

Find your solutions manual here!

El Solucionario

www.elsolucionario.net



Subscribe RSS



Find on Facebook



Follow my Tweets

Encuentra en nuestra página los Textos Universitarios que necesitas!

Libros y Solucionarios en formato digital

El complemento ideal para estar preparados para los exámenes!

*Los Solucionarios contienen TODOS los problemas del libro resueltos
y explicados paso a paso de forma clara..*

Visítanos para descargarlos GRATIS!

Descargas directas mucho más fáciles...

WWW.ELSOLUCIONARIO.NET

Biology

Investigación Operativa

Computer Science

Physics

Estadística

Chemistry

Matemáticas Avanzadas

Geometría

Termodinámica

Cálculo

Electrónica

Circuitos

Math

Business

Civil Engineering

Economía

Análisis Numérico

Mechanical Engineering

Electromagnetismo

Electrical Engineering

Álgebra

Ecuaciones Diferenciales

Find your solutions manual here!

1

INTRODUCCIÓN

Un ordenador moderno consiste de uno o más procesadores, alguna memoria principal, discos, impresoras, un teclado, una pantalla, interfaces de red y otros dispositivos de entrada/salida. Se trata de un sistema muy complejo. Resulta un trabajo extremadamente difícil escribir programas que controlen todos esos componentes y los utilicen de una forma correcta, no digamos óptima. Por esa razón, los ordenadores están equipados con una capa de software que se denomina el **sistema operativo**, cuya función es gestionar todos esos dispositivos y proporcionar a los programas del usuario una interfaz con el hardware más sencilla. Estos sistemas constituyen el tema de este libro.

En la Figura 1-1 se muestra el emplazamiento del sistema operativo. En el fondo está el hardware, que, en muchos casos, está compuesto a su vez de dos o más niveles (o capas). El nivel más bajo contiene dispositivos físicos, consistentes de chips de circuitos integrados, cables, fuentes de alimentación, tubos de rayos catódicos y otros dispositivos físicos similares. Cómo se construyen y cómo funcionan esos dispositivos es competencia del ingeniero electrónico.

A continuación viene el **nivel de la microarquitectura**, en el cual los dispositivos físicos se agrupan para formar unidades funcionales. Este nivel contiene típicamente algunos registros internos a la CPU (*Central Processing Unit*; Unidad Central de Procesamiento) y una ruta de datos conteniendo una unidad aritmético-lógica. En cada ciclo de reloj se extraen uno o dos operandos de los registros y se combinan en la unidad aritmético-lógica (por ejemplo mediante la operación de suma o el AND lógico). El resultado se almacena en uno o más registros. En algunas máquinas es el software quien controla el funcionamiento de la ruta de datos. Dicho software se denomina el **micropograma**. En otras máquinas son los circuitos del hardware quienes controlan directamente la ruta de datos.

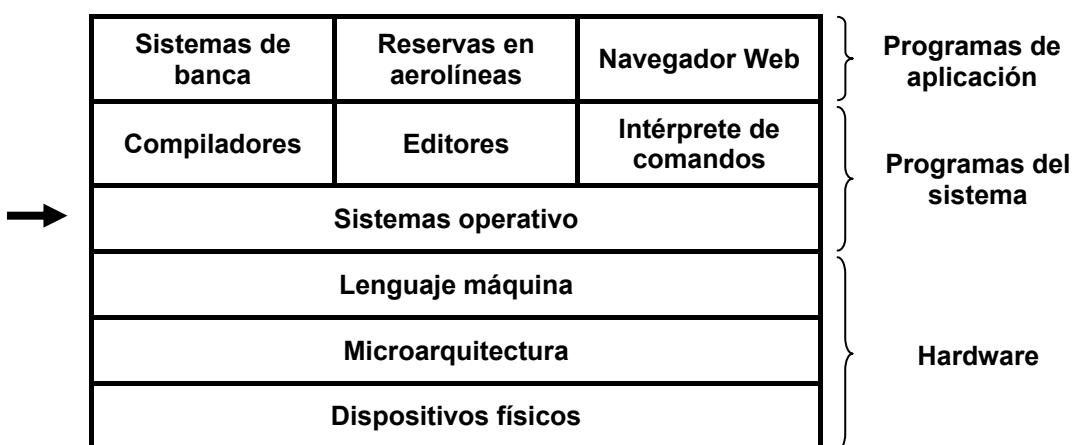


Figura 1-1. Un ordenador consta de hardware, programas del sistema y programas de aplicación.

2.5 ESBOZO DE LOS PROCESOS EN MINIX

Habiendo completado nuestro estudio de los principios de la gestión de procesos, comunicación entre procesos y planificación, vamos a echar una mirada a como se aplican en MINIX. De forma diferente que en UNIX, cuyo núcleo es un programa monolítico que no está descompuesto en módulos, MINIX es él mismo una colección de procesos que se comunican entre si y con los procesos de usuario utilizando una única primitiva de comunicación entre procesos –paso de mensajes. Este diseño proporciona una estructura más modular y flexible, haciendo fácil, por ejemplo, sustituir el sistema de ficheros entero por otro completamente diferente, sin tener ni que recompilar el núcleo.

2.5.1 La Estructura Interna de MINIX

Vamos a comenzar nuestro estudio de MINIX inspeccionando a vista de pájaro el sistema. MINIX está estructurado en cuatro capas, con cada capa realizando una función bien definida. Las cuatro capas se ilustran en la Figura 2-26.

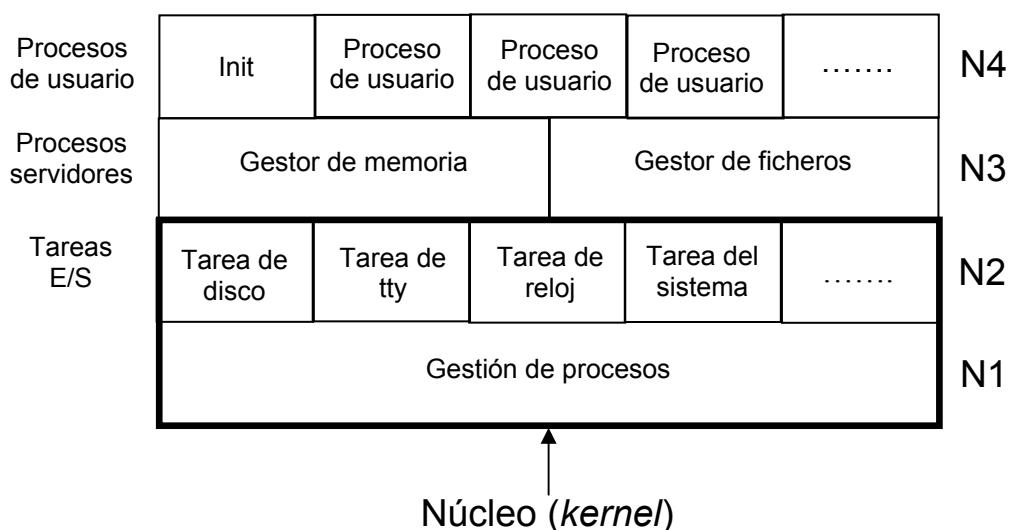


Figura 2-26. MINIX está estructurado en cuatro capas.

La capa del fondo captura todas las interrupciones y traps, realiza la planificación, y proporciona a las capas superiores un modelo de procesos secuenciales independientes que se comunican mediante mensajes. El código de esta capa tiene dos funciones principales. La primera es capturar los traps y las interrupciones, salvar y restaurar los registros, planificar, y hacer todo lo demás que se necesita para conseguir que funcione realmente la abstracción de los procesos que se proporciona a las capas superiores. La segunda es manejar la mecánica de los mensajes; comprobando la legalidad de sus destinatarios, localizando los búferes de las operaciones de enviar y recibir en la memoria, y copiando los bytes desde el emisor al receptor. Esa parte de la capa que trata el nivel inferior del manejo de las interrupciones está escrita en lenguaje ensamblador. El resto de la capa y todas las capas superiores, están escritas en C.

La capa 2 contiene los procesos de E/S, uno por tipo de dispositivo. Para distinguirlos de los procesos de usuario ordinarios, vamos a llamarlos **tareas**, pero las diferencias entre las tareas y los procesos son mínimas. En numerosos sistemas las tareas de E/S se denominan **drivers de dispositivos**; vamos a utilizar los términos “tarea” y “driver de dispositivo” de forma intercambiable. Es necesaria una tarea por cada tipo de dispositivo, incluyendo discos, impresoras, terminales, interfaces de red y relojes. Si hay otros dispositivos presentes, es necesaria una nueva tarea para cada uno de ellos. Una tarea, la tarea del sistema, es un poco diferente, ya que no corresponde a ningún dispositivo de E/S. Vamos a analizar las tareas en el siguiente capítulo.

Todas las tareas de la capa 2 y todo el código en la capa 1 están enlazados juntos en un único programa binario denominado el **núcleo**. Algunas de las tareas comparten subrutinas comunes, pero de otra manera son independientes unas de otras, se planifican independientemente, y se comunican utilizando mensajes. Los procesadores de Intel comenzando con el 286 asignan uno de sus cuatro niveles de privilegio a cada proceso. Aunque las tareas y el núcleo están compiladas juntas, cuando el núcleo y las rutinas de tratamiento de las interrupciones están ejecutándose tienen más privilegios que las tareas. Entonces el verdadero código del núcleo puede acceder a cualquier parte de la memoria y a cualquier registro del procesador – esencialmente, el núcleo puede ejecutar cualquier instrucción utilizando datos de cualquier sitio dentro del sistema. Las tareas no pueden ejecutar todas las instrucciones a nivel de máquina, ni pueden acceder a todos los registros de la CPU o a todas las partes de la memoria. Sin embargo, ellas pueden acceder a regiones de la memoria pertenecientes a procesos menos privilegiados, en orden a realizar E/S para ellos. Una tarea, la tarea del sistema, no realiza E/S en el sentido normal pero existe en orden a proporcionar servicios, tales como la copia de datos entre diferentes regiones de la memoria, para procesos a los que no se les permite realizar esas cosas por si mismos. En máquinas que no proporcionan diferentes niveles de privilegio, tales como los procesadores más antiguos de Intel, estas restricciones no pueden ser reforzadas por el hardware.

La capa 3 contiene procesos que proporcionan servicios útiles a los procesos de usuario. Estos procesos servidores se ejecutan en un nivel menos privilegiado que el núcleo y las tareas y no pueden acceder a los puertos de E/S directamente. Estos procesos tampoco pueden acceder a la memoria fuera de los segmentos que se les han asignado. El **gestor de memoria** (MM) lleva a cabo todas las llamadas al sistema de MINIX que involucran gestión de memoria, tales como FORK, EXEC y BRK. El **sistema de ficheros** (FS) lleva a cabo todas las llamadas al sistema de ficheros, tales como READ, MOUNT y CHDIR.

Como hemos señalado al comienzo del Capítulo 1, los sistemas operativos hacen dos cosas: gestionan los recursos y proporcionan una máquina extendida mediante la implementación de las llamadas al sistema. En MINIX la gestión de los recursos está situada principalmente en el núcleo (capas 1 y 2), y la interpretación de las llamadas al sistema está en la capa 3. El sistema de ficheros se ha diseñado como un “servidor” de ficheros y puede moverse a una máquina remota casi sin cambios. Esto también es válido para el gestor de memoria, aunque los servidores de memoria remotos no son tan útiles como los servidores de ficheros remotos.

En la capa 3 pueden existir servidores adicionales. Aunque MINIX tal y como se describe en este libro no incluye el servidor de red, su código fuente es parte de la distribución estándar de MINIX. El sistema puede recompilarse fácilmente para incluirlo.

Este es un buen lugar para señalar que aunque los servidores son procesos independientes, difieren de los procesos de usuario en que arrancan cuando el sistema arranca, y nunca terminan mientras el sistema esté activo. Adicionalmente, aunque ellos se ejecutan en el mismo nivel de privilegio que los procesos de usuario en términos de las instrucciones máquina que se les permite ejecutar, reciben una prioridad de ejecución más alta que los procesos de usuario. Para acomodar un nuevo servidor es necesario recompilar el núcleo. El código de arranque del núcleo instala los servidores en entradas privilegiadas de la tabla de procesos antes de que se haya permitido la ejecución de ningún proceso de usuario.

Finalmente, la capa 4 contiene todos los procesos de usuario – shells, editores, compiladores y programas *a.out* escritos por el usuario. Un sistema en ejecución tiene usualmente algunos procesos que comienzan su ejecución cuando el sistema arranca y que nunca terminan de ejecutarse. Por ejemplo, un **demonio** es un proceso de fondo que se ejecuta periódicamente o que siempre espera a que tenga lugar un suceso, tal como la llegada de un paquete por la red. En cierto sentido un demonio es un servidor que se arranca independientemente y se ejecuta como un proceso de usuario. Sin embargo, de forma diferente a los verdaderos servidores instalados en entradas privilegiadas de la tabla de procesos, tales programas no pueden recibir un tratamiento especial por parte del núcleo como el que reciben los procesos servidores de memoria y de ficheros.

2

PROCESOS Y THREADS

Vamos a embarcarnos ahora en un estudio detallado de cómo están diseñados y construidos los sistemas operativos. El concepto fundamental en cualquier sistema operativo es el concepto de *proceso* que consiste en una abstracción de lo que es un programa en ejecución. Todo lo demás depende de este concepto y es importante que el diseñador del sistema operativo (y el estudiante) comprenda lo antes posible lo que es un proceso.

2.1 PROCESOS

Todos los ordenadores modernos pueden hacer varias cosas a la vez. Mientras un ordenador está ejecutando un programa de usuario puede perfectamente estar leyendo de un disco e imprimiendo texto en una pantalla o una impresora. En un sistema multiprogramado la CPU también conmuta de unos programas a otros, ejecutando cada uno de ellos durante decenas o cientos de milisegundos. Aunque, estrictamente hablando, en cualquier instante de tiempo la CPU sólo está ejecutando un programa, en el transcurso de 1 segundo ha podido estar trabajando sobre varios programas, dando entonces a los usuarios la impresión de un cierto paralelismo. En este contexto a veces la gente habla de **pseudoparalelismo**, en contraste con el auténtico paralelismo del hardware de los sistemas **multiprocesador** (que tienen dos o más CPUs compartiendo la misma memoria física). Seguir la pista de múltiples actividades paralelas resulta muy complicado para las personas. Por ese motivo los diseñadores del sistema operativo han desarrollado a través de los años un modelo conceptual evolucionado (el de los procesos secuenciales) que permite tratar el paralelismo de una forma más fácil. Este modelo, sus usos, y algunas de sus consecuencias constituyen el tema de este capítulo.

2.1.1 El Modelo de los Procesos Secuenciales

En este modelo, todo el software ejecutable en el ordenador, incluyendo a veces al propio sistema operativo, se organiza en un número de **procesos secuenciales**, o simplemente **procesos** para acortar. Un proceso es justamente un programa en ejecución, incluyendo los valores actuales del contador de programa, registros y variables. Conceptualmente cada proceso tiene su propia CPU virtual. En realidad, por supuesto, la CPU real conmuta sucesivamente de un proceso a otro, pero para entender el sistema, es mucho más fácil pensar sobre una colección de procesos ejecutándose en (pseudo) paralelo, que intentar seguir la pista de cómo la CPU conmuta de un programa a otro. Esta rápida conmutación de un proceso a otro en algún orden se denomina **multiprogramación** como vimos en el capítulo 1.

En la Figura 2-1(a) vemos un ordenador multiprogramado con cuatro programas en memoria. En la Figura 2-1(b) vemos cuatro procesos cada uno con su propio flujo de control (es decir su propio contador de programa lógico), y cada uno ejecutándose independientemente de los otros. Por supuesto que existe un único contador de programa físico, por lo que cada vez que un proceso retoma su ejecución, su contador de programa lógico debe cargarse en el contador de programa real. Cuando el proceso agota el intervalo de tiempo que se le ha concedido, se salva

su contador de programa físico en su contador de programa lógico en memoria. En la Figura 2-1(c) vemos que desde la perspectiva de un intervalo de tiempo suficientemente largo, todos los procesos han progresado, pero que en cualquier instante dado solamente un único proceso está realmente ejecutándose.

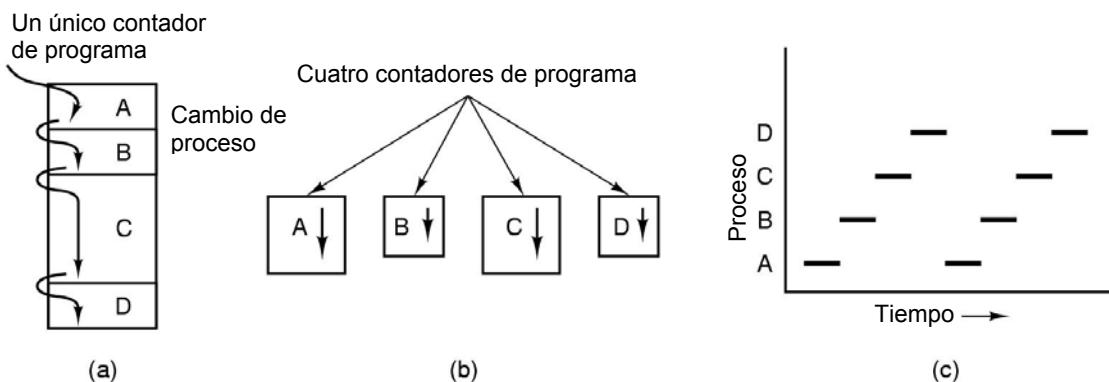


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en cada momento.

Con la CPU comutando de un proceso a otro, la velocidad a la cual un proceso realiza su computación no es uniforme y probablemente ni siquiera es reproducible si los mismos procesos se ejecutan de nuevo. Por ese motivo los procesos no deben programarse bajo suposiciones preconcebidas sobre su velocidad de ejecución. Consideremos, por ejemplo, un proceso de E/S que restaura los ficheros de un backup en cinta. El proceso comienza poniendo en marcha la cinta, ejecuta 10.000 veces un bucle para esperar a que la cinta adquiera la velocidad adecuada, y en ese preciso momento envía un comando para leer el primer registro de la cinta. Si la CPU decide comutar a otro proceso durante el bucle de retardo, el proceso de la cinta puede no volver a ejecutarse antes de que el primer registro sobrepase la cabeza de lectura. Cuando un proceso tiene requerimientos de tiempo real críticos como el anterior, esto es, que ciertos sucesos particulares deben ocurrir dentro de un número de milisegundos especificado, entonces es necesario tomar medidas especiales para asegurar que efectivamente esos sucesos ocurran dentro de esos límites de tiempo. Sin embargo, normalmente la mayoría de los procesos no se ven afectados por la multiprogramación subyacente de la CPU o por las velocidades relativas de los diferentes procesos.

La diferencia entre un proceso y un programa es sutil, pero crucial. Para explicar esto puede servirnos de ayuda una analogía. Consideremos un científico informático con aptitudes culinarias que está preparando una tarta de cumpleaños para su hija. Para ello dispone de una receta de la tarta de cumpleaños y una cocina bien surtida con todos los ingredientes: harina, huevos, azúcar, extracto de vainilla, etc. En esta analogía, la receta representa el programa (es decir un algoritmo expresado mediante alguna notación apropiada), el científico informático representa el procesador (CPU), y los ingredientes de la tarta representan los datos de entrada. El proceso es la actividad consistente en nuestro pastelero leyendo la receta, añadiendo los ingredientes y preparando la tarta.

Imaginemos ahora que el hijo del científico informático entra corriendo y gritando, diciendo que le ha picado una abeja. El científico informático apunta por donde iba en la receta (salva el estado del proceso actual), coge un libro de primeros auxilios y comienza a seguir las instrucciones para la cura. Aquí vemos cómo el procesador comuta de un proceso (preparar la tarta) a un proceso de mayor prioridad (administrar cuidados médicos), cada uno de los cuales sigue un programa diferente (la receta frente al libro de primeros auxilios). Una vez que termina de curar la picadura de la abeja, el científico vuelve a su tarta, continuando en el punto donde la dejó.

La idea clave aquí es que un proceso es una actividad de algún tipo. Tiene un programa, entrada, salida y un estado. Un único procesador puede compartirse entre varios procesos utilizando un algoritmo de planificación que determine cuándo hay que detener el trabajo sobre un proceso y pasar a atender a otro diferente.

2.1.2 Creación de Procesos

Los sistemas operativos necesitan asegurar de alguna forma que puedan existir todos los procesos necesarios. En sistemas muy sencillos, o en sistemas diseñados para ejecutar tan solo una única aplicación (por ejemplo el controlador de un microondas), puede conseguirse que cuando el sistema termine de arrancar estén presentes ya todos los procesos que puedan necesitarse en el futuro. Sin embargo, en sistemas de propósito general es necesaria alguna manera de poder crear y destruir los procesos según sea necesario durante la operación del sistema. Vamos a fijarnos ahora en algunas de estas cuestiones.

Los cuatro principales sucesos que provocan la creación de nuevos procesos son:

1. La inicialización del sistema
2. La ejecución por parte de un proceso (en ejecución) de una llamada al sistema de creación de un nuevo proceso.
3. La petición por parte del usuario de la creación de un nuevo proceso.
4. El inicio de un trabajo en batch.

Cuando un sistema operativo arranca, se crean típicamente varios procesos. Algunos de esos procesos son procesos de superficie (o en primer plano), esto es, procesos que interactúan con los usuarios (humanos) y realizan trabajo para ellos. Otros son procesos de fondo (o en segundo plano), que no están asociados con usuarios particulares, sino que tienen alguna función específica. Por ejemplo, un proceso de fondo puede diseñarse para que se encargue de aceptar el correo electrónico entrante, de manera que esté durmiendo la mayor parte del día pero vuelva repentinamente a la vida tan pronto como llegue algún correo. Otro proceso de fondo puede diseñarse para aceptar peticiones entrantes de páginas web residentes en esa máquina, despertándose cada vez que llegue una nueva petición para servir una cierta página. Los procesos que se ejecutan como procesos de fondo para llevar a cabo alguna actividad tal como el correo electrónico, las páginas web, las news o la impresión de ficheros de salida, etc, se denominan **demonios**. Los sistemas grandes tienen comúnmente docenas de ellos. En UNIX, el programa *ps* puede utilizarse para listar los procesos que están en marcha. En Windows 95/98/Me tecleando CTRL-ALT-SUPR una vez, se muestra todo lo que está en marcha. En Windows 2000 se utiliza el administrador de tareas.

Adicionalmente a los procesos creados en el momento del arranque, también pueden crearse nuevos procesos después. A menudo un proceso en ejecución puede hacer llamadas al sistema para crear uno o más procesos nuevos para que le ayuden en su trabajo. Crear nuevos procesos es particularmente útil cuando el trabajo a realizar puede formularse fácilmente en términos de varios procesos relacionados, pero por otra parte independientes, que interactúan entre sí. Por ejemplo, si se está extrayendo una gran cantidad de datos a través de una red de comunicación para su procesamiento subsiguiente, puede ser conveniente crear un proceso para extraer los datos y ponerlos en un buffer compartido mientras que un segundo proceso va retirando los datos del buffer y los procesa. En el caso de un sistema multiprocesador podemos conseguir que todo el trabajo se haga efectivamente más rápido si permitimos que esos dos procesos se ejecuten cada uno en una CPU diferente.

En sistemas interactivos los usuarios pueden arrancar un programa tecleando un comando o pinchando (dos veces) con el ratón sobre un ícono. Realizando cualquiera de esas acciones conseguimos que comience un nuevo proceso y se ejecute el programa correspondiente. En sistemas UNIX basados en comandos y ejecutando X Windows, el nuevo proceso creado se ejecuta sobre la ventana en la cual se le activó. En Microsoft Windows, cuando un proceso comienza no tiene ninguna ventana asignada, aunque puede crear una (o más), y la mayoría de los programas efectivamente eso es lo que hacen. En ambos sistemas, los usuarios pueden tener múltiples ventanas abiertas a la vez, cada una de ellas ejecutando algún proceso. Utilizando el ratón, el usuario puede seleccionar una ventana e interactuar con el proceso, por ejemplo, proporcionando datos de entrada cuando sean necesarios.

La última situación que provoca la creación de procesos se aplica sólo a los sistemas en batch que podemos encontrar en los grandes mainframes. En esos sistemas los usuarios pueden lanzar (*submit*) al sistema trabajos en batch (posiblemente de forma remota). Cuando el sistema operativo detecta que dispone de todos los recursos necesarios para poder ejecutar otro trabajo, crea un nuevo proceso y ejecuta sobre él el siguiente trabajo que haya en la cola de entrada.

Técnicamente, en todos los casos, se crea un nuevo proceso haciendo que un proceso ya existente ejecute una llamada al sistema de creación de un nuevo proceso. El proceso que hace la llamada puede ser un proceso de usuario, un proceso del sistema invocado desde el teclado o el ratón, o un proceso gestor de los trabajos en batch. Lo que ese proceso hace es ejecutar una llamada al sistema para crear el nuevo proceso. Esa llamada al sistema solicita al sistema operativo que cree un nuevo proceso, indicándole directa o indirectamente, qué programa debe ejecutar sobre él.

En UNIX sólo existe una llamada al sistema para crear un nuevo proceso: **fork**. Esta llamada crea un clon (una copia exacta) del proceso que hizo la llamada. Después del **fork**, los dos procesos, el padre y el hijo, tienen la misma imagen de memoria, las mismas variables de entorno y los mismos ficheros abiertos. Eso es todo lo que hay. Usualmente, a continuación el proceso hijo ejecuta **execve** o una llamada al sistema similar para cambiar su imagen de memoria y pasar a ejecutar un nuevo programa. Por ejemplo cuando un usuario teclea un comando del shell como por ejemplo, *sort*, el shell ejecuta un **fork** para crear un proceso hijo, el cual es el que realmente ejecuta el programa correspondiente al *sort*. La razón de realizar estos dos pasos es permitir al hijo que manipule los descriptores de fichero del shell después del **fork** pero antes de que el **execve** lleve a cabo la redirección de la entrada estándar, la salida estándar y la salida de errores estándar.

Lo anterior contrasta con lo que sucede en Windows, donde mediante una única llamada al sistema de Win32, **CreateProcess**, se realiza tanto la creación del proceso como la carga del programa correcto dentro del nuevo proceso. Esta llamada tiene 10 parámetros que incluyen entre ellos el programa que hay que ejecutar, los parámetros de la línea de comandos que va a recibir el programa, varios atributos de seguridad, bits que controlan si se heredan los ficheros abiertos, información sobre la prioridad del proceso, una especificación de la ventana que hay que crear (en su caso) para el proceso, y un puntero a una estructura (un registro) en la que se envíe de retorno toda la información sobre el nuevo proceso creado, al proceso que hace la llamada. Adicionalmente a **CreateProcess**, Win32 cuenta con unas 100 llamadas al sistema más, para gestionar y sincronizar los procesos, así como para operaciones relacionadas.

Tanto en UNIX como en Windows, después de crear un proceso, tanto el padre como el hijo cuentan con sus propios espacios de direcciones disjuntos. Si cualquiera de los procesos modifica una palabra en su espacio de direcciones, ese cambio es invisible para cualquier otro proceso. En UNIX, el espacio de direcciones inicial del hijo es una *copia* del espacio de direcciones del padre, pero hay dos espacios de direcciones distintos involucrados; la memoria no escribible se comparte (algunas implementaciones de UNIX comparten el área de código entre los dos, ya que el código nunca se modifica). Sin embargo es posible que un nuevo

proceso creado comparta algunos de los demás recursos del padre, tales como los ficheros abiertos. En Windows, los espacios de direccionamiento del padre y el hijo son diferentes desde el primer momento.

2.1.3 Terminación de los Procesos

Tras la creación de un proceso comienza su ejecución realizando el trabajo que se le ha encomendado. Sin embargo nada dura para siempre, ni siquiera los procesos. Pronto o tarde el nuevo proceso debe terminar, usualmente debido a una de las siguientes causas:

1. El proceso completa su trabajo y termina (voluntariamente).
2. El proceso detecta un error y termina (voluntariamente).
3. El sistema detecta un error fatal del proceso y fuerza su terminación.
4. Otro proceso fuerza la terminación del proceso (por ejemplo en UNIX mediante la llamada al sistema `kill`).

La mayoría de los procesos terminan debido a que han completado su trabajo. Cuando un compilador ha compilado el programa que se le ha dado, el compilador ejecuta una llamada al sistema para decirle al sistema operativo que ha finalizado. Esta llamada es `exit` en UNIX y `ExitProcess` en Windows. Los programas orientados a la pantalla soportan también la terminación voluntaria. Los procesadores de texto, navegadores y programas similares cuentan siempre con un ícono o una opción de menú para que el usuario pueda pinchar con el ratón indicándole al proceso que borre cualquier fichero temporal que esté abierto y a continuación termine.

La segunda causa de terminación es que el proceso descubra un error fatal. Por ejemplo, si un usuario teclea el comando

```
cc foo.c
```

para compilar el programa *foo.c* sin que exista tal fichero, el compilador simplemente termina. Generalmente los procesos interactivos orientados a la pantalla no dan por concluida su ejecución cuando reciben parámetros erróneos. En vez de eso despliegan una ventana de diálogo emergente solicitando al usuario que intente introducir de nuevo los parámetros correctos.

La tercera causa de terminación es la aparición de un error causado por el proceso, a menudo debido a un error de programación. Algunos ejemplos son: la ejecución de una instrucción ilegal, una referencia a una posición de memoria inexistente, o una división por cero. En algunos sistemas (por ejemplo en UNIX), un proceso puede indicar al sistema operativo que quiere tratar por sí mismo ciertos errores, en cuyo caso el proceso recibe una señal que lo interrumpe, en vez de terminar bruscamente al ocurrir uno de tales errores previstos.

La cuarta razón por la cual un proceso puede terminar, es que un proceso ejecute una llamada al sistema diciéndole al sistema operativo que mate a algún otro proceso. En UNIX esta llamada es `kill` (matar). La función Win32 correspondiente es `TerminateProcess`. En ambos casos el proceso asesino debe contar con la debida autorización. En algunos sistemas, cuando un proceso termina, bien sea voluntariamente o no, el sistema mata automáticamente también a todos los procesos que pudiera haber creado el proceso. Sin embargo, ni UNIX ni Windows funcionan de esa manera.

2.1.4. Jerarquías de Procesos

En algunos sistemas, cuando un proceso crea otro proceso, el proceso padre y el proceso hijo, continúan estando asociados de cierta manera. El proceso hijo puede a su vez crear más procesos formando una jerarquía de procesos. De forma diferente a las plantas y animales que se reproducen de forma sexual, un proceso tiene un único parente (pero cero, uno, dos o más hijos).

En UNIX, un proceso y todos sus hijos y demás descendientes forman juntos un grupo de procesos. Cuando un usuario envía una señal desde el teclado (como por ejemplo tecleando Ctrl-C), la señal se propaga a todos los miembros del grupo de procesos actualmente asociados con el teclado (normalmente todos los procesos activos que fueron creados en la ventana actual). Individualmente, cada proceso puede capturar la señal, ignorar la señal o emprender la acción por defecto, que es la de ser matado por la señal recibida.

Otro ejemplo del papel que puede jugar la jerarquía de procesos es cómo se inicializa UNIX durante su arranque. Hay un proceso especial presente en la imagen de arranque denominado *init*. Cuando este proceso comienza su ejecución lee un fichero donde figura el número de terminales con que cuenta el sistema. Acto seguido *init* crea mediante la llamada al sistema *fork* un nuevo proceso por terminal. Estos procesos esperan a que alguien se conecte al sistema a través del correspondiente terminal. Cada vez que un usuario logra conectarse, el proceso asociado al terminal ejecuta un shell para aceptar comandos. A su vez estos comandos pueden dar lugar a la creación de más procesos. En definitiva, todos los procesos en el sistema pertenecen a un único árbol que tiene al proceso *init* como raíz.

Por el contrario, Windows no ofrece ningún concepto de jerarquía de procesos. Todos los procesos son iguales. El único lugar donde hay algo parecido a una jerarquía de procesos es que cuando se crea un proceso, su proceso parente recibe un puntero a un conjunto de información (lo que se denomina un **handle**) que puede utilizar para controlar al proceso hijo. Sin embargo, el parente es libre de pasar o no esa información a algún otro proceso, lo que significa que no está asegurado por el sistema el mantenimiento de la jerarquía de los procesos creados. Los procesos en UNIX no pueden desentenderse de sus hijos.

2.1.5 Estados de los Procesos

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, los procesos necesitan a menudo interactuar con otros procesos. Un proceso puede generar los datos de salida que otro proceso utiliza como entrada. En el comando del shell

```
cat capitulo1 capitulo2 capitulo3 | grep arbol
```

el primer proceso, que ejecuta *cat*, produce como salida la concatenación de los tres ficheros. El segundo proceso, que ejecuta *grep*, selecciona todas las líneas que contienen la palabra “arbol”. Dependiendo de la velocidad relativa de los dos procesos (que depende tanto de la complejidad relativa de los programas como del tiempo de CPU del que haya dispuesto cada proceso), puede ocurrir que *grep* esté listo para ejecutarse, pero que no haya ninguna entrada esperando a ser procesada por él. En ese caso el proceso que ejecuta el *grep* debe bloquearse hasta que esté disponible alguna entrada.

Cuando un proceso se bloquea, lo hace porque desde un punto de vista lógico no puede continuar, normalmente debido a que está esperando por datos de entrada que aún no están disponibles. También es posible para un proceso que esté conceptualmente preparado y sea capaz de ejecutarse, que esté parado debido a que el sistema operativo ha decidido temporalmente asignar la CPU a otro proceso. Estas dos situaciones son completamente diferentes. En el primer caso, la suspensión es inherente a la situación (no puede procesarse el

comando del usuario hasta que no halla terminado de teclearse). En el segundo caso, se trata simplemente de una cuestión técnica del sistema (no hay suficientes CPUs para dar a cada proceso su propio procesador privado). En la Figura 2-2 podemos ver un diagrama de estados mostrando los tres estados en los que puede estar un proceso:

1. En ejecución (utilizando realmente la CPU en ese instante).
2. Preparado (ejecutable; detenido temporalmente para permitir que otro proceso se ejecute).
3. Bloqueado (incapaz de ejecutarse hasta que tenga lugar algún suceso externo).

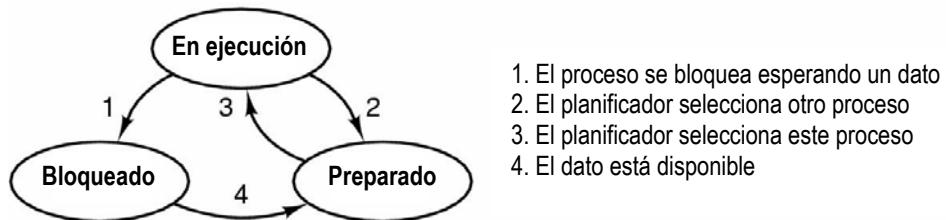


Figura 2-2. Un proceso puede estar en estado de ejecución, bloqueado o preparado. Se muestran las transiciones entre esos estados.

Desde un punto de vista lógico, los dos primeros estados son similares. En ambos casos el proceso está dispuesto a ejecutarse, sólo que en el segundo caso temporalmente no existe ninguna CPU disponible para él. El tercer estado es diferente de los dos primeros ya que el proceso no puede ejecutarse, ni siquiera en el caso de que la CPU no tuviera ninguna otra cosa que hacer.

Como puede verse son posibles cuatro transiciones entre esos tres estados. La transición 1 tiene lugar cuando un proceso descubre que no puede continuar. En algunos sistemas el proceso debe ejecutar una llamada al sistema, tal como `block` o `pause`, para pasar al estado bloqueado. En otros sistemas, incluyendo UNIX, cuando un proceso lee de una tubería (*pipe*) o de un fichero especial (como por ejemplo `/dev/tty0` correspondiente al primer terminal) y no existe ninguna entrada disponible, el proceso se bloquea automáticamente.

Las transiciones 2 y 3 se deben a la actuación del planificador de procesos (una parte del sistema operativo) sin que el proceso tenga conocimiento de ellas. La transición 2 tiene lugar cuando el planificador (*scheduler*) decide que el proceso en ejecución se ha ejecutado durante un tiempo suficientemente largo, y es hora de dejar a otro proceso que reciba algún tiempo de CPU. La transición 3 tiene lugar cuando todos los demás procesos han recibido ya su justa parte del tiempo de CPU, siendo hora ya de que el primer proceso consiga la CPU para ejecutarse de nuevo. El tema de la planificación, esto es de decidir qué proceso debe ejecutarse, cuándo y por cuánto tiempo, es un tema muy importante que trataremos posteriormente en este capítulo. Se han diseñado muchos algoritmos para tratar de equilibrar los objetivos contrapuestos de eficiencia en el funcionamiento del sistema en su conjunto y de justicia en el trato a los procesos individuales. Estudiaremos algunos de ellos posteriormente en este capítulo.

La transición 4 tiene lugar cuando por fin se produce el suceso externo por el cual estaba esperando un proceso (tal como la llegada de algún nuevo dato de entrada). Si no está ejecutándose ningún otro proceso en ese instante, tiene lugar la transición 3 y el proceso prosigue con su ejecución donde había quedado bloqueado. En otro caso el proceso tiene que esperar en el estado *preparado* durante algún tiempo hasta que esté disponible la CPU y le toque el turno.

Utilizando el modelo de los procesos, es mucho más fácil pensar sobre lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas correspondientes a comandos tecleados por un usuario. Otros procesos son parte del sistema y desarrollan tareas tales como procesar peticiones de servicio de ficheros o gestionar los detalles del manejo de un disco o una unidad de cinta. Cuando llega una interrupción procedente del disco, el sistema toma la decisión de detener la ejecución del proceso actual y ejecutar el proceso asociado al disco, que estaba anteriormente bloqueado esperando a que llegara esa interrupción. Así, en vez de pensar en términos de interrupciones, podemos pensar en términos de procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando tienen que esperar a que ocurra algo. Cuando el disco ha terminado de leerse, o el carácter por fin se teclea, el proceso que esperaba ese suceso se desbloquea y pasa a ser elegible para ejecutarse de nuevo.

Esta perspectiva da lugar al modelo mostrado en la Figura 2-3. En ella el nivel inferior del sistema operativo es el planificador, con una variedad de procesos por encima de él. Todo el manejo de las interrupciones y los detalles de cómo arrancar y detener los procesos quedan ocultos bajo lo que hemos denominado aquí el planificador de procesos, el cual no representa realmente demasiado código. El resto del sistema operativo está bonitamente estructurado en forma de múltiples procesos. Sin embargo son pocos los sistemas operativos reales que están tan bonitamente estructurados como este.

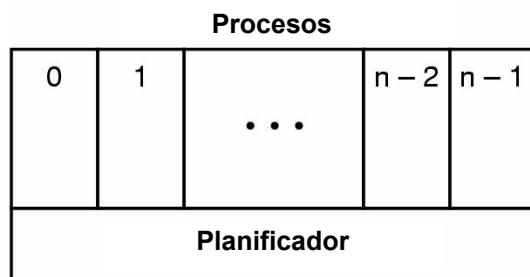


Figura 2-3. La capa inferior de un sistema estructurado en procesos maneja las interrupciones y la planificación de los procesos. Por encima de esa capa están los procesos secuenciales.

2.1.6 Implementación de Procesos

Para implementar el modelo de los procesos el sistema operativo mantiene una tabla (un array de registros o estructuras), denominada la **tabla de procesos**, con una entrada por proceso. Algunos autores denominan a cada una de esas entradas **descriptor de proceso** o **bloque de control de proceso**. Estas entradas contienen información sobre el estado de cada proceso, su contador de programa, su puntero de pila, su asignación de memoria, el estado de sus ficheros abiertos, la información relativa a su planificación y a la contabilidad de los recursos que ha consumido, así como cualquier otra información sobre el proceso que deba guardarse cuando el proceso commute del estado de *en ejecución* al estado de *preparado* o *bloqueado*, de forma que su ejecución pueda retomarse posteriormente como si nunca se hubiera detenido.

La Figura 2-4 muestra algunos de los campos más importantes que aparecen en el descriptor de proceso de cualquier sistema operativo típico. Los campos en la primera columna están relacionados con la gestión de los procesos. Las otras dos columnas tienen que ver con la gestión de memoria y la gestión de ficheros, respectivamente. Hay que señalar que los campos concretos que tienen los descriptores de la tabla de procesos varían mucho de un sistema operativo a otro, pero la figura da una idea general del tipo de información que es necesario mantener para la gestión de los procesos.

| Gestión de procesos | Gestión de memoria | Gestión de ficheros | descriptor de proceso |
|---|--|--|-----------------------|
| Registros Contador de programa (PC) Registro de estado (SR o PSW) Puntero de pila (SP) Estado del proceso Prioridad Parámetros de planificación Identificador de proceso (pid) Proceso padre Grupo del proceso Señales Instante de comienzo Tiempo de CPU utilizado Tiempo de CPU de los hijos Tiempo restante para la siguiente alarma | Puntero al segmento de código Puntero al segmento de datos Puntero al segmento de pila | Directorio raíz Directorio de trabajo Descriptores de ficheros Identificador de usuario (uid) Identificador de grupo (gid) | |

Figura 2-4. Algunos de los campos de una entrada típica de la tabla de procesos.

Una vez presentada la tabla de procesos, es posible precisar un poco más cómo es posible ofrecer la ilusión de la existencia de múltiples procesos secuenciales que desarrollan su actividad concurrentemente, sobre una máquina con una única CPU y muchos dispositivos de E/S. Cada clase de dispositivos de E/S (como por ejemplo las disqueteras, los discos duros, los timers o los terminales) tiene asociada una posición de memoria (a menudo situada en las posiciones más bajas de la memoria) que se denomina el **vector de interrupción** utilizado por ese tipo de dispositivos. Esta posición de memoria contiene la dirección de la **rutina de tratamiento de la interrupción** que atiende a ese tipo de dispositivos. Supongamos que el proceso de usuario número 3 se está ejecutando cuando de repente la CPU recibe una interrupción del disco duro. En ese momento el hardware de las interrupciones apila (en la pila actual) el contador de programa del proceso de usuario número 3, la palabra de estado del programa (es decir su registro de estado) y posiblemente algunos otros registros hardware mas. A continuación la CPU salta a la dirección especificada en el vector de interrupción del disco. Eso es todo lo que hace el hardware. Desde ese momento toma el control el software, más concretamente la rutina de tratamiento de la interrupción.

Todas las interrupciones comienzan salvando los registros (a menudo en el descriptor del proceso actualmente en ejecución). A continuación se desapila la información apilada anteriormente por el hardware (en el momento de la interrupción), estableciéndose el puntero de pila para que apunte a una pila temporal utilizada por el proceso controlador (en este caso del disco). Las acciones anteriores, tales como salvar los registros y establecer el puntero de pila, no pueden expresarse en lenguajes de alto nivel como C, de manera que las realiza una pequeña rutina en lenguaje ensamblador, usualmente la misma para todas las interrupciones ya que el trabajo de salvar los registros es siempre el mismo, sin importar cuál sea la causa de la interrupción.

Cuando esta rutina termina, realiza una llamada a un procedimiento en C para llevar a cabo el resto del trabajo para este tipo de interrupción específico. Estamos suponiendo que el sistema operativo está escrito en C (que es la elección usual para todos los sistemas operativos reales). Cuando ese procedimiento termina su trabajo, posiblemente provocando que algún proceso pase al estado de preparado, se llama al planificador para determinar qué proceso preparado se ejecuta a continuación. Después de eso, se devuelve el control al código en lenguaje ensamblador para cargar los registros y el mapa de memoria del nuevo proceso que ha

seleccionado el planificador y se retoma su ejecución. En la Figura 2-5 se resume el tratamiento de las interrupciones y la planificación. Es necesario insistir en que los detalles varían considerablemente de un sistema operativo a otro.

1. El hardware apila el contador de programa, etc.
2. El hardware carga el nuevo contador de programa desde el vector de interrupción.
3. Una rutina de lenguaje ensamblador salva los registros.
4. Una rutina de lenguaje ensamblador establece una nueva pila.
5. Se ejecuta la rutina de tratamiento de la interrupción escrita en C (normalmente lee y guarda en un búfer el dato de entrada).
6. El **planificador** decide qué procedimiento ejecutar a continuación.
7. Un procedimiento escrito en C retorna al código en ensamblador.
8. Una rutina de lenguaje ensamblador (el **dispatcher**) pasa a ejecución el proceso seleccionado por el planificador.

2.2 THREADS

En los sistemas operativos tradicionales, cada proceso tiene su propio espacio de direcciones y un único flujo (hilo) de control. De hecho, casi es esa la definición de proceso. Sin embargo, frecuentemente hay situaciones en las que es deseable contar con múltiples hilos de control (threads) en el mismo espacio de direcciones ejecutándose quasi-paralelamente, como si fueran procesos separados (excepto que comparten el mismo espacio de direcciones). En las secciones siguientes vamos a discutir esas situaciones y sus implicaciones.

2.2.1 El Modelo de los Threads

Como hemos expuesto largamente, el modelo de los procesos se basa en dos conceptos independientes: el agrupamiento de los recursos y la ejecución secuencial de un programa. A veces es útil separar esos dos conceptos, y es aquí donde entran en juego los threads.

Una forma de ver un proceso es que es una manera de agrupar juntos recursos relacionados. Un proceso tiene un espacio de direcciones contenido código y datos del programa, así como otros recursos. Estos recursos pueden incluir ficheros abiertos, procesos hijos, alarmas pendientes, controladores de señales, información de contabilidad, etc. Poniendo juntos todos esos recursos en la forma de un proceso, es posible gestionarlos más fácilmente.

El otro concepto que incluye un proceso es el de hilo de ejecución, usualmente denominado un thread. El thread tiene un contador de programa que indica cuál es la siguiente instrucción a ejecutar. Tiene además registros que contienen sus variables de trabajo actuales. Tiene una pila, que contiene una especie de historia de su ejecución, con una trama por cada procedimiento al que se ha llamado pero del que no se ha retorna todavía. Aunque un thread debe ejecutarse en algún proceso, el thread y su proceso son conceptos diferentes y pueden tratarse de forma separada. Los procesos se utilizan para agrupar recursos juntos; los threads son las entidades planificadas para su ejecución en la CPU.

Lo que los threads añaden al modelo de los procesos es que permiten que haya múltiples ejecuciones en un mismo entorno determinado por un proceso, y esto con un alto grado de independencia de esas ejecuciones. Tener múltiples threads ejecutándose en paralelo dentro de un proceso es análogo a tener múltiples procesos ejecutándose en paralelo dentro de un ordenador. En el primer caso, los threads comparten el espacio de direcciones, los ficheros abiertos y otros recursos. En el segundo caso, los procesos comparten la memoria física, los discos, las impresoras y otros recursos. Debido a que los threads tienen algunas de las propiedades de los procesos, a veces reciben la denominación de **procesos ligeros** (*lightweight process*). También se utiliza el término de **multihilo** (*multithreaded*) para describir la situación en la cual se permite que haya múltiples threads en el mismo proceso.

En la Figura 2-6(a) se representan tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un único thread de control. En contraste, en la Figura 2-6(b) se representa un único proceso con tres threads de control. Aunque en ambos casos tenemos tres threads, en la Figura 2-6(a) cada uno de los threads opera en un espacio de direcciones diferente, mientras que en la Figura 2-6(b) todos los threads comparten el mismo espacio de direcciones.

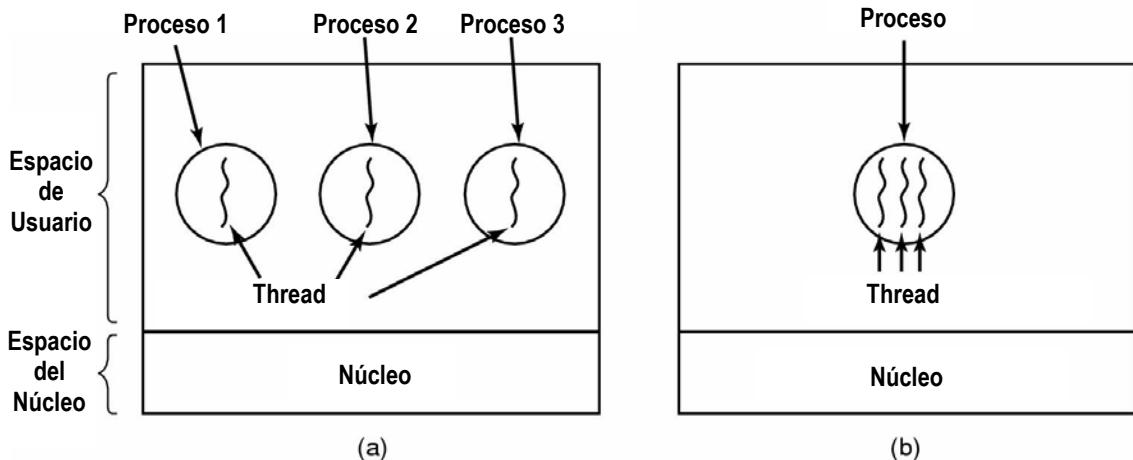


Figura 2-6. (a) Tres procesos cada uno con un thread. (b) Un proceso con tres threads.

Cuando un proceso multihilo se ejecuta sobre un sistema con una única CPU, los threads deberán hacer turnos para ejecutarse. En la Figura 2-1 vimos como funciona la multiprogramación de procesos. El sistema operativo crea la ilusión de que hay varios procesos secuenciales que se ejecutan en paralelo a base de ir conmutando la CPU entre esos procesos. Los sistemas multihilo funcionan de la misma manera. La CPU va conmutándose rápidamente de unos threads a otros proporcionando la ilusión de que los threads se están ejecutando en paralelo, aunque sobre una CPU virtual más lenta que la real. Con tres threads intensivos en computación dentro de un proceso, los threads aparentan estar ejecutándose en paralelo, cada uno sobre una CPU con un tercio de la velocidad de la CPU real.

Los diferentes threads de un proceso no son tan independientes como si fueran diferentes procesos. Todos los threads tienen exactamente el mismo espacio de direcciones, lo que significa en particular que comparten las mismas variables globales. Ya que cualquier thread puede acceder a cualquier dirección de memoria dentro del espacio de direcciones del proceso, un thread puede leer, escribir o incluso borrar completamente la pila de cualquier otro thread. No existe ninguna protección entre los threads debido a que (1) es imposible establecer ninguna medida protección, y (2) es innecesario que haya protección. De forma distinta que con diferentes procesos, que pueden corresponder a diferentes usuarios y que pueden ser hostiles uno con otro, un proceso sólo puede tener un único usuario propietario, quien cuando crea múltiples threads lo hace presumiblemente con la idea de que puedan cooperar, no luchar. Todos los threads comparten, además del espacio de direcciones, el mismo conjunto de ficheros abiertos, procesos hijos, alarmas y señales, etc. como se muestra en la Figura 2-7. Entonces la organización de la Figura 2-6(a) puede utilizarse cuando los tres procesos están esencialmente no relacionados, mientras que Figura 2-6(b) puede ser apropiada cuando los tres threads son realmente parte del mismo trabajo y cooperan activa y estrechamente uno con otro.

Los elementos de la primera columna de la Figura 2-7 corresponden a propiedades de los procesos y no a propiedades de los threads. Por ejemplo si un thread abre un fichero, ese fichero es visible por todos los otros threads del proceso, que pueden ponerse inmediatamente a leer y escribir en él. Esto es lógico ya que el proceso es la unidad de gestión de recursos, y no el thread. Si cada thread tuviera su propio espacio de direcciones, ficheros abiertos, alarmas pendientes, etc. se trataría de un proceso separado. Lo que estamos tratando de conseguir con el concepto de thread es la capacidad de que múltiples hilos de ejecución comparten un conjunto de recursos de manera que puedan trabajar estrechamente juntos para realizar alguna tarea.

| Atributos propios de los procesos | Atributos propios de los threads |
|------------------------------------|----------------------------------|
| Espacio de direcciones | Contador de programa |
| Variables globales | Registros |
| Ficheros abiertos | Pila |
| Procesos hijos | Estado |
| Alarmas pendientes | |
| Señales y controladores de señales | |
| Información de contabilidad | |

Figura 2-7. La primera columna lista algunos elementos compartidos por todos los threads de un proceso. La segunda columna lista algunos elementos privados de cada thread.

Igual que un proceso tradicional (es decir un proceso con un único thread), un thread puede estar en cualquiera de los estados: en ejecución, bloqueado, preparado o terminado. Un thread en ejecución tiene actualmente la CPU y está activo. Un thread bloqueado está esperando a que algún suceso lo desbloquee. Por ejemplo cuando un thread realiza una llamada al sistema para leer del teclado, ese thread se queda bloqueado hasta que se presiona alguna tecla. Un thread puede bloquearse esperando a que tenga lugar algún suceso externo o a que algún otro thread lo desbloquee. Un thread preparado está planificado para ejecutarse y lo hace tan pronto como le llega su turno. Las transiciones entre los estados de un thread son las mismas que las transiciones entre los estados de un proceso y se ilustran en la Figura 2-2.

Es importante que nos demos cuenta de que cada thread tiene su propia pila, como se muestra en la Figura 2-8. La pila de cada thread contiene una trama (o registro de activación) por cada procedimiento al que se ha llamado pero del que todavía no se ha retornado. Esta trama contiene las variables locales del procedimiento y la dirección de retorno a utilizar cuando termine la llamada al procedimiento. Por ejemplo, si el procedimiento *X* llama al procedimiento *Y* y éste a su vez llama al procedimiento *Z*, entonces mientras *Z* se ejecuta la pila contiene todas las tramas de *X*, *Y* y *Z*. Normalmente cada thread llama a diferentes procedimientos y por lo tanto tiene una historia de ejecución diferente. Esa es la razón por la que cada thread necesita su propia pila.

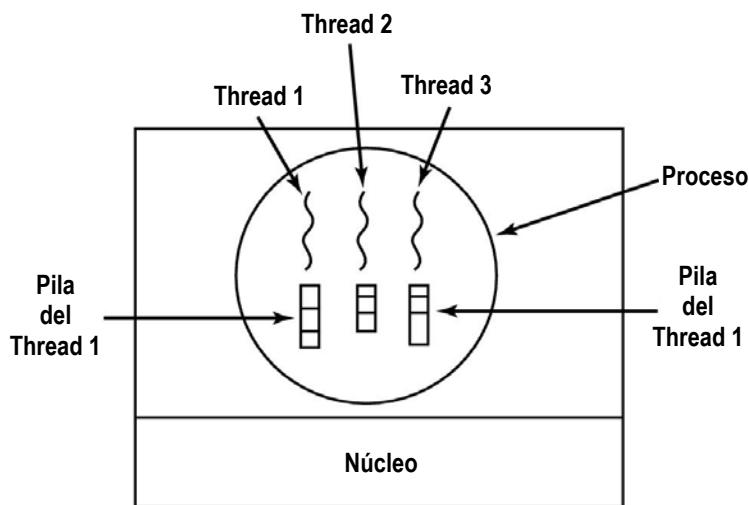


Figura 2-8. Cada thread tiene su propia pila.

Cuando estamos en un sistema multihilo, los procesos normalmente comienzan teniendo un único thread. Este thread tiene la capacidad de crear nuevos threads llamando a un procedimiento de biblioteca, por ejemplo *thread_create*. Típicamente uno de los parámetros de *thread_create* especifica el nombre de un procedimiento que debe ejecutar el nuevo thread. No es necesario (y a veces incluso es imposible) especificar nada más sobre el espacio de direcciones del nuevo thread ya que éste se ejecuta automáticamente en el mismo espacio de direcciones que el thread que lo ha creado. A veces los threads mantienen una relación jerárquica de tipo padre-hijo, pero a menudo no existe tal relación, siendo todos los threads iguales. Con o sin relación jerárquica, siempre se devuelve al thread que hace la llamada de creación un identificador de thread que sirve de nombre para el nuevo thread.

Cuando un thread concluye su trabajo puede dar por terminada su vida llamando a un procedimiento de biblioteca, por ejemplo, *thread_exit*. En ese momento el thread se desvanece dejando ya de ser planificable. En algunos sistemas de threads, un thread puede esperar a que termine otro thread (específico), a través de una llamada al sistema, como por ejemplo, *thread_wait*. Este procedimiento bloquea al proceso que lo invoca hasta que un thread (específico) termina. Desde este punto de vista la creación y terminación de los threads se parece mucho a la creación y terminación de los procesos, teniendo aproximadamente las mismas opciones.

