de definir adecuadamente un marco de referencia que describa el comportamiento normal del sistema. Sin embargo, la investigación continúa para mejorar las técnicas de detección de anomalías. El soft-



ware de detección de intrusiones está evolucionando para implementar signaturas, algoritmos de detección de anomalías y otros algoritmos, y para combinar los resultados con el fin de obtener una tasa de detección de anomalías más precisa.

15.6.4 Protección frente a virus

Como hemos visto, los virus pueden causar estragos en los sistemas y a menudo lo hacen. La protección frente a virus es, por tanto, una de las principales cuestiones de seguridad. A menudo se emplean programas antivirus para proporcionar esta protección. Algunos de estos programas son efectivos sólo frente a algunos virus concretos conocidos; estos programas funcionan buscando en todos los programas del sistema el patrón específico de instrucciones conocidas que definen el virus. Cuando encuentran un patrón conocido, eliminan las instrucciones, **desinfectando** el programa. Los programas antivirus pueden disponer de catálogos de miles de virus, que son los que tratan de buscar.

Tanto los virus como el software antivirus son cada vez más sofisticados. Algunos virus se modifican a sí mismos cuando infectan otro software, para evitar el método básico de búsqueda de patrones utilizado por los programas antivirus. A su vez, los programas antivirus ahora buscan familias de patrones en lugar de un solo patrón para identificar un virus. De hecho, algunos programas antivirus implementan una variedad de algoritmos de detección. Pueden descomprimir virus comprimidos antes de comprobar una firma; algunos también buscan anomalías en los procesos. Por ejemplo, un proceso que abra un archivo ejecutable para escribir es sospechoso, a menos que sea un compilador. Otra técnica popular es la de ejecutar un programa en lo que se denomina un **cajón de arena**, que es una sección emulada o controlada del sistema. El software antivirus analiza el comportamiento del código en el cajón de arena antes de dejar que se ejecute sin monitorización. Algunos programas antivirus también realizan un análisis integral en lugar de limitarse a analizar los archivos contenidos en el sistema de archivos: buscan en los sectores de arranque, en la memoria, en los correos electrónicos de entrada y de salida, en los archivos descargados, en los archivos almacenados en dispositivos extraíbles u otros medios de almacenamiento, etc.

La mejor protección frente a los virus informáticos es la prevención, o la práctica de la **informática segura**. Adquirir software de primera mano a los distribuidores y evitar el uso de copias gratuitas o pirateadas procedentes de fuentes públicas o del intercambio de discos constituye el método más seguro para evitar la infección. Sin embargo, incluso las copias nuevas de aplicaciones software legales no son inmunes a la infección por virus: se han producido casos en los que empleados disgustados de una empresa de software han infectado las copias maestras de los programas software para dañar económicamente a la empresa fabricante. Para los virus de tipo macro, una defensa consiste en cambiar los documentos de Word a un formato alternativo denominado **RTF** (rich text format). A diferencia del formato nativo de Word, RTF no incluye la capacidad de adjuntar macros.

Otra forma de defensa consiste en evitar abrir cualquier adjunto de los mensajes de correo electrónico de usuarios desconocidos. Desafortunadamente, la historia ha demostrado que las vulnerabilidades del correo electrónico aparecen tan rápido como son resueltas. Por ejemplo, en el año 2000, el virus *love bug* se extendió muy rápidamente, mostrándose como una nota de amor enviada por un amigo del receptor. Una vez que se abría el *script* adjunto de Visual Basic, el virus se propagaba enviándose a sí mismo a los primeros usuarios de la lista de contactos del usuario infectado. Afortunadamente, excepto por el atasco de los sistemas de correo electrónico y de las bandejas de entrada de los usuarios, ese virus era relativamente inofensivo. Sin embargo, derrotaba de forma efectiva la estrategia defensiva de abrir sólo los adjuntos procedentes de personas conocidas por el receptor. Un método de defensa más efectivo es el de evitar abrir cualquier adjunto de correo electrónico que contenga código ejecutable. Algunas compañías imponen ahora esta política, eliminando todos los adjuntos de los mensajes de correo electrónico entrantes.

Hay otra salvaguarda que, aunque no previene infecciones, permite la detección temprana. Un usuario debe comenzar por reformatear por completo el disco duro, especialmente el sector de arranque, que es con frecuencia el objetivo de los ataques de los virus. Después, sólo debe descar-

garse software seguro, calculándose a continuación una firma de cada programa mediante un algoritmo suficientemente seguro de cálculo de resúmenes. La lista resultante de nombres de archivo y resúmenes asociados debe mantenerse libre de accesos no autorizados. Periódicamente, o cada vez que se ejecute un programa, el sistema operativo calcula de nuevo la firma y la compara con la incluida en la lista original; de este modo, cualquier diferencia sirve como advertencia de una posible infección. Esta técnica puede combinarse con otras. Por ejemplo, puede utilizarse un análisis antivirus complejo (como por ejemplo la técnica del cajón de arena) y sólo si el programa pasa la prueba se crea una firma para él. Si las firmas son iguales la siguiente vez que se ejecute el programa, no será necesario volver a realizar un análisis antivirus del mismo.

15.6.5 Auditoría, contabilización y registro

La auditoría, la contabilización y la elaboración de registros pueden disminuir el rendimiento del sistema, pero resultan útiles en diversas áreas, incluyendo la de la seguridad. La tarea de registro puede ser general o específica. Pueden registrarse todas las ejecuciones de llamadas al sistema, para poder analizar el comportamiento (o el mal comportamiento) del sistema, pero normalmente sólo se registran los sucesos sospechosos. Los fallos de autenticación y los fallos de autorización pueden decírnos bastante acerca de los intentos de penetrar en el sistema.

La contabilización es otra arma potencial dentro del conjunto de herramientas de los administradores de seguridad. Puede utilizarse para detectar cambios en el rendimiento, los cuales a su vez pueden revelar problemas de seguridad. Una de las primeras intromisiones en computadoras UNIX fue detectada por Cliff Stoll cuando examinaba los registros de contabilización y localizó una anomalía.

15.7 Cortafuegos para proteger los sistemas y redes

A continuación vamos a volver sobre la cuestión de cómo puede conectarse de forma segura una computadora de confianza a una red que no sea de confianza. Una solución consiste en utilizar un cortafuegos para separar los sistemas de confianza y de no confianza. Un cortafuegos es una computadora, dispositivo o encaminador que se sitúa entre el sistema seguro y el que no lo es. Un cortafuegos de red limita el acceso a la red entre los dos **dominios de seguridad** y monitoriza y registra todas las conexiones. También puede limitar las conexiones basándose en la dirección de origen o de destino, el puerto de origen o de destino o la dirección de la conexión. Por ejemplo, los servidores web usan HTTP para comunicarse con los exploradores web. Por tanto, puede utilizarse un cortafuegos para permitir que sólo el tráfico HTTP pase desde todos los *hosts* externos al cortafuegos hacia el servidor web situado detrás del cortafuegos. Por ejemplo, el gusano Internet de Morris utilizaba el protocolo finger para entrar en las computadoras, por lo que podríamos usar el cortafuegos para impedir el tráfico entrante de finger.

De hecho, un cortafuegos de red permite dividir la red en múltiples dominios. Una implementación habitual define varios dominios distintos: el dominio no seguro constituido por Internet; otro dominio semi-seguro, denominado **zona desmilitarizada** (DMZ, demilitarized zone); y un tercer dominio formado por las computadoras de la empresa (Figura 15.10). Las conexiones entre Internet y las computadoras DMZ, y entre las computadoras de la empresa e Internet se permiten, pero no se permiten las comunicaciones entre Internet o las computadoras DMZ y las computadoras de la empresa. Opcionalmente, pueden permitirse comunicaciones controladas entre la zona DMZ y una o más computadoras de la empresa: por ejemplo, un servidor web en la zona DMZ puede necesitar consultar un servidor de base de datos de la red corporativa. Sin embargo, con un cortafuegos, el acceso está regulado y, si alguien irrumpiera en cualquiera de los sistemas DMZ, seguiría sin poder acceder a las computadoras de la empresa.

Por supuesto, el propio cortafuegos debe ser seguro y resistente a los ataques; de otro modo, su capacidad para dotar de seguridad a las conexiones puede verse comprometida. Además, los cortafuegos no impiden los ataques de tipo túnel, es decir, los ataques contenidos dentro de protocolos o conexiones que el cortafuegos permita. Por ejemplo, un cortafuegos no detendrá un ataque por desbordamiento de búfer a un servidor web, ya que la conexión HTTP está permitida y es

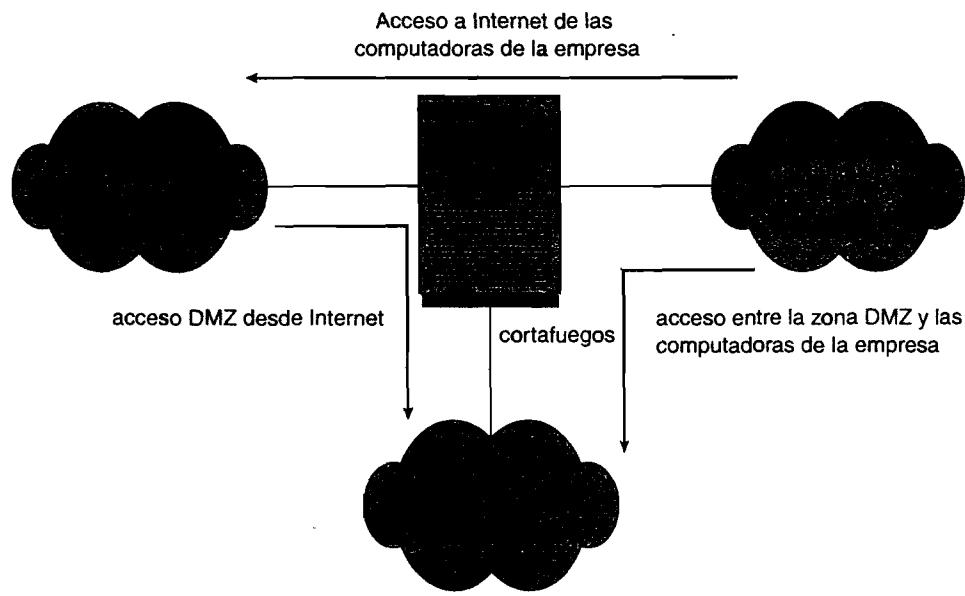


Figura 15.10 Separación de dominios mediante un cortafuegos.

el propio contenido de la conexión HTTP lo que se utiliza como mecanismo de ataque. Del mismo modo, los ataques por denegación de servicio pueden afectar a los cortafuegos al igual que a otras máquinas. Otra vulnerabilidad de los cortafuegos es la **suplantación**, que es un ataque en el que un *host* no autorizado pretende ser un *host* autorizado, cumpliendo algunos de los criterios de autorización. Por ejemplo, si una regla del cortafuegos permite una conexión de un *host* e identifica a dicho *host* por su dirección IP, entonces otro *host* podría enviar paquetes usando esa misma dirección y atravesar así el cortafuegos.

Además de los cortafuegos de red más comunes, existen otras clases de cortafuegos más novedosas, cada una ellas con sus ventajas e inconvenientes. Un **cortafuegos personal** es una capa software incluida en el sistema operativo o que se añade como una aplicación. En lugar de limitar la comunicación entre dominios de seguridad, limita la comunicación a (y posiblemente desde) un determinado *host*. Un usuario puede añadir un cortafuegos personal a su PC con el fin de denegar a un caballo de Troya el acceso a la red a la que esté conectado el PC. Un cortafuegos de tipo **proxy de aplicación** entiende los protocolos que utilizan las aplicaciones a lo largo de la red, como por ejemplo SMTP, que se usa para la transferencia de correo electrónico. Un *proxy* de aplicación acepta una conexión de la misma forma en que lo haría un servidor SMTP y luego inicia una conexión con el servidor SMTP de destino original. El *proxy* puede monitorizar el tráfico a medida que reenvía el mensaje, localizando y desactivando comandos ilegales, los intentos de explotar errores en el software, etc. Algunos cortafuegos están diseñados para un protocolo específico: por ejemplo, un **cortafuegos XML** tiene el propósito específico de analizar el tráfico XML y de bloquear el código XML no permitido o mal definido. Los **cortafuegos de llamadas al sistema** se ubican entre las aplicaciones y el *kernel*, monitorizando la ejecución de las llamadas al sistema. Por ejemplo, en Solaris 10, la funcionalidad de “menor privilegio” implementa una lista de más de cincuenta llamadas al sistema que los procesos pueden o no permitir realizar. Por ejemplo, un proceso que no tenga necesidad de crear otros procesos puede desactivar dicha capacidad.

15.8 Clasificaciones de seguridad informática

El Departamento de Defensa de los EE. UU especifica cuatro clasificaciones de seguridad para los sistemas: A, B, C y D. Esta especificación se usa ampliamente en dicho país para determinar la seguridad de una instalación y modelar soluciones de seguridad, como vamos a ver aquí. La clasificación de nivel más bajo es la división D, o protección mínima. La división D incluye sólo una

clase y se usa para los sistemas que no cumplen los requisitos de ninguna de las otras clases de seguridad. Por ejemplo, MS-DOS y Windows 3.1 pertenecen a la división D.

La división C, el siguiente nivel de seguridad, proporciona una protección discrecional y mecanismos de atribución de responsabilidad a los usuarios y de control de sus acciones, mediante el uso de las capacidades de auditoría. La división C tiene dos niveles: C1 y C2. Un sistema de clase C1 incorpora algún tipo de controles que permiten a los usuarios proteger información privada e impedir a otros usuarios la lectura o la destrucción accidental de sus datos. Un entorno C1 es aquél en el que una serie de usuarios cooperantes acceden a un conjunto de datos con el mismo nivel de confidencialidad. La mayoría de las versiones de UNIX son de clase C1.

La suma total de todos los sistemas de protección de un sistema informático (hardware, software, firmware) que permiten imponer correctamente una determinada política de seguridad se denomina **base informática de confianza** (TCB, trusted computer base). La TCB de un sistema C1 controla el acceso a los archivos por parte de los usuarios, permitiendo al usuario especificar y controlar la compartición de objetos de forma individual, mediante su nombre o mediante grupos definidos. Además, la TCB requiere que los usuarios se identifiquen antes de iniciar cualquier actividad en la que la TCB tenga que mediar. Esta identificación se lleva a cabo mediante un mecanismo protegido o una contraseña; la TCB protege los datos de autenticación de modo que no sean accesibles por parte de usuarios no autorizados.

Un sistema de clase C2 añade un control de acceso de nivel individual a los requisitos de un sistema C1. Por ejemplo, los derechos de acceso de un archivo pueden especificarse en el nivel de un solo individuo. Además, el administrador del sistema puede auditar selectivamente las acciones de uno o más usuarios basándose en su identidad individual. La TCB también se protege a sí misma de la modificación de su código o de sus estructuras de datos. Además, ninguna información generada por un usuario anterior estará disponible para otro usuario que acceda a un objeto de almacenamiento que haya sido devuelto al sistema. Algunas versiones seguras especiales de UNIX han sido certificadas en el nivel C2.

Los sistemas de protección obligatoria de la división B tienen todas las propiedades de un sistema de clase C2 y además asignan un nivel de seguridad a cada objeto. La TCB de la clase B1 mantiene el nivel de seguridad de cada objeto del sistema; dicho nivel se usa para tomar decisiones relativas al control de acceso obligatorio. Por ejemplo, un usuario en el nivel confidencial no podría acceder a un archivo en el nivel de secreto (que tiene un nivel de seguridad más alto). El TCB también indica el nivel de seguridad en la parte superior y la inferior de cada página que sea legible por una persona. Además de la información normal de autenticación (nombre de usuario y contraseña), la TCB mantiene también el nivel de seguridad y las autorizaciones de los usuarios y soporta al menos dos niveles de seguridad. Estos niveles son jerárquicos, por lo que un usuario puede acceder a cualquier objeto que tenga un nivel de seguridad igual o menor que el suyo propio. Por ejemplo, un usuario del nivel secreto podría acceder a un archivo del nivel confidencial en ausencia de otros controles de acceso. Los procesos también se aíslan usando espacios de direcciones diferentes.

Un sistema de clase B2 amplía los niveles de seguridad a cada recurso del sistema, como por ejemplo los objetos de almacenamiento. A los dispositivos físicos se les asignan niveles de seguridad mínimos y máximos que el sistema utiliza para forzar las restricciones impuestas por los entornos físicos en los que se encuentren los dispositivos. Además, un sistema B2 soporta canales encubiertos y mecanismos de auditoría de los sucesos que podrían llevar a la explotación de un canal encubierto.

Un sistema de clase B3 permite la creación de listas de control de acceso que especifican los usuarios o grupos que *no* tienen acceso a un objeto con un nombre determinado. La TCB también contiene un mecanismo para supervisar los sucesos que pueden indicar una violación de la política de seguridad. El mecanismo envía una notificación al administrador de seguridad y, si es necesario, termina el suceso de la forma menos disruptiva posible.

La clasificación de nivel superior es la división A. Arquitectónicamente, un sistema de clase A1 es funcionalmente equivalente a un sistema B3, pero usa especificaciones de diseño y técnicas de verificación formales, que conceden un alto grado de certeza de que la TCB se ha implementado correctamente. Los sistemas por encima de la clase A1 pueden ser diseñados y desarrollados por personal de confianza en una instalación segura.

El uso de una TCB simplemente asegura que el sistema pueda imponer determinados aspectos de una política de seguridad; la TCB no especifica cuál debe ser esa política. Normalmente, un entorno informático dado (en los EE.UU.) puede desarrollar una política de seguridad con vistas a su **certificación** y hacer que ese plan sea **aprobado** por una agencia de seguridad, como por ejemplo NCSC (National Computer Security Center). Determinados entornos informáticos pueden requerir otro tipo de certificación, como la que suministra TEMPEST, que protege frente a escuchas electrónicas ilegales. Por ejemplo, un sistema con certificación TEMPEST tiene terminales apantallados para impedir la fuga de campos electromagnéticos. Este apantallamiento asegura que los equipos externos a la sala o al edificio donde se encuentre el terminal no puedan detectar qué información está mostrando éste.

15.9 Un ejemplo: Windows XP

Microsoft Windows XP es un sistema operativo de propósito general diseñado para soportar una variedad de métodos y características de seguridad. En esta sección, vamos a examinar los mecanismos que Windows XP utiliza para implementar las funciones de seguridad. Para obtener más información sobre Windows XP, véase el Capítulo 22.

El modelo de seguridad de Windows XP se basa en la noción de **cuentas de usuario**. Windows XP permite la creación de cualquier número de cuentas de usuario, que pueden agruparse de cualquier manera. El acceso a los objetos del sistema puede entonces permitirse o denegarse como se desee. Los usuarios se identifican ante el sistema mediante un ID de seguridad único. Cuando un usuario inicia una sesión, Windows XP crea un **testigo de acceso de seguridad** que incluye el ID de seguridad del usuario, los ID de seguridad de los grupos de los que el usuario sea miembro y una lista de los privilegios especiales que tenga el usuario. Ejemplos de privilegios serían la realización de copias de seguridad de archivos y directorios, apagar la computadora, iniciar una sesión interactivamente y cambiar la hora del sistema. Todo proceso que Windows XP ejecute en nombre de un usuario recibirá una copia del testigo de acceso. El sistema usa los ID de seguridad del testigo de acceso para conceder o denegar el acceso a los objetos del sistema cuando el usuario, o un proceso en nombre del usuario, intenta acceder al objeto. La autenticación de una cuenta de usuario normalmente se realiza a través de un nombre de usuario y de una contraseña, aunque el diseño modular de Windows XP permite el desarrollo de paquetes de autenticación personalizados. Por ejemplo, puede utilizarse un escáner retinal (u ocular) para verificar que el usuario es quien dice ser.

Windows XP utiliza la idea de sujeto para asegurar que los programas ejecutados por un usuario no obtengan un acceso mayor al sistema que el que el usuario esté autorizado a tener. Un **sujeto** se usa para controlar y gestionar los permisos para cada programa que ejecute un usuario y está compuesto por el testigo de acceso del usuario y por el programa que actúa en nombre del usuario. Dado que Windows XP opera con un modelo cliente-servidor, se usan dos clases de sujetos para controlar el acceso: sujetos simples y sujetos de servidor. Un ejemplo de **sujeto simple** sería el típico programa de aplicación que un usuario ejecuta después de iniciar una sesión. Al sujeto simple se le asigna un **contexto de seguridad** basado en el testigo de acceso de seguridad del usuario. Un **sujeto de servidor** es un proceso implementado como un servidor protegido que utiliza el contexto de seguridad del cliente cuando actúa en nombre de éste.

Como se ha mencionado en la Sección 15.7, la auditoría es una técnica de seguridad muy útil. Windows XP tiene mecanismos de auditoría incorporados que permiten monitorizar muchas amenazas de seguridad comunes. Algunos ejemplos son la auditoría de fallos en los sucesos de inicio y fin de sesión para detectar los intentos de ruptura de contraseñas, las auditorías de éxito para los sucesos de inicio y fin de sesión con el fin de detectar la actividad a horas extrañas, auditorías de fallo y de éxito de los accesos de escritura para archivos ejecutables para detectar apariciones de virus, y la auditoría de fallo o éxito del acceso a archivos para detectar el acceso a archivos confidenciales.

Los atributos de seguridad de un objeto en Windows XP se describen mediante un **descriptor de seguridad**. El descriptor de seguridad contiene el ID de seguridad del propietario del objeto (el cual puede cambiar los permisos de acceso), un ID de seguridad de grupo usado sólo por el sub-

sistema POSIX, una lista discrecional de control de acceso que identifica qué usuarios o grupos tienen permiso (y no tienen permiso) de acceso y una lista de control de acceso del sistema que controla qué mensajes de auditoría generará el sistema. Por ejemplo, el descriptor de seguridad del archivo *foo.bar* podría tener el propietario *avi* y la siguiente lista discrecional de control de acceso:

- *avi*– todos los accesos
- grupo *cs*– acceso de lectura-escritura
- usuario *cliff*– sin acceso

Además, puede existir una lista de control de acceso del sistema para las escrituras de auditoría realizadas por cualquiera.

Una lista de control de acceso está formada por entradas de control de acceso que contienen el ID de seguridad del individuo y una máscara de acceso que define todas las posibles acciones sobre el objeto, con un valor de AccessAllowed (permitido) o AccessDenied (no permitido) para cada acción. Los archivos en Windows XP pueden tener los siguientes tipos de acceso: ReadData, WriteData, AppendData, Execute, ReadExtendedAttribute, WriteExtendedAttribute, ReadAttributes y WriteAttributes. Podemos ver fácilmente cómo esto permite un detallado grado de control sobre el acceso a los objetos.

Windows XP clasifica los objetos bien como objetos contenedores o como objetos no contenedores. Los **objetos contenedores**, tales como directorios, pueden contener lógicamente a otros objetos. Por omisión, cuando se crea un objeto dentro de un objeto contenedor, el nuevo objeto hereda los permisos del objeto padre. Del mismo modo, si el usuario copia un archivo de un directorio en un nuevo directorio, el archivo heredará los permisos del directorio de destino. Los **objetos no contenedores** no heredan ningún otro permiso. Además, si se cambia un permiso en un directorio, los nuevos permisos no se aplicarán automáticamente a los archivos y subdirectorios existentes; el usuario debe aplicarlos explícitamente si lo desea.

El administrador del sistema puede prohibir la impresión en una impresora del sistema durante todo o parte de un día y puede usar el Monitor de rendimiento de Windows XP como ayuda para detectar los problemas potenciales. En general, Windows XP proporcionando características muy adecuadas que ayudan a conseguir un entorno informático seguro. Sin embargo, muchas de estas características no están habilitadas de manera predeterminada, lo que puede ser una de las razones de la multitud de brechas de seguridad experimentadas por los sistemas Windows XP. Otra razón es el vasto número de servicios que Windows XP inicia al arrancar el sistema y la cantidad de aplicaciones que normalmente se instalan en un sistema Windows XP. En un entorno multiusuario real, el administrador del sistema debería formular un plan de seguridad e implementarlo, usando las características que Windows XP proporciona y otras herramientas de seguridad.

15.10 Resumen

La protección es un problema interno. Por el contrario, la seguridad debe tener en cuenta tanto el propio sistema informático como el entorno (personas, edificios, empresas, objetos valiosos y amenazas) dentro del que se utiliza el sistema.

Los datos almacenados en el sistema informático deben protegerse frente a accesos no autorizados, destrucción o alteración maliciosa e introducción accidental de incoherencias. Es más fácil establecer una protección frente a la pérdida accidental de la coherencia de los datos que proteger del acceso malicioso a los datos. La protección absoluta de la información almacenada en un sistema informático frente a abusos malintencionados es imposible; pero el coste que tendrá pagar el autor de ese abuso puede ser lo suficientemente alto como para disuadirle en la mayoría de los casos, si no en todos, de intentar acceder a dicha información sin la apropiada autorización.

Pueden lanzarse varios tipos de ataques contra los programas y contra las computadoras, bien en forma de ataques individuales o de ataques masivos. Las técnicas de desbordamiento de búfer y de pila permiten a los atacantes cambiar su nivel de acceso al sistema. Los virus y los gusanos

se auto-replican, infectando a veces miles de computadoras. Los ataques por denegación de servicio impiden el uso legítimo de los sistemas objeto del ataque.

Las técnicas de cifrado limitan el dominio de los receptores de datos, mientras que la autenticación limita el dominio de los emisores. El cifrado se emplea para proporcionar confidencialidad a los datos que se van a almacenar o a transferir. El cifrado simétrico requiere una clave compartida, mientras que el cifrado asimétrico utiliza una clave pública y otra clave privada. La autenticación, cuando se combina con el cálculo de valores *hash*, puede demostrar que los datos no han sido modificados.

Los métodos de autenticación del usuario se utilizan para identificar a los usuarios legítimos de un sistema. Además de la protección estándar mediante un nombre de usuario y una contraseña, se utilizan varios métodos de autenticación. Por ejemplo, las contraseñas de un solo uso cambian de una sesión a otra para evitar los ataques por repetición. La autenticación mediante dos factores requiere dos formas de autenticación, como por ejemplo una calculadora hardware con un PIN de activación. La autenticación mediante múltiples factores utiliza tres o más formas de autenticación. Estos métodos disminuyen enormemente la probabilidad de falsificar una autenticación.

Los métodos para prevenir o detectar incidentes de seguridad incluyen los sistemas de detección de intrusiones, el software antivirus, la auditoría y registro de los sucesos del sistema, la monitorización de los cambios en el software del sistema, la monitorización de las llamadas al sistema y los cortafuegos.

Ejercicios

- 15.1 Los ataques por desbordamiento de búfer pueden evitarse adoptando una mejor metodología de programación o usando un soporte hardware especial. Analice estas soluciones.
- 15.2 Una contraseña puede llegar a ser conocida por otros usuarios de diferentes formas. ¿Existe un método sencillo que permita detectar que se ha producido un suceso de este tipo? Explique su respuesta.
- 15.3 La lista de todas las contraseñas se guarda dentro del sistema operativo. Por tanto, si un usuario consigue leer esta lista, la protección de las contraseñas se verá comprometida. Sugiera un esquema que evite este problema. (Consejo: utilice representaciones internas y externas diferentes.)
- 15.4 ¿Cuál es el propósito de usar un aleatorizador junto con la contraseña proporcionada por el usuario? ¿Dónde debería almacenarse el aleatorizador y cómo debería emplearse?
- 15.5 Una adición experimental a UNIX permite a un usuario conectar un programa **watchdog** a un archivo. El programa **watchdog** se invoca cada vez que otro programa solicita acceder al archivo, y el **watchdog** concede o deniega el permiso correspondiente. Exponga dos ventajas y dos desventajas de usar programas **watchdog** como mecanismo de seguridad.
- 15.6 El programa COPS de UNIX explora un determinado sistema para detectar posibles agujeros de seguridad y alertar al usuario de los potenciales problemas. ¿Cuáles son los dos peligros potenciales de usar un sistema así para cuestiones de seguridad? ¿Cómo se pueden limitar o eliminar esos peligros?
- 15.7 Explique un medio mediante el cual los gestores de sistemas conectados a Internet podrían diseñar sus sistemas para limitar o eliminar el daño causado por un gusano. ¿Cuáles son los inconvenientes de realizar el cambio sugerido?
- 15.8 Argumente en favor o en contra de la sentencia judicial fallada contra Robert Morris, Jr., por el desarrollo y lanzamiento del gusano Internet descrito en la Sección 15.3.1.
- 15.9 Realice una lista de seis posibles problemas de seguridad que afecten al sistema informático de un banco. Para cada uno de los elementos de su lista, indique si esos problemas están

relacionados con la seguridad física, con la seguridad humana o con la del sistema operativo.

- 15.10 ¿Cuáles son las dos ventajas de cifrar los datos almacenados en un sistema informático?
- 15.11 ¿Qué programas informáticos utilizados habitualmente son propensos a los ataques por interposición? Explique las posibles soluciones para prevenir esta forma de ataque.
- 15.12 Compare los esquemas de cifrado simétrico y asimétrico y explique bajo qué circunstancias debería utilizarse uno u otro en un sistema distribuido.
- 15.13 ¿Por qué $D(k_e, N)(E(k_d, N)(m))$ no proporciona una autenticación del emisor? ¿Qué usos pueden hacerse de esta operación de cifrado?
- 15.14 Explique cómo puede utilizarse el algoritmo de cifrado asimétrico para conseguir los siguientes objetivos:
 - a. Autenticación: el receptor sabe que sólo el emisor puede haber generado el mensaje.
 - b. Secreto: sólo el receptor puede descifrar el mensaje.
 - c. Autenticación y secreto: sólo el receptor puede descifrar el mensaje y el receptor sabe que sólo el emisor puede haber generado el mensaje.
- 15.15 Considere un sistema que genera 10 millones de registros de auditoría por día. Suponga también que se producen una media de 10 ataques por día en este sistema y que cada ataque se refleja en 20 registros. Si el sistema de detección de intrusiones tiene una tasa de alarmas reales igual a 0,6 y una tasa de falsas alarmas de 0,0005, ¿qué porcentaje de las alarmas generadas por el sistema se corresponderá con intrusiones reales?

Notas bibliográficas

Explicaciones de carácter general relacionadas con la seguridad se proporcionan en Hsiao et al. [1979], Landwehr [1981], Denning [1982], Pfleeger [2003], Tanenbaum [2003] y Russell y Gangemi [1991]. También es de interés general la información que proporciona Lobel [1986]. Kurose y Ross [2005] se ocupan de las redes de computadoras.

Los problemas relacionados con el diseño y la verificación de los sistemas de seguridad se tratan en Rushby [1981] y Silverman [1983]. En Schell [1983] se describe un *kernel* de seguridad para una microcomputadora multiprocesador. En Rushby y Randell [1983] se describe un sistema de seguridad distribuido.

Morris y Thompson [1979] se ocupan de la seguridad de las contraseñas. Morshedian [1986] presenta métodos para luchar contra el pirateo de contraseñas. La autenticación de contraseñas en comunicaciones inseguras se trata en Lamport [1981]. El problema de la ruptura de contraseñas se examina en Seely [1989]. Lehmann [1987] y Reid [1987] analizan los modos de penetración en las computadoras. En Thompson [1984] se tratan las cuestiones relacionados con cuándo puede considerarse de confianza un programa informático.

Grampp y Morris [1984], Wood y Kochan [1985], Farrow [1986a], Farrow [1986b], Filipski y Hanko [1986], Hecht et al. [1988], Kramer [1988], y Garfinkel et al. [2003] realizan diversos tratamientos de la seguridad en UNIX. Bershad y Pinkerton [1988] presentan la extensión basada en watchdog a BSD UNIX. El paquete de análisis de seguridad COPS para UNIX fue escrito por Farmer en la Universidad de Purdue. Está disponible para los usuarios en Internet a través del programa FTP del host [ftp.uu.net](ftp://ftp.uu.net) en el directorio </pub/security/cops>.

Spafford [1989] presenta una discusión técnica detallada sobre los gusanos de Internet. El artículo de Spafford aparece junto con otros tres en una sección especial sobre el gusano Morris de Internet en *Communications of the ACM* (Volumen 32, Número 6, Junio de 1989).

En Bellovin [1989] se describen los problemas de seguridad asociados con la serie de protocolos TCP/IP. Los mecanismos comúnmente empleados para prevenir dichos ataques se exponen en Bellovin [2003]. Otro método para proteger a las redes de ataques internos consiste en dotar de

seguridad a la topología o a los mecanismos de descubrimiento de rutas. Kent et al. [2000], Hu et al. [2002], Zapata y Asokan [2002], y Hu y Perring [2004] presentan soluciones para el encaminamiento seguro. Savage et al. [2000] examina el ataque por denegación de servicio distribuido y propone soluciones de rastreo IP inverso para abordar el problema. Perlman [1988] propone un método para diagnosticar fallos cuando la red contiene encaminadores maliciosos.

Información sobre virus y gusanos puede encontrarse en <http://www.viruslist.com>, así como en Ludwig [1998] y Ludwig [2002]. Otros sitios que contienen información actualizada sobre seguridad son <http://www.trusecure.com> y <http://www.eeye.com>. Un buen documento sobre los peligros de la monocultura informática es el que puede encontrarse en <http://www.ccianet.org/papers/cyberinsecurity.pdf>.

Diffie y Hellman [1976] y Diffie y Hellman [1979] fueron los primeros investigadores que propusieron el uso del esquema de cifrado de clave pública. El algoritmo presentado en la Sección 15.4 se basa en el esquema de cifrado de clave pública; fue desarrollado por Rivest et al. [1978]. Lempel [1979], Simmons [1979], Denning y Denning [1979], Gifford [1982], Denning [1982], Ahituv et al. [1987], Schneier [1996] y Stallings [2003] exploran el uso de la criptografía en los sistemas informáticos. Alk [1983], Davies [1983], Denning [1983] y Denning [1984] analizan el tema de la protección de firmas digitales.

El gobierno de EE. UU., por supuesto, también se ocupa de la seguridad. El *Department of Defense Trusted Computer System Evaluation Criteria* (DoD [1985]), conocido también como *Orange Book* (libro naranja) describe un conjunto de niveles de seguridad y las características que un sistema operativo debe tener para incluirle en cada clasificación de seguridad. La lectura de este documento constituye un buen punto de partida para comprender los problemas de seguridad. El *Kit de recursos para estaciones de trabajo de Microsoft Windows NT* (Microsoft [1996]) describe el modelo de seguridad de NT y cómo utilizar dicho modelo.

El algoritmo RSA se presenta en Rivest et al. [+1978]. Puede encontrarse información sobre las actividades de NIST relacionadas con AES en <http://www.nist.gov/aes/>; también en este sitio puede encontrarse información sobre otros estándares criptográficos para los Estados Unidos. En <http://home.netscape.com/eng/ssl3/> puede encontrarse una exposición completa sobre SSL 3.0. En 1999, SSL 3.0 fue modificado ligeramente y presentado en un documento RFC de IETF con el nombre TLS.

El ejemplo de la Sección 15.6.3, que ilustra el impacto de la tasa de falsas alarmas en la efectividad de los sistemas IDS, está basado en Axelsson [1999]. Una descripción más completa del programa swatch y de su uso con syslog puede encontrarse en Hansen y Atkins [1993]. La descripción de Tripwire de la Sección 15.6.5 está basada en Kim y Spanfford [1993]. En Forrest et al. [1996] se describe un trabajo de investigación sobre la detección de anomalías basada en las llamadas al sistema.



Parte Seis

Sistemas distribuidos

Un sistema distribuido es una colección de procesadores que no comparten ni memoria, ni una señal de reloj. En lugar de ello, cada procesador tiene su propia memoria local y los procesadores se comunican entre sí a través de líneas de comunicación, como por ejemplo redes de área local o de área extensa. Los procesadores que componen un sistema distribuido varían tanto en tamaño como en función. Dichos sistemas pueden incluir pequeños dispositivos de mano o de tiempo real, computadoras personales, estaciones de trabajo y grandes sistemas informáticos de tipo mainframe.

Un sistema de archivos distribuido es un sistema de servicio de archivos cuyos usuarios, servidores y dispositivos de almacenamiento están dispersos entre los nodos de un sistema distribuido. De forma correspondiente, la actividad de servicio debe llevarse a cabo a través de la red; en lugar de un único repositorio de datos centralizado, existen múltiples dispositivos de almacenamiento independientes.

Entre los beneficios de un sistema distribuido se incluyen proporcionar a los usuarios acceso a los recursos mantenidos por el sistema y, por tanto, acelerar los cálculos y mejorar la disponibilidad de los datos y la fiabilidad. Sin embargo, como el sistema está distribuido, debe proporcionar mecanismos para la comunicación y sincronización de procesos, para tratar los problemas de interbloqueo y para gestionar aquellos fallos que no se encuentran en los sistemas centralizados.





Estructuras de los sistemas distribuidos

Un sistema distribuido es una colección de procesadores que no comparten ni memoria, ni una señal de reloj. En lugar de ello, cada procesador tiene su propia memoria local. Los procesadores se comunican entre sí a través de varios tipos de redes de comunicaciones, como buses de alta velocidad o líneas telefónicas. En este capítulo vamos a hablar de la estructura general de los sistemas distribuidos y de las redes que los interconectan. Resaltaremos las diferencias principales entre esos sistemas y los sistemas centralizados, en lo que respecta al diseño de sistemas operativos. En el Capítulo 17 se analizarán los sistemas de archivos distribuidos. Después, en el Capítulo 18 describiremos los métodos necesarios para que los sistemas operativos distribuidos puedan coordinar sus acciones.

OBJETIVOS DEL CAPÍTULO

- Proporcionar una panorámica de alto nivel de los sistemas distribuidos y de las redes que los interconectan.
- Analizar la estructura general de los sistemas operativos distribuidos.

16.1 Motivación

Un **sistema distribuido** es una colección de procesadores débilmente acoplados interconectados a través de una red de comunicaciones. Desde el punto de vista de un procesador concreto de un sistema distribuido, el resto de los procesadores y sus respectivos recursos son remotos, mientras que sus propios recursos son locales.

Los procesadores de un sistema distribuido pueden variar en cuanto a tamaño y a función. Pueden incluir pequeños microprocesadores, estaciones de trabajo, minicomputadoras y grandes sistemas informáticos de propósito general. Estos procesadores se designan mediante una serie de nombres, como *sitios*, *nodos*, *computadoras*, *máquinas* y *hosts*, dependiendo del contexto en el que se los mencione. Utilizaremos principalmente *nodo* para indicar la ubicación de una máquina y *host* para referirnos a un sistema específico en un determinado nodo. Generalmente, uno de los *hosts* de un nodo, el *servidor*, dispone de un recurso que otro *host* en otro nodo, el *cliente* (o usuario) desea utilizar. En la Figura 16.1 se muestra la estructura general de un sistema distribuido.

Hay cuatro razones principales para construir sistemas distribuidos: *compartición de recursos*, *aceleración de los cálculos*, *fiabilidad* y *comunicación*. En esta sección, vamos a analizar brevemente cada una de estas razones.

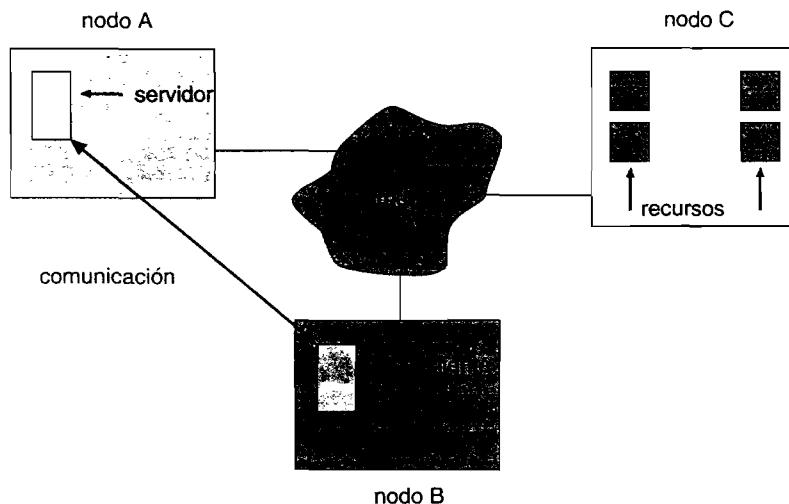


Figura 16.1 Un sistema distribuido.

16.1.1 Compartición de recursos

Si hay varias nodos (con capacidades diferentes) conectados entre sí, un usuario en uno de los nodos puede utilizar los recursos disponibles en otro. Por ejemplo, un usuario en el nodo A podría estar utilizando una impresora láser ubicada en el nodo B. Mientras tanto, un usuario en B podría acceder a un archivo que residiera en A. En general, la **compartición de recursos** en un sistema distribuido proporciona mecanismos para compartir archivos en sitios remotos, procesar información en una base de datos distribuida, imprimir archivos en sitios remotos, utilizar dispositivos hardware especializados remotos (como por ejemplo un procesador vectorial de alta velocidad) y realizar otras operaciones.

16.1.2 Aceleración de los cálculos

Si un determinado cálculo puede dividirse en una serie de cálculos de menor envergadura que puedan ejecutarse de forma concurrente, un sistema distribuido nos permitirá distribuir esos subcálculos entre los diversos nodos; los subcálculos podrán ejecutarse de manera concurrente, permitiéndonos obtener una **aceleración de los cálculos**. Además, si un nodo concreto está actualmente sobrecargado de trabajo, parte de las tareas pueden desplazarse a otros sitios que estén experimentando una menor carga. Esta transferencia de trabajos se denomina **compartición de carga**. La compartición automática de carga, en la que el sistema operativo distribuido transfiere automáticamente los trabajos, no resulta todavía común en los sistemas comerciales.

16.1.3 Fiabilidad

Si falla uno de los nodos de un sistema distribuido, los nodos restantes pueden continuar operando, proporcionando al sistema una mayor fiabilidad. Si el sistema está compuesto de múltiples instalaciones autónomas de gran tamaño (es decir, computadoras de propósito general), el fallo de una de ellas no debería afectar a las restantes. Sin embargo, si el sistema está compuesto de máquinas pequeñas, cada una de las cuales es responsable de alguna función crucial del sistema (como la E/S de caracteres para los terminales o el sistema de archivos), entonces un único fallo puede detener la operación de todo el sistema. En general, si existe la suficiente redundancia (tanto en el hardware como en los datos), el sistema podrá continuar operando incluso aunque fallen algunos de los nodos.

El fallo de un nodo debe ser detectado por el sistema y puede que sea necesario llevar a cabo las acciones apropiadas para recuperarse del fallo. El sistema no debe continuar utilizando los servicios de ese nodo. Además, si la función del nodo fallido puede ser asumida por otro nodo, el sis-

tema debe garantizar que la transferencia de esa función se produzca correctamente. Finalmente, cuando se recupere o repare el nodo fallido, deben existir mecanismos para volver a integrarlo con suavidad en el sistema. Como se expone en los Capítulos 17 y 18, estas acciones plantean problemas de gran dificultad para los que existen muchas posibles soluciones.

16.1.4 Comunicación

Cuando hay varios sitios conectados entre sí mediante una red de comunicaciones, los usuarios de los diferentes nodos tienen la oportunidad de intercambiar información. A bajo nivel, se pasan **mensajes** entre los sistemas de forma similar a como se pasan mensajes entre procesos en los sistemas de mensajería monomáquina de los que hemos hablado en la Sección 3.4. Con un mecanismo de paso de mensajes, toda la funcionalidad de nivel superior que podemos encontrar en los sistemas autónomos puede expandirse para abarcar el sistema distribuido. Dichas funciones incluyen la transferencia de archivos, los inicios de sesión, el correo electrónico y las llamadas a procedimientos remotos (RPC).

La ventaja de un sistema distribuido es que estas funciones pueden llevarse a cabo a través de grandes distancias. Dos personas situadas en nodo geográficamente distantes pueden, por ejemplo, colaborar en un proyecto. Transfiriendo los archivos del proyecto, conectándose a los sistemas remotos de la otra persona para ejecutar programas e intercambiando mensajes de correo electrónico para coordinar el trabajo, los usuarios minimizan las limitaciones inherentes al trabajo a distancia. Los autores de este libro lo hemos escrito colaborando de dicha manera.

Las ventajas de los sistemas distribuidos han hecho que todo el sector se vea sometido a una tendencia hacia la **reducción de escala (downsizing)**. Muchas empresas están sustituyendo sus computadoras de tipo *mainframe* por redes de estaciones de trabajo o de computadoras personales. Las empresas obtienen así una mejor funcionalidad por un mismo coste, más flexibilidad a la hora de ubicar los recursos y expandir las instalaciones, mejores interfaces de usuario y un mantenimiento mucho más sencillo.

16.2 Tipos de sistemas operativos distribuidos

En esta sección, vamos a describir las dos categorías generales de sistemas operativos orientados a red: sistemas operativos de red y sistemas operativos distribuidos. Los sistemas operativos de red son más sencillos de implementar, pero los usuarios encuentran generalmente más dificultades para acceder a ellos y utilizarlos que con los sistemas operativos distribuidos, que proporciona mayor funcionalidad.

16.2.1 Sistemas operativos de red

Un **sistema operativo de red** proporciona un entorno en el que los usuarios, que son conscientes de la multiplicidad de máquinas, pueden acceder a recursos remotos iniciando una sesión en la máquina remota apropiada o transfiriendo datos desde la máquina remota a su propia máquina.

16.2.1.1 Inicio de sesión remoto

Una función importante de un sistema operativo de red es permitir a los usuarios iniciar remotamente una sesión. Internet proporciona el protocolo **telnet** para este propósito. Para ilustrar esta funcionalidad, Supongamos que un usuario en una universidad española quiere realizar un determinado procesamiento en “cs.yale.edu”, una computadora situada en la universidad de Yale. Para poder hacer esto, el usuario debe disponer de una cuenta válida en dicha máquina. Para iniciar la sesión remotamente, el usuario ejecuta el comando

```
telnet cs.yale.edu
```

Este comando hace que se forme una conexión de tipo *socket* entre la máquina local de la universidad española y la computadora “cs.yale.edu”. Después de establecida esta conexión; el soft-

ware de red crea un enlace transparente bidireccional para que todos los caracteres introducidos por el usuario se envíen a un proceso en “cs.yale.edu” y toda la salida de dicho proceso se envíe de vuelta al usuario. El proceso en la máquina remota solicita al usuario que introduzca un nombre de inicio de sesión y una contraseña. Una vez recibida la información correcta, el proceso actúa como un *proxy* para el usuario, que podrá realizar sus cálculos en la máquina remota exactamente igual que cualquier usuario local.

16.2.1.2 Transferencia remota de archivos

Otra función principal de un sistema operativo de red consiste en proporcionar un mecanismo para la **transferencia remota de archivos** de una máquina a otra. En dicho entorno, cada computadora mantiene su propio sistema de archivos local. Si un usuario de un sitio (como por ejemplo “cs.uvm.edu”, que corresponde a la Universidad de Vermont) quiere acceder a un archivo ubicado en otra computadora (como “cs.yale.edu”), entonces el archivo debe copiarse explícitamente desde la computadora situada en Yale a la otra computadora.

Internet proporciona un mecanismo para este tipo de transferencias mediante el protocolo de transferencia de archivos (FTP). Suponga que un usuario en “cs.uvm.edu” quiere copiar un programa Java denominado *Server.java* que reside en “cs.yale.edu”. El usuario debe primero invocar el programa FTP ejecutando

```
ftp cs.yale.edu
```

El programa pide entonces al usuario un nombre de inicio de sesión y una contraseña. Una vez recibida la información correcta, el usuario debe conectarse al subdirectorio en el que resida el archivo *Server.java* y luego copiar el archivo ejecutando

```
get Server.java
```

Según este esquema, la ubicación del archivo no es transparente para el usuario; los usuarios deben conocer exactamente dónde está cada archivo. Además, no existe una verdadera compartición de archivos, porque el usuario sólo puede *copiar* un archivo de un nodo a otro. Por tanto, pueden existir varias copias del mismo archivo, lo que representa un desperdicio de espacio. Además, si estas copias se modifican, las diversas copias serán incoherentes.

Observe que, en nuestro ejemplo, el usuario en la Universidad de Vermont debe tener permiso de inicio de sesión en “cs.yale.edu.”. FTP también proporciona un mecanismo para permitir que un usuario que no disponga de una cuenta en la computadora de Yale copie archivos remotamente. Esta copia remota se lleva a cabo mediante el método de “FTP anónimo”, que funciona de la forma siguiente. El archivo que hay que copiar (es decir, *Server.java*) debe colocarse en un subdirectorio especial (por ejemplo, *ftp*), configurándose la protección de manera que se permita que el público lea el archivo. Un usuario que desee leer el archivo utilizará el comando *ftp* como antes. Cuando se le pregunte al usuario el nombre del inicio de sesión, el usuario deberá introducir el nombre “anonymous” y una contraseña arbitraria.

Una vez realizado el inicio de sesión anónimo, el sistema deberá tomar las medidas adecuadas para garantizar que este usuario parcialmente autorizado no acceda a archivos inapropiados. Generalmente, al usuario sólo se le permite acceder a aquellos archivos que se encuentren en el árbol de directorios del usuario “anonymous”. Todos los archivos que estén ubicados allí serán accesibles por parte de los usuarios anónimos, respetando siempre el esquema usual de protección de archivos que se utilice en esa máquina. Sin embargo, los usuarios anónimos no pueden acceder a archivos situados fuera de este árbol de directorios.

El mecanismo FTP se implementa de forma similar al mecanismo telnet. Existe un demonio en el sitio remoto que detecta las solicitudes de conexión al puerto FTP del sistema. Despues de realizar la autenticación del inicio de sesión, se permite al usuario al usuario ejecutar comandos remotamente. A diferencia del demonio telnet, que ejecuta cualquier comando por cuenta del usuario, el demonio FTP sólo responde a un conjunto predefinido de comandos relacionados con los archivos, entre los que se incluyen:

- *get*: transfiere un archivo desde la máquina remota hasta la máquina local.

- put: transfiere desde la máquina local hasta la máquina remota.
- ls o dir: enumera los archivos situados en el directorio actual en la máquina remota.
- cd: cambia el directorio actual en la máquina remota.

Existen también diversos comandos para cambiar los modos de transferencia (para archivos binarios o ASCII) y para determinar el estado de la conexión.

Un punto importante acerca de telnet y de FTP es que ambos mecanismos requieren que el usuario efectúe un cambio de paradigma. FTP requiere que el usuario conozca un conjunto de comandos enteramente distinto de los comandos normales del sistema operativo. telnet requiere un cambio más pequeño: el usuario debe conocer los comandos apropiados para el sistema remoto. Por ejemplo, un usuario en una máquina Windows que se conecte mediante telnet a una máquina UNIX deberá utilizar comandos UNIX mientras dure la sesión telnet. Las diversas funcionalidades de un sistema resultan más cómodas para los usuarios si no obligan a éstos a utilizar un conjunto diferente de comandos. Los sistemas operativos distribuidos están diseñados para resolver este problema.

16.2.2 Sistemas operativos distribuidos

En un sistema operativo distribuido, los usuarios acceden a los recursos remotos de la misma forma en la que acceden a los recursos locales. La migración de datos y de procesos entre un sitio y otro está bajo el control del sistema operativo distribuido.

16.2.2.1 Migración de datos

Suponga que un usuario en el sitio A quiere acceder a datos (como por ejemplo un archivo) que residen en el sitio B. El sistema puede transferir los datos utilizando uno de dos métodos básicos. Una técnica de **migración de datos** consiste en transferir todo el archivo al sitio A. A partir de ese momento, todo acceso al archivo es local. Cuando el usuario deja de necesitar acceder al archivo, se devuelve una copia del archivo (si ha sido modificado) al sitio B. Incluso si sólo se ha hecho un pequeño cambio en un archivo de gran tamaño, será necesario transferir todos los datos. Este mecanismo puede considerarse una especie de sistema FTP automatizado. Esta técnica se empleaba en el sistema de archivos Andrew, como se explica en el Capítulo 17, pero se vio que era muy poco eficiente.

La otra técnica consiste en transferir al sitio A únicamente aquellas partes del archivo que sean realmente *necesarias* para la tarea inmediata que haya realizar. Si se requiere posteriormente otra parte, se realizará otra transferencia. Cuando el usuario no desee acceder más al archivo, será necesario devolver al sitio B cualquier parte del mismo que haya sido modificada (observe la similitud con el mecanismo de paginación bajo demanda). El protocolo del sistema de archivos en red (NFS) de Sun Microsystems utiliza este método (Capítulo 17), al igual que las versiones más recientes de Andrew. El protocolo SMB de Microsoft (que se ejecuta sobre TCP/IP o sobre el protocolo NetBEUI de Microsoft) también permite compartir archivos a través de una red. SMB se describe en el Apéndice C.6.1.

Claramente, si sólo se está accediendo a una pequeña parte de un archivo de gran tamaño, resulta preferible la segunda de estas técnicas. Sin embargo, si se está accediendo a una parte significativa del archivo, resulta más eficiente copiar el archivo completo. Con ambos métodos, la migración de datos incluye más cosas a parte de la mera transferencia de datos de un sitio a otro. El sistema deberá también realizar diversas traducciones de los datos si los dos sitios implicados no son directamente compatibles (por ejemplo, si utilizan diferentes representaciones de códigos de caracteres o representan los enteros con un número distinto de bits o con un orden diferente de los mismos).

16.2.2.2 Migración de cálculos

En algunas circunstancias, podemos querer transferir los cálculos a través del sistema, en vez de transferir los datos; esta técnica se denomina **migración de los cálculos**. Por ejemplo, considere

un trabajo que necesita acceder a varios archivos de gran tamaño que residen en sitios diferentes, con el fin de obtener un resumen de esos archivos. Sería más eficiente acceder a los archivos en los sitios donde residen y devolver los resultados deseados al sitio que ha iniciado los cálculos. Generalmente, si el tiempo de transferencia de los datos es mayor que el tiempo para ejecutar el comando remoto, debe utilizarse el comando remoto.

Dicho cálculo puede llevarse a cabo de diferentes formas. Suponga que el proceso P quiere acceder a un archivo situado en el nodo A. El acceso al archivo se lleva a cabo en el nodo A y podría iniciarse mediante una llamada RPC. Las llamadas RPC utilizan un **protocolo de datagramas** (UDP en Internet) para ejecutar una rutina en un sistema remoto (Sección 3.6.2). El proceso P invoca un procedimiento predefinido en el sitio A. Ese procedimiento se ejecuta de la forma apropiada y luego devuelve los resultados a P.

Alternativamente, el proceso P puede enviar un *mensaje* al sitio A. El sistema operativo en el sitio A crea entonces un nuevo proceso Q cuya función consiste en llevar a cabo la tarea designada. Cuando el proceso Q completa su ejecución, devuelve el resultado necesario a P a través del sistema de mensajería. Con este esquema, el proceso P puede ejecutarse concurrentemente con el proceso Q y, de hecho, puede tener varios procesos ejecutándose concurrentemente en diversos sitios.

Ambos métodos pueden emplearse para acceder a varios archivos que residan en diversos nodos. Una llamada RPC puede provocar la invocación de otra llamada RPC o incluso la transferencia de mensajes a otro nodo. De forma similar, el proceso Q podría, durante el curso de su ejecución, enviar un mensaje a otro nodo, que a su vez crearía otro proceso. Este proceso puede devolver un mensaje a Q o repetir el ciclo.

16.2.2.3 Migración de procesos

Una extensión lógica de la migración de cálculos es la **migración de procesos**. Cuando se planifica un proceso para su ejecución, no siempre se ejecuta en el sitio en el que se ha iniciado. El proceso completo, o partes del mismo, puede ejecutarse en diferentes sitios. Este esquema puede emplearse por diversas razones:

- **Equilibrado de carga.** Los procesos (o subprocesos) pueden distribuirse por la red con el fin de equilibrar la carga de trabajo.
- **Aceleración de los cálculos.** Si puede dividirse un único proceso en una serie de subprocesos que puedan ejecutarse concurrentemente en diferentes sitios, entonces podrá reducirse el tiempo total de ejecución del proceso.
- **Preferencias hardware.** El proceso puede tener características que lo hagan más adecuado para su ejecución en algún procesador especializado (como por ejemplo una inversión de matrices en un procesador vectorial, en lugar de en un microprocesador).
- **Preferencias software.** El proceso puede requerir software que sólo esté disponible en un nodo concreto y es posible que ese software no pueda ser transferido, o que sea menos costoso transferir el propio proceso.
- **Acceso a datos.** Al igual que en la migración de cálculos si los datos que se están utilizando en los cálculos son numerosos, puede ser más eficiente hacer que un proceso se ejecute remotamente que transferir todos los datos.

Se utilizan dos técnicas complementarias para mover procesos en una red informática. En la primera, el sistema puede tratar de ocultar a ojos del cliente el hecho de que el proceso ha migrado. Este esquema tiene la ventaja de que el usuario no tiene que codificar su programa explícitamente para llevar a cabo la migración. Este método se emplea usualmente para conseguir un equilibrado de carga y una aceleración de los cálculos entre sistemas homogéneos, ya que este tipo de sistemas no necesitan que el usuario proporcione ninguna información de entrada para ayudarlos a ejecutar programas remotamente.

La otra técnica consiste en permitir (o exigir) al usuario que especifique explícitamente cómo debe migrar el proceso. Este método suele emplearse cuando es necesario mover el proceso para satisfacer unas preferencias hardware o software.

Probablemente se haya dado cuenta el lector de que la Web presenta muchos de los aspectos de un entorno informático distribuido. Ciertamente, proporciona mecanismos de migración de datos (entre un servidor web y un cliente web) y también de migración de los cálculos. Por ejemplo, un cliente web puede hacer que se ejecute una operación de base de datos en un servidor web. Por último, gracias a Java, proporciona una forma de migración de procesos. Las *applets* Java se envían desde el servidor al cliente, donde se las ejecuta. Un sistema operativo de red proporciona la mayoría de estas funciones, pero un sistema operativo distribuido hace que esas funciones sean transparentes y fácilmente utilizables. El resultado es un conjunto de funcionalidades potente y fácil de usar, lo que constituye una de las razones para el enorme crecimiento de la World Wide Web.

16.3 Estructura de una red

Existen básicamente dos tipos de redes: **redes de área local** (LAN, local-area network) y **redes de área extensa** (WAN, wide-area network). La principal diferencia entre los dos tipos es la forma en que están distribuidas geográficamente. Las redes de área local están compuestas de procesadores distribuidos en un área pequeña (como por ejemplo un único edificio o un conjunto de edificios adyacentes), mientras que las redes de área extensa están compuestas de una serie de procesadores autónomos distribuidos en una área de gran tamaño (como por ejemplo todo un país). Estas diferencias implican variaciones importantes en la velocidad y la fiabilidad de la red de comunicaciones y están reflejadas en el diseño del propio sistema operativo distribuido.

16.3.1 Redes de área local

Las redes de área local surgieron a principios de la década de 1970 como sustituto de los grandes sistemas informáticos basados en *mainframe*. Para muchas empresas, resulta más económico disponer de una serie de pequeñas computadoras, cada una con sus propias aplicaciones autocontenidoas, que adquirir un único sistema de gran tamaño. Puesto que resulta probable que cada pequeña computadora necesite un conjunto completo de dispositivos periféricos (como por ejemplo discos e impresoras) y como también es probable que tenga lugar algún tipo de compartición de datos en toda organización, resulta un paso natural conectar estos sistemas de pequeño tamaño mediante una red.

Las redes LAN, como ya hemos mencionado, están usualmente diseñadas para cubrir un área geográfica pequeña (como por ejemplo un edificio o un conjunto de edificios adyacentes) y se utilizan generalmente en un entorno de oficina. Todos los nodos en dicho tipo de sistemas están próximos entre sí, por lo que los enlaces de comunicación tienden a tener una mayor velocidad y una menor tasa de errores que sus equivalentes en las redes de área extensa. Se necesitan cables de alta calidad (costosos) para obtener esta alta velocidad y fiabilidad. También es posible utilizar el cable exclusivamente para el tráfico de la red de datos. Para distancias mayores el coste de utilizar cable de alta calidad es enorme y el uso exclusivo del cable tiende a ser prohibitivo.

Los enlaces más comunes en una red de área local son los de par trenzado y de fibra óptica. Las configuraciones más comunes son el bus multiacceso y las redes en anillo y en estrella. Las velocidades de comunicación van desde 1 megabit por segundo, para redes tales como AppleTalk, infrarrojos y la nueva red de radio local Bluetooth, hasta 1 gigabit por segundo para gigabit Ethernet. La velocidad más común es de 10 megabits por segundo y esa es la que se emplea en 10BaseT Ethernet. 100BaseT Ethernet requiere un cable de mayor calidad, pero trabaja a 100 megabits por segundo y está siendo cada vez más común. También está creciendo el uso de redes FDDI basada en fibra óptica. La red RDXI utiliza un mecanismo de paso de testigo y trabaja a más de 100 megabits por segundo.

Una LAN típica puede estar compuesta por varias computadoras de diferentes tipos (desde computadoras de tipo *mainframe* a portátiles o dispositivos PDA), diversos dispositivos periféricos

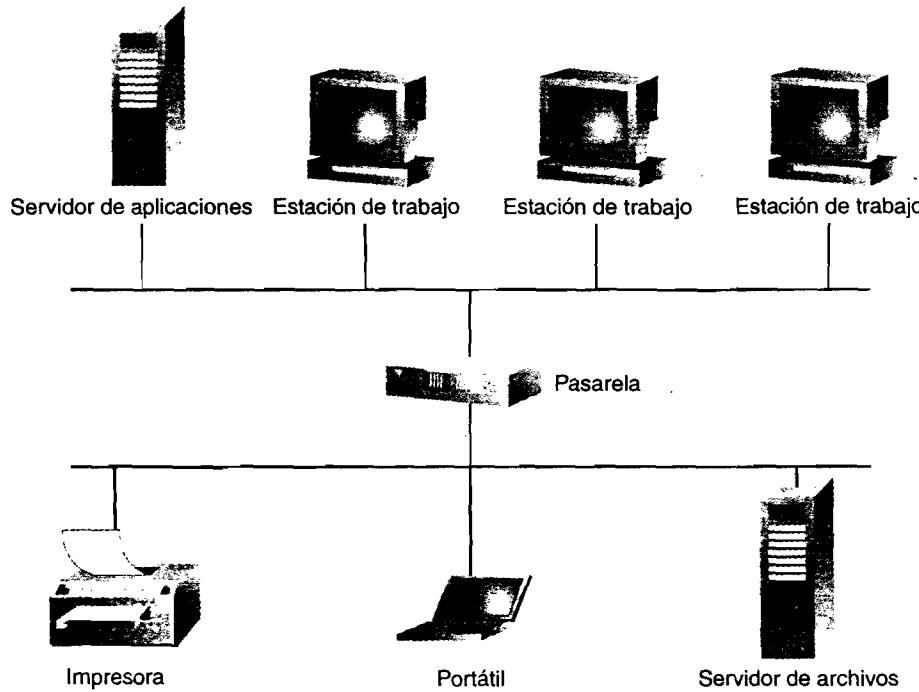


Figura 16.2 Red de área local.

compartidos (como impresoras láser y unidades de cinta magnética) y una o más pasarelas (procesadores especializados) que proporcionan acceso a otras redes (Figura 16.2). Para construir redes LAN se utiliza comúnmente un esquema Ethernet. Una red Ethernet no tiene ningún controlador central, porque se trata de un bus multiacceso, por lo que pueden añadirse fácilmente nuevos *hosts* a la red. El protocolo Ethernet está definido por el estándar IEEE 802.3.

16.3.2 Redes de área extensa

Las redes de área extensa surgieron a finales de la década de 1960, fundamentalmente como un proyecto académico de investigación destinado a proporcionar un mecanismo eficiente de comunicación entre sitios, permitiendo que una amplia comunidad de usuarios compartiera el hardware y el software de forma cómoda y económica. La primera red WAN que se diseñó y desarrolló fue *Arpanet*. Iniciada en 1968, Arpanet ha crecido enormemente, pasando de ser una red experimental compuesta de cuatro nodos a una red mundial de redes, Internet, compuesta por millones de sistemas informáticos.

Puesto que los nodos de una WAN están físicamente distribuidos en un área geográfica de gran tamaño, los enlaces de comunicaciones son, de forma predeterminada, relativamente lentos y poco fiables. Los enlaces más típicos son las líneas telefónicas, las líneas arrendadas (líneas de datos dedicadas), los enlaces de microondas y los canales de comunicación vía satélite. Estos enlaces de comunicaciones están controlados por procesadores de comunicaciones especiales (Figura 16.3), que se encargan de definir la interfaz mediante la cual se comunicarán los diversos sitios a través de la red, así como de transferir la información entre los diversos sitios.

Por ejemplo, la red WAN Internet permite que una serie de *hosts* geográficamente separados se comuniquen entre sí. Las computadoras *host* suelen diferir entre sí en lo que respecta a su tipo, velocidad, longitud de palabra, sistema operativo, etc. Los *hosts* se encuentran normalmente situados en redes LAN que están conectadas a su vez a Internet a través de redes regionales o nacionales. Las redes regionales, como NSFnet en el noreste de los EE. UU., están entrelazadas mediante **encaminadores (routers)** (Sección 16.5.2) para formar una única red mundial. Las conexiones entre las redes utilizan frecuentemente un servicio del sistema telefónico denominado T1, que proporciona una tasa de transferencia de 1,544 megabits por segundo a través de una línea arren-

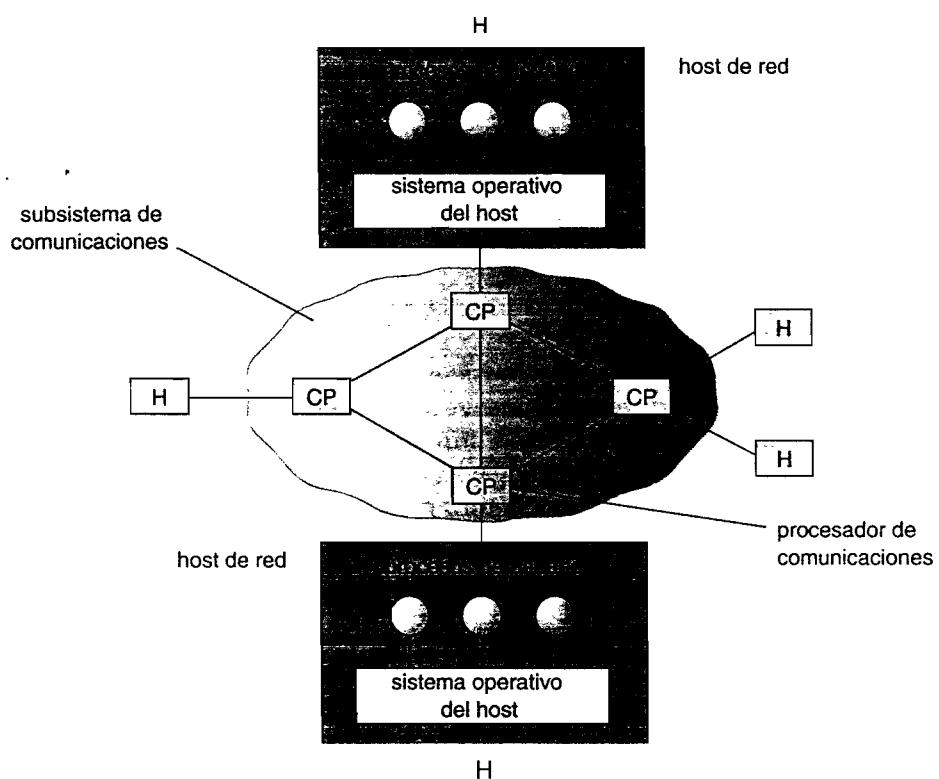


Figura 16.3 Procesadores de comunicaciones en una red de área extensa.

dada. Para aquellos sitios que requieran un acceso más rápido a Internet, las líneas T1 se agrupan en unidades que trabajan en paralelo para proporcionar una mayor tasa de transferencia. Por ejemplo, una línea T3 está compuesta de 28 conexiones T1 y tiene una tasa de transferencia de 45 megabits por segundo. Los encaminadores controlan la ruta que cada mensaje sigue a través de la red. Este proceso de encaminamiento puede ser dinámico, para incrementar la eficiencia de las comunicaciones, o estático, para reducir los riesgos de seguridad o para permitir que se calculen los costes de las comunicaciones.

Otras redes WAN utilizan líneas telefónicas estándar como medio principal de comunicación. Un **módem** es un dispositivo que acepta datos digitales desde el lado de conexión con la computadora y los convierte a las señales analógicas empleadas en el sistema telefónico. Otro módem situado en el nodo de destino convierte de nuevo la señal analógica a forma digital, con lo que el destino puede recibir los datos. La red de noticias UNIX, UUCP permite que en los sistemas se comuniquen unos con otros en momentos predeterminados, a través de módem, para intercambiar mensajes. Los mensajes se encaminan a continuación a otros sistemas cercanos y, de esta forma, o bien se propagan a todos los *hosts* de la red (mensajes públicos) o se terminan transfiriendo a su destino (mensajes privados). Las redes WAN son, generalmente, más lentas que las redes LAN; sus velocidades de transmisión van de 1.200 bits por segundo a más de 1 megabit por segundo. UUCP se ha visto sustituido por PPP, el protocolo punto a punto. PPP funciona sobre conexiones por módem, permitiendo que las computadoras domésticas se conecten a Internet.

16.4 Topología de red

Los nodos de un sistema distribuido pueden conectarse físicamente de diversas maneras y cada configuración tiene sus ventajas y desventajas. Podemos comparar las distintas configuraciones usando los siguientes criterios:

- **Coste de instalación.** El coste de enlazar físicamente los nodos que forman el sistema.

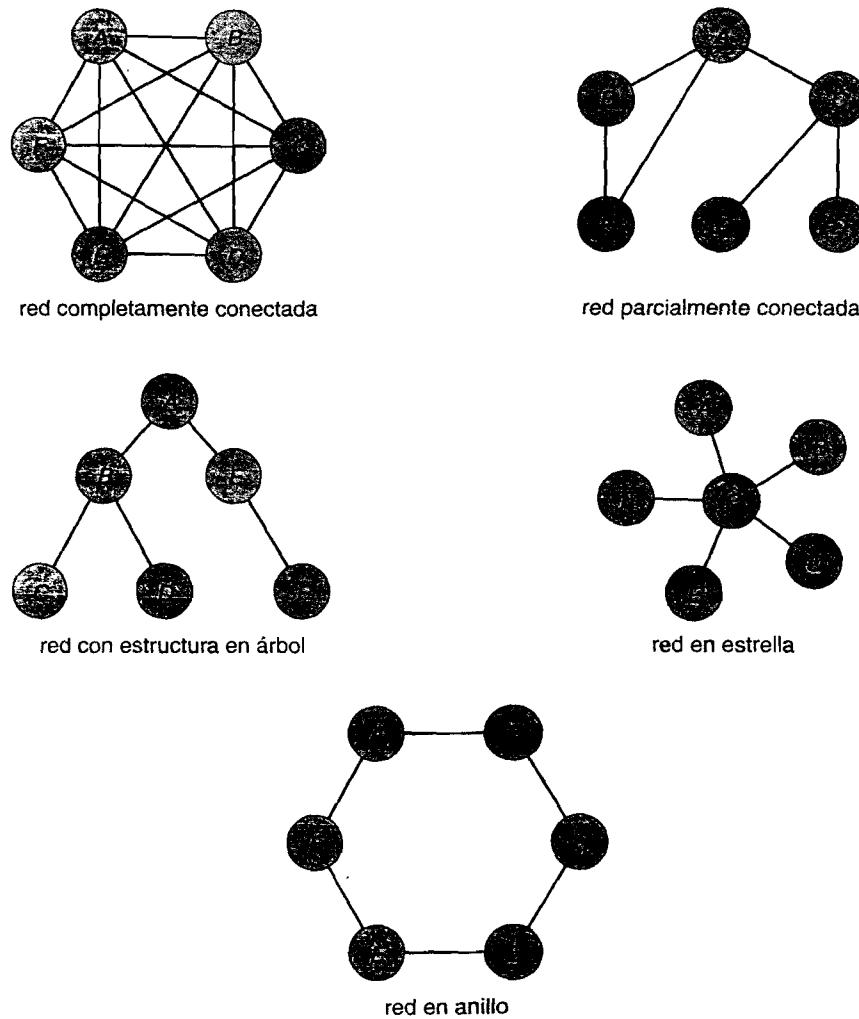


Figura 16.4 Topología de red.

- **Coste de comunicaciones.** El coste en tiempo y en dinero para enviar un mensaje desde el nodo A al nodo B.
- **Disponibilidad.** El grado hasta el que puede accederse a los datos a pesar del fallo de algunos enlaces o nodos.

Las diversas topologías se muestran en la Figura 16.4 en forma de grafos cuyos nodos se corresponden con los sitios del sistema. Una arista desde el nodo A al nodo B se corresponde con un enlace de comunicaciones directo entre los dos sitios. En una red completamente conectada, cada sitio está conectado directamente con todos los demás sitios. Sin embargo, el número de enlaces crece según el cuadrado del número de sitios, lo que representa un coste de instalación enorme. Como consecuencia, las redes completamente conectadas no resultan prácticas en ningún sistema de gran tamaño.

En una **red parcialmente conectada**, existen enlaces directos entre algunas parejas de sitios, pero no entre todas. De este modo, el coste de instalación de una configuración como ésta es menor que para una red completamente conectada. Sin embargo, si dos sitios A y B no están conectados directamente, los mensajes del uno al otro deben **encaminarse** a través de una secuencia de enlaces de comunicaciones. Este requisito hace que el coste de comunicaciones sea más alto.

Si falla un enlace de comunicaciones, los mensajes que habrían sido transmitidos a través de ese enlace deberán ser reencaminados. En algunos casos, puede encontrarse otra ruta a través de la red, de modo que los mensajes serán capaces de alcanzar su destino. En otros casos, un fallo

puede implicar que deje de existir una conexión entre algunas parejas de sitios. Cuando un sistema queda dividido en dos (o más) subsistemas que carecen de conexiones entre ellos, se dice que está particionado. Usando esta definición, un subsistema (o partición) puede estar compuesto por un único nodo.

Entre los diversos tipos de redes parcialmente conectadas se incluyen las redes con estructura de árbol, las redes en anillo y las redes en estrella, como se muestra en la Figura 16.4. Cada uno de estos tipos de redes tiene diferentes características de fallo y diferentes costes de instalación de y de comunicaciones. Los costes de instalación y de comunicaciones son relativamente bajos para las redes con estructura de árbol; sin embargo, el fallo de un único enlace en una red de este tipo puede provocar que la red quede particionada. En una red en anillo, deben fallar al menos dos enlaces para que se produzca una partición, por lo que la red en anillo tiene un grado más alto de disponibilidad que una red con estructura de árbol; sin embargo, el coste de comunicaciones es alto, ya que un mensaje puede tener que atravesar un gran número de enlaces. En una red en estrella, el fallo de un único enlace provoca una partición de la red, pero una de las particiones está compuesta por un único sitio; dicha partición puede tratarse como si fuera un fallo de un único nodo. La red en estrella tiene también un bajo coste de comunicaciones, ya que cada sitio está a una distancia de como mucho dos enlaces de cualquier otro sitio. Sin embargo, si falla el sitio central, todos los sitios del sistema quedan desconectados.

16.5 Estructura de comunicaciones

Ahora que hemos explorado los aspectos físicos de las redes, vamos a volver nuestra atención a su funcionamiento interno. El diseñador de una red de comunicaciones debe contemplar cinco cuestiones básicas:

- **Nombrado y resolución de nombres.** ¿Cómo se localizan mutuamente dos procesos para poder comunicarse?
- **Estrategias de encaminamiento.** ¿Cómo se envían los mensajes a través de la red?
- **Estrategias de paquetes.** ¿Los paquetes se envían individualmente o como una secuencia?
- **Estrategias de conexión.** ¿Cómo envían dos procesos una secuencia de mensajes?
- **Contienda.** ¿Cómo resolvemos las demandas conflictivas de uso de la red, dado que se trata de un recurso compartido?

En las siguientes secciones, vamos a analizar cada una de estas cuestiones.

16.5.1 Nombrado y resolución de nombres

El primer componente de una red de comunicaciones es el mecanismo de nombrado de los sistemas de la red. Para que un proceso en el sitio A intercambie información en el sitio B, cada uno de ellos debe ser capaz de especificar al otro. Dentro de un sistema informático autónomo, cada proceso tiene un identificador de proceso y la dirección de los mensajes puede especificarse utilizando dicho identificador. Pero, puesto que los sistemas en red no comparten memoria, un *host* del sistema no tiene ningún conocimiento inicial acerca de los procesos existentes en otros *hosts*.

Para resolver este problema, los procesos de los sistemas remotos se identifican generalmente mediante la pareja <nombre de host, identificador>, donde *nombre de host* es un nombre único dentro de la red e *identificador* puede ser un identificador de proceso y otro número único dentro de ese *host*. Un *nombre de host* es usualmente un identificador alfanumérico, en lugar de un número, para que a los usuarios les resulte más fácil especificarlo. Por ejemplo, el sitio A puede tener una serie de máquinas denominadas *homer*, *marge*, *bart* y *lisa*. Obviamente, *Bart* es mucho más fácil de recordar que 12814831100.

Los nombres resultan cómodos para que los seres humanos los utilicen, pero los computadoras prefieren los números por motivos de velocidad y simplicidad. Por esta razón, debe existir un mecanismo para **resolver** el nombre de *host* en un identificador de *host* que describa el sistema de

destino al hardware de interconexión por red. Este mecanismo de resolución es similar al acoplamiento nombre-dirección que tiene lugar durante la compilación, montaje, carga y ejecución de los programas (Capítulo 8). En el caso de los nombres de *host* existen dos posibilidades. En primer lugar, cada *host* puede tener un archivo de datos que contenga los nombres y direcciones de todos los demás *hosts* alcanzables a través de la red (esto es similar al acoplamiento en tiempo de compilación); el problema con este modelo es que añadir o eliminar un *host* de la red requiere actualizar los archivos de datos en todas las máquinas. La alternativa es distribuir la información entre una serie de sistemas de la red. La red deberá entonces utilizar un protocolo para distribuir y extraer la información; este esquema es similar al acoplamiento en tiempo de ejecución. El primer método era el que se usaba originalmente en Internet; sin embargo, a medida que Internet fue creciendo, dejó de ser práctico, por lo que ahora se utiliza el segundo método, el **sistema de nombres de dominio** (DNS, domain-name system).

DNS especifica la estructura de nombrado de los *hosts*, así como la resolución de nombres en direcciones. El direccionamiento lógico de los *hosts* de Internet se lleva a cabo mediante un nombre compuesto por múltiples partes. Los nombres están estructurados comenzando por la parte más específica de la dirección y continuando hasta la parte más general, estando separados por puntos los distintos campos. Por ejemplo, *bob.cs.brown.edu* hace referencia al *host bob* en el Departamento de Informática (cs) de la Universidad de Brown dentro del dominio *edu*. (Otros dominios de nivel superior incluyen *com* para los sitios comerciales y *org* para las organizaciones no comerciales, así como un dominio para cada país conectado a la red, para aquellos sistemas que se especifiquen según el país y no según el tipo de organización.) Generalmente, el sistema resuelve las direcciones examinando los componentes del nombre de *host* en orden inverso. Cada componente tiene un **servidor de nombres** (que es simplemente un proceso en un sistema) que acepta un nombre y devuelve la dirección del servidor de nombres responsable de dicho nombre. En el paso final, se contactará con el servidor de nombres correspondiente al *host* en cuestión y se obtendrá como respuesta un identificador de *host*. Para nuestro sistema de ejemplo, *bob.cs.brown.edu*, se llevarían a cabo los siguientes pasos como resultado de una solicitud realizada por un proceso en el sistema A para comunicarse con *bob.cs.brown.edu*:

1. El *kernel* del sistema A envía una solicitud al servidor de nombres correspondiente al dominio *edu*, pidiendo la dirección del servidor de nombres *brown.edu*. El servidor de nombres del dominio *edu* debe estar en una dirección conocida para poder consultarle.
2. El servidor de nombres para *edu* devuelve la dirección del *host* donde reside el servidor de nombres para *brown.edu*.
3. El *kernel* del sistema A consulta entonces al servidor de nombres ubicado en esta dirección y le pregunta acerca de *cs.brown.edu*.
4. Se devuelve una dirección y una solicitud a dicha dirección para conocer el identificador de *bob.cs.brown.edu* devuelve, finalmente, un identificador en forma de dirección Internet para dicho *host* (por ejemplo, 128.148.31.100)

Este protocolo puede parecer poco eficiente, pero usualmente se mantienen cachés locales en cada servidor de nombres con el fin de acelerar el proceso. Por ejemplo, el servidor de nombres *edu* podría tener *brown.edu* en su caché e informaría al sistema A de que puede resolver dos partes del nombre, devolviendo un puntero al servidor de nombres *cs.brown.edu*. Por supuesto, el contenido de estas cachés debe refrescarse a lo largo del tiempo, por si acaso se traslada el servidor de nombres o cambia su dirección. De hecho, este servicio es tan importante que se han implementado numerosas optimizaciones en el protocolo, así como diversas salvaguardas. Considere los que sucedería si el servidor de nombres principal para el dominio *edu* fallara: ¿Podría pasar que no pudiera resolverse la dirección de ningún *host* del dominio *edu*, lo que haría que todos ellos dejaran de ser alcanzables? La solución consiste en utilizar servidores de nombres secundarios de reserva que dupliquen el contenido de los servidores principales.

Antes de que se introdujera el servicio de nombres de dominio, todos los *hosts* de Internet necesitaban tener una copia de un archivo que contuviera los nombres y direcciones de todos los *hosts* de la red. Todos los cambios en este archivo tenían que registrarse en un sitio centralizado

(*hosts* SRI-NIC) y todos los *hosts* tenían que copiar periódicamente el archivo actualizado desde SRI-NIC para poder contactar a los nuevos sistemas o localizar *hosts* cuyas direcciones hubieran cambiado. Con el servicio de nombres de dominio, cada nodo servidor de nombres es responsable de actualizar la información correspondiente a los *hosts* de dicho dominio. Por ejemplo, todos los cambios de *host* en la Universidad de Brown son responsabilidad del servidor de nombres correspondiente al *brown.edu* y no hay porqué informar de esos cambios a nadie más. Las búsquedas DNS extraerán automáticamente la información actualizada, ya que se contacta directamente con *brown.edu*. Dentro de cada dominio puede haber subdominios autónomos, con el fin de distribuir aún más la responsabilidad relativa a los cambios de los nombres y los identificadores de los *hosts*.

Java proporciona la API necesaria para diseñar un programa que aplique los nombres IP a direcciones IP. Al programa mostrado en la Figura 16.5 se le pasa un nombre IP (como por ejemplo, "bob.cs.brown.edu") en la línea de comandos y el programa muestra la dirección IP del *host* o devuelve un mensaje indicando que no se pudo resolver el nombre de *host*. *InetAddress* es una clase Java que representa un nombre o dirección IP. Al método estático *getByName()* perteneciente a la clase *InetAddress* se le pasa la representación de un nombre IP en forma de cadena de caracteres y el método devuelve la correspondiente dirección *InetAddress*. El programa invoca entonces el método *getHostAddress()*, que utiliza DNS internamente para buscar la dirección IP del *host* designado.

Generalmente, el sistema operativo es responsable de aceptar de sus procesos un mensaje destinado a <nombre de host, identificador> y transferir dicho mensaje al *host* apropiado. El *kernel* del *host* de destino será entonces responsable de transferir el mensaje al proceso indicado por el identificador. Este intercambio no es en absoluto trivial y se describe en la Sección 16.5.4.

16.5.2 Estrategias de encaminamiento

Cuando un proceso en el sitio A quiere comunicarse con otro proceso en el sitio B, ¿cómo se envía el mensaje? Si sólo hay una ruta física desde A a B (como por ejemplo en una red en estrella o con estructura de árbol), el mensaje debe enviarse a través de esa ruta. Sin embargo, si hay múltiples rutas físicas desde A a B, existirán diversas opciones de encaminamiento. Cada sitio tiene una **tabla de encaminamiento** que indica las rutas alternativas que pueden usarse para enviar un mensaje a otros sitios. La tabla puede incluir información acerca de la velocidad y el coste de las diversas rutas de comunicaciones y puede actualizarse según sea necesario, bien manualmente o mediante programas que intercambien información de encaminamiento. Los tres esquemas más comunes de encaminamiento son el **encaminamiento fijo**, **encaminamiento virtual** y el **encaminamiento dinámico**.

```

/*
 * Utilización: java DNSLookUp <nombre IP>
 * es decir, java DNSLookUp www.wiley.com
 */

public class DNSLookUp {
    public static void main(String[] args) {
        InetAddress hostAddress;

        try {
            hostAddress = InetAddress.getByName(args[0]);
            System.out.println(hostAddress.getHostAddress());
        }
        catch (UnknownHostException uhe) {
            System.err.println("Host desconocido: " + args[0]);
        }
    }
}

```

Figura 16.5 Programa Java que ilustra una búsqueda DNS.

- **Encaminamiento fijo.** Se especifica de antemano una ruta desde A hasta B y esa ruta no cambia a menos que un fallo hardware la inutilice. Usualmente, se selecciona la ruta más corta para minimizar los costes de comunicaciones.
- **Encaminamiento virtual.** Se fija una ruta desde A hasta B mientras dura una sesión. Las diferentes sesiones en las que se envíen mensajes desde A hasta B pueden utilizar rutas distintas. Una sesión puede ser corta, como por ejemplo en la transferencia de un archivo, o larga, abarcando por ejemplo todo el período requerido para un inicio remoto de sesión.
- **Encaminamiento dinámico.** La ruta utilizada para enviar un mensaje desde el sitio A hasta el sitio B se selecciona en el momento de seleccionar el mensaje. Puesto que la decisión se toma dinámicamente, a cada uno de los mensajes se le puede asignar una ruta distinta. El sitio A tomará una decisión para enviar el mensaje al sitio C; el sitio C, a su vez, decidirá enviarlo al sitio D, y así sucesivamente. Eventualmente, habrá un sitio que entregue el mensaje a B. Usualmente, cada sitio envía el mensaje a otro sitio utilizando el enlace que esté siendo menos usado en ese instante concreto.

Existe una serie de compromisos inherentes a estos tres esquemas. El encaminamiento fijo no puede adaptarse a los fallos de los enlaces ni a los cambios en la carga de tráfico. En otras palabras, si se ha establecido una ruta entre A y B, los mensajes deben enviarse a lo largo de esa ruta, incluso aunque la ruta se haya visto interrumpida o esté siendo utilizada más intensamente que alguna de las otras rutas posibles. Podemos solventar parcialmente este problema utilizando el encaminamiento virtual y evitarlo completamente empleando el encaminamiento dinámico. El encaminamiento fijo y el virtual garantizan que los mensajes enviados desde A hasta B se entreguen en el mismo orden en el que fueron enviados. Con el encaminamiento dinámico, los mensajes pueden llegar desordenados. Podemos resolver este problema añadiendo un número de secuencia a cada mensaje.

El encaminamiento dinámico es el más complicado de configurar y operar; sin embargo, es la mejor forma de gestionar el encaminamiento en entornos complicados. UNIX proporciona tanto un encaminamiento fijo, para utilizarlo en los *hosts* pertenecientes a redes simples, como un encaminamiento dinámico, para los entornos de red complejos; también se pueden mezclar los dos esquemas. Dentro de un sitio, los *hosts* pueden simplemente necesitar conocer cómo alcanzar al sistema que conecta la red local con otras redes (por ejemplo, redes de ámbito corporativo o Internet). Dicho tipo de nodo se conoce con el nombre **pasarela**. Cada *host* individual dispone de una ruta estática hasta la pasarela, aunque la propia pasarela utiliza encaminamiento dinámico para poder alcanzar a cualquier otro *host* del resto de la red.

Los encaminadores son las entidades, dentro de la red informática, responsables de encaminar los mensajes. Un encaminador puede ser una computadora *host* con un software de encaminamiento o un dispositivo de propósito especial; en cualquiera de los dos casos, el encaminador debe tener al menos dos conexiones de red, ya que de otro modo no tendría a donde encaminar los mensajes. El encaminador decide si tiene que pasar cada mensaje concreto desde la red por la que lo ha recibido a cualquier otra red conectada al encaminador. El dispositivo toma esta decisión examinando la dirección Internet de destino del mensaje. El encaminador comprueba sus tablas para determinar la ubicación del *host* de destino, o al menos la ubicación de la red a la que enviará el mensaje para que éste progrese hacia el *host* de destino. En el caso del encaminamiento estático, esta tabla sólo se modifica mediante actualizaciones manuales (se carga un nuevo archivo en el encaminador). Con el encaminamiento dinámico, se utiliza un **protocolo de encaminamiento** entre los encaminadores para informar a éstos de los cambios experimentados por la red y para permitirles actualizar sus tablas de encaminamiento automáticamente. Las pasarelas y encaminadores son, normalmente, dispositivos hardware dedicados que ejecutan un código grabado en firmware.

16.5.3 Estrategias de paquetes

Los mensajes son, generalmente, de longitud variable. Para simplificar el diseño del sistema, normalmente implementamos las comunicaciones mediante mensajes de longitud fija denominados

paquetes, tramas, o datagramas. Todo mensaje implementado en un único paquete puede enviar-
se a su destino como un **mensaje sin conexión**. Los mensajes sin conexión pueden ser **no fiables**,
en cuyo caso el emisor no tiene ninguna garantía de que el paquete alcance su destino, ni tam-
po-
co puede determinar si lo ha hecho. Alternativamente, el paquete puede ser **fiable**, usualmente,
en este caso, se devuelve un paquete desde el destino para indicar que el paquete original ha lle-
gado (por supuesto, el paquete de respuesta podría perderse por el camino). Si un mensaje es
demasiado largo para caber en un solo paquete, o si es necesario intercambiar paquetes entre los
dos interlocutores, se establece una conexión para permitir el intercambio fiable de múltiples
paquetes.

16.5.4 Estrategias de conexión

Una vez que los mensajes son capaces de alcanzar su destino, los procesos pueden establecer
sesiones de comunicaciones para intercambiar información. Las parejas de procesos que quieran
comunicarse a través de la red pueden conectarse de diversas maneras. Los tres esquemas más
comunes son la **conmutación de circuitos**, la **conmutación de mensajes** y la **conmutación de
paquetes**.

- **Conmutación de circuitos.** Si dos procesos quieren comunicarse, se establece entre ellos
un enlace físico permanente. Este enlace permanecerá asignado mientras dure la sesión de
comunicaciones y ningún otro proceso podrá usar ese enlace durante este período (incluso
aunque los dos procesos no se estén comunicando activamente durante un tiempo). Este
esquema similar al que se usa en el sistema telefónico: una vez que se ha abierto una línea
de comunicaciones entre dos interlocutores (es decir, el interlocutor A llama al interlocutor B),
nadie más puede utilizar este circuito hasta que se termine la comunicación explícita-
mente (por ejemplo, cuando los interlocutores cuelgan).
- **Conmutación de mensajes.** Si dos procesos quieren comunicarse, se establece un enlace
temporal mientras dure la transferencia de un mensaje. Los enlaces físicos se asignan diná-
micamente entre las partes según sea necesario y sólo se asignan durante breves períodos
de tiempo. Cada mensaje es un bloque de datos con información del sistema (como por
ejemplo, el origen, el destino y los códigos ECC de corrección de errores) que permite a la
red de comunicaciones entregar el mensaje a su destino correctamente. Este esquema es
similar al sistema de correo tradicional: cada carta es un mensaje que contiene tanto la direc-
ción de destino como la de origen (remitente). Pueden enviarse múltiples mensajes (de dife-
rentes usuarios) a través del mismo enlace.
- **Conmutación de paquetes.** Cada mensaje lógico puede ser dividido en una serie de
paquetes. Cada paquete puede ser enviado a su destino por separado y debe, por tanto,
incluir una dirección de origen y de destino junto con los datos. Además, los diversos
paquetes pueden seguir rutas distintas a través de la red. Los paquetes deben recomponer-
se en mensajes según van llegando. Observe que no pasa nada por dividir los datos en
paquetes, encaminarlos por separado y recomponerlos en el destino. Sin embargo, descom-
poner de esta forma una señal de audio (por ejemplo una comunicación telefónica), podría
provocar una gran confusión si no se hiciera con cuidado.

Existen varios compromisos obvios entre estos diferentes esquemas. La conmutación de circui-
tos requiere un tiempo de establecimiento considerable y puede hacer que se malgaste ancho de
banda de la red, pero el coste asociado al envío de cada mensaje es menor. Por otro lado, los meca-
nismos de conmutación de mensajes y de paquetes requieren menos tiempo de establecimiento,
aunque el coste de la información administrativa asociada con cada mensaje es mayor. Asimismo,
en la conmutación de paquetes, cada mensaje debe dividirse en paquetes y luego recomponerse.
La conmutación de paquetes es el método más comúnmente utilizado en las redes de datos, por-
que es el que hace un mejor uso del ancho de banda de la red.

16.5.5 Contienda

Dependiendo de la topología de la red, un enlace puede conectar más de dos sitios de la red informática y es posible que varios de estos sitios quieran transmitir información a través de ese enlace simultáneamente. Esta situación se produce principalmente en las redes en anillo y en las que tienen estructura de bus multiacceso. En este caso, la información transmitida puede corromperse. Si lo hace, deberá ser descartada y será necesario notificar a los sitios acerca de este problema, para que puedan retransmitir la información. Si no se adoptan medidas especiales, la situación podría repetirse, dando como resultado una degradación en el rendimiento. Para evitar las colisiones repetidas, se han desarrollado diversas técnicas, incluyendo la detección de colisiones y el paso de testigo.

- **CSMA/CD.** Antes de transmitir un mensaje a través de un enlace, el nodo debe ponerse a la escucha para determinar si se está transmitiendo actualmente otro mensaje a través de ese enlace; esta técnica se denomina **detección de portadora con acceso múltiple** (CSMA, carrier sense with multiple access). Si el enlace está libre, el nodo puede comenzar a transmitir. En caso contrario, deberá esperar (y continuar escuchando) hasta que el enlace esté libre. Si dos o más sitios comienzan a transmitir exactamente al mismo tiempo (porque los dos creen que ningún otro sitio está usando el enlace), entonces registrarán una **detección de colisiones** (CD, collision detection) y dejarán de transmitir. Cada uno de los sitios volverá a intentarlo de nuevo después de un cierto intervalo de tiempo aleatorio. El problema principal de este enfoque es que, cuando el sistema está muy cargado, pueden producirse muchas colisiones, degradándose así el rendimiento. De todos modos, CSMA/CD ha sido utilizado con éxito en los sistemas Ethernet, que son los sistemas más comunes para redes de área local. Una estrategia para limitar el número de colisiones consiste en limitar el número de *hosts* por cada red Ethernet. Añadir más *hosts* a una red congestionada podría dar como resultado una disminución en la tasa de transferencia a través de la red. A medida que los sistemas van siendo más rápidos, son capaces de enviar más paquetes por cada segmento de tiempo. Como resultado, el número de sistemas por cada red Ethernet está en general decreciendo, para mantener en un nivel razonable las prestaciones de la red.
- **Paso de testigo.** Hay un tipo de mensaje especial, denominado **testigo**, que circula continuamente en el sistema (usualmente una estructura en anillo). El sitio que deseé transmitir información deberá esperar hasta que le llegue el testigo. Entonces, ese nodo elimina el testigo del anillo y comienza a transmitir sus mensajes. Cuando el nodo completa su turno de paso de mensajes, vuelve a transmitir el testigo. Esta acción, a su vez, permite que otro sitio reciba y elimine el testigo y comience a transmitir sus mensajes. Si el testigo se pierde, el sistema debe detectar esa perdida y volver a generar un testigo. Usualmente, el sistema hace esto convocando una **elección** para seleccionar un determinado sitio en el que se generará un nuevo testigo. Posteriormente, en la Sección 18.6, presentaremos un algoritmo de elección. El esquema de paso de testigo es el que se ha adoptado en los sistemas IBM y HP/Apollo. La ventaja de una red de paso de testigo es que las prestaciones son constantes. Añadir nuevos sitios a la red puede alargar el tiempo de espera necesario para recibir el testigo, pero no provoca una gran reducción de las prestaciones, como puede suceder en Ethernet. Sin embargo, en las redes con una baja carga de tráfico, Ethernet es más eficiente, porque los sistemas pueden enviar mensajes en cualquier momento.

16.6 Protocolos de comunicaciones

Cuando estamos diseñando una red de comunicaciones, debemos enfrentarnos a la complejidad inherente a la coordinación de una serie de operaciones asíncronas que se comunican dentro de un entorno potencialmente lento y sujeto a errores. Además, los sistemas de la red deben acordar un protocolo o un conjunto de protocolos para determinar los nombres de *host*, para localizar *hosts* en la red, para establecer conexiones, etc. Podemos simplificar el problema de diseño (y la correspondiente implementación) repartiendo el problema en múltiples niveles. Cada nivel de un siste-

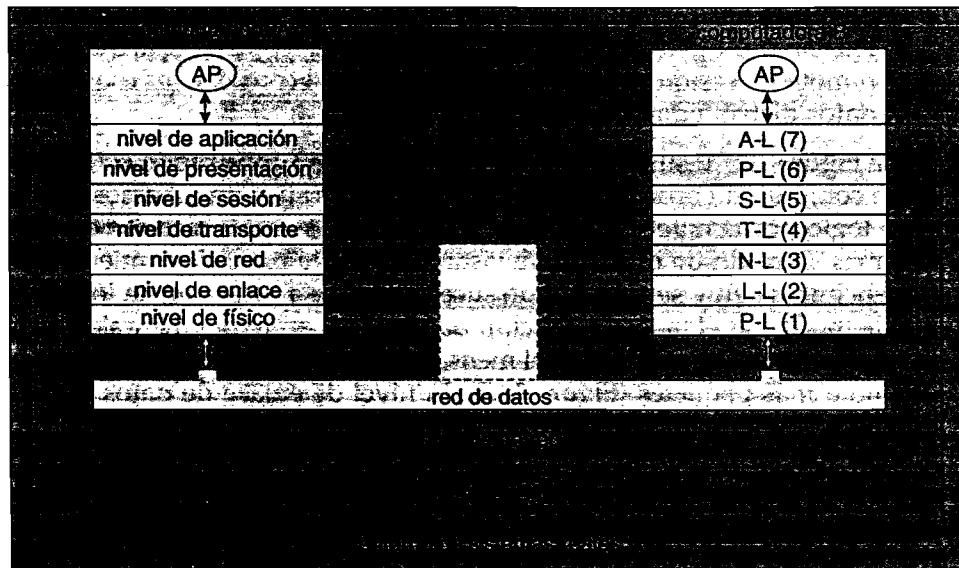


Figura 16.6 Dos computadoras comunicándose según el modelo de red ISO.

ma se comunica con el nivel equivalente en otros sistemas. Típicamente, cada nivel tiene sus propios protocolos y la comunicación tiene lugar entre niveles correspondientes utilizando un protocolo específico. Los protocolos pueden implementarse en hardware o en software. Por ejemplo, la Figura 16.6 muestra las comunicaciones lógicas entre dos computadoras, estando los tres niveles inferiores implementados en hardware. De acuerdo con las recomendaciones de ISO (International Standards Organization), denominamos a los distintos niveles de la forma siguiente:

- Nivel físico.** El nivel físico es responsable de gestionar los detalles tanto eléctricos como mecánicos de la transmisión física de un flujo de bits. En el nivel físico, los sistemas que se estén comunicando deben acordar la representación eléctrica del 0 y 1 binarios, de modo que cuando se envíen los datos como un flujo de señales eléctricas, el receptor sea capaz de interpretar los datos adecuadamente como datos binarios. Este nivel se implementa en el hardware del dispositivo de interconexión por red.
- Nivel de enlace de datos.** El nivel de enlace de datos es responsable de gestionar las *tramas*, o partes de paquetes de longitud fija, incluyendo los mecanismos de detección y recuperación de errores que hayan podido producirse en el nivel físico.
- Nivel de red.** El nivel de red es responsable de proporcionar conexiones y de encaminar los paquetes dentro de la red de comunicaciones, incluyendo la gestión de las direcciones de los paquetes salientes, la decodificación de las direcciones de los paquetes entrantes y el mantenimiento de información de encaminamiento para responder adecuadamente a los cambios en los niveles de carga. Los encaminadores trabajan a este nivel.
- Nivel de transporte.** El nivel de transporte es responsable del acceso a bajo nivel a la red y de la transferencia de mensajes entre clientes, incluyendo el particionamiento de los mensajes en paquetes, el mantenimiento del orden de los paquetes, el control de flujo y la generación de direcciones físicas.
- Nivel de sesión.** El nivel de sesión es responsable de implementar las sesiones, o protocolos de comunicación interprocesos. Típicamente, estos protocolos son las propias comunicaciones requeridas para los inicios de sesión remotos y para las transferencias de archivos y de correo electrónico.
- Nivel de presentación.** El nivel de presentación es responsable de resolver las diferencias de formato entre los distintos sitios de la red, incluyendo las conversiones de caracteres y los modos dúplex-semidúplex (eco de caracteres).

7. Nivel de aplicación. El nivel de aplicación es responsable de interactuar directamente con los usuarios. Este nivel se encarga de la transferencia de archivos, de los protocolos de inicio remoto de sesión y del correo electrónico, así como de los esquemas utilizados para bases de datos distribuidas.

La Figura 16.7 resume la pila de protocolos ISO, mostrando el flujo físico de los datos (una pila es un conjunto de protocolos que cooperan entre sí). Como hemos comentado, desde el punto de vista lógico, cada nivel de una pila de protocolos se comunica con el nivel equivalente en los otros sistemas. Pero desde el punto de vista físico, cada mensaje comienza en el nivel de aplicación (o por encima de él) y se va pasando sucesivamente a cada uno de los niveles inferiores. Cada nivel puede modificar el mensaje e incluir datos de cabecera del mensaje destinados al nivel equivalente en el lado receptor. Al final, el mensaje alcanza el nivel de red de datos y se transfiere en forma de uno o más paquetes (Figura 16.8). El nivel de enlace de datos del sistema de destino recibirá estos datos y el mensaje pasará hacia arriba a través de la pila de protocolos, siendo analizado, modificado y liberado de las cabeceras a lo largo del camino. Finalmente, alcanzará el nivel de aplicación para que lo pueda usar el proceso receptor.

El modelo ISO formaliza parte del trabajo anterior realizado en el campo de los protocolos de red, pero fue desarrollado a finales de la década de 1970 y actualmente no se utiliza de forma generalizada. Quizá la pila de protocolos más ampliamente adoptada sea el modelo TCP/IP, que se utiliza en la práctica totalidad de los sitios Internet. La pila de protocolos TCP/IP tiene menos

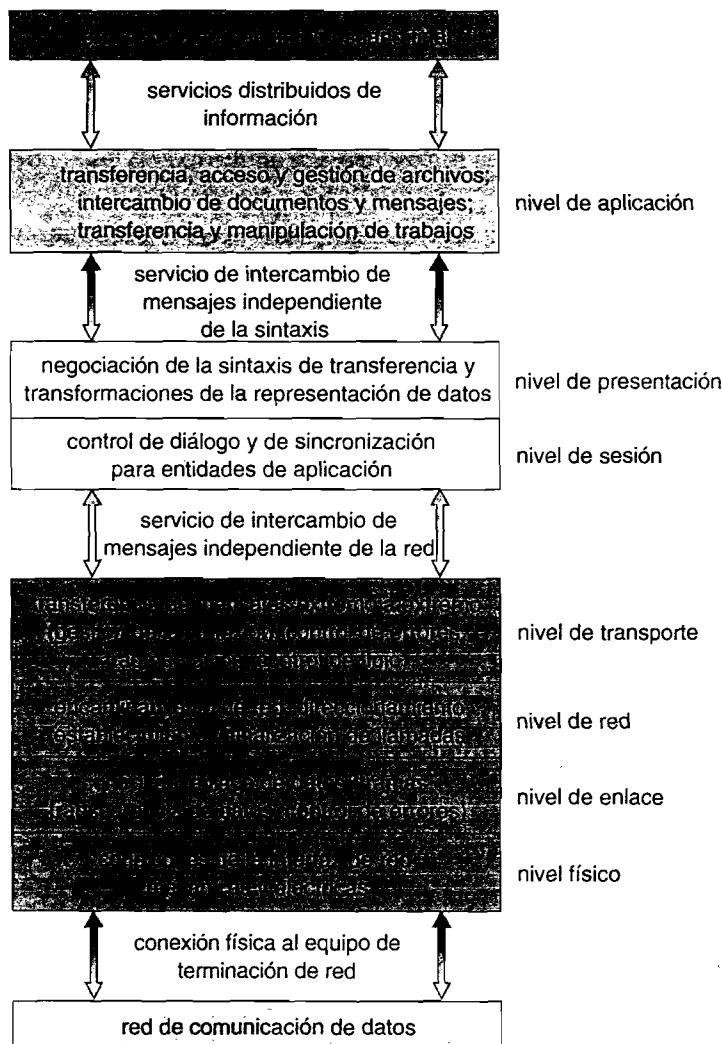


Figura 16.7 La pila del protocolo ISO.

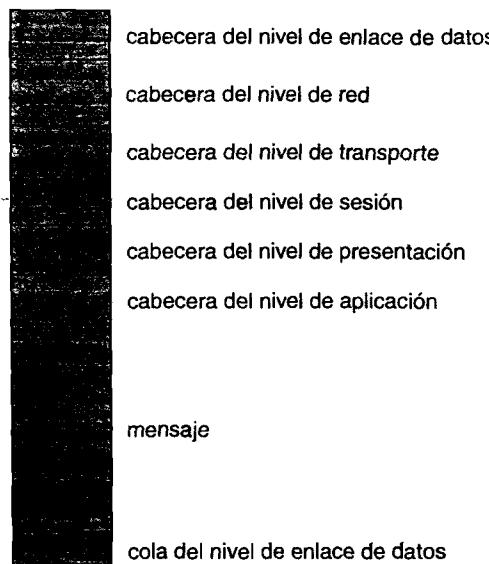


Figura 16.8 Un mensaje de red ISO.

niveles que el modelo ISO. Teóricamente, debido a que combina varias funciones en cada nivel, es más difícil de implementar pero más eficiente que las redes ISO. La relación entre los modelos ISO y TCP/IP se muestra en la Figura 16.9. El nivel de aplicación TCP/IP identifica varios protocolos ampliamente utilizados en Internet, incluyendo HTTP, FTP, Telnet, DNS y SMTP. El nivel de transporte identifica el protocolo sin conexión, no fiable, denominado **protocolo de datagramas de usuario** (UDP, user datagram protocol) y el protocolo fiable orientado a conexión, denominado **protocolo de control de transmisión** (TCP, transmission control protocol). El protocolo Internet (IP) es responsable de encaminar los datagramas IP a través de Internet. El modelo TCP/IP no identifica formalmente ningún nivel de enlace o físico, permitiendo que el tráfico TCP/IP se curse a través de cualquier red física. En la Sección 16.9, consideraremos el modelo TCP/IP sobre una red Ethernet.

16.7 Robustez

Un sistema distribuido puede sufrir diversos tipos de fallos hardware. El fallo de un enlace, el fallo de un sitio y la pérdida de un mensaje son los tipos más comunes. Para garantizar la robustez del sistema, podemos detectar todos estos fallos, reconfigurando el sistema de forma que el procesamiento pueda continuar y efectuando la correspondiente recuperación cuando se repare un sitio o un enlace.

16.7.1 Detección de fallos

En un entorno sin memoria compartida, generalmente no podemos diferenciar entre los fallos de enlace, los fallos de sitios y las pérdidas de mensajes. Usualmente, lo único que podemos detectar es que se ha producido uno de estos fallos. Una vez detectado un fallo, será necesario tomar las medidas apropiadas. Las acciones que resulten apropiadas dependerán de cada aplicación concreta.

Para detectar los fallos de enlaces y de sitios, utilizamos un procedimiento de **contacto**. Suponga que los sitios A y B tienen un enlace físico directo entre ellos. A intervalos fijos, ambos sitios le envían al otro un mensaje de *Estoy funcionando*. Si el sitio A no recibe este mensaje dentro de un período de tiempo predeterminado, puede asumir que el sitio B ha fallado, que el enlace entre A y B ha fallado o que el mensaje procedente de B se ha perdido. En este punto, el sitio A tiene dos opciones. Puede esperar otro período de tiempo para recibir un mensaje de *Estoy funcionando* procedente de B o puede enviar a B un mensaje *¿Estás funcionando?*

- Si ha fallado un enlace directo entre A y B, esta información debe difundirse a todos los sitios del sistema, para poder actualizar correspondientemente las diversas tablas de encaminamiento.
- Si el sistema cree que un sitio ha fallado (porque ya no se puede alcanzar ese sitio), será necesario enviar la correspondiente notificación a todos los sitios del sistema, para que no intenten utilizar más los servicios del sitio que ha fallado. El fallo de un sitio que actúa como coordinador central de alguna actividad (como por ejemplo la de detección de interbloqueos) requerirá la elección de un nuevo coordinador. De forma similar, si el sitio que ha fallado forma parte de un anillo lógico, será necesario construir un nuevo anillo lógico. Observe que, si el sitio no ha fallado (es decir, si está funcionando pero no resulta alcanzable) podemos encontrarnos en la desagradable situación de que dos sitios actúen como coordinador. Cuando la red queda dividida, los dos coordinadores (cada uno en su partición) pueden iniciar acciones conflictivas. Por ejemplo, si los coordinadores son responsables de implementar los mecanismos de exclusión mutua, podemos encontrarnos en la situación en que dos procesos se estén ejecutando simultáneamente dentro de sus secciones críticas.

16.7.3 Recuperación de fallos

Cuando se repara un enlace o un sitio fallido, es necesario volver a integrarlo en el sistema de forma grácil y suave.

- Suponga que ha fallado un enlace entre A y B. Cuando se repare, habrá que notificárselo tanto a A como a B. Podemos llevar a cabo esta notificación repitiendo continuamente el procedimiento de contacto descrito en la Sección 16.7.1.
- Suponga que el sitio B ha fallado. Cuando se recupere, deberá notificar a todos los demás sitios que ya está funcionando de nuevo. El sitio B puede entonces tener que recibir información de los otros sitios para actualizar sus tablas locales; por ejemplo, puede necesitar recibir información de las tablas de encaminamiento, una lista de los sitios que no estén funcionando o los mensajes y el correo electrónico que no hubieran sido entregados. Si el sitio no había fallado, sino que simplemente no era alcanzable, esta información seguirá siendo necesaria.

16.8 Cuestiones de diseño

Hacer que la multiplicidad de procesadores y de dispositivos de almacenamiento sea transparente para los usuarios ha constituido uno de los desafíos principales para muchos diseñadores. Idealmente, un sistema distribuido debe aparecer a ojos de sus usuarios como si fuera un sistema convencional centralizado. La interfaz de usuario de un sistema distribuido transparente no debería distinguir entre los recursos locales y remotos. En otras palabras, los usuarios deberían poder acceder a los recursos remotos como si fueran locales, y el sistema distribuido debería encargarse de localizar los recursos y de gestionar las apropiadas interacciones.

Otro aspecto de la transparencia es la movilidad de los usuarios. Sería deseable permitir a los usuarios iniciar una sesión en cualquier máquina del sistema, en lugar de forzarles a utilizar una máquina específica. Un sistema distribuido transparente facilita la movilidad de los usuarios trasladando el entorno del usuario (por ejemplo, su directorio principal) al lugar donde el usuario haya iniciado la sesión. Tanto el sistema de archivos Andrew de CMU como el Proyecto Athena del MIT proporcionan esta funcionalidad a gran escala; NFS puede proporcionarla a una escala más pequeña.

Otra cuestión de diseño es la relativa a la tolerancia a fallos. Utilizamos el término *tolerancia a fallos* en un sentido amplio. Los fallos de comunicaciones, los fallos de las máquinas (que provocan la detención de las mismas), los fallos catastróficos de los dispositivos de almacenamiento y el desgaste de los medios de almacenamiento deben tolerarse en mayor o menor grado. Un sistema tolerante a fallos debe continuar funcionando, quizás en forma degradada, cuando se enfren-

tengan un papel igual en la operación del sistema y donde, por tanto, cada máquina tenga un cierto grado de autonomía. En la práctica, resulta virtualmente imposible cumplir con este principio. Por ejemplo, la incorporación de máquinas sin disco viola la simetría funcional, ya que las estaciones de trabajo dependen de un disco central. De todos modos, la autonomía y la simetría son objetivos importantes a los que deberíamos aspirar.

Una aproximación práctica a la configuración simétrica y autónoma es la **agrupación en cluster** mediante la que el sistema se divide en una colección de *clusters* semiautónomos. Un *cluster* está compuesto por un conjunto de máquinas y un servidor de *cluster* dedicado. Para que las referencias a recursos de un *cluster* a otro sean relativamente infrecuentes, cada servidor de *cluster* debe satisfacer las solicitudes de sus propias máquinas la mayor parte del tiempo. Por supuesto, este esquema depende de la capacidad de localizar las referencias a recursos y de situar las unidades componentes de la forma apropiada. Si el *cluster* está bien equilibrado, es decir, si el servidor que está a cargo del mismo es suficiente para satisfacer todas las demandas del *cluster*, se lo puede utilizar como bloque componente modular para poder ampliar la escala del sistema.

Decidir la estructura de procesos del servidor constituye uno de los problemas principales en el diseño de cualquier servicio. Se supone que los servidores deben operar de manera eficiente durante los picos de carga, cuando sea necesario servir simultáneamente a cientos de clientes activos. Obviamente, no resulta una buena elección utilizar un servidor de un único proceso, porque cada vez que una solicitud necesite E/S de disco, todo el servicio quedará bloqueado. Resulta más conveniente asignar un proceso a cada cliente; sin embargo, será necesario tener en cuenta el coste de los frecuentes cambios de contexto entre los procesos. Un problema relacionado es el que se presenta debido a que todos los procesos del servidor necesitan compartir información.

Una de las mejores soluciones para definir la arquitectura de un servidor consiste en utilizar procesos ligeros, o hebras, de los que hemos hablado en el Capítulo 4. Podemos pensar en un grupo de procesos ligeros como si fueran múltiples hebras de control asociadas con algunos recursos compartidos. Usualmente, un proceso ligero no está asociado a ningún cliente concreto, sino que se dedica a servir solicitudes de diferentes clientes. La planificación de las hebras puede ser con desalojo o sin desalojo. Si se permite que las hebras continúen ejecutándose hasta completarse (sin desalojo), no será necesario proteger explícitamente sus datos compartidos. En caso contrario, deberá utilizarse algún mecanismo explícito de bloqueo. Claramente, resulta esencial utilizar algún tipo de esquema basado en procesos ligeros para poder disponer de servidores escalables.

16.9 Un ejemplo: conexión en red

Vamos a volver ahora al problema de la resolución de nombres que hemos presentado en la Sección 16.5.1 y vamos a examinar la operación en lo que respecta a la pila de protocolos TCP/IP utilizada en Internet. Vamos a considerar el procesamiento necesario para transferir un paquete entre *hosts* situados en redes Ethernet diferentes.

En una red TCP/IP, cada *host* tiene un nombre y un número Internet asociado de 32 bits (el identificador de *host*). Ambas cadenas deben ser únicas y, para poder gestionar adecuadamente el espacio de nombres, están segmentadas. El nombre es jerárquico (como se explica en la Sección 16.5.1) y describe el nombre de *host* y luego la organización con la que el *host* está asociado. El identificador de *host* se divide en un número de red y un número de *host*. La proporción de división varía dependiendo del tamaño de la red. Una vez que los administradores de Internet asignan un número de red, el sitio con dicho número puede asignar libremente los identificadores de *host*.

El sistema que envía el mensaje consulta sus tablas de encaminamiento para localizar un encaminador con el fin de comenzar el envío del paquete. Los encaminadores usan la parte de red del identificador de *host* para transferir el paquete desde su red de origen hasta su red de destino. El sistema de destino recibirá entonces el paquete. Este paquete puede ser un mensaje completo o ser simplemente un componente de un mensaje, siendo entonces necesarios más paquetes para poder recomponer el mensaje y pasárselo al nivel TCP/UDP para su transmisión al proceso de destino.

Ahora sabemos cómo se desplaza un paquete desde su red de origen hasta su destino, pero ¿cómo se desplaza un paquete dentro de una red desde el emisor (*host* o encaminador) hasta el

receptor? Cada dispositivo Ethernet tiene una cadena de bytes distintiva, denominada **dirección de control de acceso al medio** (MAC, medium access control) que se le asigna para propósitos de direccionamiento. Dos dispositivos dentro de una LAN se comunican entre sí utilizando únicamente este número. Si un sistema necesita enviar datos a otro sistema, el *kernel* genera un paquete del **protocolo de resolución de direcciones** (ARP, address resolution protocol) que contiene la dirección IP del sistema de destino. Este paquete se difunde a todos los demás sistemas presentes en dicha red Ethernet.

Los mensajes de difusión utilizan una dirección de red especial (usualmente, la dirección máxima) para señalizar que todos los *hosts* deben recibir y procesar el paquete. Los mensajes de difusión no son reenviados por las pasarelas, por lo que sólo los reciben los sistemas situados en la red local. Sólo el sistema cuya dirección IP se corresponda con la dirección IP de la solicitud ARP responderá y devolverá su dirección MAC al sistema que realizó la consulta. Por razones de eficiencia, el *host* almacena la pareja de direcciones IP-MAC en una tabla interna. Las entradas de la caché están sujetas a un proceso de **envejecimiento**, de modo que una entrada termina por eliminarse de la caché si no se requiere un acceso a dicho sistema durante un período de tiempo determinado. De esta forma, los *hosts* eliminados de una red terminan por olvidarse. Para aumentar las prestaciones, las entradas ARP correspondientes a los *hosts* más comúnmente utilizados pueden estar precodificadas en la caché ARP.

Una vez que un dispositivo Ethernet ha anunciado su identificador de *host* y su dirección, puede comenzar la comunicación. Un proceso puede especificar el nombre de un *host* con el que quiera comunicarse. El *kernel* toma dicho nombre y determina el número Internet del destino, utilizando una búsqueda DNS. El mensaje se pasa desde el nivel de aplicación al nivel hardware a través de los niveles software intermedios. En el nivel hardware, el paquete (o paquetes) tiene la dirección Ethernet al principio, mientras que la cola del paquete indica el final del mismo y contiene una **suma de comprobación** para detectar los posibles daños que el paquete sufra (Figura 16.10). El paquete es colocado en la red por el dispositivo Ethernet. La sección de datos del paquete puede contener parte de los datos del mensaje original, o la totalidad del mismo, pero también puede contener algunas de las cabeceras de nivel superior que forman el mensaje. En otras palabras, todas las partes del mensaje original deben enviarse desde el original al destino y todas las cabeceras por encima del nivel 802.3 (nivel de enlace de datos) se incluyen como datos en los paquetes Ethernet.

Si el destino se encuentra en la misma red local que el origen, el sistema puede consultar su caché ARP, localizar la dirección Ethernet del *host* y transmitir el paquete a través de los cables de conexión. El dispositivo Ethernet de destino detecta entonces su dirección en el paquete y lo lee, pasándolo hacia arriba a través de la pila de protocolos.

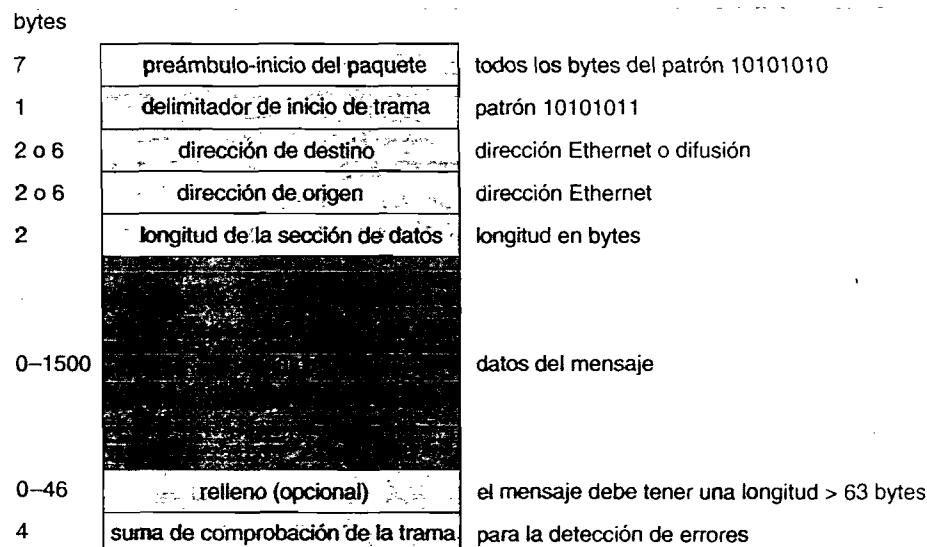


Figura 16.10 Un paquete Ethernet.

Si el sistema de destino se encuentra en una red distinta a la del origen, el sistema de origen localiza un encaminador apropiado dentro de su red y le envía el paquete. Los encaminadores pasan entonces el paquete a través de la WAN hasta que alcanza su red de destino. El encaminador que conecta la red de destino consulta su caché ARP, localiza el número Ethernet del destino y envía el paquete a dicho *host*. En todas estas transferencias, la cabecera del nivel de enlace de datos puede variar, ya que se utiliza la dirección Ethernet del siguiente encaminador de la cadena, pero las restantes cabeceras del paquete siguen siendo las mismas hasta que el paquete es recibido y procesado por la pila de protocolos y el *kernel* se lo pasa finalmente al proceso receptor.

16.10 Resumen

Un sistema distribuido es una colección de procesadores que no comparten memoria ni una señal de reloj. En lugar de ello, cada procesador tiene su propia memoria local y los procesadores se comunican entre sí a través de diversas líneas de comunicaciones, como buses de alta velocidad y líneas telefónicas. Los procesadores de un sistema distribuido varían en cuanto a tamaño y a función. Pueden incluir pequeños microprocesadores, estaciones de trabajo, minicomputadoras y grandes sistemas informáticos de propósito general.

Los procesadores del sistema están conectados a través de una red de comunicaciones, que puede estar configurada de diversas maneras. La red puede estar parcial y totalmente conectada y puede ser un árbol, una estrella, un anillo o un bus multiacceso. El diseño de la red de comunicaciones debe incluir las estrategias de encaminamiento y conexión y debe resolver los problemas de la contienda y la seguridad.

Un sistema distribuido proporciona al usuario acceso a los recursos ofrecidos por el sistema. El acceso a un recurso compartido puede proporcionarse mediante mecanismos de migración de datos, de migración de cálculos o de migración de procesos.

Las pilas de protocolos, tal como se especifican en los modelos de niveles de red, modifican los mensajes, añadiéndoles información para garantizar que alcancen su destino. Debe utilizarse un sistema de nombrado como DNS para traducir entre un nombre de *host* y una dirección de red, y puede ser necesario otro protocolo (como ARP) para traducir el número de red a una dirección de dispositivo de destino (por ejemplo, una dirección Ethernet). Si los sistemas están ubicados en redes separadas, se necesitan encaminadores para pasar los paquetes desde la red de origen hasta la red de destino.

Un sistema distribuido puede sufrir diversos tipos de fallos hardware. Para que un sistema distribuido sea tolerante a fallos, debe detectar los fallos hardware y reconfigurar el sistema. Cuando se repara el fallo, debe reconfigurarse el sistema de nuevo.

Ejercicios

- 16.1 ¿Cuál es la diferencia entre migración de cálculos y migración de procesos? ¿Cuál de los dos mecanismos es más fácil de implementar y por qué?
- 16.2 Compare las diversas topologías de red en términos de los siguientes atributos:
 - a. Fiabilidad.
 - b. Ancho de banda disponible para comunicaciones concurrentes.
 - c. Coste de la instalación.
 - d. Equilibrio de carga en las responsabilidades de encaminamiento.
- 16.3 Aunque el modelo ISO de interconexión por red especifica siete niveles de funcionalidad, la mayoría de los sistemas informáticos utiliza menos niveles para implementar una red. ¿Por qué utilizan menos niveles? ¿Qué problemas puede causar el uso de menos niveles?

Capítulo 16 Estructuras de los sistemas distribuidos

- 16.4 Explique por qué el doblar la velocidad de los sistemas en un segmento Ethernet puede provocar una degradación en el rendimiento de la red. ¿Qué cambios ayudarían a resolver este problema?
- 16.5 ¿Cuáles son las ventajas de utilizar dispositivos hardware dedicados para los encaminadores y pasarelas? ¿Cuáles son las desventajas de utilizar estos dispositivos en lugar de emplear computadoras de propósito general?
- 16.6 ¿Qué ventajas tiene utilizar un servidor de nombres en lugar de emplear tablas de *hosts* estáticas? ¿Qué problemas o complicaciones se derivan del uso de servidores de nombres? ¿Qué métodos podrían utilizarse para reducir la cantidad de tráfico que los servidores de nombres generan para satisfacer las solicitudes de traducción?
- 16.7 Los servidores de nombres están organizados de manera jerárquica. ¿Cuál es el propósito de utilizar una organización jerárquica?
- 16.8 Considere un nivel de red que detecta las colisiones y retransmite el mensaje inmediatamente al detectar una colisión. ¿Qué problemas pueden surgir con esta estrategia? ¿Cómo podrían corregirse?
- 16.9 Los niveles inferiores del modelo de red ISO proporcionan un servicio de datagramas, sin garantía de entrega de los mensajes. Se utiliza un protocolo de nivel de transporte como tcp para proporcionar fiabilidad. Explique las ventajas y desventajas de soportar una entrega de mensajes fiable en el nivel más bajo posible.
- 16.10 ¿Cuál es el impacto que tiene sobre el comportamiento de las aplicaciones la utilización de una estrategia de encaminamiento dinámico? ¿Para qué tipos de aplicaciones resulta beneficioso utilizar un encaminamiento virtual en lugar de un encaminamiento dinámico?
- 16.11 Ejecute el programa mostrado en la Figura 16.5 y determine las direcciones IP de los siguientes nombres de *host*:
- www.wiley.com
 - www.cs.yale.edu
 - www.javasoft.com
 - www.westminstercollege.edu
 - www.ietf.org
- 16.12 Considere un sistema distribuido con dos sitios A y B. Razoné si el sitio A puede distinguir entre los siguientes sucesos:
- a. B sufre un fallo.
 - b. El enlace entre A y B sufre un fallo.
 - c. B está extremadamente sobrecargado y su tiempo de respuesta es cien veces superior al normal.
- ¿Qué implicaciones tiene su respuesta en lo que respecta a la recuperación en los sistemas distribuidos?
- 16.13 El protocolo HTTP original utilizaba TCP/IP como protocolo de red subyacente. Por cada página, gráfico o *applet*, se establecía, utilizaba y finalizaba una sesión TCP separada. Debido al coste asociado al establecimiento y finalización de conexiones TCP/IP este método de implementación presentaba problemas de rendimiento. ¿Sería una buena alternativa utilizar UDP en lugar de TCP? ¿Qué otros cambios podrían hacerse para mejorar el rendimiento de HTTP?
- 16.14 ¿Qué utilidad tiene un protocolo de resolución de direcciones? ¿Por qué es mejor usar uno de esos protocolos en lugar de hacer que cada *host* lea cada paquete con el fin de determinar la dirección de destino?

nar el destino de dicho paquete? ¿Necesitan dicho tipo de protocolo las redes de paso de testigo? Razone su respuesta.

- 16.15 ¿Cuáles son las ventajas y desventajas de hacer que la red informática sea transparente para el usuario?

Notas bibliográficas

Tanenbaum [2003], Stallings [2000b] y Kurose y Ross [2005] proporcionaron panorámicas generales de las redes informáticas. Williams [2001] cubre la conexión de computadoras en red desde el punto de vista de la arquitectura informática.

Internet y sus protocolos se describen en Comer [1999] y Comer [2000]. Puede encontrarse información sobre TCP/IP en Stevens [1994] y Stevens [1995]. La programación en redes UNIX se describe en Stevens [1997] y Stevens [1998].

Coulouris et al. [2001] y Tanenbaum y van Steen [2002] ofrecen una exposición sobre las estructuras de sistemas operativos distribuidos.

El equilibrado de carga y la compartición de carga se tratan en Harchol-Balter y Downey [1997], y Vee y Hsu [2000]. Harish y Ownes [1999] describe el equilibrado de carga en los servidores DNS. La migración de procesos se trata en Jul et al. [1988], Douglis y Ousterhout [1991], Han y Ghosh [1998] y Milojicic et al. [2000]. Sirer et al. [1999] examina las cuestiones relacionadas con una máquina virtual distribuida para sistemas distribuidos.



Sistemas de archivos distribuidos



En el capítulo anterior, hemos analizado el tema de la construcción de redes y de los protocolos de bajo nivel necesarios para poder transferir mensajes entre sistemas. Ahora vamos a examinar uno de los posibles usos de esta infraestructura. Un **sistema de archivos distribuido** (DFS, distributed file system) es una implementación distribuida del modelo clásico de compartición de tiempo de un sistema de archivos, en el que múltiples usuarios comparten archivos y recursos de almacenamiento (Capítulo 11). El propósito de un sistema DFS es emplear el mismo tipo de compartición cuando los archivos están físicamente dispersos entre los sitios que componen un sistema distribuido.

En este capítulo, vamos a describir como puede diseñarse e implementarse un sistema distribuido. En primer lugar, analizaremos algunos conceptos comunes en los que se basan los sistemas DFS. A continuación, ilustraremos dichos conceptos examinando un sistema DFS bastante relevante: el sistema de archivos Andrew (AFS).

OBJETIVOS DEL CAPÍTULO

- Explicar el mecanismo de nombrado que proporciona la independencia y la transparencia de ubicación.
- Describir los diversos métodos para acceder a archivos distribuidos.
- Comparar los servidores de archivos distribuidos con o sin memoria del estado.
- Mostrar cómo la replicación de archivos en diferentes máquinas constituye un útil mecanismo de redundancia para mejorar la disponibilidad.
- Presentar el sistema de archivos Andrew (AFS, Andrew file system) como ejemplo de sistema de archivos distribuido.

17.1 Conceptos esenciales

Como hemos indicado en el capítulo anterior, un sistema distribuido es una colección de computadoras débilmente acopladas interconectadas por una red de comunicaciones. Estas computadoras pueden compartir archivos físicamente dispersos utilizando un sistema de archivos distribuido (DFS, distributed file system). En este capítulo utilizamos el término *DFS* para referirnos a los sistemas de archivos distribuidos en general, no al producto comercial Transarc DFS. A este último le denominaremos *Transarc DFS*. Asimismo, NFS hace referencia a NFS versión 3, a menos que se indique lo contrario.

Para explicar la estructura de un sistema DFS, necesitamos definir los términos *servicio*, *servidor* y *cliente*. Un servicio es una entidad software que se ejecuta en una o más máquinas y que proporciona un tipo particular de función a los clientes. Un *servidor* es el software de servicio que se ejecuta en una única máquina. Un cliente es un proceso que puede invocar un servicio utilizando

un conjunto de operaciones que forman su **interfaz de cliente**. En ocasiones, se define una interfaz de menor nivel para la propia interacción entre unas máquinas y otras; se trata de la **interfaz intermáquinas**.

Utilizando esta terminología, decimos que un sistema de archivos proporciona servicios de archivos a los clientes. Una interfaz de cliente para un servicio de archivos está formada por un conjunto de operaciones primitivas de archivos, como la de creación de un archivo, la de borrado de un archivo, la de lectura de un archivo y la de escritura en un archivo. El componente hardware principal que un servidor de archivos controla es un conjunto de dispositivos de almacenamiento secundario locales (usualmente, discos magnéticos) en los que se almacenan y de los que se extraen los archivos de acuerdo con las solicitudes de los clientes.

Un DFS es un sistema de archivos cuyos clientes, servidores y dispositivos de almacenamiento están dispersos entre las máquinas de un sistema distribuido. Correspondiente, la actividad de servicio tiene que ser llevada a cabo a través de la red. En lugar de existir un único repositorio de datos centralizado, el sistema tiene frecuentemente dispositivos de almacenamiento múltiples e independientes. Como veremos en este texto, la configuración e implementación concretas de un DFS pueden variar de un sistema a otro. En algunas configuraciones, los servidores se ejecutan sobre máquinas dedicadas; en otras, una máquina puede actuar a la vez como servidor y como cliente. Un DFS se puede implementar como parte de un sistema distribuido o, alternativamente, mediante un nivel software cuya tarea consiste en gestionar la comunicación entre sistemas operativos convencionales y sistemas de archivos. Las características distintivas de un DFS son la multiplicidad y la autonomía de los clientes y servidores del sistema.

Idealmente, un DFS debe aparecer a ojos de sus clientes como si fuera un sistema de archivos centralizado convencional. La multiplicidad y la dispersión de sus servidores y de sus dispositivos de almacenamiento deben ser invisibles para los usuarios. En otras palabras, la interfaz de cliente de un DFS no debe distinguir entre los archivos locales y remotos. Es responsabilidad del DFS localizar los archivos y organizar el transporte de los datos. Un DFS transparente facilita la movilidad de los usuarios trasladando el entorno del usuario (es decir, el directorio principal) al lugar donde el usuario haya iniciado la sesión.

La medida más importante de rendimiento de un DFS es la cantidad de tiempo necesaria para satisfacer solicitudes de servicio. En los sistemas convencionales, este tiempo está compuesto por el tiempo de acceso a disco y por una pequeña cantidad de tiempo de procesamiento invertido por la CPU. Sin embargo, en un DFS, cada acceso remoto tiene un coste adicional, debido a la estructura distribuida. Este coste incluye el tiempo para entregar la solicitud a un servidor, así como el tiempo para que el cliente obtenga la respuesta a través de la red. Para cada una de las direcciones de comunicación, además de la transferencia de la información, tenemos el coste de CPU requerido para ejecutar el software del protocolo de comunicaciones. El rendimiento de un DFS puede considerarse como otra dimensión de la transparencia del sistema DFS. En otras palabras, el rendimiento de un DFS ideal debería ser comparable al de un sistema de archivos convencional.

El hecho que un DFS gestione un conjunto de dispositivos dispersos de almacenamiento constituye la característica distintiva más importante de los sistemas DFS. El espacio de almacenamiento total gestionado por un DFS está compuesto de espacios de almacenamiento más pequeños que están separados y que se ubican de forma remota. Usualmente, estos espacios de almacenamiento constituyentes se corresponden con conjuntos de archivos. Una **unidad componente** es el conjunto de archivos más pequeño que puede almacenarse en una sola máquina, independientemente de otras unidades. Todos los archivos que pertenezcan a la misma unidad componente deben residir en la misma ubicación.

17.2 Nombrado y transparencia

El **nombrado** es una correspondencia entre los objetos lógicos y físicos. Por ejemplo, los usuarios tratan con objetos lógicos de datos representados por nombres de archivos, mientras que el sistema manipula bloques físicos de datos almacenados en las pistas de los discos. Usualmente, los usuarios hacen referencia a los archivos mediante un nombre textual. El nombre textual se hace corresponder con un identificador numérico de menor nivel que a su vez se hace corresponder

con los bloques de disco. Esta correspondencia multinivel proporciona a los usuarios una abstracción de un archivo que oculta los detalles de cómo y en qué lugar del disco está almacenado el archivo.

En un DFS transparente, se añade una nueva dimensión a esta abstracción: la de ocultar en qué lugar de la red reside el archivo. En un sistema de archivos convencional, el rango de esa correspondencia de nombrado son las direcciones dentro de un disco. En un DFS, este rango se expande para incluir la máquina específica en cuyo disco está almacenado el archivo. Si vamos un paso más allá con el concepto de tratar los archivos como abstracciones, nos encontramos con la posibilidad de la **replicación de archivos**. Dado un nombre de archivo, la correspondencia de nombrado devuelve un conjunto de ubicaciones correspondientes a las réplicas del archivo. Según esta abstracción, tanto la existencia de múltiples copias como sus ubicaciones están ocultas.

17.2.1 Estructuras de nombrado

Necesitamos diferenciar dos ideas relacionadas en lo que respecta a las correspondencias de nombres en un DFS:

1. **Transparencia de ubicación.** El nombre de un archivo no revela ninguna información acerca de la ubicación física de almacenamiento del archivo.
2. **Independencia de ubicación.** No es necesario modificar el nombre de un archivo cuando varía la ubicación física de almacenamiento del archivo.

Ambas definiciones son relativas al nivel de nombrado del que hemos hablado anteriormente, dado que los archivos tienen nombres diferentes en los distintos niveles (es decir, nombres textuales de nivel de usuario e identificadores numéricos de nivel del sistema). Un esquema de nombrado independiente de la ubicación constituye una correspondencia dinámica, ya que puede hacer corresponder el mismo nombre con ubicaciones diferentes en dos instantes distintos. Por tanto, la independencia de la ubicación es una propiedad más estricta que la de la transparencia de ubicación.

En la práctica, la mayoría de los sistemas DFS actuales proporcionan una correspondencia estática de los nombres de nivel de usuario, que es transparente respecto a la ubicación. Sin embargo, estos sistemas no soportan la **migración de archivos**; es decir, resulta imposible cambiar automáticamente la ubicación de un archivo. Por tanto, la noción de independencia de la ubicación es irrelevante para estos sistemas. Los archivos están asociados permanentemente con un conjunto específico de bloques de disco. Los archivos y los discos pueden desplazarse de unas máquinas a otras manualmente, pero la migración de archivos implica una acción automática iniciada por el SO. Sólo AFS y unos cuantos sistemas de archivos experimentales permiten la independencia de ubicación y la movilidad de archivos. AFS soporta la movilidad de archivos principalmente con propósitos administrativos. Un protocolo proporciona el mecanismo de migración de las unidades componentes de AFS para satisfacer las solicitudes de alto nivel de los usuarios, sin cambiar ni los nombres de nivel de usuario ni los nombres de bajo nivel de los correspondientes archivos.

Unos cuantos aspectos nos permiten diferenciar aún mejor los conceptos de independencia de ubicación y de transparencia estática de ubicación:

- La separación entre datos y ubicación, tal como se implementa mediante la independencia de ubicación, proporciona una mejor abstracción para los archivos. Cada nombre de archivo debe denotar los atributos más significativos del archivo, que son sus contenidos y no la ubicación. Los archivos independientes de la ubicación pueden contemplarse como contendores lógicos de datos que no están asociados con ninguna ubicación de almacenamiento específico. Si sólo se soporta la transparencia estática de ubicación, el nombre de archivo seguirá denotando un conjunto específico, aunque oculto, de bloques físicos de disco.
- La transparencia estática de ubicación proporciona a los usuarios una forma cómoda de compartir datos. Los usuarios pueden compartir archivos remotos simplemente nombrando los archivos en una forma transparente con respecto a la ubicación, como si los archivos fueran locales. Sin embargo, resulta bastante engorroso compartir el espacio de almacenamiento.

miento, porque los nombres lógicos siguen estando asociados estáticamente a los dispositivos físicos de almacenamiento. La independencia de ubicación promueve la compartición del propio espacio de almacenamiento, además de la compartición de los objetos de datos. Cuando los archivos pueden ser movilizados, el espacio de almacenamiento total del sistema parece un único recurso virtual. Una de las posibles ventajas de este mecanismo es la capacidad de equilibrar la utilización de los discos en todo el sistema.

- La independencia de ubicación separa la jerarquía de nombres de la jerarquía de dispositivos de almacenamiento y de la estructura de interconexión de las computadoras. Por contraste, si se emplea una transparencia estática de ubicación (aunque los nombres sean transparentes), podemos determinar fácilmente la correspondencia entre las unidades componentes y las máquinas. Las máquinas están configuradas según un patrón similar al de la estructura de nombres. Esta configuración puede restringir la arquitectura del sistema innecesariamente y entrar en conflicto con otras consideraciones. Un servidor a cargo de un directorio raíz constituye un ejemplo de estructura que está dictada por la jerarquía de nombres y que contradice las directrices de descentralización.

Una vez completada la separación entre nombres y ubicaciones, los clientes pueden acceder a los archivos que residen en sistemas servidores remotos. De hecho, estos clientes pueden ser sin disco y depender de los servidores para que estos les proporcionen todos los archivos, incluyendo el *kernel* del sistema operativo. Sin embargo, harán falta protocolos especiales para la secuencia de arranque. Considere el problema de enviar el *kernel* a una estación de trabajo sin disco. La estación de trabajo sin disco no tiene ningún *kernel*, por lo que no puede utilizar el código DFS para extraer el *kernel*. En lugar de ello, se invoca un protocolo de arranque especial, que está almacenado en la memoria de sólo lectura (ROM, read-only memory) del cliente. Este protocolo permite la conexión por red y extrae únicamente un archivo especial (el *kernel* o el código de arranque) desde una ubicación fija. Una vez que se ha copiado el *kernel* desde la red y que se ha cargado, su DFS hace que todos los demás archivos del sistema operativo estén disponibles. Son numerosas las ventajas de los clientes sin disco, incluyendo un menor coste (porque las máquinas cliente no necesitan ningún disco) y una mayor comodidad (cuando tiene lugar una actualización del sistema operativo, sólo es necesario modificar el servidor). Las desventajas son la mayor complejidad de los protocolos de arranque y el menor rendimiento como consecuencia de utilizar una red en lugar de un disco local.

La tendencia actual es que los clientes utilicen tanto discos locales como servidores remotos de archivos. Los sistemas operativos y el software de interconexión de red se almacenan localmente; los sistemas de archivos que contienen datos del usuario (y posiblemente aplicaciones) se almacenan en sistemas de archivos remotos. Algunos sistemas cliente pueden almacenar también aplicaciones comúnmente utilizadas, como por ejemplo procesadores de texto y exploradores web, en el sistema de archivos local. Otras aplicaciones menos comúnmente utilizadas pueden ser extraídas desde el servidor de archivos remoto y cargados en el cliente según sea necesario. La principal razón para proporcionar a los clientes sistemas de archivos locales en lugar de sistemas puros sin disco es que las unidades de disco están aumentando rápidamente de capacidad y disminuyendo de coste, apareciendo nuevas generaciones de unidades de disco, aproximadamente cada año. No se puede decir lo mismo de las redes, que evolucionan cada varios años. En conjunto, los sistemas están creciendo más rápidamente que las redes, por lo que hace falta un esfuerzo adicional para limitar el acceso a la red con el fin de aumentar la tasa de procesamiento del sistema.

17.2.2 Esquemas de nombrado

Existen tres técnicas principales para construir esquemas de nombrado en un DFS. Con el enfoque más simple, cada archivo se identifica mediante una combinación de su nombre de *host* y de su nombre local, lo que garantiza un nombre único en todo el sistema. En Ibis, por ejemplo, cada archivo se identifica únicamente mediante el nombre *host:nombre-local*, donde *nombre-local* es una ruta estilo UNIX. Este esquema de nombrado no es ni transparente con respecto a la ubicación ni independiente con respecto a la ubicación; de todos modos, pueden utilizarse las mismas ope-

raciones de archivos, tanto para los archivos remotos como para los locales. El DFS está estructurado como una colección de unidades componentes aisladas, cada una de las cuales es un sistema de archivos convencional completo. Con esta primera técnica, las unidades componentes permanecen aisladas, aunque proporcionan mecanismos para hacer referencia a los archivos remotos. A lo largo de este texto no analizaremos con más detalle este esquema.

La segunda técnica fue popularizada por el sistema de archivos de red (NFS, network file system) de Sun. NFS es el componente del sistema de archivos de ONC+, un paquete de interconexión por red soportado por muchos fabricantes de UNIX. NFS proporciona un medio para asociar directorios remotos a los directorios locales, proporcionando así la apariencia de un árbol de directorios coherente. Las primeras versiones de NFS sólo permitían acceder transparentemente a los directorios remotos previamente montados. Con la aparición de la característica de **automontaje**, el montaje se realiza según es necesario, basándose en una tabla de puntos de montaje y nombres de estructura de archivos. Los componentes se integran para soportar una compartición transparente, aunque esta integración está limitada y no es uniforme, porque cada máquina puede asociar directorios remotos distintos a su árbol. La estructura resultante es bastante versátil.

Podemos conseguir una total integración de los sistemas de archivos componentes utilizando la tercera de las técnicas. Con ella, hay una única estructura global de nombres que abarca a todos los archivos del sistema. Idealmente, la estructura de sistemas de archivos compuesta es isomorfa a la estructura de un sistema de archivos convencional. Sin embargo, en la práctica, los numerosos archivos especiales (por ejemplo, los archivos de dispositivos UNIX y los directorios binarios específicos de la máquina) hacen que este objetivo sea difícil de alcanzar.

Para evaluar las estructuras de nombrado, tenemos que analizar su **complejidad administrativa**. La estructura más compleja y más difícil de mantener es la estructura NFS. Puesto que puede asociarse cualquier directorio remoto en cualquier lugar del árbol de directorios local, la jerarquía resultante puede ser altamente no estructurada. Si un servidor deja de estar disponible, dejará también de estar disponible algún conjunto arbitrario de directorios en diferentes máquinas. Además, hay un mecanismo de acreditación separado que controla qué máquinas están autorizadas a asociar determinados directorios a su árbol. Por tanto, un usuario puede ser capaz de acceder a un árbol de directorios remoto en un cliente, pero denegársele el acceso en otro cliente.

17.2.3 Técnicas de implementación

La implementación de un convenio transparente de nombrado requiere algún tipo de mecanismo para establecer la correspondencia entre el nombre de un archivo y la ubicación asociada. Para que este mecanismo de correspondencia sea manejable, debemos agregar conjuntos de archivos en unidades componentes y realizar la correspondencia basándonos en las unidades componentes, en lugar de realizarla por separado para cada archivo. Este proceso de agregación también simplifica las tareas administrativas. Los sistemas de tipo UNIX utilizan el árbol de directorios jerárquico para proporcionar esta correspondencia entre nombres y ubicaciones y para agregar los archivos recursivamente en directorios.

Para mejorar la disponibilidad de esta información crucial de correspondencia, podemos utilizar mecanismos de replicación, de caché local o ambos. Como hemos indicado anteriormente, el concepto de independencia de la ubicación implica que esas correspondencias cambian a lo largo del tiempo; por tanto, si replicamos esas correspondencias, se vuelve imposible la actualización de esta información de una manera simple y coherente. Una técnica para eliminar este obstáculo consiste en introducir **identificadores de archivo independientes de la ubicación** de bajo nivel. Los nombres de archivo textuales se asignan a identificadores de archivo de bajo nivel que indican a qué unidad componente pertenece el archivo. Estos identificadores siguen siendo independientes de la ubicación. Se los puede replicar y almacenar en caché libremente sin que se vean invalidados por la migración de unidades componentes. El precio que hay que pagar de manera inevitable es la necesidad de un segundo nivel de correspondencias, que haga corresponder a cada unidad componente una ubicación determinada y que necesita un mecanismo de actualización simple y coherente. Si implementamos los árboles de directorio de tipo UNIX utilizando estos identificadores de bajo nivel independientes de la ubicación, toda la jerarquía será invariante con

respecto a la migración de las unidades componentes. El único aspecto que cambia es la correspondencia entre las unidades componentes y las ubicaciones.

Una forma común de implementar identificadores de bajo nivel consiste en utilizar nombres estructurados. Estos nombres son cadenas de bits que están formadas, usualmente, por dos partes. La primera parte identifica la unidad componente a la que pertenece el archivo. La segunda parte identifica el archivo concreto dentro de esa unidad. También son posibles algunas otras variantes con más partes. Sin embargo, la invariancia de los nombres estructurados es que todas las partes individuales del nombre son unívocas en todo momento sólo dentro del contexto de las partes. Podemos conseguir la unicidad en todo momento teniendo cuidado de no reutilizar un nombre que esté siendo usado todavía, añadiendo un número suficiente de bits adicionales (este método se utiliza en AFS) o empleando una marca temporal como otra parte del nombre (como se hace en Apollo Domain). Otra forma de ver este proceso es que estamos tomando un sistema transparente con respecto a la ubicación, tal como Ibis y añadiendo otro nivel de abstracción para producir un esquema de nombrado independiente con respecto a la ubicación.

La agregación de archivos en unidades componentes y la identificación de identificadores de archivo de bajo nivel independientes respecto a la ubicación son técnicas ejemplificadas en AFS.

17.3 Acceso remoto a archivos

Supongamos que un usuario solicita acceder a un archivo remoto. El servidor donde se almacena el archivo ha sido localizado por el esquema de nombrado y con eso puede tener lugar la propia transferencia de datos.

Una forma de llevar a cabo esta transferencia es mediante un **mecanismo de servicio remoto**, mediante el cual las solicitudes de acceso se entregan al servidor, la máquina servidora realiza los accesos y los resultados se devuelven al usuario. Una de las formas más comunes de implementar un servicio remoto es el paradigma de llamadas a procedimientos remotos (RPC, remote procedure call), del que hemos hablado en el Capítulo 3. Existe una analogía directa entre los métodos de acceso a disco en los sistemas de archivos convencionales y el método de servicio remoto en un DFS. Utilizar el método de servicio remoto es análogo a realizar un acceso a disco por cada solicitud de acceso.

Para garantizar un rendimiento razonable del mecanismo de servicio remoto, podemos utilizar algún tipo de caché. En los sistemas de archivos convencionales, la razón de utilizar el almacenamiento en caché es reducir la E/S de disco (incrementando así la velocidad), mientras que en un DFS, el objetivo es reducir tanto el tráfico red como el de E/S de disco. En el siguiente análisis, vamos a describir la implementación del mecanismo de caché en un DFS y a compararla con el paradigma básico de servicio remoto.

17.3.1 Esquema básico de caché

El concepto de almacenamiento en caché es simple. Si los datos necesarios para satisfacer la solicitud de acceso no se encuentran ya en la caché, entonces se trae una copia de dichos datos desde el servidor hasta el sistema cliente. Los accesos se realizan sobre la copia almacenada en caché. La idea es retener en la caché los bloques de disco a los que se ha accedido recientemente, de modo que los accesos repetidos a la misma información puedan gestionarse localmente, sin necesidad de tráfico red adicional. La implementación de algún tipo de sustitución (por ejemplo, la de los datos menos recientemente utilizados) hace que el tamaño de la caché se mantenga acotado. No existe correspondencia directa entre los accesos y el tráfico dirigido al servidor. Los archivos pueden seguir identificándose con una copia maestra que reside en la máquina servidora, pero una serie de copias del archivo (o de partes del mismo) estarán dispersas en las diferentes cachés. Cuando se modifica una copia almacenada en caché, será necesario reflejar los cambios en la copia maestra, con el fin de preservar la correspondiente semántica de coherencia. El problema de mantener las copias de caché coherentes con el archivo maestro se denomina **problema de la coherencia de caché**, y hablaremos de dicho problema en la Sección 17.3.4. Los mecanismos de caché de un DFS también podrían denominarse **memoria virtual de red**, ya que actúan de forma similar a

la forma virtual de paginación bajo demanda, salvo porque el almacenamiento de respaldo no es, usualmente, un disco local sino un servidor remoto. NFS permite montar remotamente el espacio de intercambio, de modo que puede verdaderamente implementar una memoria virtual a través de la red, aunque por supuesto con un considerable impacto sobre las prestaciones.

La granularidad de los datos almacenados en caché en un DFS puede variar, pudiendo definirse esa granularidad en el nivel de bloque de archivo o en el de un archivo completo. Usualmente, se almacenan en caché más datos de los necesarios para satisfacer un único acceso, de modo que se pueda dar servicio a muchos accesos mediante los datos almacenados en caché. Este procedimiento se parece bastante al mecanismo de lectura anticipada de disco (Sección 11.6.2). AFS almacena en caché los archivos utilizando fragmentos de gran tamaño (64 KB). Los otros sistemas analizados en este capítulo permiten almacenar en caché bloques individuales, a medida que sean demandados por los clientes. Incrementar el tamaño de la unidad de caché permite aumentar la tasa de aciertos, pero también incrementa la penalización correspondiente a los fallos, porque cada fallo requerirá transferir más datos. También incrementa la posibilidad de que aparezcan problemas de coherencia. Para seleccionar la unidad de almacenamiento en caché, debemos tener en cuenta parámetros tales como la unidad de transferencia de red y la unidad de servicio del protocolo RPC (si se utiliza un protocolo RPC). La unidad de transferencia de red (para Ethernet, un paquete) es de aproximadamente 1,5 KB, por lo que las unidades de datos de caché que tengan un mayor tamaño necesitarán ser desensambladas para poder entregarlas y re-ensambladas después de recibirlas.

El tamaño de bloque y el tamaño total de la caché resultan evidentemente importantes para los esquemas de almacenamiento de bloques en caché. En los sistemas tipo UNIX, los tamaños de bloque más comunes son 4 KB y 8 KB. Para las memorias caché de gran tamaño (más de 1 MB), resulta ventajoso utilizar tamaños de bloque grandes (por encima de 8 KB). Para las memorias caché más pequeñas, los tamaños de bloque grande son menos ventajosos, porque hacen que se almacenen menos bloques en la caché y, por tanto, que se consiga una menor tasa de aciertos.

17.3.2 Ubicación de la caché

¿Dónde deben almacenarse los datos de caché, en el disco o en la memoria principal? Las cachés de disco tienen una clara ventaja sobre las cachés de memoria principal: son bastante más fiables. Las modificaciones efectuadas en los datos almacenados en caché se perderán cuando se sufra un fallo catastrófico de la máquina, si la caché se almacena en memoria volátil. Además, si los datos de caché se almacenan en disco, seguirán estando allí durante la recuperación, y no será necesario volver a extraerlos. Las cachés de memoria principal tienen, de todos modos, varias ventajas:

- Las cachés de memoria principales permiten que las estaciones de trabajo no utilicen disco.
- Resulta más rápido acceder a los datos almacenados en una caché de memoria principal que a los de una caché de disco.
- La tecnología está evolucionando para dar memorias de mayor tamaño y menor coste. Las previsiones son que las ventajas de velocidad que se podrán conseguir de esta manera compensarán con creces las ventajas de la caché de disco.
- Las cachés de servidor (utilizadas para acelerar la E/S de disco) residirán en memoria principal, independientemente de dónde residan las cachés de usuario; si utilizamos también cachés de memoria principal en las máquinas de usuario, podemos definir un único mecanismo de caché para utilizarlo tanto en los servidores como en los clientes.

Hay muchas implementaciones de los mecanismos de acceso remoto que pueden considerarse como híbridos de los mecanismos de caché y de los de servicio remoto. En NFS, por ejemplo, la implementación está basada en un servicio remoto, pero complementado con cachés de memoria del lado del cliente y del lado del servidor para mejorar la velocidad. De forma similar, la implementación de Sprite está basada en mecanismos de caché, pero en ciertas circunstancias se adopta un método basado en servicio remoto. Por tanto, para evaluar los dos métodos, debemos primero determinar hasta qué punto se pone el énfasis en uno u otro de los métodos.

El protocolo NFS y la mayoría de las implementaciones no proporcionan ninguna caché de disco. Las implementaciones recientes de NFS en Solaris (Solaris 2.6 y superior) incluyen una opción de caché de disco del lado del cliente, el sistema de archivos **cachefs**. Una vez que el cliente NFS lee bloques de un archivo desde el servidor, los almacena en la caché de memoria y también en la caché de disco. Si se vacía la copia de memoria, o si se produce un re-arranque del sistema, se consulta la caché de disco. Si un bloque necesario no se encuentra ni en la memoria ni en la caché de disco **cachefs**, se realiza una llamada RPC al servidor con el fin de extraer el bloque, y dicho se bloque se almacena tanto en la caché de disco como en la caché de memoria, para que lo utilice el cliente.

17.3.3 Política de actualización de la caché

La política que se utilice para escribir los bloques de datos modificados en la copia maestra del servidor tiene un efecto crítico sobre la fiabilidad y las prestaciones del sistema. La política más simple consiste en escribir los datos en disco en cuanto se los coloca en cualquier caché. La ventaja de esta **política de escritura directa** es la fiabilidad: es muy poca la información que se pierde cuando un sistema cliente sufre un fallo catastrófico. Sin embargo, esta política requiere que cada acceso de escritura espere hasta que se envíe la información al servidor, por lo que la velocidad de escritura es muy baja. El almacenamiento en caché con un mecanismo de escritura directa es equivalente a utilizar un servicio remoto para los accesos de escritura y aprovechar la caché únicamente para los accesos de lectura.

Una alternativa es la **política de escritura diferida**, también denominada **caché de escritura diferida**; con este tipo de política, lo que hacemos es retardar las actualizaciones de la copia maestra. Las modificaciones se escriben en la caché y luego se envían al servidor en un momento posterior. Esta política tiene dos ventajas sobre la de escritura directa. En primer lugar, puesto que las escrituras se realizan en la caché, los accesos de escritura se completan mucho más rápidamente. En segundo lugar, los datos pueden ser sobrescritos antes de enviarlos al servidor, en cuyo caso sólo será necesario escribir en el servidor la última actualización. Lamentablemente, los esquemas de escritura diferida introducen problemas de fiabilidad, ya que los datos no escritos se perderán si una máquina de usuario sufre un fallo catastrófico.

Las diversas variantes de la política de escritura diferida se diferencian en lo que respecta al momento en que se vuelcan en el servidor los bloques de datos modificados. Una alternativa consiste en volcar un bloque cuando esté a punto de ser expulsado de la caché de cliente. Esta opción puede proporcionar unas buenas prestaciones, pero algunos bloques pueden permanecer en la caché de cliente durante mucho tiempo antes de escribirlos en el servidor. Un compromiso entre esta alternativa y la política de escritura directa consiste en explorar la caché a intervalos regulares y volcar aquellos bloques que hayan sido modificados desde la última exploración, utilizando el mismo método que UNIX emplea para explorar su caché local. Sprite utiliza esta política, con un intervalo de 30 segundos. NFS emplea también esta política para los datos de archivo, pero una vez que se realiza una escritura en el servidor durante un volcado de caché, es necesario esperar a que esa escritura se realice de forma efectiva en el disco del servidor antes de poder considerar la completa. NFS trata los metadatos (datos de directorio y datos de atributos de los archivos) de forma diferente. Los cambios en los metadatos se envían de manera síncrona al servidor. De este modo, se evita la pérdida de información sobre la estructura de archivos y la corrupción de la estructura de directorios cuando se produce un fallo catastrófico en el cliente o en el servidor.

Para NFS con **cachefs**, las escrituras también se realizan en caché de disco local en el momento de llevarlas a cabo en el servidor, con el fin de mantener la coherencia entre todas las copias. Por tanto, NFS con **cachefs** mejora las prestaciones con respecto al sistema NFS estándar en la solicitudes de lectura para las que haya un acierto de caché **cachefs** pero reduce las prestaciones para las solicitudes de lectura o de escritura en las que se produzca un fallo de caché. Al igual que con todos los mecanismos de caché, resulta fundamental tener una alta tasa de aciertos de caché, con el fin de mejorar la velocidad. En la Figura 17.1 se ilustra el sistema **cachefs** y su uso de los mecanismos de caché con escritura directa y escritura diferida.

Otra variante de la escritura diferida consiste en escribir los datos en el servidor en el momento de cerrar el archivo. Esta **política de escritura durante el cierre** se utiliza en AFS. En el caso de

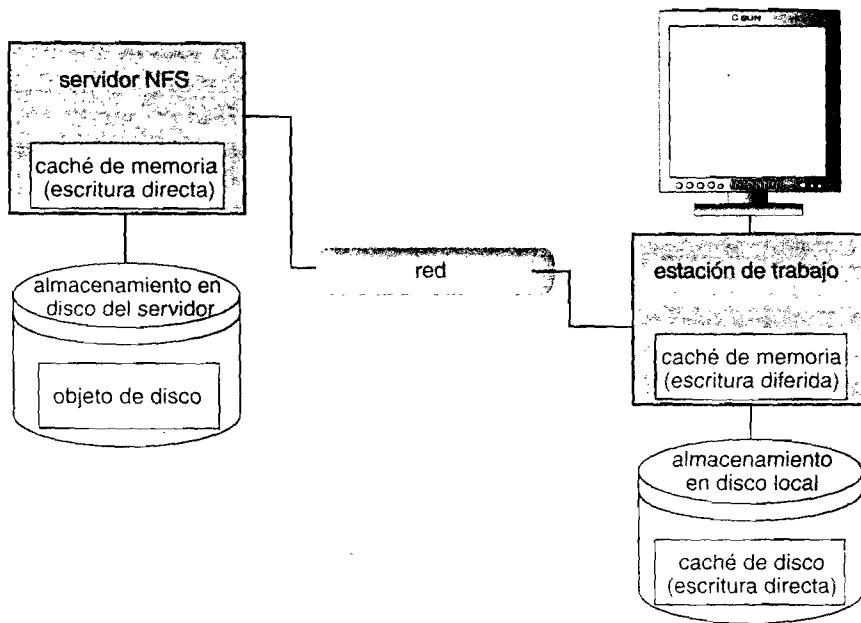


Figura 17.1 Utilización de los mecanismos de caché en cachefs.

los archivos que se abran durante cortos períodos de tiempo o que se modifiquen muy raramente, esta política no reduce significativamente el tráfico de red. Además, la política de escritura durante el cierre requiere que el proceso que realiza el cierre se quede a la espera mientras que se está escribiendo el archivo, lo que reduce la ventaja de prestaciones ofrecida por las escrituras diferidas. Sin embargo, para los archivos que están abiertos durante largos períodos de tiempo o que se modifiquen frecuentemente, las ventajas de velocidad de esta política con respecto a la escritura diferida con un volcado más frecuente resultan bastante evidentes.

17.3.4 Coherencia

Las máquinas cliente se enfrentan al problema de decidir si una copia de los datos almacenada en la caché local es coherente o no con la copia maestra (y puede por tanto ser utilizada). Si la máquina cliente determina que los datos de su caché están desactualizados, no podrá ya darse servicio a las solicitudes de acceso utilizando dichos datos de la caché. Será necesario almacenar en la caché una copia actualizada de los datos. Existen dos técnicas para verificar la validez de los datos almacenados en la caché.

- 1. Inicio por parte cliente.** El cliente inicia una comprobación de validez en la que contacta con el servidor y comprueba si los datos locales son coherentes con la copia maestra. La frecuencia de esta comprobación de validez es el aspecto fundamental de esta técnica y determina la semántica de coherencia resultante. Esta frecuencia puede variar, pudiendo realizarse la comprobación para cada acceso o sólo en el primer acceso a un archivo (básicamente durante la apertura de un archivo); también puede utilizarse cualquier otra frecuencia entre estos dos extremos. Todos los accesos que lleven aparejada una comprobación de validez sufrirán un cierto retraso, comparados con los accesos a los que se dé servicio inmediatamente utilizando los datos de la caché. Alternativamente, esas comprobaciones pueden iniciarse a intervalos de tiempo fijos. Dependiendo de su frecuencia, las comprobaciones de validez pueden imponer una gran carga tanto a la red como al servidor.
- 2. Inicio por parte del servidor.** El servidor registra, para cada cliente, los archivos (o partes de los mismos) que estos tienen almacenados en caché. Cuando el servidor detecta una incoherencia potencial, debe reaccionar a la misma. Una posibilidad de incoherencia se produce cuando dos clientes distintos que operan con modos conflictivos almacenan en caché un

mismo archivo. Si se implementa la semántica de UNIX (Sección 10.5.3), podemos resolver la incoherencia potencial haciendo que el servidor juegue un papel activo. El servidor deberá ser notificado cada vez que se abra un archivo, y deberá indicarse el modo deseado (lectura o escritura) para cada apertura archivo. El servidor puede entonces actuar cuando detecte que el archivo ha sido abierto simultáneamente en modos conflictivos, desactivando en ese caso el mecanismo de caché para ese archivo concreto. En la práctica, la desactivación del mecanismo de caché provoca la comutación a un modo de operación basado en el paradigma de servicio remoto.

17.3.5 Comparación entre el mecanismo de caché y el de servicio remoto

Esencialmente, la elección entre el mecanismo de caché y el de servicio remoto representa un compromiso entre las mejoras potenciales de prestaciones y la mayor simplicidad de implementación. Vamos a evaluar este compromiso indicando las ventajas y desventajas de los dos métodos.

- Cuando se utiliza un mecanismo de caché, la caché local puede gestionar de manera eficiente un número sustancial de los accesos remotos. Si aprovechamos las características de localidad de los patrones de acceso a los archivos, el mecanismo de caché resulta todavía más atractivo. De este modo, la mayoría de los accesos remotos podrán servir tan rápido como los accesos locales. Además, sólo es necesario contactar con los servidores ocasionalmente, en lugar de hacerlo para cada acceso. En consecuencia, se reduce tanto la carga del servidor como el tráfico de red, y se aumenta la escalabilidad del sistema. Por contraste, cuando se usa el método de servicio remoto, todo acceso remoto requiere una comunicación a través de la red. Las desventajas en lo que respecta a tráfico de red, carga de servidor y prestaciones resultan obvias.
- La carga total administrativa de red es menor cuando se transmiten grandes fragmentos de datos (como se hace en el caso de los mecanismos de caché) que cuando se transmite una serie de respuestas a solicitudes específicas (como sucede con el método de servicio remoto). Además, las rutinas de acceso a disco en el servidor pueden optimizarse en mayor medida si se sabe de antemano que las solicitudes siempre estarán dirigidas a extraer grandes segmentos continuos de datos, en lugar de bloques de disco aleatorios.
- El problema de la coherencia de caché es la principal desventaja del mecanismo de almacenamiento en caché. Cuando los patrones de acceso muestran pocas escrituras, los mecanismos de caché resultan claramente mejores. Sin embargo, cuando las escrituras son frecuentes, los mecanismos utilizados para resolver el problema de la coherencia implican un sobrecoste bastante sustancial en términos de prestaciones, de tráfico de red y de carga del servidor.
- Para que el mecanismo de caché pueda resultar ventajoso, debe utilizarse en máquinas que tengan un disco local o una gran cantidad de memoria principal. Los accesos remotos en las máquinas sin disco y con una pequeña cantidad de memoria deben realizarse utilizando el método del servicio remoto.
- Con los mecanismos de caché, puesto que los datos se transfieren en masa entre el servidor y el cliente, en lugar de en respuesta a las necesidades específicas de una operación de archivo, la interfaz intermáquina de bajo nivel es distinta de la interfaz de usuario que se utiliza a un nivel superior. Por contraste, el paradigma del servicio remoto es simplemente una extensión a través de la red de la interfaz con el sistema de archivos local. Por tanto, la interfaz intermáquina se corresponde con la interfaz de usuario.

17.4 Servicios con y sin memoria del estado

Existen dos técnicas para almacenar la información del lado del servidor cuando un cliente accede a archivos remotos. O bien el servidor lleva la cuenta de cada archivo al que esté accediendo

cada cliente, o simplemente se limita a proporcionar bloques a medida que los clientes los solicitan, sin ningún tipo de conocimiento de cómo se utilizan esos bloques. En el primero de los casos, el servicio proporcionado tiene *memoria del estado*, en el segundo caso, *no tiene memoria del estado*.

El escenario típico de un **servicio de archivos con memoria del estado** sería el siguiente: un cliente debe realizar una operación `open()` sobre un archivo antes de acceder a dicho archivo. El servidor extrae la información acerca del archivo desde su disco, la almacena en la memoria y proporciona al cliente un identificador de conexión que es exclusivo de ese cliente y de ese archivo concreto que se ha abierto. En términos UNIX, el servidor extrae el inodo y proporciona al cliente un descriptor de archivo que sirve como índice para una tabla interna de inodos. Este identificador se emplea para los accesos subsiguientes hasta que termina la sesión. Un servicio con memoria del estado está caracterizado como una conexión entre el cliente y el servidor durante la sesión. Durante el cierre del archivo, o mediante un mecanismo de recolección de memoria, el servidor deberá en un momento u otro reclamar el espacio de memoria principal utilizado por los clientes que ya no estén activos. El punto clave en lo que respecta a la tolerancia a fallos en un servicio con memoria del estado es que el servidor mantiene información en memoria principal acerca de sus clientes. AFS es un servicio de archivos con memoria del estado.

Un **servicio de archivos sin memoria del estado** evita la información de estado haciendo que cada solicitud sea auto contenida. En otras palabras, cada solicitud identifica el archivo y la posición dentro del archivo (para los accesos de lectura y escritura) de modo completo. El servidor no necesita mantener una tabla de archivos abiertos en memoria principal, aunque usualmente lo hace por razones de eficiencia. Además, no hay necesidad de establecer y terminar una conexión mediante operaciones `open()` y `close()`. Estas operaciones son totalmente redundantes, ya que cada operación de archivo es totalmente autónoma y no se considera parte de una sesión. El proceso cliente abriría el archivo y dicha apertura no provocaría el envío de mensajes remotos. Las lecturas y escrituras tendrían lugar como mensajes remotos (o búsquedas en caché). El cierre final por parte del cliente implicaría únicamente, de nuevo, una operación local. NFS es un servicio de archivos sin memoria del estado.

La ventaja de un servicio con memoria del estado frente a otro que no lo tenga es el incremento en las prestaciones. La información acerca de los archivos se almacena en caché dentro de la memoria principal, con lo que se puede acceder fácilmente a la misma utilizando el identificador de conexión, evitándose así accesos a disco. Además, un servidor con memoria del estado sabe si un archivo está abierto para acceso secuencial y puede, por tanto, leer de manera anticipada los bloques siguientes. Los servidores sin memoria del estado no pueden hacer esto, ya que no tienen ningún conocimiento de cuál es el propósito de las solicitudes de los clientes.

La distinción entre un servicio con memoria del estado y otro que no lo tenga resulta más evidente cuando consideramos los efectos de un fallo catastrófico que tenga lugar durante la actividad de servicio. Un servidor con memoria del estado perderá todos sus datos de estado volátiles durante el fallo. Para poder efectuar una recuperación grácil del servidor será necesario restaurar su estado, usualmente mediante un protocolo de recuperación basado en un diálogo con los clientes. Otras formas de recuperación menos gráciles requieren que se aborden las operaciones que estuvieran teniendo lugar en el momento de producirse el fallo. En el caso de que lo que fallen sean los clientes, el problema es distinto. El servidor necesitará darse cuenta de que se ha producido ese fallo para poder reclamar el espacio asignado para registrar el estado de los procesos del cliente que ha fallado. Este fenómeno se denomina en ocasiones **detección y eliminación de huérfanos**.

Los servidores sin memoria del estado no presentan estos problemas, ya que el servidor puede, después del arranque, responder a cualquier solicitud auto contenida sin ninguna dificultad. Por tanto, los efectos de los fallos de servidor y de los mecanismos correspondientes de recuperación son prácticamente inapreciables. No existe ninguna diferencia entre un servidor lento y un servidor que esté efectuando una recuperación desde el punto de vista del cliente. El cliente continuará retransmitiendo su solicitud en caso de no recibir ninguna respuesta.

La desventaja de utilizar el robusto servicio de una máquina sin memoria del estado es que los mensajes de solicitud son más largos y el procesamiento de las solicitudes más lento, ya que no se dispone de ninguna información interna para acelerar el procesamiento. Además, el servicio sin

memoria del estado impone restricciones adicionales al diseño del DFS. En primer lugar, puesto que cada solicitud identifica el archivo de destino, es necesario utilizar un esquema de nombrado de bajo nivel que sea global y uniforme para todo el sistema. Traducir los nombres remotos en nombres locales para cada solicitud hace que el procesamiento de las solicitudes sea todavía más lento. En segundo lugar, puesto que los clientes retransmiten las solicitudes relativas a operaciones de archivos, estas operaciones deben ser idempotentes; es decir, cada operación debe tener el mismo efecto y devolver la misma salida si se ejecuta varias veces consecutivamente. Los accesos de lectura y escritura autocontenidos son idempotentes, siempre y cuando utilicen un contador absoluto de bytes para indicar la posición dentro del archivo a la que quieren acceder, y que no dependan de un desplazamiento incremental (como se hace en las llamadas al sistema `read()` y `write()` de UNIX). Sin embargo, debemos tener cuidado a la hora de implementar las operaciones destructivas (como por ejemplo el borrado de un archivo) con el fin de hacerlas también idempotentes.

En algunos entornos, es absolutamente necesario utilizar un servicio con memoria del estado. Si el servidor emplea el método de validación de caché con inicio por parte del servidor, no puede proporcionar un servicio sin memoria del estado, ya que mantiene un registro de qué archivos tiene cada cliente almacenados en su caché.

NFS V4

Nuestro tratamiento de NFS ha considerado hasta el momento únicamente la versión 3 (es decir, V3) de NFS. El estándar NFS más reciente es la versión 4 (V4), y dicha versión difiere de manera importante de las versiones anteriores. El cambio más significativo es que el protocolo es ahora *con memoria del estado*, lo que significa que el servidor mantiene el estado de la sesión de cliente desde el momento que se abre el archivo remoto hasta que se cierra. Por tanto, el protocolo NFS proporciona ahora operaciones `open()` y `close()`, mientras que las versiones anteriores de NFS (que no tenían memoria del estado) no proporcionaban dichas operaciones. Además, las versiones anteriores especificaban protocolos separados para montar sistemas de archivos remotos y para bloquear archivos remotos. V4 proporciona todas estas características con un único protocolo. En particular, el protocolo `mount` ha sido eliminado, permitiendo a NFS trabajar con cortafuegos de red. El protocolo `mount` era un conocido agujero de seguridad en las implementaciones de NFS.

Además, V4 ha mejorado la capacidad de los clientes para almacenar datos de archivos en la caché local. Esta característica mejora las prestaciones del sistema de archivos distribuido, ya que los clientes pueden resolver más accesos a archivos desde la caché local en lugar de tener que acudir al servidor. V4 permite a los clientes solicitar también a los servidores que bloquen archivos. Si el servidor concede la solicitud, el cliente mantiene el bloqueo hasta liberarlo o hasta que caduque la concesión (también se permite a los clientes renovar las concesiones existentes). Tradicionalmente, los sistemas basados en UNIX proporcionan un bloqueo de archivos sugerido, mientras que los sistemas operativos Windows utilizan un bloqueo obligatorio. Para permitir que NFS funcione adecuadamente con sistemas no UNIX, V4 proporciona ahora también un mecanismo de bloqueo obligatorio. Los nuevos mecanismos de bloqueo y de almacenamiento en caché están basados en el concepto de delegación, por el que el servidor delega las responsabilidades sobre el contenido y el bloqueo de un archivo en el cliente que ha solicitado el bloqueo. Ese cliente delegado mantiene en caché esa versión del archivo, y otros clientes pueden solicitar a ese cliente delegado un acceso al contenido del archivo, hasta que el cliente delegado renuncie al bloqueo y a la delegación.

Por último, mientras que las versiones anteriores de NFS estaban basadas en el protocolo de red UDP, V4 está basada en TCP, lo que le permite ajustarse mejor a las cargas de tráfico variables experimentadas por la red. La delegación de responsabilidades en los clientes reduce la carga en el servidor y mejora la coherencia de caché.

La forma en que UNIX utiliza los descriptores de archivo y los desplazamientos implícitos es, inherentemente, un mecanismo con memoria del estado. Los servidores deben mantener tablas para establecer la correspondencia entre los descriptores de archivo y los inodos y deben almacenar el desplazamiento actual dentro de cada archivo. Este requisito es la razón por la que NFS, que emplea un servicio sin memoria del estado, no emplea descriptores de archivo e incluye un desplazamiento explícito en cada acceso.

17.5 Replicación de archivos

La replicación de archivos en diferentes máquinas dentro de un sistema de archivos distribuido constituye un útil mecanismo de redundancia para mejorar la disponibilidad. La replicación multimáquina puede aumentar también las prestaciones: seleccionar una réplica cercana para dar servicio a una solicitud de acceso da como resultado un tiempo de servicio más corto.

El requisito básico de un esquema de replicación es que las diferentes réplicas del mismo archivo residan en máquinas que sean independientes en lo respecta a los fallos, es decir, que la disponibilidad de una réplica no se vea afectada por la disponibilidad del resto de las réplicas. Este requisito obvio implica que la gestión de la replicación es, inherentemente, una actividad opaca en lo que respecta a la ubicación. Será necesario proporcionar mecanismos para poder colocar una réplica en una máquina concreta.

Resulta deseable ocultar los detalles de la replicación a ojos de los usuarios. El establecimiento de la correspondencia entre un nombre de archivo replicado y una réplica concreta es tarea del esquema de nombrado. La existencia de réplicas debe ser invisible para los niveles superiores. Sin embargo, en los niveles inferiores, es necesario distinguir unas réplicas de otras utilizando nombres de bajo nivel distintos. Otro requisito de transparencia es que se debe proporcionar mecanismos de control de la replicación en los niveles superiores. Esos mecanismos de control de la replicación incluyen poder determinar el grado de replicación y la colocación de las réplicas. En determinadas circunstancias, puede que convenga proporcionar estos detalles a los usuarios. Locus, por ejemplo, proporciona a los usuarios y a los administradores del sistema mecanismos para control del esquema de replicación. El problema principal asociado con las réplicas es su actualización. Desde el punto de vista de un usuario, todas las réplicas de un archivo denotan la misma entidad lógica, por lo que cualquier actualización en una réplica debe reflejarse en todas las réplicas restantes. De manera más precisa, será necesario preservar la semántica de coherencia relevante cuando los accesos a las réplicas se contemplen como accesos virtuales a los archivos lógicos de las réplicas. Si la coherencia no tiene una importancia fundamental, puede sacrificarse en aras de la disponibilidad y de las prestaciones. En este compromiso fundamental dentro del área de la tolerancia a fallos, la elección que hay que hacer es entre preservar la coherencia a toda costa, creando así el potencial de un bloqueo indefinido, o sacrificar la coherencia en algunas (en teoría, pocas) circunstancias de fallo catastrófico, para poder garantizar que las operaciones progresen de manera continua. Locus, por ejemplo, emplea de manera intensiva los mecanismos de replicación y sacrifica la coherencia en caso de que se produzca una partición de la red, con el fin de garantizar la disponibilidad de los accesos de lectura y escritura. Ibis utiliza una variante de la técnica de copia principal. El dominio de las correspondencias de nombres es una pareja *<identificador-réplica-principal, identificador-réplica-local>*. Si no existe ninguna réplica local, se utiliza un valor especial. Por tanto, estas correspondencias son relativas a cada máquina concreta. Si la réplica local es la principal, la pareja contendrá dos identificadores idénticos. Ibis soporta un mecanismo de replicación bajo demanda, que es una política de control de replicación automático similar a los mecanismos de almacenamiento en caché de archivos completos. Con una replicación bajo demanda, la lectura de una réplica no local, hace que esa réplica se almacene localmente en la caché, generando así una nueva réplica no principal. Las actualizaciones se realizan únicamente en la réplica principal y hacen que las demás réplicas queden invalidadas, enviándose los oportunos mensajes. No se garantiza la invalidación atómica y serializada de todas las réplicas no principales. Por ello, puede darse el caso de que una réplica obsoleta se considere válida. Para satisfacer los accesos de escritura remotos, lo que se hace es migrar la copia principal hasta la máquina solicitante.

17.6 Un ejemplo: AFS

Andrew es un entorno informático distribuido diseñado e implementado en la Universidad Carnegie Mellon. El sistema de archivos Andrew (AFS) constituye el mecanismo subyacente de compartición de información entre los clientes del entorno. Transarc Corporation asumió el desarrollo de AFS y luego fue adquirida por IBM. Desde entonces, IBM ha producido varias implementaciones comerciales de AFS. El sistema AFS fue posteriormente elegido como sistema DFS por una determinada coalición de empresas del sector, el resultado fue Transarc DFS, que forma parte del entorno informático distribuido (DCE, distributed computing environment) de la organización OSF. En el año 2000, el laboratorio Transarc Lab de IBM anunció que AFS iba a pasar a ser un producto de código abierto (denominado Open AFS), disponible bajo la licencia pública de IBM, por lo que Transarc DFS fue cancelado como producto comercial.

Open AFS está disponible para la mayor parte de las versiones comerciales de UNIX, así como para los sistemas Linux y Microsoft Windows. Muchos fabricantes de UNIX, además de Microsoft, soportan el sistema DCE y su sistema de archivos DFS, que está basado en AFS, y el trabajo continúa para hacer que DCE sea un DFS interplataforma universalmente aceptado. Puesto que AFS y Transarc DFS son muy similares, vamos a describir AFS en esta sección; cuando hagamos referencia a Transarc DFS lo indicaremos de forma explícita.

AFS trata de resolver muchos de los problemas de los sistemas DFS más simples, como NFS, y es el DFS no experimental más rico en funcionalidad. Incluye un espacio de nombres uniforme, mecanismos de compartición de archivos independientes de la ubicación, cachés del lado del cliente con coherencia de caché y un sistema de autenticación segura basado en Kerberos. También incluye mecanismos de caché del lado del servidor en la forma de réplicas, con características de alta disponibilidad gracias a la comutación automática a una réplica en caso de que el servidor de origen deje de estar disponible. Uno de las características más formidables de AFS es la escalabilidad. El sistema Andrew permite interconectar más de 5000 estaciones de trabajo. Entre AFS y Transarc DFS, hay ya centenares de implementaciones de este sistema en todo el mundo.

17.6.1 Introducción

AFS distingue entre *máquinas cliente* (en ocasiones se denominan *estaciones de trabajo*) y *máquinas servidoras* dedicadas. Los servidores y clientes ejecutaron originalmente únicamente el sistema UNIX BSD 4.2, pero AFS ha sido portado a muchos sistemas operativos. Los clientes y servidores se interconectan mediante una red de redes LAN o WAN.

A los clientes se les presenta un espacio dividido de nombres de archivo: un espacio de nombres local y un espacio de nombres compartido. Los servidores dedicados, que se denominan *Vice* por el nombre del software que ejecutan, presentan el espacio de nombres compartido a los clientes en forma de una jerarquía de archivos homogénea, idéntica y transparente a la ubicación. El espacio de nombres local es el sistema de archivos raíz de cada estación de trabajo, del que desciende el espacio de nombres compartido. Las estaciones de trabajo ejecutan el protocolo *Virtue* para comunicarse con Vice y cada una de ellas debe tener un disco local en el que almacenan su espacio de nombres local. Los servidores son, colectivamente, responsables del almacenamiento y la gestión del espacio de nombres compartido. El espacio de nombres local es pequeño, distinto para cada estación de trabajo y contiene programas del sistema esenciales para la operación autónoma de las máquinas y para mejorar las correspondientes prestaciones. También son locales los archivos temporales y los archivos que el propietario de la estación de trabajo, por razones de confidencialidad, quiera explícitamente almacenar de modo local.

Contemplados con una granularidad más fina, clientes y servidores están estructurados en *clusters* e interconectados mediante una WAN. Cada *cluster* consta de una colección de estaciones de trabajo en una LAN y por un representante de Vice denominado *servidor de cluster*, y cada *cluster* se conecta a la WAN mediante un encaminador. La descomposición en *clusters* se lleva a cabo principalmente para resolver el problema del aumento de escala. Para optimizar las prestaciones, las estaciones de trabajo deben utilizar el servidor de su propio *cluster* la mayor parte del tiempo, con el fin de hacer que las referencias a archivos entre *cluster* sean relativamente infrecuentes.

La arquitectura del sistema de archivos también está basada en consideraciones de escala. El procedimiento heurístico básico consiste en descargar trabajo de los servidores a los clientes, a la vista de la experiencia que indica que la velocidad de la CPU del servidor es el cuello de botella del sistema. De acuerdo con este principio heurístico, el mecanismo fundamental seleccionado para las operaciones remotas con archivos consiste en almacenar archivos en caché en grandes fragmentos (64 KB). Esta característica reduce la latencia de apertura de los archivos y permite que las lecturas y escrituras se dirijan a la copia almacenada en caché, sin implicar frecuentemente a los servidores.

Brevemente, he aquí algunas características adicionales del diseño de AFS:

- **Movilidad de los clientes.** Los clientes pueden acceder a cualquier archivo del espacio de nombres compartido desde cualquier estación de trabajo. El cliente puede percibir una cierta degradación inicial de las prestaciones debido al almacenamiento en caché de los archivos cuando esté accediendo a esos archivos desde una estación de trabajo distinta de la usual.
- **Seguridad.** La interfaz Vice se considera la frontera de confianza, porque ningún programa cliente se ejecuta en las máquinas Vice. Las funciones de autenticación y de transmisión segura se proporcionan como parte de un paquete de comunicaciones basado en conexión que utiliza el paradigma RPC. Después de la autenticación mutua, un servidor Vice y un cliente se comunican mediante mensajes cifrados. El cifrado se realiza mediante dispositivos hardware o (más lentamente) por software. La información de los clientes y de los grupos está almacenada en una base de datos de protección replicada en cada servidor.
- **Protección.** AFS proporciona listas de acceso para proteger los directorios, además de los bits normales de UNIX para protección de archivos. La lista de acceso puede contener información sobre aquellos usuarios que están autorizados a acceder a un directorio, además de información acerca de aquellos usuarios que *no* están autorizados a acceder a él. De este modo, resulta muy sencillo especificar que todo el mundo excepto una determinada persona puede acceder a un directorio. AFS soporta los tipos de acceso de lectura, escritura, búsqueda, inserción, administración, bloqueo y borrado.
- **Heterogeneidad.** La definición de una interfaz clara con Vice resulta clave para la integración de diversos sistemas operativos y estaciones de trabajo con distinto hardware. Para facilitar la heterogeneidad, algunos archivos del directorio *bin* local son enlaces simbólicos que apuntan a archivos ejecutables específicos de la máquina que residen en Vice.

17.6.2 Espacio de nombres compartido

El espacio de nombres compartido de AFS está formado por unidades componentes denominadas **volúmenes**. Los volúmenes son unidades componentes inusualmente pequeñas. Típicamente, están asociados con los archivos de un único cliente. En cada partición de disco residen unos cuantos volúmenes y estos pueden crecer (hasta una determinada cuota) y reducirse de tamaño. Conceptualmente, los volúmenes se combinan mediante un mecanismo similar al mecanismo de montaje de UNIX. Sin embargo, la diferencia de granularidad es significativa, ya que en UNIX sólo se puede montar una partición de disco completa (que contiene un sistema de archivo). Los volúmenes son una unidad administrativa fundamental y juegan un papel vital en la identificación y localización de los archivos individuales.

Un archivo o directorio Vice está identificado por un identificador de bajo nivel denominado **fid**. Cada entrada de directorio AFS establece la correspondencia entre un componente de nombre de ruta y un identificador *fid*. Un *fid* tiene 96 bits de longitud y tres componentes de igual longitud: un *número de volumen*, un *número de vnodo* y un *unificador*. El *número de vnodo* se utiliza como índice en una matriz que contiene los inodos de los archivos de un único volumen. El *unificador* permite reutilizar los números de *vnodo*, manteniendo compactas de este modo ciertas estructuras de datos. Los identificadores *fid* son transparentes con respecto a la ubicación; por tanto, los desplazamientos de archivos de un servidor a otro no invalidan los contenidos de los directorios almacenados en caché.

La información de ubicación se mantiene independientemente para cada volumen en una **base de datos de ubicación de volumen** replicada en cada servidor. Los clientes pueden identificar la ubicación de todos los volúmenes del sistema consultando esta base de datos. La agregación de archivos en volúmenes hace que sea posible mantener la base de datos de ubicación con un tamaño maneable.

Para equilibrar el espacio de disco disponible y el grado de utilización de los servidores, es necesario migrar los volúmenes entre particiones de disco y servidores. Cuando se envía un volumen a su nueva ubicación, se deja en su servidor original una información de re-envío temporal para que la base de datos de ubicación no tenga que actualizarse de manera síncrona. Mientras se está transfiriendo el volumen, el servidor original puede continuar gestionando las actualizaciones, que se envían posteriormente al nuevo servidor. En un determinado momento, el volumen se desactiva brevemente para poder procesar las modificaciones recientes; a continuación, el nuevo volumen pasa a estar disponible otra vez en el nuevo sitio. La operación de movimiento del volumen es atómica; si alguno de los dos servidores sufre un fallo catastrófico, se aborta la operación completa.

El sistema soporta la replicación de sólo lectura con un nivel de granularidad equivalente al de un volumen completo para los archivos ejecutables por el sistema y para los archivos raramente actualizados en los niveles superiores del espacio de nombres Vice. La base de datos de ubicación de los volúmenes especifica el servidor que contiene la única copia de lectura-escritura de un volumen y una lista de los sitios de replicación de sólo lectura.

17.6.3 Operaciones con archivos y semánticas de coherencia

El principio fundamental de arquitectura en AFS es el almacenamiento en caché de archivos completos extraídos de los servidores. Correspondientemente, una estación de trabajo cliente interacciona con los servidores Vice sólo durante la apertura y el cierre de los archivos, e incluso esta interacción no siempre es necesaria. La lectura y escritura de archivos no provoca una interacción remota (por contraste con el método de servicio remoto). Esta distinción clave tiene consecuencias profundas en lo que respecta a las prestaciones, así como en lo que se refiere a la semántica de las operaciones con archivos.

El sistema operativo de cada estación de trabajo intercepta las llamadas al sistema de archivos y las re-envía a un proceso de nivel de cliente situado en esa estación de trabajo. Este proceso, denominado *Venus*, almacena en caché los archivos de Vice cuando estos son abiertos y almacena las copias modificadas de los archivos otra vez en los servidores de donde provenían, en el momento de cerrarlos. Venus puede contactar a Vice sólo cuando se abre o se cierra un archivo; la lectura y escritura de bytes individuales de un archivo se realizan directamente en la copia almacenada en caché, puenteando a Venus. Como resultado, las escrituras que se realicen en ciertos sitios no son visibles inmediatamente en los demás.

El mecanismo de caché se aprovecha también para las futuras operaciones de apertura del archivo almacenado en caché. Venus asume que las entradas de la caché (archivos o directorios) son válidas, a menos que se le notifique lo contrario. Por tanto, Venus no necesita contactar a Vice al abrir un archivo con el fin de validar la copia almacenada en caché. El mecanismo necesario para soportar esta política, denominado de **retrollamada**, reduce enormemente el número de solicitudes de validación de caché recibidas por los servidores. Funciona de la forma siguiente: cuando un cliente almacena en caché un archivo o directorio, el servidor actualiza su información de estado para registrar este hecho. Decimos entonces que el cliente tiene una retrollamada para dicho archivo. El servidor enviará una notificación al cliente antes de autorizar a otro cliente a que modifique el archivo. En dicho caso, decimos que el servidor elimina la retrollamada de ese archivo para el primero de esos clientes. Un cliente puede utilizar un archivo de la caché para abrirlo sólo cuando el archivo tenga una retrollamada. Si un cliente cierra un archivo después de modificarlo, todos los demás clientes que tengan este archivo en caché perderán sus retrollamadas. Por tanto, cuando estos clientes abran el archivo posteriormente, tendrán que obtener la nueva versión del servidor.

El *kernel* realiza directamente la lectura y escritura de bytes en un archivo, en la copia almacenada en caché, sin la intervención de Venus. Venus vuelve a obtener el control cuando se cierra el

archivo. Si el archivo se ha modificado localmente, actualiza el archivo en el servidor apropiado. Así, las únicas ocasiones en las que Venus contacta a los servidores Vice son al abrir archivos que no se encuentren en la caché o cuya retrollamada haya sido revocada y al cerrar archivos que hayan sido modificados localmente.

Básicamente, AFS implementa una semántica de sesión. Las únicas excepciones son las operaciones de archivo distintas de las lecturas y escrituras primitivas (como los cambios de protección en el nivel de directorio), que son visibles en todos los puntos de la red inmediatamente después de completarse la operación.

A pesar del mecanismo de retrollamada, sigue existiendo una pequeña cantidad de tráfico de validación de caché, usualmente para sustituir las retrollamadas perdidas debido a fallos de máquina o de la red. Cuando se re-arranca una estación, Venus considera que todos los archivos y directorios almacenados en caché son sospechosos y genera una solicitud de validación de caché cuando se utiliza por primera vez cada una de esas entradas.

El mecanismo de retrollamada fuerza a cada servidor a mantener la información de retrollamada y a cada cliente a mantener información de validez. Si la cantidad de información de retrollamada mantenida por un servidor es excesiva, el servidor puede anular las retrollamadas y reclamar algo de espacio de almacenamiento notificando unilateralmente a los clientes que se ha revocado la validez de sus archivos almacenados en caché. Si el estado de retrollamadas mantenido por Venus se desincroniza con respecto al correspondiente estado mantenido por los servidores, pueden aparecer incoherencias.

Venus también almacena en caché el contenido de los directorios y los enlaces simbólicos para la traducción de nombres de ruta. Cada componente del nombre de ruta es extraído y se establece una retrollamada para el mismo si no está ya almacenado en caché, o si el cliente no dispone de una retrollamada para él. Venus realiza las búsquedas en los directorios extraídos localmente, utilizando los identificadores *fid*. No se re-envía ninguna solicitud desde un servidor a otro. Al finalizar el recorrido de un nombre de ruta, todos los directorios intermedios y el archivo objetivo se encontrarán en la caché, con las correspondientes retrollamadas. Las futuras llamadas de apertura para este archivo no implicarán ninguna comunicación por red, a menos que una retrollamada quedara revocada para alguno de los componentes del nombre de la ruta.

La única excepción para la política de almacenamiento en caché son las modificaciones a directorios que se hagan directamente en el servidor responsable de dicho directorio por razones de integridad. La interfaz Vice tiene una serie de operaciones bien definidas para este propósito. Venus se encarga de reflejar los cambios en su copia almacenada en caché, con el fin de evitar tener que extraer de nuevo el directorio.

17.6.4 Implementación

Los procesos cliente se comunican con el *kernel* UNIX utilizando el conjunto habitual de llamadas al sistema. El *kernel* está modificado ligeramente para detectar las referencias a los archivos Vice en las operaciones relevantes y para re-enviar las solicitudes al proceso Venus del nivel de cliente de la estación de trabajo.

Venus realiza la traducción de los nombres de ruta componente a componente como se ha descrito anteriormente. Dispone de una caché de correspondencias que asocia los volúmenes con las ubicaciones de servidor, con el fin de evitar interrogar al servidor con respecto a la ubicación de un volumen ya conocida. Si un volumen determinado no está presente en esta caché, Venus contacta a cualquier servidor con el que ya tenga una conexión, solicita la información de ubicación e introduce dicha información en la caché de correspondencias. A menos que Venus tenga ya una conexión con el servidor, establecerá una nueva conexión y utilizará ésta para extraer el archivo o directorio. El establecimiento de la conexión es necesario para propósitos de autenticación y seguridad. Cuando se encuentra un archivo objetivo y se almacena en caché, se crea una copia en el disco local. Venus devuelve entonces el control al *kernel*, que abre la copia almacenada en caché y devuelve su correspondiente descriptor al proceso cliente.

El sistema de archivos UNIX se usa como sistema de almacenamiento de bajo nivel tanto para los clientes como para los servidores AFS. La caché del cliente es un directorio local en el disco de

la estación de trabajo. Dentro de este directorio se encuentra una serie de archivos cuyos nombres son contenedores de las entradas de la caché. Tanto los procesos Venus como los procesos de servidor acceden directamente a los archivos UNIX utilizando los inodos, con el fin de evitar la costosa rutina de traducción entre nombres de ruta e inodos (*namei*). Puesto que la interfaz interna con los inodos no es visible para los procesos de nivel de cliente (tanto Venus como los procesos de servidor son procesos de nivel de cliente), se ha añadido un conjunto apropiado de llamadas al sistema adicionales. DFS utiliza su propio sistema de archivos con mecanismos de diario para mejorar las prestaciones y la fiabilidad con respecto a UFS.

Venus gestiona dos cachés separadas: una para la información de estado y otra para los datos. Emplea un sencillo algoritmo LRU (least-recently-used) para mantener acotado el tamaño de cada una de estas cachés. Cuando se elimina un archivo de la caché, Venus notifica al servidor apropiado que debe eliminar la correspondiente retrollamada para este archivo. La caché de estado se almacena en memoria virtual para permitir servir rápidamente las llamadas al sistema *stat()* (que devuelven el estado de los archivos). La caché de datos reside en el disco local, pero el mecanismo de búfer de E/S de UNIX realiza un cierto almacenamiento en caché de los bloques de disco que es completamente transparente para Venus.

Un único proceso de nivel de cliente en cada servidor de archivos se encarga de dar servicio a todas las solicitudes de archivo de los clientes. Este proceso utiliza un paquete basado en procesos ligeros con un sistema de planificación no apropiativo para dar servicio concurrentemente a muchas solicitudes de clientes. El paquete RPC está integrado con el paquete de procesos ligeros, permitiendo así que el servidor de archivos realice o sirva concurrentemente una RPC por cada proceso ligero. El paquete RPC está construido por encima de una abstracción de datagramas de bajo nivel. La transferencia completa de archivos se implementa como efecto colateral de las llamadas RPC. Existe una conexión RPC por cada cliente, pero no existe ninguna asociación a priori de los procesos ligeros con estas conexiones. En lugar de ello, un conjunto de procesos ligeros se encarga de dar servicio a las solicitudes de cliente de todas las conexiones. El uso de un único proceso servidor multihebra permite almacenar en caché las estructuras de datos necesarias para servir las solicitudes. Como desventaja, un fallo catastrófico de un único proceso de servidor tiene el efecto desastroso de paralizar a ese servidor concreto.

17.7 Resumen

Un DFS es un sistema de servicio de archivos cuyos clientes, servidores y dispositivos de almacenamiento están dispersos entre los diversos nodos de un sistema distribuido. Correspondientemente, la actividad de servicio debe llevarse a cabo a través de la red; en lugar de un único repositorio centralizado de datos hay múltiples dispositivos de almacenamiento independientes.

Idealmente, un DFS debe aparecer a ojos de sus clientes como un sistema de archivos convencional centralizado. La multiplicidad y dispersión de sus servidores y dispositivos de almacenamiento debe ser transparente, es decir, la interfaz cliente de un DFS no debe hacer distinciones entre los archivos locales y remotos. Es responsabilidad del DFS localizar los archivos y encargarse del transporte de los datos. Un DFS transparente facilita la movilidad de los clientes, desplazando el entorno del cliente al sitio donde éste haya iniciado la sesión.

Existen diversas técnicas para implementar técnicas de nombrado en un DFS. En la más simple de ellas, los archivos se nombran utilizando una combinación del nombre de host y del nombre local, lo que garantiza un nombre único en todo el sistema. Otra técnica, popularizada por NFS, proporciona un medio para asociar directorios remotos a los directorios locales, dando así la apariencia de un árbol de directorios coherente.

Las solicitudes para acceder a un archivo remoto se suelen gestionar mediante dos métodos complementarios. Con el método de servicio remoto, las solicitudes de acceso se entregan al servidor; la máquina servidora realiza los accesos y devuelve los resultados al cliente. Con el mecanismo basado en caché, si los datos necesarios para satisfacer las solicitudes de acceso no están ya almacenados en la caché, se trae una copia de los datos desde el servidor hasta el cliente. Los accesos se realizan sobre la copia almacenada en caché. La idea consiste en retener en la caché los bloques de disco a los que se ha accedido recientemente, de modo que los accesos repetidos a la

misma información puedan gestionarse de modo local, sin necesidad de ningún tráfico de red adicional. Se emplea una política de sustitución para mantener acotado el tamaño de la caché. El problema de mantener la coherencia de las copias almacenadas en caché y del archivo maestro se denomina problema de coherencia de caché.

Existen dos técnicas en lo respecta a la gestión de la información del lado del servidor. El servidor puede controlar cada archivo al que acceden los clientes, o puede simplemente proporcionar los bloques a medida que los clientes los soliciten, sin tener ningún conocimiento de cómo se van a usar esos bloques. Estas técnicas son los paradigmas de servicio con memoria del estado y sin memoria del estado.

La replicación de archivos en diferentes máquinas constituye una útil herramienta de redundancia para la mejora de la disponibilidad. La replicación multimáquina puede ser ventajosa también desde el punto de vista de las prestaciones, ya que seleccionar una réplica cercana para dar servicio a una solicitud de acceso hace que el tiempo de servicio disminuya.

AFS es sistema DFS muy rico en funcionalidad que se caracteriza por su independencia con respecto a la ubicación y su transparencia de ubicación. También impone una semántica de coherencia bastante significativa. El sistema emplea mecanismos de almacenamiento en caché y de replicación para mejorar las prestaciones.

Ejercicios

- 17.1 ¿Cuáles son los beneficios de un DFS, comparado con un sistema de archivos en un sistema centralizado?
- 17.2 ¿Cuáles de los sistemas DFS de ejemplo presentados en este capítulo gestionaría de la forma más eficiente una aplicación de base de datos multicliente de gran envergadura? Razone su respuesta.
- 17.3 Indique si AFS y NFS proporcionan lo siguiente: (a) transparencia de ubicación y (b) independencia con respecto a la ubicación.
- 17.4 ¿En qué circunstancias definiría un cliente un DFS transparente respecto a la ubicación? ¿En qué circunstancias preferiría un DFS independiente con respecto a la ubicación? Explique las razones para estas preferencias.
- 17.5 ¿Qué aspectos de un sistema distribuido seleccionaría para un sistema que se ejecutara en una red completamente fiable?
- 17.6 Considere AFS, que es un sistema de archivos distribuido con memoria del estado. ¿Qué acciones habría que llevar a cabo para recuperarse de un fallo catastrófico de servidor, con el fin de preservar la coherencia garantizada por el sistema?
- 17.7 Compare las técnicas de almacenamiento en caché de los bloques de disco localmente (en un sistema cliente) y remotamente (en un servidor).
- 17.8 AFS está diseñado para soportar un gran número de clientes. Explique tres técnicas utilizadas para hacer de AFS un sistema escalable.
- 17.9 Explique las ventajas y desventajas de realizar la traducción de nombres de rutas haciendo que el cliente envíe la ruta completa al servidor, solicitando una traducción del nombre completo de ruta del archivo.
- 17.10 ¿Cuáles son las ventajas de almacenar los objetos en memoria virtual, como hace Apollo Domain? ¿Cuáles son las desventajas?
- 17.11 Describa algunas de las diferencias fundamentales entre AFS y NFS (véase el Capítulo 11).
- 17.12 Explique si los clientes de los siguientes sistemas pueden obtener datos incoherentes o no actualizados desde el servidor de archivos y, en caso afirmativo, en qué casos podría ocurrir esto.

- a. AFS
- b. Sprite
- c. NFS

Notas bibliográficas

Pueden encontrar una serie de análisis relativos a la coherencia y al control de recuperación de los archivos replicados en Davcev y Burkhard [1985]. La gestión de los archivos replicados en un entorno UNIX fue tratada por Brereton [1986] y Purdin et al. [1987]. Wah [1984] analizó la cuestión de la colocación de los archivos en sistemas informáticos distribuidos. En Svobodova [1984] se proporciona una panorámica detallada de servidores de archivos, principalmente, centralizados.

El sistema de archivos de red (NFS) de Sun fue presentado por Callaghan [2000] y Sandberg et al. [1985]. El sistema AFS se analiza en Morris et al. [1986], Howard et al. [1988] y Satyanarayanan [1990]. Hay información disponible acerca de OpenAFS en la dirección web <http://www.openafs.org>.

En este texto no hemos tratado en detalle muchos sistemas DFS distintos que también resultan interesantes, como UNIX United, Sprite y Locus. UNIX United se describe en Brownbridge et al. [1982]. El sistema Locus se presenta en Popek y Walter [1985]. El sistema Sprite se describe en Ousterhout et al. [1988] y Nelson et al. [1988]. Los sistemas de archivos distribuidos para dispositivos de almacenamiento móviles se analizan en Kistler y Satyanarayanan [1992] y Sobti et al. [2004]. Se han llevado a cabo considerables investigaciones en el campo de los sistemas de archivos distribuidos en *cluster* (Anderson et al. [1995], Lee y Thekkath [1996], Thekkath et al. [1997] y Anderson et al. [2000]). Los sistemas de almacenamiento distribuidos para entornos de área extensa y gran escala se presentan en Dabek et al. [2001] y Kubiatowicz et al. [2000].



Coordinación distribuida

En el Capítulo 6, hemos descrito diversos mecanismos que permiten a los procesos sincronizar sus acciones. También hemos expuesto diversos esquemas para garantizar la atomicidad de una transacción que se ejecute de manera aislada o concurrentemente con otras transacciones. En el Capítulo 7, hemos descrito diversos métodos que un sistema operativo puede utilizar para resolver el problema de los interbloqueos. En este capítulo, vamos a ver cómo se pueden ampliar los mecanismos centralizados de sincronización a los entornos distribuidos. También expondremos métodos para gestionar los interbloqueos en sistemas distribuidos.

OBJETIVOS DEL CAPÍTULO

- Describir diversos métodos para conseguir la exclusión mutua en un sistema distribuido.
- Explicar cómo pueden implementarse las transacciones atómicas en un sistema distribuido.
- Mostrar cómo pueden modificarse algunos de los esquemas de control de concurrencia vistos en el Capítulo 6 para usarlos en un entorno distribuido.
- Presentar esquemas para gestionar la prevención de interbloqueos, la evitación de interbloqueos y la detección de interbloqueos en un sistema distribuido.

18.1 Ordenación de sucesos

En un sistema centralizado, siempre podemos determinar el orden en el que han tenido lugar dos sucesos, ya que el sistema tiene una única memoria común y un único reloj. Muchas aplicaciones pueden necesitar que se determine ese orden. Por ejemplo, en un esquema de asignación de recursos, especificamos que un recurso sólo puede utilizarse *después* de haber sido concedido. Sin embargo, un sistema distribuido no tiene ninguna memoria común ni ningún reloj común. Por tanto, a veces es imposible decir cuál de dos sucesos ocurrió en primer lugar. La relación *ha sucedido antes* representa sólo una ordenación parcial de los sucesos en los sistemas distribuidos. Puesto que la capacidad de definir una ordenación total resulta crucial en muchas aplicaciones, vamos a presentar un algoritmo distribuido para ampliar la relación *ha sucedido antes* con el fin de obtener una ordenación total coherente de todos los sucesos del sistema.

18.1.1 La relación Ha sucedido antes

Puesto que estamos considerando únicamente procesos secuenciales, todos los sucesos que se ejecuten dentro de un único proceso estarán completamente ordenados. También, por la ley de la causalidad, un mensaje sólo puede recibirse después de haber sido enviado. Por tanto, podemos

definir la relación *ha sucedido antes* (denotada por \rightarrow) para un conjunto de sucesos de la forma siguiente (asumiendo que el envío y la recepción de un mensaje constituye un suceso):

1. Si A y B son sucesos del mismo proceso, y A se ha ejecutado antes que B, entonces $A \rightarrow B$.
2. Si A es el suceso correspondiente al envío de un mensaje por un proceso y B es el suceso correspondiente a la recepción de dicho mensaje por parte de otro proceso, entonces $A \rightarrow B$.
3. Si $A \rightarrow B$ y $B \rightarrow C$ entonces $A \rightarrow C$.

Puesto que un suceso no puede ocurrir antes que sí mismo, la relación \rightarrow es una ordenación parcial no reflexiva.

Si dos sucesos, A y B, no están relacionados por la relación \rightarrow (es decir, A no sucedió antes que B, y B no sucedió antes que A), entonces decimos que estos dos sucesos se ejecutaron **concurrentemente**. En este caso, ninguno de los sucesos puede afectar causalmente al otro. Sin embargo, si $A \rightarrow B$, entonces es posible que el suceso A afecte causalmente al suceso B.

La mejor manera de ilustrar las definiciones de concurrencia y de la relación *ha sucedido antes* es un diagrama espacio-temporal como el de la Figura 18.1. La dirección horizontal representa el espacio (es decir, diferentes procesos), y la dirección vertical representa el tiempo. Las líneas verticales etiquetadas denotan procesos (o procesadores). Los puntos etiquetados denotan sucesos. Una línea ondulada denota un mensaje enviado de un proceso a otro. Los sucesos son concurrentes si y sólo si no existe ninguna ruta entre los mismos.

Por ejemplo, he aquí algunos de los sucesos relacionados por la relación *ha sucedido antes de la* Figura 18.1:

$$\begin{aligned} p_1 &\rightarrow q_2 \\ r_0 &\rightarrow q_4 \\ q_3 &\rightarrow r_4 \\ p_1 &\rightarrow q_4 \quad (\text{ya que } p_1 \rightarrow q_2 \text{ y } q_2 \rightarrow q_4) \} \end{aligned}$$

He aquí algunos de los sucesos concurrentes del sistema

$$\begin{aligned} q_0 \text{ y } p_2 \\ r_0 \text{ y } q_3 \\ r_0 \text{ y } p_3 \\ q_3 \text{ y } p_3 \end{aligned}$$

No podemos conocer cuál de dos sucesos concurrentes, como q_0 y p_2 , ha tenido lugar primero. Sin embargo, puesto que ninguno de los dos sucesos puede afectar al otro (no hay ninguna forma que uno de ellos sepa si el otro ya ha ocurrido), no es importante cuál de ellos haya tenido lugar primero. Sólo es importante que cualesquiera procesos que se preocupen acerca del orden de cualesquiera dos sucesos acuerden un cierto orden arbitrario.

18.1.2 Implementación

Para determinar que un suceso A ha tenido lugar antes que un suceso B, necesitamos un reloj común o un conjunto de relojes perfectamente sincronizado. Puesto que un sistema distribuido no está disponible ninguna de estas dos opciones, tenemos que definir la relación *ha sucedido antes sin utilizar relojes físicos*.

Con cada suceso del sistema asociemos una **marca temporal**. Podemos entonces definir el requisito de **ordenación global**: para cada pareja de sucesos A y B, si $A \rightarrow B$, entonces la marca temporal de A es inferior a la marca temporal de B (más adelante veremos que el enunciado inverso no tiene porqué ser cierto).

¿Cómo imponemos el requisito de ordenación global en un entorno distribuido? Definimos dentro de *cada* proceso P_i un **reloj lógico**, LC_i . El reloj lógico puede implementarse como un simple contador que se incrementa cada dos sucesos sucesivos que se ejecuten dentro de un proceso. Puesto que el reloj lógico tiene un valor **monótonamente creciente**, asignará un número único a cada suceso, y si un suceso A tiene lugar antes que el suceso B en el proceso P_i , entonces $LC_i(A)$

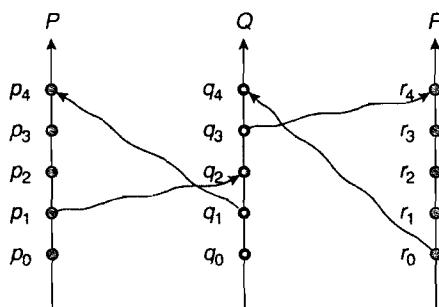


Figura 18.1 Tiempo relativo para tres procesos concurrentes.

$< LC_i(B)$. La marca temporal para un suceso es el valor del reloj lógico para dicho suceso. Este esquema garantiza que para cualesquiera dos sucesos dentro del mismo proceso se cumpla el requisito de ordenación global.

Lamentablemente, este esquema no garantiza que se cumpla el requisito de ordenación global entre varios procesos. Para ilustrar el problema, considere dos procesos P_1 y P_2 que se comunican entre sí. Suponga que P_1 envía un mensaje a P_2 (suceso A) con $LC_1(A) = 200$, y que P_2 recibe el mensaje (suceso B) con $LC_2(B) = 195$ (porque el procesador de P_2 es más lento que el procesador de P_1 , y los tics de su reloj lógico van más lentos). Esta situación viola nuestro requisito, ya que $A \rightarrow B$, pero la marca temporal de A es superior que la marca temporal de B.

Para resolver esta dificultad, obligamos a cada proceso a hacer avanzar su reloj lógico cuando reciba un mensaje cuya marca temporal sea superior al valor actual de su propio reloj lógico. En particular, si el proceso P_i recibe un mensaje (suceso B) con marca temporal t y $LC_i(B) \leq t$, entonces deberá avanzar su reloj de modo que $LC_i(B) = t + 1$. Así, en nuestro ejemplo, cuando P_2 recibe el mensaje de P_1 , avanzará su reloj lógico de modo que $LC_2(B) = 201$.

Finalmente, para conseguir una ordenación total, sólo necesitamos observar que, con nuestro esquema de ordenación de marcas temporales, si las marcas temporales de dos sucesos, A y B, son iguales, entonces los sucesos son concurrentes. En este caso, podemos utilizar los números de identidad de los procesos para romper los empates y crear una ordenación total. El uso de marcas temporales se analiza más en detalle en la Sección 18.4.2.

18.2 Exclusión mutua

En esta sección, vamos a presentar una serie de algoritmos diferentes para implementar la exclusión mutua en un entorno distribuido. Vamos a suponer que el sistema consta de n procesos, cada uno de los cuales reside en un procesador diferente. Para simplificar nuestra exposición, vamos a suponer que los procesos están numerados de manera única desde 1 hasta n , y que existe una correspondencia uno-a-uno entre procesos y procesadores (es decir, cada proceso tiene su propio procesador).

18.2.1 Enfoque centralizado

En un enfoque centralizado para la implementación del mecanismo de exclusión mutua, se elige uno de los procesos del sistema para que coordine la entrada en la sección crítica. Cada proceso que desee invocar la exclusión mutua enviará un mensaje de *solicitud* al coordinador. Cuando el proceso reciba un mensaje de *respuesta* del coordinador, podrá entrar en su sección crítica. Después de salir de su sección crítica, el proceso enviará un mensaje de *liberación* al coordinador y continuará con su ejecución.

Al recibir un mensaje de *solicitud*, el coordinador comprueba si hay algún otro proceso en su sección crítica. Si no hay ningún otro proceso en su sección crítica, el coordinador devuelve inmediatamente un mensaje de *respuesta*. En caso contrario, se pone la solicitud en cola. Cuando el coordinador recibe un mensaje de *liberación*, extrae uno de los mensajes de *solicitud* de la cola (de

acuerdo con algún algoritmo de planificación) y envía un mensaje de *respuesta* al proceso solicitante.

Debería quedar claro que este algoritmo garantiza la exclusión mutua. Además, si la política de planificación del coordinador es equitativa (como por ejemplo una planificación del tipo FCFS, donde el primero en llegar será el primero en ser servido) no puede producirse ningún problema de muerte por inanición. Este esquema requiere tres mensajes por cada entrada en una sección crítica: una *solicitud*, una *respuesta* y una *liberación*.

Si falla el proceso coordinador, entonces un nuevo proceso deberá tomar su lugar. En la Sección 18.6, describiremos algunos algoritmos para elegir un nuevo coordinador. Una vez elegido el nuevo coordinador, éste deberá sondear a todos los procesos del sistema para reconstruir la cola de *solicitudes*. Una vez construida la cola, el procesamiento puede continuar.

18.2.2 Enfoque completamente distribuido

Si queremos distribuir el proceso de toma de decisiones por todo el sistema, entonces la solución es bastante más complicada. Una de las técnicas, que se describe a continuación, utiliza un algoritmo basado en el esquema de ordenación de sucesos descrito en la Sección 18.1.

Cuando un proceso P_i quiere entrar en su sección crítica, genera una nueva marca temporal, TS , y envía el mensaje *solicitud* (P_i, TS) a todos los procesos del sistema (incluyendo el mismo). Al recibir un mensaje de *solicitud*, un proceso puede responder inmediatamente (es decir, enviar un mensaje de *respuesta* a P_i), o puede diferir el envío de la respuesta (por ejemplo, porque ya se encuentra dentro de su sección crítica). Un proceso que haya recibido un mensaje de *respuesta* de todos los demás procesos del sistema podrá entrar en su sección crítica, poniendo en cola las solicitudes entrantes y diferiendo la respuesta a las mismas. Después de salir de su sección crítica, el proceso enviará mensajes de *respuesta* a todas las solicitudes diferidas.

La decisión de si el proceso P_i debe responder inmediatamente a un mensaje de *solicitud* (P_j, TS) o diferir la respuesta está basada entre tres factores:

1. Si el proceso P_i se encuentra en su sección crítica, entonces diferirá su respuesta a P_j .
2. Si el proceso P_i no quiere entrar en su sección crítica, entonces responderá de forma inmediata a P_j .
3. Si el proceso P_i quiere entrar en su sección crítica pero todavía no lo ha hecho, entonces comparará la marca temporal de su propia *solicitud* con la marca temporal de la solicitud entrante realizada por el P_j . Si la marca temporal de su propia *solicitud* es mayor que la de la solicitud entrante, entonces enviará inmediatamente una respuesta a P_j (porque P_j realizó la solicitud primero). En caso contrario, diferirá la respuesta.

El comportamiento de este algoritmo exhibe las siguientes características ventajosas:

- Se consigue la exclusión mutua.
- Se garantiza que el algoritmo está libre de interbloqueos.
- Se garantiza que el algoritmo está libre del fenómeno de muerte por inanición, ya que la entrada en la sección crítica se planifica mediante la ordenación por marcas temporales. La ordenación por marcas temporales garantiza que se dé servicio a los servicios según un orden FCFS.
- El número de mensajes por cada entrada en la sección crítica es $2 \times (n - 1)$. Este número es el número mínimo de mensajes requeridos por cada entrada en una sección crítica cuando los procesos actúan de forma independiente y concurrente.

Para ilustrar cómo funciona el algoritmo, vamos a considerar un sistema compuesto por los procesos P_1 , P_2 y P_3 . Supongamos que los procesos P_1 y P_3 desean entrar en sus secciones críticas. El proceso P_1 envía entonces un mensaje de *solicitud* (P_1 , marca temporal = 10) a los procesos P_2 y P_3 , mientras que el proceso P_3 envía un mensaje *solicitud* (P_3 , marca temporal = 4) a los procesos P_1 y P_2 . Las marcas temporales 4 y 10 han sido obtenidas a partir de los relojes lógicos descritos en

la Sección 18.1. Cuando el P_2 recibe estos mensajes de *solicitud*, responde inmediatamente. Cuando el proceso P_1 recibe la *solicitud* del proceso P_3 , responde inmediatamente, ya que la marca temporal (10) de su propio mensaje de *solicitud* es mayor que la marca temporal (4) del proceso P_3 . Cuando el proceso P_3 recibe el mensaje de *solicitud* del proceso P_1 , difiere su respuesta, ya que la marca temporal (4) de su propio mensaje de *solicitud* es menor que la marca temporal (10) del mensaje procedente del proceso P_1 . Al recibir las respuestas de ambos procesos P_1 y P_2 , el proceso P_3 puede entrar en su sección crítica. Después de salir de su sección crítica, el proceso P_3 envía una respuesta al proceso P_1 , el cual puede entonces entrar en su propia sección crítica.

Sin embargo, puesto que este esquema requiere la participación de todos los procesos del sistema, presenta tres consecuencias indeseables:

1. Los procesos necesitan conocer la identidad de todos los demás procesos del sistema. Cuando un nuevo proceso se une al grupo de procesos que participan en el algoritmo de exclusión mutua, será necesario llevar a cabo las siguientes acciones:

- a. El proceso debe recibir los nombres de los restantes procesos del grupo.
 - b. El nombre del nuevo proceso debe distribuirse a todos los demás procesos del grupo.

Esta tarea no es tan trivial como puede parecer, ya que algunos mensajes de *solicitud* y de *respuesta* pueden estar ya circulando en el sistema en el momento que el nuevo proceso se une al grupo. El lector interesado puede encontrar más detalles en la documentación referenciada en la Notas bibliográficas.

2. Si uno de los procesos falla, entonces todo el esquema se colapsa. Podemos resolver esta dificultad monitorizando continuamente el estado de todos los procesos del sistema. Si un proceso falla, se envía una notificación a todos los demás procesos, para que ya no manden más mensajes de *solicitud* al proceso fallido. Cuando un proceso se recupera, debe iniciar el procedimiento que le permita volver a unirse al grupo.
 3. Los procesos que no hayan entrado en su sección crítica deben efectuar pausas frecuentes para asegurar a otros procesos que pretendan entrar en la sección crítica.

Debido a estas dificultades, este protocolo está mejor adaptado a conjuntos pequeños y establebles de procesos cooperativos.

18.2.3 Técnica basada en paso de testigo

Otro método de proporcionar la exclusión mutua consiste en circular un testigo entre los procesos del sistema. Un **testigo** es un tipo especial de mensaje que se pasa por todo el sistema. La posesión del testigo permite al poseedor entrar en la sección crítica. Puesto que sólo hay un testigo, sólo podrá haber un proceso en su sección crítica en cada momento.

Supongamos que los procesos del sistema están organizados *lógicamente* según una **estructura en anillo**. La propia red física de comunicaciones no tiene que ser un anillo. Siempre que los procesos estén conectados entre sí, resulta posible implementar ese anillo lógico. Para implementar la exclusión mutua, lo que hacemos es pasar el testigo alrededor del anillo. Cuando un proceso recibe el testigo, puede entrar en su sección crítica, manteniendo el testigo. Después de que el proceso salga de su sección crítica, el testigo vuelve a hacer circular. Si el proceso que recibe el testigo no quiere entrar en su sección crítica, pasará el testigo a su vecino. Este esquema es similar al algoritmo 1 del Capítulo 6, pero se utiliza un testigo en lugar de una variable compartida.

Si el anillo es unidireccional, está garantizado que no aparezcan problemas de muerte por inanición. El número de mensajes necesarios para implementar la exclusión mutua puede variar desde un mensaje por entrada, en el caso de un alto grado de contienda (es decir, cuando todos los procesos quieren entrar en su sección crítica), hasta un número infinito de mensajes, en el caso de que el grado de contienda sea bajo (es decir, ningún proceso quiere entrar en su sección crítica).

Debemos tomar en consideración dos tipos de fallo. En primer lugar, si se pierde el testigo, es necesario convocar una elección para generar un nuevo testigo. En segundo lugar, si un proceso

falla, es necesario establecer un nuevo anillo lógico. En la Sección 18.6 presentamos un algoritmo de elección, existiendo otros algoritmos posibles. El desarrollo de un algoritmo para reconstruir el anillo se deja como tarea para el lector en el Ejercicio 18.9.

18.3 Atomicidad

En el Capítulo 6, hemos presentado el concepto de transacción atómica, que es una unidad de programa que debe ejecutarse **atómicamente**. Es decir, o bien todas las operaciones asociadas con ella se ejecutan hasta completarse, o bien no se lleva a cabo ninguna. Cuando estamos tratando con sistemas distribuidos, garantizar la atomicidad de una transacción resulta mucho más complicado que en un sistema centralizado. Esta dificultad surge porque puede haber varios sitios participando en la ejecución de una misma transacción. El fallo de uno de estos sitios, o el fallo de un enlace de comunicaciones que interconecte los sitios, puede dar como resultado un procesamiento erróneo.

Garantizar que la ejecución de las transacciones en el sistema distribuido preserve la atomicidad es función del **coordinador de la transacción**. Cada sitio tiene su propio coordinador local de transacción, que es responsable de coordinar la ejecución de todas las transacciones iniciadas en dicho sitio. Para cada transacción, el coordinador es responsable de lo siguiente:

- Iniciar la ejecución de la transacción.
- Descomponer la transacción en una serie de subtransacciones y distribuirlas a los sitios apropiados para su ejecución.
- Coordinar la terminación de la transacción, que puede dar como resultado que las transacciones se confirmen en todos los sitios o se aborten en todos los sitios.

Asumimos que cada sitio local mantiene un registro local para propósitos de recuperación.

18.3.1 El protocolo de confirmación en dos fases

Para garantizar la atomicidad, todos los sitios en los que se haya ejecutado una transacción T deben acordar el resultado final de la ejecución. T debe confirmarse en todos los sitios o abortarse en todos los sitios. Para garantizar esta propiedad, el coordinador de la transacción T debe ejecutar un **protocolo de confirmación**. Entre los protocolos de confirmación más simples y más ampliamente utilizados se encuentra el **protocolo de confirmación en dos fases (2PC)**, que describiremos a continuación.

Sea T una transacción iniciada en el sitio S_i y sea C_i el coordinador de la transacción en S_i . Cuando T completa su ejecución, es decir, cuando todos los sitios en los que T se ha ejecutado informen a C_i de que T se ha completado, entonces C_i dará comienzo al protocolo 2PC.

- **Fase 1.** C_i añade la entrada $\langle \text{prepare } T \rangle$ al registro y fuerza la escritura de esa entrada en almacenamiento estable. A continuación envía un mensaje $\text{prepare } (T)$ a todos los sitios en que T se haya ejecutado. Al recibir el mensaje, el gestor de la transacción en dicho sitio determina si está preparado para confirmar su parte correspondiente de T . Si la respuesta es *no*, añade una entrada $\langle \text{no } T \rangle$ al registro y responde enviando un mensaje $\text{abort } (T)$ a C_i . Si la respuesta es *yes*, añade una entrada $\langle \text{ready } T \rangle$ al registro y escribe en almacenamiento estable todas las entradas de registro correspondientes a T . El gestor de la transacción responde entonces con un mensaje $\text{ready } (T)$ a C_i .
- **Fase 2.** Cuando C_i ha recibido las respuestas al mensaje $\text{prepare } (T)$ de todos los sitios, o cuando ha transcurrido un intervalo de tiempo previamente especificado desde que envió el mensaje $\text{prepare } (T)$, C_i puede determinar si la transacción T puede ser confirmada o abortada. La transacción T podrá ser confirmada si C_i ha recibido un mensaje $\text{ready } (T)$ de todos los sitios participantes. En caso contrario, la transacción T deberá ser abortada. Dependiendo del veredicto, se añade una entrada $\langle \text{commit } T \rangle$ o $\langle \text{abort } T \rangle$ al registro y se escribe dicha entrada en almacenamiento estable. En este punto, el destino de la transacción ya ha sido

fijado. A continuación, el coordinador enviará un mensaje *commit* (T) o *abort* (T) a todos los sitios participantes. Cuando un sitio reciba este mensaje, lo almacenará en el registro.

Un sitio en el que T se haya ejecutado puede abortar T incondicionalmente en cualquier momento antes de enviar el mensaje *ready* (T) al coordinador. El mensaje *ready* (T) es, en efecto, una promesa efectuada por un sitio para cumplir la orden del coordinador de confirmar T o abortar T . Un sitio sólo puede hacer esa promesa cuando la información esté guardada en almacenamiento estable. En caso contrario, si el sitio sufriera un fallo catastrófico después de informar de que estaba preparado para confirmar T , podría no ser capaz de cumplir con su promesa.

Puesto que se requiere unanimidad para confirmar una transacción, el destino de T estará fijado tan pronto como uno de los sitios responda con el mensaje *abort* (T). Observe que el sitio coordinador S_i puede decidir unilateralmente abortar T , ya que es uno de los sitios donde T se ha ejecutado. El veredicto final concerniente a T se determina en el momento en que el coordinador escribe dicho veredicto (confirmar o abortar) en el registro y lo guarda en almacenamiento estable. En algunas implementaciones del protocolo 2PC, los sitios envían un mensaje de *acknowledge* (T) al coordinador al final de la segunda fase del protocolo. Cuando el coordinador ha recibido el mensaje de *acknowledge* (T) de todos los sitios, añade la entrada $\langle\text{complete } T\rangle$ al registro.

18.3.2 Gestión de fallos en 2PC

Ahora vamos a examinar en detalle cómo responde 2PC a distintos tipos de fallos. Como veremos, una de las principales desventajas del protocolo 2PC es que el fallo del coordinador puede dar lugar a un bloqueo y la decisión de si confirmar o abortar T puede tener que ser pospuesta hasta que C_i se recupere.

18.3.2.1 Fallo de un sitio participante

Cuando un sitio participante S_k se recupera de un fallo, debe examinar su registro para determinar el destino de aquellas transacciones que estuvieran en mitad de su ejecución en el momento de producirse el fallo. Sea T una de tales transacciones. ¿Cómo puede tratar S_k la transacción T ? Vamos a considerar cada una de las posibles alternativas:

- El registro contiene una entrada $\langle\text{commit } T\rangle$. En este caso, el sitio ejecuta *redo*(T) para rehacer la transacción.
- El registro contiene una entrada $\langle\text{abort } T\rangle$. En este caso, el sitio ejecuta *undo*(T) para deshacer la transacción.
- El registro contiene una entrada $\langle\text{ready } T\rangle$. En este caso, el sitio debe consultar a C_i para determinar el destino que tuvo T . Si C_i está funcionando, notificará a S_k si la transacción T se confirmó o se abortó. En el primero de los casos, el sitio ejecutará *redo*(T); en el segundo de los casos, ejecutará *undo*(T). Si C_i no está funcionando, S_k debe tratar de averiguar el destino de T consultando a otros sitios. Para ello, envía un mensaje *query-status* (T) a todos los sitios del sistema. Al recibir tal mensaje, un sitio debe consultar su registro para determinar si T se ha ejecutado allí y, en caso afirmativo, si se confirmó o abortó. Entonces, enviará una notificación S_k acerca del resultado. Si no hay ningún sitio que tenga la información apropiada, (es decir, que pueda informar de si T se confirmó o abortó), entonces S_k no puede abortar ni confirmar T . La decisión concerniente a T se pospone hasta que S_k pueda obtener la información necesaria. En este caso, S_k debe periódicamente volver a enviar el mensaje *query-status* (T) a los otros sitios, y continuar haciéndolo hasta que se recupere un sitio que contenga la información necesaria. El sitio en el que reside C_i siempre tiene esa información necesaria.
- El registro no contiene ninguna entrada de control (abort, commit, ready) concerniente a T . La ausencia de entradas de control implica que S_k sufrió un fallo antes de responder al mensaje *prepare* (T) de C_i . Puesto que el fallo de S_k quiere decir que no ha podido enviar la res-

puesta, C_i deberá, según nuestro algoritmo, haber abortado T . Por tanto, S_k deberá ejecutar $\text{undo}(T)$.

18.3.2.2 Fallo del coordinador

Si falla el coordinador en mitad de la ejecución del protocolo de confirmación para la transacción T , entonces los sitios participantes deben decidir el destino de T . Veremos que, en cierto casos, los sitios participantes no pueden decidir si hay que confirmar o abortar T y, por tanto, dichos sitios deberán esperar a que se recupere el coordinador fallido.

- Si un sitio activo contiene una entrada $\langle\text{commit } T\rangle$ en su registro, entonces habrá que confirmar T .
- Si un sitio activo contiene una entrada $\langle\text{abort } T\rangle$ en su registro, entonces habrá que abortar T .
- Si algún sitio activo *no* contiene una entrada $\langle\text{ready } T\rangle$ en su registro, entonces el coordinador fallido C_i no puede haber decidido confirmar T . Podemos extraer esta conclusión porque un sitio que no disponga de una entrada $\langle\text{ready } T\rangle$ en su registro no puede haber enviado un mensaje $\text{ready}(T)$ a C_i . Sin embargo, el coordinador puede haber decidido abortar T . En lugar de esperar a que C_i se recupere, resulta preferible en este caso abortar T .
- Si no se cumple ninguno de los casos anteriores, entonces todos los sitios activos tienen que tener una entrada $\langle\text{ready } T\rangle$ en sus registros, pero ninguna otra entrada adicional de control (como por ejemplo $\langle\text{abort } T\rangle$ o $\langle\text{commit } T\rangle$). Puesto que el coordinador ha fallado, es imposible determinar si se ha tomado una decisión o, en caso afirmativo, qué decisión ha sido, y será imposible efectuar esa determinación hasta que el coordinador se recupere. Por tanto, los sitios activos deberán esperar hasta que C_i se recupere. Mientras el destino de T siga siendo dudoso, T puede continuar reteniendo recursos del sistema. Por ejemplo, si se han utilizado mecanismos de bloqueo, T puede tener bloqueados determinados datos en una serie de sitios activos. Dicha situación resulta indeseable, porque pueden pasar horas o días antes de que C_i vuelva a estar activo. Durante este tiempo, otras transacciones pueden verse forzadas a esperar a que termine T . Como resultado, los datos no sólo no estarán disponibles en el sitio fallido (C_i) sino que tampoco en los sitios activos. La cantidad de datos no disponibles se incrementa a medida que crece el tiempo de parada de C_i . Esta situación se denomina problema del *bloqueo*, porque T está bloqueada pendiente de la recuperación del sitio C_i .

18.3.2.3 Fallo de la red

Cuando falla un enlace, los mensajes que están siendo encaminados a través del enlace no llegarán intactos a sus destinos. Desde el punto de vista de los sitios conectados a través de ese enlace, los otros sitios parecerán haber fallado. Por tanto, nuestros esquemas anteriores se aplican también a este caso.

Cuando fallan diversos enlaces, la red puede quedar dividida. En este caso, existen dos posibilidades. El coordinador y todos los participantes pueden quedar incluidos en una de las particiones; en este caso, el fallo no tiene ningún efecto sobre el protocolo de confirmación. Alternativamente, el coordinador y los participantes pueden quedar distribuidos entre varias particiones; en este caso, se pierden los mensajes entre algunos participantes y el coordinador, reduciendo el caso al de un fallo de enlace.

18.4 Control de concurrencia

A continuación, vamos a analizar la cuestión del control de concurrencia. En esta sección, veremos cómo pueden modificarse algunos de los esquemas de control de concurrencia expuestos en el Capítulo 6 para utilizarlos en un entorno distribuido.

El gestor de transacciones de un sistema de base de datos distribuido gestiona la ejecución de aquellas transacciones (o subtransacciones) que acceden a datos almacenados en un sitio local. Cada una de esas transacciones puede ser una transacción local (es decir, una transacción que sólo se ejecuta en un sitio) o parte de una transacción global (es decir, una transacción que se ejecuta en varios sitios). Cada gestor de transacciones responsable de mantener un registro para propósitos de recuperación y de participar en un esquema apropiado de control de concurrencia para coordinar la ejecución concurrente de las transacciones que se estén ejecutando en dicho sitio. Como veremos, es necesario modificar los esquemas de concurrencia descritos en Capítulo 6 para adaptarlos al caso de la distribución de transacciones.

18.4.1 Protocolos de bloqueo

Los protocolos de bloqueo en dos fases descritos en el Capítulo 6 pueden utilizarse en un entorno distribuido. El único cambio necesario es en la forma de implementar el gestor de bloqueo. Aquí, vamos a presentar varios posibles esquemas. El primero de ellos trata con el caso en el que no se permita ninguna replicación de datos. Los otros esquemas se aplican al caso más general en el que los datos pueden estar replicados en varios sitios. Como en el Capítulo 6, vamos a presuponer la existencia de los **modos de bloqueo exclusivo y compartido**.

18.4.1.1 Esquema no replicado

Si no hay ningún dato replicado en el sistema, entonces podemos aplicar de la forma siguiente los esquemas de bloqueo descritos en la Sección 6.9: cada sitio mantiene un gestor de bloqueos local cuya función consiste en administrar las solicitudes de bloqueo y desbloqueo para los elementos de datos almacenados en ese sitio. Cuando una transacción desea bloquear el elemento de datos Q en el sitio S_i , simplemente envía un mensaje al gestor de bloqueos del sitio S_i solicitando un determinado cerrojo (o bloqueo) en un modo de bloqueo particular. Si el elemento de datos Q está bloqueado en un modo incompatible, entonces se difiere la respuesta a la solicitud hasta que dicha solicitud puede ser aprobada. Una vez que se ha determinado que la solicitud de bloqueo puede concederse, el gestor de bloqueos devuelve un mensaje al iniciador indicando que se ha concedido la solicitud de bloqueo.

Este esquema presenta la ventaja de que su implementación es muy simple. Requiere dos transferencias de mensajes para gestionar las solicitudes de bloqueo y una transferencia de mensaje para gestionar las solicitudes de desbloqueo. Sin embargo, la gestión de los interbloqueos es más compleja. Puesto que las solicitudes de bloqueo y desbloqueo ya no se realizan en un único sitio, deberán modificarse los diferentes algoritmos de gestión de interbloqueos expuestos en el Capítulo 7; estas modificaciones se detallan en la Sección 18.5

18.4.1.2 Técnica basada en un único coordinador

En los sistemas que permiten la replicación de datos pueden utilizarse diversos esquemas de control de concurrencia. Con la técnica basada en un único coordinador, el sistema mantiene un único gestor de bloqueos que reside en un único sitio elegido, como por ejemplo S_i . Todas las solicitudes de bloqueo y de desbloqueo se dirigen al sitio S_i . Cuando una transacción necesita bloquear un elemento de datos, envía una solicitud de bloqueo a S_i . El gestor de bloqueos determina si puede concederse el bloqueo de forma inmediata. En caso afirmativo, envía un mensaje en ese sentido al sitio en el que se inició la solicitud de bloqueo. En caso contrario, se retarda la respuesta a la solicitud hasta que ésta pueda ser concedida; cuando se la pueda conceder, se envía un mensaje al sitio en que se inició la solicitud de bloqueo. La transacción puede leer el elemento de datos en *cualquiera* de los sitios en los que resida una réplica de dicho elemento de datos. En el caso de una operación `write`, deben implicarse en la escritura todos los sitios en los que resida una réplica del elemento de datos.

Este esquema tiene las siguientes ventajas:

- **Implementación simple.** Este esquema requiere dos mensajes para gestionar las solicitudes de bloqueo y un mensaje para gestionar las solicitudes de desbloqueo.
- **Gestión simple de los interbloqueos.** Puesto que todas las solicitudes de bloqueo y de desbloqueo se realizan en un mismo sitio, pueden aplicarse directamente en este entorno los algoritmos de gestión de interbloqueos expuestos en el Capítulo 7.

Entre las desventajas de este esquema se encuentran las siguientes:

- **Cuello de botella.** El sitio S_i se convierte en un cuello de botella, ya que hay que procesar en él todas las solicitudes.
- **Vulnerabilidad.** Si el sitio S_i falla, se pierde el controlador de concurrencia. En ese caso, o bien se detiene el procesamiento, o bien se utiliza un esquema de recuperación.

Podemos conseguir un compromiso entre estas ventajas y desventajas utilizando **técnica basada en coordinadores múltiples**, en la que la función de gestión de los bloqueos se distribuye entre varios sitios. Cada gestor de bloqueos administra las solicitudes de bloqueo y de desbloqueo para un subconjunto de elementos de datos, y esos gestores de bloqueos residen en sitios distintos. Esta distribución reduce el grado en que los coordinadores pueden llegar a convertirse en cuellos de botella, pero complica la gestión de interbloqueos, ya que las solicitudes de bloqueo y de desbloqueo no se realizan en un único sitio.

18.4.1.3 Protocolo de mayoría

El protocolo de mayoría es una modificación del esquema para datos no replicado que hemos presentado anteriormente. El sistema mantiene un gestor de bloqueos en cada sitio. Cada gestor controla los bloqueos para todos los datos o réplicas de datos almacenados en ese sitio. Cuando una transacción quiere bloquear un elemento de datos Q que esté replicado en n sitios diferentes deberá enviar una solicitud de bloqueo a más de la mitad de los n sitios en los que Q esté almacenado. Cada gestor de bloqueos determina si puede concederse el bloqueo inmediatamente (en lo que a él respecta). Como antes, la respuesta se retarda hasta que la solicitud pueda concederse. La transacción no operará con el elemento de datos Q hasta que haya obtenido con éxito un bloqueo sobre una mayoría de las réplicas del elemento de datos.

Este esquema trata con los datos replicados de una manera descentralizada, evitando así las desventajas del control central. Sin embargo, también tiene sus desventajas:

- **Implementación.** El protocolo de mayoría es más complicado de implementar que los esquemas anteriores. Requiere $2(n/2 + 1)$ mensajes para gestionar las solicitudes de bloqueo y $(n/2 + 1)$ mensajes para gestionar las solicitudes de desbloqueo.
- **Gestión de los interbloqueos.** Puesto que las solicitudes de bloqueo y de desbloqueo no se realizan en un mismo sitio, es necesario modificar los algoritmos de gestión de interbloqueos (Sección 18.5). Además, puede producirse un interbloqueo incluso si sólo está bloqueado un único elemento de datos. Como ilustración, considere un sistema con cuatro sitios y con un esquema de replicación total. Suponga que las transacciones T_1 y T_2 quieren bloquear el elemento de datos Q en modo exclusivo. La transacción T_1 puede conseguir bloquear Q en los sitios S_1 y S_3 , mientras que la transacción T_2 puede conseguir bloquear Q en los sitios S_2 y S_4 . Entonces, cada una de las dos transacciones deberá esperar para adquirir el tercer bloqueo y, por tanto, se habrá producido un interbloqueo.

18.4.1.4 Protocolo preferencial

El protocolo preferencial es similar al protocolo de mayoría. La diferencia es que las solicitudes de bloqueos compartidos tienen un tratamiento más favorable que las solicitudes de bloques exclusivos. El sistema mantiene un gestor de bloqueos en cada sitio. Cada gestor se encarga de gestionar los bloqueos para todos los elementos de datos almacenados en ese sitio. Los bloqueos compartidos y exclusivos se gestionan de forma distinta.

- **Bloqueos compartidos.** Cuando una transacción necesita bloquear el elemento de datos Q , simplemente solicita un bloqueo sobre Q al gestor de bloqueos de uno de los sitios que contenga una réplica de Q .
- **Bloqueos exclusivos.** Cuando una transacción necesita bloquear el elemento de datos Q , solicita un bloqueo sobre Q al gestor de bloqueos de cada uno de los sitios que contenga una réplica de Q .

Como antes, la respuesta a la solicitud se retarda hasta que puede ser concedida.

Este esquema tiene la ventaja de imponer una menor carga adicional a las operaciones de lectura que el protocolo de mayoría. Esta ventaja es especialmente significativa en esos casos comunes en los que la frecuencia de las lecturas es mucho mayor que la frecuencia de las escrituras. Sin embargo, el coste adicional de las escrituras constituye una de las desventajas de este esquema. Además, el protocolo preferencial comparte con el protocolo de mayoría la desventaja de que la gestión de los interbloqueos resulta muy compleja.

18.4.1.5 Copia principal

Otra alternativa más consiste en seleccionar una de las réplicas como copia principal. Así, para cada elemento de datos Q , la copia principal de Q debe residir en un solo sitio que denominaremos *sitio principal de Q* . Cuando una transacción necesita bloquear un elemento de datos Q , solicita un bloqueo en el sitio principal de Q . Como antes, la respuesta a la solicitud se retarda hasta que la solicitud pueda concederse.

Este esquema nos permite gestionar el control de concurrencia de los datos replicados de forma bastante similar al caso de los datos no replicados. La implementación del método resulta simple. Sin embargo, si falla el sitio principal de Q , Q será inaccesible aún cuando puedan ser accesibles otros sitios que contenga una réplica.

18.4.2 Marcas temporales

La idea principal que subyace al esquema de marcas temporales explicado en la Sección 6.9 es que a cada transacción se le da una marca temporal *única*, que se utiliza para decidir el orden de serialización. Nuestra primera tarea, entonces, a la hora de generalizar el esquema centralizado a un esquema distribuido consiste en desarrollar un método para generar marcas temporales únicas. Entonces, podremos aplicar directamente nuestros protocolos anteriores a los entornos no replicados.

18.4.2.1 Generación de marcas temporales únicas

Existen dos métodos principales para generar marcas temporales únicas; uno de ellos es centralizado y el otro distribuido. En el esquema centralizado, se elige un único sitio para distribuir las marcas temporales. El sitio puede utilizar un contador lógico o su propio reloj local para dicho propósito.

En un esquema distribuido, cada sitio genera una marca temporal local única utilizando un contador lógico o el reloj local. La marca temporal única global se obtiene mediante la concatenación de la marca temporal única local con el identificador del sitio, que debe ser único (Figura 18.2). El orden de concatenación es importante. Utilizamos el identificador del sitio en la posición menos significativa para garantizar que las marcas temporales globales generadas en un sitio no sean siempre mayores que las generadas en otro sitio. Compare esta técnica de generación de marcas temporales únicas con la que hemos presentado en la Sección 18.1.2 para generar nombres únicos.

Podemos seguir teniendo problemas si uno de los sitios genera marcas temporales locales a una tasa más rápida que otros sitios. En dicho caso, el contador lógico del sitio más rápido tendrá un valor mayor que el de los otros sitios. Por tanto, todas las marcas temporales generadas por el sitio más rápido serán mayores que las generadas por los otros sitios. Se necesita un mecanismo

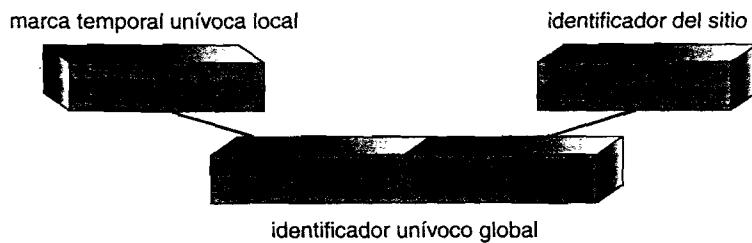


Figura 18.2 Generación de marcas temporales unívocas.

para garantizar que las marcas temporales locales se generen de manera equitativa en todo el sistema. Para conseguir una generación equitativa de marcas temporales, definimos dentro de cada sitio S_i un reloj lógico (LC_i), que genera la marca temporal local (véase la Sección 18.1.2). Para garantizar que los diversos relojes lógicos estén sincronizados, imponemos la condición de que cada sitio S_i deberá hacer avanzar su reloj lógico cada vez que una transacción T_i con marca temporal $\langle x, y \rangle$ visite dicho sitio y x sea mayor que el valor actual de LC_i . En este caso, el sitio S_i hará avanzar su reloj lógico al valor $x + 1$.

Si se utiliza el reloj del sistema para generar marcas temporales, entonces las marcas temporales se asignarán de manera equitativa, siempre que no haya ningún sitio cuyo reloj del sistema sea demasiado rápido o demasiado lento. Puesto que los relojes pueden no ser perfectamente precisos, debe utilizarse una técnica similar a la que se emplea con los relojes lógicos para garantizar que ningún reloj se adelante o se retrase demasiado con respecto a ningún otro.

18.4.2.2 Esquema de ordenación de marcas temporales

El esquema básico de marcas temporales presentado en la Sección 6.9 puede ampliarse de manera sencilla a un sistema distribuido. Como en el caso centralizado, pueden provocarse anulaciones en cascada si no se utiliza ningún mecanismo para impedir que una transacción lea el valor de un elemento de datos que todavía no haya sido confirmado. Para eliminar las anulaciones en cascada, podemos combinar el esquema básico de marcas temporales de la Sección 6.9 con el protocolo 2PC de la Sección 18.3, con el fin de obtener un protocolo que garantice la serializabilidad sin anulaciones en cascada. Dejamos el desarrollo de dicho algoritmo como ejercicio para el lector.

El esquema básico de marcas temporales que acabamos de describir tiene la indeseable propiedad de que los conflictos entre las transacciones se resuelven mediante anulaciones, en lugar de mediante esperas. Para aliviar este problema, podemos almacenar en búfer las diversas operaciones `read` y `write` (es decir, *retardarlas*) hasta algún momento en que podamos estar seguros de que estas operaciones puedan llevarse a cabo sin provocar anulaciones. Una operación `read` por parte de T_i deberá ser retardada si existe una transacción T_j que va a realizar una operación `write`, pero todavía no lo ha hecho y $TS(T_j) < TS(T_i)$. De forma similar, una operación `write(x)` por parte de T_i deberá ser retardada si existe una transacción T_j que vaya a realizar una operación `read(x)` o `write(y)` y $TS(T_j) < TS(T_i)$.

Hay disponibles diversos métodos para garantizar esta propiedad. Uno de ellos, denominado **esquema conservador de ordenación de marcas temporales**, requiere que cada sitio mantenga una cola de lectura y una cola de escritura compuesta por todas las solicitudes `read` y `write` que vayan a ejecutarse en el sitio y que deban ser retardadas para preservar la propiedad anterior. No vamos a presentar dicho esquema aquí; de nuevo, dejamos el desarrollo del algoritmo como ejercicio para el lector.

18.5 Gestión de interbloqueos

Los algoritmos de prevención, evasión y detección de interbloqueos presentados en el Capítulo 7 pueden ampliarse para utilizarlos en un sistema distribuido. En esta sección, describiremos varios de estos algoritmos distribuidos.

18.5.1 Prevención y evasión de interbloqueos

Los algoritmos de prevención y evasión de interbloqueos presentados en el Capítulo 7 pueden utilizarse en un sistema distribuido, siempre que se realicen las apropiadas modificaciones. Por ejemplo, podemos utilizar la técnica de prevención de interbloqueos basada en ordenación de recursos definiendo simplemente una ordenación global entre todos los recursos del sistema. En otras palabras, a todos los recursos del sistema completo se les asignan números únicos, y un proceso podrá solicitar un recurso (en cualquier procesador) con el número único i sólo si no está en posesión de un recurso cuyo número único sea superior a i . De forma similar, podemos utilizar el algoritmo del banquero en un sistema distribuido designando uno de los procesos del sistema (el *banquero*) como proceso que mantiene la información necesaria para poder implementar el algoritmo del banquero. Todas las solicitudes de recursos deberán canalizarse a través del banquero.

El esquema de prevención de interbloqueos basado en una ordenación global de los recursos resulta simple de implementar en un entorno distribuido y no requiere un sobrecoste administrativo excesivo. El algoritmo del banquero también puede implementarse fácilmente, pero puede que el sobrecoste sea excesivo. El propio banquero puede llegar a ser un cuello de botella, ya que el número de mensajes enviados y recibidos por él puede ser muy grande. Por tanto, el esquema del banquero no parece tener un uso práctico en los sistemas distribuidos.

A continuación vamos a considerar un nuevo esquema de prevención de interbloqueos basado en la técnica de ordenación de marcas temporales con apropiación de recursos. Aunque esta técnica permite gestionar cualquier situación de interbloqueo que pueda surgir en un sistema distribuido, vamos a considerar por simplicidad únicamente el caso de una única instancia de cada tipo de recurso.

Para controlar la apropiación, asignamos un número único de prioridad a cada proceso. Estos números se usan para decidir si un determinado proceso P_i debe esperar por otro proceso P_j . Por ejemplo, podemos hacer que P_i espere por P_j si P_i tiene una prioridad mayor que la de P_j ; en caso contrario, P_i se anula. Este esquema evita los interbloqueos porque, para cada arista $P_i \rightarrow P_j$ en el grafo de espera, P_i tiene una prioridad mayor que P_j . Como consecuencia, no puede existir un ciclo.