Otra llamada común relacionada con los threads es *thread_yield*, que permite que un thread abandone voluntariamente la CPU para permitir que se ejecute algún otro thread. Tal llamada es importante ya que no existe ninguna interrupción de reloj que dé soporte al tiempo compartido como en el caso de los procesos. Por tanto para los threads es importante ser corteses y ceder voluntariamente la CPU de cuando en cuando para dar la oportunidad de que se ejecuten otros threads. Hay otras llamadas que permiten a un thread esperar a que otro thread termine algún trabajo, o a que un thread anuncie que ha terminado algún trabajo, etc.

Aunque los threads resultan frecuentemente útiles, introducen un cierto número de complicaciones en el modelo de programación. Para comenzar consideremos los efectos de la llamada al sistema *fork* de UNIX. Si el proceso padre tiene un cierto número de threads, ¿debe también tenerlos el proceso hijo? Si no fuera así, el proceso podría no funcionar correctamente, ya que todos los threads del padre podrían ser esenciales en su comportamiento.

Sin embargo, si el proceso hijo se crea con el mismo número de threads que el padre, ¿qué sucede si un thread del padre se bloquea en una llamada a *read*, por ejemplo del teclado? ¿Resulta ahora que los dos threads están ahora bloqueados por el teclado, uno en el padre y otro en el hijo? Cuándo se termina de teclear una línea, ¿obtienen los dos threads una copia de esa línea? ¿Sólo el padre? ¿Sólo el hijo? Este mismo problema se plantea también en relación con las conexiones de red abiertas.

Otra clase de problemas se relaciona con el hecho de que los threads comparten numerosas estructuras de datos. ¿Qué sucede si un thread cierra un fichero mientras otro thread está leyendo todavía? Supongamos que un thread se da cuenta de que queda demasiado poca memoria disponible y empieza a hacer acopio de más memoria. Si en ese momento tiene lugar un cambio de thread en ejecución y el nuevo thread aprecia también que queda demasiado poca memoria, comenzará a su vez a reservar más memoria. En esa situación es muy probable que se asigne el doble de la memoria necesaria. Estos problemas pueden resolverse con algún esfuerzo, pero requieren un análisis y diseño cuidadoso para conseguir que los programas multihilo funcionen correctamente.

2.2.2 Utilización de los Threads

Habiendo descrito qué son los threads, es el momento de exponer la razón de porqué es deseable disponer de ellos en el sistema operativo. La razón principal para tener threads es que son numerosas las aplicaciones en las que hay varias actividades que están en marcha simultáneamente. De vez en cuando, alguna de estas actividades puede bloquearse. En esa situación el modelo de programación resulta más sencillo si descomponemos tal aplicación en varios threads secuenciales que se ejecutan en paralelo.

Hemos visto este argumento antes. Es precisamente el mismo argumento que hicimos para justificar la conveniencia de disponer de procesos. En vez de pensar en términos de interrupciones, timers y cambios de contexto, podemos pensar en términos de procesos paralelos. Sólo que ahora con los threads introducimos un nuevo elemento: la capacidad de las entidades paralelas para compartir entre ellas un espacio de direcciones y todos sus datos. Esta capacidad es esencial para ciertas aplicaciones, y es por lo que el tener simplemente múltiples procesos (con sus espacios de direcciones separados) no puede funcionar en estos casos.

Un segundo argumento para tener threads es que ya que no tienen ningún recurso ligado a ellos, son más fáciles de crear y destruir que los procesos. En numerosos sistemas, la creación de un thread puede realizarse 100 veces más rápido que la creación de un proceso. Cuando el número de threads necesita cambiar dinámicamente y rápidamente, esa propiedad es efectivamente útil.

Una tercera razón para tener threads es también un argumento sobre el rendimiento. Los threads no proporcionan ninguna ganancia en el rendimiento cuando todos ellos utilizan intensamente la CPU, sino cuando hay una necesidad substancial tanto de cálculo en la CPU como de E/S, de manera que teniendo threads se puede conseguir que esas dos actividades se solapen, acelerando la ejecución de la aplicación.

Finalmente, los threads son útiles sobre sistemas con varias CPUs, donde es posible un paralelismo auténtico. Volveremos sobre esta cuestión en el capítulo 8.

Probablemente es más fácil apreciar porqué los threads son útiles dando algunos ejemplos concretos. Como primer ejemplo, consideremos un procesador de texto. La mayoría de los procesadores de texto visualizan en la pantalla el documento que se está creando formateado exactamente como aparecería una vez impreso. En particular, todos los saltos de línea y de página aparecen en su posición correcta final, de forma que el usuario puede inspeccionarlos y modificar el documento si es necesario (por ejemplo eliminando las líneas viudas y huérfanas – es decir las líneas de párrafos incompletos que aparecen en la parte de arriba y de abajo de una página, las cuales se consideran estéticamente desagradables).

Supongamos que el usuario está escribiendo un libro. Desde el punto de vista del autor es más cómodo meter el libro entero en un único fichero con el fin de hacer más fácil la búsqueda por temas, realizar sustituciones globales, etc. Alternativamente, puede ponerse cada capítulo en un fichero separado. Sin embargo teniendo cada sección y subsección como un fichero separado es un fastidio cuando hay que realizar modificaciones globales al libro entero ya que en ese caso puede ser necesario tener que editar cientos de ficheros. Por ejemplo si el estándar propuesto xxxx se aprueba justo antes de que el libro vaya a la imprenta, entonces es necesario sustituir en el último minuto todas las apariciones de “el estándar propuesto xxxx” por “el estándar xxxx”. Si el libro entero es un único fichero, normalmente es suficiente con un único comando para realizar todas las sustituciones. Por el contrario, si el libro se extiende sobre 300 ficheros, cada fichero debe editarse por separado.

Consideremos ahora qué sucede cuando el usuario borra repentinamente una frase de la página 1 de un documento de 800 páginas. Después de revisar la página modificada para asegurarse de que es correcta, el usuario puede querer realizar otra modificación en la página 600 por lo que introduce un comando diciéndole al procesador de texto que vaya a esa página (posiblemente pidiéndole que busque una frase que sólo aparece en esa página). En ese caso, el procesador de texto se ve forzado a reformatear inmediatamente todo el libro hasta la página 600 ya que el procesador no puede saber cuál es la primera línea de la página 600 hasta que no haya procesado todas las páginas anteriores. Aquí puede producirse una espera considerable antes de que pueda visualizarse la página 600, encontrándonos entonces con un usuario descontento con el procesador de texto.

En este caso los threads pueden ayudarnos. Supongamos que el procesador de texto está escrito como un programa con dos threads. Un thread interactúa con el usuario y el otro realiza el reformato como una actividad de fondo. Tan pronto como se borra la frase de la página 1, el thread interactivo indica al thread de reformato que reformatee todo el libro. Mientras tanto, el thread interactivo continúa atendiendo al teclado y al ratón y responde a comandos sencillos como realizar el *scroll* de la página 1 mientras el otro thread sigue trabajando frenéticamente en un segundo plano. Con un poco de suerte, el reformato se completa antes de que el usuario pida ver la página 600, de forma que en ese momento puede visualizarse instantáneamente.

Llegados a este punto, ¿por qué no añadir un tercer thread? Muchos procesadores de texto ofrecen la posibilidad de salvar automáticamente todo el fichero en el disco cada pocos minutos para proteger al usuario de la pérdida de su trabajo diario a causa de un programa que se bloquea, una caída del sistema o un fallo del suministro eléctrico. El tercer thread puede ocuparse de los backups (copias de seguridad, respaldos) en el disco sin interferir con los otros dos. La situación con los tres threads se muestra en la Figura 2-9.

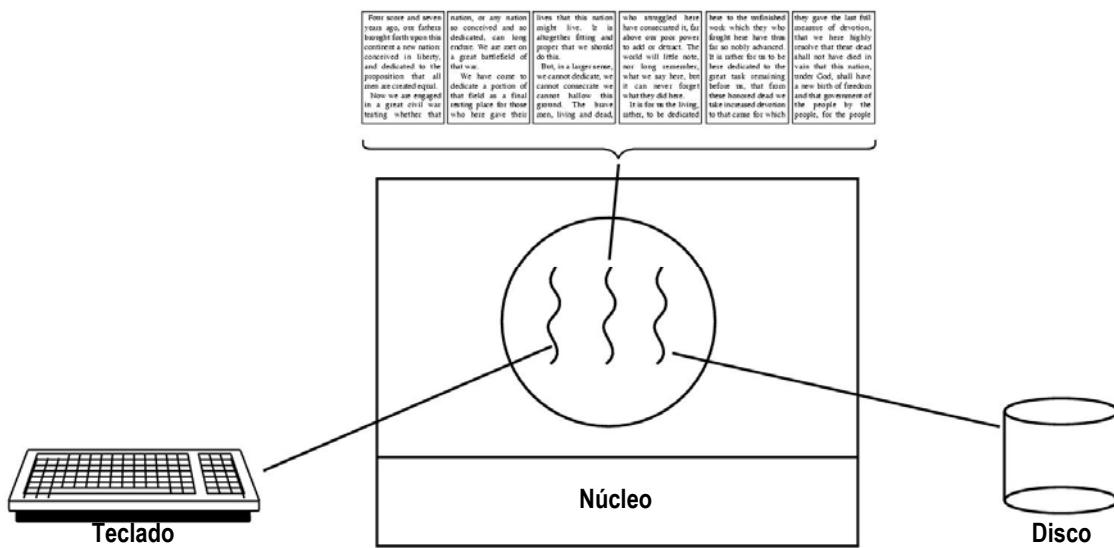


Figura 2-9. Un procesador de texto con tres threads.

Si el programa correspondiente al procesador de texto tuviera tan sólo un único thread, se tendría que cada vez que comenzase un backup al disco, deberían ignorarse los comandos procedentes del teclado y del ratón hasta el momento en que el backup terminase. Está claro que el usuario podría percibir esa lentitud como un pobre rendimiento. De forma alternativa, podríamos permitir que las señales del teclado y del ratón interrumpieran el backup al disco, haciendo posible obtener un buen rendimiento pero volviendo a caer en un complejo modelo de programación dirigido por interrupciones. Con tres threads, el modelo de programación es más simple. El primer thread se limita a interactuar con el usuario. El segundo thread reformatea el

documento cuando se le ordena. Finalmente el tercer thread escribe el contenido de la RAM al disco periódicamente.

Está claro que aquí no funcionaría bien tener tres procesos debido a que los tres threads necesitan operar sobre el documento. Teniendo tres threads en vez de tres procesos, se consigue que al compartir una memoria común todos tengan acceso al documento que se está editando.

Se da una situación análoga en otros muchos programas interactivos. Por ejemplo, una hoja de cálculo es un programa que permite a un usuario mantener una matriz, algunos de cuyos elementos son datos proporcionados por el usuario. Otros elementos de la matriz se calculan a partir de los datos de entrada utilizando fórmulas potencialmente muy complejas. Cuando un usuario modifica un elemento puede ser necesario recalcular muchos otros elementos. Teniendo un thread de fondo que se dedique a rehacer los cálculos, el thread interactivo puede permitir al usuario realizar cambios adicionales mientras se realizan los cálculos. De forma similar puede dedicarse un tercer thread para que realice backups periódicos al disco por su cuenta.

Consideremos todavía otro ejemplo de dónde pueden ser útiles los threads: un servidor para un sitio World Wide Web. El servidor recibe peticiones de páginas y envía las páginas solicitadas de vuelta a los clientes. En la mayoría de los sitios web, se accede más frecuentemente a algunas páginas que a otras. Por ejemplo, se accede mucho más a la página de entrada de Sony que a una página situada en las profundidades del árbol de páginas contenido las especificaciones técnicas de alguna cámara de vídeo particular. Los servidores web aprovechan este hecho para mejorar su rendimiento manteniendo en memoria una colección de páginas utilizadas muy frecuentemente, con lo que evitan tener que acceder al disco para obtenerlas. Tal colección se denomina una **caché**, utilizándose esta técnica también en muchos otros contextos.

En la Figura 2-10 se muestra una forma de organizar el servidor web. Aquí un thread, el **despachador** (*dispatcher*), lee las peticiones de servicio que se le hacen a través de la red. Después de examinar la petición, elige un **thread obrero** ocioso (y por tanto bloqueado) y le pasa la petición, escribiendo posiblemente un puntero al mensaje en una palabra especial asociada con cada thread. En ese momento el despachador desbloquea al thread obrero, que pasa al estado de preparado.

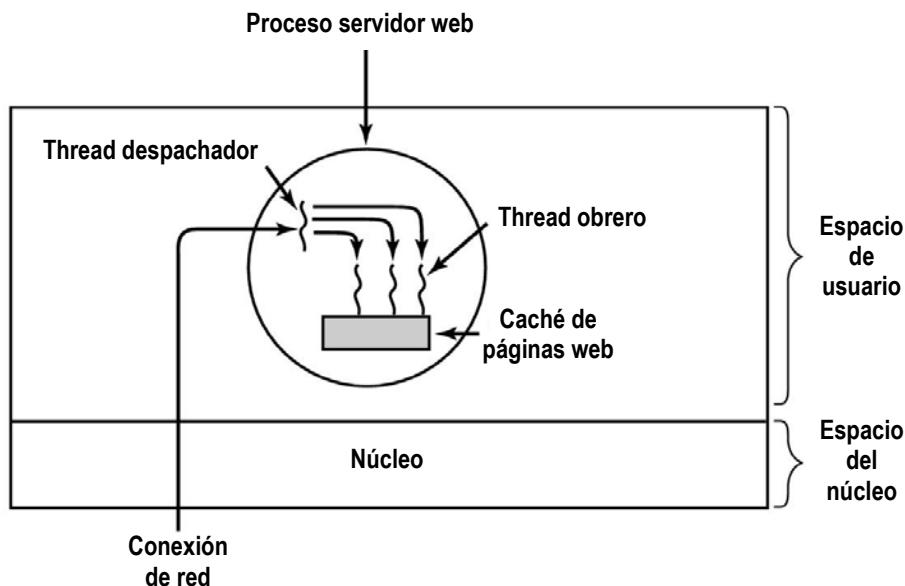


Figura 2-10. Un servidor web multihilado.

Cuando el thread obrero pasa a ejecución, comprueba si la petición puede satisfacerse desde la caché de páginas web, a la cual tienen acceso todos los threads. En caso contrario el thread arranca una operación `read` para obtener la página desde el disco y bloquearse hasta que la operación de disco se complete. Cuando el thread se bloquea sobre la operación del disco, se elige a otro thread para ejecutarse, posiblemente el despachador, en orden a aceptar más trabajo, o posiblemente a otro thread obrero que esté ahora preparado para ejecutarse.

Este modelo permite escribir el servidor como una colección de threads secuenciales. El programa del despachador consiste de un bucle infinito que obtiene una petición de servicio y la trata a través de un thread obrero. El código de cada thread obrero se reduce a un bucle infinito en el que se acepta una petición encomendada por el despachador y se comprueba si la página está presente en la caché de páginas web. Si lo está, se envía la página de vuelta al cliente y el thread obrero se bloquea esperando que se le encomiende una nueva petición. Si la página no está en la caché, dicha página se busca en el disco, tras lo cual se la envía de vuelta al cliente, bloqueándose a continuación el thread obrero a la espera de una nueva petición.

En la Figura 2-11 se muestra un esbozo simplificado del código. Aquí, como en el resto de este libro, se supone que `TRUE` representa la constante 1. Además, `peticion` y `pagina` son estructuras apropiadas para almacenar una petición de servicio y una página web, respectivamente.

```
while (TRUE) {
    obtener_siguiente(&peticion);
    encomendar_trabajo(&peticion);
}
(a)
```



```
while (TRUE) {
    esperar_a_que_haya_trabajo(&peticion);
    buscar_pagina_en_la_cache(&peticion, &pagina);
    if (no_esta_en_la_cache(&pagina))
        leer_pagina_del_disco(&peticion, &pagina);
    servir_pagina(&pagina);
}
(b)
```

Figura 2-11. Un esbozo simplificado del código para la Figura 2-10. (a) Thread despachador. (b) Thread obrero.

Consideremos la forma en que tendría que haberse escrito el servidor web en ausencia de threads. Una posibilidad sería que el servidor operase como un único thread. El bucle principal del servidor web toma una petición, la examina y la atiende hasta que se completa antes de pasar a la siguiente petición. El servidor estará ocioso mientras espera por una lectura del disco y mientras tanto no podrá procesar ninguna de las otras peticiones de servicio que están pendientes. Si el servidor web se ejecuta sobre una máquina dedicada, como normalmente es el caso, la CPU simplemente estará ociosa mientras el servidor web se encuentra esperando por el disco. El resultado neto es que van a poder procesarse muchas menos peticiones de páginas web por segundo. Por tanto la estructuración del servidor en múltiples threads produce una considerable mejora de su rendimiento, programándose cada thread secuencialmente de la forma usual.

Hasta aquí hemos visto dos diseños posibles: un servidor web multihilo y un servidor web con un único thread. Supongamos que los threads no están disponibles pero que los diseñadores del sistema se encuentran con que es inaceptable la pérdida de eficiencia debida a la utilización de un único thread. Si el sistema dispone de una versión no bloqueante de la llamada al sistema `read`, es posible un tercer enfoque del problema. Cuando entra una nueva petición el único thread existente la examina. Si puede satisfacerse desde la caché, perfecto, pero si no es así, se arranca una operación no bloqueante de lectura del disco. .

El servidor guarda el estado de la petición actual en una tabla y pasa a atender el siguiente evento. El siguiente evento puede ser bien una nueva petición de otra página web, o una respuesta del disco a una operación de lectura previa. Si se trata de una nueva petición, se atiende como la anterior. Si se trata de una respuesta del disco, se extrae de la tabla la información relevante y se procesa la respuesta. Con E/S del disco no bloqueante, las respuestas del disco tendrán probablemente la forma de una señal o una interrupción.

En este diseño, el modelo de los “procesos secuenciales” que teníamos en los dos primeros casos se pierde. El estado del procesamiento de las peticiones debe salvarse y restaurarse explícitamente cada vez que el servidor comuta de trabajar sobre una petición a trabajar sobre otra. Realmente lo que se está haciendo es simular los threads y sus pilas de la manera más difícil. Un diseño como este en el que cada procesamiento tiene un estado guardado y existe algún conjunto de eventos que pueden suceder para alterar su estado se denomina una **máquina de estados finitos** (o un **autómata finito determinista**). Este concepto se utiliza muy a menudo en informática.

Debe estar claro ahora qué es lo que nos ofrecen los threads. Ellos hacen posible mantener la idea de procesos secuenciales que hacen llamadas al sistema bloqueantes (por ejemplo E/S del disco) sin que se pierda el paralelismo. Las llamadas al sistema bloqueantes hacen más sencilla la programación, mientras que el paralelismo mejora el rendimiento. El servidor web con un único thread mantiene la sencillez de las llamadas al sistema bloqueantes, pero pierde en rendimiento. El tercer enfoque consigue un alto rendimiento gracias al paralelismo, pero utiliza llamadas no bloqueantes e interrupciones, por lo que es muy difícil de programar. Estos tres modelos se resumen en la Figura 2-12.

| Modelo | Características |
|------------------------------|--|
| Threads | Paralelismo y llamadas al sistema bloqueantes |
| Proceso monohilo | Sin paralelismo y llamadas al sistema bloqueantes |
| Autómata finito determinista | Paralelismo, llamadas al sistema no bloqueantes e interrupciones |

Figura 2-12. Tres formas de construir un servidor.

Un tercer ejemplo donde son útiles los threads es en aplicaciones que deben procesar cantidades muy grandes de datos. El enfoque más normal es leer un bloque de datos, procesarlo y a continuación escribirlo de nuevo. Aquí el problema es que si sólo están disponibles llamadas al sistema bloqueantes, el proceso se bloquea mientras los datos están entrando y mientras los resultados se están escribiendo. Tener la CPU ociosa mientras hay una gran cantidad de trabajo por realizar es claramente un despilfarro que debe evitarse siempre que sea posible.

Los threads ofrecen una solución. El proceso puede estructurarse en un thread de entrada, un thread de procesamiento y un thread de salida. El thread de entrada lee los datos dejándolos en un búfer de entrada. El thread de procesamiento toma datos del búfer de entrada, los procesa, y deja el resultado en un búfer de salida. El thread de salida escribe esos resultados en el disco. De esta manera la entrada, la salida y el procesamiento pueden ir haciéndose al mismo tiempo. Por supuesto, este modelo sólo funciona si una llamada al sistema bloquea sólo al thread que la invoca, y no al proceso entero que contiene a ese thread.

2.2.3 Implementación de los Threads en el Espacio del Usuario

Existen dos formas fundamentales de implementar un paquete de threads: en el espacio del usuario y en el núcleo (*kernel*). La elección entre esas dos implementaciones resulta moderadamente controvertida, siendo también posible una implementación híbrida. Vamos a pasar a describir estos métodos, junto con sus ventajas y desventajas.

El primer método consiste en poner el paquete de threads enteramente en el espacio de usuario. En consecuencia el núcleo del sistema no sabe nada de su existencia. En lo que concierne al núcleo, sólo se gestionan procesos ordinarios con un único thread. La primera, y más obvia, ventaja es que el paquete de threads a nivel de usuario puede implementarse sobre un sistema operativo que no soporte threads. Todos los sistemas operativos tradicionales entran dentro de esta categoría, y algunos de ellos siguen incluso actualmente sin dar soporte a los threads.

Todas estas implementaciones tienen la misma estructura general, que se ilustra en la Figura 2-13(a). Los threads se ejecutan en lo alto de un **sistema en tiempo de ejecución** (*runtime system*), que es una colección de procedimientos que gestiona los threads. Hemos visto cuatro de estos procedimientos anteriormente: *thread_create*, *thread_exit*, *thread_wait* y *thread_yield*, pero normalmente hay algunos procedimientos más.

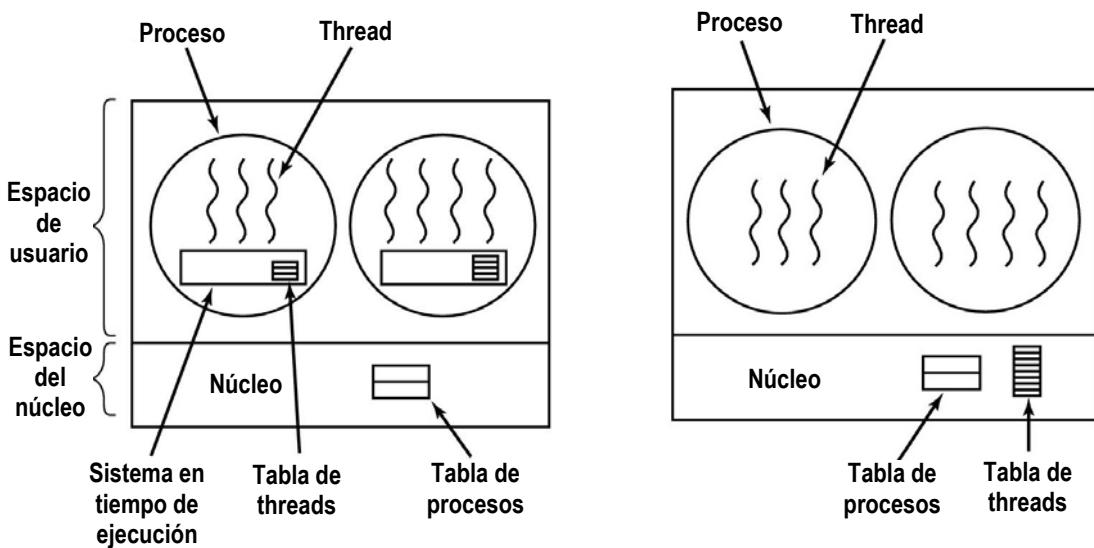


Figura 2-13. (a) Un paquete de threads a nivel de usuario.
(b) Un paquete de threads gestionado por el núcleo.

Cuando los threads se gestionan en el espacio del usuario, cada proceso necesita su propia **tabla de threads** privada para llevar el control de sus threads. Esta tabla es análoga a la tabla de procesos del núcleo, salvo que sólo controla las propiedades propias de los threads tales como el contador de programa del thread, su puntero de pila, sus registros, su estado, etc. La tabla de threads está gestionada por el sistema en tiempo de ejecución. Cuando un thread pasa al estado preparado o bloqueado, la información necesaria para proseguir posteriormente con su ejecución se guarda en la tabla de threads, exactamente de la misma forma que el núcleo almacena la información sobre los procesos en la tabla de procesos.

Cuando un thread hace algo que puede provocar que se bloquee localmente, como por ejemplo, esperar a que otro thread en su mismo proceso complete algún trabajo, entonces llama a un procedimiento del sistema en tiempo de ejecución. Este procedimiento comprueba si el thread debe pasar al estado bloqueado. Si es así, el procedimiento guarda los registros del thread (que son los mismos que los suyos propios) en la tabla de threads, busca en la tabla un thread preparado para ejecutarse, y recarga los registros de la máquina con los valores que se tienen guardados correspondientes al nuevo thread. Tan pronto como se comutan el puntero de pila y el contador de programa, el nuevo thread vuelve automáticamente otra vez a la vida. Si la máquina tiene una instrucción para guardar todos los registros, y otra para cargarlos todos, la comutación de un thread a otro puede hacerse ejecutando un puñado de instrucciones. Este tipo de comutación de threads es al menos un orden de magnitud más rápido que hacer un trap al núcleo, lo que representa un potente argumento a favor de los paquetes de threads a nivel de usuario.

Sin embargo existe una diferencia clave con respecto a los procesos. Cuando un thread termina de ejecutarse por el momento, por ejemplo cuando realiza una llamada a *thread_yield*, el código de *thread_yield* puede salvar él mismo la información del thread en la tabla de threads. Además, puede llamar al thread planificador para que escoja otro thread para pasarlo a ejecución. El procedimiento que salva el estado del thread y el planificador no son más que procedimientos locales, por lo que su invocación es mucho más eficiente que realizar una llamada al núcleo. Entre otras cuestiones, no es necesario ningún trap, no es necesario ningún cambio de contexto, no es necesario vaciar la memoria caché, etc. Esto hace que la planificación de los threads sea muy rápida.

Los threads a nivel de usuario tienen también otras ventajas. Para empezar, permiten que cada proceso tenga su propio algoritmo de planificación ajustado a sus necesidades. Para algunas aplicaciones, como por ejemplo aquellas con un thread recolector de basura, es una ventaja adicional el no tener que preocuparse sobre si un thread se ha quedado detenido en un momento inadecuado. También se dimensionan mejor, ya que los threads a nivel del núcleo requieren invariablemente en el núcleo algún espacio para la tabla de threads y algún espacio de pila, lo que puede resultar un problema cuando el número de threads es muy grande.

A pesar de su mayor eficiencia, los paquetes de threads a nivel de usuario tienen algunos serios problemas. El primero de ellos es el problema de cómo se implementan las llamadas al sistema bloqueantes. Supongamos que un thread lee desde el teclado antes de que se haya pulsado ninguna tecla. Es inaceptable permitir que el thread haga realmente esa llamada al sistema, ya que eso detendría a todos los threads del proceso. Uno de los principales motivos con que hemos justificado la necesidad de incorporar al sistema los threads fue el permitir utilizar llamadas bloqueantes pero sin que el bloqueo de un thread afectase a los demás. Con llamadas al sistema bloqueantes, es difícil imaginarse cómo puede conseguirse este objetivo fácilmente.

Podemos modificar todas las llamadas al sistema para que no sean bloqueantes (por ejemplo haciendo que la llamada **read** sobre el teclado retorne inmediatamente 0 bytes si no hay caracteres en el búfer del teclado), pero exigir cambios sobre el sistema operativo es muy poco atractivo. Aparte de eso, uno de los argumentos para incorporar los threads a nivel de usuario fue precisamente que pudieran ejecutarse en los sistemas operativos *existentes*. Adicionalmente, cambiar la semántica de **read** requeriría también realizar cambios en muchos de los programas de usuario ya escritos.

Es posible otra alternativa en aquellos casos en los que es posible determinar con antelación si una llamada al sistema va a producir un bloqueo. En algunas versiones de UNIX, existe una llamada al sistema, **select**, que permite al que la invoca saber si una determinada llamada **read** que se propone realizar va a bloquearse o no. Cuando se dispone de esta llamada, el procedimiento de librería **read** puede reemplazarse por uno nuevo que primero realiza una llamada a **select** y luego hace la llamada a **read** si es segura (es decir si no va a producir un bloqueo). Si por el contrario la llamada **read** va a producir un bloqueo, la llamada no se hace, y en su lugar se ejecuta otro thread. La siguiente vez que el sistema en tiempo de ejecución tome el control, puede comprobar de nuevo si la llamada **read** es ya segura. Este enfoque requiere reescribir partes de la librería de llamadas al sistema, es ineficiente y nada elegante, pero no queda otra elección. El código añadido alrededor de la llamada al sistema para hacer las comprobaciones sobre la seguridad de la llamada se denomina un **jacket** (chaqueta) o **wrapper** (envoltorio).

Un problema análogo al de las llamadas al sistema bloqueantes es el problema de las faltas de página. Estudiaremos ese problema en el capítulo 4, pero de momento es suficiente con decir que los ordenadores pueden configurarse de forma que no todo el programa esté en memoria principal a la vez. Si el programa llama o salta a una instrucción que no está en memoria, tiene lugar una falta de página y el sistema operativo debe ir y tomar las instrucciones no encontradas (y sus vecinas) obteniéndolas del disco. Eso es lo que se denomina una *falta de página*. El proceso se bloquea mientras se localizan y leen las instrucciones necesarias. Si un thread provoca una falta de página, el núcleo, que ni siquiera sabe de la existencia de los threads, bloquea en consecuencia al proceso entero hasta que la E/S del disco provocada por la falta de página se complete, y eso incluso aunque haya otros threads ejecutables dentro del proceso.

Otro problema con los paquetes de threads a nivel de usuario es que si un thread comienza a ejecutarse, ningún otro thread en ese proceso podrá volver a ejecutarse mientras que el primer thread no ceda voluntariamente la CPU. Dentro de un único proceso, no existen interrupciones de reloj, lo que hace imposible planificar los threads de una forma tipo round-robin (es decir turnándose periódicamente). A menos que un thread entre en el sistema en tiempo de ejecución por su propia voluntad, el planificador nunca tiene la oportunidad de pasar a ejecución a otro thread.

Una posible solución al problema de la ejecución indefinidamente prolongada de los threads es hacer que el sistema en tiempo de ejecución solicite una señal de reloj (interrupción) una vez cada segundo para obtener el control, pero eso es también algo rudimentario y complicado de programar. Las interrupciones periódicas del reloj no siempre son posibles con una alta frecuencia, pero incluso aunque lo sean, generan una considerable sobrecarga total en el sistema. Además, un thread también puede necesitar su propia interrupción de reloj, la cual podría interferir con el uso que el sistema en tiempo de ejecución estaría ya haciendo del reloj.

Otro, y probablemente el argumento más demoledor en contra de a los threads a nivel de usuario es que, en general, los programadores desean utilizar los threads precisamente en las aplicaciones donde los threads se bloquean muy a menudo, como por ejemplo en un servidor web multihilo. Esos threads están constantemente haciendo llamadas al sistema. Una vez que el thread ha hecho un trap al núcleo para llevar a cabo una llamada al sistema, no significa mucho más trabajo para el núcleo commutar a otro thread si el primero se bloquea, y dejar que el núcleo haga eso elimina la necesidad de hacer constantemente llamadas al sistema **select** para comprobar si las llamadas al sistema **read** son seguras. Para aplicaciones que esencialmente mantienen muy ocupada la CPU bloqueándose raramente, ¿cuál es la ventaja de disponer de threads? Nadie puede proponer seriamente calcular los n primeros números primos o jugar al ajedrez utilizando threads, debido a que no hay nada que ganar haciéndolo de esa manera.

2.2.4 Implementación de los Threads en el Núcleo

Ahora vamos a considerar el caso en el que el núcleo conoce y gestiona los threads. En ese caso, como se muestra en la Figura 2-13(b), no es necesario ningún sistema en tiempo de ejecución dentro de cada proceso. Igualmente no existe ninguna tabla de threads en cada proceso. En vez de eso, el núcleo mantiene una tabla de threads que sigue la pista de todos los threads en el sistema. Cuando un thread desea crear un nuevo thread o destruir uno que ya existe, hace una llamada al núcleo, que es el que se encarga efectivamente de su creación o destrucción actualizando la tabla de threads del sistema.

La tabla de threads del núcleo guarda los registros de cada thread, su estado y otra información. La información es la misma que con threads a nivel de usuario, pero ahora esa información está en el núcleo en vez de en el espacio de usuario (dentro del sistema en tiempo de ejecución). Esta información es un subconjunto de la información que los núcleos tradicionales mantienen sobre cada uno de sus procesos con un solo thread, esto es, el estado del proceso. Adicionalmente, el núcleo mantiene también la tabla de procesos para seguir la pista de los procesos.

Todas las llamadas que puedan bloquear a un thread se implementan como llamadas al sistema, a un coste considerablemente más alto que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando se bloquea un thread, el núcleo tiene la opción de decidir si pasa a ejecutar otro thread del mismo proceso (si es que hay alguno preparado), o pasa a ejecutar un thread de un proceso diferente. Con threads a nivel de usuario el sistema en tiempo de ejecución tiene que seguir ejecutando threads de su propio proceso hasta que el núcleo le arrebate la CPU (o hasta que no le queden threads preparados para ejecutarse).

Debido al coste relativamente alto de crear y destruir los threads en el núcleo, algunos sistemas toman un enfoque medio-ambientalmente correcto reciclando sus threads. Cuando se destruye un thread, éste se marca como un thread no ejecutable, pero sin que se vean afectadas de otro modo sus estructuras de datos del núcleo. Posteriormente, cuando haya que crear un nuevo thread, se procederá a reactivar un antiguo thread marcado, ahorrando de esta manera algo de la sobrecarga de la creación normal de un thread. También es posible el reciclado de threads para threads a nivel de usuario, pero ya que la sobrecarga de la gestión de los threads es mucho menor, el incentivo para reciclar resulta menos atractivo.

Los threads a nivel del núcleo no requieren ninguna llamada al sistema nueva de tipo no bloqueante. Además, si un thread de un proceso provoca una falta de página, el núcleo puede comprobar fácilmente si el proceso tiene todavía threads ejecutables, y en ese caso pasar a ejecutar uno de ellos mientras espera a que la página que provocó la falta se cargue desde el disco. Su principal desventaja es que el coste de una llamada al sistema es considerable, de manera que si las operaciones con threads (creación, terminación, etc.) son frecuentes, puede incurrirse en una elevada sobrecarga para el sistema.

2.2.5 Implementaciones Híbridas

Se han investigado varias líneas que intentan combinar las ventajas de los threads a nivel de usuario con las ventajas de los threads a nivel del núcleo. Una línea consiste en utilizar threads a nivel del núcleo y multiplexar threads a nivel de usuario sobre algunos o todos los threads a nivel del núcleo, como se muestra en la Figura 2-14.

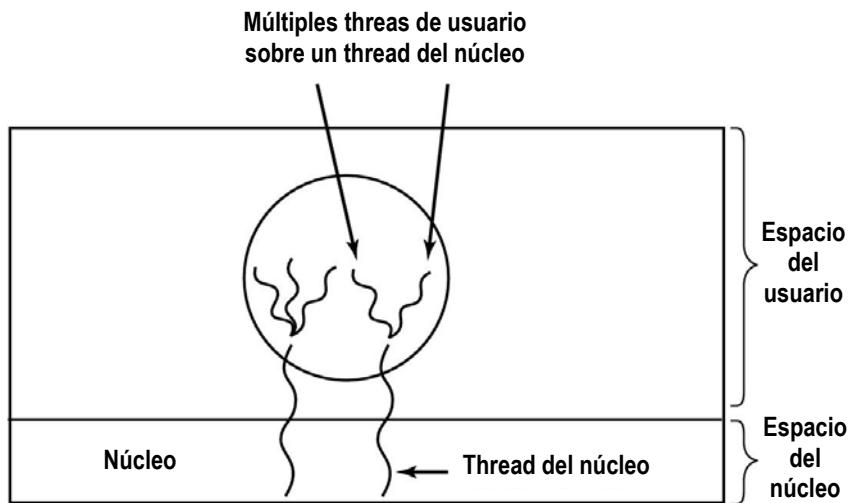


Figura 2-14. Multiplexación de los threads a nivel de usuario sobre los threads a nivel del núcleo.

En este diseño el núcleo sólo tiene conocimiento de los threads a nivel del núcleo, ocupándose de su planificación. Algunos de estos threads pueden tener multiplexados sobre ellos múltiples threads a nivel de usuario. Estos threads a nivel de usuario se crean, se destruyen y se planifican de la misma manera que los threads a nivel de usuario de un proceso que se ejecuta sobre un sistema operativo sin capacidad multihilera. En este modelo, cada thread a nivel de núcleo tiene algún conjunto de threads a nivel de usuario que lo utilizan por turnos para ejecutarse.

2.2.6 Activaciones del Planificador

Varios investigadores han intentado combinar la ventaja de los threads a nivel de usuario (buen rendimiento) con la ventaja de los threads a nivel del núcleo (no tener que utilizar un montón de trucos para que las cosas funcionen). A continuación vamos a describir uno de esos enfoques diseñado por Anderson y otros (1992), denominado **activaciones del planificador**. Se discuten trabajos relacionados en los artículos de Edler y otros (1998) y Scott y otros (1990).

Los objetivos del trabajo sobre activaciones del planificador son imitar la funcionalidad de los threads a nivel del núcleo, pero con el mejor rendimiento y la mayor flexibilidad usualmente asociada con los paquetes de threads implementados en el espacio del usuario. En particular, los threads a nivel del usuario no tienen que hacer llamadas al sistema especiales no bloqueantes o comprobar por adelantado si es seguro realizar ciertas llamadas al sistema. Sin embargo, cuando un thread se bloquea en una llamada al sistema o debido a una falta de página, debe ser posible ejecutar otros threads dentro del mismo proceso, supuesto que haya algún thread preparado.

La eficiencia se consigue evitando transiciones innecesarias entre el espacio del usuario y del núcleo. Por ejemplo si un thread se bloquea esperando a que otro thread haga algo, no existe ninguna razón para involucrar al núcleo, ahorrando así la sobrecarga de la transición núcleo-usuario. El sistema en tiempo de ejecución presente en el espacio de usuario puede bloquear al thread que requiere la sincronización y planificar uno nuevo por sí mismo.

Cuando se utilizan las activaciones del planificador, el núcleo asigna un cierto número de procesadores virtuales a cada proceso y deja que el sistema en tiempo de ejecución (en el espacio de usuario) asigne threads a procesadores. Este mecanismo puede utilizarse también sobre un multiprocesador donde los procesadores virtuales pueden ser CPUs reales. El número de procesadores virtuales asignados a un proceso es inicialmente de uno sólo, pero el proceso puede pedir más y puede también devolver procesadores que ya no necesita. El núcleo puede también recuperar procesadores virtuales asignados anteriormente, en orden a asignárselos a otros procesos más necesitados.

La idea básica que hace que este esquema funcione es que cuando el núcleo detecta que un thread se ha bloqueado (por ejemplo, porque el thread ha ejecutado una llamada al sistema bloqueante o ha provocado una falta de página), el núcleo se lo notifica al sistema en tiempo de ejecución del proceso, pasándole como parámetros sobre la pila el número del thread en cuestión y una descripción del suceso que ha tenido lugar. La notificación ocurre mediante la activación por parte del núcleo del sistema en tiempo de ejecución en una dirección de comienzo conocida, de forma más o menos análoga a una señal en UNIX. A este mecanismo se le denomina una **llamada ascendente** (*upcall*).

Una vez que se ha activado de esa manera, el sistema en tiempo de ejecución puede replanificar sus threads, normalmente marcando el thread actual como bloqueado y tomando otro thread de la lista de preparados, estableciendo sus registros y rearrancándolo. Posteriormente, cuando el núcleo detecte que el thread original puede ejecutarse de nuevo (por ejemplo porque la tubería que estaba tratando de leer ahora contiene datos, o porque se ha terminado de cargar desde el disco la página correspondiente a una falta de página), el núcleo hace otra llamada ascendente al sistema en tiempo de ejecución para informar de este suceso. El sistema en tiempo de ejecución, puede decidir en ese momento, si rearrastra inmediatamente el thread bloqueado, o lo pone en la lista de preparados para que se ejecute posteriormente.

Cuando tiene lugar una interrupción hardware mientras se está ejecutando un thread de usuario, la CPU interrumpida pasa a modo núcleo. Si la interrupción está provocada por un suceso que no tiene que ver con el proceso interrumpido, tal como la finalización de la E/S de otro proceso, entonces cuando termina la rutina de tratamiento de la interrupción se vuelve a poner al thread interrumpido en el mismo estado en el que estaba antes de la interrupción. Si, por el contrario, el proceso tiene parte en la interrupción, tal como en la llegada de una página necesaria por uno de los threads del proceso, el thread interrumpido no se rearrastra. En vez de eso, se suspende ese thread y se arranca el sistema en tiempo de ejecución sobre esa CPU virtual, con el estado del thread interrumpido en la pila. Corresponde entonces al sistema en tiempo de ejecución decidir qué thread planificar sobre esa CPU: el thread interrumpido, el thread nuevamente preparado, o algún tercer thread elegido.

Una objeción a las activaciones del planificador es que es un mecanismo que depende fundamentalmente de las llamadas ascendentes, un concepto que viola la estructura inherente en cualquier sistema estructurado en capas. Normalmente, la capa n ofrece ciertos servicios que la capa $n + 1$ puede invocar, pero la capa n no puede invocar a procedimientos que están en la capa $n + 1$. Las llamadas ascendentes no respetan ese principio fundamental.

2.2.7 Threads Emergentes

Frecuentemente los threads son muy útiles en sistemas distribuidos. Un ejemplo importante es cómo se tratan los mensajes de entrada, por ejemplo peticiones de servicio. El enfoque tradicional es tener un proceso o thread que está bloqueado en una llamada al sistema `receive` esperando a que llegue un mensaje de entrada. Cuando llega un mensaje, ese proceso o thread acepta el mensaje y lo procesa.

Sin embargo, es posible un enfoque completamente diferente, en el cual la llegada de un mensaje provoca que el sistema cree un nuevo thread para tratar el mensaje. Un tal thread se denomina un **thread emergente** (*pop-up thread*) y se ilustra en la Figura 2-15. Una ventaja clave de los threads emergentes es que ya que son trigo nuevo, no tienen ninguna historia – registros, pila, etc. que deba restaurarse. Cada uno comienza limpio y cada uno es idéntico a todos los demás. Esto hace posible crear muy rápidamente tales threads. Al nuevo thread se le da el mensaje de entrada a procesar. El resultado de utilizar los threads emergentes es que la latencia entre la llegada del mensaje y el comienzo del procesamiento puede hacerse muy pequeña.

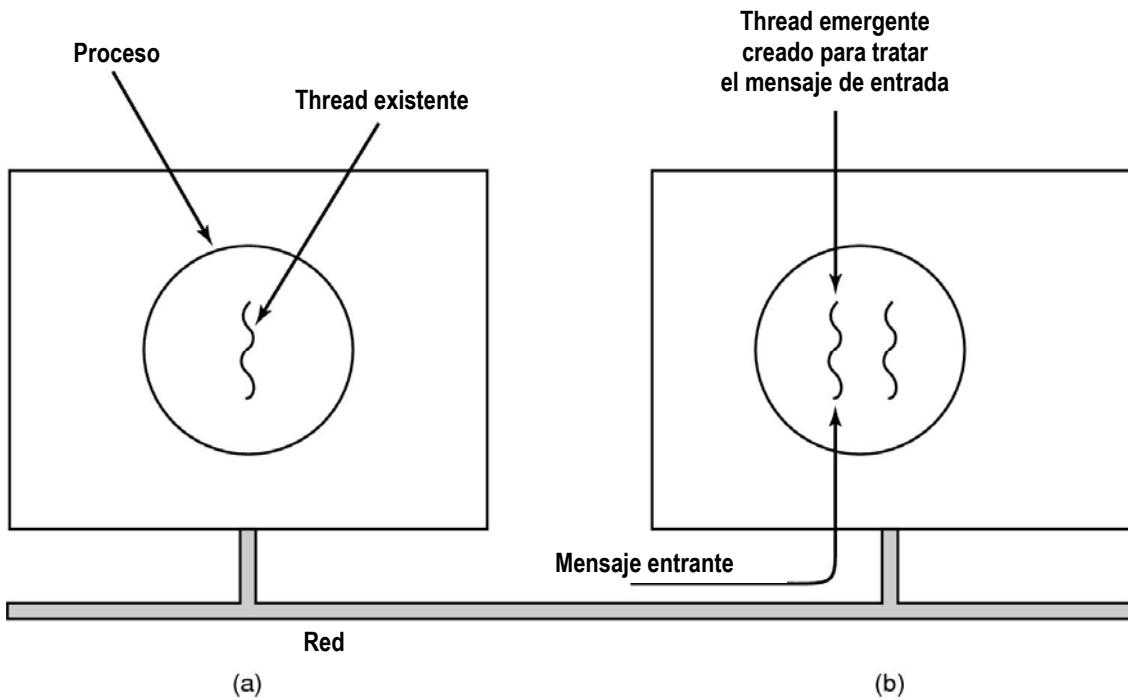


Figura 2-15. Creación de un nuevo thread cuando llega un mensaje.

(a) Antes de que llegue el mensaje. (b) Despues de que llegue el mensaje.

Es necesario tener las cosas planeadas por adelantado cuando se utilizan threads emergentes. Por ejemplo, ¿en qué proceso se ejecuta el thread? Si el sistema dispone de threads ejecutándose en el contexto del núcleo, el thread puede ejecutarse allí (que es por lo que no hemos mostrado el núcleo en la Figura 2-15). Hacer que el thread emergente se ejecute en el espacio del núcleo es usualmente más fácil y rápido que ponerlo en el espacio del usuario. También, un thread emergente en el espacio del núcleo puede acceder más fácilmente a todas las tablas del núcleo y a los dispositivos de E/S, que pueden ser necesarios para el procesamiento de interrupciones. Del otro lado, un thread del núcleo con errores de programación puede hacer mucho más daño que un thread de usuario igualmente erróneo. Por ejemplo, si el thread se ejecuta durante demasiado tiempo y no existe manera de expulsarlo, es posible que se pierdan los datos que llegan.

2.2.8 Conversión de Código Secuencial en Código Multihilo.

Muchos programas existentes se escribieron para ejecutarse como procesos con un único thread. El convertir esos programas en multihilo es mucho más complicado de lo que pueda parecer en un primer momento. A continuación vamos a examinar unas pocas de las dificultades que suelen aparecer.

Como punto de partida, el código de un thread consta normalmente de múltiples procedimientos, de la misma forma que un proceso. Estos procedimientos pueden tener variables locales, variables globales y parámetros. Las variables locales y los parámetros no provocan ningún problema, pero las variables que son globales a un thread pero no son globales al programa entero sí que los provocan. Éstas son variables que son globales en el sentido de que las usan varios procedimientos dentro del thread (de la misma forma que pueden usar cualquier variable global), pero los otros threads desde un punto de vista lógico no deben acceder a ellas.

Como ejemplo, consideremos la variable *errno* mantenida por UNIX. Cuando un proceso (o un thread) hace una llamada al sistema que falla, el código del error se deja en *errno*. En la Figura 2-16, el thread 1 ejecuta la llamada al sistema **access** para saber si tiene permiso para acceder a un cierto fichero. El sistema operativo devuelve la respuesta en la variable global *errno*. Después de que el control ha retorna do al thread 1, pero antes de que el thread 1 haya podido leer *errno*, el planificador decide que el thread 1 ha tenido suficiente tiempo de CPU por el momento y decide conmutar al thread 2. El thread 2 ejecuta una llamada **open** que también falla, lo que provoca que *errno* se sobrescriba y que el código de error del **access** del thread 1 se pierda para siempre. Cuando el thread 1 pase de nuevo a ejecución, leerá el valor erróneo y se comportará incorrectamente.

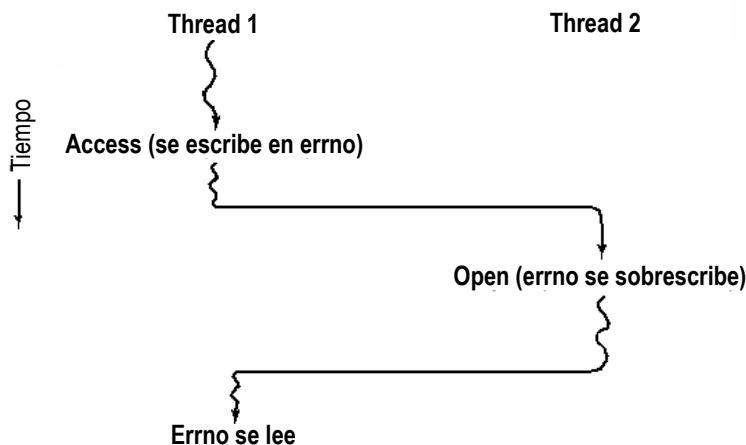


Figura 2-16. Conflictos entre threads en el uso de una variable global.

Son posibles varias soluciones a este problema. Una es prohibir completamente el uso de variables globales. Sin embargo por muy buena causa que sea este ideal, entra en conflicto con la mayoría del software existente. Otra solución es asignar a cada thread sus propias variables globales privadas, como se muestra en la Figura 2-17. De esta forma cada thread tiene su propia copia privada de *errno* y otras variables globales, evitando así cualquier conflicto. Efectivamente, esta decisión crea un nuevo nivel de ámbito, variables que son visibles a todos los procedimientos de un thread, adicionalmente a los niveles de ámbito existentes de variables visibles sólo por un procedimiento y variables visibles desde cualquier punto del programa.

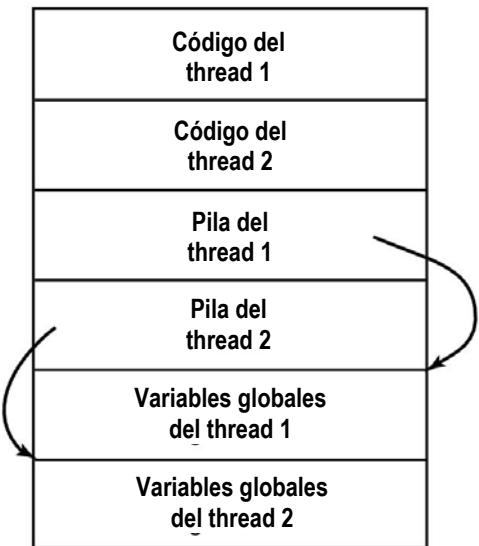


Figura 2-17. Los threads pueden tener variables globales privadas.

Sin embargo, el acceso a las variables privadas es un poco complicado, ya que la mayoría de los lenguajes de programación tienen una forma de expresar las variables locales y las variables globales, pero no formas intermedias. Es posible asignar un bloque de memoria para las variables globales y pasárselo a cada procedimiento en el thread, como un parámetro extra. Aunque difícilmente resulta una solución elegante, funciona.

Alternativamente, pueden introducirse nuevos procedimientos de biblioteca para crear, establecer y leer estas variables globales a lo largo del thread. La primera llamada podría ser como esta:

```
create_global("bufptr");
```

Esta llamada asigna memoria a un puntero denominado *bufptr* sobre el heap o en un área de memoria especial reservada para el thread que ha llamado. No importa donde se asigne la memoria, ya que sólo el thread que la invoca tiene acceso a esa variable global. Si otro thread crea una variable global con el mismo nombre, ese otro thread obtiene una posición de memoria diferente que no entra en conflicto con la existente.

Son necesarias dos llamadas para acceder a las variables globales: una para escribir en ellas y otra para leerlas. Para escribir, puede servir algo como:

```
set_global("bufptr", &buf);
```

Esta llamada almacena el valor de un puntero en la posición de memoria previamente creada por la llamada a *create_global*. Para leer una variable global, la llamada puede ser como la siguiente:

```
bufptr = read_global("bufptr");
```

Esa llamada devuelve la dirección almacenada en la variable global, así puede accederse a sus datos.

El siguiente problema para convertir un programa secuencial en un programa multihilo es que la mayoría de los procedimientos de biblioteca no son reentrantes. Esto es, no fueron diseñados para permitir que se realice una segunda llamada a cualquier procedimiento no habiendo finalizado todavía una llamada anterior. Por ejemplo, el envío de un mensaje sobre la red puede perfectamente haberse programado de forma que el mensaje se ensambla en un búfer

establecido dentro de la biblioteca, haciendo luego un trap al núcleo para enviarlo. ¿Qué ocurre si un thread ha ensamblado su mensaje en el búfer, y luego una interrupción del reloj fuerza que la CPU conmute a un segundo thread que sobrescribe inmediatamente el búfer con su propio mensaje?

De forma similar, los procedimientos de asignación de memoria, tales como *malloc* en UNIX, mantienen tablas cruciales sobre la utilización de la memoria, por ejemplo, una lista enlazada de bloques de memoria de distintos tamaños disponibles. Mientras *malloc* está ocupado actualizando estas listas, las listas pueden estar temporalmente en un estado inconsistente, con punteros que apuntan a ninguna parte. Si tiene lugar un cambio de thread mientras las tablas son inconsistentes y un thread diferente hace una nueva llamada, puede que se utilice un puntero inválido, conduciendo esto a un cuelgue del programa. Resolver adecuada y efectivamente todos estos problemas significa tener que reescribir completamente la biblioteca.

Una solución diferente es proporcionar a cada procedimiento una chaqueta (*jacket*) de código que establezca un bit para marcar la biblioteca indicando que está en uso. Cualquier intento por parte de otro thread de utilizar el procedimiento de biblioteca no habiéndose completado todavía una llamada previa, bloquea al thread. Aunque puede conseguirse que esta solución funcione, provoca una enorme reducción del paralelismo potencialmente existente.

A continuación, vamos a considerar las señales. Algunas señales son lógicamente específicas de los threads, mientras que otras no. Por ejemplo, si un thread invoca *alarm*, tiene sentido que la señal resultante se dirija al thread que la invocó. Sin embargo, cuando los threads se implementan completamente en el espacio del usuario, el núcleo no sabe nada sobre los threads, de forma que difícilmente puede dirigir la señal al thread correcto. Una complicación adicional ocurre si un proceso sólo puede tener pendiente a la vez una alarma y varios threads llaman a *alarm* de forma independiente.

Otras señales, tales como las interrupciones del teclado, no son específicas de ningún thread. ¿Quién debe capturarlas? ¿Un thread designado? ¿Todos los threads? ¿Un thread emergente nuevamente creado? Además, ¿qué sucede si un thread cambia los controladores de las señales sin avisar a los otros threads? ¿Y qué ocurre si un thread quiere capturar una señal particular (pongamos por ejemplo, la pulsación de CTRL-C por parte del usuario), y otro thread espera esa señal para terminar el proceso? Esta situación puede darse si uno o más threads ejecutan procedimientos de una biblioteca estándar y otros están escritos por el usuario. Claramente, sus deseos son incompatibles. En general, la gestión de las señales es ya bastante difícil de gestionar en un entorno con un único thread. El pasar a un entorno multihilo no hace que sean más fáciles de tratar.

Un último problema introducido por los threads es la gestión de la pila. En muchos sistemas, cuando se desborda la pila de un proceso, el núcleo tan sólo proporciona de forma automática más pila a ese proceso. Cuando un proceso tiene múltiples threads, es necesario tener varias pilas. Si el núcleo desconoce la existencia de estas pilas, no puede hacer que crezcan automáticamente tras una falta de página. De hecho, puede que el núcleo ni siquiera se entere de que una falta de memoria esté relacionada con el crecimiento de una pila.

Ciertamente estos problemas no son insuperables, pero constituyen una prueba de que la introducción de threads en un sistema existente sin un rediseño substancial del sistema no puede funcionar de ninguna manera. Como mínimo es necesario redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas. Y todas esas cosas deben hacerse de tal manera que se mantenga la compatibilidad hacia atrás con los programas existentes para el caso particular de un proceso con un único thread. Para encontrar información adicional sobre los threads, véase Hauser y otros, 1993; y Marsh y otros, 1991.

2.3 COMUNICACIÓN ENTRE PROCESOS

Frecuentemente los procesos necesitan comunicarse con otros procesos. Por ejemplo, en una tubería del shell, la salida del primer proceso debe pasarse al segundo proceso, conservando los datos a su llegada el orden de partida. Por tanto existe una necesidad de comunicación entre los procesos, preferiblemente de una forma bien estructurada y sin utilizar interrupciones. En las siguientes secciones examinaremos algunas de las cuestiones relacionadas con esta **comunicación entre procesos** o **IPC** (*InterProcess Communication*).

Muy brevemente, hay tres cuestiones aquí. Ya hemos aludido a la primera cuestión anteriormente: cómo puede un proceso pasar información a otro. La segunda cuestión consiste en asegurar que dos o más procesos no se interfieran mientras realizan tareas críticas (pensemos en dos procesos que intentan apoderarse del último megabyte de memoria disponible). La tercera cuestión tiene que ver con el secuenciamiento correcto cuando existen dependencias: si el proceso *A* produce datos que el proceso *B* imprime, *B* tiene que esperar hasta que *A* produzca algún dato antes de comenzar a imprimir. Vamos a examinar estas tres cuestiones a partir de la siguiente sección.

Es importante señalar que dos de las esas tres cuestiones planteadas para los procesos se aplican exactamente igual a los threads. La primera – el paso de la información – resulta mucho más sencilla para los threads, al compartir el mismo espacio de direcciones (la comunicación entre threads que estén en diferentes espacios de direcciones puede tratarse como una comunicación entre procesos). Sin embargo, las otras dos cuestiones – la de evitar las interferencias y la de secuenciar correctamente – sí que se aplican exactamente igual a los threads. Tanto es así que existen los mismos problemas y se aplican las mismas soluciones. A continuación vamos a analizar el problema en el contexto de los procesos, pero deberá tenerse siempre presente que los mismos problemas y soluciones se aplican también a los threads.

2.3.1 Condiciones de Carrera

En algunos sistemas operativos, los procesos que trabajan juntos pueden compartir algún área común sobre la que cada proceso puede leer y escribir. La memoria compartida puede estar en memoria principal (posiblemente en una estructura de datos del núcleo) o puede ser un fichero compartido; la ubicación de la memoria compartida no cambia la naturaleza de la comunicación o los problemas que pueden surgir. Para ver cómo funciona la comunicación entre procesos en la práctica, vamos a considerar un ejemplo sencillo pero muy común: un spooler de impresión. Cuando un proceso quiere imprimir un archivo, pone el nombre del fichero en un **directorio especial de spool**. Otro proceso, el **demonio de la impresora**, comprueba periódicamente si hay ficheros para imprimir, y en caso de haberlos, los imprime para a continuación suprimir su nombre del directorio.

Imaginemos que nuestro directorio de spool tiene un número muy grande de entradas, numeradas 0, 1, 2, ..., cada una de ellas capaz de contener un nombre de fichero. Imaginemos también que se tienen dos variables compartidas, *out*, que apunta al siguiente fichero a imprimir, e *in*, que apunta a la siguiente entrada libre en el directorio. Estas dos variables pueden caber perfectamente en un fichero de dos palabras disponible para todos los procesos. En un cierto momento, las entradas de la 0 a la 3 están vacías (al haberse imprimido ya los ficheros correspondientes) y las entradas de la 4 a la 6 están ocupadas (con los nombres de los ficheros encolados para su impresión). Más o menos simultáneamente, los procesos *A* y *B* deciden cada uno de ellos mandar a la cola de impresión un fichero. Esta situación se muestra en la Figura 2-18.

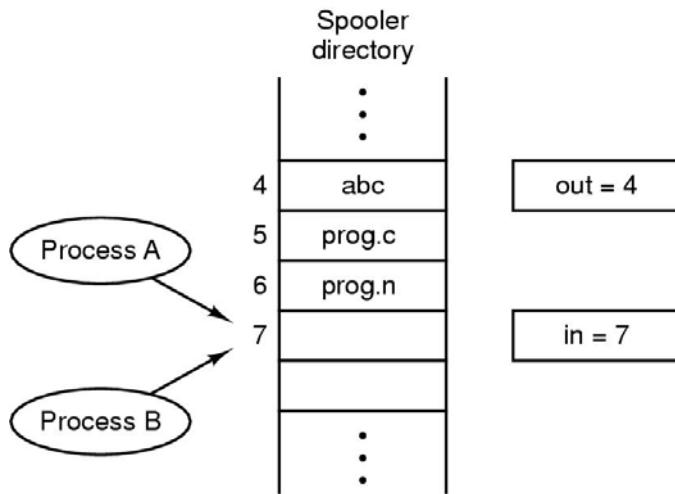


Figura 2-18. Dos procesos intentan acceder a memoria compartida al mismo tiempo.

Aplicando en esta situación la ley de Murphy (“Si algo puede ir mal, irá mal”) podría suceder lo siguiente. El proceso *A* lee *in* y guarda el valor 7 en una variable local denominada *next_free_slot*. Justo a continuación tiene lugar una interrupción del reloj y la CPU decide que el proceso *A* se ha ejecutado ya lo suficiente, por lo que conmuta al proceso *B*. El proceso *B* lee entonces *in*, obteniendo también un 7. Igualmente, *B* almacena ese valor en su variable local *next_free_slot*. En ese momento ambos procesos piensan que la siguiente entrada disponible es la 7.

Ahora el proceso *B* continúa ejecutándose. En un momento dado *B* guarda el nombre del fichero que quiere imprimir en la entrada 7 y actualiza *in* con el valor 8. Tras esa actualización *B* da por finalizada la operación de impresión y se pone a hacer otras cosas.

Eventualmente, el proceso *A* vuelve a pasar a ejecución, prosiguiendo por donde se quedó. El proceso *A* consulta *next_free_slot*, encuentra un 7, y escribe el nombre de su fichero en la entrada 7, borrando el nombre que había puesto previamente el proceso *B*. A continuación *B* calcula *next_free_slot* + 1, obteniendo un 8 que procede a guardar en la variable *in*.

Ahora el directorio de spool es internamente consistente, de manera que el demonio de impresión no nota nada que sea erróneo. Sin embargo el usuario del proceso *B* nunca llegará a recibir la impresión del fichero que encargó. Ese usuario puede permanecer esperando junto al cuarto de impresoras durante años, al estar completamente seguro de que ha ordenado correctamente la impresión de su fichero. La dura realidad es que ese fichero nunca se imprimirá. Situaciones como estas, donde dos o más procesos están leyendo o escribiendo sobre datos compartidos y el resultado final depende de quien se ejecute precisamente en cada momento, se denominan **condiciones de carrera**. La depuración de los programas que contienen condiciones de carrera no es nada divertida. Los resultados de todas las baterías de prueba realizadas pueden ser correctos, pero de vez en cuando puede sobrevenir un raro e inexplicable error.

2.3.2 Regiones Críticas

¿Cómo podemos evitar las condiciones de carrera? Aquí y en muchas otras situaciones donde se comparte memoria, ficheros, o cualquier otra cosa, la clave para evitar problemas es encontrar alguna forma de impedir que más de un proceso lea y escriba sobre el dato compartido al mismo tiempo. Dicho en otras palabras, lo que necesitamos es **exclusión mútua**, esto es,

alguna manera de asegurar que si un proceso está utilizando una variable compartida (o fichero compartido) los demás procesos estarán excluidos de hacer uso de esa misma variable. El problema anterior con el spooler de impresión ocurrió debido a que el proceso *B* comenzó a utilizar una de las variables compartidas antes de que el proceso *A* hubiera terminado con ella. La elección de las operaciones primitivas adecuadas para lograr la exclusión mutua es una de las principales cuestiones de diseño en cualquier sistema operativo, y un tema que vamos a examinar con mucho detalle en las secciones siguientes.

El problema de evitar las condiciones de carrera puede formularse también de manera abstracta. Parte del tiempo, un proceso está ocupado haciendo cálculos internos y otras cosas que no conducen a condiciones de carrera. Sin embargo, a veces un proceso tiene que acceder a memoria compartida o ficheros, o hacer otras tareas críticas que sí pueden conducir a condiciones de carrera. La parte del programa donde se accede a la memoria compartida se denomina la **región crítica** o **sección crítica**. Si pudiéramos organizar las cosas de forma que nunca estén dos procesos a la vez en sus regiones críticas, podríamos evitar las condiciones de carrera.

Aunque ese requerimiento evita las condiciones de carrera, no es suficiente para asegurar que los procesos concurrentes cooperan correcta y eficientemente utilizando datos compartidos. Deben cumplirse las cuatro condiciones siguientes para obtener una solución satisfactoria:

1. Ningún par de procesos pueden estar simultáneamente dentro de sus regiones críticas.
2. No debe hacerse ninguna suposición sobre la velocidad o el número de CPUs.
3. Ningún proceso fuera de su región crítica puede bloquear a otros procesos.
4. Ningún proceso deberá tener que esperar infinitamente para entrar en su región crítica.

En sentido abstracto, el comportamiento que se desea para los procesos es el que se muestra en la Figura 2-19. En ella el proceso *A* entra en su región crítica en el instante T_1 . Un poco después, en el instante T_2 , el proceso *B* intenta entrar en su región crítica pero no lo consigue debido a que otro proceso está ya en la región crítica y sólo podemos permitir que haya uno en cada momento. Consecuentemente, *B* tiene que suspenderse temporalmente hasta que *A* en el instante T_3 abandona la región crítica, permitiendo que *B* entre inmediatamente. Eventualmente *B* abandona (en T_4) la región crítica de manera que volvemos a estar en la situación original en la cual ningún proceso está dentro de la región crítica.

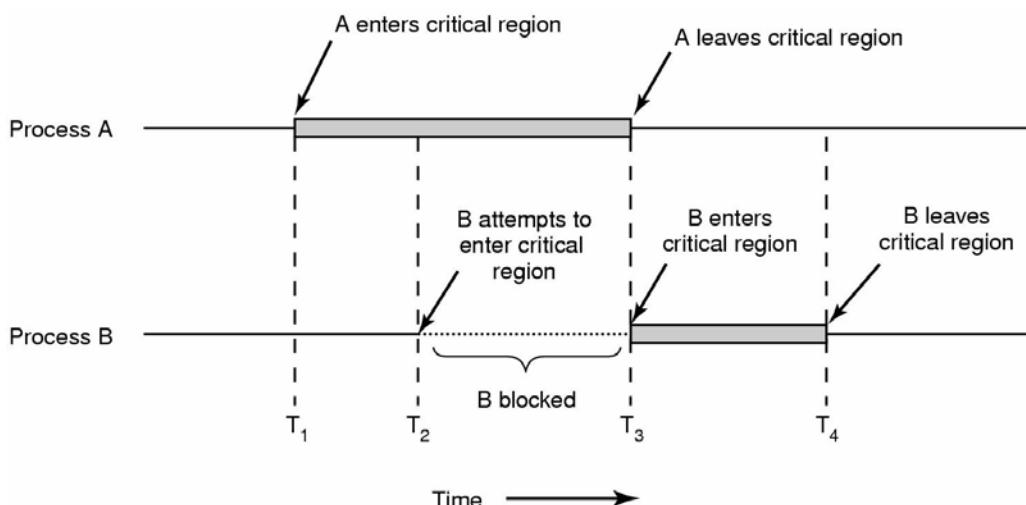


Figura 2-19. Exclusión mutua utilizando regiones críticas.

2.3.3 Exclusión Mutua con Espera Activa

En esta sección vamos a examinar varias propuestas para la consecución de la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso pueda entrar en su región crítica y provocar algún problema.

Inhibición de Interrupciones

La solución más sencilla es hacer que cada proceso inhiba todas las interrupciones nada más entrar en su región crítica y que las rehabilite nada más salir de ella. Con las interrupciones inhibidas, no puede ocurrir ninguna interrupción del reloj. Después de todo, la CPU sólo se conmuta a otro proceso como resultado de una interrupción del reloj o de otro dispositivo, y con las interrupciones inhibidas la CPU no puede comutarse a otro proceso. Entonces, una vez que un proceso ha inhibido las interrupciones, puede examinar y actualizar la memoria compartida sin temor a intromisiones de otros procesos.

En general este enfoque no es demasiado atractivo debido a que es muy peligroso conceder al usuario el poder de inhibir las interrupciones. Supongamos que uno de los procesos inhibe las interrupciones y nunca vuelve a activarlas. Eso podría significar el final del sistema. Además, si el sistema es un sistema multiprocesador, es decir cuenta con dos o más CPUs, la inhibición de las interrupciones por parte de un proceso afecta sólo a la CPU que ejecutó la instrucción **disable**. Por tanto, los demás procesos podrían perfectamente seguir ejecutándose en las demás CPUs, y en particular acceder también a la memoria compartida.

Por otra parte, frecuentemente es conveniente que el propio núcleo inhiba las interrupciones durante la ejecución de unas cuantas instrucciones mientras actualiza variables o listas. Por ejemplo, la llegada de una interrupción justo en el momento en el que la lista de procesos preparados estuviese en un estado inconsistente podría ocasionar condiciones de carrera verdaderamente peligrosas para el sistema. La conclusión es: la inhibición de interrupciones es a menudo una técnica útil dentro del propio sistema operativo, pero no es apropiada como un mecanismo de exclusión mutua general para los procesos de usuario.

Variables Cerrojo

Como segundo intento vamos a probar con una solución por software. Consideremos que disponemos de una variable compartida (variable cerrojo) cuyo valor es inicialmente 0. Cuando un proceso quiere entrar en su región crítica, primero examina la variable cerrojo. Si el cerrojo está a 0, el proceso lo pone a 1 y entra en la región crítica. Si el cerrojo ya estaba a 1, el proceso espera hasta que vuelva a valer 0. Entonces, un 0 significa que ningún proceso está en la región crítica, y un 1 significa que algún proceso está en su región crítica.

Desafortunadamente, esta idea contiene exactamente el mismo error fatal que ya vimos en el ejemplo del directorio del spooler de impresión. Supongamos que un proceso lee el cerrojo y observa que vale 0. Antes de que pueda poner el cerrojo a 1, se planifica otro proceso que pasa a ejecución y pone el cerrojo a 1. Cuando el primer proceso se ejecute de nuevo, pondrá a 1 también el cerrojo, y tendremos a dos procesos en sus regiones críticas al mismo tiempo.

Podríamos pensar en sortear este problema leyendo primero la variable cerrojo, y comprobando una segunda vez su valor justo antes de escribir el 1 en ella. Sin embargo esto tampoco funciona. Ahora la condición de carrera sucede si el segundo proceso modifica el cerrojo inmediatamente después de que el primer proceso termine hacer la segunda de las comprobaciones del cerrojo.

Alternancia Estricta

En la Figura 2-20 se muestra una tercera aproximación al problema de la exclusión mutua. Este fragmento de programa está escrito en C, igual que casi todos los demás de este libro. La elección de C se debe a que todos los sistemas operativos reales están siempre virtualmente escritos en C (o ocasionalmente en C++), y casi nunca en lenguajes como Java, Modula 3 o Pascal. C es potente, eficiente y predecible, siendo éstas características críticas para la escritura de sistemas operativos. Java, por ejemplo, no es predecible, debido a que se le puede agotar la memoria disponible en momentos críticos, necesitando invocar al recolector de basura en el instante más inoportuno. Esto no puede suceder en C debido a que no existe ningún recolector de basura en C. En Prechelt, 2000, se proporciona una comparación cuantitativa entre C, C++, Java y cuatro lenguajes más.

```

while (TRUE) {
    while (turno != 0 /* nada */;
        region_critica();
        turno = 1 ;
        region_no_critica() ;
}

```

(a)

```

while (TRUE) {
    while (turno != 1) /* nada */ ;
    region_critica();
    turno = 0 ;
    region_no_critica();
}

```

(b)

Figura 2-20. Una propuesta de solución al problema de la región crítica. (a) Proceso 0. (b) Proceso 1. En ambos casos, hay que tener el cuidado de fijarse en el punto y coma que indica la terminación de las instrucciones while.

En la Figura 2-20, la variable entera *turno*, inicializada a 0, indica a quien le corresponde el turno de entrar en la región crítica y examinar o actualizar la memoria compartida. En un primer momento, el proceso 0 inspecciona *turno*, observa que vale 0 y entra en su región crítica. El proceso 1 también se encuentra con que *turno* vale 0 y por lo tanto espera en un bucle vacío comprobando continuamente *turno* para ver cuando pasa a valer 1. La comprobación continua de una variable hasta que contenga algún valor determinado se denomina **espera activa** (*busy waiting*). Usualmente debe evitarse, ya que derrocha tiempo de CPU. La utilización de la espera activa sólo es apropiada cuando existan unas expectativas razonables de que la espera va a ser muy corta. Un cerrojo que utiliza espera activa se denomina un **spin lock**.

Cuando el proceso 0 abandona la región crítica. Pone la variable *turno* a 1 con el fin de permitir que el proceso 1 entre en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de forma que los dos procesos están en sus regiones no críticas, con la variable *turno* a 0. Ahora el proceso 0 ejecuta rápidamente una vuelta completa, saliendo de su región crítica y poniendo *turno* a 1. En este punto *turno* vale 1 y ambos procesos se encuentran de nuevo ejecutándose en sus regiones no críticas.

De repente el proceso 0 termina su región no crítica y vuelve a lo alto de su bucle. Desafortunadamente, ahora no se le permite entrar en su región crítica, debido a que *turno* es 1 y el proceso 1 está ocupado con su región no crítica. Así el proceso 0 queda atrapado en su bucle `while` hasta que el proceso 1 ponga turno a 0. Dicho de otra manera, el establecimiento de turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.

Esta situación viola la condición 3 que habíamos establecido: el proceso 0 está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al ejemplo del spooler de impresión discutido anteriormente, si asociamos la región crítica con la lectura y escritura en el directorio del spooler, al proceso 0 no se le permitiría imprimir otro fichero debido a que el proceso 1 está haciendo cualquier otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen de forma estricta en el acceso a sus regiones críticas, por ejemplo, en la impresión de ficheros mediante spooling. No se le permite a nadie que mande a la cola de impresión dos ficheros seguidos. Aunque este algoritmo consigue evitar todas las condiciones de carrera, no es realmente un candidato serio a ser la solución deseada, ya que viola la condición 3 exigida.

Solución de Peterson

Combinando la idea de establecer turnos con la idea de utilizar variables cerrojo y variables de aviso, el matemático holandés, T. Dekker, fue el primero en diseñar una solución por software para el problema de la exclusión mutua que no requiere alternancia estricta. Para una discusión del algoritmo de Dekker, ver (Dijkstra, 1965).

En 1981, G.L. Peterson descubrió una forma mucho más sencilla de conseguir la exclusión mutua, dejando la solución de Dekker obsoleta. El algoritmo de Peterson se muestra en la Figura 2-21. Este algoritmo consiste en dos procedimientos escritos en ANSI C, lo que significa que hay que declarar los prototipos de todas las funciones. Sin embargo, para ahorrar espacio, no vamos a mostrar los prototipos ni en este ni en los ejemplos posteriores.

```
#define FALSE 0
#define TRUE 1
#define N      2           /* numero de procesos */

int turno ;                /* ¿de quien es el turno? */
int interesado[N] ;        /* array inicializado a 0's (FALSE's) */

void entrar_en_region ( int proceso )    /* proceso vale 0 o 1 */
{
    int otro ;                /* numero del otro proceso */
    otro = 1 - proceso ;     /* el complementario del proceso */
    interesado[proceso] = TRUE ;   /* indica que esta interesado */
    turno = proceso ;         /* intenta obtener el turno */
    while (turno == proceso
          && interesado[otro] == TRUE) /* instrucción vacia */;
}

void abandonar_region ( int proceso )    /* proceso: el que esta saliendo */
{
    interesado[proceso] = FALSE ;       /* indica la salida de la region critica */
}
```

Figura 2-21. Solución de Peterson para conseguir exclusión mutua.

Antes de utilizar las variables compartidas (es decir, antes de entrar en su región crítica), cada proceso invoca *entrar_en_region* con su propio número de proceso, 0 o 1, como parámetro. Esta llamada puede provocar su espera, si es necesario, hasta que sea seguro entrar. Cuando termine de utilizar las variables compartidas, el proceso invoca *abandonar_region* para indicar que ha terminado y para permitir a los demás procesos entrar, si así lo desean.

Vamos a ver como funciona esta solución. Inicialmente no hay ningún proceso en su región crítica. Ahora el proceso 0 llama a *entrar_en_region*. Indica su interés poniendo a TRUE su componente del array *interesado* y pone el valor 0 en *turno*. Ya que el proceso 1 no está interesado, *entrar_en_region* retorna inmediatamente. Si el proceso 1 llama ahora a *entrar_en_region*, quedará atrapado en esa función hasta que *interesado[0]* pase a valer *FALSE*, algo que sólo ocurre cuando el proceso 0 llama a *abandonar_region* para salir de la región crítica.

Consideremos ahora el caso en que ambos procesos llaman a *entrar_en_region* casi simultáneamente. Ambos deben asignar su número de proceso a la variable *turno*. La asignación que va a prevalecer es la del último proceso que realice la asignación; siendo anulada la primera de las asignaciones por la sobreescritura. Supongamos que el proceso 1 es el último que hace la asignación, de manera que *turno* es 1. Cuando ambos procesos entran en la sentencia *while*, el proceso 0 la ejecuta cero veces y entra en su región crítica. El proceso 1 se pone a dar vueltas en el bucle y no entra en su región crítica hasta que el proceso 0 sale de su región crítica.

La Instrucción TSL (Test and Set Lock)

Vamos a examinar ahora una propuesta de solución que requiere un poco de ayuda del hardware. Numerosos ordenadores, especialmente los que están diseñados con la idea en mente de que dispongan de múltiples procesadores, cuentan con la instrucción

TSL RX,LOCK

(Test and Set Lock) que funciona de la siguiente manera. Lee el contenido de la palabra de memoria *lock* en el registro RX y a continuación escribe un valor distinto de cero en la dirección de memoria *lock*. Se garantiza que la operación de lectura de la palabra y su escritura son operaciones indivisibles – ningún otro procesador puede acceder a la palabra de memoria hasta que se termine la instrucción. La CPU que ejecuta la instrucción TSL bloquea el bus de memoria para prohibir al resto de CPUs acceder a la memoria hasta que termine la instrucción.

Para utilizar la instrucción TSL, debemos utilizar una variable compartida, *lock*, para coordinar el acceso a la memoria compartida. Cuando *lock* vale 0, cualquier proceso puede ponerla a 1 utilizando la instrucción TSL y luego leer o escribir en la memoria compartida. Una vez hecho esto, el proceso vuelve a poner *lock* a 0 utilizando una instrucción move ordinaria.

¿Cómo puede utilizarse esta instrucción para prevenir que dos procesos entren simultáneamente en sus regiones críticas? La solución se muestra en la Figura 2-22, en la forma de una subrutina de cuatro instrucciones escrita en un lenguaje ensamblador ficticio (pero típico). La primera instrucción copia el antiguo valor de *lock* al registro, poniendo a continuación *lock* a 1. Luego el antiguo valor de *lock* se compara con 0. Si no es cero, el *lock* fue puesto a 1 previamente, de manera que el programa debe retroceder al principio y comprobar de nuevo su valor. Más pronto o más tarde volverá a valer 0 (cuando el proceso que está actualmente en su región crítica salga de ella), y la subrutina retornará, con el cerrojo *lock* echado. Borrar el cerrojo es sencillo. El programa se limita a poner un 0 en *lock*. No es necesaria ninguna instrucción especial.

```

entrar_en_region:
    TSL RREGISTER,LOCK      ; copia en el registro el cerrojo y lo pone a 1
    CMP REGISTER,#0          ; ¿valía 0 el cerrojo?
    JNE entrar_en_region    ; si no valía 0, lo intentamos de nuevo
    RET                      ; retorno al punto de llamada; entra en la r.c.

abandonar_region:
    MOVE LOCK,#0             ; poner nuevamente el 0 en lock
    RET                      ; retorno al punto de llamada

```

Figura 2-22. Entrada y salida de una región crítica utilizando la instrucción TSL.

Resulta trivial ahora una solución al problema de la región crítica. Un proceso antes de entrar en su región crítica invoca a *entrar_en_region*, haciendo espera activa hasta quede libre

el cerrojo; entonces echa el cerrojo y retorna. Después de ejecutar la región crítica el proceso llama a *abandonar_region*, que pone un 0 en *lock*. Como con todas las soluciones que se basan en regiones críticas, el proceso debe llamar a *entrar_en_región* y *abandonar_region* en los momentos adecuados para que el método funcione. Si un proceso hace trampa, la exclusión mutua puede fallar.

2.3.4 Sleep y Wakeup

Tanto la solución de Peterson como la solución con TSL son correctas, pero ambas tienen el defecto de requerir espera activa. En esencia, lo que estas dos soluciones hacen es lo siguiente: cuando un proceso quiere entrar en su región crítica, comprueba si se le permite la entrada. Si no es así, el proceso se mete en un bucle vacío esperando hasta que sí se le permita entrar.

Este método no sólo gasta tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos un ordenador con dos procesos, *H* con alta prioridad y *L* con baja prioridad. Las reglas de planificación son tales que *H* pasa a ejecución inmediatamente siempre que se encuentre en el estado de preparado. En un cierto momento, estando *L* en su región crítica, *H* pasa al estado preparado (por ejemplo, debido a que se completa una operación de E/S que lo mantenía bloqueado). De inmediato *H* comienza la espera activa, pero ya que *L* nunca se planifica mientras *H* esté ejecutándose, *L* nunca tendrá la oportunidad de abandonar su región crítica, con lo que *H* quedará para siempre dando vueltas al bucle de espera activa. A esta situación se la denomina a veces como el **problema de la inversión de las prioridades**.

A continuación vamos a estudiar algunas de las primitivas de comunicación entre procesos que bloquean a los procesos en vez de consumir tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las primitivas más simples es el par de llamadas al sistema *sleep* y *wakeup*. *Sleep* es una llamada al sistema que provoca que el proceso que la invoca se bloquee, esto es, se suspenda hasta que otro proceso lo despierte. La llamada *wakeup* tiene un parámetro, que es el proceso a ser despertado. Alternativamente, tanto *sleep* como *wakeup* podrían tener ambas un parámetro, una posición de memoria utilizada para ajustar los *sleeps* con los *wakeups*.

El Problema del Productor-Consumidor

Como un ejemplo de cómo pueden utilizarse estas primitivas, vamos a considerar el problema del **productor-consumidor** (también conocido como el problema del **búfer acotado**). Dos procesos comparten un búfer de tamaño fijo común. Uno de ellos, el productor, mete información en el búfer, y el otro, el consumidor, la saca. Es posible generalizar el problema para tener *m* productores y *n* consumidores, pero sólo vamos a considerar el caso de un productor y un consumidor, ya que bajo esa suposición se simplifica la solución.

Surge un problema cuando el productor quiere meter un nuevo elemento en el búfer encontrándose éste ya completamente lleno. La solución es que el productor se duerma, no despertándose hasta que el consumidor saque uno o más elementos del búfer. Similarmente, si el consumidor quiere sacar un elemento del búfer y ve que el búfer está vacío, debe dormirse hasta que el productor meta algo en el búfer y lo despierte.

Este enfoque parece muy sencillo, pero conduce al mismo tipo de condiciones de carrera que vimos anteriormente en el ejemplo del spooler de impresión. Para controlar el número de elementos en el búfer, vamos a necesitar una variable, *contador*. Si el número máximo de elementos que puede contener el búfer es *N*, el código del productor debe comenzar comprobando si el valor de *contador* es *N*. Si lo es, el productor debe dormirse; si no lo es, el productor puede añadir un nuevo elemento al búfer e incrementar *contador*.

El código del consumidor es similar: primero comprueba si el valor de *contador* es 0. Si lo es se irá a dormir; si no es cero, sacará un elemento del búfer y decrementará el contador. Cada uno de los procesos comprueba también si puede despertar al otro, y en ese caso, lo despierta. El código tanto del productor como del consumidor se muestra en la Figura 2-23.

```
#define N 100                                /* numero de entradas en el bufer */
int contador = 0 ;                            /* numero de elementos en el bufer */

void productor (void)
{
    int elemento ;

    while (TRUE) {                                /* repetir siempre */
        elemento = producir_elemento( ) ;          /* generar el siguiente elemento */
        if (contador == N) sleep( ) ;                /* si el buffer está lleno, dormirse */
        meter_elemento(elemento) ;                  /* meter el elemento en el bufer */
        contador = contador + 1 ;                   /* incrementar el contador de elementos */
        if (contador == 1) wakeup(consumidor) ;       /* ¿estaba el bufer vacío? */
    }
}

void consumidor ( void )
{
    int elemento ;

    while (TRUE) {                                /* repetir siempre */
        if (contador == 0) sleep( ) ;                /* si el bufer está vacío, dormirse */
        elemento = sacar_elemento( ) ;              /* sacar el elemento del bufer */
        contador = contador - 1 ;                   /* decrementar el contador de elementos */
        if (contador == N - 1) wakeup(productor) ;    /* ¿estaba el bufer lleno? */
        consumir_elemento(elemento) ;
    }
}
```

Figura 2-23. El problema del productor-consumidor con una condición de carrera fatal.

Para expresar llamadas al sistema tales como **sleep** y **wakeup** en C, vamos a mostrarlas como llamadas a procedimientos de biblioteca. No son parte de la librería estándar de C, pero presumiblemente podrían estar disponibles en cualquier sistema que realmente contase con estas llamadas al sistema. Los procedimientos *meter_elemento* y *sacar_elemento*, que no se muestran, se encargan de los pormenores de meter elementos en el búfer y sacar elementos del búfer.

Vamos a mostrar ahora la condición de carrera. Ésta puede ocurrir debido a que no se restringe el acceso a *contador*. Así puede ocurrir la siguiente situación en la que el búfer está vacío y el consumidor acaba de leer *contador* para ver si es 0. En ese instante el planificador decide detener temporalmente la ejecución del consumidor y comenzar a ejecutar el productor. El productor mete un primer elemento en el búfer, incrementa *contador* y observa que vale ahora 1. De acuerdo con eso, el productor razona que el consumidor debe de estar durmiendo, por lo que invoca a *wakeup* con el fin de despertar al consumidor.

Desafortunadamente, ya que el consumidor lógicamente no está todavía dormido, la señal del que intenta despertarlo se pierde. Cuando el consumidor vuelve a ejecutarse, comprobará el valor de *contador* que leyó previamente, encontrando que su valor es 0, por lo que procederá ahora ya sí a dormirse. Más pronto o más tarde el productor, a base de meter nuevos elementos, terminará de llenar el búfer y también se dormirá. Llegados a este punto ambos procesos permanecerán durmiendo para siempre.

Aquí, la esencia del problema es la pérdida de una señal enviada para despertar a un proceso que no estaba (todavía) dormido. Si dicha señal no se hubiera perdido, todo hubiera funcionado bien. Un parche rápido sería modificar las reglas añadiendo un bit de espera por la señal que despierta al proceso (*wakeup waiting bit*). Este bit se activa cuando se envía una señal para despertar a un proceso que todavía está despierto. Posteriormente, cuando el proceso intente dormirse, si encuentra ese bit activo, simplemente lo desactiva permaneciendo despierto. Este bit de espera es como una especie de hucha que guarda las señales que se envían para despertar a un proceso.

Mientras que el bit de espera anterior salva la situación en este ejemplo sencillo, es fácil construir ejemplos con tres o más procesos en los cuales un único bit de este tipo es insuficiente. Podemos hacer otro parche y añadir un segundo bit, o posiblemente 8 o 32, pero en principio el problema seguiría ahí.

2.3.5 Semáforos

Esa era la situación en el año 1965, cuando E.W.Dijkstra (1965) sugirió utilizar una variable entera para contar el número de señales enviadas para despertar un proceso guardadas para su uso futuro. En su propuesta introdujo un nuevo tipo de variable, denominado **semáforo**. El valor de un semáforo puede ser 0, indicando que no se ha guardado ninguna señal, o algún valor positivo de acuerdo con el número de señales pendientes para despertar al proceso.

Dijkstra propuso establecer dos operaciones sobre los semáforos, **bajar** y **subir** (las cuales generalizan las operaciones **sleep** y **wakeup**, respectivamente). La operación **bajar** sobre un semáforo comprueba si el valor es mayor que 0. Si lo es, simplemente decrementa el valor (es decir utiliza una de las señales guardadas). Si el valor es 0, el proceso procede a dormirse sin completar la operación **bajar** por el momento. La comprobación del valor del semáforo, su modificación y la posible acción de dormirse, se realizan como una única **acción atómica** indivisible. Está garantizado que una vez que comienza una operación sobre un semáforo, ningún otro proceso puede acceder al semáforo hasta que la operación se completa o hasta que el proceso se bloquea. Esta atomicidad es absolutamente esencial para resolver los problemas de sincronización y evitar que se produzcan condiciones de carrera.

La operación **subir** incrementa el valor del semáforo al cual se aplica. Si uno o más procesos estuviesen dormidos sobre ese semáforo, incapaces de completar una anterior operación de **bajar**, el sistema elige a uno de ellos (por ejemplo de forma aleatoria) y se le permite completar esa operación **bajar** pendiente. Entonces, después de ejecutarse un **subir** sobre un semáforo con procesos dormidos en él, el semáforo sigue valiendo 0, pero habrá un proceso menos, dormido en él. La operación de incrementar el semáforo y despertar un proceso es también indivisible. La operación de **subir** nunca bloquea al proceso que la ejecuta, de la misma forma que en el modelo anterior nunca se bloquea un proceso ejecutando una operación **wakeup**.

Como comentario, en el artículo original de Dijkstra, se utilizan los nombres P y V en vez de **bajar** y **subir**, respectivamente, pero ya que estos nombres no tienen ningún significado nemotécnico más que para los holandeses (e incluso para ellos esos nombres resultan también poco sugerentes) hemos preferido utilizar en su lugar los nombres **bajar** y **subir**. Los semáforos se introdujeron por primera vez en el lenguaje de programación Algol 68.

Resolución del Problema del Productor-Consumidor Utilizando Semáforos

Los semáforos resuelven el problema de la pérdida de señales para desbloquear un proceso, como se muestra en la Figura 2-24. Es esencial que los semáforos estén implementados de forma indivisible. La manera normal es implementar **bajar** y **subir** como llamadas al sistema, encargándose el sistema operativo de inhibir brevemente todas las interrupciones mientras comprueba el valor del semáforo, lo actualiza y bloquea el proceso, si es necesario. Como todas estas acciones requieren tan solo unas pocas instrucciones, no se provoca ningún daño al sistema inhibiendo las interrupciones durante ese corto lapso de tiempo. Si se están utilizando varias CPUs, es necesario proteger cada semáforo mediante una variable cerrojo, utilizando la instrucción TSL para asegurar que tan sólo una CPU examina a la vez el semáforo. Asegúrese de entender que la utilización de TSL para prevenir que varias CPUs accedan al semáforo al mismo tiempo es muy diferente de la utilización de espera activa por parte del productor o del consumidor esperando a que el otro proceso vacíe o llene el búfer. La operación sobre el semáforo va a durar tan sólo unos pocos microsegundos, mientras que el productor o el consumidor pueden tardar intervalos de tiempo arbitrariamente largos.

Esta solución utiliza tres semáforos: uno denominado *entrada_llena* para contar el número de entradas que están llenas, uno denominado *entrada_vacia* para contar el número de entradas vacías, y otro denominado *mutex* (exclusión mutua; *mutual exclusion*) para asegurar que el productor y el consumidor no acceden al búfer al mismo tiempo. Inicialmente *entrada_llena* vale 0, *entrada_vacia* es igual al número total de entradas que tiene el búfer, y *mutex* vale 1. Los semáforos que se inicializan con el valor 1 y que se utilizan para asegurar que tan solo un proceso pueda entrar en su región crítica en cada momento, se denominan **semáforos binarios**. La exclusión mutua está garantizada si cada proceso hace un **bajar** justo antes de entrar en su región crítica, y un **subir** justo después de abandonarla.

Ahora que ya tenemos a nuestra disposición una buena primitiva de comunicación entre procesos, vamos a retroceder y echar una mirada de nuevo a la secuencia de una interrupción mostrada en la Figura 2-5. La forma natural de ocultar las interrupciones en un sistema que cuenta con semáforos, es asociar a cada dispositivo de E/S un semáforo inicializado a 0. Justo después de poner en marcha un dispositivo de E/S el proceso que lo controla hace un **bajar** sobre el semáforo asociado, bloqueándose de forma inmediata. Cuando llega la interrupción, la rutina de tratamiento de la interrupción hace un **subir** sobre el semáforo asociado, lo que provoca que el proceso correspondiente quede listo (preparado) para ejecutarse de nuevo. En este modelo, el paso 5 de la Figura 2-5 consiste en realizar un **subir** sobre el semáforo del dispositivo, de forma que en el paso 6 el planificador sea capaz de ejecutar el controlador del dispositivo. Por supuesto, en caso de que ahora haya varios procesos preparados, el planificador puede elegir ejecutar a continuación un proceso distinto de mayor importancia. Posteriormente en este capítulo vamos a ver algunos de los algoritmos utilizados en la planificación de los procesos.

```

#define N 100                                /* numero total de entradas en el bufer */
typedef int semaphore;                      /* semaforo como tipo especial de int */
semaphore mutex = 1;                        /* controla el acceso a la region critica */
semaphore entrada_vacia = N;                /* numero de entradas vacias del bufer */
semaphore entrada_llena = 0;                 /* numero de entradas ocupadas del b. */

void productor ( void )
{
    int elemento;

    while (TRUE) {
        elemento = producir_elemento( );
        bajar(&entrada_vacia);
        bajar(&mutex);
        meter_elemento(elemento);
        subir(&mutex);
        subir(&entrada_llena);
    }
}

void consumidor ( void )
{
    int elemento;

    while (TRUE) {
        bajar(&entrada_llena);           /* bucle infinito */
        bajar(&mutex);                 /* decrementar el contador de ocupadas */
        elemento = sacar_elemento(elemento); /* entrar en la region critica */
        /* sacar un elemento del bufer */
        subir(&mutex);                 /* abandonar la region critica */
        subir(&entrada_vacia);          /* incrementar cont. de entradas vacias */
        consumir_elemento(elemento);    /* hacer algo con el elemento */
    }
}

```

Figura 2-24. El problema del productor-consumidor utilizando semáforos.

En el ejemplo de la Figura 2-24, realmente hemos utilizado los semáforos de dos formas muy distintas. La diferencia es lo suficientemente importante para hacerla explícita. El semáforo *mutex* se utiliza para conseguir la exclusión mutua. Está diseñado para garantizar que solamente un proceso esté en cada momento leyendo o escribiendo en el búfer y sus variables asociadas. Esta exclusión mutua es necesaria para prevenir el caos. Vamos a estudiar más en detalle la exclusión mutua y cómo conseguirla, en la sección siguiente.

El otro uso de los semáforos es la **sincronización**. Los semáforos *entrada_llena* y *entrada_vacia* son necesarios para garantizar que ocurran o no ciertas secuencias de sucesos. En este caso concreto, esos semáforos aseguran que el productor detiene su ejecución cuando el búfer está lleno, y que el consumidor detiene su ejecución cuando el búfer está vacío. Este uso es diferente de la exclusión mutua.

2.3.6 Variables de Exclusión Mútua

Cuando no es necesaria la capacidad del semáforo para contar, es frecuente el uso de una versión simplificada de los semáforos, que denominaremos **variables de exclusión mutua** (variables *mutex*). Estas variables sólo son apropiadas para conseguir la exclusión mutua en el

acceso a algún recurso compartido o fragmento de código. Se pueden implementar de una forma sencilla y eficiente, lo que las hace especialmente útiles en paquetes de threads que están implementados enteramente en el espacio del usuario.

Una variable de exclusión mutua es una variable que puede estar en dos estados: desbloqueada o bloqueada. Consecuentemente, sólo es necesario 1 bit para representarla, aunque en la práctica se utiliza a menudo un entero, con 0 significando desbloqueada y todos los demás valores significando bloqueada. Hay dos procedimientos que se utilizan con las variables de exclusión mutua. Cuando un thread (o proceso) necesita acceder a una región crítica, llama a *mutex_lock*. Si la variable de exclusión mutua está actualmente desbloqueada (lo que significa que la región crítica está disponible), la llamada consigue su objetivo y el thread que la invoca es libre para entrar en la región crítica.

Por otro lado, si la variable de exclusión mutua está bloqueada, el thread que hace la llamada se bloquea hasta que el thread que está en la región crítica la termine y llame a *mutex_unlock*. Si hay varios threads que están bloqueados en la variable de exclusión mutua, se elige aleatoriamente uno de ellos para que se apropie de la región crítica.

Debido a que las variables de exclusión mutua son tan sencillas, pueden implementarse fácilmente en el espacio de usuario con tal de disponer de la instrucción TSL. El código correspondiente a *mutex_lock* y a *mutex_unlock* para utilizarlo con un paquete de threads a nivel de usuario se muestra en la Figura 2-25.

```

mutex_lock:
    TSL REGISTER,MUTEX ; copiar mutex al registro y poner mutex a 1
    CMP REGISTER,#0      ; ¿valía mutex cero?
    JZE ok               ; si era cero, mutex no estaba bloqueada,
                          ; por lo que se retorna
    CALL thread_yield    ; mutex está ocupada; se planifica otro thread
    JMP mutex_lock       ; se intenta de nuevo luego
ok:   RET                ; retorno al punto de llamada;
                          ; se entra en la región crítica

mutex_unlock:
    MOVE MUTEX,#0        ; ponemos un 0 en mutex
    RET                  ; retorno al punto de llamada

```

Figura 2-25. Implementación de *mutex_lock* y *mutex_unlock*.

El código de *mutex_lock* es similar al código de *entrar_en_region* de la Figura 2-22 pero con una diferencia crucial. Cuando *entrar_en_region* falla intentando entrar en la región crítica, se sigue comprobando el cerrojo de forma repetida (espera activa). Eventualmente, llega una interrupción del reloj y se planifica algún otro proceso para su ejecución. Más pronto o más tarde el proceso que posea el cerrojo conseguirá ejecutarse y lo desbloqueará.

Con los threads, la situación es diferente debido a que no existe ningún reloj que detenga a los threads que se han ejecutado durante demasiado tiempo. Consecuentemente, un thread que intenta apropiarse de un cerrojo mediante espera activa, permanecerá por siempre en el bucle de espera y nunca conseguirá el cerrojo debido a que no está permitiendo que se ejecute ningún otro thread y que se libere el cerrojo.

Aquí es donde se presenta la diferencia entre *entrar_en_region* y *mutex_lock*. Cuando *mutex_lock* falla intentando apropiarse del cerrojo, llama a *thread_yield* para ceder la CPU a otro thread. Consecuentemente no existe espera activa. La próxima vez que se ejecute el thread, volverá a comprobar el cerrojo.

Ya que *thread_yield* es precisamente una llamada al thread planificador en el espacio de usuario, se ejecuta muy rápidamente. Como consecuencia, ni *mutex_lock* ni *mutex_unlock* requieren ninguna llamada al núcleo. De esta manera, mediante el uso de esas dos primitivas los threads a nivel de usuario pueden sincronizarse enteramente dentro del espacio de usuario, utilizando procedimientos que requieren tan sólo un puñado de instrucciones.

El sistema de variables de exclusión mutua que hemos descrito anteriormente es un conjunto de llamadas supersimplificado. Como sucede con todo el software, siempre existe una demanda de funciones más especializadas y sofisticadas, y las primitivas de sincronización no son una excepción. Por ejemplo, a veces un paquete de threads ofrece una llamada *mutex_trylock* que o bien adquiere el cerrojo o bien devuelve un código de fallo, pero que no se bloquea. Esta llamada dota al thread de la flexibilidad de poder decidir qué hacer a continuación cuando hay alguna alternativa a la pura y simple espera.

Hasta ahora hemos pasado por alto una cuestión que es necesario al menos comentar explícitamente. Con un paquete de threads en el espacio de usuario no existe ningún problema con que varios threads accedan simultáneamente a la misma variable de exclusión mutua ya que todos los threads operan en un espacio de direcciones común. Sin embargo, con la mayoría de las soluciones anteriores, tales como el algoritmo de Peterson y los semáforos, existe una suposición tácita de que varios procesos tienen acceso a al menos alguna memoria compartida, quizás una única palabra, pero algo. ¿Si los procesos tienen espacios de direcciones disjuntos, como hemos dicho constantemente, como es posible que puedan compartir la variable turno del algoritmo de Peterson, o semáforos, o todo un búfer común?

Hay dos respuestas. La primera es que algunas de las estructuras de datos compartidas, tales como los semáforos, pueden almacenarse en el núcleo accediéndose a ellas a través de llamadas al sistema. Este enfoque elimina completamente el problema. La segunda de las respuestas es que la mayoría de los sistemas operativos modernos (incluyendo UNIX y Windows) ofrecen a los procesos alguna forma de compartir alguna porción de sus espacios de direcciones con otros procesos. De esta manera pueden compartirse los búferes y otras estructuras de datos. En el peor de los casos, en el cual nada de lo anterior es posible, siempre puede utilizarse un fichero compartido.

Si dos o más procesos comparten la mayoría o incluso todo su espacio de direcciones, la distinción entre procesos y threads queda un poco difuminada pero sin embargo sigue presente. Dos procesos que comparten un espacio de direcciones común todavía tienen diferentes ficheros abiertos, alarmas de temporización y otras propiedades propias de los procesos, mientras que los threads dentro de un mismo proceso comparten todas esas características. Lo que sí es siempre cierto es que varios procesos que comparten un espacio de direcciones común nunca consiguen la eficiencia de los threads a nivel del usuario ya que el núcleo está profundamente involucrado en su gestión.

2.3.7 Monitores

¿Verdad que la comunicación entre procesos utilizando semáforos parece fácil? De ninguna manera. Veamos en detalle cuál es el orden de los **bajar** antes de meter o sacar elementos del búfer en la Figura 2-24. Supongamos que invertimos el orden de los dos **bajar** en el código del productor, de manera que *mutex* se decrementa antes de *entrada_vacia* en vez de después de ella. Si el búfer estuviera completamente lleno, el productor se bloquearía, con *mutex* valiendo 0. Consecuentemente, la siguiente vez que el consumidor intente acceder al búfer, hará un **bajar** sobre *mutex*, valiendo ahora 0, y también se bloqueará. Ambos procesos quedarían bloqueados para siempre y nunca podrían hacer más trabajo. Esta desafortunada situación se denomina un interbloqueo (*deadlock*). Vamos a estudiar los interbloqueos en detalle en el Capítulo 3.

Hemos mostrado este problema para poner de manifiesto lo cuidadosos que debemos ser cuando utilizamos semáforos. Un sutil error y todo puede quedar paralizado irreversiblemente. Es parecido a programar en lenguaje ensamblador, sólo que peor, debido a que los errores son condiciones de carrera, interbloqueos y otras formas de comportamiento erróneo impredecible e irreproducible.

Para hacer más fácil escribir programas correctos, Hoare (1974) y Brinch Hansen (1975) propusieron una primitiva de sincronización de alto nivel denominada **monitor**. Sus propuestas difieren ligeramente, como describiremos después. Un monitor es una colección de procedimientos, variables y estructuras de datos que están todos agrupados juntos en un tipo especial de módulo o paquete. Los procesos pueden llamar a los procedimientos de un monitor siempre que quieran, pero no se les permite acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor. La Figura 2-26 ilustra un monitor escrito en un lenguaje de programación imaginario, Pidgin Pascal. (Pidgin: lengua basada en otras, que tiene una gramática y un vocabulario muy restringido, en definitiva Cutre-Pascal).

```
monitor ejemplo
    i : integer ;
    c : condition ;

    procedure productor () ;
    ...
    end ;

    procedure consumidor () ;
    ...
    end ;

end monitor ;
```

Figura 2-26. Un monitor.

Los monitores tienen una importante propiedad que los hace útiles para conseguir exclusión mutua: en cualquier instante solamente un proceso puede estar activo dentro del monitor. Los monitores son construcciones del lenguaje, por lo que el compilador sabe que son especiales, de manera que puede tratar las llamadas a los procedimientos del monitor de forma diferente que a otras llamadas a procedimientos normales. En la mayoría de los casos, cuando un proceso llama a un procedimiento de un monitor, las primeras instrucciones del procedimiento comprueban si cualquier otro proceso está actualmente activo dentro del monitor. En ese caso, el proceso que hizo la llamada debe suspenderse hasta que el otro proceso abandone el monitor. Si ningún otro proceso está utilizando el monitor, el proceso que hizo la llamada puede entrar inmediatamente.

Corresponde al compilador implementar la exclusión mutua sobre las entradas al monitor. La forma más común de implementarla es utilizando una variable de exclusión mutua o un semáforo binario. Debido a que es el compilador y no el programador el que organiza las cosas para conseguir la exclusión mutua, resulta mucho más difícil que se produzcan errores. En cualquier caso, la persona que escribe el monitor no tiene que preocuparse de cómo consigue asegurar la exclusión mutua el compilador dentro del monitor. Es suficiente con que sepa que metiendo todas las regiones críticas en procedimientos del monitor, nunca habrá dos procesos que ejecuten sus regiones críticas a la vez.

Aunque, como hemos visto anteriormente, los monitores proporcionan una manera fácil de conseguir la exclusión mutua, eso no es suficiente. Necesitamos también un medio para poder bloquear a los procesos cuando no deban proseguir su ejecución. En el problema del productor-consumidor es muy fácil ubicar todas las comprobaciones de búfer lleno y búfer vacío en procedimientos de monitor, ¿pero cómo podemos bloquear al productor cuando se encuentra que el búfer está lleno?

La solución consiste en la introducción de **variables de condición**, junto con dos operaciones sobre ellas, **wait** y **signal**. Cuando un procedimiento del monitor descubre que no puede continuar (por ejemplo, cuando el productor encuentra que el búfer está lleno), debe llevar a cabo una operación **wait** sobre alguna variable de condición, pongamos que se llame *lleno*. Esta acción provoca que el proceso que la invoca se bloquee, permitiendo también que algún proceso al que se le hubiese impedido previamente la entrada en el monitor entre ahora.

Este otro proceso, por ejemplo el consumidor, puede despertar a su compañero que está dormido realizando una operación **signal** sobre la variable de condición sobre la que está esperando su compañero. Para evitar tener dos procesos activos al mismo tiempo en el monitor, necesitamos una regla que nos diga qué ocurre después de un **signal**. Hoare propuso dejar que el proceso recién despertado se ejecute inmediatamente, suspendiéndose el otro proceso. Brinch Hansen propuso solucionar diplomáticamente el problema imponiendo que cuando un proceso realice un **signal** *debe* salir del monitor inmediatamente. En otras palabras, una instrucción **signal** sólo puede aparecer como una última instrucción que se ejecute en un procedimiento del monitor. Vamos a adoptar la propuesta de Brinch Hansen debido a que es más sencilla tanto desde el punto de vista conceptual como en lo relativo a su implementación. Si se realiza un **signal** sobre una variable de condición sobre la que están esperando varios procesos, sólo uno de ellos revive, siendo el planificador del sistema el encargado de determinar ese proceso.

Como comentario, existe también una tercera solución no propuesta ni por Hoare ni por Brinch Hansen. Ésta consistiría en dejar que el proceso que realiza el **signal** continúe ejecutándose, permitiéndose nuevamente la ejecución del proceso que espera sólo después de que el proceso que hizo el **signal** abandone el monitor.

Las variables de condición no son contadores, ya que no acumulan las señales para un uso posterior como hacen los semáforos. Entonces si una variable de condición recibe una señal sin que haya ningún proceso esperándola, la señal se pierde para siempre. En otras palabras el **wait** debe ocurrir antes que el **signal**. Esta regla simplifica mucho la implementación, y en la práctica no representa ningún problema ya que, si es necesario, es fácil seguir la pista del estado de cada proceso utilizando variables adicionales. Antes de llevar a cabo un **signal**, el proceso puede ver si esa operación es necesaria, o no, inspeccionando esas variables.

En la Figura 2-27 se muestra el esqueleto de la solución del productor-consumidor utilizando monitores expresada en el lenguaje de programación ficticio Pidgin Pascal. La ventaja de utilizar Pidgin Pascal aquí es que, además de su pureza y simplicidad, sigue de un modo exacto el modelo de Hoare/Brinch Hansen.

```

monitor ProductorConsumidor
  condition lleno, vacio ;
  integer contador ;

  procedure meter ( elemento : integer ) ;
  begin
    if contador = N then wait(lleno) ;
    meter_elemento(elemento) ;
    contador := contador + 1 ;
    if contador = 1 then signal(vacio)
  end ;

  function sacar ( ) : integer ;
  begin
    if contador = 0 then wait(vacio) ;
    sacar := sacar_elemento ;
    contador := contador - 1 ;
    if contador = N - 1 then signal(lleno)
  end ;

  contador := 0 ;

end monitor ;

procedure productor ;
begin
  while true do
    begin
      elemento := producir_elemento () ;
      ProductorConsumidor.meter(elemento) ;
    end
  end ;

procedure consumidor ;
begin
  while true do
    begin
      elemento := ProductorConsumidor.sacar () ;
      consumir_elemento(elemento)
    end
  end ;

```

Figura 2-27. Un esbozo de solución del problema del productor-consumidor con monitores. En cada momento sólo está activo un procedimiento del monitor. El buffer tiene N entradas.

Alguien puede pensar que las operaciones **wait** y **signal** se parecen a las operaciones **sleep** y **wakeup**, que vimos anteriormente que conducían a condiciones de carrera fatales. Aunque *son* muy similares, existe una diferencia crucial: **sleep** y **wakeup** fallaban debido a que mientras que un proceso estaba tratando de dormirse, el otro proceso estaba tratando de despertarlo. Con los monitores esto no puede ocurrir. La exclusión mutua que proporcionan automáticamente los monitores en el acceso a sus procedimientos garantiza que si, por ejemplo, el productor dentro del procedimiento del monitor descubre que el búfer está lleno, siempre será capaz de completar la operación **wait** sin tener que preocuparse sobre la posibilidad de que el planificador ceda el control al consumidor justamente antes de que el **wait** se complete. El

consumidor ni siquiera puede entrar en el monitor hasta que el `wait` termine y el productor pase al estado bloqueado.

Aunque Pidgin Pascal es un lenguaje imaginario, algunos lenguajes de programación reales soportan los monitores, aunque no siempre en la forma diseñada por Hoare y Brinch Hansen. Un tal lenguaje es Java. Java es un lenguaje orientado a objetos que soporta threads a nivel de usuario y que permite agrupar juntos varios métodos (procedimientos) formando clases. Añadiendo la palabra reservada `synchronized` a la declaración de un método, Java garantiza que una vez que un thread comienza a ejecutar ese método, no se le permite a ningún otro thread comenzar a ejecutar cualquier otro método `synchronized` de esa clase.

En la Figura 2-28 se da una solución en Java al problema del productor-consumidor utilizando monitores. La solución consta de cuatro clases. La clase más externa, `ProductorConsumidor`, crea y arranca dos threads, `p` y `c`. La segunda y tercera clases, `productor` y `consumidor` contienen, respectivamente, el código del productor y el del consumidor. Finalmente, la clase `nuestro_monitor`, es el monitor y contiene dos métodos sincronizados que se utilizan para insertar y sacar elementos del búfer compartido. Aquí, contrariamente a los ejemplos previos, sí que hemos mostrado completamente el código de `meter` y `sacar`.

Los threads productor y consumidor son funcionalmente idénticos a sus contrapartidas en todos nuestros ejemplos anteriores. El productor consiste en un bucle infinito en el que se genera un dato y se mete en el búfer común. El consumidor consiste igualmente en un bucle infinito en el que se saca un dato del búfer común y se hace alguna cosa divertida con él.

La parte interesante de este programa es la clase `nuestro_monitor`, que contiene el búfer, sus variables de administración y dos métodos sincronizados. Cuando el productor está activo dentro del `meter`, éste sabe con seguridad que el consumidor no puede estar activo dentro del `sacar`, por lo que resulta segura la actualización de las variables y del búfer sin temor a la aparición de condiciones de carrera. La variable `contador` controla el número de elementos que hay en el búfer. Esta variable puede tomar cualquier valor desde 0 hasta N inclusive. La variable `lo` es el índice de la entrada en el búfer de donde va a extraerse el siguiente elemento. Similarmente, `hi` es el índice de la entrada del búfer donde va a colocarse el siguiente elemento que se introduzca en él. Se permite que `lo = hi`, lo que significa que en el búfer hay 0 o N elementos. El valor de `contador` nos dice cuál de los dos casos es el que realmente se está dando.