Una dificultad de este esquema es la posibilidad de que se produzca el fenómeno de muerte por inanición. Algunos procesos con prioridades extremadamente bajas pueden terminar siendo siempre anulados. Esta dificultad puede evitarse utilizando marcas temporales. A cada proceso del sistema se le asigna una marca temporal única en el momento de crearlo. Se han propuesto dos esquemas de prevención de interbloqueos complementarios que utilizan marcas temporales:

- 1. El esquema espera-muere.** Este esquema se basa en una técnica no apropiativa. Cuando el proceso P_i solicita un recurso que actualmente está en manos de P_j , sólo se permite a P_i esperar si tiene una marca temporal que es menor que la de P_j (es decir, si P_i es más antiguo que P_j). En caso contrario, P_i se anula (*muere*). Por ejemplo, suponga que los procesos P_1 , P_2 y P_3 tienen las marcas temporales 5, 10 y 15, respectivamente. Si P_1 solicita un recurso que está en posesión de P_2 , P_1 esperará. Si P_3 solicita un recurso que está en posesión de P_2 , P_3 será anulado.
- 2. El esquema herido-espera.** Este esquema se basa en una técnica apropiativa y es la contrapartida del esquema espera-muere. Cuando el proceso P_i solicita un recurso que actualmente está en posesión de P_j , sólo se permite a P_i esperar si tiene una marca temporal mayor que la de P_j (es decir, P_i es más reciente que P_j). En caso contrario, P_j se anula (P_j es *herido* por P_i). Volviendo a nuestro ejemplo anterior con los procesos P_1 , P_2 y P_3 , si P_1 solicita un recurso que está en posesión de P_2 , entonces se le quitará el recurso a P_2 , y P_2 será anulado. Si P_3 solicita un recurso que está en posesión de P_2 , entonces P_3 tendrá que esperar.

Ambos esquemas pueden evitar el fenómeno de la muerte por inanición siempre que no asigne una nueva marca temporal a un proceso en el momento de anularlo. Puesto que las marcas temporales siempre se incrementan, un proceso que sea anulado llegará con el tiempo a tener la marca

temporal más pequeña, con lo que no volverá a ser anulado de nuevo. Sin embargo, existen significativas diferencias en la forma en que los dos esquemas operan.

- En el esquema espera-muere, un proceso más antiguo deberá esperar a que otro más reciente libere su recurso. Por tanto, cuanto más antiguo sea el proceso, más tenderá a esperar. Por contraste, en el esquema herido-espera, los procesos más antiguos nunca esperan por un proceso más reciente.
- En el esquema espera-muere, si un proceso P_i muere y es anulado porque ha solicitado un recurso mantenido por el proceso P_j , entonces P_i puede volver a realizar la misma secuencia de solicitudes en el momento que se reinicie. Si el recurso continúa siendo mantenido por P_j , entonces P_i volverá a morir. Por tanto, P_i puede morir varias veces antes de adquirir el recurso necesario. Compare esta serie de sucesos con lo que ocurre en el esquema herido-espera. El proceso P_i es herido y anulado porque P_j ha solicitado un recurso que él tenía. Cuando P_i se reinicia y solicita el recurso que ahora está en posesión de P_j , P_i tendrá que esperar. Por tanto, se producen menos anulaciones en el esquema herido-espera.

El problema principal con ambos esquemas es que pueden tener lugar anulaciones innecesarias.

18.5.2 Detección de interbloqueos

El algoritmo de prevención de interbloqueos puede provocar apropiaciones de recursos incluso aunque no se haya producido ningún interbloqueo. Para evitar las apropiaciones innecesarias, podemos utilizar un algoritmo de detección de interbloqueos. Para ello, construimos un gráfico de espera que describa el estado de asignación de los recursos; puesto que estamos asumiendo que sólo hay un único recurso de cada tipo, la presencia de un ciclo en el grafo de espera representará un interbloqueo.

El problema principal en un sistema distribuido consiste en decidir cómo mantener el grafo de espera. Vamos a ilustrar este problema describiendo varias técnicas comunes para tratar con esta cuestión. Estos esquemas requieren que cada sitio mantenga un grafo de espera *local*. Los nodos del grafo se corresponden con todos los procesos (tanto locales como no locales) que actualmente posean o estén solicitando algunos de los recursos locales a ese sitio. Por ejemplo, en la figura 18.3 tenemos un sistema que consta de dos sitios, cada uno de los cuales mantiene su propio grafo de espera local. Observe que los procesos P_2 y P_3 aparecen en ambos grafos, indicando que los procesos han solicitado recursos pertenecientes a ambos sitios.

Estos grafos de espera locales se construyen de la forma usual para los procesos y recursos locales. Cuando un proceso P_i del sitio S_1 necesita un recurso que está en posesión del proceso P_j en el sitio S_2 , P_i envía un mensaje de solicitud al sitio S_2 . Entonces, se inserta la arista $P_i \rightarrow P_j$ en el grafo de espera local del sitio S_2 .

Claramente, si alguno de los grafos de espera locales tiene un ciclo, querrá decir que se ha producido un interbloqueo. Sin embargo, el hecho de que no encontramos ningún ciclo en ninguno de los grafos de espera locales no quiere decir que no haya ningún interbloqueo. Para ilustrar este problema, consideremos el sistema mostrado en la Figura 18.3. Cada uno de los grafos de espera

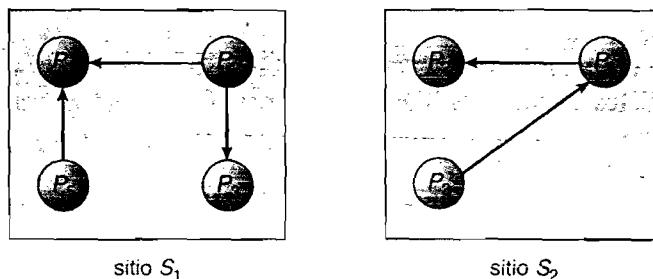


Figura 18.3 Dos grafos de espera locales.

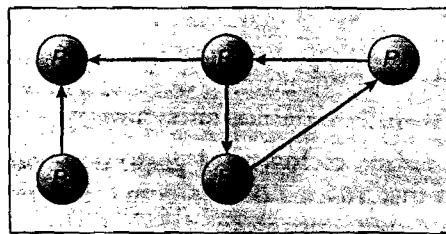


Figura 18.4 Grafo de espera global para la Figura 18.3.

es acíclico; sin embargo, existe un interbloqueo en el sistema. Para demostrar que no se ha producido ningún interbloqueo, debemos demostrar que la **unión** de todos los grafos locales es acíclica. El grafo (Figura 18.4) que obtenemos al realizar la unión de los dos grafos de espera de la Figura 18.3 sí que contiene un ciclo, lo que implica que el sistema está interbloqueado.

Hay disponibles varios métodos para organizar el grafo de espera en un sistema distribuido. A continuación vamos a describir varios de los esquemas más comunes.

18.5.2.1 Enfoque centralizado

En el enfoque centralizado, se construye un grafo de espera global como la unión de todos los grafos de espera locales. Ese grafo global es mantenido mediante un **único** proceso: el **coordinador de detección de interbloqueos**. Puesto que existe un retardo de comunicación en el sistema, debemos distinguir entre dos tipos de grafos de espera. El grafo *real* describe el estado real, pero desconocido, del sistema en cualquier instante de tiempo, tal como sería visto por un observador omnisciente. El grafo *construido* es una aproximación generada por el coordinador durante la generación de su algoritmo. El grafo construido debe generarse de modo que, cada vez que se invoque el algoritmo de detección, los resultados obtenidos sean correctos. Por *correctos* queremos decir lo siguiente:

- Si existe un interbloqueo, se informará de ello apropiadamente.
- Si se informa de que existe un interbloqueo, entonces el sistema se encuentra, ciertamente, en estado de interbloqueo.

Como veremos, no resulta sencillo construir algoritmos que sean correctos desde este punto de vista.

El grafo de espera puede construirse en tres instantes distintos de tiempo:

1. Cada vez que se inserta o elimina una nueva arista en uno de los grafos de espera locales.
2. Periódicamente, cuando se haya producido una serie de cambios en un grafo de espera.
3. Cada vez que el coordinador de detección de interbloqueos necesite invocar el algoritmo de detección de ciclos.

Cuando se invoca el algoritmo de detección de interbloqueos, el coordinador analiza su grafo global. Si encuentra un ciclo, selecciona una *victima* para anularla. El coordinador deberá notificar a todos los sitios que se ha seleccionado un proceso concreto como víctima. A su vez, esos sitios anularán el proceso víctima.

Consideremos la opción 1. Cada vez que se inserte o elimine una arista en un grafo local, el sitio local deberá enviar un mensaje al coordinador para notificarle la modificación. Al recibir tal mensaje, el coordinador actualiza su grafo global.

Alternativamente (opción 2), un sitio puede enviar varios de esos cambios en un único mensaje de manera periódica. Volviendo sobre nuestro ejemplo anterior, el proceso coordinador mantendrá el grafo de espera global tal como se muestra en la Figura 18.4. Cuando el sitio S_2 inserte la arista $P_3 \rightarrow P_4$ en su grafo de espera local, también enviará un mensaje al coordinador. De forma

similar, cuando el sitio S_1 borre la arista $P_5 \rightarrow P_1$ porque P_1 ha liberado un recurso que estaba siendo solicitado por P_5 , se enviará un mensaje apropiado al coordinador.

Observe que, independientemente de la opción que se utilice, pueden producirse anulaciones innecesarias, como resultado de dos situaciones:

1. Pueden existir **falsos ciclos** en el grafo de espera global. Para ilustrar este punto, vamos a considerar una instantánea del sistema tal como se muestra en la Figura 18.5. Suponga que P_2 libera el recurso que posee en el sitio S_1 , lo que provoca el borrado de la arista $P_1 \rightarrow P_2$ en el sitio S_1 . El proceso P_2 solicita entonces un recurso que está en posesión de P_3 en el sitio S_2 , lo que da lugar a la adición de la arista $P_2 \rightarrow P_3$ en el sitio S_2 . Si el mensaje *insert* $P_2 \rightarrow P_3$ del sitio S_2 llega antes que el mensaje *delete* $P_1 \rightarrow P_2$ del sitio S_1 , el coordinador podría detectar el falso ciclo $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$ después de ejecutar la operación *insert* (pero antes de ejecutar la operación *delete*). En este caso, se podría iniciar el procedimiento de recuperación de interbloqueos, a pesar de que no se ha producido verdaderamente ningún interbloqueo.
2. También pueden producirse anulaciones innecesarias en aquellos casos en que se haya producido verdaderamente un interbloqueo y se haya seleccionado una víctima, pero *al mismo tiempo* uno de los procesos haya sido abortado por razones no relacionadas con el interbloqueo (por ejemplo, cuando un proceso ha excedido su tiempo asignado). Por ejemplo, supongamos que el sitio S_1 de la Figura 18.3 decide abortar P_2 . Al mismo tiempo, el coordinador ha descubierto un ciclo y ha seleccionado a P_3 como víctima. Ahora tanto P_2 como P_3 serán anulados, aunque sólo era necesario anular P_2 .

Consideremos ahora un algoritmo centralizado de detección de interbloqueos que utiliza la opción 3 y que detecte todos los interbloqueos que verdaderamente se produzcan y no detecte ningún falso interbloqueo. Para evitar informar de falsos interbloqueos, necesitamos que a las solicitudes de los diferentes sitios se les añadan identificadores únicos (o marcas temporales). Cuando el proceso P_i en el sitio S_1 solicita un recurso de P_j en el sitio S_2 , se envía un mensaje de solicitud con marca temporal TS . Entonces se inserta la arista $P_i \rightarrow P_j$ con la etiqueta TS en el grafo de espera local de S_1 . Esta arista se inserta en el grafo de espera local del sitio S_2 sólo si el sitio S_2 ha recibido el mensaje de solicitud y no puede conceder inmediatamente el recurso solicitado. Una solicitud de P_i a P_j en el mismo sitio será gestionada de la forma usual, sin asociar ninguna marca temporal con la arista $P_i \rightarrow P_j$. El algoritmo de detección es el siguiente:

1. El controlador envía un mensaje de inicio a cada sitio del sistema.
2. Al recibir este mensaje, cada sitio envía su grafo de espera local al coordinador. Cada uno de estos grafos de espera locales contiene toda la información local que el sitio tiene acerca del estado del grafo real. El grafo refleja un estado instantáneo del sitio, pero no está sincronizado con respecto a ningún otro sitio.
3. Cuando el controlador ha recibido una respuesta de cada sitio, construye un grafo de la forma siguiente:
 - a. El grafo construido contiene un vértice por cada proceso del sistema.
 - b. El grafo tiene una arista $P_i \rightarrow P_j$ si y sólo si existe una arista $P_i \rightarrow P_j$ en uno de los grafos de espera o una arista $P_i \rightarrow P_j$ con alguna etiqueta TS en más de un grafo de espera.

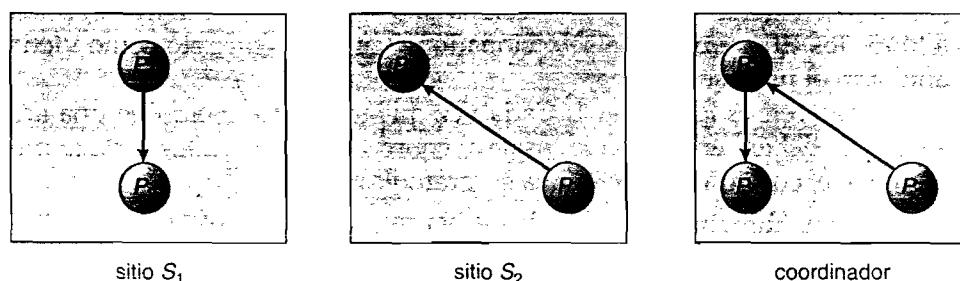


Figura 18.5 Grafos de espera locales y globales.

Si el grafo construido contiene un ciclo, entonces el sistema estará en estado de interbloqueo. Si el grafo construido no contiene un ciclo, entonces el sistema no estaba interbloqueado en el momento de invocar el algoritmo de detección como resultado de los mensajes de iniciación enviados por el coordinador (en el paso 1).

18.5.2.2 Enfoque completamente distribuido

En el **algoritmo de detección de interbloqueos completamente distribuido**, todos los controladores comparten igualmente la responsabilidad de detectar los interbloqueos. Cada sitio construye un grafo en espera que representa una parte del grafo total, dependiendo del comportamiento dinámico del sistema. La idea es que, si existe un interbloqueo, aparecerá un ciclo en al menos uno de los grafos parciales. Vamos a presentar uno de tales algoritmos, que implica la construcción de grafos parciales en cada sitio.

Cada sitio mantiene su propio grafo de espera local. Un grafo de espera local, en este esquema, difiere de los descritos anteriormente, en el sentido de que añadimos un nodo adicional P_{ex} al grafo. Existirá una arista $P_i \rightarrow P_{ex}$ en el grafo si P_i está esperando por un elemento de datos en otro sitio que esté actualmente en posesión de *cualquier* proceso. De forma similar, existirá una arista $P_{ex} \rightarrow P_j$ en el grafo si hay algún proceso en otro sitio que está esperando a adquirir un recurso que actualmente está en posesión de P_j en este sitio local.

Para ilustrar esta situación, consideremos de nuevo los dos grafos de espera locales de la Figura 18.3. La adición del nodo P_{ex} a ambos grafos da como resultado los grafos de espera locales que se muestran en la Figura 18.6.

Si un grafo de espera local contiene un ciclo que no implica al nodo P_{ex} , entonces el sistema estará en estado interbloqueado. Sin embargo, si un grafo local contiene un ciclo que implica a P_{ex} , entonces esto quiere decir que existe la *posibilidad* de que haya un interbloqueo. Para ver si existe efectivamente un interbloqueo, debemos invocar un algoritmo distribuido de detección de interbloqueos.

Supongamos que, en el sitio S_i , el grafo de espera local contiene un ciclo que implica al nodo P_{ex} . Este ciclo debe ser de la forma

$$P_{ex} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \dots \rightarrow P_{k_n} \rightarrow P_{ex}$$

que indica que el proceso P_{k_n} en el sitio S_i está esperando a adquirir un elemento de datos localizado en algún otro sitio, como por ejemplo S_j . Al descubrir este ciclo, el sitio S_i envía al ciclo S_j un mensaje de detección de interbloqueo que contiene la información acerca de dicho ciclo.

Cuando el sitio S_j recibe este mensaje de detección de interbloqueos, actualiza su grafo de espera local con la nueva información y a continuación analiza el grafo de espera recién reconstruido para ver si hay algún ciclo que no implique a P_{ex} . Si existe uno de esos ciclos, se habrá encontrado un interbloqueo y se invocará un esquema de recuperación apropiado. Si se descubre un ciclo que implique a P_{ex} entonces S_j transmitirá un mensaje de detección de interbloqueos al sitio apropiado, como por ejemplo, S_k . El sitio S_k , a su vez, repite el mismo procedimiento. De este modo, después de un número finito de turnos, se podrá descubrir un interbloqueo o se detendrá el algoritmo de detección de interbloqueos.

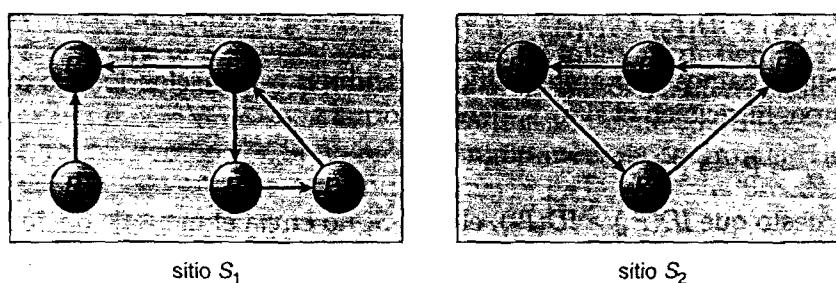


Figura 18.6 Grafos de espera locales aumentados de la Figura 18.3.

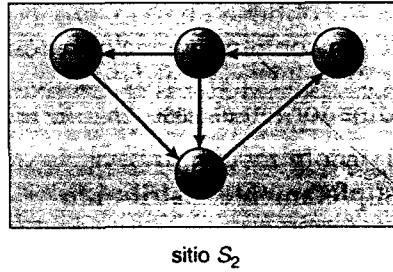


Figura 18.7 Grafo de espera local aumentado de la Figura 18.6 en el sitio S_2 .

Para ilustrar este procedimiento, consideremos los grafos de espera locales de la Figura 18.6. Supongamos que el sitio S_1 descubre el ciclo

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}$$

Puesto que P_3 está esperando a adquirir un elemento de datos del sitio S_2 , se transmite desde el sitio S_1 al sitio S_2 un mensaje de detección de interbloqueos que describa dicho ciclo. Cuando S_2 recibe este mensaje, actualiza su grafo de espera local, obteniendo el grafo de espera de la Figura 18.7. Este grafo contiene el ciclo

$$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2,$$

que no incluye el nodo P_{ex} . Por tanto, el sistema está en un estado de interbloqueo y será necesario invocar un esquema de recuperación apropiado.

Observe que el resultado sería el mismo si el sitio S_2 descubriera el ciclo primero en su grafo de espera local y enviara el mensaje de detección de interbloqueos al sitio S_1 . En el peor de los casos, ambos sitios descubrirían el ciclo a aproximadamente al mismo tiempo, y los dos mensajes de detección de interbloqueos se enviarían: uno desde S_1 a S_2 y otro desde S_2 a S_1 . Esta situación provoca una transferencia de mensajes innecesaria y un sobrecoste adicional en forma de actualización de los dos grafos de espera locales y de la búsqueda de ciclos en ambos grafos.

Para reducir el tráfico de mensajes, asignamos a cada proceso P_i un identificador único, que denotaremos por $ID(P_i)$. Cuando el sitio S_k descubre que su grafo de espera local contiene un ciclo que implica al nodo 1 P_{ex} en la forma

$$P_{ex} \rightarrow P_{K_1} \rightarrow P_{K_2} \rightarrow \dots \rightarrow P_{K_n} \rightarrow P_{ex},$$

envía un mensaje de detección de interbloqueos a otro sitio sólo si

$$ID(P_{K_n}) < ID(P_{K_1})$$

En caso contrario, el sitio S_k continuará con su ejecución normal, dejando la responsabilidad de iniciar el algoritmo detección de interbloqueos a algún otro sitio.

Para ilustrar este esquema, consideremos de nuevo los grafos de espera mantenidos en los sitios S_1 y S_2 de la Figura 18.6. Suponga que

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4).$$

Supongamos que ambos sitios descubren estos ciclos locales a aproximadamente el mismo tiempo. El ciclo en el sitio S_1 tiene la forma

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Puesto que $ID(P_3) > ID(P_2)$, el sitio S_1 no envía el mensaje de detección de interbloqueos al sitio S_2 .

El ciclo en el sitio S_2 es de la forma

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}.$$

Puesto que $ID(P_2) < ID(P_3)$, el sitio S_2 sí que envía un mensaje de detección de interbloqueos al sitio S_1 , el cual, al recibir el mensaje, actualizará su grafo de espera local. El sitio S_1 buscará entonces un ciclo en el grafo y descubrirá que el sistema está en estado de interbloqueo.

18.6 Algoritmos de elección

Como hemos apuntado en la Sección 18.3, muchos algoritmos distribuidos emplean un proceso coordinador que realiza funciones necesarias para otros procesos del sistema. Estas funciones incluyen garantizar la exclusión mutua, mantener un grafo de espera global para la detección de interbloqueos, reemplazar un testigo perdido y controlar un dispositivo de entrada o de salida en el sistema. Si el proceso coordinador falla debido al fallo del sitio en que reside, el sistema puede continuar su ejecución sólo si restaura una nueva copia del coordinador en algún otro sitio. Los algoritmos que determinan dónde debe reiniciarse una nueva copia del coordinador se denominan **algoritmos de elección**.

Los algoritmos de elección asumen que hay un número de prioridad único asociado con cada proceso activo del sistema. Para facilitar la notación, vamos a asumir que el número de prioridad del proceso P_i es i . Para simplificar las exposiciones, vamos a suponer también que existe una correspondencia biúnica entre los procesos y los sitios y vamos a referirnos, por tanto, a ambos como si fueran procesos. El coordinador será siempre el proceso con el número de prioridad más alta. De este modo, cuando un coordinador falle, el algoritmo deberá elegir el proceso activo que tenga el número de prioridad más grande. Este número debe enviarse a cada proceso activo del sistema. Además, el algoritmo debe proporcionar un mecanismo para que un proceso recuperado pueda identificar al coordinador actual.

En esta sección vamos a presentar ejemplos de algoritmo de elección para dos configuraciones diferentes de sistemas distribuidos. El primer algoritmo se aplica a los sistemas en los que cada proceso puede enviar un mensaje a cada uno de los otros procesos del sistema. El segundo algoritmo se aplica a los sistemas organizados en forma de anillo (lógico o físico). Ambos algoritmos requieren n^2 mensajes para realizar una elección, donde n es el número de procesos del sistema. Vamos a suponer que un proceso que haya fallado sabe, al recuperarse, que ha fallado y que toma por tanto las acciones apropiadas para volver a unirse al conjunto de procesos activos.

18.6.1 El algoritmo de imposición

Suponga que el proceso P_i envía una solicitud que no es respondida por el coordinador dentro de un intervalo de tiempo T . En esta situación, se asume que el coordinador ha fallado y que P_i tratará de elegirse a sí mismo como nuevo coordinador. Esta tarea puede llevarse a cabo utilizando el siguiente algoritmo.

El proceso P_i envía un mensaje de elección a todos los procesos que tengan un número de prioridad más alto. El proceso P_i esperará entonces durante un intervalo de tiempo T a recibir una respuesta de alguno de estos procesos.

Si no recibe ninguna respuesta dentro del tiempo T , el proceso P_i asumirá que todos los procesos con prioridad mayor que i han fallado y se elegirá a sí mismo como nuevo coordinador. El proceso P_i reiniciará una nueva copia del coordinador y enviará un mensaje para informar a todos los procesos activos con números de prioridad menores que i de que P_i es el nuevo coordinador.

Sin embargo, si se recibe una respuesta, P_i arrancará un intervalo de tiempo T' , esperando recibir un mensaje que le informe de que se ha elegido a un proceso con un número de prioridad más alto (es decir, que algún otro proceso se está eligiendo a sí mismo coordinador y debe informar del resultado dentro del tiempo T'). Si no recibe ningún mensaje dentro del tiempo T' , entonces se asume que el proceso con número de prioridad mayor ha fallado, y el proceso P_i debe reiniciar el algoritmo.

Si P_i no es el coordinador, entonces, en cualquier instante de la ejecución, P_i puede recibir uno de los dos mensajes siguientes del proceso P_j :

1. P_j es el nuevo coordinador ($j > i$). El proceso P_i , a su vez, registrará esta información.

2. P_j ha iniciado una elección ($j < i$). El proceso P_i envía una respuesta a P_j y comenzará su propio algoritmo de elección, supuesto que P_i no haya iniciado ya dicho mecanismo de elección.

El proceso que complete su algoritmo tendrá el número de prioridad más alto y será elegido como coordinador. Asimismo, habrá enviado su número a todos los procesos activos con prioridad más baja. Después de que un proceso fallido se recupere, comenzará inmediatamente a ejecutar el mismo algoritmo. Si no hay ningún proceso activo con un número de prioridad más alta, el proceso recuperado forzará a todos los procesos con número de prioridad más bajo a que le dejen convertirse en el proceso coordinador, incluso si actualmente existe un coordinador activo con un número de prioridad más bajo. Por esta razón, este algoritmo se denomina **algoritmo de imposición**.

Podemos demostrar la operación del algoritmo con un ejemplo simple de un sistema compuesto por los procesos P_1 a P_4 . Las operaciones son las siguientes:

1. Todos los procesos están activos; P_4 es el proceso coordinador.
2. P_1 y P_4 fallan. P_2 determina que P_4 ha fallado al enviar una solicitud que no es respondida dentro del tiempo T . P_2 comienza entonces su algoritmo de elección enviando una solicitud a P_3 .
3. P_3 recibe la solicitud, responde a P_2 , y comienza su propio algoritmo enviando una solicitud de elección a P_4 .
4. P_2 recibe la respuesta de P_3 y comienza a esperar un intervalo T' .
5. P_4 no responde dentro del intervalo T , por lo que P_3 se elige a sí mismo como nuevo coordinador y envía el número 3 a P_2 y P_1 . (P_1 no recibe el número, ya que ha fallado.)
6. Posteriormente, cuando P_1 se recupere, enviará una solicitud de elección a P_2 , P_3 y P_4 .
7. P_2 y P_3 responderán a P_1 y comenzarán su propio algoritmo de elección. P_3 volverá a ser elegido, a través de la misma secuencia de sucesos que antes.
8. Finalmente, P_4 se recupera y notifica a P_1 , P_2 y P_3 que es el coordinador actual (P_4 no envía ninguna solicitud de elección, ya que es el proceso con el número de prioridad más alto del sistema)

18.6.2 El algoritmo del anillo

El **algoritmo del anillo** presupone que los enlaces son unidireccionales y que cada proceso envía sus mensajes al vecino situado a su derecha. La estructura de datos principal usada por el algoritmo es la **lista de activos**, una lista que contiene los números de prioridad de todos los procesos activos en el sistema en el momento de terminar el algoritmo; cada proceso mantiene su propia lista de activos. El algoritmo funciona de la siguiente forma:

1. Si el proceso P_i detecta un fallo del coordinador, crea una nueva lista de activos que estará inicialmente vacía. A continuación, envía el mensaje $elect(i)$ a su vecino de la derecha y añade el número i a su lista de activos.
2. Si P_i recibe un mensaje $elect(j)$ del proceso situado a su izquierda, debe responder de una de tres maneras:
 - a. Si se trata del primer mensaje $elect$ que ha visto o enviado, P_i creará una nueva lista de activos con los números i y j . A continuación, enviará un mensaje $elect(i)$, seguido del mensaje $elect(j)$.
 - b. Si $i \neq j$, es decir, si el mensaje recibido no contiene el número de P_i , entonces P_i añade j a su lista de activos y re-envía el mensaje a su vecino de la derecha.
 - c. Si $i = j$, es decir, si P_i recibe el mensaje $elect(i)$, entonces la lista de activos de P_i contiene ahora los números de todos los procesos activos del sistema. El proceso P_i ahora

puede determinar el número mayor de la lista de activos con el fin de identificar un nuevo proceso coordinador.

Este algoritmo no especifica cómo puede determinar un proceso que se está recuperando el número del proceso coordinador actual. Una de las posibles soluciones exige que el proceso que esté en recuperación envíe un mensaje de consulta. Este mensaje se re-envía alrededor del anillo hasta el coordinador actual, que a su vez envía una respuesta que contiene su propio número.

18.7 Procedimientos de acuerdo

Para que un sistema sea fiable, necesitamos un mecanismo que permita a un conjunto de procesos acordar un *valor* común. Dicho acuerdo puede no tener lugar, por varias razones. En primer lugar, el medio de comunicación puede tener errores, que provoquen la pérdida o la corrupción de mensajes. En segundo lugar, los propios procesos pueden fallar, provocando un comportamiento impredecible de los procesos. Lo mejor que cabe esperar en este caso es que los procesos fallen de manera limpia, deteniendo su ejecución sin desviarse de su patrón de ejecución normal. En el peor de los casos, los procesos pueden enviar mensajes corruptos o incorrectos a otros procesos, o incluso colaborar con otros procesos fallidos en un intento de destruir la integridad del sistema.

El problema de los generales bizantinos proporciona una analogía para esta situación. Varias divisiones del ejército bizantino, cada una de ellas comandada por su propio general, rodean un campamento enemigo. Los generales bizantinos deben alcanzar un acuerdo sobre si atacar o no al enemigo al alba. Resulta crucial que todos los generales se pongan de acuerdo, ya que si sólo atacaran algunas de las divisiones, el ejército sería derrotado. Las diversas divisiones están geográficamente dispersas y los generales pueden comunicarse entre sí sólo a través de mensajeros que van corriendo de un campamento a otro. Los generales pueden no ser capaces de alcanzar un acuerdo por al menos dos razones principales:

1. Los mensajeros pueden ser capturados por el enemigo y pueden, por tanto, no poder entregar sus mensajes. Esta situación se corresponde con una comunicación no fiable dentro de un sistema informático y se analiza más detalladamente en la Sección 18.7.1.
2. Los generales pueden ser *traidores*, que estén intentando impedir que los generales *leales* alcancen un acuerdo. Esta situación se corresponde con el caso de los procesos fallidos dentro de un sistema informático y se analiza más en detalle en la Sección 18.7.2.

18.7.1 Comunicaciones no fiables

Vamos a suponer primero que, si los procesos fallan, lo hacen de una forma limpia y que el medio de comunicación es no fiable. Supongamos que el proceso P_i en el sitio S_1 , que ha enviado un mensaje al proceso P_j del sitio S_2 , necesita saber si P_j ha recibido el mensaje, para poder decidir cómo continuar con sus cálculos. Por ejemplo, P_i puede decidir calcular una determinada función *foo* si P_j ha recibido su mensaje o calcular una función *boo* si P_j no ha recibido el mensaje (debido por ejemplo a algún fallo de hardware).

Para detectar fallos, podemos usar un esquema de temporización similar al descrito en la Sección 16.7.1. Cuando P_i envía un mensaje, también especifica un intervalo de tiempo durante el cual esperará a recibir un mensaje de confirmación procedente de P_j . Cuando P_j reciba el mensaje, envía inmediatamente una confirmación a P_i . Si P_i recibe el mensaje de confirmación dentro del intervalo de tiempo especificado, puede concluir con seguridad que P_j ha recibido el mensaje. Sin embargo, si se produce un fin de temporización, entonces P_i necesitará retransmitir su mensaje y esperar una confirmación. Este procedimiento continuará hasta que P_i obtenga el mensaje de confirmación o hasta que el sistema le notifique que el sitio S_2 ha fallado. En el primer caso, calculará S ; en el segundo caso, calculará F . Observe que, si éstas son las únicas dos alternativas viables, P_i debe esperar hasta que reciba una notificación de que se ha producido alguna de las dos situaciones.

Supongamos ahora que P_j también necesita saber que P_i ha recibido su mensaje de confirmación, para poder decidir cómo continuar con sus cálculos. Por ejemplo, P_j puede querer calcular *foo* sólo si está seguro de que P_i ha obtenido su confirmación. En otras palabras, P_i y P_j calcularán *foo* si y sólo si ambos han acordado hacerlo. Se puede demostrar que, en presencia de fallos, no es posible llevar a cabo esta tarea. Más precisamente, no resulta posible en un entorno distribuido que los procesos P_i y P_j acuerden de manera completa sus respectivos estados.

Para demostrar esta afirmación, vamos a suponer que existiera una secuencia mínima de transferencias de mensajes tal que, después de haber sido entregados los mensajes, ambos procesos acordaran calcular *foo*. Sea m' el último mensaje enviado por P_i a P_j . Puesto que P_i no sabe si su mensaje llegará a P_j (puesto que el mensaje puede perderse debido a un fallo), P_i ejecutará *foo* independientemente del resultado de la entrega del mensaje. Por tanto, m' podría eliminarse de la secuencia sin afectar al proceso de decisión. Por tanto, la secuencia original no era mínima, lo que contradice nuestra suposición y demuestra que no existe tal secuencia. Los procesos no pueden nunca estar seguros de que ambos calcularán *foo*.

18.7.2 Procesos fallidos

Ahora vamos a suponer que el medio de comunicación es fiable pero que los procesos pueden fallar de formas impredecibles. Consideré un sistema formado por n procesos, de los cuales no más de m son fallidos. Suponga que cada proceso P_i tiene algún valor privado de V_i . Queremos desarrollar un algoritmo que permita a cada uno de los procesos no fallidos P_i construir un vector $X_i = (A_{i,1}, A_{i,2}, \dots, A_{i,n})$ tal que se cumplan las siguientes condiciones:

1. Si P_j es un proceso no fallido, entonces $A_{i,j} = V_j$.
2. Si P_i y P_j son ambos procesos no fallidos, entonces $X_i = X_j$.

Existen muchas soluciones a este problema y todas ellas comparten las siguientes propiedades:

1. Sólo puede desarrollarse un algoritmo correcto si $n \geq 3 \times m + 1$.
2. El retardo de caso peor para alcanzar un acuerdo es proporcional a $m + 1$ retardos de transferencia de mensaje.
3. El número de mensajes requeridos para alcanzar el acuerdo es grande. Ninguno de los procesos es totalmente de confianza, por lo que todos los procesos deben recopilar toda la información y tomar sus propias decisiones.

En lugar de presentar una solución general, que sería complicada, vamos a presentar un algoritmo para el caso más simple, en el que $m = 1$ y $n = 4$. El algoritmo requiere dos turnos de intercambio de información:

1. Cada proceso envía su valor privado a los otros tres procesos.
2. Cada proceso envía la información que ha obtenido en la primera ronda a todos los demás procesos.

Un proceso fallido puede, obviamente, negarse a enviar mensajes. En este caso, un proceso no fallido puede seleccionar un valor arbitrario y pretender que dicho valor ha sido enviado por el proceso fallido.

Una vez que se han completado estas dos rondas, un proceso no fallido P_i puede construir su vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ del siguiente modo:

1. $A_{i,i} = V_i$.
2. Para $j \neq i$, si al menos dos de los tres valores informados para el proceso P_j (en las dos rondas de intercambio) concuerdan, entonces se usa el valor mayoritario para asignar un valor a $A_{i,j}$. En caso contrario, se utiliza un valor predeterminado (por ejemplo, *nil*) para asignar el valor a $A_{i,j}$.

18.8 Resumen

En un sistema distribuido sin memoria común y sin un reloj común, a veces es imposible determinar el orden exacto en que se producen los sucesos. La relación *ha sucedido antes* es sólo una ordenación parcial de los sucesos en un sistema distribuido. Pueden utilizarse marcas temporales para proporcionar una ordenación de sucesos coherente.

Puede implementarse la exclusión mutua en un entorno distribuido de diversas maneras. Con el enfoque centralizado, se elige uno de los procesos del sistema para coordinar la entrada en la sección crítica. Con el enfoque completamente distribuido, la toma de decisiones se distribuye por todo el sistema. Un posible algoritmo distribuido, que es aplicable a las redes con estructura de anillo es la técnica de paso de testigo.

Para garantizar la atomicidad, todos los sitios en los que una transacción T se haya ejecutado deben acordar cuál es el resultado final de la ejecución. T debe confirmarse en todos los sitios o abortarse en todos los sitios. Para garantizar esta propiedad, el coordinador de la transacción T debe ejecutar un protocolo de confirmación. El protocolo de confirmación más ampliamente utilizado es el protocolo 2PC.

Podemos modificar los diversos esquemas de control de concurrencia utilizados en los sistemas centralizados para emplearlos en un entorno distribuido. En el caso de los protocolos de bloqueo, sólo necesitamos modificar la forma en que se implementa el gestor de bloqueos. En el caso de los esquemas de marcas temporales y de validación, el único cambio necesario es el desarrollo de un mecanismo para generar marcas temporales únicas globales. El mecanismo puede limitarse a concatenar una marca temporal local con el identificador del sitio, o puede hacer avanzar los relojes locales cada vez que se reciba un mensaje que tenga una marca temporal mayor.

El método principal para tratar con los interbloqueos en un entorno distribuido es la detección de interbloqueos. El problema fundamental es el de cómo mantener el grafo de espera. Entre los métodos existentes para organizar el grafo de espera se incluyen la técnica centralizada y la técnica completamente distribuida.

Algunos algoritmos distribuidos requieren el uso de un coordinador. Si el coordinador falla debido a un fallo del sitio en el que reside, el sistema sólo podrá continuar operando si reinicia una nueva copia del coordinador en algún otro sitio. Puede hacer esto manteniendo un coordinador de reserva que esté listo para asumir la responsabilidad si el coordinador falla. Otra posible técnica consiste en elegir el nuevo coordinador después de que el antiguo haya fallado. Los algoritmos que determinan dónde debe reiniciarse una copia del coordinador se denominan algoritmos de elección. Pueden usarse dos algoritmos: el algoritmo de imposición y el algoritmo del anillo, para elegir un nuevo coordinador en el caso de que se produzca un fallo.

Ejercicios

- 18.1 Explique las ventajas y desventajas de los dos métodos que hemos presentado para la generación de marcas temporales globalmente únicas.
- 18.2 El esquema de marcas temporales basadas en reloj lógico que se ha presentado en este capítulo proporciona la siguiente garantía: Si el suceso A ocurre antes que el suceso B, entonces la marca temporal de A será menor que la marca temporal de B. Sin embargo, observe que no podemos ordenar dos sucesos basándonos únicamente en sus marcas temporales. El hecho de que un suceso C tenga una marca temporal inferior a la marca temporal del suceso D no quiere decir necesariamente que el suceso C haya tenido lugar antes que el suceso D; C y D podrían ser sucesos concurrentes en el sistema. Proponga diversas formas en las que podría ampliarse el esquema de marcas temporales basado en reloj lógico para distinguir entre sucesos concurrentes y sucesos que puedan ordenarse mediante la relación *ha sucedido antes*.
- 18.3 Imagine que nuestra empresa está construyendo una red informática y que se nos pide que escribamos un algoritmo para conseguir la exclusión mutua distribuida. ¿Qué esquema propondría? Razone su respuesta.

- 18.4 ¿Por qué la detección de interbloqueos es mucho más costosa en un entorno distribuido que en un entorno centralizado?
- 18.5 Imagine que nuestra empresa está construyendo una red informática y que se nos pide que escribamos un esquema para tratar con el problema de los interbloqueos.
- ¿Utilizaría un esquema de detección de interbloqueos o un esquema de prevención de interbloqueos?
 - Si fuera a utilizar un esquema de prevención de interbloqueos, ¿cuál utilizaría? Razona tu respuesta.
 - Si fuera a utilizar un esquema de detección de interbloqueos, ¿cuál utilizaría? Razona tu respuesta.
- 18.6 ¿En qué circunstancias se comporta mejor el esquema espera-muere que el esquema herido-expresa, a la hora de conceder recursos a una serie de transacciones que se estén ejecutando de manera concurrente?
- 18.7 Considera las técnicas centralizada y completamente distribuida de detección de interbloqueos. Compare los dos algoritmos en términos de la complejidad de los mensajes.
- 18.8 Considera el siguiente algoritmo jerárquico de detección de interbloqueos, en el que el grafo de espera global está distribuido entre una serie de controladores diferentes, que están organizados en forma de árbol. Cada controlador que no sea una hoja del árbol, mantiene un grafo de espera que contiene información relevante correspondiente a los grafos de los controladores contenidos en el subárbol que se encuentra por debajo suyo. En particular, sean S_A , S_B y S_C una serie de controladores tales que S_C es el ancestro común más bajo de S_A y S_B (S_C debe ser único, ya que estamos considerando el caso de un árbol). Supongamos que el nodo T_i aparece en el grafo de espera local de los controladores S_A y S_B . Entonces T_i también debe aparecer en el grafo de espera local de
- el controlador S_C
 - todos los controladores situados en la trayectoria que va desde S_C a S_A
 - todos los controladores situados en la trayectoria que va desde S_C a S_B
- Además, si T_i y T_j aparecen en el grafo de espera del controlador S_D y existe una trayectoria desde T_i a T_j en el grafo de espera de uno de los hijos de S_D , entonces deberá existir una arista $T_i \rightarrow T_j$ en el grafo de espera de S_D .
- Demuestra que, si existe un ciclo en cualquiera de los grafos de espera, entonces el sistema está interbloqueado.
- 18.9 Diseña un algoritmo de elección para anillos bidireccionales que sea más eficiente que el que hemos presentado en este capítulo. ¿Cuántos mensajes se necesitan para n procesos?
- 18.10 Considera una configuración en la que los procesadores no estén asociados con identificadores únicos, pero en la que el número total de procesadores sea conocido y los procesadores estén organizados en forma de un anillo bidireccional. ¿Resulta posible diseñar un algoritmo de elección para esta configuración?
- 18.11 Considera un fallo que tenga lugar durante el algoritmo 2PC para una transacción. Para cada posible fallo, explícate de qué manera garantiza el algoritmo 2PC la atomicidad de la transacción a pesar del fallo.
- 18.12 Considera el siguiente modelo de fallo para una serie de procesadores fallidos. Los procesadores siguen el protocolo pero pueden fallar en instantes inesperados de tiempo. Cuando los procesadores fallan, simplemente dejan de funcionar y no continúan participando en el sistema distribuido. Dado este modelo de fallo, diseña un algoritmo para alcanzar un acuerdo entre un conjunto de procesadores. Analiza las condiciones bajo las que podrá alcanzarse dicho acuerdo.

Notas bibliográficas

El algoritmo distribuido para extender la relación *ha sucedido antes* y transformarla en una ordenación total coherente de todos los sucesos del sistema fue desarrollada por Lamport [1978]. Puede encontrar análisis adicionales relativos a la utilización del tiempo lógico para caracterizar un comportamiento de los sistemas distribuidos en Fidge [1991], Raynal y Singhal [1996], Babaoglu y Marzullo [1993], Schwarz y Mattern [1994], y Mattern [1988].

El primer algoritmo general para la implementación de la exclusión mutua en un entorno distribuido también fue desarrollado por Lamport [1978a]. El esquema de Lamport requiere $3 \times (n - 1)$ mensajes por cada entrada en una sección crítica. Subsiguientemente, Ricart y Agrawala [1981] propusieron un algoritmo distribuido que sólo requiere $2 \times (n - 1)$ mensajes. Su algoritmo se presenta en la Sección 18.2.2. Un algoritmo proporcional a la raíz cuadrada para la exclusión mutua en sistemas distribuidos es el descrito por Maekawa [1985]. El algoritmo de paso de testigo para sistemas con estructura de anillo presentado en la Sección 18.2.3 fue desarrollado por Lelann [1977]. Carvalho y Roucairol [1983] expusieron la exclusión mutua en las redes informáticas y Agrawal y Abbadi [1991] describen una solución eficiente y tolerante a fallos para la exclusión mutua distribuida. Raynal [1991] presenta una taxonomía simple de los archivos distribuidos de exclusión mutua.

La cuestión de la sincronización distribuida fue analizada en Reed y Kanodia [1979] (entorno de memoria compartida), Lamport [1978a], Lamport [1978b] y Schneider [1982] (procesos completamente disjuntos). Una solución distribuida al problema de la cena de los filósofos es la presentada por Chang [1980].

El protocolo 2PC fue desarrollado por Lampson y Sturgis [1976] y Gray [1978]. Mohan [1983] analizó dos versiones modificadas de 2PC, denominadas “presuposición de la confirmación” y “presuposición del aborto”, que reducen la sobrecarga de 2PC por el procedimiento de definir suposiciones predeterminadas en lo que respecta al destino de las transacciones.

Algunos artículos que tratan los problemas de implementar el concepto de transacción en una base de datos distribuida fueron presentados por Gray [1981], Traiger et al. [1982] y Spector y Schwarz [1983]. Puede encontrar análisis detallados sobre el tema del control de concurrencia en sistemas distribuidos en Bernstein et al. [1987]. En Rosenkrantz et al. [1978] analizan el algoritmo de prevención de interbloqueos distribuido basado en marcas temporales. El esquema de detección de interbloqueos completamente distribuido presentado en la Sección 18.5.2 fue desarrollado por Obermarck [1982]. El esquema jerárquico de detección de interbloqueos del Ejercicio 18.4 apareció en Menasce y Muntz [1979]. Knapp [1987] y Singhal [1989] incluyen un repaso del tema de la detección de interbloqueos en sistemas distribuidos. Los interbloqueos también pueden detectarse tomando instantáneas de un sistema distribuido, como se explica en Chandy [1985].

El problema de los generales bizantinos se expone en Lamport et al. [1982] y Pease et al. [1980]. El algoritmo de imposición fue presentado por García-Molina [1982] y el algoritmo de elección para un sistema con estructura de anillo fue escrito por Lelann [1977].

Parte Siete

Sistemas de propósito especial

Nuestro tratamiento de los temas de sistemas operativos se ha centrado hasta ahora principalmente en los sistemas informáticos de propósito general. Existen, sin embargo, sistemas de propósito especial cuyos requisitos son distintos de los de muchos de los sistemas que hemos descrito hasta ahora.

Un *sistema de tiempo real* es un sistema informático que no sólo requiere que los resultados calculados sean "correctos", sino también que esos resultados se produzcan dentro de un periodo de tiempo especificado. Los resultados producidos después de finalizar ese período pueden no tener (incluso aunque sean correctos) ningún valor útil. Para dichos sistemas, es necesario modificar muchos algoritmos de planificación de los sistemas operativos tradicionales, con el fin de cumplir con los estrictos requisitos de temporización.

Un *sistema multimedia* debe ser capaz de manejar no sólo datos convencionales, como archivos de texto, programas y documentos de procesadores de texto, sino también datos multimedia. Los datos multimedia están compuestos por datos de flujo continuo (audio y vídeo) además de por datos convencionales. Los datos de flujo continuo (como las imágenes de un vídeo) deben suministrarse respetando ciertas restricciones de tiempo (por ejemplo, 25 imágenes por segundo). Las demandas relativas a la gestión de datos de flujo continuo requieren realizar cambios significativos en la estructura de los sistemas operativos, especialmente en lo que respecta a la gestión de la memoria, del disco y de la red.

Sistemas de tiempo real

Nuestro tratamiento de los temas relativos a los sistemas operativos se ha centrado principalmente en los sistemas informáticos de propósito general (por ejemplo, los sistemas de sobremesa y sistemas servidores). En este capítulo, vamos a volver nuestra atención a los sistemas informáticos de tiempo real. Los requisitos de los sistemas de tiempo real difieren de los de muchos de los sistemas que hemos descrito hasta el momento, principalmente porque los sistemas de tiempo real deben producir los resultados dentro de ciertos límites de tiempo. En este capítulo, vamos a proporcionar una panorámica de los sistemas informáticos en tiempo real y a describir cómo deben construirse para satisfacer los estrictos requisitos de tiempo de estos sistemas.

OBJETIVOS DEL CAPÍTULO

- Explicar los requisitos de temporización de los sistemas de tiempo real.
- Distinguir entre sistemas de tiempo real estrictos y no estrictos.
- Explicar las características que definen a los sistemas de tiempo real.
- Describir los algoritmos de planificación para los sistemas de tiempo real estrictos.

19.1 Introducción

Un sistema de tiempo real es un sistema informático que no sólo requiere que los resultados calculados sean “correctos”, sino que también esos resultados se produzcan dentro de un período de tiempo especificado. Los resultados producidos después de que ese período de tiempo haya transcurrido pueden no tener (incluso aunque sean correctos) ningún valor real. Como ilustración, considere un robot autónomo que se encarga de distribuir el correo en un complejo de oficinas. Si el sistema de control de visión identifica una pared *después* de que el robot haya impactado contra ella, el sistema no habrá satisfecho los requisitos previstos, incluso aunque haya identificado correctamente la pared. Compare este requisito de temporización con las demandas mucho menos estrictas de otros sistemas. En un sistema informático de sobremesa interactivo, resulta deseable proporcionar un rápido tiempo de respuesta al usuario interactivo, pero no es obligatorio hacerlo. Algunos sistemas (como los sistemas de procesamiento por lotes) pueden que no tengan, incluso, ningún requisito de temporización en absoluto.

Los sistemas de tiempo real que se ejecutan sobre hardware informático tradicional se utilizan en un amplio rango de aplicaciones. Además, muchos sistemas de tiempo real están integrados en “dispositivos especializados”, como electrodomésticos normales, (por ejemplo, hornos de microondas y lavavajillas), dispositivos digitales de consumo (por ejemplo, cámaras y reproductores MP3) y dispositivos de comunicaciones (por ejemplo, teléfonos móviles y dispositivos de mano Black-berry). También están presentes en otros aparatos de mayor tamaño, como automóviles o

aeroplanos. Un **sistema integrado** es un dispositivo informático que forma parte de un sistema de mayor tamaño, en el que la presencia de ese dispositivo informático a menudo no resulta obvia para el usuario.

Como ejemplo, considere un sistema integrado para el control de un lavavajillas. El sistema integrado puede ofrecer diversas opciones para planificar la operación del lavavajillas: la temperatura del agua, el tipo de lavado (ligero o en profundidad) o incluso un temporizador que indica cuándo debe comenzar a funcionar el lavavajillas. Lo más probable es que el usuario del lavavajillas no sea consciente de que existe una computadora integrada en el electrodoméstico. Como ejemplo adicional, considere un sistema integrado que controla el sistema antibloqueo de los frenos en un automóvil. Cada rueda del automóvil tiene un sensor que detecta el grado de deslizamiento y de tracción en cada momento y cada sensor envía continuamente sus datos al controlador del sistema. Tomando los resultados de estos sensores, el controlador le dice al mecanismo de frenado de cada rueda cuanta presión de frenado debe aplicar. De nuevo, para el usuario (en este caso, el conductor del vehículo), la presencia de un sistema informático integrado puede no resultar aparente. Sin embargo, es importante destacar, que no todos los sistemas integrados son sistemas de tiempo real. Por ejemplo, un sistema integrado que controle un horno doméstico puede no tener ningún requisito de tiempo real en absoluto.

Algunos sistemas de tiempo real se identifican como **sistemas de seguridad crítica**. En un sistema de seguridad crítica, la operación incorrecta (usualmente debido a que no se cumple con un requisito de temporización) provoca algún tipo de "catástrofe". Como ejemplos de sistemas de seguridad crítica podemos citar los sistemas armamentísticos, los sistemas antibloqueo de los frenos, los sistemas de gestión de vuelo, los sistemas integrados que tengan relación con la salud, como por ejemplo los marcapasos. En estos escenarios, el sistema de tiempo real *debe* responder a los sucesos con los requisitos de temporización especificados; en caso contrario, podrían producirse graves daños o incluso algo peor. Sin embargo, una mayoría significativa de los sistemas integrados no son sistemas de seguridad crítica, incluyendo por ejemplo las máquinas fax, los hornos microondas, los relojes de pulsera y los dispositivos de red como conmutadores y encaminadores. Para estos dispositivos, si no se cumplen los requisitos de temporización, el peor de los resultados posibles es, en muchas ocasiones, un usuario insatisfecho.

La informática de tiempo de real puede clasificarse en dos tipos distintos: estricta y no estricta. Un **sistema de tiempo real estricto** tiene los requisitos más fuertes, que garantizan que las tareas de tiempo real críticas se completen dentro del período especificado. Los sistemas de seguridad crítica son, normalmente, sistemas de tiempo real estrictos. Los **sistemas de tiempo real no estrictos** no tienen requisitos tan fuertes, y se limitan a garantizar que las tareas de tiempo real críticas tengan prioridad sobre otras tareas y que retengán dicha prioridad hasta complementarse. Muchos sistemas operativos comerciales, así como Linux, proporcionan soporte de tiempo real no estricto.

19.2 Características del sistema

En esta sección, vamos a explorar las características de los sistemas de tiempo real y a analizar diversas cuestiones relacionadas con el diseño de sistemas operativos de tiempo real tanto estrictos como no estrictos.

Entre las características típicas de muchos sistemas operativos de tiempo real podemos citar:

- Tienen un único propósito.
- Son de pequeño tamaño.
- Son de bajo coste y se producen en masa.
- Tienen requisitos de temporización específicos.

A continuación, vamos a examinar cada una de estas características.

A diferencia de las máquinas de tipo PC, que se pueden utilizar para muchas cosas, un sistema en tiempo real sirve normalmente a un único propósito, como por ejemplo controlar el sistema

antibloqueo de los frenos o reproducir música en un reproductor MP3. ¡Resulta bastante poco probable que un sistema de tiempo real que controle el sistema de navegación de una aeronave se utilice también para reproducir un DVD! El diseño de un sistema operativo de tiempo real refleja su naturaleza concentrada en un único propósito y es a menudo bastante simple.

Existen muchos sistemas de tiempo real en entornos en los que el espacio físico está restringido. Considere la cantidad de espacio disponible en un reloj de pulsera o en un horno microondas: es considerablemente menor que el que hay disponible en una computadora de sobremesa. Debido a las restricciones de espacio, la mayoría de los sistemas de tiempo real carecen tanto de la potencia de procesamiento de la CPU como de la cantidad de memoria disponibles en las computadoras de sobremesa estándar. Mientras que la mayoría de los sistemas contemporáneos de sobremesa y de servidor utilizan procesadores de 32 o 64 bits, muchos sistemas de tiempo real se ejecutan sobre procesadores de 8 o 16 bits. De forma similar, un PC de sobremesa puede tener varios gigabytes de memoria física, mientras que un sistema de tiempo real puede tener menos de un megabyte. Denominamos **huella** de un sistema a la cantidad de memoria necesaria para ejecutar un sistema operativo y sus aplicaciones. Puesto que la cantidad de memoria es limitada, la mayoría de los sistemas operativos deben tener huellas de pequeño tamaño.

A continuación, pensemos en dónde se implementan los sistemas en tiempo real. A menudo los podemos encontrar en electrodomésticos y dispositivos de consumo. Dispositivos como cámaras digitales, hornos de microondas y termostatos se fabrican en masa dentro de entornos de mercado en los que el coste es un factor fundamental. Por tanto, los microprocesadores para los sistemas de tiempo real también tienen que poder fabricarse masivamente y a bajo coste.

Una técnica para reducir el coste de un controlador integrado consiste en emplear una técnica alternativa para organizar los componentes del sistema informático. En lugar de organizar la computadora de acuerdo con la estructura mostrada en la Figura 19.1, en la que una serie de buses proporcionan el mecanismo de interconexión con los componentes individuales, muchos controladores para sistemas integrados utilizan una estrategia conocida con el nombre de **sistema en un chip** (SOC, system-on-chip). En este tipo de sistemas, la CPU, la memoria (incluyendo la caché), la unidad de gestión de memoria (MMU, memory-management-unit) y los puertos periféricos asociados, como los puertos USB, están contenidos en un único circuito integrado. Esta estrategia SOC suele ser menos costosa que la organización orientada a bus de la Figura 19.1.

Fijémonos ahora en la característica final de los sistemas de tiempo real que hemos identificado anteriormente: los requisitos específicos de temporización. Se trata, de hecho, de la característica definitoria de este tipo de sistemas. Correspondientemente, la característica definitoria de los sistemas operativos de tiempo real tanto estrictos como no estrictos consiste en dar soporte a los requisitos de temporización de las tareas en tiempo real, y en el resto del capítulo nos vamos a centrar en esta cuestión. Los sistemas operativos de tiempo real cumplen con los requisitos de temporización utilizando algoritmos de planificación que proporcionan a los procesos de tiempo real las más altas prioridades de planificación. Además, los planificadores deben garantizar que

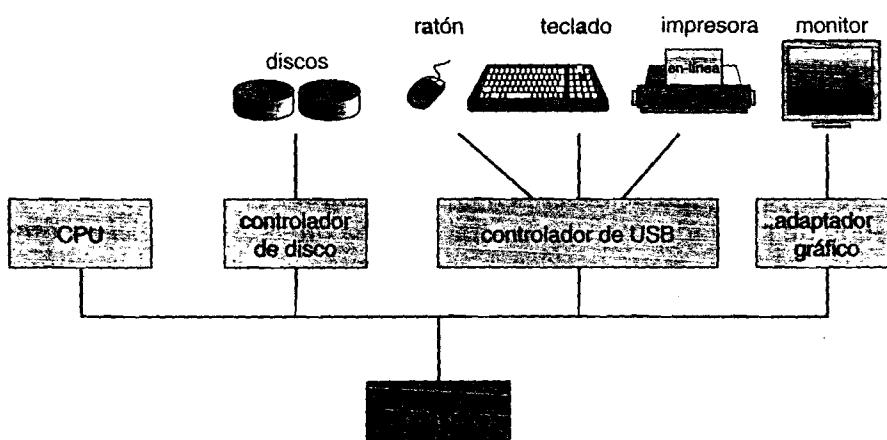


Figura 19.1 Organización orientada a bus.

la prioridad de una tarea de tiempo real no se degrada con el tiempo. Una segunda técnica, en cierto modo relacionada con la anterior, que se utiliza para satisfacer los requisitos de temporización consiste en minimizar el tiempo de respuesta a sucesos como las interrupciones.

19.3 Características de un kernel de tiempo real

En esta sección, vamos a analizar las características necesarias para diseñar un sistema operativo que permita procesos en tiempo real. Sin embargo, antes de comenzar, vamos a analizar lo que normalmente *no* se necesita para un sistema en tiempo real. Comenzaremos examinando varias de las características proporcionadas en muchos de los sistemas operativos que hemos analizado hasta ahora en el texto, incluyendo Linux, UNIX y las diversas versiones de Windows. Estos sistemas proporcionan normalmente soporte para las siguientes características:

- Una diversidad de dispositivos periféricos, tales como pantallas gráficas, unidades de CD y de DVD.
- Mecanismos de protección y seguridad.
- Múltiples usuarios.

Soportar estas características requiere a menudo un *kernel* sofisticado y de gran tamaño, por ejemplo, Windows XP tiene más de 40 millones de líneas de código fuente. Por contraste, un sistema operativo de tiempo real típico suele tener un diseño muy simple, que a menudo está escrito en miles, en lugar de en millones, de líneas de código fuente. No cabe razonablemente esperar que estos sistemas simples puedan soportar las características que hemos enumerado anteriormente.

¿Pero por qué los sistemas de tiempo real no proporcionan estas características, que son cruciales para los sistemas estándar de sobremesa y de servidor? Existen varias razones, pero las fundamentales son tres. En primer lugar, porque la mayoría de los sistemas en tiempo real sirven a un único propósito, por lo que no requieren muchas de las características que podemos encontrar en un PC de sobremesa. Considere un reloj de pulsera digital: obviamente, no necesita disponer de una unidad de disco o de DVD, ni tampoco una memoria virtual. Además, un sistema de tiempo real no incluye la noción de usuario: el sistema simplemente permite un número pequeño de tareas, que a menudo están a la espera de una serie de entradas que tienen que recibir de dispositivos hardware (sensores, aparatos de visión, etc.). En segundo lugar, las características soportadas por los sistemas operativos estándar de sobremesa son imposibles de proporcionar sin utilizar procesadores rápidos y grandes cantidades de memoria. Tanto unos como otra no están disponibles en los sistemas de tiempo de real debido a restricciones de espacio, como hemos explicado anteriormente. Además, muchos sistemas de tiempo real carecen del suficiente espacio como para soportar unidades de disco periféricas o pantallas gráficas, aunque algunos sistemas admiten sistemas de archivos utilizando memoria no volátil (NVRAM). En tercer lugar, soportar las características que resultan comunes en los entornos informáticos estándar de sobremesa incrementaría enormemente el coste de los sistemas de tiempo real, lo que podría hacer que dichos sistemas no resultaran factibles desde un punto de vista económico.

Hay una serie de consideraciones adicionales que entran en juego cuando consideramos el tema de la memoria virtual en un sistema de tiempo real. Proporcionar las características de memoria virtual que se describen en el Capítulo 9 requiere que el sistema incluya una unidad de gestión de memoria (MMU) para traducir las direcciones lógicas a direcciones físicas. Sin embargo, las unidades MMU tienden a incrementar el coste y el consumo de potencia del sistema. Además, el tiempo requerido para traducir las direcciones lógicas a las direcciones físicas, especialmente en el caso de que se produzca un fallo de búfer de traducción (TLB, translation lookaside buffer), puede ser prohibitivo en un entorno de tiempo real. En el resto de esta sección, vamos a examinar varias técnicas que se utilizan para traducir direcciones en los sistemas de tiempo real.

La Figura 19.2 ilustra tres diferentes estrategias para la gestión de traducción de direcciones que los diseñadores de sistemas operativos en tiempo real pueden utilizar. En este escenario, la

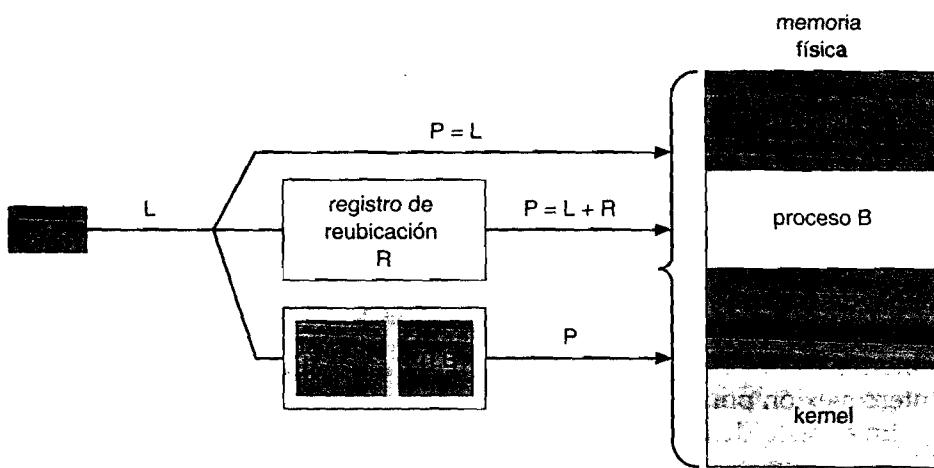


Figura 19.2 Traducción de direcciones en los sistemas de tiempo real.

CPU genera la dirección lógica L que debe hacerse corresponder con la dirección física P . La primera técnica consiste en puentear las direcciones lógicas y hacer que la CPU genere directamente las direcciones físicas. Esta técnica (denominada **modo de direccionamiento real**) no emplea mecanismos de memoria virtual y equivale, en la práctica, a decir que P equivale a L . Un problema con el modo de direccionamiento real es la ausencia de protección de memoria entre los procesos. El modo de direccionamiento real puede también requerir que los programadores especifiquen la ubicación física donde deben cargarse sus programas en memoria. Sin embargo, la ventaja de esta técnica es que el sistema es bastante rápido, ya que no se pierde ningún tiempo en traducir las direcciones. El modo de direccionamiento real resulta bastante común en los sistemas integrados con restricciones de tiempo real estrictas. De hecho, algunos sistemas operativos de tiempo real que se ejecutan sobre microprocesadores que contienen una MMU suelen desactivar la MMU para incrementar la velocidad haciendo directamente referencia a las direcciones físicas.

Una segunda estrategia para traducir las direcciones consiste en utilizar una técnica similar a la del registro de reubicación dinámica mostrado en la Figura 8.4. En este escenario, se configura un registro de reubicación R con la ubicación de memoria en la que se ha cargado un programa. La dirección física P se genera sumando el contenido del registro de reubicación R con L . Algunos sistemas de tiempo real configuran la MMU para que funcione de esta forma. La ventaja obvia de esta estrategia es que la MMU puede traducir fácilmente las direcciones lógicas a direcciones físicas utilizando la ecuación $P = L + R$. Sin embargo, este sistema continuará sufriendo la desventaja de que no existe protección de memoria entre los procesos.

La última técnica consiste en que el sistema de tiempo real proporcione una funcionalidad completa de memoria virtual, como se describe en el Capítulo 9. En este caso, la traducción de direcciones tiene lugar mediante tablas de páginas y un búfer de traducción o TLD. Además de permitir cargar un programa en cualquier posición de memoria, esta estrategia también proporciona protección de memoria entre los procesos. Para los sistemas que no tengan unidades de disco conectadas, puede que no sea posible implementar los mecanismos de paginación bajo demanda y de intercambio. Sin embargo, los sistemas pueden proporcionar dichas características utilizando memorias flash NVRAM. Los sistemas LynxOS y OnCore Systems son ejemplos de sistemas operativos de tiempo real que proporcionan soporte completo de memoria virtual.

19.4 Implementación de sistemas operativos de tiempo real

Teniendo presentes las muchas posibles variaciones, vamos a identificar ahora las características necesarias para implementar un sistema operativo en tiempo real. Esta lista no pretende ser exhaustiva, algunos sistemas proporcionan más características de las que a continuación enumeramos, mientras que otros sistemas proporcionan menos:

- Planificación apropiativa basada en prioridades.
- *Kernel* apropiativo.
- Latencia minimizada.

Una característica notable que hemos omitido de esta lista es el soporte para la conexión por red. Sin embargo, decidir si deben soportarse protocolos de red, como por ejemplo TCP/IP, resulta bastante simple: si el sistema de tiempo real debe conectarse a una red, el sistema operativo deberá proporcionar capacidades de interconexión por red. Por ejemplo, un sistema que recopile datos de tiempo real y los transmita a un servidor, deberá obviamente incluir características de interconexión por red. Alternativamente, un sistema integrado autocontenido que no requiera ninguna interacción con otros sistemas informáticos no tendrá ningún requisito obvio para la interconexión por red.

En el resto de esta sección, examinaremos los requisitos básicos enumerados anteriormente e identificaremos cómo se les puede implementar en un sistema operativo de tiempo real.

19.4.1 Planificación basada en prioridades

La característica más importante de un sistema operativo de tiempo real consiste en responder inmediatamente a un proceso de tiempo real, en cuanto dicho proceso necesite la CPU. Como resultado, el planificador para un sistema de tiempo real debe permitir un algoritmo basado en prioridades con apropiación. Recuerde que los algoritmos de planificación basados en prioridades asignan a cada proceso una prioridad, dependiendo de su importancia, a las tareas más importantes se les asignan prioridades más altas que a aquellas que se considera que son menos importantes. Si el planificador también permite un mecanismo de apropiación, un proceso que se esté ejecutando actualmente en la CPU será desalojado si otro proceso de mayor prioridad pasa a estar disponible para ejecución.

Los algoritmos de planificación apropiativos basados en prioridades se han tratado en detalle en el Capítulo 5, donde presentamos ejemplos de las características de planificación de tiempo real no estricto de los sistemas operativos Solaris, Windows XP y Linux. Cada uno de estos sistemas asigna a los procesos de tiempo real la más alta prioridad de planificación. Por ejemplo, Windows XP tiene 32 niveles diferentes de prioridad; los niveles más altos (valores de prioridad 16 a 31) están reservados para los procesos de tiempo real. Solaris y Linux tienen esquemas de asignación de prioridades similares.

Observe sin embargo, que proporcionar un planificador apropiativo basado en prioridades sólo garantiza una funcionalidad de tiempo real no estricta. Los sistemas de tiempo real estrictos deben además garantizar que las tareas de tiempo real reciban servicio de acuerdo con sus requisitos de temporización, y para poder proporcionar tales garantías pueden necesitar características de planificación adicionales. En la Sección 19.5 analizamos los algoritmos de planificación adecuados para los sistemas de tiempo real estrictos.

19.4.2 Kernels apropiativos

Los *kernels* no apropiativos no permiten desalojar un proceso que se esté ejecutando en modo *kernel*; el proceso en modo *kernel* continuará ejecutándose hasta que salga del modo *kernel*, se bloquee o voluntariamente ceda el control de la CPU. Por el contrario, un *kernel* apropiativo permitirá desalojar una tarea que se esté ejecutando en modo *kernel*. El diseño de un *kernel* apropiativo puede resultar bastante difícil, y las aplicaciones tradicionales orientadas al usuario, como hojas de cálculo, procesadores de texto y exploradores web normalmente no requieren esos rápidos tiempos de respuesta. Como resultado, algunos sistemas operativos de sobremesa convencionales (como Windows XP) son no apropiativos.

Sin embargo, para satisfacer los requisitos de temporización de los sistemas de tiempo real (y en particular de los sistemas de tiempo real estrictos) resulta obligatorio utilizar un *kernel* apropiativo. En caso contrario, una tarea de tiempo real podría tener que esperar un período de tiempo arbitrariamente largo, mientras hubiera otra tarea activa en el *kernel*.

Existen diversas estrategias para hacer que un *kernel* sea apropiativo. Una de ellas consiste en insertar **puntos de desalojo** en las llamadas al sistema de larga duración. En cada punto de desalojo se comprueba si existe la necesidad de ejecutar un proceso de alta prioridad. En caso de que exista, se produce un cambio de contexto. Después, cuando el proceso de alta prioridad termine de ejecutarse, el proceso interrumpido continuará con su llamada al sistema. Los puntos de desalojo sólo pueden colocarse en puntos *seguros* del *kernel*, es decir, sólo en aquellos puntos donde las estructuras del *kernel* no estén siendo modificadas. Una segunda estrategia para hacer que un *kernel* sea apropiativo consiste en utilizar mecanismos de sincronización, como los que hemos descrito en el Capítulo 6. Con este método, el *kernel* siempre puede ser apropiativo, porque cualquier dato del *kernel* que se esté actualizando estará protegido frente a posibles modificaciones por parte del proceso de alta prioridad.

19.4.3 Minimización de la latencia

Consideremos la naturaleza dirigida por sucesos de un sistema de tiempo real: normalmente, el sistema está esperando a que se produzca un suceso en tiempo real. Los sucesos pueden producirse en el software (por ejemplo, cuando caduca un contador) o en el hardware (como por ejemplo cuando un vehículo con control remoto detecta que se está aproximando a un rectángulo). Cuando se produce un suceso, el sistema debe responder a él y darle servicio lo más rápidamente posible. Denominamos **latencia del suceso** a la cantidad de tiempo que transcurre desde el momento que tiene lugar el suceso hasta el momento en el que se le da servicio (Figura 19.3).

Normalmente, los diferentes sucesos tienen distintos requisitos de latencia. Por ejemplo, el requisito de latencia para un sistema antibloqueo de frenos puede ser de entre 3 y 5 milisegundos, lo que quiere decir que desde el momento en que una rueda detecta por primera vez que está deslizándose, el sistema que controla el mecanismo de antibloqueo de los frenos tiene entre 3 y 5 milisegundos para responder a esa situación y controlarla. Cualquier respuesta que requiera más tiempo puede provocar que el vehículo quede fuera de control. Por contraste, un sistema integrado que controle el radar en una aeronave puede tolerar un período de latencia de varios segundos.

Hay dos tipos de latencias que afectan al rendimiento de los sistemas de tiempo real:

1. Latencia de interrupción.
2. Latencia de despacho.

La **latencia de interrupción** se refiere al período de tiempo que transcurre entre la llegada de una interrupción a la CPU y el instante en que comienza la rutina de servicio de dicha interrupción. Cuando se produce una interrupción, el sistema operativo debe primero completar la instrucción que está ejecutando y determinar el tipo de interrupción que ha ocurrido. Luego, debe guardar el estado del proceso actual antes de dar servicio a la interrupción utilizando la rutina de servicio de interrupción (ISR, interrupt service routine) específica. El tiempo total requerido para

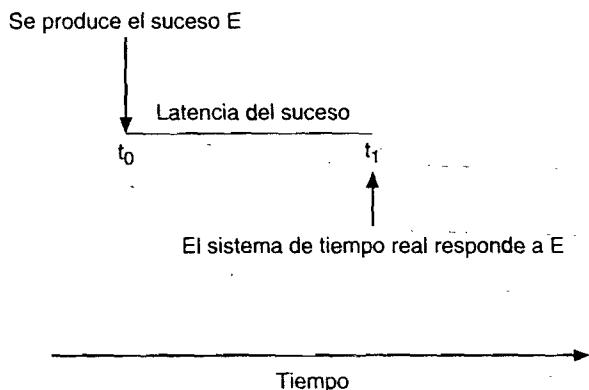


Figura 19.3 Latencia de un suceso.

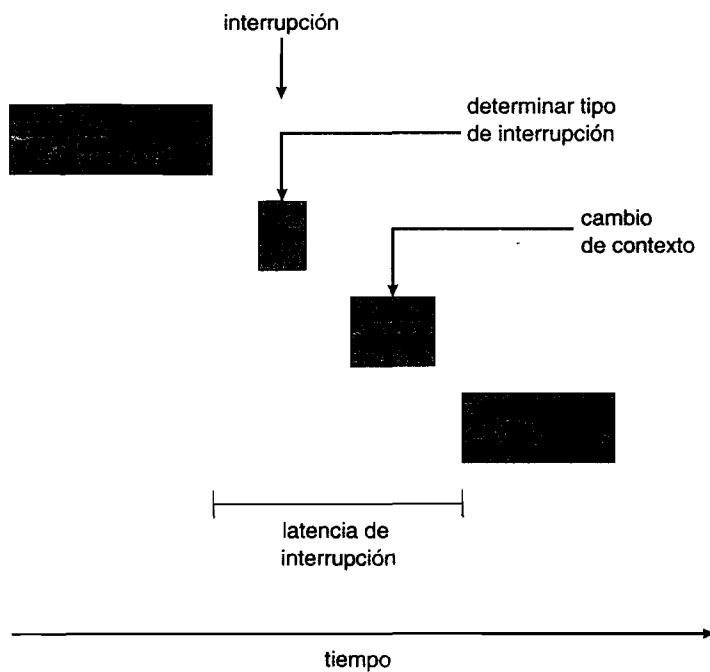


Figura 19.4 Latencia de interrupción.

realizar estas tareas es la latencia de interrupción (Figura 19.4). Obviamente, para los sistemas operativos de tiempo real resulta crucial minimizar la latencia de interrupción, con el fin de garantizar que las tareas de tiempo real reciban una atención inmediata.

Un factor importante que contribuye a la latencia de interrupción es la cantidad de tiempo que las interrupciones pueden estar desactivadas mientras se actualizan las estructuras de datos del *kernel*. Los sistemas operativos de tiempo real requieren que las interrupciones estén desactivadas durante períodos muy cortos de tiempo. Sin embargo, para los sistemas de tiempo real estrictos, no sólo es necesario minimizar la latencia de interrupción, sino que de hecho es preciso acotarla, con el fin de garantizar el comportamiento determinista que los *kernels* de tiempo real estricto deben exhibir.

La cantidad de tiempo requerida para que el despachador de planificación detenga un proceso e inicie otro se conoce como **latencia de despacho**. Para proporcionar a las tareas de tiempo real un acceso inmediato a la CPU, es obligatorio que el sistema operativo de tiempo real minimice esta latencia. La técnica más efectiva para tener una latencia de despacho bajo consiste en proporcionar un *kernel* apropiativo.

En la Figura 19.5 se muestra un diagrama que ilustra el concepto de latencia de despacho. La fase de conflicto de la latencia de despacho tiene dos componentes:

1. El desalojo de cualquier proceso que se esté ejecutando en el *kernel*.
2. La liberación por parte de los procesos de baja prioridad de los recursos necesarios para el proceso de alta prioridad.

Como ejemplo, en Solaris, la latencia de despacho con el mecanismo de apropiación desactivado es de más de 100 milisegundos. Si se activa el mecanismo de apropiación, esta latencia se reduce a menos de un milisegundo.

Una cuestión que puede afectar a la latencia de despacho es la que surge cuando un proceso de mayor prioridad necesita leer o modificar datos del *kernel* a los que esté actualmente accediendo un proceso de menor prioridad (o una cadena de procesos de menor prioridad). Puesto que los datos del *kernel* están normalmente protegidos mediante un cerrojo, el proceso de mayor prioridad tendrá que esperar a que otro de menor prioridad finalice con el recurso. Esta situación se complica aún más si el proceso de menor prioridad es desalojado en favor de otro proceso que

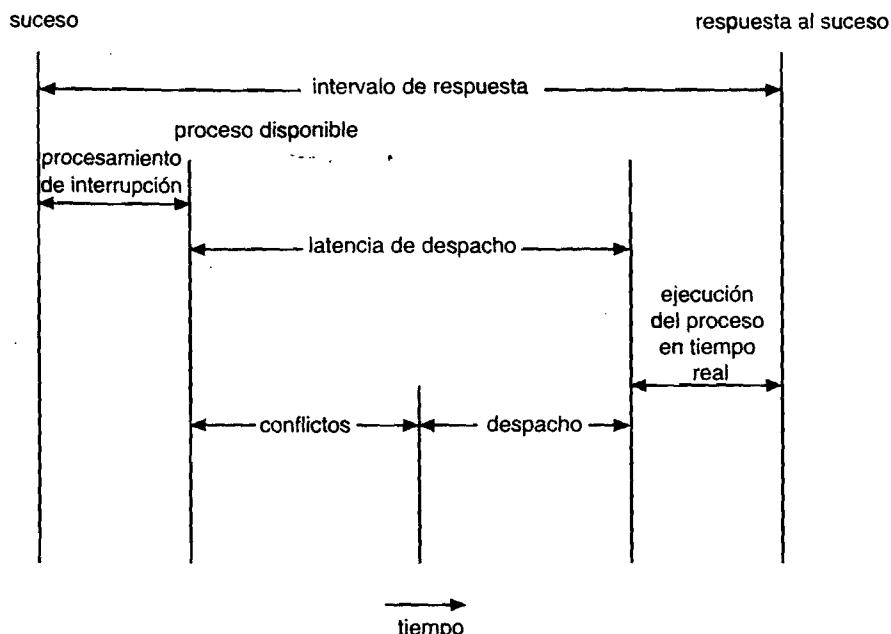


Figura 19.5 Latencia de despacho.

tenga una prioridad mayor. Como ejemplo, vamos a suponer que tenemos tres procesos L , M y H , cuyas prioridades siguen el orden $L < M < H$. Supongamos que el proceso H requiere el recurso R , al que está actualmente accediendo el proceso L . Normalmente, el proceso H esperaría a que L finalizara de utilizar el recurso R . Sin embargo, suponga ahora que el proceso M pasa a ser ejecutable desalojando así al proceso L . Indirectamente, un proceso con una menor prioridad (el proceso M) ha afectado al tiempo que el proceso H debe esperar para que L libere el recurso R .

Este problema, conocido como **inversión de prioridades**, puede resolverse utilizando el denominado **protocolo de herencia de prioridades**. De acuerdo con este protocolo, todos los procesos que estén accediendo a recursos que necesite otro proceso de mayor prioridad heredan esa mayor prioridad hasta que hayan terminado con los recursos en cuestión. Una vez que han terminado, sus prioridades revierten a los valores originales. En el ejemplo anterior, un protocolo de herencia de prioridad permitiría al proceso L heredar temporalmente la prioridad del proceso H , evitando así que el proceso M se apropiara de la CPU. Cuando el proceso L hubiera terminado de utilizar el recurso R , renunciaría a su prioridad heredada de H y asumiría su prioridad original. Como el recurso R está ahora disponible, es el proceso H (y no M) el que se ejecutará a continuación.

19.5 Planificación de la CPU en tiempo real

Nuestro tratamiento de la planificación hasta ahora se ha centrado principalmente en los sistemas de tiempo no estrictos. Como hemos mencionado, sin embargo, el mecanismo de planificación para dichos sistemas no proporciona ninguna garantía sobre cuándo se planificará para ejecución un proceso crítico; el sistema tan sólo garantiza que se dará a ese proceso preferencia sobre otros procesos no críticos. Los sistemas de tiempo real estrictos tienen requisitos más fuertes: es necesario dar servicio a una tarea de acuerdo con el plazo prefijado; proporcionar ese servicio después de que pase el plazo equivale, a todos los efectos, a no proporcionar el servicio en absoluto.

Vamos a considerar ahora el tema de la planificación para los sistemas de tiempo real estrictos. Sin embargo, antes de continuar con los detalles de los algoritmos de planificación individuales, debemos definir ciertas características de los procesos que hay que planificar. En primer lugar, los procesos se consideran **periódicos**. Es decir, esos procesos requieren la CPU a intervalos constantes (periodos). Cada proceso periódico tiene un tiempo de procesamiento fijo t una vez que

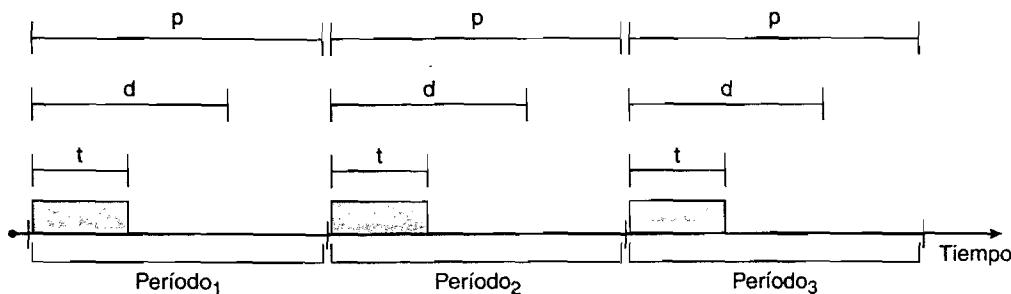


Figura 19.6 Tarea periódica.

adquiere la CPU, un plazo d antes del cual debe ser servido por la CPU y un período p . La relación del tiempo de procesamiento, el plazo y el período puede expresarse como $0 \leq t \leq d \leq p$. La **tasa** de una tarea periódica es $1/p$. La Figura 19.6 ilustra la ejecución de un proceso periódico a lo largo del tiempo. Los planificadores pueden aprovechar esta relación y asignar las prioridades de acuerdo con el plazo de un proceso periódico o con su tasa requerida.

Lo que resulta inusual acerca de esta forma de planificación es que un proceso puede tener que anunciar al planificador sus requisitos relativos al plazo de servicio. Entonces, utilizando una técnica conocida con el nombre de **algoritmo de control de admisión**, el planificador puede, o bien admitir el proceso, garantizando que se completará en el tiempo prefijado, o rechazar la solicitud como imposible, si no puede garantizar que se vaya a dar servicio a la tarea antes de alcanzar el plazo.

En las siguientes secciones, vamos a explorar los algoritmos de planificación dirigidos a tratar de satisfacer los requisitos relativos al plazo en los sistemas de tiempo real estrictos.

19.5.1 Planificación por prioridad monótona en tasa

El algoritmo de **planificación por prioridad monótona en tasa** (rate-monotonic scheduling) planifica las tareas periódicas utilizando una política de prioridad estática con apropiación. Si se está ejecutando un proceso de menor prioridad y otro proceso de mayor prioridad pasa a estar disponible para ejecución, este proceso desalojará al proceso de menor prioridad. Al entrar en el sistema, a cada tarea periódica se le asigna una prioridad que está en función inversa a su período: cuanto más corto sea el período, mayor será la prioridad y cuanto más largo sea el período, menor será la prioridad. La razón que subyace a esta política consiste en asignar una mayor prioridad a aquellas tareas que requieren la CPU más a menudo. Además, la planificación por prioridad monótona en tasa presupone que el tiempo de procesamiento de un proceso periódico es igual en cada ráfaga de la CPU, es decir, que cada vez que un proceso adquiere la CPU, la duración de su correspondiente ráfaga de CPU es la misma.

Vamos a considerar un ejemplo. Tenemos dos procesos P_1 y P_2 . Los períodos para P_1 y P_2 son 50 y 100, respectivamente; es decir, $p_1 = 50$ y $p_2 = 100$. Los tiempos de procesamiento son $t_1 = 20$ para P_1 y $t_2 = 35$ para P_2 . El plazo para cada proceso requiere que el proceso complete su ráfaga de CPU antes de que comience el siguiente período.

Primero debemos preguntarnos si es posible planificar estas tareas de manera que cada una de ellas satisfaga sus requisitos de plazo. Si medimos la utilización de CPU por parte de un proceso P_i como la relación entre su ráfaga de procesamiento y su período (t_i / p_i) la utilización de la CPU por parte de P_1 será $20 / 50 = 0,40$ y la de P_2 será $35 / 100 = 0,35$, dando una utilización total de la CPU del 75 por ciento. Por tanto, parece que sí que podemos planificar estas tareas de tal forma que ambas cumplan con sus plazos y sigan dejando a la CPU una serie de ciclos disponibles.

En primer lugar, vamos a suponer que asignamos a P_2 una mayor prioridad que a P_1 . La ejecución de P_1 y P_2 se muestra en la Figura 19.7. Como podemos ver, P_2 comienza primero su ejecución y se completa en el instante 35. En este punto, comienza P_1 , que completa su ráfaga de CPU en el instante 55. Sin embargo, el primer plazo para P_1 era en el instante 50, por lo que el planificador ha hecho que P_1 no sea capaz de completarse antes del plazo.



Figura 19.7 Planificación de tareas cuando P_2 tiene una prioridad mayor que P_1 .

Ahora vamos a suponer que utilizamos una planificación por prioridad monótona en tasa, en la que asignamos a P_1 una mayor prioridad que a P_2 , ya que el período de P_1 es más corto que el de P_2 . La ejecución de estos procesos se muestra en la Figura 19.8. P_1 comienza primero y completa su rafaga de CPU en el instante 20, cumpliendo así con su primer plazo. P_2 comienza a ejecutarse en este punto y se ejecuta hasta el instante 50. En este momento, es desalojado por P_1 , aunque todavía faltan 5 milisegundos para completar su rafaga. P_1 completa su rafaga de CPU en el instante 70, momento en el que el planificador reanuda la ejecución de P_2 . P_2 completa su rafaga de CPU en el instante 75, cumpliendo también con su primer plazo. El sistema quedará inactivo hasta el instante 100, cuando se vuelva a planificar la ejecución de P_1 .

La planificación por prioridad monótona en tasa se considera óptima, en el sentido de que si un conjunto procesos no puede planificarse utilizando este algoritmo, no podrá ser planificado por ningún otro algoritmo que asigne prioridades estáticas. Examinemos a continuación un conjunto de procesos que no pueden planificarse utilizando el algoritmo monótono con respecto a la tasa. Supongamos que el proceso P_1 tiene un período de $p_1 = 50$ y una rafaga de CPU de $t_1 = 25$. Para P_2 , los valores correspondientes son $p_2 = 80$ y $t_2 = 35$. La planificación por prioridad monótona en tasa asignaría al proceso P_1 una prioridad más alta, ya que es el proceso que tiene el período más corto. La utilización total de la CPU por parte de los dos procesos será $(25 / 50) + (35 / 80) = 0,94$ y, por tanto, parece lógico que pudieran planificarse los dos procesos y seguir dejando un 6 por ciento de tiempo de CPU disponible. El diagrama de Gantt que muestra la planificación de los procesos P_1 y P_2 es el que aparece en la Figura 19.9. Inicialmente, P_1 se ejecuta hasta que completa su rafaga de CPU en el instante 25. Entonces empieza a ejecutarse el proceso P_2 y se ejecuta hasta el instante 50, siendo desalojado por el proceso P_1 . En este punto, P_2 sigue necesitando 10 milisegundos para terminar su rafaga de CPU. El proceso P_1 se ejecuta hasta el instante 75; sin embargo, P_2 no podrá completar su rafaga de CPU antes de su plazo, en el instante 80.

Por tanto, a pesar de ser óptima, la planificación por prioridad monótona en tasa tiene una limitación: la utilización de la CPU está acotada y no siempre es posible maximizar por completo los recursos de CPU. El caso peor de utilización de la CPU para la planificación de N procesos es:

$$2(2^{1/n} - 1)$$

Con un proceso en el sistema, la utilización de la CPU es del 100 por ciento, pero esa tasa de utilización cae aproximadamente al 69 por ciento a medida que el número de procesos se aproxima a infinito. Con dos procesos, la tasa utilización de la CPU está acotada entorno al 83 por ciento. La utilización combinada de la CPU para los dos procesos planificados en las Figuras 19.7 y 19.8 es del 75 por ciento, por lo que está garantizado que el algoritmo de planificación por prioridad monótona en tasa puede planificarlos de modo que sean capaces de cumplir con sus plazos. Sin embargo, para los procesos planificados de la Figura 19.9, la tasa combinada de utilización de la CPU es de aproximadamente el 94 por ciento, por tanto, la planificación por prioridad monótona en tasa no puede garantizar que puedan planificarse esos procesos para cumplir con sus plazos.

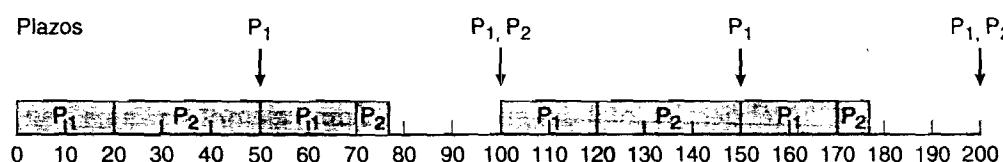


Figura 19.8 Planificación por prioridad monótona en tasa.

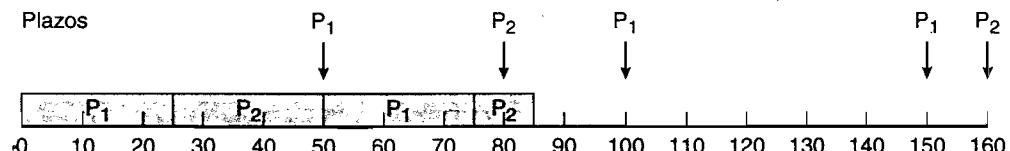


Figura 19.9 Problema de no cumplimiento de plazos con la planificación por prioridad monótona en tasa.

19.5.2 Planificación por prioridad en finalización de plazo

El algoritmo de planificación por prioridad en finalización de plazo (EDF, earliest-deadline-first) asigna dinámicamente las prioridades de acuerdo con la finalización de los plazos. Cuanto más próximo esté un plazo, mayor será la prioridad, y cuanto más lejano esté el plazo, menor será la prioridad. Con una política EDF, cuando un proceso pasa a ser ejecutable, debe anunciar sus requisitos de plazos al sistema. Puede que sea necesario ajustar las prioridades para reflejar el siguiente plazo de ese proceso que acaba de pasar a ser ejecutable. Observe que este algoritmo difiere del algoritmo por prioridad monótona en tasa en que las prioridades eran fijas.

Para ilustrar la planificación EDF, vamos a planificar de nuevo los procesos mostrados en la Figura 19.9, en la que no habíamos sido capaces de cumplir con los plazos empleando una planificación por prioridad monótona en tasa. Recuerde que P_1 tiene los valores $p_1 = 50$ y $t_1 = 25$ y que P_2 tiene los valores $p_2 = 80$ y $t_2 = 35$. La planificación EDF de estos procesos se muestra en la Figura 19.10. El proceso P_1 tiene su finalización de plazo, por lo que su prioridad inicial es más alta que la del proceso P_2 . El proceso P_2 comienza a ejecutarse al final de la ráfaga de CPU de P_1 . Sin embargo, mientras que el algoritmo de planificación por prioridad monótona en tasa permitía a P_1 desalojar a P_2 al principio de su siguiente período, en el instante 50, el algoritmo de planificación EDF permite que el proceso P_2 continúe ejecutándose. P_2 tiene ahora una mayor prioridad que P_1 porque su siguiente plazo (en el instante 80) está más próximo que el de P_1 (en el instante 100). De este modo, tanto P_1 como P_2 pueden cumplir con su primer plazo. El proceso P_1 comienza a ejecutarse de nuevo en el instante 60 y completa su segunda ráfaga de CPU en el instante 85, cumpliendo también con su segundo plazo en el instante 100. P_2 comenzará a ejecutarse en este punto, para ser desalojado por P_1 al comienzo de su siguiente período en el instante 100. P_2 es desalojado porque P_1 tiene una finalización de plazo anterior (instante 150) que P_2 (instante 160). En el instante 125, P_1 completa su ráfaga de CPU y P_2 reanuda su ejecución, finalizando en el instante 145 y cumpliendo también con su plazo. El sistema estará inactivo hasta el instante 150, en el que se planifica de nuevo la ejecución de P_1 .

A diferencia del algoritmo por prioridad monótona en tasa, la planificación EDF no requiere que los procesos sean periódicos, ni tampoco que necesiten una cantidad constante de tiempo de CPU por cada ráfaga de ejecución. El único requisito es que cada proceso indique al planificador su próximo plazo en el momento de pasar a ser ejecutable. El atractivo de la planificación EDF es que resulta óptima desde el punto de vista teórico. En teoría, permite planificar los procesos de modo que cada uno de ellos pueda cumplir con sus requisitos relativos a los plazos de ejecución, y la utilización de la CPU será del 100 por ciento. Sin embargo, en la práctica, es imposible conseguir este nivel de utilización de la CPU debido al coste de los cambios de contexto entre un proceso y el coste del tratamiento de las interrupciones.

19.5.3 Planificación con cuota proporcional

El algoritmo de planificación con cuota proporcional funciona asignando T cuotas entre todas las aplicaciones. Cada aplicación puede recibir N cuotas de tiempo, garantizando así que la aplicación tenga un N/T del tiempo total del procesador. Como ejemplo, vamos a suponer que hay un total de $T = 100$ cuotas que hay que dividir entre tres procesos, A , B y C . Asignamos 50 cuotas a A , 15 cuotas a B y 20 cuotas a C . Este esquema asegura que A tendrá un 50 por ciento del tiempo total de procesador, B tendrá un 15 por ciento y C tendrá un 20 por ciento.

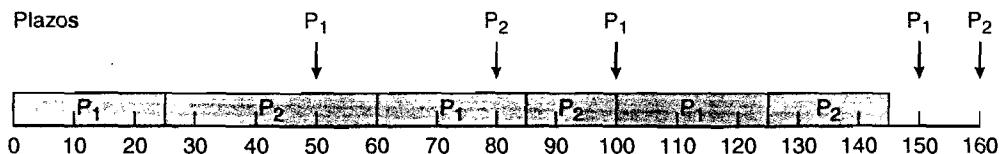


Figura 19.10 Planificación por prioridad en finalización de plazo.

El algoritmo de planificación con cuota proporcional debe trabajar en conjunción con una política de control de admisión, para garantizar que cada aplicación reciba su cuota de tiempo asignada. La política de control de admisión sólo admitirá a un cliente que solicite un número concreto de cuotas si hay suficientes cuotas disponibles. En nuestro ejemplo, hemos asignado $50 + 15 + 20 = 75$ cuotas del total de 100. Si un nuevo proceso *D* solicitará 30 cuotas, el controlador de admisión denegaría a *D* la entrada en el sistema.

19.5.4 Planificación en Pthread

El estándar POSIX también proporciona extensiones para informática en tiempo real: POSIX.1b. En esta sección, vamos a analizar parte de la API Pthread de POSIX relacionada con la planificación de hebras en tiempo real. Pthreads define dos clases de planificación para las hebras en tiempo real:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO planifica las hebras de acuerdo con una política en que el primero en llegar es el primero en ser servido, usando una cola FIFO, como se indica en la Sección 5.3.1. Sin embargo, no existe ningún reparto de franjas temporales entre las hebras de igual prioridad. Por tanto, la hebra de tiempo real de prioridad más alta situada al principio de la cola FIFO recibirá la CPU y continuará ejecutándose hasta terminar, o hasta quedar bloqueada. SCHED_RR (donde RR quiere decir *round-robin*, es decir, planificación por turnos) es similar a SCHED_FIFO salvo porque incluye una distribución de franjas temporales entre hebras de igual prioridad. Pthreads proporciona una clase adicional de planificación (SCHED_OTHER) pero su implementación no está definida y es específica del sistema, pudiendo por tanto comportarse de manera diferente en distintos sistemas.

La API Pthread especifica las siguientes dos funciones para consultar y establecer la política de planificación:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

El primer parámetro de ambas funciones es un puntero al conjunto de atributos de la hebra. El segundo parámetro puede ser un puntero a un entero que defina la política de planificación actual [para `pthread_attr_getsched_policy()`] o un valor entero (SCHED_FIFO, SCHED_RR o SCHED_OTHER) para la función `pthread_attr_setsched_policy()`. Ambas funciones devuelven valores distintos de cero en caso de que se produzca un error.

En la Figura 19.11 se muestra un programa Pthread que usa esta API. Este programa determina en primer lugar la política de planificación actual y después selecciona el algoritmo de planificación SCHED_OTHER.

19.6 VxWorks 5.x

En esta sección, vamos a describir VxWorks, un popular sistema operativo de tiempo real que proporciona soporte de tiempo real estricto. VxWorks, desarrollado comercialmente por Wind River Systems, se utiliza ampliamente en vehículos, dispositivos de consumo e industriales y en equipos de red como comutadores y encaminadores. VxWorks también se emplea para controlar los dos robots, *Spirit* y *Opportunity*, que comenzaron a explorar el planeta Marte en 2004.

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* obtener atributos predeterminados */
    pthread_attr_init(&attr);

    /* obtener la política de planificación actual */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Imposible obtener política.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* establecer la política de planificación - FIFO, RR, u OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
        fprintf(stderr, "Imposible establecer política.\n");

    /* crear las hebras */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* terminar cada una de las hebras */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Cada hebra tomará el control en esta función */
void *runner(void *param)
{
    /* realizar alguna tarea ... */