Los métodos sincronizados de Java difieren de los monitores clásicos en un aspecto esencial: Java no dispone de variables de condición. En su lugar, ofrece dos procedimientos `wait` y `notify` que son los equivalentes de `sleep` y `wakeup` salvo que cuando se utilizan dentro de métodos sincronizados no están expuestos a que se produzcan condiciones de carrera. En teoría, el método `wait` puede ser interrumpido, y eso es por lo que se ha incluido el código que lo rodea. Java requiere que el tratamiento de excepciones se haga explícito. Para nuestros propósitos, supondremos que `go_to_sleep` es la forma de conseguir que un proceso se duerma.

```

public class ProductorConsumidor {
    static final int N = 100 ;                                // constante que fija el tamaño del búfer
    static productor p = new productor( ) ;                  // instancia un nuevo thread productor
    static consumidor c = new consumidor( ) ;                // instancia un nuevo thread consumidor
    static nuestro_monitor mon = new nuestro_monitor( ) ;   // instancia un nuevo monitor

    public static void main(String args[ ]) {
        p.start( ) ;      // arranca el thread productor
        c.start( ) ;      // arranca el thread consumidor
    }

    static class productor extends Thread {
        public void run( ) { // el metodo run contiene el codigo del thread
            int elemento ;
            while (true) { // bucle del productor
                elemento = producir_elemento( );
                mon.meter(elemento) ;
            }
        }
        private int producir_elemento( ) { ... }           // producir realmente
    }

    static class consumidor extends Thread {
        public void run( ) { // el metodo run contiene el codigo del thread
            int elemento ;
            while (true) { // bucle del consumidor
                elemento = mon.sacar( );
                consumir_elemento(elemento) ;
            }
        }
        private void consumir_elemento(int elemento) { ... } // consumir realmente
    }

    static class nuestro_monitor {                         // esto es un monitor
        private int buffer[ ] = new int [N] ;
        private int contador = 0, lo = 0, hi = 0 ;          // contadores e índices

        public synchronized void meter (int val) {
            if (contador == N) go_to_sleep( ) ; // si el bufer esta lleno, dormirse
            bufer[hi] = val ;                 // meter un elemento en el bufer
            hi = (hi + 1) % N ;              // entrada donde colocar el siguiente elemento
            contador = contador + 1 ;        // un elemento mas ahora en el bufer
            if (contador == 1) notify( ) ;    // si el consumidor esta dormido, despertarlo
        }

        public synchronized void sacar ( ) {
            int val ;
            if (contador == 0) go_to_sleep( ) ; // si el bufer esta vacio, dormirse
            val = bufer[lo] ;                // sacar un elemento del bufer
            lo = (lo + 1) % N ;              // entrada de donde extraer el siguiente elemento
            contador = contador - 1 ;       // un elemento menos en el bufer
            if (contador == N - 1) notify( ) ; // si el productor esta dormido, despertarlo
            return val ;
        }

        private void go_to_sleep( ) { try { wait( ) ; } catch(interruptedException exc) {} ; }
    }
}

```

Figura 2-28. Una solución al problema del productor-consumidor en Java.

Mediante la automatización de la exclusión mutua de las regiones críticas, los monitores consiguen que la programación paralela sea mucho menos propensa a errores que con los semáforos. Aún así, los monitores tienen también algunos inconvenientes. No es por nada que nuestros dos ejemplos de monitores correspondan a Pidgin Pascal y Java en vez de C, como sucede con los demás ejemplos en este libro. Como hemos dicho anteriormente, los monitores son un concepto del lenguaje de programación. El compilador debe reconocerlos y añadir el código necesario para conseguir la exclusión mutua. C, Pascal y la mayoría de los demás lenguajes no disponen de monitores, de manera que no es razonable esperar que sus compiladores refuercen ninguna regla de exclusión mutua. De hecho, ¿cómo puede el compilador ni tan siquiera saber qué procedimientos están en los monitores y cuáles no?

Estos mismos lenguajes no cuentan tampoco con semáforos, pero resulta muy fácil añadírselos: Lo único que se necesita hacer es añadir dos pequeñas rutinas en código ensamblador a la biblioteca para tener a nuestra disposición las llamadas al sistema **subir** y **bajar**. Los compiladores ni siquiera tienen porqué enterarse de que existen. Por supuesto el sistema operativo tiene que tener conocimiento de los semáforos, pero al menos si disponemos de un sistema operativo basado en semáforos, podemos seguir escribiendo los programas de usuario en C o C++ (o incluso en lenguaje ensamblador si somos lo bastante masoquistas). Con los monitores, necesitamos un lenguaje que ya los tenga integrados.

Otro problema con los monitores, y también con los semáforos, es que fueron diseñados para resolver el problema de la exclusión mutua sobre una o más CPUs que tienen acceso a una memoria común. Poniendo los semáforos en la memoria compartida y protegiéndolos con la instrucción **TSL**, podemos evitar las condiciones de carrera. Cuando pasamos a un entorno diferente como un sistema distribuido consistente en múltiples CPUs (cada una de las cuales tiene su propia memoria privada) conectadas a través de una red de área local, esas primitivas son inaplicables. La conclusión es que los semáforos son primitivas de un nivel demasiado bajo y que los monitores no están disponibles más que en unos pocos lenguajes de programación. Además, ninguna de esas primitivas proporciona intercambio de información entre diferentes máquinas. Necesitamos alguna otra cosa.

2.3.8 Paso de Mensajes

Esa otra cosa es el **paso de mensajes**. Este método de comunicación entre procesos utiliza dos primitivas, **enviar** y **recibir**, que igual que los semáforos y de forma diferente a los monitores son llamadas al sistema en vez de construcciones del lenguaje de programación. Como tales, pueden incluirse fácilmente en librerías de procedimientos, tales como

```
enviar(destinatario, &mensaje) ;
y
recibir(remitente, &mensaje) ;
```

La primera llamada envía un mensaje a un destinatario dado y la segunda recibe un mensaje de un remitente dado (o de cualquiera, si al receptor no le importa el remitente). Si no está disponible ningún mensaje, el receptor puede bloquearse hasta que llegue uno. De forma alternativa, puede retornar inmediatamente indicando un código de error.

Cuestiones de Diseño para Sistemas de Paso de Mensajes

Los sistemas de paso de mensajes plantean numerosos problemas desafiantes y cuestiones de diseño que no se presentan con los semáforos o los monitores, especialmente si los procesos que se comunican residen en máquinas diferentes conectadas por una red. Por ejemplo, los mensajes pueden perderse a través de la red. Para hacer frente a la pérdida de mensajes, el remitente y el receptor pueden convenir que tan pronto como se recibe un mensaje,

el receptor debe responder enviando un mensaje especial de **acuse** (*acknowledgement*) de la recepción. Si el remitente no recibe el mensaje de acuse dentro de un cierto intervalo de tiempo, procede a retransmitir el mensaje.

Consideremos ahora qué ocurre si el mensaje original se recibe correctamente, pero el correspondiente mensaje de acuse se pierde. El remitente retransmitirá el mensaje, de manera que el receptor lo recibirá dos veces. Es esencial que el receptor sea capaz de distinguir un nuevo mensaje, de la retransmisión de uno antiguo. Usualmente este problema se resuelve añadiendo números de secuencia consecutivos a cada mensaje original. Si al receptor le llega un mensaje con el mismo número de secuencia que otro mensaje anterior, el receptor sabe que ese mensaje es un duplicado que puede ignorarse. Comunicarse de una forma efectiva en un entorno con paso de mensajes no fiable es una parte importante del estudio de las redes de ordenadores. Para más información, véase (Tanenbaum, 1996).

Los sistemas de mensajes también deben de dar respuesta a la pregunta de cómo referenciar o nombrar a los procesos, de forma que el proceso especificado por una llamada a enviar o recibir lo sea de una forma no ambigua. Otro aspecto de los sistemas de mensajes es la **autenticación**: ¿cómo puede estar seguro el cliente de que se está comunicando con el servidor de ficheros real, y no con un impostor?

En el otro extremo del espectro existen cuestiones de diseño que son importantes cuando el remitente y el receptor están en la misma máquina. Una de esas cuestiones es el rendimiento. Copiar los mensajes de un proceso a otro es siempre más lento que hacer una operación sobre un semáforo o entrar en un monitor. Se ha dedicado mucho esfuerzo para conseguir que el paso de mensajes sea eficiente. Cheriton (1984), por ejemplo, sugirió limitar el tamaño de los mensajes a lo que sea posible meter en los registros de la máquina, realizando el paso de mensajes utilizando los registros.

El Problema del Productor-Consumidor con Paso de Mensajes

Vamos a ver ahora como puede resolverse el problema del productor-consumidor con paso de mensajes y sin memoria compartida. En la Figura 2-29 se da una solución. Suponemos que todos los mensajes son del mismo tamaño y que los mensajes enviados pero no recibidos todavía los almacena automáticamente el sistema operativo. En esta solución se utiliza un total de N mensajes, de forma análoga a las N entradas de un búfer en memoria compartida. El consumidor comienza enviando al productor N mensajes vacíos. Siempre que el productor tiene un elemento para dar al consumidor, toma un mensaje vacío y responde enviando uno lleno. De esta manera, el número total de mensajes en el sistema permanece constante a lo largo del tiempo, de forma que todos los mensajes pueden almacenarse invirtiendo una cantidad fija de memoria que se conoce desde el principio.

Si el productor trabaja más rápido que el consumidor, todos los mensajes terminan por estar llenos, esperando a ser recibidos por el consumidor; el productor se bloqueará, esperando recibir un mensaje vacío para poder continuar. Si el consumidor trabaja más rápido, entonces ocurre a la inversa: todos los mensajes quedarán vacíos esperando a que el productor los llene; el consumidor se bloqueará, esperando a que llegue un mensaje lleno.

```

#define N 100                                /* número de entradas en el bufer */

void productor (void)
{
    int elemento ;
    mensaje m ;                            /* contiene temporalmente el mensaje */

    while (TRUE) {
        elemento = producir_elemento( ) ;   /* generar algo para meter en el bufer */
        recibir(consumidor, &m) ;           /* espera a que llegue un mensaje vacío */
        construir_mensaje(&m, elemento) ;  /* construir el mensaje a enviar */
        enviar(consumidor, &m) ;            /* enviar elemento al consumidor */
    }
}

void consumidor(void)
{
    int elemento, i ;
    mensaje m ;

    for (i = 0; i < N; i++)                /* enviar N mensajes vacíos */
        enviar(productor, &m) ;
    while (TRUE) {
        recibir(productor, &m) ;           /* tomar un mensaje con un elemento */
        elemento = extraer_elemento(&m) ;  /* extraer el elemento del mensaje */
        enviar(productor, &m) ;            /* devolver el mensaje vacío */
        consumir_elemento(elemento) ;      /* hacer algo con el elemento */
    }
}

```

Figura 2-29. El problema del productor-consumidor con N mensajes.

Existen numerosas variantes de paso de mensajes. Para los principiantes, vamos a tratar cómo se especifica la dirección de los mensajes. Una forma es asignar a cada proceso una única dirección y dirigir los mensajes a los procesos. Una forma diferente es inventar una nueva estructura de datos, que denominaremos **buzón**. Un buzón es un lugar donde podemos depositar un cierto número de mensajes, especificado normalmente cuando se crea el buzón. Cuando se utilizan buzones, los parámetros correspondientes a la dirección en las llamadas al sistema **enviar** y **recibir** son buzones, no procesos. Cuando un proceso intenta enviar un mensaje a un buzón que está lleno, el proceso se suspende hasta que se recoge algún mensaje de los depositados en el buzón, dejando sitio para un nuevo mensaje.

En el caso del problema del productor-consumidor, tanto el productor como el consumidor podrían crear buzones suficientemente grandes para contener N mensajes. El productor podría enviar mensajes contenido datos al buzón del consumidor, y el consumidor podría enviar mensajes vacíos al buzón del productor. Cuando se utilizan buzones el mecanismo de almacenamiento intermedio de los mensajes es claro: el buzón de destino almacena los mensajes que se han enviado al proceso de destino pero que no han sido recibidos todavía.

El extremo opuesto a tener buzones sería eliminar cualquier almacenamiento intermedio de los mensajes. Cuando se sigue este enfoque, si el **enviar** se realiza antes que el **recibir**, el proceso que envía deberá bloquearse hasta que tenga lugar el **recibir**, momento en el cual el mensaje puede copiarse directamente desde el remitente al receptor, sin ningún almacenamiento intermedio. Similamente, si el **recibir** tiene lugar primero, el receptor se bloquea hasta que se

realiza una operación **enviar**. Esta estrategia se conoce con el nombre de **cita** (*rendezvous*). Es más fácil de implementar que un esquema de mensajes con almacenamiento intermedio pero es mucho menos flexible ya que se fuerza al remitente y al receptor a trabajar juntos de una forma síncrona.

El paso de mensajes se utiliza comúnmente en sistemas de programación paralela. Un sistema de paso de mensajes bien conocido es **MPI** (*Message-Passing Interface*). Se utiliza muy ampliamente en computación científica. Para encontrar más información sobre MPI, véase por ejemplo (Gropp y otros, 1994; y Snir y otros, 1996).

2.3.9 Barreras

Nuestro último mecanismo de sincronización está pensado para grupos de procesos en vez de para situaciones con tan sólo dos procesos como en el caso del productor-consumidor. Algunas aplicaciones se dividen en fases imponiéndose la regla de que ningún proceso puede pasar a la fase siguiente hasta que todos los procesos estén listos para comenzar esa siguiente fase. Este comportamiento puede conseguirse disponiendo una **barrera** al final de cada fase. Cuando un proceso alcanza la barrera, se bloquea hasta que todos los procesos hayan alcanzado la barrera. La forma de operar de una barrera se ilustra en la Figura 2-30.

En la Figura 2-30(a) vemos cuatro procesos aproximándose a una barrera. Lo que esto significa es que esos procesos están realizando cálculos y no han alcanzado todavía el final de la fase actual. Después de un rato, el primer proceso termina todos los cálculos que le corresponden durante la primera fase, por lo que ejecuta la primitiva **barrera**, generalmente llamando a un procedimiento de librería. En ese momento el proceso se suspende. Un poco después, un segundo y un tercer proceso terminan la primera fase y ejecutan también la primitiva **barrera**. Esta situación se ilustra en la Figura 2-30(b). Finalmente, cuando el último proceso, *C*, alcanza la barrera, se produce el desbloqueo de todos los procesos a la vez, como se muestra en la Figura 2-30(c).

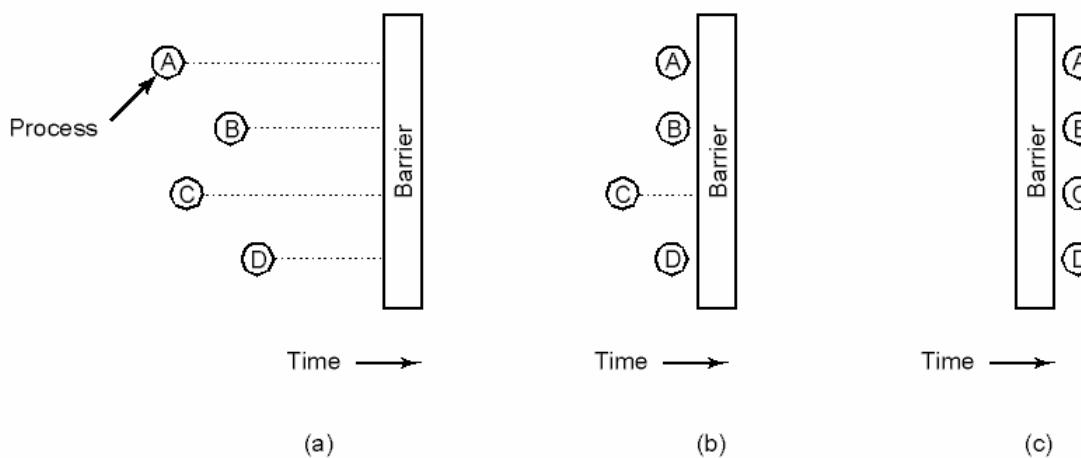


Figura 2-30. Utilización de una barrera. (a) Los procesos se aproximan a una barrera. (b) Todos los procesos salvo uno están bloqueados en la barrera. (c) Cuando llega el último proceso a la barrera, todos los procesos consiguen traspasarla.

Como ejemplo de un problema que requiere el uso de barreras, consideremos un problema de propagación del calor en física o ingeniería. Es típico que haya una matriz que

contiene algunos valores iniciales. Los valores pueden representar temperaturas en diferentes puntos de una plancha de metal. La idea puede ser calcular cuánto tiempo tarda en propagarse a través de la plancha metálica el calor de una llama aplicado a una de sus esquinas.

Comenzando con los valores actuales, se aplica una transformación a la matriz para obtener la segunda versión de la matriz, por ejemplo, aplicando las leyes de la termodinámica que determinan que un momento después todas las temperaturas se han elevado en una cantidad ΔT . A continuación el proceso se repite una y otra vez, obteniéndose las temperaturas en los puntos muestreados como una función del tiempo a medida que la plancha se calienta. Así el algoritmo produce una serie de matrices a lo largo del tiempo.

Supongamos ahora que la matriz es muy grande (pongamos que de 1 millón de filas por 1 millón de columnas), de manera que se necesitan numerosos procesos paralelos (posiblemente sobre una máquina multiprocesador) para acelerar los cálculos. Tenemos por tanto que diferentes procesos trabajan sobre diferentes partes de la matriz calculando los nuevos elementos a partir de los anteriores de acuerdo con las leyes de la física. Sin embargo ningún proceso puede comenzar la iteración $n + 1$ hasta que no se complete la iteración n , esto es, hasta que todos los procesos hayan terminado su trabajo actual. La forma de conseguir este objetivo es programar cada proceso para que ejecute una operación **barrera** después de que haya terminado su parte de la iteración actual. Cuando todos los procesos alcancen esa barrera estará terminada la nueva matriz (que es el dato de entrada de la siguiente iteración), y todos los procesos se verán desbloqueados simultáneamente para comenzar la siguiente iteración.

2.5 PLANIFICACIÓN

En un ordenador multiprogramado es frecuente que en un momento dado haya múltiples procesos compitiendo por el uso de la CPU al mismo tiempo. Esta situación se da siempre que dos o más procesos están simultáneamente en el estado preparado. Si sólo hay una CPU disponible, es necesario hacer una elección para determinar cual de esos procesos será el siguiente que se ejecute. La parte del sistema operativo que realiza esa elección se denomina el **planificador** (*scheduler*), y el algoritmo que se utiliza para esa elección se denomina el **algoritmo de planificación**. Estos conceptos constituyen el tema principal de las siguientes secciones.

Muchas de las ideas que se aplican a la planificación de procesos se aplican también a la planificación de los threads, aunque algunas son diferentes. Inicialmente vamos a concentrarnos en la planificación de procesos. Posteriormente trataremos explícitamente la planificación de los threads.

2.5.1 Introducción a la Planificación

En los viejos tiempos de los sistemas de batch con entrada en forma de imágenes de tarjetas perforadas grabadas en cinta magnética, el algoritmo de planificación era muy sencillo: ejecutar el siguiente trabajo que aparece grabado en la cinta. Con la llegada de los sistemas en tiempo compartido, el algoritmo de planificación se hizo más complejo debido a que generalmente había muchos usuarios esperando recibir un servicio inmediato. Algunos mainframes todavía combinan los servicios de batch y los de tiempo compartido, requiriendo que el planificador decida si el proceso que va a continuación debe ser un trabajo en batch o el de un usuario interactivo sentado ante un terminal. (Aunque un trabajo en batch puede consistir en la ejecución sucesiva de múltiples programas, en esta sección vamos a suponer que sólo consiste en la ejecución de un único programa). Debido a que el tiempo de CPU es un recurso escaso en estas máquinas, un buen planificador puede conseguir grandes mejoras en el rendimiento percibido y la satisfacción del usuario. Consecuentemente, se ha invertido una gran cantidad de trabajo en diseñar algoritmos de planificación muy ingeniosos y eficientes.

La situación ha cambiado en dos aspectos con la llegada de los ordenadores personales. En primer lugar, la mayoría del tiempo sólo hay un proceso activo. Es poco probable que un usuario editando un documento con su procesador de texto esté además compilando simultáneamente un programa como proceso de fondo. Cuando el usuario teclea un comando del procesador de texto, el planificador no tiene que trabajar mucho para darse cuenta de cual es el siguiente proceso a ejecutar – el procesador de texto es el único candidato existente.

En segundo lugar, a través de los años los ordenadores se han hecho mucho más rápidos por lo que la CPU raramente será ya un recurso escaso. La mayoría de los programas para ordenadores personales están limitados por la velocidad a la cual el usuario puede introducir los datos (mediante el teclado o el ratón), no por la velocidad a la cual la CPU puede procesarlos. Incluso las compilaciones, una de los mayores sumideros de ciclos de CPU en el pasado, requieren tan solo de unos pocos microsegundos en la actualidad. Por otra parte cuando tenemos dos programas que se están ejecutando a la vez, tales como un procesador de texto y una hoja de cálculo, poco importa cual de los dos va primero ya que muy probablemente el usuario está esperando a que terminen los dos. Como consecuencia, la planificación no resulta demasiado preocupante en los PCs más sencillos. [Por supuesto, hay aplicaciones que prácticamente se comen viva a la CPU: renderizar una hora de vídeo de alta resolución puede requerir el procesamiento de imágenes a nivel industrial para cada uno de los 108.000 fotogramas en NTSC (90.000 en PAL), pero estas aplicaciones son más bien la excepción y no la regla.]

Cuando pasamos a hablar de las estaciones de trabajo y servidores de gama alta, la situación cambia notablemente. Aquí es frecuente tener múltiples procesos que compiten por la

CPU, de manera que la planificación vuelve a ser muy importante. Por ejemplo, cuando la CPU tiene que elegir entre ejecutar un proceso que actualiza la pantalla después de que el usuario haya cerrado una ventana, y ejecutar un proceso que envía el correo electrónico pendiente, esa elección provoca enormes diferencias en la percepción de cómo responde el ordenador. Si cerrar la ventana tarda 2 segundos mientras que el correo electrónico se envía, el usuario pensará muy probablemente que el sistema es extremadamente lento, mientras que demorar el envío del correo en 2 segundos puede que incluso ni se note. En este caso sí que importa mucho la planificación de los procesos.

El planificador, además de tener que escoger el proceso correcto para ejecutar, tiene que preocuparse también de realizar un uso eficiente de la CPU, debido a que la commutación de procesos es muy costosa. Para empezar, debe producirse el paso de modo usuario a modo supervisor. Luego debe salvase el estado del proceso actual, incluyendo el almacenamiento de los registros en la tabla de procesos de forma que puedan recargarse posteriormente. En muchos sistemas, también debe salvase el mapa de memoria (por ejemplo, los bits de referencia a memoria en la tabla de páginas). A continuación debe seleccionarse un nuevo proceso mediante la ejecución del algoritmo de planificación. Después de eso, la MMU (Unidad de Gestión de Memoria) debe volverse a cargar con el mapa de memoria del nuevo proceso. Finalmente, debe arrancarse el nuevo proceso. Adicionalmente a todo eso, la commutación del proceso usualmente invalida toda la memoria caché, forzando que sea recargada dinámicamente desde la memoria principal dos veces (tras entrar en el núcleo y tras abandonarlo). En resumen, la realización de demasiadas commutaciones de procesos por segundo puede consumir una cantidad significativa de tiempo de CPU, por lo que debemos ser prudentes.

Comportamiento de los Procesos

Casi todos los procesos alternan ráfagas de computación con peticiones de E/S (disco), como se muestra en la Figura 2-37. Normalmente la CPU ejecuta instrucciones durante un cierto intervalo de tiempo sin detenerse, y luego realiza una llamada al sistema para leer de un fichero o escribir en un fichero. Cuando se completa la llamada al sistema, la CPU vuelve a realizar cálculos hasta que necesita más datos o hasta que tiene que escribir más datos, y así sucesivamente. Hay que aclarar que algunas operaciones de E/S cuentan aquí como computación. Por ejemplo, cuando la CPU copia bits a una RAM de vídeo para actualizar la pantalla, esto es computación, no realización de E/S, debido a que la CPU está utilizándose. La E/S en este sentido es cuando un proceso pasa al estado de bloqueado esperando a que un dispositivo externo termine un cierto trabajo.

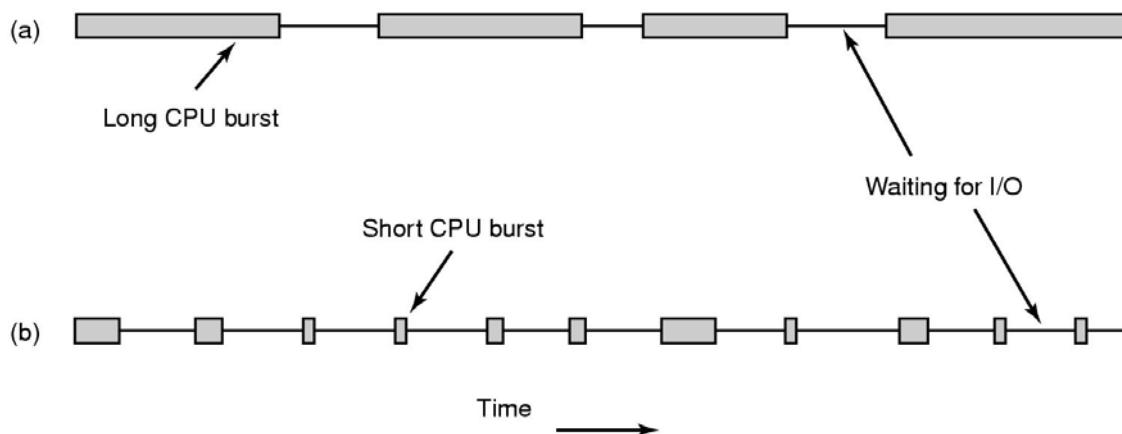


Figura 2-37. Ráfagas de uso de CPU alternadas con períodos de espera por E/S. (a) Un proceso intensivo en CPU. (b) Un proceso intensivo en E/S.

Lo importante a destacar en la Figura 2-37 es que algunos procesos, tales como el de la Figura 2-37(a), invierten la mayor parte de su tiempo computando, mientras que otros tales como el de la Figura 2-37(b), invierten la mayor parte de su tiempo esperando por la terminación de operaciones de E/S. El primer proceso se dice que es **intensivo en computación (CPU-bound)**; el segundo se dice que es **intensivo en E/S (I/O-bound)**. Los procesos intensivos en computación tienen normalmente ráfagas de CPU muy largas y por lo tanto esperan por E/S poco frecuentes, mientras que los procesos intensivos en E/S tienen ráfagas de CPU muy cortas y por lo tanto esperan por E/S muy frecuentes. Hay que darse cuenta de que el factor clave es la longitud de la ráfaga de CPU, no la longitud de la ráfaga de E/S. Los procesos intensivos en E/S lo son debido a que realizan poca computación entre peticiones de E/S sucesivas, y no debido a que tengan peticiones de E/S especialmente largas. Se requiere el mismo tiempo para leer un bloque del disco sin que importe lo mucho o lo poco que se tarde en procesar los datos después de que hayan llegado.

Es necesario destacar que a medida que las CPUs se hacen más rápidas, los procesos tienden a ser más intensivos en E/S. Este efecto se produce debido a que las CPUs se están mejorando mucho más rápido que los discos. Como consecuencia, es muy probable que la planificación de los procesos intensivos en E/S se convierta en un tema muy importante en el futuro. La idea básica aquí es que si un proceso intensivo en E/S quiere ejecutarse, debe dársele esa oportunidad rápidamente de forma que pueda formular su petición al disco, y mantener así el disco ocupado.

Cuando Planificar

Una cuestión clave relacionada con la planificación es en qué momento realizar las decisiones de planificación. Resulta que hay una gran variedad de situaciones en las que es necesaria la planificación. En primer lugar, cuando se crea un nuevo proceso, se necesita decidir si se ejecuta el proceso padre o el proceso hijo. Ya que ambos procesos están en el estado preparado, se trata de una decisión de planificación normal y puede hacerse de cualquiera de las dos maneras, esto es, el planificador puede legítimamente elegir ejecutar a continuación bien el padre, o bien el hijo.

En segundo lugar, debe hacerse una decisión de planificación cuando un proceso termina. Ya que no puede seguirse ejecutando ese proceso (puesto que ya no existe), debe elegirse a algún otro proceso de entre el conjunto de procesos preparados. Si no hay ningún proceso preparado, normalmente se ejecuta un **proceso ocioso** proporcionado por el sistema.

En tercer lugar, cuando un proceso se bloquea en E/S, sobre un semáforo, o por alguna otra razón, es necesario seleccionar algún otro proceso para ejecutarlo. A veces el motivo del bloqueo puede jugar algún papel en la elección. Por ejemplo, si *A* es un proceso importante y está esperando a que *B* salga de su región crítica, el dejar que *B* se ejecute a continuación puede permitir que salga de su región crítica, haciendo posible que *A* pueda continuar. Sin embargo, el problema es que generalmente el planificador no cuenta con la información necesaria para poder tomar en cuenta esta dependencia entre los procesos.

En cuarto lugar, cuando tiene lugar una interrupción debida a una E/S, también es necesario tomar una decisión de planificación. Si la interrupción proviene de un dispositivo de E/S que ha completado ahora su trabajo, entonces algún proceso que estaba bloqueado esperando por la E/S puede ahora estar preparado para ejecutarse. Corresponde al planificador decidir si debe ejecutarse el nuevo proceso preparado, si debe continuar ejecutándose el proceso en ejecución en el momento de la interrupción, o si debe ejecutarse algún tercer proceso.

Si un reloj de hardware proporciona interrupciones periódicas a 50 Hz, 60 Hz, o a alguna otra frecuencia, puede tomarse una decisión de planificación en cada interrupción de reloj o en cada *k*-ésima interrupción de reloj. Los algoritmos de planificación pueden dividirse en dos

categorías si tenemos en cuenta cómo se comportan ante las interrupciones de reloj. Un algoritmo de planificación **no expulsor** (*nonpreemptive*) escoge un proceso para ejecutar dejándole que se ejecute hasta que se bloquee (bien debido a una E/S o esperando por otro proceso) o hasta que el proceso ceda voluntariamente la CPU. Incluso si el proceso se ejecuta durante horas, el sistema nunca forzará que el proceso se suspenda. Por tanto, durante las interrupciones del reloj nunca se toma ninguna decisión de planificación. Después de terminado el procesamiento de la interrupción del reloj, siempre se retoma para su ejecución al mismo proceso que estaba ejecutándose antes de la interrupción.

En contraste, un algoritmo de planificación **expulsor** (*preemptive*) escoge un proceso y permite que se ejecute durante un máximo de tiempo prefijado. Si el proceso está todavía ejecutándose al final de ese intervalo de tiempo, se le suspende y el planificador escoge algún otro proceso para ejecutarse (si es que hay alguno disponible). Para que sea posible una planificación expulsora es necesario contar con una interrupción de reloj que tenga lugar al final del intervalo de tiempo para ceder el control de la CPU de nuevo al planificador. Si no está disponible ningún reloj, la única opción posible es la planificación no expulsora.

Categorías de Algoritmos de Planificación

De forma nada sorprendente sucede que en entornos diferentes se necesitan algoritmos de planificación diferentes. Esto se debe a que cada área de aplicación (y cada tipo de sistema operativo) tiene objetivos diferentes. En otras palabras, lo que el planificador debe optimizar no es lo mismo en todos los sistemas. Es necesario distinguir aquí tres entornos:

1. Batch.
2. Interactivo.
3. Tiempo Real.

En los sistemas en batch, no existen usuarios que estén esperando impacientemente por una rápida respuesta ante sus terminales. En consecuencia, son aceptables los algoritmos no expulsores, o los algoritmos expulsores con largos períodos de tiempo para cada proceso. Con este enfoque se reduce el número de cambios de proceso, mejorando por tanto el rendimiento.

En un entorno con usuarios interactivos es indispensable que haya expulsiones para impedir que un proceso acapare la CPU, negando cualquier servicio de la CPU a los demás. Incluso aunque ningún proceso tenga intención de ejecutarse eternamente, es posible que debido a un error en el programa un proceso mantenga parados a todos los demás indefinidamente. La expulsión es necesaria para impedir ese comportamiento.

En los sistemas con restricciones de tiempo real, por extraño que parezca, la expulsión es algunas veces innecesaria debido a que los procesos saben que no pueden ejecutarse durante largos períodos de tiempo y usualmente hacen su trabajo y rápidamente se bloquean. La diferencia con los sistemas interactivos es que los sistemas en tiempo real sólo ejecutan programas pensados como parte de una misma aplicación. Los sistemas interactivos por el contrario son sistemas de propósito general y pueden ejecutar programas arbitrarios no cooperantes o incluso maliciosos.

Objetivos de los Algoritmos de Planificación

En orden a diseñar un algoritmo de planificación es necesario tener alguna idea de qué es lo que debe hacer un buen algoritmo de planificación. Algunos objetivos dependen del entorno (sistema en batch, interactivo o de tiempo real), pero hay también algunos otros

objetivos que son deseables en todos los casos. Algunos objetivos aparecen listados en la Figura 2-38. Vamos a discutirlos uno a uno a continuación.

Todos los sistemas

- Justicia – dar a cada proceso la parte de CPU que le corresponde
- Reforzamiento de la política – cuidar de que la política establecida se cumpla
- Equilibrio – mantener todas las partes del sistema ocupadas

Sistemas de batch

- Rendimiento – maximizar el número de trabajos ejecutados por hora
- Tiempo de retorno – minimizar el tiempo entre la entrada y la conclusión de un trabajo
- Utilización de la CPU – mantener la CPU ocupada todo el tiempo

Sistemas interactivos

- Tiempo de respuesta – responder rápidamente a las peticiones
- Proporcionalidad – satisfacer las expectativas de los usuarios

Sistemas de tiempo real

- Cumplir los límites de tiempo – evitar la pérdida de datos
- Predecibilidad – evitar la degradación de la calidad en sistemas multimedia

Figura 2-38. Algunos objetivos del algoritmo de planificación bajo diferentes circunstancias.

Bajo cualquier circunstancia, es importante la justicia (*fairness*). Procesos similares deben recibir servicios similares. Dar a un proceso mucho más tiempo de CPU que a otro proceso equivalente no es justo. Por supuesto, diferentes categorías de procesos pueden recibir un trato muy diferente. Pensemos en el control de seguridad y en el procesamiento de las nóminas en un centro de computación de un reactor nuclear.

Algo que está relacionado con la justicia es el reforzamiento de las políticas del sistema. Si la política local es que los procesos de control de la seguridad se ejecuten siempre que quieran hacerlo, incluso si eso significa que las nóminas se retrasen 30 segundos, el planificador tiene que asegurar el cumplimiento de esta política.

Otro objetivo general es mantener ocupadas todas las partes del sistema siempre que sea posible. Si la CPU y todos los dispositivos de E/S pueden mantenerse ocupados todo el tiempo, se realizará más trabajo por segundo que si alguno de los componentes está ocioso. Por ejemplo en un sistema en batch el planificador tiene el control de qué trabajos están cargados en memoria para ejecutarse. Tener en memoria juntos algunos procesos intensivos en CPU y algunos procesos intensivos en E/S es una idea mejor que primero cargar y ejecutar todos los trabajos intensivos en CPU y a continuación una vez que se hayan terminado, cargar y ejecutar todos los trabajos intensivos en E/S. Si se utiliza la última estrategia, mientras se ejecutan los procesos intensivos en CPU, esos procesos lucharán por conseguir la CPU y el disco permanecerá ocioso. Luego, cuando se carguen los trabajos intensivos en E/S, esos procesos lucharán por utilizar el disco y la CPU permanecerá ociosa. Es por tanto preferible mantener en memoria una mezcla de procesos cuidadosamente estudiada para conseguir que todo el sistema esté funcionando a la vez.

Los administradores de grandes centros de cálculo donde se ejecutan muchos trabajos normalmente utilizan tres métricas para estimar lo bueno que es el servicio ofrecido por sus sistemas: rendimiento, tiempo de retorno y utilización de la CPU. El **rendimiento** (*throughput*) es el número de trabajos por hora terminados por el sistema. A igualdad de otros factores,

terminar 50 trabajos por hora es mucho mejor que terminar 40 trabajos por hora. El **tiempo de retorno** (*turnaround time*) es el tiempo medio estadístico que transcurre desde que un trabajo en batch se introduce en el sistema hasta que se completa. Mide por tanto cuánto tiempo tiene que esperar el usuario medio para obtener la salida de su trabajo. La regla es aquí: lo pequeño es bello.

Un algoritmo de planificación que maximice el rendimiento no tiene necesariamente por qué minimizar el tiempo de retorno. Por ejemplo, dada una mezcla de trabajos cortos y trabajos largos, un planificador que ejecute siempre los trabajos cortos y nunca los trabajos largos puede conseguir un rendimiento excelente (muchos trabajos cortos por hora) pero a expensas de obtener un tiempo de retorno terriblemente malo para los trabajos largos. Si continuamente están llegando trabajos cortos a una velocidad constante, es posible que los trabajos largos nunca lleguen a ejecutarse, provocando que el tiempo de retorno medio se haga infinito al tiempo que se consigue un alto rendimiento.

La utilización de la CPU es también una cuestión importante en los sistemas de batch debido a que en los mainframes donde se ejecutan los sistemas de batch la CPU representa todavía el principal coste. Por tanto los administradores de los centros de cálculo sienten remordimientos cuando la CPU no está ejecutando trabajos durante todo el tiempo. Sin embargo, realmente no se trata de una buena métrica. Lo que realmente importa es cuántos trabajos por hora salen del sistema (rendimiento) y cuánto tiempo se requiere para que un trabajo se nos devuelva terminado (tiempo de retorno). Emplear como métrica la utilización de la CPU es igual que hacer la valoración de los coches basándonos en el número de revoluciones por hora del motor.

Para los sistemas interactivos, especialmente los sistemas en tiempo compartido y los servidores, se aplican diferentes objetivos a los ya vistos. El más importante es minimizar el **tiempo de respuesta**, es decir el tiempo entre que se introduce un comando y se obtiene el resultado. Sobre un ordenador personal en el que se está ejecutando un proceso de fondo (por ejemplo leer y almacenar el correo electrónico procedente de la red), cualquier petición del usuario para ejecutar un programa o abrir un fichero tiene precedencia sobre el proceso de fondo. El que todas las peticiones interactivas vayan antes se percibe como que se está dando un buen servicio.

Una cuestión relacionada es lo que puede denominarse la **proporcionalidad**. Los usuarios tienen una idea inherente (pero a menudo incorrecta) de cuánto tiempo deben tardar las cosas. Cuando una petición que se percibe complicada tarda mucho tiempo, los usuarios lo aceptan sin problemas, pero cuando una petición que se percibe como sencilla tarda mucho tiempo, los usuarios se irritan. Por ejemplo, si pinchando con el ratón sobre un ícono que llama a un proveedor de Internet utilizando un módem analógico se tarda 45 segundos en establecer la conexión, el usuario probablemente aceptará esto como una de esas cosas que tiene la vida. Por otro lado, si pinchando con el ratón sobre un ícono que cierra la conexión se tarda 45 segundos, el usuario probablemente estará a los 30 segundos profiriendo maldiciones, para acabar a los 45 segundos echando espuma por la boca. Este comportamiento se debe a la percepción del usuario común de que realizar una llamada de teléfono y establecer una conexión *se supone* que tarda mucho más tiempo que simplemente colgar el teléfono. En algunos casos (tales como este), el planificador no puede hacer nada para mejorar el tiempo de respuesta, pero en otros casos sí que puede, especialmente cuando la tardanza se debe a una pobre elección del orden de los procesos.

Los sistemas en tiempo real tienen diferentes propiedades que los sistemas interactivos, y por tanto diferentes objetivos de planificación. Los sistemas en tiempo real se caracterizan por tener límites de tiempo que deben, o al menos deberían, cumplirse. Por ejemplo, si un ordenador está controlando un dispositivo que produce datos a una velocidad constante, cualquier retraso en la ejecución del proceso que recoge los datos dentro del límite de tiempo establecido puede

provocar la pérdida de algunos datos. Por lo tanto el objetivo primordial en un sistema de tiempo real es cumplir con todos (o la mayoría) de los límites de tiempo especificados.

En algunos sistemas de tiempo real, especialmente en aquellos relacionados con la multimedia, es muy importante la predecibilidad. El incumplimiento ocasional de algunos límites de tiempo no resulta fatal, pero si el proceso de audio se ejecuta demasiado erráticamente, la calidad del sonido puede deteriorarse rápidamente. Lo mismo sucede con el vídeo, pero el oído es mucho más sensible a las perturbaciones que el ojo. La planificación de procesos debe ser altamente predecible y regular para conseguir evitar este problema. Vamos a estudiar los algoritmos de planificación en sistemas de batch e interactivos en este capítulo, pero vamos a diferir mayormente nuestro estudio de la planificación en los sistemas de tiempo real hasta que tratemos los sistemas operativos multimedia en el capítulo 7.

2.5.2 Planificación en Sistemas en Batch

Es ahora el momento de dejar las cuestiones de planificación más generales para tratar algoritmos de planificación específicos. En esta sección vamos a examinar los algoritmos utilizados en sistemas en batch. En las siguientes secciones examinaremos los algoritmos correspondientes a los sistemas interactivos y a los sistemas de tiempo real. Es necesario señalar que algunos algoritmos se utilizan tanto en los sistemas en batch como en los sistemas interactivos. Vamos a estudiar esos algoritmos más tarde, concentrándonos ahora en algoritmos que sólo son apropiados para sistema en batch.

Primero en Llegar Primero en Ser Servido

Probablemente el más simple de todos los algoritmos de planificación es el algoritmo no expulsor **primero en llegar primero en ser servido** (abreviadamente **FCFS** del inglés *First-Come First-Served*). Con este algoritmo, se asigna la CPU a los procesos respetando el orden en el que la han solicitado. Básicamente, existe una única cola de procesos preparados. Cuando por la mañana entra en el sistema el primer trabajo procedente del exterior, se le concede la CPU inmediatamente permitiéndosele que se ejecute durante todo el tiempo que quiera. A medida que van llegando nuevos trabajos, se van metiendo al final de la cola. Cuando el proceso en ejecución se bloquea, se ejecuta a continuación el primer proceso de la cola. Cuando un proceso bloqueado pasa a preparado, se mete al final de la cola de preparados como si fuera un trabajo recién llegado.

La gran ventaja de este algoritmo es que es fácil de entender e igualmente fácil de programar. También es justo en el mismo sentido que es justo que se asignen las últimas entradas de un partido o de un concierto a la gente que ha estado haciendo cola desde las 2 de la madrugada. Con este algoritmo, una única lista enlazada lleva la cuenta de todos los procesos preparados. Escoger un proceso para su ejecución requiere simplemente sacar un proceso del principio de la cola. Añadir un nuevo trabajo o un proceso que se ha desbloqueado no requiere más que incorporarlo al final de la cola. ¿Qué puede haber más simple?

Desafortunadamente, este algoritmo tiene también una gran desventaja. Supongamos que tenemos un proceso intensivo en CPU que se ejecuta en ráfagas de 1 segundo, y muchos procesos intensivos en E/S que utilizan poco tiempo de CPU pero requieren 1000 lecturas del disco para completarse. El proceso intensivo en CPU se ejecuta durante 1 segundo, para a continuación leer un bloque del disco. Ahora todos los procesos intensivos en E/S se ejecutan y ponen en marcha lecturas del disco. Cuando el proceso intensivo en CPU obtiene su bloque del disco se ejecuta durante otro segundo, seguido en rápida sucesión por todos los procesos intensivos en E/S.

El resultado neto es que cada proceso intensivo en E/S leerá un bloque por segundo y requerirá 1000 segundos para terminar. Sin embargo utilizando un algoritmo de planificación

que expulse al proceso intensivo en CPU cada 10 milisegundos, los procesos intensivos en E/S podrían terminar en 10 segundos en vez de los 1000 segundos, y esto sin ralentizar demasiado el proceso intensivo en CPU.

Primero el Trabajo más Corto

Vamos a fijarnos ahora en otro algoritmo no expulsor propio de sistemas en batch que asume que los tiempos de ejecución se conocen por anticipado. Por ejemplo, en una compañía de seguros es posible predecir con mucha precisión cuánto tiempo va a tardar en ejecutarse un batch de 1000 reclamaciones, puesto que todos los días se realiza un trabajo similar. Cuando están esperando para comenzar su ejecución varios trabajos de la misma importancia en la cola de entrada, el planificador escoge **primero el trabajo más corto** (abreviando SJF del inglés *Shortest Job First*). Véase la Figura 2-39. En ella encontramos cuatro trabajos *A*, *B*, *C* y *D* con tiempos de ejecución de 8, 4, 4 y 4 minutos, respectivamente. Pasándolos a ejecución en ese orden, se obtiene un tiempo de retorno para *A* de 8 minutos, para *B* de 12 minutos, para *C* de 16 minutos y para *D* de 20 minutos, obteniendo una media de 14 minutos.



Figura 2-39. Un ejemplo de planificación el trabajo más corto primero.

(a) Ejecutando cuatro trabajos en el orden original. (b) Ejecutando los mismos trabajos en el orden el trabajo más corto primero.

Vamos a considerar ahora la ejecución de esos cuatro trabajos utilizando la planificación el trabajo más corto primero, como se muestra en la Figura 2-39(b). Los tiempos de retorno que se obtienen ahora son 4, 8, 12 y 20 minutos resultando una media de 11 minutos. El algoritmo del trabajo más corto primero es probablemente óptimo. Consideremos el caso de cuatro trabajos, con tiempos de ejecución de *a*, *b*, *c* y *d*, respectivamente. El primer trabajo termina en el momento *a*, el segundo termina en el momento *a + b*, y así los demás. El tiempo de retorno medio es por tanto $(4a + 3b + 2c + d)/4$. Es claro que *a* contribuye más a la media que los demás tiempos de ejecución, por lo cual debe ser el trabajo más corto, con *b* a continuación, luego *c* y finalmente *d* como el más largo al afectar tan solo a su propio tiempo de retorno. Podemos aplicar igual de bien el mismo argumento a cualquier número de trabajos.

Es necesario señalar que el algoritmo del trabajo más corto el primero sólo es óptimo cuando todos los trabajos están disponibles simultáneamente. Como contraejemplo, consideremos cinco trabajos, de *A* hasta *E*, con tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3. Inicialmente, sólo podemos elegir a *A* o a *B*, ya que los otros tres trabajos no han llegado todavía. Utilizando el algoritmo del trabajo más corto el primero ejecutaremos los trabajos en el orden *A*, *B*, *C*, *D* y *E* obteniendo una espera media de 4,6. Sin embargo si los ejecutamos en el orden *B*, *C*, *D*, *E* y *A* la espera media obtenida es de 4,4.

Tiempo Restante más Corto a Continuación

El algoritmo denominado **tiempo restante más corto a continuación** (*shortest remaining time next*) es una versión expulsora del trabajo más corto el primero. Con este algoritmo, el planificador elige siempre el proceso al cual le queda menos tiempo de ejecución. Aquí también se requiere conocer el tiempo de ejecución por adelantado. Cuando llega un nuevo trabajo, se compara su tiempo total con el tiempo de ejecución que le queda al proceso

actual. Si el nuevo proceso necesita menos tiempo para terminar que el proceso actual, el proceso actual se suspende, arrancándose el nuevo trabajo. Este esquema permite que los nuevos procesos cortos reciban un buen servicio.

Planificación a Tres Niveles

Desde una cierta perspectiva, los sistemas en batch permiten planificar a tres niveles diferentes, como se ilustra en la Figura 2-40. Tan pronto como llegan los trabajos al sistema, se los coloca en un primer momento en una cola de entrada residente en el disco. El **planificador de admisión** decide qué trabajos se admiten en el sistema. Los demás trabajos se quedan en la cola de entrada hasta que sean seleccionados. Un algoritmo típico de control de la admisión puede tener como objetivo obtener una mezcla adecuada de trabajos intensivos en CPU y trabajos intensivos en E/S. De forma alternativa, el objetivo puede ser que los trabajos cortos puedan ser admitidos rápidamente mientras que los largos tengan que esperar. El planificador de admisión es libre para dejar algunos trabajos en la cola de entrada y admitir otros que llegan después, a su criterio.

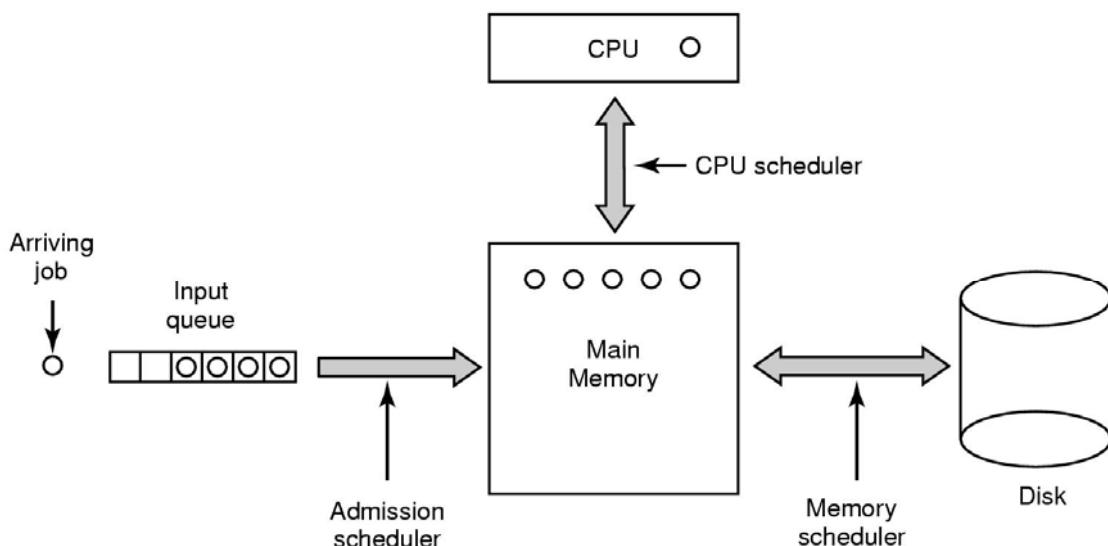


Figura 2-40. Planificación a tres niveles.

Una vez que un trabajo ha sido admitido en el sistema, es posible crear para él un proceso que pueda competir por la CPU. Sin embargo, puede también ocurrir que el número de procesos sea tan grande que no exista suficiente espacio para todos ellos en la memoria. En este caso, algunos de los procesos tienen que salir de la memoria guardándose en el área de intercambio del disco. El segundo nivel de planificación consiste en decidir qué procesos deben mantenerse en memoria y qué procesos deben mantenerse en el disco. Vamos a llamar a este planificador el **planificador de memoria**, ya que determina qué procesos van a estar en memoria y qué procesos van a estar en el disco.

Esta decisión debe revisarse frecuentemente para permitir a los procesos que están en el disco recibir algún servicio. Sin embargo, probablemente esta revisión no debe de realizarse más de una vez por segundo, o incluso menos, ya que cargar un proceso desde el disco resulta muy costoso. Si los contenidos de la memoria principal se transvasan de memoria al disco y del disco a memoria demasiado a menudo, se estará gastando una gran cantidad de ancho de banda del disco, ralentizando la E/S de ficheros.

Para optimizar el rendimiento del sistema en conjunto, el planificador de memoria debe decidir cuidadosamente cuantos procesos desea que residan en memoria, lo que se denomina el **grado de multiprogramación**, y qué tipo de procesos. Si tiene información sobre qué procesos son intensivos en CPU y cuáles son intensivos en E/S, puede tratar de mantener una mezcla adecuada de esos tipos de procesos en memoria. Como un enfoque muy crudo, si una cierta clase de proceso realiza cálculos durante el 20% del tiempo, mantener en torno a cinco de esos procesos en memoria es un número adecuado para tener a la CPU completamente ocupada. Vamos a examinar un modelo de multiprogramación ligeramente mejor en el capítulo 4 de gestión de memoria.

Para tomar sus decisiones, el planificador de memoria inspecciona periódicamente cada proceso en el disco para decidir si lo carga o no en memoria. Entre los criterios que puede utilizar para tomar esa decisión están los siguientes:

1. ¿Cuánto tiempo ha pasado desde que el proceso se intercambió al disco?
2. ¿Cuánto tiempo de CPU ha tenido el proceso recientemente?
3. ¿Qué grande es el proceso? (Los procesos pequeños no son problema)
4. ¿Cuán importante es el proceso?

El tercer nivel de planificación consiste realmente en escoger uno de los procesos preparados que hay en memoria para ejecutarlo a continuación. A menudo se le denomina el **planificador de la CPU** y es aquél en el que piensa toda la gente cuando se habla del “planificador”. Aquí puede utilizarse cualquier algoritmo apropiado, bien expulsor o no expulsor. Estos algoritmos incluyen los descritos anteriormente así como un número de algoritmos que van a describirse en la siguiente sección.

2.5.3 Planificación en Sistemas Interactivos

Vamos a ver ahora algunos algoritmos que pueden utilizarse en los sistemas interactivos. Todos ellos pueden utilizarse también como planificadores de la CPU en sistemas en batch. Mientras que la planificación a tres niveles no es posible aquí, sí que es posible e incluso común una planificación a dos niveles (planificador de memoria y planificador de la CPU). A continuación vamos a concentrarnos en el planificador de la CPU.

Planificación Round-Robin

Vamos a ver ahora algunos algoritmos de planificación específicos. Uno de los más antiguos, simples, justos y de uso más extendido es **round robin**. A cada proceso se le asigna un intervalo de tiempo, denominado su **quantum** (o también **rodaja de CPU**), durante el que se le permite ejecutarse. Si el proceso sigue ejecutándose todavía al final del quantum, se le expulsa de la CPU, concediéndole la CPU a algún otro proceso. Si el proceso se bloquea (o termina) antes de que transcurra el quantum se realiza por supuesto la comutación de la CPU. Round robin es fácil de implementar. Todo lo que necesita el planificador es mantener una lista de los procesos ejecutables, como se muestra en la Figura 2-41(a). Cuando el proceso consume su quantum, se le pone al final de la lista, como se muestra en la Figura 2-41(b).

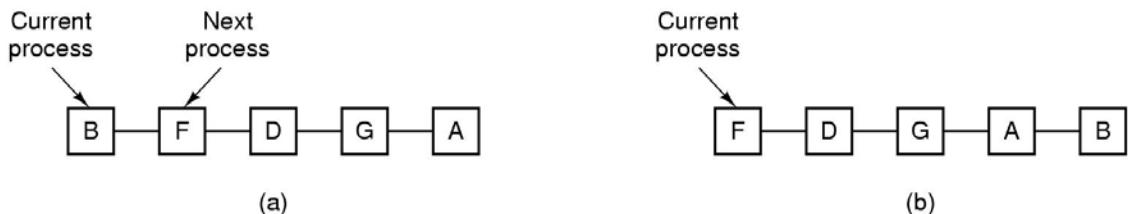


Figura 2-41. Planificación round-robin. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables después de que *B* haya utilizado su quantum.

La única cuestión interesante en relación con round robin es el tamaño del quantum. La conmutación de un proceso a otro requiere una cierta cantidad de tiempo para llevar a cabo la administración – salvar y cargar los registros y los mapas de memoria, actualizar diferentes tablas y listas, vaciar y recargar la memoria caché, etc. Supongamos que esta **conmutación de proceso o cambio de contexto**, como se denomina a veces, requiere 1 milisegundo, incluyendo la conmutación de los mapas de memoria, el vaciado y la recarga de la memoria caché, etc. Supongamos también que el quantum se ha establecido en 4 milisegundos. Con estos parámetros, después de hacer trabajo útil durante 4 milisegundos, la CPU debe gastar 1 milisegundo en la conmutación del proceso. De esta manera en el mejor de los casos el veinte por ciento del tiempo de la CPU se va a gastar en una sobrecarga administrativa. Claramente esto es demasiado.

Para mejorar la eficiencia de la CPU, podemos establecer el quantum en, pongamos 100 milisegundos. Ahora el tiempo gastado es sólo del 1 por ciento. Pero consideremos qué ocurre en un sistema en tiempo compartido si diez usuarios interactivos pulsan la tecla de retorno de carro casi al mismo tiempo. Diez procesos van a aparecer de repente en la lista de procesos ejecutables. Si la CPU está ociosa, el primer proceso comienza inmediatamente, el segundo no puede comenzar hasta 100 milisegundos después, y así sucesivamente. El desafortunado último proceso puede tener que esperar hasta 1 segundo antes de tener la posibilidad de ejecutarse, suponiendo que todos los demás procesos aprovechan completamente sus *quanta* (plural de quantum en latín). La mayoría de los usuarios podrían percibir una respuesta a un comando corto con una espera de 1 segundo como una respuesta muy lenta.

Otro factor a tener en cuenta es que si el quantum se establece más largo que la ráfaga de CPU media, entonces raramente ocurrirá ninguna expulsión. Por el contrario, la mayoría de los procesos realizarán una operación bloqueante antes de que el quantum se agote, provocando un cambio de proceso. Eliminar las expulsiones mejora el rendimiento debido a que los cambios de proceso sólo ocurren cuando son lógicamente necesarios, es decir, cuando un proceso se bloquea y no puede continuar.

La conclusión puede formularse como sigue: fijar un quantum demasiado corto provoca demasiados cambios de proceso y reduce la eficiencia de la CPU, pero fijar un quantum demasiado largo puede provocar un pobre tiempo de respuesta a peticiones interactivas cortas. Un quantum en torno de entre 20 y 50 milisegundos es a menudo un compromiso razonable.

Planificación por Prioridades

La planificación round robin supone tácitamente que todos los procesos son igualmente importantes. Frecuentemente, la gente que posee y opera un ordenador multiusuario tiene ideas muy diferentes al respecto. En una universidad, la jerarquía puede ser primero los decanos, luego los profesores, las secretarias, los conserjes y finalmente los estudiantes. La necesidad de tener en cuenta factores externos conduce a la **planificación por prioridades**. La idea básica es

trivial: a cada proceso se le asigna una prioridad, y siempre es el proceso ejecutable con mayor prioridad al que se le permite ejecutarse.

Incluso en un PC con un único propietario, puede haber múltiples procesos, algunos más importantes que otros. Por ejemplo, a un proceso demonio que envía el correo electrónico como proceso de fondo se le puede asignar una menor prioridad que a un proceso que visualiza en tiempo real una película de vídeo en la pantalla.

Para impedir que los procesos de mayor prioridad se ejecuten de manera indefinida, el planificador podría hacer decrecer la prioridad del proceso que se está ejecutando en cada tic del reloj (es decir, en cada interrupción de reloj). Si esta acción provoca que la prioridad descienda por debajo de la del siguiente proceso de mayor prioridad, tendrá lugar un cambio de proceso. Alternativamente, se podría asignar a cada proceso un quantum de tiempo máximo durante el cual puede ejecutarse. Una vez consumido ese quantum, el proceso con la siguiente prioridad más alta tendrá la oportunidad de ejecutarse.

Las prioridades pueden asignarse a los procesos de forma estática o dinámica. En un ordenador militar, los procesos iniciados por los generales podrían comenzar con prioridad 100; los iniciados por coronelos, con 90; los de los comandantes con 80; los de los capitanes con 70; los de los tenientes con 60, y así en forma sucesiva. Alternativamente, en un centro de cálculo comercial, los trabajos de alta prioridad podrían costar 100 dólares la hora; los de mediana prioridad, 75 dólares la hora, y los de baja prioridad 50 dólares la hora. El sistema UNIX tiene un comando, *nice*, que permite a un usuario reducir de manera voluntaria la prioridad de su proceso, a fin de ser amable con los demás usuarios. Nadie lo utiliza nunca.

El sistema también podría asignar prioridades de forma dinámica para conseguir ciertos objetivos del sistema. Por ejemplo, algunos procesos están completamente dedicados a hacer E/S y gastan la mayor parte de su tiempo esperando a que terminen las operaciones de E/S. Cada vez que un proceso de ese tipo quiera la CPU, debe concedérsela de inmediato para que pueda iniciar su siguiente solicitud de E/S, la cual puede proceder en paralelo con otro proceso que sí haga uso de la CPU. Hacer que los procesos intensivos en E/S esperen mucho tiempo por la CPU sólo consigue tenerlos ocupando la memoria durante un tiempo innecesariamente largo. Un algoritmo sencillo para dar un buen servicio a los procesos intensivos en E/S consiste en asignarles como prioridad el valor $1/f$, donde f es la fracción del último quantum que utilizó un proceso. Un proceso que utilizó sólo 1 milisegundo de su quantum de 50 milisegundos obtendría una prioridad de 50, mientras que uno que se ejecutó durante 25 milisegundos antes de bloquearse obtendría una prioridad de 2. Un proceso que consumió todo su quantum obtendría una prioridad de 1.

En muchos casos es conveniente agrupar los procesos en clases de prioridad y utilizar planificación por prioridades entre las clases, pero planificación round-robin dentro de cada clase. La Figura 2-42 muestra un sistema con cuatro clases de prioridad. El algoritmo de planificación es el siguiente: mientras haya procesos ejecutables en la clase de prioridad 4, se ejecutarán cada uno durante un quantum, al modo round-robin, sin ocuparse de las clases de menor prioridad. Si la clase 4 está vacía, se ejecutan los procesos de la clase 3 de forma round robin. Si las clase 4 y 3 están ambas vacías, se ejecuta la clase 2 de forma round robin, y así sucesivamente. Si no se realiza un ajuste de las prioridades ocasionalmente, puede suceder que las clases de menor prioridad sufren de inanición hasta morir.

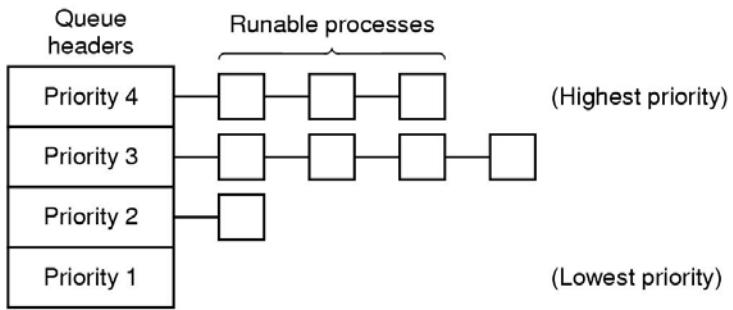


Figura 2-42. Un algoritmo de planificación con cuatro clases de prioridad.

Múltiples Colas

Uno de los primeros planificadores por prioridades se utilizó en el sistema CTSS (Corbató y otros, 1962). CTSS tenía el problema de que el cambio de proceso era muy lento porque el 7094 sólo podía contener un proceso en la memoria. Cada cambio de proceso implicaba llevar el proceso actual al disco y leer del disco el nuevo proceso. Los diseñadores del CTSS enseguida se dieron cuenta de que era más eficiente conceder a los procesos intensivos en CPU un quantum grande de una vez, en lugar de darles pequeños quanta más frecuentemente (con el fin de reducir el intercambio). Por otra parte, dar a todos los procesos un quantum grande aumentaría el tiempo de respuesta como ya vimos. Su solución fue establecer clases de prioridad. Los procesos de la clase más alta se ejecutaban durante un quantum, los procesos de la siguiente clase más alta se ejecutaban durante dos quanta. Los procesos de la siguiente clase se ejecutaban durante cuatro quanta, y así sucesivamente. Cada vez que un proceso se gastaba todos los quanta que se le habían asignado, se le bajaba a la clase inmediatamente inferior.

Por ejemplo, consideremos un proceso que necesitaba la CPU de manera continuada durante 100 quanta. En un principio se le concedía un quantum, después del cual se le intercambiaba al disco. La siguiente vez recibía dos quanta antes de ser intercambiado al disco. En las siguientes ejecuciones recibía 4, 8, 16, 32 y 64 quanta, aunque sólo usaba 37 de los 64 quanta finales para terminar su trabajo. Sólo se necesitaban siete intercambios (incluida la carga inicial) en vez de los 100 que se necesitarían con un algoritmo round robin puro. Además a medida que el proceso bajaba de nivel en las colas de prioridad, se le ejecutaba con una frecuencia cada vez menor, pudiéndose dedicar la CPU a otros procesos interactivos cortos.

Se adoptó la siguiente política para evitar que un proceso que en un principio necesitaba ejecutarse durante un tiempo largo, pero que después se volvía interactivo, fuera castigado de forma indefinida. Cada vez que se oprimía la tecla de retorno de carro en un terminal, el proceso perteneciente a ese terminal se pasaba a la clase de más alta prioridad, bajo el supuesto de que estaba a punto de volverse interactivo. Un buen día, un usuario que tenía un proceso intensivo en CPU descubrió que si oprimía la tecla de retorno de carro de su terminal a intervalos aleatorios, cada pocos segundos, conseguía maravillas en su tiempo de respuesta. Ese usuario comunicó su descubrimiento a todos sus amigos. La moraleja de la historia es: hacer que las cosas funcionen en la práctica es mucho más difícil que hacer que funcionen en principio.

Se han usado muchos otros algoritmos para asignar procesos a clases de prioridad. Por ejemplo, el influyente sistema XDS 940 (Lampson, 1968), construido en Berkeley, tenía cuatro clases de prioridad llamadas terminal, E/S, quantum corto y quantum largo. Cuando un proceso que estaba esperando entradas de un terminal por fin se despertaba, se le colocaba en la clase de más alta prioridad (terminal). Cuando un proceso esperaba un bloqueo de disco pasaba al

estado preparado, se le colocaba en la segunda clase. Si un proceso seguía ejecutable cuando su quantum expiraba, se le colocaba inicialmente en la tercera clase. Pero si un proceso gastaba su quantum muchas veces seguidas sin bloquearse por el terminal u otro tipo de E/S, se le pasaba a la cola más baja. Muchos otros sistemas utilizan algo parecido para favorecer a los usuarios y procesos interactivos a expensas de los de segundo plano.

Proceso más Corto a Continuación

Debido a que el trabajo más corto primero siempre produce el mínimo tiempo de respuesta medio en los sistemas en batch, resultaría bonito poder utilizarlo también con procesos interactivos. Hasta cierto punto, tal cosa es posible. Por lo general, los procesos interactivos siguen el patrón de esperar por un comando, ejecutar el comando, esperar un comando, ejecutar el comando, y así sucesivamente. Si vemos la ejecución de cada comando como un “trabajo” separado, podríamos minimizar el tiempo de respuesta global ejecutando el más corto primero. El único problema es determinar cuál de los procesos preparados es el más corto.

Una estrategia consiste en hacer estimaciones basadas en el comportamiento anterior y ejecutar el proceso que tenga el tiempo de ejecución estimado más corto. Supongamos que el tiempo estimado por comando para un terminal dado es T_0 . Ahora supongamos que la siguiente ejecución de un comando proveniente de ese terminal tarda T_1 . Podríamos actualizar nuestra estimación calculando una suma ponderada de estas dos estimaciones, es decir, $aT_0 + (1 - a)T_1$. Mediante la elección del valor de a , podemos decidir que el proceso de estimación olvide rápidamente las ejecuciones pasadas o que las recuerde durante mucho tiempo. Con $a = 1/2$, obtenemos las siguientes estimaciones sucesivas

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Después de tres nuevas ejecuciones, el peso de T_0 en la vigente estimación habrá bajado a 1/8.

La técnica de estimar el siguiente valor de una serie, calculando la media ponderada del último valor medido y su estimación anterior, se conoce como **envejecimiento** (*aging*), y puede aplicarse en muchas situaciones en las que es preciso hacer una predicción con base en los valores anteriores. El envejecimiento es especialmente fácil de aplicar cuando $a = 1/2$, pues basta con sumar el valor nuevo a su estimación y dividir la suma por 2 (desplazándola un bit a la derecha).

Planificación Garantizada

Un enfoque a la planificación completamente diferente es la de hacer promesas reales a los usuarios sobre el rendimiento y luego cumplirlas en la práctica. Una promesa que es realista y fácil de cumplir es la siguiente: si hay n usuarios conectados, cada uno recibirá aproximadamente $1/n$ de la potencia de la CPU. Similarmente, en un sistema monousuario en el que se están ejecutando n procesos, si todos los demás factores son iguales, cada uno deberá recibir un $1/n$ de los ciclos de CPU.

Para cumplir esta promesa, el sistema necesita llevar la cuenta de cuánto tiempo de CPU ha recibido cada proceso desde su creación. Luego se calcula el tiempo de CPU al que cada uno tiene derecho, que sería el tiempo desde la creación dividido entre n . Puesto que se conoce el tiempo de CPU que ha utilizado cada proceso, se puede calcular el cociente del tiempo de CPU consumido realmente entre el tiempo al que el proceso tiene derecho. Un cociente de 0,5 significa que el proceso sólo ha recibido la mitad de lo que debería haber recibido, y un cociente de 2,0 significa que el proceso ha recibido el doble de lo que le correspondía. El algoritmo consiste entonces en ejecutar el proceso cuyo cociente es más bajo, hasta que su cociente rebase al de su competidor más cercano.

Planificación por Lotería

Aunque hacer promesas a los usuarios y luego cumplirlas es una idea excelente, no es fácil de implementar. Sin embargo, es posible utilizar otro algoritmo para obtener resultados igual de predecibles con una implementación mucho más sencilla: el de la **planificación por lotería** (Waldspurger y Weihl, 1994).

La idea básica consiste en dar a los procesos como “décimos de lotería” para los distintos recursos del sistema, tales como el tiempo de CPU. Siempre que deba tomarse una decisión de planificación, se escoge un décimo al azar, concediéndose el recurso al proceso que lo tenga. Si este sistema se aplica a la planificación de la CPU, el sistema podría celebrar un “sorteo” 50 veces por segundo, dando como premio al ganador los siguientes 20 milisegundos de tiempo de CPU.

Parafraseando a George Orwell: “Todos los procesos son iguales, pero algunos son más iguales que otros”. Podríamos dar más décimos a los procesos más importantes para aumentar sus posibilidades de ganar. Si se emiten 100 décimos y un proceso tiene 20 de ellos, tendrá una probabilidad del 20% de ganar cada sorteo. A la larga, este proceso recibirá aproximadamente un 20% de la CPU. En contraste con un planificador por prioridades, donde es muy difícil establecer lo que significa tener una prioridad de 40, aquí la regla está muy clara: un proceso poseyendo una fracción f de los décimos obtendrá una fracción f del recurso en cuestión.

La planificación por lotería tiene varias propiedades interesantes. Por ejemplo, si llega un nuevo proceso y recibe cierto número de décimos, en el siguiente sorteo tendrá una probabilidad de ganar proporcional al número de décimos que tenga. Dicho de otro modo, la planificación por lotería es muy sensible en el sentido de que responde muy rápidamente a los cambios.

Los procesos que cooperan pueden intercambiar décimos si lo desean. Por ejemplo, si un proceso cliente envía un mensaje a un proceso servidor y luego se bloquea, podría entregar todos sus décimos al servidor para mejorar la probabilidad de que sea el servidor quien se ejecute a continuación. Cuando el servidor termine, devolverá los décimos al cliente para que pueda ejecutarse otra vez. De hecho, si no hay clientes, los servidores no necesitan décimos.

La planificación por lotería puede utilizarse para resolver problemas difíciles de manejar con otros métodos. Un ejemplo es un servidor de vídeo en el que varios procesos alimentan flujos de vídeo a sus clientes, pero con diferentes frecuencias de visualización. Supongamos que los procesos necesitan imágenes a razón de 10, 20 y 25 imágenes por segundo. Si se asignan a tales procesos 10, 20 y 25 décimos respectivamente, se repartirán en forma automática la CPU en la proporción correcta, es decir 10 : 20 : 25.

Planificación por Reparto Justo

Hasta aquí hemos dado por hecho que cada proceso se planifica por sus propios méritos sin considerar quién es su dueño. El resultado es que si el usuario 1 inicia 9 procesos y el usuario 2 inicia sólo un proceso, y se utiliza round robin o prioridades estáticas, el usuario 1 recibirá el 90% del tiempo de CPU y el usuario 2 recibirá sólo el 10%.

A fin de prevenir esa situación, algunos sistemas toman en cuenta quien es el propietario del proceso antes de planificarlo. En este modelo, a cada usuario se le asigna cierta fracción del tiempo de CPU y el planificador escoge los procesos de manera que se respete ese reparto. Por ejemplo, si a dos usuarios se les prometió el 50% del tiempo de CPU, cada uno recibirá esa fracción, sin importar cuántos procesos cree cada uno.

Consideremos el caso de un sistema con dos usuarios, a cada uno de los cuales se le ha prometido el 50% de la CPU. El usuario 1 tiene cuatro procesos, A, B, C y D , y el usuario 2 sólo tiene un proceso, E . Si se emplea la planificación round robin, la siguiente sería una posible secuencia de planificación que se ajusta a todas las restricciones:

A E B E C E D E A E B E C E D E . . .

Por otra parte, si el usuario 1 tiene derecho al doble de tiempo de CPU que el usuario 2, podríamos tener

A B E C D E A B E C D E . . .

Por supuesto, existen muchas otras posibilidades, y pueden explotarse, dependiendo del concepto de justicia que se utilice.

2.5.4 Planificación en Sistemas de Tiempo Real

Un sistema de tiempo real es uno en el cual el tiempo juega un papel esencial. Típicamente, se tiene uno o más dispositivos físicos externos al ordenador que generan estímulos a los cuales debe reaccionar el ordenador de la manera apropiada y dentro de un plazo de tiempo prefijado. Por ejemplo, el ordenador interno de un reproductor de discos compactos recibe los bits tal y como salen de la unidad y debe convertirlos en música en un intervalo de tiempo muy ajustado. Si el cálculo tarda demasiado, la música sonará rara. Otros sistemas en tiempo real monitorizan pacientes en la unidad de cuidados intensivos de un hospital, controlan el piloto automático de un avión y controlan los robots en una fábrica automatizada. En todos estos casos, producir la respuesta correcta demasiado tarde es a menudo tan malo como no producir ninguna respuesta.

Los sistemas en tiempo real se clasifican generalmente en sistemas de **tiempo real estricto** (*hard real time*) y sistemas de **tiempo real moderado** (*soft real time*). En los sistemas de tiempo real estricto hay plazos absolutos que deben cumplirse, pase lo que pase. En los sistemas de tiempo real moderado el incumplimiento ocasional de un plazo aunque es indeseable, es sin embargo tolerable. En ambos casos, el comportamiento en tiempo real se logra dividiendo el programa en varios procesos cuyo comportamiento es predecible y conocido por adelantado. Generalmente, tales procesos son cortos y pueden terminar su trabajo en mucho menos de un segundo. Cuando se detecta un suceso externo, el planificador debe planificar los procesos de tal modo que se cumplan todos los plazos.

Los sucesos a los que un sistema de tiempo real debe tener que responder pueden clasificarse como **periódicos** (que se presentan a intervalos regulares) o **aperiódicos** (cuya ocurrencia es impredecible). Un sistema puede tener que responder a múltiples flujos de sucesos periódicos. Dependiendo de cuánto tiempo se requiere para procesar cada suceso, podría no ser siquiera posible atenderlos a todos. Por ejemplo, si hay m sucesos periódicos y el suceso i tiene lugar con un periodo P_i y su tratamiento requiere C_i segundos de tiempo de CPU, entonces el sistema sólo podrá soportar esa carga si

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Un sistema en tiempo real que cumpla este criterio se dice que es **planificable**.

Por ejemplo, consideremos un sistema de tiempo real moderado con tres sucesos periódicos, con períodos de 100, 200 y 500 milisegundos, respectivamente. Si el tratamiento de estos sucesos requiere 50, 30 y 100 milisegundos de tiempo de CPU respectivamente, el sistema es planificable porque $0,5 + 0,15 + 0,2 < 1$. Si se añade un cuarto suceso con un período de 1 segundo, el sistema seguirá siendo planificable, en tanto que ese proceso no necesite más de 150 milisegundos de tiempo de CPU para su tratamiento. En este cálculo está implícita la suposición de que la sobrecarga por el cambio del contexto de los procesos es tan pequeña que puede ignorarse.

Los algoritmos de planificación para tiempo real pueden ser estáticos o dinámicos. Los primeros toman sus decisiones de planificación antes de que el sistema comience a ejecutarse. Los segundos toman las decisiones en tiempo de ejecución. La planificación estática sólo funciona si se está perfectamente informado por anticipado sobre el trabajo que debe realizarse y los plazos que deben cumplirse. Los algoritmos de planificación dinámica no tienen estas restricciones. Vamos a diferir nuestro estudio de algoritmos específicos hasta que tratemos los sistemas de tiempo real multimedia en el capítulo 7.

2.5.5 Política Frente a Mecanismo

Hasta ahora, hemos supuesto tácitamente que todos los procesos del sistema pertenecen a usuarios diferentes y, que por lo tanto, están compitiendo por la CPU. Aunque muchas veces eso es cierto, hay ocasiones en las que un proceso tiene muchos hijos ejecutándose bajo su control. Por ejemplo, un proceso de un sistema de gestión de una base de datos podría tener muchos hijos. Cada hijo podría estar trabajando sobre una petición distinta, o cada uno podría tener una función específica que realizar (análisis de consultas, acceso al disco, etc.). Es muy posible que el proceso principal sepa exactamente cuáles de sus hijos son más importantes (o cuáles son críticos en tiempo), y cuáles lo son menos. Desafortunadamente, ninguno de los planificadores discutidos anteriormente acepta información de los procesos de usuario sobre sus decisiones de planificación. Como resultado, el planificador raramente realiza la mejor elección posible.

La solución a este problema es separar el **mecanismo de planificación de la política de planificación**. Lo que esto significa es que el algoritmo de planificación está parametrizado de alguna manera, pudiendo los procesos de usuario especificar los parámetros concretos que deben utilizarse. Vamos a considerar otra vez el ejemplo de la base de datos. Supongamos que el núcleo utiliza un algoritmo de planificación por prioridades pero ofrece una llamada al sistema con la cual un proceso puede establecer (y modificar) las prioridades de sus hijos. De esta manera, el padre puede controlar en detalle la forma en la que se planifican sus hijos, incluso aunque él mismo no sea quien realice la planificación. En este caso, el mecanismo está en el núcleo pero la política la establece un proceso de usuario.

2.5.6 Planificación de Threads

Cuando varios procesos tienen cada uno múltiples threads, tenemos dos niveles de paralelismo: procesos y threads. La planificación en tales sistemas presenta notables diferencias dependiendo de si se soportan los threads a nivel de usuario o a nivel del núcleo (o si se dan ambas posibilidades).

Consideremos primero los threads a nivel de usuario. Puesto que el núcleo no sabe nada de la existencia de los threads, el núcleo opera como lo hace siempre, escogiendo un proceso, por ejemplo *A*, y cediéndole el control durante su quantum de tiempo. El thread planificador dentro de *A* es el que decide qué thread se ejecuta, por ejemplo *A1*. Puesto que no hay interrupciones de reloj para multiprogramar los threads, este thread puede seguir ejecutándose todo el tiempo que quiera. Si consume todo el quantum del proceso, el núcleo seleccionará otro proceso para ejecutar.

Cuando finalmente el proceso *A* vuelve a ejecutarse, el thread *A1* reanudará su ejecución y continuará consumiendo todo el tiempo de *A* hasta que termine. Sin embargo, su comportamiento antisocial no afectará a los demás procesos, los cuales recibirán lo que el planificador considere que les corresponde, sin importar lo que esté sucediendo dentro del proceso *A*.

Consideremos ahora el caso en el que los threads de *A* tienen relativamente poco trabajo que hacer por cada ráfaga de CPU, por ejemplo 5 milisegundos de trabajo dentro de un quantum de 50 milisegundos. Consecuentemente, cada thread se ejecutará durante un periodo corto de tiempo para luego ceder la CPU al thread planificador. Esto podría dar lugar a la secuencia *A1, A2, A3, A1, A2, A3, A1, A2, A3, A1* antes de que el núcleo comute al proceso *B*. Esta situación se ilustra en la Figura 2-43(a).

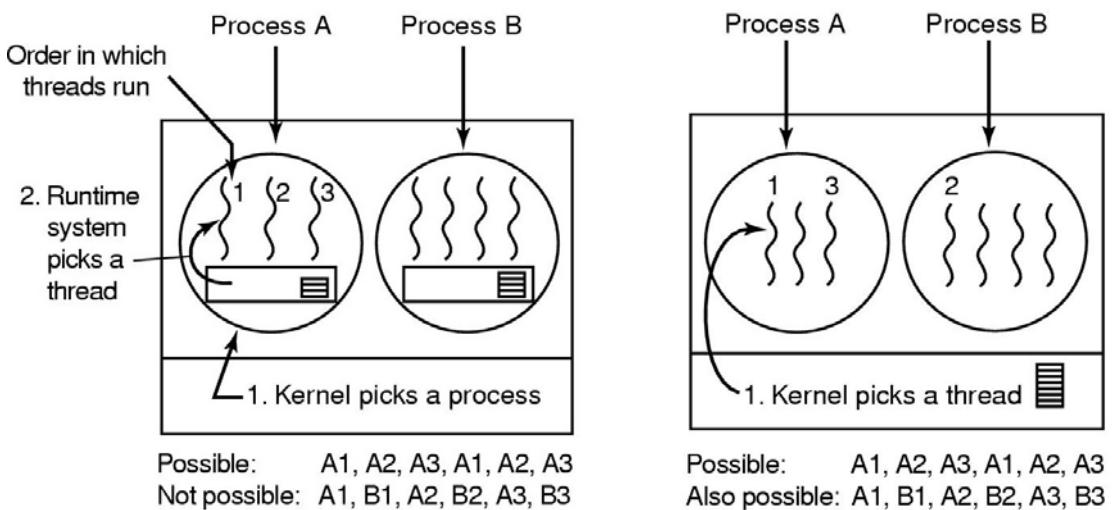


Figura 2-43. (a) Una posible planificación de threads a nivel de usuario con un quantum de proceso de 50 milisegundos y threads que se ejecutan durante 5 milisegundos en cada ráfaga de CPU. (b) Una posible planificación de threads a nivel del núcleo con las mismas características que en (a).

El algoritmo de planificación utilizado por el sistema en tiempo de ejecución puede ser cualquiera de los que hemos descrito anteriormente. En la práctica, los más comunes son la planificación round-robin y la planificación por prioridades. La única restricción es la ausencia de un reloj para interrumpir a un thread que ha estado ejecutándose durante demasiado tiempo.

Consideremos ahora la situación con threads a nivel del núcleo. Aquí el núcleo escoge un thread específico para ejecutar. No es obligatorio que tome en cuenta a qué proceso pertenece el thread, pero puede hacerlo si lo desea. Se concede un quantum al thread y se fuerza la suspensión del thread cuando el quantum se agote. Con un quantum de 50 milisegundos y threads que se bloquean después de cada 5 milisegundos, el orden de los threads durante un periodo dado de 30 milisegundos podría ser *A1, B1, A2, B2, A3, B3*, lo cual no sería posible con estos parámetros si los threads estuvieran en el nivel del usuario. Esta situación se ilustra, en parte, en la Figura 2-43(b).

La mayor diferencia entre los threads a nivel de usuario y a nivel del núcleo es el rendimiento. Una conmutación de threads a nivel de usuario requiere un puñado de instrucciones de lenguaje máquina. En el caso de los threads a nivel del núcleo se requiere un cambio de contexto completo, cambiando el mapa de memoria e invalidando la caché, lo que

resulta en una comutación varios órdenes de magnitud más lenta. Por otra parte, si se usan threads a nivel del núcleo y uno se bloquea esperando por una E/S, no se bloquea todo el proceso, como sí sucede con los threads a nivel de usuario.

Puesto que el núcleo sabe que comutar de un thread del proceso *A* a uno del proceso *B* es más costoso que ejecutar un segundo thread del proceso *A* (porque hay que cambiar el mapa de memoria y vaciar la caché), puede tomar esa información en cuenta para tomar una decisión. Por ejemplo, si hay dos threads de la misma importancia, pero uno de ellos pertenece al mismo proceso al que pertenecía un thread que acaba de bloquearse, y el otro pertenece a un proceso distinto, se debería dar preferencia al primero.

Otro factor importante es que los threads a nivel de usuario pueden utilizar un planificador de threads específico para la aplicación. Por ejemplo, consideremos el servidor web de la Figura 2-10. Supongamos que un thread trabajador acaba de bloquearse y que el thread despachador y dos threads trabajadores están listos. ¿Qué thread debería ejecutarse a continuación? El sistema en tiempo de ejecución, knowing lo que hace cada thread, puede fácilmente seleccionar al despachador para que éste a su vez pueda poner en marcha a otro thread trabajador. Esta estrategia maximiza la cantidad de paralelismo en un entorno en el que los threads trabajadores se bloquean con frecuencia en espera de E/S del disco. Con threads a nivel del núcleo, el núcleo nunca puede saber lo que hace cada thread (aunque es posible asignar a los threads diferentes prioridades). En general, los planificadores de threads específicos de la aplicación pueden afinar la aplicación mucho mejor que el núcleo.

2.6 INVESTIGACIÓN SOBRE PROCESOS Y THREADS

En el capítulo 1 mencionamos algunas de las investigaciones actuales sobre la estructura de los sistemas operativos. En este capítulo y los que siguen veremos investigaciones concentradas sobre aspectos mas concretos, comenzando con los procesos. Como irá quedando claro, algunos temas están sufriendo muchos menos cambios que otros. La mayoría de las investigaciones tienden a ser sobre temas nuevos, más que sobre los que se han estudiado durante décadas.

El concepto de proceso es un ejemplo de algo que está bien establecido. Casi todos los sistemas tienen una noción de proceso como contenedor para agrupar recursos relacionados, tales como un espacio de direcciones, threads, ficheros abiertos, permisos de protección, etc. Diferentes sistemas realizan el agrupamiento de formas ligeramente distintas, pero se trata simplemente de diferencias ingenieriles. La idea básica ya no da pie a muchas controversias y hay pocas investigaciones nuevas sobre el tema.

Los threads son una idea más reciente que los procesos, por lo que todavía se están realizando investigaciones sobre ellos. Hauser y otros (1993) examinaron la forma en la que los programas reales utilizan los threads y se encontraron con 10 paradigmas diferentes del uso de los threads. La planificación de threads (tanto en sistemas monoprocesador como multiprocesador) sigue siendo el tema favorito de algunos investigadores (Blufome y Leiserson, 1994; Buchanan y Chien, 1997; Corbalán y otros, 2000; Chandra y otros 2000; Duda y Cheriton, 1999; Ford y Susarla, 1996; y Petrou y otros, 1999). Sin embargo, pocos diseñadores de sistemas reales se pasan el día angustiados por la falta de un algoritmo de planificación de threads decente, de manera que aparentemente este tipo de investigaciones se realizan más por propia iniciativa de los investigadores que por la existencia de una demanda efectiva.

La sincronización y la exclusión mutua están estrechamente relacionadas con los threads. En los años setenta y ochenta, estos temas se investigaron hasta el agotamiento, por lo que ahora no se está trabajando mucho al respecto, y lo que se está haciendo tiende a concentrarse en el rendimiento (por ejemplo, Liedtke, 1993), en herramientas para detectar errores de sincronización (Savage y otros, 1997), o en nuevas modificaciones de conceptos

antiguos (Tai y Carver, 1996; Trono, 2000). Finalmente, siguen produciéndose y apareciendo informes de paquetes de threads que cumplen con el estándar POSIX (Alfieri, 1994; Millar, 1999).

2.7 RESUMEN

Con la finalidad de ocultar los efectos de las interrupciones, los sistemas operativos proporcionan un modelo conceptual consistente en procesos secuenciales que se ejecutan en paralelo. Los procesos pueden crearse y destruirse dinámicamente. Cada proceso tiene su propio espacio de direcciones.

Resulta útil en algunas aplicaciones disponer de múltiples threads de control dentro de un único proceso. Estos threads se planifican de manera independiente y cada uno tiene su propia pila, pero todos los threads de un proceso comparten un espacio de direcciones común. Los threads pueden implementarse en el espacio de usuario o en el núcleo.

Los procesos pueden comunicarse entre sí empleando primitivas de comunicación entre procesos, como semáforos, monitores o mensajes, que sirven para garantizar que nunca haya dos procesos en sus regiones críticas al mismo tiempo, situación que conduce al caos. Un proceso puede estar en ejecución, preparado o bloqueado, y puede cambiar de estado cuando él u otro proceso ejecute una de las primitivas de comunicación entre procesos. La comunicación entre threads es similar.

Las primitivas de comunicación entre procesos pueden servir para resolver problemas como el del productor-consumidor. Incluso con estas primitivas, hay que tener cuidado para evitar errores e interbloqueos.

Se conocen muchos algoritmos de planificación. Algunos se usan primordialmente en sistemas de batch, como el del trabajo más corto el primero. Otros son comunes en sistemas tanto de batch como en sistemas interactivos, e incluyen la planificación round robin, por prioridades, colas multinivel, planificación garantizada, planificación por lotería y planificación por reparto justo. Algunos sistemas separan con claridad el mecanismo de planificación de la política de planificación, lo que permite a los usuarios tener el control del algoritmo de planificación.

3

GESTIÓN DE MEMORIA

La memoria es un recurso importante que debe ser cuidadosamente gestionado. Aunque el ordenador doméstico medio de nuestros días tiene miles de veces más memoria que el IBM 7094 (el mayor ordenador del mundo a principios de la década de los años sesenta) el tamaño de los programas está creciendo mucho más rápido que el tamaño de las memorias. Para parafrasear la ley de Parkinson: “Los programas se expanden hasta llenar toda la memoria disponible para contenerlos”. En este capítulo vamos a estudiar la forma en la que los sistemas operativos gestionan la memoria.

Idealmente a todo programador le gustaría poder contar con una memoria infinitamente grande, infinitamente rápida y que fuese además no volátil, esto es, que no perdiese su contenido en ausencia de energía eléctrica. Llegados aquí, ¿porqué no pedir además que esa memoria sea también suficientemente barata? Desafortunadamente la tecnología no proporciona tales memorias. Consecuentemente, la mayoría de los ordenadores disponen de una **jerarquía de memoria**, con una pequeña cantidad de memoria caché muy rápida, cara y volátil, decenas de megabytes de memoria principal (RAM) moderadamente rápida, moderadamente cara y volátil, y decenas o cientos de gigabytes de memoria de disco lenta, barata y no volátil. Corresponde al sistema operativo coordinar la utilización de esos tres tipos de memoria.

La parte del sistema operativo que gestiona la jerarquía de memoria se denomina el **gestor de memoria**. Su trabajo es seguir la pista de qué partes de la memoria están en uso y cuáles no lo están, con el fin de poder asignar memoria a los procesos cuando la necesiten, y recuperar esa memoria cuando dejen de necesitarla, así como gestionar el intercambio entre memoria principal y el disco cuando la memoria principal resulte demasiado pequeña para contener a todos los procesos.

En este capítulo vamos a investigar diferentes esquemas de gestión de memoria, que van desde los más simples hasta los más sofisticados. Vamos a comenzar por el principio, fijándonos primero en el sistema de gestión de memoria más sencillo posible para luego ir progresando gradualmente a gestores de memoria más y más elaborados.

Como señalamos en el Capítulo 1, la historia tiende a repetirse ella misma en el mundo de los ordenadores. Así, aunque los esquemas de gestión de memoria más sencillos ya no se utilizan en los ordenadores personales, siguen utilizándose todavía en algunos asistentes personales (*palmtops*), sistemas empotrados y sistemas de tarjeta inteligente. Por este motivo, es necesario estudiarlos todavía.

4.1 GESTIÓN DE MEMORIA BÁSICA

Los sistemas de gestión de memoria pueden dividirse en dos clases: los que mueven procesos de la memoria principal al disco y del disco a la memoria principal durante su ejecución (intercambio y paginación), y los que no lo hacen. Los segundos son más sencillos, por lo que vamos a estudiarlos primero. Más tarde en el capítulo examinaremos el intercambio y la paginación. A lo largo de este capítulo el lector debe tener presente que el intercambio y la paginación son principalmente mecanismos artificiales motivados por la falta de memoria principal suficiente para contener todos los programas a la vez. Si la memoria principal llegara a ser tan grande que siempre hubiera la suficiente, los argumentos a favor de un tipo de esquema de gestión de memoria u otro podrían volverse obsoletos.

Por otra parte, como ya mencionamos anteriormente, el software parece estar creciendo incluso con más rapidez que la memoria, por lo que es posible que siempre se necesite una gestión de memoria eficiente. En la década de 1980, muchas universidades ejecutaban un sistema de tiempo compartido con docenas de usuarios (más o menos satisfechos) sobre un ordenador VAX de tan solo 4 MB. En la actualidad, Microsoft recomienda tener por lo menos 64 MB para un sistema Windows 2000 monousuario. La tendencia hacia la multimedia demanda mayores cantidades de memoria, así que probablemente se seguirá necesitando una buena gestión de memoria, al menos, durante la próxima década.

4.1.1 Monoprogramación sin Intercambio ni Paginación

El esquema de gestión de memoria más sencillo posible consiste en ejecutar sólo un programa a la vez, repartiendo la memoria entre ese programa y el sistema operativo. En la Figura 4-1 se muestran tres variaciones de este tema. El sistema operativo podría estar en el fondo de la memoria RAM (*Random Access Memory*), como se muestra en la Figura 4.1(a), o podría estar en ROM (*Read-Only Memory*) en lo alto de la memoria, como se muestra en la Figura 4-1(b), o los drivers de los dispositivos podrían estar en la parte más alta de la memoria en una ROM y el resto del sistema en la parte baja de la RAM, como en la Figura 4-1(c). El primer modelo se utilizó antiguamente en mainframes y miniordenadores pero actualmente es muy raro su uso. El segundo modelo se usa en algunos ordenadores palmtop y en sistemas empotrados. El tercer modelo se usó en los primeros ordenadores personales (ejecutando por ejemplo MS-DOS), donde la parte del sistema que está en ROM se denomina el **BIOS** (*Basic Input Output System*).

Cuando el sistema está organizado de esta manera, sólo puede ejecutarse un proceso a la vez. Tan pronto como el usuario teclea un comando, el sistema operativo copia el programa solicitado del disco a la memoria y lo ejecuta. Cuando el proceso termina, el sistema operativo muestra un carácter de *prompt* y espera por un nuevo comando. Cuando recibe el comando, carga un nuevo programa en la memoria, sobrescribiendo el primero.

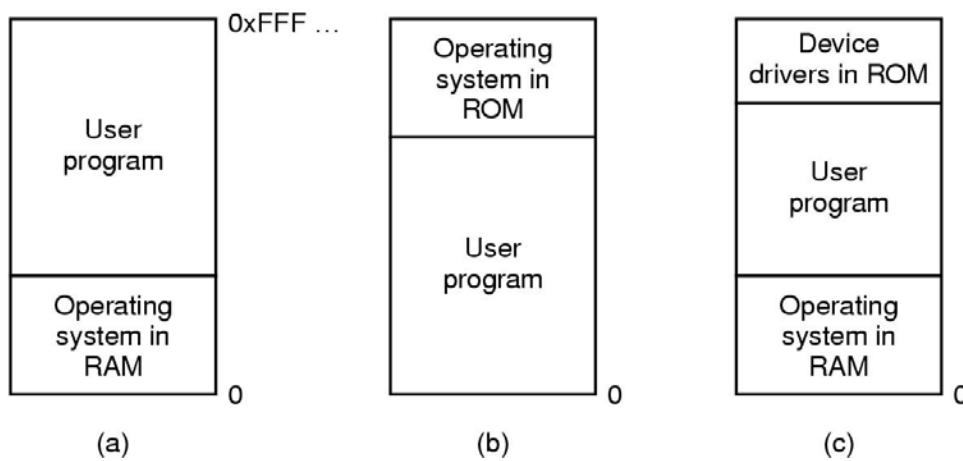


Figura 4-1. Tres formas de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

4.1.2 Multiprogramación con Particiones Fijas

Con la excepción de los sencillos sistemas empotrados, la monoprogramación está ya absolutamente en desuso. La mayoría de los sistemas modernos permiten la ejecución de múltiples procesos al mismo tiempo. El tener múltiples procesos ejecutándose a la vez significa que cuando un proceso se bloquea esperando a que termine una operación de E/S, otro proceso puede seguir haciendo uso de la CPU. Así la multiprogramación aumenta la utilización de la CPU. Los servidores de red siempre han tenido la capacidad de ejecutar múltiples procesos (para diferentes clientes) al mismo tiempo, pero en la actualidad la mayoría de las máquinas clientes (es decir, de escritorio) también cuentan con esta capacidad.

La forma más fácil de conseguir la multiprogramación es simplemente dividir la memoria en n particiones (posiblemente de diferentes tamaños). Esta división puede, por ejemplo, realizarse manualmente cuando se pone en marcha el sistema.

Cuando llega un trabajo, puede colocarse en la cola de entrada de la partición más pequeña en la que cabe. Puesto que en este esquema las particiones son fijas, cualquier espacio en una partición no utilizado por un trabajo se desperdicia. En la Figura 4-2(a) vemos el aspecto que tiene este sistema de particiones fijas y colas de entrada separadas.

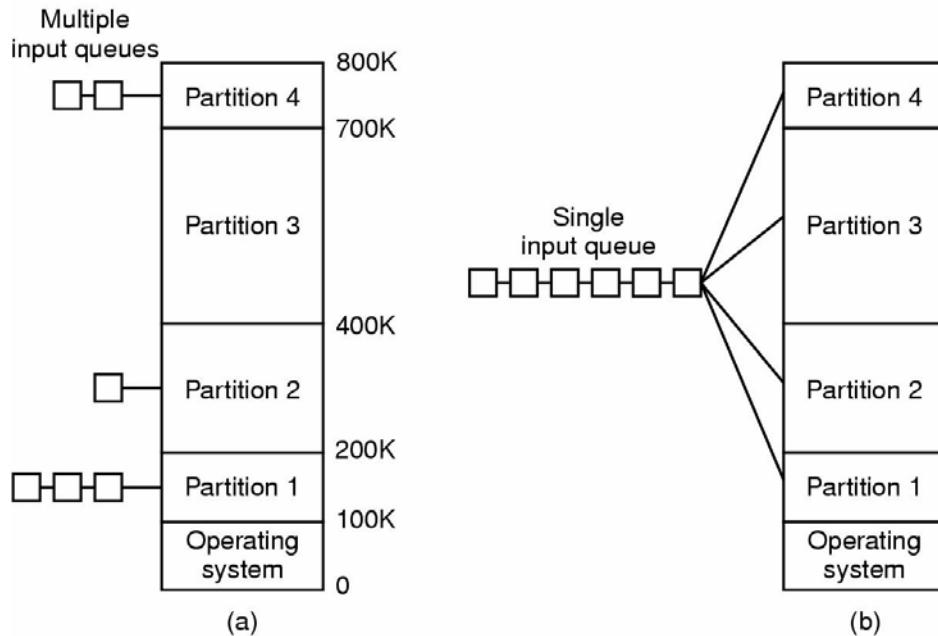


Figura 4-2. (a) Particiones de memoria fijas con colas de entrada separadas para cada partición. (b) Particiones de memoria fijas con una única cola de entrada.

La desventaja de ordenar los trabajos que llegan en colas separadas se hace evidente cuando la cola de una partición grande está vacía pero la cola de una partición pequeña está llena, como sucede con las particiones 1 y 3 de la Figura 4-2(a). Aquí los trabajos pequeños tienen que esperar para entrar en la memoria, a pesar de que hay más que suficiente memoria libre. Una organización alternativa sería mantener una única cola, como en la Figura 4-2(b). Cada vez que se desocupe una partición, se cargará en la partición vacía y se ejecutará en ella el trabajo más cercano al frente de la cola que quepa en esa partición. Puesto que no es deseable desperdiciar una partición grande con un trabajo pequeño, una estrategia diferente sería examinar toda la cola de entrada cada vez que quede libre una partición, y escoger el trabajo más grande que quepa en ella. Adviértase que este último algoritmo discrimina a los trabajos pequeños porque no los considera merecedores de toda una partición, cuando usualmente es deseable dar a los trabajos más pequeños (que suelen ser interactivos) el mejor servicio, no el peor.

Una solución es tener al menos una partición pequeña. Tal partición permitiría que se ejecutases los trabajos pequeños sin tener que asignarles una partición grande.

Otra estrategia sería tener una regla que estableciese que un trabajo elegible para ejecutarse no puede pasarse por alto más de k veces. Cada vez que se le pasa por alto, se le otorga un punto. Cuando haya adquirido k puntos, ya no se le podrá ignorar.

Este sistema, con particiones fijas establecidas por la mañana por el operador sin posibilidad de modificarse después, fue utilizado en el OS/360 de los grandes mainframes de IBM durante muchos años. Se le denominó **MFT** (*Multiprogramming with a Fixed number of Tasks* u OS/MFT). Es sencillo de entender e igualmente sencillo de implementar: los trabajos que llegan se encolan hasta que esté disponible una partición apropiada, momento en el cual el trabajo se carga en esa partición y se ejecuta hasta que termina. Actualmente, son pocos, por no decir ninguno, los sistemas operativos que utilizan este modelo.

4.1.3 Modelización de la Multiprogramación

Cuando se utiliza multiprogramación, es posible mejorar la utilización de la CPU. Dicho groseramente, si un proceso típico realiza cálculos sólo el 20% del tiempo que está en la memoria, y se tienen cinco procesos en la memoria a la vez, la CPU deberá estar ocupada todo el tiempo. Sin embargo, este modelo es excesivamente optimista ya que supone que los cinco procesos nunca van a estar esperando por E/S todos al mismo tiempo.

Se obtiene un modelo mejor observando cómo se comporta la utilización de la CPU desde un punto de vista probabilístico. Supongamos que un proceso pasa una fracción p de su tiempo esperando a que terminen sus operaciones de E/S. Si hay n procesos en la memoria a la vez, la probabilidad de que todos los n procesos estén esperando por E/S (en cuyo caso la CPU estará inactiva) es p^n . La utilización de la CPU viene entonces dada por la fórmula

$$\text{utilización de la CPU} = 1 - p^n$$

La Figura 4-3 muestra la utilización de la CPU en función de n , que se denomina el **grado de multiprogramación**.

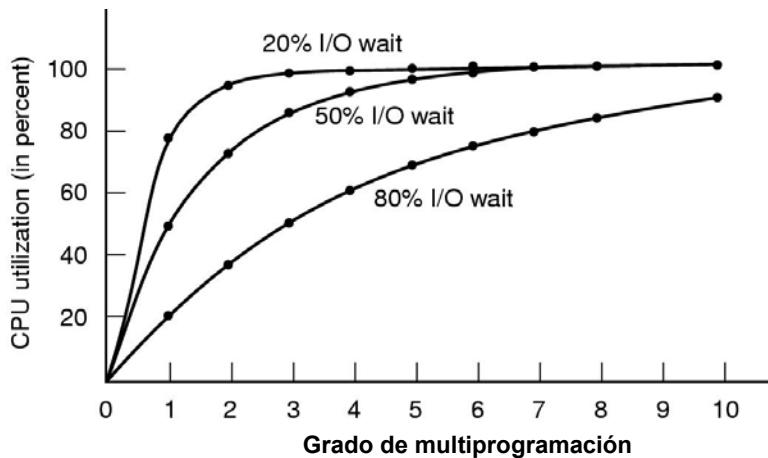


Figura 4-3. Utilización de la CPU en función del número de procesos en la memoria.

En la figura queda claro que si los procesos pasan el 80% de su tiempo esperando por E/S, deberá haber al menos 10 procesos en la memoria a la vez para conseguir que el desaprovechamiento de la CPU esté por debajo del 10%. Si pensamos en que un proceso interactivo que espera a que un usuario teclee algo en un terminal está en estado de espera por una E/S, debe quedarnos claro que los tiempos de espera por E/S del 80% o más no son algo inusual. Pero incluso en los sistemas en batch, los procesos que realizan mucha E/S de disco a menudo alcanzan o superan ese porcentaje.

En aras a conseguir una absoluta precisión, debe señalarse que el modelo probabilístico que acabamos de describir es sólo una aproximación, ya que supone implícitamente de que los n procesos son independientes, lo que significa que es perfectamente aceptable para un sistema con cinco procesos en memoria que tenga tres en ejecución y dos esperando. Pero con una única CPU no es posible tener tres procesos ejecutándose a la vez, así un proceso que pase al estado preparado mientras la CPU está ocupada tendrá necesariamente que esperar. Así los procesos no son independientes. Puede construirse un modelo más preciso utilizando la teoría de colas, pero la afirmación – de que la multiprogramación permite a los procesos usar la CPU cuando en otro caso estaría ociosa – sigue siendo válida, a pesar de que las nuevas curvas de la Figura 4-3 obtenidas con ese modelo resulten un poco diferentes.

Aunque el modelo de la Figura 4-3 es simplista, sin embargo puede servir para hacer predicciones específicas, aunque aproximadas, sobre el rendimiento de la CPU. Por ejemplo, supongamos que un ordenador tiene 32 MB de memoria, de la cual el sistema operativo ocupa 16 MB y cada programa de usuario ocupa 4 MB. Estos tamaños permiten que estén en memoria a la vez cuatro programas de usuario. Con una espera por E/S media del 80%, tendremos una utilización de la CPU (ignorando la sobrecarga del sistema operativo) de $1 - 0,8^4$, o sea, aproximadamente el 60%. La adición de otros 16 MB de memoria permitiría al sistema pasar de multiprogramación de cuatro vías a multiprogramación de ocho vías, con lo que el aprovechamiento de la CPU subiría al 83%. En otras palabras, los 16 MB adicionales elevan el rendimiento un 38%.

Añadiendo otros 16 MB sólo se podría aumentar la utilización de la CPU del 83% al 93%, elevándose el rendimiento tan sólo un 12%. Utilizando este modelo, el propietario del ordenador podría decidir que la primera adición representa una buena inversión, pero que la segunda no se justifica suficientemente.

4.1.4 Análisis del Rendimiento de un Sistema Multiprogramado

El modelo anterior también puede servir para analizar sistemas en batch. Por ejemplo, consideremos un centro de cálculo cuyos trabajos en media esperan por E/S el 80% del tiempo. Cierta mañana se presentan cuatro trabajos, como se muestra en la Figura 4-4(a). El primer trabajo llega a las 10:00 y requiere cuatro minutos de tiempo de CPU. Con una espera por E/S del 80%, el trabajo utiliza sólo 12 segundos de tiempo de CPU por cada minuto que está en la memoria, aunque ningún otro trabajo esté compitiendo con él por la CPU. Los otros 48 segundos se gastan esperando a que termine la E/S. Por lo tanto, el trabajo tendrá que permanecer en la memoria por lo menos 20 minutos para obtener los 4 minutos de trabajo de CPU, aunque no haya competencia por el uso de la CPU.

De las 10:00 a las 10:10 el trabajo 1 está solo en la memoria y realiza 2 minutos de trabajo. Cuando llega el trabajo 2 a las 10:10, la utilización de la CPU aumenta de 0,20 a 0,36, gracias al mayor grado de multiprogramación (ver la Figura 4-3). Sin embargo con la planificación round-robin, cada trabajo recibe la mitad del tiempo de CPU, así que cada uno efectúa 0,18 minutos de trabajo de CPU por cada minuto que esté en la memoria. Obsérvese que la adición de un segundo trabajo le cuesta al primer trabajo sólo el 10% de su rendimiento: pasa de obtener 0,20 minutos de CPU por minuto de tiempo real, a 0,18 minutos de CPU por minuto de tiempo real.

El tercer trabajo llega a las 10:15. Hasta aquí el trabajo 1 ha recibido 2,9 minutos de CPU y el trabajo 2 ha tenido 0,9 minutos. Con multiprogramación de tres vías, cada trabajo recibe 0,16 minutos de tiempo de CPU por minuto de tiempo real, como se ilustra en la Figura 4-4(b). Entre las 10:15 y las 10:20 cada uno de los tres trabajos recibe 0,8 minutos de tiempo de CPU. A las 10:20 llega el cuarto trabajo. La Figura 4-4(c) muestra la secuencia completa de sucesos.

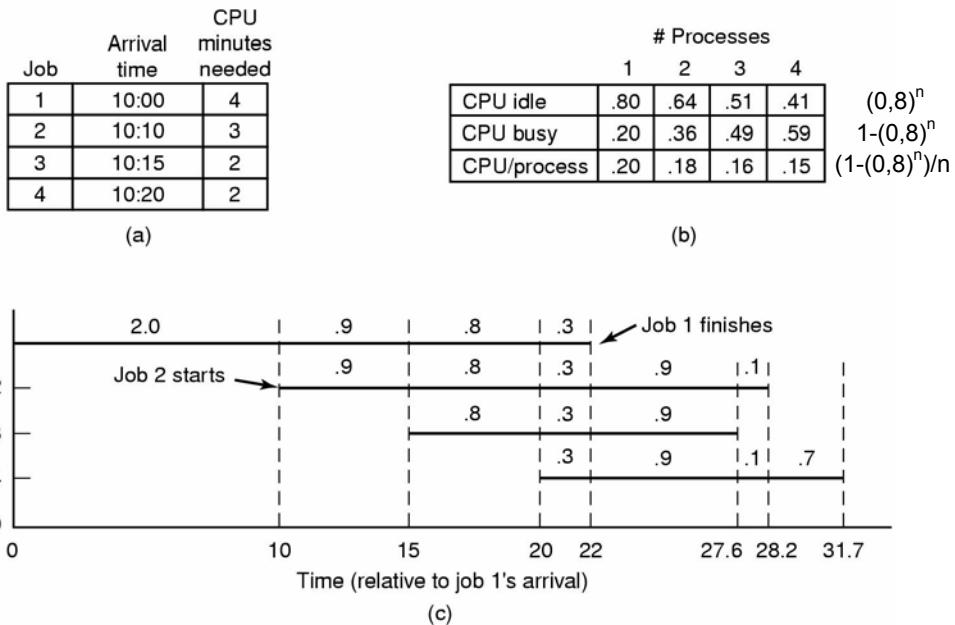


Figura 4-4. (a) Llegada y requerimientos de CPU de cuatro trabajos. (b) Utilización de la CPU para 1 a 4 procesos con el 80% de espera por E/S. (c) Secuencia de sucesos a medida que los trabajos llegan y terminan. Los números por encima de las líneas horizontales indican cuanto tiempo de CPU, en minutos, recibe cada trabajo en cada intervalo.

4.1.5 Reubicación y Protección

La multiprogramación introduce dos problemas fundamentales que deben resolverse: reubicación y protección. Examinemos la Figura 4-2. De la figura es claro que diferentes trabajos deben ejecutarse en direcciones diferentes. Cuando se enlaza un programa (es decir, cuando se combinan el programa principal, los procedimientos escritos por el usuario y los procedimientos de biblioteca en un único espacio de direcciones), el enlazador (*linker*) necesita saber en qué dirección de la memoria comenzará el programa.

Por ejemplo, supongamos que la primera instrucción es una llamada a un procedimiento que está en la dirección absoluta 100 dentro del archivo binario producido por el enlazador. Si este programa se carga en la partición 1 (en la dirección 100K), esa instrucción saltará a la dirección absoluta 100, que está dentro del sistema operativo. Lo que se necesita es una llamada a 100K + 100. Si el programa se carga en la partición 2, la llamada deberá llevarse a cabo como una llamada a 200K + 100, y así sucesivamente. Este problema se conoce como el problema de la **reubicación**.

Una posible solución consiste en modificar las instrucciones a medida que el programa se carga en la memoria. Los programas cargados en la partición 1 tendrán cada dirección incrementada en 100K, los programas cargados en la partición 2 tendrán sus direcciones incrementadas en 200K, y así sucesivamente. Para realizar la reubicación durante la carga de esa manera, el enlazador deberá incluir en el programa binario una lista o mapa de bits que indique qué palabras del programa son direcciones a reubicar y cuáles son códigos de operación, constantes u otras cosas que no deben reubicarse. OS/MFT trabajaba de esta manera.

**reubicación
estática**

La reubicación durante la carga no resuelve el problema de la protección. Un programa malicioso siempre puede construir una nueva instrucción y saltar a ella. Puesto que los programas en este sistema utilizan direcciones de memoria absolutas en vez de direcciones relativas a un registro, no hay ninguna manera de impedir que un programa construya una instrucción que lea o escriba en cualquier palabra de la memoria. En los sistemas multiusuario es altamente indeseable dejar que los procesos lean y escriban en la memoria perteneciente a otros usuarios.

La solución que IBM escogió para proteger el 360 fue dividir la memoria en bloques de 2 KB y asignar un código de protección de 4 bits a cada bloque. El registro de estado, denominado en el 360 PSW (*Program Status Word*), contenía una clave de 4 bits. El hardware del 360 provocaba una excepción tras cualquier intento por parte del proceso en ejecución de acceder a un bloque de memoria cuyo código de protección difiriera de la clave contenida en la PSW. Puesto que sólo el sistema operativo puede modificar los códigos de protección y la clave, se impedía así que los procesos de usuario interfíresen unos con otros y con el sistema operativo mismo.

Otra solución alternativa para tanto el problema de la reubicación como el de la protección consiste en equipar la máquina con dos registros especiales de hardware, llamados el **registro de base** y el **registro de límite**. Cuando se planifica un proceso, se carga el registro de base con la dirección donde comienza su partición, y el registro de límite se carga con la longitud de la partición. Cada vez que se genera una dirección de memoria, se le suma de forma automática el contenido del registro de base antes de enviarla a la memoria. Por ejemplo, si el registro base contiene el valor 100K, una instrucción CALL 100 se convierte efectivamente en una instrucción CALL 100K + 100, sin que la instrucción en sí se modifique. También se comparan las direcciones con el registro de límite para asegurar que no intentan direccionar memoria fuera de la partición actual. El hardware protege los registros de base y de límite para evitar que los programas de usuario los modifiquen.

Una desventaja de este esquema es la necesidad de efectuar una suma y una comparación cada vez que se hace referencia a la memoria. Las comparaciones pueden hacerse rápidamente, pero las sumas son lentas debido al tiempo de propagación del acarreo a menos que se utilicen circuitos especiales de suma.