    pthread_exit(0);
}

```

Figura 19.11 API de planificación de Pthread.

La organización de VxWorks se muestra en la Figura 19.12. VxWorks está centrado en torno al *microkernel Wind*. Recuerde de nuestras explicaciones en la Sección 2.7.3, que los *microkernels* están diseñados de modo que el *kernel* del sistema operativo proporcione un número mínimo de características; las utilidades adicionales, como la conexión por red, los sistemas de archivos y los gráficos, se proporcionan en bibliotecas situadas fuera del *kernel*. Esta técnica ofrece muchas ventajas

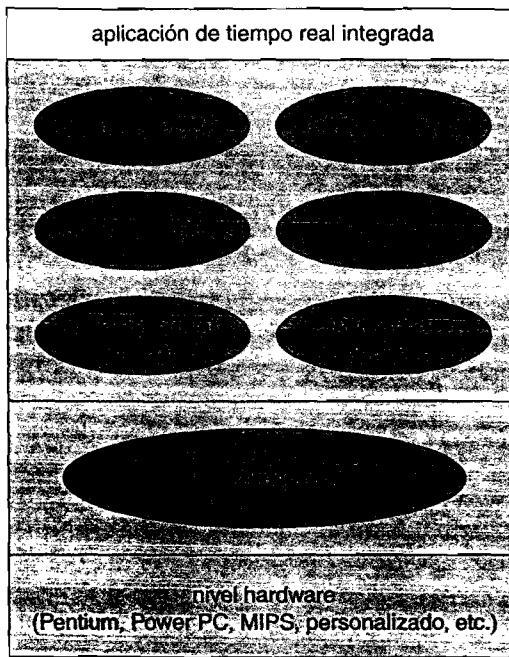


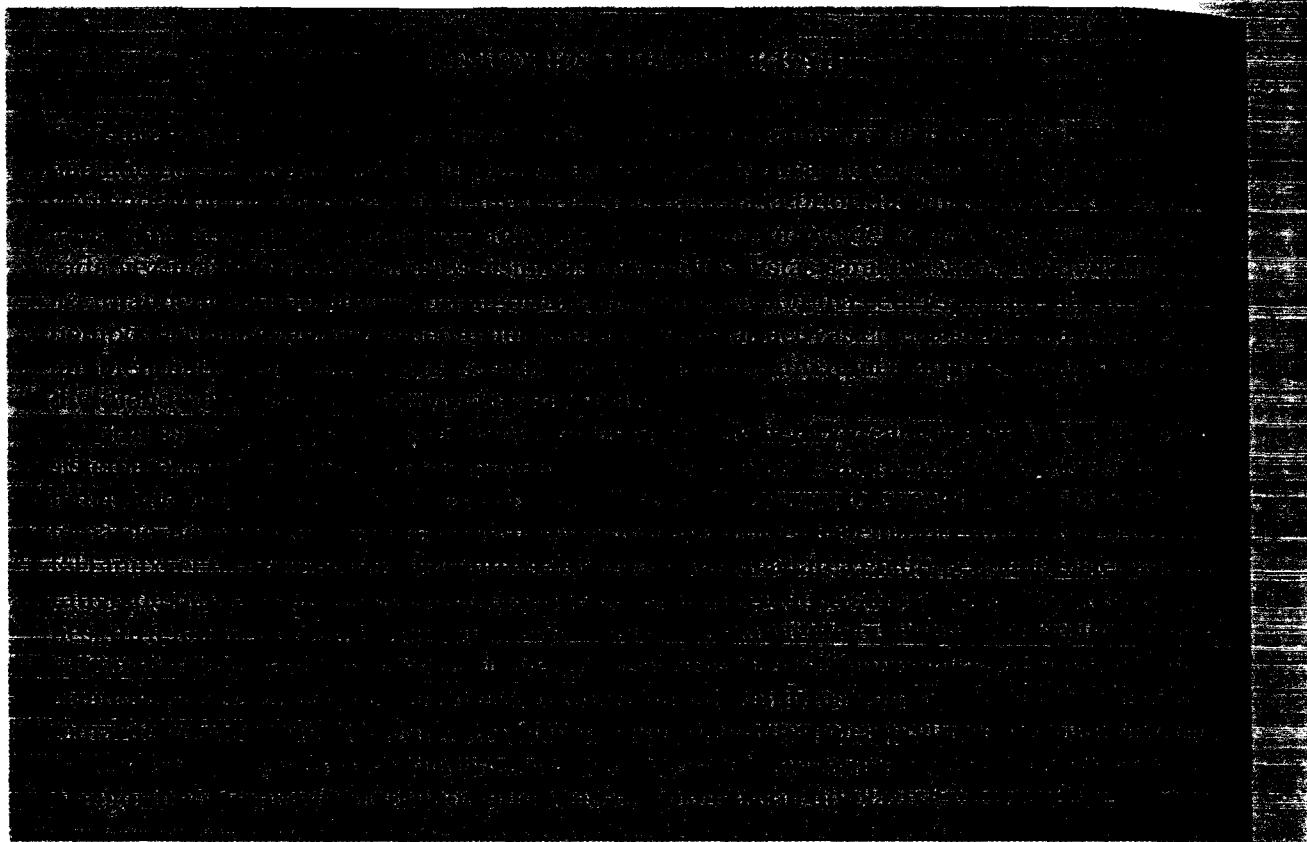
Figura 19.12 La organización de VxWorks.

incluyendo la minimización del tamaño del *kernel*, que constituye una característica deseable para todo sistema integrado que requiera una huella de memoria pequeña.

El *microkernel* Wind soporta las siguientes características básicas:

- **Procesos.** El *microkernel* Wind proporciona soporte para procesos individuales y hebras (utilizando la API Pthread). Sin embargo, de forma similar a Linux, VxWorks no distingue entre procesos y hebras haciendo referencia a ambos con el nombre de tareas.
- **Planificación.** Wind proporciona dos modelos separados de planificación: planificación por turnos apropiativa y no apropiativa con 256 niveles diferentes de prioridad. El planificador también soporta la API POSIX para hebras de tiempo real que se analiza en la Sección 19.5.4.
- **Interrupciones.** El *microkernel* Wind también gestiona interrupciones. Para cumplir con los requisitos de tiempo real estrictos, los tiempos de latencia de despacho y de interrupción están acotados.
- **Comunicación interprocesos.** El *microkernel* Wind proporciona mecanismos tanto de memoria compartida como de paso de mensajes para la comunicación entre las distintas tareas. Wind también permite que las tareas se comuniquen utilizando una técnica denominada *pipes*, que son un mecanismo que se comporta del mismo modo que una cola FIFO, pero permite a las tareas comunicarse en un archivo especial, el *pipe*. Para proteger los datos compartidos por las distintas tareas, VxWorks proporciona semáforos y cerrojos mútex, con un protocolo de herencia de prioridades con el fin de evitar el fenómeno de inversión de prioridad.

Fuera del *microkernel*, VxWorks incluye varias bibliotecas de componentes que proporcionan soporte para POSIX, Java, redes TCP/IP, etc. Todos los componentes son opcionales, permitiendo al diseñador de un sistema integrado personalizar el sistema de acuerdo con sus necesidades específicas. Por ejemplo, si no se requiere la conexión por red, puede excluirse la biblioteca TCP/IP de la imagen del sistema operativo. Esta estrategia permite al diseñador del sistema operativo incluir sólo las características requeridas, minimizando así el tamaño (o huella de memoria) del sistema operativo.



VxWorks adopta un enfoque interesante en lo que respecta a la gestión de memoria, permitiendo dos niveles de memoria virtual. El primer nivel, que es bastante simple, permite controlar la caché por separado para cada página. Esta política hace que las aplicaciones puedan especificar ciertas páginas como no almacenables en caché. Cuando hay una serie de datos que están siendo compartidos por distintas tareas que se ejecutan en una arquitectura multiprocesador, resulta posible almacenar los datos compartidos en cachés separadas que sean locales a cada procesador individual. A menos que una arquitectura soporte una política de coherencia de caché para garantizar que los mismos datos que residan en dos cachés no puedan ser distintos, dichos datos compartidos no deben almacenarse en caché y deben residir, en su lugar, únicamente en memoria principal, para que todas las tareas puedan mantener una vista coherente de los datos.

El segundo nivel de memoria virtual requiere el componente opcional de memoria virtual VxVMI (Figura 19.12), junto con un soporte de una unidad de gestión de memoria (MMU) por parte del procesador. Cargando este componente opcional en los sistemas con una MMU, VxWorks permite a las tareas marcar ciertas áreas de datos como *privadas*. Un área de datos marcada como privada sólo podrá ser accedida por la tarea a la que pertenezca. Además, VxWorks permite declarar como de sólo lectura las páginas que contienen el código del *kernel* junto con el vector de interrupción. Esta característica resulta muy útil, ya que VxWorks no distingue entre los modos de usuario y de *kernel*; todas las aplicaciones se ejecutan en modo *kernel*, lo que proporciona a la aplicación acceso al espacio de direcciones completo del sistema.

19.7 Resumen

Un sistema de tiempo real es un sistema informático que requiere que se obtengan los resultados antes de un cierto plazo; los resultados obtenidos después cumplirse el plazo son completamente inútiles. En los dispositivos de consumo e industriales están integrados muchos sistemas de tiempo real. Existen dos tipos de sistema de tiempo real: estrictos y no estrictos. Los sistemas de tiempo real no estrictos son los menos restrictivos, limitándose a asignar a las tareas de tiempo

real una prioridad de planificación mayor que a las otras tareas. Los sistemas de tiempo real estrictos deben garantizar que se preste servicio a las tareas de tiempo real dentro de los plazos definidos. Además de por los requisitos de temporización estrictos, los sistemas de tiempo pueden caracterizarse también por el hecho de que se utilizan para un único propósito y se ejecutan sobre dispositivos de pequeño tamaño y bajo coste.

Para cumplir con los requisitos de temporización, los sistemas operativos de tiempo real deben emplear diversas técnicas. El planificador para un sistema de tiempo real debe poder trabajar con un algoritmo basado en prioridades con apropiación. Además, el sistema operativo debe permitir que las tareas que se ejecutan en el *kernel* sean desalojadas en favor de las tareas de tiempo real con mayor prioridad. Los sistemas operativos de tiempo real también tratan de resolver los problemas específicos de temporización minimizando la latencia de interrupción como la de despacho.

Entre los algoritmos de planificación de tiempo real podemos citar el algoritmo de planificación por prioridad monótona en tasa y el algoritmo por prioridad en finalización de plazo. La planificación por prioridad monótona en tasa asigna una mayor prioridad a las tareas que requieren la CPU más a menudo, asignando la prioridad más baja a aquellas que hacen un uso más infrecuente de la CPU. La planificación por prioridad en finalización de plazo asigna la prioridad de acuerdo con los plazos previstos: cuanto más próximo esté un plazo, mayor será la prioridad del proceso correspondiente. La planificación con cuota proporcional utiliza la técnica de dividir el tiempo de procesador en una serie cuotas y asignar un número de cuotas a cada proceso, garantizando así a cada proceso su cuota proporcional de tiempo de CPU. La API Pthread proporciona también diversas funciones para la planificación de hebras en tiempo real.

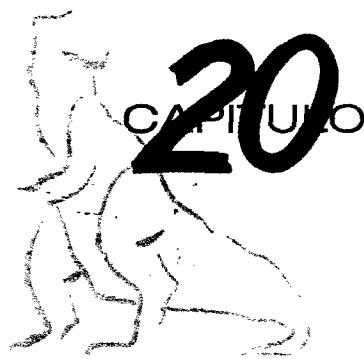
Ejercicios

- 19.1 Indique si resulta más apropiada la planificación de tiempo real estricta o no estricta en los siguientes entornos:
 - a. Termostato en una vivienda.
 - b. Sistema de control para una central de energía nuclear.
 - c. Sistema de ahorro en un vehículo.
 - d. Sistema de aterrizaje en un avión comercial.
- 19.2 Indique de qué manera se podría resolver el problema de inversión de prioridades en un sistema de tiempo real. Indique también si las soluciones podrían implementarse dentro del contexto de un planificador con cuota proporcional.
- 19.3 El *kernel* de Linux 2.6 puede construirse sin sistema de memoria virtual. Explique por qué esta característica puede resultar atractiva para los diseñadores de sistemas en tiempo real.
- 19.4 ¿En qué circunstancias es la planificación por prioridad monótona en tasa peor que la planificación por prioridad de finalización de plazo, a la hora de satisfacer los plazos asociados con cada proceso?
- 19.5 Considere dos procesos, P_1 y P_2 , donde $p_1 = 50$, $t_1 = 25$, $p_2 = 75$ y $t_2 = 30$.
 - a. ¿Pueden planificarse estos dos procesos utilizando un algoritmo de planificación monótona en tasa? Ilustre su respuesta utilizando un diagrama de Gantt.
 - b. Ilustre la planificación de estos dos procesos utilizando el algoritmo EDF de planificación por prioridad en finalización de plazo.
- 19.6 ¿Cuáles son los diversos componentes de las latencias de interrupción y de despacho?
- 19.7 Explique por qué las latencias de interrupción y de despacho deben estar acotadas en un sistema de tiempo real estricto.

Notas bibliográficas

Los algoritmos de planificación para los sistemas de tiempo real estrictos, como la planificación monótona en tasa y la planificación por prioridad en finalización de plazo, fueron presentados en Liu y Layland [1973]. Otros algoritmos de planificación, así como una serie de extensiones a los algoritmos anteriores se presentan en Jensen et al. [1985], Lehoczky et al. [1989], Audsley et al. [1991], Mok [1983] y Stoica et al. [1996]. Mok [1983] describía un algoritmo de asignación dinámica de prioridades denominado planificación con prioridad de la menor laxitud. Stoica et al [1996] analiza el algoritmo de cuota proporcional. Puede encontrar información útil relativa a diversos sistemas operativos populares que se utilizan en sistemas integrados en <http://rtlinux.org>, <http://windriver.com> y <http://qnx.com>. El tema de las líneas futuras de investigación y los problemas más importantes abordados por esas investigaciones en el campo de los sistemas integrados se analiza en un artículo de Stankovic [1996].

Sistemas multimedia



En los capítulos anteriores, nos hemos centrado generalmente en la forma en la que los sistemas operativos gestionan los datos convencionales, como archivos de texto, programas, archivos binarios, documentos de procesadores de textos y hojas de cálculo. Sin embargo, los sistemas operativos pueden tener que manejar también otros tipos de datos. Una tendencia reciente dentro del ámbito tecnológico es la de la incorporación de **datos multimedia** en los sistemas informáticos. Los datos multimedia están compuestos de flujos continuos de datos (audio y vídeo), además de archivos convencionales. Los datos de flujo continuo difieren de los datos convencionales en que los primeros (por ejemplo, las imágenes de vídeo) deben suministrarse de acuerdo con ciertas restricciones de tiempo (por ejemplo, 24 imágenes por segundo). En este capítulo, vamos a analizar los requisitos de los datos de flujo continuo. También veremos con más detalle en qué sentido difieren esos datos de los datos convencionales y cómo afectan esas diferencias en el diseño de los sistemas operativos que se utilizan en los sistemas multimedia.

OBJETIVOS DEL CAPÍTULO

- Identificar las características de los datos multimedia.
- Examinar varios algoritmos que se utilizan para comprimir los datos multimedia.
- Explorar los requisitos que los datos multimedia imponen a los sistemas operativos, incluyendo los temas de planificación de la CPU y del disco, y de gestión de red.

20.1 ¿Qué es la multimedia?

El término *multimedia* describe un amplio rango de aplicaciones que se utilizan hoy en día de modo bastante popular. Entre éstas se incluyen los archivos de audio y vídeo, como por ejemplo los archivos de audio MP3, las películas de DVD y videoclips de corta duración correspondientes a anuncios de películas o noticias de interés informativo que pueden descargarse a través de Internet. Las aplicaciones multimedia también incluyen las difusiones en vivo (*webcasts*, que son difusiones realizadas a través de la World Wide Web) de discursos o de eventos deportivos e incluso las cámaras web en vivo que permiten a un espectador de una determinada ciudad observar a los clientes de un café de París. Las aplicaciones multimedia no tienen por qué ser sólo audio o sólo vídeo; en lugar de ello, una aplicación multimedia suele incluir una combinación de ambos tipos de datos. Por ejemplo, una película puede estar compuesta por pistas separadas de audio y de vídeo. Asimismo, las aplicaciones multimedia no sólo se distribuyen a computadoras personales de sobremesa. Cada vez más, también se distribuyen estos datos a dispositivos de menor tamaño, incluyendo los asistentes digitales personales (PDA, personal digital assistant) y teléfonos

móviles. Por ejemplo, un agente de bolsa puede hacer que se le envíen las cotizaciones en tiempo real a su PDA.

En esta sección, vamos a explorar diversas características de los sistemas multimedia y a examinar cómo pueden enviarse archivos multimedia desde un servidor a un sistema cliente. También veremos determinados estándares comunes para la representación de archivos multimedia de audio y vídeo.

20.1.1 Suministro de datos

Los datos multimedia se almacenan en el sistema de archivos al igual que cualquier otro tipo de datos. La principal diferencia entre un archivo normal y un archivo multimedia es que al archivo multimedia hay que acceder con una tasa específica, mientras que el acceso a un archivo normal no requiere ninguna temporización especial. Utilicemos un vídeo como ejemplo de lo que queremos decir con "tasa". El vídeo está representado por una serie de imágenes, que se muestran en rápida sucesión. Cuanto más rápido se muestran las imágenes, más suaves parecen los movimientos del vídeo. En general, es necesaria una tasa de entre 24 y 30 imágenes por segundo para que los movimientos reflejados en el vídeo parezcan suaves a ojos de un espectador humano (el ojo retiene cada imagen durante un corto tiempo después de presentarla, conociéndose esta característica con el nombre de **persistencia de la visión**). Una tasa de entre 24 y 30 imágenes por segundo es lo suficientemente rápida como para que parezca un movimiento continuo). Una tasa inferior a 24 imágenes por segundo hará que los movimientos parezcan dar saltos. Es necesario que se acceda al archivo de vídeo del sistema de archivos a una tasa que sea coherente con aquella a la que se está mostrando el vídeo. Para referirnos a los datos que tienen este tipo de requisitos de tasa asociados, utilizamos el término de **datos de flujo continuo**.

Los datos multimedia pueden suministrarse a un cliente bien desde el sistema de archivos local bien desde un servidor remoto. Cuando los datos se suministran desde el sistema de archivos local, denominamos a ese suministro **reproducción local**. Como ejemplos tendríamos la visualización de un DVD en una computadora de sobremesa o la audición de un archivo MP3 en un reproductor MP3 de mano. En estos casos, los datos están contenidos en un archivo normal almacenado en el sistema de archivos local y que se reproduce (es decir, se visualiza o se escucha) desde dicho sistema.

Los archivos multimedia pueden estar también almacenados en un servidor remoto y ser entregados a un cliente a través de una red utilizando una técnica conocida con el nombre de **transmisión de flujos (streaming)**. El cliente puede ser una computadora personal o algún otro dispositivo de menor tamaño, como una computadora de mando, un PDA o un teléfono móvil. Los datos correspondientes a contenidos en vivo de tipo continuo (como por ejemplo, cámaras web en vivo) también se envían desde un servidor a uno o más clientes.

Existen dos tipos de técnicas de transmisión de flujos: descarga progresiva y flujos en tiempo real. Con una **descarga progresiva**, se descarga un archivo multimedia que contenga audio o vídeo y se almacena en el sistema de archivos local de los clientes. A medida que se descarga el archivo, el cliente puede reproducir el contenido del mismo sin tener que esperar a que se descargue todo el contenido del archivo. Puesto que el archivo multimedia se almacena en el sistema cliente, este método de la descarga progresiva resulta útil especialmente en el caso de archivos multimedia relativamente pequeños, como por ejemplo videoclips de corta duración.

La transmisión de flujos en tiempo real difiere de la descarga progresiva en que el archivo multimedia también se envía en forma de flujo al cliente, pero sin llegar a almacenarlo en él mismo; el cliente se limita a reproducir lo que le llega. Puesto que el archivo multimedia no está almacenado en el sistema cliente, los flujos en tiempo real son preferibles al método de la descarga progresiva para aquellos archivos multimedia que puedan ser demasiado grandes como para almacenarlos en el sistema, como por ejemplo vídeos de larga duración o difusiones de TV o de radio a través de Internet.

Tanto la descarga progresiva como la transmisión de flujos en tiempo real pueden permitir que el cliente se sitúe en diferentes puntos del flujo de datos, del mismo que utilizariamos las operaciones de avance y retroceso rápido en un magnetoscopio para situarnos en diferentes puntos de

la cinta de vídeo. Por ejemplo, podemos desplazarnos hasta el final de un flujo de vídeo de cinco minutos de duración o volver a reproducir una cierta sección de una película. La posibilidad de desplazarse a uno u otro lado dentro del flujo multimedia se conoce con el nombre de **acceso aleatorio**.

Existen dos tipos de flujos en tiempo real: flujos en vivo y flujos a la carta. Los **flujos en vivo** (*live streaming*) se utilizan para suministrar en directo la información multimedia referida a un evento, como por ejemplo un concierto o una conferencia, a medida que el evento tiene lugar. La difusión de un programa de radio a través de Internet es un ejemplo de flujo en tiempo real en vivo. De hecho, uno de los autores de este texto suele escuchar su emisora de radio favorita de una cierta ciudad mientras trabaja en su casa, situada en otra ciudad distinta, y para ello utiliza el flujo multimedia que se transmite en vivo a través de Internet. Los flujos de tiempo real en vivo se utilizan también para aplicaciones tales como cámaras web en directo y videoconferencias. Debido a que el suministro de los datos se realiza en directo, este tipo de flujos en tiempo real no permite que los clientes accedan aleatoriamente a diferentes puntos del flujo multimedia. Además, el suministro en vivo implica que un cliente que quiera ver (o escuchar) un flujo de datos concreto en vivo que ya haya comenzado se “unirá” a la sesión “tarde”, perdiéndose por tanto las partes anteriores de ese flujo de datos. Pasa exactamente lo mismo que con una difusión en vivo de un programa de TV o de radio por medios convencionales. Si comenzamos a ver un programa de noticias diez minutos después de que éste haya comenzado, nos perderemos los diez primeros minutos de esa difusión.

Los **flujos a la carta** se utilizan para suministrar flujos multimedia tales como películas o grabaciones de conferencias ya celebradas. La diferencia entre los flujos en vivo y los flujos a la carta es que estos últimos no tienen lugar a medida que transcurre el evento. Así, por ejemplo, mientras que contemplar un flujo multimedia en vivo es como contemplar las noticias en la televisión, ver un flujo a la carta es como ver una película en un reproductor de DVD en el momento que nos resulte más conveniente. La noción de “llegar tarde” al programa no existe. Dependiendo del tipo de flujo a la carta, un cliente puede o no disponer de acceso aleatorio a dicho flujo multimedia.

Como ejemplos de productos para difusión de flujos multimedia muy populares podemos citar RealPlayer, Apple QuickTime y Windows Media Player (Reproductor de Windows Media). Estos productos incluyen tanto servidores que distribuyen los flujos multimedia como software cliente que se utiliza para la reproducción.

20.1.2 Características de los sistemas multimedia

Los requisitos de los sistemas multimedia difieren de los de las aplicaciones tradicionales. En general, los sistemas multimedia pueden tener las características siguientes:

1. Los archivos multimedia pueden ser de gran tamaño. Por ejemplo, un archivo de vídeo MPEG-1 de 100 minutos de duración necesita, aproximadamente, 1,125 GB de espacio de almacenamiento; 100 minutos de TV de alta definición (HDTV, high-definition television) requiere aproximadamente 15 GB de almacenamiento. Así, un servidor que almacene cientos o miles de archivos de vídeo digital puede requerir varios terabytes de espacio de almacenamiento.
2. Los datos de flujo continuo puede requerir tasas de datos muy altas. Considere, por ejemplo, un archivo de vídeo digital, en el que una imagen de vídeo en color se muestre con una resolución de 800×600 . Si utilizamos 24 bits para representar el color de cada píxel (lo que nos permite tener 2^{24} colores diferentes, es decir, unos 16 millones de colores), una única imagen requerirá $800 \times 600 \times 24 = 11.520.000$ bits de datos. Si las imágenes se muestran a una tasa de 30 imágenes por segundo, hará falta un ancho de banda superior a 345 Mbps.
3. Las aplicaciones multimedia son sensibles a los retardos temporales durante la reproducción. Una vez que se suministra un archivo de flujo continuo a un cliente, el suministro debe continuar a una cierta tasa durante la reproducción del contenido; en caso contrario, el oyente o espectador percibirá una serie de pausas durante la presentación.

20.1.3 Cuestiones relativas al sistema operativo

Para que un sistema informático pueda suministrar datos de flujo continuo, debe garantizar que se cumplan los requisitos específicos que los datos multimedia tienen respecto a la tasa y a la temporización; estos requisitos se conocen también con el nombre de requisitos de **calidad de servicio** (QoS, quality of service).

Proporcionar estas garantías QoS afecta a diversos componentes del sistema informático e influyen sobre algunos aspectos del sistema operativo como la planificación de la CPU, la planificación de disco y la gestión de red. Como ejemplos específicos podríamos citar:

1. La compresión y la decodificación puede requerir una capacidad de procesamiento significativa por parte de la CPU.
2. Las tareas multimedia deben planificarse con ciertas prioridades para garantizar que se cumplen los requisitos de temporización de los datos multimedia continuos.
3. De forma similar, los sistemas de archivos deben ser eficientes para poder satisfacer los requisitos de tasa de los datos de flujo continuo.
4. Los protocolos de red deben soportar los requisitos relativos al ancho de banda al mismo tiempo que minimizan el retardo y las fluctuaciones de transmisión.

En las siguientes secciones, vamos a tratar éstas y otras cuestiones relacionadas con los requisitos de calidad de servicio. Pero primero, necesitamos presentar una panorámica de las diversas técnicas utilizadas para comprimir los datos multimedia. Como hemos indicado anteriormente, la compresión impone unas demandas muy significativas a la CPU.

20.2 Compresión

Debido a los requisitos de tamaño y de tasa de los sistemas multimedia, los archivos multimedia se suelen comprimir, pasándolos de su formato original a otro que ocupe mucho menos espacio. Una vez que se ha comprimido un archivo, éste ocupa mucho menos espacio de almacenamiento y puede ser suministrado a un cliente más rápidamente. La compresión resulta particularmente importante cuando el contenido se suministra a través de una conexión de red. Al hablar de la compresión de archivos hacemos referencia a la **tasa de compresión**, que es la relación entre el tamaño del archivo original y el tamaño del archivo comprimido. Por ejemplo, un archivo de 800 KB que se comprima a 100 KB, tendrá una tasa de compresión de 8:1.

Una vez que se ha comprimido (**codificado**) un archivo, es necesario descomprimirlo (**decodificarlo**) antes de poder acceder a él. Hay una característica del algoritmo de compresión utilizado que afecta a la descompresión posterior del archivo. Los algoritmos de compresión pueden ser de dos tipos: **con pérdidas** y **sin pérdidas**. En la compresión con pérdidas, parte de los datos originales se pierden al decodificar el archivo, mientras que la compresión sin pérdidas garantiza que siempre pueda restaurarse a su forma original el archivo comprimido. En general, las técnicas de compresión con pérdidas proporcionan tasas de compresión mucho mayores. Obviamente, sin embargo, sólo ciertos tipos de datos pueden tolerar la compresión con pérdidas: imágenes, audio y vídeo. Los algoritmos de compresión con pérdidas funcionan a menudo eliminando ciertos datos, como por ejemplo, las frecuencias muy altas o muy bajas que el oído humano no puede detectar. Algunos algoritmos de compresión con pérdidas utilizados para datos de vídeo operan almacenando únicamente las diferencias entre cada dos imágenes sucesivas. Los algoritmos de compresión sin pérdidas se utilizan para comprimir archivos de texto, como por ejemplo programas informáticos, porque es necesario poder restaurar esos archivos a su estado original.

Hay comercialmente disponibles diversos esquemas de compresión para datos multimedia de flujo continuo. En esta sección, vamos a centrarnos en el formato MPEG definido por la organización Moving Picture Experts Group.

MPEG hace referencia a un conjunto de formatos de archivo y de estándares de compresión para el vídeo digital. Puesto que el vídeo digital contiene a menudo también una parte de audio, cada uno de los estándares está dividido en tres niveles. Los niveles 3 y 2 se aplican a las partes

de audio y de vídeo del archivo multimedia. El nivel 1 se conoce con el **nivel del sistema** y contiene información de temporización que permite al reproductor MPRG multiplexar las partes de audio y de vídeo de forma que estén sincronizadas durante la reproducción. Hay tres estándares MPEG principales: MPEG-1, MPEG-2 y MPEG-4.

MPEG-1 se utiliza para vídeo digital y para su flujo de audio asociado. La resolución de MPEG-1 es de 352×288 a 24 imágenes por segundo, con una tasa de bits de hasta 1,5 Mbps. Esto proporciona una calidad ligeramente inferior a la de un magnetoscopio convencional. Los archivos de audio MP3 (un soporte muy popular para el almacenamiento de música) utilizan el nivel de audio (nivel 1) de MPEG-1. Para vídeo, MPEG-1 puede conseguir tasas de compresión de hasta 200:1, aunque en la práctica esas tasas de compresión son muy inferiores. Puesto que MPEG-1 no requiere altas tasas de datos se utiliza a menudo para descargar videoclips de corta duración a través de Internet.

MPEG-2 proporciona una mayor calidad que MPEG-1 y se utiliza para comprimir películas DVD y televisión digital (incluyendo la televisión de alta definición, HDTV). MPEG-2 identifica diversos **niveles** y **perfíles** de compresión de vídeo. El nivel hace referencia a la resolución del vídeo, mientras que el perfil caracteriza la calidad del mismo. En general, cuanto mayor sea el nivel de resolución y cuanto mayor sea la calidad del vídeo mayor será la tasa de datos requerida. Las tasas de bits típica para archivos con codificación MPEG-2 están entre 1,5 Mbps y 15 Mbps. Puesto que MPEG-2 requiere tasas de datos más altas, a menudo no resulta adecuado para la distribución de vídeo a través de una red, por lo que se utiliza generalmente para reproducción local.

MPEG-4 es el más reciente de los estándares y se utiliza para transmitir audio, vídeo y gráficos, incluyendo capas de animación bidimensionales y tridimensionales. La animación hace posible que los usuarios interactúen con el archivo durante la reproducción. Por ejemplo, un potencial comprador doméstico puede descargar un archivo MPEG-4 y realizar un recorrido virtual a través de una casa que esté considerando comprar, desplazándose de habitación en habitación. Otra característica de MPEG-4 es que proporciona un nivel de calidad escalable, permitiendo el suministro de los datos a través de conexiones de red relativamente lentas, por ejemplo, un módem de 56 Kbps o a través de redes de área local con tasas de varios megabits por segundo. Además, al proporcionar un nivel escalable de calidad, los archivos de audio y de vídeo MPEG-4 pueden suministrarse a dispositivos inalámbricos, incluyendo computadoras de mano, dispositivos PDA y teléfonos móviles.

Los tres estándares MPEG mencionados llevan a cabo una compresión con pérdidas con el fin de conseguir altas tasas de compresión. La idea fundamental que subyace a la compresión MPEG consiste en almacenar las diferencias entre las tramas sucesivas. No vamos a profundizar en los detalles de cómo MPEG realiza la compresión; el lector interesado puede consultar las Notas bibliográficas proporcionadas al final del capítulo.

20.3 Requisitos de los kernels multimedia

Como resultado de las características descritas en la Sección 20.1.2, las aplicaciones multimedia requieren a menudo una serie de niveles de servicio del sistema operativo que difieren de los requisitos de las aplicaciones tradicionales, como los procesadores de texto, compiladores y hojas de cálculo. Las cuestiones más importantes son, quizás, los requisitos de temporización y de tasa, ya que la reproducción de datos de audio y vídeo exige que esos datos se suministren dentro de unos determinados límites temporales y a una tasa fija. Las aplicaciones tradicionales no suelen tener esas restricciones de temporización y de tasa.

Las tareas que requieren datos a intervalos (o **períodos**) constantes se conocen con el nombre de **procesos periódicos**. Por ejemplo, un vídeo MPEG-1 puede requerir una tasa de 30 imágenes por segundo durante la reproducción. Mantener esta tasa exige que se suministre una imagen aproximadamente $1/30$ o 33,4 milésimas de segundo. Para poner esto en el contexto de los límites de temporización, vamos a suponer que la imagen F_j sigue a la imagen F_i en la reproducción del vídeo y que esa imagen F_j fue mostrada en el instante T_0 . El instante límite para visualizar la imagen F_j es 33,4 milisegundos después del instante T_0 . Si el sistema operativo no es capaz de mostrar esa imagen en dicho instante, la imagen será omitida del flujo de datos.

Como hemos mencionado anteriormente, los requisitos de tasa y los instantes límites de suministro se conocen con el nombre de requisitos de calidad de servicio (QoS). Existen tres niveles QoS:

1. **Servicios sin garantías.** El sistema hace lo que puede para satisfacer los requisitos; sin embargo, no se proporciona ninguna garantía.
2. **QoS no estricta.** Este nivel trata los diferentes tipos de tráfico de diferentes formas, proporcionando a determinados flujos una mayor prioridad que a otros. Sin embargo, al igual que con el servicio sin garantías no se garantiza ninguna característica concreta.
3. **QoS estricta.** Se garantizan los requisitos de calidad de servicio.

Los sistemas operativos tradicionales (los sistemas de los que hemos hablado en este texto hasta el momento) sólo proporcionan normalmente un servicio sin garantías, y confían en la técnica de **sobredimensionamiento**; en otras palabras, esos sistemas simplemente presuponen que la cantidad de recursos disponibles tenderá a ser mayor que lo que la carga de trabajo de caso peor demande. Si la demanda excede a la capacidad de recursos disponible, será necesario una intervención manual y habrá que eliminar del sistema uno o más procesos. Sin embargo, los sistemas multimedia de nueva generación no pueden realizar esas suposiciones. Estos sistemas deben proporcionar a las aplicaciones de flujo continuo las garantías requeridas por la QoS estricta. Por tanto, en lo que resta de nuestra exposición, cuando hagamos referencias a QoS, estaremos refiriéndonos a la QoS estricta. A continuación, vamos a estudiar varias técnicas que permiten a los sistemas multimedia proporcionar dichas garantías de nivel de servicio.

Hay diversos parámetros que definen la QoS para las aplicaciones multimedia, entre los que se incluyen:

- **Tasa de transferencia.** La tasa de transferencia es la cantidad total de procesamiento realizado durante un determinado intervalo. Para las aplicaciones multimedia, la tasa de transferencia es la tasa de datos requerida.
- **Retardo.** El retardo hace referencia al tiempo transcurrido entre el instante en el que se realiza una solicitud y el instante en que se produce el resultado deseado. Por ejemplo, en un sistema multimedia el retardo sería el tiempo que transcurre entre el momento en que un cliente solicita un flujo de datos y el momento en que ese flujo se suministra al cliente.
- **Fluctuación.** La fluctuación está relacionada con el retardo, pero mientras que el retardo hace referencia al tiempo que tiene esperar un cliente para recibir un flujo multimedia, la fluctuación hace referencia a los retardos que se producen durante la reproducción del flujo. Ciertas aplicaciones multimedia, como los flujos en tiempo real a la carta, pueden tolerar este tipo de retardo. Sin embargo, la fluctuación se considera generalmente inaceptable para las aplicaciones de flujo continuo, porque puede hacer que se produzcan pausas de larga duración (o imágenes perdidas) durante la reproducción. Los clientes pueden compensar a menudo estas fluctuaciones almacenando en búfer una cierta cantidad de datos (por ejemplo, cinco segundos datos) antes de comenzar la reproducción.
- **Fiabilidad.** La fiabilidad hace referencia al modo en que se gestionan los errores durante la transmisión y procesamiento de datos multimedia de carácter continuo. Pueden producirse errores debido a que se pierdan paquetes en la red o a retardos de procesamiento en la CPU. En estos y otros escenarios, los errores no pueden corregirse, ya que los paquetes suelen llegar demasiado tarde como para resultar útiles.

El nivel de calidad de servicio puede ser **negociado** entre el cliente y el servidor. Por ejemplo, los datos de flujo continuo pueden comprimirse con diferentes niveles de calidad: cuanto mayor sea la calidad, mayor será la tasa de datos requerida. Un cliente puede negociar una tasa de datos específica con un servidor, acordando un cierto nivel de calidad para la reproducción. Además, muchos reproductores multimedia permiten al cliente configurar el reproductor de acuerdo con la velocidad de la conexión del cliente a la red. Esto permite al cliente recibir un servicio de flujos

multimedia con una tasa de datos que sea específica de su conexión concreta. Así, el cliente negocia la calidad de servicio con el proveedor de contenido.

Para proporcionar garantías QoS, los sistemas operativos utilizan técnicas de **control de admisión**, que consisten simplemente en admitir una solicitud de servicio sólo si el servidor tiene los suficientes recursos como para satisfacer la solicitud. Podemos ver este tipo de técnicas de control de admisión bastante a menudo en nuestras vidas cotidianas, por ejemplo, en una sala de cine sólo se admiten tantos clientes como butacas haya en la sala (también hay otras muchas situaciones en la vida cotidiana en las que las políticas de control de admisión serían deseables, pero no se llevan a la práctica). Si no se utiliza ninguna política de control de admisión en un entorno multimedia, las demandas impuestas al sistema pueden llegar a ser tan altas, que el sistema sea incapaz de garantizar el nivel QoS acordado.

En el Capítulo 6, hemos hablado de cómo utilizar los semáforos como método para implementar una política simple de control de admisión. En este escenario, existe un número finito de recursos no compartibles. Cuando se solicita un recurso, sólo se concede si existen suficientes recursos disponibles; en caso contrario, se obliga a esperar al proceso solicitante hasta que haya disponible un recurso. Pueden utilizarse semáforos para implementar una política de control de admisión, inicializando primero un semáforo con el número de recursos disponibles. Cada solicitud de un recurso se realiza mediante una operación `wait()` sobre el semáforo; un recurso se libera invocando una operación `signal()` sobre ese semáforo. Una vez que todos los recursos se están utilizando, las siguientes llamadas a `wait()` se bloquearán hasta que haya la correspondiente llamada a `signal()`.

Una técnica común para implementar los mecanismos de control de admisión consiste en utilizar **reservas de recursos**. Por ejemplo, los recursos de un servidor de archivos pueden incluir la CPU, la memoria, el sistema de archivos, los dispositivos y la red (Figura 20.1). Observe que los recursos pueden ser exclusivos o compartidos y que puede haber una o más instancias de cada tipo de recurso. Para utilizar un recurso, un cliente debe hacer de antemano una solicitud de reserva del recurso. Si no se puede conceder la solicitud, se deniega la reserva. Los esquemas de control de admisión asignan un **gestor de recurso** a cada tipo de recurso. Las solicitudes de recursos tienen requisitos QoS asociados, como por ejemplo, tasas de datos requeridas. Cuando llega una solicitud referida a un cierto recurso, el gestor de recursos determina si ese recurso puede satisfacer los requisitos QoS de la solicitud. En caso negativo, la solicitud puede ser rechazada, o bien puede negociarse un nivel QoS inferior entre el cliente y el servidor. Si se acepta la solicitud, el gestor de recursos reserva los recursos para el cliente solicitante, asegurándole así los requisitos

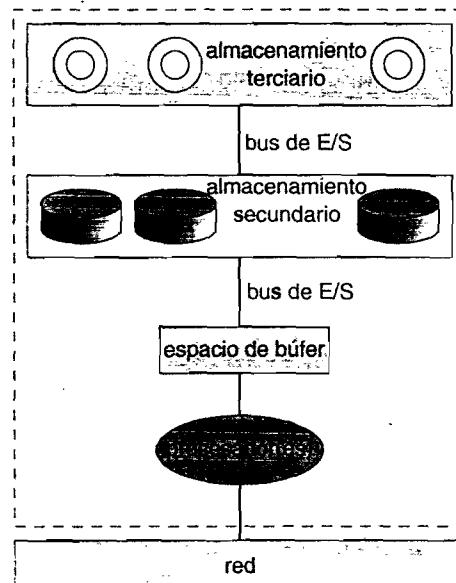


Figura 20.1. Recursos en un servidor de archivos.

QoS deseados. En la Sección 20.7.2, examinaremos el algoritmo de control de admisión utilizado para garantizar el nivel QoS en el servidor multimedia CineBlitz.

20.4 Planificación de la CPU

En el Capítulo 19, en el que nos hemos ocupado de los sistemas de tiempo real, distinguíamos entre **sistemas de tiempo real no estrictos** y **sistemas de tiempo real estrictos**. Los sistemas de tiempo real no estrictos simplemente proporcionan una cierta prioridad de planificación a los procesos críticos. Un sistema de tiempo real no estricto garantiza que los procesos críticos tengan preferencia sobre los procesos no críticos, pero no proporciona ninguna garantía relativa al modo en que se planificará la ejecución de esos procesos críticos. Sin embargo, un requisito típico de los flujos multimedia continuos es que los datos deben ser suministrados a los clientes antes de un cierto instante límite; los datos que no lleguen antes de ese límite resultan inútiles. Por tanto, los sistemas multimedia requieren una planificación de tiempo real estricta para asegurarse de que una tarea crítica reciba servicio dentro de un período de tiempo garantizado.

Otro tema relativo a la planificación es si el algoritmo de planificación utiliza una **prioridad estática** o **prioridad dinámica**, una distinción de la que hemos hablado por primera vez en el Capítulo 5. La diferencia entre ambas es que la prioridad de un proceso permanecerá constante si el planificador le asigna una prioridad estática. Los algoritmos de planificación que asignan prioridades dinámicas permiten que las prioridades cambien a lo largo del tiempo. La mayoría de los sistemas operativos utilizan prioridades dinámicas a la hora de planificar tareas que no sean de tiempo real, con el fin de proporcionar una mayor prioridad a los procesos interactivos. Sin embargo, cuando se planifica la ejecución de tareas de tiempo real, la mayoría de los sistemas asignan prioridades estáticas, ya que el diseño del planificador resulta menos complejo.

Pueden utilizarse varias de las estrategias de tiempo real expuestas en la Sección 19.5 para satisfacer los requisitos QoS de tasa y de temporización de las aplicaciones con flujos multimedia continuos.

20.5 Planificación de disco

Hemos hablado de la planificación de disco por primera vez en el Capítulo 12. Allí, nos centrábamos principalmente en los sistemas que gestionan datos convencionales, para estos sistemas, los objetivos de planificación son la equidad y la tasa de procesamiento. Como resultado, la mayoría de los planificadores de disco tradicionales emplean alguna variante de los algoritmos SCAN (Sección 12.4.3) o C-SCAN (Sección 12.4.4).

Sin embargo, los archivos con datos multimedia continuos tienen dos restricciones que no se encuentran, generalmente, en los archivos de datos convencionales: instantes límites de temporización y requisitos de tasa. Estas dos restricciones deben satisfacerse para asegurar las garantías QoS, y los algoritmos de planificación de disco deben optimizarse de acuerdo con dichas restricciones. Lamentablemente, estas dos restricciones entran a menudo en conflicto. Los archivos multimedia continuos requieren normalmente unos anchos de banda muy altos para satisfacer sus requisitos relativos a la tasa de datos. Puesto que los discos tienen tasas de transferencia relativamente bajas y latencias relativamente altas, los planificadores de disco deben reducir la latencia para garantizar un alto ancho de banda. Sin embargo, reducir la latencia puede resultar en una política de planificación que no asigne las prioridades de acuerdo con los instantes límites de procesamiento. En esta sección, vamos a abordar dos algoritmos de planificación de disco que satisfacen los requisitos QoS para los sistemas multimedia de flujo continuo.

20.5.1 Planificación por prioridad en finalización de plazo

Ya vimos el algoritmo EDF (earliest-deadline-first, por prioridad en finalización de plazo) en la Sección 19.5.2 como ejemplo de algoritmo de la planificación de la CPU que asigna las prioridades de acuerdo con los plazos. EDF también puede utilizarse como algoritmo de planificación de disco;

en este contexto, EDF utiliza una cola para ordenar las solicitudes de acuerdo con el instante en que cada solicitud deba terminar de servirse. EDF es similar al algoritmo SSTF (shortest-seek-time-first, prioridad por el tiempo de búsqueda más corto), del que hablamos en la Sección 12.4.2, salvo porque en lugar de dar servicio a la solicitud más próxima al cilindro actual, se sirven las solicitudes de acuerdo con su plazo; en consecuencia, se da servicio primero a la solicitud cuyo plazo sea más próximo al instante actual.

Un problema con esta técnica es que dar servicio a las solicitudes estrictamente de acuerdo con los plazos puede provocar unos tiempos de búsqueda del cabezal de disco más altos, ya que los cabezales de disco pueden moverse aleatoriamente por el mismo, sin tener en cuenta para nada en cuenta la posición actual. Por ejemplo, suponga que un cabezal está actualmente en el cilindro 75 y que la cola de cilindros (ordenada de acuerdo con los plazos) contiene los valores 98, 183, 105. Con una planificación EDF estricta, el cabezal de disco se movería del cilindro 75 al 98, luego al 183, y después volvería al 105. Observe que el cabezal pasa sobre el cilindro 105 al desplazarse desde el 98 al 183. En consecuencia, hubiera sido posible que el planificador de disco diera servicio a la solicitud relativa al cilindro 105 mientras se desplazaba hasta el 183, sin que ello afectara a la preservación de los requisitos de temporización límite relativos al cilindro 183.

20.5.2 Planificación SCAN-EDF

El problema fundamental con la planificación EDF estricta es que ignora la posición de los cabezales de lectura-escritura sobre el disco; resulta posible, en consecuencia, que los cabezales se muevan de un lado a otro del disco, provocando unos tiempos de búsqueda inaceptables que afecten de manera negativa a la tasa de transferencia del disco. Recuerde que este mismo problema ya nos lo encontramos al analizar la planificación FCFS (Sección 12.4.1). Entonces, vimos que para resolver este problema había que adoptar la planificación SCAN, en la que el brazo del disco se mueve en una dirección por toda la superficie dando servicio a las solicitudes de acuerdo con su proximidad al cilindro actual. Una vez que el brazo del disco alcanza el final del mismo, comienza a moverse en la dirección contraria. Esta estrategia permite optimizar los tiempos de búsqueda.

SCAN-EDF es un algoritmo híbrido que combina la planificación EDF con la planificación SCAN. SCAN-EDF comienza con una ordenación EDF, pero da servicio a las solicitudes que tengan el mismo plazo utilizando una ordenación SCAN. ¿Qué sucede si hay varias solicitudes con diferentes plazos, pero que están relativamente próximas? En este caso, SCAN-EDF puede agrupar las solicitudes en lotes, utilizando una ordenación SCAN para dar servicio a todas las solicitudes comprendidas en un mismo lote. Existen muchas técnicas para agrupar en lotes las solicitudes con plazos similares; el único requisito es que la reordenación de solicitudes dentro de un lote no debe impedir que se dé servicio a una solicitud antes de cumplir su plazo. Si los plazos están distribuidos de manera homogénea, los lotes pueden organizarse en grupos de un cierto tamaño, como por ejemplo de 10 solicitudes por lote.

Otra técnica consiste en agrupar por lotes las solicitudes cuyos plazos difieran menos de un determinado umbral, como por ejemplo 100 milisegundos. Consideremos un ejemplo en el que vamos a agrupar las solicitudes por lotes de la siguiente manera. Supongamos que tenemos las solicitudes especificadas en la tabla de la página siguiente, para cada una de las cuales se indica su plazo (en milisegundos) y el cilindro que se solicita.

Supongamos que estamos en el instante 0, que el cilindro al que actualmente se está dando servicio es el 50 y que el cabezal del disco se está moviendo hacia el cilindro 51. De acuerdo con nuestro esquema de agrupación por lotes, las solicitudes D y F se encontrarán en el primer lote; A, G y H estarán en el lote 2; B, E y J en el lote 3; y C e I en el último lote. Las solicitudes dentro de cada lote se ordenarán de acuerdo con el algoritmo SCAN. Así, en el lote 1, daremos primero servicio a la solicitud F y luego a la solicitud D. Observe que nos estamos moviendo hacia abajo en lo que respecta a los números de cilindro desde 85 a 31. En el lote 2, primero damos servicio a la solicitud A y después los cabezales empiezan a moverse en sentido creciente de número de cilindro dando servicio a la solicitud H y luego a G. El lote 3 se sirve en el orden E, B, J. Las solicitudes I y C se sirven en el lote final.

solicitud	plazo	cilindro
100	201	316
101	302	015
102	403	314
103	504	655
104	605	554
105	706	453
106	807	352
107	908	251
108	009	150

20.6 Gestión de red

Quizá el problema más importante de calidad de servicio en los sistemas multimedia es el que se refiere a la preservación de los requisitos de tasa. Por ejemplo, si un cliente quiere ver un vídeo comprimido con MPEG-1, la calidad de servicio dependerá enormemente de la capacidad del sistema para suministrar las imágenes a la tasa requerida.

Nuestro tratamiento de los algoritmos de la planificación de la CPU y de disco se ha centrado en cómo pueden utilizarse estas técnicas para mejorar los requisitos de calidad de servicio de las aplicaciones multimedia. Sin embargo, si el archivo multimedia se está transmitiendo a través de una red (quizá, Internet), una serie de cuestiones relativas al modo en que la red suministra los datos multimedia puede afectar significativamente a la capacidad de satisfacer los requisitos QoS. En esta sección, vamos a analizar diversas cuestiones de red relacionadas con los requisitos específicos de los flujos de datos continuos.

Antes de continuar, merece la pena destacar que las redes informáticas en general (e Internet en particular) no proporcionan actualmente protocolos de red que permitan asegurar el suministro de los datos con requisitos específicos de temporización (existen algunos protocolos propietarios, especialmente los que ejecutan en los encaminadores de Cisco, que sí permiten asignar prioridad a ciertos tipos de tráficos de red, con el fin de satisfacer los requisitos QoS. El uso de dichos protocolos propietarios no está generalizado en Internet y, por tanto, no afectan a nuestra exposición).

Cuando se encaminan los datos a través de una red, resulta probable que la transmisión se encuentre con problemas de congestión, retardos y otros problemas que afectan al tráfico de red; todos esos problemas escapan al control de la fuente de los datos. Para los datos multimedia con requisitos de temporización, las cuestiones de temporización deben sincronizarse entre las dos máquinas interlocutoras: el servidor que suministra el contenido y el cliente que lo reproduce.

Un protocolo que contempla las cuestiones de temporización es el **protocolo de transporte en tiempo real** (RTP, real-time transport protocol). RTP es un estándar de Internet para el suministro de datos en tiempo real incluyendo audio y vídeo. Puede utilizarse para transportar información multimedia en formatos tales como archivos de audio MP3 y archivos de vídeo comprimidos con MPEG. RTP no proporciona ninguna garantía QoS; en lugar de ello, proporciona características que permiten al receptor eliminar las fluctuaciones introducidas por los retardos y la congestión de la red.

En las siguientes secciones, vamos a considerar otras dos técnicas para la gestión de los requisitos específicos de los datos de flujo continuo.

20.6.1 Unidifusión y multidifusión

En general, hay disponibles tres métodos para suministrar contenido desde un servidor a un cliente a través de una red:

- **Unidifusión (*unicast*)**. El servidor entrega el contenido a un único cliente. Si el contenido se está entregando a más de un cliente, el servidor debe establecer una sesión separada de unidifusión con cada cliente.
- **Difusión (*broadcast*)**. El servidor entrega el contenido a todos los clientes, independientemente de si estos quieren recibirla o no.
- **Multidifusión (*multicast*)**. El servidor entrega el contenido a un grupo de receptores que han indicado que quieren recibir el contenido. Este método constituye, en cierta forma, una solución intermedia entre la unidifusión y la difusión.

Uno de los problemas que afecta al suministro mediante unidifusión es que el servidor debe establecer una sesión de unidifusión separada con cada cliente. Esto constituye un desperdicio de recursos especialmente obvio en el caso de los flujos de tiempo real en vivo, ya que el servidor debe realizar varias copias del mismo contenido, una para cada cliente. Evidentemente la difusión pura no siempre resulta apropiada, ya que puede que no todos los clientes deseen recibir el flujo multimedia (baste decir que la difusión pura suele utilizarse únicamente en redes de área local y no resulta posible en la Internet pública).

La multidifusión parece representar un compromiso razonable, ya que permite al servidor entregar una única copia del contenido a todos los clientes que hayan indicado que quieren recibirla. La dificultad de la multidifusión desde un punto de vista práctico, es que los clientes deben estar físicamente próximos al servidor o a algún encaminador intermedio que retransmita el contenido del servidor original. Si la ruta desde el servidor hasta el cliente debe atravesar una serie de encaminadores intermedios, esos encaminadores también deben permitir la multidifusión. Si estas condiciones no se cumplen, los retardos que se experimentarán durante el encaminamiento pueden hacer que se violen los requisitos de temporización de los flujos multimedia continuos. En el caso peor, si un cliente está conectado a un encaminador intermedio que no permite la multidifusión, el cliente no podrá recibir el flujo de multidifusión.

Actualmente, la mayoría de los flujos multimedia continuos se suministran a través de canales de unidifusión; sin embargo, la multidifusión se utiliza en diversas áreas en las que la organización del servidor y de los clientes se conoce de antemano. Por ejemplo, una gran empresa con varias sedes en todo el país puede ser capaz de garantizar que todos los sitios estén conectados a encaminadores de multidifusión y se encuentren relativamente próximos a esos encaminadores. En consecuencia, esa empresa podrá suministrar una presentación efectuada por el Director General utilizando un mecanismo de multidifusión de vídeo.

20.6.2 Protocolo de flujos de tiempo real

En la Sección 20.1.1, hemos descrito algunas características de los flujos multimedia. Como hemos indicado allí, los usuarios pueden ser capaces de acceder aleatoriamente al flujo multimedia, quizás rebobinando o efectuando una pausa del mismo modo que harían con un magnetoscopio tradicional. ¿Cómo es posible esto?

Para resolver esta cuestión, vamos a analizar la forma en que se suministran los flujos multimedia a los clientes. Una posibilidad consiste en enviar los datos en forma de flujo multimedia desde un servidor web estándar utilizando el protocolo de transporte de hipertexto (HTTP, hyper-text transport protocol), que es el protocolo utilizado para suministrar documentos desde un servidor web. Muy a menudo, los clientes utilizan un **reproductor de medios** como QuickTime, RealPlayer o el Windows Media Player, con el fin de reproducir flujos multimedia transmitidos desde un servidor web estándar. Normalmente, el cliente solicita primero un **metaarchivo**, que contiene la ubicación (posiblemente identificada mediante un localizador uniforme de recursos o URL, uniform resource locator) del archivo que contiene el flujo multimedia. Este metaarchivo se suministra al explorador web del cliente, y el explorador arranca entonces el reproductor multimedia apropiado de acuerdo con el tipo de datos multimedia especificado en el metaarchivo. Por ejemplo, un flujo Real Audio requeriría utilizar RealPlayer, mientras que el Windows Media Player puede utilizarse para reproducir flujos multimedia de tipo Windows. El reproductor de medios contacta entonces con el servidor web y solicita el flujo multimedia. Ese flujo se suminis-

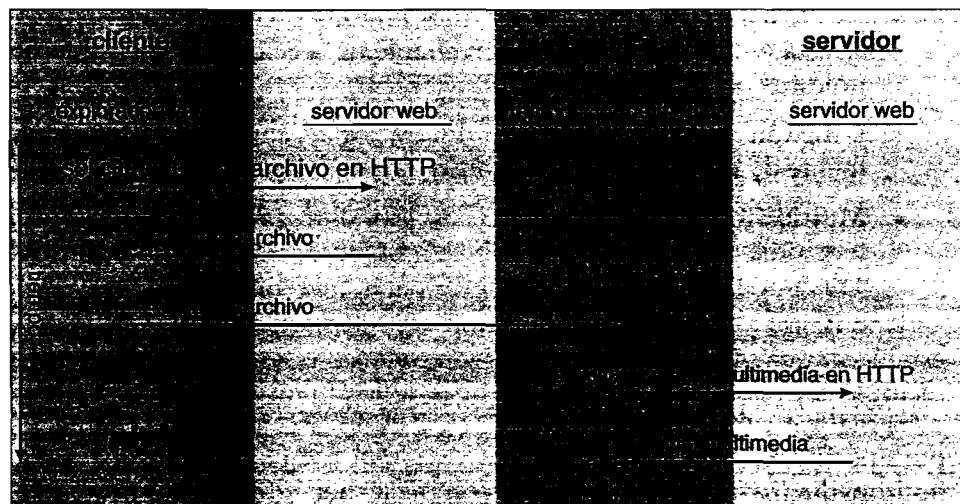


Figura 20.2 Flujos multimedia desde un servidor web convencional.

tra desde el servidor web al reproductor de medios usando solicitudes HTTP estándar. Este proceso se esboza en la Figura 20.2.

El problema de suministrar los flujos multimedia desde un servidor web estándar es que HTTP se considera un protocolo **sin memoria del estado**; por tanto, el servidor web no mantiene el estado de la conexión con el cliente. Como resultado, es difícil que el cliente efectúe una pausa durante el suministro de un contenido multimedia, ya que efectuar esa pausa requeriría que el servidor web supiera dónde comenzar dentro del flujo multimedia cuando el cliente quisiera reanudar la reproducción.

Una estrategia alternativa consiste en utilizar un servidor de flujos multimedia especializado, diseñado específicamente para la transmisión de datos multimedia. Uno de los protocolos diseñados para la comunicación entre los servidores de flujos multimedia y los reproductores de medios es el denominado protocolo de flujos en tiempo real o (RTSP, real-time streaming protocol). La ventaja principal de RTSP sobre HTTP es que se establece una conexión con memoria del estado entre el cliente y el servidor, lo que permite al cliente hacer una pausa o situarse en posiciones aleatorias del flujo multimedia durante la reproducción. El suministro de flujos multimedia mediante RTSP es similar al suministro mediante HTTP (Figura 20.2) en el sentido de que el metaarchivo se suministra utilizando un servidor web convencional. Sin embargo, en lugar de emplear un servidor web, los datos multimedia se suministran desde el servidor de flujos empleando el protocolo RTSP. La operación de RTSP se muestra en la Figura 20.3.

RTSP define varios comandos como parte de su protocolo; estos comandos se envían desde un cliente a un servidor de flujos RTSP. Entre los comandos principales podemos citar:

- **SETUP.** El servidor asigna recursos para una sesión cliente.
- **PLAY.** El servidor suministra un flujo multimedia a una sesión cliente establecida mediante un comando SETUP.
- **PAUSE.** El servidor suspende el suministro de un flujo multimedia pero mantiene los recursos asignados a la sesión.
- **TEARDOWN.** El servidor termina la conexión y libera los recursos asignados a la sesión.

Los comandos pueden ilustrarse mediante una máquina de estados para el servidor, como se muestra en la Figura 20.4. Como puede ver en la figura, el servidor RTSP puede encontrarse en uno de tres posibles estados: **init** (inicial), **ready** (preparado) y **playing** (reproducción). Las transiciones entre estos tres estados tienen lugar cuando el servidor recibe uno de los comandos RTSP del cliente.

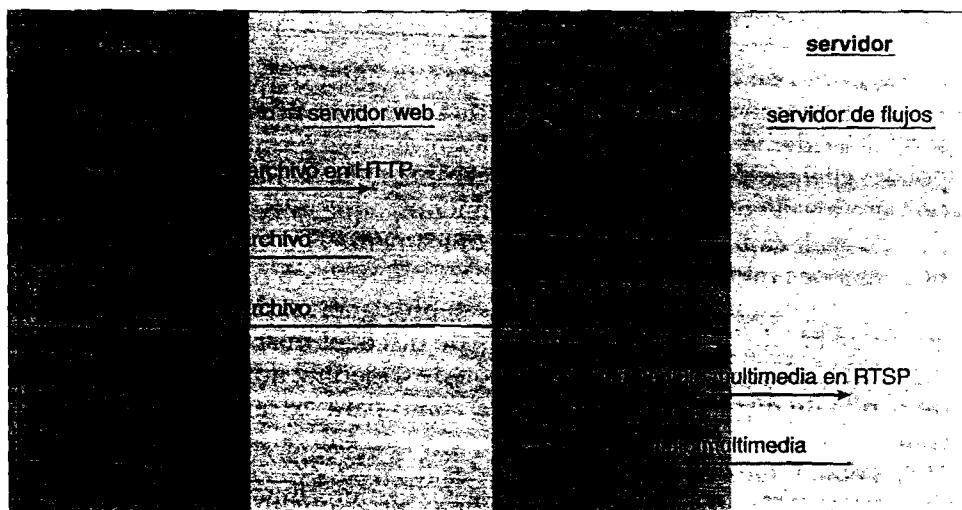


Figura 20.3 Protocolo de flujos en tiempo real (RTSP).

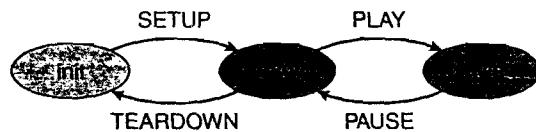


Figura 20.4 Máquina de estados finita que representa el protocolo RTSP

Utilizar RTSP en lugar de HTTP para los flujos multimedia ofrece muchas otras ventajas, aunque éstas están principalmente relacionadas con las cuestiones de red y caen más allá del alcance de este libro. El lector interesado puede consultar las Notas bibliográficas que se proporcionan al final del capítulo, donde encontrará diversas referencias a información adicional.

20.7 Un ejemplo: CineBlitz

El servidor de almacenamiento multimedia CineBlitz es un servidor multimedia de altas prestaciones que soporta tanto datos multimedia continuos con requisitos de tasa de transmisión (como por ejemplo, vídeo y audio) y datos convencionales que no tienen requisito de tasa asociados (como por ejemplo texto e imágenes). CineBlitz denomina a aquellos clientes que tienen requisitos de tasa **clientes de tiempo real**, mientras que **clientes no de tiempo real** no tienen restricciones relativas a la tasa. CineBlitz garantiza poder satisfacer los requisitos de tasa de los clientes de tiempo real implementando un controlador de admisión, de modo que sólo se admita a un cliente si hay suficientes recursos para permitir la extracción de los datos a la tasa requerida. En esta sección, vamos a analizar los algoritmos de planificación de disco y de control de admisión de CineBlitz.

20.7.1 Planificación de disco

El planificador de disco de CineBlitz da servicio a las solicitudes en ciclos. Al principio de cada ciclo de servicio, las solicitudes se colocan en orden C-SCAN (Sección 12.4.4). Recuerde de nuestras explicaciones anteriores relativas a C-SCAN que los cabezales de disco se mueven desde un extremo del disco al otro. Sin embargo, en lugar de invertir la dirección cuando alcanzan el extremo del disco, como en la planificación de disco SCAN pura (Sección 12.4.3), los cabezales del disco vuelven directamente al comienzo del disco.

20.7.2 Control de admisión

El algoritmo de control de admisión de CineBlitz debe monitorizar las solicitudes de los clientes tanto de tiempo real como de no tiempo real garantizando que ambas clases de clientes reciban servicio. Además, el controlador de admisión debe proporcionar las garantías de tasa requeridas por los clientes de tiempo real. Para garantizar la equidad, sólo se reserva una fracción p del tiempo para los clientes de tiempo real, mientras que el resto, $1 - p$, se reserva para los clientes no de tiempo real. Aquí, vamos a analizar únicamente el control de admisión para los clientes de tiempo real; por tanto, el término *cliente* hará referencia a un cliente de tiempo real.

El controlador de admisión de CineBlitz monitoriza diversos recursos del sistema, como el ancho de banda de disco y la latencia de disco, al mismo tiempo que controla el espacio de búfer disponible. El controlador de admisión de CineBlitz admite un cliente sólo si hay suficiente ancho de banda de disco y espacio de búfer como para extraer los datos para el cliente a la tasa requerida.

CineBlitz pone en cola las solicitudes $R_1, R_2, R_3, \dots, R_n$ de archivos multimedia continuos, siendo r_i la tasa de datos requerida para una cierta solicitud R_i . Las solicitudes de la cola se sirven en orden cíclico, utilizando una técnica conocida como **doble búfer**, en la que a cada solicitud R_i se le asigna un búfer de tamaño $2 \times T \times r_i$.

Durante cada ciclo I , el servidor debe:

1. Extraer los datos del disco y almacenarlos en el búfer ($I \bmod 2$).
2. Transferir datos desde el búfer ($(I + 1) \bmod 2$) al cliente.

Este proceso se ilustra en la Figura 20.5. Para N clientes el espacio de búfer total B requerido es:

$$\sum_{i=1}^N 2 \times T \times r_i \leq B \quad (20.1)$$

La idea fundamental que subyace al controlador de admisión de CineBlitz consiste en acotar las solicitudes de entrada en la cola de acuerdo con los siguientes criterios:

1. Primero se estima el tiempo de servicio de cada solicitud.
2. Sólo se admite una solicitud si la suma de los tiempos de servicio estimados para todas las solicitudes admitidas no excede la duración del ciclo de servicio T .

Supongamos que se extraen $T \times r_i$ bits durante cada ciclo para cada cliente de tiempo real R_i con tasa r_i . Si R_1, R_2, \dots, R_n son los clientes actualmente activos en el sistema, entonces el controlador de admisión deberá garantizar que los tiempos totales para extraer $T \times r_1, T \times r_2, \dots, T \times r_n$ bits para los correspondientes clientes de tiempo real no excedan de T . En el resto de esta sección, vamos a analizar los detalles relativos a esta política de admisión.

Si b es el tamaño de un bloque de disco, entonces el número máximo de bloques de disco que pueden extraerse para la solicitud R_k durante cada ciclo es $\lceil (T \times r_k)/b \rceil + 1$. El 1 en esta fórmula

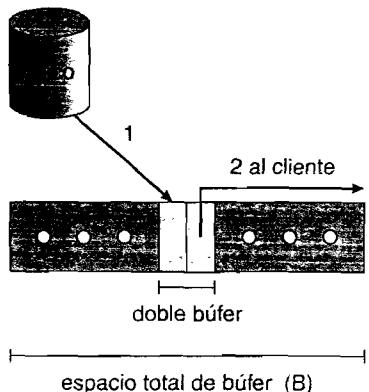


Figura 20.5 Doble búfer en CineBlitz.

se debe al hecho de que si $T \times r_k$ es menor que b , entonces es posible que $T \times r_k$ bits estén distribuidos entre la última parte de un bloque de disco y el comienzo del siguiente, lo que hará que se extraigan dos bloques. Sabemos que la extracción de un bloque implica (a) un cierto tiempo de búsqueda hasta situarse sobre la pista que contiene ese bloque y (b) el retardo rotacional requerido para que los datos contenidos en la pista deseada pasen por debajo del cabezal del disco. Como ya hemos dicho, CineBlitz utiliza un algoritmo de planificación de disco C-SCAN, de modo que los bloques del disco se extraen según el orden de sus posiciones en el disco.

Si t_{seek} y t_{rot} representan los tiempos de caso peor de búsqueda y de retardo rotacional, la latencia máxima requerida para servir N solicitudes será

$$2 \times t_{seek} + \sum_{i=1}^N \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot}. \quad (20.2)$$

En esta ecuación, el componente $2 \times t_{seek}$ hace referencia a la latencia máxima de búsqueda en el disco que se experimenta durante un ciclo. El segundo componente refleja la suma de los tiempos de extracción de los bloques del disco multiplicada por el retardo rotacional de caso peor.

Si la tasa de transferencia del disco es r_{disk} , entonces el tiempo requerido para transferir $T \times r_k$ bits de datos para la solicitud R_k será $(T \times r_k) / r_{disk}$. Como resultado, el tiempo total necesario para extraer $T \times r_1, T \times r_2, \dots, T \times r_n$ bits para las solicitudes R_1, R_2, \dots, R_n será la suma de la ecuación 20.2 y

$$\sum_{i=1}^N \frac{T \times r_i}{r_{disk}} \quad (20.3)$$

Por tanto, el controlador de admisión de CineBlitz sólo admitirá un nuevo cliente R_i si al menos hay disponibles $2 \times T \times r_i$ bits de espacio de búfer libre para el cliente y se satisface la siguiente ecuación:

$$2 \times t_{seek} + \sum_{i=1}^N \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot} + \sum_{i=1}^N \frac{T \times r_i}{r_{disk}} \leq T. \quad (20.4)$$

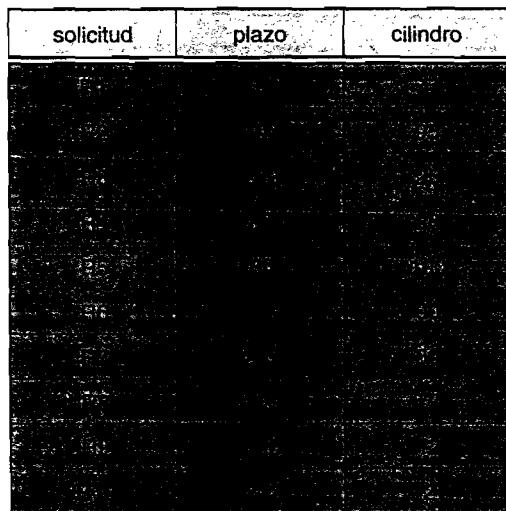
20.8 Resumen

Las aplicaciones multimedia resultan de uso común en los sistemas informáticos modernos. Los archivos multimedia incluyen archivos de vídeo y de audio, que pueden ser suministrados a sistemas tales como computadoras de sobremesa, asistentes digitales personales y teléfonos móviles. La distinción principal entre datos multimedia y datos convencionales es que los datos multimedia tienen requisitos específicos relativos a la tasa y a los plazos de suministro. Puesto que los archivos multimedia tienen requisitos específicos de temporización, a menudo deben comprimirse antes de ser suministrados a un cliente para que éste los reproduzca. Los datos multimedia pueden ser suministrados bien desde el sistema de archivos local o desde un servidor multimedia a través de una conexión de red, utilizando una técnica denominada **flujos multimedia**.

Los requisitos de temporización de los datos multimedia se conocen como requisitos de calidad de servicio, y los sistemas operativos convencionales a menudo no pueden realizar garantías de calidad de servicio. Para proporcionar calidad de servicio, los sistemas multimedia deben proporcionar algún tipo de control de admisión, incluyendo un mecanismo mediante el que el sistema acepte una solicitud sólo si es capaz de satisfacer el nivel de calidad de servicio especificado por la solicitud. Proporcionar garantías de calidad de servicio requiere evaluar el modo en que el sistema operativo realiza la planificación de la CPU, planificación de disco y la gestión de red. Tanto la planificación de la CPU como del disco suelen utilizar como criterio de planificación los requisitos de las tareas de procesamiento multimedia relativos a los plazos de procesamiento. La gestión de red requiere utilizar protocolos que resuelvan los problemas de retardo y de fluctuación causados por la red, así como proporcionar a los clientes mecanismos mediante los que puedan poner en pausa el flujo multimedia durante la reproducción o desplazarse a diferentes posiciones dentro del flujo multimedia.

Ejercicios

- 20.1 Proporcione ejemplos de aplicaciones multimedia que se ejecuten a través de Internet.
- 20.2 Indique las diferencias entre el mecanismo de descarga progresiva y los flujos en tiempo real.
- 20.3 ¿Cuáles de los siguientes tipos de aplicaciones multimedia en tiempo real pueden tolerar un retardo? ¿Cuáles pueden tolerar la fluctuación?
- Flujos de tiempo real en vivo.
 - Flujos de tiempo real a la carta.
- 20.4 Explique qué técnicas podrían utilizarse para satisfacer los requisitos de calidad de servicio para aplicaciones multimedia en los siguientes componentes de un sistema:
- Planificador de procesos.
 - Planificador de disco
 - Gestor de memoria.
- 20.5 Explique por qué los protocolos Internet tradicionalmente usados para la transmisión de datos no son suficientes para proporcionar las garantías de calidad de servicio requeridas por un sistema multimedia. Explique qué cambios se necesitan para proporcionar las garantías de calidad de servicio.
- 20.6 Suponga que se está mostrando un archivo de vídeo digital a una tasa de 30 imágenes por segundo, la resolución de cada imagen es de 640×480 y se utilizan 24 bits para representar cada color. Suponiendo que no se utilice ningún tipo de compresión, ¿cuál es el ancho de banda necesario para suministrar este archivo? A continuación, suponiendo que el archivo se haya comprimido con una tasa de compresión de 200:1 ¿Cuál será el ancho de banda necesario para suministrar el archivo comprimido?
- 20.7 Una aplicación multimedia está compuesta por un conjunto que contiene 100 imágenes, 10 minutos de vídeo y 10 minutos de audio. Los tamaños comprimidos de las imágenes, del vídeo y del audio son 500 MB, 550 MB y 8 MB, respectivamente. Las imágenes se han comprimido con una tasa de 15:1, y el vídeo y el audio se han comprimido con sendas tasas de 200:1 y 10:1, respectivamente. ¿Cuáles eran los tamaños de las imágenes, del vídeo y del audio antes de la compresión?
- 20.8 Suponga que queremos comprimir un archivo de vídeo digital utilizando tecnología MPSG-1. La tasa de bits objetivo es de 1,5 Mbps. Si el vídeo se muestra con una resolución de 352×240 a 30 imágenes por segundo utilizando 24 bits para representar cada color, ¿Cuál será la tasa de compresión necesaria para conseguir la tasa de bits deseada?
- 20.9 Considere dos procesos, P_1 y P_2 , donde $p_1 = 50$, $t_1 = 25$, $p_2 = 75$ y $t_2 = 30$.
- ¿Pueden planificarse estos dos procesos utilizando una planificación por prioridad monótona en tasa? Ilustre su respuesta mediante un diagrama de Gantt.
 - Ilustre la planificación de estos dos procesos usando el algoritmo de planificación por prioridad en finalización de plazo (EDF).
- 20.10 La tabla de la página siguiente contiene una serie de solicitudes con sus plazos y cilindros asociados. Suponga que agrupamos por lotes las solicitudes de 100 en 100 milisegundos. El cabezal de disco se encuentra actualmente en el cilindro 94 y se está desplazando hacia el cilindro 95. Si se usa la planificación de disco SCAN-EDF, ¿cómo se agruparán por lotes las solicitudes y cuál será el orden de las solicitudes dentro de cada lote?
- 20.11 Repita la cuestión anterior, pero esta vez agrupando las solicitudes de 75 en 75 milisegundos.



- 20.12** Compare la unidifusión, la multidifusión y la difusión pura en cuanto técnicas para el suministro de contenido a través de una red informática.
- 20.13** Describa por qué HTTP suele ser insuficiente para la distribución de flujos multimedia.
- 20.14** ¿Qué principio operativo utiliza el sistema CineBlitz para implementar el control de admisión de las solicitudes relativas a archivos multimedia?

Notas bibliográficas

Fuhr [1994] proporciona una introducción general a los sistemas multimedia. Puede encontrar una explicación sobre diversos temas relativos al suministro de multimedia a través de redes de comunicaciones en Kurose y Ross [2005]. El soporte multimedia de los sistemas operativos se analiza en Steinmetz [1995] y Leslie et al. [1996]. La tema de la gestión de recursos tales como la capacidad de procesamiento y los búferes de memoria se trata en Mercer et al. [1994] y Druschel y Peterson [1993]. Narasimha Reddy y Wyllie [1994] proporciona una buena introducción a los problemas relacionados con la utilización de la E/S en un sistema multimedia. Puede encontrar un análisis relativo al modelo de programación apropiado para el desarrollo de aplicaciones multimedia en Regehr et al. [2000]. Un sistema de control de admisión para un planificador por prioridad monótona en tasa se presenta en Lauzac et al. [2003]. Bolosky et al. [1997] presenta un sistema para servir datos de vídeo y analiza los problemas de gestión de la planificación que surgen en dicho sistema. Puede encontrar los detalles relativos al protocolo de flujos en tiempo real en <http://www.rtsp.org>. Tudor [1995] proporciona un tutorial sobre MPEG-2; puede encontrar un tutorial similar sobre técnicas de compresión de vídeo en <http://www.wave-report.com/tutorials/VC.htm>.



Parte Ocho

Casos de estudio

Ahora podemos integrar los conceptos presentados en este libro, describiendo una serie de sistemas operativos reales. Hemos elegido dos de tales sistemas para ese análisis detallado: Linux y Windows XP. Hemos Elegido Linux por varias razones: es muy popular, está disponible de manera gratuita y constituye un sistema UNIX completo. Esto proporciona a los estudiantes de sistemas operativos de leer (y modificar) código fuente real de un sistema operativo.

También analizamos Windows XP con gran detalle. Este reciente sistema operativo de Microsoft está ganando popularidad no sólo en el mercado de las máquinas autónomas, sino también en el de los servidores para grupos de trabajo. Hemos elegido Windows XP principalmente porque nos proporciona la oportunidad de analizar un sistema operativo moderno que tiene un diseño e implementación drásticamente distintos de los UNIX.

Además, vamos a presentar brevemente otros sistemas operativos influyentes. Hemos seleccionado el orden de presentación con el fin de resaltar las similitudes y diferencias entre los sistemas; dicho orden no es estrictamente cronológico y no refleja la importancia relativa de unos sistemas u otros.

Finalmente, se analizan otros tres sistemas existentes. FreeBSD es otro sistema UNIX, pero mientras que Linux combina características de varios sistemas UNIX distintos, FreeBSD está basado en el modelo BSD de UNIX. El código fuente de FreeBSD está disponible gratuitamente al igual que el de Linux. Mach es un moderno sistema operativo que proporciona compatibilidad con BSD UNIX. Windows por su parte es otro sistema operativo moderno de Microsoft para Intel Pentium y versiones posteriores de ese microprocesador; es compatible con las aplicaciones MS-DOS y Microsoft Windows.



El sistema Linux

Este capítulo presenta un análisis detallado del sistema operativo Linux. Examinando un sistema real completo, podemos ver cómo se relacionan entre sí los conceptos que hemos presentado hasta el momento y cómo pueden llevarse a la práctica.

Linux es una versión de UNIX que ha ganado una gran popularidad en los últimos años. En este capítulo, vamos a examinar la historia y el desarrollo de Linux y a presentar las interfaces de usuario y de programador que Linux incluye, interfaces que se basan en gran medida en la tradición UNIX. También analizaremos los métodos internos mediante los que Linux implementa dichas interfaces. Linux es un sistema operativo en rápida evolución y este capítulo describe los desarrollos existentes hasta el *kernel* Linux 2.6 que fue presentado a finales de 2003.

OBJETIVOS DEL CAPÍTULO

- Presentar la historia del sistema operativo UNIX del que se deriva Linux y los principios en los que se basó el diseño de Linux.
- Analizar el modelo de procesos de Linux e ilustrar el modo en que Linux planifica los procesos y proporciona los mecanismos de comunicación interprocesos.
- Examinar el tema de la gestión de memoria en Linux.
- Analizar el modo en que Linux implementa los sistemas de archivos y gestiona los dispositivos de E/S.

21.1 Historia de Linux

Linux tiene un aspecto y estilo similares a los de cualquier otro sistema UNIX; de hecho, la compatibilidad UNIX ha sido uno de los principales objetivos de diseño del proyecto Linux. Sin embargo, Linux es mucho más reciente que la mayoría de los sistemas UNIX, ya que su desarrollo se inició en 1991, cuando un estudiante finlandés, Linus Torvalds, escribió y bautizó con el nombre de **Linux**, un *kernel* pequeño pero autocontenido para el procesador 80386, que fue el primer verdadero procesador de 32-bits dentro de la gama de Intel de procesadores compatibles con las máquinas PC.

Muy poco después de que comenzara su desarrollo, el código fuente de Linux fue puesto a disposición de todo el mundo de manera gratuita a través de Internet. Como resultado, la historia de Linux representa un caso de colaboración por parte de múltiples usuarios de todo el mundo, que se comunicaban casi exclusivamente a través de Internet. A partir de un *kernel* inicial que implementaba parcialmente un pequeño subconjunto de los servicios del sistema UNIX, el sistema Linux creció para incluir buena parte de la funcionalidad UNIX.

En los primeros días, el desarrollo de Linux giraba principalmente alrededor del *kernel* central del sistema operativo: el programa ejecutivo privilegiado que gestiona todos los recursos del sis-

tema e interactúa directamente con el hardware de la máquina. Sin embargo, hace falta mucho más que este *kernel* para construir un sistema operativo completo, por supuesto. Resulta útil realizar la distinción entre el *kernel* de Linux y un sistema Linux. El *kernel* de Linux es un software completamente original desarrollado por la comunidad Linux partiendo de cero. El **sistema Linux**, como lo conocemos hoy en día, incluye una multitud de componentes, algunos de ellos escritos partiendo de cero y otros que se han tomado prestados de otros proyectos de desarrollo; incluso otros componentes han sido creado en colaboración con otros equipos de desarrollo.

El sistema Linux básico es un entorno estándar para aplicaciones y programación de usuario, pero no impone ningún método estándar de gestión del conjunto completo de funcionalidades disponibles. A medida que Linux ha ido madurando, surgió la necesidad de disponer de otra capa de funcionalidad por encima del sistema Linux. Esta necesidad ha sido satisfecha por las diversas distribuciones Linux existentes. Una **distribución Linux** incluye todos los componentes estándar del sistema Linux más una serie de herramientas administrativas para simplificar la instalación inicial y la subsiguiente actualización de Linux, y para gestionar la instalación y eliminación de otros paquetes del sistema. Una distribución moderna también incluye, normalmente, herramientas para la gestión de los sistemas de archivos, para la creación y gestión de las cuentas de usuario, para la administración de redes, exploradores web, procesadores de textos, etc.

21.1.1 El *kernel* de Linux

El primer *kernel* de Linux presentado al público fue la versión 0.01 el 14 de mayo de 1991. No disponía de capacidad de interconexión por red, sólo se ejecutaba sobre procesadores Intel compatibles con el 80386 y hardware PC, y tenía un soporte para controladores de dispositivos extremadamente limitado. El subsistema de memoria virtual era también bastante básico y no incluía ningún soporte para archivos mapeados en memoria; sin embargo, incluso esta versión inicial soportaba la utilización de páginas compartidas con un mecanismo de copia durante la escritura. El único sistema de archivos permitido era el sistema de archivos Minix, los primeros *kernels* de Linux se desarrollaron utilizando una plataforma Minix. Sin embargo, el *kernel* sí que implementaba adecuadamente los procesos UNIX con espacios de direcciones protegidos.

La siguiente versión importante, Linux 1.0, fue lanzada el 14 de marzo de 1994. Esta versión culminó tres años de rápido desarrollo del *kernel* de Linux. Quizá la novedad más importante era la conexión por red: la versión 1.0 incluía soporte para los protocolos de red estándar TCP/IP de UNIX, así como una interfaz *socket* compatible con BSD para la programación de red. Se añadió soporte de controladores de dispositivo para ejecutar IP sobre una red Ethernet o (utilizando los protocolos PPP o SLIP) sobre líneas de comunicación serie o vía módem.

El *kernel* 1.0 también incluía un nuevo sistema de archivos muy mejorado, sin las limitaciones del sistema de archivos Minix original, y soportaba diversos controladores SCSI para el acceso a discos a alta velocidad. Los desarrolladores ampliaron el subsistema de memoria virtual para permitir la paginación sobre los archivos de intercambio y el mapeo de memoria de archivos arbitrarios (pero en la versión 1.0 sólo se implementaba mapeo en memoria de sólo lectura).

También se incluyó en esta versión una gama completa de soporte hardware adicional. Aunque todavía restringido a la plataforma PC de Intel, el soporte hardware había crecido para incluir unidades de disquete y dispositivos CD-ROM, así como tarjetas de sonido, diversos ratones y teclados internacionales. El *kernel* proporcionaba emulación de coma flotante para aquellos usuarios del procesadores 80386 que no dispusieran de coprocesador matemático 80387; el sistema implementaba asimismo, mecanismos de **comunicación interprocesos** (IPC) estilo UNIX System V, incluyendo memoria compartida, semáforos y colas de mensajes. También se proporcionaba un soporte simple para cargar y descargar dinámicamente módulos del *kernel*.

En este punto, dio comienzo el desarrollo de la versión 1.1 del *kernel*, pero posteriormente se publicaron numerosos parches de la versión 1.0 para corregir diversos errores. Fue en este punto cuando se adoptó el patrón estándar de la numeración de las versiones del *kernel* de Linux. Los *kernels* con un número de versión secundaria impar, como 1.1, 1.3 y 2.1 son *kernels de desarrollo*; los números de versión secundaria pares hacen referencia a *kernels de producción* estables. Las actualizaciones de los *kernels* estables sólo buscan solventar errores detectados mientras que los

kernels de desarrollo pueden incluir funcionalidad más reciente que todavía no haya sido suficientemente probada.

En marzo de 1995, se publicó la versión 1.2 del *kernel*. Esta versión no ofrecía tantas mejoras de funcionalidad como la versión 1.0, pero soportaba una variedad mucho más amplia de dispositivos hardware, incluyendo la nueva arquitectura de bus hardware PCI. Los desarrolladores añadieron también otra característica típica de las máquinas tipo PC (el soporte para el modo 8086 virtual de la CPU 80386), con el fin de permitir la emulación del sistema operativo DOS en las computadoras de tipo PC. También actualizaron la pila de protocolos para proporcionar soporte para el protocolo IPX y completaron la implementación IP incluyendo herramientas de gestión de cuentas y funcionalidad de tipo cortafuegos.

La versión 1.2 del *kernel* fue el último de los *kernels* Linux sólo para PC. La distribución fuente para Linux 1.2 incluía soporte parcialmente implementado para procesadores SPARC, Alpha y MIPS, pero la integración completa de estas otras arquitectura no comenzó hasta después de que se publicara el *kernel* 1.2 estable.

La versión Linux 1.2 se concentró en proporcionar un más amplio soporte hardware y una implementación más completa de la funcionalidad existente. En aquel momento se estaban desarrollando muchas nuevas funcionalidades, pero la integración del nuevo código dentro del código fuente principal del *kernel* había sido diferida hasta después de que se publicara el *kernel* estable 1.2. Como resultado, el equipo de desarrollo de la versión 1.3 añadió una gran cantidad de funcionalidades nuevas al *kernel*.

Este trabajo culminó con el lanzamiento de Linux 2.0 en junio de 1996. Con este lanzamiento se incrementó el número de versión principal para reflejar dos nuevas capacidades: el soporte para múltiples arquitecturas, incluyendo una versión Alpha nativa completamente de 64 bits, y el soporte para arquitecturas multiprocesador. Hay también distribuciones Linux basadas en la versión 2.0 disponibles para los procesadores de la serie Motorola 68000 y para los sistemas SPARC de Sun. Hay también una versión derivada de Linux que se ejecuta sobre el *microkernel* Mach y que funciona en sistemas PC y PowerMac.

Los cambios en la versión 2.0 no se detuvieron ahí. El código de gestión de memoria fue mejorado sustancialmente con el fin de proporcionar una caché unificada para los datos del sistema de archivos, independientemente del mecanismo de caché de los dispositivos de bloque. Como resultado de este cambio, el *kernel* ofrecía unas prestaciones de memoria virtual y del sistema de archivos mucho mayores. Por primera vez, el mecanismo de caché del sistema de archivos se extendió a los sistemas de archivos en red, y se añadió también soporte para regiones mapeadas en memoria en las que se pudiera escribir.

El *kernel* 2.0 también mejoraba significativamente las prestaciones de las comunicaciones TCP/IP, y se añadieron asimismo diversos nuevos protocolos de red, incluyendo AppleTalk, AX.25 para la comunicación a través de redes de radioaficionados y soporte RDSI. Se añadió la capacidad de montar software de red remoto y volúmenes SMB (Microsoft LanManager).

Otras mejoras importantes de la versión 2.0 fueron el soporte para las hebras internas del *kernel*, para gestionar dependencias entre módulos cargables y para la carga automática de módulos a la carta. Se mejoró en buena medida la configuración dinámica del *kernel* en tiempo de ejecución, utilizando una nueva interfaz de configuración estandarizada. Entre las características añadidas se incluían las cuotas del sistema de archivos y las clases para planificación de procesos en tiempo real compatibles con POSIX.

Las mejoras continuaron con el lanzamiento de Linux 2.2 en enero de 1999. Se añadió una versión para sistemas UltraSPARC y se mejoraron las capacidades de comunicación por red, añadiendo un sistema de cortafuegos más flexible, unos mejores mecanismos de encaminamiento y de gestión de tráfico y soporte para ventanas de transmisión TCP de gran tamaño y mecanismos de confirmación selectiva. Con esta versión podían leerse discos Acorn, Apple y NT, y se mejoró el sistema de archivo NFS, añadiendo un demonio NFS en modo *kernel*. Los mecanismos de bloqueo para tratamiento de señales, interrupciones y ciertas interrupciones de E/S se rediseñaron utilizando una granularidad más fina que antes, con el fin de mejorar la velocidad en el multiprocesamiento simétrico (SMP).

Las mejoras en las versiones 2.4 y 2.6 del *kernel* incluyen un mayor soporte para los sistemas SMP, sistemas de archivos con registro diario y mejoras en el sistema de gestión de memoria. El

planificador de procesos ha sido modificado en la versión 2.6, proporcionando un eficiente algoritmo de planificación con complejidad $O(1)$. El *kernel* Linux 2.6 es ahora apropiativo, permitiendo desalojar un proceso mientras se está ejecutando en modo *kernel*.

21.1.2 El sistema Linux

En muchos aspectos, el *kernel* de Linux forma la base del proyecto Linux, pero hay también otros componentes que intervienen en la construcción de un sistema operativo Linux completo. Mientras que el *kernel* de Linux está compuesto completamente de código escrito partiendo de cero específicamente para el proyecto Linux, buena parte del software de soporte que forma un sistema Linux no es exclusivo de Linux, sino común a diversos sistemas operativos de tipo UNIX. En particular, Linux utiliza muchas herramientas desarrolladas como parte del sistema operativo BSD de Berkeley, del sistema X Window del MIT y del proyecto GNU de la organización Free Software Foundation.

Esta compartición de herramientas ha tenido lugar en ambas direcciones. Las principales bibliotecas del sistema de Linux tiene su origen en el proyecto GNU, pero la comunidad Linux ha mejorado enormemente esas bibliotecas resolviendo las omisiones, ineficiencias y errores detectados. Otros componentes, como el **compilador C de GNU** (gcc), ya tenían una calidad lo suficientemente alta como para poder usarlos directamente en Linux. Las herramientas de administración de red utilizadas en Linux derivan de código que fue desarrollado para el sistema BSD 4.3, pero a cambio otros derivados más recientes de BSD, como FreeBSD, han tomado prestado código de Linux. Como ejemplos, podríamos citar la biblioteca matemática de emulación de coma flotante para Intel y los controladores de dispositivos hardware de sonido para PC.

El sistema Linux, en su conjunto, es mantenido por una red poco estructurada de desarrolladores que colaboran a través de Internet, existiendo pequeños grupos de personas individuales que son responsables de mantener la integridad de determinados componentes específicos. Hay un pequeño número de sitios públicos de archivos ftp (file-transfer-protocol, protocolo de transferencia de archivos) en Internet que actúan como repositorio comúnmente aceptado para dichos componentes. La comunidad Linux mantiene también el documento **File System Hierarchy Standard** (estándar de jerarquía del sistema de archivos) como medio de mantener la compatibilidad entre los diversos componentes del sistema. Este estándar especifica la disposición global de un sistema de archivos estándar de Linux; determina los nombres de los directorios en los que deben almacenarse los archivos de configuración, las bibliotecas, los ejecutables del sistema y los archivos de datos de tiempo de ejecución.

21.1.3 Distribuciones de Linux

En teoría, cualquiera puede instalar un sistema Linux descargando de los sitios ftp las últimas revisiones de los componentes del sistema necesarios y compilándolas. En los primeros días de Linux, esto era precisamente lo que un usuario de Linux tenía que hacer. Sin embargo, a medida que Linux ha ido madurando, diversas personas y grupos han tratado de hacer más sencillo este trabajo proporcionando un conjunto de paquetes estándar precompilado para realizar una instalación de manera sencilla.

Estas colecciones, o distribuciones, incluyen muchas más cosas que el sistema Linux básico. Normalmente, incluyen también utilidades adicionales de gestión e instalación del sistema, así como paquetes precompilados y listos para instalar muchas de las herramientas comunes UNIX, como servidores de noticias, exploradores web, herramientas de edición y procesamiento de textos e incluso juegos.

Las primeras distribuciones gestionaban estos paquetes simplemente proporcionando un mecanismo para desempaquetar todos los archivos en los lugares apropiados. Sin embargo, una de las contribuciones más importantes de las distribuciones más modernas es un mecanismo de gestión avanzada de paquetes. Las distribuciones actuales de Linux incluyen una base de datos de control de los paquetes que permite instalar, actualizar o eliminar paquetes de manera sencilla.

La distribución SLS, que se remonta a los primeros días de Linux, fue la primera colección de paquetes Linux reconocible como una distribución completa. Sin embargo, aunque podía instalarse como una sola entidad, SLS carecía de las herramientas de gestión de paquetes que ahora se incluyen en todas las distribuciones Linux. La distribución **Slackware** representó una gran mejora en lo que respecta a la calidad global, aun cuando su sistema de gestión de paquetes seguía siendo un tanto pobre; continúa siendo una de las distribuciones más ampliamente instaladas dentro de la comunidad Linux.

Desde el lanzamiento de Slackware, han surgido muchas distribuciones comerciales y no comerciales de Linux. **Red Hat** y **Debian** son distribuciones bastante populares; la primera ha sido creada por una empresa comercial de soporte Linux, mientras que la segunda es de la comunidad de software libre Linux. Otras versiones de Linux con soporte comercial son las distribuciones de **Caldera**, **Craftworks** y **WorkGroup Solutions**. El gran seguimiento que Linux ha tenido en Alemania ha hecho que existan varias distribuciones dedicadas en alemán, incluyendo las versiones de **SuSE** y **Unifix**. Existen demasiadas distribuciones Linux en circulación como para enumerarlas todas aquí. Sin embargo, esa diversidad de distribuciones no impide la compatibilidad entre unas distribuciones de Linux y otras. La mayoría de las distribuciones utilizan, o al menos entienden, el formato de archivo de paquete RPM, y los paquetes comerciales distribuidos en este formato pueden instalarse y ejecutarse sobre cualquier distribución que acepte archivos RPM.

21.1.4 Licencias Linux

El *kernel* de Linux se distribuye de acuerdo con la licencia pública general de GNU (GPL, general public license), cuyos términos establece la organización Free Software Foundation. Linux no es un software de dominio público. El término **dominio público** implica que los autores han renunciado a los derechos de propiedad intelectual sobre el software, pero los derechos de propiedad intelectual del código de Linux siguen estando en las manos de los diversos autores del código. Sin embargo, Linux es **software libre**, en el sentido de que cualquiera puede copiarlo, modificarlo, utilizarlo en la forma que desee y proporcionar a otros sus propias copias sin ningún tipo de restricción.

Las principales consecuencias de las condiciones de licencia de Linux son que nadie que utilice Linux, o que cree su propio derivado de Linux (lo que puede perfectamente hacerse), puede hacer que el producto derivado sea propietario. El software distribuido de acuerdo con la licencia GPL no puede redistribuirse como producto exclusivamente binario. Si alguien distribuye algún software que incluya cualquier componente cubierto por la licencia GPL, entonces, bajo los términos de la licencia GPL, debe distribuirse también el código fuente junto con la distribución ejecutable. (Esta restricción no prohíbe la creación ni la venta de distribuciones software en código exclusivamente binario, siempre y cuando cualquiera que reciba ese código binario tenga también la oportunidad de obtener el código fuente, por un precio de distribución razonable.)

21.2 Principios de diseño

En su diseño global, Linux se asemeja a cualquier otra implementación tradicional de UNIX no basada en *microkernel*. Es un sistema multiusuario y multitarea con un conjunto completo de herramientas compatibles con UNIX. El sistema de archivos de Linux respeta la semántica UNIX tradicional y en lo que respecta al modelo de red estándar de UNIX, éste se ha implementado en su totalidad. Los detalles internos del diseño de Linux se han visto muy influidos por la historia del desarrollo de este sistema operativo.

Aunque Linux se ejecuta sobre una amplia variedad de plataformas, fue desarrollado exclusivamente sobre una arquitectura PC. Una buena parte de aquellos primeros desarrollados fue acometida por personas entusiastas, en lugar de por organizaciones de investigación o empresas comerciales bien dotadas de fondos, por lo que desde el principio Linux trató de obtener la máxima funcionalidad posible a partir de unos recursos limitados. Hoy en día, Linux puede ejecutarse holgadamente sobre una máquina multiprocesadora con cientos de megabytes de memoria

principal y muchos gigabytes de espacio de disco, pero también sigue siendo capaz de operar de manera útil en menos de 4 MB de RAM.

A medida que los PC se fueron haciendo más potentes y a medida que fue disminuyendo el coste de la memoria y de los discos duros, los *kernels* Linux originales, de carácter minimalista, fueron creciendo para implementar más funcionalidades de UNIX. La velocidad y la eficiencia siguen siendo objetivos de diseño importantes, pero buena parte del trabajo actual y reciente realizado en Linux se ha concentrado en el tercero de los principales objetivos de diseño: la estandarización. Uno de los precios que hay que pagar por la diversidad de las implementaciones UNIX actualmente disponibles es que el código fuente escrito para una de las versiones puede o no compilarse o ejecutarse correctamente en otra. Incluso cuando están presentes las mismas llamadas al sistema en dos sistemas UNIX diferentes, esas llamadas no se comportan necesariamente de la misma forma exacta. El estándar POSIX comprende un conjunto de especificaciones de diferentes aspectos de comportamiento del sistema operativo. Existen documentos POSIX para la funcionalidad más común del sistema operativo y para extensiones tales como hebras de proceso y operaciones en tiempo real. Linux está diseñado para ser compatible con los documentos POSIX relevantes; hay al menos dos distribuciones Linux que han conseguido la certificación POSIX oficial.

Puesto que presenta interfaces estándar tanto al programador como al usuario, Linux no ofrece muchas sorpresas a cualquiera que esté familiarizado con UNIX. No vamos a detallar esas interfaces aquí; las secciones dedicadas a la interfaz del programador (Sección A.3) y a la interfaz de usuario (Sección A.4) de BSD se aplican también a Linux. Sin embargo, de manera predeterminada, la interfaz de programación de Linux se adhiere a la semántica de UNIX SVR4, en lugar de ajustarse al comportamiento de BSD. Hay disponibles conjuntos separados de bibliotecas para implementar la semántica de BSD en aquellas situaciones en las que los comportamientos son significativamente distintos.

Existen otros muchos estándares en el mundo UNIX, pero la certificación completa de Linux de acuerdo con esos estándares va algo lenta, porque a menudo esa certificación sólo puede obtenerse pagando una determinada licencia, y los gastos necesarios para certificar el cumplimiento de la mayoría de los estándares por parte de un sistema operativo resulta sustancial. Sin embargo, soportar una amplia base de aplicaciones es importante para cualquier sistema operativo, por lo que la implementación de estándares es uno de los objetivos principales del desarrollo de Linux, incluso aunque esa implementación no esté certificada formalmente. Además del estándar POSIX básico, Linux soporta actualmente las extensiones para hebras de POSIX (Pthreads) y un subconjunto de las extensiones POSIX para el control de procesos en tiempo real.

21.2.1 Componentes de un sistema Linux

El sistema Linux está formado por tres cuerpos principales de código, en línea con la mayoría de las implementaciones UNIX más tradicionales:

- 1. Kernel.** El *kernel* es responsable de mantener todas las abstracciones importantes del sistema operativo, incluyendo elementos tales como la memoria virtual y los procesos.
- 2. Bibliotecas del sistema.** Las bibliotecas del sistema definen un conjunto estándar de funciones mediante las que las aplicaciones pueden interactuar con el *kernel*. Estas funciones implementan buena parte de la funcionalidad del sistema operativo que no necesita los privilegios completos del código del *kernel*.
- 3. Utilidades del sistema.** Las utilidades del sistema son programas que realizan tareas individuales y especializadas de gestión. Algunas utilidades del sistema pueden ser invocadas para inicializar y configurar algunos aspectos del sistema; otras (conocidas como *demonios* en la terminología UNIX) pueden ejecutarse de manera permanente, gestionando tareas tales como responder a las conexiones entrantes de red, aceptar solicitudes de inicio de sesión por parte de los terminales y actualizar los archivos de registro.

La Figura 21.1 ilustra los diversos componentes que forman un sistema Linux completo. La distinción más importante aquí es entre el *kernel* y todo lo demás. Todo el código del *kernel* se ejecu-

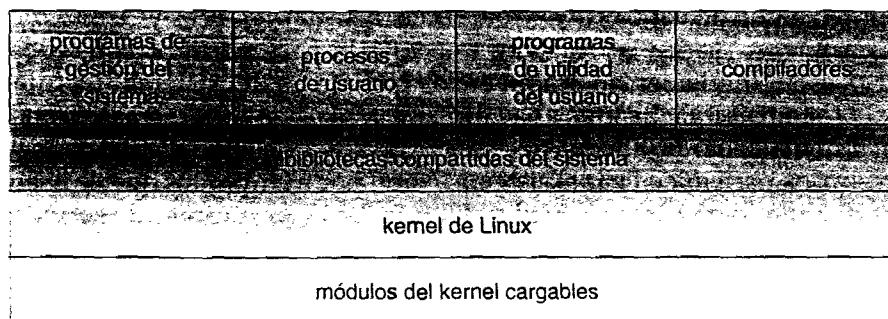


Figura 21.1 Componentes del sistema Linux.

ta en el modo privilegiado del procesador con acceso completo a todos los recursos físicos de la computadora. Linux se refiere a este modo privilegiado con el término **modo *kernel***. En Linux, no hay ningún código en modo usuario incluido dentro del *kernel*. Todo el código de soporte del sistema operativo que no necesite ejecutarse en modo *kernel* se coloca en las bibliotecas del sistema.

Aunque varios sistemas operativos modernos han adoptado una arquitectura de paso de mensajes para el funcionamiento interno de su *kernel*, Linux mantiene el modelo histórico de UNIX: El *kernel* está construido como un código binario monolítico. La principal razón es mejorar las prestaciones: puesto que todo el código y las estructuras de datos del *kernel* ocupan un único espacio de direcciones, no es necesario ningún cambio de contexto cuando un proceso invoca una función del sistema operativo o cuando se produce una interrupción hardware. No es sólo el código principal de planificación y de memoria virtual el que ocupa este espacio de direcciones, todo el código del *kernel*, incluyendo todos los controladores de dispositivos, los sistemas de archivos y el código de interconexión de red está presente en el mismo espacio de direcciones unificado.

Incluso aunque todos los componentes del *kernel* comparten esta misma área, sigue existiendo espacio para la modularidad. De la misma forma que las aplicaciones de usuario pueden cargar bibliotecas compartidas en tiempo de ejecución para integrar una parte de código necesaria, también el *kernel* de Linux puede cargar (y descargar) módulos dinámicamente en tiempo de ejecución. El *kernel* no necesita conocer de antemano qué módulos pueden ser cargados: esos módulos son, verdaderamente, componentes cargables independientes.

El *kernel* de Linux forma la base del sistema operativo Linux. Proporciona toda la funcionalidad necesaria para ejecutar procesos y también proporciona servicios del sistema para que las aplicaciones dispongan de un acceso arbitrado y protegido a los recursos hardware. El *kernel* implementa todas las características requeridas para poder calificarlo como un verdadero sistema operativo. Sin embargo, visto en sí mismo, el sistema operativo proporcionado por el *kernel* de Linux no se parece en nada a un sistema UNIX. Le faltan muchas de las características adicionales de UNIX, y la funcionalidad que proporcionan no está necesariamente en el formato en el que aplicación UNIX la esperaría. El *kernel* no mantiene directamente la interfaz con el sistema operativo que es visible para las aplicaciones que se están ejecutando. En lugar de ello, las aplicaciones realizan llamadas a las bibliotecas del sistema, que a su vez invocan según sea necesario a los servicios del sistema operativo.

Las bibliotecas del sistema proporcionan muchos tipos de funcionalidad. En el nivel más simple, permiten a las aplicaciones realizar solicitudes de servicio al sistema del *kernel*. Realizar una llamada al sistema implica transferir el control desde el modo de usuario no privilegiado al modo de *kernel* privilegiado; los detalles de esta transferencia varían de una arquitectura a otra. Las bibliotecas se encargan de recopilar los argumentos de la llamada al sistema y, en caso necesario, ordenar esos argumentos en la forma especial necesaria para poder hacer la llamada al sistema.

Las bibliotecas también pueden proporcionar versiones más complejas de las llamadas al sistema básicas. Por ejemplo, las funciones de gestión de archivo con búfer del lenguaje C están implementadas en las bibliotecas del sistema, proporcionando un control más avanzado de la E/S de archivo que permiten las llamadas al sistema básicas del *kernel*. Las bibliotecas también proporcionan rutinas que no se corresponden en absoluto con ninguna llamada al sistema, como por

ejemplo algoritmos de ordenación, funciones matemáticas y rutinas de manipulación de cadenas de caracteres. Todas las funciones necesarias para permitir la ejecución de aplicaciones UNIX o POSIX están implementadas aquí en las bibliotecas del sistema.

El sistema Linux incluye una amplia variedad de programas en modo usuario, tanto utilidades del sistema como utilidades de usuario. Las utilidades del sistema incluyen todos los programas necesarios para inicializar el sistema, como por ejemplo los que se usan para configurar dispositivos de red y para cargar módulos del *kernel*. Los programas de servidor que se ejecutan de modo continuo también se consideran utilidades del sistema; dichos programas gestionan solicitudes de inicio de sesión del usuario, conexiones de red entrantes y las colas de impresión.

No todas las utilidades estándar implementan funciones clave de administración del sistema. El entorno de usuario de UNIX contiene un gran número de utilidades estándar para llevar a cabo simplemente tareas cotidianas, como listar el contenido de los directorios, mover y borrar archivos y visualizar el contenido de un archivo. Otras utilidades más complejas pueden realizar funciones de procesamiento de textos, como ordenar datos textuales y buscar patrones en el texto de entrada. Juntas, estas utilidades forman un conjunto de herramientas estándar que los usuarios esperarán encontrar en cualquier sistema UNIX; aunque no realizan ninguna función del sistema operativo, son una parte importante del sistema Linux básico.

21.3 Módulos del kernel

El *kernel* de Linux tiene la capacidad de cargar y descargar secciones arbitrarias del código del *kernel* bajo demanda. Estos módulos cargables *kernel* se ejecutan en modo *kernel* privilegiado y como consecuencia disponen de acceso completo a todas las capacidades hardware de la máquina sobre la que se ejecutan. En teoría, no hay ninguna restricción en cuanto a lo que se permite hacer a un módulo del *kernel*; típicamente, un módulo puede implementar un controlador de dispositivo, un sistema de archivos o un protocolo de red.

Los módulos del *kernel* resultan útiles por varias razones. El código fuente de Linux es gratuito, por lo que cualquier persona que quiera escribir código del *kernel*, puede compilar un *kernel* modificar y rearrancar el sistema para cargar la nueva funcionalidad; sin embargo, recompilar, remontar y recargar el código completo constituye un ciclo de desarrollo bastante engorroso si lo que estamos desarrollando es un nuevo controlador. Si utilizamos módulos del *kernel* no es necesario construir un nuevo *kernel* para probar un nuevo controlador: el controlador puede compilarse por separado y cargarse sobre el *kernel* que se esté ejecutando. Por supuesto, una vez que se ha escrito un nuevo controlador, se puede distribuir en forma de módulo, de modo que otros usuarios puedan beneficiarse del mismo sin tener que reconstruir sus *kernels*.

Este último punto tiene otra implicación importante. Puesto que está cubierto por la licencia GPL, el *kernel* de Linux no puede distribuirse con componentes propietarios añadidos, a menos que dichos nuevos componentes también se distribuyan según la licencia GPL y se ponga el código fuente a disposición de aquellos usuarios que los soliciten. La interfaz de módulos del *kernel* permite que otras personas u organizaciones interesadas escriban y distribuyan, de acuerdo con sus propios términos, controladores de dispositivos o sistemas de archivos que no podrían distribuirse de acuerdo con los términos de licencia de GPL.

Los módulos del *kernel* permiten construir un sistema Linux con un *kernel* estándar mínimo, sin incorporar ningún controlador de dispositivo adicional. Cualquier controlador de dispositivo que el usuario necesite puede ser cargado explícitamente por el sistema en el arranque o ser cargado automáticamente por el sistema bajo demanda, siendo descargado después cuando no se esté utilizando. Por ejemplo, un controlador de CD-ROM podría cargarse cuando se monte un CD y descargarse de memoria cuando se desmonte el CD del sistema de archivos.

El soporte de módulos en Linux tiene tres componentes:

1. El componente de **gestión de módulos** permite cargar módulos en memoria y que estos se comuniquen con el resto del *kernel*.
2. El módulo de **registros de controladores** permite a los módulos informar al resto del *kernel* de que hay disponible un nuevo controlador.

3. El **mecanismo de resolución de conflictos** permite a los diferentes controladores de dispositivo reservar recursos hardware y proteger dichos recursos del uso accidental por parte de otro controlador.

21.3.1 Gestión de módulos

Cargar un módulo requiere algo más que cargar su contenido binario en la memoria del *kernel*. El sistema debe también asegurarse de que todas las referencias que el módulo realice a los puntos de entrada o símbolos del *kernel* se actualicen para apuntar a las ubicaciones correctas dentro del espacio de direcciones del *kernel*. Linux lleva a cabo esta actualización de referencias dividiendo el trabajo de cargar un módulo en dos secciones separadas: la gestión de secciones de código de los módulos en memoria del *kernel* y la gestión de los símbolos que se permite a los módulos que referencien.

Linux mantiene una tabla interna de símbolos en el *kernel*. Esta tabla de símbolos no contiene el conjunto completo de símbolos definido en el *kernel* durante la compilación de éste; en lugar de ello, cada símbolo debe ser exportado explícitamente por el *kernel*. El conjunto de símbolos exportado constituye una interfaz bien definida mediante la que un módulo puede interactuar con el *kernel*.

Aunque exportar de símbolos desde una función del *kernel* requiere una solicitud explícita por parte del programador, no se necesita hacer ningún esfuerzo especial para importar dichos símbolos en un módulo. El desarrollador de un módulo simplemente utiliza el mecanismo estándar de montaje externo del lenguaje C. Todos los símbolos externos a los que haga referencia el módulo y que no estén declarados en el mismo se marcan simplemente como no resueltos en el código binario final del módulo generado por el compilador. Cuando hay que cargar un módulo en el *kernel*, una utilidad del sistema analiza primero el módulo en busca de estas referencias no resueltas. Todos los símbolos que sea necesario resolver se buscan en la tabla de símbolos del *kernel*, introduciéndose en el código del módulo las direcciones correctas de dichos símbolos dentro del *kernel* que se esté actualmente ejecutando. Sólo después de esto se pasa el módulo al *kernel* para cargarlo. Si esta utilidad del sistema no puede resolver alguna referencia contenida en el módulo buscándola en la tabla de símbolos del *kernel*, se rechaza la carga del módulo.

La carga del módulo se realiza en dos etapas. En primer lugar, la utilidad cargadora de módulos pide al *kernel* que reserve un área continua de la memoria virtual del *kernel* para el módulo. El *kernel* devuelve la dirección de la memoria asignada y la utilidad cargadora puede utilizar esta dirección para reubicar el código máquina del módulo de acuerdo con la dirección de carga correcta. Una segunda llamada al sistema pasa entonces el módulo, junto con cualquier tabla de símbolos que el nuevo módulo quiera exportar al *kernel*. El propio módulo se copiará ahora tal cual en el espacio previamente asignado, actualizándose la tabla de símbolos del *kernel* con los nuevos símbolos, con el fin de permitir que sean utilizados por otros módulos que todavía no se hayan cargado.

El componente final de gestión de módulos es el solicitante de módulos. El *kernel* define una interfaz de comunicaciones a la que puede conectarse cualquier programa de gestión de módulos. Una vez establecida esta conexión, el *kernel* informará al proceso de gestión cada vez que un proceso solicite un controlador de dispositivo, un sistema de archivos o un servicio de red que no esté actualmente cargado, dando a ese gestor la oportunidad de cargar dicho servicio. La solicitud original de servicio se completará una vez que el módulo haya sido cargado. El proceso gestor consulta regularmente al *kernel* para ver si se siguen utilizando cada uno de los módulos dinámicamente cargados, y los descarga cuando ya no están siendo activamente utilizados.

21.3.2 Registro de controladores

Una vez cargado un módulo, pasa a ser una simple región aislada de memoria hasta que da a conocer al resto del *kernel* la nueva funcionalidad que ese módulo proporciona. El *kernel* mantiene una serie de tablas dinámicas de todos los controladores conocidos y proporciona un conjunto de rutinas mediante las que se pueden añadir o eliminar controladores de esas tablas en cualquier

instante. El *kernel* invoca la rutina de arranque de un módulo en el momento de cargar de ese módulo e invoca también la rutina de limpieza antes de descargarlo: estas rutinas son las responsables de registrar la funcionalidad del módulo ante el sistema.

Un módulo puede registrar muchos tipos de controladores y puede registrar más de un controlador si así lo desea. Por ejemplo, un controlador de dispositivo podría querer registrar dos mecanismos separados para acceder a ese dispositivo. Las tablas de registro incluyen los elementos siguientes:

- **Controladores de dispositivos.** Estos controladores incluyen dispositivos de caracteres (como impresoras, terminales y ratones), dispositivos de bloques (incluyendo todas las unidades de disco) y dispositivos de interfaz de red.
- **Sistemas de archivo.** El sistema de archivos puede ser cualquier cosa que implemente las rutinas de llamada al sistema de archivos virtual de Linux. Puede implementar un formato para almacenar archivos en disco, pero también podría ser perfectamente un sistema de archivos en red, como NFS, o un sistema de archivos virtual cuyo contenido se genere bajo demanda, como el sistema de archivos /proc de Linux.
- **Protocolos de red.** Un módulo puede implementar un protocolo completo de comunicación por red como IPX, o simplemente un nuevo conjunto de reglas de filtrado de paquetes para un cortafuegos de red.
- **Formato binario.** Este formato especifica una forma de reconocer y cargar un nuevo tipo de archivo ejecutable.

Además, un módulo puede registrar un nuevo conjunto de entradas en las tablas *sysctl* y *proc*, para permitir configurar dinámicamente dicho módulo (Sección 21.7.4).

21.3.3 Resolución de conflictos

Las implementaciones de UNIX comerciales suelen venderse para ejecutarlas sobre el propio hardware del fabricante. Una ventaja de las soluciones de un solo fabricante es que el fabricante del software tiene una idea bastante correcta acerca de las configuraciones hardware que son posibles. Sin embargo, el hardware IBM PC se presenta en un amplísimo número de configuraciones con una gran cantidad de posibles controladores para dispositivos como tarjetas de red, controladoras SCSI y adaptadoras de pantalla de vídeo. El problema de gestionar la configuración hardware se complica cuando se soportan los controladores de dispositivos modulares, ya que el conjunto actualmente activo de dispositivos pasa a variar dinámicamente.

Linux proporciona un mecanismo centralizado de resolución de conflictos como ayuda para arbitrar en el acceso a ciertos recursos hardware. Sus objetivos son los siguientes:

- Evitar que los módulos entren en conflicto a la hora de acceder a los recursos hardware.
- Evitar que las **autocomprobaciones** (las pruebas que un controlador de dispositivo hace para autodetectar la configuración del dispositivo) interfieran con los controladores de dispositivo existentes.
- Resolver los conflictos entre múltiples controladores que están tratando de acceder al mismo hardware; por ejemplo, cuando el controlador de la impresora paralelo y el controlador de red de línea paralelo IP (PLIP, parallel-line IP) tratan de comunicarse con el puerto de impresora paralelo.

Para conseguir estos objetivos, el *kernel* mantiene listas de los recursos hardware asignados. El PC tiene un número limitado de puertos de E/S posibles (direcciones dentro de su espacio de direcciones hardware de E/S), líneas de interrupción y canales DMA; cuando algún controlador de dispositivo quiere acceder a uno de esos recursos, lo que se espera es que reserve primero el recurso, solicitándolo a la base de datos del *kernel*. Este requisito permite además al administrador del sistema determinar exactamente qué recursos han sido asignados a cada controlador en cualquier instante determinado.

Los módulos deben emplear este mecanismo para reservar de antemano los recursos hardware que esperen utilizar. Si se rechaza la solicitud de reserva porque el recurso no está presente o porque ya está en uso, es responsabilidad del módulo decidir qué hacer. El módulo puede dar por cancelada la inicialización y solicitar que se le descargue si no puede continuar, o puede seguir adelante empleando recursos hardware alternativos.

21.4 Gestión de procesos

Un proceso es el contexto básico en el que se da servicio a todas las actividades solicitadas por el usuario dentro del sistema operativo. Para ser compatible con otros sistemas UNIX, Linux debe utilizar un modelo de proceso similar a los de las otras versiones de UNIX. Sin embargo, Linux opera de manera diferente a UNIX en algunos aspectos clave. En esta sección, vamos a repasar el modelo tradicional de procesos de UNIX descrito de la Sección A.3.2 y vamos a presentar el propio modelo de hebras de Linux.

21.4.1 El modelo de procesos fork() y exec()

El principio básico de la gestión de procesos UNIX consiste en separar dos operaciones: la creación de un proceso y la ejecución de un nuevo programa. Un nuevo proceso se crea con la llamada al sistema `fork()` y un nuevo programa se ejecuta después de una llamada a `exec()`. Se trata de dos funciones separadas netamente distintas. Puede crearse un nuevo proceso con `fork()` sin que se ejecute un nuevo programa y el nuevo subproceso simplemente continuará ejecutando exactamente el mismo programa que el proceso padre estaba ejecutando. De la misma forma, para ejecutar un programa no se requiere que se cree primero un nuevo proceso. Cualquier proceso puede invocar `exec()` en cualquier momento. El programa que se esté actualmente ejecutando se terminará inmediatamente y el nuevo programa comenzará a ejecutarse en el contexto del proceso existente.

Este modelo presenta la ventaja de su gran simplicidad. En lugar de tener que especificar cada detalle de un nuevo programa en la llamada al sistema que ejecuta dicho programa, los nuevos programas se ejecutan en su entorno ya existente. Si un proceso padre quiere modificar el entorno en el que hay que ejecutar un nuevo programa, puede crear un nuevo proceso, y luego, mientras se sigue ejecutando el programa original en el proceso hijo, realizar las llamadas al sistema que requiera para modificar ese proceso hijo antes de ejecutar finalmente el nuevo programa.

En UNIX, por tanto, un proceso abarca toda la información que el sistema operativo tiene que mantener para controlar el contexto de una sola ejecución de un único programa. En Linux, podemos descomponer este contexto en una serie de secciones específicas. Las propiedades de los procesos pueden clasificarse en tres grupos: identidad del proceso, entorno y contexto.

21.4.1.1 Identidad del proceso

La identidad del proceso está compuesta principalmente de los siguientes elementos:

- **ID de proceso (PID).** Cada proceso tiene un identificador único. Los identificadores PID se utilizan para especificar procesos al sistema operativo cuando una aplicación realiza una llamada al sistema para señalizar, modificar o esperar a otro proceso. Una serie de identificadores adicionales asocian el proceso con un grupo de procesos (normalmente como un árbol de procesos que se han creado mediante un único comando de usuario `fork`) y con un inicio de sesión concreto.
- **Credenciales.** Cada proceso debe tener un ID de usuario asociado y uno o más identificadores de grupo (los grupos de usuarios se explican en la Sección 10.6.2) que determinan los derechos de un proceso para acceder a archivos y recursos del sistema.
- **Personalidad.** Las personalidades de los procesos no se encuentran tradicionalmente en los sistemas UNIX, pero en Linux cada proceso tiene un identificador de personalidad aso-

ciado que puede modificar ligeramente la semántica de ciertas llamadas al sistema. Las bibliotecas de emulación emplean las personalidades principalmente para solicitar que las llamadas al sistema sean compatibles con ciertas variantes de UNIX.

La mayor parte de estos identificadores están bajo el control limitado del propio proceso. Los identificadores de grupo de procesos y de sesión pueden cambiarse si el proceso quiere iniciar un nuevo grupo o sesión. Sus credenciales pueden modificarse realizando las comprobaciones de seguridad apropiadas. Sin embargo, el PID principal de un proceso no puede cambiar e identifica únicamente al proceso hasta que éste termina.

21.4.1.2 Entorno de un proceso

El entorno de un proceso se hereda de su proceso padre y está compuesto por dos vectores con terminación nula: el vector de argumentos y el vector de entorno. El **vector de argumentos** simplemente enumera los argumentos de línea de comandos para invocar al programa en ejecución; normalmente comienza con el nombre del propio programa. El **vector de entorno** es una lista de parejas “NOMBRE = VALOR” que asocia una serie de variables de entorno con nombre con sus correspondientes valores textuales arbitrarios. El entorno no se almacena en la memoria del *kernel*, sino en el espacio de direcciones en modo usuario del propio proceso con el primer dato en la parte superior de la pila del proceso.

Los vectores de argumentos y de entorno no se modifican cuando se crea un nuevo proceso: el nuevo proceso hijo heredará el entorno que su padre posea. Sin embargo, cuando se invoca un nuevo programa se establece un entorno completamente nuevo. Al invocar a `exec()`, un proceso debe suministrar el entorno para el nuevo programa. El *kernel* pasa estas variables de entorno al siguiente programa, sustituyendo al entorno actual del proceso. Por lo demás, el *kernel* no interfiere con los vectores de entorno y de línea de comandos, su interpretación se deja completamente al arbitrio de las aplicaciones y bibliotecas del modo usuario.

El paso de las variables de entorno de un proceso al siguiente y la herencia de dichas variables por parte de los hijos de un proceso proporciona maneras flexibles de pasar información a los componentes del software del sistema modo usuario. Hay diversas variables de entorno importantes que tienen significados convencionales para determinadas partes de software del sistema. Por ejemplo, la variable `TERM` se configura para nombrar el tipo de terminal conectado a una sesión de usuario; muchos programas utilizan esta variable para determinar cómo realizar operaciones en la pantalla del usuario, por ejemplo, desplazar el cursor o mover una región de texto por la pantalla. Los programas de soporte multilingüe utilizan la variable `LANG` para determinar en qué idioma hay que mostrar los mensajes del sistema.

El mecanismo de variables de entorno permite personalizar el sistema operativo para cada proceso, en lugar de para todo el sistema en su conjunto. Los usuarios pueden seleccionar su propio idioma o sus propios editores independientemente unos de otros.

21.4.1.3 Contexto de un proceso

La identidad del proceso y las propiedades del entorno se suelen configurar en el momento de crear un proceso y no cambian hasta que el proceso termina. Un proceso puede decidir algunos aspectos de su identidad si necesita hacerlo, o puede también alterar su entorno. Por contraste, el contexto de un proceso es el estado del programa en ejecución en un momento determinado, así que está variando constantemente. El contexto de un proceso incluye las siguientes partes:

- **Contexto de planificación.** La parte más importante del contexto de un proceso es su contexto de planificación: la información que necesita el planificador para suspender y reiniciar el proceso. Esta información incluye las copias guardadas de todos los registros procesos. Los registros en coma flotante se almacenan por separado y se restauran sólo cuando es necesario, de modo que los procesos que no utilizan aritmética de coma flotante no tienen que pagar el coste de guardar dicho estado. El contexto de planificación también incluye información acerca de la prioridad de planificación y de las señales pendientes de ser súmi-

nistradas al proceso. Una parte clave del contexto de planificación es la pila del *kernel* del proceso, un área separada de la memoria del *kernel* reservada para uso exclusivo del código en modo *kernel*. Tanto las llamadas al sistema como las interrupciones que se produzcan mientras el proceso se está ejecutando utilizarán esta pila.

- **Contabilidad de recursos.** El *kernel* mantiene información acerca de los recursos que estén actualmente siendo consumidos por cada proceso y acerca de los recursos totales consumidos por el proceso hasta ahora, desde que fue iniciado.
- **Tabla de archivos.** La tabla de archivos es una matriz de punteros a estructuras de archivos del *kernel*. Cuando se realizan llamadas al sistema para E/S de archivos, los procesos hacen referencia a los archivos utilizando su índice correspondiente dentro de esta tabla.
- **Contexto del sistema de archivos.** Mientras que la tabla de archivos enumera los archivos abiertos existentes, el contexto de sistema de archivos se aplica a las solicitudes para abrir nuevos archivos. Aquí se almacenan el directorio raíz inicial y los directorios predeterminados que haya que utilizar para las nuevas búsquedas archivos.
- **Tabla de rutina de tratamiento de señales.** Los sistemas UNIX pueden suministrar señales asíncronas a un proceso en respuesta a diversos sucesos externos. La tabla de rutinas de tratamiento de señales define las rutinas dentro del espacio de direcciones del proceso, que hay que invocar cuando lleguen determinadas señales específicas.
- **Contexto de memoria virtual.** El contexto de memoria virtual describe el contenido completo del espacio de direcciones privado de un proceso; hablaremos de él en la Sección 21.6.

21.4.2 Procesos y hebras

Linux proporciona la llamada al sistema `fork()` con la funcionalidad tradicional de duplicar un proceso. Linux también proporciona la capacidad de crear hebras usando la llamada al sistema `clone()`. Sin embargo, Linux no distingue entre procesos y hebras. De hecho, Linux utiliza generalmente el término *tarea* (en lugar de *proceso* o *hebra*) para referirse a un flujo de control dentro de un programa. Cuando se invoca `clone()`, se le pasa un conjunto de indicadores para determinar el grado de compartición que debe haber entre las tareas padre e hija. Algunos de estos indicadores son los que a continuación se enumeran:

indicador	significado
<code>CLOONE_FS</code>	se comparte la información del sistema de archivos
<code>CLOONE_VM</code>	se comparte el mismo espacio de memoria
<code>CLOONE_SIGHAND</code>	se comparten las rutinas de tratamiento de señales
<code>CLOONE_FILES</code>	se comparte el conjunto de archivos abiertos

Por tanto, si se pasa a `clone()` los indicadores `CLOONE_FS`, `CLOONE_VM`, `CLOONE_SIGHAND` y `CLOONE_FILES`, las tareas padre e hija compartirán la misma información del sistema de archivos (como por ejemplo el directorio de trabajo actual), el mismo espacio de memoria, las mismas rutinas de tratamiento de señales y el mismo conjunto de archivos abiertos. Utilizar `clone()` de esta manera es equivalente a crear una hebra en otros sistemas, ya que la tarea padre comparte la mayor parte de sus recursos con la tarea hija. Sin embargo, si no está activado ninguno de estos indicadores en el momento de invocar `clone()`, no tendrá lugar ninguna compartición, con lo que la funcionalidad es similar a la de la llamada al sistema `fork()`.

La falta de distinción entre procesos y hebras es posible porque Linux no almacena el contexto completo de un proceso dentro de la estructura principal de datos del proceso; en lugar de ello, almacena el contexto dentro de subcontextos independientes. Por tanto, el contexto del sistema de archivos, la tabla de descriptores de archivos, la tabla de rutinas de tratamiento de señales y el contexto de memoria de un proceso se almacenan en estructuras de datos separadas. La estructu-

ra de datos del proceso simplemente contiene punteros a estas otras estructuras, por lo que cualquier número de procesos puede compartir fácilmente un subcontexto apuntando al mismo subcontexto según sea necesario.

Los argumentos de la llamada al sistema `clone()` le dicen a ésta qué subcontextos hay que copiar y cuáles hay que compartir, cuando se crea un nuevo proceso. Al nuevo proceso siempre se le proporciona una nueva identidad y un nuevo contexto de planificación; sin embargo, según los argumentos que se le pasen, puede crear nuevas estructuras de datos de subcontexto que se inicialicen como copias de las del padre, o puede configurar el nuevo proceso para que utilice las mismas estructuras de datos de subcontexto que el padre esté utilizando. La llamada al sistema `fork()` no es otra cosa que un caso especial de `clone()` en el que se copian todos los subcontextos no compartiéndose ninguno de ellos.

21.5 Planificación

La planificación es el acto de asignar el tiempo de CPU a las diferentes tareas dentro de un sistema operativo. Normalmente, pensamos en la planificación como si fuera el acto de ejecutar e interrumpir procesos, pero hay otro aspecto de la planificación que también resulta importante en Linux: la ejecución de las diversas tareas del *kernel*. Las tareas del *kernel* comprenden tanto tareas solicitadas por un proceso en ejecución, como tareas que se ejecuten internamente por cuenta de un controlador de dispositivo.

21.5.1 Planificación de procesos

Linux tiene dos algoritmos separados de planificación de procesos. Uno de ellos es un algoritmo de tiempo compartido para la planificación apropiativa y equitativa entre múltiples procesos; el otro está diseñado para tareas de tiempo real, en la que las prioridades absolutas son más importantes que la equidad.

El algoritmo de planificación utilizado para las tareas normales de tiempo compartido fue enormemente mejorado en la versión 2.5 del *kernel*. Antes de la versión 2.5, el *kernel* de Linux ejecutaba una variante del algoritmo tradicional de UNIX. Entre otros, los principales problemas del planificador tradicional de UNIX son que no proporciona un adecuado soporte para los sistemas SMP y que no puede escalarse bien a medida que crece el número de tareas en el sistema. La mejora realizada en el planificador en la versión 2.5 del *kernel* proporciona ahora un algoritmo de planificación que se ejecuta en tiempo constante [lo que se conoce como $O(1)$], independientemente del número de tareas que haya en el sistema. El nuevo planificador también proporciona un soporte mejorado para SMP, incluyendo mecanismos de afinidad de procesador y de equilibrado de carga, además de mantener la equidad y de proporcionar soporte para tareas interactivas.

El planificador de Linux es un algoritmo apropiativo basado en prioridades, con dos rangos de prioridad separados: un **rango de tiempo real**, que va de 0 hasta 99, y un rango de valores **nice**, que va de 100 a 140. Estos dos rangos se ponen en correspondencia con un esquema de prioridades global en el que los valores numéricamente inferiores indican las prioridades más altas.

A diferencia de los planificadores de muchos otros sistemas, Linux asigna a las tareas de mayor prioridad unos cuantos de tiempo más largos, y viceversa. Debido a la naturaleza original del planificador, esto resulta apropiado para Linux, como pronto veremos. En la Figura 21.2 se ilustra la relación entre las prioridades y los tamaños de los cuantos de tiempo utilizados.

Una tarea se considera elegible para ejecución en la CPU siempre que todavía le quede tiempo en su cuanto de tiempo. Cuando una tarea ha agotado su cuanto de tiempo, se la considera **caducada** y no es elegible de nuevo para ejecución hasta que todas las restantes tareas hayan agotado también su cuanto de tiempo. El *kernel* mantiene una lista de todas las tareas ejecutables en una estructura de datos denominada **cola de ejecución** (*runqueue*). Debido a su soporte para SMP, cada procesador mantiene su propia cola de ejecución y se planifica de manera independiente. Cada cola de ejecución contiene dos matrices de prioridad: matriz de tareas **activas** y matriz de tareas **caducadas**. La matriz de tareas activas contiene todas las tareas que todavía les queda tiempo en su cuanto de tiempo, mientras que la matriz de tareas caducadas contiene todas las tareas cadu-

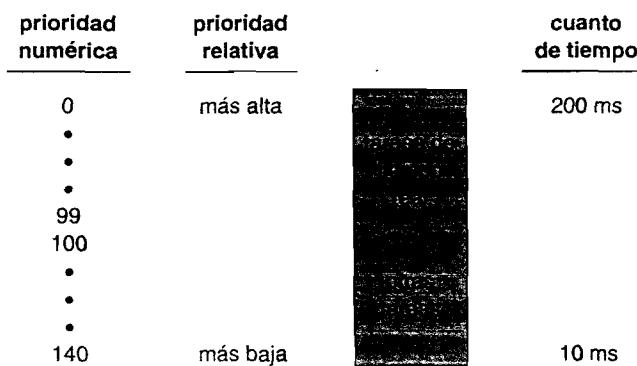


Figura 21.2 Relación entre prioridades y tamaño de los cuantos de tiempo.

cadas. Cada una de estas matrices de prioridad incluye una lista de tareas indexada de acuerdo con la prioridad de cada una (Figura 21.3). El planificador selecciona, para ejecución en la CPU, la tarea con prioridad más alta dentro de la matriz de tareas activas. En las máquinas de tipo multiprocesador, esto quiere decir que cada procesador planifica para ejecución la tarea de prioridad más alta de su propia cola de ejecución. Cuando todas las tareas han agotado sus cuantos de tiempo (cuando la matriz activa está vacía), se intercambian las dos matrices de prioridad, con lo que la matriz de tareas caducadas pasa a ser la matriz de tareas activa, y viceversa.

A las tareas se les asignan prioridades dinámicas basadas en el valor *nice* más o menos un cierto valor menor o igual a 5, dependiendo de la interactividad de la tarea. El que se sume o se reste ese valor adicional del valor *nice* de la tarea dependerá de la interactividad de la misma. La interactividad de una tarea se determina basándose en el tiempo que haya estado durmiendo mientras esperaba a que se realizaran operaciones de E/S. Las tareas más interactivas tienen, normalmente, tiempos de inactividad más largos y es, por tanto, más probable que el valor de ajuste esté más próximo a -5, ya que el planificador favorece dichas tareas interactivas. A la inversa, las tareas con tiempos de inactividad más cortos suelen estar limitadas por el tiempo de procesamiento, de modo que su prioridad se ajustará a la baja.

El recálculo de la prioridad dinámica de cada tarea tiene lugar cuando esa tarea ha agotado su cuento de tiempo y es preciso moverla a la matriz de tareas caducadas. Así, cuando se intercambian las dos matrices, todas las tareas de la nueva matriz de tareas activas tendrán asignadas las nuevas prioridades y sus correspondientes cuantos de tiempo.

La planificación en tiempo real de Linux es todavía más simple. Linux implementa las dos clases de planificación de tiempo real requeridas por POSIX.1b: la planificación FCFS (first-come, first-served) y la planificación por turnos (Secciones 5.3.1 y 5.3.4, respectivamente). En ambos casos, cada proceso tiene una prioridad, además de pertenecer a una cierta clase de planificación. Los procesos con prioridades diferentes pueden competir entre sí hasta un cierto punto cuando se utiliza una planificación de tiempo compartido; sin embargo, en la planificación de tiempo real, el planificador siempre ejecuta el proceso que tenga la mayor prioridad. Entre los procesos de igual prioridad, el planificador ejecuta aquel proceso que haya estado esperando más tiempo. La única

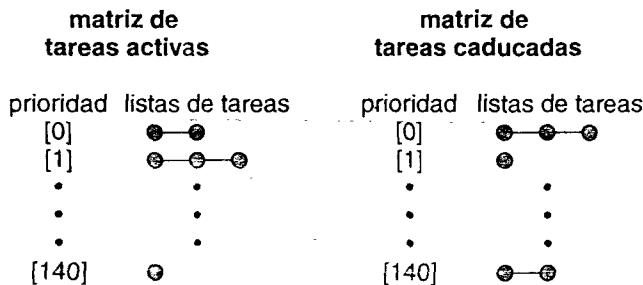


Figura 21.3 Lista de tareas indexada de acuerdo con la prioridad.

diferencia entre la planificación FCFS y la planificación por turnos es que los procesos FCFS continúan ejecutándose hasta que terminan o se bloquean, mientras que un proceso en la planificación por turnos será desalojado después de un cierto tiempo y pasará al final de la cola de planificación, por lo que los procesos de igual prioridad con planificación por turnos estarán, en la práctica, compartiendo el tiempo disponible. A diferencia de lo que sucede con las tareas normales de tiempo compartido, a las tareas de tiempo real se les asigna prioridades estáticas.

La planificación de tiempo de real de Linux es un mecanismo de tiempo real no estricto. El planificador ofrece unas garantías estrictas acerca de las prioridades relativas de los procesos de tiempo real, pero el *kernel* no ofrece ninguna garantía sobre la rapidez con que un proceso de tiempo real será planificado para ejecución una vez que ese proceso pase a ser ejecutable.

21.5.2 Sincronización del *kernel*

La forma en que el *kernel* planifica sus propias operaciones es fundamentalmente distinta de la forma en que planifica los procesos. Una solicitud para ejecución en modo *kernel* puede producirse de dos formas. Un programa en ejecución puede solicitar un servicio del sistema operativo, bien explícitamente mediante una llamada al sistema, o bien implícitamente (por ejemplo, cuando se produce un fallo de página). Alternativamente, un controlador de dispositivo puede generar una interrupción hardware que haga que la CPU comience a ejecutar una rutina de tratamiento definida por el *kernel* para dicha interrupción.

El problema que se le plantea al *kernel* es que todas estas tareas pueden tratar de acceder a las mismas estructuras internas de datos. Si una tarea del *kernel* está accediendo a alguna estructura de datos en el momento en que se ejecuta una rutina de servicio de interrupción, entonces dicha rutina de servicio no puede acceder a esos mismos datos ni modificarlos sin que se corra el riesgo de que los datos se corrompan. Este hecho se relaciona con la idea de secciones críticas, es decir, partes de código que acceden a datos compartidos y a las que no se les debe permitir que se ejecuten de manera concurrente. Como resultado, la sincronización del *kernel* implica mucho más que simplemente planificar los procesos. Se requiere un marco de trabajo que permita a las tareas del *kernel* ejecutarse sin violar la integridad de los datos compartidos.

Antes de la versión 2.6, Linux era un *kernel* no apropiativo, lo que quiere decir que un proceso que se estuviera ejecutando en modo *kernel* no podía ser desalojado, ni siquiera si pasaba a estar disponible para ejecución otro proceso de mayor prioridad. Con la versión 2.6, el *kernel* de Linux se ha hecho completamente apropiativo, de modo que ahora una tarea puede ser desalojada cuando se está ejecutando en el *kernel*.

El *kernel* de Linux proporciona cerros de bucle sin fin (*spinlock*) y semáforos (así como versiones lector-escritor de estos dos tipos de bloqueo), para los bloqueos en el *kernel*. En las máquinas SMP, el mecanismo de bloqueo fundamental es un cerrojo de bucle sin fin, el *kernel* está diseñado de forma que el cerrojo sólo se mantiene durante períodos muy cortos. En las máquinas de un solo procesador, estos cerros resultan inapropiados y se sustituyen por un sistema de activación y desactivación del mecanismo de apropiación del *kernel*. En otras palabras, en las máquinas monoprocesador, en lugar de mantener un cerrojo de bucle sin fin, la tarea desactiva el mecanismo de apropiación del *kernel*. Cuando ha finalizado con el bloqueo, en lugar de liberar un cerrojo, lo que hace la tarea es volver a activar el mecanismo de apropiación en el *kernel*. Este comportamiento se resume en la siguiente tabla:

monoprocesador	multiprocesador
Desactiva apropiación en el <i>kernel</i>	Adquiere cerrojo de bucle sin fin
Activa apropiación en el <i>kernel</i>	Liberar cerrojo de bucle sin fin

Linux utiliza una técnica interesante para activar y desactivar el mecanismo de apropiación del *kernel*. Proporciona dos llamadas al sistema simples, `preempt_disable()` y `preempt_enable()`, para desactivar y activar el mecanismo de apropiación del *kernel*. Sin embargo, ade-

más de esto, el *kernel* no es apropiativo si hay una tarea en modo *kernel* que esté manteniendo un bloqueo. Para imponer esta regla, cada tarea tiene una estructura *thread_info* que incluye el campo *preempt_count*, que es un contador que indica el número de cerrojos que mantiene la tarea. Cuando se adquiere un cerrojo, se incrementa *preempt_count*. De la misma forma, el contador se decrementa cuando se libera un cerrojo. Si el valor de *preempt_count* para la tarea que se está actualmente ejecutando es mayor que cero, no resulta seguro desalojar a la tarea ya que esa tarea mantiene al menos un cerrojo. Si el contador tiene un valor igual a cero, puede interrumpirse al *kernel* sin problemas, suponiendo que no haya ninguna llamada pendiente a *preempt_disable()*.

Los cerrojos de bucle sin fin (junto con el sistema de activación y desactivación del mecanismo de apropiación del *kernel*) se utilizan en el *kernel* sólo cuando el cerrojo se mantiene durante un tiempo muy corto. Cuando es necesario mantener un cerrojo durante períodos más largos se utilizan semáforos.

La segunda técnica de protección que utiliza Linux se aplica a las secciones críticas contenidas en las rutinas de servicio de interrupciones. La herramienta básica es el hardware de control de interrupciones del procesador. Desactivando las interrupciones (o utilizando cerrojos de bucle sin fin) durante una sección crítica, el *kernel* garantiza que se pueda continuar operando sin correr el riesgo de un acceso concurrente a las estructuras de datos compartidas.

Sin embargo, existe un precio a pagar por la desactivación de las interrupciones. En la mayoría de las arquitecturas hardware, las instrucciones de activación y desactivación de las interrupciones son costosas. Además, mientras las interrupciones permanezcan desactivadas, se interrumpen todas las operaciones de E/S y cualquier dispositivo que esté esperando a ser servido tendrá que esperar hasta que se vuelvan a activar las interrupciones; en consecuencia, las prestaciones se degradan. El *kernel* de Linux utiliza una arquitectura de sincronización que permite ejecutar largas secciones críticas por completo, sin tener que desactivar las interrupciones. Esta capacidad resulta especialmente útil en el código de comunicación por red: una interrupción en un controlador de dispositivo de red puede señalar la llegada de un paquete de red completo, lo que puede dar como resultado que se ejecute una gran cantidad de código para desensamblar, encaminar y re-enviar ese paquete dentro de la rutina de servicio de la interrupción.

Linux implementa esta arquitectura separando en dos secciones las rutinas de servicio de interrupciones: la mitad superior y la mitad inferior. La **mitad superior** es una rutina normal de servicio de interrupción y se ejecuta teniendo desactivadas las interrupciones recursivas; otras interrupciones de mayor prioridad pueden interrumpir a la rutina, pero no estarán permitidas las interrupciones de la misma prioridad o de una prioridad menor. La **mitad inferior** de una rutina de servicio se ejecuta con todas las interrupciones activadas, empleándose un planificador en miniatura que garantiza que esas mitades inferiores no se interrumpan nunca a sí mismas. El planificador de mitades inferiores se invoca automáticamente en el momento de salir de una rutina de servicio de interrupción.

Esta separación implica que el *kernel* puede completar cualquier procesamiento complejo que haya que realizar en respuesta a una interrupción sin preocuparse de que pudiera darse el caso de que se interrumpiera a sí mismo. Si se produce otra interrupción mientras se está ejecutando una mitad inferior, entonces esa interrupción puede solicitar que se ejecute esa misma mitad inferior pero la ejecución será diferida hasta que se complete la mitad inferior que se esté ejecutando actualmente. Cada ejecución de la mitad inferior puede ser interrumpida por una mitad superior pero nunca por otra mitad inferior similar.

La arquitectura basada en esa distinción mitad superior/mitad inferior se completa mediante un mecanismo para desactivar una serie de mitades inferiores seleccionadas mientras se ejecuta el código normal de primer plano del *kernel*. De este modo, pueden incluirse fácilmente secciones críticas de código en el *kernel*. Las rutinas de tratamiento de interrupciones pueden incluir el código de sus secciones críticas en las secciones inferiores; y cuando el cuerpo principal del *kernel* quiera entrar en una sección crítica, puede desactivar las mitades inferiores relevantes para evitar que otras secciones críticas le interrumpan. Al final de la sección crítica, el *kernel* puede volver a activar las mitades inferiores y ejecutar aquellas mitades inferiores que hayan sido puestas en cola por las mitades superiores de las rutinas de servicio de interrupciones durante la sección crítica.

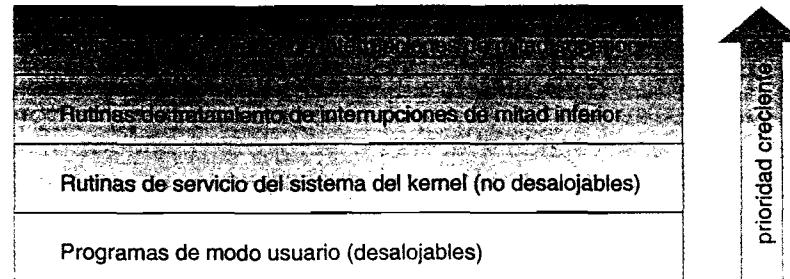


Figura 21.4 Niveles de protección de interrupción.

La Figura 21.4 resume los diversos niveles de protección de interrupciones dentro del *kernel*. Cada nivel puede ser interrumpido por código que se esté ejecutando en un nivel superior, pero nunca será interrumpido por código que se ejecute en el mismo nivel o en un nivel inferior; salvo por lo que respecta al código en modo usuario, los procesos de usuario siempre pueden ser desalojados por otro proceso cuando se produce una interrupción de planificación de tiempo compartido.

21.5.3 Multiprocesamiento simétrico

El *kernel* de Linux 2.0 fue el primer *kernel* de Linux estable en soportar hardware de **multiprocesamiento simétrico** (SMP), lo que permite que una serie de procesos separados se ejecuten en paralelo en diferentes procesadores. Originalmente, la implementación de SMP imponía la restricción de que sólo un procesador podía estar ejecutando código en modo *kernel* en cada instante.

En la versión 2.2 del *kernel*, se creó un único cerrojo de bucle sin fin del *kernel* (en ocasiones denominado BKL, big *kernel* lock) para permitir que estuvieran activos concurrentemente en el *kernel* múltiples procesos (ejecutándose en diferentes procesadores). Sin embargo, el BKL proporcionaba un nivel muy grueso de granularidad de bloqueo. Las posteriores versiones del *kernel* hicieron que la implementación SMP fuera más escalable, dividiendo este único cerrojo de bucle sin fin en múltiples cerrojos, cada uno de los cuales protege únicamente un pequeño subconjunto de las estructuras de datos del *kernel*. Dichos cerrojos de bucle sin fin se describen en la Sección 21.5.2. El *kernel* 2.6 proporcionaba mejoras adicionales en el campo SMP, incluyendo mecanismos de afinidad de procesador y algoritmos de equilibrado de carga.

21.6 Gestión de memoria

La gestión de memoria en Linux tiene dos componentes. El primero se encarga de la asignación y de la liberación de la memoria física: páginas, grupos de página y pequeños bloques de memoria. El segundo gestiona la memoria virtual, que es la memoria mapeada sobre el espacio de direcciones de los procesos que se está ejecutando. En esta sección, vamos a describir estos dos componentes y a examinar después los mecanismos mediante los que los componentes cargables de un nuevo programa se cargan dentro de la memoria virtual de un proceso en respuesta a una llamada al sistema `exec()`.

21.6.1 Gestión de memoria física

Debido a características hardware específicas, Linux separa la memoria física en tres **zonas** distintas, que identifican diferentes regiones de memoria. Las zonas se identifican como:

- ZONE_DMA
- ZONE_NORMAL
- ZONE_HIGHMEM

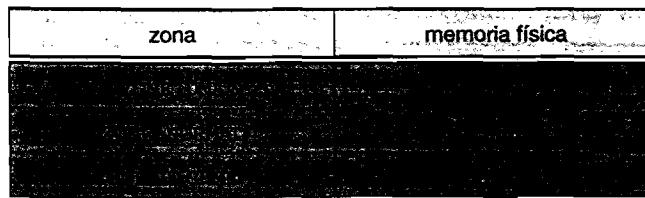


Figura 21.5 Relaciones entre zonas y direcciones físicas en el Intel 80x86.

Estas zonas son específicas de la arquitectura. Por ejemplo, en la arquitectura Intel 80x86, ciertos dispositivos ISA (industry standard architecture) sólo pueden acceder a los 16 MB más bajos de la memoria física utilizando DMA. En estos sistemas, los primeros 16 MB de la memoria física forman la zona ZONE_DMA. ZONE_NORMAL identifica la memoria física que se mapea sobre el espacio de direcciones de la CPU. Esta zona se utiliza para la mayoría de las solicitudes de memoria normales. Para las arquitecturas que no impongan un límite a la zona de memoria a la que se puede acceder mediante DMA, ZONE_DMA no está presente, utilizándose ZONE_NORMAL. Finalmente, ZONE_HIGHMEM (“high memory”) hace referencia a la memoria física que no está mapeada sobre el espacio de direcciones del *kernel*. Por ejemplo, en la arquitectura de 32 bits de Intel (en la que 2^{32} proporciona 4 GB de espacio de direcciones), el *kernel* se mapea sobre los primeros 896 MB del espacio de direcciones; la memoria restante se denomina memoria alta y se asigna a partir de ZONE_HIGHMEM. En la Figura 21.5 se muestra la relación entre las zonas y las direcciones físicas en la arquitectura Intel 80x86. El *kernel* mantiene una lista de páginas libres para cada zona. Cuando llega una solicitud de memoria física, el *kernel* satisface la solicitud usando la zona apropiada.

El gestor principal de memoria física en el *kernel* de Linux es el **asignador de páginas**. Cada zona tiene su propio asignador que es responsable de asignar y liberar todas las páginas físicas de la zona, y es capaz de asignar rangos de páginas físicamente contiguas según se vaya solicitando. El asignador utiliza un **sistema de descomposición binaria** (Sección 9.8.1), para controlar las páginas físicas disponibles. Según este esquema, se agrupan las unidades adyacentes de memoria assignable. Cada región de memoria assignable tiene una región adyacente asociada. Cuando se liberan dos regiones asociadas adyacentes, se las combina para formar una región más grande. Dicha región tiene también otra región asociada con la que se puede combinar para formar una región todavía mayor. A la inversa, si no se puede satisfacer una solicitud de memoria de pequeño tamaño, asignando una región libre pequeña ya existente, se subdividirá una región libre de mayor tamaño en dos regiones asociadas con el fin de satisfacer la solicitud. Se utilizan listas enlazadas separadas para llevar la cuenta de las regiones libres de memoria de cada uno de los tamaños permitidos; en Linux, el tamaño assignable más pequeño con este mecanismo es una única página física. La Figura 21.6 muestra un ejemplo de asignación por descomposición binaria. Se quiere asignar una región de 4 KB, pero la región más pequeña disponible es una de 16 KB. En consecuencia, esta región se descompone recursivamente hasta obtener un fragmento del tamaño deseado.

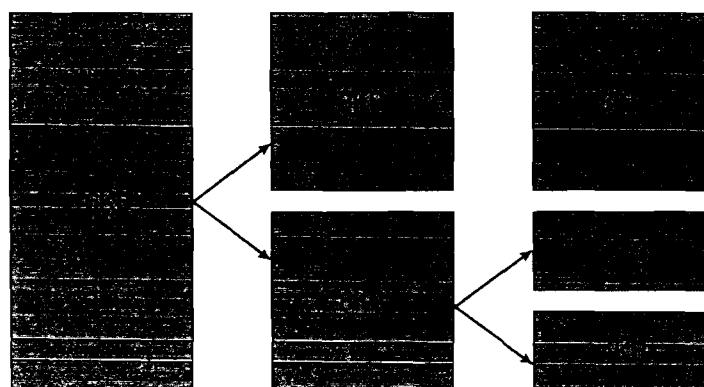


Figura 21.6 División de la memoria en el sistema de descomposición binaria.

En último término, todas las asignaciones de memoria dentro del *kernel* de Linux se realizan estáticamente (por controladores que reservan un área continua de memoria durante el arranque del sistema) o dinámicamente (por parte del asignador de página). Sin embargo, las funciones del *kernel* no tienen que usar obligatoriamente el asignador básico para reservar memoria. Varios subsistemas especializados de gestión de memoria emplean el asignador de páginas subyacentes para gestionar sus propias secciones de memoria. Los más importantes son el sistema de memoria virtual, descrito en la Sección 21.6.2, el asignador de tamaño variable `kmalloc()`; el asignador de franjas utilizado para asignar memoria para las estructuras de datos del *kernel* y la caché de páginas, utilizada para almacenar en caché las páginas pertenecientes a los archivos.

Muchos componentes del sistema operativo Linux necesitan asignar páginas completas de acuerdo con las solicitudes recibidas, pero a menudo se necesitan bloques de memoria más pequeños. El *kernel* proporciona un asignador adicional para solicitudes de tamaño arbitrario, en las que el tamaño de la solicitud no se conoce de antemano y puede ser de sólo unos pocos bytes, en lugar de una página completa. De forma análoga, a la función `malloc()` del lenguaje C, este servicio `kmalloc()` asigna páginas completa bajo demanda, pero luego las descompone en fragmentos más pequeños. El *kernel* mantiene un conjunto de listas de páginas que estén siendo utilizadas por el servicio `kmalloc()`. La asignación de memoria implica utilizar la lista apropiada y extraer de ella el fragmento disponible o asignar una nueva página y dividirla. Las regiones de memoria reservadas por el sistema `kmalloc()` se asignan permanentemente hasta que se las libera de forma explícita; el sistema `kmalloc()` no puede reubicar ni reclamar estas regiones en respuesta a las situaciones de falta de memoria.

Otra estrategia adoptada por Linux para asignar memoria del *kernel* se conoce con el nombre de asignación de franjas. Una **franja** se utiliza para asignar memoria para las estructuras de datos del *kernel* y está compuesta de una o más páginas físicamente contiguas. Cada **caché** está compuesta de una o más franjas y existe una sola caché por cada estructura de datos del *kernel* distinta: por ejemplo, una caché para la estructura de datos que representa los descriptores de procesos, otra caché para los objetos de archivos, una caché para los semáforos, etc. Cada caché se rellena con **objetos**, que son instancias de las estructuras de datos del *kernel* que esa caché representa. Por ejemplo, la caché que representa los semáforos almacena instancias de objetos semáforo, la caché que representa descriptores de procesos almacena instancias de objetos descriptor de proceso, etc. En la Figura 21.7 se muestra la relación entre franjas, cachés y objetos. La figura muestra dos objetos del *kernel* de 3 KB de tamaño y tres objetos de 7 KB de tamaño. Estos objetos están almacenados en las cachés respectivas para los objetos de 3 KB y 7 KB.

El algoritmo de asignación de franjas utiliza cachés para almacenar objetos del *kernel*. Cuando se crea una caché, se la asigna una serie de objetos, que están inicialmente marcados como **libres**.

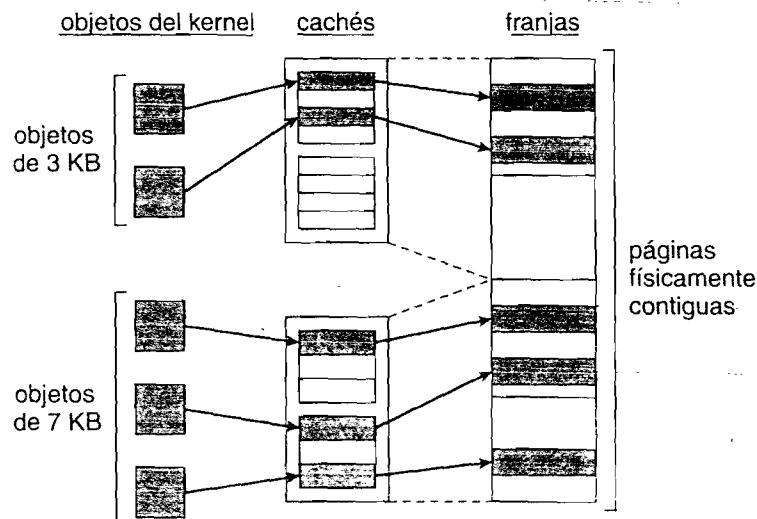


Figura 21.7 Asignador de franjas en Linux.

El número de objetos en la caché dependerá del tamaño de la franja asociada. Por ejemplo, una franja de 12 KB (compuesta de tres páginas de 4 KB contiguas) podría almacenar 6 objetos de 2 KB. Inicialmente, todos los objetos de la caché están marcados como libres. Cuando se necesita un nuevo objeto para una estructura de datos del *kernel*, el asignador puede asignar cualquier objeto libre de la caché para satisfacer la solicitud. El objeto asignado de la caché se marca como **usado**.

Consideremos un escenario en el que el *kernel* solicita memoria al asignador de franjas para un objeto que representa al descriptor de un proceso. En los sistemas Linux, un descriptor de proceso tiene el tipo `struct task_struct`, que requiere aproximadamente 1,7 KB de memoria. Cuando el *kernel* de Linux crea una nueva tarea, solicita la memoria necesaria para el objeto `struct task_struct` a la caché correspondiente. La caché satisfará la solicitud utilizando un objeto `struct task_struct` que ya haya sido asignado en una franja y que esté marcado como libre.

En Linux, una franja puede estar en uno de tres posibles estados:

1. **Llena.** Todos los objetos de la franja están marcados como utilizados.
2. **Vacía.** Todos los objetos de la franja están marcados como libres.
3. **Parcial.** La franja está compuesta de objetos tanto libres como utilizados.

El asignador de franjas primero intenta satisfacer la solicitud con un objeto libre de una franja parcial. Si no existe ninguno, se asigna un objeto libre de una franja vacía. Si no hay disponible ninguna franja vacía, se define una nueva franja formada por una serie de páginas físicamente contiguas y se la asigna a una caché; la memoria para el objeto se asigna a partir de esta franja.

Los otros dos subsistemas principales de Linux que realizan su propia gestión de páginas físicas están estrechamente relacionados entre sí. Se trata de la caché de páginas y del sistema de memoria virtual. La **caché de páginas** es la caché principal del *kernel* para los dispositivos orientados a bloques y para los archivos mapeados en memoria, y constituye el mecanismo principal mediante el que se llevan a cabo las operaciones de E/S con estos dispositivos. Tanto los sistemas de archivo nativos de Linux, basados en disco, como el sistema de archivos en red NFS utilizan la caché de páginas. La caché de páginas almacena páginas completas de los archivos y no está limitada a los dispositivos de bloques; también pueden almacenarse en caché datos relacionados con la comunicación por red. El sistema de memoria virtual gestiona el contenido del espacio de direcciones virtual de cada proceso. Estos dos sistemas interactúan estrechamente, porque leer una página de datos para cargarla en la caché de páginas requiere mapear las páginas sobre la caché de páginas empleando el sistema de memoria virtual. En las secciones siguientes, vamos a examinar el sistema de memoria virtual con mayor detalle.

21.6.2 Memoria virtual

El sistema de memoria virtual de Linux es responsable de mantener el espacio de direcciones visible para cada proceso. Este sistema crea páginas de memoria virtual bajo demanda y gestiona la carga de dichas páginas desde el disco o la escritura de esas páginas en el espacio de intercambio del disco, según sea necesario. En Linux, el gestor de memoria virtual mantiene dos vistas separadas del espacio de direcciones de un proceso: una como conjunto de regiones separadas y otra como conjunto de páginas.

La primera vista de un espacio de direcciones es la vista lógica, que describe las instrucciones que el sistema de memoria virtual ha recibido en lo que respecta a la disposición del espacio de direcciones. En esta vista, el espacio de direcciones consta de un conjunto de regiones no solapadas, donde cada región representa un subconjunto continuo del espacio de direcciones con alineación de página. Cada región se describe internamente mediante una única estructura `vm_area_struct` que define las propiedades de la región, incluyendo los permisos de lectura, escritura y ejecución del proceso para dicha región e información acerca de cualesquiera archivos que estén asociados con esa región. Las regiones de cada espacio de direcciones están enlazadas

en un árbol binario equilibrado, con el fin de permitir la búsqueda rápida de la región correspondiente a cualquier dirección virtual.

El *kernel* también mantiene una segunda vista, de carácter físico, de cada espacio de direcciones. Esta vista está almacenada en las tablas de páginas hardware del proceso. Las entradas de la tabla de páginas determinan la ubicación actual exacta de cada página de memoria virtual, independientemente de si se encuentra en disco o en la memoria física. La vista física está gestionada por un conjunto de rutinas que se invocan desde las rutinas de servicio de interrupciones software del *kernel* cada vez que un proceso trata de acceder a una página que no se encuentra actualmente en las tablas de páginas. Cada estructura `vm_area_struct` de la descripción del espacio de direcciones contiene un campo que apunta a una tabla de funciones que implementa las funciones clave de gestión de páginas para cualquier región de memoria virtual dada. Todas las solicitudes de leer o escribir una página no disponible terminan por ser despachadas a la rutina de tratamiento apropiada en la tabla de funciones de la estructura `vm_area_struct`, de modo que las rutinas centrales de gestión de memoria no necesitan conocer los detalles relativos a cómo gestionar cada uno de los tipos posibles de regiones de memoria.

21.6.2.1 Regiones de memoria virtual

Linux implementa varios tipos de regiones de memoria virtual. La primera propiedad que caracteriza a un tipo de memoria virtual es el almacenamiento de respaldo utilizado para la región, que describe el lugar del que proceden las páginas de esa región. La mayoría de las regiones de memoria utilizan como origen un archivo o nada en absoluto. Una región que no tenga nada como origen es el tipo más simple de memoria virtual. Dicha región representa lo que se denomina una **memoria demandada con ceros**. Cuando un proceso trata de leer una página en una de esas regiones, simplemente se le devuelve una página de memoria rellena con ceros.

Una región que tenga como origen un archivo actúa como una especie de visor de una sección de ese archivo. Cuando el proceso intenta acceder a una página dentro de esa región, la tabla de páginas se llena con la dirección de una página dentro de la caché del *kernel* que se corresponde con el desplazamiento apropiado dentro del archivo. La caché de páginas y las tablas de páginas del proceso utilizan la misma página de memoria física, por lo que cualquier cambio hecho en el archivo por el sistema de archivos será inmediatamente visible para cualquier proceso que haya mapeado dicho archivo en su espacio de direcciones. Cualquier número de procesos puede mapear la misma región del mismo archivo, y todos ellos terminarán usando la misma página de memoria física.

Una región de memoria virtual también queda definida por su reacción frente a las escrituras. El mapeado de una región sobre el espacio de direcciones de un proceso puede ser *privado* o *compartido*. Si un proceso escribe sobre una región mapeada de manera privada, el paginador detectará que es necesario realizar una copia durante la escritura con el fin de que los cambios continúen siendo locales al proceso. Por contraste, las escrituras en una región compartida provocan que se actualice el objeto mapeado sobre dicha región, de modo que el cambio será inmediatamente visible para cualquier proceso que esté mapeando el objeto.

21.6.2.2 Duración de un espacio de direcciones virtual

El *kernel* creará un nuevo espacio de direcciones virtual en dos situaciones distintas: cuando un proceso ejecute un nuevo programa mediante la llamada al sistema `exec()` y al crear un nuevo proceso mediante la llamada al sistema `fork()`. El primer caso resulta sencillo: cuando se ejecuta un nuevo proceso, al proceso se le asigna un espacio de direcciones virtual nuevo completamente vacío. Es responsabilidad de las rutinas que cargan el programa llenar el espacio de direcciones con regiones de memoria virtual.

El segundo caso, el de creación de un nuevo proceso mediante `fork()`, implica crear una copia completa del espacio de direcciones virtual del proceso existente. El *kernel* copia los descriptores `vm_area_struct` del proceso padre y luego crea un nuevo conjunto de tablas de páginas para el hijo. Las tablas de páginas del padre se copian directamente en las del hijo, y el contador de refe-

referencias de cada página se incrementa; así, después de la creación del proceso hijo, el hijo y el padre comparten las mismas páginas físicas de memoria en sus espacios de direcciones.

Un caso especial es el que se produce cuando la operación de copia alcanza una región de memoria virtual que está mapeada de forma privada. Todas las páginas en las que haya escrito el proceso padre dentro de dicha región serán privadas, por lo que los subsiguientes cambios hechos en esas páginas por parte del padre o del hijo no deben actualizar la página en el espacio de direcciones del otro proceso. Cuando se copian las entradas de la tabla de páginas para dichas regiones, se las configura como de sólo lectura y se las marca para aplicar la operación de copia durante la escritura. Mientras que ninguno de los procesos modifique estas páginas, los dos procesos compartirán la misma página de memoria física. Sin embargo, si algunos de los procesos tratan de modificar una página de copia durante la escritura, se comprueba el contador de referencias de la página. Si la página sigue estando compartida, entonces el proceso copia el contenido de la página en una nueva página de memoria física y utiliza, en su lugar, esa copia recién creada. Este mecanismo garantiza que se compartan entre los procesos, siempre que sea posible, las páginas de datos privados y que sólo se realizan copias cuando sea absolutamente necesario.

21.6.2.3 Intercambio y paginación

Una tarea importante que debe llevar a cabo cualquier sistema de memoria virtual es la de reubicar las páginas de memoria, extrayéndolas de la memoria física y almacenándolas en el disco cuando esa memoria sea necesaria para otras cosas. Los sistemas UNIX más antiguos realizaban esta reubicación intercambiando todo el contenido de un proceso completo, pero las versiones modernas de UNIX utilizan más el mecanismo de paginación: el movimiento de páginas individuales de memoria virtual entre memoria física y disco. Linux no implementa el mecanismo de intercambio de procesos completos, sino que utiliza exclusivamente el mecanismo más reciente de paginación.

El sistema de paginación puede dividirse en dos secciones. En primer lugar, el **algoritmo de política** decide qué páginas escribir en el disco y cuándo escribirlas. En segundo lugar, el **mecanismo de paginación** lleva a cabo la transferencia y vuelve a cargar los datos en memoria física cuando son necesarios de nuevo.

La **política de descarga de páginas** de Linux utiliza una versión modificada del algoritmo estándar del reloj (o de segunda oportunidad) descrito en la Sección 9.4.5.2. En Linux, se utiliza un reloj con múltiples pasadas y cada página tiene una *edad* que se ajusta en cada pasada del reloj. Esta edad es, para ser más precisos, una medida de la “juventud” de una página, o de cuánta actividad se ha realizado recientemente con esa página. Las páginas a las que se acceda frecuentemente tendrán un valor de edad más alto, pero la edad de las páginas a las que se acceda de manera infrecuente caerá hacia cero con cada pasada. Estos valores de edad permiten al paginador seleccionar las páginas que hay que descargar, basándose en una política del tipo LFU (least frequently used, menos frecuentemente utilizada).

El mecanismo de paginación soporta la paginación tanto con particiones y dispositivos de intercambio dedicados, como con archivos normales, aunque el intercambio con un archivo es significativamente más lento, debido al procesamiento adicional que debe realizar el sistema de archivos. Los bloques se asignan a partir de los dispositivos de intercambio de acuerdo con un mapa de bits de bloques utilizados, que se mantiene en memoria física en todo instante. El asignador utiliza un algoritmo de siguiente ajuste con el fin de tratar de escribir las páginas en secciones continuas de bloques de disco, para mejorar la velocidad. El asignador registra el hecho de que se ha descargado una página en disco, empleando una característica de las tablas de páginas de los procesadores modernos: se activa el bit de “página no presente” de la entrada de la tabla de páginas, permitiendo llenar el resto de esa entrada de la tabla de páginas con un índice que identifique dónde se ha escrito la página.

21.6.2.4 Memoria virtual del kernel

Linux reserva para su propio uso interno una región constante, dependiente de la arquitectura, del espacio de direcciones virtual de cada proceso. Las entradas de la tabla de páginas que se

corresponden con estas páginas del *kernel* se marcan como protegidas, para que esas páginas no sean visibles ni modificables cuando el procesador esté operando en modo usuario. Esta área de memoria virtual del *kernel* contiene dos regiones. La primera es un área estática que contiene referencias de la tabla de páginas a cada una de las páginas físicas de memoria disponibles en el sistema, para poder realizar una traducción simple entre direcciones físicas y virtuales cuando se esté ejecutando código del *kernel*. La parte fundamental del *kernel* junto con todas las páginas asignadas por el asignador de páginas normal reside en esta región.

El resto de la sección del espacio de direcciones reservada por el *kernel* no está reservado para ningún proceso específico. Las entradas de la tabla de páginas dentro de este rango de direcciones pueden ser modificadas por el *kernel* con el fin de que apunten a otras áreas de memoria. El *kernel* proporciona un par de funciones que permiten a los procesos usar esta memoria virtual. La función `vmalloc()` asigna un número arbitrario de páginas de memoria física que pueden no estar físicamente contiguas en una única región de memoria virtual contigua del *kernel*. La función `vremap()` hace corresponder una secuencia de direcciones virtuales con un área de memoria usada por un controlador de dispositivo para las operaciones de E/S mapeadas en memoria.

21.6.3 Ejecución y carga de programas de usuario

La ejecución de programas de usuario por parte del *kernel* de Linux se inicia mediante una invocación a la llamada al sistema `exec()`. Esta llamada hace que el *kernel* ejecute un nuevo programa dentro del proceso actual, sobreescritiendo completamente el contexto actual de ejecución con el contexto inicial del nuevo programa. El primer cometido de este servicio del sistema consiste en verificar que el proceso que realiza la invocación tiene los permisos correctos sobre el archivo que se va a ejecutar. Una vez efectuada esta comprobación, el *kernel* invoca una rutina de carga para comenzar a ejecutar el programa. El cargador no carga necesariamente el contenido del archivo de programa en memoria física, pero al menos prepara el mapeo del programa sobre memoria virtual.

No existe ninguna rutina diferenciada en Linux para cargar un nuevo programa. En su lugar, Linux mantiene una tabla de posibles funciones cargadoras, y da a cada una de esas funciones la oportunidad de que intente cargar el archivo especificado cuando se realice una llamada al sistema `exec()`. La razón inicial para la existencia esta tabla de rutinas cargadoras era que, entre las versiones de los *kernels* 1.0 y 1.2, se modificó el formato estándar de los archivos binarios de Linux. Los *kernels* más antiguos de Linux sólo comprendían el formato a.out de archivos binarios, que es un formato relativamente simple bastante común en los sistemas UNIX más antiguos. Los sistemas Linux más recientes utilizan el formato más moderno ELF, que ahora está soportado por la mayoría de las implementaciones UNIX actuales. ELF tiene una serie de ventajas sobre a.out, incluyendo una mayor flexibilidad y ampliabilidad. Pueden añadirse nuevas secciones a un binario ELF (por ejemplo, para añadir información de depuración adicional) sin que las rutinas de carga se confundan. Al permitir registrar múltiples rutinas de carga, Linux puede soportar fácilmente los formatos binarios ELF y a.out dentro de un mismo sistema.

En las Secciones 21.6.3 y 21.6.3.2, vamos a concentrarnos exclusivamente en los procesos de carga y de ejecución de los archivos binarios con formato ELF. El procedimiento para cargar archivos binarios a.out es más simple, pero su operación resulta similar.

21.6.3.1 Carga de programas en memoria

En Linux, el cargador binario no carga un archivo binario en memoria física. En lugar de ello, se mapean las páginas del archivo binario sobre regiones de memoria virtual. Sólo cuando el programa trate de acceder a una página determinada, el correspondiente fallo de página provocará la carga de dicha página en memoria física, utilizando el mecanismo de paginación bajo demanda.

Es responsabilidad del cargador binario del *kernel* configurar el mapeo de memoria inicial. Un archivo binario en formato ELF está compuesto por una cabecera, seguida de varias secciones con alineación de página. El cargador ELF funciona leyendo la cabecera y mapeando las secciones del archivo sobre regiones separadas de memoria virtual.

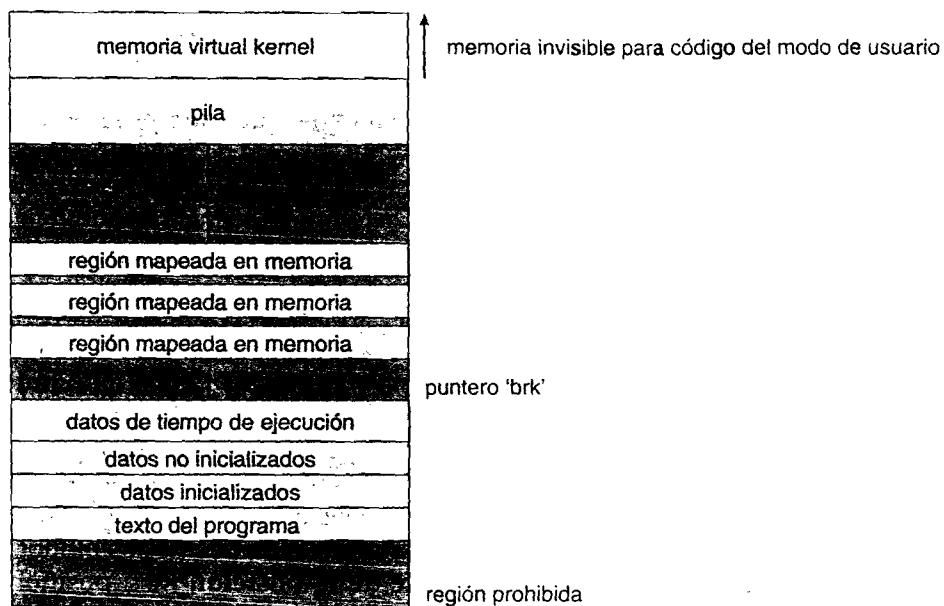


Figura 21.8 Disposición de memoria para los programas ELF.

La Figura 21.8 muestra la disposición típica de regiones de memoria inicializada por el cargador ELF. En una región reservada en uno de los extremos del espacio de direcciones, se sitúa el *kernel* en su propia región privilegiada de memoria virtual a la que no pueden acceder los programas normales en modo usuario. El resto de la memoria virtual está disponible para aplicaciones, que pueden utilizar las funciones de mapeo de memoria del *kernel* para crear regiones que mapeen una parte de un archivo o que estén disponibles para almacenar los datos de la aplicación.

El trabajo del cargador consiste en preparar el mapeo inicial de memoria con el fin de que dé comienzo la ejecución del programa. Entre las regiones que hay que inicializar se encuentran la pila y las regiones de texto y de datos del programa.

La pila se crea en la parte superior de la memoria virtual de modo usuario; la pila crece hacia abajo, en el sentido decreciente de direcciones de memoria. Incluye copias de los argumentos y de las variables de entorno proporcionadas al programa en la llamada al sistema `exec()`. El resto de regiones se crean cerca de la parte inferior de la memoria virtual. Las secciones del archivo binario que contienen texto del programa o datos de sólo lectura se mapean en memoria como región protegida contra escritura. A continuación se mapean los datos inicializados sobre los que se puede escribir, después, se mapean todos los datos no inicializados, como una región privada con ceros bajo demanda.

Directamente a continuación de estas regiones de tamaño fijo hay una región de tamaño variable que los programas pueden ampliar según sea necesario con el fin de almacenar los datos asignados en tiempo de ejecución. Cada proceso tiene un puntero, `brk`, que apunta al límite actual de esta región de datos, y los procesos pueden ampliar o reducir su región `brk` con una única llamada al sistema: `sbrk()`.

Una vez configuradas estas correspondencias de regiones de memoria, el cargador inicializa el registro contador de programa del proceso, cargando en él el punto inicial indicado en la cabecera del ELF, después de lo cual se puede planificar la ejecución del proceso.

21.6.3.2 Montaje estático y dinámico

Una vez que el programa se ha cargado y ha comenzado a ejecutarse, todo contenido necesario del archivo binario habrá sido cargado en el espacio de direcciones virtual del proceso. Sin embargo, la mayoría de los programas también necesitan ejecutar funciones de las bibliotecas del sistema, por lo que será necesario, asimismo, cargar dichas funciones de biblioteca. En el caso más simple, las funciones de biblioteca necesarias estarán integradas directamente dentro del archivo

binario ejecutable del programa. Un programa de este tipo está montado estáticamente con sus bibliotecas, y los ejecutables con montaje estático pueden comenzar a ejecutarse inmediatamente después de que termine su carga.

La principal desventaja del montaje estático es que cada programa que se genere deberá contener copias de las mismas funciones comunes exactas de la biblioteca del sistema. Resulta mucho más eficiente, tanto en términos de memoria física como del uso del espacio de disco, cargar una única vez en memoria las bibliotecas del sistema. La técnica de montaje dinámico permite que se realice esta única operación de carga.

Linux implementa el montaje dinámico en modo usuario a través de una biblioteca especial de montador. Cada programa dinámicamente montado contiene una pequeña función de montaje estático que se invoca en el momento de arrancar el programa. Esta función estática simplemente mapea la biblioteca de montaje en memoria y ejecuta el código que contiene la función. La biblioteca de montaje determina las bibliotecas dinámicas requeridas por el programa y los nombres de las variables y funciones que se necesitan de dichas bibliotecas, leyendo para ello la información contenida en una serie de secciones del archivo binario ELF. A continuación, mapea las bibliotecas en la memoria virtual y resuelve las referencias a los símbolos contenidos en dichas bibliotecas. No importa en qué punto exacto de la memoria se mapeen estas bibliotecas compartidas, porque se compilan en un **código independiente de la posición** (PIC, position-independent code), que puede ejecutarse en cualquier dirección de la memoria.

21.7 Sistemas de archivos

Linux mantiene el modelo estándar de sistemas de archivos de UNIX. En UNIX, un archivo no tiene porque ser un objeto almacenado en disco o descargado a través de la red desde un servidor de archivos remoto. En lugar de ello, los archivos UNIX pueden ser cualquier cosa capaz de procesar la entrada o la salida de un flujo de datos. Los controladores de dispositivos pueden aparecer como archivos y los canales de comunicación interprocesos o las conexiones de red también parecen archivos a ojos del usuario.

El *kernel* de Linux gestiona todos estos tipos de archivos ocultando los detalles de implementación de cada tipo de archivo concreto por debajo de una capa de software, el sistema de archivos virtual (VFS, virtual file system). Aquí, vamos a presentar primero el sistema de archivos virtual y luego nos ocuparemos del sistema de archivos estándar de Linux, ext2fs.

21.7.1 El sistema de archivos virtual

El VFS de Linux está diseñado basándose en principios de orientación a objetos. Tiene dos componentes: un conjunto de definiciones que especifican cuál es el aspecto que están autorizados a tener los objetos del sistema de archivos y una capa de software para manipular los objetos. El sistema VFS define cuatro tipos principales de objetos:

- Un **objeto inodo** representa un archivo individual.
- Un **objeto archivo** representa un archivo abierto.
- Un **objeto superbloque** representa el sistema de archivos completo.
- Un **objeto entrada de directorio (dentry)** representa una entrada de directorio individual.

Para cada uno de estos cuatro tipos de objeto, VFS define un conjunto de operaciones. Cada objeto de uno de estos tipos contiene un puntero a una tabla de funciones. La tabla de funciones enumera las direcciones de las funciones reales que implementan las operaciones definidas para ese objeto. Por ejemplo, una API abreviada para algunas de las operaciones que pueden realizarse sobre el objeto archivo incluiría:

- `int open(. . .)` — Abre un archivo.
- `ssize_t read(. . .)` — Leer de un archivo.



- `ssize_t write(. . .)` —Escribir en un archivo.
- `int mmap(. . .)` —Mapear en memoria un archivo.

La definición completa del objeto archivo se especifica en la estructura `struct file_operations`, que está ubicada en el archivo `/usr/include/linux/fs.h`. Cada implementación del objeto archivo (para un tipo de archivo específico) debe obligatoriamente implementar todas las funciones especificadas en la definición del objeto archivo.

La capa de software VFS puede realizar una operación sobre uno de los objetos del sistema de archivos invocando la función apropiada de la tabla de funciones del objeto, sin tener que conocer de antemano exactamente el tipo de objeto con el que está tratando. El VFS no sabe, ni tampoco le preocupa, si un inodo representa un archivo de red, un archivo de disco, un *socket* de red o un archivo de directorio. La función apropiada para la operación `read()` de dicho archivo siempre se encontrará en el mismo lugar dentro de su tabla de funciones, y la capa de software VFS invocará dicha función sin preocuparse de la forma en que los datos sean leídos realmente.

Los objetos inodo y archivo son los mecanismos utilizados para acceder a los archivos. Un objeto inodo es una estructura de datos que contiene punteros a los bloques de disco donde están los propios contenidos del archivo, y un objeto archivo representa un punto de acceso a los datos de un archivo abierto. Un proceso no puede acceder al contenido de un inodo sin primero obtener el objeto archivo que apunta al inodo. El objeto archivo controla en qué lugar del archivo está leyendo o escribiendo actualmente el proceso, con el fin de controlar las operaciones de E/S de archivo secuenciales. También recuerda si el proceso solicitó permisos de escritura al abrir el archivo y controla la actividad del proceso en caso necesario para realizar lecturas anticipadas adaptativas, cargando los datos del archivo en memoria antes de que el proceso los solicite, con el fin de mejorar la velocidad.

Los objetos archivo pertenecen normalmente a un único proceso, pero los objetos inodo no. Aún cuando un archivo ya no esté siendo utilizado por ningún proceso, su objeto inodo puede continuar siendo almacenado en la caché por el VFS, con el fin de mejorar la velocidad por si acaso, a corto plazo, se utiliza de nuevo el archivo. Todos los datos del archivo almacenados en caché están enlazados en una lista dentro del objeto inodo del archivo. El inodo también mantiene información estándar acerca de cada archivo, como por ejemplo su propietario, su tamaño y el instante en que ha sido modificado más recientemente.

El tratamiento que se da a los archivos de directorio es ligeramente distinto del de los demás archivos. La interfaz de programación de UNIX define una serie de operaciones con los directorios, como por ejemplo la de creación, borrado y renombrado de un archivo dentro de un directorio. Las llamadas al sistema para estas operaciones de directorio no requieren que el usuario abra los correspondientes archivos, a diferencia de lo que ocurre con la lectura o la escritura de datos. Por tanto, el VFS define estas operaciones de directorio en el objeto inodo en lugar de en el objeto archivo.

El objeto superbloque representa un conjunto conectado de archivos que forman un sistema de archivos autocontenido. El *kernel* del sistema operativo mantiene un único objeto superbloque por cada dispositivo de disco montado como sistema de archivos y por cada sistema de archivos de red que esté actualmente conectado. La responsabilidad principal del objeto superbloque consiste en proporcionar acceso a los inodos. El VFS identifica cada inodo mediante una pareja única (sistema-archivo/número inodo) y localiza el inodo que corresponde a un número de inodo concreto pidiendo al objeto superbloque que le devuelva el inodo que tiene dicho número.

Por último, un objeto entrada de directorio representa una entrada de directorio que puede incluir el nombre de un directorio dentro del nombre de ruta de un archivo (como `/usr`) o el propio archivo (como `stdio.h`). Por ejemplo, el archivo `/usr/include/stdio.h` contiene las entradas de directorio (1) `/`, (2) `usr`, (3) `include` y (4) `stdio.h`. Cada uno de estos valores está representado por su propio objeto entrada de directorio.

Como ejemplo del modo en que se utilizan los objetos entrada de directorio, considere el caso en que un proceso quisiera abrir el archivo con el nombre de ruta `/usr/include/stdio.h`, utilizando un editor. Puesto que Linux trata los nombres de directorio como archivos, traducir esta ruta requiere obtener en primer lugar el inodo de la raíz, `/`. El sistema operativo debe entonces

leer este archivo para obtener el inodo del archivo `include` y continuar con este proceso hasta obtener el inodo del archivo `stdio.h`. Puesto que la traducción de los nombres de ruta puede ser una tarea que consume mucho tiempo, Linux mantiene una caché de objetos de entrada de directorio, que consulta durante la traducción de los nombres de ruta. Obtener el inodo a partir de la caché de entradas de directorio es considerablemente más rápido que leer el archivo almacenado en disco.

21.7.2 El sistema de archivos ext2fs de Linux

Por razones históricas, el sistema de archivos estándar en disco estándar utilizado por Linux se denomina **ext2fs**. Originalmente, Linux fue programado con un sistema de archivos compatible con Minix, para facilitar el intercambio de datos con el sistema de desarrollo Minix, pero dicho sistema de archivos estaba enormemente restringido, ya que los nombres de archivo tenían un límite de 14 caracteres y el tamaño máximo del sistema de archivos era de 64 MB. El sistema de archivos Minix fue sustituido por otro nuevo sistema de archivos que se denominó **sistema de archivos extendido** (EFS, extended file system). Un posterior rediseño de este sistema de archivos para mejorar la velocidad y la escalabilidad y añadir unas cuantas características que se echaban de menos condujo al denominado **segundo sistema de archivos extendido** (ext2fs, second extended file system).

El sistema de archivos ext2fs de Linux tiene muchas características en común con el sistema FFS (Fast File System) de BSD (Sección A.7.7). Utiliza un mecanismo similar para localizar los bloques de datos que pertenecen a un archivo específico, almacenando punteros a los bloques de datos en bloques indirectos por todo el sistema de archivos con hasta tres niveles de indirección. Como en FFS, los archivos de directorio están almacenados en disco al igual que los archivos normales, aunque su contenido se interpreta de manera diferente. Cada bloque en un archivo de directorio está compuesto por una lista enlazada de entradas; cada entrada contiene la longitud de la entrada, el nombre de un archivo y el número del inodo al que esa entrada hace referencia.

Las principales diferencias entre ext2fs y FFS radican en sus respectivas políticas de asignación de disco. En FFS, el disco se asigna a los archivos en bloques de 8 KB. Estos bloques se subdividen en fragmentos de 1 KB para el almacenamiento de pequeños archivos o de bloques parcialmente llenos situados al final de los archivos. Por contraste, ext2fs no utiliza fragmentos, sino que realiza todas las tareas de asignación empleando unidades más pequeñas. El tamaño de bloque predeterminado en ext2fs es de 1 KB, aunque también se permiten bloques de 2 KB y 4 KB.

Para obtener unas altas prestaciones, el sistema operativo debe tratar de realizar las operaciones de E/S en grandes fragmentos siempre que sea posible, agrupando las solicitudes de E/S que sean físicamente adyacentes. Este agrupamiento reduce el gasto promedio adicional por cada solicitud debido a los controladores de dispositivos, los discos y el hardware de la controladora de disco. Un tamaño de solicitud de E/S de 1 KB es demasiado pequeño como para poder obtener unas buenas prestaciones, por lo que ext2fs utiliza políticas de asignación diseñadas para situar los bloques de un archivo que sean adyacentes desde el punto de vista lógico en bloques físicamente adyacentes en el disco, con el fin de poder solicitar en una única operación de E/S varios bloques de disco sucesivos.

La política de asignación de ext2fs tiene dos partes. Como en FFS, un sistema de archivos ext2fs está particionado en múltiples **grupos de bloques**. FFS emplea el concepto similar de **grupos de cilindros**, donde cada grupo se corresponde con un único cilindro de un disco físico. Sin embargo, la tecnología moderna de las unidades de disco empaquetan los sectores en el disco con diferentes densidades, y por tanto con diferentes tamaños de cilindro, dependiendo de lo lejos que esté el cabezal del disco con respecto al centro del disco. Por tanto, los grupos de cilindros de tamaño fijo no se corresponden necesariamente con la geometría del disco.

Cuando se asigna un archivo, ext2fs debe seleccionar en primer lugar el grupo de bloques de dicho archivo. Para los bloques de datos, trata de asignar el archivo al grupo de bloques al que se haya asignado el inodo del archivo. Para asignaciones de inodos, selecciona el grupo de bloques en el que resida el directorio padre del archivo (en el caso de archivos que no sean de directorio). Los archivos del directorio no se mantienen juntos, sino que se dispersan entre todos los grupos

de bloques disponibles. Estas políticas están diseñadas no sólo para mantener toda la información relacionada dentro del mismo grupo de bloques, sino también distribuir la carga del disco entre los distintos grupos de bloques del mismo, con el fin de reducir la fragmentación de cada una de las áreas del disco.

Dentro de un grupo de bloques, ext2fs trata de realizar las asignaciones de manera que sean físicamente contiguas, siempre que sea posible, reduciendo la fragmentación si puede. El sistema mantiene un mapa de bits de todos los bloques libres de cada grupo de bloques. Cuando se asignan los primeros bloques para un nuevo archivo, el sistema comienza a buscar un bloque libre a partir del principio del grupo de bloques, cuando se trata de ampliar un archivo, continúa la búsqueda a partir del bloque asignado más recientemente al archivo. Esta búsqueda se realizada en dos etapas. En primer lugar, ext2fs busca un byte libre completo dentro del mapa de bits, si no encuentra ninguno, se conformará con cualquier bit libre. La búsqueda de bytes libres pretende asignar el espacio de disco en secciones de al menos ocho bloques, siempre que sea posible.

Una vez que se ha identificado un bloque libre, se extiende hacia atrás la búsqueda hasta encontrar un bloque asignado. Cuando se encuentra un byte libre en el mapa de bits, esta búsqueda hacia atrás evita que ext2fs deje un hueco entre el bloque asignado más recientemente dentro del byte anterior distinto de cero y el byte igual a cero encontrado anteriormente. Una vez localizado el siguiente bloque que hay que asignar, mediante la búsqueda de un byte o un bit, ext2fs extiende la asignación hacia adelante hasta un máximo de 8 bloques y **preasigna** estos bloques adicionales al archivo. Esta preasignación ayuda a reducir la fragmentación durante las escrituras entrelazadas entre archivos separados y también reduce el coste de CPU de la asignación de disco, al asignar múltiples bloques simultáneamente. Los bloques preasignados se devuelven al mapa de bits de espacio libre en el momento de cerrar el archivo.

La Figura 21.9 ilustra las políticas de asignación. Cada fila representa una secuencia de bits activado/desactivado dentro de un mapa de bits de asignación, indicando los bloques utilizados y libres que hay en el disco. En el primer caso, si podemos encontrar algún bloque libre lo suficientemente cerca del comienzo de la búsqueda, asignaremos esos bloques independientemente de lo fragmentados que puedan estar. La fragmentación se compensa parcialmente por el hecho de que los bloques están próximos entre sí y probablemente pueden ser leídos todos ellos sin efectuar ningún reposicionamiento del cabezal del disco; además, asignarlos todos a un mismo

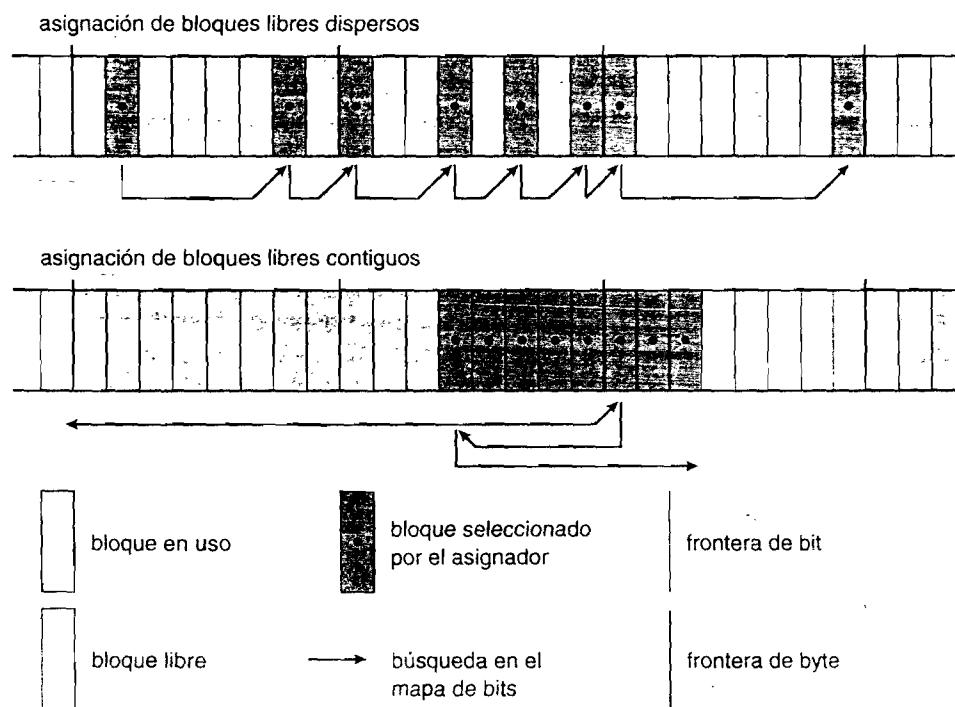


Figura 21.9 Políticas de asignación de bloques de ext2fs.

archivo es mejor, a largo plazo, que asignar bloques aislados a distintos archivos una vez que empiezan a escasear las áreas libres de gran tamaño dentro del disco. En el segundo caso, no se ha encontrado inmediatamente un bloque libre cercano, por lo que se busca hacia adelante hasta encontrar un byte libre completo dentro del mapa de bits. Si asignáramos dicho byte como una unidad, terminaríamos creando un área fragmentada de espacio libre antes de ese byte, de modo que antes de realizar la asignación volvemos hacia atrás para situar la asignación a continuación de la asignación precedente, y asignamos un conjunto de bloques a partir de ese punto con el fin de satisfacer el tamaño predeterminado de asignación que es de ocho bloques.

21.7.3 Diario

Hay disponibles muchos tipos diferentes de sistemas de archivos para los sistemas Linux. Una característica popular de los sistemas de archivos es la de llevar un **diario**, mediante el cual se escriben secuencialmente en un registro las modificaciones efectuadas en el sistema de archivos. Un conjunto de operaciones que realice una tarea específica recibe el nombre de **transacción**. Una vez escrita una transacción en el diario, se la considera confirmada y la llamada al sistema que ha modificado el sistema de archivos (por ejemplo, `write()`), puede devolver el control al proceso de usuario, permitiéndole continuar con su ejecución. Mientras tanto, las entradas del diario relacionadas con la transacción se reaplican en las propias estructuras del sistema de archivos. A medida que se realizan los cambios, se actualiza un puntero para indicar qué acciones se han completado y cuáles están todavía incompletas. Una vez completada una transacción confirmada entera se la elimina del diario. Este diario, que es en la práctica un búfer circular, puede encontrarse en una sección separada del sistema de archivos, o incluso en una sección separada del disco. Resulta más eficiente, aunque también más complejo, utilizar cabezas de lectura-escritura distintas, porque así se reducen tanto la contienda por la utilización de los cabezales como los tiempos de búsqueda.

Si el sistema sufre un fallo catastrófico, en el diario estarán almacenadas cero o más transacciones. Esas transacciones nunca llegaron a ser completamente aplicadas al sistema de archivos, aun cuando hubieran sido confirmadas por el sistema operativo, así que será necesario completarlas. Las transacciones pueden ejecutarse a partir del puntero hasta que se termine con todas las tareas pendientes y las estructuras del sistema de archivos queden en un estado coherente. El único problema aparece cuando se ha abortado una transacción, es decir, cuando hay una transacción que no había sido confirmada antes de que el sistema sufriera el fallo catastrófico. Todos los cambios correspondientes a dichas transacciones que hubieran sido aplicados al sistema de archivos deben ser deshechos, de nuevo para poder preservar la coherencia del sistema de archivos. Este proceso de recuperación es lo único que hay que hacer después de un fallo catastrófico, eliminando así todos los problemas asociados con las comprobaciones de coherencia.

Los sistemas de archivos con registro de diario también suelen ser más rápidos que los que carecen de esta característica, ya que las actualizaciones se realizan mucho más rápidamente cuando se aplican al diario que se conserva en memoria en lugar de aplicarlas a las estructuras de datos almacenadas en el disco. La razón de esta mejora es que resulta mucho más rápida la E/S secuencial que la E/S aleatoria. Las costosas escrituras aleatorias síncronas en el sistema de archivos se transforman en unas escrituras secuenciales síncronas mucho menos costosas en el registro de diario del sistema de archivos. Esos cambios, a su vez, se vuelven a aplicar asíncronamente, mediante escrituras aleatorias, en las estructuras de datos apropiadas. El resultado global es una ganancia significativa en la velocidad de las operaciones realizadas con los metadatos del sistema de archivos, como por ejemplo la creación y el borrado de archivos.

El sistema ext2fs no tiene un registro de diario. Sin embargo, quien sí lo tiene es otro sistema de archivos común disponible para los sistemas Linux, ext3, que está basado en ext2fs.

21.7.4 El sistema de archivos proc de Linux

La flexibilidad del VFS de Linux nos permite implementar un sistema de archivos que no almacene ningún dato de manera persistente, sino que simplemente proporcione una interfaz para otro

tipo de funcionalidad. El **sistema de archivos de proceso** de Linux, denominado sistema de archivos `/proc`, es un ejemplo de un sistema de archivos cuyo contenido no está almacenado en ninguna parte sino que se calcula bajo demanda de acuerdo con las solicitudes de E/S de archivos del usuario.

Los sistemas de archivos `/proc` no son exclusivos de Linux. SVR4 UNIX ya introdujo un sistema de archivos `/proc` como interfaz de gran eficiencia para los mecanismos de soporte de depuración de procesos del *kernel*: cada subdirectorio del sistema de archivos no se corresponde con un directorio de cualquier disco, sino con un proceso activo en el sistema actual. Un listado del sistema de archivos nos da un directorio por cada proceso, siendo el nombre del directorio la representación decimal ASCII del identificador de proceso (PID) único de cada proceso.

Linux implementa ese mismo sistema de archivos `/proc`, pero lo amplía en gran medida, añadiendo diversos directorios adicionales y archivos de texto en el directorio raíz del sistema de archivos. Esas nuevas entradas se corresponden con diversas estadísticas acerca del *kernel* y controladores cargados asociados. El sistema de archivos `/proc` proporciona una manera para que los programas accedan a esta información en forma de archivos de texto, que pueden ser procesados por las potentes herramientas que el entorno de usuario de UNIX estándar proporciona. Por ejemplo, en el pasado, el comando tradicional `ps` de UNIX para obtener un listado del estado de todos los procesos en ejecución se implementaba como un proceso privilegiado que leía el estado del proceso directamente de la memoria virtual del *kernel*. En Linux, este comando se implementa como un programa completamente no privilegiado que simplemente analiza y formatea la información obtenida de `/proc`.

El sistema de archivos `/proc` debe implementar dos cosas: una estructura de directorios y el contenido de los archivos almacenados en ella. Puesto que un sistema de archivos UNIX se define como un conjunto de inodos de archivo y de directorio identificados por sus números de inodo, el sistema de archivos `/proc` debe definir un número de inodo único y persistente para cada directorio y para los archivos asociados. Una vez establecida esa correspondencia puede utilizar ese número de inodo para identificar la operación requerida cada vez que un usuario trata de leer del inodo de un archivo concreto o trata de realizar una búsqueda en el inodo de un directorio concreto. Cuando se leen los datos de uno de estos archivos, el sistema de archivos/`/proc` recopila la información apropiada, la da formato textual y la coloca en el búfer de lectura del proceso solicitante.

La correspondencia entre número de inodo y el tipo de información se basa en dividir el número de inodo en dos campos distintos. En Linux, un identificador PID tiene 16 bits de longitud, pero un número de inodo tiene 32 bits. Los 16 bits más altos del número de inodo se interpretan como un PID y los restantes bits definen el tipo de información que se está solicitando acerca de dicho proceso.

Un identificador PID igual a cero no es válido, por lo que si el campo PID en un número de inodo es igual a cero, se asume que ese inodo contiene información global en lugar de contener información específica de un proceso. Hay diferentes archivos globales en `/proc` para proporcionar información tal como la versión del *kernel*, la memoria libre, las estadísticas de rendimiento y los controladores que se estén actualmente ejecutando.

No todos los números de inodo de este rango están reservados. El *kernel* puede asignar nuevas correspondencias de inodo en `/proc` dinámicamente, manteniendo un mapa de bits de los números de inodo asignados. También mantiene una estructura de datos en árbol con las entradas globales registradas del sistema de archivos `/proc`. Cada entrada contiene el número de inodo del archivo, el nombre del archivo y los permisos de acceso, junto con las funciones especiales utilizadas para generar el contenido del archivo. Los controladores pueden registrar y desregar entradas en este árbol en cualquier momento, y hay una sección especial del árbol (que aparece el directorio `/proc/sys`) reservada para las variables del *kernel*. Los archivos situados bajo este árbol son gestionados mediante un conjunto de descriptores comunes que permiten tanto leer como escribir estas variables, por lo que un administrador del sistema puede optimizar los parámetros del *kernel* simplemente escribiendo los nuevos valores deseados en formato ASCII decimal en el archivo apropiado.

Para permitir un acceso eficiente a estas variables desde las aplicaciones puede accederse al subárbol `/proc/sys` mediante una llamada al sistema especial, `sysctl()`, que lee y escribe las mis-

mas variables en formato binario, en lugar de en formato de texto, sin tener que incurrir en el coste adicional exigido por el sistema de archivos. `sysctl()` no es una funcionalidad adicional; simplemente lee el árbol de entradas dinámicas de `/proc` para decidir a qué variables dinámicas está haciendo referencia la aplicación.

21.8 Entrada y salida

Para el usuario, el sistema de E/S en Linux se parece bastante al de cualquier otro sistema UNIX. Es decir, en la medida de lo posible, todos los controladores de dispositivos aparecen como archivos normales. Un usuario puede abrir un canal de acceso a un dispositivo de la misma forma en que abre cualquier otro archivo, los dispositivos pueden aparecer como objetos dentro del sistema de archivos. El administrador del sistema puede crear archivos especiales dentro de un sistema de archivos que contengan referencias a un controlador de dispositivo específico, y un usuario que abra dicho archivo será capaz de leer y escribir en el dispositivo referenciado. Utilizando el sistema normal de protección de archivos, que determina quién puede acceder a cada archivo, el administrador puede establecer los permisos de acceso para cada dispositivo.

Linux divide todos los dispositivos en tres clases: dispositivos de bloque, dispositivos de caracteres y dispositivos de red. La Figura 21.10 ilustra la estructura global del sistema de controladores de dispositivo.

Los **dispositivos de bloque** incluyen todos los dispositivos que permiten el acceso aleatorio a bloques de datos completamente independientes y de tamaño fijo, incluyendo los discos duros y disquetes, los discos CD-ROM y la memoria flash. Los dispositivos de bloques se utilizan normalmente para almacenar sistemas de archivos, pero también está permitido el acceso directo a un dispositivo de bloque, para que los programas puedan crear y reparar el sistema de archivos que el dispositivo contiene. Las aplicaciones también pueden acceder a los dispositivos de bloque directamente si lo desean; por ejemplo, una aplicación de base de datos puede preferir revisar su propia disposición optimizada de los datos en el disco en lugar de emplear el sistema de archivos de propósito general.

Los **dispositivos de caracteres** incluyen la mayoría de los restantes dispositivos, como por ejemplo ratones y teclados. La diferencia fundamental entre dispositivos de bloque y de caracteres es el acceso aleatorio: a los dispositivos de bloques se puede acceder de forma aleatoria, mientras que a los dispositivos de caracteres sólo se puede acceder en modo serie. Por ejemplo, la operación de situarse en una cierta posición de un archivo puede estar soportada para un DVD, pero no tiene ningún sentido si de lo que hablamos es de un dispositivo de señalización como un ratón.

Los **dispositivos de red** se tratan de manera distinta que los dispositivos de bloques y de caracteres. Los usuarios no pueden transferir datos directamente a los dispositivos de red; en lugar de ello, deben comunicarse indirectamente abriendo una conexión con el subsistema de red del *kernel*. En la Sección 21.10 nos ocuparemos de la interfaz con los dispositivos de red.

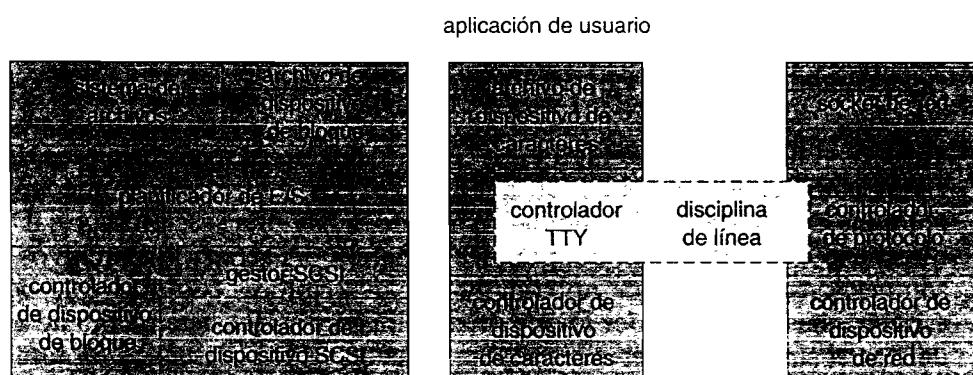


Figura 21.10 Estructura de bloques de los controladores de dispositivos.

21.8.1 Dispositivos de bloque

Los dispositivos de bloque proporcionan la interfaz principal para todos los dispositivos de disco del sistema. Las prestaciones son especialmente importantes para los discos, y el sistema de dispositivos de bloque debe proporcionar la funcionalidad necesaria para garantizar que el acceso a disco sea lo más rápido posible. Esta funcionalidad se consigue a través de la planificación de las operaciones de E/S.

En el contexto de los dispositivos de bloque, un **bloque** representa la unidad con la que el *kernel* realiza las operaciones de E/S. Cuando se lee un bloque en memoria, se almacena en un búfer. El **gestor de solicitudes** es la capa de software que gestiona la lectura y escritura del contenido del búfer desde y en el controlador de dispositivo de bloque.

Para cada controlador de dispositivo de bloque se mantiene una lista separada de solicitudes. Tradicionalmente, estas solicitudes se planificaban de acuerdo con un algoritmo de tipo ascensor unidireccional (C-SCAN) que aprovecha el orden en que se insertan y eliminan las solicitudes en la lista de cada dispositivo. Las listas de solicitudes se mantienen ordenadas según el orden creciente del número de sector inicial. Cuando se acepta una solicitud para su procesamiento por parte de un controlador de dispositivo de bloque, no se le elimina de la lista. Sólo se le elimina después de completada la operación de E/S, en cuyo momento el controlador continúa con el siguiente controlador de la lista, incluso si se han insertado nuevas solicitudes en la lista antes que la solicitud activa. A medida que se realizan nuevas solicitudes de E/S, el gestor de solicitudes trata de combinar las solicitudes existentes en la lista de cada dispositivo.

La planificación de las operaciones de E/S cambió en cierta medida con la versión 2.6 del *kernel*. El problema fundamental con el algoritmo del ascensor es que las operaciones de E/S concentradas en una región específica del disco podían provocar la muerte por inanición de las solicitudes dirigidas a otras regiones del disco. El **planificador de E/S con límite de temporización** utilizado en la versión 2.6 funciona de forma similar al algoritmo del ascensor salvo porque también asocia un plazo con cada solicitud resolviendo así el problema de muerte por inanición. De manera predeterminada, el plazo para solicitudes de lectura es de 0,5 segundos y para las de escritura de 5 segundos. El planificador con límite de temporización mantiene una **cola ordenada** de operaciones de E/S pendientes ordenadas por número de sector. Sin embargo, también mantiene otras dos colas: una **cola de lectura** para las operaciones de lectura y una **cola de escritura** para las operaciones de escritura. Estas dos colas están ordenadas de acuerdo con los plazos. Cada solicitud de E/S se coloca tanto en la cola ordenada como en la cola de lectura o de escritura, según sea apropiado. Normalmente, las operaciones de E/S van realizándose de acuerdo con la cola ordenada. Sin embargo, si vence el plazo de una solicitud en la cola de lectura o de escritura, se planifican las operaciones de E/S de la cola que contiene esa solicitud caducada. Esta política garantiza que ninguna operación de E/S tengan que esperar más tiempo que el plazo marcado.

21.8.2 Dispositivos de caracteres

Un controlador de dispositivo de caracteres puede ser casi cualquier controlador de dispositivo que no ofrezca un acceso aleatorio a bloques fijos de datos. Todos los controladores de dispositivo de caracteres registrados antes el *kernel* de Linux deben también registrar un conjunto de funciones que implementen las operaciones de E/S de archivo que el controlador soporte. El *kernel* no realiza casi ningún preprocessamiento de las solicitudes de lectura o escritura de archivo realizadas en un dispositivo de caracteres, simplemente pasa la solicitud al dispositivo en cuestión y deja que el dispositivo la trate.