El CDC 6600 – el primer supercomputador del mundo – utilizaba este esquema. La CPU Intel 8088 incorporada en el PC original de IBM, utilizaba una versión más débil de este esquema: registros de base, pero sin registros de límite. En la actualidad son pocos los ordenadores (si es que hay alguno) que utilicen este esquema.

4.2 INTERCAMBIO (SWAPPING)

En un sistema en batch es simple y efectivo organizar la memoria en particiones fijas. Cada trabajo se carga en una partición cuando llega a la cabeza de la cola, y se queda en la memoria hasta que termina. Mientras puedan mantenerse suficientes trabajos en la memoria como para que la CPU esté ocupada todo el tiempo, no hay ninguna razón para utilizar un esquema más complicado.

Con los sistemas de tiempo compartido o con los ordenadores personales orientados a gráficos, la situación es distinta. A veces no hay suficiente memoria principal para contener a todos los procesos actualmente activos, así que los procesos de más deben mantenerse en el disco y cargarse en la memoria para ejecutarse de forma dinámica.

Pueden utilizarse dos enfoques generales para la gestión de la memoria, dependiendo (en parte) del hardware disponible. La estrategia más sencilla, llamada **intercambio (swapping)**, consiste en cargar en la memoria un proceso entero, ejecutarlo durante un rato y volver a guardarlo en el disco. La otra estrategia, llamada **memoria virtual**, permite que los programas se ejecuten incluso cuando tan sólo una parte de ellos esté cargada en la memoria principal. A continuación estudiaremos el intercambio; en la sección 4.3 examinaremos la memoria virtual.

En la Figura 4.5 se ilustra el funcionamiento de un sistema con intercambio. Inicialmente sólo está en la memoria el proceso *A*. Luego se crean o se traen del disco los procesos *B* y *C*. En la Figura 4-5(d) *A* se intercambia al disco. Luego llega *D* y *B* sale. Finalmente *A* entra de nuevo. Ya que *A* está ahora en un lugar distinto, es preciso reubicar las direcciones que contiene, sea por software en el momento del intercambio, o (más probablemente) por hardware durante la ejecución del programa.

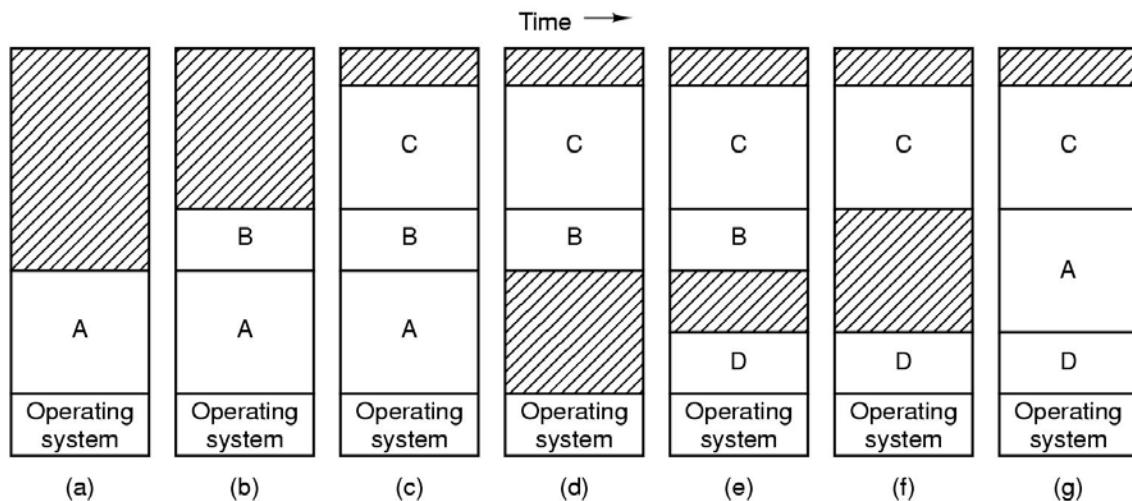


Figura 4-5. La asignación de memoria cambia a medida que los procesos entran en la memoria y salen de ella. Las regiones sombreadas representan memoria no utilizada.

La diferencia principal entre las particiones fijas de la Figura 4-2 y las particiones variables de la Figura 4-5 es que en el segundo caso el número, la ubicación y el tamaño de las particiones varía de forma dinámica a medida que los procesos llegan y se van, mientras que en el primero no cambian. La flexibilidad de no estar atados a un número fijo de particiones que podrían ser demasiado grandes o demasiado pequeñas mejora la utilización de la memoria, pero también complica la asignación y liberación de la memoria, así como su control.

Cuando el intercambio crea múltiples huecos en la memoria, es posible combinar todos esos huecos en uno solo más grande, moviendo todos los procesos hacia abajo hasta donde sea posible. Esta técnica se denomina **compactación de la memoria**. Usualmente no se realiza porque requiere mucho tiempo de CPU. Por ejemplo, en una máquina de 256 MB que puede copiar 4 bytes en 40 nanosegundos, se requerirían aproximadamente 2,7 segundos para compactar toda la memoria.

Una cuestión que es necesario plantearse es cuánta memoria debe asignarse a un proceso cuando se crea o se intercambia a la memoria. Si los procesos se crean con un tamaño fijo que nunca cambia, la asignación es sencilla: el sistema operativo asigna exactamente lo que se necesita, ni más ni menos.

Sin embargo, si los segmentos de datos de los procesos pueden crecer, por ejemplo, asignando memoria dinámicamente de un *heap*, como en muchos lenguajes de programación, se presentará un problema cada vez que un proceso trate de crecer. Si hay un hueco adyacente al proceso, podrá asignársele y el proceso podrá crecer en dicho hueco. Por otra parte, si el proceso está adyacente a otro proceso, el proceso que crece tendrá que moverse a un hueco de la memoria lo bastante grande como para contenerlo, o bien habrá que intercambiar a disco uno o más procesos para crear un hueco del tamaño suficiente. Si un proceso no puede crecer en la memoria y el área de intercambio en el disco está llena, el proceso tendrá que esperar o ser eliminado.

Si es probable que la mayoría de los procesos crezcan durante su ejecución, resulta una buena idea asignar un poco de memoria extra cada vez que se intercambie un proceso a la memoria o se cambie de lugar, a fin de reducir la sobrecarga asociada con el cambio de lugar o el intercambio de procesos que ya no caben en la memoria que se les asignó. Sin embargo, cuando se intercambien esos procesos al disco sólo deberá transferirse la memoria que realmente se esté usando; sería un derroche intercambiar también la memoria extra. En la Figura 4-6(a) vemos una configuración de memoria en la que se ha asignado espacio para crecer a dos procesos.

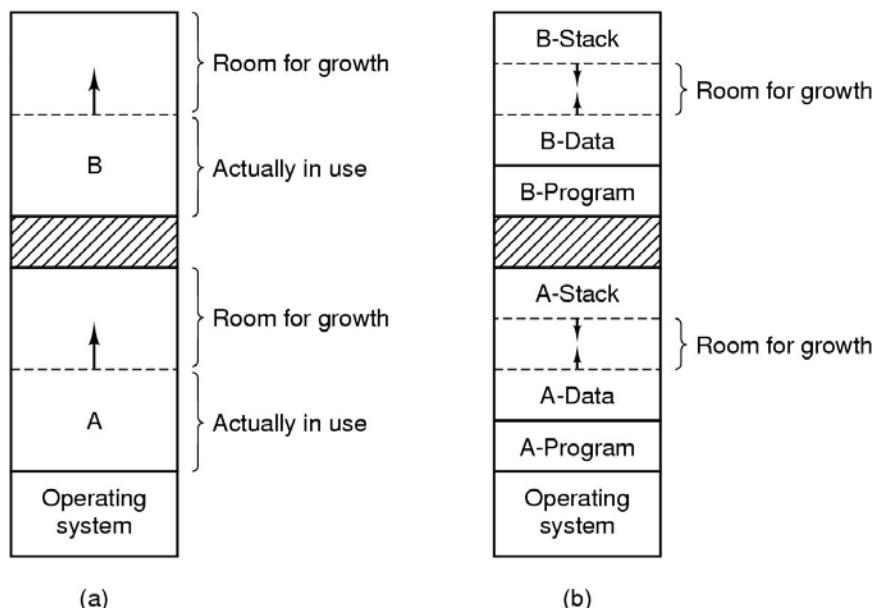


Figura 4-6. (a) Asignación de espacio para un segmento de datos que puede crecer. (b) Asignación de espacio para una pila y un segmento de datos que pueden crecer.

Si los procesos tienen dos segmentos que pueden crecer, por ejemplo, el segmento de datos que se usa como un *heap* para variables que se asignan y liberan de forma dinámica, y un segmento de pila para las variables locales normales y las direcciones de retorno, esto sugiere la organización alternativa de la Figura 4-6(b). En esa figura vemos que los procesos *A* y *B* tienen en lo alto de la memoria que se les asignó una pila que crece hacia abajo, e inmediatamente encima del programa tienen un segmento de datos que crece hacia arriba. La memoria entre ambos segmentos puede utilizarse para cualquiera de los dos segmentos que crezca. Si se agota, el proceso tendrá que moverse a un hueco con suficiente espacio, intercambiarse de la memoria al disco hasta que pueda crearse un hueco del tamaño suficiente, o eliminarse.

4.2.1 Gestión de Memoria con Mapas de Bits

Si la memoria se asigna dinámicamente, el sistema operativo debe gestionarla. En términos generales, hay dos formas de llevar el control del uso de la memoria: mapas de bits y listas de bloques libres. En esta sección y en la que sigue examinaremos los dos métodos.

Con un mapa de bits, la memoria se divide en **unidades de asignación**, que pueden ser desde unas cuantas palabras hasta varios kilobytes. A cada unidad de asignación le corresponde un bit del mapa de bits. El bit es 0 si la unidad de asignación está libre y 1 si está ocupada (o viceversa). La Figura 4-7 muestra parte de la memoria y el mapa de bits correspondiente.

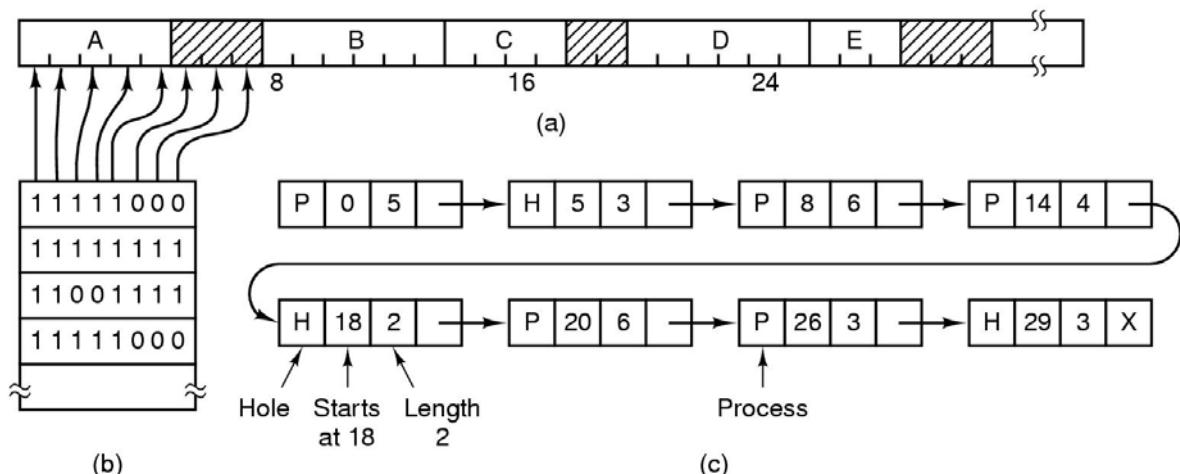


Figura 4-7. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas pequeñas corresponden a las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están desocupadas. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una cuestión de diseño importante. Cuanto más pequeña sea la unidad, mayor será el mapa de bits. Sin embargo, incluso con unidades de asignación de sólo 4 bytes, 32 bits de la memoria sólo requieren un bit en el mapa. Una memoria de $32n$ bits necesitará un mapa de n bits, así que el mapa de bits sólo ocupará $1/33$ de la memoria. Si se escoge una unidad de asignación grande, el mapa de bits será pequeño, pero podría desperdiciarse una cantidad de memoria apreciable en la última unidad de asignación del proceso si el tamaño del proceso no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits proporciona una manera sencilla de llevar el control de las palabras de memoria utilizando una cantidad de memoria fija porque su tamaño sólo depende del tamaño de la memoria y del tamaño de la unidad de asignación. El problema principal con él es que una vez que se ha decidido traer a la memoria un proceso de k unidades, el gestor de memoria debe examinar el mapa de bits en busca de una secuencia de k bits a 0 consecutivos. Buscar en el mapa una secuencia de cierta longitud es una operación lenta (porque la secuencia en el mapa puede cruzar fronteras de palabra); este es un argumento en contra del uso de los mapas de bits.

4.2.2 Gestión de Memoria con Listas Enlazadas

Otra forma de llevar el control de la memoria es mantener una lista enlazada de bloques de memoria asignados y libres, donde cada bloque es un proceso o un hueco entre dos procesos. La memoria de la Figura 4-7(a) se representa en la Figura 4-7(c) como una lista enlazada. Cada nodo de la lista representa un bloque, especificando el tipo de bloque [hueco (H) o proceso (P)], su dirección de comienzo, su longitud y un puntero al siguiente nodo.

En este ejemplo, la lista de bloques se mantiene ordenada por dirección. Esta ordenación por dirección tiene la ventaja de que cuando un proceso termina o se intercambia a disco, resulta trivial la actualización de la lista. Normalmente, un proceso que termina tiene dos vecinos (excepto si está en el extremo superior o inferior de la memoria). Dichos vecinos pueden ser procesos o huecos, lo que da lugar a las cuatro combinaciones de la Figura 4-8. En la Figura 4-8(a) la actualización de la lista requiere sustituir una P por una H. En las Figuras 4-8(b) y 4-8(c), se funden dos entradas en una sola, acortándose la lista en una entrada. En la Figura 4-8(d) se fusionan tres entradas y se eliminan dos elementos de la lista. Puesto que la entrada en la tabla de procesos correspondiente al proceso que terminó apunta normalmente a la entrada de la lista del proceso mismo, puede ser más conveniente implementar la lista como una lista doblemente enlazada, en vez de cómo una lista simplemente enlazada como la de la Figura 4-7(c). Esa estructura hace más fácil encontrar la entrada anterior para determinar si puede haber una fusión con un hueco adyacente anterior.

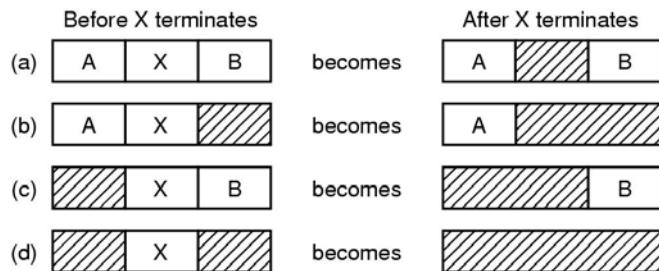


Figura 4-8. Cuatro combinaciones de vecinos para el proceso *X* que termina.

Si los procesos y huecos se mantienen en una lista ordenada por dirección, pueden utilizarse varios algoritmos para asignar memoria a un proceso recién creado (o a un proceso existente que se intercambia a memoria). Suponemos que el gestor de memoria sabe cuanta memoria debe asignar al proceso. El algoritmo más sencillo es el del **primer ajuste** (*first fit*). El gestor de memoria explora la lista de segmentos hasta encontrar un hueco lo suficientemente grande. Luego el hueco se divide en dos partes, una para el proceso y otra para la memoria no utilizada, salvo en el caso poco probable de que el ajuste sea exacto. Este algoritmo es rápido porque la búsqueda es lo más corta posible.

Una variación menor del primer ajuste es el **siguiente ajuste** (*next fit*). Su funcionamiento es similar al del primer ajuste, salvo que el algoritmo recuerda en qué punto de la lista se quedó la última vez que encontró un hueco apropiado. La siguiente vez que se le pida encontrar un hueco, comenzará la búsqueda desde ese punto de la lista, y no siempre desde su principio, como lo hace el algoritmo del primer ajuste. Las simulaciones efectuadas por Bays (1977) muestran que el algoritmo del siguiente ajuste proporciona un rendimiento ligeramente peor que el algoritmo del primer ajuste.

Otro algoritmo muy conocido es el del **mejor ajuste** (*best fit*). En este caso se recorre toda la lista para encontrar el hueco más pequeño capaz de contener al proceso. En lugar de dividir un hueco grande que podría necesitarse después, el algoritmo del mejor ajuste trata de encontrar un hueco de tamaño lo más parecido al tamaño realmente solicitado.

Como ejemplo del primer ajuste y del mejor ajuste, consideremos otra vez la Figura 4-7. Si se necesita un bloque de tamaño 2, el algoritmo del primer ajuste asignará el bloque que comienza en la unidad de asignación 5, pero el algoritmo del mejor ajuste asignará el que comienza en la unidad de asignación 18.

El algoritmo del mejor ajuste es más lento que el del primer ajuste porque debe recorrer toda la lista cada vez que se le invoca. De forma un tanto sorprendente, resulta que también desperdicia más memoria que el primer ajuste o el siguiente ajuste porque tiende a saturar la memoria de huecos diminutos y por tanto inútiles. En media, el primer ajuste genera huecos más grandes.

Para resolver el problema de dividir un hueco de tamaño casi exactamente igual al requerido, en un proceso y un hueco diminuto, podríamos considerar el **peor ajuste** (*worst fit*), es decir, escoger siempre el hueco más grande disponible, de modo que el hueco resultante sea lo suficientemente grande como para ser útil. Las simulaciones han demostrado que el peor ajuste tampoco es una idea muy buena.

Los cuatro algoritmos anteriores pueden acelerarse manteniendo listas separadas para los procesos y los huecos. De esta manera, todos ellos dedicarán toda su energía a inspeccionar huecos, no procesos. El precio inevitable que se paga por esta aceleración de la asignación es una complejidad adicional y ralentización cuando se libera la memoria, ya que los segmentos liberados deben eliminarse de la lista de procesos e insertarse en la lista de huecos.

Si se mantienen listas distintas para procesos y huecos, la lista de huecos podría mantenerse ordenada por tamaño, para que el algoritmo del mejor ajuste sea más rápido. Cuando este algoritmo recorra la lista de huecos desde el más pequeño al más grande, tan pronto como encuentre un hueco suficientemente grande, sabrá que es el más pequeño que puede asignarse y, por lo tanto, es el mejor ajuste. No es preciso buscar más, como sucedía en el esquema de una única lista. Con la lista de huecos ordenada por tamaño, los algoritmos del primer ajuste y del mejor ajuste son igual de rápidos, y el algoritmo del siguiente ajuste no tiene sentido.

Cuando los huecos están en una lista separada de los procesos, es posible una pequeña optimización. En lugar de tener una estructura de datos aparte para mantener la lista de huecos, como se hace en la Figura 4-7(c), pueden utilizarse los mismos huecos. La primera palabra de cada hueco podría indicar su tamaño, y la segunda, un puntero al siguiente hueco. Los nodos de la lista de la Figura 4-7(c), que requieren tres palabras y un bit (P/H), ya no serían necesarios.

Otro algoritmo de asignación es el del **ajuste rápido** (*quick fit*), que mantiene listas separadas para algunos de los tamaños solicitados más frecuentemente. Por ejemplo, podría mantenerse una tabla con n entradas, en la cual la primera entrada fuese un puntero a la cabeza de una lista de huecos de 4 KB, la segunda entrada es un puntero a una lista de huecos de 8 KB, la tercera entrada es un puntero a una lista de huecos de 12 KB, y así de forma sucesiva. Los huecos de por ejemplo 21 KB podrían colocarse en la lista de 20 KB, o bien, en una lista especial de huecos de tamaño raro. Con este algoritmo, la localización de un hueco del tamaño requerido es extremadamente rápida, pero tiene la misma desventaja de todos los esquemas que ordenan por tamaño, es decir, cuando un proceso termina o se intercambia al disco resulta costoso localizar a sus vecinos para ver si puede fusionarse con ellos. Si no se realiza esa fusión, la memoria podría fragmentarse rápidamente en un gran número de pequeños huecos en los que no cupiese ningún proceso.

4.3 MEMORIA VIRTUAL

Hace ya muchos años que aparecieron los primeros programas demasiado grandes para caber en la memoria disponible. La solución usualmente adoptada fue dividir el programa en trozos, llamados **recubrimientos** (*overlays*). El recubrimiento 0 era el que se ejecutaba primero. Cuando terminaba, llamaba a otro recubrimiento. Algunos sistemas de recubrimientos eran altamente complejos, permitiendo tener varios recubrimientos en memoria a la vez. Los recubrimientos se mantenían en el disco y el sistema operativo los intercambiaba entre el disco y la memoria, dinámicamente según se iban necesitando.

Aunque el sistema realizaba el trabajo real de intercambiar los recubrimientos, el programador tenía que encargarse de dividir en trozos apropiados el programa. La tarea de dividir programas grandes en pequeños trozos modulares era laboriosa y tediosa, así que no pasó mucho tiempo antes de que alguien idease una manera de dejar todo ese trabajo para el ordenador.

El método ideado (Fotheringham, 1961) se conoce ahora como **memoria virtual**. La idea básica detrás de la memoria virtual es que el tamaño combinado del programa, sus datos y su pila pueden exceder la cantidad de memoria física disponible. El sistema operativo mantiene en la memoria principal aquellas partes del programa que se están usando en cada momento, manteniendo el resto de las partes del programa en el disco. Por ejemplo, un programa de 16 MB puede ejecutarse sobre una máquina de 4 MB eligiendo cuidadosamente qué 4 MB se tendrán en la memoria en cada instante, e intercambiando partes del programa entre el disco y la memoria, según sea necesario.

La memoria virtual puede funcionar también en un sistema multiprogramado, con diversos fragmentos de muchos programas en memoria a la vez. Mientras un programa espera a que se traiga del disco una parte de si mismo, está esperando por una E/S y no puede ejecutarse, por lo que debe asignarse la CPU a otro proceso de la misma forma que en cualquier otro sistema multiprogramado.

4.3.1 Paginación

La mayoría de los sistemas con memoria virtual utilizan una técnica denominada **paginación**, que vamos a describir ahora. En cualquier ordenador, existe un conjunto de direcciones de memoria que los programas pueden producir. Cuando un programa utiliza una instrucción como

```
MOV REG,1000
```

lo hace para copiar el contenido de la dirección de memoria 1000 en REG (o viceversa, dependiendo del ordenador). Las direcciones pueden generarse empleando indexación, registros base, registros de segmento y otros métodos.

Estas direcciones generadas por el programa se denominan **direcciones virtuales** y constituyen el **espacio de direcciones virtual**. En ordenadores sin memoria virtual, la dirección virtual se coloca directamente sobre el bus de memoria y eso hace que la palabra de memoria física con esa dirección se lea o escriba. Cuando se utiliza memoria virtual, las direcciones virtuales no se envían directamente al bus de memoria, sino que van a una **unidad de gestión de memoria (MMU; Memory Management Unit)** que establece una correspondencia entre las direcciones virtuales y las direcciones físicas de la memoria, como se ilustra en la Figura 4-9.

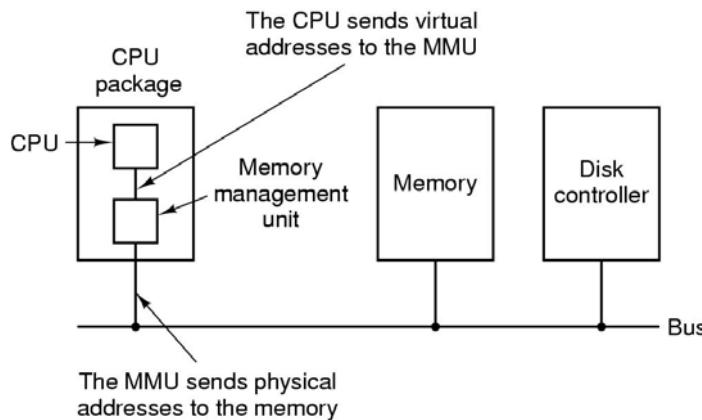


Figura 4-9. Posición y función de la MMU. Aquí se muestra la MMU formando parte del chip de la CPU porque es lo más común en la actualidad. Sin embargo, lógicamente podría ser un chip aparte y en el pasado lo era.

En la Figura 4-10 se muestra un ejemplo sencillo de cómo funciona esta correspondencia. En este ejemplo tenemos un ordenador que puede generar direcciones de 16 bits, desde 0 hasta 64K. Éstas son las direcciones virtuales. Sin embargo, este ordenador sólo tiene 32 KB de memoria física, por lo que, aunque es posible escribir programas de 64 KB, no es posible cargarlos en la memoria completamente y ejecutarlos. En el disco debe estar presente una copia completa de la imagen del programa, de hasta 64 KB, para poder cargar a la memoria partes del programa según se vayan necesitando.

El espacio de direcciones virtual se divide en unidades llamadas **páginas**. Las unidades correspondientes en la memoria física se denominan **marcos de página**. Las páginas y los marcos de página tienen siempre el mismo tamaño, que en este ejemplo es de 4 KB, aunque en sistemas reales se han usado páginas desde 512 bytes hasta 64 KB. Con un espacio de direcciones virtual de 64 KB, y con una memoria física 32 KB, tenemos 16 páginas virtuales y 8 marcos de página. Las transferencias entre la RAM y el disco siempre se efectúan en unidades de una página.

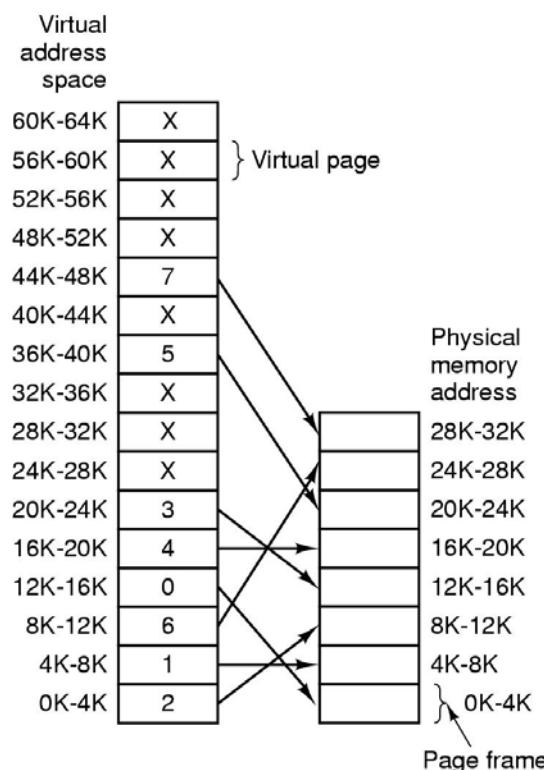


Figura 4-10. La relación entre las direcciones virtuales y las direcciones de la memoria física viene dada por la tabla de páginas.

Cuando el programa intenta acceder a la dirección 0, por ejemplo, con la instrucción

MOV REG,0

la dirección virtual 0 se envía a la MMU. La MMU ve que esa dirección virtual cae en la página 0 (0 a 4095), que de acuerdo a su correspondencia está en el marco de página 2 (8192 a 12287). La MMU transforma entonces la dirección a 8192 y coloca la dirección 8192 en el bus. La memoria no tiene ningún conocimiento de la MMU y lo único que ve es una petición de lectura o escritura de la dirección 8192, que lleva a cabo. Por lo tanto, la MMU transforma efectivamente todas las direcciones virtuales entre 0 y 4095 en las direcciones físicas entre 8192 y 12287.

De forma similar, una instrucción

MOV REG,8192

se transforma efectivamente en

MOV REG,24576

debido a que la dirección virtual 8192 está en la página virtual 2, la cual corresponde al marco de página físico 6 (direcciones físicas 24576 a 28671). Como un tercer ejemplo, la dirección virtual 20500 está 20 bytes después del comienzo de la página virtual 5 (direcciones virtuales 20480 a 24575) por lo que se corresponde con la dirección física $12288 + 20 = 12308$.

Esta capacidad para mapear las 16 páginas virtuales en cualquiera de los ocho marcos de página ajustando debidamente el mapa de la MMU no resuelve por sí misma el problema de que el espacio de direcciones virtual es más grande que la memoria física. Puesto que sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales de la Figura 4-10 tendrán correspondencia con la memoria física. Las demás, que se indican con una cruz en la figura, no tienen correspondencia. En el hardware real, un **bit de página presente/ausente** (o simplemente **bit de presencia**) lleva el control de qué páginas están físicamente presentes en la memoria.

¿Qué sucede si el programa intenta utilizar una página que no tiene correspondencia, por ejemplo, ejecutando la instrucción

MOV REG,32780

que referencia el byte 12 dentro de la página virtual 8 (que comienza en 32768)? La MMU ve que la página no tiene correspondencia (lo que se indica con una cruz en la figura) y provoca una excepción que hace que la CPU ceda el control al sistema operativo. Esta excepción se denomina una **falta de página**. El sistema operativo escoge un marco de página poco utilizado y escribe su contenido de vuelta al disco. A continuación el sistema operativo carga la página a la que se acaba de hacer referencia colocándola en el marco de página que acaba de quedar desocupado, modifica el mapa en la MMU y reinicia la instrucción interrumpida.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 a partir de la dirección física 4K y haría dos cambios en el mapa de la MMU. Primero, marcaría la entrada de la página virtual 1 como sin correspondencia para capturar los accesos futuros a las direcciones virtuales entre 4K y 8K. Luego sustituiría la cruz de la entrada correspondiente a la página virtual 8 por un 1, de modo que cuando la instrucción interrumpida vuelve a ejecutarse, transforme la dirección virtual 32780 en la dirección física 4108.

Vamos a mirar ahora dentro de la MMU para ver cómo funciona y justificar por qué motivo hemos elegido un tamaño de página que es potencia de 2. En la Figura 4-11 vemos un ejemplo de dirección virtual, 8196 (001000000000100 en binario), que se transforma utilizando el mapa de la MMU de la Figura 4-10. La dirección virtual entrante de 16 bits se divide en un número de página de 4 bits y un desplazamiento de 12 bits. Con 4 bits para el número de página, podemos tener 16 páginas, y con 12 bits para el desplazamiento, podemos direccionar todos los 4096 bytes que hay dentro de una página.

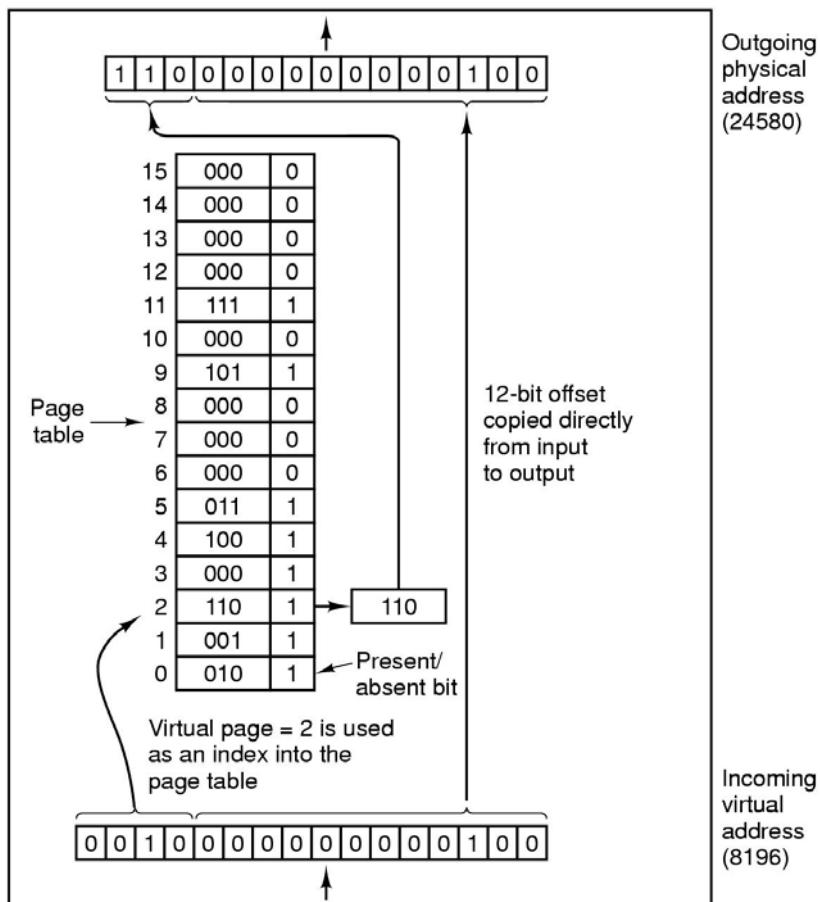


Figura 4-11. Funcionamiento interno de la MMU con 16 páginas de 4 KB.

El número de página se utiliza como un índice para consultar la **tabla de páginas**, obteniendo el número del marco de página correspondiente a esa página virtual. Si el bit de presencia es 0, se generará una excepción que cederá el control al sistema operativo. Si el bit es 1, el número de marco de página encontrado en la tabla de páginas se copia en los tres bits de mayor orden del registro de salida junto con el desplazamiento de 12 bits que se copia (sin ninguna modificación) de la dirección virtual recibida. Juntos, esos dos campos forman una dirección física de 15 bits. Finalmente, el registro de salida se vuelca al bus de memoria como la dirección de memoria física a la que efectivamente se va a acceder.

4.3.2 Tablas de Páginas

En el caso más sencillo, la traducción de direcciones virtuales a direcciones físicas se realiza como acabamos de describir. La dirección virtual se divide en un número de página virtual (bits de orden alto) y un desplazamiento (bits de orden bajo). Por ejemplo, con direcciones virtuales de 16 bits y páginas de 4 KB, los 4 bits superiores especifican una de las 16 páginas virtuales y los 12 bits inferiores especifican el desplazamiento del byte (0 a 4095) dentro de la página seleccionada. Sin embargo también es posible una división con 3 o 5 o algún otro número de bits para la página. Diferentes divisiones implican diferentes tamaños de página.

El número de página virtual se usa como un índice en la tabla de páginas para encontrar la entrada de esa página virtual. De la entrada de la tabla de páginas se obtiene el número de marco de página (sólo en el caso de que la página esté presente en memoria). El número de marco de página se pone a continuación del desplazamiento, reemplazando al número de página virtual, para formar una dirección física que puede enviarse ya a la memoria.

El propósito de la tabla de páginas es establecer una correspondencia aplicando las páginas virtuales sobre los marcos de página. Matemáticamente hablando, la tabla de páginas es una función, con el número de página virtual como argumento y el número de marco de página como resultado. Utilizando el resultado de esta función, el campo de página virtual de una dirección virtual puede reemplazarse por un campo de marco de página, formando así una dirección de memoria física.

A pesar de lo sencillo de esta descripción, hay que resolver los siguientes problemas:

1. La tabla de páginas puede ser extremadamente grande.
2. La traducción de direcciones debe realizarse muy rápidamente.

El primer punto se sigue del hecho de que los ordenadores modernos utilizan direcciones virtuales de por lo menos 32 bits. Por ejemplo, con páginas de 4 KB, un espacio de direcciones de 32 bits tiene un millón de páginas, y un espacio de direcciones de 64 bits tiene más páginas de las que quisiéramos contemplar. Con un millón de páginas en el espacio de direcciones virtual, la tabla de páginas debe tener un millón de entradas, y hay que recordar que cada proceso necesita su propia tabla de páginas (porque tiene su propio espacio de direcciones virtual).

El segundo punto es una consecuencia del hecho de que la traducción de direcciones virtuales a direcciones físicas debe realizarse cada vez que se hace referencia a la memoria. Una instrucción típica tiene una palabra de instrucción y a menudo también un operando en la memoria. Consecuentemente, es necesario hacer una, dos o más referencias a la tabla de páginas por cada instrucción. Si una instrucción tarda, digamos 4 nanosegundos, la consulta a la tabla de páginas deberá hacerse en menos de 1 nanosegundo para evitar que se convierta en un importante cuello de botella.

La necesidad de una traducción de direcciones virtuales rápida y con un gran número de páginas virtuales supone una importante restricción sobre la forma en que se construyen los ordenadores. Aunque el problema es más serio en las máquinas de gama más alta, también es una cuestión a tener en cuenta en los ordenadores de gama baja, donde el coste y la relación precio/rendimiento son críticas. En esta sección y en las siguientes examinaremos el diseño de las tablas de página en detalle y mostraremos varias soluciones hardware que se han utilizado en ordenadores reales.

El diseño más sencillo (al menos conceptualmente) es tener una única tabla de páginas consistente en un array de registros hardware rápidos, con una entrada por cada página virtual, indexado por número de página virtual, como se muestra en la Figura 4-11. Cuando se arranca un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tomados de una copia que se mantiene en memoria principal. Durante la ejecución del proceso, no son necesarias más referencias a la tabla de páginas en memoria. La ventaja de este método es que es simple y no requiere realizar ninguna referencia a la memoria durante la traducción. Una desventaja es que es potencialmente caro (si la tabla de páginas es grande). Tener que cargar la tabla de páginas completa en cada cambio de contexto resulta ineficiente.

En el otro extremo, la tabla de páginas podría residir simplemente en la memoria principal. En ese caso lo único que necesita el hardware es un único registro que apunte al principio de la tabla de páginas. Este diseño permite cambiar el mapa de memoria como parte de un cambio de contexto con sólo recargar un registro. Por supuesto, tiene la desventaja de requerir una o más referencias a memoria, para leer las entradas de la tabla de páginas durante la ejecución de cada instrucción. Por esa razón, casi nunca se adopta este enfoque en su forma más pura, aunque a continuación estudiaremos algunas variaciones que son mucho más eficientes.

Tablas de Páginas Multinivel

Para superar el problema de tener que mantener todo el tiempo en memoria enormes tablas de páginas, muchos ordenadores utilizan una tabla de páginas multinivel. En la Figura 4-12 se muestra un ejemplo sencillo. En la Figura 4-12(a) tenemos una dirección virtual de 32 bits que se divide en un campo *TP1* de 10 bits, un campo *TP2* de 10 bits y un campo *Offset* de desplazamiento de 12 bits. Puesto que los desplazamientos son de 12 bits, las páginas son de 4 KB y que hay un total de 2^{20} páginas.

El secreto del método de la tabla de páginas multinivel consiste en evitar mantener todas las tablas de páginas en memoria todo el tiempo. En particular, las que no se necesiten no deben mantenerse en ella. Por ejemplo, supongamos que un proceso necesita 12 MB, los 4 MB de la parte baja de la memoria para el texto (código) del programa, los siguientes 4 MB para los datos y los 4 MB superiores de la memoria para la pila. Entre la parte superior de los datos y la pila hay un hueco gigantesco que no se usa.

En la Figura 4-12(b) vemos cómo funciona una tabla de páginas de dos niveles en este ejemplo. A la izquierda tenemos la tabla de páginas de primer nivel, con 1024 entradas, que corresponde al campo *TP1* de 10 bits. Cuando se presenta una dirección virtual a la MMU, ésta extrae primero el campo *TP1* y usa ese valor como índice para consultar la tabla de páginas de primer nivel. Cada una de estas 1024 entradas representa 4 MB porque todo el espacio de direcciones virtual de 4 GB (es decir, de 32 bits) se ha dividido en 1024 trozos.

La entrada obtenida al consultar la tabla de páginas de primer nivel proporciona la dirección o el número de marco de página de una tabla de páginas de segundo nivel. La entrada 0 de la tabla de primer nivel apunta a la tabla de páginas correspondiente al texto del programa, la entrada 1 apunta a la tabla de páginas para los datos y la entrada 1023 apunta a la tabla de páginas para la pila. Las demás entradas (sombreadas) no se usan. Se utiliza ahora el campo *TP2* como un índice para la tabla de páginas de segundo nivel seleccionada a fin de encontrar el número de marco de página donde está la página misma.

Por ejemplo, consideremos la dirección virtual de 32 bits 0x00403004 (4.206.596 en decimal), que está a una distancia de 12.292 bytes del principio del segmento de datos. Esta dirección virtual corresponde a *TP1* = 1, *TP2* = 3 y desplazamiento *Offset* = 4. La MMU primero usa *TP1* como un índice para la tabla de páginas de primer nivel y obtiene la entrada 1, que corresponde a las direcciones entre 4M y 8M. Luego usa *TP2* como un índice para la tabla de páginas que se acaba de encontrar y extrae la entrada 3, que corresponde a las direcciones

desde 12.288 hasta 16.383 dentro de su trozo de 4M (es decir, a las direcciones absolutas de 4.206.592 a 4.210.687). Esta entrada contiene el número de marco de página donde está cargada la página que contiene a la dirección virtual 0x00403004. Si esa página no estuviera en la memoria, el bit de presencia en la entrada de la tabla sería cero, provocando una falta de página. Si la página está en la memoria, el número de marco de página tomado de la tabla de páginas de segundo nivel se combina con el desplazamiento (4) para construir una dirección física. Ésta se coloca en el bus y se envía a la memoria.

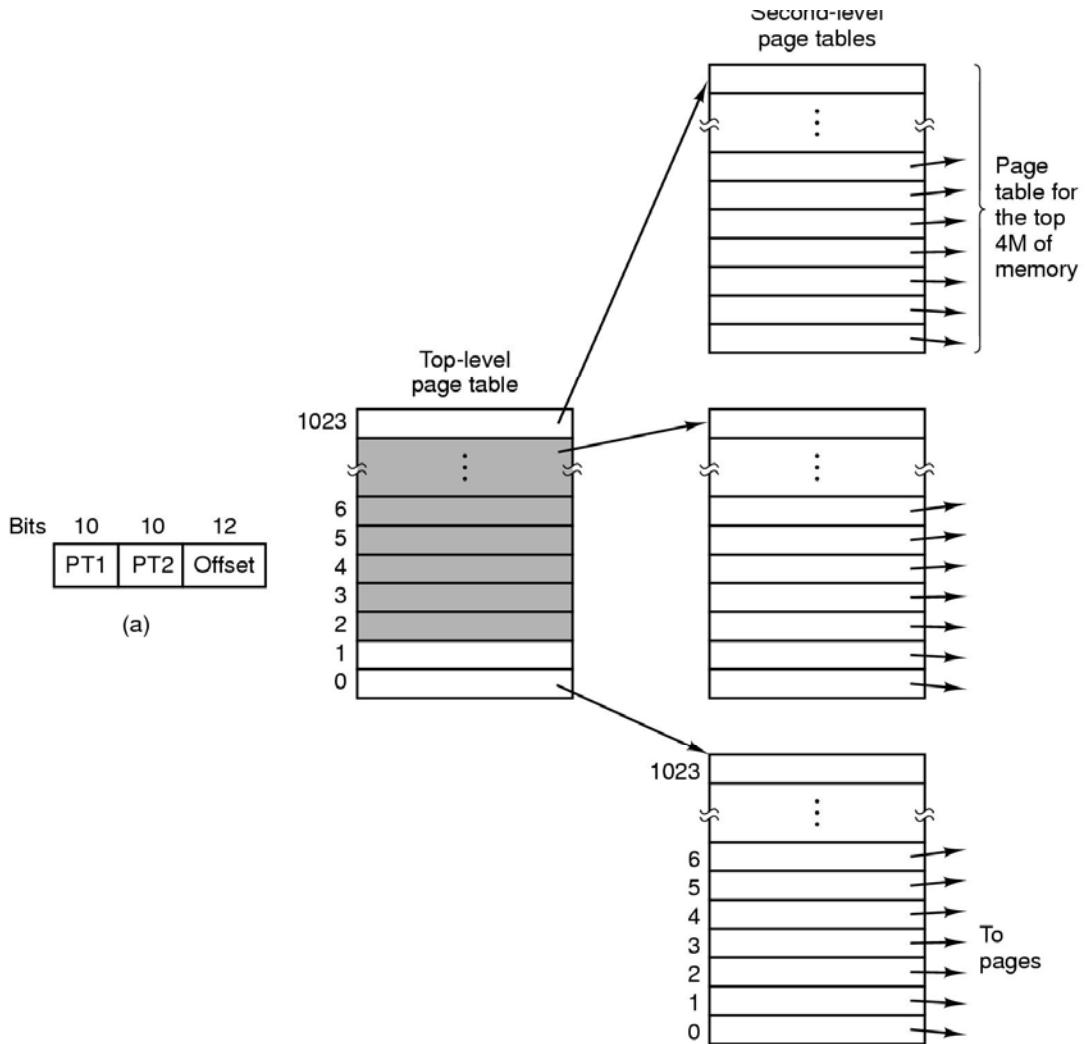


Figura 4-12. (a) Dirección de 32 bits con dos campos de tabla de páginas.
(b) Tabla de páginas de dos niveles.

Lo interesante a destacar de la Figura 4-12 es que aunque el espacio de direcciones contiene más de un millón de páginas, en realidad, sólo se necesitan cuatro tablas de páginas: la tabla de primer nivel y las de segundo nivel para las direcciones de 0 a 4M, de 4M a 8M, y los 4M superiores de la memoria. Los bits de presencia de 1021 entradas de la tabla de páginas de primer nivel están establecidos a 0, forzando una falta de página si se intenta en cualquier momento acceder a ellas. Si eso ocurre, el sistema operativo se percibirá de que el proceso está tratando de referenciar memoria a la que se supone que no debe acceder el proceso, y tomará las medidas apropiadas, como enviarle una señal o eliminarlo (por ejemplo en unix mediante la llamada al sistema kill). En este ejemplo hemos escogido números redondos para los diferentes tamaños y hemos elegido $TP1$ del mismo tamaño que $TP2$, pero por supuesto en la práctica real son posibles también otros valores.

El sistema de tabla de páginas de dos niveles de la Figura 4-12 puede expandirse a tres, cuatro o más niveles. El tener más niveles proporciona una mayor flexibilidad, pero es bastante dudoso que merezca la pena la complejidad adicional que surge cuando se utilizan más de tres niveles.

Estructura de una Entrada de la Tabla de Páginas

Después de tratar la estructura general de las tablas de páginas, vamos a pasar a tratar ahora sobre los detalles de las entradas individuales de la tabla de páginas (a menudo denominadas **descriptores de página**). La organización exacta de una entrada depende mucho de la máquina, pero el tipo de información presente es casi el mismo en todos los ordenadores. En la Figura 4-13 mostramos un ejemplo de entrada de una tabla de páginas. El tamaño varía de un ordenador a otro, pero 32 bits es un tamaño muy común. El campo más importante es el *Número de marco de página*. Después de todo, la función de la tabla de páginas es permitir encontrar ese valor. Junto a él tenemos el bit de *Presencia*. Si este bit es 1, la entrada es válida y puede usarse; si es 0, la página virtual a la que corresponde la entrada no está actualmente en memoria. Acceder a una entrada de la tabla de páginas con el bit de presencia a 0 provoca una falta de página.

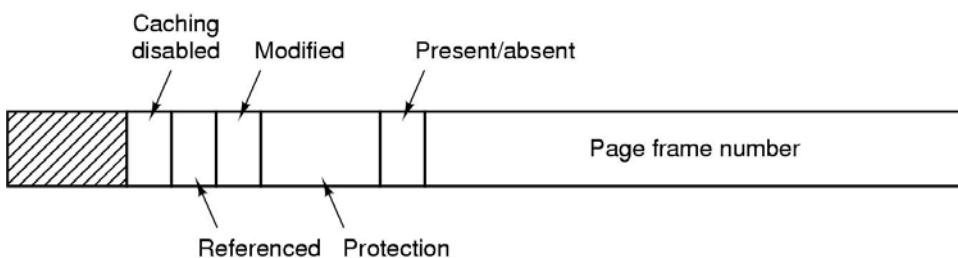


Figura 4-13. Una entrada típica de la tabla de páginas.

Los bits de *Protección* indican qué tipos de acceso están permitidos. En su forma más simple, este campo contiene un bit, que vale 0 si se permite leer y escribir, o 1 si sólo se permite leer. Un esquema más sofisticado utiliza 3 bits, para habilitar/inhibir independientemente la lectura, la escritura y la ejecución de palabras de la página (análogo a los bits *rwx*).

Los bits de página *Modificada* y *Referenciada* siguen la pista del uso de la página. Cuando se escribe en una página, el hardware activa automáticamente el bit de *Modificada*. Éste bit es útil cuando el sistema operativo decide liberar un marco de página ocupado. Si la página que contenía ha sido modificada (es decir, está “sucia”), deberá volver a escribirse en el disco; Si esa página no ha sido modificada (es decir, está “limpia”), simplemente podrá abandonarse, ya que la copia que está en el disco sigue siendo válida. El bit de *Modificada* se denomina también como el **bit de suciedad** (*dirty bit*), pues refleja ese estado de la página.

El bit de *Referenciada* se activa siempre que se referencia una página, ya sea para leer o para escribir. Su función es la de ayudar al sistema operativo a seleccionar la página que elegirá como *victima* cuando se presente una falta de página. Las páginas que no se están usando son mejores candidatas, que las páginas que sí se usan, y este bit juega un importante papel en varios de los algoritmos de sustitución de páginas que estudiaremos más adelante en este capítulo.

Finalmente el último bit permite inhabilitar el uso de la caché para la página. Esta característica es importante en el caso de páginas que contienen direcciones correspondientes a registros de dispositivos, en vez de a posiciones de memoria. Si el sistema operativo está dando

vueltas en un bucle de polling esperando a que algún dispositivo de E/S responda a un comando que se le acaba de enviar, es indispensable que el hardware siga extrayendo la palabra del dispositivo, y que no utilice una copia antigua almacenada en la caché. Con este bit, puede desactivarse el uso de la caché para esa página. Las máquinas que tienen un espacio de E/S separado y no utilizan E/S mapeada en memoria no necesitan ese bit.

Conviene aclarar que la dirección del disco que se utiliza para guardar la página cuando no está en la memoria no forma parte de la tabla de páginas. La razón es sencilla. La tabla de páginas sólo contiene la información que el hardware necesita para traducir direcciones virtuales a direcciones físicas. La información que el sistema operativo necesita para resolver las faltas de página se mantiene en tablas de software (es decir, se calcula mediante un algoritmo) dentro del sistema operativo. El hardware no necesita esa información.

4.3.3 TLBs

En la mayoría de los esquemas de paginación, las tablas de páginas se mantienen en la memoria, debido a su gran tamaño. Potencialmente, este diseño tiene un enorme impacto sobre el rendimiento. Por ejemplo, consideremos una instrucción que copia un registro en otro. En ausencia de paginación, esa instrucción sólo realiza una referencia a la memoria, para extraer la instrucción. Con paginación, pueden ser necesarias referencias adicionales a la memoria para acceder a la tabla de páginas. Puesto que la velocidad de ejecución está generalmente limitada por la velocidad con que la CPU puede obtener instrucciones y datos de la memoria, el tener que hacer dos referencias a la tabla de páginas por cada referencia a la memoria reduce el rendimiento en 2/3. Bajo estas condiciones, nadie utilizaría la paginación.

Los diseñadores de ordenadores han sido conscientes de este problema desde hace muchos años y han encontrado una solución basada en la observación de que los programas tienden a hacer un gran número de referencias a un número pequeño de páginas, y no al revés. Por lo tanto, solamente se lee con mucha frecuencia una pequeña fracción de las entradas de la tabla de páginas; mientras que el resto de entradas apenas se usan.

La solución ideada consiste en equipar al ordenador con un pequeño dispositivo de hardware que traduce direcciones virtuales a direcciones físicas, sin hacer uso de la tabla de páginas. Este dispositivo, denominado **TLB** (*Tranlation Lookaside Buffer*), también conocido como **memoria asociativa**, se ilustra en la Figura 4-14. Usualmente está dentro de la MMU y consiste en un pequeño número de entradas, ocho en este ejemplo, pero casi nunca más de 64. Cada entrada contiene información sobre una página, incluido el número de página virtual, un bit que se establece cuando la página se modifica, el código de protección (permisos para leer/escribir/ejecutar) y el marco de página físico en el cual está cargada la página. Estos campos tienen una correspondencia uno a uno con los campos de la tabla de páginas. Otro bit indica si la entrada es válida (o sea, si se está usando) o no.

| Valid | Virtual page | Modified | Protection | Page frame |
|--------------|---------------------|-----------------|-------------------|-------------------|
| 1 | 140 | 1 | RW | 31 |
| 1 | 20 | 0 | R X | 38 |
| 1 | 130 | 1 | RW | 29 |
| 1 | 129 | 1 | RW | 62 |
| 1 | 19 | 0 | R X | 50 |
| 1 | 21 | 0 | R X | 45 |
| 1 | 860 | 1 | RW | 14 |
| 1 | 861 | 1 | RW | 75 |

Figura 4-14. Una TLB para acelerar la paginación.

Un ejemplo que podría generar la TLB de la Figura 4-14 es un proceso que está ejecutando un bucle que abarca las páginas virtuales 19, 20 y 21, por lo que estas entradas de la TLB tienen códigos de protección para leer y ejecutar. Los datos principales que se están usando (por ejemplo, un array que se está procesando) están en las páginas 129 y 130. La página 140 contiene los índices que se usan en los cálculos con el array. Finalmente, la pila está en las páginas 860 y 861.

Vamos a ver ahora cómo funciona la TLB. Cuando se presenta una dirección virtual a la MMU para que la traduzca, el hardware comprueba primero si su número de página virtual está presente en la TLB o no, comparándolo con todas sus entradas de manera simultánea (es decir, en paralelo). Si encuentra ese número de página y el acceso no viola los bits de protección, el número de marco de página se toma directamente de la TLB, sin recurrir a la tabla de páginas. Si el número de página virtual está presente en la TLB pero la instrucción está tratando de escribir en una página de sólo lectura, se generará un fallo de protección, de la misma forma que se generaría a partir de la correspondiente entrada de la tabla de páginas.

Lo interesante es lo qué sucede cuando el número de página virtual no está en la TLB. La MMU detecta este fallo y realiza una consulta ordinaria de la tabla de páginas. Luego desaloja una de las entradas de la TLB y la sustituye por la entrada de la tabla de páginas que acaba de encontrar. De este modo, si la página en cuestión se vuelve a usar pronto, la segunda vez sí que se encontrará en la TLB. Cuando se desaloja una entrada de la TLB, el bit de modificada se copia de vuelta a la entrada correspondiente de la tabla de páginas en la memoria. Los demás valores ya están ahí. Cuando la TLB se carga de la tabla de páginas, todos los campos se toman de la memoria.

Software de Gestión de la TLB

Hasta aquí hemos supuesto que toda máquina con memoria virtual paginada tiene tablas de páginas que el hardware reconoce, más una TLB. En este diseño, el hardware de la MMU realiza toda la gestión de la TLB y el tratamiento de las faltas de TLB. Sólo se realizan traps al sistema operativo cuando se referencia una página que no está en la memoria.

En el pasado, esta suposición era válida. Sin embargo, muchos ordenadores RISC modernos incluidos el SPARC, MIPS, Alpha y HP PA, realizan prácticamente toda esa gestión de páginas por software. En tales máquinas, el sistema operativo carga explícitamente las entradas de la TLB. Cuando tiene lugar una falta de TLB, en vez de ser la MMU la que consulta directamente las tablas de páginas para encontrar y extraer la entrada de la página requerida, la MMU simplemente genera una falta de TLB y deja la resolución del problema en manos del sistema operativo. El sistema deberá encontrar la página, quitar una entrada de la TLB, cargar la nueva y reiniciar la instrucción que provocó el fallo. Por supuesto, todo esto debe hacerse ejecutando tan solo un puñado de instrucciones porque las faltas de TLB ocurren con mucha más frecuencia que las faltas de página.

Por sorprendente que parezca, si la TLB tiene un tamaño razonablemente grande (digamos, 64 entradas) para que la tasa de fallos de TLB no sea muy alta, la gestión de la TLB por software resulta aceptablemente eficiente. Lo que se gana principalmente con esto es tener una MMU mucho más simple, lo que deja libre una considerable área del chip de la CPU para cachés y otros recursos que pueden mejorar el rendimiento. La gestión de la TLB por software se analiza en Uhlig y otros (1994).

Se han desarrollado diversas estrategias para mejorar el rendimiento en máquinas que realizan la gestión de la TLB por software. Una de ellas busca reducir las faltas de TLB y, al mismo tiempo, reducir el coste de las faltas de TLB cuando tengan lugar (Bala y otros, 1994). Para reducir las faltas de TLB, a veces el sistema operativo puede utilizar su intuición para

intentar determinar qué páginas tienen una gran probabilidad de ser utilizadas a continuación, precargando en la TLB las entradas correspondientes. Por ejemplo, si un proceso cliente envía un mensaje a un proceso servidor que está en la misma máquina, es muy probable que el servidor tenga que ejecutarse pronto. Sabiendo esto, el sistema puede determinar, mientras procesa la llamada al sistema correspondiente al **enviar**, dónde están las páginas de código, datos y pila del servidor, y colocar sus entradas en la TLB evitando así que se produzcan algunas faltas de TLB.

La forma normal de procesar una falta de TLB, bien sea por hardware o por software, es consultar la tabla de páginas y realizar las operaciones de indexación necesarias para localizar la página a la que se hizo referencia. El problema de realizar esta búsqueda por software es que las páginas que contienen la tabla de páginas podrían no estar en la TLB, lo que provocaría faltas de TLB adicionales durante el procesamiento. Estas faltas pueden reducirse si se mantiene una caché grande de software (por ejemplo, de 4 KB) con entradas de la TLB, en un lugar fijo cuya página siempre se conserve en la TLB. Si el sistema operativo consulta primero la caché de software, podrán reducirse considerablemente las faltas de TLB.

4.3.4 Tablas de Páginas Invertidas

Las tablas de páginas tradicionales del tipo que hemos descrito hasta ahora requieren una entrada por cada página virtual, ya que están indexadas por número de página virtual. Si el espacio de direcciones consta de 2^{32} bytes, con 4096 bytes por página, se necesitarán más de un millón de entradas en la tabla de páginas. Como mínimo, la tabla ocupará 4 MB. En los sistemas más grandes es probable que sí sea factible tener tablas tan grandes.

Sin embargo, a medida que se hace más común el uso de ordenadores de 64 bits, la situación cambia drásticamente. Si el espacio de direcciones tiene ahora 2^{64} bytes, con páginas de 4 KB, necesitamos una tabla con 2^{52} entradas. Si cada entrada requiere 8 bytes, la tabla ocupará más de 30 millones de gigabytes. Dedicar tanto espacio sólo a la tabla de páginas no es factible, ni ahora ni en un futuro cercano, y tal vez nunca. Consecuentemente, se requiere una solución diferente para los espacios de direcciones virtuales paginados de 64 bits.

Una solución es la **tabla de páginas invertida**. En este diseño, hay una entrada por cada marco de página en la memoria real, en vez de una entrada por cada página del espacio de direcciones virtual. Por ejemplo, con direcciones virtuales de 64 bits, páginas de 4 KB y 256 MB de RAM, una tabla de páginas invertida sólo requiere 65.536 entradas. La entrada indica qué proceso y qué página virtual de ese proceso está cargada en el marco correspondiente.

Aunque las tablas de páginas invertidas ahorran una enorme cantidad de espacio, al menos cuando el espacio de direcciones virtual es mucho más grande que la memoria física, tienen un serio inconveniente: la traducción de direcciones virtuales a físicas es mucho más difícil. Cuando el proceso n hace referencia a la página virtual p , el hardware no puede ya encontrar la página física utilizando p como un índice sobre la tabla de páginas del proceso n , sino que debe buscar una entrada (n, p) recorriendo la tabla de páginas invertida. Además, esa búsqueda debe efectuarse en cada referencia a la memoria, y no sólo cuando se generan faltas de página. Recorrer una tabla de 64 K en cada referencia a la memoria no es la forma de conseguir que la máquina sea excepcionalmente rápida.

La forma de resolver este dilema es sacar partido de TLB. Si la TLB puede contener todas las páginas más utilizadas, la traducción puede ser tan rápida como con las tablas de páginas normales. Sin embargo, ante una falta de TLB, habrá que realizar una búsqueda por software en la tabla de páginas invertida. Un modo viable de hacer esa búsqueda a partir de la dirección virtual es tener una tabla hash. Todas las páginas virtuales que estén en la memoria en ese momento y que tengan el mismo valor hash están enlazadas, como se muestra en la Figura

4-15. Si la tabla hash tiene tantas entradas como marcos de página físicos tiene la máquina, las cadenas tendrán una longitud media de una sola entrada, lo cual acelera de forma considerable la traducción. Una vez obtenido el marco, se coloca en la TLB el nuevo par (página, marco).

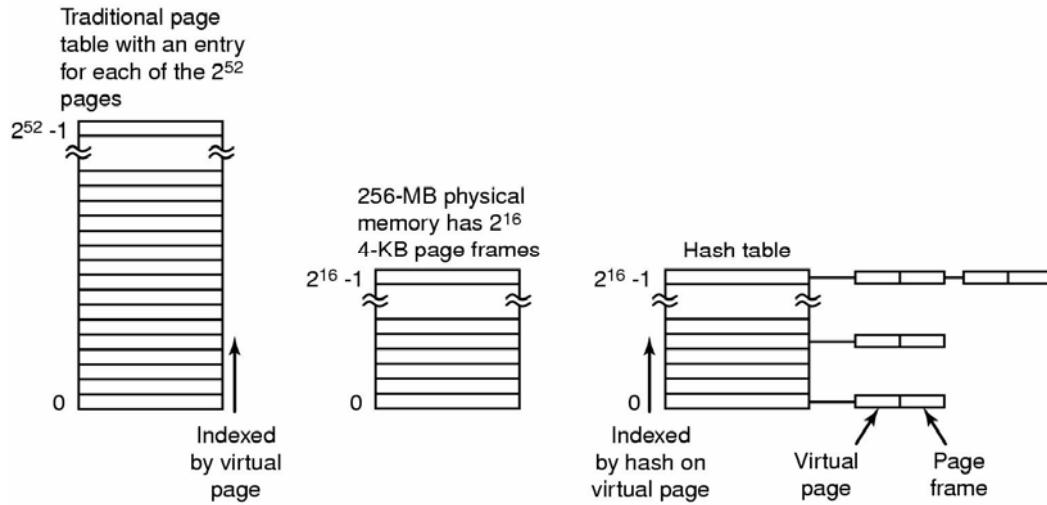


Figura 4-15. Comparación entre una tabla de páginas tradicional y una tabla de páginas invertida.

En la actualidad, las tablas de páginas invertidas se utilizan en algunas estaciones de trabajo IBM y Hewlett-Packard y se harán más comunes a medida que se generalice el uso de máquinas de 64 bits. Se presentan otros enfoques para manejar grandes memorias virtuales en Huck y Hays (1993), Talluri y Hill (1994) y Talluri y otros (1995).

4.4 ALGORITMOS DE SUSTITUCIÓN DE PÁGINAS

Cuando se presenta una falta de página, el sistema operativo tiene que escoger la página que se quitará de la memoria para hacer sitio a la página que se traerá del disco. Si la página a quitar se modificó mientras estaba en la memoria, deberá volverse a escribir en el disco para mantener la copia en el disco actualizada. Sin embargo, si la página no se ha modificado (por ejemplo, si contiene código del programa), la copia en disco estará ya actualizada y no será necesario escribirla de nuevo. La página leída simplemente sobrescribe a la que se quita de la memoria.

Aunque sería posible escoger en cada falta de página una página al azar para quitarla de la memoria, el rendimiento del sistema mejora mucho si se escoge como *victima* una página que no se usa mucho. Si se sustituye una página muy utilizada, lo más seguro es que pronto tenga que volverse a traer a la memoria, con el consiguiente gasto de tiempo extra. Se ha trabajado mucho sobre el tema de los algoritmos de sustitución de páginas, tanto desde el punto de vista teórico como experimental. A continuación describiremos algunos de los algoritmos más importantes.

Vale la pena resaltar que el problema de la “sustitución de páginas” se da también en otras áreas del diseño de los ordenadores. Por ejemplo, la mayoría de los ordenadores tienen una o más memorias cachés que contienen bloques de memoria recientemente usados de 32 o 64 bytes. Cuando se llena la caché, hay que escoger algún bloque para sustituirlo. Este problema es idéntico al de sustitución de páginas, sólo que se produce en una escala de tiempo más corta (se tiene que realizar en unos cuantos nanosegundos, no en milisegundos como la sustitución de páginas). La escala de tiempo es corta porque las faltas de bloque en la caché se resuelven trayendo un bloque de la memoria principal, lo que no implica tiempos de posicionamiento de la cabeza lectora ni latencia rotacional.

Un segundo ejemplo es un servidor web. El servidor puede mantener en su memoria caché cierto número de páginas web muy utilizadas. Sin embargo, cuando la caché se llena y se hace referencia a una nueva página, hay que decidir qué página web se sustituye. Las consideraciones son similares al caso de las páginas de la memoria virtual, salvo por el hecho de que las páginas web nunca se modifican en la caché, así que la copia en disco siempre está actualizada. En un sistema de memoria virtual, las páginas que están en la memoria principal pueden estar limpias o sucias.

4.4.1 El Algoritmo de Sustitución de Páginas Óptimo

El mejor algoritmo de sustitución de páginas posible es fácil de describir pero imposible de implementar. En el momento en que tiene lugar una falta de página, hay un cierto conjunto de páginas en la memoria. Alguna de esas páginas puede ser referenciada en la siguiente instrucción (en concreto la página que contiene esa instrucción). Otras páginas puede ser que no se refieran hasta 10, 100 o quizás 1000 instrucciones después. Cada página puede etiquetarse con el número de instrucciones que se ejecutarán antes de que se haga la primera referencia a esa página.

El algoritmo de sustitución óptimo simplemente dice que debe sustituirse la página con el valor más alto de esa etiqueta. Si faltan 8 millones de instrucciones para que se utilice cierta página y faltan 6 millones de instrucciones para que se utilice otra, la sustitución de la primera página retrasa lo más posible la falta de página que ocurrirá cuando esa página vuelva a referenciarse. Los ordenadores, lo mismo que las personas, tratan de retrasar lo más posible los sucesos desagradables.

El único problema con este algoritmo es que es imposible llevarlo a la práctica. En el momento en que se presenta la falta de página, el sistema operativo no tiene ninguna forma de saber cuándo se volverá a hacer referencia a cada una de las páginas. (Vimos una situación similar anteriormente con el algoritmo de planificación del trabajo más corto el primero: ¿cómo puede saber el sistema qué trabajo es el más corto?) De todas formas, ejecutando un programa en un simulador y llevando la cuenta de todas las referencias a las páginas, es posible implementar el algoritmo de sustitución de páginas óptimo en la *segunda ejecución*, utilizando la información de las referencias a las páginas obtenida durante la *primera ejecución*.

De este modo, es posible comparar el rendimiento de algoritmos realizables con el rendimiento del mejor algoritmo posible. Si, por ejemplo, un sistema operativo logra un rendimiento sólo un 1% peor que el del algoritmo óptimo, por más esfuerzo que se dedique a encontrar un algoritmo mejor, lo más que se conseguirá será una mejora del 1%.

A fin de evitar cualquier posible confusión, conviene aclarar que este registro de referencias a páginas es válido sólo para el programa que se acaba de simular y sólo si se procesa con los mismos datos de entrada. Por lo tanto, el algoritmo de sustitución de páginas que se derive de la simulación será específico para ese programa y esos datos de entrada. Aunque este método es útil para evaluar algoritmos de sustitución de páginas, no sirve en los sistemas reales. A continuación estudiaremos algoritmos que sí *son* útiles en los sistemas reales.

4.4.2 El Algoritmo de Sustitución de Páginas NRU

Para hacer posible que el sistema operativo pueda recoger estadísticas útiles sobre qué páginas se están usando y cuáles no, casi todos los ordenadores con memoria virtual asocian a cada página dos bits de estado R y M . R se activa cada vez que referencia la página (para leer o escribir). M se activa cada vez que se escribe en la página (es decir, se modifica). Los bits forman parte de la entrada correspondiente de la tabla de páginas, como se mostró en la Figura 4-13. Es importante tener en cuenta que dichos bits deben actualizarse en cada referencia a la memoria, por lo que es esencial que sea el hardware quien los active. Una vez activado un bit, conserva el valor 1 hasta que el sistema operativo lo desactiva a 0 por software.

Si el hardware no cuenta con esos bits, pueden simularse como sigue. Cuando se inicia un proceso todas sus entradas en la tabla de páginas se marcan como no presentes en la memoria. Tan pronto como se haga referencia a una página, tendrá lugar una falta de página. El sistema operativo activa entonces el bit R (en sus tablas internas), modifica la entrada de la tabla de páginas de modo que apunte a la página correcta, con modo de acceso READ ONLY, y reiniciará la instrucción. Si después se escribe en la página, ocurrirá otro fallo de página, lo que permitirá al sistema operativo activar el bit M y cambiar el modo de la página a READ/WRITE.

Los bits R y M pueden servir para elaborar un algoritmo de sustitución de páginas sencillo. Cuando se inicia un proceso, el sistema operativo pone a cero ambos bits para todas sus páginas. De forma periódica (por ejemplo en cada interrupción de reloj), el bit R se pone a 0 para distinguir las páginas que no se han referenciado últimamente de aquéllas que sí lo han sido.

Cuando se presenta una falta de página, el sistema operativo inspecciona todas las páginas dividiéndolas en cuatro categorías según los valores actuales de sus bits R y M :

- Clase 0: página ni referenciada, ni modificada.
- Clase 1: página no referenciada y modificada.
- Clase 2: página referenciada y no modificada.
- Clase 3: página referenciada y modificada.

Aunque a primera vista parece imposible que alguna página pertenezca a la clase 1, eso ocurre cuando una interrupción de reloj desactiva el bit R de una página de la clase 3. Las interrupciones de reloj no desactivan el bit M porque esa información es necesaria para saber si la página debe actualizarse en el disco o no. Desactivar R pero no M da lugar a una página de la clase 1.

El algoritmo de sustitución de la página **no recientemente usada** (**NRU**; *Not Recently Used*) sustituye al azar una página de la clase de número más bajo que no esté vacía. Este algoritmo se basa en la suposición implícita de que es mejor sustituir una página modificada a la que no se ha hecho referencia en por lo menos un tic de reloj (que típicamente es de unos 20 milisegundos), en vez de una página limpia que se está usando mucho. El principal atractivo de NRU es que es fácil de entender, tiene una implementación moderadamente eficiente y proporciona un rendimiento que, aunque ciertamente no es óptimo, puede ser aceptable.

4.4.3 El Algoritmo de Sustitución de Páginas FIFO

Otro algoritmo de paginación con una baja sobrecarga para el sistema es el algoritmo **primero en entrar primero en salir** (**FIFO**; *First-In First-Out*). Para ilustrar su funcionamiento, consideremos un supermercado que tiene suficientes estantes para almacenar exactamente k productos diferentes. Un día, cierta compañía introduce un nuevo tipo de producto: yogurt orgánico en polvo instantáneo que puede reconstituirse en un horno microondas. El producto es un éxito inmediato, así que nuestro supermercado finito tiene que deshacerse de un producto antiguo para poder tener el nuevo producto en stock.

Una posibilidad es buscar el producto que el supermercado ha tenido en existencias más tiempo (por ejemplo, algo que comenzó a vender hace 120 años) y deshacerse de él alegando que ya no le interesa a nadie. En efecto, el supermercado mantiene una lista enlazada de todos los productos que vende actualmente, en el orden en el que se introdujeron. El nuevo producto se coloca al final de la lista; el que está al principio, se retira del supermercado.

Como algoritmo de sustitución de páginas, puede aplicarse la misma idea. El sistema operativo mantiene una lista de todas las páginas que están actualmente en la memoria, con la más antigua al principio de la lista y la más nueva al final. Al producirse una falta de página, se sustituye la página que está al principio de la lista y la nueva se añade al final. En el caso de una tienda, FIFO podría eliminar la cera para el bigote, pero también podría eliminar harina, sal o mantequilla. En el caso de los ordenadores surge el mismo problema. Por esa razón, raramente se usa FIFO en su forma pura.

4.4.4 El Algoritmo de Sustitución de Páginas de la Segunda Oportunidad

Una modificación sencilla del algoritmo FIFO que evita el problema de sustituir una página que se usa mucho consiste en examinar el bit R de la página más antigua. Si es 0, quiere decir que la página es antigua y además no se usa. Por lo tanto, se la sustituye de inmediato. Si el bit R es 1, se pone a 0, se coloca la página al final de lista de páginas y su instante de carga se actualiza como si acabara de cargarse en la memoria. Luego se continúa la búsqueda.

El funcionamiento de este algoritmo, denominado **de la segunda oportunidad**, se muestra en la Figura 4-16. En la Figura 4-16(a) vemos que las páginas de la A a la H se mantienen en una lista enlazada, ordenadas según la hora a la que llegaron a la memoria.

Supongamos que se produce una falta de página en el instante 20. La página más antigua es la A , que llegó en el instante 0, cuando se inició el proceso. Si el bit R de A está a 0, se expulsa a A de la memoria, actualizándola en el disco (si está modificada) o abandonándola

simplemente (si está limpia). Pero si el bit R está a 1, A se coloca al final de la lista y su “instante de carga” se cambia al instante actual (20). También se desactiva el bit R . La búsqueda de una página apropiada continúa con B .

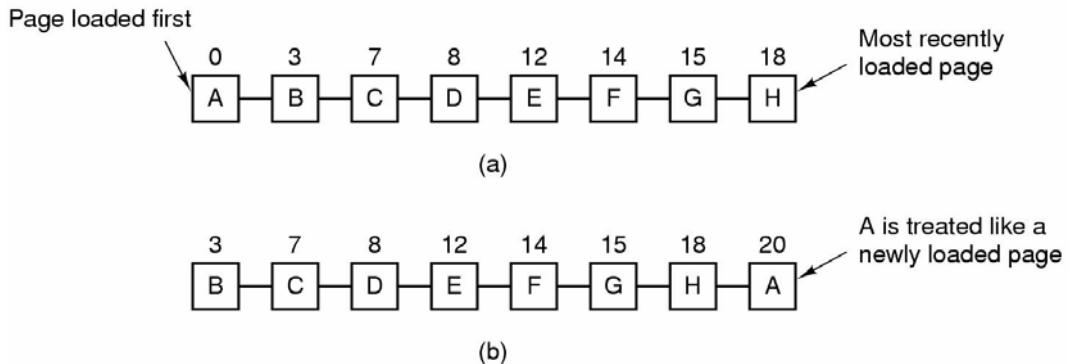


Figura 4-16. Funcionamiento del algoritmo de la segunda oportunidad. (a) Páginas en orden FIFO. (b) Lista de páginas si se produce una falta de página en el instante 20 y el bit R de la página A está activado. Los números que aparecen encima de las páginas representan sus instantes de carga.

Lo que este algoritmo está haciendo es buscar una página antigua a la que no se haya hecho referencia durante el intervalo de reloj anterior. Si se ha hecho referencia a todas las páginas, el algoritmo de la segunda oportunidad degenera en el algoritmo FIFO puro. De manera más específica, imaginemos que los bits R de todas las páginas de la Figura 4-16(a) están activados. El sistema operativo pasa las páginas una por una al final de la lista, poniendo a 0 el bit R cada vez que añade una página al final. Eventualmente, el sistema vuelve a examinar la página A , que ahora tiene su bit R a 0. En ese momento, se sustituye la página A . Así, el algoritmo siempre termina.

4.4.5 El Algoritmo de Sustitución de Páginas del Reloj

Aunque el algoritmo de la segunda oportunidad es un algoritmo razonable, es innecesariamente inefficiente porque continuamente cambia las páginas de lugar dentro de la lista. Una estrategia mejor sería mantener todos los marcos de página sobre una lista circular en la forma de un reloj, como se muestra en la Figura 4-17. Una manecilla apunta a la página más antigua.

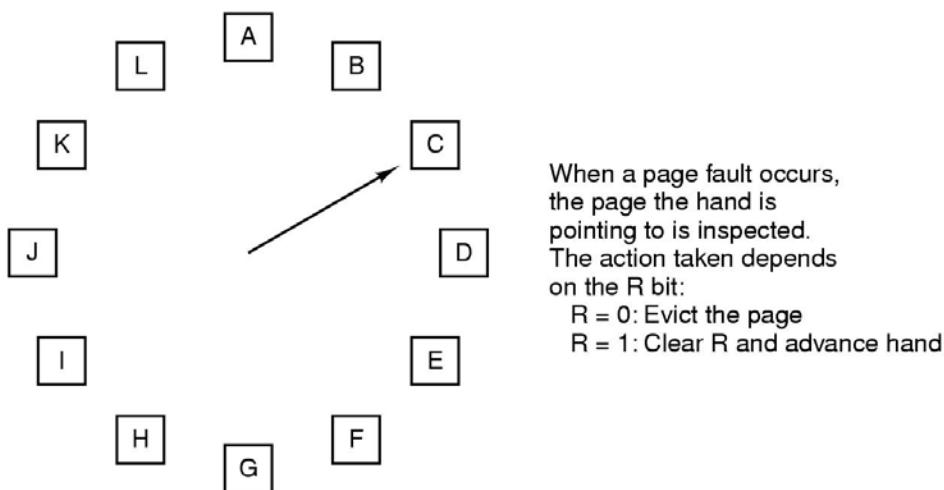


Figura 4-17. El algoritmo de sustitución de páginas del reloj.

Cuando se presenta una falta de página, se examina la página a la que apunta la manecilla. Si su bit R es 0, dicha página se sustituye, insertando la nueva en su lugar y adelantando una posición la manecilla. Si R es 1, se cambia a 0 y la manecilla se adelanta a la siguiente página. Este proceso se repite hasta hallar una página con $R = 0$. No es sorprendente que a este algoritmo se le llame el **algoritmo del reloj**. La única diferencia respecto al de la segunda oportunidad radica en su implementación.

4.4.6 El Algoritmo de Sustitución de Páginas LRU

Una buena aproximación al algoritmo óptimo se basa en la observación de que las páginas que se han utilizado mucho en las últimas instrucciones probablemente se utilizarán también mucho en las instrucciones siguientes. Por el contrario, las páginas que no se han utilizado durante siglos probablemente seguirán sin utilizarse durante mucho tiempo. Esta idea sugiere un algoritmo realizable: cuando se produzca una falta de página, sustituir la página que lleve más tiempo sin utilizarse. Tal estrategia se denomina paginación **menos recientemente utilizada** o **LRU** (*Least Recently Used*).

Aunque LRU es teóricamente realizable, tiene un coste elevado. Para implementarlo fielmente es necesario mantener una lista enlazada de todas las páginas que están en la memoria, con la página que se utilizó más recientemente al principio y la página menos recientemente utilizada al final. La dificultad consiste en que la lista debe actualizarse en cada referencia a la memoria. Encontrar una página en la lista, sacarla de la lista y reinserirla al frente es una operación muy lenta, incluso realizándose por hardware (suponiendo que pudiera construirse tal hardware).

Sin embargo, hay otras formas de implementar LRU utilizando hardware especial. Consideremos primero el método más sencillo, que requiere equipar al hardware con un contador de 64 bits, C , que se incrementa automáticamente después de cada instrucción. Además, se añade a cada entrada de la tabla de páginas un campo lo suficientemente grande como para contener al contador. Después de cada referencia a la memoria, el valor actual de C se almacena en la entrada de la tabla de páginas correspondiente a la página a la que se acaba de hacer referencia. Cuando se presenta una falta de página, el sistema operativo examina todos los contadores de la tabla de páginas para encontrar el menor. Esa página es la menos recientemente usada.

Vamos a examinar ahora un segundo algoritmo LRU por hardware. Para una máquina con n marcos de página, el hardware de LRU puede mantener una matriz de $n \times n$ bits, los cuales son todos cero al principio. Cada vez que se hace referencia al marco de página k , el hardware pone a 1 primero todos los bits de la fila k y luego pone a 0 todos los bits de la columna k . En cualquier instante, la fila cuyo valor binario sea el más bajo corresponde al marco menos recientemente utilizado, la fila con el siguiente valor más bajo será la del siguiente marco menos recientemente utilizado, y así de forma sucesiva. El funcionamiento de este algoritmo se ilustra en la Figura 4-18 para cuatro marcos de página y la secuencia de referencias a páginas

0 1 2 3 2 1 0 3 2 3

Después de la primera referencia a la página 0, tenemos la situación de la Figura 4-18(a). Después de la referencia a la página 1, tenemos la situación de la Figura 4-18(b), y así de forma sucesiva.

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |

(a)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

(b)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(c)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(d)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(e)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(f)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(g)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(h)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(i)

| Page | | | |
|------|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(j)

Figura 4-18. LRU utilizando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

4.4.7 Simulación del Algoritmo LRU por Software

Aunque los dos algoritmos LRU anteriores son en principio realizables, pocas máquinas (si es que hay alguna) cuentan con este hardware, así que son de poca utilidad para el diseñador de sistemas operativos que está creando un sistema para una máquina que no cuenta con tal hardware. Se necesita una solución que pueda implementarse por software. Una posibilidad es el algoritmo de la página **no frecuentemente utilizada** (NFU; *Not Frequently Used*). NFU requiere por cada página un contador software asociado con valor inicial cero. En cada interrupción de reloj, el sistema operativo explora todas las páginas que están en la memoria. Para cada página, el bit *R*, que es 0 o 1, se suma al contador. De hecho, los contadores son un intento de llevar la cuenta de cuán a menudo se han referenciado las páginas. Cuando se presenta una falta de página, se escoge para ser reemplazada la página con el contador más bajo.

El problema principal con NFU es que nunca olvida nada. Por ejemplo, en un compilador de varias pasadas, las páginas que se utilizaron mucho durante la pasada 1 podrían seguir teniendo un valor alto del contador en las pasadas posteriores. De hecho, si la pasada 1 es la de mayor tiempo de ejecución de todas, las páginas que contienen el código de las pasadas subsiguientes podrían tener siempre contadores más bajos que las páginas de la pasada 1. Eso haría que el sistema operativo sustituyese páginas útiles, en vez de páginas que ya no se usan.

Afortunadamente, una pequeña modificación del algoritmo NFU permite simular al LRU muy bien. Tal modificación tiene dos partes. Primero, todos los contadores se desplazan un bit a la derecha antes de añadir el bit *R*. Segundo, el bit *R* se añade al bit más a la izquierda, en vez de al bit más a la derecha.

La Figura 4-19 ilustra el funcionamiento del algoritmo modificado, conocido como algoritmo de **envejecimiento** (*aging*). Supongamos que después del primer tic de reloj los bits *R* de las páginas 0 a 5 tienen los valores 1, 0, 1, 0, 1 y 1, respectivamente (la página 0 es 1, la página 1 es 0, la página 2 es 1, etc.). Dicho de otro modo, entre el tic 0 y el 1 se hizo referencia a las páginas 0, 2, 4 y 5, lo cual activó sus bits *R*, mientras que los demás siguen a 0. Después de haber desplazado los seis contadores y haber insertado el bit *R* a la izquierda, éstos tienen los valores que se muestran en la Figura 4-19(a). Las siguientes cuatro columnas muestran los seis contadores después de los cuatro tics de reloj siguientes.

| | R bits for pages 0-5, clock tick 0 | R bits for pages 0-5, clock tick 1 | R bits for pages 0-5, clock tick 2 | R bits for pages 0-5, clock tick 3 | R bits for pages 0-5, clock tick 4 |
|------|------------------------------------|------------------------------------|------------------------------------|------------------------------------|------------------------------------|
| Page | 1 0 1 0 1 1 | 1 1 0 0 1 0 | 1 1 0 1 0 1 | 1 0 0 0 1 0 | 0 1 1 0 0 0 |
| 0 | 10000000 | 11000000 | 11100000 | 11110000 | 01111000 |
| 1 | 00000000 | 10000000 | 11000000 | 01100000 | 10110000 |
| 2 | 10000000 | 01000000 | 00100000 | 00100000 | 10001000 |
| 3 | 00000000 | 00000000 | 10000000 | 01000000 | 00100000 |
| 4 | 10000000 | 11000000 | 01100000 | 10110000 | 01011000 |
| 5 | 10000000 | 01000000 | 10100000 | 01010000 | 00101000 |

(a) (b) (c) (d) (e)

Figura 4-19. El algoritmo de envejecimiento simula por software al LRU. Se muestran seis páginas durante cinco tics de reloj. Los cinco tics se representan de la (a) a la (e).

Cuando ocurre una falta de página, se sustituye la página cuyo contador es menor. Es claro que una página a la que no se ha hecho referencia durante, digamos, cuatro tics de reloj, tendrá cuatro ceros a la izquierda en su contador y, por lo tanto, tendrá un valor más bajo que el contador de una página a la que no se ha hecho referencia durante sólo tres tics.

Este algoritmo difiere del LRU en dos aspectos. Consideremos las páginas 3 y 5 de la Figura 4-19(e). No se ha referenciado ninguna durante dos tics de reloj; ambas se referenciaron en el tic previo a esos dos. Según LRU, si es preciso sustituir una página, deberíamos escoger una de esas dos. El problema es que no sabemos cuál de ellas fue la última que se referenció en el intervalo entre el tic 1 y el 2. Al guardar sólo un bit por intervalo de tiempo hemos perdido la capacidad para distinguir las referencias que se hicieron al principio del intervalo de reloj, de las que se hicieron después. Lo único que podemos hacer es sustituir la página 3, porque la 5 también se referenció dos tics antes y la 3 no.

La segunda diferencia entre LRU y el envejecimiento es que en este último algoritmo los contadores tienen un número finito de bits, 8 bits en este ejemplo. Supongamos que dos páginas tienen un valor de contador 0. Lo único que podemos hacer es escoger una de ellas al azar. En realidad, bien podría ser que una de las páginas se hubiera referenciado por última vez hace 9 tics y que la otra se hubiera referenciado por última vez hace 1000. No hay forma de saberlo. Sin embargo, en la práctica 8 bits suelen ser suficientes si el tic de reloj dura alrededor de 20 milisegundos. Si no se ha referenciado una página en 160 milisegundos, seguramente no sea una página muy importante.

4.4.8 El Algoritmo de Sustitución de Páginas del Conjunto de Trabajo

En la paginación, en su forma más pura, los procesos comienzan su ejecución sin tener ninguna página cargada en la memoria. Tan pronto como la CPU trate de extraer la primera instrucción, se producirá una falta de página y el sistema operativo traerá del disco la página que contiene la primera instrucción. Por lo general, enseguida se producen también faltas de páginas correspondientes a las variables globales y la pila. Después de un cierto tiempo, el proceso tiene casi todas las páginas que necesita y su ejecución se desarrolla de forma estable con relativamente pocas faltas de página. Tal estrategia se denomina **paginación bajo demanda** porque las páginas se cargan sólo cuando se necesitan, nunca por adelantado.

Desde luego, es bastante fácil escribir un programa de prueba que lea sistemáticamente todas las páginas de un espacio de direcciones grande, generando tantas faltas de página que no haya suficientes marcos de memoria principal para contener a todas esas páginas. Afortunadamente, la mayoría de los procesos no funcionan así. Los procesos muestran una marcada **localización de sus referencias** a memoria, lo que significa que durante cualquier fase de su ejecución, el proceso sólo hace referencia a una fracción relativamente pequeña de sus páginas. Por ejemplo, cada pasada de un compilador de varias pasadas sólo hace referencia a una fracción de todas las páginas, y a una fracción distinta en cada pasada.

El conjunto de páginas que un proceso está utilizando en un momento dado se denomina su **conjunto de trabajo** (Denning, 1968; Denning, 1980). Si todo el conjunto de trabajo está en la memoria, el proceso se ejecuta sin provocar muchas faltas de página hasta que pasa a otra fase de su ejecución (por ejemplo, la siguiente pasada del compilador). Si la memoria disponible es demasiado pequeña para contener todo el conjunto de trabajo, el proceso provocará muchas faltas de página y se ejecutará lentamente porque la ejecución de una instrucción tarda sólo unos cuantos nanosegundos, pero la lectura de una página del disco suele tardar 10 milisegundos. A razón de una o dos instrucciones ejecutadas cada 10 milisegundos, el programa tardará siglos en terminar. Cuando un programa provoca una falta de página tras cada pocas instrucciones se dice que está produciendo **trasiego** de páginas (*thrashing*) (Denning, 1980b).

En un sistema multiprogramado, con frecuencia los procesos se transfieren al disco (es decir, todas sus páginas se expulsan de la memoria) para que otros procesos puedan usar la CPU. Surge entonces la cuestión sobre qué hacer cuando un proceso vuelve a traerse a la memoria. Técnicamente, no hay que hacer nada. El proceso simplemente provocará faltas de página hasta que se cargue su conjunto de trabajo. El problema es que tener 20, 100 o hasta 1000 faltas de página cada vez que se carga un proceso hace que la ejecución resulte muy lenta, desperdiando también mucho tiempo de CPU, ya que el sistema operativo gasta varios milisegundos de tiempo de CPU en procesar una falta de página.

Por ese motivo, muchos sistemas de paginación tratan de seguir la pista de cuál es el conjunto de trabajo de cada proceso, asegurándose de tenerlo en la memoria antes de permitir que se ejecute el proceso. Este enfoque se denomina el **modelo del conjunto de trabajo** (Denning, 1970), y está diseñado para reducir considerablemente la tasa de faltas de página. La carga de las páginas *antes* de permitir que los procesos se ejecuten se denomina también **prepaginación**. No hay que olvidar que el conjunto de trabajo varía con el tiempo.

Desde hace mucho tiempo se sabe que la mayoría de los programas no hacen referencia a sus espacios de direccionamiento de manera uniforme, sino que las referencias tienden a concentrarse sobre un pequeño número de páginas. Una referencia a memoria puede deberse a la extracción de una instrucción, a la lectura de un dato o a la escritura de un resultado. En cualquier instante de tiempo t existe un conjunto formado por todas las páginas utilizadas por las k últimas referencias a memoria. Ese conjunto, $w(k, t)$, es el conjunto de trabajo. Puesto que las $k + 1$ últimas referencias deben haber utilizado todas las páginas que usaron las k últimas referencias, y posiblemente otra, $w(k, t)$ es una función monótona no decreciente de k (en el

sentido $w(k, t) \subseteq w(k+1, t)$). El límite de $w(k, t)$ a medida que k aumenta es finito porque un programa no puede hacer referencia a más páginas de las que contiene su espacio de direcciones, y pocos programas utilizan todas y cada una de sus páginas. La Figura 4-20 muestra cómo varía el tamaño del conjunto de trabajo en función de k .

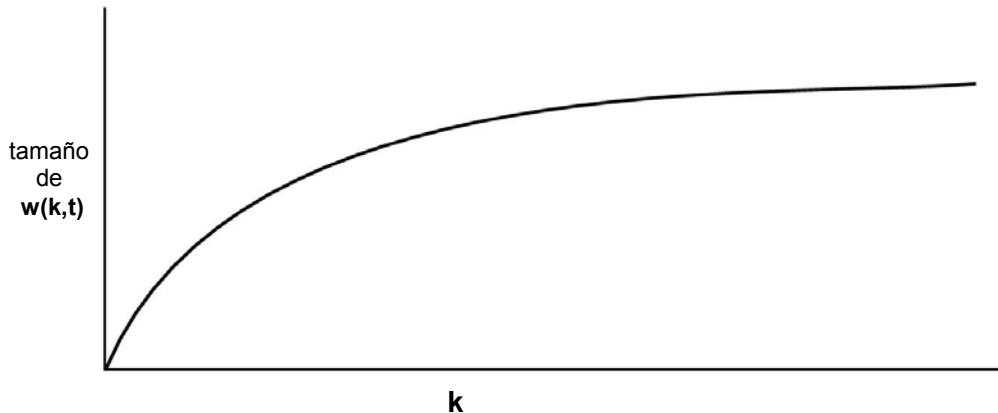


Figura 4-20. El conjunto de trabajo es el conjunto de las páginas que se han utilizado en las k últimas referencias a la memoria. La función $w(k, t)$ representa el conjunto de trabajo en el instante t .

El hecho de que la mayoría de los programas acceda de manera aleatoria a un número reducido de páginas y que ese conjunto de páginas varíe lentamente con el tiempo, explica la rápida subida inicial de la curva y la lentitud con la que sube después, cuando k es grande. Por ejemplo, un programa que está ejecutando un bucle que ocupa dos páginas y utiliza datos que ocupan cuatro páginas podría hacer referencia a las seis páginas cada 1000 instrucciones, pero la última referencia a alguna otra página podría haberse efectuado un millón de instrucciones atrás, durante la fase de inicialización. Debido a este comportamiento asintótico, el contenido del conjunto de trabajo no es muy sensible al valor de k escogido. Dicho de otro modo, hay un amplio rango de valores de k para los cuales no varía el conjunto de trabajo. Dado que el conjunto de trabajo varía lentamente con el tiempo, es posible hacer una previsión razonable sobre qué páginas se necesitarán cuando el programa se reinicie, sobre la base de su conjunto de trabajo cuando se suspendió por última vez. La prepaginación consiste en cargar estas páginas antes de permitir que el proceso reanude su ejecución.

Para implementar el modelo del conjunto de trabajo es necesario que el sistema operativo siga la pista de qué páginas están en él. Teniendo esta información obtenemos inmediatamente además un posible algoritmo de sustitución de páginas: cuando tiene lugar una falta de página, sustituir cualquier página que no esté en el conjunto de trabajo. Para implementarlo necesitamos una forma precisa de determinar qué páginas están en el conjunto de trabajo y cuáles no, en cualquier instante dado.

Como mencionamos antes, el conjunto de trabajo es el conjunto de páginas que se referenciaron en los k últimos accesos a memoria. Para implementar cualquier algoritmo de conjunto de trabajo, debe escogerse por adelantado el valor de k . Una vez seleccionado un valor, el conjunto de páginas referenciadas en los k últimos accesos a la memoria quedará determinado de manera única después de cada referencia a la memoria.

Desde luego, tener una definición operativa del conjunto de trabajo no implica que haya una forma eficiente de calcularlo en tiempo real, durante la ejecución del programa. Imaginemos un registro de desplazamiento de longitud k . Cada referencia a la memoria lo desplaza una posición a la izquierda e inserta a la derecha el número de la página a la que se

hizo referencia más recientemente. El conjunto correspondiente a los k números de página contenidos en el registro de desplazamiento sería el conjunto de trabajo. En teoría, al producirse una falta de página, el contenido del registro de desplazamiento podría leerse y ordenarse. Después de eliminar las páginas repetidas, el resultado sería el conjunto de trabajo. Sin embargo, mantener el registro de desplazamiento y procesarlo en cada falta de página tendría un coste prohibitivo, por lo que esta técnica nunca se usa.

En vez de eso, se emplean diversas aproximaciones. Una muy común consiste en desechar la idea de contar hacia atrás k referencias a la memoria y utilizar en su lugar el tiempo de ejecución. Por ejemplo, en lugar de definir el conjunto de trabajo como las páginas que se usaron durante los 10 millones de referencias a la memoria anteriores, podemos definirlo como el conjunto de páginas utilizadas durante los últimos 100 milisegundos de tiempo de ejecución. En la práctica, tal definición es tan satisfactoria como la otra y mucho más fácil de usar. Hay que señalar que cada proceso cuenta con su propio tiempo de ejecución. Por tanto, si un proceso comienza a ejecutarse en el instante T y ha tenido 40 milisegundos de tiempo de CPU en el instante real $T + 100$ milisegundos, para el cálculo del conjunto de trabajo su tiempo será de 40 milisegundos. El tiempo de CPU que ha consumido en realidad un proceso desde que se inició se denomina a menudo su **tiempo virtual actual**. Con esta aproximación, el conjunto de trabajo de un proceso es el conjunto de páginas a las que ha hecho referencia durante los últimos τ segundos de tiempo virtual.

Examinemos ahora un algoritmo de sustitución de páginas basado en el conjunto de trabajo. La idea básica es encontrar una página que no esté en el conjunto de trabajo para sustituirla. En la Figura 4-21 vemos una porción de la tabla de páginas de una cierta máquina. Dado que sólo las páginas que están en la memoria se consideran candidatas para su sustitución, el algoritmo debe ignorar las páginas ausentes de la memoria. Cada entrada contiene (por lo menos) dos elementos de información: el tiempo aproximado en que fue utilizada la página por última vez y el bit R (*Referenciada*). El rectángulo blanco vacío representa los demás campos que no se necesitan para aplicar este algoritmo, como el número de marco de página, los bits de protección y el bit M (*Modificada*).

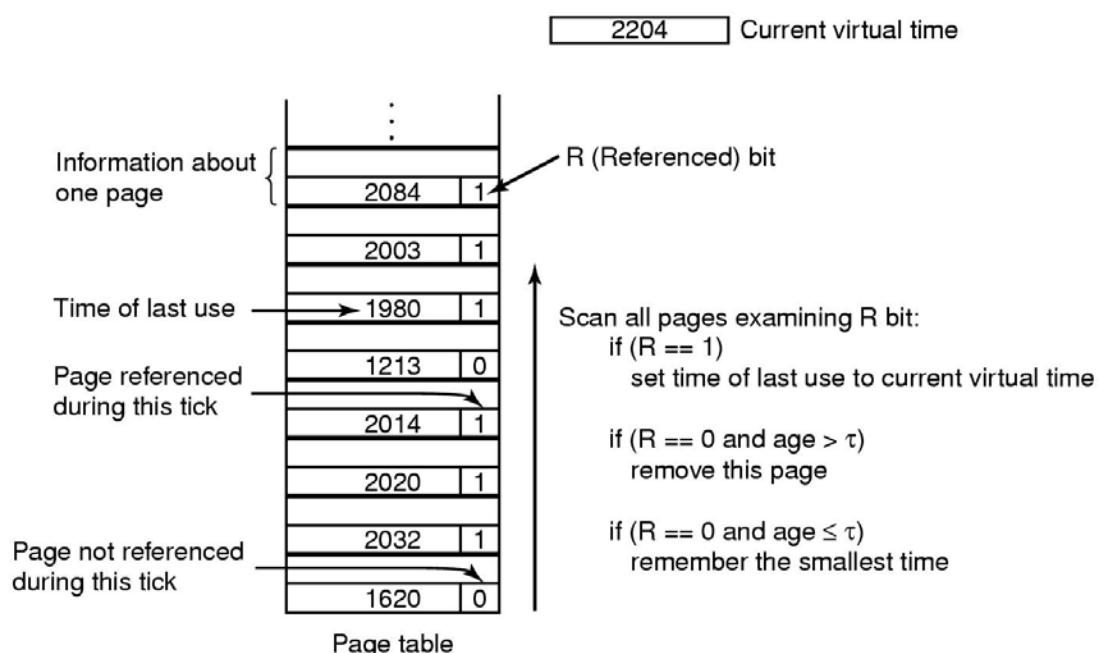


Figura 4-21. El algoritmo de sustitución del conjunto de trabajo.

El algoritmo funciona como sigue. Se supone que el hardware se encarga de activar los bits R y M , como mencionamos anteriormente. También se supone que una interrupción de reloj periódica ejecuta el código necesario para desactivar el bit de *Referenciada* en cada tic de reloj. En cada falta de página, la tabla de páginas se explora en busca de una página apropiada para sustituir.

Conforme se procesa cada entrada, se examina el bit R . Si es 1, se escribe el tiempo virtual actual en el campo *Tiempo del último uso* en la tabla de páginas, para indicar que la página se había utilizado cuando se produjo la falta de página. Puesto que se hizo referencia a la página durante el tic de reloj actual, es evidente que pertenece al conjunto de trabajo y no es candidata para su sustitución (se supone que τ abarca varios tics de reloj).

Si R es 0, quiere decir que no se ha hecho referencia a la página durante el tic de reloj actual y podría ser candidata para su sustitución. Para ver si se la debe desalojar o no, se calcula su edad, es decir, el tiempo virtual actual menos su *Tiempo del último uso* y se compara con τ . Si la edad es mayor que τ quiere decir que la página ya no está en el conjunto de trabajo, así que se sustituye, cargándose la nueva página en el marco que ella estaba ocupando. No obstante, la exploración termina de actualizar las entradas restantes de la tabla de páginas.

Por otra parte, si R es 0 pero la edad es menor o igual que τ quiere decir que la página todavía pertenece al conjunto de trabajo. Se le perdona la vida por el momento, pero se toma nota de qué página tiene mayor edad (valor más pequeño de *Tiempo del último uso*). Si se explora toda la tabla de páginas sin encontrar una candidata para ser sustituida, significa que todas las páginas están en el conjunto de trabajo. En tal caso, si se encontró una o más páginas con $R = 0$, se sustituye la de mayor edad. En el peor caso, todas las páginas se habrán referenciado durante el tic de reloj actual (y, por lo tanto, todas tienen $R = 1$), así que se escoge una al azar para sustituirla, preferiblemente una que esté limpia.

4.4.9 El Algoritmo de Sustitución de Páginas WSClock

El algoritmo del conjunto de trabajo básico es engorroso porque cada vez que se produce una falta de página explora toda la tabla de páginas hasta encontrar una candidata apropiada. **WSClock** (Carr y Hennesy, 1981) es un algoritmo mejorado que se basa en el algoritmo del reloj pero que usa también la información del conjunto de trabajo. Se utiliza mucho en la práctica por su sencillez de implementación y buen rendimiento.

La estructura de datos en que se apoya WSClock es una lista circular de marcos de página (como en el algoritmo del reloj) la cual se muestra en la Figura 4-22(a). Inicialmente, la lista está vacía. Cuando se carga la primera página, ésta se añade a la lista. A medida que se accede a nuevas páginas, éstas se incorporan a la lista formando un anillo. Cada entrada contiene el campo *Tiempo del último uso* del algoritmo del conjunto de trabajo básico, junto con el bit *R* (que se muestra) y el bit *M* (que no se muestra).

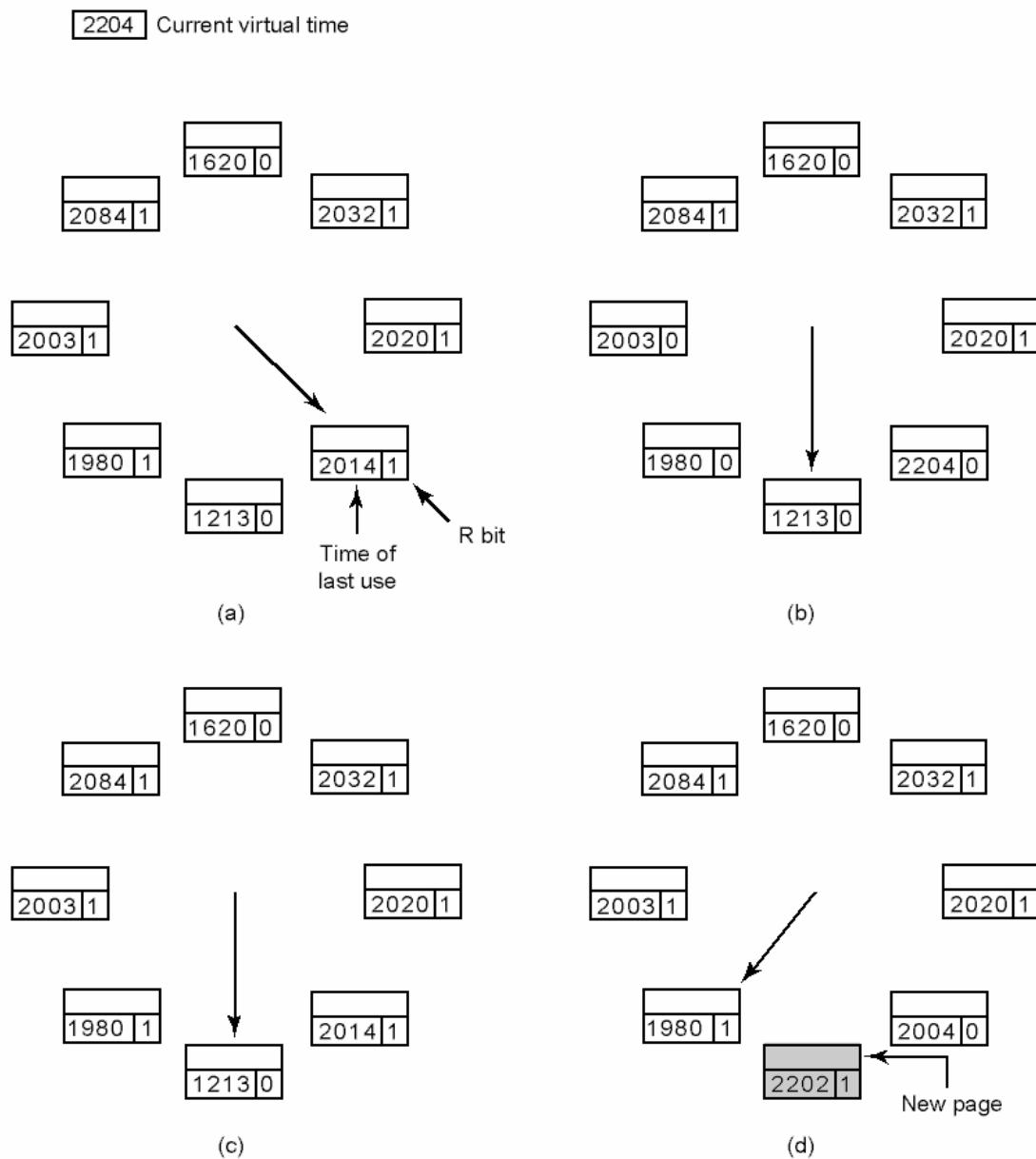


Figura 4-22. Funcionamiento del algoritmo WSClock. (a) y (b) dan un ejemplo de lo que sucede cuando $R = 1$. (c) y (d) dan un ejemplo de $R = 0$.

Al igual que en el algoritmo del reloj, cada vez que se produce una falta de página se examina primero la página a la que apunta la manecilla. Si el bit R es 1, quiere decir que la página se referenció durante el tic actual, así que no es una candidata ideal para sustituir. Por lo tanto, bit R se pone a 0, se adelanta la manecilla a la siguiente página y se repite el algoritmo con ella. El estado después de esta serie de sucesos se muestra en la Figura 4-22(b).

Consideremos ahora qué sucede si la página a la que apunta la manecilla tiene $R = 0$, como en la Figura 4-22(c). Si su edad es mayor que τ y la página está limpia, quiere decir que no está en el conjunto de trabajo y que ya hay una copia válida en el disco. La nueva página simplemente se coloca en ese marco de página, como se muestra en la Figura 4-22(d). Por otra parte, si la página está sucia, no podrá sustituirse de inmediato porque no hay una copia válida en el disco. Para evitar un cambio de proceso, se planificará su escritura en el disco, pero la manecilla se adelanta y el algoritmo continúa con la siguiente página. Después de todo, podría haber una página limpia vieja más adelante que se podría utilizar inmediatamente.

En principio, todas las páginas podrían estar planificadas para E/S de disco en una vuelta completa al reloj. Para reducir el tráfico de disco, podría fijarse un límite, y sólo permitir que se planifiquen para su escritura en el disco n páginas como máximo. Una vez alcanzado ese límite, no se planificarían nuevas escrituras.

¿Qué sucede si la manecilla da toda la vuelta y regresa a su punto de partida? Debemos distinguir dos casos:

1. Se planificó al menos una escritura.
2. No se planificó ninguna escritura.

En el primer caso, la manecilla tan solo se sigue adelantando, en busca de una página limpia. Puesto que se han planificado una o más escrituras, tarde o temprano terminará alguna, marcándose su página como limpia. La primera página limpia que se encuentre será la que se sustituya. Esta página no es necesariamente la correspondiente a la primera escritura planificada porque el controlador del disco podría reordenar las escrituras para optimizar el rendimiento del disco.

En el segundo caso, todas las páginas están en el conjunto de trabajo, pues de lo contrario se habría planificado por lo menos una escritura. A falta de información adicional, lo más sencillo es desalojar cualquier página limpia y usar su marco. Podría haberse tomado nota de la ubicación de una página limpia durante el movimiento de la manecilla. Si no hay páginas limpias, se escoge como víctima la página actual, se escribe en el disco y se desaloja.

4.4.10 Resumen de los algoritmos de sustitución de páginas

Hemos examinado diversos algoritmos de sustitución de páginas. En esta sección vamos a resumirlos brevemente. En la Figura 4-23 se listan los algoritmos que se han descrito.

| Algorithm | Comment |
|----------------------------|--|
| Optimal | Not implementable, but useful as a benchmark |
| NRU (Not Recently Used) | Very crude |
| FIFO (First-In, First-Out) | Might throw out important pages |
| Second chance | Big improvement over FIFO |
| Clock | Realistic |
| LRU (Least Recently Used) | Excellent, but difficult to implement exactly |
| NFU (Not Frequently Used) | Fairly crude approximation to LRU |
| Aging | Efficient algorithm that approximates LRU well |
| Working set | Somewhat expensive to implement |
| WSClock | Good efficient algorithm |

Figura 4-23. Algoritmos de sustitución de páginas descritos en el texto.

El algoritmo óptimo reemplaza la página que tardará más tiempo en volver a ser referenciada. Desafortunadamente, no hay forma de determinar qué página será la que tarde más tiempo en referenciarse, por lo que este algoritmo no puede utilizarse en la práctica. Sin embargo, es útil como base de comparación con otros algoritmos en un banco de pruebas.

El algoritmo NRU divide las páginas en cuatro clases según el estado de los bits R y M . Se escoge una página al azar de la clase de número más bajo. Este algoritmo es fácil de implementar, pero es muy burdo. Los hay mejores.

El algoritmo FIFO recuerda el orden en el que se cargaron las páginas en la memoria, utilizando una lista enlazada. Resulta trivial eliminar la página más antigua, pero es posible que esa página todavía esté en uso, por lo que FIFO es una mala elección.

El algoritmo de la segunda oportunidad es una modificación del FIFO que comprueba si una página se está usando o no, antes de sustituirla. Si se está usando, se le perdona la vida. Esta modificación mejora el rendimiento de forma considerable. El algoritmo del reloj es simplemente una implementación diferente del algoritmo de la segunda oportunidad. Tiene las mismas propiedades de rendimiento, pero la ejecución del algoritmo es más rápida.

LRU es un algoritmo excelente, pero no puede implementarse sin un hardware especial. Si no se cuenta con ese hardware, LRU no puede usarse. NFU es un intento burdo de aproximarse al LRU. No es muy bueno. En cambio, el algoritmo de envejecimiento es una aproximación mucho mejor al LRU y puede implementarse eficientemente. Es una buena elección.

Los dos últimos algoritmos se basan en el conjunto de trabajo. El algoritmo del conjunto del trabajo tiene un rendimiento razonable, pero su implementación es algo costosa. El algoritmo WSClock es una variante que no solo tiene un buen rendimiento, sino que también puede implementarse eficientemente.

En síntesis, los dos mejores algoritmos son el de envejecimiento y WSClock. Se basan en el algoritmo LRU y en el conjunto de trabajo, respectivamente. Ambos logran un buen rendimiento de paginación y pueden implementarse eficientemente. Existen unos cuantos algoritmos más, pero estos dos son probablemente los más importantes en la práctica.

4.5 MODELIZACIÓN DE LOS ALGORITMOS DE SUSTITUCIÓN

A través de los años, se ha realizado algún trabajo sobre la modelización de los algoritmos de sustitución de páginas desde un punto de vista teórico. En esta sección analizaremos algunas de estas ideas sólo para ver cómo funciona el proceso de modelización.

4.5.1 Anomalía de Belady

Intuitivamente, podríamos pensar que cuanto más marcos de página tenga la memoria, menos faltas de página experimentará un programa. Aunque parezca bastante sorprendente, no siempre sucede así. Belady y otros (1969) descubrieron un contraejemplo en el que FIFO provocaba más faltas de página con cuatro marcos de página que con tres. Esta extraña situación se conoce como la **anomalía de Belady**, y se ilustra en la Figura 4-24 para un programa con cinco páginas virtuales, numeradas del 0 al 4. Las páginas se referencian en el orden

0 1 2 3 0 1 4 0 1 2 3 4

En la Figura 4-24(a) vemos como con tres marcos de página se generan un total de nueve faltas de página. En la Figura 4-24(b) obtenemos diez faltas de página con cuatro marcos de página.

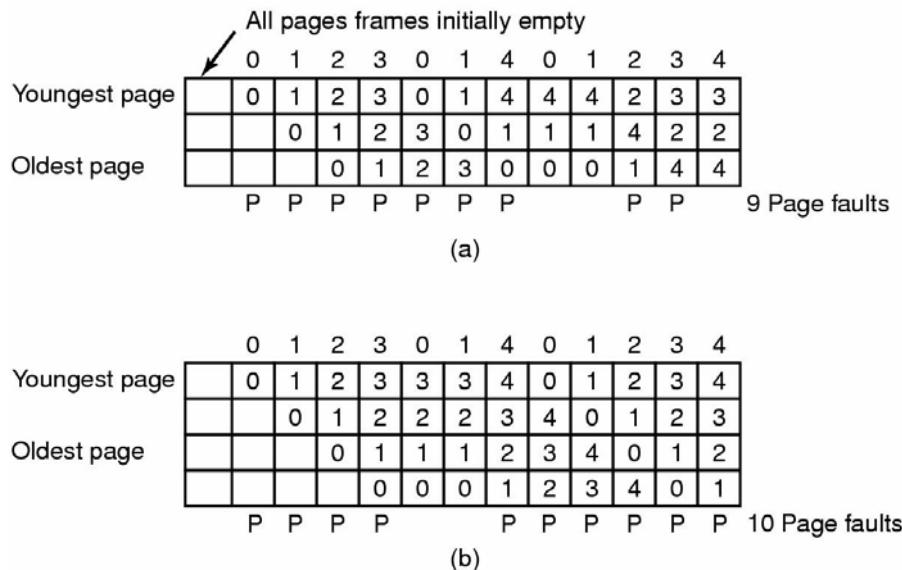


Figura 4-24. Anomalía de Belady. (a) FIFO con tres marcos de página. (b) FIFO con cuatro marcos de página. Las Ps indican las referencias que provocan faltas de página.