La excepción principal a esta regla es el subconjunto especial de controladores de dispositivo de caracteres que implementa dispositivos de terminal. El *kernel* mantiene una interfaz estándar con estos controladores mediante un conjunto de estructuras `tty_struct`. Cada una de estas estructuras proporciona búferes y un mecanismo de control de flujo para los datos procedentes del dispositivo terminal, y entrega dichos datos a una disciplina de línea.

Una **disciplina de línea** es un intérprete de la información procedente de un dispositivo de terminal. La disciplina de línea más común es la disciplina `tty`, que dirige el flujo de datos del

terminal a los flujos estándar de entrada y de salida de los procesos que el usuario esté ejecutando, permitiendo a dichos procesos comunicarse directamente con el terminal de usuario. Esta tarea se complica por el hecho de que puede haber varios de dichos procesos ejecutándose simultáneamente, y la disciplina de línea *tty* es responsable de conectar y desconectar la entrada y salida del terminal a los distintos procesos conectados, a medida que el usuario suspende o despierta dichos procesos.

Hay también otras disciplinas de línea implementadas que no tienen nada que ver con la E/S a un proceso de usuario. Los protocolos de red PPP y SLIP representan formas de codificar una conexión de red a través de un dispositivo terminal tal como una línea serie. Estos protocolos se implementan en Linux como controladores que, en uno de los extremos, aparecen ante el sistema terminal como disciplinas de línea, mientras que en el otro extremo aparecen ante el sistema de red como controladores de dispositivo de red. Una vez que se ha activado una de estas disciplinas de línea en un dispositivo terminal, los datos que aparezcan en dicho dispositivo terminal se encaminarán directamente al controlador de dispositivo de red apropiado.

21.9 Comunicación interprocesos

UNIX proporciona un entorno muy rico para que los procesos se comuniquen entre sí. La comunicación puede ser simplemente una cuestión de hacer que el otro proceso sepa que se ha producido un determinado suceso, o puede implicar la transferencia de datos de un proceso a otro.

21.9.1 Sincronización y señales

El mecanismo estándar de UNIX para informar a un proceso de que ha tenido lugar un cierto suceso es una **señal**. Las señales pueden ser enviadas por un proceso a cualquier otro proceso, existiendo una serie de restricciones que afectan a las señales enviadas a los procesos que sean propiedad de otro usuario. Sin embargo, el número de señales disponibles es limitado, y esas señales no pueden transportar información: lo único que está disponible para el proceso es el hecho de que se ha producido una señal. Las señales no son sólo generadas por los procesos. El *kernel* también genera señales internamente; por ejemplo, puede enviar una señal a un proceso servidor cuando llegan datos a través de un canal de red, a un proceso padre cuando un proceso hijo termina o a un proceso en espera cuando termina un temporizador.

Internamente, el *kernel* de Linux no utiliza señales para comunicarse con los procesos que se están ejecutando en modo *kernel*. Si un proceso en modo *kernel* está esperando a que ocurra un suceso, normalmente no utilizará señales para recibir la notificación de dicho suceso. En lugar de ello, la comunicación con el *kernel* acerca de los sucesos asíncronos entrantes se realiza mediante el uso de estados de planificación y de *wait_queue*. Estos mecanismos permiten a los procesos en modo *kernel* informarse entre sí acerca de los sucesos relevantes, y también permiten que los sucesos sean generados por los controladores de dispositivos o por el sistema de comunicación por red. Cada vez que un proceso quiere esperar a que se complete un determinado suceso, se coloca a sí mismo en una **cola de espera** asociada con dicho suceso y le dice al planificador que ya no es elegible para ejecución. Una vez que el suceso se ha completado, despertará a todos los procesos que se encuentren en la cola de espera. Este procedimiento permite que múltiples procesos esperen a que se produzca un mismo suceso. Por ejemplo, si hay muchos procesos tratando de leer un archivo del disco, entonces todos ellos serán despertados una vez que los datos hayan sido cargados en memoria.

Aunque las señales han sido siempre el mecanismo principal de comunicación de sucesos asíncronos entre procesos, Linux también implementa el mecanismo de semáforos de UNIX System V. Un proceso puede esperar en un semáforo con la misma sencillez con que puede esperar a una señal, pero los semáforos tienen dos ventajas: se pueden compartir múltiples semáforos entre múltiples procesos independientes y las operaciones sobre múltiples semáforos pueden realizarse atómicamente. Internamente, el mecanismo estándar de colas de espera de Linux sincroniza los procesos que se están comunicando mediante semáforos.

21.9.2 Transferencia de datos entre procesos

Linux ofrece varios mecanismos para la transferencia de datos entre procesos. El mecanismo *pipe* (canalización) estándar de UNIX permite que un proceso hijo herede el canal de comunicación de su padre; los datos escritos en un extremo del *pipe* pueden leerse en el otro. En Linux, los *pipes* aparecer como otro tipo más de inodo en el software del sistema de archivos virtual. Cada *pipe* tiene un par de colas de espera para sincronizar al lector y al escritor. UNIX también define un conjunto de funcionalidades de red que permiten enviar flujos de datos a procesos tanto locales como remotos. En la Sección 21.10 se cubre el tema de la comunicación por red.

Hay disponibles otros dos métodos para compartir datos entre procesos. En primer lugar, la memoria compartida ofrece una forma extremadamente rápida de transferir grandes o pequeñas cantidades de datos. Los datos escritos por un proceso en una región de memoria compartida pueden ser leídos inmediatamente por cualquier otro proceso que haya mapeado dicha región en su espacio de direcciones. La desventaja principal de la memoria compartida es que, en sí misma, no ofrece ninguna sincronización. Un proceso ni puede preguntar al sistema operativo si se ha escrito en una sección de la memoria compartida ni tampoco puede suspender su propia ejecución hasta que dicha escritura se produzca. La memoria compartida resulta particularmente potente cuando se la utiliza en conjunción con otro mecanismo de comunicación interprocesos que proporcione ese mecanismo de sincronización que a la memoria compartida le falta.

Una región de memoria compartida en Linux es un objeto persistente que puede ser creado o eliminado por los procesos. Dicho objeto se trata como si fuera un pequeño espacio de direcciones independiente. Los algoritmos de paginación de Linux pueden decidir descargar en disco las páginas de memoria compartida, de la misma forma que pueden descargar las páginas de datos de un proceso. El objeto de memoria compartida actúa como almacén de respaldo para las regiones de memoria compartida, de la misma manera que un archivo puede actuar de respaldo para una región mapeada en memoria. Cuando se mapea un archivo en una región del espacio de direcciones virtual, cualquier fallo de página que se produzca hará que se mapee en la memoria virtual la página apropiada del archivo. De la misma forma, los mapeos de memoria compartida hacen que los fallos de página mapeen las páginas de un objeto de memoria compartida persistente. También al igual que con los archivos, los objetos de memoria compartida recuerdan su contenido incluso aunque que no haya ningún proceso que los esté mapeando actualmente en memoria virtual.

21.10 Estructura de red

La comunicación por red es una de las áreas clave de funcionalidad en Linux. Linux no sólo soporta los protocolos Internet estándar utilizados en la mayoría de las comunicaciones UNIX-UNIX, sino que también implementa diversos protocolos nativos de otros sistemas operativos distintos de UNIX. En particular, puesto que Linux fue originalmente implementado principalmente en máquinas de tipo PC, en lugar de grandes estaciones de trabajo o sistemas de tipo servidor, admite muchos de los protocolos usados normalmente en redes de tipo PC, como AppleTalk e IPX.

Internamente, la comunicación por red en el *kernel* de Linux se implementa mediante tres capas de software:

1. La interfaz *socket*
2. Controladores de protocolo
3. Controladores de dispositivos de red.

Las aplicaciones de usuario realizan todas las solicitudes de comunicación por red a través de la interfaz *socket*. Esta interfaz está diseñada para asemejarse a la capa *socket* de BSD 4.3, de modo que cualquier programa diseñado para hacer uso de los *sockets* de Berkeley se podrá ejecutar en Linux sin efectuar ningún cambio en el código fuente. Esta interfaz se describe en la Sección A.9.1. La interfaz *socket* BSD es suficientemente general como para representar las direcciones de red para un amplio rango de protocolos de comunicación por red. Esta misma interfaz se emplea en Linux



no sólo para acceder a aquellos protocolos implementados en los sistemas BSD estándar, sino también a todos los protocolos soportados por el sistema.

La siguiente capa de software es la pila de protocolos, que es similar en organización a la utilizada en BSD. Cada vez que llegan nuevos datos por red a esta capa, bien procedente de un *socket* de aplicación o a través un controlador de dispositivo de red, se espera que los datos hayan sido etiquetados con un identificador que especifique el protocolo de red que contienen. Los protocolos pueden comunicarse entre sí, si así lo desean; por ejemplo, dentro del conjunto de protocolos Internet, hay una serie de protocolos separados que gestionan el encaminamiento, los informes de errores y la retransmisión fiable de los datos perdidos.

La capa de protocolos puede reescribir paquetes, crear nuevos paquetes, dividir o re-ensamblar paquetes en fragmentos o simplemente descartar los datos entrantes. En último extremo, una vez que ha finalizado de procesar un conjunto de paquetes los pasa a la siguiente capa, es decir, a la interfaz *socket* si los datos están destinados a una conexión local o al controlador de dispositivo si el paquete tiene que retransmitirse remotamente. La capa de protocolos decide el *socket* o dispositivo al cual hay que enviar el paquete.

Todas las comunicaciones entre las capas de la pila de comunicación por red se realizan pasando estructuras `skbuff`. Un estructura `skbuff` contiene un conjunto de punteros a un área continua de memoria, que representa un búfer dentro del cual pueden construirse los paquetes de red. Los datos válidos de una estructura `skbuff` no necesitan comenzar al principio del búfer, y tampoco necesitan continuar hasta el final. El código de comunicación por red puede añadir o eliminar datos de cualquiera de los dos extremos del paquete, siempre y cuando el resultado continúe cabiendo dentro de la estructura `skbuff`. Esta capacidad resulta especialmente importante en los microprocesadores modernos, en los que las mejoras en la velocidad de la CPU han sido mucho más significativas que las de la velocidad de la memoria principal. La arquitectura `skbuff` proporciona una gran flexibilidad a la hora de manipular las cabeceras de paquetes y las sumas de comprobación, evitando operaciones innecesarias de copia de los datos.

El conjunto de protocolos más importante dentro del sistema de comunicación por red de Linux es el conjunto de protocolos TCP/IP. Este conjunto comprende diversos protocolos separados. El protocolo IP implementa el encaminamiento entre distintos *hosts* situados en cualquier lugar de la red. Por encima del protocolo de encaminamiento están construidos los protocolos UDP, TCP e ICMP. El protocolo UDP transporta datagramas individuales arbitrarios entre *hosts*. El protocolo TCP implementa conexiones fiables entre hosts con una entrega ordenada y garantizada de los paquetes y con un mecanismo de retransmisión automática de los datos perdidos. El protocolo ICMP se utiliza para comunicar diversos mensajes de error y de estado entre un *host* y otro.

Los paquetes (estructuras `skbuff`) que llegan a la capa de protocolos de la pila red deben estar etiquetados con un identificador interno que indique el protocolo para el cual es relevante dicho paquete. Diferentes controladores de dispositivos de red codifican el tipo de protocolo de distintas maneras en el medio de comunicación; por tanto, el protocolo para los datos entrantes debe identificarse en el controlador de dispositivo. El controlador de dispositivos utiliza una tabla *hash* de identificadores conocidos de protocolos de red para localizar el protocolo apropiado y pasa el paquete a dicho protocolo. Pueden añadirse nuevos protocolos a la tabla *hash* en forma de módulos cargables por el *kernel*.

Los paquetes IP entrantes se entregan al controlador IP. El trabajo de esta capa de software consiste en realizar el encaminamiento. Después de decidir a dónde está destinado el paquete, esta capa re-envía el paquete al controlador de protocolo interno que resulte apropiado, con el fin de suministrarlo localmente, o vuelve a injectarlo en la cola de un controlador de dispositivo de red seleccionado, con el fin de re-enviarlo a otro *host*. Esta decisión de encaminamiento se toma utilizando dos tablas: la denominada base de información de re-envío persistente (FIB, forwarding information base) y una caché en la que se almacenan las decisiones de encaminamiento recientemente tomadas. La base FIB almacena información de configuración del encaminamiento y puede especificar rutas basándose en una dirección de destino específica o en un carácter comodín que represente múltiples destinos. La base FIB está organizada como un conjunto de tablas *hash* indexadas según la dirección de destino; las tablas que representan las rutas más específicas son las que se

exploran primero. Las búsquedas que tengan éxito en esta tabla se añaden a la tabla de caché de rutas, donde se almacenan las rutas únicamente según su destino específico, en la caché no se almacena ningún carácter comodín, de modo que las búsquedas en la caché puedan llevarse a cabo rápidamente. Las entradas de la caché de rutas caducan después de un período fijo durante el cual no se haya producido ningún acierto de caché para esa entrada.

En diversas etapas del proceso, el software IP pasa los paquetes a una sección de código separada con el fin de realizar la **gestión de cortafuegos**, es decir, el filtrado selectivo de los paquetes de acuerdo con una serie de criterios arbitrarios, normalmente por razones de seguridad. El gestor de cortafuegos mantiene una serie de **cadenas de cortafuegos** y permite comparar cada `skbuff` con cualquiera de las cadenas. Las distintas cadenas se reservan para diferentes propósitos: una se usa para los paquetes re-enviados, otra para los paquetes que constituyen una entrada para este *host* y otras para los datos generados en este *host*. Cada cadena se almacena en forma de una lista ordenada de reglas, en la que cada regla especifica una función de entre una serie de posibles funciones que regulan la decisión de cortafuegos; cada regla se complementa con una serie de datos arbitrarios que son con los que hay que ver si el paquete se corresponde.

El controlador IP realiza otras dos funciones: el desensamblado y el re-ensamblado de paquetes de gran tamaño. Si un paquete saliente es demasiado grande como para poder ponerlo en la cola de un dispositivo, simplemente se divide en una serie de **fragmentos** más pequeños, todos los cuales se ponen en la cola del controlador. En el host receptor, esos fragmentos deberán ser reensamblados. El controlador IP mantiene un objeto `ipfrag` por cada fragmento que esté a la espera de ser re-ensamblado y un objeto `ipq` por cada datagrama que esté siendo ensamblado. Los fragmentos entrantes se comparan con cada `ipq` conocido. Si se encuentra una correspondencia, se le añade el fragmento; en caso contrario, se crea un nuevo `ipq`. Una vez que ha llegado el fragmento final de un `ipq`, se construye un estructura `skbuff` completamente nueva para almacenar el nuevo paquete y este paquete se pasa al controlador IP.

Los paquetes identificados por la capa IP como destinados a este *host* se pasa a alguno de los otros controladores de protocolo. Los protocolos UDP y TCP comparten un mismo medio de asociar los paquetes con los *sockets* de origen y de destino: cada pareja de *sockets* conectados queda identificada únicamente mediante sus direcciones de origen y de destino y mediante los números de puerto de origen y de destino. Las listas de *sockets* están enlazadas en tablas *hash* que utilizan como clave estos cuatro valores de dirección-puerto, con el fin de poder buscar fácilmente el *socket* correspondiente a los paquetes entrantes. El protocolo TCP tiene que tratar con conexiones no fiables, así que mantiene listas ordenadas de paquetes salientes no confirmados, con el fin de retransmitirlos después de un cierto período de temporización, y de paquetes entrantes que han entrado desordenadamente con el fin de entregárselos al *socket* una vez que lleguen los datos que faltan.

21.11 Seguridad

El modelo de seguridad de Linux está bastante relacionado con los mecanismos típicos de seguridad de UNIX. Los problemas relativos a la seguridad pueden clasificarse en dos grupos:

1. **Autenticación.** Asegurarse de que nadie pueda acceder al sistema sin demostrar primero que tiene los correspondientes derechos de entrada.
2. **Control de acceso.** Proporcionar un mecanismo para controlar si un usuario tiene derecho de acceso a un cierto objeto e impedir el acceso a los objetos según sea necesario.

21.11.1 Autenticación

La autenticación en UNIX se realizaba tradicionalmente utilizando un archivo de contraseñas públicamente legible. La contraseña de un usuario se combina con un valor **aleatorizador** y el resultado se codifica mediante una función de transformación unidireccional y se almacena en el archivo de contraseñas. El uso de la función unidireccional implica que la contraseña original no puede deducirse a partir del archivo de contraseñas salvo por un método de prueba y error.

Cuando un usuario presenta una contraseña al sistema, la contraseña se recombina con el aleatorizador almacenado en el archivo de contraseñas y se pasa a través de la misma función unidireccional. Si el resultado se corresponde con el contenido del archivo de contraseñas, entonces la contraseña se acepta.

Históricamente, las implementaciones UNIX de este mecanismo presentaban diversos problemas. Las contraseñas estaban limitadas a menudo a ocho caracteres y el número de posibles valores aleatorizadores era tan bajo que un atacante podía fácilmente combinar un diccionario de contraseñas comúnmente utilizadas con todos los valores aleatorizadores y tener una buena probabilidad de encontrar una correspondencia con una o más de las contraseñas del archivo de contraseñas, obteniendo como resultado un acceso no autorizado a las cuentas que hubieran quedado comprometidas. Se han introducido diversas extensiones del mecanismo de contraseñas que mantienen en secreto la contraseña cifrada en un archivo que no es públicamente legible. Otras variantes permiten utilizar contraseñas de mayor tamaño o emplean métodos más seguros para codificar la contraseña. También se han introducido otros mecanismos de autenticación que limitan el número de veces que un usuario está autorizado a conectarse al sistema o que distribuyen información de autenticación a todos los sistemas relacionados de una red.

Los distribuidores de UNIX han desarrollado un nuevo mecanismo de seguridad para resolver los problemas de autenticación. El sistema de **módulos de autenticación conectables** (PAM, pluggable authentication modules) está basado en una biblioteca compartida que puede ser utilizada por cualquier componente del sistema que necesite autenticar a los usuarios. En Linux hay disponible una implementación de este sistema. PAM permite cargar módulos de autenticación bajo demanda, según se especifique en un archivo de configuración global del sistema. Si se añade un nuevo mecanismo de autenticación posteriormente, se puede insertar en el archivo de configuración, y todos los componentes del sistema podrán aprovecharse de él de forma inmediata. Los módulos PAM pueden especificar métodos de autenticación, restricciones que afecten cuentas, funciones de configuración de sesión y funciones de modificación de contraseña (de modo que, cuando los usuarios cambien sus contraseñas, puedan actualizarse a la vez todos los mecanismos de autenticación necesarios).

21.11.2 Control de acceso

El control de acceso en los sistemas UNIX, incluyendo Linux, se realizan utilizando identificadores numéricos únicos. Un identificador de usuario (uid) identifica a un único usuario a o único conjunto de derechos de acceso. Un identificador de grupo (gid) es un identificador adicional que puede emplearse para especificar aquellos derechos que pertenecen a más de un usuario.

El mecanismo de control de acceso se aplica a diversos objetos del sistema. Cada archivo disponible en el sistema está protegido por el mecanismo estándar de control de acceso; además, otros objetos compartidos, como las secciones de memoria compartida y los semáforos, emplean el mismo sistema de acceso.

Todo objeto de un sistema a UNIX sometido al control de acceso de usuario y de grupo tiene asociado un único uid y un único gid. Los procesos de usuario también tienen un único uid pero pueden tener más de un gid. Si el uid de un proceso se corresponde con el uid de un objeto, entonces el proceso tiene **derechos de usuario** o **derechos de propietario** sobre dicho objeto. Si los valores uid no se corresponden, pero algunos de los valores gid del proceso sí se corresponden con el gid del objeto, entonces se conceden al proceso **derechos de grupo**; en caso contrario, el proceso tiene **derechos del resto del mundo** sobre el objeto.

Linux realiza el control de acceso asignando a los objetos una **máscara de protección**, que especifica los modos de acceso (lectura, escritura o ejecución) que hay que conceder a los procesos que tengan acceso de propietario, de grupo o resto del mundo. Por tanto, el propietario de un objeto puede tener un acceso completo de lectura, escritura y ejecución sobre un archivo, mientras que otros usuarios de un cierto grupo podrían tener acceso de lectura, pero no de escritura y todos los demás podrían no tener ningún acceso en absoluto.

La única excepción es el uid **root** privilegiado. Un proceso con este uid especial tiene automáticamente acceso a todos los objetos del sistema, puenteando los controles de acceso normales.

Tales procesos también tienen permiso para realizar operaciones privilegiadas, como leer cualquier sección de la memoria física o abrir *sockets* de red reservados. Este mecanismo permite al *kernel* impedir que los usuarios normales accedan a estos recursos. La mayoría de los principales recursos internos del *kernel* son propiedad implícitamente del uid root.

Linux implementa el mecanismo *setuid* estándar de UNIX descrito en el Sección A.3.2. Este mecanismo permite a un programa ejecutarse con privilegios diferentes de los del usuario que está ejecutando el programa. Por ejemplo, el programa *lpr* (que coloca un trabajo en una cola de impresión) tiene acceso a las colas de impresión del sistema incluso si el usuario que está ejecutando el programa no dispone de dicho acceso. La implementación UNIX de *setuid* distingue entre un uid *real* y el uid *efectivo* de un proceso. El uid real es el del usuario que está ejecutando el programa el uid efectivo es el del propietario del archivo.

En Linux, este mecanismo está ampliado de dos formas distintas. En primer lugar, Linux implementa el mecanismo *saved user-id* de la especificación POSIX, que permite a un proceso renunciar y readquirir su uid efectivo repetidamente. Por razones de seguridad, un programa puede querer realizar la mayoría de sus operaciones en un modo seguro, renunciando a los privilegios concedidos por el estado *setuid*, pero al mismo tiempo puede querer realizar determinadas operaciones seleccionadas con todos los privilegios. Las implementaciones estándar de UNIX consiguen esta capacidad únicamente intercambiando los valores uid real y efectivo; el uid efectivo anterior se recuerda, pero el uid real del programa no siempre se corresponde con el uid del usuario que ejecuta el programa. Los uid guardados permiten a un proceso utilizar como uid efectivo su uid real y luego volver al valor anterior de uid efectivo, sin tener que modificar el uid real en ningún momento.

La segunda mejora proporcionada por Linux es la adición de una característica de los procesos que concede únicamente un subconjunto de los derechos del uid efectivo. Las propiedades *fsuid* y *fsgid* de los procesos se utilizan cuando se conceden derechos de acceso a los archivos. La propiedad correspondiente se configura cada vez que se configura el uid o gid efectivo. Sin embargo, los valores *fsuid* y *fsgid* pueden configurarse independientemente de los identificadores efectivos, lo que permite a un proceso acceder a archivos por cuenta de otro usuario sin adoptar la identidad de ese otro usuario en ninguna forma. Específicamente, los procesos de servidor pueden utilizar este mecanismo para servir archivos a un cierto usuario sin que el proceso pueda ser matado o suspendido por ese usuario.

Finalmente, Linux proporciona un mecanismo flexible para pasar los derechos de un programa a otro, un mecanismo que ha llegado a ser común en las versiones modernas de UNIX. Cuando se ha configurado un *socket* de red local entre dos procesos del sistema, cualquiera de esos procesos puede enviar al otro un descriptor de archivo que se corresponda con uno de sus archivos abiertos; el otro proceso recibirá un descriptor de archivo duplicado para ese mismo archivo. Este mecanismo permite a un cliente pasar selectivamente el acceso a un archivo a otro proceso servidor sin conceder a dicho proceso ningún otro privilegio. Por ejemplo, ya no es necesario que un servidor de impresión pueda leer todos los archivos de un usuario que envíe un nuevo trabajo de impresión; el cliente de impresión puede simplemente pasar al servidor los descriptores de los archivos que haya que imprimir denegando al servidor el acceso a los restantes archivos del usuario.

21.12 Resumen

Linux es un moderno sistema operativo gratuito basado en estándares UNIX. Ha sido diseñado para ejecutarse de manera eficiente y fiable sobre hardware común de tipo PC; también se puede ejecutar en otras plataformas. Proporciona una interfaz de programación y una interfaz de usuario compatible con los sistemas estándar UNIX y permite ejecutar un gran número de aplicaciones UNIX, incluyendo un creciente número de aplicaciones comercialmente soportadas.

Linux no ha evolucionado dentro de una burbuja. Un sistema Linux completo incluye muchos componentes que fueron desarrollados de Linux. El *kernel* básico del sistema operativo Linux es enteramente original, pero permite ejecutar buena parte del software UNIX gratuito existente, lo

que da como resultado un sistema operativo completo compatible con UNIX y completamente libre de código propietario.

El *kernel* de Linux está implementado como un *kernel* monolítico tradicional por razones de rendimiento, pero es suficientemente modular en su diseño como para permitir cargar y descargar dinámicamente la mayoría de los controladores en tiempo de ejecución.

Linux es un sistema multiusuario, que proporciona protección entre los procesos y que puede ejecutar múltiples procesos de acuerdo con un planificador de tiempo compartido. Los procesos recién creados pueden compartir partes seleccionadas de su entorno de ejecución con sus procesos padre, lo que permite la programación multihebra. La programación interprocesos está soportada tanto por mecanismos de tipo System V (colas de mensajes, semáforos y memoria compartida) como por la interfaz *socket* de BSD. A través de la interfaz *socket* puede accederse simultáneamente a múltiples protocolos de red.

Para el usuario, el sistema de archivos aparece como un árbol de directorios jerárquico que cumple con la semántica UNIX. Internamente, Linux utiliza una capa de abstracción para gestionar múltiples sistemas de archivos diferentes. Se permite el uso de sistemas de archivos orientados a dispositivo, sistemas de archivos en red y sistemas de archivos virtuales. Los sistemas de archivos orientados a dispositivo acceden al almacenamiento en disco a través de una caché de páginas que está unificada con el sistema de memoria virtual.

El sistema de gestión de memoria utiliza compartición de páginas y mecanismos de copia durante la escritura para minimizar la duplicación de los datos compartidos por los diferentes procesos. Las páginas se cargan bajo demanda la primera vez que se hace referencia a las mismas y se descargan en el almacén de respaldo de acuerdo con un algoritmo LFU cuando es necesario reclamar la memoria física.

Ejercicios

- 21.1 ¿Cuáles son las ventajas y desventajas de escribir un sistema operativo en un lenguaje de alto nivel, como C?
- 21.2 ¿En qué circunstancias resulta más apropiada la secuencia de llamadas al sistema `fork()` `exec()`? ¿Cuándo es preferible utilizar `vfork()`?
- 21.3 ¿Qué tipo de *socket* debería utilizarse para implementar un programa de transferencia de archivos entre computadoras? ¿Qué tipo debería utilizarse para un programa que comprobara periódicamente si otra computadora está encendida en la red? Razone su respuesta.
- 21.4 Linux se ejecuta sobre diversas plataformas hardware. ¿Qué pasos deben dar los desarrolladores de Linux para garantizar que el sistema sea portable a diferentes procesadores y arquitecturas de gestión de memoria, y para minimizar la cantidad de código de *kernel* específico de la arquitectura?
- 21.5 ¿Cuáles son las ventajas y desventajas de hacer que sólo parte de los símbolos definidos dentro de un *kernel* sean accesibles a un módulo cargable del *kernel*?
- 21.6 ¿Cuáles son los objetivos principales del mecanismo de resolución de conflictos usado por el *kernel* de Linux para cargar los módulos del *kernel*?
- 21.7 Explique el modo en que se utiliza la operación `clone()` de Linux para permitir el uso tanto de procesos como de hebras.
- 21.8 ¿Cómo clasificaría las hebras de Linux: cómo hebras de nivel de usuario o como hebras de nivel del *kernel*? Razone su respuesta con los argumentos apropiados.
- 21.9 ¿Qué costes adicionales tiene la creación y planificación de un proceso, si se compara con el coste de una hebra clonada?
- 21.10 El planificador de Linux implementa una planificación en tiempo real *no estricta*. ¿Qué características necesarias para ciertas tareas de programación en tiempo real son las que faltan? ¿Cómo podrían añadirse al *kernel*?

- 21.11 ¿En qué circunstancias solicitaría un proceso de usuario una operación que provocará la asignación de una región de memoria con ceros bajo demanda?
- 21.12 ¿Qué escenarios harían que se mapeara una página de memoria sobre el espacio de direcciones de un programa de usuario con el atributo de copia durante la escritura activado?
- 21.13 En Linux, las bibliotecas compartidas realizan muchas operaciones cruciales para el sistema operativo. ¿Cuál es la ventaja de mantener esta funcionalidad fuera del *kernel*? ¿Existe alguna desventaja? Razone su respuesta.
- 21.14 La estructura de directorios de un sistema operativo Linux puede comprender archivos correspondientes a diferentes sistemas de archivos, incluyendo el sistema de archivos /proc de Linux. ¿Cuáles son las implicaciones de tener que soportar diferentes tipos de sistemas de archivos en la estructura del *kernel* de Linux?
- 21.15 ¿De qué forma difiere la funcionalidad setuid de Linux de la funcionalidad setuid del UNIX estándar?
- 21.16 El código fuente de Linux está ampliamente disponible de forma gratuita a través de Internet o de diversos suministradores en CD-ROM. ¿Podría citar tres consecuencias que esta disponibilidad tiene para la seguridad del sistema Linux?

Notas bibliográficas

El sistema Linux es un producto de Internet; en consecuencia, buena parte de la documentación disponible sobre Linux está accesible de una u otra forma a través Internet. Los siguientes sitios principales contienen referencias a la mayor parte de la información útil disponible:

- Linux Cross-Reference Pages en <http://lxr.linux.no/> mantiene listados actuales del *kernel* de Linux, que pueden consultarse a través de la Web y están completamente referenciados entre sí.
- Linux-HQ en <http://www.linuxhq.com/> alberga una gran cantidad de información relacionada con los *kernels* 2.x de Linux 2.x. Este sitio también incluye vínculos con las páginas principales de la mayoría de las distribuciones Linux, así como archivos de las principales listas de correo.
- Linux Documentation Project en <http://sunsite.unc.edu/linux/> cita muchos libros sobre Linux que están disponibles en formato fuente como parte del proyector de documentación de Linux. El proyecto también alberga las guías tutoriales *How-To*, que contienen una serie de consejos y sugerencias relativas a diversos aspectos de este sistema operativo.
- La guía *Kernel Hackers'Guide* es una guía basada en Internet de los detalles internos generales del *kernel*. Este sitio en constante expansión se encuentra en <http://www.redhat.com:8080/HyperNews/get/khg.html>.
- El sitio web Kernel Newbies (<http://www.kernelnewbies.org/>) proporciona diversos recursos introductorios al *kernel* de Linux para los principiantes.

También hay disponibles muchas listas de correo dedicadas a Linux. Las más importantes son las mantenidas por un gestor de listas de correo con el que se puede conectar en la dirección de correo electrónico majordomo@vger.rutgers.edu. Envíe un mensaje de correo electrónico a esta dirección incluyendo una única línea con el texto "help" en el cuerpo del mensaje para obtener información sobre cómo acceder al servidor de listas y cómo suscribirse a alguna de ellas.

Por último, el propio sistema Linux puede obtenerse a través de Internet. Pueden obtenerse distribuciones Linux completas en los sitios web de las correspondientes empresas, y la comunidad Linux también mantiene archivos de los componentes actuales del sistema en diversos lugares de Internet. Los más importantes son:

- <ftp://tsx-11.mit.edu/pub/linux/>

- <ftp://sunsite.unc.edu/pub/Linux/>
- <ftp://linux.kernel.org/pub/linux/>

Además de investigar los recursos disponibles en Internet, puede aprender acerca de los detalles internos del *kernel* de Linux en Bovet y Cesati [2002] y Love [2004].

Windows XP

El sistema operativo Microsoft Windows XP es un sistema operativo multitarea apropiativo de 32/64-bits para los microprocesadores AMD K6/K7, Intel IA32\IA64 y posteriores. Sucesor de Windows NT y Windows 2000, Windows XP también trata de reemplazar al sistemas operativo Windows 95/98. Los objetivos clave del sistema operativo son la seguridad, la fiabilidad, la facilidad de uso, la compatibilidad con aplicaciones Windows y POSIX, las altas prestaciones, ampliabilidad, la portabilidad y el soporte internacional. En este capítulo, vamos a analizar los objetivos clave de Windows XP, la arquitectura en niveles que los hace tan fácil de utilizar, el sistema de archivos, las funciones de conexión por red y la interfaz de programación.

OBJETIVOS DEL CAPÍTULO

- Explorar los principios en los que se basa el diseño de Windows XP y los componentes específicos que forman el sistema.
- Comprender cómo Windows XP puede ejecutar programas diseñados para otros sistemas operativos.
- Proporcionar unas explicaciones detalladas del sistema de archivos de Windows XP.
- Ilustrar los protocolos de red soportados en Windows XP.
- Analizar la interfaz disponible para los programadores de sistemas y aplicaciones.

22.1 Historia

A mediados de 1980, Microsoft e IBM cooperaron en el desarrollo del sistema operativo OS/2, que fue escrito en lenguaje ensamblador para sistemas Intel 80286 monoprocesador. En 1988, Microsoft decidió comenzar de nuevo y desarrollar un sistema operativo portable basado en una “nueva tecnología” (o NT, new technology) que soportara las interfaz de programación de aplicaciones (API, application-programming interface) tanto de OS/2 como de POSIX. En octubre de 1988, Dave Cutler, el arquitecto del sistema operativo VAX/VMS de DEC fue contratado por Microsoft y se le encargo construir ese nuevo sistema operativo.

Originalmente, el equipo de desarrollo pretendía usar para NT la API OS/2 como entorno nativo, pero durante el desarrollo NT fue modificado para utilizar la API Windows de 32 bits (o API Win32), como resultado de la popularidad de Windows 3.0. Las primera versiones de NT fueron Windows NT 3.1 y Windows NT 3.1 Advanced Server (en aquella época, la versión actual del sistema operativo Windows de 16 bits era la 3.1.) La versión 4.0 de Windows NT adoptó la interfaz de usuario de Windows 95 e incorporó software de servidor web y explorador web Internet. Además, las rutinas de interfaz de usuario y todo el código gráfico se incluyeron en el *kernel* para mejorar las prestaciones, aunque eso tuvo el efecto colateral de reducir la fiabilidad del sistema. Aunque las versiones anteriores de NT habían sido portadas a otras arquitecturas de microproce-

sador, la versión Windows 2000, lanzada en febrero de 2000, dejó de dar soporte para otros procesadores distintos de Intel (y compatibles) debido a consideraciones de mercado. Windows 2000 incorporó varios cambios significativos respecto a Windows NT. Se añadió Active Directory (un servicio de directorio basado en X.500), un mejor soporte de red y de dispositivos portátiles, soporte para dispositivos plug-and-play, un sistema distribuido de archivos y soporte para más procesadores y más memoria.

En octubre de 2001, se presentó Windows XP como actualización del sistema operativo de sobremesa Windows 2000 y como sustituto de Windows 95/98. En 2002, estuvieron disponibles las versiones de servidor de Windows XP (denominadas Windows .Net Server). Windows XP actualiza la interfaz gráfica de usuario (GUI, graphical user interface) con un diseño visual que aprovecha los más recientes avances hardware e incorpora muchas nuevas características que incrementan la **facilidad de uso**. Se han añadido numerosas funciones para reparar automáticamente los problemas de las aplicaciones y del propio sistema operativo. Windows XP proporciona unas posibilidades más simples de configuración de la red y de los dispositivos (incluyendo comunicación inalámbrica de configuración cero, mensajería instantánea, flujos multimedia y vídeo/fotografía digital). También se han aumentado enormemente las prestaciones tanto para las máquinas de sobremesa como para los grandes sistemas multiprocesador, y la fiabilidad y la seguridad son aún mejores que las de Windows 2000.

Windows XP utiliza una arquitectura cliente-servidor (como Mach) para poder implementar múltiples personalidades del sistema operativo, como por ejemplo la API Win32 y POSIX, con una serie de procesos de nivel de usuario, denominados subsistemas. La arquitectura de subsistemas permite realizar mejoras en una de las personalidades del sistema operativo sin afectar a la compatibilidad con las aplicaciones de las restantes personalidades.

Windows XP es un sistema operativo multiusuario, que permite el acceso simultáneo a través de servicios distribuidos o mediante múltiples instancias de la interfaz gráfica de usuario, a través del servidor de terminales de Windows. Las versiones de servidor de Windows XP permiten el establecimiento de varias sesiones simultáneas del servidor de terminales, desde sistemas Windows de sobremesa. Las versiones de sobremesa del servidor de terminales multiplexan el teclado, el ratón y el monitor entre las distintas sesiones de terminal virtual, para cada uno de los usuarios que hayan iniciado la sesión. Esta característica, denominada conmutación rápida de usuario, permite a los usuarios desalojarse unos a otros en la consola de un PC sin tener que cerrar y volver a abrir una sesión en el sistema.

Windows XP es la primera versión de Windows en incorporar soporte de 64 bits. El sistema de archivos nativo NT (NTFS) y muchas de las API Win32 han utilizado siempre enteros de 64 bits en todos los lugares apropiados, por lo que la extensión principal a 64 bits en Windows XP se refiere principalmente al soporte de direcciones de gran tamaño.

Existen dos versiones de sobremesa de Windows XP. Windows XP Professional, es la opción preferida para sistemas de escritorio utilizados por usuarios avanzados, tanto en entornos domésticos como de oficina. Para los usuarios domésticos que están efectuando la migración de Windows 95/98, Windows XP Personal proporciona la fiabilidad y la facilidad de uso de Windows XP, aunque carece de las características más avanzadas para poder trabajar con Active Directory o para ejecutar aplicaciones POSIX.

Los miembros de la familia Windows .Net Server utilizan los mismos componentes principales que las versiones de sobremesa, pero añaden una serie de características necesarias para aplicaciones tales como granjas de servidores web, servidores de archivos y de impresión, sistemas en *cluster* y máquinas para grandes centros de datos. Esas máquinas para grandes centros de datos pueden tener hasta 64 GB de memoria y 32 procesadores en los sistemas IA32 y hasta 128 GB y 64 procesadores en los sistemas IA64.

22.2 Principios de diseño

Los objetivos de diseño de Microsoft para Windows XP incluyen la seguridad, la fiabilidad, la compatibilidad con aplicaciones Windows y POSIX, las altas prestaciones, la ampliabilidad, la portabilidad y el soporte internacional.

22.2.1 Seguridad

Los objetivos de **seguridad** de Windows XP requerían algo más que una simple adherencia a los estándares que permitieron que Windows NT 4.0 recibiera una clasificación de seguridad C-2 por parte del gobierno de EE. UU. (lo que significa un nivel moderado de protección frente al software defectuoso de los ataques maliciosos). Se combinó una revisión y pruebas profundas del código con la utilización de herramientas automáticas sofisticadas de análisis para identificar e investigar los defectos potenciales que pudieran representar vulnerabilidades de seguridad del sistema.

22.2.2 Fiabilidad

Windows 2000 era el sistema operativo más fiable y estable que Microsoft había lanzando hasta la fecha. Buena parte de esta fiabilidad se debía a la madurez del código fuente, a las profundas pruebas de carga del sistema y a las herramientas de detección automática de numerosos errores graves en los controladores. Los requisitos de **fiabilidad** para Windows XP eran todavía más estrictos. Microsoft utilizó profundas revisiones de código, tanto manuales como automáticas, para identificar más de 63.000 líneas en los archivos fuente que pudieran contener potenciales problemas no detectados por las pruebas, después de lo cual revisó cada una de las áreas para verificar que el código era correcto.

Windows XP amplía la verificación de los controladores con el fin de detectar errores más sutiles, mejora las facilidades para detectar los errores de programación en el código de nivel de usuario y somete a los dispositivos, controladores y aplicaciones de terceras fuentes, a un riguroso proceso de certificación. Además, Windows XP añade nuevas facilidades para monitorizar el estado del PC, incluyendo la descarga de parches para los problemas, antes de que estos lleguen a ser experimentados por los usuarios. La fiabilidad percibida de Windows XP también se mejoró haciendo que la interfaz gráfica de usuario fuera más fácil de utilizar; para ello se empleó un mejor diseño visual, unos menús más simples y unas mejoras mensurables en la facilidad con la que los usuarios pueden descubrir cómo realizar determinadas tareas comunes.

22.2.3 Compatibilidad con aplicaciones Windows y POSIX

Windows XP no sólo es una actualización de Windows 2000, sino también sustituto para Windows 95/98. Windows 2000 se centraba principalmente en la compatibilidad para aplicaciones de ofimática. Los requisitos para Windows XP incluían una mucho mayor compatibilidad con las aplicaciones de consumo que se ejecutan sobre Windows 95/98. La **compatibilidad de aplicaciones** es difícil de conseguir, porque cada aplicación comprueba para ver si está instalada una versión concreta de Windows, puede tener una cierta dependencia con respecto a las particularidades de implementación de las API, puede exhibir errores latentes de aplicación que estuvieran ocultos en el sistema anterior, etc.

Windows XP introduce un nivel de compatibilidad que se sitúa entre las aplicaciones y las API Win32. Este nivel hace que Windows XP parezca (casi completamente) compatible con las versiones anteriores de Windows, cuyos errores es capaz de emular. Windows XP, al igual que las versiones de NT anteriores, mantiene el soporte para ejecutar muchas aplicaciones de 16 bits utilizando un nivel de conversión que traduce las llamadas a la API de 16 bits a sus llamadas de 32 bits equivalentes. De forma similar, la versión de 64 bits de Windows XP proporciona un nivel de conversión que traduce las llamadas a la API de 32bits en llamadas de 64 bits nativas. El soporte POSIX en Windows XP ha sido notablemente mejorado. Ahora hay disponible un nuevo subsistema POSIX denominado Interix. La mayoría del software compatible con UNIX actualmente disponible se puede compilar y ejecutar en Interix sin ninguna modificación.

22.2.4 Altas prestaciones

Windows XP está diseñado para proporcionar unas **altas prestaciones** en los sistemas de sobremesa (que están fundamentalmente restringidos por el rendimiento de E/S), en los sistemas ser-

vidores (en los que la CPU es a menudo el cuello de botella) y en los grandes entornos multihebra y multiprocesador (donde la gestión de los mecanismos de bloqueo y de las líneas de caché resultan claves para la escalabilidad). Las altas prestaciones han sido un objetivo de importancia creciente en Windows XP. Windows 2000 con SQL 2000 sobre un hardware Compaq consiguió unas marcas TPC-C extraordinarias en el momento de su lanzamiento.

Para satisfacer los requisitos de prestaciones, NT utiliza diversas técnicas como la E/S asíncrona, protocolos optimizados para las redes (por ejemplo, mecanismos de bloqueo optimista de datos distribuidos y consolidación de las solicitudes en lotes), gráficos basados en el *kernel* y mecanismos sofisticados de almacenamiento en caché de los datos del sistema de archivos. Los algoritmos de gestión de memoria y de sincronización están diseñados teniendo en cuenta las consideraciones de rendimiento relativas a las líneas de caché y a los multiprocesadores.

Windows XP ha conseguido mejorar aún más las prestaciones, reduciendo la longitud de la ruta de código en las funciones críticas, utilizando mejores algoritmos y estructuras de datos separadas para cada procesador, utilizando mecanismos de coloreados de memoria para máquinas NUMA (non-uniform memory access, acceso de memoria no uniforme), e implementando protocolos de bloqueo más escalables, como por ejemplo los cerrojos de bucle sin fin en cola. Los nuevos protocolos de bloqueo ayudan a reducir los ciclos del bus del sistema e incluyen colas y listas de procesos bloqueados libres, operaciones atómicas de lectura-modificación-escritura (como por ejemplo las operaciones de incremento interbloqueado) y otras técnicas de bloqueo avanzado.

Los subsistemas que constituyen Windows XP se comunican entre sí de forma eficiente mediante una funcionalidad de llamadas a procedimientos locales (LPC, local procedure call), que proporcionan un sistema de mensajería de altas prestaciones. Salvo cuando se están ejecutando en el despachador del *kernel*, las hebras de los subsistemas de Windows XP pueden ser desalojadas por otras hebras de mayor prioridad. De este modo, el sistema puede responder rápidamente a los sucesos externos. Además, Windows XP está diseñado para el multiprocesamiento simétrico; las máquinas multiprocesador pueden ejecutar varias hebras al mismo tiempo.

22.2.5 Ampliabilidad

El concepto de **ampliabilidad** hace referencia a la capacidad de un sistema operativo para mantenerse actualizado con respecto a los avances en tecnología informática. Para poder facilitar los cambios a lo largo del tiempo, los desarrolladores implementaron Windows XP utilizando una arquitectura de niveles. El programa ejecutivo (Executive) de Windows XP se ejecuta en modo *kernel* o en modo protegido y proporciona los servicios básicos del sistema. Por encima del ejecutivo, hay varios subsistemas de servidor que trabajan en modo usuario. Entre unos subsistemas y otros se encuentra el **subsistema de entorno** que emula diferentes sistemas operativos. De este modo, puede ejecutarse sobre Windows XP, en el entorno apropiado, programas escritos para MS-DOS, Microsoft Windows y POSIX (en la Sección 22.4 podrá encontrar más información sobre los subsistemas de entorno). Debido a la estructura modular, pueden añadirse subsistemas de entornos sin afectar al ejecutivo. Además, Windows XP utiliza controladores cargables en el sistema de E/S, por lo que pueden añadirse nuevos sistemas de archivos, nuevos tipos de dispositivos de E/S y nuevos mecanismos de interconexión por red mientras el sistema continúa funcionando. Windows XP emplea un modelo cliente-servidor como el sistema operativo Mach, y permite el uso de mecanismos de procesamiento distribuido basados en llamadas a procedimientos remotos (RPC) como define la organización Open Software Foundation.

22.2.6 Portabilidad

Un sistema operativo es **portable** si se le puede desplazar de una arquitectura hardware a otra con un número de cambios relativamente pequeño. Windows XP está diseñado para ser portable. Al igual que sucede con el sistema operativo UNIX, la mayor parte del sistema está escrito en C y C++. Casi todo el código dependiente del procesador está aislado en una biblioteca de montaje dinámico (DLL, dynamic link library) denominada **nivel de abstracción del hardware** (HAL, hardware-abstraction layer). Una DLL es un archivo que se mapea dentro del espacio de direcciones de un proceso, de modo que todas las funciones de la DLL parezcan ser parte de ese proceso. Los nive-

les superiores del *kernel* de Windows XP dependen de las interfaces HAL en vez de depender del hardware subyacente, lo que aumenta la portabilidad de Windows XP. El nivel HAL manipula el hardware directamente, aislando al resto de Windows XP de las diferencias hardware que puedan existir entre las diversas plataformas sobre las que se ejecuta.

Aunque Windows 2000 sólo se vendía, por razones de mercado, para plataformas Intel compatibles con IA32, también fue probado sobre plataformas Alpha de DEC e IA32 hasta poco antes del lanzamiento con el fin de garantizar la portabilidad. Windows XP se ejecuta sobre los procesadores compatibles con las especificaciones IA32 e IA64. Microsoft es consciente de la importancia del desarrollo y las pruebas multiplataforma, ya que, desde el punto de vista práctico, el no mantener la portabilidad equivale a renunciar a un mercado que ciertamente existe.

22.2.7 Soporte internacional

Windows XP también está diseñado para su utilización en entornos **internacionales y multiculturales**. Proporciona soporte para diferentes configuraciones nacionales mediante la API de **soporte de idioma nacional** (NLS, national-language-support). La API NLS proporciona rutinas especializadas para dar formato a las fechas, las horas y las unidades monetarias de acuerdo con las diversas costumbres nacionales. Las comparaciones de cadenas de caracteres están especializadas, para tener en cuenta las diferencias entre conjuntos de caracteres. UNICODE es el conjunto nativo de caracteres de Windows XP. Este sistema operativo permite también el uso del conjunto de caracteres ANSI, convirtiéndoles a caracteres UNICODE antes de manipularlos (conversión de 8-bits a 16-bits). Las cadenas de caracteres utilizadas por el sistema se mantienen en archivos de recursos que pueden ser sustituidos fácilmente para localizar el sistema en diferentes idiomas. Pueden usarse concurrentemente múltiples configuraciones locales, lo cual es importante para personas y empresas que operen en entornos multilingües.

22.3 Componentes del sistema

La arquitectura de Windows XP es un sistema de módulos organizado en niveles, como se muestra en la Figura 22.1. Los niveles principales son el nivel HAL, el *kernel* y el subsistema ejecutivo, todos los cuales se ejecutan en modo protegido, junto con una colección de subsistemas y servicios que se ejecutan en modo usuario. Los subsistemas de modo usuario pueden clasificarse en dos categorías: los subsistemas de entorno, que emulan diferentes sistemas operativos y **subsistemas de protección**, que proporcionan funciones de seguridad. Una de las principales ventajas de este tipo de arquitectura es que las interacciones entre los distintos módulos resultan simples. En el resto de esta sección se describen estos niveles y subsistemas.

22.3.1 Nivel de abstracción hardware

El nivel HAL es la capa de software que oculta las diferencias de tipo hardware a ojos de los niveles superiores del sistema operativo, con el fin de hacer que Windows XP sea portable. El nivel HAL exporta una interfaz de máquina virtual que es utilizada por el despachador del *kernel*, por el ejecutivo y por los controladores de dispositivos. Una de las ventajas de esta técnica es que sólo se necesita una versión de cada controlador de dispositivo; esa versión se podrá ejecutar sobre todas las plataformas hardware sin necesitar portar el código del controlador. El nivel HAL también proporciona soporte para el multiprocesamiento simétrico. Los controladores de dispositivo mapean los dispositivos y acceden a ellos directamente, pero los detalles administrativos de mapeo de memoria, de configuración de los buses de E/S, de configuración del mecanismos DMA y de tratamiento de las características de cada placa madre son proporcionados por la interfaces HAL.

22.3.2 Kernel

El *kernel* de Windows XP proporciona la base para el ejecutivo y para los distintos subsistemas. El *kernel* permanece cargado en memoria y su ejecución nunca puede ser desalojada. Tiene cuatro

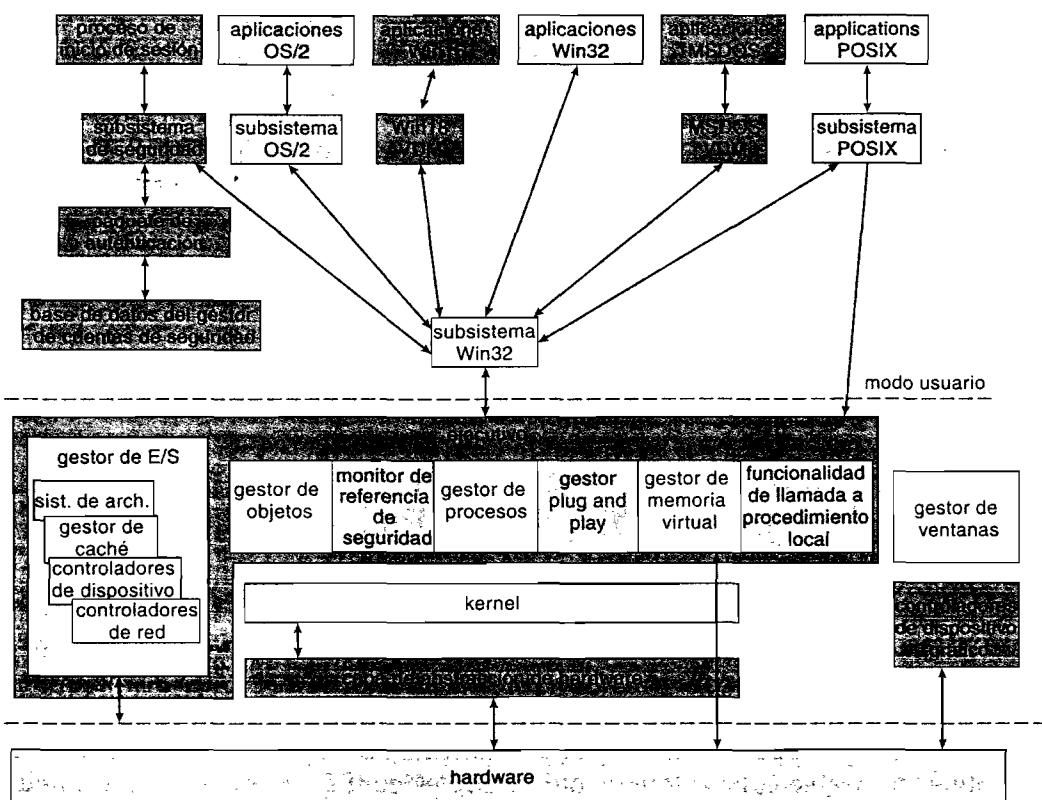


Figura 22.1 Diagrama de bloques de Windows XP.

responsabilidades principales: planificación de hebras, tratamiento de interrupciones y excepciones, sincronización del procesador a bajo nivel y recuperación después de un fallo de alimentación.

El *kernel* está orientado a objetos. Un *tipo de objeto* en Windows 2000 es un tipo de dato definido por el sistema que tiene un conjunto de atributos (valores de datos) y un conjunto de métodos (por ejemplo, funciones u operaciones). Un *objeto* es una instancia de un tipo de objeto. El *kernel* lleva a cabo su trabajo utilizando un conjunto de objetos del *kernel* cuyos atributos almacenan los datos del *kernel* y cuyos métodos realizan las actividades del *kernel*.

22.3.2.1 Despachador del kernel

El despachador del *kernel* proporciona la base para el ejecutivo y para los distintos subsistemas. La mayor parte del despachador nunca es descargada de memoria por el mecanismo de paginación y su ejecución nunca puede ser desalojada. Sus principales responsabilidades son la planificación de hebras, la implementación de primitivas de sincronización, la gestión de temporizadores, las interrupciones software (llamadas a procedimientos asíncronas y diferidas) y el despacho de excepciones.

22.3.2.2 Hebras y planificación

Al igual que otros muchos sistemas operativos modernos, Windows XP utiliza procesos y hebras para estructurar el código ejecutable. Los procesos tienen un espacio de direcciones de memoria virtual y una serie de informaciones que se utilizan para inicializar cada hebra, como por ejemplo una prioridad base y una afinidad para uno o más procesadores. Cada proceso tiene una o más hebras, cada una de las cuales es una unidad ejecutable que el *kernel* se encarga de despachar. Cada hebra tiene su propio estado de planificación, incluyendo su prioridad real, la afinidad de procesador y la información de utilización de la CPU.

Los seis posibles estados de las hebras son: preparada, lista, en ejecución, en espera, en transición y terminada. El estado **preparada** indica que la hebra está esperando para ser ejecutada. En cada decisión de planificación, la hebra preparada de mayor prioridad se pasa al estado de **lista**, lo que quiere decir que será la siguiente hebra que se ejecute. En un sistema multiprocesador, cada proceso mantiene una hebra en el estado de lista. Una hebra estará en ejecución cuando esté ejecutándose sobre un procesador. Esa hebra se ejecutará hasta que sea desalojada por una hebra de mayor prioridad, hasta que termine, hasta que finalice su tiempo de ejecución asignado (**cuanto**) o hasta que se bloquee en espera de un objeto del despachador, como por ejemplo un suceso que señale la terminación de una operación de E/S. Una hebra se encontrará en el estado de **espera** cuando esté esperando a que se señale un objeto del despachador. Una nueva hebra estará en el estado de **transición** cuando esté esperando a los recursos necesarios para su ejecución. Una hebra entrará en el estado de **terminada** cuando haya finalizado su ejecución.

El despachador utiliza un esquema de prioridades de 32 niveles para determinar el orden de ejecución de las hebras. Las prioridades se dividen en dos clases: clase variable y clase de tiempo real. La clase variable contiene hebras cuyas prioridades van de 0 a 15, mientras que la clase de tiempo real contiene hebras cuyas prioridades están comprendidas entre 16 y 31. El despachador utiliza una cola diferente para cada prioridad de planificación y recorre el conjunto de colas desde la prioridad más alta a la más baja hasta que encuentra una hebra que esté lista para ejecutarse. Si una hebra tiene una afinidad de procesador concreta pero el procesador no está disponible, el despachador se la salta y continúa buscando una hebra preparada que sí que quiera ejecutarse en el procesador disponible. Si no encuentra una hebra preparada, el despachador ejecuta una hebra especial denominada hebra de inactividad.

Cuando se agota el cuento de tiempo de una hebra, la interrupción del reloj efectúa una llamada a procedimiento diferida (DPC, deferred procedure call) al procesador en la que se señala el final del cuento, con el fin de volver a planificar el procesador. Si la hebra desalojada pertenece a la clase de prioridad variable, su prioridad se reduce. La prioridad no puede nunca reducirse por debajo de la prioridad base. Este mecanismo de reducción de prioridad de la hebra tiende a limitar el consumo de CPU por parte de las hebras limitadas por un procesador. Cuando una hebra de prioridad variable finaliza una operación de espera, el despachador aumenta su prioridad. El grado de aumento dependerá del dispositivo por el que la hebra estuviera esperando, por ejemplo, una hebra que estuviera esperando una operación de E/S de teclado obtendría un gran aumento de prioridad, mientras que una hebra que estuviera esperando por una operación de disco obtendría sólo un aumento de prioridad moderado. Esta estrategia tiende a proporcionar buenos tiempos de respuesta a las hebras interactivas que utilicen un ratón y un sistema de ventanas; también permite que las hebras limitadas por E/S mantengan ocupados a los dispositivos de E/S, al mismo tiempo que se permite que las hebras limitadas por procesador utilicen en segundo plano los ciclos de CPU que queden libres. Esta estrategia se emplea en diversos sistemas operativos de tiempo compartido, incluyendo UNIX. Además, la hebra asociada con la ventana GUI activa del usuario recibe un aumento de prioridad, con el fin de optimizar su tiempo de respuesta.

Las decisiones de planificación se toman cuando una hebra entra en el estado de preparada o de espera, cuando una hebra termina o cuando una aplicación modifica la prioridad o la afinidad de procesador de una cierta hebra. Si una hebra de tiempo real con mayor prioridad pasa a estar preparada mientras se está ejecutando una hebra de menor prioridad, esa hebra de menor prioridad será desalojada. Este mecanismo apropiativo proporciona a las hebras de tiempo real un acceso preferencial a la CPU en los momentos en que esas hebras necesiten dicho acceso. Sin embargo, Windows XP no es un sistema operativo de tiempo real estricto, porque no garantiza que una hebra de tiempo real pueda comenzar a ejecutarse dentro de un límite de tiempo concreto.

22.3.2.3 Implementación de las primitivas de sincronización

Las estructuras de datos clave del sistema operativo se gestionan como objetos, utilizando funcionalidades comunes para la asignación, para el recuento de las referencias y para la seguridad. Los **objetos del despachador** controlan el despacho y la sincronización dentro del sistema. Ejemplos de estos objetos son los sucesos, los mutantes, los mutex, los semáforos, los procesos, las hebras y

los temporizadores. El **objeto suceso** se utiliza para registrar un determinado suceso y para sincronizarlo con alguna acción. Los sucesos de notificación se encargan de señalizar a todas las hebras que estén en espera, mientras que los sucesos de sincronización señalan a una única hebra en espera. Los objetos **mutantes** proporcionan exclusión mutua en modo *kernel* o en modo usuario, teniendo esa exclusión mutua aparejada la noción de propiedad. Los **mútex**, disponibles sólo en modo *kernel*, proporcionan exclusión mutua libre de interbloqueos. Un **objeto semáforo** actúa como un contador o puerta para controlar el número de hebras que acceden a un recurso. El **objeto hebra** es la entidad que se encarga de despachar el despachador del *kernel* y está asociada con un **objeto proceso**, que encapsula un espacio de direcciones virtual. Los **objetos temporizadores** se emplean para controlar el tiempo y para señalizar los fines de temporización cuando las operaciones son muy largas y es necesario interrumpirlas, o cuando se necesita planificar una actividad periódica.

A muchos de los objetos del despachador se puede acceder en modo usuario, mediante una operación de apertura que devuelve un descriptor. El código en modo usuario sondea y/o espera a recibir los descriptores, para sincronizarse con otras hebras, así como con el sistema operativo (véase la Sección 22.7.1).

22.3.2.4 Interrupciones software: llamadas a procedimientos asíncronas y diferidas

El despachador implementa dos tipos de interrupciones software: llamadas a procedimientos asíncronas y llamadas a procedimientos diferidas. Las llamadas a procedimientos asíncronas (APC, Asynchronous procedure call) interrumpen a una hebra en ejecución e invocan un procedimiento. Las APC se usan para iniciar la ejecución de una nueva hebra, para terminar procesos y para notificar que se ha completado una operación de E/S asíncrona. Las APC se ponen en cola para ser atendidas por hebras específicas y permiten al sistema ejecutar tanto código del sistema como del usuario dentro del contexto de un proceso.

Las llamadas a procedimientos diferidas (DPC, deferred procedure call) se utilizan para posponer el procesamiento de interrupciones. Después de tratar todos los procesos bloqueados por la interrupción de un dispositivo, la rutina de servicio de interrupción (ISR, interrupt service routine) planifica el procesamiento restante poniendo en cola una DPC. El despachador planifica las interrupciones software con una prioridad menor que las interrupciones de dispositivo, por lo que las llamadas DPC no bloquean otras rutinas ISR. Además de diferir el procesamiento de las interrupciones de dispositivos, el despachador utiliza las llamadas DPC para procesar los sucesos de caducidad de los temporizadores y para desalojar la ejecución de una hebra al final de su cuento de planificación.

La ejecución de las llamadas DPC impide que las hebras sean planificadas para su ejecución en el procesador actual y también evita que las APC señalicen la terminación de las operaciones de E/S. Esto se hace para que las rutinas DPC no tarden demasiado tiempo en completarse. Como alternativa, el despachador mantiene un conjunto de hebras de trabajo; las rutinas ISR y las llamadas DPC ponen en cola una serie de tareas para que las ejecuten las hebras de trabajo. Las rutinas DPC están restringidas, de modo que no puedan provocar fallos de página, invocar servicios del sistema o realizar cualquier otra acción que pudiera resultar en un intento de bloquear la ejecución de un objeto del despachador. A diferencia de las APC, las rutinas DPC no realizan ningún tipo de suposición acerca del contexto del proceso que está ejecutando el procesador.

22.3.2.5 Excepciones e interrupciones

El despachador del *kernel* también proporciona rutinas para tratar las excepciones e interrupciones generadas por el hardware o por el software. Windows XP define diversas excepciones independientes de la arquitectura, entre las que se incluyen:

- Violación de acceso a memoria.
- Desbordamiento de entero.
- Desbordamiento o subdesbordamiento de coma flotante.

- División entera por cero.
- División en coma flotante por cero.
- Instrucción ilegal.
- Alineación incorrecta de los datos.
- Instrucción privilegiada.
- Error de lectura de página.
- Violación de acceso.
- Cuota de archivo de paginación excedida.
- Punto de ruptura del depurador.
- Ejecución paso a paso del depurador.

Las rutinas de tratamiento se encargan de gestionar excepciones simples. La gestión más sofisticada de las excepciones la lleva a cabo el despachador de excepciones del *kernel*. El **despachador de excepciones** crea un registro de excepción que contiene la razón de la excepción y localiza una rutina de tratamiento de excepciones apropiada.

Cuando se produce una excepción en modo *kernel*, el despachador de excepciones simplemente invoca una rutina para localizar la rutina de tratamiento correspondiente. Si no se encuentra una rutina de tratamiento apropiada se produce un error del sistema fatal y se presenta al usuario la “pantalla azul” que indica que el sistema ha fallado.

El tratamiento de las excepciones es más complejo para los procesos en modo usuario, porque los subsistemas de entorno (como por ejemplo el sistema POSIX) activan un puerto de depurador y un puerto de excepciones para cada proceso que crean. Si hay registrado un puerto de depurador, la rutina de tratamiento de la excepción envía la excepción a dicho puerto. Si no se encuentra el puerto del depurador o ese puerto no permite tratar la excepción, el despachador trata de localizar una rutina de tratamiento de excepciones apropiada. Si no encuentra dicha rutina, se invoca de nuevo el depurador para capturar el error con vistas a su depuración. Si no hay ningún depurador ejecutándose, se envía el mensaje al puerto de excepciones del proceso, para proporcionar al subsistema de entorno una oportunidad de traducir dicha excepción. Por ejemplo, el entorno POSIX traduce los mensajes de excepción Windows XP a señales POSIX antes de enviarlas a la hebra que ha provocado la excepción. Finalmente, si ninguna de las otras alternativas funciona, el *kernel* se limita a terminar simplemente el proceso que contiene la hebra que provocó la excepción.

El despachador de interrupciones del *kernel* se encarga de gestionar las interrupciones, llamando a una rutina de servicio de interrupción (ISR) suministrada por un controlador de dispositivo, o a una rutina de tratamiento del *kernel*. La interrupción se representa mediante un objeto interrupción que contiene toda la información necesaria para gestionarla. Utilizar un objeto interrupción hace que resulte sencillo asociar las rutinas de servicio con su interrupción correspondiente sin tener que acceder directamente al hardware de interrupciones.

Las diferentes arquitecturas de procesador, como por ejemplo Intel y Alpha de DEC tienen diferentes tipos y números de interrupciones. Para facilitar la portabilidad, el despachador de interrupciones hace corresponder las interrupciones hardware con un conjunto estándar de interrupciones predefinidas. Cada interrupción tiene una prioridad asociada y el servicio de las interrupciones se realiza según el orden de prioridad. En Windows XP hay 32 niveles de solicitud de interrupción (IRQL, interrupt request level). Ocho de ellos están reservados para ser utilizados por el *kernel*; los 24 niveles restantes representan interrupciones hardware que tienen lugar a través del nivel HAL (aunque la mayor parte de los sistemas IA32 sólo utilizan 16). En la Figura 22.2 se definen las interrupciones de Windows XP.

El *kernel* utiliza una **tabla de despacho de interrupciones** para asociar cada nivel de interrupción con una rutina de servicio. En una computadora multiprocesador, Windows XP mantiene una tabla de despacho de interrupciones separada para cada procesador, y el IRQL de cada procesador

niveles de interrupción	tipos de interrupciones
31	error de dispositivo, comprobación de errores
30	fallo de memoria
29	notificación de la ejecución de software, actualización de la memoria caché de los procesadores
28	reloj de tiempo real, llamada de tiempo
27	permiso
3	interrupciones temporales, devoluciones de procedimientos de interruptor, despacho y llamada a procedimientos de interruptor
2	llamada a procedimiento de interruptor
0	pasiva

Figura 22.2 Niveles de solicitud de interrupción de Windows XP.

puede configurarse de manera independiente con el fin de enmascarar las interrupciones. Todas las interrupciones con un nivel igual o inferior al IRQL de un procesador estarán bloqueadas hasta que el IRQL sea reducido por una hebra del nivel del *kernel* o por una rutina ISR en el momento de volver del procesamiento de una interrupción. Windows XP aprovecha esta propiedad y utiliza las interrupciones software para realizar llamadas APC y DPC, para realizar funciones del sistema tales como la sincronización de hebras con la terminación de las operaciones E/S, para comenzar las tareas de despacho de hebras y para gestionar los temporizadores.

22.3.3 Executive

El sistema ejecutivo (Executive) de Windows XP proporciona un conjunto de servicios utilizado por todos los subsistemas de entorno. Los servicios se agrupan del siguiente modo: gestor de objetos, gestor de memoria virtual, gestor de procesos, funcionalidad de llamada a procedimientos locales, gestor de E/S, gestor de caché, monitor de referencia de seguridad, gestores plug-and-play y de seguridad, registro y arranque.

22.3.3.1 Gestor de objetos

Para gestionar las entidades de modo *kernel*, Windows XP utiliza un conjunto genérico de interfaces que son manipuladas por los programas en modo usuario. Windows XP denomina a estas entidades *objetos*, y el componente del programa ejecutivo que los manipula es el **gestor de objetos**. Cada proceso tiene una tabla de objetos que contiene entradas utilizadas para controlar los objetos usados por el proceso. El código en modo usuario accede a estos objetos empleando un valor opaco denominado *descriptor* que es devuelto por muchas de las funciones de diversas API (interfaces de programación de aplicaciones). Los descriptores de objetos también pueden crearse duplicando un descriptor existente del mismo proceso o de otro proceso distinto. Como ejemplo de objetos podríamos citar los semáforos, los mutex, los sucesos, los procesos y las hebras; todos ellos son *objetos del despachador*. Las hebras pueden bloquearse en el despachador del *kernel*, en espera de que alguno de estos objetos sea señalizado. Las API de los procesos, de las hebras y de la memoria virtual utilizan los descriptores de los procesos y hebras para identificar el proceso o la hebra sobre la que hay que actuar. Otros ejemplos de objetos serían los archivos, las secciones, los puertos y diversos objetos de E/S internos. Los objetos archivo se emplean para mantener el estado de apertura de los archivos y dispositivos. Las secciones se utilizan para mapear archivos. Los archivos abiertos se describen en términos de los objetos archivo. Los puntos terminales de comunicación local se implementan como objetos puerto.

El gestor de objetos mantiene el espacio interno de nombres de Windows XP. A diferencia de UNIX, que asocia el espacio de nombres del sistema con el sistema de archivos, Windows XP utiliza un espacio de nombres abstracto y conecta los sistemas de archivos como dispositivos.

El gestor de objetos proporciona interfaces para definir tanto tipos de objetos como instancias de objetos, traduciendo los nombres a objetos, manteniendo el espacio de nombres abstracto (mediante directorios internos y enlaces simbólicos) y encargándose de gestionar la creación y eliminación de objetos. Los objetos se suelen gestionar usando contadores de referencias dentro del código en modo protegido y descriptores dentro del código en modo usuario. Sin embargo, algunos componentes de modo *kernel* utilizan las mismas API que el código en modo usuario, así que emplean descriptores para manipular los objetos. Si es necesario que un descriptor persista una vez que ha terminado el proceso actual, se puede marcar como descriptor del *kernel* y almacenar en la tabla de objetos del proceso del sistema. El espacio de nombres abstracto no persiste entre un arranque del sistema y otro, sino que se crea a partir de la información de configuración almacenada en el registro del sistema, a partir de los datos recopilados durante la fase de descubrimiento de dispositivos plug-and-play y a partir de las operaciones de creación de objetos por parte de los componentes del sistema.

El sistema ejecutivo de Windows XP permite proporcionar un **nombre** a cada objeto. Un proceso puede crear un proceso nominado, mientras que un segundo proceso puede abrir un descriptor de dicho objeto y compartirlo con el primer proceso. Los procesos también pueden compartir objetos duplicando los descriptores, en cuyo caso no es necesario proporcionar un nombre a los objetos.

Los nombres pueden ser permanentes o temporales. Un nombre permanente representa una entidad, como por ejemplo una unidad de disco, que sigue existiendo incluso aunque no haya ningún proceso accediendo a la misma. Por el contrario, un nombre temporal sólo existe mientras un proceso posea un descriptor de dicho objeto.

Los nombres de los objetos están estructurados como los nombres de ruta de los archivos en MS-DOS y UNIX. Los directorios del espacio de nombres se representan mediante **objetos directorio** que contienen los nombres de todos los objetos del directorio. El espacio de nombres de objetos se amplía mediante la adición de objetos dispositivo que representan volúmenes que contienen sistemas de archivos.

Los objetos se manipulan mediante un conjunto de funciones virtuales, para las que se proporcionan implementaciones para cada tipo de objeto: `create()`, `open()`, `close()`, `delete()`, `query_name()`, `parse()` y `security()`. Conviene hacer algunas aclaraciones sobre las tres últimas funciones:

- `query_name()` se invoca cuando una hebra tiene una referencia a un objeto, pero quiere conocer el nombre del objeto.
- `parse()` es utilizada por el gestor de objetos para buscar un objeto una vez que se conoce su nombre.
- `security()` se invoca para realizar comprobaciones de seguridad en todas las operaciones sobre objetos, como por ejemplo un procesador abre o cierra un objeto, realiza modificaciones en el descriptor de seguridad o duplica un manejador de un objeto.

El procedimiento de análisis sintáctico (`parse`) se utiliza para ampliar el espacio de nombres abstracto con el fin de incluir archivos. La traducción de un nombre de ruta a un objeto archivo comienza en la raíz del espacio de nombres abstracto. Los componentes del nombre de ruta están separados mediante caracteres de barra invertida ('\\') en lugar de los caracteres de barra inclinada ('/') utilizados en UNIX. Durante el proceso de traducción, se busca cada componente en el directorio actual de análisis sintáctico del espacio de nombres. Los nodos internos dentro del espacio de nombres pueden ser directorios o enlaces simbólicos. Si se encuentra un objeto hoja y no queda ya ningún componente en el nombre de ruta, se devuelve el objeto hoja. En caso contrario, se invoca el procedimiento de análisis sintáctico del objeto hoja, pasándole el resto del nombre de ruta.

Los procedimientos de análisis sintáctico sólo se usan con un pequeño número de objetos pertenecientes a la interfaz de Windows, al gestor de configuración (registro) y, especialmente, a los objetos dispositivos que representan a los sistemas de archivos.

El procedimiento de análisis sintáctico para el tipo de objeto dispositivo asigna un objeto archivo e inicia una operación de E/S de apertura o de creación sobre el sistema de archivos. Si la operación tiene éxito, se rellenan los campos del objeto archivo para describir el archivo.

En resumen, el nombre de ruta de un archivo se emplea para recorrer el espacio de nombres del gestor de objetos, traduciendo el nombre de ruta absoluto original a una pareja (objeto dispositivo, nombre de ruta relativo). Esta pareja se pasa entonces al sistema de archivos a través del gestor de E/S, que se encarga de proporcionar el objeto archivo. El propio objeto archivo no tiene ningún nombre, efectuándose las referencias al mismo a través de un descriptor.

Los sistemas de archivos UNIX tienen **enlaces simbólicos** que permiten utilizar múltiples alias para un mismo archivo. El **objeto enlace simbólico** implementado por el gestor de objetos de Windows XP se utiliza dentro del espacio de nombres abstracto, no para proporcionar alias de archivos dentro de un sistema de archivos. Aún así, los enlaces simbólicos resultan muy útiles; se emplean para organizar el espacio de nombres, de forma similar a la organización del directorio /devices en UNIX. También se usan para asignar nombres de unidad a las letras de unidad estándar de MS-DOS. Las letras de unidad son enlaces simbólicos que pueden reasignarse a voluntad del usuario o del administrador.

Las letras de unidad son uno de los puntos en los que el espacio de nombres abstracto de Windows XP no tiene un carácter global. Cada usuario que inicia una sesión tiene su propio conjunto de letras de unidad para evitar que los usuarios se interfieran entre sí. Por contraste, las sesiones de servidor de terminales comparten todos los procesos dentro de una sesión. Los objetos **BaseNamedObjects** contienen los objetos con nombre creados por la mayor parte de las aplicaciones.

Aunque el espacio de nombres no es directamente visible a través de una red, el método **parse()** del gestor de objetos se utiliza como ayuda para acceder a un objeto con nombre almacenado en otro sistema. Cuando un proceso intenta abrir un objeto que reside en una computadora remota, el gestor de objetos llama al método de análisis sintáctico **parse** para el objeto dispositivo correspondiente a un redirector de red. Esto provoca una operación de E/S que accede al archivo a través de la red.

Los objetos son instancias de un **tipo de objeto**. El tipo de objeto especifica cómo hay que asignar las instancias, las definiciones de los campos de datos y la implementación del conjunto estándar de funciones virtuales utilizadas para todos los objetos. Estas funciones implementan operaciones tales como la asignación de nombres a los objetos, el cierre y borrado de objetos y la aplicación de medidas de seguridad.

El gestor de objetos mantiene dos contadores para cada objeto. El contador de punteros es el número de referencias diferentes realizadas a un objeto. El código en modo protegido que hace referencia a los objetos debe mantener una referencia al objeto con el fin de asegurar que el objeto no será borrado mientras esté en uso. Por su parte, el contador de descriptores representa el número de entradas de la tabla de descriptores que hacen referencia a un objeto. Cada descriptor está también reflejado en el contador de referencias.

Cuando se cierra un descriptor de un objeto, se invoca la rutina de cierre de ese objeto. En el caso de los objetos archivo, esta llamada hace que el gestor de E/S realice una operación de limpieza en el momento de cerrar el último descriptor. Esta operación de limpieza indica al sistema de archivos que el archivo ya no está siendo accedido en modo usuario, por lo que se pueden eliminar las restricciones de compartición, los bloqueos de rangos y otras informaciones de estado específicas de la correspondiente rutina de apertura.

Cada cierre de un descriptor elimina una referencia del contador de punteros, pero los componentes internos del sistema pueden contener referencias adicionales. Cuando se elimina la última referencia, se invoca el procedimiento de borrado del objeto. Utilizando de nuevo los objetos archivo como ejemplo, el procedimiento de borrado haría que el gestor de E/S enviara al sistema de archivos una operación de cierre relativa al objeto archivo. Esto hace que el sistema de archivos elimine la asignación de las estructuras de datos internas que hubieran sido asignadas para el objeto archivo.

Después de completarse el procedimiento de borrado para un objeto temporal, el objeto se borra de memoria. Los objetos pueden hacerse permanentes (al menos en lo que respecta al arranque actual del sistema) pidiendo al gestor de objetos que defina una referencia adicional al objeto. De este modo, los objetos permanentes no se borran incluso cuando se elimine la última referencia situada fuera del gestor de objetos. Cuando se vuelve a hacer temporal un objeto per-

manentemente, el gestor de objetos elimina esa referencia adicional. Si esa referencia adicional era la última, se borra el objeto. Los objetos permanentes no resultan muy comunes y se usan principalmente para los dispositivos, para el establecimiento de correspondencias entre unidades y letras de unidad y para los objetos directorio y objetos de enlace simbólico.

La tarea del gestor de objetos consiste en supervisar la utilización de todos los objetos gestionados. Cuando una hebra quiere utilizar un objeto, invoca el método `open()` del gestor para obtener una referencia al objeto. Si el objeto está siendo abierto desde una API en modo usuario, la referencia se inserta en la tabla de objetos del proceso y se devuelve un descriptor.

Un proceso obtiene un descriptor creando un objeto, abriendo un objeto existente, recibiendo un descriptor duplicado de otro proceso o heredando un descriptor de un **proceso padre**, de forma similar a la manera en que un proceso UNIX obtiene un descriptor de archivo. Estos descriptores se almacenan en la **tabla de objetos del proceso**. Cada entrada de la tabla de objetos contiene los derechos de acceso al objeto e indica si el descriptor debe ser heredado por los **procesos hijo**. Cuando un proceso termina, Windows XP cierra automáticamente todos los descriptores abiertos por el proceso.

Los **descriptores** son una interfaz estandarizada para todos los tipos de objetos. Al igual que un descriptor de archivo en UNIX, un descriptor de objeto es un identificador único para un proceso y que confiere a éste la capacidad de acceder y manipular un recurso del sistema. Los descriptores se pueden duplicar dentro de un proceso o entre un proceso y otro. Este último caso se utiliza cuando se crean procesos hijo y cuando se implementan contextos de ejecución fuera del proceso.

Puesto que el gestor de objetos es la única entidad que genera descriptores de objetos, representa el lugar natural en el que realizar las comprobaciones de seguridad. El gestor de objetos comprueba si un proceso tiene el derecho de acceder a un objeto en el momento en que el proceso trata de abrir dicho objeto. El gestor de objetos también se encarga de verificar que se respeten las cuotas, como por ejemplo la cantidad máxima de memoria que un proceso puede utilizar; para ello, atribuye al proceso la memoria ocupada por todos los objetos a los que hace referencia e impedirá que se le asigne más memoria cuando el proceso haya excedido la cuota asignada.

Cuando el proceso de inicio de sesión autentica a un usuario, se asocia un testigo de acceso con el proceso del usuario. Ese testigo de acceso contiene información tal como el ID de seguridad, los ID de grupo, los privilegios, el grupo principal y la lista predeterminada de control de acceso. Los servicios y objetos a los que el usuario puede acceder están determinados por estos atributos.

El testigo que controla el acceso está asociado con la hebra que efectúa el acceso. Normalmente, no se utiliza un testigo de hebra, empleándose de manera predeterminada el testigo del proceso, pero los servicios necesitan a menudo ejecutar código por cuenta de sus clientes. Por ello, Windows XP permite que las hebras adopten temporalmente una personalidad, utilizando el testigo de un cliente. De este modo, el testigo de la hebra no tiene por qué coincidir necesariamente con el testigo del proceso.

En Windows XP, cada objeto está protegido mediante una lista de control de acceso que contiene los ID de seguridad y los derechos de acceso concedidos. Cuando una hebra trata de acceder a un objeto, el sistema compara el ID de seguridad del testigo de acceso de la hebra con la lista de control de acceso del objeto para determinar si ese acceso debe permitirse. La comprobación se efectúa sólo en el momento de abrir un objeto, por lo que no resulta posible denegar el acceso después de efectuada la apertura. Los componentes del sistema operativo que se ejecutan en modo *kernel* puentean estas comprobaciones de acceso, ya que se asume que el código en modo *kernel* es de confianza. Por tanto, el código en modo *kernel* puede tratar de evitar las vulnerabilidades de seguridad, como por ejemplo dejar inhabilitadas las comprobaciones mientras se crea un descriptor accesible en modo usuario dentro de un proceso que no sea de confianza.

Generalmente, el creador del objeto determina la lista de control de acceso al mismo. Si no se suministra una lista explícitamente, la rutina de apertura del tipo de objeto puede definir una lista predeterminada, o bien puede obtenerse una lista predeterminada a partir del objeto que representa el testigo de acceso del usuario.

El testigo de acceso tiene un campo que controla la auditoría de los accesos al objeto. Las operaciones auditadas se registran en el registro de seguridad del sistema junto con una identifica-

ción del usuario. Un administrador puede monitorizar este registro para descubrir los intentos de irrupción en el sistema o los intentos de acceder a objetos protegidos.

22.3.3.2 Gestor de memoria virtual

El componente ejecutivo que gestiona el espacio virtual de direcciones, la asignación de memoria física y la paginación es el **gestor de memoria virtual** (VM, virtual memory). El diseño del gestor VM asume que el hardware subyacente da soporte a la traducción de direcciones virtuales a físicas, un mecanismo de paginación y un mecanismo de coherencia transparente de caché en los sistemas multiprocesador, además de permitir que se asignen múltiples entradas de la tabla de páginas al mismo marco de página física. El gestor VM en Windows XP utiliza un esquema de gestión basado en página con un tamaño de página de 4 KB en los procesadores compatibles con IA32 y de 8 KB en las máquinas IA64. Las páginas de datos asignadas a un proceso que no se encuentran en memoria física se almacenan en los **archivos de paginación** en disco o se mapean directamente a un archivo normal en un sistema de archivos local o remoto. Las páginas también pueden marcarse para llenarse con ceros bajo demanda, lo que escribe una serie de ceros en la página antes de asignarla, borrando así el contenido anterior.

En los procesadores IA32, cada proceso tiene un espacio virtual de direcciones de 4 GB. Los 2 GB superiores son fundamentalmente idénticos para todos los procesos y los utiliza Windows XP en modo *kernel* para acceder al código y a las estructuras de datos del sistema operativo. Las áreas clave de la región en modo *kernel* que no son idénticas para todos los procesos son el **automapa de tabla de páginas**, el **hiperespacio** y el **espacio de sesión**. El hardware hace referencia a las tablas de páginas de un proceso utilizando los números de marco físico de página. El gestor VM hace corresponder las tablas de páginas con una única región de 4 MB dentro del espacio de direcciones del proceso, de modo que se accede a ellas mediante direcciones virtuales. El hiperespacio mapea la información del conjunto de trabajo actual del proceso sobre el espacio de direcciones de modo *kernel*.

El espacio de sesión se utiliza para compartir los controladores de Win32 y otros controladores específicos de la sesión entre todos los procesos, dentro de una misma sesión de servidor de terminales, en lugar de para todos los procesos del sistema. Los 2 GB inferiores son específicos de cada proceso y a ellos pueden acceder tanto las hebras de modo *kernel* como las de modo usuario. Determinadas configuraciones de Windows XP sólo reservan 1 GB para el uso del sistema operativo, permitiendo a los procesos emplear un espacio de direcciones de 3 GB. Ejecutar el sistema en el modo 3 GB reduce drásticamente el almacenamiento de datos en caché dentro del *kernel*. Sin embargo, para las aplicaciones de gran envergadura que gestionan su propia E/S, como por ejemplo las bases de datos SQL, esta pérdida de tamaño de caché se ve compensada por la ventaja de disponer de un espacio de direcciones en modo usuario de mayor tamaño.

El gestor VM de Windows XP utiliza un proceso en dos pasos para asignar la memoria al usuario. El primer paso *reserva* una parte del espacio virtual de direcciones del proceso. El segundo paso *confirma* la asignación, asignando espacio de memoria virtual (memoria física o espacio en los archivos de paginación). Windows XP limita la cantidad de espacio de memoria virtual que un proceso consume, por el procedimiento de imponer una cuota máxima que restringe la memoria confirmada. Los procesos liberan la memoria que ya no utilizan con el fin de liberar memoria virtual para que la puedan utilizar otros procesos. Las API utilizadas para reservar direcciones virtuales y para confirmar memoria virtual toman como parámetro un descriptor de un objeto proceso. Esto permite que un proceso controle la memoria virtual del otro. Los subsistemas de entorno gestionan la memoria de sus procesos cliente de esta manera.

Por razones de rendimiento, el gestor VM permite que un proceso privilegiado bloquee una serie de páginas seleccionadas en la memoria física, garantizando así que esas páginas no desaparecen al archivo de paginación. Los procesos pueden también asignar memoria física y luego mapear regiones de la misma dentro de su espacio virtual de direcciones. Los procesadores IA32 con extensión de direcciones físicas (PAE, physical address extension) pueden tener hasta 64 GB de memoria física dentro de un mismo sistema. Esta memoria no puede mapearse de una sola vez dentro del espacio de direcciones de un proceso, pero Windows XP hace que se pueda acceder a

ella utilizando las API de extensión de ventanas de direcciones (AWE, address windowing extension), que asignan memoria física y luego mapean regiones de direcciones virtuales pertenecientes al espacio de direcciones del proceso sobre parte de la memoria física. La funcionalidad AWE es utilizada principalmente por aplicaciones de gran envergadura, como la base de datos de SQL.

Windows XP implementa la memoria compartida definiendo lo que se denomina un **objeto sección**. Después de obtener un descriptor de un objeto sección, los procesos mapean la parte de memoria que necesitan dentro de su espacio de direcciones. Cada una de esas partes se denomina **vista**. Un proceso puede redefinir su vista de un objeto con el fin de acceder al objeto completo, pasando de una región a otra.

Un proceso puede controlar de diferentes formas el uso de un objeto sección de memoria compartida. El tamaño máximo de una sección puede estar acotado. Asimismo, la sección puede estar respaldada por espacio de disco dentro del archivo de paginación del sistema o dentro de un archivo normal (lo que se denominaría **archivo mapeado en memoria**). Las secciones pueden estar *basadas*, lo que quiere decir que la sección aparece en la misma dirección virtual para todos los procesos que quieren acceder a ella. Finalmente, la protección de memoria de las páginas componentes de la sección puede configurarse como de sólo lectura, de lectura-escritura, de lectura-escritura-ejecución, de sólo ejecución, sin acceso o de copia durante la escritura. Conviene explicar estos dos últimos modos de configurar la protección:

- Una *página sin acceso* genera una excepción cada vez que se intenta acceder a ella; esa excepción se utiliza, por ejemplo, para comprobar si un programa erróneo está tratando de realizar una iteración más allá del final de una matriz. Tanto el asignador de memoria en modo usuario como el asignador especial del *kernel* utilizado por el verificador de dispositivos pueden configurarse para mapear cada asignación al final de una página que esté seguida por otra página sin acceso, con el fin de detectar los desbordamientos de búfer.
- El *mecanismo de copia durante la escritura* mejora la eficiencia del uso de memoria física por parte del gestor VM. Cuando dos procesos quieren tener copias independientes de un objeto, el gestor VM coloca una única copia compartida en memoria virtual y activa la propiedad de copia durante la escritura para esa región de memoria. Si uno de los procesos trata de modificar los datos contenidos en una página de copia durante la escritura, el gestor VM hace una copia privada de la página para ese proceso.

La traducción de direcciones virtuales en Windows XP utiliza una tabla de página multinivel. Para los procesadores IA32 que no tengan habilitada la extensión de direcciones físicas, cada proceso tiene un **directorio de páginas** que contiene 1.024 **entradas de directorio de páginas** (PDE, page-directory entry), cada una de las cuales tiene un tamaño de 4 bytes. Cada PDE apunta a una **tabla de páginas** que contiene 1.024 **entradas de tabla de páginas** (PTE, page-table entry), cada una de las cuales tiene también 4 bytes. Cada PTE apunta a un **marco de páginas** de 4 KB en la memoria física. El tamaño total de todas las tablas de página para un proceso es de 4 MB, por lo que el gestor VM descarga las tablas individuales a disco cada vez que sea necesario, utilizando el mecanismo de paginación. En la Figura 22.3 se muestra un diagrama de esta estructura,

El hardware hace referencia al directorio de páginas y a las tablas de páginas mediante sus direcciones físicas. Para mejorar el rendimiento, el gestor VM automapea el directorio de páginas y las tablas de página sobre una región de 4 MB de direcciones virtuales. El automapa permite que el gestor VM traduzca una dirección virtual a la correspondiente PDE o PTE sin necesidad de accesos a memoria adicionales. Cuando se cambia el contexto de un proceso, sólo se necesita modificar una única entrada del directorio de páginas, con el fin de hacerla corresponder con las nuevas tablas de páginas del proceso. Por diversas razones, el hardware requiere que cada directorio de páginas o tabla de páginas ocupe una única página. Por tanto, el número de entradas PDE o PTE que caben en una página determina el modo en que se traducen las direcciones virtuales.

Vamos a describir a continuación la manera de traducir las direcciones virtuales a direcciones físicas en los procesadores compatibles con IA32 (que no tengan habilitada la extensión PAE). Un campo de 10 bits permite representar todos los valores de 0 a 1.023. Por tanto, un valor de 10 bits puede seleccionar cualquier entrada del directorio de páginas o de una tabla de páginas. Esta propiedad se emplea cuando se traduce un puntero de dirección virtual a una dirección de byte

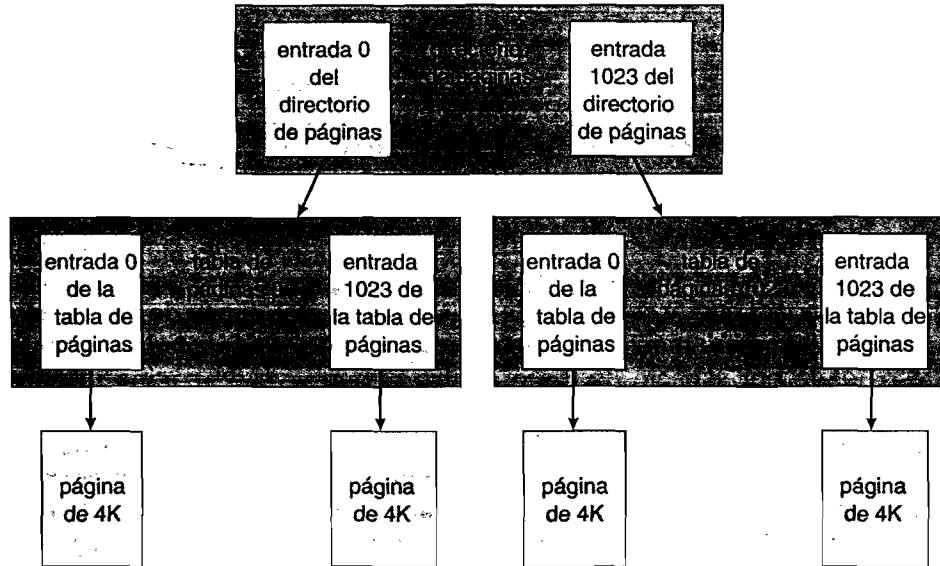


Figura 22.3 Disposición de las tablas de páginas.

dentro de la memoria física. Una dirección virtual de memoria de 32 bits se divide en tres valores, como se muestra en la Figura 22.4. Los primeros 10 bits de la dirección virtual se utilizan como índice para acceder al directorio de páginas. Esta dirección selecciona una entrada del directorio de páginas (PDE), que contiene el marco de página físico de una tabla de páginas. La unidad de gestión de memoria (MMU) utiliza los siguientes 10 bits de la dirección virtual para seleccionar una PTE de la tabla de páginas. La entrada PTE especifica un marco de página dentro de la memoria física. Los restantes 12 bits de la dirección virtual son el desplazamiento correspondiente a un byte específico dentro del marco de página. La MMU crea un puntero al byte específico de memoria física concatenando los 20 bits de la PTE con los 12 bits inferiores de la dirección virtual. De este modo, la PTE de 32 bits dispone de 12 bits para describir el estado de la página física. El hardware IA32 reserva 3 de esos bits para su uso por el sistema operativo, mientras que los restantes bits especifican si se ha accedido a la página o se ha escrito en ella, los atributos de caché, el modo de acceso, si se trata de una página global y si la PTE es válida.

Los procesadores IA32 que tengan habilitada la extensión PAE utilizan entradas PDE y PTE de 64 bits para representar el campo del número de marco de página, que tiene un tamaño mayor de 24 bits. Por esto, los directorios de página de segundo nivel y las tablas de páginas contienen sólo 512 entradas PDE y PTE, respectivamente. Para proporcionar 4 GB de espacio virtual de direcciones hace falta un nivel adicional de directorio de páginas que contenga cuatro entradas PDE. La traducción de direcciones virtuales de 32 bits utiliza 2 bits como índice para el directorio de nivel superior y 9 bits para los directorios de páginas segundo nivel y para las tablas de páginas.

Para evitar tener que traducir todas las direcciones virtuales buscando las entradas PDE y PTE, los procesadores emplean un búfer de traducción directa (TLB, translation-lookaside buffer), que contiene una caché de memoria asociativa que asigna páginas virtuales a entradas PTE. A diferencia de la arquitectura IA32, en la que el TLB es mantenido por la MMU hardware, la arquitectura

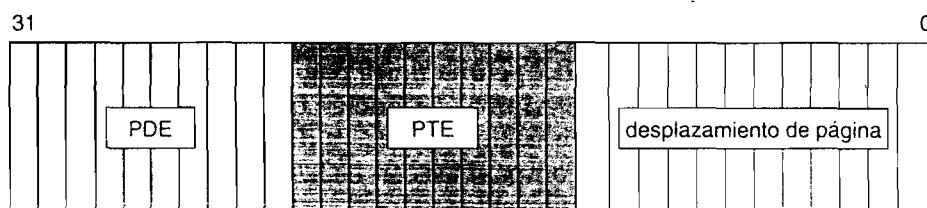


Figura 22.4 Traducción de direcciones virtuales a físicas en la arquitectura IA32.

IA64 invoca una rutina de interrupción software para poder realizar las traducciones que no se pueden llevar a cabo a partir del contenido del TLB. Esto proporciona flexibilidad al gestor VM a la hora de elegir las estructuras de datos que haya que utilizar. En Windows XP, se selecciona una estructura en árbol de tres niveles para traducir las direcciones virtuales de modo usuario en la especificación IA64.

En los procesadores IA64, el tamaño de página es de 8 KB, pero las entradas PTE ocupan 64 bits, por lo que cada página seguirá conteniendo sólo 1.024 (lo que equivale a 10 bits) entradas PDE o PTE. Por tanto, con 10 bits para la PDE de nivel superior, 10 bits para el segundo nivel, 10 bits para la tabla de páginas y 13 bits de desplazamiento de página, la parte del usuario del espacio virtual de direcciones de un proceso en Windows XP sobre arquitectura IA64 es de 8 TB (lo que equivale a 43 bits). La limitación de 8 TB en la versión actual de Windows XP es más restrictiva que las capacidades del procesador IA64, pero representa un compromiso entre el número de referencias de memoria requeridas para gestionar los fallos de TLB y el tamaño del espacio de direcciones soportado en modo usuario.

Una página física puede encontrarse en uno de seis posibles estados: válida, libre, borrada, modificada, reserva, corrupta o en transición.

- Una página *válida* es aquella que está siendo usada por un proceso activo.
- Una página *libre* es una página que no está referenciada por una entrada PTE.
- Una página *borrada* es una página libre que ha sido rellenada de ceros y está lista para ser inmediatamente utilizada con el fin de satisfacer fallos de relleno de ceros bajo demanda.
- Una página *modificada* es aquella en la que ha escrito un proceso y debe enviarse a disco antes de asignarla a otro proceso.
- Una página de *reserva* es una copia de información que ya está almacenada en disco. Las páginas de reserva pueden ser páginas que no hayan sido modificadas, páginas modificadas que ya hayan sido escritas en el disco o páginas que hayan sido extraídas de manera anticipada para aprovechar la característica de localidad de los datos.
- Una página *corrupta* no es utilizable porque se ha detectado un error hardware.
- Por último, una página *en transición* es aquella que está en camino desde el disco a un marco de página asignado dentro de la memoria física.

Cuando el bit válido/inválido en una entrada PTE es cero, el gestor VM define el formato de los otros bits. Las páginas no válidas pueden tener diversos estados que estarán representados por los bits contenidos en la entrada PTE. Las páginas del archivo de páginas que nunca hayan sido cargadas como consecuencia de un fallo página se marcan como páginas de relleno de ceros bajo demanda. Los archivos mapeados a través de objetos sección tienen asociado un puntero a dicho objeto sección. Las páginas que hayan sido escritas en el archivo de páginas contendrán suficiente información como para encontrar la página en disco, y así sucesivamente.

En la Figura 22.5 se muestra la estructura real de la PTE del archivo de páginas. La PTE contiene 5 bits de protección de página, 20 bits para el desplazamiento dentro del archivo de páginas, 4 bits para seleccionar el archivo de paginación y 3 bits que describen el estado de la página. Una PTE del archivo de páginas se marca como dirección virtual no válida de cara a la MMU. Puesto que el código ejecutable y los archivos mapeados en memoria disponen ya de una copia en el disco, no necesitan espacio dentro de un archivo de paginación. Si una de estas páginas no se encuentra en la memoria física, la estructura de la PTE es la siguiente: el bit más significativo se utiliza para especificar la protección de página, los siguientes 28 bits se usan como índice dentro de una estructura de datos del sistema que indica un archivo y un desplazamiento dentro del archivo correspondientes a la página, y los 3 bits inferiores especifican el estado de la página.

Las direcciones virtuales no válidas también pueden encontrarse en diversos estados temporales que forman parte de los algoritmos de paginación. Cuando se elimina una página del conjunto de trabajo de un proceso, se la mueve a la lista de páginas modificadas (que habrá que escribir en el disco) o directamente a la lista de páginas de reserva. Si se escribe en la lista de páginas de reserva, se reclamará la página sin volver a leerla del disco cuando se la necesite de nuevo,



Figura 22.5 Entrada de la tabla de páginas del archivo de páginas. El bit válido/inválido es cero.

siempre y cuando la página no haya sido trasladada todavía a la lista de páginas libres. Siempre que sea posible, el gestor VM utilizará ciclos de inactividad de la CPU para escribir ceros en la lista de páginas libres y así moverlas a la lista de páginas borradas. Las páginas en transición son aquellas a las que se les ha asignado una página física y que están esperando a que termine la operación de E/S de paginación antes de marcar la entrada PTE como válida.

Windows XP utiliza los objetos sección para describir aquellas páginas que son compartibles entre distintos procesos. Cada proceso tiene su propio conjunto de tablas de páginas virtuales, pero el objeto sección también incluye un conjunto de tablas de páginas que contiene las entradas PTE maestra (o prototípicas). Cuando una PTE dentro de la tabla de páginas de un proceso está marcada como válida, apunta al marco de página físico que contiene a esa página, como debe ser en los procesadores IA32, en los que la MMU hardware lee las tablas de páginas directamente de la memoria. Pero cuando una página compartida pasa a ser no válida, se modifica la PTE para que apunte a la PTE prototípica asociada con el objeto sección.

Las tablas de páginas asociadas con un objeto sección son virtuales en el sentido de que se las crea y se las elimina según sea necesario. Las únicas PTE prototípicas necesarias son aquellas que describen páginas para las que existe una vista actualmente mapeada. Esto mejora enormemente el rendimiento y permite utilizar de forma más eficiente las direcciones virtuales del kernel.

La PTE prototípica contiene la dirección del marco de página y los bits de protección y de estado. Así, el primer acceso a una página compartida por un proceso genera un fallo de página. Después del primer acceso, los accesos siguientes se realizan de la forma normal. Si un proceso escribe en una página de copia durante la escritura marcada como de sólo lectura en la PTE, el gestor VM hace una copia de la página y marca la PTE como escribible, después de lo cual el proceso dejará en la práctica de tener una página compartida. Las páginas compartidas nunca aparecen en el archivo de páginas, sino que se las puede localizar en el sistema de archivos.

El gestor VM controla las páginas de memoria física mediante una base de datos de marcos de página. Existe una entrada por cada página de memoria física que haya en el sistema. Esta entrada apunta a la PTE, que a su vez apunta al marco de página, de modo que el gestor VM puede controlar el estado de la página. Los marcos de página que no estén referenciados por una PTE válida se enlazan en una serie de listas de acuerdo con el tipo de página concreto, como por ejemplo páginas borradas, modificadas o libres.

Si se marca una página física compartida como válida para cualquier proceso, esa página no puede eliminarse de la memoria. El gestor VM mantiene un contador de entradas PTE válidas para cada página, dentro de la base de datos de marcos de página. Cuando el contador pasa a valer cero, la página física puede reutilizarse, después de escribir su contenido en el disco (si la página había sido modificada y estaba marcada como "sucia").

Cuando se produce un fallo de página, el gestor VM localiza una página física para albergar los datos. Para las páginas con relleno de ceros bajo demanda, la primera opción consiste en localizar una página que ya haya sido borrada. Si no hay ninguna disponible, se selecciona una página de la lista de páginas libres o de la lista de páginas de reserva, y se borran los datos de la página antes de continuar. Si la página que ha provocado el fallo ha sido marcada como página en transición, ya estará siendo leída del disco o habrá sido eliminada y continuará estando disponible en la lista de páginas de reserva o de páginas modificadas. Entonces, la hebra esperará a que la operación de E/S se complete o, en los últimos casos, reclamará la página de la lista apropiada.

2. La función `CreateProcess()` en el proceso original invoca entonces una API de gestor de procesos del ejecutivo NT para proceder a la creación del proceso.
3. El gestor de procesos invoca al gestor de objetos para crear un objeto proceso y devuelve el descriptor del objeto a la API Win32.
4. La API Win32 invoca al gestor de procesos de nuevo para crear una hebra para el proceso, y devuelve los descriptores del nuevo proceso y de la nueva hebra.

Las API de Windows XP para manipular la memoria virtual y las hebras y para duplicar los descriptores toman como parámetro un descriptor de proceso, de modo que los subsistemas puedan realizar operaciones por cuenta de un nuevo proceso sin tener que ejecutar código directamente dentro del contexto del nuevo proceso. Una vez creado un nuevo proceso, se crea la hebra inicial y se entrega una llamada a procedimiento asíncrona a la hebra para provocar el comienzo de la ejecución, que tendrá lugar en el cargador de imagen de modo usuario. El cargador es una biblioteca `ntdll.dll`, que es una biblioteca de montaje que se mapea automáticamente en todo proceso de nueva creación. Windows XP también soporta un estilo de creación de procesos de tipo `fork()` de UNIX, con el fin de dar soporte al subsistema de entorno POSIX. Aunque el entorno API Win32 invoca al gestor de procesos desde el proceso cliente, POSIX emplea la naturaleza inter-procesos de las API Windows XP para crear el nuevo proceso desde el proceso de subsistema.

El gestor de procesos también implementa la entrega y puesta en cola de llamadas a procedimiento asíncronas (APC, asynchronous procedure call) a las hebras. Las APC son utilizadas por el sistema para iniciar la ejecución de las hebras, para completar la E/S, para terminar las hebras y procesos y para asociar depuradores. El código en modo usuario también puede tener en cola una APC dirigida a una hebra, para entregar notificaciones de tipo señal. Para soportar la especificación POSIX, el gestor de procesos proporciona diversas API que envían alertas a las hebras con el fin de desbloquearlas después de que éstas hayan efectuado llamadas al sistema.

El soporte de depuración en el gestor de procesos incluye la capacidad de suspender y reanudar hebras y de crear hebras que comiencen en modo suspendido. También hay diversas API del gestor de procesos que permiten configurar y consultar el contexto de registros de una hebra y acceder a la memoria virtual de otro proceso.

Pueden crearse hebras dentro del proceso actual y también se las puede inyectar dentro de otro proceso. Dentro del ejecutivo, las hebras existentes pueden asociarse temporalmente a otro proceso. Las hebras de trabajo que necesitan ejecutarse dentro del contexto de un proceso que haya originado una solicitud de trabajo utilizan este método.

El gestor de procesos también soporta la característica de suplantación. Una hebra que se esté ejecutando dentro de un proceso con un testigo de seguridad que pertenezca a un usuario puede configurar un testigo específico de la hebra perteneciente a otro usuario. Esta funcionalidad resulta fundamental dentro del modelo cliente-servidor, en el que los servicios necesitan actuar por cuenta de diversos clientes, los cuales tendrán diferentes ID de seguridad.

22.3.3.4 Funcionalidad de llamadas a procedimientos locales

La implementación de Windows XP utiliza un modelo cliente-servidor. Los subsistemas de entorno son servidores que implementan personalidades concretas del sistema operativo. El modelo cliente-servidor se emplea para implementar diversos servicios del sistema operativo, que no pertenecen a los subsistemas de entorno. La gestión de seguridad, la gestión de impresión, los servicios web, los sistemas de archivos de red, las funciones plug-and-play y muchas otras características se implementan utilizando este modelo. Para reducir el consumo de memoria, se suelen agrupar múltiples servicios dentro de unos cuantos procesos, que utilizan la funcionalidad de gestión de conjuntos de hebras en modo usuario para compartir hebras y esperar a la recepción de mensajes (véase la Sección 22.3.3).

El sistema operativo utiliza la funcionalidad de llamada a procedimiento local (LPC, local procedure call) para intercambiar solicitudes y resultados entre los procesos cliente y servidor dentro de una misma máquina. En particular, la funcionalidad LPC se utiliza para solicitar servicios de los diversos subsistemas Windows XP. LPC es similar en muchos aspectos a los mecanismos RPC

utilizados por muchos sistemas operativos para procesamiento distribuido a través de las redes, pero LPC está optimizado para usarlo dentro de un único sistema. La implementación Windows XP del mecanismo RPC de OSF (Open Software Foundation) a menudo utiliza la funcionalidad LPC como transporte dentro de la máquina local.

LPC es un mecanismo de paso de mensajes. El proceso servidor publica un objeto puerto de conexión que es visible globalmente. Cuando un cliente desea un servicio de un subsistema, abre un descriptor del objeto puerto de conexión del subsistema y envía una solicitud de conexión a dicho puerto. El servidor crea un canal y devuelve un descriptor al cliente. El canal está compuesto por un par de puertos privados de comunicación: uno para los mensajes del cliente al servidor y otro para los mensajes del servidor al cliente. Los canales de comunicación soportan un mecanismo de retrollamada, por lo que el cliente y el servidor pueden aceptar solicitudes en aquellos casos en los que normalmente estarían esperando recibir una respuesta.

Cuando se crea un canal LPC, debe especificarse una de tres posibles técnicas de paso de mensajes.

1. La primera técnica es adecuada para mensajes de pequeño tamaño (hasta un par de cientos de bytes). En este caso, la cola de mensajes del puerto se emplea como almacenamiento intermedio y los mensajes se copian de un proceso al otro.
2. La segunda técnica se emplea para mensajes de mayor tamaño. En este caso, se crea un objeto sección de memoria compartida para el canal. Los mensajes enviados a través de la cola de mensajes del puerto contienen un puntero que hace referencia al objeto sección junto con la necesaria información de tamaño. Esto evita la necesidad de copiar mensajes de gran longitud. El emisor coloca los datos en la sección compartida y el receptor los puede consultar directamente.
3. La tercera técnica utiliza las API que leen y escriben directamente en el espacio de direcciones de un proceso. La funcionalidad LPC proporciona funciones y mecanismos de sincronización, de modo que un servidor pueda acceder a los datos almacenados en un cliente.

El gestor de ventanas de la API Win32 utiliza su propia forma de paso de mensajes, que es independiente del mecanismo LPC ejecutivo. Cuando un cliente solicita una conexión que utiliza mensajes del gestor de ventanas, el servidor construye tres objetos: (1) una hebra dedicada de servidor para gestionar las solicitudes, (2) un objeto sección de 64 KB y (3) un objeto pareja de sucesos. Un *objeto pareja de sucesos* es un objeto de sincronización empleado por el subsistema API Win32 para proporcionar una notificación cuando la hebra cliente haya copiado un mensaje al servidor API Win32, o viceversa. El objeto sección pasa los mensajes y el objeto pareja de sucesos se encarga de la sincronización.

El mecanismo de mensajería del gestor de ventanas tiene varias ventajas:

- El objeto sección elimina la necesidad de copiar mensajes, ya que representa una región de memoria compartida.
- El objeto pareja de sucesos elimina el gasto adicional implicado en la utilización del objeto puerto para pasar mensajes que contienen punteros de información de longitud.
- La hebra dedicada de servidor elimina el coste adicional de determinar qué hebra cliente está llamando al servidor, ya que existe una hebra de servidor por cada hebra cliente.
- El *kernel* asigna una preferencia de planificación a estas hebras dedicadas de servidor, con el fin de mejorar las prestaciones.

22.3.3.5 Gestor de E/S

El gestor de E/S es responsable de los sistemas de archivos, controladores de dispositivos y controladores de red. Se encarga de controlar qué controladores de dispositivo, controladores de filtrado y sistemas de archivos se han cargado, y también gestiona los búferes para las solicitudes de E/S. Coopera con el gestor VM para proporcionar un mecanismo de E/S de archivo mapeada

en memoria y controla el gestor de caché de Windows XP que se encarga de gestionar el almacenamiento en caché para todo el sistema de E/S. El gestor de E/S es fundamentalmente asíncrono. La E/S síncrona se proporciona esperando explícitamente a que se complete una operación de E/S. El gestor de E/S proporciona varios modelos de terminación asíncrona de las operaciones de E/S, incluyendo la configuración de sucesos, la entrega de llamadas APC a la hebra iniciadora y el uso de puertos de terminación de E/S, que permiten que una única hebra procese los sucesos de terminación de E/S correspondientes a otras muchas hebras.

Los controladores de dispositivos están organizados como una lista para cada dispositivo (lo que se denomina una pila de controlador o de E/S, debido al modo en que se añaden los controladores de dispositivo). El gestor de E/S convierte las solicitudes que recibe a un formato estándar denominado **paquete de solicitud de E/S** (IRP, request packet I/O). A continuación, re-envía el IRP al primer controlador de la pila para su procesamiento. Después de que cada controlador procese el IRP, invoca al gestor de E/S, bien para re-enviar el paquete al siguiente controlador de la pila o, si ha finalizado todo el procesamiento, para completar las operaciones realizadas con el IRP.

La terminación de esas operaciones puede producirse en un contexto distinto del correspondiente a la solicitud de E/S original. Por ejemplo, si un controlador está llevando a cabo su parte de una operación de E/S y se ve obligado a bloquearse durante un período de tiempo prolongado, puede poner en cola la IRP para que una hebra de trabajo continúe con el procesamiento dentro del contexto del sistema. En la hebra original, el controlador devuelve un código de estado que indica que la solicitud de E/S está pendiente, de modo que la hebra pueda seguir ejecutándose en paralelo con la operación de E/S. Los paquetes IRP también pueden procesarse en rutinas de servicio de interrupciones y completarse dentro de un contexto arbitrario. Puesto que puede ser necesario realizar un cierto procesamiento final dentro del contexto que inició la operación de E/S, el gestor de E/S utiliza una APC para realizar el procesamiento final de terminación de la E/S dentro del contexto de la hebra original.

El modelo de pila es muy flexible. A medida que se construye una pila de controladores, los diversos controladores tienen la oportunidad de insertarse dentro de la pila como **controladores de filtrado**. Los controladores de filtrado pueden examinar y, potencialmente, modificar cada una de las operaciones de E/S. La gestión de montaje, la gestión de particiones y los mecanismos de duplicación de duplicación en bandas de los discos son algunos ejemplos de funcionalidad implementada mediante controladores de filtrado que se ejecutan por debajo del sistema de archivos dentro de la pila. Los controladores de filtrado del sistema de archivos se ejecutan por encima del sistema de archivos y se los utiliza para implementar funcionalidad tal como la gestión de almacenamiento jerárquico, la instanciación simple de archivos para arranque remoto y la conversión dinámica de formatos. Otros fabricantes de software utilizan también controladores de filtrado del sistema de archivos para implementar mecanismos de detección de virus.

Los controladores de dispositivos de Windows XP se escriben de acuerdo con la especificación WDM (Windows Driver Model). Este modelo establece todos los requisitos para los controladores de dispositivo, incluyendo cómo apilar controladores de filtrado, cómo compartir código común para gestionar las solicitudes plug-and-play y de administración de energía, cómo construir lógica de cancelación correcta, etc.

Debido a la complejidad del modelo WDM, escribir un controlador de dispositivo WDM para cada nuevo dispositivo de hardware puede requerir una cantidad de trabajo excesiva. Afortunadamente, el modelo puerto/minipuerto hace que esto sea innecesario. Dentro de una clase de dispositivos similares, como por ejemplo controladores de audio, dispositivos SCSI o controladoras Ethernet, cada instancia de un dispositivo comparte un controlador común de la clase denominado **controlador de puerto**. El controlador de puerto implementa las operaciones estándar para la clase y luego invoca las rutinas específicas del dispositivo dentro del **controlador de minipuerto** de éste, con el fin de implementar la funcionalidad específica que ese dispositivo proporcione.

22.3.3.6 Gestor de caché

En muchos sistemas operativos, del almacenamiento en caché se encarga el sistema de archivos. Por el contrario, Windows XP proporciona una funcionalidad de caché centralizada. El gestor de

caché coopera estrechamente con el gestor VM con el fin de proporcionar servicios de caché para todos los componentes que están bajo el control del gestor de E/S. El almacenamiento en caché en Windows XP está basado en archivos en lugar de en bloque sin formato.

El tamaño de la caché cambia dinámicamente de acuerdo con la memoria libre que haya disponible en el sistema. Recuerde que los 2 GB superiores del espacio de direcciones de un proceso forman el área del sistema, que estará disponible dentro del contexto de todos los procesos. El gestor VM asigna hasta un cincuenta por ciento de este espacio a la caché del sistema. El gestor de caché mapea archivos sobre este espacio de direcciones y emplea las capacidades del gestor VM para gestionar la E/S de archivos.

La caché está dividida en bloques de 256 KB. Cada bloque de la caché puede almacenar una vista (es decir, como una región mapeada en memoria) de un archivo. Cada bloque de caché está descrito en un **bloque de control de dirección virtual** (VACB, virtual address control block), que almacena la dirección virtual y el desplazamiento dentro del archivo para la vista, así como el número de procesos que están utilizando la vista. Los VACB residen en una única matriz, de cuyo mantenimiento se encarga el gestor de caché.

Para cada archivo abierto, el gestor de caché mantiene una matriz índice VACB independiente que describe el almacenamiento en caché para el archivo completo. Esta matriz tiene una entrada por cada fragmento de 256 KB del archivo; de este modo, por ejemplo, un archivo de 2 MB tendría una matriz índice VACB de 8 entradas. Cada entrada de la matriz índice VACB apunta al VACB si dicha parte del archivo se encuentra en la caché; en caso contrario, la entrada tendrá un valor nulo. Cuando el gestor de E/S recibe una solicitud de lectura de archivo de nivel de usuario, envía un paquete IRP a la pila de controladores de dispositivo en la que reside el archivo. El sistema de archivos trata de buscar los datos solicitados mediante el gestor de caché (a menos que la solicitud especifique que debe realizarse una lectura no dirigida a la memoria caché). El gestor de caché calcula qué entrada de la matriz índice VACB de dicho archivo se corresponde con el desplazamiento en bytes de la solicitud. Esa entrada apuntará a la vista almacenada en la caché o será inválida. Si es inválida, el gestor de caché asignará un bloque de caché (y la entrada correspondiente en la matriz VACB) y mapeará la vista sobre el bloque de caché. El gestor de caché tratará entonces de copiar los datos del archivo mapeado al búfer del proceso que ha realizado la invocación. Si la copia tiene éxito, la operación se completa. Si la copia falla, lo hará debido a un fallo de página que provocará que el gestor VM envíe al gestor de E/S una solicitud de lectura no dirigida a la caché. El gestor de E/S enviará otra solicitud hacia abajo a través de la pila de controladores solicitando esta vez una operación de *paginación*, que puentea al gestor de caché y lee los datos del archivo directamente en la página asignada al gestor de caché. Al terminar la operación, se modifica el VACB para que apunte a esa página. Los datos, que ahora estarán en la caché, se copian en el búfer del proceso que ha realizado la invocación, y la solicitud de E/S original se completa. La Figura 22.6 muestra un resumen de estas operaciones.

Siempre que sea posible, para las operaciones síncronas sobre los archivos almacenados en caché, las operaciones de E/S gestionadas por el **mecanismo de E/S rápida**. Este mecanismo es similar a la E/S normal basada en paquetes IRP, pero invoca la pila de controladores directamente, en lugar de pasar hacia abajo un paquete IRP. Puesto que no se utiliza un IRP, la operación no debe bloquearse durante un período largo de tiempo y no debe ponerse en una cola para ser procesada por una hebra de trabajo. Por tanto, cuando la operación alcanza el sistema de archivos e invoca al gestor de caché, la operación fallará si la información no se encuentra ya en la caché. El gestor de E/S tratará entonces de llevar a cabo la operación utilizando la ruta normal basada en paquetes IRP.

Las operaciones de lectura de nivel de *kernel* son similares, excepto en que puede accederse a los datos directamente desde la caché, en lugar de copiarlos a un búfer en el espacio de usuario. Para utilizar los metadatos del sistema de archivos (estructuras de datos que describen el sistema de archivos), el *kernel* emplea la interfaz de mapeo del gestor de caché con el fin de leer los metadatos. Para modificar los metadatos, el sistema de archivos utiliza la interfaz de anclaje del gestor de caché. Anclar una página hace que esa página quede fija en un marco de página de memoria física, de modo que el gestor VM no pueda moverla ni descargarla como resultado de una operación de paginación. Después de actualizar los metadatos, el sistema de archivos solicita al gestor

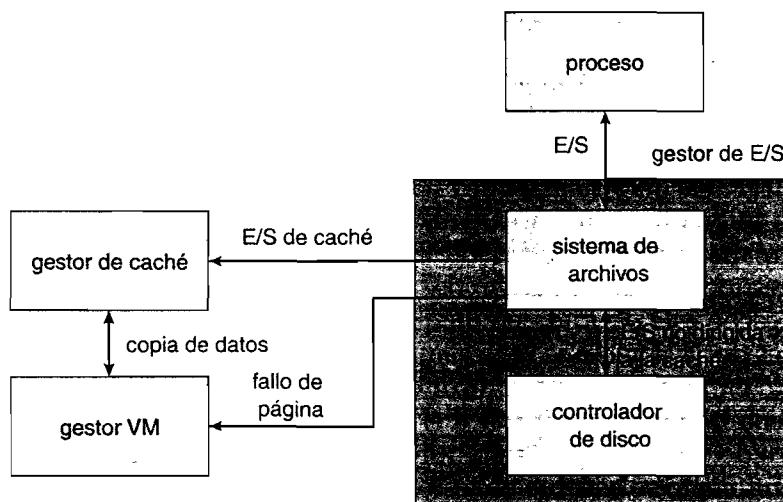


Figura 22.6 E/S de archivo.

de caché que elimine el anclaje de la página. Las páginas modificadas se marcan como sucias, para que el gestor VM las vuelque al disco. Los metadatos se almacenan en un archivo.

Para mejorar las prestaciones, el gestor de caché mantiene un pequeño historial de las solicitudes de lectura y trata de predecir las solicitudes futuras a partir de este historial. Si el gestor de caché encuentra un patrón en las tres solicitudes anteriores, como por ejemplo un acceso secuencial hacia delante o hacia atrás, extrae anticipadamente los datos y los almacena en la caché antes de que la aplicación envíe la siguiente solicitud. De esta forma, la aplicación encontrará los datos en la caché y no necesitará esperar a que se produzca una operación de E/S de disco. Las funciones `OpenFile()` y `CreateFile()` de la API Win32 admiten como parámetro un indicador `FILE_FLAG_SEQUENTIAL_SCAN`, que es una sugerencia para que el gestor de caché trate de extraer anticipadamente 192 KB más allá de las solicitudes efectuadas por la hebra. Normalmente, Windows XP realiza operaciones de E/S en fragmentos de 64 KB o 16 páginas; por tanto, esta lectura anticipada equivale a tres veces la cantidad normal de datos leídos.

El gestor de caché también es responsable de decir al gestor VM que vuelque el contenido de la caché en el disco. El comportamiento predeterminado del gestor de caché consiste en aplicar un mecanismo de escritura retardada: acumula las escrituras durante 4 o 5 segundos y luego despertar a la hebra escritora de caché. Cuando se necesita emplear un mecanismo de escritura directa, un proceso puede configurar un indicador en el momento de abrir un archivo, o bien el proceso puede invocar una función explícita de volcado de caché.

Un proceso que realice escrituras con gran rapidez podría llegar a llenar todas las páginas libres de caché antes de que la hebra escritora de caché tenga la oportunidad de despertarse y volver las páginas al disco. El escritor de caché evita que un proceso inunde el sistema de la siguiente forma: cuando la cantidad de memoria libre en la caché es muy baja, el gestor de caché bloquea temporalmente los procesos que estén intentando escribir datos y despertar a la hebra escritora de caché, con el fin de que vuelque las páginas al disco. Si el proceso que realiza escrituras con gran rapidez es un redirector de red para un sistema de archivos de red, bloquearlo durante demasiado tiempo podría provocar fines de temporización en las transferencias de red con sus consiguientes retransmisiones. Estas retransmisiones representarían un desperdicio del ancho de banda de red. Para evitar este desperdicio, los redirectores de red pueden ordenar al gestor de caché que limite el número de escrituras pendientes en la caché.

Puesto que los sistemas de archivos en red necesitan transferir datos entre el disco y la interfaz de red, el gestor de caché también proporciona una interfaz DMA para poder mover los datos directamente. Esta transferencia directa de los datos evita la necesidad de copiarlos en un búfer intermedio.

22.3.3.7 Monitor de referencia de seguridad

La centralización de la gestión de las entidades del sistema en el gestor de objetos permite a Windows XP utilizar un mecanismo uniforme para llevar a cabo las validaciones de acceso en tiempo de ejecución y las comprobaciones de auditoría para todas las entidades del sistema a las que puedan acceder los usuarios. Cada vez que un proceso abre un descriptor de un objeto, el **monitor de referencia de seguridad** (SRM, security reference monitor) comprueba el testigo de seguridad del proceso y la lista de control de acceso del objeto para ver si el proceso tiene los derechos necesarios.

El SRM también es responsable de manipular los privilegios en los testigos de seguridad. Se requieren privilegios especiales para que los usuarios realicen operaciones de copia de seguridad o restauración en los sistemas de archivos, para evitar ciertas comprobaciones como administrador, para depurar los procesos, etc. Los testigos también pueden marcarse como restringidos en lo que a sus privilegios se refiere, para que no puedan acceder a objetos que estén disponibles para la mayoría de los usuarios. Los testigos restringidos se utilizan principalmente para limitar el daño que puede provocar la ejecución de código que no sea de confianza.

Otra responsabilidad del SRM es llevar un registro de los sucesos de auditoría de seguridad. La clasificación de seguridad C-2 requiere que el sistema tenga la capacidad de detectar y registrar todos los intentos de acceder a recursos del sistema, para que resulte más sencillo trazar los intentos de acceso no autorizados. Puesto que el SRM es responsable de realizar las comprobaciones de acceso, es él quien genera la mayoría de las entradas de registro de auditoría en el registro de sucesos de seguridad.

22.3.3.8 Gestores plug-and-play y de administración de energía

El sistema operativo utiliza el **gestor plug-and-play** (PnP) para reconocer los cambios en la configuración hardware y adaptarse a los mismos. Para que el mecanismo PnP funcione, tanto el dispositivo como el controlador deben soportar el estándar PnP. El gestor PnP reconoce automáticamente los dispositivos instalados y detecta los cambios en dichos dispositivos mientras el sistema sigue funcionando. El gestor también controla los recursos utilizados por cada dispositivo, así como los recursos potenciales que podrían llegar a utilizarse y se responsabiliza de cargar los controladores apropiados. Esta gestión de los recursos hardware (principalmente interrupciones y rangos de memoria de E/S) tiene el objetivo de determinar una configuración hardware en la que todos los dispositivos sean capaces de funcionar correctamente.

Por ejemplo, si el dispositivo B puede utilizar la interrupción 5 y el dispositivo A puede usar la 5 o la 7, entonces el gestor PnP asignará la 5 a B y la 7 a A. En las versiones anteriores, el usuario podía tener que eliminar el dispositivo A y reconfigurarlo para que utilizara la interrupción 7 antes de poder instalar el dispositivo B. Eso obligaba al usuario a estudiar los recursos del sistema antes de instalar nuevo hardware y determinar qué dispositivos estaban usando cada recurso hardware. La proliferación de tarjetas PCMCIA, de estaciones de acoplamiento de portátiles y de dispositivos USB, IEEE 1394, Infiniband y otros dispositivos conectables en caliente también exige soportar la existencia de recursos dinámicamente configurables.

El gestor PnP gestiona la reconfiguración dinámica de la forma siguiente: en primer lugar, obtiene una lista de dispositivos de cada controlador de bus (por ejemplo, PCI, USB). Después carga el controlador instalado (o instala uno en caso necesario) y envía una solicitud `add-device` al controlador apropiado para cada dispositivo. El gestor PnP determina la asignación óptima de recursos y envía una solicitud `start-device` a cada controlador, junto con la asignación de recursos correspondiente a ese dispositivo. Si es necesario reconfigurar un dispositivo, el gestor PnP envía una solicitud `query-stop`, que pregunta al controlador si puede desactivarse temporalmente el dispositivo. Si el controlador puede desactivar el dispositivo, entonces se completan todas las operaciones pendientes y se impide que comiencen nuevas operaciones. Despues el gestor PnP envía una solicitud `stop` tras lo cual puede reconfigurar el dispositivo con otra solicitud `start-device`.

El gestor PnP también soporta otras solicitudes, como `query-remove`. Esta solicitud, que se utiliza cuando el usuario se está preparando para expulsar un dispositivo PCCARD, funciona de forma similar a `query-stop`. La solicitud `surprise-remove` se utiliza cuando un dispositivo falla o, más probablemente, un usuario extrae un dispositivo PCCARD sin detenerlo primero. La solicitud `remove` dice al controlador que deje de usar el dispositivo y que libere todos los recursos asignados al mismo.

Windows XP soporta sofisticados mecanismos de administración de energía. Aunque estas funcionalidades son útiles en los sistemas domésticos para reducir el consumo de energía, su principal aplicación es la facilidad de uso (un acceso más rápido) y la ampliación del tiempo de vida de las baterías en los equipos portátiles. El sistema y los dispositivos individuales pueden ponerse en modo de baja energía (denominado modo de suspensión o hibernación) cuando no se están usando, con lo que la batería se utilizará principalmente para retener los datos en la memoria física (RAM). El sistema puede volver a activarse automáticamente cuando se reciban paquetes a través de la red, cuando se produzca una llamada a través de una línea telefónica conectada a un módem, o cuando un usuario abra una computadora portátil o pulse un cierto botón de re-arranque del sistema. Windows XP también puede *hibernar* un sistema, almacenando el contenido de la memoria física en disco y apagando completamente la máquina, para posteriormente restaurar el sistema en algún instante posterior antes de continuar con la ejecución.

También hay disponibles otras estrategias para reducir el consumo de energía. En lugar de dejar que la CPU ejecute un bucle de inactividad en los momentos que no tenga otra cosa que hacer, Windows XP pone el sistema en un estado que requiere un menor consumo de energía, si la CPU se está infrautilizando. Windows XP reduce la velocidad de reloj de la CPU, lo que permite ahorrar una cantidad significativa de energía.

22.3.3.9 El Registro

Windows XP mantiene buena parte de su configuración en una base de datos interna conocida como el **Registro**. Una base de datos del Registro se denomina **colmena**. Existen colmenas separadas para la información del sistema, las preferencias predeterminadas del usuario, la instalación software y para la seguridad. Puesto que la información contenida en la **colmena del sistema** es necesaria para poder arrancar el sistema, el gestor del Registro se implementa como componente del programa ejecutivo. Cada vez que el sistema arranca correctamente, almacena la colmena del sistema como *última colmena correcta conocida*.

Si el usuario instala software, como por ejemplo un controlador de dispositivo, que genere una configuración de la colmena del sistema que no permita el arranque, el usuario puede normalmente arrancar utilizando la última configuración correcta conocida.

Los daños en la colmena del sistema debidos a la instalación de aplicaciones y controladores de otros fabricantes suelen ser tan comunes, que Windows XP tiene un componente denominado **Restaurar el sistema**, que guarda periódicamente las colmenas, además de otra información de estado del software, como por ejemplo los ejecutables de los controladores y los archivos de configuración, para poder restaurar el sistema a un estado correcto anterior en aquellos casos en los que el sistema arranque pero haya dejado de funcionar en la forma esperada.

22.3.3.10 Arranque

El arranque de un PC Windows XP comienza cuando se alimenta el hardware y el sistema BIOS comienza a ejecutarse a partir de la ROM. El sistema BIOS identifica el **dispositivo del sistema** a partir del cual hay que arrancar, y carga y ejecuta el cargador de arranque situado al inicio del disco. Este cargador conoce la información suficiente acerca del formato del sistema de archivos como para cargar el programa NTLDR del directorio raíz del dispositivo del sistema. NTLDR se utiliza para determinar qué **dispositivo de arranque** contiene el sistema operativo. A continuación, el NTLDR carga la biblioteca HAL, el *kernel* y la colmena del sistema desde el dispositivo de arranque. Gracias a la colmena del sistema determina qué controladores de dispositivo son necesarios para arrancar el sistema (los *controladores de arranque*) y los carga. Por último, NTLDR comienza la ejecución del *kernel*.

El *kernel* inicializa el sistema y crea dos procesos. El **proceso del sistema** contiene todas las hebras de trabajo internas y nunca se ejecuta en modo usuario. El primer proceso en modo usuario es el SMSS, que es similar al proceso INIT (initialización) de UNIX. SMSS realiza tareas adicionales de inicialización del sistema, incluyendo el establecimiento de los archivos de paginación y la carga de los controladores de dispositivo, y crea los procesos WINLOGON y CSRSS. CSRSS es el subsistema API Win32. WINLOGON se encarga de poner en marcha el resto del sistema, incluyendo el subsistema de seguridad LSASS y los restantes servicios necesarios para ejecutar el sistema.

El sistema optimiza el proceso de arranque precargando archivos desde el disco basándose en los arranques anteriores del sistema. Los patrones de acceso a disco durante el arranque también se utilizan para disponer los archivos del sistema en disco con el fin de reducir las operaciones de E/S necesarias. El número de procesos requeridos para arrancar el sistema se reduce agrupando los servicios dentro de un único proceso. Todas estas técnicas permiten reducir enormemente el tiempo de arranque del sistema. Por supuesto, el tiempo de arranque del sistema es menos importante que anteriormente debido a las capacidades de suspensión e hibernación de Windows XP, que permiten a los usuarios apagar sus computadoras y luego continuar rápidamente en el punto en que se habían quedado.

22.4 Subsistemas de entorno

Los subsistemas de entorno son procesos en modo usuario apilados sobre los servicios ejecutivos nativos de Windows XP y que permiten a Windows XP desarrollar programas desarrollados para otros sistemas operativos, incluyendo Windows de 16 bits, MS-DOS y POSIX. Cada subsistema de entorno proporciona un único entorno de aplicación.

Windows XP utiliza el subsistema API Win32 como el principal entorno de operación, por lo que es este subsistema el que arranca todos los procesos. Cuando se ejecuta una aplicación, el subsistema API Win32 invoca al gestor VM para cargar el código ejecutable de la aplicación. El gestor de memoria devuelve un código de estado a Win32 que indica el tipo de ejecutable. Si no es un ejecutable nativo API Win32, el entorno API Win32 comprueba si se está ejecutando el subsistema de entorno apropiado, si no es así, lo arranca como proceso en modo usuario. El subsistema toma entonces el control del arranque de la aplicación.

Los subsistemas de entorno utilizan la funcionalidad LPC para proporcionar servicios del sistema operativo a los procesos cliente. La arquitectura de subsistemas de Windows XP evita que las aplicaciones mezclen rutinas API de diferentes entornos. Por ejemplo, una aplicación API Win32 no puede hacer una llamada al sistema POSIX, porque sólo puede haber un subsistema de entorno asociado con cada proceso.

Puesto que cada subsistema se ejecuta como un proceso separado en modo usuario, un fallo catastrófico en uno de los subsistemas no tiene efectos sobre los procesos. La excepción es la API Win32, que proporciona todas las capacidades de interfaz con el teclado, el ratón y la pantalla gráfica. Si falla, el sistema se detiene y requiere un re-arranque.

El entorno API Win32 clasifica las aplicaciones como aplicaciones gráficas o basadas en caracteres, siendo una *aplicación basada en caracteres* aquella que cree que la salida interactiva va a una ventana (de comandos) basada en caracteres. La API Win32 transforma la salida de una aplicación basada en caracteres a una representación gráfica dentro de la ventana de comandos. Esta transformación resulta sencilla: cada vez que se invoca una rutina de salida, el subsistema de entorno llama a una rutina Win32 para mostrar el texto. Puesto que el entorno API Win32 realiza esta función para todas las ventanas basadas en caracteres, permite transferir texto de la pantalla entre unas ventanas y otras a través del portapapeles. Esta transformación funciona tanto para las aplicaciones MS-DOS como para las aplicaciones de línea de comandos POSIX.

22.4.1 Entorno MS-DOS

El entorno MS-DOS no tiene la complejidad de los otros subsistemas de entorno de Windows XP. Este entorno se proporciona mediante una aplicación API Win32 denominada **máquina DOS virtual** (VDM, virtual DOS machine). Puesto que la VDM es un proceso en modo usuario, se pagina y

se despacha como cualquier otra aplicación de Windows XP. La VDM tiene una **unidad de ejecución de instrucciones**, para ejecutar o emular instrucciones Intel 486. La VDM también proporciona rutina para emular el BIOS ROM MS-DOS y los servicios de la interrupción software “int 21” y disponen de controladores de dispositivo virtuales para la pantalla, el teclado y los puertos de comunicaciones. La VDM está basada en el código fuente de MS-DOS 5.0 y asigna al menos 620 KB de memoria a la aplicación.

La *shell* de comandos de Windows XP es un programa que crea una ventana que se asemeja a un entorno MS-DOS. Puede ejecutar programas tanto de 16 como de 32 bits. Cuando se ejecuta una aplicación MS-DOS, la *shell* de comandos arranca un proceso VDM para ejecutar el programa.

Si Windows XP está ejecutándose sobre un procesador compatible con IA32, las aplicaciones gráficas se ejecutan en modo de pantalla completa y las aplicaciones de caracteres pueden ejecutarse en modo de pantalla completa o en una ventana. No todas las aplicaciones MS-DOS se ejecutan sobre la VDM. Por ejemplo, algunas aplicaciones MS-DOS acceden directamente al hardware de disco, así que no podrán ejecutarse en Windows XP porque el acceso a disco está restringido con el fin de proteger el sistema de archivos. En general, las aplicaciones MS-DOS que acceden directamente al hardware no podrán funcionar en Windows XP.

Puesto que MS-DOS no es un entorno multitarea, algunas aplicaciones se han escrito de tal modo que “acaparan” la CPU. Por ejemplo, la utilización de bucles de espera puede provocar retardos o pausas en la ejecución. El planificador del despachador del *kernel* detecta dichos retardos y ajusta automáticamente el uso de la CPU, pero esto puede hacer que la aplicación funcione incorrectamente.

22.4.2 Entorno Windows de 16 bits

El entorno de ejecución Win16 se proporciona mediante una VDM que incorpora un software adicional denominado *Windows on Windows* (WOW32 para las aplicaciones de 16 bits); este software proporciona las rutinas del *kernel* Windows 3.1 y las rutinas de interfaz para las funciones de gestión de ventanas y de interfaz de dispositivo gráfico (GDI, graphical-device-interface). Las rutinas de interfaz invocan a las subrutinas apropiadas de la API Win32, convirtiendo las direcciones de 16 bits en direcciones de 32 bits. Las aplicaciones que utilizan la estructura interna del gestor de ventanas de 16 bits o de GDI, pueden no funcionar, porque la implementación subyacente de la API Win32 es, por supuesto, diferente del verdadero Windows de 16 bits.

WOW32 puede ejecutarse concurrentemente con otros procesos en Windows XP, pero se asemeja a Windows 3.1 en muchos aspectos. Sólo se puede ejecutar una aplicación Win16 en cada instante, todas las aplicaciones son monohebra y residen en el mismo espacio de direcciones y todas ellas comparten la misma cola de entrada. Estas características implican que una aplicación que detenga la recepción de datos de entrada bloqueará a todas las demás aplicaciones Win16, al igual que en Windows 3.x, y una aplicación Win16 puede hacer que sufran un fallo catastrófico otras aplicaciones Win16 corrompiendo el espacio de direcciones. Sin embargo, pueden coexistir múltiples entornos Win16, utilizando el comando *start /separate aplicaciónWin16* en la línea de comandos.

Son relativamente pocas las aplicaciones de 16 bits que los usuarios necesiten continuar ejecutando en Windows XP, pero algunas de ellas son, por ejemplo, programas comunes de instalación y configuración. Por ello, el entorno WOW32 continúa existiendo principalmente debido a que una serie de aplicaciones de 32 bits no pueden instalarse en Windows XP sin dicho entorno.

22.4.3 Entorno Windows de 32 bits en IA64

El entorno nativo para Windows en las máquinas compatibles con IA64 utiliza direcciones de 64 bits y el conjunto de instrucciones nativo IA64. Para ejecutar programas IA32 en este entorno, se requiere un nivel de traducción para transformar las llamadas a la API Win32 de 32 bits en las correspondientes llamadas de 64 bits, de la misma manera que las aplicaciones de 16 bits requieren una traducción en los sistemas IA32. Por ello, Windows de 64 bits soporta el entorno WOW64. Las implementaciones de 32 y 64 bits de Windows son esencialmente idénticas, y el procesador

IA64 proporciona un mecanismo de ejecución directa de las instrucciones IA32, por lo que WOW64 consigue un nivel mayor de compatibilidad que WOW32.

22.4.4 Entorno Win32

El subsistema principal en Windows XP es la API Win32. Ejecuta aplicaciones API Win32 y gestiona toda la E/S de teclado, ratón y pantalla. Puesto que es el entorno de control, está diseñado para ser extremadamente robusto, y a esta robustez contribuyen diversas características de la API Win32. A diferencia de los procesos en el entorno Win16, cada proceso Win32 tiene su propia cola de entrada. El gestor de ventanas despacha todas las entradas recibidas en el sistema a la cola de entrada del proceso apropiado, por lo que un proceso fallido no bloqueará la entrada a otros procesos.

El *kernel* de Windows XP también proporciona mecanismos multitarea apropiativos, lo que permite al usuario terminar las aplicaciones que hayan fallado o que ya no sean necesarias. La API Win32 también valida todos los objetos antes de utilizarlos, para evitar fallos catastróficos que podrían producirse si una aplicación tratara de emplear un descriptor no válido o incorrecto. El subsistema API Win32 verifica el tipo de objeto al que el descriptor apunta antes de utilizar dicho objeto. Los contadores de referencias que el gestor de objetos mantiene evitan que los objetos sean borrados mientras están siendo utilizados y evitan también que se los use después de haber sido borrados.

Para conseguir un alto grado de compatibilidad con los sistemas Windows 95/98, Windows XP permite a los usuarios especificar que ciertas aplicaciones individuales se ejecuten utilizando una **capa de modificación**, que modifica la API Win32 para aproximarse más exactamente al comportamiento esperado por las aplicaciones antiguas. Por ejemplo, algunas aplicaciones esperan ver una versión concreta del sistema y fallan en las nuevas versiones. Frecuentemente, las aplicaciones tienen errores latentes que quedan expuestos debido a los cambios en la implementación. Por ejemplo, la utilización de memoria después de haberla liberado puede provocar una corrupción sólo si cambia el orden de reutilización de la memoria por parte del cúmulo de memoria; asimismo una aplicación puede realizar suposiciones acerca de los errores que puedan ser devueltos por una rutina o acerca del número de bits válidos en una dirección. Al ejecutar una aplicación con la capa de modificación Windows 95/98 habilitada, el sistema proporciona un comportamiento mucho más próximo al de Windows 95/98, aunque a expensas de un rendimiento reducido y una interoperabilidad limitada con las otras aplicaciones.

22.4.5 Subsistema POSIX

El subsistema POSIX está diseñado para ejecutar aplicaciones POSIX escritas de acuerdo con el estándar POSIX, el cual está basado en el modelo UNIX. Las aplicaciones POSIX pueden ser iniciadas por el subsistema API Win32 o por otra aplicación POSIX. Las aplicaciones POSIX utilizan el servidor PSXSS .EXE del subsistema POSIX, la biblioteca de montaje dinámico POSIX PSXDLL .DLL y el gestor de sesión de consola POSIX, POSIX .EXE.

Aunque el estándar POSIX no especifica los temas de impresión, las aplicaciones POSIX pueden utilizar impresoras de forma transparente a través del mecanismo de redirección de Windows XP. Las aplicaciones POSIX tienen acceso a todos los sistemas de archivos existentes en el sistema Windows XP; el entorno POSIX impone una serie de permisos de tipo UNIX en los árboles de directorios.

Debido a cuestiones de planificación, el sistema POSIX en Windows XP no se incluye con el sistema, sino que se comercializa por separado para los servidores y sistemas de sobremesa profesionales. Proporciona un nivel mucho más alto de compatibilidad con las aplicaciones UNIX que las versiones anteriores de NT. La mayoría de las aplicaciones UNIX comúnmente disponibles se pueden compilar y ejecutar sin ningún cambio sobre la última versión de Interix.

22.4.6 Subsistemas de inicio de sesión y de seguridad

Antes de que un usuario pueda acceder a ningún objeto en Windows XP, dicho usuario debe ser autenticado por el servicio de inicio de sesión WINLOGON. WINLOGON se encarga de responder a

la secuencia de atención segura (Control-Alt-Supr.). La secuencia de atención segura es un mecanismo obligatorio para evitar que una aplicación pueda actuar como caballo de Troya. Sólo WINLOGON puede interceptar esta secuencia para mostrar una pantalla de inicio de sesión, modificar contraseñas y bloquear la estación de trabajo. Para poder ser autenticado, un usuario debe disponer de una cuenta y proporcionar la contraseña correspondiente. Alternativamente, el usuario puede iniciar una sesión empleando una tarjeta inteligente y un número de identificación personal, de acuerdo con las políticas de seguridad (o directivas de seguridad de acuerdo con la terminología de Microsoft) que se estén aplicando dentro del dominio.

El subsistema local de autoridad de seguridad (LSASS, local security authority subsystem) es el proceso que genera los testigos de acceso que representan a los usuarios en el sistema. Este subsistema invoca un paquete de autenticación para llevar a cabo la autenticación utilizando la información procedente del subsistema de inicio de sesión o del servidor de red. Normalmente, el paquete de autenticación simplemente consulta la información de cuentas en una base de datos local y comprueba si la contraseña es correcta. A continuación, el subsistema de seguridad genera el testigo de acceso para el ID de usuario, que contendrá los privilegios apropiados, los límites de cuota y los identificadores de grupo. Cuando el usuario trate de acceder a un objeto en el sistema, como por ejemplo, abriendo un descriptor del objeto, se pasa el testigo de acceso al monitor de referencia de seguridad, que comprueba los privilegios y cuotas. El paquete de autenticación predeterminado para los dominios Windows XP es Kerberos. LSASS también se encarga de implementar políticas de seguridad tales como las contraseñas fuertes, es responsable de autenticar a los usuarios y se encarga de realizar el cifrado de los datos y las claves.

22.5 Sistema de archivos

Históricamente, los sistemas MS-DOS han utilizado el sistema de archivos FAT (FAT, file-allocation table, tabla de asignación de archivos). El sistema de archivos FAT de 16 bits tiene diversas desventajas, incluyendo el fenómeno de la fragmentación interna, un límite de tamaño de 2 GB y una falta de protección de acceso para los archivos. El sistema de archivos FAT de 32 bits resolvió los problemas de tamaño y de fragmentación, pero su rendimiento y sus características siguen siendo pobres por comparación con los sistemas de archivos modernos. El sistema NTFS es mucho mejor. Fue diseñado para incluir numerosas funciones, incluyendo mecanismos de recuperación de datos, seguridad, tolerancia a fallos, soporte de sistemas de archivos y archivos de gran tamaño, múltiples flujos de datos, nombres UNICODE, archivos dispersos, cifrado, mecanismo de diafragma, copias de volúmenes ocultas y compresión de archivos.

Windows XP utiliza NTFS como su sistema de archivos básico, así que nos vamos a centrar aquí en ese sistema de archivos. Sin embargo, Windows XP continúa utilizando FAT16 para leer discos duros y otros soportes extraíbles. Y, a pesar de las ventajas de NTFS, FAT32 continúa siendo importante para la interoperabilidad de soportes físicos con sistemas Windows 95/98. Windows XP permite emplear tipos de sistemas de archivos adicionales para los formatos comunes utilizados en los soportes CD y DVD.

22.5.1 Estructura interna de NTFS

La entidad fundamental en NTFS es el volumen. Los volúmenes se crean con la utilidad de gestión de discos lógicos de Windows XP y están basados en una partición de disco lógico. Un volumen puede ocupar una parte de un disco, un disco completo o incluso varios discos.

NTFS no trata con los sectores individuales del disco, sino que utiliza *clusters* como unidades de asignación de disco. Un *cluster* es un conjunto de sectores de disco, cuyo número es una potencia de 2. El tamaño de *cluster* se configura en el momento de formatear el sistema de archivos NTFS. El tamaño de *cluster* predeterminado es igual al tamaño de sector para los volúmenes de hasta 512 MB, es igual a 1 KB para volúmenes de hasta 1 GB, 2 KB para volúmenes de hasta 2 GB y 4 KB para los volúmenes de mayor tamaño. Este tamaño de *cluster* es mucho menor que el del sistema de archivos FAT de 16 bits, y este pequeño tamaño reduce la cantidad de fragmentación interna. Como ejemplo, considere un disco de 1,6 GB con 16.000 archivos. Si utilizamos un sistema de

archivos FAT-16, se podrían perder 400 MB debido a la fragmentación interna por el tamaño de *cluster* es de 32 KB. En NTFS, sólo se perderían 17 MB al almacenar los mismos archivos.

NTFS utiliza **números lógicos de cluster** (LCN, logical cluster numbers) como direcciones de disco. El sistema asigna esos identificadores numerando los *clusters* desde el comienzo del disco hasta el final, de manera secuencial. Utilizando este esquema, el sistema puede calcular el desplazamiento en disco físico (en bytes) multiplicando el LCN por el tamaño de *cluster*.

Un archivo en NTFS no es un simple flujo de bytes como en MS-DOS o UNIX; en lugar de ello, es un objeto estructurado compuesto de **atributos** con tipo. Cada atributo de un archivo es un flujo de bytes independiente que puede ser creado, borrado, leído y escrito. Algunos tipos de atributo son estándar para todos los archivos, incluyendo el nombre de archivo (o nombres, si el archivo tiene alias, como por ejemplo un nombre corto MS-DOS), la fecha de creación y el descriptor de seguridad que especifica el control de acceso. Los datos de usuario se almacenan en los **atributos de datos**.

La mayoría de los archivos de datos tradicionales tienen un atributo de datos *sin nombre* que contiene todos los datos del archivo. Sin embargo, pueden crearse flujos de datos adicionales con nombres explícitos. Por ejemplo, en archivos Macintosh almacenados en un servidor Windows XP, el subarchivo de recursos es un flujo de datos nominado. Las interfaces IProp del modelo COM (Component Object Model, modelo de objetos componentes) utiliza un flujo de datos nominado para almacenar propiedades de los archivos normales, incluyendo miniaturas de las imágenes. En general, pueden añadirse atributos según sea necesario y a esos atributos se accede utilizando una sintaxis *nombre_archivo:atributo*. NTFS devuelve el tamaño del atributo sin nombre sólo en respuesta a las operaciones de consulta de archivo, como por ejemplo cuando se ejecuta el comando `dir`.

Todos los archivos en NTFS se describen mediante uno o más registros dentro de una matriz almacenada en un archivo especial denominado tabla maestra de archivos (MFT, master file table). El tamaño de un registro se determina cuando se crea el sistema de archivos y puede ir de 1 a 4 KB. Los atributos de pequeño tamaño se almacenan en el propio registro de la MFT y se denominan **atributos residentes**. Los atributos de mayor tamaño, como por ejemplo el cuerpo de los datos sin nombre, se denominan **atributos no residentes** y se almacenan en una o más **extensiones contiguas** del disco; en el registro MFT se almacena un puntero a cada extensión. Para un archivo de pequeño tamaño, incluso el propio atributo de datos puede caber dentro del registro MFT. Si un archivo tiene muchos atributos o si está altamente fragmentado (de modo que se necesitan muchos punteros para apuntar a todos los fragmentos), puede que un sólo registro de la MFT no sea lo suficientemente grande. En este caso, el archivo se describe mediante un registro denominado **registro base del archivo**, que contiene punteros a una serie de registros de desbordamiento que almacenan los atributos y punteros adicionales.

Cada archivo de un volumen NTFS tiene un ID único denominado **referencia de archivo**. La referencia de archivo es un valor de 64 bits compuesto por un número de archivos de 48 bits y de un número de secuencia de 16 bits. El número de archivo es el número de registro (es decir, la posición en la matriz) dentro de la MFT que describe el archivo. El número de secuencia se incrementa cada vez que se reutiliza una entrada de la MFT. El número de secuencia permite a NTFS realizar comprobaciones internas de coherencia, como por ejemplo detectar una referencia obsoleta a un archivo borrado después de que la correspondiente entrada a la MFT haya sido reutilizada para un nuevo archivo.

22.5.1.1 Árbol B+ de NTFS

Al igual que en MS-DOS y UNIX, el espacio de nombres de NTFS está organizado como una jerarquía de directorios. Cada directorio utiliza una estructura de datos denominada **árbol B+** para almacenar un índice de los nombres de archivo contenidos en ese directorio. Se utiliza un árbol B+ porque elimina el coste de reorganizar el árbol y tiene la propiedad de que la longitud de cada ruta desde la raíz del árbol hasta un nodo hoja es idéntica. La **raíz del índice** de un directorio contiene el nivel superior del árbol B+. Para un directorio de gran tamaño, este nivel superior contendrá punteros a extensiones del disco donde se almacena el resto del árbol. Cada entrada del directorio contiene el nombre y la referencia del archivo, así como una copia de la marca tempo-

ral de la última actualización y del tamaño del archivo, valores ambos tomados de los atributos del archivo residentes en la MFT. En el directorio se almacenan copias de esta información, para poder generar de manera eficiente listados de directorio. Puesto que todos los nombres de archivo, tamaños y fechas de actualización están disponibles en el propio directorio, no hay necesidad de extraer estos atributos de las entradas MFT de cada uno de los archivos.

22.5.1.2 Metadatos NTFS

Los metadatos del volumen NTFS se almacenan en archivos. El primero de estos archivos es la MFT. El segundo archivo, que se utiliza durante la recuperación si resulta dañada la MFT, contiene una copia de las primeras 16 entradas de la MFT. Los siguientes archivos también son de propósito especial e incluyen el archivo de registro, el archivo de volumen, la tabla de definición de atributos, el directorio raíz, el archivo de mapa de bits, el archivo de arranque y el archivo de *clusters* erróneos. A continuación se describe el papel de cada uno de estos archivos.

- El **archivo de registro** almacena todas las actualizaciones de metadatos realizadas en el archivo.
- El **archivo de volumen** contiene el nombre del volumen, la versión de NTFS que ha formateado el volumen y un bit que indica si el volumen puede haber resultado corrompido y necesita comprobar su coherencia.
- La **tabla de definición de atributos** indica qué tipos de atributos se usan en el volumen y qué operaciones pueden realizarse con cada uno de ellos.
- El **directorío raíz** es el directorio del nivel superior dentro de la jerarquía del sistema de archivos.
- El **archivo de mapa de bits** indica qué *clusters* del volumen están asignados a archivos y cuáles otros están libres.
- El **archivo de arranque** contiene el código de arranque de Windows XP y debe estar ubicado en una dirección concreta del disco, para que pueda ser localizado fácilmente por un cargador de arranque simple almacenado en ROM. El archivo de arranque también contiene la dirección física de la MFT.
- El **archivo de clusters erróneos** registra las áreas incorrectas que el volumen pueda contener; NTFS utiliza este registro para propósitos de recuperación de errores.

22.5.2 Recuperación

En muchos sistemas de archivos simples, un fallo de alimentación en el instante incorrecto puede dañar las estructuras de datos del sistema de archivos tan gravemente que el volumen completo resulta corrompido. Muchas versiones de UNIX almacenan metadatos redundantes en el disco, y se recuperan de los fallos catastróficos utilizando el programa `fsck` para comprobar todas las estructuras de datos del sistema de archivos y restaurarlas obligatoriamente a un estado coherente. La restauración de estas estructuras implica a menudo borrar los archivos dañados y liberar *clusters* de datos que hubieran sido escritos con datos del usuario pero no hubieran sido registrados adecuadamente dentro de las estructuras de metadatos del sistema de archivos. Esta comprobación puede ser un proceso muy lento y puede hacer que se pierdan cantidades significativas de datos.

NTFS adopta una técnica diferente para garantizar la robustez del sistema de archivos. En NTFS, todas las actualizaciones de las estructuras de datos del sistema de archivos se realizan dentro de transacciones. Antes de modificar una estructura de datos, la transacción escribe una entrada de registro que contiene información de rehacer y deshacer; después de haber modificado las estructuras de datos, la transacción escribe una entrada de confirmación en el registro para indicar que la transacción ha tenido éxito.

Después de un fallo catastrófico, el sistema puede restaurar las estructuras de datos del sistema de archivos a un estado coherente procesando las entradas del registro, primero rehaciendo las operaciones de las transacciones confirmadas y luego deshaciendo las operaciones de las transacciones que no hubieran sido confirmadas adecuadamente antes del fallo catastrófico. Periódicamente (usualmente cada cinco segundos), se escribe en el registro un punto de comprobación. El sistema no necesita las entradas de registro previas al punto de comprobación para recuperarse de un fallo catastrófico. Dichas entradas pueden descartarse, con lo que el archivo de registro no crecerá de forma ilimitada. La primera vez en que se accede a un volumen NTFS después del arranque del sistema, NTFS lleva a cabo automáticamente una recuperación del sistema de archivos.

Este esquema no garantiza que todos los contenidos de los archivos de usuario sean correctos después de un fallo catastrófico; sólo garantiza que las estructuras de datos del sistema de archivos (los archivos de metadatos no estén dañados y reflejen un cierto estado coherente que existía antes del fallo catastrófico). Teóricamente, resultaría posible ampliar este esquema basado en transacciones para cubrir también a los archivos de usuario, y puede que Microsoft lo haga en el futuro.

El registro está almacenado en el tercio de los archivos de metadatos colocado, al principio del volumen. Se crea con un tamaño fijo máximo en el momento de formatear el sistema de archivos. El registro tiene dos secciones: el **área de registro**, que es una cola circular de entradas de registro y el **área de reinicio**, que almacena información de contexto, como por ejemplo la posición dentro del área de registro donde NTFS debe comenzar a leer durante una recuperación. De hecho, el área de reinicio alberga dos copias de su información, por lo que sigue siendo posible la recuperación si una de las copias resulta dañada durante el fallo.

La funcionalidad de registro es proporcionada por el **servicio del archivo de registro** de Windows XP. Además de escribir las entradas de registro y efectuar acciones de recuperación, el servicio del archivo de registro controla el espacio libre existente en dicho archivo. Si el espacio libre llega a ser demasiado bajo, el servicio del archivo de registro pone en cola las transacciones pendientes y NTFS detiene todas las nuevas operaciones de E/S. Después de que se completen las operaciones que estuvieran en curso, NTFS invoca al gestor de caché para volcar en disco todos los datos y luego reinicia el archivo de registro y aplica las transacciones que estuvieran en cola.

22.5.3 Seguridad

La seguridad de un volumen NTFS se deriva del modelo de objetos de Windows XP. Cada archivo NTFS hace referencia a un descriptor de seguridad, que contiene el testigo de acceso del propietario del archivo y una lista de control de acceso, que indica los privilegios de acceso concedidos a cada usuario que tenga acceso al archivo.

En operación normal, NTFS no impone permisos durante el recorrido de los directorios contenidos en los nombres de ruta de los archivos. Sin embargo, por compatibilidad con POSIX, pueden habilitarse estas comprobaciones. Las comprobaciones de recorrido son inherentemente más costosas, ya que el análisis sintáctico moderno de los nombres de ruta de los archivos utiliza técnicas de detección de prefijos, en lugar de realizar una apertura componente a componente de los nombres de directorio.

22.5.4 Gestión de volúmenes y tolerancia a fallos

FtDisk es el controlador de disco tolerante a fallos de Windows XP. Cuando se instala, proporciona diversas maneras de combinar múltiples unidades de disco en un único volumen lógico, para mejorar las prestaciones, la capacidad o la fiabilidad.

22.5.4.1 Conjunto de volúmenes

Una forma de combinar múltiples disco consiste en concatenarlos lógicamente para formar un volumen lógico de gran tamaño, como se muestra en la Figura 22.7. En Windows XP, este

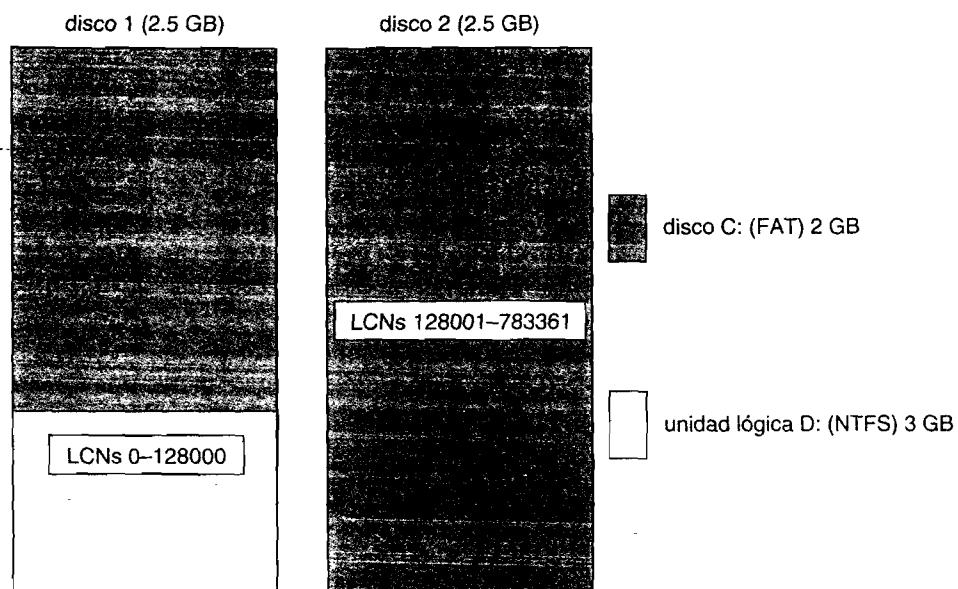


Figura 22.7 Conjunto de volúmenes con dos unidades.

volumen lógico, denominado **conjunto de volúmenes**, puede estar compuesto de hasta 32 particiones físicas. Un conjunto de volúmenes que contenga un volumen NTFS puede ampliarse sin perturbar los datos que ya estén almacenados en el sistema de archivos. Los metadatos de mapa de bits contenidos en el volumen NTFS se amplían simplemente para cubrir el espacio recién añadido. NTFS continúa utilizando el mismo mecanismo LCN que utiliza para un único disco físico, y el FtDisk se encarga de traducir cada desplazamiento dentro del volumen lógico al correspondiente desplazamiento dentro de un disco concreto.

22.5.4.2 Conjunto de distribución en bandas

Otra forma de combinar múltiples particiones físicas consiste en entrelazar sus bloques por turnos con el fin de formar lo que denomina **conjunto de distribución en bandas**, como se muestra en la Figura 22.8. Este esquema también se denomina RAID nivel 0 o **bandas de disco**. FtDisk utiliza una tamaña de banda de 64 KB: los primeros 64 KB del volumen lógico se almacenan en la primera partición física, los segundos 64 KB se almacenan en la segunda partición física, y así sucesivamente, hasta que cada una de las particiones ha proporcionado 64 KB de espacio. Después, la asignación comienza de nuevo por el primer disco, asignando el segundo bloque de

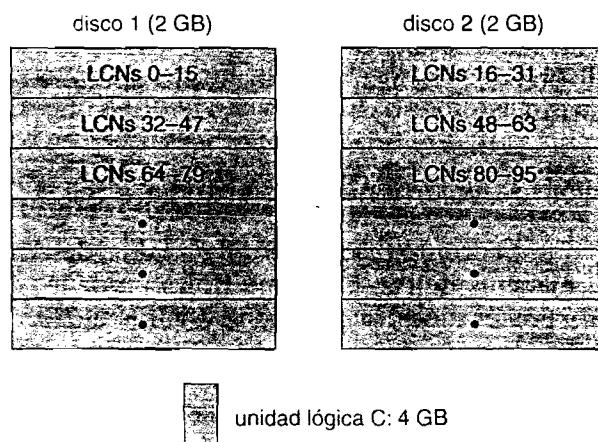


Figura 22.8 Conjunto de distribución en bandas con dos discos.

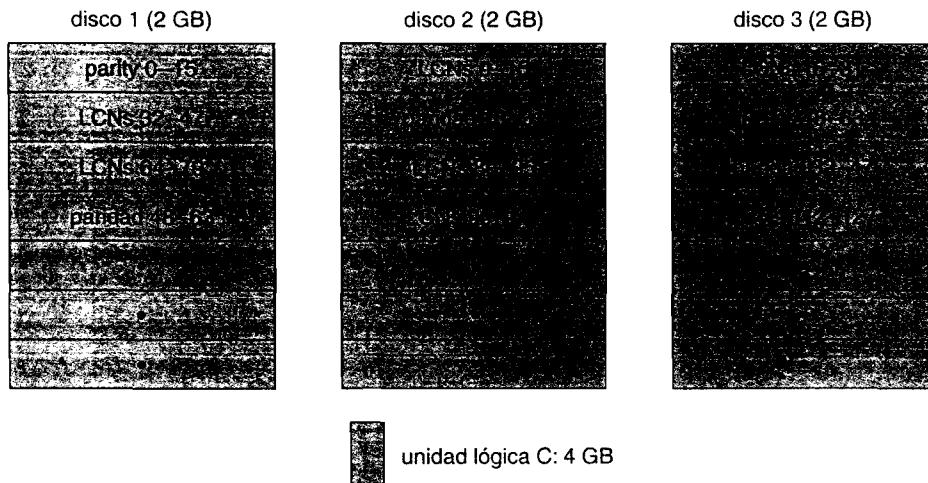


Figura 22.9 Conjunto de distribución en bandas con paridad, utilizando tres discos.

64 KB. Un conjunto de distribución en bandas forma un único volumen de gran tamaño, pero esa peculiar distribución física permite mejorar el ancho de banda de E/S, porque para una operación de E/S de gran tamaño, todos los discos pueden transferir sus datos en paralelo.

22.5.4.3 Conjunto de distribución en bandas con paridad

Una variante de esta idea es el denominado **conjunto de distribución en bandas con paridad**, que se muestra en Figura 22.9. Este esquema también se denomina RAID nivel 5. Suponga que un conjunto de distribución en banda tuviera ocho discos. Siete de estos discos almacenarían bandas de datos, estando cada banda almacenada en un disco y el octavo disco almacenaría una banda de paridad para cada banda de datos. La banda de paridad contiene el resultado de aplicar la operación `exclusive or` byte a byte a las otras bandas de datos. Si alguna de las ocho bandas resultara destruida, el sistema puede reconstruir los datos calculando el resultado de la operación `exclusive or` de las siete restantes. Esta capacidad de reconstruir los datos hace que la matriz de discos tenga una probabilidad mucho menor de perder datos en caso de que se produzca un fallo de disco.

Observe que una actualización de una de las bandas de datos también requiere volver a calcular la banda de datos. Por tanto, siete escrituras concurrentes en siete diferentes bandas de datos requerirían también actualizar siete bandas de paridad. Si las bandas de paridad estuvieran todas en el mismo disco, dicho disco podría tener una carga de E/S siete veces mayor que la de los discos de datos. Para evitar crear este cuello de botella, distribuimos las bandas de paridad entre todos los discos, asignándolas por turnos a cada uno de ellos. Para construir un conjunto de distribución en bandas con paridad, necesitamos un mínimo de tres particiones de igual tamaño localizadas en tres discos separados.

22.5.4.4 Espejo de disco

Un esquema aún más robusto es el que se denomina **espejo de disco** o RAID nivel 1, la Figura 22.10 ilustra este esquema. Un **conjunto espejo** comprende dos particiones de igual tamaño situadas en dos discos distintos. Cuando una aplicación escribe datos en un conjunto espejo, FtDisk escribe los datos en ambas particiones, de modo que el contenido de las dos particiones será idéntico. Si una de las particiones falla, FtDisk dispone de otra copia almacenada en el espejo. Los conjuntos espejo también pueden mejorar las prestaciones, porque las solicitudes de lectura pueden distribuirse entre las dos copias en espejo, asignando a cada una de las copias la mitad de la carga de trabajo. Para proteger frente al fallo de una controladora de disco podemos conectar los dos dis-

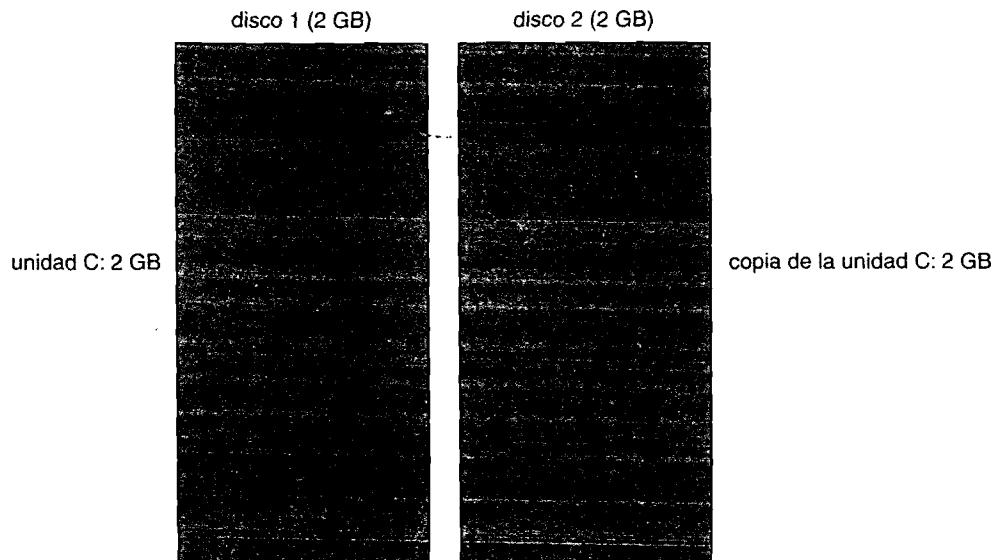


Figura 2.10 Conjunto espejo con dos unidades de disco.

cos de un conjunto espejo a dos controladoras de disco separadas. Esta disposición se denomina **conjunto dúplex**.

22.5.4.5 Sectores de reserva y reasignación de *clusters*

Para resolver el problema de los fallos de los sectores de disco, FtDisk utiliza una técnica hardware denominada de sectores de reserva y NTFS emplea una técnica denominada reasignación de *clusters*. Los **sectores de reserva** es una capacidad hardware proporcionada por muchas unidades de disco. Cuando se formatea una unidad de disco, se crea un mapa que asigna los números de bloques lógicos a los sectores correctos del disco; durante esta operación se dejan también sectores adicionales sin asignar, como reserva. Si falla un sector, FtDisk ordena a la unidad de disco que lo sustituya por uno de los sectores de reserva. La **reasignación de *clusters*** es una técnica software realizada por el sistema de archivos. Si falla un bloque del disco, NTFS lo sustituye por un bloque diferente no asignado, modificando los punteros afectados en la MFT. NTFS también anota que ese bloque erróneo nunca debe volver a ser asignado a ningún archivo.

Cuando un bloque de disco se vuelve defectuoso, el resultado usual es una pérdida de datos. Pero pueden combinarse las técnicas de sectores de reserva o de asignación de *clusters* con volúmenes tolerantes a fallos con el fin de que el usuario no perciba ese fallo de un bloque de disco. Si falla una operación de lectura, el sistema reconstruye los datos que faltan leyendo la copia en espejo o calculando el valor de paridad exclusive or en un conjunto de distribución en bandas con paridad. Los datos reconstruidos se almacenan en una nueva ubicación que se obtiene aplicando las técnicas de sectores de reserva o de reasignación de *clusters*.

22.5.5 Compresión y cifrado

NTFS puede comprimir los datos de archivos individuales o todos los archivos de datos de un directorio. Para comprimir un archivo, NTFS divide los datos del archivo en **unidades de compresión**, que son bloques de 16 *clusters* contiguos. Cuando se escribe cada unidad de compresión, se aplica un algoritmo de compresión de datos. Si el resultado cabe en menos de 16 *clusters*, se almacena la versión comprimida. A la hora de leer, NTFS puede determinar si los datos han sido comprimidos. En caso afirmativo, la longitud de la unidad de compresión almacenada será inferior a 16 *clusters*. Para mejorar la velocidad a la hora de leer unidades de compresión contiguas, NTFS extrae los datos anticipadamente y los descomprime, adelantándose a las solicitudes de la aplicación.

Para los archivos dispersos o para los archivos que contengan fundamentalmente ceros, NTFS emplea otra técnica para ahorrar espacio. Los *clusters* que sólo contienen ceros porque nunca se ha escrito en ellos, no están realmente asignados ni se los almacena en disco. En lugar de ello, se dejan huecos en la secuencia de números de *cluster* virtuales almacenada en la entrada MFT del archivo. A la hora de leer el archivo, si el sistema encuentra un hueco en los números de cluster virtuales, NTFS se limita a llenar de ceros esa parte del búfer del proceso que haya realizado la invocación. Esta técnica también se emplea en el sistema operativo UNIX.

NTFS permite el cifrado de archivos, pudiéndose especificar que se cifren archivos individuales o directorios completos. El sistema de seguridad gestiona las claves utilizadas y hay disponible un servicio de recuperación de claves para recuperar las claves perdidas.

22.5.6 Puntos de montaje

Los puntos de montaje son un tipo de enlace simbólico específicos de los directorios de NTFS. Proporcionan un mecanismo para que los administradores organicen los volúmenes de disco de una forma más flexible que utilizando nombres globales (como por ejemplo las letras de unidad). Los puntos de montaje se implementan como un enlace simbólico con una serie de datos asociados que contienen el verdadero nombre del volumen. En el futuro, los puntos de montaje sustituirán completamente a las letras de unidad, aunque existirá un período de transición largo, debido a la dependencia que muchas aplicaciones tienen del esquema de letras de unidad.

22.5.7 Diario de cambios

NTFS mantiene un diario que describe todos los cambios que se han realizado en el sistema de archivos. Los servicios de modo usuario pueden recibir notificaciones de los cambios sufridos por el diario e identificar así qué archivos han sido modificados. El servicio de indexación de contenidos utiliza el diario de cambios para identificar los archivos que hay que volver a indexar. El servicio de replicación de archivos lo utiliza para identificar los archivos que deben replicarse a través de la red.

22.5.8 Copias ocultas de volúmenes

Windows XP implementa la capacidad de poner un volumen en un estado conocido y luego crear una copia oculta que puede utilizarse para disponer de una reserva con una vista coherente del volumen. La realización de una copia oculta de un volumen es un tipo de mecanismo de copia durante la escritura, en el que los bloques modificados después de haber creado la copia oculta se añaden a la copia. Para conseguir que el bloque esté en un estado coherente se requiere que las aplicaciones cooperen, ya que el sistema no puede saber cuándo los datos utilizados por una aplicación se encuentran en un estado estable a partir del cual pueda reiniciarse la aplicación de forma segura.

La versión de servidor de Windows XP utiliza copias ocultas para mantener de manera eficiente versiones antiguas de los archivos almacenados en los servidores de archivos. Esto permite a los usuarios ver documentos almacenados en los servidores de archivos en el estado que tenían en algún instante anterior. El usuario puede utilizar esta característica para recuperar archivos que accidentalmente se hayan borrado, o simplemente para examinar versiones anteriores del archivo, todo ello sin necesidad de introducir una cinta con una copia de seguridad.

22.6 Conexión de red

Windows XP permite utilizar tanto redes entre iguales (peer-to-peer) como redes cliente-servidor. También dispone de funcionalidades para la gestión de red. Los componentes de red de Windows XP proporcionan mecanismos de transporte de datos, comunicación interprocesos, compartición de archivos a través de la red y envío de trabajos a impresoras remotas.

22.6.1 Interfaces de red

Para describir los mecanismos de comunicación por red en Windows XP, en primer lugar debemos mencionar dos de las interfaces internas de comunicación por red: la **especificación de interfaz de dispositivo de red** (NDIS, network device interface specification) y la **interfaz de controlador de transporte** (TDI, transport driver interface). La interfaz NDIS fue desarrollada en 1989 por Microsoft y 3Com para separar los adaptadores de red de los protocolos de transporte de modo que cualquiera de ellos pudiera modificarse sin afectar al otro. NDIS reside en la interfaz entre los niveles de control de enlace de datos y de control de acceso al medio dentro del modelo OSI y permite que diversos protocolos operen sobre muchos adaptadores de red diferentes. En términos del modelo OSI, la TDI es la interfaz entre el nivel de transporte (nivel 4) y el nivel de sesión (nivel 5). Esta interfaz permite que cualquier componente del nivel de sesión utilice cualquiera de los mecanismos de transporte disponibles (un razonamiento similar fue el que condujo al mecanismo de *streams* en UNIX). La TDI soporta mecanismos de transportes tanto orientados a conexión como sin conexión y dispone de funciones para enviar cualquier tipo de datos.

22.6.2 Protocolos

Windows XP implementa los protocolos de transporte como controladores. Estos controladores pueden cargarse y descargarse en el sistema dinámicamente, aunque en la práctica suele ser necesario reiniciar el sistema después de un cambio. Windows XP se suministra con diversos protocolos de comunicación por red. A continuación, vamos a analizar algunos de los protocolos utilizados en Windows XP para proporcionar diversas funcionalidades de comunicación por red.

22.6.2.1 Protocolo SMB

El **protocolo SMB** (server-message-block, bloque de mensajes de servidor) fue introducido por primera vez en MS-DOS 3.1. El sistema emplea el protocolo para enviar solicitudes de E/S a través de la red. El protocolo SMB utiliza cuatro tipos de mensajes. Los mensajes Session control (control de sesión) son comandos que inician y terminan una conexión de redirección como un recurso compartido situado en el servidor. Un redirector usa mensajes File (archivo) para acceder a los archivos del servidor. El sistema utiliza mensajes Printer (impresora) para enviar datos a una cola de impresión remota y para recibir información de estado y el mensaje Message se emplea para comunicarse con otras estaciones de trabajo. El protocolo SMB fue publicado con el nombre de **Common Internet File System** (CIFS) y está soportado por diversos sistemas operativos.

22.6.2.2 Protocolo NetBIOS

El **protocolo NetBIOS** (network basic input/output system, sistema básico de entrada/salida de red) es una interfaz de abstracción hardware para redes, análoga a la interfaz de abstracción hardware BIOS desarrollada para las máquinas de tipo PC que ejecutan MS-DOS. NetBIOS, desarrollado a principios de la década de los años 80, se han convertido en una interfaz estándar de programación por red. NetBIOS se utiliza para establecer nombres lógicos en la red, para establecer conexiones lógicas, o **sesiones**, entre dos nombres lógicos de la red y para permitir la transferencia fiable de datos en una sesión mediante solicitudes NetBIOS o SMB.

22.6.2.3 Protocolo NetBEUI

El **protocolo NetBEUI** (NetBIOS extended user interface, interfaz de usuario extendida NetBIOS) fue introducido por IBM en 1985 como protocolo simple y eficiente de comunicación para redes de hasta 254 máquinas. Es el protocolo predeterminado para las redes entre iguales Windows 95 y para Windows para Grupos de trabajo. Windows XP utiliza NetBEUI cuando quiere compartir recursos con estas redes. Entre las limitaciones de NetBEUI podemos citar que utiliza el nombre real de las computadoras como dirección de las mismas y que no soporta el encaminamiento de los datos.

22.6.2.4 Protocolo TCP/IP

El **conjunto de protocolos TCP/IP** (transmission control protocol/Internet protocol, protocolo de control de transmisión/protocolo Internet) que se utiliza en Internet, ha llegado a convertirse en un estándar de facto para las estructuras de comunicación por red. Windows XP emplea TCP/IP para conectarse con una amplia variedad de sistemas operativos y plataformas hardware. El paquete TCP/IP de Windows incluye el protocolo SNM (simple network-management, gestión simple de red), el protocolo DHCP (dynamic host-configuration protocol, protocolo dinámico de configuración de host), el servicio WINS (Windows Internet name service, servicio de nombres Internet de Windows) y NetBIOS.

22.6.2.5 Protocolo PPTP

El **protocolo PPTP** (point-to-point tunneling protocol, protocolo de túnel punto a punto) es un protocolo proporcionado por Windows XP para poder comunicar módulos de servidor de acceso remoto que se ejecuten en máquinas servidor Windows XP con otros sistemas cliente que esté conectados a través de Internet. Los servidores de acceso remoto pueden cifrar los datos enviados a través de la conexión y soportan **redes privadas virtuales** (VPN, virtual private network) multiproocolo a través de Internet.

22.6.2.6 Protocolos Novell NetWare

Los protocolos Novell NetWare (servicios de datagramas IPX sobre el nivel de transporte SPX) se utilizan ampliamente en las redes LAN de tipo PC. El protocolo NWLink de Windows XP conecta NetBIOS con las redes NetWare. En combinación con un redirector (como por ejemplo Client Service for NetWare de Microsoft o NetWare Client for Windows de Novell), este protocolo permite a un cliente Windows XP conectarse con un servidor NetWare.

22.6.2.7 Protocolo WebDAV

El protocolo WebDAV (Web distributed authoring and versioning, autoría y versionado distribuidos a través de Web) es un protocolo basado en http para la autoría en colaboración a través de la red. Windows XP incluye un redirector WebDAV dentro del sistema de archivos. Al incluir el WebDAV directamente dentro del sistema de archivos, puede integrarse con otras funcionalidades, como la de cifrado. De este modo, los archivos personales pueden almacenarse con seguridad en espacios públicos de la red.

22.6.2.8 Protocolo AppleTalk

El protocolo **AppleTalk** fue diseñado por Apple como conexión de bajo coste para permitir a las computadoras Macintosh compartir archivos. Los sistemas Windows XP pueden compartir archivos e impresoras con computadoras Macintosh a través de AppleTalk si hay un servidor Windows XP en la red que esté ejecutando el paquete de servicios de Windows para Macintosh.

22.6.3 Mecanismos de procesamiento distribuido

Aunque Windows XP no es un sistema operativo distribuido, sí que permite utilizar aplicaciones distribuidas. Entre los mecanismos que dan soporte al procesamiento distribuido en Windows XP podemos citar NetBIOS, las *pipes* con nombre y los buzones de correo, los *sockets* de Windows, el mecanismo RPC, el lenguaje de definición de interfaces de Microsoft y, finalmente, COM.

22.6.3.1 NetBIOS

En Windows XP, las aplicaciones NetBIOS pueden comunicarse a través de la red utilizando NetBEUI, NWLink o TCP/IP.

22.6.3.2 Pipes con nombre

Las *pipes* con nombre son un mecanismo de mensajería orientados a conexión. Las *pipes* con nombre fueron originalmente desarrolladas como interfaz de alto nivel con las conexiones NetBIOS a través de la red. Un proceso también puede utilizar *pipes* con nombre para comunicarse con otros procesos situados en la misma máquina. Puesto que a las *pipes* con nombre se accede a través de la interfaz del sistema de archivos, los mecanismos de seguridad usados para los objetos archivos también se aplican a las *pipes* con nombre.

El nombre de una *pipe* con nombre tiene un formato denominado **convenio uniforme de nombrado** (UNC, uniform naming convention). Un nombre UNC se asemeja a un nombre típico de un archivo remoto. El formato de un nombre UNC es \\nombre_servidor\nombre_recurso \x\y\z, donde *nombre_servidor* identifica a un servidor de la red; *nombre_recurso* identifica cualquier recurso que esté a disposición de los usuarios de la red, como por ejemplo un directorio, un archivo, una *pipe* con nombre o una impresoras, y la parte \x\y\z es un nombre de ruta normal de un archivo.

22.6.3.3 Buzones de correo

Los **buzones de correo** son un mecanismo de mensajería sin conexión. No son fiables cuando se accede a ellos a través de la red, en el sentido de que un mensaje enviado a un buzón de correo puede perderse antes de que el receptor llegue a leerlo. Los buzones de correo se emplean para aplicaciones de difusión, como por ejemplo para localizar componentes en la red y también son usados por el servicio explorador de la computadora Windows.

22.6.3.4 Winsock

Winsock es la API de *sockets* de Windows XP. Winsock es una interfaz de nivel de sesión bastante compatible con los *sockets* de UNIX, aunque tiene algunas extensiones propias de Windows XP. Proporciona una interfaz estándar con muchos protocolos de transporte que pueden tener diferentes esquemas de direccionamiento, de modo que cualquier aplicación Winsock puede ejecutarse sobre cualquier pila de protocolos compatible con Winsock.

22.6.3.5 Llamadas a procedimiento remoto

Una llamada a procedimiento remoto (RPC, remote procedure call) es un mecanismo cliente-servidor que permite a una aplicación en una máquina realizar una llamada a procedimiento que invoque código situado en otra máquina. El cliente llama a un procedimiento local (una *rutina stub*) que empaqueta los argumentos dentro de un mensaje y los envía a través de la red a un proceso servidor concreto. La rutina *stub* del lado del cliente se bloquea entonces en espera de una respuesta. Mientras tanto, el servidor desempaquetá el mensaje, invoca el procedimiento, empaquetá los resultados que hay que devolver en un mensaje y devuelve éste a la rutina *stub* del cliente. La rutina *stub* del cliente se desbloquea, recibe el mensaje, desempaquetá los resultados de la llamada RPC y los devuelve al proceso que realizó la invocación. Este empaquetamiento de los argumentos a veces se denomina **conformación** (*marshalling*). El mecanismo RPC de Windows XP se ajusta al ampliamente utilizado estándar para entornos de informática distribuida basada en mensajes RPC, por lo que los programas escritos con llamadas RPC de Windows XP son altamente portables. El estándar RPC es muy detallado y oculta muchas de las diferencias de arquitectura existentes entre unas computadoras y otras, como por ejemplo los tamaños de los números binarios y el orden de los bytes y bits dentro de las palabras usadas en una máquina concreta, al especificar formatos de datos estándar para los mensajes RPC.

Windows XP puede enviar mensajes RPC utilizando NetBIOS, usando Winsock en redes TCP/IP o empleando *pipes* con nombre sobre redes LAN Manager. La funcionalidad LPC, explicada anteriormente, es similar a RPC, excepto en que en el caso de LPC los mensajes se pasan entre dos procesos que se están ejecutando en la misma computadora.

22.6.3.6 Lenguaje de definición de interfaces de Microsoft

Resulta tedioso y bastante proclive a errores escribir el código para conformar y transmitir los argumentos en el formato estándar, extraerlos y ejecutar el procedimiento remoto, conformar y enviar los resultados devueltos, y extraer y devolver esos resultados al proceso invocador. Afortunadamente, sin embargo, buena parte de este código puede generarse automáticamente a partir de una simple descripción de los argumentos y de los resultados de retorno.

Windows XP proporciona el **Lenguaje de definición de interfaces de Microsoft** para describir los nombres, los argumentos y resultados de los procedimientos remotos. El compilador de este lenguaje genera archivos de cabecera que declaran las rutinas *stub* para los procedimientos remotos, así como los tipos de datos de los mensajes de argumentos y valores de retorno. También genera el código fuente para las rutinas *stub* utilizadas en el lado del cliente y para el mecanismo de extracción y despacho en el lado del servidor. Cuando se monta la aplicación, se incluyen las rutinas *stub*. Cuando la aplicación ejecuta la rutina *stub* RPC, el código generado se encarga del resto.

22.6.3.7 Modelo COM

El **modelo COM** (component object model, modelo de objetos componentes) es un mecanismo para la comunicación interprocesos que fue desarrollada para Windows. Los objetos COM proporcionan una interfaz bien definida para manipular los datos del objeto. Por ejemplo, COM es la infraestructura utilizada por la **tecnología OLE** (object linking and embedding, montaje e incrustación de objetos) de Microsoft para insertar hojas de cálculo en documentos de Microsoft Word. Windows XP tiene una extensión distribuida denominada DCOM que puede utilizarse a través de una red mediante RPC con el fin de proporcionar un método transparente de desarrollo de aplicaciones distribuidas.

22.6.4 Redireccionamiento y servidores

En Windows XP, una aplicación puede utilizar la API de E/S de Windows XP para acceder a archivos de una computadora remota como si fueran locales, siempre y cuando la computadora remota esté ejecutando un servidor CIFS, como el proporcionado por Windows XP o por los sistemas Windows anteriores. Un **redactor** es el objeto del lado cliente que re-envía las solicitudes de E/S relativas a archivos remotos, solicitudes que serán satisfechas por un servidor. Para mejorar las prestaciones y la seguridad, los redireccionadores y servidores se ejecutan en modo *kernel*.

Analizando más en detalle este tema, el acceso a un archivo remoto se produce de la siguiente manera:

1. La aplicación invoca al gestor de E/S para solicitar que se abra un archivo con un nombre de archivo en formato UNC estándar.
2. El gestor de E/S construye un paquete de solicitud de E/S, como se describe en la Sección 22.3.3.5.
3. El gestor de E/S detecta que el acceso se refiere a un archivo remoto e invoca un controlador denominado **proveedor múltiple UNC** [multiple universal-naming-convention provider (MUP)].
4. El MUP envía el paquete de solicitud de E/S asíncronamente a todos los redireccionadores registrados.
5. Un redactor que pueda satisfacer la solicitud responderá al MUP. Para evitar preguntar a todos los redireccionadores la misma cuestión en el futuro, MUP utiliza una caché para recordar qué redactor puede gestionar este archivo.
6. El redactor envía la solicitud de red al sistema remoto.
7. Los controladores de red del sistema remoto reciben la solicitud y la pasan al controlador del servidor.

8. El controlador del servidor entrega la solicitud al controlador apropiado del sistema de archivos local.
9. Se invoca el controlador de dispositivo apropiado para acceder a los datos.
10. Los resultados se devuelven al controlador del servidor, el cual los envía al redirector solicitante. El redirector devuelve entonces los datos a la aplicación que realizó la invocación a través del gestor de E/S.

Para las aplicaciones que utilizan la API de red de la API Win32 en lugar de los servicios UNC, se lleva a cabo un proceso similar, salvo porque se emplea un módulo denominado encaminador multiproveedor, en lugar de utilizar MUP.

En aras de la portabilidad, los redirectores y servidores utilizan la API TDI para el transporte de red. Las propias solicitudes se expresan en un protocolo de mayor nivel, que es de forma predefinida el protocolo SMB mencionado en la Sección 22.6.2. La lista de redirectores se mantiene en la base de datos del Registro del sistema.

22.6.4.1 Sistema de archivos distribuido

Los nombres UNC no siempre resultan cómodos, por que puede haber disponibles múltiples servidores de archivos para proporcionar un mismo contenido, y los nombres UNC incluyen explícitamente el nombre del servidor. Windows XP soporta también un protocolo de **sistema de archivos distribuido** (DFS, distributed file system) que permite a un administrador de red servir los archivos desde múltiples servidores, utilizando un espacio de nombres distribuido.

22.6.4.2 Redirección de carpetas y caché del lado del cliente

Para mejorar la interactividad de las máquinas PC empleadas por los usuarios de empresas que cambian frecuentemente de una computadora a otra, Windows XP permite a los administradores asignar a los usuarios **perfiles móviles**, que mantienen las preferencias de los usuarios y otros datos de configuración en los servidores. Entonces, se utiliza un mecanismo de **redirección de carpetas** para almacenar automáticamente los documentos y otros archivos de los usuarios en un servidor. Este mecanismo funciona bien hasta que una de las computadoras deja de estar conectada a la red, como por ejemplo una computadora portátil en un avión. Para proporcionar a los usuarios acceso fuera de línea a sus archivos redirigidos, Windows XP utiliza un mecanismo de **caché del lado del cliente** (CSC, client-side caching). CSC se utiliza cuando la computadora está en línea para mantener copias de los archivos del servidor en la máquina local, con el fin de mejorar las prestaciones. Los archivos se vuelcan en el servidor en cuanto son modificados. Si la computadora se desconecta, los archivos seguirán estando disponibles y la actualización del servidor se diferirá hasta la siguiente vez que la computadora esté en línea y disponga de un enlace de red con las adecuadas prestaciones.

22.6.5 Dominios

Muchos entornos de red tienen grupos naturales de usuarios, como por ejemplo los estudiantes de un laboratorio de informática o los empleados de cierto departamento de una empresa. Frecuentemente, surge la necesidad de que todos los miembros del grupo sean capaces de acceder a una serie de recursos compartidos situados en las diversas computadoras que forman el grupo. Para gestionar los derechos globales de acceso dentro de estos grupos, Windows XP utiliza el concepto de dominio. Anteriormente, estos dominios no tenían ningún tipo de relación con el sistema de nombres de dominio (DNS, domain-name system) que asigna nombre de *host* de Internet a direcciones IP. Sin embargo, ahora ambos conceptos están estrechamente relacionados.

Específicamente, un dominio de Windows XP es un grupo de estaciones de trabajo y servidores Windows XP que comparten una política de seguridad y una base de datos de usuarios comunes. Puesto que Windows XP ahora utiliza el protocolo Kerberos para las cuestiones de confianza y de autenticación, un dominio Windows XP es lo mismo que un dominio Kerberos. Las versiones

anteriores de NT utilizaban la idea de controladores de dominio principales y de reserva; ahora todos los servidores de un dominio son controladores de dominio. Además, las versiones anteriores requerían que se definieran relaciones de confianza unidireccionales entre los dominios. Windows XP emplea un enfoque jerárquico basado en DNS y permite establecer relaciones de confianza transitivas que pueden fluir hacia arriba y hacia abajo de la jerarquía. Esta técnica reduce el número de relaciones de confianza requeridas para n dominios, desde $n * (n - 1)$ a $O(n)$. Las estaciones de trabajo del dominio confían en que el controlador de dominio proporcione información correcta acerca de los derechos de acceso de cada usuario (a través del testigo de acceso del usuario). Todos los usuarios continúan teniendo la capacidad de restringir el acceso a sus propias estaciones de trabajo, independientemente de lo que cualquier controlador de dominio pueda decir.

22.6.5.1 Bosques y árboles de dominio

Dado que una empresa puede tener muchos departamentos y un centro educativo muchas clases, a menudo es necesario gestionar múltiples dominios dentro de una misma organización. Un **árbol de dominios** es una jerarquía de denominación DNS contigua para la gestión de múltiples dominios. Por ejemplo, *bell-labs.com* podría ser la raíz del árbol, teniendo *research.bell-labs.com* y *pez.bell-labs.com* como hijos (correspondientes a los dominios *research* y *pez*). Un **bosque** es un conjunto de nombres no contiguos. Un ejemplo serían los árboles *bell-labs.com* y/o *lucent.com*. Sin embargo, un bosque puede estar formado también por un único árbol de dominios.

22.6.5.2 Relaciones de confianza

Pueden establecerse relaciones de confianza entre dominios de tres formas distintas: unidireccionales, transitivas y cruzadas. Las versiones de NT hasta la 4.0 sólo permitían relaciones de confianza unidireccionales. Una **relación de confianza unidireccional** es exactamente lo que su nombre indica: informamos al dominio A de que puede confiar en el dominio B. Sin embargo, B no confiará en A, a menos que se configure otra relación de confianza. En una **relación de confianza transitiva**, si A confía en B y B confía en C, entonces A, B y C confiarán todos entre sí, ya que las relaciones de confianza transitivas son bidireccionales de manera predeterminada. La relaciones de confianza transitivas están habilitadas por omisión para los nuevos dominios de un árbol y sólo pueden configurarse entre dominios que formen parte de un mismo bosque. El tercer tipo, las **relaciones de confianza cruzadas**, resulta útil para reducir el tráfico de autenticación. Suponga que los dominios A y B son nodos hoja y que los usuarios de A emplean a menudo recursos que se encuentran en B. Si se utiliza una relación de confianza transitiva normal, las solicitudes de autenticación deben recorrer el árbol hasta alcanzar el ancestro común de los dos nodos hoja; pero si A y B tienen establecida una relación de confianza cruzada, la información de autenticación se envía directamente al otro nodo.

22.6.6 Active Directory

Active Directory es la implementación de Windows XP de los servicios LDAP (lightweight directory-access protocol, protocolo ligero de acceso a directorio). Active Directory almacena la información topológica acerca del dominio, mantiene las cuentas de usuario y de grupo basadas en el dominio, con sus correspondientes contraseñas, y proporciona un repositorio de dominio para tecnologías tales como las **políticas de grupo** y la **duplicación inteligente en espejo** (Intellimirror).

Los administradores emplean las políticas de grupo para establecer estándares relativos a las preferencias y al software de las máquinas de sobremesa. Para muchos grupos de Tecnologías de la Información de las grandes empresas, la uniformidad reduce enormemente el coste relacionando con la infraestructura informática. La tecnología de duplicación inteligente en espejo se utiliza junto con las políticas de grupo para especificar qué software debe estar disponible para cada clase de usuario, e incluso para instalar automáticamente el software bajo demanda desde un servidor corporativo.

22.6.7 Resolución de nombres en redes TCP/IP

En una red IP, la **resolución de nombres** es el proceso de convertir un nombre de computadora en una dirección IP, como por ejemplo traduciendo *www.bell-labs.com* a 135.104.1.14. Windows XP proporciona varios métodos de resolución de nombres, incluyendo WINS (Windows Internet name service, servicio de nombres Internet de Windows), resolución de nombres de difusión, DNS (domain-name system, sistema de nombres de dominio), un archivo hosts y un archivo LMHOSTS. Muchos sistemas operativos emplean la mayor parte de estos métodos, por lo que aquí solo describiremos WINS.

En WINS, dos o más servidores WINS mantienen una base de datos dinámica de correspondencias entre nombres y direcciones IP, junto con un software cliente que permite consultar a los servidores. Se utilizan al menos dos servidores para que el servicio WINS pueda sobrevivir a un fallo de servidor y para distribuir entre múltiples máquinas la carga de trabajo de la resolución de nombres.

WINS utiliza el protocolo DHCP (dynamic host-configuration protocol, protocolo dinámico de configuración de host). DHCP actualiza automáticamente las configuraciones de direcciones en la base de datos WINS, sin que el usuario o el administrador intervengan de la forma siguiente: cuando arranca un cliente DHCP, difunde un mensaje *discover* y cada servidor DHCP que recibe el mensaje responde con un mensaje *offer* que contiene una dirección IP e información de configuración para el cliente. El cliente selecciona una de las configuraciones y envía un mensaje *request* al servidor DHCP seleccionado. El servidor DHCP responde con la dirección IP y con la información de configuración que indicó anteriormente, así como una **concesión** de dicha dirección. La concesión proporciona al cliente el derecho a utilizar esa dirección IP durante un período de tiempo especificado. Cuando ha transcurrido la mitad del período de concesión, el cliente tratará de renovar la concesión de esa dirección. Si la concesión no se renueva, el cliente deberá obtener una nueva concesión.

22.7 Interfaz de programación

La API Win32 es la interfaz fundamental para emplear las capacidades ofrecidas por Windows XP. Esta sección describe cinco aspectos principales de la API Win32: acceso a los objetos del *kernel*, compartición de objetos entre procesos, gestión de procesos, comunicación interprocesos y gestión de memoria.

22.7.1 Acceso a los objetos del kernel

El *kernel* de Windows XP proporciona muchos servicios que los programas de aplicación pueden utilizar. Los programas de aplicación obtienen estos servicios manipulando objetos del *kernel*. Un proceso obtiene acceso a un objeto del *kernel* denominado XXX llamando a la función *CreateXXX* para abrir un descriptor de XXX. Este descriptor es único para el proceso. Dependiendo del proceso que se esté abriendo, si la función *Create()* falla, puede devolver 0, o puede devolver una constante especial denominada **INVALID_HANDLE_VALUE**. Un proceso puede cerrar cualquier descriptor invocando la función *CloseHandle()* y el sistema puede borrar el objeto si el contador de procesos que están utilizando el objeto cae a 0.

22.7.2 Compartición de objetos entre procesos

Windows XP proporciona tres formas para compartir objetos entre los procesos. La primera forma es que un proceso hijo herede un descriptor del objeto. Cuando el padre invoca a la función *CreateXXX*, suministra una estructura **SECURITY_ATTRIBUTES** con el campo **bInheritHandle** configurado con el valor TRUE. Este campo crea un descriptor heredable. A continuación, se crea el proceso hijo, pasando un valor TRUE al argumento **bInheritHandle** de la función *CreateProcess()*. La Figura 22.11 muestra un ejemplo de código que crea un descriptor de semáforo heredado por un proceso hijo.

```

SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa, 1, 1, NULL);
char command_line[132];
ostrstream ostring(command_line, sizeof(command_line));
ostring << a_semaphore << ends;
CreateProcess("another_process.exe", command_line,
NULL, NULL, TRUE, . . . );

```

Figura 22.11 Código que permite a un hijo compartir un objeto heredando un descriptor.

Suponiendo que el proceso hijo conozca qué descriptores están compartidos, el padre y el hijo pueden llevar a cabo la comunicación interprocesos a través de los procesos compartidos. En el ejemplo de la Figura 22.11, el proceso hijo obtiene el valor de descriptor gracias al primer argumento de la línea de comandos y luego comparte el semáforo con el proceso padre.

La segunda forma de compartir objetos es que uno de los procesos proporcione al objeto un nombre en el momento de crearlo y que el segundo proceso lo abra utilizando dicho nombre. Este método tiene dos desventajas: Windows XP no proporciona ninguna manera de comprobar si ya existe un objeto con el nombre elegido y el espacio de nombres de objetos es global, con independencia del tipo de objeto. Por ejemplo, dos aplicaciones pueden crear un objeto *pipe* con nombre aunque estuvieran tratando de crear dos objetos distinguibles y posiblemente distintos.

Los objetos con nombre tienen la ventaja de que dos procesos no relacionados pueden compartirlos fácilmente. El primer proceso invoca una de las funciones *CreateXXX* y suministra un nombre dentro del parámetro *lpszName*. El segundo proceso obtiene un descriptor para compartir del objeto invocando *OpenXXX()* (o *CreateXXX*) con el mismo nombre, como se muestra en el ejemplo de la Figura 22.12.

La tercera forma de compartir objetos es mediante la función *DuplicateHandle()*, que duplica un descriptor. Este método requiere algún otro método adicional de comunicación interprocesos para poder pasar el descriptor duplicado. Dado un descriptor de un proceso y el valor del descriptor dentro de ese proceso, un segundo proceso puede obtener un descriptor del mismo objeto y compartirlo. En la Figura 22.13 se muestra un ejemplo de este método.

22.7.3 Gestión de procesos

En Windows XP, un **proceso** es una instancia en ejecución de una aplicación y una **hebra** es una unidad de código que puede ser planificada por el sistema operativo. Por tanto, un proceso contiene una o más hebras. Un proceso arranca cuando algún otro proceso invoca la rutina *CreateProcess()*. Esta rutina carga las bibliotecas de montaje dinámico utilizadas por el proceso y crea una **hebra principal**. Pueden crearse hebras adicionales mediante la función *CreateThread()*. Cada hebra se crea con su propia pila, que tendrá un tamaño predetermina-

```

// Proceso A
.
.
.

HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, "MySEM1");
.

.

// Proceso B
.
.
.

HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
FALSE, "MySEM1");

```

Figura 22.12 Código para compartir un objeto mediante una búsqueda de nombres.

```

// El proceso A quiere proporcionar al proceso B acceso a un
semáforo

// Proceso A
HANDLE a_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
// enviar el valor del semáforo al proceso B
// utilizando un mensaje o un objeto de memoria compartida
. . .

// Proceso B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    process_id_of_A);

HANDLE b_semaphore;

DuplicateHandle(process_a, a_semaphore,
    GetCurrentProcess(), &b_semaphore,
    0, FALSE, DUPLICATE_SAME_ACCESS);

// Utilizar b_semaphore para acceder al semáforo

```

Figura 22.13 Código para compartir un objeto pasando un descriptor.

do de un 1 MB, a menos que se especifique lo contrario dentro de uno de los argumentos de `CreateThread()`. Puesto que algunas funciones de tiempo de ejecución de C mantienen el estado en variables estáticas, como por ejemplo `errno`, las aplicaciones multihebra necesitan protegerse frente a los accesos no sincronizados. La función envoltorio `beginthreadex()` proporciona la adecuada sincronización.

22.7.3.1 Descriptores de instancia

Cada biblioteca de montaje dinámico o archivo ejecutable cargados en el espacio de direcciones de un proceso se identifica mediante un **descriptor de instancia**. El valor del descriptor de instancia es, en la práctica, la dirección virtual en la que se ha cargado el archivo. Una aplicación puede obtener el descriptor de un módulo dentro de su espacio de direcciones pasando el nombre al módulo a la función `GetModuleHandle()`. Si se pasa `NULL` como nombre, se obtiene como resultado la dirección base del proceso. Los 64 KB inferiores del espacio de direcciones no se utilizan, por lo que un programa erróneo que tratara de eliminar la referencia de un puntero `NULL` provocaría una violación de acceso.

Las prioridades en el entorno API Win32 están basadas en el modelo de planificación de Windows XP, pero no pueden seleccionarse todos los valores de prioridad. La API Win32 utiliza cuatro clases de prioridad:

1. `IDLE_PRIORITY_CLASS` (nivel de prioridad 4).
2. `NORMAL_PRIORITY_CLASS` (nivel de prioridad 8).
3. `HIGH_PRIORITY_CLASS` (nivel de prioridad 13).
4. `REALTIME_PRIORITY_CLASS` (nivel de prioridad 24).

Los procesos son típicamente miembros de la clase `NORMAL_PRIORITY_CLASS`, a menos que el padre del proceso fuera de la clase `IDLE_PRIORITY_CLASS` o se especificar otra clase en el momento de invocar `CreateProcess`. La clase de prioridad de un proceso puede modificarse mediante la función `SetPriorityClass()` o pasando un argumento al comando `START`. Por ejemplo, el comando `START/REALTIME cbserver.exe` ejecutaría el programa indicado dentro de la clase `REALTIME_PRIORITY_CLASS`. Sólo los usuarios con el privilegio *increase scheduling*

priority (incrementar prioridad de planificación) pueden mover un proceso a la clase `REALTIME_PRIORITY_CLASS`. Por omisión, los administradores y usuarios avanzados tienen este privilegio.

22.7.3.2 Regla de planificación

Cuando un usuario está ejecutando un programa interactivo, el sistema necesita proporcionar unas prestaciones especialmente buenas al proceso. Por esta razón, Windows XP tiene una regla especial de planificación para los procesos de la clase `NORMAL_PRIORITY_CLASS`. Windows XP distingue entre el proceso de primer plano que está actualmente seleccionado en la pantalla y los procesos de segundo plano que no están actualmente seleccionados. Cuando un proceso pasa a primer plano, Windows XP aumenta el cuento de planificación en un cierto factor, que normalmente es igual a 3 (este factor puede modificarse mediante la opción Rendimiento, disponible a través del Panel de control en la sección Sistema). Este incremento proporciona al proceso de primer plano un tiempo de ejecución tres veces mayor, antes de que se produzca un desalojo debido a los mecanismos de tiempo compartido.

22.7.3.3 Prioridades de las hebras

Una hebra comienza con una prioridad inicial que estará determinada por su clase. La prioridad puede modificarse mediante la función `SetThreadPriority()`. Esta función toma un argumento que especifica una prioridad relativa a la prioridad base de su clase.

- `THREAD_PRIORITY_LOWEST`: base - 2
- `THREAD_PRIORITY_BELOW_NORMAL`: base - 1
- `THREAD_PRIORITY_NORMAL`: base + 0
- `THREAD_PRIORITY_ABOVE_NORMAL`: base + 1
- `THREAD_PRIORITY_HIGHEST`: base + 2

También se utilizan otros dos valores para ajustar la prioridad. Recuerde de la Sección 22.3.2.1 que el *kernel* tiene dos clases de prioridad: 16-31 para la clase de tiempo real y 0-15 para la clase de prioridad variable. `THREAD_PRIORITY_IDLE` configura la prioridad con el valor 16 para las hebras de tiempo real y con el valor 0 para las hebras de prioridad variable. `THREAD_PRIORITY_TIME_CRITICAL` configura la prioridad con el valor 31 para las hebras de tiempo real y con el valor 15 para las hebras de prioridad variable.

Como se ha explicado en la Sección 22.3.2.1, el *kernel* ajusta la prioridad de una hebra dinámicamente, dependiendo de si la hebra está limitada por E/S o por CPU. La API Win32 proporciona un método para inhabilitar este ajuste mediante las funciones `SetProcessPriorityBoost()` y `SetThreadPriorityBoost()`.

22.7.3.4 Sincronización de hebras

Una hebra puede crearse en un **estado suspendido**; la hebra no se ejecutará hasta que otra hebra la convierta en elegible mediante la función `ResumeThread()`. La función `SuspendThread()` realiza la labor opuesta. Estas funciones configuran un contador, por lo que si se suspende una hebra dos veces, deberá ser reanudada dos veces antes de poder ser ejecutada. Para sincronizar el acceso concurrente a los objetos compartido por parte de las hebras, el *kernel* proporciona objetos de sincronización, como semáforos y objetos mútex.

Además, puede conseguirse la sincronización de hebras utilizando las funciones `WaitForSingleObject()` y `WaitForMultipleObjects()`. Otro método de sincronización en la API Win32 es el de sección crítica. Una sección crítica es una región sincronizada de código que sólo puede ser ejecutada por una hebra en cada momento. Una hebra establece una sección crítica invocando `InitializeCriticalSection()`. La aplicación deberá llamar a

`EnterCriticalSection()` antes de entrar en la sección crítica y a `LeaveCriticalSection()` después de salir de ella. Estas dos rutinas garantizan que, si múltiples hebras tratan de entrar concurrentemente en la sección crítica, sólo se permitirá continuar a una de ellas en cada instante. Las otras esperarán dentro de la rutina `EnterCriticalSection()`. El mecanismo de la sección crítica es más rápido que utilizar los objetos de sincronización del *kernel*, porque no asigna objetos del *kernel* hasta que no encuentre una situación de contienda por la sección crítica.

22.7.3.5 Fibras

Una **fibra** es un código en modo usuario que se planifica de acuerdo con un algoritmo de planificación definido por el usuario. Un proceso puede tener múltiples fibras, al igual que puede tener múltiples hebras. Una de las principales diferencias entre hebras y fibras es que, mientras que las hebras pueden ejecutarse concurrentemente, sólo está permitido que se ejecute una fibra cada vez, incluso aunque se esté usando un hardware multiprocesador. Este mecanismo se incluye en Windows XP para facilitar la tarea de portar las aplicaciones UNIX heredadas que hubieran sido escritas para un modelo de ejecución basado en fibras.

El sistema crea una fibra invocando `ConvertThreadToFiber()` o `CreateFiber()`. La principal diferencia entre estas funciones es que `CreateFiber()` no comienza a ejecutar la fibra creada. Para comenzar la ejecución, la aplicación deberá invocar `SwitchToFiber()`. La aplicación puede terminar una fibra llamando a la función `DeleteFiber()`.

22.7.3.6 Conjunto de hebras

La creación y borrado repetitivos de hebras puede resultar muy costosa para las aplicaciones y servicios que realizan pequeñas cantidades de trabajo en cada una de esas hebras. Los conjuntos de hebras proporcionan programas en modo usuario con tres servicios: una cola en la que pueden incluirse solicitudes de trabajo (mediante la API `QueueUserWorkItem()`), una API que puede utilizarse para asociar retrollamadas con descriptores para los cuales realizarse esperas (`RegisterWaitForSingleObject()`), y una serie de interfaces API para asociar retrollamadas con sucesos de fin de temporización (`CreateTimerQueue()` y `CreateTimerQueueTimer()`).

El objetivo de los conjunto de hebras es aumentar el rendimiento. Las hebras son relativamente costosas y cada procesador sólo puede estar ejecutando un determinado trabajo en cada momento, independientemente de cuantas hebras se utilicen. El conjunto de hebras trata de reducir el número de hebras en espera, retardando ligeramente las solicitudes de trabajo (por el procedimiento de reutilizar cada hebra para muchas solicitudes), al mismo tiempo que se proporcionan suficientes hebras como para utilizar de manera efectiva los procesadores de la máquina. Las API de retrollamada para espera y fin de temporización permiten al conjunto de hebras reducir aún más el número de hebras de un proceso, utilizando muchas menos hebras de las que serían necesarias si un proceso tuviera que dedicar una hebra para dar servicio a cada descriptor para el que se produce una espera y a cada suceso de fin de temporización.

22.7.4 Comunicación interprocesos

Las aplicaciones API Win32 gestionan la comunicación interprocesos de varias formas distintas. Una forma es compartiendo objetos del kernel. Otra posible forma consiste en pasar mensajes, una técnica que resulta especialmente popular en las aplicaciones GUI Windows. Una hebra puede enviar un mensaje a otra hebra o a una ventana invocando `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()` o `SendMessageCallback()`. La diferencia entre las dos primeras funciones y las tres últimas es que las primeras son asíncronas: vuelven inmediatamente y la hebra invocante no sabe si el mensaje ha sido realmente entregado. Las otras tres rutinas son síncronas: bloquean al llamar hasta que el mensaje ha sido entregado y procesado.

Además de enviar un mensaje, una hebra puede enviar datos dentro del mensaje. Puesto que los procesos tienen espacios de direcciones separados, es necesario copiar los datos. El sistema

copia los datos invocando `SendMessage()` para enviar un mensaje de tipo `WM_COPYDATA` con una estructura de datos `COPYDATASTRUCT` que contiene la longitud y la dirección de los datos que hay que transferir. Cuando se envía el mensaje, Windows XP copia los datos en un nuevo bloque de memoria y proporciona la dirección virtual del nuevo bloque al proceso receptor.

A diferencia de las hebras en el entorno Windows de 16 bits, toda hebra de la API Win32 tiene su propia cola de entrada a través de la cual recibe sus mensajes (todas las entradas se reciben mediante mensajes). Esta estructura es más fiable que la cola de entrada compartida de Windows de 16 bits, porque con colas separadas, ya no resulta posible que una aplicación bloqueada impida que las otras aplicaciones reciban su entrada. Si una aplicación API Win32 no invoca `GetMessage()` para tratar los sucesos relativos a su cola de entrada, la cola termina llenándose y, después de cinco segundos, el sistema marca la aplicación como "No responde".

22.7.5 Gestión de memoria

La API Win32 proporciona diversas formas para que una aplicación utilice la memoria: memoria virtual, archivos mapeados en memoria, cúmulos de memoria y almacenamiento local en las hebras.

22.7.5.1 Memoria virtual

Una aplicación invoca `VirtualAlloc()` para reservar o confirmar memoria virtual y `VirtualFree()` para anular la reserva o liberar la memoria. Estas funciones permiten a la aplicación especificar la dirección virtual en la que se asigna la memoria y operan sobre múltiplos del tamaño de página de la memoria, debiendo la dirección inicial de cada región asignada ser mayor que `0x10000`. En la Figura 22.14 se muestran ejemplos de estas funciones.

Un proceso puede bloquear parte de sus páginas confirmadas en la memoria física, invocando la función `VirtualLock()`. El número máximo de páginas que un proceso puede bloquear es de 30, a menos que el proceso haya invocado primero a `SetProcessWorkingSetSize()` para incrementar el tamaño máximo del conjunto de trabajo.

22.7.5.2 Mapeo de archivos en memoria

Otra forma de utilizar la memoria por parte de las aplicaciones consiste en mapear un archivo en memoria dentro de su espacio de direcciones. El mapeo de memoria es también una forma muy cómoda para que dos procesos comparten memoria. Ambos procesos mapean el mismo archivo en su memoria virtual. El mapeo de memoria es un proceso multietapa, como se puede ver en el ejemplo de la Figura 2.15.

```
// asignar 16 MB en la parte superior de nuestro espacio
// de direcciones
void *buf = VirtualAlloc(0, 0x1000000, MEM_RESERVE | MEM_TOP_DOWN,
    PAGE_READWRITE);

// confirmar los 8 MB superiores del espacio asignado
VirtualAlloc(buf + 0x800000, 0x800000, MEM_COMMIT, PAGE_READWRITE);
// hacer algo con la memoria
.

// ahora anular la reserva de la memoria
VirtualFree(buf + 0x800000, 0x800000, MEM_DECOMMIT);
// liberar todo el espacio de direcciones asignado
VirtualFree(buf, 0, MEM_RELEASE);
```

Figura 22.14 Fragmentos de código para la asignación de memoria virtual.

```

// abrir el archivo o crearlo si no existe
HANDLE hfile = CreateFile("somefile", GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// crear un mapeo de archivo de 8 MB de tamaño
HANDLE hmap = CreateFileMapping(hfile, PAGE_READWRITE,
    SEC_COMMIT, 0, 0x800000, "SHM_1");
// obtener ahora una vista del espacio mapeado
void *buf = MapViewOfFile(hmap, FILE_MAP_ALL_ACCESS,
    0, 0, 0x800000);
// hacer algo con el archivo mapeado
.

// eliminar el mapeo del archivo
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);

```

Figura 22.15 Fragmentos de código para mapear en memoria un archivo.

Si un proceso quiere mapear un cierto espacio de direcciones simplemente para compartir una región de memoria con otro proceso, no hace falta ningún archivo. El proceso llamará a `CreateFileMapping()` con un descriptor de archivo igual a `0xffffffff` y definirá un tamaño concreto. El objeto de mapeo de memoria resultante puede ser compartido por herencia, por búsqueda de nombres o por duplicación.

22.7.5.3 Cúmulos de memoria

Los cúmulos de memoria proporcionan una tercera forma de utilización de memoria por parte de las aplicaciones. Un cúmulo de memoria en el entorno Win32 es una región reservada del espacio de direcciones. Cuando un proceso API Win32 se inicializa, se crea con un **cúmulo de memoria predeterminado** de un 1 MB. Puesto que muchas funciones de la API Win32 utilizan el cúmulo de memoria predeterminado, el acceso al cúmulo está sincronizado para proteger las estructuras de datos de asignación del espacio del cúmulo frente a posibles daños provocados por las actualizaciones concurrentes realizadas por múltiples hebras.

LA API Win32 proporciona diversas funciones de gestión de cúmulos de memoria para que un proceso pueda asignar y gestionar un cúmulo privado. Estas funciones son `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()` y `HeapDestroy()`. La API Win32 también proporciona las funciones `HeapLock()` y `HeapUnlock()` para permitir que una hebra obtenga acceso exclusivo a un cúmulo de memoria. A diferencia de `VirtualLock()`, estas funciones sólo se encargan de la sincronización, y no bloquean las páginas en memoria física.

22.7.5.4 Almacenamiento local en una hebra

La cuarta forma en que las aplicaciones pueden utilizar la memoria es mediante un mecanismo de almacenamiento local en una hebra. Las funciones que utilizan datos globales o estáticos no suelen funcionar adecuadamente en entornos multihebra. Por ejemplo, la función C de tiempo de ejecución `strtok()` utiliza una variable estática para controlar su posición actual mientras está analizando sintácticamente una cadena. Para que dos hebras concurrentes puedan ejecutar `strtok()` de manera correcta, necesitan variables separadas que indiquen la *posición actual*. El mecanismo de almacenamiento local en una hebra permite disponer de un almacenamiento global separado para cada hebra. Este mecanismo de almacenamiento proporciona métodos tanto estáticos como dinámicos para crear un espacio de almacenamiento local a la hebra. En la Figura 22.16 se ilustra el método dinámico.

```

// reservar espacio para una variable
DWORD var_index = TlsAlloc();
// asignarla el valor 10
TlsSetValue(var_index, 10);
// obtener el valor

int var = TlsGetValue(var_index);
// liberar el índice
TlsFree(var_index);

```

Figura 22.16 Código para el almacenamiento dinámico local a una hebra.

Para utilizar una variable estática local a la hebra, la aplicación declara la variable de la forma siguiente, con el fin de garantizar que toda hebra tenga su propia copia privada:

```
__declspec(thread) DWORD} cur_pos = 0;
```

22.8 Resumen

Microsoft ha diseñado Windows XP para que sea un sistema operativo ampliable y portable que pueda aprovechar las nuevas técnicas y el nuevo hardware disponible. Windows XP soporta múltiples entornos operativos y hardware de multiprocesamiento simétrico, incluyendo procesadores de 32 bits y de 64 bits y computadoras NUMA. La utilización de objetos del *kernel* para proporcionar servicios básicos, junto con el soporte para el modelo cliente-servidor, permite a Windows XP soportar una amplia variedad de entornos de aplicación. Por ejemplo, Windows XP puede ejecutar programas compilados para MS-DOS, Windows 16, Windows 95, Windows XP y POSIX. Proporciona mecanismos de memoria virtual, caché integrada y planificación apropiativa. Windows XP dispone de un modelo de seguridad más robusto que el de los anteriores sistemas operativos de Microsoft e incluye características de internacionalización. Windows XP se ejecuta sobre una amplia variedad de computadoras, de modo que los usuarios pueden seleccionar y actualizar el hardware para ajustarse al presupuesto disponible y a sus requisitos de prestaciones sin tener que modificar las aplicaciones que ejecutan.

Ejercicios

- 22.1 ¿En qué circunstancias se utilizaría la funcionalidad de llamadas a procedimientos diferidas de Windows XP?
- 22.2 ¿Qué es un descriptor y cómo puede un proceso obtener un descriptor?
- 22.3 Describa el esquema gestión utilizado por el gestor de memoria virtual. ¿Cómo hace el gestor VM para mejorar las prestaciones?
- 22.4 Describa una aplicación útil de la funcionalidad de páginas sin acceso proporcionada en Windows XP.
- 22.5 Los procesadores IA64 contienen registros que pueden utilizarse para direccionar un espacio de direcciones de 64bits. Sin embargo, Windows XP limita el espacio de direcciones de los programas de usuario a 8 TB, lo que se corresponde con 43 bits. ¿Por qué se tomó esta decisión?
- 22.6 Describa las tres técnicas utilizadas para la comunicación de datos en una llamada a procedimiento local. ¿Cuáles son las opciones de configuración más decisivas a la hora de aplicar las diferentes técnicas de paso de mensajes?
- 22.7 ¿Qué es lo que se utiliza para gestionar la caché en Windows XP? ¿Cómo se gestiona la caché?

- 22.8 ¿Cuál es el propósito del entorno Win16? ¿Qué limitaciones se imponen a los programas que se ejecutan dentro de este entorno? ¿Cuáles son las garantías de protección proporcionadas entre diferentes aplicaciones que se ejecuten en un entorno Windows 16? ¿Cuáles son las garantías de protección proporcionadas entre una aplicación que se ejecute en un entorno Windows 16 y una aplicación de 32 bits?
- 22.9 Describa dos procesos de modo usuario que Windows XP proporcione para poder ejecutar programas desarrollados para otros sistemas operativos.
- 22.10 ¿En qué sentido difiere la estructura de directorios empleada en NTFS de la estructura usada en los sistemas operativos UNIX?
- 22.11 ¿Qué es un proceso y cómo se gestiona en Windows XP?
- 22.12 ¿Qué es la abstracción de fibras proporcionada por Windows XP? ¿En qué sentido difiere de la abstracción de hebras?

Notas bibliográficas

Solomon y Russinovich [2000] proporcionan una introducción a Windows XP y considerables detalles técnicos acerca de las interioridades y componentes del sistema. Tate [2000] es una buena referencia sobre la utilización de Windows XP. El Kit de recursos de Microsoft Windows XP Server (Microsoft [2000b]) es un conjunto de seis volúmenes muy útil para utilizar e implantar Windows XP. Microsoft Developer Network Library (Microsfot [2000a]), que se publica trimestralmente, suministra una amplia información sobre Windows XP y otros productos Microsoft.

Iseminger [2000] proporciona una buena referencia sobre la tecnología Active Directory de Windows XP. Richter [1997] incluye un análisis detallado sobre la escritura de programas utilizando la API Win32. Silberschatz et al. [2001] contiene un buen análisis de los árboles B+.

Sistemas operativos influyentes



Ahora que ya entendemos los conceptos fundamentales de los sistemas operativos (planificación de la CPU, gestión de memoria, procesos, etc.), estamos en disposición de examinar cómo se han aplicado estos conceptos en diversos sistemas operativos antiguos y altamente influyentes. Algunos de ellos (como por ejemplo el sistema XDS-940 y THE) fueron sistemas de los que sólo se construyó un único ejemplar; otros (como OS/360) se utilizan ampliamente. El orden de presentación resalta las similitudes y diferencias entre los sistemas, de modo que no es estrictamente cronológico ni están ordenados según importancia. Para abordar con seriedad el estudio de los sistemas operativos, es necesario familiarizarse con todos estos sistemas.

A medida que describamos los sistemas pioneros, incluiremos referencias a las lecturas adicionales. La lectura de los artículos, escritos por los diseñadores de los sistemas, resulta importante tanto por su contenido técnico como por su estilo.

23.1 Sistemas pioneros

Las primeras computadoras eran máquinas físicamente enormes que se ejecutaban desde una consola. El programador, que también era el operador del sistema informático, escribía un programa y luego lo gestionaba directamente desde la consola del operador. Primero, había que cargar manualmente el programa en memoria mediante los conmutadores del panel frontal (una instrucción cada vez), mediante cinta de papel o utilizando tarjetas perforadas. Después, se pulsaban los botones apropiados para establecer la dirección de comienzo e iniciar la ejecución del programa. A medida que el programa se ejecutaba, el programador/operador podía monitorizar su ejecución mediante los indicadores luminosos de la consola. Si se descubrían errores, el programador podía parar el programa, examinar el contenido de la memoria y de los registros, y depurar el programa directamente desde la consola. La salida se imprimía o se perforaba en cinta de papel en tarjetas para su posterior impresión.

23.1.1 Sistemas informáticos dedicados

A medida que fue pasando el tiempo, se desarrolló software y hardware adicional. Los lectores de tarjetas, las impresoras de líneas y las cintas magnéticas llegaron a ser comunes. Se diseñaron ensambladores, cargadores y montadores para facilitar la tarea de programación y se crearon bibliotecas de funciones comunes. Estas funciones comunes podían entonces copiarse en un nuevo programa sin tener que escribirlas de nuevo, proporcionando una reusabilidad del software.

Las rutinas que realizaban las operaciones de E/S eran especialmente importantes. Cada nuevo dispositivo de E/S tenía sus propias características y requería una cuidadosa programación. Para cada dispositivo de E/S se escribía una subrutina especial denominada controlador de dispositivo. Un controlador de dispositivo sabe cómo deben utilizarse los búferes, indicadores, registros, bits de control y bits de estado de un dispositivo concreto. Cada tipo de dispositivo tiene su propio controlador. Una tarea simple, como por ejemplo leer un carácter de un lector de cinta de

papel, podría requerir complejas secuencias de operaciones específicas del dispositivo. En lugar de escribir el código necesario cada vez, se utilizaba simplemente el controlador del dispositivo incluido en la biblioteca.

Posteriormente, aparecieron compiladores para FORTRAN, COBOL y otros lenguajes, haciendo la tarea de programación mucho más fácil aunque complicando la operación de la computadora. Por ejemplo, para preparar un programa FORTRAN para su ejecución, el programador necesitaba primero cargar el compilador FORTRAN en la computadora. El compilador normalmente se mantenía en cinta magnética, por lo que era necesario montar la cinta apropiada en una unidad de cinta. El programa se leía mediante un lector de tarjetas y se escribía en otra cinta. El compilador FORTRAN producía una salida en lenguaje ensamblador, que necesita ser ensamblada. Este procedimiento requería montar otra cinta con el ensamblador. La salida del ensamblador necesita, a su vez, ser montada con las rutinas de biblioteca necesarias. Finalmente, la versión objeto binaria del programa quedaba lista para su ejecución, pudiéndosela cargar en la memoria y depurarla desde la consola, como antes.

Para ejecutar cada trabajo hacia falta un considerable **tiempo de preparación**. Cada trabajo consistía en muchos pasos separados:

1. Cargar la cinta del compilador de FORTRAN.
2. Ejecutar el compilador.
3. Descargar la cinta del compilador.
4. Cargar la cinta del ensamblador.
5. Ejecutar el ensamblador.
6. Descargar la cinta del ensamblador.
7. Cargar el programa objeto.
8. Ejecutar el programa objeto.

Si se producía un error durante cualquiera de los pasos, el programador/operador podía tener que comenzar otra vez de nuevo desde el principio. Cada paso del trabajo podía implicar la carga y la descarga de cintas magnéticas, cintas de papel y tarjetas perforadas.

El tiempo de preparación de cada trabajo era un auténtico problema. Mientras que se montaban las cintas o el programador operaba en la consola, la CPU permanecía inactiva. Recuerde que, en los primeros días, había muy pocas computadoras y éstas eran muy caras. Una computadora podía constar millones de euros, sin incluir el coste de operación debido a la alimentación eléctrica, la refrigeración, los programadores, etc. Por tanto, el tiempo de computadora era extremadamente valioso y sus propietarios querían que se utilizara el máximo tiempo posible. Necesitaban una alta utilización con el fin de obtener un retorno de inversión máximo.

23.1.2 Sistemas informáticos compartidos

La solución se basó en dos aspectos distintos. En primer lugar, se decidió que convenía contratar un operador informático profesional, con el fin de que el programador no tuviera que operar la máquina. En cuanto un trabajo finalizaba, el operador podía comenzar con el siguiente. Puesto que el operador tenía más experiencia con el montaje de cintas que el programador, el tiempo de preparación se redujo. El programador proporcionaba las tarjetas o cintas necesarias, así como una corta descripción de cómo había que ejecutar el trabajo. Por supuesto, el operador no podía depurar un programa incorrecto de la consola, ya que no comprendía las complejidades del programa. Por tanto, en el caso de producirse un error de programa, se realizaba un volcado de la memoria y de los registros y el programador tenía que realizar la depuración a partir del volcado. Ese volcado de la memoria y de los registros permitía al operador continuar inmediatamente con el siguiente trabajo, aunque dejaba al programador un problema de depuración más complejo.

En segundo lugar, los trabajos con necesidades similares se agruparon por lotes con el fin de ejecutarlos en la computadora como un grupo, para así reducir el tiempo de preparación. Por

ejemplo, suponga que el operador hubiera recibido un trabajo FORTRAN, otro trabajo COBOL y otro trabajo FORTRAN. Si los ejecutara en ese orden, tendría que efectuar la preparación para FORTRAN (cargar las cintas del compilador, etc.), luego efectuar la preparación para COBOL y luego hacerla de nuevo para FORTRAN. Sin embargo, al ejecutar los dos programas FORTRAN como un lote, podría ahorrarse uno de los tiempos de preparación para FORTRAN, reduciendo de este modo el tiempo de operador.

Sin embargo, seguían existiendo problemas. Por ejemplo, cuando un trabajo se detenía, el operador tenía que darse cuenta de que se había detenido (observando la consola), determinar *por qué* se había detenido (terminación normal o anormal), volcar la memoria y los registros (en caso necesario), cargar el dispositivo apropiado con el siguiente trabajo y reiniciar la computadora. Durante esta transición de un trabajo al siguiente, la CPU permanecía inactiva.

Para aprovechar este tiempo de inactividad, se desarrollaron mecanismos de **secuenciamiento automático de trabajos**; con esta técnica nacieron los primeros sistemas operativos rudimentarios. Un pequeño programa, denominado **monitor residente**, se utilizaba para transferir el control automáticamente de un trabajo al siguiente (Figura 23.1). El monitor residente se encuentra siempre en memoria (por eso se le llama *residente*).

Al encender la computadora, se invocaba el monitor residente, el cual transfería el control a un programa. Cuando el programa terminaba, devolvía el control al monitor residente, que continuaba con el programa siguiente. De este modo, el monitor residente efectuaba un secuenciamiento automático de un programa a otro y de un trabajo a otro.

Pero, ¿cómo sabía el monitor residente qué programa ejecutar? Anteriormente, al operador se le proporcionaba una breve descripción de qué programas había que ejecutar y con qué datos. Para proporcionar esta información directamente al monitor se introdujeron las **tarjetas de control**. La idea es muy simple: además del programa o de los datos para un trabajo, el programador incluía las tarjetas de control, que contenían directivas para el monitor residente que indicaban el programa que había que ejecutar. Por ejemplo, un programa de usuario normal podría requerir que se ejecutara uno de tres programas: el compilador FORTRAN (FTN), el ensamblador (ASM) o el programa de usuario (RUN). Para cada uno de estos programas se utilizaba una tarjeta de control diferente:

\$FTN– Ejecutar el compilador FORTRAN.

\$ASM– Ejecutar el ensamblador.

\$RUN– Ejecutar el programa de usuario.

Estas tarjetas le decían al monitor residente qué programas debía ejecutar.

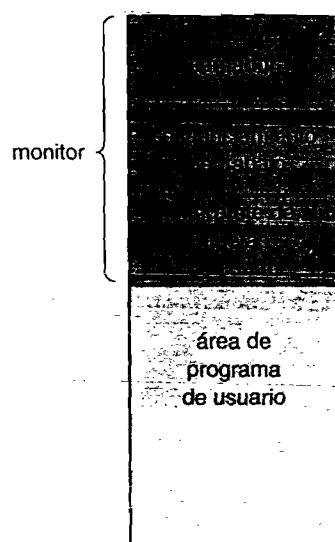


Figura 23.1 Disposición de memoria para un monitor residente.

Podemos utilizar dos tarjetas de control adicionales para definir los límites de cada trabajo:

\$JOB—Primera tarjeta de un trabajo.

\$END—Última tarjeta de un trabajo.

Estas dos tarjetas también eran útiles para propósitos de contabilización de los recursos de máquina utilizados por el programador. Podían utilizarse parámetros para definir el nombre del trabajo, el número de cuenta a la que había efectuar el cargo, etc. Asimismo, podían definirse otras tarjetas de control para otras funciones, como por ejemplo solicitar al operador que cargara o descargara una cinta.

Uno de los problemas de las tarjetas de control es cómo distinguirlas de las tarjetas de datos o de programa. La solución usual consiste en identificarlas mediante un carácter o patrón especial. Varios sistemas utilizaban el carácter de dólar (\$) en la primera columna para identificar una tarjeta de control; otros sistemas utilizaban un código diferente. El lenguaje de control de trabajos (JCL, Job Control Language) de IBM utilizaba barras inclinadas (//) en las primeras dos columnas. La Figura 23.2 muestra un mazo de tarjetas de ejemplo para un sistema simple de procesamiento por lotes.

Por tanto, un monitor residente tiene varias partes identificables:

- El **intérprete de tarjetas de control** es responsable de leer y ejecutar las instrucciones de las tarjetas en el punto de ejecución.
- El **cargador** es invocado por el intérprete de tarjetas de control para cargar programas del sistema y programas de aplicación en la memoria a ciertos intervalos.
- Los **controladores de dispositivo** son utilizados tanto por el intérprete de tarjetas de control como por el cargador para realizar operaciones de E/S a través de los dispositivos del sistema. A menudo, el sistema y los programas de aplicación se montan con estos mismos controladores de dispositivo, proporcionando continuidad en su operación y ahorrando espacio de memoria y tiempo de programación.

Estos sistemas de procesamiento por lotes funcionan bastante bien. El monitor residente proporciona un secuenciamiento automático de trabajos dirigidos por las tarjetas de control. Cuando una tarjeta de control indica que hay que ejecutar un programa, el monitor carga el programa en memoria y le transfiere el control. Cuando el programa se completa devuelve el control al monitor, que lee la siguiente tarjeta de control, carga el programa apropiado, y así sucesivamente. Este ciclo se repite hasta que se han interpretado todas las tarjetas de control del trabajo. Entonces, el monitor continúa automáticamente con el siguiente trabajo.

Este cambio a los sistemas de procesamiento por lotes con secuenciamiento automático de trabajos se realizó con el fin de mejorar el rendimiento. El problema, dicho en pocas palabras, es que

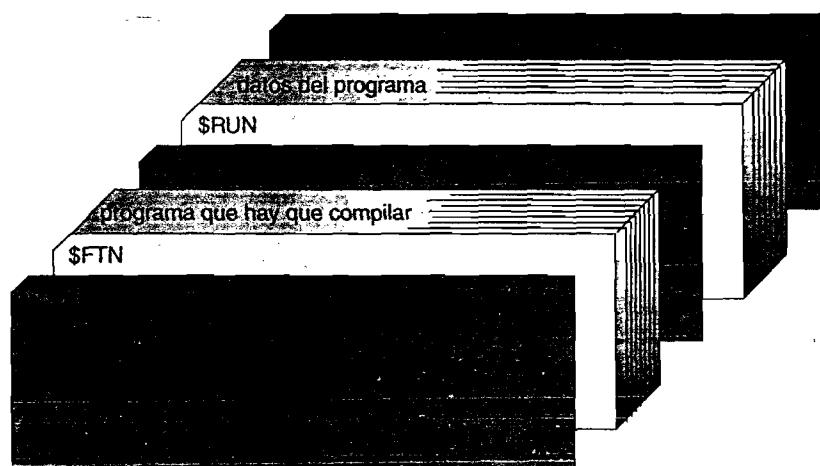


Figura 23.2 Mazo de tarjetas para un sistema simple de procesamiento por lotes.

los humanos son considerablemente más lentos que la computadora. En consecuencia, resulta conveniente sustituir las operaciones humanas por software del sistema operativo. El secuenciamiento automático de trabajos elimina la necesidad de que las personas lleven a cabo el secuenciamiento de trabajos y también el tiempo de preparación.

Sin embargo, como hemos apuntado anteriormente, incluso con esta solución, la CPU estaba inactiva a menudo. El problema era la velocidad de los dispositivos de E/S mecánicos, que son intrínsecamente más lentos que los dispositivos electrónicos. Incluso una CPU lenta funciona en el rango de los microsegundos, ejecutando cada segundo miles de instrucciones. Un lector de tarjetas rápido, por contraste, podía leer 1.200 tarjetas por minuto (es decir, 20 tarjetas por segundo). Por tanto, la diferencia de velocidad entre la CPU y sus dispositivos de E/S podía ser de tres órdenes de magnitud o mayor. Con el tiempo, por supuesto, las mejoras en la tecnología permitieron construir dispositivos de E/S más rápidos. Lamentablemente, las velocidades de CPU se incrementaron con mayor rapidez aún, por lo que el problema no sólo no se resolvió sino que se vio exacerbado.

23.1.3 E/S solapada

Una solución común al problema de la E/S era sustituir los lentos lectores de tarjetas (dispositivos de entrada) e impresoras de líneas (dispositivos de salida) por unidades de cinta magnética. La mayoría de los sistemas informáticos a finales de la década de 1950 y principios de 1960 eran sistemas de procesamiento por lotes que leían utilizando lectores de tarjetas y escribían en impresoras de líneas o perforadoras de tarjetas. Sin embargo, en lugar de hacer que la CPU leyera directamente de las tarjetas, lo que se hizo fue copiar primero las tarjetas en cinta magnética mediante un dispositivo separado. Cuando la cinta estaba suficientemente llena, se la descargaba y se llevaba hasta la computadora. Cuando hacia falta una tarjeta para un programa, se leía el registro equivalente de la cinta. De forma similar, la salida se escribía en cinta y el contenido de la cinta se imprimía posteriormente. Los lectores de tarjetas y las impresoras de líneas se operaban fuera de línea, en lugar de ser controlados por la computadora principal (Figura 23.3).

Una ventaja obvia de la operación fuera de línea era que la computadora principal ya no estaba restringida por la velocidad de los lectores de tarjeta y las impresoras de líneas, sino sólo por la velocidad de las unidades de cinta magnética que eran mucho más rápidas. La técnica de utilizar cintas magnéticas para toda la E/S podía aplicarse para diversos tipos de equipos, como lectores de tarjetas, perforadoras de tarjetas, trazadoras gráficas, lectores de cintas de papel e impresoras.

La verdadera ganancia derivada de la operación fuera de línea proviene de la facilidad de utilizar múltiples sistemas de conversión lector a cinta y cinta a impresora para una misma CPU. Si la CPU puede procesar las entradas dos veces más rápido de que el lector puede leer las tarjetas, entonces dos lectores que estuvieran trabajando simultáneamente podrían producir suficientes cintas magnéticas como para mantener ocupada a la CPU. Sin embargo, existe también una desventaja: un retardo mayor a la hora de ejecutar cada trabajo concreto. El trabajo debe primero leerse en cinta, después debe esperar hasta que sea leído un número suficiente de otros trabajos en la

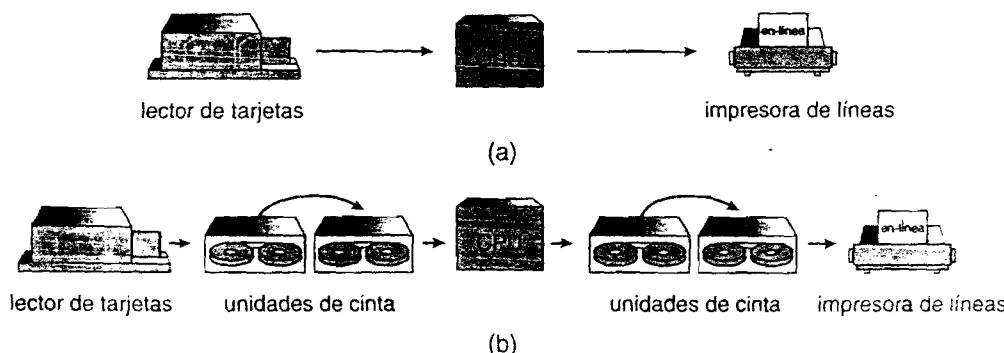


Figura 23.3 Operación de los dispositivos de E/S (a) en línea y (b) fuera de línea.

cinta, con el fin de “llenarla”. Después, había que rebobinar la cinta, descargarla, llevarla a mano hasta la CPU y montarla en una unidad de cinta libre. Por supuesto, este proceso no resulta poco razonable para los sistemas de procesamiento por lotes, ya que pueden agruparse múltiples trabajos similares en una cinta antes de llevarla a la computadora.

Aunque la preparación fuera de línea de los trabajos continuó durante un cierto tiempo, pronto fue sustituida en la mayoría de los sistemas. Los sistemas de disco comenzaron a estar ampliamente disponibles y representaban una enorme mejora con respecto a la operación fuera de línea. El problema con los sistemas de cinta era que el lector de tarjetas no podía escribir en un extremo de la cinta mientras que la CPU estuviera leyendo del otro extremo. Era necesario escribir la cinta completa antes de rebobinarla y leerla, porque las cintas son, por naturaleza, **dispositivos de acceso secuencial**. Los sistemas de disco eliminaban este problema al ser **dispositivos de acceso aleatorio**. Puesto que el cabezal se mueve de un área del disco a otra, un disco puede conmutar rápidamente del área del disco que esté siendo usada por el lector de tarjetas para almacenar nuevas tarjetas, a la posición que la CPU necesita para leer la “siguiente” tarjeta.

En un sistema de disco, las tarjetas se leen directamente desde el lector de tarjetas al disco. La ubicación de las imágenes se registra en una tabla mantenida por el sistema operativo. Cuando se ejecuta un trabajo, el sistema operativo satisface sus solicitudes de entrada de lector de tarjetas leyendo del disco. De forma similar, cuando el trabajo solicita a la impresora imprimir una línea, esa línea se copia en un búfer del sistema y se escribe en el disco. Cuando se ha completado el trabajo, se imprime verdaderamente la salida. Esta forma de procesamiento se denomina *spooling* (Figura 23.4); el nombre es un acrónimo de *simultaneous peripheral operation on line* (operación simultánea de periféricos en línea). El *spooling*, en esencia, utiliza el disco como un inmenso búfer para leer lo más anticipadamente posible en los dispositivos de salida y para almacenar los archivos de salida hasta que los dispositivos de salida estén en disposición de aceptarlos.

El *spooling* también se utiliza para el procesamiento de datos en sitios remotos. La CPU envía los datos a través de determinadas rutas de comunicaciones a una impresora remota (o acepta un trabajo de entrada completo de un lector de tarjetas remoto). El procesamiento remoto se realiza a su propia velocidad, sin intervención de la CPU. La CPU sólo necesita que se le notifique cuando se ha completado el procesamiento, para poder enviar el siguiente lote de datos.

El *spooling* solapa la E/S de un trabajo con los cálculos de otros trabajos. Incluso en un sistema simple, el gestor de *spooling* puede estar leyendo la entrada de un trabajo mientras imprime la salida de un trabajo diferente. Durante este tiempo, pueden también estar ejecutándose otros trabajos, o bien esos trabajos pueden estar leyendo sus “tarjetas” del disco e “imprimiendo” su línea de salida en el disco.

El *spooling* tiene un efecto beneficioso directo sobre el rendimiento del sistema. El único coste asociado es el espacio de disco y unas pocas tablas, y a cambio se pueden solapar los cálculos de un trabajo con las operaciones de E/S de otros trabajos. Por tanto, el *spooling* puede mantener trabajando tanto a la CPU como a los dispositivos de E/S a una tasa mucho más alta. El *spooling* con-

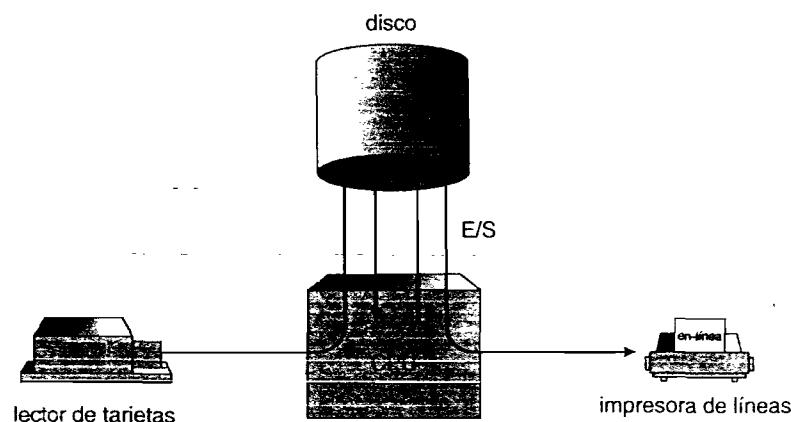


Figura 23.4 Spooling.

duce de manera natural al concepto de multiprogramación, que es la base de todos los sistemas operativos modernos.

23.2 Atlas

El sistema operativo Atlas (Kilburn et al. [1961], Howarth et al. [1961]) fue diseñado en la Universidad de Manchester, Inglaterra, a finales de los años 50 y principios de los 60. Muchas de sus características básicas que eran novedosas en aquel momento han llegado a convertirse en componentes estándar de los sistemas operativos modernos. Los controladores de dispositivo eran una de las partes principales del sistema. Además, se añadieron llamadas al sistema mediante un conjunto de instrucciones especiales denominado *códigos extra*.

Atlas era un sistema operativo de procesamiento por lotes con *spooling*. El *spooling* permitía al sistema planificar los trabajos de acuerdo con la disponibilidad de los dispositivos periféricos, como unidades de cinta magnética, lectores de cinta de papel, perforadoras de cinta de papel, impresoras de líneas, lectores de tarjetas y perforadoras de tarjetas.

Sin embargo, la característica más destacable de Atlas era su gestión de la memoria. La memoria de núcleo era novedosa y muy cara en aquella época. Muchas computadoras, como la IBM 650, utilizaban un tambor como sistema principal. El sistema Atlas empleaba un tambor como memoria principal, pero tenía una pequeña cantidad de la memoria de núcleo que se usaba como caché para el tambor. Se empleaba un mecanismo de paginación bajo demanda para transferir automáticamente información entre la memoria de núcleo y el tambor.

El sistema Atlas utilizaba una computadora de fabricación británica con palabras de 48 bits. Las direcciones tenían 24 bits, pero estaban codificadas en decimal, lo que sólo permitía dirigir 1 millón de palabras. En aquella época, este espacio de direcciones era extremadamente grande. La memoria física de Atlas era un tambor con 98 KB-palabras y un núcleo magnético de 16 KB-palabras. La memoria se dividía en páginas de 512 palabras, proporcionando 32 marcos de memoria física. Una memoria asociativa de 32 registros implementaba la correspondencia entre direcciones virtuales y direcciones físicas.

Si se producía un fallo de página, se invocaba un algoritmo de sustitución de páginas. Uno de los marcos de memoria se mantenía siempre vacío, para que la transferencia del tambor pudiera iniciarse inmediatamente. El algoritmo de sustitución de páginas trataba de predecir el comportamiento futuro del acceso a memoria basándose en el comportamiento anterior. Cada vez que se accedía a un marco, se activaba un bit de referencia para el mismo. Los bits de referencia se leían en memoria cada 1.024 instrucciones y se retenían los últimos 32 valores de estos bits. Este historial se utilizaba para definir el tiempo transcurrido desde la referencia más reciente (t_1) y el intervalo entre las dos últimas referencias (t_2). Las páginas para sustitución se elegían en el orden siguiente:

1. Cualquier página con $t_1 > t_2 + 1$. Se consideraba que dicha página ya no estaba en uso.
2. Si $t_1 \leq t_2$ para todas las páginas, entonces se sustituía la página con el valor $t_2 - t_1$ más grande.

El algoritmo de sustitución de páginas asume que los programas acceden a la memoria en bucles. Si el tiempo entre las dos últimas referencias es t_2 , entonces se espera que se produzca otra referencia t_2 unidades de tiempo más tarde. Si no se produce una referencia ($t_1 > t_2$), se asume que la página ya no está siendo utilizada y esa página se sustituye. Si todas las páginas siguen estando en uso, se sustituye a la página que no va a ser necesaria durante el período más largo de tiempo. El tiempo esperado hasta la siguiente referencia es $t_2 - t_1$.

23.3 XDS-940

El sistema operativo XDS-940 (Lichtenberger y Pirtle [1965]) fue diseñado en la Universidad de California en Berkeley. Al igual que el sistema Atlas, utilizaba un mecanismo de paginación para la gestión de memoria. Sin embargo, a diferencia del sistema Atlas, era un sistema de tiempo compartido.

La paginación se utilizaba únicamente para la reubicación, y no para la paginación bajo demanda. La memoria virtual de cualquier proceso de usuario estaba formada por hasta 16 KB-palabras, mientras que la memoria física estaba formada por 64 KB-palabras. Cada página tenía 2 KB-palabras y la tabla de páginas se almacenaba en registros. Puesto que la memoria física era más grande que la memoria virtual, podía haber varios procesos de usuario simultáneamente en memoria. El número de usuarios podía incrementarse compartiendo páginas que contuvieran código re-entrante de sólo lectura. Los procesos se mantenían en un tambor y se intercambiaban para cargarlos en memoria según fuera necesario.

El sistema XDS-940 fue construido a partir de un sistema XDS-930 modificado. Las modificaciones eran las típicas que se realizan en una computadora básica para poder escribir apropiadamente un sistema operativo. Se añadió un modo monitor de usuario y se definieron como privilegiadas ciertas instrucciones como las de detención y E/S. Los intentos de ejecutar una instrucción privilegiada en modo usuario provocaban una interrupción dirigida al sistema operativo.

Se añadió una instrucción de llamada al sistema al conjunto de instrucciones en modo usuario. Esta instrucción se utilizaba para crear nuevos recursos, como archivos, permitiendo al sistema operativo gestionar los recursos físicos. Los archivos, por ejemplo, se asignan en bloques de 256-palabras del espacio de almacenamiento del tambor. Se empleaba un mapa de bits para gestionar los bloques libres del tambor. Cada archivo tenía un bloque de índice con punteros a los bloques de datos reales. Los bloques de índice estaban encadenados.

El sistema XDS-940 también proporcionaba llamadas al sistema para permitir a los procesos crear, iniciar, suspender y destruir subprocesos. Un programador podía construir un sistema de procesos y los distintos procesos podían compartir memoria para comunicación y sincronización. La creación de procesos definía una estructura en árbol, en la que un proceso era la raíz y sus subprocesos eran los nodos situados por debajo de ella en el árbol. Cada uno de los subprocesos podía, a su vez, crear más subprocesos.

23.4 THE

El sistema operativo THE (Dijkstra [1968], McKeag y Wilson [1976]) fue diseñado en la Technische Hogeschool en Eindhoven, Holanda. Era un sistema de procesamiento por lotes que se ejecutaba en una computadora de fabricación holandesa, la EL X8, con 32 KB de palabras de 27-bits. El sistema era realmente notable por su limpio diseño, particularmente por su estructura en niveles, y por su uso de un conjunto de procesos concurrentes que empleaban semáforos para sincronización.

Sin embargo, a diferencia del sistema XDS-940, el conjunto de procesos en el sistema THE era estático. El propio sistema operativo estaba diseñado como un conjunto de procesos cooperantes. Además, se crearon cinco procesos de usuario que servían como agentes activos para compilar, ejecutar e imprimir programas de usuario. Cuando se terminaba un trabajo, el proceso volvía a la cola de entrada para seleccionar otro trabajo.

Se utilizaba un algoritmo de planificación de la CPU basado en prioridades. Las prioridades se recalculaban cada dos segundos y eran inversamente proporcionales al tiempo de CPU utilizado recientemente (es decir, en los últimos 8 a 10 segundos). Este esquema proporcionaba una prioridad más alta a los procesos limitados por E/S y a los procesos nuevos.

La gestión de memoria estaba limitada por la falta de soporte hardware. Sin embargo, puesto que el sistema era limitado y los programas de usuario sólo podían escribirse en Algol, se empleó un esquema software de paginación. El compilador Algol generaba automáticamente llamadas a las rutinas del sistema, lo que garantizaba que la información solicitada estuviera en memoria, produciéndose intercambios según fuera necesario. El almacén de respaldo era un tambor de 512 KB-palabras. Se utilizaba un tamaño de página de 512 palabras con una estrategia LRU de sustitución de páginas.

Otra de las principales preocupaciones abordada por el sistema THE era el control de los interbloqueos. Para evitar los interbloqueos se empleaba el algoritmo del banquero.

Estrechamente relacionado con el sistema THE está el sistema Venus (Liskov [1972]). El sistema Venus también tenía un diseño estructurado en niveles y utilizaba semáforos para la sincroniza-

ción de los procesos. Sin embargo, los niveles inferiores del diseño estaban implementados en microcódigo, lo que proporcionaba un sistema mucho más rápido. La gestión de memoria se cambió por una memoria segmentada paginada. El sistema también se diseñó como sistema de tiempo compartido en lugar de para procesamiento por lotes.

23.5 RC 4000

El sistema RC 4000, como el sistema THE, era muy notable principalmente por conceptos de diseño. Fue diseñado para la computadora Danish 4000 por Regnecentralen, particularmente por Brinch-Hansen (Brinchhansen [1970], Brinchhansen [1973]). El objetivo no era diseñar un sistema de procesamiento por lotes ni un sistema de tiempo compartido, ni ningún otro sistema específico. En lugar de ello, el objetivo era crear un núcleo de sistema operativo, o *kernel*, sobre el que pudiera construirse un sistema operativo completo. Por tanto, la estructura del sistema era en niveles, y sólo se suministraron los niveles inferiores, que era los que formaban el *kernel*.

El *kernel* soportaba una colección de procesos concurrentes, utilizándose un planificador de CPU con un algoritmo de asignación por turnos. Aunque los procesos podían compartir memoria, el mecanismo principal de comunicación y sincronización era el **sistema de mensajería** proporcionado por el *kernel*. Los procesos podían intercambiarse entre sí intercambiando mensajes de tamaño fijo que tenían una longitud de ocho palabras. Todos los mensajes se almacenaban en búferes que se extraían de un conjunto de búferes común. Cuando ya no se necesitaba un búfer de mensaje, se le devolvía al conjunto común.

Con cada proceso había asociada una **cola de mensajes**. La cola contenía todos los mensajes que hubieran sido enviados a dicho proceso, pero que todavía no hubieran sido recibidos. Los mensajes se eliminaban de la cola en orden FIFO. El sistema soportaba cuatro operaciones primitivas que se ejecutan atómicamente:

- **send-message** (*in receiver, in message, out buffer*)
- **wait-message** (*out sender, out message, out buffer*)
- **send-answer** (*out result, in message, in buffer*)
- **wait-answer** (*out result, out message, in buffer*)

Las dos últimas operaciones permitían a los procesos intercambiar varios mensajes al mismo tiempo.

Estas primitivas requerían que los procesos dieran servicio a su cola de mensajes en orden FIFO y que se bloquearan mientras otros procesos estaban gestionando sus mensajes. Para eliminar estas restricciones, los desarrolladores proporcionaron dos primitivas adicionales de comunicación, que permitían a un proceso esperar la llegada del siguiente mensaje o responder y dar servicio a su cola de mensajes en cualquier orden:

- **wait-event** (*in previous-buffer, out next-buffer, out result*)
- **get-event** (*out buffer*)

Los dispositivos de E/S también se trataban como procesos. Los controladores de dispositivo eran fragmentos de código que convertían las interrupciones de dispositivo y los registros en mensajes. Así, un proceso escribía en un terminal enviando a ese terminal un mensaje. El controlador de dispositivo recibía el mensaje y presentaba el carácter por el terminal. Los caracteres de entrada interrumpían al sistema y se transferían a un controlador de dispositivo, el cual creaba un mensaje con el carácter de entrada y lo enviaba a un proceso que estuviera en espera.

23.6 CTSS

El sistema CTSS (Compatible Time-Sharing System) (Corbato et al. [1962]) fue diseñado en el MIT como sistema experimental de tiempo compartido. Se implementó en un IBM 7090 y llegó a soportar hasta 32 usuarios interactivos. A los usuarios se les proporcionaba un conjunto de comandos

interactivos que les permitían manipular archivos y compilar y ejecutar programas a través de un terminal.

El 7090 tenía una memoria de 32 KB formada por palabras de 36 bits. El monitor utilizaba palabras de 5 KB dejando 27 KB para los usuarios. Las imágenes de memoria de los usuarios se intercambiaban entre la memoria y un tambor de alta velocidad. La planificación de la CPU empleaba un algoritmo de cola de realimentación multinivel. El cuento de tiempo para el nivel i era de $2 * i$ unidades de tiempo. Si un programa no terminaba su ráfaga de CPU en un cuento de tiempo, se le movía al siguiente nivel de la cola, dándole el doble de tiempo. El programa en el nivel superior (con el cuento más corto) se ejecutaba en primer lugar. El nivel inicial de cada programa se determinaba según su tamaño, de modo que el cuento de tiempo fuera al menos igual de grande que el tiempo de intercambio.

CTSS tuvo un gran éxito y fue utilizado hasta 1972. Aunque estaba limitado, fue de capaz de demostrar que el tiempo compartido era una forma de procesamiento cómoda y práctica. Uno de los resultados de CTSS fue promover el desarrollo de sistemas de tiempo compartido. Otro resultado fue el desarrollo de MULTICS.

23.7 MULTICS

El sistema operativo MULTICS (Corbató y Vyssotski [1965], Organick [1972]) fue diseñado en el MIT como una extensión natural de CTSS. El sistema CTSS y otros sistemas pioneros de tiempo compartido tuvieron tanto éxito que incentivaron el deseo inmediato de desarrollar rápidamente otros sistemas mejores y de mayor envergadura. A medida que fueron estando disponibles las computadoras de gran tamaño, los diseñadores de CTSS decidieron crear una utilidad de tiempo compartido. Los servicios informáticos se proporcionarían, según este concepto, como la energía eléctrica: los grandes sistemas informáticos estarían conectados por hilos telefónicos a terminales situados en oficina y domicilios de toda la ciudad. El sistema operativo sería un sistema de tiempo compartido ejecutándose continuamente con un vasto sistema de archivos compuesto por programas y datos compartidos.

MULTICS fue diseñado por un equipo de desarrolladores del MIT, de GE (que posteriormente vendió su división de informática a Honeywell) y de Bell Laboratories (que se salió del proyecto en 1969). Se modificó la computadora básica GE 635 para desarrollar un nuevo sistema informático denominado GE 645, y cuya principal diferencia era la adición de un hardware de memoria con segmentación paginada.

Cada dirección virtual estaba compuesta por un número de segmento de 18 bits y un desplazamiento de palabra de 16 bits. Los segmentos se paginaban utilizando páginas de 1 KB-palabras. Se utilizó el algoritmo de sustitución de páginas de segunda oportunidad.

El espacio virtual de direcciones segmentado estaba mezclado conceptualmente con el sistema de archivos. Cada segmento era un archivo. Los segmentos se direccionaban utilizando el nombre del archivo y el propio sistema de archivos era una estructura en árbol multinivel, que permitía a los usuarios crear sus propias estructuras de subdirectorios.

Al igual que CTSS, MULTICS utilizaba una cola de realimentación multinivel para la planificación de la CPU. La protección se conseguía mediante una lista de acceso asociada a cada archivo y un conjunto de anillos de protección para los procesos en ejecución. El sistema, que estaba escrito, casi por completo, en PL/1, tenía unas 300.000 líneas de código. Posteriormente, se amplió para transformarlo en un sistema multiprocesador, permitiendo retirar una CPU para tareas de mantenimiento mientras el sistema continuaba funcionando.

23.8 IBM OS/360

La línea más larga de desarrollo de sistemas operativos es, sin ninguna duda, la de las computadoras IBM. Las primeras computadoras IBM, como la IBM 7090 y la IBM 7094, constituyen ejemplos notables del desarrollo de subrutinas de E/S comunes, seguido del desarrollo de un monitor residente y de mecanismos de instrucciones privilegiadas, protección de memoria y procesamien-

to por lotes simple. Estos sistemas se desarrollaron de manera separada, a menudo en distintas sedes de la compañía, que funcionaban independientemente. Como resultado, IBM se encontró varias computadoras distintas, con diferentes lenguajes y diferentes software del sistema.

Para resolver esta situación se desarrolló el IBM/360. El IBM/360 se diseñó como una familia de computadoras que abarcaba el rango completo que iba desde máquinas para pequeñas empresas hasta las grandes máquinas de cálculo científico. Sólo faltaba un conjunto de programas software para todos estos sistemas, ya que todos ellos utilizaban el mismo sistema operativo: el OS/360 (Mealy et al. [1966]). Esta decisión se tomó para reducir los problemas de mantenimiento en IBM y para permitir a los usuarios transferir programas y aplicaciones libremente desde un sistema IBM a otro.

Lamentablemente, OS/360 cometió el error de tratar de resolver todos los problemas de todos los tipos de personas al mismo tiempo. Como resultado, no llevaba a cabo ninguna de sus tareas especialmente bien. El sistema de archivos incluía un campo de tipo que definía el tipo de cada archivo, y se definieron diferentes tipos de archivo para registros de longitud fija y de longitud variable y para archivos con bloques y sin bloques. Se utilizaba un mecanismo de asignación contigua, por lo que usuario tenía que adivinar el tamaño de cada archivo de salida. El lenguaje JCL (Job Control Language, lenguaje de control de trabajos) añadía parámetros para todas las opciones posibles, haciendo que fuera incomprensible para el usuario medio.

Las rutinas de gestión de memoria estaban dictadas por la arquitectura. Aunque se utilizó un modo de direccionamiento con registro base, el programa podía acceder al registro base y modificarlo, de modo que la CPU generaba direcciones absolutas. Este mecanismo impedía la reubicación dinámica, con lo que los programas estaban ligados a la memoria física desde el momento de su carga. Se generaron dos versiones distintas del sistema operativo: OS/MFT, que utilizaba regiones fijas y OS/MVT, que empleaba regiones variables.

El sistema fue escrito en lenguaje ensamblador por miles de programadores, lo que dio como resultado millones de líneas de código. El propio sistema operativo requería grandes cantidades de memoria para el código y las tablas. El gasto de procesamiento del sistema operativo requería a menudo un 50 por ciento de los ciclos de CPU totales. A lo largo de los años, se lanzaron nuevas versiones para añadir nuevas características y corregir errores. Sin embargo, la corrección de un error causaba a menudo otro en alguna parte remota del sistema, por lo que el número de errores conocidos en el sistema permaneció prácticamente constante.

La memoria virtual se añadió al OS/360 con el cambio a la arquitectura IBM 370. El hardware subyacente proporcionaba una memoria virtual segmentada-paginada. Las nuevas versiones del sistema operativo utilizaban este hardware de manera diferente. OS/VS1 creaba un espacio virtual de direcciones de gran tamaño y ejecutaba OS/MFT en dicha memoria virtual. Por tanto, el propio sistema operativo estaba paginado, además de estarlos los programas de usuario. OS/VS2 Versión 1 ejecutaba OS/MVT en memoria virtual. Por último, OS/VS2 Versión 2, que ahora se denomina MVS, proporcionaba a cada usuario su propia memoria virtual.

MVS sigue siendo, básicamente, un sistema operativo de procesamiento por lotes. El sistema CTSS se ejecutaba en un IBM 7094, pero el MIT decidió que el espacio de direcciones del 360, el sucesor del IBM 7094, era demasiado pequeño para MULTICS, por lo que cambiaron de fabricante. IBM decidió entonces crear su propio sistema de tiempo compartido, el TSS/360 (Lett y Konigsford [1968]). Al igual que MULTICS, TSS/360 pretendía ser una utilidad de tiempo compartido a gran escala. La arquitectura básica 360 fue modificada en el modelo 67 para proporcionar memoria virtual. Diversas organizaciones adquirieron el 360/67 como anticipación al TSS/360.

Sin embargo, el TSS/360 se retrasó, por lo que se desarrollaron otros sistemas de tiempo compartido como solución temporal hasta que el TSS/360 estuviera disponible. De este modo, se añadió una opción de tiempo compartido (TSO, time-sharing option) al OS/360. El centro de investigación Cambridge de IBM desarrolló CMS como sistema monousuario de CP/67 para proporcionar una máquina virtual sobre la que se pudiera ejecutar (Meyer y Seawright [1970], Parmelee et al. [1972]).

Cuando fue lanzado finalmente el TSS/360 fue un fracaso. Era demasiado complejo y demasiado lento. Como resultado, ninguna organización hizo sustituir su sistema temporal por el TSS/360. Hoy en día, el tiempo compartido en los sistemas IBM se proporciona principalmente mediante TSO sobre MVS o mediante CMS sobre CP/67 (renombrado VM).

Tanto TSS/360 como MULTICS no tuvieron un gran éxito comercial. ¿Qué era lo que fallaba con estos sistemas? Parte del problema era que estos sistemas avanzados eran demasiado complejos y demasiado voluminosos como para poder ser comprendidos por quienes los tenían que utilizar. Otro problema era la suposición de que las capacidades informáticas se proporcionarían mediante grandes computadoras remotas. Ahora sabemos que la mayor parte del procesamiento informático se realiza mediante pequeñas máquinas individuales (computadoras personales), no mediante sistemas complejos y remotos de tiempo compartido que traten de resolver todos los problemas de todos los usuarios.

23.9 Mach

El sistema operativo Mach tiene su antecesor en el sistema operativo Accent desarrollado en la Universidad CMU (Carnegie Mellon University) (Rashid y Robertson [1981]). La filosofía y el sistema de comunicaciones de Mach se derivan de Accent, pero otras partes significativas del sistema (por ejemplo, el sistema de memoria virtual y la gestión de tareas y de hebras) fueron desarrollados partiendo de cero (Rashid [1986], Tevanian et al. [1989] y Accetta et al. [1986]). El planificador de Mach se describe en detalle en Tevanian et al. [1987a] y Black [1990]. Tevanian et al. [1987b] presentó una primera versión del sistema de memoria compartida y de mapeo de memoria de Mach.

El sistema operativo Mach fue diseñado teniendo en mente los tres siguientes objetivos críticos:

1. Emular un UNIX BSD 4.3 de modo que los archivos ejecutables de un sistema UNIX pudieran funcionar correctamente sobre Mach.
2. Ser un sistema operativo moderno que soportara distintos modelos de memoria, y tanto procesamiento paralelo como distribuido.
3. Disponer de un *kernel* que fuera más fácil y sencillo de modificar que BSD 4.3.

El desarrollo de Mach siguió un camino evolutivo a partir de los sistemas UNIX BSD. El código de Mach fue desarrollado inicialmente dentro del *kernel* BSD 4.2, sustituyendo componentes del *kernel* BSD por componentes de Mach a medida que éstos iban siendo completados. Los componentes de BSD se actualizaron a BSD 4.3 cuando esta versión apareció. En 1986, los subsistemas de memoria virtual y de comunicaciones funcionaban sobre la familia de computadoras VAX de DEC, incluyendo las versiones multiprocesador del VAX. Poco después se lanzarían las versiones para el RT/PC de IBM y para las estaciones de trabajo SUN 3. Después, en 1987 se terminaron las versiones multiprocesador para Encore Multimax y Sequent Balance, incluyendo el soporte de tareas y hebras; en ese mismo año se lanzaron las primeras versiones oficiales del sistema, la versión 0 y la versión 1.

Hasta la versión 2, Mach proporcionaba compatibilidad con los correspondientes sistemas BSD, incluyendo gran parte del código de BSD en el *kernel*. Las nuevas características y capacidades de Mach hicieron que el *kernel* de estas versiones fuera mayor que el *kernel* BSD correspondiente. Mach 3 desplazó el código BSD fuera del *kernel*, dejando un *microkernel* mucho más pequeño. El sistema sólo implementa las características de Mach en el *kernel*, todo el código específico de UNIX se ejecuta en servidores de modo usuario. Excluir el código específico de UNIX del *kernel* permite sustituir BSD por otro sistema operativo, y también la ejecución simultánea de múltiples interfaces de sistema operativo sobre el *microkernel*. Además, de BSD, se han desarrollado implementaciones en modo usuario para DOS, para el sistema operativo Macintosh y para OSF/1. Esta técnica presenta similitudes con el concepto de máquina virtual, pero aquí la máquina virtual está definida por software (la interfaz del *kernel* de Mach), en lugar de por hardware. Con la versión 3.0, Mach pasó a estar disponible sobre una amplia variedad de sistemas, incluyendo máquina monoprocesador de SUN, Intel, IBM y DEC, y sistemas multiprocesador de DEC, Sequent y Encore.

Mach obtuvo una gran atención del sector informático cuando OSF (Open Software Foundation) anunció en 1989 que utilizaría Mach 2.5 como base para su sistema operativo OSF/1. El lanzamiento inicial de OSF/1 tuvo lugar un año después, y este sistema entró en competencia

con UNIX System V, versión 4, el sistema operativo más extendido en ese momento entre los miembros de UI (UNIX International). Entre los miembros de OSF había empresas tecnológicas de gran importancia como IBM, DEC y HP. Desde entonces, OSF ha cambiado de dirección y sólo el UNIX de DEC está basado en el *kernel* de Mach.

Mach 2.5 también forma la base del sistema operativo de la estación de trabajo NeXT, una de las criaturas de Steve Jobs, que fue fundador de Apple Computer.

A diferencia de UNIX, que fue desarrollado sin tener en cuenta el multiprocesamiento. Mach incorpora soporte de multiprocesamiento en todo el sistema. Su soporte de multiprocesamiento es enormemente flexible, abarcando todo el rango desde los sistemas de memoria compartida hasta sistemas en los que los procesadores no comparten ninguna memoria. Mach utiliza procesos ligeros, en la forma de múltiples hebras de ejecución dentro de una tarea (espacio de direcciones), para permitir el multiprocesamiento y el procesamiento paralelo. Su intensivo uso de mensajes como único método de comunicación garantiza que los mecanismos de protección sean completos y eficientes. Integrando los mensajes como el sistema de memoria virtual, Mach también garantiza que los mensajes se puedan gestionar de manera eficiente. Finalmente, al hacer que el sistema de memoria virtual utilice mensajes para comunicarse con los demonios que gestionan el almacén de respaldo, Mach proporciona una gran flexibilidad en el diseño e implementación de estas tareas de gestión de objetos de memoria. Proporcionando llamadas al sistema de bajo nivel o primitivas, a partir de las cuales pueden construirse funciones más complejas, Mach reduce el tamaño del *kernel*, al mismo tiempo que permite emular sistemas operativos en el nivel del usuario de forma bastante similar a los sistemas de máquina virtual de IBM.

Las anteriores ediciones de este libro incluían un capítulo completo dedicado a Mach. Este capítulo, tal como aparecía en la cuarta edición, está disponible en la Web (<http://www.os-book.com>).

23.10 Otros sistemas

Por supuesto, existen otros sistemas operativos y la mayoría de ellos tienen propiedades interesantes. El sistema operativo MCP para la familia de computadoras Burroughs (McKeag y Wilson [1976]) fue el primero en ser escrito en un lenguaje de programación de sistemas. Soportaba mecanismos de segmentación y múltiples procesadores. El sistema operativo SCOPE para el CDC 6600 (McKeag y Wilson [1976]) también era un sistema multiprocesador. La coordinación y sincronización de los múltiples procesos estaban sorprendentemente bien diseñadas. Tenex (Bobrow et al. [1972]) fue un sistema de paginación bajo demanda pionero para el PDP-10, que ha tenido una gran influencia en los sistemas de tiempo compartido posteriores, como por ejemplo el TOPS-20 para DEC-20. El sistema operativo VMS para el VAX está basado en el sistema operativo RSX del PDP-11. CP/M fue el sistema operativo más común para las microcomputadoras de 8 bits, de los cuales quedan ya pocos ejemplares; MS-DOS es el sistema más común para las microcomputadoras de 16 bits. Las interfaces gráficas de usuario (GUI, Graphical user interface) se han hecho populares debido a que facilitan la utilización de las computadoras; el sistema operativo Macintosh y Microsoft Windows son los dos líderes en este área.

Ejercicios

- 23.1 Explique las consideraciones que el operador informático tenía en cuenta a la hora de decidir las secuencias con las que los programas debían ejecutarse en los primeros sistemas informáticos que se operaban de forma manual.
- 23.2 ¿Qué mecanismos de optimización se utilizaron para minimizar la discrepancia entre las velocidades de la CPU y de la E/S en los primeros sistemas informáticos?
- 23.3 Considere el algoritmo de sustitución de páginas utilizado por Atlas. ¿Cuáles son las diferencias respecto al algoritmo del reloj explicado en la Sección 9.4.5.2?
- 23.4 Considere la cola de realimentación multinivel utilizada por CTSS y MULTICS. Suponga que un programa utiliza constantemente siete unidades de tiempo cada vez que se le planifica,

antes de llevar a cabo una operación de E/S y bloquearse. ¿Cuántas unidades de tiempo se asignarán a este programa cuando se planifique para ejecución en diferentes momentos?

- 23.5 ¿Cuáles son las implicaciones de soportar la funcionalidad BSD mediante servidores de modo usuario dentro del sistema operativo Mach?

Bibliografía

- [Abbot 1984] C. Abbot, "Intervention Schedules for Real-Time Programming", *IEEE Transactions on Software Engineering*, Vol. SE-10, Nº 4 (1984), págs. 268-274.
- [Accett et al. 1986] M. Accett, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian y M. Young, "Mach: A New Kernel Foundation for Unix Development", *Proceedings of the Summer USENIX Conference* (1986), págs. 93-112.
- [Agrawal y Abbadi 1991] D. P. Agrawal y A. E. Abbadi, "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion", *ACM Transactions on Computer Systems*, Vol. 9, Nº 1 (1991), págs. 1-20.
- [Agre 2003] P. E. Agre, "P2P and the Promise of Internet Equality", *Communications of the ACM*, Vol. 46, Nº 2 (2003), págs. 39-42.
- [Ahituv et al. 1987] N. Ahituv, Y. Lapid y S. Neumann, "Processing Encrypted Data", *Communications of the ACM*, Vol. 30, Nº 9 (1987), páginas 777-780.
- [Ahmed 2000] I. Ahmed, "Cluster Computing: A Glance al Recent Events", *IEEE Concurrency*, Vol. 8, Nº 1 (2000).
- [Akl 1983] S. G. Akl, "Digital Signatures: A Tutorial Survey", *Computer*, Vol. 16, Nº 2 (1983), págs. 15-24.
- [Akyurek y Salem 1993] S. Akyurek y K. Salem, "Adaptative Block Rearrangement", *Proceedings of the International Conference on Data Engineering* (1993), págs. 182-189.
- [Alt 1993] H. Alt, "Removable Media in Solaris", *Proceedings of the Winter USENIX Conference* (1993), págs. 281-287.
- [Anderson 1990] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors", *IEEE Trans. Parallel Distrib. Syst.*, Vol. 1, Nº 1 (1990), págs. 6-16.
- [Anderson et al. 1989] T. E. Anderson, E. D. Lazowska y H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Computers*, Vol. 38, Nº 12 (1989), págs. 1631-1644.
- [Anderson et al. 1991] T. E. Anderson, B. N. Bershad, E. D. Lazowska y H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 95-109.
- [Anderson et al. 1995] T. E. Anderson, M. D., Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli y R. Y Wang, "Serverless Network File Systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1995), págs. 109-206.

- [Anderson et al. 2000] D. Anderson, J. Chase y A. Vahdat, "Interposed Request Routing for Scalable Network Storage", *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation* (2000).
- [Asthana and Finkelstein 1995] P. Asthana y B. Finkelstein, "Superdense Optical Storage", *IEEE Spectrum*, Vol. 32, Nº 8 (1995), páginas 25-31.
- [Audsley et al. 1991] N. C. Audsley, A. Burns, M. F. Richardson y A. J. Wellings, "Hard Real-Time Scheduling: The Deadline Monotonic Approach", *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software* (1991).
- [Axelsson 1999] S. Axelsson, "The Base-Rate Fallacy and Its Implications for Intrusion Detection", *Proceedings of the ACM Conference on Computer and Communications Security* (1999), págs. 1-7.
- [Babaoglu y Marzullo 1993] O. Babaoglu y K. Marzullo. "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms", págs. 55-56. Addison-Wesley (1993).
- [Bach 1987] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall (1987).
- [Back et al. 2000] G. Back, P. Tullman, L. Stoller, W. C. Hsieh y J. Lepreau, "Techniques for the Design of Java Operating Systems", *2000 USENIX Annual Technical Conference* (2000).
- [Baker et al. 1991] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff y J. K. Ousterhout, "Measurements of a Distributed File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 198-212.
- [Balakrishnan et al. 2003] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris e I. Stoica, "Looking Up Data in P2P Systems", *Communications of the ACM*, Vol. 46, Nº 2 (2003), págs. 43-48.
- [Baldwin 2002] J. Baldwin, "Locking in the Multithreaded FreeBSD Kernel", *USENIX BSD* (2002).
- [Basu et al. 1995] A. Basu, V. Buch, W. Vogels y T. von Eicken, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", *Proceedings of the ACM Symposium on Operating Systems Principles* (1995).
- [Bayer et. 1978] R. Bayer, R. M. Graham y G. Seegmuller, editores, *Operating Systems-An Advanced Course*, Springer Verlag (1978).
- [Bays 1977] C. Bays, "A Comparison of Next-Fit, First-Fit and Best-Fit", *Communications of the ACM*, Vol. 20, Nº 3 (1977), págs. 191-192.
- [Belady 1966] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer", *IBM Systems Journal*, Vol. 5, Nº 2 (1966), págs. 78-101.
- [Belady et al. 1969] L. A. Belady, R. A. Nelson y G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", *Communications of the ACM*, Vol. 12, Nº 6 (1969), págs. 349-353.
- [Bellovin 1989] S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite", *Computer Communications Review*, Vol. 19:2, (1989), págs. 32-48.
- [Ben-Ari 1990] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice Hall (1990).
- [Benjamin 1990] C. D. Benjamin, "The Role of Optical Storage Technology for NASA", *Proceedings, Storage and Retrieval Systems and Applications* (1990), págs. 10-17.
- [Bernstein y Goodman 1980] P. A. Bernstein y N. Goodman, "Time-Stamp-Based Algorithms for Concurrency Control in Distributed Database Systems", *Proceedings of the International Conference on Very Large Databases* (1980), págs. 285-300.

- [Bernstein et al. 1987] A. Bernstein, V. Hadzilacos y N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).
- [Bershad 1993] B. Bershad, "Practical Considerations for Non-Blocking Concurrent Objects", *IEEE International Conference on Distributed Computing Systems* (1993), págs. 264-273.
- [Bershad and Pinkerton 1988] B. N. Bershad y C. B. Pinkerton, "Watchdogs: Extending the Unix File System", *Proceedings of the Winter USENIX Conference* (1988).
- [Bershad et al. 1990] B. N. Bershad, T. E. Anderson, E. D. Lazowska y H. M. Levy, "Lightweight Remote Procedure Call", *ACM Transactions on Computer Systems*, Vol. 8, Nº 1 (1990), págs. 37-55.
- [Bershad et al. 1995] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers y C. Chambers, "Extensibility Safety and Performance in the SPIN Operating System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1995), págs. 267-284.
- [Beveridge y Wiener 1997] J. Beveridge y R. Wiener, *Multithreading Applications in Win32*, Addison-Wesley (1997).
- [Birrell 1989] A. D. Birrell, "An Introduction to Programming with Threads". Technical Report 35, DEC-SRC (1989).
- [Birrell y Nelson 1984] A. D. Birrell y B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, Nº 1 (1984), págs. 39-59.
- [Black 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, Vol. 23, Nº 5 (1990), págs. 35-43.
- [Blumofe y Leiserson 1994] R. Blumofe y C. Leiserson, "Scheduling Multithreaded Computations by Work Stealing", *Proceedings of the Annual Symposium on Foundations of Computer Science* (1994), págs. 356-368.
- [Bobrow et al. 1972] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy y R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, Vol. 15, Nº 3 (1972).
- [Bolosky et al. 1997] W. J. Bolosky, R. P. Fitzgerald y J. R. Douceur, "Distributed Schedule Management in the Tiger Video Fileserver", *Proceedings of the ACM Symposium on Operating Systems Principles* (1997), págs. 212-223.
- [Bonwick 1994] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator", *USENIX Summer* (1994), págs. 87-98.
- [Bonwick y Adams 2001] J. Bonwick y J. Adams, "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources", *Proceedings of the 2001 USENIX Annual Technical Conference* (2001).
- [Bovet y Cesati 2002] D. P. Bovet y M. Cesati, *Understanding the Linux Kernel, Second Edition*, O'Reilly & Associates (2002).
- [Brain 1996] M. Brain, *Win32 System Services, Second Edition*, Prentice Hall (1996).
- [Brent 1989] R. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation", *ACM Transactions on Programming Languages and Systems*, Vol. 11, Nº 3 (1989), págs. 388-403.
- [Brereton 1986] O. P. Brereton, "Management of Replicated Files in a UNIX Environment", *Software-Practice and Experience*, Vol. 16, (1986), páginas 771-780.
- [Brinch-Hansen 1970] P. Brinch-Hansen, "The Nucleus of a Multiprogramming System", *Communications of the ACM*, Vol. 13, Nº 4 (1970), págs. 238-241 y 250.
- [Brinch-Hansen 1972] P. Brinch-Hansen, "Structured Multiprogramming", *Communications of the ACM*, Vol. 15, Nº 7 (1972), págs. 574-578.

- [Brinch-Hansen 1973] P. Brinch-Hansen, *Operating System Principles*, Prentice Hall (1973).
- [Brookshear 2003] J. G. Brookshear, *Computer Science: An Overview, Seventh Edition*, Addison-Wesley (2003).
- [Brownbridge et al. 1982] D. R. Brownbridge, L. F. Marshall y B. Randell, "The Newcastle Connection or UNIXes of the World Unite!", *Software—Practice and Experience*, Vol. 12, Nº 12 (1982), págs. 1147-1162.
- [Burns 1978] J. E. Burns, "Mutual Exclusion with Linear Waiting Using Binary Shared Variables", *SIGACT News*, Vol. 10, Nº 2 (1978), págs. 42-47.
- [Butenhof 1997] D. Butenhof, *Programming with POSIX Threads*, Addison-Wesley (1997).
- [Buyya 1999] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*, Prentice Hall (1999).
- [Callaghan 2000] B. Callaghan, *NFS Illustrated*, Addison-Wesley (2000).
- [Calvert y Donahoo 2001] K. Calvert y M. Donahoo, *TCP/IP Sockets in Java: Practical Guide for Programmers*, Morgan Kaufmann (2001).
- [Cantrill et al. 2004] B. M. Cantrill, M. W. Shapiro y A. H. Leventhal, "Techniques for the Design of Java Operating Systems", *2004 USENIX Annual Technical Conference* (2004).
- [Carr y Hennessy 1981] W. R. Carr y J. L. Hennessy, "WSClock-A Simple and Effective Algorithm for Virtual Memory Management", *Proceedings of the ACM Symposium on Operating Systems Principles* (1981), págs. 87-95.
- [Carvalho y Roucairol 1983] O. S. Carvalho y G. Roucairol, "On Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. 26, Nº 2 (1983), págs. 146-147.
- [Chandy y Lamport 1985] K. M. Chandy y L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 3, Nº 1 (1985), págs. 63-75.
- [Chang 1980] E. Chang, "N-Philosophers: An Exercise in Distributed Control", *Computer Networks*, Vol. 4, Nº 2 (1980), págs. 71-76.
- [Chang y Mergen 1988] A. Chang y M. F. Mergen, "801 Storage: Architecture and Programming", *ACM Transactions on Computer Systems*, Vol. 6, Nº 1 (1988), págs. 28-50.
- [Chase et al. 1994] J. S. Chase, H. M. Levy, M. J. Feeley y E. D. Lazowska, "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems*, Vol. 12, Nº 4 (1994), págs. 271-307.
- [Chen et al. 1994] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz y D. A. Patterson, "RAID: High-Performance, Reliable Secondary Storage", *ACM Computing Survey*, Vol. 26, Nº 2 (1994), págs. 145-185.
- [Cheswick et al. 2003] W. Cheswick, S. Bellovin y A. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*, segunda edición, Addison-Wesley (2003).
- [Cheung y Loong 1995] W. H. Cheung y A. H. S. Loong, "Exploring Issues of Operating Systems Structuring: From Microkernel to Extensible Systems", *Operating Systems Review*, Vol. 29, (1995), págs. 4-16.
- [Chi 1982] C. S. Chi, "Advances in Computer Mass Storage Technology", *Computer*, Vol. 15, Nº 5 (1982), págs. 60-74.
- [Coffman et al. 1971] E. G. Coffman, M. J. Elphick y A. Shoshani, "System Deadlocks", *Computing Surveys*, Vol. 3, Nº 2 (1971), págs. 67-78.
- [Cohen y Jefferson 1975] E. S. Cohen y D. Jefferson, "Protection in the Hydra Operating System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1975), págs. 141-160.

- [Cohen y Woodring 1997] A. Cohen y M. Woodring, *Win32 Multithreaded Programming*, O'Reilly & Associates (1997).
- [Comer 1999] D. Comer, *Internetworking with TCP/IP, Volume II, Third Edition*, Prentice Hall (1999).
- [Comer 2000] D. Comer, *Internetworking with TCP/IP, Volume II, Fourth Edition*, Prentice Hall (2000).
- [Corbato y Vyssotsky 1965] F. J. Corbato y V. A. Vyssotsky, "Introduction and Overview of the MULTICS System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), págs. 185-196.
- [Corbato et al. 1962] F. J. Corbato, M. Merwin-Daggett y R. C. Daley, "An Experimental Time-Sharing System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), págs. 335-344.
- [Coulouris et al. 2001] G. Coulouris, J. Dollimore y T. Kindberg, *Distributed Systems Concepts and Designs, Third Edition*, Addison Wesley (2001).
- [Courtois et al. 1971] P. J. Courtois, F. Heymans y D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'", *Communications of the ACM*, Vol. 14, Nº 10 (1971), págs. 667-668.
- [Culler et al. 1998] D. E. Culler, J. P. Singh y A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc. (1998).
- [Custer 1994] H. Custer, *Inside the Windows NT File System*, Microsoft Press (1994).
- [Dabek et al. 2001] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris e I. Stoica, "Wide-Area Cooperative Storage with DFS", *Proceedings of the ACM Symposium on Operating Systems Principles* (2001), págs. 202-215.
- [Daley y Dennis 1967] R. C. Daley y J. B. Dennis, "Virtual Memory, Processes, and Sharing in Multics", *Proceedings of the ACM Symposium on Operating Systems Principles* (1967), págs. 121-128.
- [Davcev y Burkhard 1985] D. Davcev y W. A. Burkhard, "Consistency and Recovery Control for Replicated Files", *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), págs. 87-96.
- [Davies 1983] D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail", *Computer*, Vol. 16, Nº 2 (1983), págs. 55-62.
- [deBruijn 1967] N. G. deBruijn, "Additional Comments on a Problem in Concurrent Programming and Control", *Communications of the ACM*, Vol. 10, Nº 3 (1967), págs. 137-138.
- [Deitel 1990] H. M. Deitel, *An Introduction to Operating Systems, Second Edition*, Addison-Wesley (1990).
- [Denning 1968] P. J. Denning, "The Working Set Model for Program Behavior", *Communications of the ACM*, Vol. 11, Nº 5 (1968), págs. 323-333.
- [Denning 1982] D. E. Denning, *Cryptography and Data Security*, Addison-Wesley (1982).
- [Denning 1983] D. E. Denning, "Protecting Public Keys and Signature Keys", *Computer*, Vol. 16, Nº 2 (1983), págs. 27-35.
- [Denning 1984] D. E. Denning, "Digital Signatures with RSA and Other Public-Key Cryptosystems", *Communications of the ACM*, Vol. 27, Nº 4 (1984), págs. 388-392.
- [Denning y Denning 1979] D. E. Denning y P. J. Denning, "Data Security", *ACM Comput. Surv.*, Vol. 11, Nº 3 (1979), págs. 227-249.
- [Dennis 1965] J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems", *Communications of the ACM*, Vol. 8, Nº 4 (1965), págs. 589-602.
- [Dennis y Horn 1966] J. B. Dennis y E. C. V. Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM*, Vol. 9, Nº 3 (1966), págs. 143-155.

- [Di Pietro y Mancini 2003] R. Di Pietro y L. V. Mancini, "Security and Privacy Issues of Handheld and Wearable Wireless Devices", *Communications of the ACM*, Vol. 46, Nº 9 (2003), págs. 74-79.
- [Diffie y Hellman 1979] W. Diffie y M. E. Hellman, "Privacy and Authentication", *Proceedings of the IEEE*(1979), págs. 397-427.
- [Dijkstra 1965a] E. W. Dijkstra. "Cooperating Sequential Processes". Technical Report, Technological University, Eindhoven, Holanda (1965).
- [Dijkstra 1965b] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, Nº 9 (1965), pág. 569.
- [Dijkstra 1968] E. W. Dijkstra, "The Structure of the THE Multiprogramming System", *Communications of the ACM*, Vol. 11, Nº 5 (1968), págs. 341-346.
- [Dijkstra 1971] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes", *Acta Informatica*, Vol. 1, Nº 2 (1971), págs. 115-138.
- [DoD 1985] *Trusted Computer System Evaluation Criteria*. Department of Defense (1985).
- [Dougan et al. 1999] C. Dougan, P. Mackerras y V. Yodaiken, "Optimizing the Idle Task and Other MMU Tricks", *Proceedings of the Symposium on Operating System Design and Implementation* (1999).
- [Douglis y Ousterhout 1991] F. Douglis y J. K. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation", *software*, Vol. 21, Nº 8 (1991), págs. 757-785.
- [Douglis et al. 1994] F. Douglis, F. Kaashoek, K. Li, R. Caceres, B. Marsh y J. A. Tauber, "Storage Alternatives for Mobile Computers", *Proceedings of the Symposium on Operating Systems Design and Implementation*(1994), págs. 25-37.
- [Douglis et al. 1995] F. Douglis, P. Krishnan y B. Bershad, "Adaptive Disk Spin-Down Policies for Mobile Computers", *Proceedings of the USENIX Symposium on Mobile and Location Independent Computing* (1995), págs. 121-137.
- [Draves et al. 1991] R. P. Draves, B. N . Bershad, R. F. Rashid y R. W. Dean, "Using continuations to implement thread management and communication in operating systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 122-136.
- [Druschel y Peterson 1993] P. Druschel y L. L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", *Proceedings of the ACM Symposium on Operating Systems Principles* (1993), págs. 189-202.
- [Eastlake 1999] D. Eastlake, "Domain Name System Security Extensions", *Network Working Group, Request for Comments: 2535* (1999).
- [Eisenberg y McGuire 1972] M. A. Eisenberg y M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", *Communications of the ACM*, Vol. 15, Nº 11 (1972), pág. 999.
- [Ekanadham y Bernstein 1979] K. Ekanadham y A. J. Bernstein, "Conditional Capabilities", *IEEE Transactions on Software Engineering*, Vol. SE-5, Nº 5 (1979), págs. 458-464.
- [Engelschall 2000] R. Engelschall, "Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation", *Proceedings of the 2000 USENIX Annual Technical Conference* (2000).
- [Eswaran et al. 1976] K- P. Eswaran, H. N. Gray, R. A. Lorie e I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Vol. 19, Nº 11 (1976), págs. 624-633.
- [Fang et al. 2001] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh y S. A. McKee, "Reevaluating Online Superpage Promotion with Hardware Support", *Proceedings of the International Symposium on High-Performance Computer Architecture*, Vol. 50, Nº 5 (2001).

- [Farrow 1986a]** R. Farrow, "Security for Superusers, or How to Break the UNIX System", *UNIX World* (Mayo 1986), págs. 65-70.
- [Farrow 1986b]** R. Farrow, "Security Issues and Strategies for Users", *UNIX World* (Abril 1986), págs. 65-71.
- [Feitelson y Rudolph 1990]** D. Feitelson y L. Rudolph, "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control", *Proceedings of the International Conference on Parallel Processing* (1990).
- [Fidge 1991]** C. Fidge, "Logical Time in Distributed Computing Systems", *Computer*, Vol. 24, Nº 8 (1991), págs. 28-33.
- [Filipski y Hanko 1986]** A. Filipski y J. Hanko, "Making UNIX Secure", *Byte* (Abril 1986), págs. 113-128.
- [Fisher 1981]** J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, Vol. 30, Nº 7 (1981), págs. 478-490.
- [Folk y Zoellick 1987]** M. J. Folk y B. Zoellick, *File Structures*, Addison-Wesley (1987).
- [Forrest et al. 1996]** S. Forrest, S. A. Hofmeyr y T. A. Longstaff, "A Sense of Self for UNIX Processes", *Proceedings of the IEEE Symposium on Security and Privacy* (1996), págs. 120-128.
- [Fortier 1989]** P. J. Fortier, *Handbook of LAN Technology*, McGraw-Hill (1989).
- [FreeBSD 1999]** FreeBSD, *FreeBSD Handbook*, The FreeBSD Documentation Project (1999).
- [Freedman 1983]** D. H. Freedman, "Searching for Denser Disks", *Infosystems* (1983), pág. 56.
- [Fuhrt 1994]** B. Fuhrt, "Multimedia Systems: An Overview", *IEEE MultiMedia*, Vol. 1, Nº 1 (1994), págs. 47-59.
- [Fujitani 1984]** L. Fujitani, "Laser Optical Disk: The Coming Revolution in On-Line Storage", *Communications of the ACM*, Vol. 27, Nº 6 (1984), páginas 546-554.
- [Gait 1988]** J. Gait, "The Optical File Cabinet: A Random-Access File System for Write-On Optical Disks", *Computer*, Vol. 21, Nº 6 (1988).
- [Ganapathy y Schimmel 1998]** N. Ganapathy y C. Schimmel, "General Purpose Operating System Support for Multiple Page Sizes", *Proceedings of the USENIX Technical Conference* (1998).
- [Ganger et al. 2002]** G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt y T. Pinckney, "Fast and Flexible Application-Level Networking on Exokernel Systems", *ACM Transactions on Computer Systems*, Vol. 20, Nº 1 (2002), págs. 49-83.
- [García-Molina 1982]** H. García-Molina, "Elections in Distributed Computing Systems", *IEEE Transactions on Computers*, Vol. c-31, Nº 1 (1982).
- [Garfinkel et al. 2003]** S. Garfinkel, G. Spafford y A. Schwartz, *Practical UNIX & Internet Security*, O'Reilly & Associates (2003).
- [Gibson et al. 1997a]** G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg y J. Zelenka. "Filesystems for Network-Attached Secure Disks". Technical Report, CMU-CS-97-112 (1997).
- [Gibson et al. 1997b]** G. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg y J. Zelenka, "File Server Scaling with Network-Attached Secure Disks", *Measurement and Modeling of Computer Systems* (1997), págs. 272-284.
- [Gifford 1982]** D. K. Gifford, "Cryptographic Sealing for Information Secrecy and Authentication", *Communications of the ACM*, Vol. 25, Nº 4 (1982), págs. 274-286.
- [Goldberg et al. 1996]** I. Goldberg, D. Wagner, R. Thomas y E. A. Brewer, "A Secure Environment for Untrusted Helper Applications", *Proceedings of the 6th Usenix Security Symposium* (1996).

- [Golden y Pechura 1986]** D. Golden y M. Pechura, "The Structure of Microcomputer File Systems", *Communications of the ACM*, Vol. 29, Nº 3 (1986), páginas 222-230.
- [Golding et al. 1995]** R. A. Golding, P. B. II, C. Staelin, T. Sullivan y J. Wilkes, "Idleness is Not Sloth", *USENIX Winter* (1995), págs. 201-212.
- [Golm et al. 2002]** M. Golm, M. Felser, C. Wawersich y J. Kleinoder, "The JX Operating System", *2002 USENIX Annual Technical Conference* (2002).
- [Gong et al. 1997]** L. Gong, M. Mueller, H. Prafullchandra y R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2", *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (1997).
- [Goodman et al. 1989]** J. R. Goodman, M. K. Vernon y P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (1989), págs. 64-75.
- [Gosling et al. 1996]** J. Gosling, B. Joy y G. Steele, *The Java Language Specification*, Addison-Wesley (1996).
- [Govindan y Anderson 1991]** R. Govindan y D. P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 68-80.
- [Grampp y Morris 1984]** F. T. Grampp y R. H. Morris, "UNIX Operating-System Security", *AT&T Bell Laboratories Technical Journal*, Vol. 63, (1984), págs. 1649-1672.
- [Gray 1978]** J. N. Gray, "Notes on Data Base Operating Systems", in **[Bayer et al. 1978]** (1978), págs. 393-481.
- [Gray 1981]** J. N. Gray, "The Transaction Concept: Virtues and Limitations", *Proceedings of the International Conference on Very Large Databases* (1981), págs. 144-154.
- [Gray 1997]** J. Gray, *Interprocess Communications in UNIX*, Prentice Hall (1997).
- [Gray et al. 1981]** J. N. Gray, P. R. McJones y M. Blasgen, "The Recovery Manager of the System R Database Manager", *ACM Computing Survey*, Vol. 13, Nº 2 (1981), págs. 223-242.
- [Greenawalt 1994]** P. Greenawalt, "Modeling Power Management for Hard Disks", *Proceedings of the Symposium on Modeling and Simulation of Computer Telecommunication Systems* (1994), págs. 62-66.
- [Grosshans 1986]** D. Grosshans, *File Systems Design and Implementation*, Prentice Hall (1986).
- [Grosso 2002]** W. Grosso, *Java RMI*, O'Reilly & Associates (2002).
- [Habermann 1969]** A. N. Habermann, "Prevention of System Deadlocks", *Communications of the ACM*, Vol. 12, Nº 7 (1969), págs. 373-377, 385.
- [Hall et al. 1996]** L. Hall, D. Shmoys y J. Wein, "Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms", *SODA: ACM-SIAM Symposium on Discrete Algorithms* (1996).
- [Halsall 1992]** F. Halsall, *Data Communications, Computer Networks and Open Systems*, Addison-Wesley (1992).
- [Hamacher et al. 2002]** C. Hamacher, Z. Vranesic y S. Zaky, *Computer Organization, Fifth Edition*, McGraw-Hill (2002).
- [Han y Ghosh 1998]** K. Han y S. Ghosh, "A Comparative Analysis of Virtual Versus Physical Process-Migration Strategies for Distributed Modeling and Simulation of Mobile Computing Networks", *Wireless Networks*, Vol. 4, Nº 5 (1998), págs. 365-378.

- [Hansen y Atkins 1993] S. E. Hansen y E. T. Atkins, "Automated System Monitoring and Notification With Swatch", *Proceedings of the USENIX Systems Administration Conference* (1993).
- [Harchol-Balter y Downey 1997] M. Harchol-Balter y A. B. Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing", *ACM Transactions on Computer Systems*, Vol. 15, Nº 3 (1997), págs. 253-285.
- [Harish y Owens 1999] V. C. Harish y B. Owens, "Dynamic Load Balancing DNS", *Linux Journal*, Vol. 1999, Nº 64 (1999).
- [Harker et al. 1981] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana y L. G. Taft, "A Quarter Century of Disk File Innovation", *IBM Journal of Research and Development*, Vol. 25, Nº 5 (1981), págs. 677-689.
- [Harrison et al. 1976] M. A. Harrison, W. L. Ruzzo y J. D. Ullman, "Protection in Operating Systems", *Communications of the ACM*, Vol. 19, Nº 8 (1976), págs. 461-471.
- [Hartman y Ousterhout 1995] J. H. Hartman y J. K. Ousterhout, "The Zebra Striped Network File System", *ACM Transactions on Computer Systems*, Vol. 13, Nº 3 (1995), págs. 274-310.
- [Havender 1968] J.W. Havender, "Avoiding Deadlock in Multitasking Systems", *IBM Systems Journal*, Vol. 7, Nº 2 (1968), págs. 74-84.
- [Hecht et al. 1988] M. S. Hecht, A. Johri, R. Aditham y T. J. Wei, "Experience Adding C2 Security Features to UNIX", *Proceedings of the Summer USENIX Conference* (1988), págs. 133-146.
- [Hennessy y Patterson 2002] J. L. Hennessy y D. A. Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*, Morgan Kaufmann Publishers (2002).
- [Henry 1984] G. Henry, "The Fair Share Scheduler", *AT&T Bell Laboratories Technical Journal* (1984).
- [Herlihy 1993] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 15, Nº 5 (1993), págs. 745-770.
- [Herlihy y Moss 1993] M. Herlihy y J. E. B. Moss, "Transactional Memory: Architectural Support For Lock-Free Data Structures", *Proceedings of the Twentieth Annual International Symposium on Computer Architecture* (1993).
- [Hitz et al. 1995] D. Hitz, J. Lau y M. Malcolm, "File System Design for an NFS File Server Appliance", *Technical Report TR3002* (http://www.netapp.com/tech_library/3002.html), NetApp (1995).
- [Hoagland 1985] A. S. Hoagland, "Information Storage Technology – A Look at the Future", *Computer*, Vol. 18, Nº 7 (1985), págs. 60-68.
- [Hoare 1972] C. A. R. Hoare, "Towards a Theory of Parallel Programming", in [Hoare y Perrott 1972] (1972), páginas 61-71.
- [Hoare 1974] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, Vol. 17, Nº 10 (1974), págs. 549-557.
- [Hoare y Perrott 1972] C. A. R. Hoare y R. H. Perrott, editors, *Operating Systems Techniques*, Academic Press (1972).
- [Holt 1971] R. C. Holt, "Comments on Prevention of System Deadlocks", *Communications of the ACM*, Vol. 14, Nº 1 (1971), págs. 36-38.
- [Holt 1972] R. C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, Nº 3 (1972), págs. 179-196.
- [Holub 2000] A. Holub, *Taming Java Threads*, Apress (2000).

- [Hong et al. 1989] J. Hong, X. Tan y D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System", *IEEE Transactions on Computers*, Vol. 38, Nº 12 (1989), págs. 1736-1744.
- [Howard et al. 1988] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan y R. N. Sidebotham, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, Vol. 6, Nº 1 (1988), págs. 55-81.
- [Howarth et al. 1961] D. J. Howarth, R. B. Payne y F. H. Sumner, "The Manchester University Atlas Operating System, Part II: User's Description", *Computer Journal*, Vol. 4, Nº 3 (1961), págs. 226-229.
- [Hsiao et al. 1979] D. K. Hsiao, D. S. Kerr y S. E. Madnick, *Computer Security*, Academic Press (1979).
- [Hu y Perrig 2004] Y.-C. Hu y A. Perrig, "SPV: A Secure Path Vector Routing Scheme for Securing BGP", *Proceedings of ACM SIGCOMM Conference on Data Communication* (2004).
- [Hu et al. 2002] Y.-C. Hu, A. Perrig y D. Johnson, "Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks", *Proceedings of the Annual International Conference on Mobile Computing and Networking* (2002).
- [Hyman 1985] D. Hyman, *The Columbus Chicken Statute and More Bonehead Legislation*, S. Greene Press (1985).
- [Iacobucci 1988] E. Iacobucci, *OS/2 Programmer's Guide*, Osborne McGraw-Hill (1988).
- [IBM 1983] *Technical Reference*. IBM Corporation (1983).
- [Iliffe y Jodeit 1962] J. K. Iliffe y J. G. Jodeit, "A Dynamic Storage Allocation System", *Computer Journal*, Vol. 5, Nº 3 (1962), págs. 200-209.
- [Intel 1985a] *iAPX 286 Programmer's Reference Manual*. Intel Corporation (1985).
- [Intel 1985b] *iAPX 86/88, 186/188 User's Manual Programmer's Reference*. Intel Corporation (1985).
- [Intel 1986] *iAPX 386 Programmer's Reference Manual*. Intel Corporation (1986).
- [Intel 1990] *i486 Microprocessor Programmer's Reference Manual*. Intel Corporation (1990).
- [Intel 1993] *Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual*. Intel Corporation (1993).
- [Iseminger 2000] D. Iseminger, *Active Directory Services for Microsoft Windows 2000. Technical Reference*, Microsoft Press (2000).
- [Jacob y Mudge 1997] B. Jacob y T. Mudge, "Software-Managed Address Translation", *Proceedings of the International Symposium on High Performance Computer Architecture and Implementation* (1997).
- [Jacob y Mudge 1998a] B. Jacob y T. Mudge, "Virtual Memory in Contemporary Microprocessors", *IEEE Micro Magazine*, Vol. 18, (1998), páginas 60-75.
- [Jacob y Mudge 1998b] B. Jacob y T. Mudge, "Virtual Memory: Issues of Implementation", *IEEE Computer Magazine*, Vol. 31, (1998), págs. 33-43.
- [Jacob y Mudge 2001] B. Jacob y T. Mudge, "Uniprocessor Virtual Memory Without TLBs", *IEEE Transactions on Computers*, Vol. 50, Nº 5 (2001).
- [Jacobson y Wilkes 1991] D. M. Jacobson y J. Wilkes. "Disk Scheduling Algorithms Based on Rotational Position". Technical Report HPL-CSP-91-7 (1991).
- [Jensen et al. 1985] E. D. Jensen, C. D. Locke y H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems", *Proceedings of the IEEE Real-Time Systems Symposium* (1985), págs. 112-122.

- [Johnstone y Wilson 1998] M. S. Johnstone y P. R. Wilson, "The Memory Fragmentation Problem: Solved?", *Proceedings of the First International Symposium on Memory management* (1998), págs. 26-36.
- [Jones y Liskov 1978] A. K. Jones y B. H. Liskov, "A Language Extension for Expressing Constraints on Data Access", *Communications of the ACM*, Vol. 21, Nº 5 (1978), págs. 358-367.
- [Jul et al. 1988] E. Jul, H. Levy, N. Hutchinson y A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, Vol. 6, Nº 1 (1988), págs. 109-133.
- [Kaashoek et al. 1997] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti y K. Mackenzie, "Application performance and flexibility on exokernel systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1997), págs. 52-65.
- [Katz et al. 1989] R. H. Katz, G. A. Gibson y D. A. Patterson, "Disk System Architectures for High Performance Computing", *Proceedings of the IEEE* (1989).
- [Kay y Lauder 1988] J. Kay y P. Lauder, "A Fair Share Scheduler", *Communications of the ACM*, Vol. 31, Nº 1 (1988), págs. 44-55.
- [Kent et al. 2000] S. Kent, C. Lynn y K. Seo, "Secure Border Gateway Protocol (Secure-BGP)", *IEEE Journal on Selected Areas in Communications*, Vol. 18, Nº 4 (2000), páginas 582-592.
- [Kenville 1982] R. F. Kenville, "Optical Disk Data Storage", *Computer*, Vol. 15, Nº 7 (1982), págs. 21-26.
- [Kessels 1977] J. L. W. Kessels, "An Alternative to Event Queues for Synchronization in Monitors", *Communications of the ACM*, Vol. 20, Nº 7 (1977), págs. 500-503.
- [Khanna et al. 1992] S. Khanna, M. Sebree y J. Zolnowsky, "Realtime Scheduling in SunOS 5.0", *Proceedings of the Winter USENIX Conference* (1992), págs. 375-390.
- [Kieburz y Silberschatz 1978] R. B. Kieburz y A. Silberschatz, "Capability Managers", *IEEE Transactions on Software Engineering*, Vol. SE-4, Number 6 (1978), págs. 467-477.
- [Kieburz y Silberschatz 1983] R. B. Kieburz y A. Silberschatz, "Access Right Expressions", *ACM Transactions on Programming Languages and Systems*, Vol. 5, Nº 1 (1983), páginas 78-96.
- [Kilburn et al. 1961] T. Kilburn, D. J. Howarth, R. B. Payne y F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization", *Computer Journal*, Vol. 4, Nº 3 (1961), págs. 222-225.
- [Kim y Spafford 1993] G. H. Kim y E. H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker", *Technical Report, Purdue University* (1993).
- [King 1990] R. P. King, "Disk Arm Movement in Anticipation of Future Requests", *ACM Transactions on Computer Systems*, Vol. 8, Nº 3 (1990), págs. 214-229.
- [Kistler y Satyanarayanan 1992] J. Kistler y M. Satyanarayanan, "Disconnected Operation in the Coda File System", *ACM Transactions on Computer Systems*, Vol. 10, Nº 1 (1992), págs. 3-25.
- [Kleinrock 1975] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*, Wiley-Interscience (1975).
- [Knapp 1987] E. Knapp, "Deadlock Detection in Distributed Databases", *Computing Surveys*, Vol. 19, Nº 4 (1987), págs. 303-328.
- [Knowlton 1965] K. C. Knowlton, "A Fast Storage Allocator", *Communications of the ACM*, Vol. 8, Nº 10 (1965), págs. 623-624.
- [Knuth 1966] D. E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 9, Nº 5 (1966), págs. 321-322.

Bibliografía

- [Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition*, Addison-Wesley (1973).
- [Koch 1987] P. D. L. Koch, "Disk File Allocation Based on the Buddy System", *ACM Transactions on Computer Systems*, Vol. 5, Nº 4 (1987), págs. 352-370.
- [Kopetz y Reisinger 1993] H. Kopetz y J. Reisinger, "The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronisation Problem", *IEEE Real-Time Systems Symposium* (1993), págs. 131-137.
- [Kosaraju 1973] S. Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", *Operating Systems Review*, Vol. 7, Nº 4 (1973), págs. 122-126.
- [Kramer 1988] S. M. Kramer, "Retaining SUID Programs in a Secure UNIX", *Proceedings of the Summer USENIX Conference* (1988), págs. 107-118.
- [Kubiatowicz et al. 2000] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells y B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage", *Proc. of Architectural Support for Programming Languages and Operating Systems* (2000).
- [Kurose y Ross 2005] J. Kurose y K. Ross, *Computer Networking – A Top-Down Approach Featuring the Internet, Third Edition*, Addison-Wesley (2005).
- [Lamport 1974] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem", *Communications of the ACM*, Vol. 17, Nº 8 (1974), págs. 453-455.
- [Lamport 1976] L. Lamport, "Synchronization of Independent Processes", *Acta Informatica*, Vol. 7, Nº 1 (1976), págs. 15-34.
- [Lamport 1977] L. Lamport, "Concurrent Reading and Writing", *Communications of the ACM*, Vol. 20, Nº 11 (1977), págs. 806-811.
- [Lamport 1978a] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems", *Computer Networks*, Vol. 2, Nº 2 (1978), págs. 95-114.
- [Lamport 1978b] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, Nº 7 (1978), págs. 558-565.
- [Lamport 1981] L. Lamport, "Password Authentication with Insecure Communications", *Communications of the ACM*, Vol. 24, Nº 11 (1981), págs. 770-772.
- [Lamport 1986] L. Lamport, "The Mutual Exclusion Problem", *Communications of the ACM*, Vol. 33, Nº 2 (1986), págs. 313-348.
- [Lamport 1987] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, Nº 1 (1987), págs. 1-11.
- [Lamport 1991] L. Lamport, "The Mutual Exclusion Problem Has Been Solved", *Communications of the ACM*, Vol. 34, Nº 1 (1991), págs. 110.
- [Lamport et al. 1982] L. Lamport, R. Shostak y M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, Vol. 4, Nº 3 (1982), págs. 382-401.
- [Lampson 1969] B. W. Lampson, "Dynamic Protection Structures", *Proceedings of the AFIPS Fall Joint Computer Conference* (1969), págs. 27-38.
- [Lampson 1971] B. W. Lampson, "Protection", *Proceedings of the Fifth Annual Princeton Conference on Information Systems Science* (1971), págs. 437-443.
- [Lampson 1973] B. W. Lampson, "A Note on the Confinement Problem", *Communications of the ACM*, Vol. 10, Nº 16 (1973), págs. 613-615.
- [Lampson y Redell 1979] B.W. Lampson y D. D. Redell, "Experience with Processes and Monitors in Mesa", *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)* (1979), págs. 43-44.

- [Lampson y Sturgis 1976] B. Lampson y H. Sturgis, "Crash Recovery in a Distributed Data Storage System", *Technical Report, Xerox Research Center* (1976).
- [Landwehr 1981] C. E. Landwehr, "Formal Models of Computer Security", *Computing Surveys*, Vol. 13, Nº 3 (1981), págs. 247-278.
- [Lann 1977] G. L. Lann, "Distributed Systems – Toward a Formal Approach", *Proceedings of the IFIP Congress* (1977), págs. 155-160.
- [Larson y Kajla 1984] P. Larson y A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access", *Communications of the ACM*, Vol. 27, Nº 7 (1984), págs. 670-677.
- [Lauzac et al. 2003] S. Lauzac, R. Melhem y D. Mosse, "An Improved Rate-Monotonic Admission Control and Its Applications", *IEEE Transactions on Computers*, Vol. 52, Nº 3 (2003).
- [Lee 2003] J. Lee, "An End-User Perspective on File-Sharing Systems", *Communications of the ACM*, Vol. 46, Nº 2 (2003), págs. 49-53.
- [Lee y Thekkath 1996] E. K. Lee y C. A. Thekkath, "Petal: Distributed Virtual Disks", *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), págs. 84-92.
- [Leffler et al. 1989] S. J. Leffler, M. K. McKusick, M. J. Karels y J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley (1989).
- [Lehmann 1987] F. Lehmann, "Computer Break-Ins", *Communications of the ACM*, Vol. 30, Nº 7 (1987), págs. 584-585.
- [Lehoczky et al. 1989] J. Lehoczky, L. Sha y Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour", *Proceedings of 10th IEEE Real-Time Systems Symposium* (1989).
- [Lempel 1979] A. Lempel, "Cryptology in Transition", *Computing Surveys*, Vol. 11, Nº 4 (1979), págs. 286-303.
- [Leslie et al. 1996] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns y E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications", *IEEE Journal of Selected Areas in Communications*, Vol. 14, Nº 7 (1996), págs. 1280-1297.
- [Lett y Konigsford 1968] A. L. Lett y W. L. Konigsford, "TSS/360: A Time-Shared Operating System", *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), págs. 15-28.
- [Leutenegger y Vernon 1990] S. Leutenegger y M. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", *Proceedings of the Conference on Measurement and Modeling of Computer Systems* (1990).
- [Levin et al. 1975] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack y W. A. Wulf, "Policy/Mechanism Separation in Hydra", *Proceedings of the ACM Symposium on Operating Systems Principles* (1975), págs. 132-140.
- [Levine 2003] G. Levine, "Defining Deadlock", *Operating Systems Review*, Vol. 37, Nº 1 (2003).
- [Lewis y Berg 1998] B. Lewis y D. Berg, *Multithreaded Programming with Pthreads*, Sun Microsystems Press (1998).
- [Lewis y Berg 2000] B. Lewis y D. Berg, *Multithreaded Programming with Java Technology*, Sun Microsystems Press (2000).
- [Lichtenberger y Pirtle 1965] W. W. Lichtenberger y M. W. Pirtle, "A Facility for Experimentation in Man-Machine Interaction", *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), págs. 589-598.

- [Lindholm y Yellin 1999] T. Lindholm y F. Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley (1999).
- [Ling et al. 2000] Y. Ling, T. Mullen y X. Lin, “Analysis of Optimal Thread Pool Size”, *Operating System Review*, Vol. 34, Nº 2 (2000).
- [Lipner 1975] S. Lipner, “A Comment on the Confinement Problem”, *Operating System Review*, Vol. 9, Nº 5 (1975), págs. 192-196.
- [Lipton 1974] R. Lipton. “On Synchronization Primitive Systems”. Ph.D. Thesis, Carnegie-Mellon University (1974).
- [Liskov 1972] B. H. Liskov, “The Design of the Venus Operating System”, *Communications of the ACM*, Vol. 15, Nº 3 (1972), págs. 144-149.
- [Liu y Layland 1973] C. L. Liu y J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment”, *Communications of the ACM*, Vol. 20, Nº 1 (1973), págs. 46-61.
- [Lobel 1986] J. Lobel, *Foiling the System Breakers: Computer Security and Access Control*, McGraw-Hill (1986).
- [Loo 2003] A.W. Loo, “The Future of Peer-to-Peer Computing”, *Communications of the ACM*, Vol. 46, Nº 9 (2003), págs. 56-61.
- [Love 2004] R. Love, *Linux Kernel Development*, Developer’s Library (2004).
- [Lowney et al. 1993] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O’Donnell y J. C. Ruttenberg, “The Multiflow Trace Scheduling Compiler”, *Journal of Supercomputing*, Vol. 7, Nº 1-2 (1993), págs. 51-142.
- [Lucco 1992] S. Lucco, “A Dynamic Scheduling Method for Irregular Parallel Programs”, *Proceedings of the Conference on Programming Language Design and Implementation* (1992), págs. 200-211.
- [Ludwig 1998] M. Ludwig, *The Giant Black Book of Computer Viruses, Second Edition*, American Eagle Publications (1998).
- [Ludwig 2002] M. Ludwig, *The Little Black Book of Email Viruses*, American Eagle Publications (2002).
- [Lumb et al. 2000] C. Lumb, J. Schindler, G. R. Ganger, D. F. Nagle y E. Riedel, “Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives”, *Symposium on Operating Systems Design and Implementation* (2000).
- [Maekawa 1985] M. Maekawa, “A Square Root Algorithm for Mutual Exclusion in Decentralized Systems”, *ACM Transactions on Computer Systems*, Vol. 3, Nº 2 (1985), págs. 145-159.
- [Maher et al. 1994] C. Maher, J. S. Goldick, C. Kerby y B. Zumach, “The Integration of Distributed File Systems and Mass Storage Systems”, *Proceedings of the IEEE Symposium on Mass Storage Systems* (1994), págs. 27-31.
- [Marsh et al. 1991] B. D. Marsh, M. L. Scott, T. J. LeBlanc y E. P. Markatos, “First-Class User-Level Threads”, *Proceedings of the 13th ACM Symposium on Operating Systems Principle* (1991), págs. 110-121.
- [Massalin y Pu 1989] H. Massalin y C. Pu, “Threads and Input/Output in the Synthesis Kernel”, *Proceedings of the ACM Symposium on Operating Systems Principles* (1989), págs. 191-200.
- [Mattern 1988] F. Mattern, “Virtual Time and Global States of Distributed Systems”, *Workshop on Parallel and Distributed Algorithms* (1988).
- [Mattson et al. 1970] R. L. Mattson, J. Gecsei, D. R. Slutz y I. L. Traiger, “Evaluation Techniques for Storage Hierarchies”, *IBM Systems Journal*, Vol. 9, Nº 2 (1970), págs. 78-117.

- [Mauro y McDougall 2001] J. Mauro y R. McDougall, *Solaris Internals: Core Kernel Architecture*, Prentice Hall (2001).
- [McCanne y Jacobson 1993] S. McCanne y V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", *USENIX Winter* (1993), págs. 259–270.
- [McGraw y Andrews 1979] J. R. McGraw y G. R. Andrews, "Access Control in Parallel Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, Nº 1 (1979), págs. 1–9.
- [McKeag y Wilson 1976] R. M. McKeag y R. Wilson, *Studies in Operating Systems*, Academic Press (1976).
- [McKeon 1985] B. McKeon, "An Algorithm for Disk Caching with Limited Memory", *Byte*, Vol. 10, Nº 9 (1985), págs. 129–138.
- [McKusick et al. 1984] M. K. McKusick, W. N. Joy, S. J. Leffler y R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, Vol. 2, Nº 3 (1984), págs. 181–197.
- [McKusick et al. 1996] M. K. McKusick, K. Bostic y M. J. Karels, *The Design and Implementation of the 4.4 BSD UNIX Operating System*, John Wiley and Sons (1996).
- [McVoy y Kleiman 1991] L.W. McVoy y S. R. Kleiman, "Extent-like Performance from a UNIX File System", *Proceedings of the Winter USENIX Conference* (1991), págs. 33–44.
- [Mealy et al. 1966] G. H. Mealy, B. I. Witt y W. A. Clark, "The Functional Structure of OS/360", *IBM Systems Journal*, Vol. 5, Nº 1 (1966).
- [Mellor-Crummey y Scott 1991] J. M. Mellor-Crummey y M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, Number 1 (1991), páginas 21–65.
- [Menasce y Muntz 1979] D. Menasce y R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases", *IEEE Transactions on Software Engineering*, Vol. SE-5, Nº 3 (1979), págs. 195–202.
- [Mercer et al. 1994] C.W. Mercer, S. Savage y H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", *International Conference on Multimedia Computing and Systems* (1994), págs. 90–99.
- [Meyer y Seawright 1970] R. A. Meyer y L. H. Seawright, "A Virtual Machine Time-Sharing System", *IBM Systems Journal*, Vol. 9, Nº 3 (1970), págs. 199–218.
- [Microsoft 1986] Microsoft MS-DOS User's Reference and Microsoft MS-DOS Programmer's Reference. Microsoft Press (1986).
- [Microsoft 1996] Microsoft Windows NT Workstation Resource Kit. MicrosoftPress (1996).
- [Microsoft 2000a] Microsoft Developer Network Development Library. Microsoft Press (2000).
- [Microsoft 2000b] Microsoft Windows 2000 Server Resource Kit. Microsoft Press (2000).
- [Microsystems 1995] S. Microsystems, *Solaris Multithreaded Programming Guide*, Prentice Hall (1995).
- [Milenkovic 1987] M. Milenkovic, *Operating Systems: Concepts and Design*, McGraw-Hill (1987).
- [Miller y Katz 1993] E. L. Miller y R. H. Katz, "An Analysis of File Migration in a UNIX Supercomputing Environment", *Proceedings of the Winter USENIX Conference* (1993), págs. 421–434.
- [Milojicic et al. 2000] D. S. Milojevic, F. Douglis, Y. Paindaveine, R. Wheeler y S. Zhou, "Process Migration", *ACM Comput. Surv.*, Vol. 32, Nº 3 (2000), págs. 241–299.
- [Mockapetris 1987] P. Mockapetris, "Domain Names—Concepts and Facilities", *Network Working Group, Request for Comments: 1034* (1987).

- [Mohan y Lindsay 1983]** C. Mohan y B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proceedings of the ACM Symposium on Principles of Database Systems* (1983).
- [Mok 1983]** A. K. Mok. "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment". Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA (1983).
- [Morris 1973]** J. H. Morris, "Protection in Programming Languages", *Communications of the ACM*, Vol. 16, Nº 1 (1973), págs. 15-21.
- [Morris y Thompson 1979]** R. Morris y K. Thompson, "Password Security: A Case History", *Communications of the ACM*, Vol. 22, Nº 11 (1979), págs. 594-597.
- [Morris et al. 1986]** J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S.H. Rosenthal y F. D. Smith, "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, Vol. 29, Nº 3 (1986), págs. 184-201.
- [Morshedian 1986]** D. Morshedian, "How to Fight Password Pirates", *Computer*, Vol. 19, Nº 1 (1986).
- [Motorola 1993]** *PowerPC 601 RISC Microprocessor User's Manual*. Motorola Inc. (1993).
- [Myers y Beigl 2003]** B. Myers y M. Beigl, "Handheld Computing", *Computer*, Vol. 36, Nº 9 (2003), págs. 27-29.
- [Navarro et al. 2002]** J. Navarro, S. Lyer, P. Druschel y A. Cox, "Practical, Transparent Operating System Support for Superpages", *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (2002).
- [Needham y Walker 1977]** R. M. Needham y R. D. H. Walker, "The Cambridge CAP Computer and Its Protection System", *Proceedings of the Sixth Symposium on Operating System Principles* (1977), págs. 1-10.
- [Nelson et al. 1988]** M. Nelson, B. Welch y J. K. Ousterhout, "Caching in the Sprite Network File System", *ACM Transactions on Computer Systems*, Vol. 6, Nº 1 (1988), págs. 134-154.
- [Norton y Wilton 1988]** P. Norton y R. Wilton, *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*, Microsoft Press (1988).
- [Nutt 2004]** G. Nutt, *Operating Systems: A Modern Perspective, Third Edition*, Addison-Wesley (2004).
- [Oaks y Wong 1999]** S. Oaks y H. Wong, *Java Threads, Second Edition*, O'Reilly & Associates (1999).
- [Obermarck 1982]** R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, Nº 2 (1982), págs. 187-208.
- [O'Leary y Kitts 1985]** B. T. O'Leary y D. L. Kitts, "Optical Device for a Mass Storage System", *Computer*, Vol. 18, Nº 7 (1985).
- [Olsen y Kenley 1989]** R. P. Olsen y G. Kenley, "Virtual Optical Disks Solve the On-Line Storage Crunch", *Computer Design*, Vol. 28, Nº 1 (1989), págs. 93-96.
- [Organick 1972]** E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).
- [Ortiz 2001]** S. Ortiz, "Embedded OSs Gain the Inside Track", *Computer*, Vol. 34, Nº 11 (2001).
- [Ousterhout 1991]** J. Ousterhout. "The Role of Distributed State". In CMU Computer Science: a 25th Anniversary Commemorative (1991), R. F. Rashid, Ed., Addison-Wesley (1991).
- [Ousterhout et al. 1985]** J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer y J. G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), págs. 15-24.

- [Ousterhout et al. 1988] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson y B. B. Welch, "The Sprite Network-Operating System", *Computer*, Vol. 21, Nº 2 (1988), págs. 23-36.
- [Parameswaran et al. 2001] M. Parameswaran, A. Susarla y A. B. Whinston, "P2P Networking: An Information-Sharing Alternative", *Computer*, Vol. 34, Nº 7 (2001).
- [Parmelee et al. 1972] R. P. Parmelee, T. I. Peterson, C. C. Tillman y D. Hatfield, "Virtual Storage and Virtual Machine Concepts", *IBM Systems Journal*, Vol. 11, Nº 2 (1972), págs. 99-130.
- [Parnas 1975] D. L. Parnas, "On a Solution to the Cigarette Smokers' Problem Without Conditional Statements", *Communications of the ACM*, Vol. 18, Nº 3 (1975), págs. 181-183.
- [Patil 1971] S. Patil. "Limitations y Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes". Technical Report, MIT (1971).
- [Patterson et al. 1988] D. A. Patterson, G. Gibson y R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988).
- [Pease et al. 1980] M. Pease, R. Shostak y L. Lamport, "Reaching Agreement in the Presence of Faults", *Communications of the ACM*, Vol. 27, Nº 2 (1980), págs. 228-234.
- [Pechura y Schoeffler 1983] M. A. Pechura y J. D. Schoeffler, "Estimating File Access Time of Floppy Disks", *Communications of the ACM*, Vol. 26, Nº 10 (1983), págs. 754-763.
- [Perlman 1988] R. Perlman, *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology (1988).
- [Peterson 1981] G. L. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, Nº 3 (1981).
- [Peterson y Davie 1996] L. L. Peterson y B. S. Davie, *Computer Networks: a Systems Approach*, Morgan Kaufmann Publishers Inc. (1996).
- [Peterson y Norman 1977] J. L. Peterson y T. A. Norman, "Buddy Systems", *Communications of the ACM*, Vol. 20, Nº 6 (1977), páginas 421-431.
- [Pfleeger y Pfleeger 2003] C. Pfleeger y S. Pfleeger, *Security in Computing, Third Edition*, Prentice Hall (2003).
- [Philbin et al. 1996] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas y K. Li, "Thread Scheduling for Cache Locality", *Architectural Support for Programming Languages and Operating Systems* (1996), págs. 60-71.
- [Pinilla y Gill 2003] R. Pinilla y M. Gill, "JVM: Platform Independent vs. Performance Dependent", *Operating System Review* (2003).
- [Polychronopoulos y Kuck 1987] C. D. Polychronopoulos y D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", *IEEE Transactions on Computers*, Vol. 36, Nº 12 (1987), págs. 1425-1439.
- [Popek 1974] G. J. Popek, "Protection Structures", *Computer*, Vol. 7, Nº 6 (1974), páginas 22-33.
- [Popek y Walker 1985] G. Popek y B. Walker, editors, *The LOCUS Distributed System Architecture*, MIT Press (1985).
- [Prieve y Fabry 1976] B. G. Prieve y R. S. Fabry, "VMIN – An Optimal Variable Space Page-Replacement Algorithm", *Communications of the ACM*, Vol. 19, Nº 5 (1976), págs. 295-297.
- [Psaltis y Mok 1995] D. Psaltis y F. Mok, "Holographic Memories", *Scientific American*, Vol. 273, Nº 5 (1995), págs. 70-76.
- [Purdin et al. 1987] T. D. M. Purdin, R. D. Schlichting y G. R. Andrews, "A File Replication Facility for Berkeley UNIX", *Software – Practice and Experience*, Vol. 17, (1987), págs. 923-940.

- [Purdom, Jr. y Stigler 1970] P. W. Purdom, Jr. y S. M. Stigler, "Statistical Properties of the Buddy System", *J. ACM*, Vol. 17, Nº 4 (1970), págs. 683–697.
- [Quinlan 1991] S. Quinlan, "A Cached WORM", *Software – Practice and Experience*, Vol. 21, Nº 12 (1991), págs. 1289–1299.
- [Rago 1993] S. Rago, *UNIX System V Network Programming*, Addison-Wesley (1993).
- [Rashid 1986] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network Operating System", *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986).
- [Rashid y Robertson 1981] R. Rashid y G. Robertson, "Accent: A Communication-Oriented Network Operating System Kernel", *Proceedings of the ACM Symposium on Operating System Principles* (1981).
- [Raynal 1986] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press (1986).
- [Raynal 1991] M. Raynal, "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms", *Operating Systems Review*, Vol. 25, Nº 1 (1991), págs. 47–50.
- [Raynal y Singhal 1996] M. Raynal y M. Singhal, "Logical Time: Capturing Causality in Distributed Systems", *Computer*, Vol. 29, Nº 2 (1996), págs. 49–56.
- [Reddy y Wyllie 1994] A. L. N. Reddy y J. C. Wyllie, "I/O issues in a Multimedia System", *Computer*, Vol. 27, Nº 3 (1994), págs. 69–74.
- [Redell y Fabry 1974] D. D. Redell y R. S. Fabry, "Selective Revocation of Capabilities", *Proceedings of the IRIA International Workshop on Protection in Operating Systems* (1974), págs. 197–210.
- [Reed 1983] D. P. Reed, "Implementing Atomic Actions on Decentralized Data", *ACM Transactions on Computer Systems*, Vol. 1, Nº 1 (1983), págs. 3–23.
- [Reed y Kanodia 1979] D. P. Reed y R. K. Kanodia, "Synchronization with Eventcounts and Sequences", *Communications of the ACM*, Vol. 22, Nº 2 (1979), págs. 115–123.
- [Regehr et al. 2000] J. Regehr, M. B. Jones y J. A. Stankovic, "Operating System Support for Multimedia: The Programming Model Matters", *Technical Report MSR-TR-2000-89, Microsoft Research* (2000).
- [Reid 1987] B. Reid, "Reflections on Some Recent Widespread Computer Break-Ins", *Communications of the ACM*, Vol. 30, Nº 2 (1987), págs. 103–105.
- [Ricart y Agrawala 1981] G. Ricart y A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", *Communications of the ACM*, Vol. 24, Nº 1 (1981), págs. 9–17.
- [Richards 1990] A. E. Richards, "A File System Approach for Integrating Removable Media Devices and Jukeboxes", *Optical Information Systems*, Vol. 10, Nº 5 (1990), págs. 270–274.
- [Richter 1997] J. Richter, *Advanced Windows*, Microsoft Press (1997).
- [Riedel et al. 1998] E. Riedel, G. A. Gibson y C. Faloutsos, "Active Storage for Large-Scale Data Mining and Multimedia", *Proceedings of 24th International Conference on Very Large Data Bases* (1998), págs. 62–73.
- [Ripeanu et al. 2002] M. Ripeanu, A. Imrnitchi y I. Foster, "Mapping the Gnutella Network", *IEEE Internet Computing*, Vol. 6, Nº 1 (2002).
- [Rivest et al. 1978] R. L. Rivest, A. Shamir y L. Adleman, "On Digital Signatures and Public Key Cryptosystems", *Communications of the ACM*, Vol. 21, Nº 2 (1978), págs. 120–126.
- [Rodeheffer y Schroeder 1991] T. L. Rodeheffer y M. D. Schroeder, "Automatic reconfiguration in Autonet", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 183–97.

- [Rosenblum y Ousterhout 1991] M. Rosenblum y J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1991), págs. 1-15.
- [Rosenkrantz et al. 1978] D. J. Rosenkrantz, R. E. Stearns y P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems", *ACM Transactions on Database Systems*, Vol. 3, Nº 2 (1978), págs. 178-198.
- [Ruemmler y Wilkes 1991] C. Ruemmler y J. Wilkes. "Disk Shuffling". Technical Report, Hewlett-Packard Laboratories (1991).
- [Ruemmler y Wilkes 1993] C. Ruemmler y J. Wilkes, "Unix Disk Access Patterns", *Proceedings of the Winter USENIX Conference* (1993), páginas 405-420.
- [Ruemmler y Wilkes 1994] C. Ruemmler y J. Wilkes, "An Introduction to Disk Drive Modeling", *Computer*, Vol. 27, Nº 3 (1994), págs. 17-29.
- [Rushby 1981] J. M. Rushby, "Design and Verification of Secure Systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1981), págs. 12-21.
- [Rushby y Randell 1983] J. Rushby y B. Randell, "A Distributed Secure System", *Computer*, Vol. 16, Nº 7 (1983), págs. 55-67.
- [Russell y Gangemi 1991] D. Russell y G. T. Gangemi, *Computer Security Basics*, O'Reilly & Associates (1991).
- [Saltzer y Schroeder 1975] J. H. Saltzer y M. D. Schroeder, "The Protection of Information in Computer Systems", *Proceedings of the IEEE* (1975), págs. 1278-1308.
- [Sandberg 1987] R. Sandberg, *The Sun Network File System: Design, Implementation and Experience*, Sun Microsystems (1987).
- [Sandberg et al. 1985] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh y B. Lyon, "Design and Implementation of the Sun Network Filesystem", *Proceedings of the Summer USENIX Conference* (1985), págs. 119-130.
- [Sargent y Shoemaker 1995] M. Sargent y R. Shoemaker, *The Personal Computer from the Inside Out, Third Edition*, Addison-Wesley (1995).
- [Sarisky 1983] L. Sarisky, "Will Removable Hard Disks Replace the Floppy?", *Byte* (1983), págs. 110-117.
- [Satyanarayanan 1990] M. Satyanarayanan, "Scalable, Secure and Highly Available Distributed File Access", *Computer*, Vol. 23, Nº 5 (1990), págs. 9-21.
- [Savage et al. 2000] S. Savage, D. Wetherall, A. R. Karlin y T. Anderson, "Practical Network Support for IP Traceback", *Proceedings of ACM SIGCOMM Conference on Data Communication* (2000), páginas 295-306.
- [Schell 1983] R. R. Schell, "A Security Kernel for a Multiprocessor Microcomputer", *Computer* (1983), págs. 47-53.
- [Schindler y Gregory 1999] J. Schindler y G. Gregory, "Automated Disk Drive Characterization", *Technical Report, Carnegie-Mellon University* (1999).
- [Schlichting y Schneider 1982] R. D. Schlichting y F. B. Schneider, "Understanding and Using Asynchronous Message Passing Primitives", *Proceedings of the Symposium on Principles of Distributed Computing* (1982), págs. 141-147.
- [Schneider 1982] F. B. Schneider, "Synchronization in Distributed Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 4, Nº 2 (1982), págs. 125-148.
- [Schneier 1996] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley and Sons (1996).

- [Schrage 1967]** L. E. Schrage, "The Queue M/G/1 with Feedback to Lower Priority Queues", *Management Science*, Vol. 13, (1967), págs. 466–474.
- [Schwarz y Mattern 1994]** R. Schwarz y F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail", *Distributed Computing*, Vol. 7, Nº 3 (1994), páginas 149–174.
- [Seely 1989]** D. Seely, "Password Cracking: A Game of Wits", *Communications of the ACM*, Vol. 32, Nº 6 (1989), páginas 700–704.
- [Seltzer et al. 1990]** M. Seltzer, P. Chen y J. Ousterhout, "Disk Scheduling Revisited", *Proceedings of the Winter USENIX Conference* (1990), págs. 313–323.
- [Seltzer et al. 1993]** M. I. Seltzer, K. Bostic, M. K. McKusick y C. Staelin, "An Implementation of a Log-Structured File System for UNIX", *USENIX Winter* (1993), págs. 307–326.
- [Seltzer et al. 1995]** M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains y V. N. Padmanabhan, "File System Logging versus Clustering: A Performance Comparison", *USENIX Winter* (1995), págs. 249–264.
- [Shrivastava y Panzieri 1982]** S. K. Shrivastava y F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism", *IEEE Transactions on Computers*, Vol. C-31, Nº 7 (1982), págs. 692–697.
- [Silberschatz et al. 2001]** A. Silberschatz, H. F. Korth y S. Sudarshan, *Database System Concepts*, Fourth Edition, McGraw-Hill (2001).
- [Silverman 1983]** J. M. Silverman, "Reflections on the Verification of the Security of an Operating System Kernel", *Proceedings of the ACM Symposium on Operating Systems Principles* (1983), págs. 143–154.
- [Silvers 2000]** C. Silvers, "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD", *USENIX Annual Technical Conference – FREENIX Track* (2000).
- [Simmons 1979]** G. J. Simmons, "Symmetric and Asymmetric Encryption", *Computing Surveys*, Vol. 11, Nº 4 (1979), págs. 304–330.
- [Sincerbox 1994]** G. T. Sincerbox, editor, *Selected Papers on Holographic Storage*, Optical Engineering Press (1994).
- [Singhal 1989]** M. Singhal, "Deadlock Detection in Distributed Systems", *Computer*, Vol. 22, Nº 11 (1989), págs. 37–48.
- [Sirer et al. 1999]** E. G. Sirer, R. Grimm, A. J. Gregory y B. N. Bershad, "Design and Implementation of a Distributed Virtual Machine for Networked Computers", *Symposium on Operating Systems Principles* (1999), páginas 202–216.
- [Smith 1982]** A. J. Smith, "Cache Memories", *ACM Computing Surveys*, Vol. 14, Nº 3 (1982), págs. 473–530.
- [Smith 1985]** A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems*, Vol. 3, Nº 3 (1985), págs. 161–203.
- [Sobti et al. 2004]** S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy y R. Wang, "Segank: A Distributed Mobile Storage System", *Proceedings of the Third USENIX Conference on File and Storage Technologies* (2004).
- [Solomon 1998]** D. A. Solomon, *Inside Windows NT*, Second Edition, Microsoft Press (1998).
- [Solomon y Russinovich 2000]** D. A. Solomon y M. E. Russinovich, *Inside Microsoft Windows 2000*, Third Edition, Microsoft Press (2000).
- [Spafford 1989]** E. H. Spafford, "The Internet Worm: Crisis and Aftermath", *Communications of the ACM*, Vol. 32, Nº 6 (1989), págs. 678–687.

- [Spector y Schwarz 1983] A. Z. Spector y P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", *ACM SIGOPS Operating Systems Review*, Vol. 17, Nº 2 (1983), págs. 18-35.
- [Stallings 2000a] W. Stallings, *Local and Metropolitan Area Networks*, Prentice Hall (2000).
- [Stallings 2000b] W. Stallings, *Operating Systems, Fourth Edition*, Prentice Hall (2000).
- [Stallings 2003] W. Stallings, *Cryptography and Network Security: Principles and Practice, Third Edition*, Prentice Hall (2003).
- [Stankovic 1982] J. S. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus Messages", *Computer*, Vol. 15, Nº 4 (1982).
- [Stankovic 1996] J. A. Stankovic, "Strategic Directions in Real-Time and Embedded Systems", *ACM Computing Surveys*, Vol. 28, Nº 4 (1996), págs. 751-763.
- [Staunstrup 1982] J. Staunstrup, "Message Passing Communication Versus Procedure Call Communication", *Software – Practice and Experience*, Vol. 12, Nº 3 (1982), págs. 223-234.
- [Steinmetz 1995] R. Steinmetz, "Analyzing the Multimedia Operating System", *IEEE MultiMedia*, Vol. 2, Nº 1 (1995), págs. 68-84.
- [Stephenson 1983] C. J. Stephenson, "Fast Fits: A New Method for Dynamic Storage Allocation", *Proceedings of the Ninth Symposium on Operating Systems Principles* (1983), páginas 30-32.
- [Stevens 1992] R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley (1992).
- [Stevens 1994] R. Stevens, *TCP/IP Illustrated Volume 1: The Protocols*, Addison-Wesley (1994).
- [Stevens 1995] R. Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley (1995).
- [Stevens 1997] W. R. Stevens, *UNIX Network Programming – Volume I*, Prentice Hall (1997).
- [Stevens 1998] W. R. Stevens, *UNIX Network Programming – Volume II*, Prentice Hall (1998).
- [Stevens 1999] W. R. Stevens, *UNIX Network Programming Interprocess Communications – Volume 2*, Prentice Hall (1999).
- [Stoica et al. 1996] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke y G. Plaxton, "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems", *IEEE Real-Time Systems Symposium* (1996).
- [Su 1982] Z. Su, "A Distributed System for Internet Name Service", *Network Working Group, Request for Comments: 830* (1982).
- [Sugerman et al. 2001] J. Sugerman, G. Venkitachalam y B. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", *2001 USENIX Annual Technical Conference* (2001).
- [Sun 1990] *Network Programming Guide*. Sun Microsystems (1990).
- [Svobodova 1984] L. Svobodova, "File Servers for Network-Based Distributed Systems", *ACM Computing Survey*, Vol. 16, Nº 4 (1984), págs. 353-398.
- [Talluri et al. 1995] M. Talluri, M. D. Hill y Y. A. Khalidi, "A New Page Table for 64-bit Address Spaces", *Proceedings of the ACM Symposium on Operating Systems Principles* (1995).
- [Tamches y Miller 1999] A. Tamches y B. P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels", *USENIX Symposium on Operating Systems Design and Implementation* (1999).
- [Tanenbaum 1990] A. S. Tanenbaum, *Structured Computer Organization, Third Edition*, Prentice Hall (1990).

- [Tanenbaum 2001] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall (2001).
- [Tanenbaum 2003] A. S. Tanenbaum, *Computer Networks, Fourth Edition*, Prentice Hall (2003).
- [Tanenbaum y Van Renesse 1985] A. S. Tanenbaum y R. Van Renesse, "Distributed Operating Systems", *ACM Computing Survey*, Vol. 17, Nº 4 (1985), págs. 419–470.
- [Tanenbaum y van Steen 2002] A. Tanenbaum y M. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall (2002).
- [Tanenbaum y Woodhull 1997] A. S. Tanenbaum y A. S. Woodhull, *Operating System Design and Implementation, Second Edition*, Prentice Hall (1997).
- [Tate 2000] S. Tate, *Windows 2000 Essential Reference*, New Riders (2000).
- [Tay y Ananda 1990] B. H. Tay y A. L. Ananda, "A Survey of Remote Procedure Calls", *Operating Systems Review*, Vol. 24, Nº 3 (1990), págs. 68–79.
- [Teorey y Pinkerton 1972] T. J. Teorey y T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies", *Communications of the ACM*, Vol. 15, Nº 3 (1972), págs. 177–184.
- [Tevanian et al. 1987a] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper y M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the Summer USENIX Conference* (1987).
- [Tevanian et al. 1987b] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky y R. Sanzi. "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach". Technical Report, Carnegie-Mellon University (1987).
- [Tevanian et al. 1989] A. Tevanian, Jr. y B. Smith, "Mach: The Model for Future Unix", *Byte* (1989).
- [Thekkath et al. 1997] C. A. Thekkath, T. Mann y E. K. Lee, "Frangipani: A Scalable Distributed File System", *Symposium on Operating Systems Principles* (1997), páginas 224–237.
- [Thompson 1984] K. Thompson, "Reflections on Trusting Trust", *Communications of ACM*, Vol. 27, Nº 8 (1984), págs. 761–763.
- [Thorn 1997] T. Thorn, "Programming Languages for Mobile Code", *ACM Computing Surveys*, Vol. 29, Nº 3 (1997), págs. 213–239.
- [Toigo 2000] J. Toigo, "Avoiding a Data Crunch", *Scientific American*, Vol. 282, Nº 5 (2000), págs. 58–74.
- [Traiger et al. 1982] I. L. Traiger, J. N. Gray, C. A. Galtieri y B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems", *ACM Transactions on Database Systems*, Vol. 7, Nº 3 (1982), págs. 323–342.
- [Tucker y Gupta 1989] A. Tucker y A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proceedings of the ACM Symposium on Operating Systems Principles* (1989).
- [Tudor 1995] P. N. Tudor. "MPEG-2 video compression tutorial". IEEE Coloquium on MPEG-2 - What it is and What it isn't (1995).
- [Vahalia 1996] U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).
- [Vee y Hsu 2000] V. Vee y W. Hsu, "Locality-Preserving Load-Balancing Mechanisms for Synchronous Simulations on Shared-Memory Multiprocessors", *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation* (2000), págs. 131–138.
- [Venners 1998] B. Venners, *Inside the Java Virtual Machine*, McGraw-Hill (1998).
- [Wah 1984] B. W. Wah, "File Placement on Distributed Computer Systems", *Computer*, Vol. 17, Nº 1 (1984), págs. 23–32.

- [Wahbe et al. 1993a] R. Wahbe, S. Lucco, T. E. Anderson y S. L. Graham, "Efficient Software-Based Fault Isolation", *ACM SIGOPS Operating Systems Review*, Vol. 27, Nº 5 (1993), págs. 203–216.
- [Wahbe et al. 1993b] R. Wahbe, S. Lucco, T. E. Anderson y S. L. Graham, "Efficient Software-Based Fault Isolation", *ACM SIGOPS Operating Systems Review*, Vol. 27, Nº 5 (1993), págs. 203–216.
- [Wallach et al. 1997] D. S. Wallach, D. Balfanz, D. Dean y E. W. Felten, "Extensible Security Architectures for Java", *Proceedings of the ACM Symposium on Operating Systems Principles* (1997).
- [Wilkes et al. 1996] J. Wilkes, R. Golding, C. Staelin y T. Sullivan, "The HP AutoRAID Hierarchical Storage System", *ACM Transactions on Computer Systems*, Vol. 14, Nº 1 (1996), págs. 108–136.
- [Williams 2001] R. Williams, *Computer Systems Architecture – A Networking Approach*, Addison-Wesley (2001).
- [Williams 2002] N. Williams, "An Implementation of Scheduler Activations on the NetBSD Operating System", *2002 USENIX Annual Technical Conference, FREENIX Track* (2002).
- [Wilson et al. 1995] P. R. Wilson, M. S. Johnstone, M. Neely y D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review", *Proceedings of the International Workshop on Memory Management* (1995), págs. 1–116.
- [Wolf 2003] W. Wolf, "A Decade of Hardware/Software Codesign", *Computer*, Vol. 36, Nº 4 (2003), págs. 38–43.
- [Wood y Kochan 1985] P. Wood y S. Kochan, *UNIX System Security*, Hayden (1985).
- [Woodside 1986] C. Woodside, "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers", *IEEE Transactions on Software Engineering*, Vol. SE-12, Nº 10 (1986), págs. 1041–1048.
- [Worthington et al. 1994] B. L. Worthington, G. R. Ganger y Y. N. Patt, "Scheduling Algorithms for Modern Disk Drives", *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1994), págs. 241–251.
- [Worthington et al. 1995] B. L. Worthington, G. R. Ganger, Y. N. Patt y J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems* (1995), págs. 146–156.
- [Wulf 1969] W. A. Wulf, "Performance Monitors for Multiprogramming Systems", *Proceedings of the ACM Symposium on Operating Systems Principles* (1969), págs. 175–181.
- [Wulf et al. 1981] W. A. Wulf, R. Levin y S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill (1981).
- [Yeong et al. 1995] W. Yeong, T. Howes y S. Kille, "Lightweight Directory Access Protocol", *Network Working Group, Request for Comments: 1777* (1995).
- [Young et al. 1987] M. Young, A. Tevanian, R. Rashid, D. Golub y J. Eppinger, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proceedings of the ACM Symposium on Operating Systems Principles* (1987), págs. 63–76.
- [Yu et al. 2000] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy y T. E. Anderson, "Trading Capacity for Performance in a Disk Array", *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation* (2000), págs. 243–258.
- [Zabatta y Young 1998] F. Zabatta y K. Young, "A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor", *Proceedings of the 2nd USENIX Windows NT Symposium* (1998).

- [Zahorjan y McCann 1990] J. Zahorjan y C. McCann, "Processor Scheduling in Shared-Memory Multiprocessors", *Proceedings of the Conference on Measurement and Modeling of Computer Systems* (1990).
- [Zapata y Asokan 2002] M. Zapata y N. Asokan, "Securing Ad Hoc Routing Protocols", *Proc. 2002 ACM Workshop on Wireless Security* (2002).
- [Zhao 1989] W. Zhao, editor, *Special Issue on Real-Time Operating Systems*, *Operating System Review* (1989).

Créditos

Figura 1.9: de Hennessy y Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, © 2002, Morgan Kaufmann Publishers, Figura 5.3, pág. 394. Reimpreso con permiso del editor.

Figura 3.9: de Iacobucci, *OS/2 Programmer's Guide*, © 1988, McGraw-Hill, Inc., Nueva York, Nueva York. Figura 1.7, pág. 20. Reimpreso con permiso del editor.

Figura 6.8: de Khanna/Sebree/Zolnowsky, "Realtime Scheduling in SunOS 5.0" Proceedings of Winter USENIX, enero 1992, San Francisco, California. Reimpreso con permiso de los autores.

Figura 6.10 adaptada con permiso de Sun Microsystems, Inc.

Figura 9.21: del manual *80386 Programmer's Reference Manual*, Figura 5-12, pág. 5-12. Reimpreso con permiso de Intel Corporation, Copyright, / Intel Corporation 1986.

Figura 10.16: de *IBM Systems Journal*, Vol. 10, Nº 3, © 1971, International Business Machines Corporation. Reimpreso con permiso de IBM Corporation.

Figura 12.9: de Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, © 1989 por Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figura 7.6, pág. 196. Reimpreso con permiso del editor.

Figura 13.9: de *Pentium Processor User's Manual: Architecture and Programming Manual*, Vol. 3, Copyright 1993. Reimpresó con permiso de Intel Corporation.

Figuras 15.4, 15.5, y 15.7: de Halsall, *Data Communications, Computer Networks, and Open Systems*, Third Edition, © 1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Figura 1.9, pág. 14, Figura 1.10, pág. 15 y Figura 1.11, pág. 18. Reimpreso con permiso del editor.

Secciones de los capítulos 7 y 17 de Silberschatz/Korth, *Database System Concepts*, Third Edition, Copyright 1997, McGraw-Hill, Inc., Nueva York, Nueva York. Sección 13.5, págs. 451-454, 14.1.1, págs. 471-742, 14.1.3, págs. 476-479, 14.2, págs. 482-485, 15.2.1, págs. 512-513, 15.4, págs. 517-518, 15.4.3, págs. 523-524, 18.7, págs. 613-617, 18.8, págs. 617-622. Reimpreso con permiso del editor.

Figura A.1: de Quarterman/Wilhelm, *UNIX, POSIX and Open Systems: The Open Standards Puzzle*, © 1993, por Addison-Wesley Publishing Co., Inc. Reading, Massachusetts. Figura 2.1, pág. 31. Reimpreso con permiso del editor.



índice

- .NET Framework, 62
100BaseT Ethernet, 563
10BaseT Ethernet, 563
16-bits, entorno Windows de, 740
2PC protocolo. Véase Protocolo de confirmación en dos fases
32-bits, entorno Windows de, 740-741
- A**
- Absoluto, nombre de ruta, 349
Acceso:
anónimo, 357
control de acceso en Linux, 708-709
controlado, 361
derechos de, 486, 496-497
directo (archivos), 342-343
directo a memoria (DMA), 10, 457-458
latencia de, 439
secuencial (archivos), 342
Acceso a memoria, tiempo efectivo de, 261
Acceso aleatorio, 653
tiempo de (discos), 408
Acceso efectivo, tiempo de, 287
Acceso remoto a archivos (sistemas de archivos distribuidos), 590-594
coherencia, 593-594
esquema básico de caché, 590-591
mecanismo de caché y servicio remoto, 594
política de actualización de la caché, 592-593
ubicación de la caché, 591-592
a archivos, 337, 361-363
Aceleración de los cálculos, 558
Acíclico, grafo, 351
ACL (access-control list), 362
Acorazado, virus, 520
Activaciones del planificador, 127-128
Active Directory (Windows XP), 755
Administración de archivos, 49
llamadas al sistema, 46-48
Administración de dispositivos, llamadas al sistema, 48
Admisión, control de, 657, 664
AES (advanced encryption standard), 528
Afinidad del procesador, 152
Afinidad dura, 152
Afinidad suave, 152
Agrupación en cluster, 579
asimétrico, 13
en Windows XP, 323
Agrupamiento, 387
Agujeros, 253
Ajuste automático del conjunto de trabajo (Windows XP), 323
Algoritmo
de control de admisión, 642
de firma digital, 531
de imposición, 623-624
de los bits de referencia adicionales, 299
de planificación:
 de colas multinivel realimentadas, 150-151
 mediante colas multinivel, 148-150
 por prioridad monótona en tasa, 642-644
 por prioridades, 145-146
 por turnos, 146-148
de política (Linux), 693
de seguridad, 230
de solicitud de recursos, 230-231
de sustitución local (algoritmo se sustitución basado en prioridades), 306
del anillo, 624-625
del banquero, 229-232
del reloj, 299
Almacén de respaldo, 249
Almacenamiento. Véase también Almacenamiento masivo,
estructura de
de utilidad, 431
densidad de, 446
estable, 198
holográfico, 435
no volátil, 9, 198
secundario, 8, 369. Véase también Discos
terciario, 21
volátil, 9-10, 198
Almacenamiento en caché, 22-23, 467
del lado del cliente, 754
doble, 390

Almacenamiento en caché (Cont.)

escritura diferida, 592
y servicios remotos, 594

Almacenamiento masivo, estructura de, 407-409
algoritmos de planificación de disco, 412-417

C-SCAN, 416
FCFS, 413
LOOK, 416
SCAN, 415
selección, 416-417
SSTF, 413-415

almacenamiento terciario, 433-442

cintas magnéticas, 434-435

discos extraíbles, 433-434

rendimiento, 438-442

soporte del sistema operativo, 435-438

tecnologías futuras, 435

cintas magnéticas, 409

conexión de un disco:

conectado a la red, 411-412

conectado al *host*, 410-411

redes de área de almacenamiento, 412

de reserva en caliente, 430

discos magnéticos, 407-409

estructura de disco, 409

extensiones, 431

gestión de disco:

bloque de arranque, 419-420

bloques defectuosos, 420-21

formateo de discos, 418-419

gestión del espacio de intercambio, 421-423

implementación de un almacenamiento estable, 432-433

RAID, estructuras, 423-432

mejora de la fiabilidad, 424-425

mejoras en las prestaciones, 425-426

problemas, 432

RAID, niveles, 426-431

Alta disponibilidad, 13**Ámbito de contienda del proceso, 154****Ámbito de contienda del sistema, 154****Amenazas relacionadas con los programas, 513-520**

bomba lógica, 514-515

caballo de Troya, 513-514

desbordamiento de pila y de búfer, 515-518

puerta trasera, 514

virus, 518-520

Amplificación de derechos (Hydra), 498**Análisis de desperdicios, 512****Análisis de redes de colas, 163****Ancho de banda:**

disco, 412

efectivo, 438

sostenido, 438

Anclar, 735**Andrew, sistema de archivos (AFS), 598-602**

espacio de nombres compartido, 599-600

implementación, 601-602

operaciones con archivos, 600-601

Anónima, memoria, 423**Anónimo, acceso, 357****APC (asynchronous procedure calls), 125, 720****APC. Véase Llamadas asíncronas a procedimientos****API Win32, 312-313, 713-714, 741**

API. Véase Interfaz de programación de aplicaciones

Apple Computers, 38**AppleTalk, protocolo, 751****Aprobación, 549****Apropiación de recursos; recuperación de interbloqueos mediante, 235****Apropiativa, planificación, 139-140****Árbol B+ (NTFS), 743-744****Árboles de dominio, 755****Archivado en cinta, 435****Archivo mapeado en memoria, 727****Archivo, objeto, 376, 696****Archivo, virus de, 519****Archivos, 20, 333-334. Véase también Directorios**

atributos de, 334-335

bloqueo de, 337-339

codificados, 654

compartidos inmutables, 360

de paginación (Windows XP), 726

de procesamiento por lotes, 339

definición, 333

ejecutables, 74

estructura de almacenamiento para, 345

estructura interna de, 340-341

extensiones de, 339-340

métodos de acceso, 342-345

acceso directo, 342-343

acceso secuencial, 342

operaciones con, 335-339

protección, 361-365

mediante acceso a archivos, 361-363

mediante contraseñas/permisos, 363-364

recuperación de, 392-393

Área de registro, 745**Área de reinicio, 745****Arista de asignación, 220****Arista de declaración, 228****Arista de solicitud, 220****ARP (address resolution protocol), 580****Arquitectura, 11-14**

basada en bus, 10

basada en comutador, 10

de Windows XP, 717-718

sistemas de un solo procesador, 11-13

sistemas en *cluster*, 13-14

sistemas multiprocesador, 11-13

Arranque, 64, 738-739

bloque de control de, 371

del sistema, 64

dual, sistemas de, 374

partición de, 419

sector de, 419

virus de, 519

ASID. Véase Identificadores del espacio de direcciones**Asignación:**

de almacenamiento dinámico, problema de, 253, 379

de espacio de disco, 378-386

asignación contigua, 378-380

asignación enlazada, 380-382

asignación indexada, 383-384

y prestaciones, 384-386

- de franjas, 316-317
 - de la memoria del *kernel*, 314-317
 - de memoria contigua, 252
 - de recursos (servicios del sistema operativo), 36-37
 - descomposición binaria, 315-316
 - equitativa, 303
 - proporcional, 303
 - Asignación de marcos**, 302-305
 - algoritmo de, 293
 - asignación equitativa, 303
 - asignación proporcional, 303-304
 - global y local, 304-305
 - Asignador de páginas (Linux)**, 689
 - Asignador de potencias de 2**, 315
 - Asignador de recursos, sistema operativo como**, 5
 - Asimétrico, cifrado, 528
 - Asíncrono, dispositivo, 460
 - Asistente personal digital (PDA, personal digital assistants), 10, 27
 - ATA (advanced technology attachment), buses**, 409
 - Ataque**
 - de denegación de servicio, 510
 - de día cero, 542
 - de reproducción, 510
 - por interposición, 510
 - Atlas, sistema operativo**, 771
 - Atomicidad**, 610-612
 - Atributos no residentes**, 743
 - Atributos residentes**, 743
 - Autenticación**:
 - de dos factores, 539
 - en Linux, 707-708
 - en Windows, 742
 - ruptura, 510
 - y cifrado, 530-531
 - Autenticación de usuario**, 535-539
 - biométrica, 539
 - contraseñas, 536-539
 - Autocomprobaciones**, 680
 - Automapa de tabla de páginas**, 726
 - Automática, variable**, 515
 - Automontaje, funcionalidad**, 589
 - Autoridades de certificación**, 533
- B**
- Bandas de disco**, 746
 - Base de datos de marcos de página**, 730
 - Base de datos de ubicación de volumen (NFS V4)**, 600
 - Base del segmento**, 270
 - Base informática de confianza (TCB)**, 548
 - Base, registro**, 244, 245
 - Básico, sistema de archivos**, 370
 - Bayes, teorema de**, 543
 - Belady, anomalía de**, 295
 - Biblioteca C**, 44
 - Bibliotecas compartidas**, 248-249, 282
 - Bibliotecas de hebras**, 117-123
 - acerca de, 117-118
 - hebras de Win32, 118-121
 - hebras Java, 121-123
 - Pthreads, 118
 - Bibliotecas de montaje dinámico (DLL)**, 248-249, 716
 - Bibliotecas del sistema (Linux)**, 676, 677
 - Binario, semáforo**, 179
 - Biométrica**, 539
 - Bit**:
 - de modificación (sucio), 292
 - de modo, 17
 - de referencia, 299
 - válido-inválido, 262
 - Bloque de control de dirección virtual (VACB)**, 735
 - Bloque de mensajes de servidor (SMB)**, 750
 - Bloque(s)**, 41, 253, 341-342
 - de arranque, 64, 371, 419
 - de control de archivo (FCB), 371
 - de control de arranque, 371
 - de control de proceso o bloque de control de tarea (PCB, process control block), 75-76
 - de control de volumen, 371
 - de índice, esquema indexado, 383-384
 - defectuosos, 420-421
 - definición, 703
 - directo, 384
 - dispositivos de, 459-460
 - doblemente indirecto, 384
 - índice a, 343
 - indirecto de un nivel, 384
 - lógicos, 409
 - número relativo de, 343
 - tríplemente indirecto, 384
 - Bloqueante, E/S**, 463-464
 - Bloqueo-clave, esquema de**, 494
 - Bloqueo(s)**, 175, 494
 - compartido, 337
 - de archivos obligatorio, mecanismos de, 337-338
 - de archivos sugerido, mecanismos de, 337-338
 - en la API Java, 337-338
 - exclusivo, 338
 - indefinido (muerte por inanición), 146, 181-182
 - lector-escritor, 185
 - mútex, 179, 219-220
 - obligatorio, 337
 - sugeridos, 337
 - Borrado de archivos**, 335
 - Bosques**, 755
 - Bucle arbitrado, (FC-AL)**, 41
 - Búfer**, 703
 - almacenamiento en, 91-92, 465-467, 664
 - caché de, 389
 - circular, 394
 - de páginas, algoritmo de, 301
 - de traducción directa (TLB), 260, 728
 - definición, 465
 - Bus, 408-409**
 - ATA, 409
 - de expansión, 450
 - definición, 450
 - PCI, 450
 - USB, 409
 - Búsqueda en archivos**, 335
 - Búsqueda, tiempo de (discos)**, 408, 412
 - Buzones de correo**, 752
 - Buzones de correo**, 90

C

- Caballo de Troya**, 513-514
Cabecera de stream, 473
Caché:
 como búfer de memoria, 244
 de búfer, 389
 de búfer unificado, 390, 391
 de páginas, 389, 390, 691
 definición, 467
 en Linux, 690
 en Windows XP, 734-736
 franjas en, 316
 RAM no volátil, 425
 y acceso a archivos remotos:
 política de actualización, 592-593
 ubicación de la caché, 591-592
 y coherencia, 593-594
 y prestaciones, 389
cachefs, sistema de archivos, 592
Cadena de referencia, 293
Cajón de arena (Tripwire, sistema de archivos), 545
Cálculos, migración de, 561-562
Cambio de contexto, 80-81, 475-476
Cambio de fase, discos de, 434
Canales encubiertos, 514
Cancelación asíncrona de hebras, 124
Cancelación de hebras, 124
Cancelación diferida de hebras, 124
CAP de Cambridge, sistema, 499-500
Capacidad cero (de colas), 92
Capacidad de datos, 499
Capacidad ilimitada (de colas), 92
Capacidad limitada (de colas), 92
Capacidad software, 499
Caracteres, dispositivos de (Linux), 703-704
Carga:
 dinámica, 248
 en Linux, 694-695
 tiempo de, 247
 y ejecución de programas, 50
Cargador, 768
Cargador de clases, 60
Carpetas, 38
Casi en línea, almacenamiento, almacenamiento, 435
CAV (constant angular velocity), 410
CD (collision detection, detección de colisiones), 572
Cerrojos. Véase Bloqueos
Certificación, 549
Certificado digital, 533
Ciclos:
 de ráfagas de CPU y de E/S, 138-139
 en CineBlitz, 663
 robo de, 458
Cifrada, contraseña, 537-538
Cifrado, 526-533
 asimétrico, 528-530
 autenticación, 530-531
 de bloqueo, 528
 de flujo, 528
 distribución de claves, 531-533
 simétrico, 527-528
 virus, 520
 Windows XP, 748
CIFS (common Internet file system), 358
CineBlitz, 663-665
Cintas de traza, 164
Cintas magnéticas, 409, 434-435
Circuitos, conmutación de, 571
Circular, búfer, 394
Clases (Java), 503
Clave de tipo, 468
Claves, 494, 497, 526
 distribución de, 531-533
 flujo de, 528
 privada, 529
 pública, 529
CLI (command-line interface), 35
Cliente(s):
 de tiempo real, 663
 definición, 585
 en SSL, 534-535
 interfaz de, 586
 no de tiempo real, 663
 sin disco, 588
Cliente-servidor, modelo, 357-358
C-LOOK, algoritmo de planificación, 416
close(), operación, 336
Cluster, 418, 579, 743
 asimétrico, 13
 reasignación de, 748
 servidor de, 598
 sistemas en, 13-14
 tabla de páginas en, 267
CLV (constant linear velocity), 410
Código:
 absoluto, 246
 de autenticación de mensajes (MAC), 531
 de corrección de errores (ECC), 418, 426
 de tipo adicional, 468
 fuente, virus de, 520
 independiente de la posición (PIC), 696
 intermedio (bytecode), 60
 puro, 263
 reentrante, 263
Coherencia, 594-595
 de caché, 23, 593-594
 semántica de, 359-360
Cola(s), 78-79
 capacidad de, 92
 de bloqueos (*tunrstile*), 194
 de dispositivo, 467
 de ejecución, 160, 684
 de entrada, 245
 de escritura, 703
 de espera, 704
 de lectura, 703
 de mensajes, 773
 de procesos preparados, 78, 79, 250
 de trabajos, 15, 78
 del dispositivo, 78
 ordenada, 703
Colisiones (de nombres de archivo), 378
Colmena del sistema, 738

- COM, modelo**, 753
Compactación, 255, 380
Compatible, dispositivo, 460
Compartición:
 carga, 151, 558
 del procesador, 147
 recursos, 558
 tiempo, 15
 y paginación, 263-264
Compartición de archivos, 355-360
 con múltiples usuarios, 356
 en redes, 356-359
 modelo cliente-servidor, 357-358
 modos de fallo, 358-359
 sistemas de información distribuidos, 358
 semántica de coherencia, 359-360
Compartido, bloqueo (cerrojo), 337
Compartir la carga, 151, 558
Compilación, tiempo de, 246
Compilador C de GNU (gcc), 674
Compilador just-in-time, 62
Compilador, imposición de reglas basadas en, 500-503
Complejidad administrativa, 589
Componentes del sistema Linux, 672, 676-678
Compresión:
 con pérdidas, 654
 en sistemas multimedia, 654-655
 en Windows XP, 748
 sin pérdidas, 654
 tasa de, 654
 unidades de, 748
Comprobación de coherencia, 392
Computadoras de mano, 4
Computadoras de red, 29
Computadoras personales (PC), 3
Comunicación interprocesos (IPC, interprocess communication), 86-92
 en Linux, 672, 704-705
 en los sistemas cliente-servidor, 96-103
 invocación de métodos remotos, 102-103
 llamadas a procedimientos remotos, 99-102
 sockets, 97-99
 Mach, ejemplo, 93-95
 memoria compartida en POSIX, ejemplo, 92-93
 sistemas de memoria compartida, 87-89
 sistemas de paso de mensajes, 89-92
 Windows XP, ejemplo, 95-96
Comunicaciones:
 directas, 89
 en sistemas operativos distribuidos, 559
 indirectas, 90
 interproceso. Véase Comunicación interprocesos
 llamadas al sistema, 48-49
 no fiables, 625-626
 programas del sistema, 50
 servicios del sistema operativo, 36
Concentrador de terminales, 475
Condición de carrera, 173
Condiciones de error, 280
Conectado a la red, almacenamiento, 411-412
Conectado al host, almacenamiento, 410
Conexión en cascada, 450
Confidencialidad, ruptura de la, 510
Confinamiento, problema del, 492
Conformación, 752
Conjunto:
 de buzones de correo, 95
 de distribución en bandas, 746-747
 de entrada, 193
 de trabajo, 307, 310
 de volúmenes, 745
 dúplex, 748
 espejo, 747
Conjuntos compartidos:
 de hebras, 125-126
 de páginas libres, 290
Conmutación:
 de circuitos, 571
 de dominio, 486
 de mensajes, 571
 de paquetes, 571
Contador de programa, 19, 74
Contador, semáforo, 179
Contexto (del proceso), 80
Contexto de seguridad (Windows XP), 549-550
Contienda, 572
Contraseñas, 535-539
 cifradas, 537-538
 de un solo uso, 538-539
 emparejadas, 538
 vulnerabilidades de, 536-537
Control de acceso basado en roles (RBAC), 495
Control de concurrencia, 612-616
 algoritmos de, 201
 con marcas temporales, 615-616
 con protocolos de bloqueo, 613-615
Control de flujo, 473
Control de procesos, llamadas al sistema, 42-46
Control, registro de, 452
Controlador de dispositivo, 10, 370, 471, 768
Controlador de puerto, 734
Controladora(s), 409, 450-451
 de acceso directo a memoria, 457
 de disco, 409
 de dispositivos, 6, 471. Véase también Sistemas de E/S
 definición, 450
 hardware de interrupciones, 454
 host, 409
Controladores de filtrado, 734
Convenio uniforme de nombrado (UNC), 752
Convoy, efecto, 143
Cooperativa, planificación, 139
Coordinación distribuida:
 algoritmos de elección, 623-625
 atomicidad, 610-612
 control de concurrencia, 612-616
 exclusión mutua, 607-610
 interbloqueos, 616-623
 detección, 618-623
 prevención/evasión, 617-618
 ordenación de sucesos, 605-607
 procedimientos de acuerdo, 625-626

- Coordinador de la transacción**, 610
Copia de seguridad, 392
 completa, 393
 incremental, 393
Copia durante la escritura, técnica de, 289–290
Cortafuegos, 29, 546–547
 cadenas de, 707
 de llamadas al sistema, 547
 gestión de, 707
 personal, 547
CPU (central processing unit), 4, 243–245
CPU y E/S, ciclo de ráfaga de, 138–139
CPU, ráfaga de, 138
Crackers, 510
Creación:
 de archivos, 335
 de procesos, 81–85
Criptografía, 525–535
 cifrado, 526–533
 implementación, 533–534
 SSL, ejemplo, 534–535
CSC (client-side caching, caché del lado del cliente), 754
C-SCAN, algoritmo de planificación, 416
CSMA (carrier sense with multiple access), 572
CTSS, sistema operativo, 773–774
Cuaderno de un solo uso, 539
Cualificador adicional del código de tipo, 468
Cuanto de tiempo, 146, 719
Cúmulos de memoria, 74, 762
Cuota equitativa (Solaris), 157
Cuota proporcional, planificación con, 644–645
Cuotas, 158
- D**
- d (desplazamiento de página)**, 255
Datagramas, 571
Datos:
 archivos de, 334
 de flujo continuo, 652
 distribución en bandas de, 425
 específicos de una hebra, 126–127
 multimedia, 27
 recuperación de, 392–393
DCOM, 753
Dedicado, dispositivo, 460
Defectuosos, bloques, 420–421
Degradación suave, 12
Delegación (NFS V4), 596
Demonio, proceso, 487
Denegación de servicio, ataque de, 510
Densidad de almacenamiento, 446
Depuradores, 43
Derechos:
 auxiliares (*Hydra*), 498
 de grupo (Linux), 708
 de propietario (Linux), 708
 de usuario (Linux), 708
 del resto del mundo (Linux), 708
DES (data-encryption standard), 528
Desafío (contraseñas), 538
Desbordamiento de búfer, ataque, 515–518
Desbordamiento de pila, ataque, 515–518
- Descarga progresiva**, 652
Descomposición binaria, sistema de, 315–316
Descriptor(es), 722, 725
 de archivo, 373
 de direcciones virtuales (VAD), 731
 de instancia, 758–759
 de seguridad (Windows XP), 549
Deslizamiento de sectores, 421
Despachador, 140
Desplazamiento de página (d), 255
Detección:
 basada en firma, 542
 de anomalías, 542
 de colisiones (CD), 572
 de errores, 36
 de interbloqueos completamente distribuido, algoritmo, 621–623
 de interbloqueos, coordinador de, 619
 de intrusiones, 541–545
 de portadora con acceso múltiple (CSMA), 572
 tiempo de, 380
 y eliminación de huérfanos, 595
Detenerse, 244
DFS (distributed file system, sistema de archivos distribuido), 357, 585–604
 acceso remoto a archivos, 590–595
 coherencia, 593–594
 esquema básico de caché, 590–591
 mecanismo de caché y servicio remoto, 594–595
 política de actualización de la caché, 592–593
 ubicación de la caché, 591–592
AFS, ejemplo de, 598–602
 espacio de nombres compartido, 599–600
 implementación, 601–602
 operaciones con archivos, 600–601
 definición, 585
 nombrado, 586–588
 replicación de archivos, 597
 servicios con y sin memoria del estado, 594–597
 sin memoria, 359
 Windows XP, 754
Día cero, ataques de, 542
Diagrama de colas, 78
Diario, 700
Diario de cambios (Windows XP), 749
Difusión, 580, 661
Dinámica, carga, 248
Dinámica, protección, 486
Dirección(es):
 de bucle, 98
 de control de acceso al medio (MAC), 580
 definición, 454
 física, 247
 Internet, 568
 lineal, 272
 lógica, 247
 virtual, 247
Direccionamiento real, modo de, 637
Directo, bloque, 384
Directorios:
 actual, 349
 de archivos de usuario (UFD), 347

- de dispositivo, 345. *Véase también* Directories de páginas, 727
 maestro de archivos (MFD), 347
 objeto (Windows XP), 724
- Directories**, 345-354
 con estructura de árbol, 348-350
 de dos niveles, 347-348
 de un único nivel, 346-347
 en forma de grafo general, 353-354
 en un grafo acíclico, 350-352
 estructura de almacenamiento, 345
 implementación de, 377-378
 recuperación de, 392-393
- Disciplina de línea**, 703
- Disco(s)**, 407-409. *Véase también* Almacenamiento masivo, estructura de
 algoritmos de planificación de disco, 412-417
 C-SCAN, 416
 FCFS, 413
 LOOK, 416
 SCAN, 415
 selección, 416-417
 SSTF, 413-415
 asignación de espacio de, 378-386
 asignación contigua, 378-380
 asignación enlazada, 380-382
 asignación indexada, 383-384
 y prestaciones, 384-386
 bloque de arranque, 419
 bloques defectuosos, 420-421
 brazo del, 408
 conectado a la red, 411-412
 conectado al host, 410
 controladora de, 409
 de arranque, 64, 419
 de cambio de fase, 434
 de estado sólido, 22
 de lectura-escritura, 434
 de sólo lectura, 434
 del sistema, 419
 disquete, 408-409
 eficiencia y prestaciones, 388-392
 electrónico, 9
 estructura, 409
 extraibles, 433-434
 formateo a bajo nivel, 409
 formateo, 418-419
 gestión del espacio libre, 386-388
 magnético, 8
 magneto-ópticos, 434
 redes de área de almacenamiento, 412
 sin formato, 302
 uso eficiente de, 388
 WORM, 434
- Diseño de sistemas operativos:**
 Linux, 675-678
 mecanismos y políticas, 51
 objetivos, 50-51
 sistemas operativos distribuidos, 577-579
 Windows XP, 714-717
- Dispersión**, 267, 282
- Dispositivos**
 de acceso aleatorio, 460, 770
 de acceso secuencial, 770
 de almacenamiento terciario, 21
 de sólo escritura, 460
 del sistema, 738
- Disquetes**, 408-409
- Distribución en bandas de nivel bit**, 425
- Distribución en bandas de nivel de bloque**, 425
- Distribución fuera de banda**, 532
- Distribuciones Linux**, 672, 674-675
- DLL**. *Véase* Biblioteca de montaje dinámico
- DLM (distributed lock manager)**, 14
- DMA (direct-memory-access) controladora**, 457
- DMA**. *Véase* Acceso directo a memoria
- DMZ (demilitarized zone)**, 546
- DNS (domain-name system)**, 358, 568
- Doble búfer**, 466, 664
- Doble caché**, 390
- Doblemente indirecto, bloque**, 384
- Dominio de aplicación**, 62
- Dominio de protección**, 486
- Dominio público**, 675
- Dominio, conmutación**, 486
- Dominios**, 358, 754-756
 de seguridad, 546
- DOS (denial-of-service, denegación de servicio), ataques de**, 510
- Dos niveles, directorios en**, 347-348
- DPC (deferred procedure calls)**, 720
- DRAM (dynamic random-access memory)**, 7-8
- Duplicación en espejo**, 424
- Duplicación inteligente en espejo**, 755
- DVMA (direct virtual memory access)**, 458
- E**
- E/S (entrada/salida)**, 3, 9-11
 canal de, 475, 476
 directa, 461
 gestor de, 733-736
 interbloqueo de, 321-323
 mapeada en memoria, 313
 programada, 314
 puertos de, 314
 ráfaga de, 138
 rápida, mecanismo de, 735
 sin formato, 461
 solapada, 769-771
- E/S, sistema(s) de**, 449-480
 hardware, 450-458
 acceso directo a memoria, 457-458
 interrupciones, 453-456
 sondeo, 452-453
 interfaz de las aplicaciones, 458-464
 dispositivos de bloques y de caracteres, 461-462
 dispositivos de red, 462
 E/S bloqueante y no bloqueante, 463-464
 relojes y temporizadores, 462-463
kernel, 464-470
 almacenamiento en búfer, 465-467
 almacenamiento en caché, 467

- E/S, sistema(s) de, (Cont.)**
- estructuras de datos del, 469–470
 - gestión de colas y reserva de dispositivos, 467–468
 - planificación de E/S, 464–465
 - protección, 468–469
 - tratamiento de errores, 468
 - y subsistemas de E/S, 470
- Linux**, 702–704
- dispositivos de bloque, 703
 - dispositivos de caracteres, 703–704
- rendimiento**, 475–478
- STREAMS**, mecanismos de, 473–475
- transformación de solicitudes del *kernel* en operaciones hardware, 471–473
- ECC (error-correcting code, código de corrección de errores)**, 418, 426
- EDF (earliest-deadline-first)**, planificación, 644, 658–659
- Efectivo**, ancho de banda, 438
- Eficiencia**, 3, 388–389
- EIDE (enhanced integrated drive electronics) buses**, 408
- Ejecución**,
- de instrucciones, ciclo de, 243–245
 - de programas (servicios del sistema operativo), 36
 - de programas de usuario, 694
 - tiempo de, 247
- Ejecutables, archivos**, 74, 334
- Elección**, 572
- Elección, algoritmos de**, 623–625
- Empaquetar**, 341
- En ejecución, estado**, 75
- En ejecución, estado de hebra (Windows XP)**, 719
- En espera, estado**, 75
- En línea, compactación de espacio**, 380
- Encaminamiento:**
- de bloques cifrados, 528
 - de interrupciones, 454
 - dinámico, 569–570
 - en redes parcialmente conectadas, 566
 - fijo, 569–570
 - tabla de, 569
 - virtual, 569–570
 - y comunicación de redes, 569–570
- Encapsulación (Java)**, 505
- Enlaces:**
- de comunicaciones, 89
 - definición, 351
 - duros o no simbólicos, 352
 - resolución, 351
 - simbólico, 724
- Enmascarable, interrupción**, 454
- Entornos informáticos**, 28–31
- cliente-servidor, 29–30
 - sistema basado en la Web, 30–31
 - sistema entre iguales, 30
 - tradicional, 28–29
- Entrada de datos, registro de**, 452
- Entrada de directorio (objeto)**, 376, 696
- Entrada de directorio de páginas (PDE)**, 727
- Entrada de pila**, 516–517
- Envejecimiento**, 146, 580
- EPROM (erasable programmable read only memory)**, 64
- Equilibrado de carga**, 152–153
- Equitativa, asignación**, 303
- Error(es):**
- blandos, 418
 - duros, 421
 - tratamiento de, 468
- Escalabilidad**, 578
- Escalar privilegios**, 25
- Escaneo de puertos**, 524
- Escape (sistemas operativos)**, 460
- Escritores**, 183–185
- Escritorio**, 38
- Escritura**
- anticipada, 391
 - asíncrona, 391
 - de archivos, 335
 - diferida, caché de, 592
 - diferida, política de, 592
 - síncrona, 391
- Espacio de direcciones:**
- lógico y físico, 247–248
 - virtuales, 280, 692–693
- Espacio de disco**
- asignación contigua, 378–380
 - asignación enlazada, 380–382
 - asignación indexada, 383–384
 - sin formato, 345
- Espacio de intercambio**, 286
- Espacio de nombres compartido**, 599–600
- Espacio de nombres local**, 598
- Espacio de sesión**, 726
- Especificación de interfaz de dispositivo de red (NDIS)**, 750
- Espejo de disco**, 747
- Espera activa**, 180, 453
- Espera circular, condición de (interbloqueos)**, 225–226
- Espera condicional, estructura**, 191
- Espera, estado de hebra (Windows XP)**, 719
- Espera-muere, esquema**, 617
- Esqueleto**, 102
- Esquema combinado, bloque de índice**, 384
- Esquema de redundancia P + Q**, 428
- Esquema de temporización**, 576, 625–626
- Estable, almacenamiento**, 198, 432–442
- Estaciones de trabajo**, 4
- Estado**
- de la arquitectura, 153
 - del proceso, 74–75
 - información de, 359
 - no señalizado, 195
 - registro de, 452
 - señalizado, 195
 - suspendido, 759
- Estándar avanzado de cifrado (AES)**, 528
- Estática, protección**, 486
- Estricto, sistema de tiempo real**, 634, 658
- Estructura**
- de árbol, directorios con, 348–350
 - en anillo, 609
 - en niveles (estructura del sistema operativo), 53–55
 - simple del sistema operativo, 53
- Etiqueta**, 494
- Evaluación analítica**, 162
- Evaluación de riesgos**, 540–541

- Evaluación de vulnerabilidades**, 540-541
Excepción, 16
Excepciones (con interrupciones), 455
Exclusión mutua, 220, 607-610
 - condición de (interbloqueos), 224
 - enfoque centralizado, 607
 - enfoque completamente distribuido, 608-609
 - técnicas basadas en paso de testigo, 609-610**Exclusivo, bloqueo (cerrojo)**, 337
exec(), llamada al sistema, 123
Expansión, bus de, 450
Exportación, lista de, 397
Extensión (espacio contiguo), 380
Extensiones, 743
Extensiones del kernel, 57
Extremo de controlador (STREAM), 473
- F**
- Facilidad de trazado dinámico de Solaris 10**, 47
Facilidad de uso, 4, 714
Fallos:
 - detección de, 575-576
 - durante la escritura de bloques, 432-433
 - gestión de (protocolo 2PC), 611-612
 - recuperación de, 577
 - tiempo medio entre, 424**Falsos negativos**, 543
Falsos positivos, 543
Fase de conflicto (de la latencia de despacho), 640
FAT (file-allocation table), 382
FC (fiber channel) buses, 409
FC (fiber channel), 411
FC-AL (arbitrated loop), 411
FCB (file-control block), 371
FCFS, algoritmo de planificación, 142-143, 413
Fiabilidad:
 - de sistemas operativos distribuidos, 558-559
 - de Windows XP, 715
 - en sistemas multimedia, 656
 - paquetes, 571**Fibras**, 760
fid (NFS V4), 599
File descriptor (descriptor de archivo), 373
File handle (descriptor de archivo), 373
File System Hierarchy Standard, documento, 674
FileLock (Java), 337
FireWire, 410
Firma digital, 531
Firmware, 6, 64
Física, dirección, 247
Físico, espacio de direcciones, 247-248
Fluctuación, 656
Flujo
 - a la carta, 653
 - de caracteres, dispositivo de, 459-460, 703-704
 - de clave, 528
 - en tiempo real, 652, 661-663
 - en vivo (*live streaming*), 653**fork para memoria virtual**, 290
fork(), llamadas al sistema, 123
fork() y exec(), modelo de procesos (Linux), 681-683
Formateo, 418-419
de discos a bajo nivel, 409, 418-419
físico, 418
lógico, 418
Fragmentación, 254-255
 - externa, 254-255, 379
 - interna, 255, 342**Fragmentos de paquetes**, 707
Franjas, 345
 - asignación de, 316-317, 690**Frecuencia de fallos de página (PFF)**, 309
FTP (file transfer protocol), 560-561
ftp, 357
Fuente, archivo, 334
Fuera de línea, compactación de espacio, 380
Funciones seguras respecto a las señales, 110
- G**
- Gantt, diagrama de**, 142
GB (gigabyte), 5
gcc (compilador C de GNU), 674
GDT (global descriptor table), 272
Generación del sistema (SYSGEN, system generation), 63-64
Generales bizantinos, problema de los, 625
Gestión de almacenamiento, 20-24
 - en caché, 22-23
 - jerárquico (HSM), 437-438
 - masivo, 21
 - sistemas de E/S, 23-24**Gestión de colas**, 467-468
Gestión de la caché, 22
Gestión de memoria, 19-20
 - en Linux, 688-696
 - ejecución y carga de programas de usuario, 694-696
 - memoria física, 688-691
 - memoria virtual, 691-694
 - en Windows XP, 761-763
 - almacenamiento local en una hebra, 762
 - cúmulos de memoria, 762
 - mapeo de archivos de memoria, 761
 - memoria virtual, 761**Gestión de procesos**, 19
 - en Linux, 681-684
 - fork() y exec(), modelo de procesos, 681-682
 - procesos y hebras, 683-684
 - Gestión de red en sistemas multimedia, 660-663
 - Gestión de volúmenes (Windows XP), 745-748
 - Gestión del ciclo de vida de la información (ILM), 438
 - Gestión del espacio libre en discos, 386-388
 - agrupamiento, 387
 - lista enlazada, 387
 - recuento, 388
 - vector de bits, 386-387
 - Gestión del sistema de archivos, 20-21
 - Gestor**
 - de bloqueos distribuido (DLM, distributed lock manager), 14
 - de memoria virtual (VM), 726-731
 - de procesos (Windows XP), 731-732
 - de recurso, 657
 - de solicitudes, 703
 - plug-and-play (PnP), 737-738

- Gigabyte (GB),** 5
Global, sustitución, 304
GNU Portable Threads, 116
Grado de multiprogramación, 80
Grafo acíclico, 351
 - directorios en un, 350-352**Grafo de asignación de recursos del sistema,** 220-223
Grafo de asignación de recursos, algoritmo, 228-229
Grafo general, directorio en forma de, 353-354
Green, hebras, 116
Grupos de bloques, 698
Grupos de cilindros, 698
Gusanos, 521-524
- H**
- Ha sucedido antes, relación,** 605-606
HAL (hardware-abstraction layer), 717
Hardware, 4
 - de sistemas de E/S, 450-458
 - acceso directo a memoria, 457-458
 - interrupciones, 453-456
 - sondeo, 452-453
 - objetos, 485
 - para almacenar tablas de páginas, 259-261
 - sincronización, 175-178
- Hash:**
 - funciones, 530
 - tablas, 377-378
 - tabla de páginas, 266
 - valor (resumen de mensaje), 530
- Hebras.** Véase también Multihebra
 - cancelación, 124
 - componentes de, 113
 - conjuntos compartidos, 125-126
 - de usuario, 115
 - en Linux, 128-130, 682-683
 - en Windows XP, 128, 129, 718-719, 757, 759-760
 - funciones, 113-115
 - inactiva, 158
 - Java, 121-123
 - kernel, 115
 - objetivo, 124
 - planificación de, 154-155
 - principal, 757
 - y el modelo de proceso, 76-77
- Herido-expresa, esquema,** 617
Hijos, 81
Hiperespacio, 726
Holográfico, almacenamiento, 435
Homogeneidad, 151
Host, adaptadora, 450
Host, controladora, 409
HSM (hierarchical storage management), 437
Huella, 635
Husmear, 536
Hydra, 498-499
- I**
- IBM QS/360,** 774-776
Identidad de cliente suplantada, 357
Identidad del proceso (Linux), 681-682
- Identificadores:**
 - de archivo, 334
 - de archivos independientes de la ubicación, 589
 - de grupo, 25
 - de usuario (ID), 24
 - efectivo, 25
 - para archivos, 335
 - del espacio de direcciones (ASID), 260-261
 - de proceso (pid), 81
- IDP (intrusion-prevention system,sistema de prevención de intrusiones),** 542
IDS (intrusion-detection system,sistema de detección de intrusiones), 542
IKE, protocolo, 533
ILM (information life-cycle management), 438
Implementación:
 - de algoritmos de planificación de la CPU, 165
 - de máquinas virtuales, 59
 - de sistema operativos de tiempo real, 637-641
 - kernels* apropiativos, 638-639
 - minimización de la latencia, 639-641
 - planificación basada en prioridades, 638
 - de sistemas operativos, 52
 - de técnicas de nombrado transparente, 589-590
- Independencia de ubicación,** 587
Independientes, discos, 425
Índice, 343
 - bloque de, 383
 - multinivel, 384
- Indirecto, bloque,** 384
Información de estado, 49
Informática segura, 545
Ingeniería social, 512
Inicio de sesión
 - de red, 358
 - remoto, 559
 - unificado seguro, 358
- Inodo, objeto,** 376, 696
InServ, matriz de almacenamiento, 431
Inspección de pila, 504
Instrucciones privilegiadas, 17
Integridad, ruptura de, 510
Intel Pentium, procesador, 271-275
Intellimirror, 755
Interbloqueos, 181-182, 616-622
 - con cerrojos mútex, 219-220
 - condiciones necesarias para, 219-220
 - de E/S, 321-323
 - definición, 217
 - detección de, 232-235, 618-622
 - una sola instancia de cada tipo de recurso, 232-233
 - utilización del algoritmo de, 234
 - varias instancias de cada tipo de recurso, 233-234
 - evasión de, 223, 226-232
 - con el algoritmo de estado seguro, 227-228
 - con el algoritmo del banquero, 229-232
 - con el grafo de asignación de recursos, 228-229
 - grafos de asignación de recursos del sistema, 220-223
 - métodos para tratar los, 223-224
 - modelo de sistema, 217-218
 - prevención de, 223-226
 - condición de espera circular, 225-226

- condición de exclusión mutua, 224
 condición de retención y espera, 224–225
 no apropiativo (sin desalojo), 225
 prevención y evasión de, 617–618
 recuperación de, 235–236
 mediante apropiación de recursos, 235
 mediante terminación de procesos, 235
- Intercambiador (jukebox) robotizado**, 437
- Intercambiador perezoso**, 283
- Intercambio**, 16, 80, 249–252, 283
 en Linux, 693
 espacio de, 286
 gestión del espacio de, 421–423
 paginación e, 421
- Interfaz:**
 cliente, 585
 de controlador de transporte (TDI), 750
 de línea de comandos (CLI), 35
 de llamadas al sistema, 40
 de proceso por lotes, 35–36
 de programación de aplicaciones (API, application program interface), 39–40
 de red de Windows XP, 750
 de usuario (UI), 36–37
 definición, 459
 gráfica de usuario (GUI), 37–39
 intermáquinas, 586
- Interfaz de las aplicaciones (sistemas de E/S)**, 458–464
 dispositivos de bloques y de caracteres, 461–462
 dispositivos de red, 462
 E/S bloqueante y no bloqueante, 463–464
 relojes y temporizadores, 462–463
- Intermáquinas, interfaz**, 586
- Interna, fragmentación**, 255, 342
- Internet, dirección**, 568
- Internet, protocolo (IP)**, 533–534
- Interposición, ataque de**, 510
- Intérprete de comandos**, 37–38
- Intérprete de tarjetas de control**, 768
- Interrupción(es)**, 6, 453–456
 de fallo de página, 284
 definición, 453
 en Linux, 687–688
 encadenamiento de, 454
 niveles de prioridad de, 455
 vector de, 454
- Intrusos**, 510
- Inversión de prioridad**, 195, 641
- Invertida, tabla de páginas**, 267–268, 320
- Invocación de métodos remotos (RMI, remote method invocation)**, 102–103
- IP (Internet Protocol)**, 533–534
- IPC. Véase Comunicación interprocesos**
- IPSec**, 533
- IRP (paquete de solicitud de E/S)**, 734
- ISCSI**, 411
- ISO, modelo de referencia**, 533
- ISO, pila del protocolo**, 574
- J**
- Java:**
 bloqueo de archivos en, 337–338
- monitores en, 193
 protección basada en el lenguaje, 503–505
- JVM (Java Virtual Machine)**, 60–63
- K**
- KB (kilobyte)**, 5
- Kerberos**, 742
- Kernel**, 5, 464–470
 almacenamiento en búfer, 465–467
 almacenamiento en caché, 467
 apropiativo, 174, 638–639
 de desarrollo (Linux), 672
 de producción (Linux), 672
 de tiempo real, 636–637
 estructuras de datos, 469–470
 gestión de colas y reserva de dispositivos, 467–468
 Linux, 676, 678
 no apropiativos, 174
 planificación de E/S, 464–465
 protección, 468–469
 sincronización de tareas (en Linux), 686–688
 sistemas multimedia, 655–658
 tratamiento de errores, 468
 Windows XP, 717–722
 y subsistemas de E/S, 470
- Kernel, módulos (Linux)**, 678–681
 gestión de módulos, 679
 registro de controladores, 679–680
 resolución de conflictos, 680–681
- Kerr, efecto**, 434
- Kilobyte (KB)**, 5
- L**
- LAN. Véase Red de área local**
- Latencia**
 de despacho, 140, 639–641
 de interrupción, 639–640
 del suceso, 639
 en sistemas de tiempo real, 639–641
 rotacional (discos), 408, 412
- LCN (logical cluster numbers)**, 743
- LDAP (lightweight directory-access protocol)**, 358, 755
- LDT (local descriptor table)**, 272
- Lectores**, 183–185
- Lectura de archivos**, 335
- Lectura-escritura, discos de**, 434
- Lectura-modificación-escritura, ciclo de**, 428
- Lenguaje de definición de interfaces de Microsoft**, 753
- LFU (least-frequently used), algoritmo de sustitución de páginas**, 300
- Liberación retardada, técnica**, 391
- Libres, objetos**, 316, 690
- Libros de código**, 539
- Licencias software**, 208
- Límite del segmento**, 270
- Límite, registro**, 244, 245
- Línea de solicitud de interrupción**, 453
- Lineal, dirección**, 272
- Linux**, 671–712
 adicción de una llamada al sistema al *kernel* de Linux (proyecto), 67–70
 comunicación interprocesos, 704–705

Linux (Cont.)

- ejemplo de planificación, 160-161
- ejemplos de hebras, 129-130
- en sistemas Pentium, 273-275
- en tiempo real, 648
- estructura de red, 705-707
- gestión de memoria, 688-696
 - ejecución y carga de programas de usuario, 694-696
 - memoria física, 688-691
 - memoria virtual, 691-694
- gestión de procesos, 681-684
 - modelo de procesos fork() y exec(), 681-683
 - procesos y hebras, 683-684
- gestión del espacio de intercambio, 424
- historia, 671-675
 - descripción del sistema, 674
 - distribuciones, 674-675
 - licencias, 675
 - primer kernel, 672-674
- modelo de seguridad, 707-709
 - autenticación, 707-708
 - control de acceso, 708-709
- módulos del kernel, 678-681
- planificación, 684-688
 - de procesos, 684-686
 - multiprocesamiento simétrico, 688
 - sincronización del kernel, 686-688
- principios de diseño, 675-678
- representación de un proceso en, 77
- sincronización en, 196
- sistema de E/S, 702-704
 - dispositivos de bloque, 703
 - dispositivos de caracteres, 703-704
- sistemas de archivos, 696-702
 - diario, 700
 - ext2fs, 698-700
 - proc, 700-702
 - virtual, 696-698

Lista

- de acceso (NFS V4), 599
- de activos, 624
- de capacidades, 493-494
- de control de acceso (ACL), 362
- de espacio libre, 386
- enlazada, 387
 - lineal (archivos), 377
- Little, fórmula de**, 163
- Llamadas a procedimiento diferidas (DPC)**, 720
- Llamadas a procedimiento remoto (RPC)**, 752
- Llamadas a procedimientos locales (LPC)**, 716, 732-733
- Llamadas al sistema (llamada del monitor)**, 6, 39-49
 - administración de archivos, 46-48
 - administración de dispositivos, 48
 - comunicaciones, 48-49
 - control de procesos, 42-46
 - funcionamiento, 39-40
 - mantenimiento de información, 48
 - y API, 39-40
- Llamadas asíncronas a procedimientos (APC)**, 125, 720
- Local, sustitución**, 304
- Localidad de referencia**, 285
- lock()**, operación, 337

Lógica, dirección, 247

- Lógico, bloque**, 409
- Lógico, espacio de direcciones**, 247-248
- Lógico, registro**, 342
- Lógico, reloj**, 606
- Lógico, sistema de archivos**, 370
- LOOK, algoritmo de planificación**, 416
- LPC (local procedure call)**, 716, 732-733
- LRU (least-recently-used), algoritmo de sustitución de páginas**, 296-298
- LRU, algoritmo de sustitución de páginas mediante aproximación**, 298-300

M

- MAC (medium access control), dirección**, 580
- MAC (message-authentication code)**, 531
- Mach, sistema operativo**, 55, 93-95, 776-777
- Macintosh, sistema operativo**, 341
- Macro, virus de**, 519
- Maestra, clave**, 497
- Magnética, cinta**, 409, 434-435
- Magnético, disco**, 8, 407-409. *Véase también Discos*
- Magneto-óptico, disco**, 434
- Mainframe**, 4
- MAN (metropolitan-area network)**, 25
- Manipulación del sistema de archivos (servicios del sistema operativo)**, 36
- Mantenimiento de información, llamadas al sistema**, 48
- Mapa de intercambio**, 423
- Mapeo de archivos de memoria**, 761
- Mapeo de memoria**, 252, 309-314
 - de archivos, 311
 - definición, 309
 - E/S, mapeada en memoria, 313
 - en la API Win32, 312-313
 - en Linux, 694-695
 - mecanismo básico, 309-311
- Máquina DOS virtual (VDM)**, 739-740
- Máquinas virtuales**, 58-63
 - beneficios, 59-60
 - idea fundamental, 58
 - implementación, 59
 - máquina virtual Java, ejemplo, 60-63
 - VMware, ejemplo, 60
- Marca temporal**, 606, 615-616
- Marcas temporales, esquema conservador de ordenación de**, 616
- Marco de páginas**, 727
- Marcos**, 255
 - victima, 292
- Máscara de protección (Linux)**, 708
- Mascarada**, 510
- Matchmaker**, 101
- Matrices**, 280
- Matriz de acceso**, 489-493
 - definición, 489
 - implementación de, 493-495
 - revocación de derechos de acceso, 496-497
 - y control de acceso, 495-496
- Matriz de almacenamiento**, 424
- Matriz de tareas activas (Linux)**, 684
- Matriz de tareas caducadas (Linux)**, 684

- Máximo del conjunto de trabajo (Windows XP),** 323
MB (megabyte), 5
MBR (master book record), 419
MCP, sistema operativo, 777
Mecanismos, 51
 - de equilibrado de carga, 31
 - de paginación (Linux), 693
 - de protección retroimplementados, 364**Medios de almacenamiento extraibles,** 435–438
 - cintas magnéticas, 409, 434–435
 - denominación de archivos, 437
 - discos magnéticos, 407–409
 - discos, 433–434
 - gestión del almacenamiento jerárquico, 437–438
 - interfaz de aplicación, 436–437**Megabyte (MB),** 5
Mejor ajuste, estrategia de, 254
Memoria:
 - acceso directo a memoria virtual, 458
 - acceso directo a memoria, 10
 - anónima, 423
 - asignación de, 253–254
 - compartida, 86, 282
 - de acceso aleatorio (RAM, random-access memory), 7
 - de núcleo, 771
 - de sólo lectura (ROM, read-only memory), 64, 419
 - demandada con ceros, 692
 - dinámica de acceso aleatorio (DRAM), 7–8
 - física, 16, 279–280, 688–691
 - lógica, 16, 280. Véase también Memoria virtual
 - principal. Véase Memoria principal
 - secundaria, 286
 - semiconductor, 8
 - sobreasignación, 290
 - virtual. Véase Memoria virtual
 - virtual unificada, 390**Memoria principal,**
 - a través de la red, 591
 - asignación contigua de, 252–255
 - mapeo, 252
 - métodos, 253–254
 - protección, 252
 - y fragmentación, 254–255
 - e intercambio, 249–252
 - espacio de direcciones lógico y físico, 247–248
 - Intel Pentium, ejemplo:
 - con Linux, 273–275
 - paginación, 273
 - segmentación, 272
 - paginación para gestión de, 255–259
 - hardware, 259–261
 - Intel Pentium, ejemplo, 273
 - método básico, 255–259
 - paginación jerárquica, 264–266
 - protección, 261–263
 - tablas de páginas *hash*, 266
 - tablas de páginas invertidas, 267–268
 - y páginas compartidas, 263–264
 - segmentación para gestión de, 269–271
 - hardware, 270–271
 - Intel Pentium, ejemplo, 271–275
 - método básico, 269–270

y carga dinámica, 248
 y hardware, 244–245
 y montaje dinámico, 248–249
 y reasignación de direcciones, 245–247

Memoria virtual, 16, 279–282

 - acceso directo a memoria virtual (DVMA), 458
 - archivos mapeados en memoria, 309–314
 - E/S mapeada en memoria, 313
 - en la API Win32, 312–313
 - mecanismo básico, 309–311
 - asignación de la memoria del *kernel*, 314–317
 - del *kernel*, 693
 - en Linux, 691–694
 - en Solaris, 324–325
 - en Windows XP, 323–324
 - paginación bajo demanda, 282–289
 - alcance del TLB, 319–320
 - con tablas de páginas invertidas, 320
 - e instrucciones de reinicio, 286–287
 - estructura de los programas, 320–321
 - interbloqueo de E/S, 321–323
 - mecanismo básico, 283–287
 - prepaginación, 317–318
 - pura, 285
 - tamaño de página, 318–319
 - y rendimiento, 287–289
 - separación de la memoria lógica de la memoria física, 282
 - sustitución de páginas para conservar, 290–302
 - algoritmos de búfer de páginas, 301
 - mecanismo básico, 291–294
 - sustitución de páginas basada en contador, 300
 - sustitución de páginas FIFO, 294–295
 - sustitución de páginas LRU, 296–298
 - sustitución de páginas mediante aproximación LRU, 298–300
 - sustitución óptima de páginas, 296
 - y aplicaciones, 301
 - tamaño, 280
 - unificada, 390
 - y asignación de marcos, 302–305
 - asignación equitativa, 303
 - asignación global y local, 304–305
 - asignación proporcional, 303–304
 - y sobrepaginación, 305–309
 - causa de, 305–307
 - frecuencia de fallos de página, 309
 - modelo del conjunto de trabajo, 307–308
 - y técnica de copia durante la escritura, 289–290

MEMS (micro-electronic mechanical systems), 435

Mensajes:
 - cola de, 773
 - conmutación de, 571
 - en sistemas operativos distribuidos, 559
 - sin conexión, 571

Metaarchivo, 661

Metadatos, 359, 744

Método de particiones múltiples, 253

Métodos (Java), 503

Mezcla de procesos, 80

MFD (master file directory), 347

MFU, algoritmo de sustitución de páginas, 301

Microcomputadora, 4

- Microkernels**, 55-57
- Microsoft Windows**. Véase Windows
- Migración:**
- comandada, 152
 - de archivos, 587
 - de cálculos, 561-562
 - de datos, 561
 - de procesos, 562-563
 - solicitada, 152
- Minidiscos**, 345
- Mínimo del conjunto de trabajo (Windows XP)**, 323
- Mínimo privilegio, principio del**, 484-485
- Minipuerto, controlador de**, 734
- MMU**. Véase Unidad de gestión de memoria
- Modelo:**
- de localidad, 306
 - de memoria compartida, 49, 87-89
 - de paso de mensajes, 48, 89-92
 - de referencia ISO, 533
 - del conjunto de trabajo, 307-308
 - determinista, 162-163
 - multihébra muchos-a-muchos, 116-117
 - multihébra muchos-a-uno, 115-116
 - multihébra uno-a-uno, 116
- Modificación, bits de**, 292
- Modificación de archivos**, 50
- Modificación de mensaje**, 510
- Modo**
- bit de, 17
 - de bloqueo compartido, 613
 - de bloqueo, exclusivo, 613
 - de espera en caliente, 13
 - kernel*, 17, 677
 - simétrico, 13
 - usuario, 17
- Modos de fallo (directorios)**, 358-359
- Módulo de organización de archivos**, 370
- Módulos**, 56-57, 473
- Módulos de autenticación conectables (PAM)**, 708
- Monitor de referencia de seguridad**, 737
- Monitor residente**, 767
- Monitores**, 186-193
- implementación de, usando semáforos, 190-191
 - reanudación de procesos dentro de, 191-193
 - solución al problema de la cena de los filósofos usando, 189-190
 - utilización, 187-189
- Monocultura**, 520
- Montaje**, 373
- dinámico, 248-249, 695-696
 - dinámico y estático, 248-249
 - tabla de, 375, 471
- Morris, Robert**, 521-524
- Movilidad de los usuarios**, 397
- MPEG**, archivos, 654-655
- MS-DOS**, 739-740
- Muerte por inanición**. Véase Bloqueo indefinido
- MULTICS**, sistema operativo, 488-489, 774
- Multidifusión**, 660-661
- Multihebra:**
- activaciones del planificador, 127-128
 - beneficios, 113-115
 - cancelación de una hebra, 123-124
 - conjuntos compartidos de hebras, 125-126
 - datos específicos de la hebra, 126-127
 - llamadas al sistema *exec()*, 123
 - llamadas al sistema *fork()*, 123
 - modelos, 115-117
 - tratamiento de señales, 124-125
- Multimedia**, 651-652
- como término, 651-652
 - cuestiones relativas al sistema operativo, 654
 - datos, 27, 652-653
- Multiprocesamiento:**
- asimétrico, 12, 151
 - simétrico (SMP), 12, 151, 153, 688
- Multiprogramación**, 14-16, 80
- Multitarea**, 15
- MUP (proveedor múltiple UNC)**, 753
- Mútex:**
- adaptativo, 194
 - en Windows XP, 720
- N**
- NDIS (network device interface specification)**, 750
- Negociación**, 452, 656
- NetBEUI, protocolo**, 750
- NetBIOS, protocolo**, 750
- NFS (network file system,sistema de archivos de red)**, 395-400
- operaciones remotas, 400
 - protocolo de montaje, 397
 - protocolo NFS, 398-399
 - traducción de nombres de ruta, 399-400
- NFS V4**, 596
- NFS, protocolo**, 398-399
- NIS (network information service)**, 358
- Nivel**
- de abstracción de hardware (HAL), 717
 - de aplicación, 574
 - de enlace de datos, 573
 - de presentación, 573
 - de red, 573
 - de sesión, 573
 - de transporte, 573
 - del sistema, 655
 - físico, 573
 - físico, seguridad, 511
 - humano, seguridad, 512
- Niveles**, 655
- de prioridad de interrupción, 455
 - de protocolos de red, 533
 - RAID, 426-431
- NLS (national-language-support), API**, 717
- No apropiativo (sin desalojo), (interbloqueos)**, 225
- No bloqueante I/O**, 463-464
- No enmascarable, interrupción**, 454
- No estrictos, sistemas de tiempo real**, 634, 658
- No repudio**, 531
- Nombrado**, 89-91, 358
- de archivos, 334
 - definición, 586
- LDAP (lightweight directory-access protocol)**, 358
- sistema de nombres de dominio, 358

- y comunicación de redes, 567-569
Nombre de ruta, 347-348
 absoluto, 349
 relativo, 349
Nombres:
 en Windows XP, 722-723
 resolución de, 567-569, 756
Novell NetWare, protocolos, 751
NTFS, 742-744
Núcleo, memoria de, 771
Nuevo, estado, 75
Número:
 de bloque relativo, 343
 de página (p), 255
 de prioridad, 191
 mágico (archivos), 340
 personal de identificación (PIN), 539
Números lógicos de cluster, 743
NVRAM (RAM no volátil), 9
 caché, 425
- O**
- Objeto(s):**
 archivo, 334
 contenedores (Windows XP), 550
 despachadores, 195
 en Windows XP, 719
 en caché, 316
 en Linux, 690
 en Windows XP, 722-726
 hardware y software, 485
 libres, 316
 listas de acceso para, 493
 locales (no remotos), 103
 no contenedores (Windows XP), 550
 no remotos (locales), 103
 sección, 96
 trabajo, 731
 usados, 317
- OLE (object linking and embedding)**, 753
open(), operación, 336
Operaciones conflictivas, 201
Operaciones de E/S (servicios del sistema operativo), 36
Operaciones remotas, 400
Ordenación de sucesos, 605-607
Ordenación global, 606
Organización con códigos de corrección de errores (ECC)
 de tipo memoria, 426
Organización de paridad con entrelazado de bits, 426
Organización de paridad con entrelazado de bloques,
 426-427
OS/2, sistema operativo, 713
- P**
- p (número de página)**, 255
pageout (Solaris), 324-325
Paginación, 255-264
 con prioridad, 325
 e intercambio, 421
 en Linux, 693
 Intel Pentium, ejemplo, 273
 invertida, 267-268
 jerárquica, 264-266
 método básico, 255-259
 soporte hardware, 259-261
 tablas de páginas *hash*, 266
 y páginas compartidas, 263-264
 y protección de memoria, 261-263
Paginación bajo demanda, 282-289
 alcance del TLB, 319-320
 con tablas de páginas invertidas, 320
 definición, 282
 e instrucciones de reinicio, 286-287
 estructura de los programas, 320-321
 interbloqueo de E/S, 321-323
 mecanismo básico, 283-287
 prepaginación, 317-318
 pura, 285
 tamaño de página, 318-319
 y rendimiento, 287-289
Paginador, 283
Páginas:
 compartidas, 263-264
 definición, 255
 residentes en memoria, 284
PAM (pluggable authentication modules), 708
Paquete de solicitud de E/S (IRP), 734
Paquetes, 102, 571, 707
Paridad distribuida con entrelazado de bloque, 428
Partición(es), 253, 345, 373-375
 de arranque, 419
 de tamaño fijo, 253
 en bruto (sin formato), 373, 422
 raíz, 374
Particionar discos, 418
Pasarela, 570
Paso de mensajes, 86
 asíncrono, 91
 síncrono, 91
Paso de testigo, 572, 609-610
PCB. Véase Bloque de control de proceso
PCI, bus, 450
PCS (process-contention scope, ámbito de contienda del proceso), 154
PDA. Véase Asistente personal digital
PDE (page-directory entries), 727
Peor ajuste, estrategia de, 254
Pérdida de datos, tiempo medio de, 425
Perfiles, 655
 Perfiles móviles, 754
Períodos, 655
Permisos, 365
Persistencia de la visión, 652
Peterson, solución de, 174-175
PFF (page-fault frequency), 309
Phishing, 512
PIC (position-independent code), 696
Pid (identificador de proceso), 81
Pila, algoritmo de, 297
Pilas, 41, 74
PIN (personal identification number), 539
PIO (programmed I/O, E/S programada), 314, 457
Pipe, mecanismo, 705
Pipes con nombre, 752

- Pistas de disco**, 408
- Planificación:**
- algoritmos de planificación de disco, 412–417
 - C-SCAN, 416
 - FCFS, 413
 - LOOK, 416
 - SCAN, 415
 - selección, 416–417
 - SSTF, 413–415
 - apropiativa, 139–140
 - basada en prioridades, 638
 - con cuota proporcional, 644–645
 - cooperativa, 139
 - CPU. Véase Planificación de la CPU de disco:
 - CineBlitz, 663
 - en sistemas multimedia, 658–659
 - de hebras, 137, 154–155
 - de procesos:
 - en Linux, 684–688
 - planificación de hebras y, 137
 - de trabajos, 15
 - E/S, 464–465
 - en Linux, 684–688
 - multiprocesamiento simétrico, 688
 - planificación de procesos, 684–686
 - sincronización del *kernel*, 686–688
 - en Pthread, 645, 646
 - en Windows XP, 718–719, 758–760
 - no serie, 201
 - por prioridad en finalización de plazo, 644
 - por prioridad monótona en tasa, 642–644
 - serie, 201
- Planificación de la CPU, 15
 - acerca de, 137–138
 - algoritmos de, 142–151
 - criterios, 140–141
 - evaluación de, 161–165
 - implementación de, 165
 - planificación FCFS, 142–143
 - planificación mediante colas multinivel, 148–150
 - planificación mediante colas multinivel realimentadas, 150–151
 - planificación por prioridades, 145–146
 - planificación por turnos, 146–148
 - planificación SJF, 143–145
 - apropiativa, 139–140
 - despachador, papel del, 140
 - en sistemas de tiempo real, 641–645
 - con cuota proporcional, 644–645
 - planificación en Pthread, 645
 - por prioridad en finalización de plazo, 644
 - por prioridad monótona en tasa, 642–644
 - en sistemas multimedia, 658
 - en sistemas multiprocesador, 151–153
 - mecanismos multihebra simétricos, 153
 - métodos, 151–153
 - y afinidad del procesador, 152
 - y equilibrado de carga, 152–153
 - modelos para, 161–165
 - análisis de redes de colas, 163
 - e implementación, 165
- modelo determinista, 162–163
- simulaciones, 164–165
- planificador a corto plazo, papel del, 138
- y ciclo de ráfagas de CPU y de E/S, 138–139
- Planificación de sistemas multiprocesador**, 151–153
 - ejemplos de:
 - Linux, 160–161
 - Solaris, 156–158
 - Windows XP, 158–160
 - mecanismos multihebra simétricos, 153
 - métodos, 151–152
 - y afinidad del procesador, 152
 - y equilibrado de carga, 152
- Planificador a corto plazo (planificador de la CPU)**, 79, 138
- Planificador de E/S con límite de temporización**, 703
- Planificador de procesos**, 77–78
 - a corto plazo, 79
 - a largo plazo, 79
 - a medio plazo, 80
- Planificador de trabajos**, 79
- Planificadores**, 79–80
- Platos (discos)**, 407
- PnP, gestor**, 737–738
- Polimórfico, virus**, 520
- Política**, 51
 - de descarga de páginas (Linux), 693
 - de escritura diferida, 592
 - de escritura durante el cierre, 592
 - de grupo, 755
 - de seguridad, 540
- Portabilidad**, 716
- Portales**, 29
- Posesión de la capacidad**, 494
- Posicionamiento, tiempo de (discos)**, 408
- POSIX**, 713, 715
 - ejemplo de sistema IPC, 93–93
 - en Windows XP, 741
- PPTP, protocolo**, 751
- Practicidad**, 3
- Prepaginación**, 317–318
- Preparada, estado de hebra (Windows XP)**, 719
- Preparado, estado**, 75
- Prestaciones:**
 - asignación de espacio de disco, 384–386
 - de Windows XP, 715–716
 - rendimiento con almacenamiento terciario, 438–442
 - coste, 440–442
 - fiabilidad, 440
 - velocidad, 438–440
 - rendimiento en sistemas de E/S, 475–478
- Primer ajuste, estrategia de**, 254
- Principio del mínimo privilegio**, 484–485
- Prioridad**
 - dinámica, 658
 - estática, 658
 - fija (Solaris), 157
 - paginación con, 325
- Privada, clave**, 529
- Problema de la cena de los filosófos**, 185–186, 189–190
- Problema de los lectores-escritores**, 183–185
- Problema del barbero dormilón**, 207
- Problema del búfer limitado**, 181–182

- Procedimientos de acuerdo**, 625–626
Procesadores de comunicaciones, 564
Procesadores frontales, 475
Procesamiento distribuido, mecanismos de, 751–752
Procesamiento por lotes, archivos de, 339
Proceso del sistema (Windows XP), 739
Proceso, objeto (Windows XP), 720
Procesos, 15
 - componentes, 74
 - comunicación entre. Véase **Comunicación interprocesos**
 - concepto, 73–74
 - contexto del, 80, 682–683
 - cooperativos, 86
 - definición, 73
 - despachados, 79
 - en Linux, 683–684
 - en primer plano, 148
 - en segundo plano, 148
 - en Windows XP, 757
 - entorno de, 682
 - estado, 75
 - fallidos, 626
 - hebras ejecutadas por, 76–77
 - hijo, 725
 - independientes, 86
 - limitados por E/S y limitados por la CPU, 80
 - migración de, 562–563
 - monohebra, 113
 - multihébra. Véase **Multihébra**
 - operaciones sobre los, 81–86
 - creación, 81–85
 - terminación, 85–86
 - padre, 81, 725
 - periódicos, 655
 - pesados, 113
 - planificación de, 77–81
 - programas y, 20, 74, 75
 - trabajos y, 74
 - y cambios de contexto, 80–81
- Programa de arranque**, 419, 521
Programa de control, 5
Programa, archivos de, 334
Programas:
 - de arranque, 6, 64
 - de usuario (tareas de usuario), 73, 694
 - del sistema, 49–50
 - informáticos. Véase **Programas de aplicación**
 - y procesos, 74, 75. Véase también **Programas de aplicación**
- Programas de aplicación**, 3
 - desinfección de, 544–545
 - procesamiento en varios pasos de, 246
 - procesos y, 19
 - utilidades del sistema, 49–50
- Promedio exponencial**, 144
Protección, 361–364, 483–507
 - archivos, 334
 - control de acceso para, 362–363
 - de E/S, 468–469
 - de memoria, 252
 - de sistemas de archivos, 361–365
 - de virus, 545–546
 - dominio de, 485–489
- estructura, 486–487
MULTICS, ejemplo, 488–489
UNIX, ejemplo, 487–488
 en entorno paginado, 261–263
 en sistemas informáticos, 24–25
 matriz de acceso como modelo de, 489–493
 - control de acceso, 495–496
 - implementación, 493–495
 - objetivos de la, 483–484
 - permisos, 365
 - principio del mínimo privilegio, 484–485
 - retroimplementado, 364
 - revocación de derechos de acceso, 496–497
 - seguridad y, 509
 - servicio del sistema operativo, 37
 - sistemas basados en capacidades, 497–500
 - Hydra, 498–499
 - sistema CAP de Cambridge, 499–500
 - sistemas basados en el lenguaje, 500–505
 - imposición de reglas basadas en compilador, 500–503
 - Java, 503–505
 - tratamiento de errores, 468
- Protocolo(s)**:
 - basados en marcas temporales, 203–204
 - de bloqueo en dos fases, 202
 - de bloqueo, 202–203, 613–616
 - de confirmación en dos fases (2PC), 610–612
 - de confirmación, 610
 - de control de transmisión (TCP), 575
 - de datagramas de usuario (UDP), 575
 - de encaminamiento, 570
 - de herencia de prioridad, 195, 641
 - de mayoría, 614
 - de montaje, 397
 - de nivel de transporte (TCP), 533
 - de resolución de direcciones (ARP), 580
 - de transporte en tiempo real (RTP), 660
 - en redes Windows XP, 750–753
 - ligero de acceso al directorio (LDAP), 358, 755
 - preferencial, 614–615
 - sin memoria del estado, 662
- Proxy, cortafuegos**, 547
Proyecto, 158
Prueba de penetración, 540
PTBR (page-table base register), 260
Pthread, planificación en, 645, 646
Pthreads, 118
 - planificación, 154–155
 - sincronización en, 197
- PTLR (page-table length register)**, 263
Pública, clave, 529
Puerta trasera, 460, 514
Puertos, 314, 450
Pull migration, 152
Puntero de archivo, 337
Puntero de entrada de pila, 516–517
Puntero de posición actual del archivo, 335
Punto de montaje, 354, 749
Puntos de cancelación, 124
Puntos de comprobación, 200
Puntos de desalojo, 639
Push migration, 152

R

- RAID (redundant arrays of inexpensive disks)**, 423-432
 estructuración, 424
 matriz, 424
 mejora de la fiabilidad, 424-425
 mejoras en las prestaciones, 425
 niveles, 426-431
 problemas, 432
- Raíz del índice**, 743
- Raíz, partición**, 374
- RAM (random-access memory)**, 7
- RAM no volátil (NVRAM)**, 9
 caché, 425
- Rango de tiempo real (planificadores Linux)**, 684
- Ranuras de página**, 423
- RBAC (role-based access control)**, 495
- RC 4000, sistema operativo**, 773
- Reasignación de direcciones**, 245-247
- Reasignación**, 245
- Recolección de memoria**, 61, 353
- Reconfiguración**, 576-577
- Recorte web**, 28
- Recuento**, 388
- Recuperación:**
 comprobación de coherencia, 392
 copia de seguridad y restauración, 393-394
 de archivos y directorios, 392
 de fallos, 577
 de interbloqueos, 235-236
 mediante terminación de procesos, 235
 por apropiación de recursos, 235
 Windows XP, 744-745
- Recursos, compartición de**, 558
- Red**
 de área de almacenamiento (SAN), 14, 411, 412
 de área extensa (WAN, wide-area network), 15, 25, 564-565
 de área local (LAN, local area network), 13, 25, 563-564
 de área metropolitana (MAN, metropolitan-area network), 25
 de área pequeña, 25
 dispositivo de, 462, 702
 inalámbrica, 28
 inicio de sesión de, 358
 parcialmente conectada, 566
 privada virtual (VPN), 533, 751
- Redes**. Véase también Red de área local (LAN); Red de área extensa (WAN)
 amenazas, 520-525
 cuestiones de diseño, 577-579
 de área metropolitana (MAN), 25
 de área pequeña, 25
 definición, 25
 ejemplo, 579-581
 en Linux, 705-707
 en Windows XP, 749-756
 Active Directory, 755
 dominios, 754-755
 interfaces, 750
 procesamiento distribuido, mecanismo de, 751-753
 protocolos, 750-751
- redirectores y servidores, 753-755
 resolución de nombres, 756
- estructura de comunicaciones**, 567-572
 contienda, 572
 estrategias de conexión, 571
 estrategias de encaminamiento, 569-570
 estrategias de paquetes, 570-571
 nombrado y resolución de nombres, 567-569
- protocolos de comunicaciones**, 572-575
 robustez, 575-577
 seguridad, 512
 tipos de, 563
 topología, 565-567
- Redirectores**, 753-754
- Reducción de escala (downsizing)**, 559
- Redundancia**, 424. Véase también RAID
- Reed-Solomon, códigos**, 428
- Reentrant, código (código puro)**, 263
- Referencia de archivo**, 743
- Referencia, bits de**, 299
- Referencia, cadena de**, 293
- Regiones de memoria virtual**, 692
- Registrador de pulsaciones de tecla**, 520
- Registro(s)**, 41, 50, 738
 base, 244, 245
 base de la tabla de páginas (PTBR), 260
 base del archivo, 743
 de controladores, módulo (Linux), 679-680
 de direcciones de memoria, 247
 de escritura anticipada, 198-199
 de instrucciones, 8
 de lectura anticipada, 198-199
 de longitud de la tabla de páginas (PTLR), 263
 de reubicación, 247
 límite, 244, 245
 lógicos, 342
 maestro de arranque, 419
 para tablas de páginas, 259-260
- Regla de planificación**, 759
- Regla del 50 por ciento**, 254
- Relación de confianza**, 755
 cruzada, 755
 transitiva, 755
 unidireccional, 755
- Relativo, nombre de ruta**, 349
- release()**, operación, 337
- Relleno de ceros bajo demanda**, técnica, 290
- Reloj, algoritmo del**, 299
- Reloj de la CPU**, 244
- Reloj lógico**, 606
- Relojes**, 463-464
- Remoto, sistema de archivos**, 356-359
- Rendezvous**, 91
- Reparación, tiempo medio de**, 424
- Replicación**, 430
 de archivos (sistemas de archivos distribuidos), 597
- Reposiciónamiento (dentro de archivos)**, 335
- Representación de datos externa (XDR)**, 100
- Representación de un proceso en Linux**, 77
- Reproducción local**, 652
- Reproducción, ataques de**, 510
- Reproductor de medios**, 661

- Reserva**
de dispositivos, 467-468
de sectores, 420
de recursos, 657-658
en caliente, discos de, 430
- Resolución de conflictos, módulo (Linux)**, 680-681
- Resolución**, 319
de enlaces, 351
de nombres, 567-569
y tamaño de página, 318
- Responsabilidad (servicios del sistema operativo)**, 37
- Restauración:**
de datos, 392-393
del estado, 80
- Restaurar el sistema (Windows XP)**, 738
- Resumen de mensajes (valor hash)**, 530
- Retardo**, 656
- Retención y espera, condición de (interbloqueos)**, 224-225
- Retrollamada**, 600
- Reubicación, registro de**, 247
- Revocación de derechos de acceso**, 496-497
- RMI**. Véase Invocación de métodos remotos
- Robo de ciclos**, 458
- Robo de servicio**, 510
- Robustez**, 575-577
- Roles**, 495
- ROM**. Véase Memoria de sólo lectura
- RSX, sistema operativo**, 777
- RTF (rich text format)**, 545
- R-timestamp**, 203
- RTP (real-time transport), protocolo**, 660
- Ruptura de la confidencialidad**, 510
- Ruptura de la disponibilidad**, 510
- Ruptura de la integridad**, 510
- Ruta de búsqueda**, 348
- Rutina de servicio de interrupciones, mitad inferior**, 687
- Rutina de servicio de interrupciones, mitad superior**, 687
- Rutina de tratamiento de suprallamada**, 127
- RW (lectura-escritura), formato**, 21
- S**
- Salida de datos, registro de**, 452
- Salvaguarda del estado**, 80
- SAN (storage-area network)**, 14, 411, 412
- SATA (serial ATA) buses**, 409
- SCAN (ascensor), algoritmo de planificación**, 415, 659
- SCAN circular (C-SCAN), algoritmo de planificación**, 416
- SCOPE, sistema operativo**, 777
- Script kiddies**, 517
- SCS (system-contention scope, ámbito de contienda del sistema)**, 154
- SCSI (small computer-systems interface)**, 10
buses, 409
destinos, 411
iniciador, 411
- Sección**
de datos (del proceso), 74
de entrada, 173
de salida, 173
de texto (del proceso), 74
objeto, 727
restante, 173
- Sección crítica, problema de la**, 173-174
Peterson, solución de, 174-175
y semáforos, 178-182
e interbloqueos, 181-182
implementación, 180-181
inanición, 181-182
utilización, 179
y sincronización hardware, 175-178
- Secreto maestro (SSL)**, 534
- Secreto premaestro (SSL)**, 534
- Sector de arranque de la partición**, 371
- Sector de disco**, 408
- Sectores de reserva**, 748
- Secuencia segura**, 227
- Secuenciales, dispositivos**, 460
- Secuenciamiento automático de trabajos**, 767
- Secuestro de sesión (*hijacking*)**, 511
- Secundaria, memoria**, 286
- Segmentación**, 269-271
definición, 269
hardware, 270-271
Intel Pentium, ejemplo, 271-275
método básico, 269-270
- Segmentos, tabla de**, 270
- Segunda oportunidad, algoritmo de sustitución de páginas (algoritmo del reloj)**, 299
- Segundo sistema de archivos extendido (ext2fs)**, 698-700
- Seguridad**. Véase también Amenazas relacionadas con los programas; Protección; Autenticación de usuario
amenazas del sistema y de la red, 520-525
denegación de servicio, 524-525
escaneo de puertos, 524
gusanos, 521-524
autenticación de usuario, 535-539
biométrica, 539
contraseñas, 536-539
clasificaciones de, 547-549
como problema, 509-513
cortafuegos, 546-547
criptografía como herramienta de, 525-535
cifrado, 526-533
implementación, 533-534
SSL, ejemplo, 534-535
- de los tipos (Java), 504
- en Linux, 707-709
autenticación, 707-708
control de acceso, 708-709
- en sistemas informáticos, 24
- en Windows XP, 549-550, 715
- implementación de, 539-546
auditoría, 546
contabilización, 546
detección de intrusiones, 541-545
evaluación de la vulnerabilidad, 540-541
política de seguridad, 540
protección frente a virus, 545-546
registro, 546
- mediante la oscuridad, 541
- niveles de, 511-512
- protección y, 509
- servicio del sistema operativo, 37
- Windows XP, 745

- Seguro, sistema**, 509
Semáforos, 178-182
 - binario, 179
 - contador, 179
 - definición, 178
 - e inanición, 181-182
 - e interbloqueos, 181-182
 - implementación de monitores usando, 190-191
 - implementación, 180-181
 - utilización, 179
 Windows XP, 720
- Semántica**:
- de archivos compartidos inmutables, 360
 - de coherencia, 359-360
 - de copia, 466
 - de sesión, 360
- Semilla**, 538
- Señales**:
- Linux, 704
 - UNIX, 110, 124-125
- Serialización**, 200-202
 - de objetos, 103
- Servicio de información de red (NIS)**, 358
- Servicio del archivo de registro**, 745
- Servicio remoto, mecanismo de**, 594
- Servicios de archivos con memoria del estado**, 594-595
- Servicios de archivos sin memoria del estado**, 594-595
- Servicios del sistema operativo**, 35-37
- Servidor de cluster**, 598
- Servidores**, 4
 - blade*, 13
 - definición, 585
 - en SSL, 533-534
- Sesión de archivo**, 360
- Sesión, semántica de**, 360
- Sesiones de comunicaciones**, 571
- Shells**, 37, 108-110
- Signaturas**, 542
- Simétrico, cifrado**, 527-528
- Simulaciones**, 164
- Sin conexión, mensajes**, 571
- Sin desalojo, planificación**, 138
- Sin disco, cliente**, 588
- Sin formato, disco**, 302, 373
- Sin formato, particiones**, 422
- Sincronización de procesos**
 - acerca de, 171-173
 - ejemplos:
 - Java, 193
 - Linux, 196-197
 - Pthreads, 197
 - Solaris, 194-195
 - Windows XP, 195-196
 - monitores, 186-193
 - implementación usando semáforos, 190-191
 - rearudación de procesos dentro de, 191-193
 - solución al problema de la cena de los filósofos, 189-190
 - utilización, 187-189
 - problema de la cena de los filósofos, 185-186, 189-190
 - problema de la sección crítica, 173-174
 - Peterson, solución de, 174-175
- solución hardware, 175-178
- problema de los lectores-escritores, 183-185
- problema del búfer limitado, 182-183
- semáforos para, 178-182
- y transacciones atómicas, 197-204
 - modelo del sistema, 197-198
 - puntos de comprobación, 199-200
 - recuperación basada en registro, 198-199
 - transacciones concurrentes, 200-204
- Sincronización**, 91. *Véase también Sincronización de procesos*
- Síncrono, dispositivo**, 460
- Sistema**
 - archivos del, 348
 - basado en la Web, 30-31
 - cliente, 28
 - común de archivos Internet (CIFS), 358
 - de archivos de proceso, /proc (Linux), 700-702
 - de archivos de red (NFS). *Véase NFS*
 - de descomposición binaria (Linux), 689
 - de detección de intrusiones, 542
 - de nombres de dominio (DNS), 358, 568
 - de prevención de intrusiones, 542
 - en ejecución, 64
 - entre iguales, 30
 - integrado, 634
 - operativo de red, 26, 559-561
 - tolerante a fallos, 577
- Sistemas de archivos**, 333, 369-371
 - básico, 370
 - creación de, 345-346
 - de red (NFS), 395-400
 - en niveles, 370
 - extendido, 371
 - implementación de, 371-377
 - montaje, 373
 - particiones, 373-375
 - sistemas virtuales, 375-377
 - Linux, 696-702
 - lógico, 370
 - montaje de, 354-355
 - orientados a transacciones y basados en registro, 393-395
 - problemas de diseño con, 370
 - remotos, 356-359
 - WAFL, 400-402
- Sistemas de bases de datos**, 197
- Sistemas de información distribuidos (servicios de denominación distribuidos)**, 358
- Sistemas de mano**, 27-28
- Sistemas de protección basados en capacidades**, 497-500
 - Hydra, 498-499
 - sistema CAP de Cambridge, 499-500
- Sistemas de protección basados en el lenguaje**, 500-505
 - imposición de reglas basadas en compilador, 500-503
 - Java, 503-505
- Sistemas de seguridad crítica**, 634
- Sistemas de servidor de archivos**, 28
- Sistemas de un solo procesador**, 11, 137
- Sistemas distribuidos**, 25-26, 555-583
 - definición, 557
 - sistemas operativos de red, 559-561
 - sistemas operativos distribuidos, 561-563
 - ventajas de los, 557-559

- Sistemas en tiempo real**, 26–27, 633–650
 características no necesarias en, 636
 características, 634–636
 definición, 633
 estrictos, 634, 658
 huella, 635
 implementación, 637–641
 kernels apropiativos, 638–639
 minimización de la latencia, 639–641
 planificación basada en prioridades, 638
 no estrictos, 634, 658
 planificación de la CPU, 641–645
 traducción de direcciones, 636–637
 VxWorks, ejemplo, 645–648
- Sistemas fuertemente acoplados**, 13–14
- Sistemas informáticos**:
- almacenamiento en, 7–9
 - amenazas, 520–525
 - arquitectura
 - sistemas de un solo procesador, 11–13
 - sistemas en *cluster*, 13–14
 - sistemas multiprocesador, 11–13
 - de propósito especial, 26–28
 - sistemas de mano, 27–28
 - sistemas embebidos en tiempo real, 26–27
 - sistemas multimedia, 27
 - estructura de E/S, 9–11
 - funcionamiento de, 6–7
 - gestión de almacenamiento, 20–24
 - almacenamiento en caché, 22–23
 - gestión del almacenamiento masivo, 21
 - sistemas de E/S, 23–24
 - gestión de memoria, 19–20
 - gestión de procesos, 19
 - gestión del sistema de archivos, 20–21
 - interactivo, 15
 - protección en, 24–25
 - seguridad en, 24
 - seguridad, 510
 - sistemas de propósito general, 26–28
 - sistemas de mano, 27–28
 - sistemas embebidos en tiempo real, 26–27
 - sistemas multimedia, 27
 - sistemas distribuidos, 25–26
 - tradicionales, 28–29
 - Sistemas mecánicos microelectrónicos (MEMS), 435
 - Sistemas multimedia, 27, 651–667
 - características, 653
 - CineBlitz, ejemplo, 663–665
 - compresión, 654–655
 - gestión de red, 660–663
 - kernels* en, 655–657
 - planificación de disco, 657–660
 - planificación de la CPU, 657
 - Sistemas multiprocesador (sistemas paralelos, sistemas fuertemente acoplados), 11–14
 - Sistemas operativos, 1
 - asignador de recursos, 5
 - características, 3
 - controlados mediante interrupciones, 16
 - definición, 3, 5–6
 - distribuidos, 561–563
 - en tiempo real, 26–27
 - estructura, 14–16, 52–57
 - en niveles, 53–55
 - microkernels*, 55–57
 - módulos, 56–57
 - simple, 53
 - funcionamiento, 3–6
 - huésped, 60
 - implementación, 52
 - interfaz de usuario, 4–5, 37–39
 - mechanismos, 51
 - objetivos del diseño, 50–51
 - operaciones del:
 - modo, 16–18
 - y temporizador, 18
 - pioneros, 765–771
 - E/S solapada, 769–771
 - sistemas informáticos compartidos, 766–769
 - sistemas informáticos dedicados, 765–766
 - políticas, 51
 - red, 25
 - seguridad en, 512
 - servicios proporcionados, 35–37
 - vista del sistema, 5
 - Sistemas paralelo, 11–14
 - SJF, algoritmo de planificación, 143–145
 - SMB (server-message-block), 750
 - SMP. Véase Multiprocesamiento simétrico
 - Sniffing(husmear)*, 536
 - Sobreasignación (de memoria), 290
 - Sobrepaginación, 305–309
 - causa de, 305–307
 - definición, 305
 - frecuencia de fallos de página, estrategia, 309
 - modelo del conjunto de trabajo, 307–308
 - SOC (system-on-chip, sistema en un chip), estrategia, 635
 - Socket, 97–99
 - interfaz, 462
 - TCP, 97
 - TCP orientados a conexión, 97
 - UDP, 97
 - UDP sin conexión, 97
 - Software, interrupción, 456
 - Software, objetos, 485
 - Solaris:
 - ejemplo de planificación, 156–158
 - gestión del espacio de intercambio, 423
 - memoria virtual en, 324–325
 - sincronización en, 194–195
 - Sólo escritura, dispositivos de, 460
 - Sólo lectura, discos de, 434
 - Sólo lectura, dispositivos de, 460
 - Sondeo, 452–453
 - Soporte de idioma nacional (NLS), 717
 - Soporte de lenguajes de programación, 50
 - Soporte internacional, 717
 - Sostenido, ancho de banda, 438
 - Spooling*, 770–771
 - Spyware*, 513
 - SSL 3.0, 534–535
 - SSTF, algoritmo de planificación, 413–415
 - Stream, módulos, 473

Índice

- STREAMS, mecanismo, 473-475**
Stub, rutinas, 752
Stubs, 102, 249
Subarchivo de datos, 341
Subarchivo de recursos, 341
Subsistema de E/S, 23
 kernel, 6, 464-470
 procedimientos supervisado : por, 470
Subsistemas de entorno, 716
Subsistemas de protección (Windows XP), 717
Suceso, objeto (Windows XP), 720
Sucesos, 195
Sucesos, ordenación de, 605-607
Sucio (o de modificación), bit, 292
Sujeto de servidor (Windows XP), 549
Sujeto simple (Windows XP), 549
Suma de comprobación, 580
Superbloque, 372
Superbloque, objeto, 376, 696
Suplantación, 547
Supralllamadas, 127
Sustitución basada en prioridades, algoritmo de, 306
Sustitución de páginas, 290-302. Véase también Asignación de marcos
 algoritmo, 293
 algoritmos de búfer de páginas, 301
 global y local, 304
 mecanismo básico, 291-294
 sustitución de páginas basada en contador, 300
 sustitución de páginas FIFO, 294-295
 sustitución de páginas LRU, 296-298
 sustitución de páginas mediante aproximación LRU, 298-300
 sustitución óptima de páginas, 296
 y aplicaciones, 301
Sustitución de sectores, 420
Sustitución óptima de páginas, algoritmo, 296
- T**
- Tabla de páginas, 255-259, 285, 727**
 con correspondencia directa, 264
 en *cluster*, 267
 hardware para almacenar, 259-261
 hash, 266
 invertidas, 267-268, 320
Tabla(s), 280
 de archivos abiertos de cada proceso, 372
 de archivos abiertos, 336
 de asignación de archivos (FAT), 382
 de contenidos del volumen, 345
 de despacho de interrupciones (Windows XP), 721
 de encamínamiento, 569
 de montaje, 375, 471
 de objetos, 725
 de segmentos, 270
 global de archivos abiertos, 372
 global de descriptores (GDT), 272
 hash, 377-378
 local de descriptores (LDT), 272
 maestra de archivos, 372
Tamaño de página, 318-319
- Tareas:**
 caducadas (Linux), 684
 Linux, 682-683
 VxWorks, 645
Tarjetas de control, 44, 767, 768
Tarjetas de E/S con control maestro del bus, 457
Tasa
 de acierto, 261
 de fallos de página, 288
 de procesamiento, 141
 de transferencia, 656
TCB (trusted computer base), 548
TCP (transmission control protocol), 575
TCP/IP, conjunto de protocolos, 751
TDI (transport driver interface), 750
Técnica basada en coordinadores múltiples (control de concurrencia), 614
Técnica de precodificación, 90
Tecnología hiperhebra, 153
telnet, 559
Temporizador, objeto, 720
Temporizadores, 462-463
 de intervalo programable, 462
 variable, 18
Tenex, sistema operativo, 777
Terciario, almacenamiento. Véase Almacenamiento terciario
Terminación:
 de procesos, 85-86, 235
 en cascada, 86
Terminada, estado de hebra (Windows XP), 719
Terminado, estado, 75
Testigo, 572, 609
Testigo de acceso de seguridad (Windows XP), 549
Texto, archivos de, 334
THE, sistema operativo, 772-773
Tiempo:
 compartido (multitarea), 15
 de acceso efectivo, 287
 de carga, 247
 de compilación, 246
 de ejecución, 141, 247
 de espera, 141
 de respuesta, 15, 141
 efectivo de acceso a memoria, 261
 medio de pérdida de datos, 425
 medio de reparación, 424
 medio entre fallos, 424
 real, clase de, 158
Tipo abstracto de datos, 335
Tipos de objetos, 376, 724
TLB (translation look-aside buffer). Véase Búfer de traducción directa
 alcance del, 319-320
 fallo de, 260
Tolerancia a fallos, 12, 578, 745-748
Topología de red, 565-567
Torvalds, Linus, 671
Trabajos, procesos y, 74
Traducción de nombres de ruta, 399-400
Tramas, 571
Transacciones 197. Véase también Transacciones atómicas abortadas, 198

- anuladas, 198
confirmadas, 198
definición, 700
en Linux, 700
en sistemas de archivos con estructura de registro, 393-395
- Transacciones atómicas**, 176, 197-204
concurrentes, 200-204
y protocolos basados en marcas temporales, 202-203
y protocolos de bloqueo, 202-203
y serialización, 200-202
modelo del sistema, 197-198
registro de escritura anticipada, 198-199
y puntos de comprobación, 199-200
- Transarc DFS**, 598
- Transferencia de datos dirigida por interrupción**, 314
- Transferencia remota de archivos**, 560-561
- Transformaciones de caja negra**, 528
- Transición**, estado de hebra (Windows XP), 719
- Transmisión de flujos (streaming)**, 652
- Transparencia**, 577-579, 586-587
- Tratamiento de señales**, 110, 124-125
definidas por el usuario, 124
predeterminadas, 124
- Triple DES**, 528
- Triplemente indirecto, bloque**, 384
- Tripwire**, sistema de archivos, 544
- Tunel**, virus, 520
- twofish**, algoritmo, 528
- U**
- Ubicación de archivo**, 334
- Ubicación, independencia de**, 587
- Ubicación, transparencia de**, 587
- UDP (user datagram protocol)**, 575
- UFD (user file directory)**, 347
- UFS (UNIX file system)**, 371
- UI (user interface)**, 36-37
- UID efectivo**, 25
- uid root (Linux)**, 708
- UNC (uniform naming convention)**, 752
- UNC, proveedor múltiple (MUP)**, 753
- UNICODE**, 717
- Unidad central de procesamiento**. Véase CPU
- Unidad componente**, 586
- Unidad de ejecución de instrucciones**, 740
- Unidad de gestión de memoria (MMU, memory-management unit)**, 247-248, 728
- Unidades lógicas**, 411
- Unidifusión**, 660-661
- UNIX, sistema operativo**:
comunicación de dominio en, 487-488
intercambio en, 251
permisos en, 365
semántica de coherencia, 360
señales, 110, 124-125
shell y función historial (proyecto), 108-111
sistema de archivos (UFS), 371
y Linux, 737
- Usados, objetos**, 317, 691
- USB (universal serial bus)**, 409
- Usuario(s)**, 4-5, 356
- cuentas de, 549
movilidad de, 397
- Utilidad, almacenamiento de**, 431
- Utilidades del sistema**, 49-50, 676-677
- Utilización de recursos**, 4
- V**
- Vaciar**, 260
- VAD (virtual address descriptors)**, 731
- Válido-inválido**, bit, 262
- Valor de tiempo real (Linux)**, 160
- Valor normal (nice) (Linux)**, 160, 685
- Variable automática**, 515
- Variable, clase**, 158
- VDM**. Véase Máquina DOS virtual
- Vector**:
de argumentos, 682
de bits (mapa de bits), 386
de entorno, 682
de interrupciones, 7, 454
programa, 521
- Velocidad angular constante (CAV)**, 410
- Velocidad de operaciones**:
para dispositivos de E/S, 459, 460
- Velocidad de transferencia (discos)**, 408, 409
- Velocidad lineal constante (CLV)**, 410
- Velocidad relativa**, 173
- Ventanas de explorador emergentes**, 514
- vfork() (fork para memoria virtual)**, 290
- VFS (virtual file system)**, 375-377, 696-698
- Víctima, marco**, 292
- Virtual, dirección**, 247
- Virus**, 518-520, 545-546
cifrado, 520
de código fuente, 520
encubierto, 520
lanzador de, 518
multiparte, 520
polimórfico, 520
- Vista**, 727
- VMS, sistema operativo**, 777
- VMware**, 60
- vnodo**, 376
número de (NFS V4), 599
- Volumen, bloque de control de**, 371
- Volúmenes**, 345, 599
copias ocultas de, 749
- von Neumann, arquitectura de**, 8
- VPN (virtual private network)**, 533, 751
- VxWorks**, 645-648
- W**
- WAFL, sistema de archivos**, 400-402
- WAN**. Véase Red de área extensa
- WebDAV, protocolo**, 751
- Win32, biblioteca de hebras de**, 118-121
- Windows 2000**, 714, 717
- Windows NT**, 713-714
- Windows XP**, 713-764
altas prestaciones, 715-716
ampliabilidad, 716
compatibilidad con aplicaciones, 715

Windows XP (Cont.)

- componentes del sistema, 717-739
 - executive. Véase Windows XP executive
 - kernel*, 717-722
 - nivel de abstracción hardware, 717
- conexión de red, 749-756
 - Active Directory, 755
 - dominios, 754-755
 - interfaces, 750
 - procesamiento distribuido, mecanismos de, 751-753
 - protocolos, 750-751
 - redirectores y servidores, 753-754
 - resolución de nombres, 756
- ejemplo de hebras, 128, 129
- ejemplo de planificación, 158-160
- ejemplo de sistema IPC, 95-96
 - Windows de 16 bits, 740
 - Windows de 32 bits, 740-741
- fiabilidad de, 715
- historia, 713-714
- interfaz de programación, 756-763
 - acceso a los objetos del *kernel*, 756
 - compartición de objetos entre procesos, 756-757
 - comunicación interprocesos, 760-761
 - gestión de memoria, 761-763
 - gestión de procesos, 757-760
- memoria virtual en, 323-324
- portabilidad, 716
- principios de diseño, 714-717
- seguridad, 715
- sincronización en, 195-196
- sistema de archivos, 742-749
 - compresión y cifrado, 748
 - copias ocultas de volúmenes, 749
 - diario de cambios, 749
 - gestión de volúmenes y tolerancia a fallos, 745-746
- NTFS, árbol B+, 743-744
- NTFS, estructura interna, 742-744
- NTFS, metadatos, 744
- puntos de montaje, 749
- recuperación, 744-745

seguridad, 745

- subsistema de seguridad, 741-742

subsistemas de entorno, 739-742

- inicio de sesión, 741-742

MS-DOS, 739-740**POSIX, 741****seguridad, 741-742****Win32, 741****versiones de sobremesa, 714****Windows XP executive, 722-739**

- arranque, 738-739

gestor de caché, 734-736**gestor de E/S, 733-734****gestor de memoria virtual, 726-731****gestor de objetos, 722-726****gestor de procesos, 731-732****gestores plug-and-play y de administración de energía, 737-738****llamada a procedimientos locales, 732-733****monitor de referencia de seguridad, 737****Registro, 738****Windows, intercambio en, 251****Winsock, 752****Witness, 226****World Wide Web, 357****WORM (escritura una vez, lectura muchas veces), formato, 21****WORM (write-once, read-many-times), discos, 434****W-timestamp, 203****X****XDR (external data representation), 100****XDS-940, sistema operativo, 771-772****Xerox, 38****XML, cortafuegos, 547****Z****Zombi, sistemas, 524****Zona desmilitarizada (DMZ), 546****Zonas (Linux), 688**

Otro momento decisivo en la evolución de los sistemas operativos

Los sistemas operativos con huella pequeña, como los utilizados en los dispositivos de mano que emplean los bebés de dinosaurio de la portada, representan sólo una de las múltiples aplicaciones avanzadas que podrá encontrar el libro *Fundamentos de sistemas operativos, séptima edición* de Silberschatz, Galvin y Gagne.

Al mantenerse actualizado, preservando la relevancia y adaptándose a las necesidades didácticas más recientes, este texto fundamental continúa siendo el libro de referencia para cursos de sistemas operativos. Esta séptima edición no sólo presenta los sistemas más recientes y relevantes, sino que también los analiza más en profundidad para presentar los conceptos fundamentales que han permanecido constantes a lo largo de la evolución de los sistemas operativos actuales. Con esta fuerte base conceptual, los estudiantes pueden comprender con mayor facilidad los detalles relacionados con cada sistema específico.

Novedades

- Tratamiento más detallado de la perspectiva del usuario en el Capítulo 1.
- Aumento del material de diseño de todos los elementos de un sistema operativo.
- Un nuevo capítulo sobre sistemas de tiempo real y sistemas embebidos (Capítulo 19).
- Un nuevo capítulo sobre sistemas multimedia (Capítulo 20).
- Material adicional sobre seguridad y protección.
- Material adicional sobre programación distribuida.
- Nuevos ejercicios al final de cada capítulo.
- Nuevos proyectos de programación al final de cada capítulo.
- Nuevo tratamiento de los aspectos didácticos, centrado en el estudiante con el fin de mejorar el proceso de aprendizaje.

www.mcgraw-hill.es



9 788448 146412

The McGraw-Hill Companies

ISBN: 84-481-4641-7