4.5.2 Algoritmos de Pila

Muchos investigadores en ciencias de la computación quedaron intrigados ante la anomalía de Belady y comenzaron a investigarla. Estos trabajos condujeron al desarrollo de toda una teoría sobre los algoritmos de sustitución de páginas y sus propiedades. Aunque en su mayor parte tales trabajos están fuera del alcance del presente libro, daremos a continuación una breve introducción. Si se desean más detalles consultese Maekawa y otros (1987).

Todos esos trabajos parten de la observación de que un proceso genera una serie de referencias a la memoria a medida que se ejecuta. Cada referencia a la memoria corresponde a una página virtual específica. Por tanto, en lo conceptual, el acceso de un proceso a la memoria puede representarse mediante una lista (ordenada) de números de página. Esta lista se denomina la **serie de referencias** y juega un papel central en la teoría. Por simplicidad, en el resto de esta sección consideraremos únicamente el caso de una máquina con un único proceso, de modo que en cada máquina habrá una única serie de referencias determinista (si hubiera múltiples procesos, tendríamos que tomar en cuenta la intercalación de sus series de referencia, debido a la multiprogramación).

Un sistema de paginación puede caracterizarse por tres elementos:

1. La serie de referencias del proceso en ejecución.
2. El algoritmo de sustitución de páginas.
3. El número, m , de marcos de página con que cuenta la memoria.

Conceptualmente, podemos imaginar un intérprete abstracto que funciona como sigue. Se mantiene un array interno, M , que sigue la pista del estado de la memoria. M tienen tantas entradas como páginas virtuales tiene el proceso, número que llamaremos n . Además, dicho array se divide en dos partes. La parte superior, con m entradas, contiene todas las páginas que están actualmente en la memoria. La parte inferior, con $n - m$ entradas, contiene todas las páginas a las que se ha hecho referencia alguna vez pero que se han intercambiado a disco y ya no están en la memoria. En un principio, M es el conjunto vacío, ya que no se ha hecho referencia a ninguna página, y no hay páginas en la memoria.

Al comenzar su ejecución, el proceso empieza a referenciar las páginas de la serie de referencias, una por una. Para cada referencia, el intérprete comprueba si la página ya está en la memoria (es decir, en la parte superior de M). Si no está, hay una falta de página. Si hay un marco vacío en la memoria (es decir, si la parte superior de M contiene menos de m números de página), la página se carga y se inserta en la parte superior de M , desplazando un lugar hacia abajo el contenido de M . Esta situación sólo se presenta al principio de la ejecución. Si la memoria se llena (es decir, si la parte superior de M contiene m entradas), se invoca el algoritmo de sustitución de páginas para que desaloje una página de la memoria. En el modelo, lo que sucede es que una página se pasa de la parte superior de M a la parte inferior, y el número de la página requerida se anota en la parte superior. Además, la parte superior y la inferior podrían reordenarse por separado.

Para dejar más claro el funcionamiento del intérprete, examinaremos un ejemplo concreto utilizando el algoritmo de sustitución de páginas LRU. En este ejemplo supondremos que el espacio de direcciones virtual tiene 8 páginas y que la memoria física tiene 4 marcos de página. En la parte superior de la Figura 4-25 tenemos una serie de referencias que consiste en las 24 páginas:

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

Debajo de la serie de referencias tenemos 25 columnas de 8 elementos cada una. La primera columna, que está vacía, refleja el estado de M antes de iniciarse la ejecución. Cada columna sucesiva muestra M después de procesarse la siguiente referencia de la serie mediante la aplicación del algoritmo de sustitución de páginas utilizado por el sistema. El contorno grueso denota la parte superior de M , es decir, las primeras cuatro entradas, que corresponden a marcos de página en la memoria. Las páginas que figuran dentro del rectángulo grueso están en la memoria, y las que están debajo se han intercambiado al disco.

| Reference string | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 3 | 3 | 5 | 5 | 3 | 1 | 1 | 1 | 7 | 1 | 3 | 4 | 1 | |
|------------------|----------|----------|----------|----------|----------|----------|----------|----------|---|----------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 7 | 4 | 7 | 7 | 3 | 3 | 5 | 3 | 3 | 3 | 3 | 1 | 7 | 1 | 3 | 4 | |
| | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 3 | 3 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 3 | 3 | 7 | 1 | 3 | | | |
| | 0 | 2 | 1 | 3 | 5 | 4 | 6 | 6 | 6 | 6 | 4 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 7 | 7 | | | | |
| | 0 | 2 | 1 | 1 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 5 | | | |
| | 0 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | 6 | | | |
| | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| Page faults | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | P | | |
| Distance string | ∞ | 4 | ∞ | 4 | 2 | 3 | 1 | 5 | 1 | 2 | 6 | 1 | 1 | 4 | 2 | 3 | 5 | 3 |

Figura 4-25. Estado del array M , después de procesarse cada elemento de la serie de referencias. La distancia en la serie de referencias se explicará en la siguiente sección.

La primera página de la serie de referencias es 0, así que se introduce en la parte superior de la memoria, como se muestra en la segunda columna. La segunda página es la 2, así que se introduce en lo alto de la tercera columna. Esta acción provoca que 0 baje. En este ejemplo, una página nuevamente cargada siempre se introduce en lo alto, y todo lo demás se desplaza hacia abajo, según sea necesario.

Cada una de las siete primeras páginas de la serie de referencias provoca una falta de página. Las primeras cuatro pueden tratarse sin sustituir ninguna página, pero a partir de la referencia a la página 5, la carga de una nueva página requiere sustituir una página antigua.

La segunda referencia a la página 3 no provoca una falta de página, porque esa página ya está en la memoria. No obstante, el intérprete la quita de donde estaba y la coloca en lo alto, como se muestra. El proceso continúa durante un tiempo, hasta que se hace referencia a la página 5. Esta página se pasa de la parte inferior de M a la superior (es decir, se carga en memoria procedente del disco). Cada vez que se hace referencia a una página que no está dentro del rectángulo con borde grueso, tiene lugar una falta de página, como se indica con las P s de debajo de la matriz.

Resumamos ahora algunas de las propiedades de este modelo. Primera, cuando se hace referencia a una página, siempre se la coloca en lo alto de M . Segunda, si la página solicitada ya estaba en M , todas las páginas que estaban por encima de ella bajan una posición. Una transición desde el interior del rectángulo grueso hacia fuera de él corresponde al desalojo de una página de la memoria. Tercera, las páginas que estaban debajo de la página referenciada no se mueven. De esta manera, los contenidos sucesivos de M representan exactamente el comportamiento del algoritmo LRU.

Aunque en este ejemplo utilizamos LRU, el modelo funciona igualmente bien con otros algoritmos de sustitución de páginas. En particular, hay una clase de algoritmos que es especialmente interesante: los algoritmos que tienen la propiedad:

$$M(m, r) \subseteq M(m + 1, r)$$

que debe cumplirse para cualquier número de marcos m y cualquier índice r de la serie de referencias. Esto significa que el conjunto de páginas incluido en la parte superior de M para una memoria con m marcos de página después de r referencias a la memoria está también incluido en M para una memoria con $m + 1$ marcos de página. En otras palabras, si aumentamos el tamaño de la memoria en un marco de página y volvemos a ejecutar el proceso, en todos los

puntos durante la ejecución todas las páginas que estaban presentes en la primera ocasión también lo estarán en la segunda, junto con una página adicional.

Si examinamos la Figura 4-25 y pensamos un poco en cómo funciona, debe quedar claro que LRU tiene esta propiedad. Algunos otros algoritmos (por ejemplo, el algoritmo de sustitución de páginas óptimo) también la tienen, pero FIFO no. Llamamos **algoritmos de pila** a los que tienen esta propiedad. Estos algoritmos no sufren la anomalía de Belady y por ese motivo son muy del agrado de los teóricos de la memoria virtual.

4.5.2 La Serie de Distancias Asociada a la Serie de Referencias

En el caso de los algoritmos de pila, resulta conveniente representar la serie de referencias de una forma más abstracta que con los números de página reales. En adelante, denotaremos una referencia a una página por la distancia desde lo alto de la pila hasta la entrada donde se colocó la página. Por ejemplo, la referencia a la página 1 en la última columna de la Figura 4-25 es una referencia a una página que está a una distancia 3 de lo alto de la pila (porque la página 1 estaba en tercer lugar *antes* de la referencia). Diremos que las páginas que todavía no se han referenciado y por lo tanto todavía no están en la pila (es decir, no están en M) están a una distancia ∞ . La **serie de distancias** para la Figura 4-25 se indica en la parte baja de la figura.

Hay que darse cuenta de que la distancia en la serie de referencias no sólo depende de la serie de referencias, sino también del algoritmo de sustitución. Con la misma serie de referencias original, un algoritmo de sustitución distinto tomaría diferentes decisiones respecto a qué páginas desalojar. Como resultado, se produce una sucesión de pilas diferente.



Las propiedades estadísticas de la serie de distancias tienen un gran impacto sobre el rendimiento del algoritmo de sustitución. En la Figura 4-26(a) vemos la función de densidad de probabilidad para los valores, d , de una serie de distancias (ficticia). Casi todos los elementos de la serie de referencias tienen una distancia entre 1 y k . Con una memoria de k marcos de página, habrá pocas faltas de página.

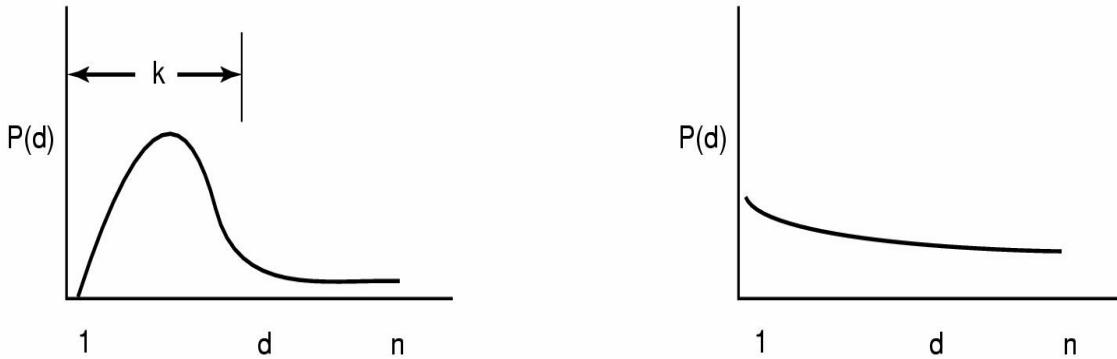


Figura 4-26. Funciones de densidad de probabilidad para dos distancias en series de referencias hipotéticas.

En contraste, en la Figura 4-26(b), las distancias de las referencias están tan dispersas que la única manera evitar un gran número de faltas de página es dar al programa tantos marcos de página como páginas virtuales tenga. Dar con un programa como este es auténtica mala suerte.

4.5.3 Predicción de la Tasa de Faltas de Página

Una de las propiedades agradables de la serie de distancias es que puede servir para predecir el número de faltas de página que se producirán con memorias de diferentes tamaños. Vamos a demostrar cómo puede realizarse ese cálculo basándonos en el ejemplo de la Figura 4-25. El objetivo es realizar una pasada por serie de distancias y, a partir de la información obtenida, predecir el número de faltas de página que tendría el proceso en memorias con 1, 2, 3, ..., n marcos de página, donde n es el número de páginas virtuales del espacio de direcciones del proceso.

El algoritmo comienza explorando la serie de distancias, página por página, y llevando la cuenta de las veces que aparece la distancia 1, las veces que aparece la distancia 2, etc. C_i es el número de veces que aparece la distancia i . En la Figura 4-27(a) se ilustra el vector C para las distancias en la serie de referencias de la Figura 4-25. En este ejemplo, sucede 4 veces que la página referenciada ya estaba en lo alto de la pila. En 3 ocasiones se referenció la página que estaba una posición más abajo, y así de forma sucesiva. C_∞ es el número de veces que aparece ∞ como distancia en la serie de referencias.

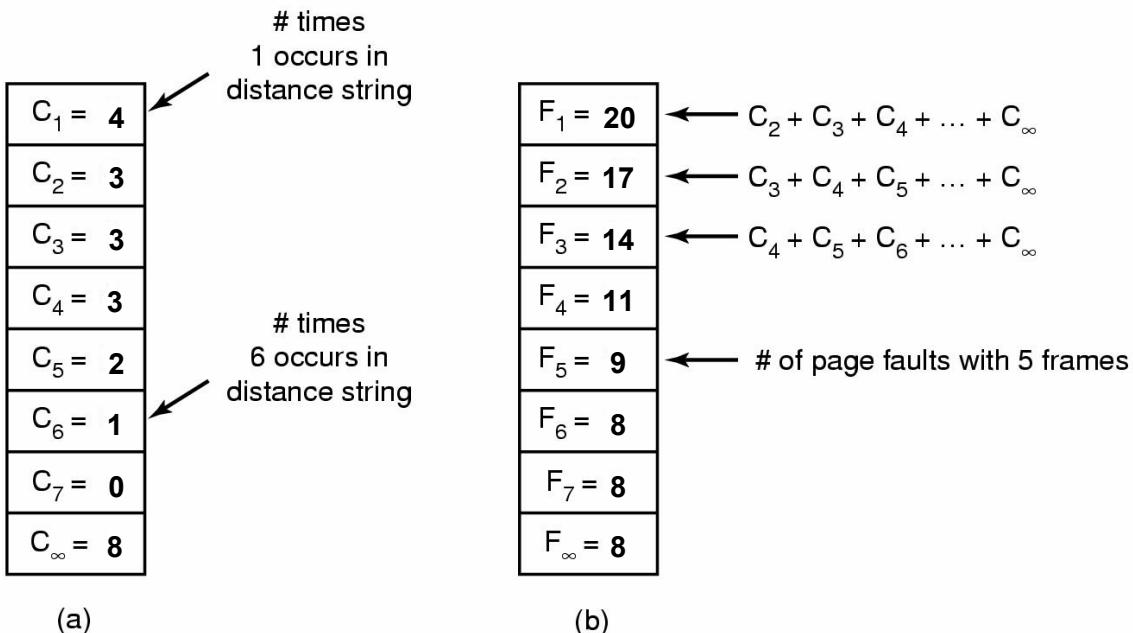


Figura 4-27. Cálculo de la tasa de faltas de página a partir de la serie de distancias. (a) El vector C . (b) El vector F .

Calculamos ahora el vector F de acuerdo a la fórmula:

$$F_m = \sum_{k=m+1}^n C_k + C_\infty$$

El valor de F_m es el número de faltas de página que se producirían con la serie de referencias dada y m marcos de página. Con la serie de referencias de la Figura 4-25, el vector F es el que se muestra en la Figura 4-27(b). Por ejemplo, F_1 es 20, lo que significa que, con una memoria de un solo marco de página, de las 24 referencias de la cadena, todas provocarían faltas de página excepto las cuatro que se hacen justo a la página que se acaba de referenciar anteriormente.

Para entender por qué funciona esta fórmula, regresemos al rectángulo grueso de la Figura 4-25. m es el número de marcos de página, que aparecen en la parte superior de M . Una falta de página se produce cada vez que la distancia en la serie de referencias es $m + 1$ o más. El sumatorio de la fórmula anterior calcula el total de las veces que aparecen tales distancias. Este modelo puede utilizarse también para hacer otras predicciones (Maekawa y otros, 1987).

4.6 CUESTIONES DE DISEÑO PARA SISTEMAS PAGINADOS

En las secciones anteriores hemos explicado cómo funciona la paginación y hemos descrito algunos de los algoritmos de sustitución de páginas básicos, mostrando finalmente cómo modelizarlos. Sin embargo el conocimiento tan sólo de la mecánica no es suficiente. En el diseño de un sistema, se necesita algo más para conseguir que el sistema funcione bien. Es como la diferencia entre saber cómo mover la torre, el caballo, el alfil y las demás piezas del ajedrez, y ser un buen jugador. En las siguientes secciones examinaremos otros aspectos que los diseñadores de los sistemas operativos deben considerar con detenimiento para obtener un buen rendimiento de un sistema de paginación.

4.6.1 Políticas de Asignación Local y Global

En las secciones anteriores hemos estudiado varios algoritmos para elegir la página que será reemplazada cuando se produzca una falta de página. La cuestión más importante asociada con esta elección (que hasta ahora nos hemos cuidado de esconder bajo la alfombra) es cómo debe repartirse la memoria entre todos los procesos ejecutables que compiten por ella.

Echemos un vistazo a la Figura 4-28(a). En ella, tres procesos *A*, *B* y *C* constituyen el conjunto de procesos ejecutables. Supongamos que *A* provoca una falta de página. ¿El algoritmo de sustitución de páginas debe tratar de encontrar la página menos recientemente utilizada considerando sólo las seis páginas que *A* tiene asignadas en la actualidad, o debe considerar todas las páginas que están en la memoria? Si sólo se consideran las páginas de *A*, la de menor edad es *A*5, y llegaremos a la situación de la Figura 4-28(b).

Por otra parte, si se sustituye la página de menor edad sin considerar a qué proceso pertenece, se escogerá la página *B*3 y tendremos la situación de la Figura 4-28(c). El algoritmo de la Figura 4-28(b) se dice que es un algoritmo de sustitución de páginas **local**, mientras que el de la Figura 4-28(c) se dice que es un algoritmo de sustitución de páginas **global**. Los algoritmos locales corresponden de hecho a asignar a cada proceso una fracción fija de la memoria. Los algoritmos globales asignan dinámicamente los marcos de páginas entre los procesos ejecutables. Así, el número de marcos de página asignados a cada proceso varía con el tiempo.

| | Age |
|----|-----|
| A0 | 10 |
| A1 | 7 |
| A2 | 5 |
| A3 | 4 |
| A4 | 6 |
| A5 | 3 |
| B0 | 9 |
| B1 | 4 |
| B2 | 6 |
| B3 | 2 |
| B4 | 5 |
| B5 | 6 |
| B6 | 12 |
| C1 | 3 |
| C2 | 5 |
| C3 | 6 |

| | Age |
|----|-----|
| A0 | |
| A1 | |
| A2 | |
| A3 | |
| A4 | |
| A5 | |
| B0 | |
| B1 | |
| B2 | |
| B3 | |
| B4 | |
| B5 | |
| B6 | |
| C1 | |
| C2 | |
| C3 | |

| | Age |
|----|-----|
| A0 | |
| A1 | |
| A2 | |
| A3 | |
| A4 | |
| A5 | |
| B0 | |
| B1 | |
| B2 | |
| B3 | |
| B4 | |
| B5 | |
| B6 | |
| C1 | |
| C2 | |
| C3 | |

Figura 4-28. Sustitución de páginas local y global. (a) Configuración original.
(b) Sustitución de páginas local. (c) Sustitución de páginas global.

En general, los algoritmos globales funcionan mejor, sobre todo cuando el tamaño del conjunto de trabajo puede variar durante el tiempo de vida de los procesos. Si se utiliza un algoritmo local y el conjunto de trabajo crece, habrá trasiego, incluso si hay muchos marcos de página desocupados. Si el conjunto de trabajo se encoge, los algoritmos locales desperdician memoria. Si se utiliza un algoritmo global, el sistema debe decidir continuamente cuántos marcos de página asignará a cada proceso. Una forma de hacerlo es monitorizar el tamaño del conjunto de trabajo mediante los bits de envejecimiento (como en la Figura 4-19), pero ese enfoque no previene necesariamente el trasiego. El tamaño del conjunto de trabajo puede cambiar en microsegundos, mientras que los bits de envejecimiento son una medida burda que abarca varios tics de reloj.

Otro enfoque consiste en tener un algoritmo para asignar marcos de página a los procesos. Una posible vía es determinar periódicamente el número de procesos ejecutables y repartir entre ellos a partes iguales todos los marcos de página. Así, con 12.416 marcos de página disponibles (es decir no ocupados por el sistema operativo) y 10 procesos, cada proceso recibiría 1.241 marcos. Los 6 restantes integrarían una reserva de la que se echaría mano para afrontar futuras faltas de página.

Aunque este método parece equitativo, no tiene mucho sentido asignar el mismo número de marcos de página a un proceso de 10 KB y a otro de 300 KB. En vez de eso, los marcos podrían repartirse proporcionalmente al tamaño total de cada proceso, de modo que un proceso de 300 KB obtendría 30 veces más que uno de 10 KB. Tal vez sería prudente asignar a cada proceso una cantidad mínima para que pueda ejecutarse por pequeño que sea. Por ejemplo, en algunas máquinas, una sola instrucción con dos operandos podría requerir hasta seis páginas en memoria porque la instrucción misma, el operando de origen y el operando de destino podrían estar todos atravesando fronteras de páginas consecutivas. Con una asignación de sólo cinco páginas en memoria los programas que contuviesen tales instrucciones no podrían ejecutarse de ninguna manera.

Si se utiliza un algoritmo global, es posible iniciar cada proceso con cierto número de páginas en memoria proporcional a su tamaño, pero la asignación deberá actualizarse dinámicamente a medida que avance la ejecución de los procesos. Una forma de gestionar la asignación es utilizar el algoritmo de la **frecuencia de faltas de página** (PFF; *Page Fault Frequency*). Este algoritmo determina cuándo hay que aumentar o reducir el número de marcos asignados al proceso, pero no dice nada sobre qué página hay que sustituir concretamente si se produce una falta de página; sólo controla el tamaño de la asignación realizada.

Se sabe que para una amplia clase de algoritmos de sustitución de páginas, incluyendo al LRU, la tasa de faltas de página disminuye a medida que se asignan más páginas, como ya explicamos. Ésta es la suposición en la que se basa PFF. Esta propiedad se ilustra en la Figura 4-29.

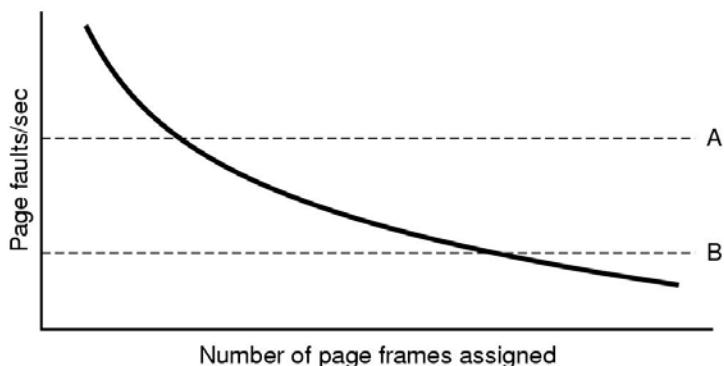


Figura 4-29. La tasa de faltas de página como una función del número de marcos de página asignados.

Medir la tasa de faltas de página es sencillo: basta con contar el número de faltas de página por segundo, calculando quizás también una media para los segundos transcurridos. Una forma fácil de hacerlo es sumar el número de segundos presente a la media de ejecución actual y dividir el resultado entre 2. La línea de guiones marcada con una *A* corresponde a una tasa de faltas de página inaceptablemente alta, por lo que se asignarían más marcos de página a ese proceso para reducir la tasa de faltas de página. La línea de guiones marcada con una *B* corresponde a una tasa de faltas de página tan baja que podría concluirse que el proceso tiene demasiada memoria. En este caso, podríamos quitarle marcos de página. Por lo tanto, PFF trata de mantener la tasa de paginación de cada proceso dentro de límites aceptables.

Es importante señalar que algunos algoritmos de sustitución de páginas pueden operar con una política de sustitución local o global. Por ejemplo, FIFO puede reemplazar la página más antigua de toda la memoria (algoritmo global) o la página más antigua del proceso actual (algoritmo local). Similarmente, LRU, o alguna de sus aproximaciones, puede sustituir la página menos recientemente utilizada de toda la memoria (algoritmo global) o la menos recientemente utilizada por el proceso actual (algoritmo local). En algunos casos la elección entre una política local o global es independiente del algoritmo.

Por otra parte, hay algoritmos de sustitución de páginas para los que sólo tiene sentido una estrategia local. En particular, los algoritmos del conjunto de trabajo y WSClock se refieren a un proceso específico y deben aplicarse en ese contexto. En realidad no existe un conjunto de trabajo para la máquina en su totalidad, y tratar de usar la unión de todos los conjuntos de trabajo podría hacer que se perdiera la propiedad de localidad y no funcionaría bien.

4.6.2 Control de Carga

Incluso con el mejor algoritmo de sustitución de páginas y una asignación global óptima de marcos de página a los procesos, puede suceder que el sistema tenga trasiego. De hecho, siempre que los conjuntos de trabajo combinados de todos los procesos exceden la capacidad de la memoria, cabe esperar trasiego. Un síntoma de esta situación es que el algoritmo PFF indica que algunos procesos necesitan más memoria, pero ningún proceso necesita menos. En ese caso, es imposible asignar más memoria a los procesos que la necesitan sin perjudicar a otros procesos. La única solución real es deshacerse temporalmente de algunos procesos.

La forma de reducir el número de procesos que compiten por la memoria es intercambiar algunos de ellos al disco y liberar todas las páginas que tenían asignadas. Por ejemplo, puede intercambiarse al disco un proceso repartiendo todos sus marcos de página entre otros procesos que sufren trasiego. Si el trasiego cesa, el sistema podrá operar durante un rato en estas condiciones. Si el trasiego no cesa, habrá que intercambiar otro proceso al disco, y así hasta que desaparezca el trasiego. Entonces, incluso con paginación, sigue siendo necesario el intercambio, sólo que ahora se utiliza para reducir la demanda potencial de memoria, en vez de para recuperar bloques de la memoria para uso inmediato.

El intercambio de procesos al disco para aliviar la carga de la memoria es una reminiscencia de la planificación a dos niveles, en la que algunos procesos se colocan en el disco y se utiliza un planificador a corto plazo para los procesos restantes. Claramente, las dos ideas pueden combinarse, intercambiando a disco tan solo los procesos suficientes para lograr que la tasa de faltas de página sea aceptable. Periódicamente, algunos procesos se cargarán desde el disco y otros se intercambiarán al disco.

Sin embargo, otro factor a considerar es el grado de multiprogramación. Como vimos en la Figura 4-4, cuando el número de procesos en memoria principal es demasiado bajo, la CPU puede estar ociosa durante períodos de tiempo considerables. Esta consideración es un argumento a favor de considerar no sólo el tamaño de los procesos y la tasa de paginación al decidir qué procesos se intercambian al disco, sino también sus características, tales como si son intensivos en CPU o en E/S, así como las características de los demás procesos.

4.6.3 Tamaño de Página

El tamaño de la página suele ser un parámetro que puede elegir el sistema operativo. Incluso si el hardware se diseñó con páginas de, por ejemplo, 512 bytes, el sistema operativo puede fácilmente ver las páginas 0 y 1, 2 y 3, 4 y 5, etcétera como páginas de 1 KB asignándoles siempre dos marcos de página de 512 bytes consecutivos.

Determinar el tamaño de página óptimo requiere equilibrar varios factores en conflicto. Como resultado, no existe un tamaño óptimo para todos los casos. Para comenzar, hay dos factores que favorecen el uso de páginas pequeñas. Un segmento de código, datos o pila cualquiera no tiene porqué llenar completamente un número entero de páginas. En promedio, la mitad de la última página estará vacía, y ese espacio adicional se desperdicia. Este desperdicio se denomina **fragmentación interna**. Si hay n segmentos en la memoria y las páginas son de p bytes, se desperdiciarán $np/2$ bytes debido a la fragmentación interna. Este razonamiento nos lleva a escoger páginas pequeñas.

El otro argumento en este sentido se hace evidente si consideramos un programa que consta de ocho fases sucesivas de 4 KB cada una. Con páginas de 32 KB, será necesario asignar al programa 32 KB todo el tiempo. Con páginas de 16 KB, solo necesitará 16 KB. Con páginas de 4 KB o menores sólo requerirá 4 KB en cualquier instante dado. En general, un tamaño de página grande provoca que más partes no utilizadas de los programas estén en la memoria.

Por otro lado, el uso de páginas pequeñas significa que los programas van a necesitar muchas páginas y, por lo tanto, una tabla de páginas más grande. Un programa de 32 KB sólo necesita cuatro páginas de 8 KB, pero 64 páginas de 512 bytes. Las transferencias entre la memoria y el disco suelen efectuarse en unidades de una página, invirtiéndose la mayor parte del tiempo en el posicionamiento del brazo y el retraso rotacional, por lo que transferir una página pequeña requiere casi el mismo tiempo que transferir una grande. Podrían necesitarse 64×10 milisegundos para cargar 64 páginas de 512 bytes, pero sólo 4×12 milisegundos para cargar cuatro páginas de 8 KB.

En algunas máquinas, la tabla de páginas debe cargarse en registros de hardware cada vez que la CPU conmuta de un proceso a otro. En tales máquinas, el tiempo requerido para cargar esos registros aumenta a medida que disminuye el tamaño de página. Además, el espacio ocupado por la tabla aumenta a medida que se reduce el tamaño de página.

Este último punto permite un análisis matemático. Supongamos que el tamaño medio de los procesos es s bytes y que el tamaño de las páginas es p bytes. Supongamos además que cada entrada de la tabla de páginas ocupa e bytes. Por lo tanto, el número aproximado de páginas que se requieren por proceso es s/p , ocupando se/p bytes de espacio para la tabla de páginas. El desperdicio de memoria en la última página del proceso, debido a la fragmentación interna es $p/2$. Por lo tanto, la sobrecarga total debida a la tabla de páginas y a la pérdida por fragmentación interna es la suma de esos dos términos:

$$\text{sobrecarga} = se/p + p/2$$

El primer término (tamaño de la tabla de páginas) es grande cuando el tamaño de página es pequeño. El segundo término (fragmentación interna) es grande cuando el tamaño de página es grande. El tamaño óptimo debe estar en algún punto intermedio. Si obtenemos la primera derivada respecto a p y la igualamos a cero, obtenemos la ecuación

$$-se/p^2 + 1/2 = 0$$

De esa ecuación podemos deducir una fórmula que da el tamaño de página óptimo (considerando sólo la memoria que se desperdicia por fragmentación y el tamaño de la tabla de páginas). El resultado es:

$$p = \sqrt{2se}$$

Con $s = 1$ MB y $e = 8$ bytes por entrada de la tabla de páginas, el tamaño de página óptimo es de 4 KB. Los ordenadores comerciales han manejado páginas desde 512 bytes hasta 64 KB. Un valor típico utilizado era 1 KB, pero actualmente son más comunes 4 KB u 8 KB. A medida que crecen las memorias, el tamaño de página tiende a aumentar (pero no de manera lineal). Aumentando cuatro veces el tamaño de la RAM casi nunca se llega a duplicar el tamaño de la página.

4.9 INVESTIGACIÓN SOBRE GESTIÓN DE MEMORIA

La gestión de memoria (especialmente los algoritmos de paginación) fue en algún momento un área fructífera de investigación, pero casi toda esa actividad parece haber cesado, al menos en el caso de los sistemas de propósito general. La mayoría de los sistemas reales utilizan alguna variación del algoritmo del reloj, ya que es fácil de implementar y es relativamente efectivo. Una excepción reciente, sin embargo, es el rediseño del sistema de memoria virtual de UNIX BSD versión 4.4 (Cranor y Parulkar, 1999).

Donde sí se siguen realizando investigaciones sobre paginación es en sistemas de propósito especial y en nuevos tipos de sistemas. Algunos de estos trabajos buscan formas de permitir que los procesos de usuario hagan el tratamiento de sus propias faltas de página y realicen su propia gestión de memoria, quizás de forma específica para la aplicación (Engler y otros, 1995). Un área en la cual las aplicaciones podrían necesitar realizar su propia paginación de manera especial es en el área de multimedia, por lo que algunas investigaciones han tratado sobre eso (Hand, 1999). Otro área que tiene algunos requerimientos especiales es la de los comunicadores personales de bolsillo (Abutaleb y Li, 1997; Wan y Lin, 1997). Un último área es la de los sistemas con espacios de direcciones de 64 bits compartidos por muchos procesos (Talluri y otros, 1995).

4.10 RESUMEN

En este capítulo hemos examinado la gestión de memoria. Vimos que los sistemas más sencillos no realizan ni intercambio ni paginación. Una vez que un programa se carga en la memoria, permanece allí hasta que termina. Algunos sistemas operativos sólo permiten mantener un proceso a la vez en la memoria, mientras que otros soportan multiprogramación.

El paso siguiente es el intercambio. Cuando se utiliza intercambio, el sistema puede contener más procesos de los que caben en la memoria. Los procesos para los que no hay espacio suficiente se intercambian al disco. Puede llevarse el control del espacio desocupado en la memoria y en el disco con un mapa de bits o una lista de huecos.

Los ordenadores modernos suelen tener alguna forma de memoria virtual. En su forma más simple, el espacio de direcciones de cada proceso se divide en bloques del mismo tamaño que se denominan páginas y que pueden cargarse en cualquier marco de página disponible en la memoria. Hay muchos algoritmos de sustitución de páginas; dos de los mejores algoritmos son el de envejecimiento y WSClock.

Los sistemas de paginación pueden modelizarse abstrayendo la serie de referencias a las páginas del programa y utilizándola con diferentes algoritmos. Estos modelos pueden servir para hacer algunas predicciones respecto al comportamiento de la paginación.

No basta con escoger un algoritmo de sustitución para lograr que los sistemas de paginación funcionen bien; hay que cuidar aspectos como la determinación del conjunto de trabajo, la política de asignación de memoria y el tamaño de las páginas.

La segmentación ayuda a manejar estructuras de datos que cambian de tamaño durante la ejecución y simplifica el enlazado y la compartición. También facilita proporcionar protección específica a diferentes segmentos. A veces se combinan la segmentación y la paginación para crear una memoria virtual bidimensional. Los sistemas MULTICS y Pentium de Intel manejan segmentación y paginación.

6

SISTEMAS DE FICHEROS

Todas las aplicaciones necesitan almacenar y recuperar información. Mientras un proceso está ejecutándose puede almacenar cierta cantidad de información dentro de su propio espacio de direcciones. Sin embargo, esa capacidad de almacenamiento está limitada por el tamaño del espacio de direcciones virtual. Para algunas aplicaciones ese tamaño es adecuado, pero para otras, tales como la reserva de billetes de avión, la banca o el registro de las operaciones realizadas por una empresa, resulta demasiado pequeño.

Un segundo problema con el que nos encontramos al guardar la información dentro del espacio de direccionamiento de un proceso es que cuando el proceso termina, la información se pierde. Para muchas aplicaciones (por ejemplo para las bases de datos) la información debe ser retenida durante semanas, meses o incluso para siempre. Es inaceptable permitir que la información se desvanezca cuando termina el proceso que la utiliza. Además, tampoco debe perderse aunque el proceso se destruya repentinamente debido a una caída del sistema.

Un tercer problema es que frecuentemente es necesario que múltiples procesos accedan a (partes de) la información al mismo tiempo. Si disponemos de una guía telefónica almacenada dentro del espacio de direccionamiento de un único proceso, sólo ese proceso va a poder acceder a ella. La manera de resolver este problema es hacer que la información sea ella misma independiente de cualquier proceso.

Entonces tenemos ya tres requerimientos esenciales para el almacenamiento a largo plazo de la información:

1. Debe poder almacenarse una cantidad de información muy grande.
2. La información debe permanecer tras la terminación del proceso que la usa.
3. Debe permitir que múltiples procesos puedan acceder a la información concurrentemente

La solución usual a todos estos problemas es almacenar la información sobre discos y otros medios externos en unidades denominadas **ficheros**. Los procesos pueden entonces leerlos y crear nuevos ficheros si es necesario. La información almacenada en los ficheros debe ser persistente, esto es, no debe verse afectada por la creación y terminación de los procesos. Un fichero sólo puede desaparecer cuando su propietario lo borre de forma explícita.

Los ficheros están gestionados por el sistema operativo. La forma en la cual están estructurados, cómo se nombran, se acceden, se utilizan, se protegen e implementan son temas principales en el diseño de los sistemas operativos. Globalmente, a esa parte del sistema operativo que trata los ficheros se la conoce como el **sistema de ficheros** y es el tema de este capítulo.

Desde el punto de vista de los usuarios, el aspecto más importante de un sistema de ficheros es su apariencia, es decir, qué constituye un fichero, como se nombran y se protegen los ficheros, qué operaciones se permiten, etc. Los detalles de si para seguir la pista de la memoria libre se utilizan listas enlazadas o mapas de bits, o el detalle de cuántos sectores hay en un bloque lógico, son cuestiones de menos interés, aunque son de gran importancia para los

diseñadores del sistema de ficheros. Por esa razón, hemos estructurado el capítulo en varias secciones. Las dos primeras secciones tienen que ver con la interfaz del usuario con los ficheros y con los directorios, respectivamente. A continuación se discutirá en detalle la forma en la cual se implementa el sistema de ficheros. Finalmente, daremos algunos ejemplos de sistemas de ficheros reales.

6.1 FICHEROS

En las páginas siguientes examinaremos los ficheros desde la perspectiva del usuario; es decir, cómo se utilizan y qué propiedades tienen.

6.1.1 Nombres de fichero

Los ficheros son un mecanismo de abstracción que permite almacenar información en el disco y leerla después. Esto debe hacerse de tal modo que el usuario no tenga que enterarse de los detalles de cómo y dónde está almacenada la información, y de cómo funcionan en realidad los discos.

Tal vez la característica más importante de cualquier mecanismo de abstracción es la forma en la que se da nombre a los objetos que se manejan, así que comenzaremos nuestro estudio de los sistemas de ficheros con el tema de los nombres de fichero. Cuando un proceso crea un fichero, le asigna un nombre. Cuando el proceso termina, el fichero sigue existiendo y otros programas pueden tener acceso a él utilizando su nombre.

Las reglas exactas para nombrar ficheros varían un tanto de un sistema a otro, pero todos los sistemas operativos actuales permiten usar cadenas de una a ocho letras como nombres de fichero válidos. Así *andrea*, *bruce* y *cathy* son posibles nombres de fichero. Es común que se permitan también dígitos y caracteres especiales, de modo que nombres como *2*, *urgent!* y *Fig.2-14* también son válidos en muchos casos. Muchos sistemas de ficheros reconocen nombres de hasta 255 caracteres de longitud.

Algunos sistemas de ficheros distinguen entre mayúsculas y minúsculas, pero otros no. UNIX pertenece a esta primera categoría; MS-DOS, a la segunda. Por tanto, en un sistema UNIX los siguientes nombres corresponden a tres ficheros distintos: *maria*, *Maria* y *MARIA*. En MS-DOS, todos esos nombres se refieren al mismo fichero.

Quizá valga la pena hacer aquí una pequeña digresión en lo tocante a los nombres de fichero. Tanto Windows 95 como Windows 98 utilizan el sistema de ficheros de MS-DOS, y por lo tanto heredaron muchas de sus propiedades, como la forma de construir nombres de fichero. Además, Windows NT y Windows 2000 reconocen el sistema de ficheros de MS-DOS y por ende también heredan sus propiedades. Sin embargo, los últimos dos sistemas también tienen un sistema de ficheros nativo (NTFS) que tiene diferentes propiedades (como nombres de fichero en Unicode). En este capítulo, cuando nos refiramos al sistema de ficheros de Windows, estaremos hablando del sistema de ficheros de MS-DOS, que es el único que reconocen todas las versiones de Windows. Trataremos el sistema de ficheros nativo de Windows 2000 en el capítulo 11.

Muchos sistemas de ficheros manejan nombres de fichero con dos partes, separadas por un punto, como en *prog.c*. La parte que sigue al punto se denomina **extensión del fichero**, y normalmente indica algo acerca del fichero. En MS-DOS, por ejemplo, los nombres de fichero tienen de uno a ocho caracteres, más una extensión opcional de uno a tres caracteres. En UNIX, el tamaño de la extensión, si la hay, se deja al criterio del usuario, y un fichero podría incluso tener dos o más extensiones, como en *prog.c.Z*, donde *.Z* se utiliza por lo común para indicar que el fichero (*prog.c*) se comprimió utilizando el algoritmo de compresión Ziv-Lempel. En la Figura 6-1 se presentan algunas de las extensiones de fichero más comunes y su significado.

| Extension | Meaning |
|-----------|---|
| file.bak | Backup file |
| file.c | C source program |
| file.gif | Compuserve Graphical Interchange Format image |
| file.hlp | Help file |
| file.html | World Wide Web HyperText Markup Language document |
| file.jpg | Still picture encoded with the JPEG standard |
| file.mp3 | Music encoded in MPEG layer 3 audio format |
| file.mpg | Movie encoded with the MPEG standard |
| file.o | Object file (compiler output, not yet linked) |
| file.pdf | Portable Document Format file |
| file.ps | PostScript file |
| file.tex | Input for the TEX formatting program |
| file.txt | General text file |
| file.zip | Compressed archive |

Figura 6-1. Algunas extensiones de fichero comunes.

En algunos sistemas (como UNIX) las extensiones de fichero son sólo un convenio y el sistema operativo no vigila que se utilicen de alguna manera específica. Un fichero llamado *fichero.txt* puede ser algún tipo de fichero de texto, pero el nombre sirve más para recordar ese hecho a su propietario que para comunicar alguna información real al ordenador. En cambio, un compilador de C podría insistir en que cualquier fichero que vaya a compilar termine en *.c*, y podría negarse a compilarlo de no ser así.

Los convenios de este tipo son útiles en especial cuando el mismo programa puede manejar varios tipos de ficheros distintos. Por ejemplo, podría suministrarse al compilador de C una lista de varios ficheros que debe compilar y enlazar, algunos de ellos en C y otros en lenguaje ensamblador. En tal caso, la extensión se vuelve indispensable para que el compilador sepa cuáles son los ficheros en C, cuáles están en lenguaje ensamblador y cuáles son de otro tipo.

En contraste, Windows tiene conocimiento de las extensiones y les asigna un significado. Los usuarios (o procesos) pueden registrar extensiones de cara al sistema operativo y especificar, para cada una, qué programa es el “dueño” de la extensión. Cuando un usuario hace doble clic sobre un nombre de fichero, se inicia el programa asociado a su extensión de fichero, con el nombre de fichero como parámetro. Por ejemplo, si se hace doble clic en *fichero.doc*, se iniciará el programa Word de Microsoft, y éste abrirá *fichero.doc* como primer documento a editar.

6.1.2 Estructura de los ficheros

Los ficheros pueden estructurarse de varias maneras. En la Figura 6-2 se ilustran tres posibilidades comunes. El fichero de la Figura 6-2(a) es una sucesión no estructurada de bytes. En efecto, el sistema operativo no sabe qué contiene el fichero, ni le interesa; lo único que ve son bytes. Cualquier significado que tenga el fichero deberán atribuírselo los programas en el nivel de usuario. Tanto UNIX como Windows utilizan este enfoque.

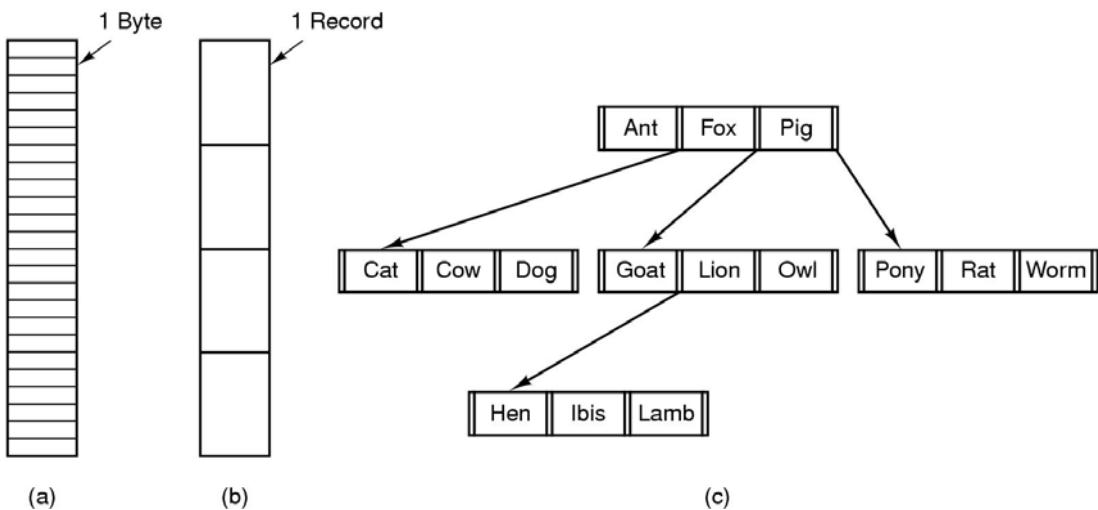


Figura 6-2. Tres tipos de ficheros. (a) Sucesión de bytes. (b) sucesión de registros. (c) Árbol.

Hacer que el sistema operativo vea los ficheros únicamente como sucesiones de bytes ofrece el máximo de flexibilidad. Los programas de usuario pueden colocar lo que deseen en sus ficheros y darles el nombre que les convenga. El sistema operativo no ayuda, pero tampoco estorba. Esto es muy importante para los usuarios que desean hacer cosas fuera de lo común.

El primer paso de estructuración se muestra en la Figura 6-2(b). En este modelo, un fichero es una sucesión de registros de longitud fija, cada uno de los cuales tiene cierta estructura interna. Un aspecto fundamental de la idea de que un fichero es una sucesión de registros es la idea de que la operación de lectura devuelve un registro y que la operación de escritura sobrescribe o añade un registro. Como nota histórica, en las décadas pasadas, cuando reinaba la tarjeta perforada de 80 columnas, muchos sistemas operativos de mainframe basaban sus sistemas de ficheros en ficheros formados por registros de 80 caracteres, que de hecho eran imágenes de tarjetas. Esos sistemas también reconocían ficheros de registros de 132 caracteres, destinados a impresoras (que en esa época eran grandes impresoras de cadena que imprimían 132 columnas). Los programas leían las entradas en unidades de 80 caracteres y escribían sus salidas en unidades de 132 caracteres, aunque los últimos 52 podían ser espacios, claro. Ningún sistema actual de uso general funciona ya así.

El tercer tipo de estructura de fichero se muestra en la Figura 6-2(c). En esta organización un fichero consiste en un árbol de registros, no todos necesariamente de la misma longitud, cada uno de los cuales contiene un campo **clave** en una posición fija del registro. El árbol está ordenado según el campo clave, con objeto de poder hallar con rapidez una clave en particular.

La operación básica aquí no es obtener el “siguiente” registro, aunque también puede hacerse, sino obtener el que tenga una clave dada. En el caso del fichero del zoológico de la Figura 6-2(c), se le podría pedir al sistema que obtenga el registro cuya clave es *Pony*, por ejemplo, sin preocuparse por su posición exacta en el fichero. Además, es posible añadir registros nuevos al fichero y dejar que sea el sistema operativo, no el usuario, quien decida dónde colocarlos. Es obvio que este tipo de fichero es muy distinto de los flujos de bytes no estructurados que se usan en UNIX y Windows, pero se utiliza en forma amplia en los grandes ordenadores mainframe que todavía se emplean en el procesamiento comercial de datos.

6.1.3 Tipos de ficheros

Muchos sistemas operativos reconocen varios tipos de ficheros. UNIX y Windows, por ejemplo, tienen ficheros regulares y directorios. UNIX también tiene ficheros especiales de bloques y de caracteres. Los **ficheros regulares** son los que contienen información del usuario. Todos los ficheros de la Figura 6-2 son ficheros regulares. Los **directorios** son ficheros del sistema que sirven para mantener la estructura del sistema de ficheros, y los estudiaremos más adelante. Los **ficheros especiales de caracteres** tienen que ver con la entrada/salida, y sirven para modelar dispositivos de E/S de tipo serie como terminales, impresoras y redes. Los **ficheros especiales de bloques** sirven para modelar discos. En este capítulo nos referiremos primordialmente a los ficheros regulares.

Los ficheros regulares son normalmente ficheros ASCII o ficheros binarios. Los ficheros ASCII consisten en líneas de texto. En algunos sistemas, cada línea termina con un carácter de retorno de carro; en otros se usa el carácter de salto de línea. Algunos sistemas (como MS-DOS) utilizan ambos. No es necesario que todas las líneas sea de la misma longitud.

La gran ventaja de los ficheros ASCII es que pueden visualizarse e imprimirse tal cual, y pueden editarse con cualquier editor de texto. Además, si un gran número de programas utiliza ficheros ASCII como su entrada y su salida, es fácil conectar la salida de un programa con la entrada de otro, como en las tuberías del shell. (La fontanería entre procesos no es nada fácil, pero interpretar la información ciertamente sí que lo es si se utiliza un convenio estándar para expresarla, tal como ASCII.)

Otros ficheros son binarios, lo que significa simplemente que no son ficheros ASCII. Si se escriben en una impresora se produce un listado incomprendible que parece estar lleno de basura. Normalmente, estos ficheros tienen alguna estructura interna conocida por los programas que los usan.

Por ejemplo en la Figura 6-3(a) vemos un fichero binario ejecutable sencillo tomado de una versión de UNIX. Aunque desde el punto de vista técnico el fichero no es más que una sucesión de bytes, el sistema operativo sólo puede ejecutar un fichero si éste tiene el formato correcto. Este fichero tiene cinco secciones: encabezado, texto, datos, bits de reubicación y tabla de símbolos. El encabezado comienza con lo que se conoce como un **número mágico**, el cual identifica el fichero como ejecutable (para evitar la ejecución accidental de un fichero que no tenga este formato). Luego vienen los tamaños de los diversos componentes del fichero, la dirección de la primera instrucción a ejecutar y algunos bits que actúan como indicadores. Después del encabezado vienen el texto y los datos del programa propiamente dicho. Éstos se cargan en la memoria y se reubican empleando los bits de reubicación. La tabla de símbolos sirve para depurar el programa.

Nuestro segundo ejemplo de fichero binario es también un fichero de UNIX. Consiste de una colección de procedimientos de biblioteca (módulos) compilados pero sin enlazar. Cada procedimiento va precedido por un encabezado que indica su nombre, la fecha en que se creó, el propietario, un código de protección y el tamaño. Al igual que en el fichero ejecutable, los encabezados de módulo están llenos de números binarios. Si se escribieran por una impresora se obtendría algo ilegible.

Todo sistema operativo debe reconocer al menos un tipo de fichero: sus propios ficheros ejecutables, pero algunos reconocen más. El viejo sistema TOPS-20 (para el sistema DEC 20) llegaba al extremo de examinar la hora de creación de cualquier fichero a ejecutar. Luego buscaba el fichero fuente y veía si había sido modificado desde la hora en que se había creado el binario. En tal caso, recompilaba automáticamente el fichero fuente. En términos de UNIX, sería como si el programa *make* se hubiera integrado en el shell. Las extensiones de fichero eran

obligatorias para que el sistema pudiera saber qué programa binario se había obtenido de qué fichero fuente.

Tener ficheros fuertemente tipados de esta forma provoca problemas cuando el usuario hace algo que los diseñadores del sistema no contemplaron. Consideremos, por ejemplo, un sistema en el que los ficheros de salida de los programas tienen la extensión *.dat* (ficheros de datos). Si un usuario escribe un formateador de programas que lee un fichero *.c* (programa en C), lo convierte (por ejemplo, a un esquema de sangrado convencional) y luego escribe el programa convertido como salida, el fichero de salida será de tipo *.dat*. Si el usuario trata de compilar este fichero con el compilador de C, el sistema se negará porque no tiene la extensión correcta. El sistema rechazará también cualquier intento por copiar *fichero.dat* en *fichero.c* por considerarlo incorrecto (para proteger al usuario contra equivocaciones).

Aunque esta “amabilidad con el usuario” podría ayudar a los novatos, exaspera a los usuarios experimentados, porque deben dedicar un tiempo considerable a buscar formas de sustraerse al concepto que tiene el sistema operativo de lo que es razonable y lo que no lo es.

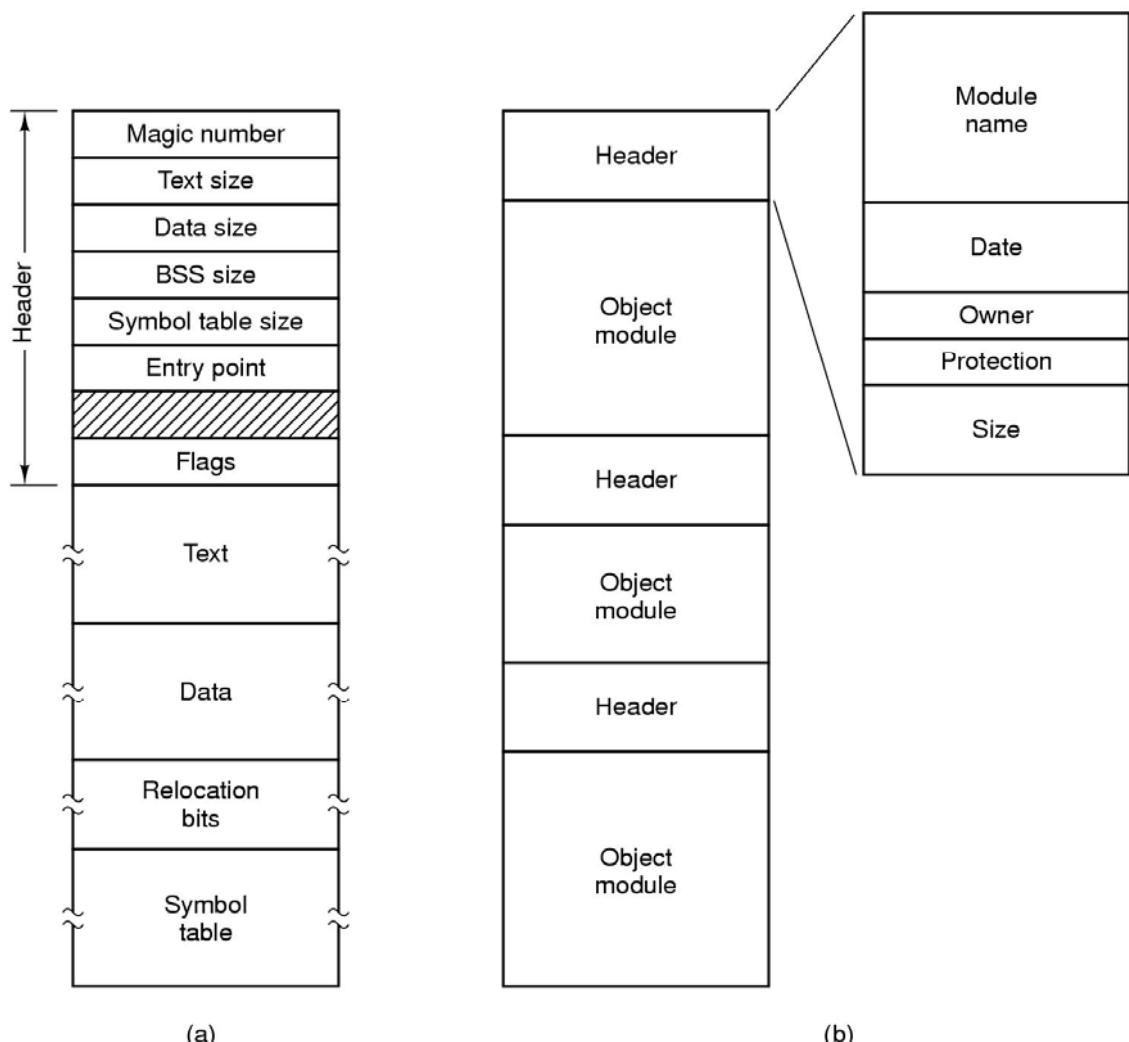


Figura 6-3. (a) Fichero ejecutable. (b) Un fichero de biblioteca.

6.1.4 Acceso a ficheros

Los primeros sistemas operativos sólo permitían un tipo de acceso a los ficheros: **acceso secuencial**. En aquellos sistemas, un proceso podía leer todos los bytes o registros de un fichero por orden, comenzando por el principio, pero no podía efectuar saltos para leerlos en otro orden. Lo que sí podía hacerse con los ficheros secuenciales era “rebobinarlos” para poder leerlos tantas veces como se quisiera. Los ficheros secuenciales eran apropiados cuando el medio de almacenamiento era la cinta magnética, no el disco.

Cuando comenzaron a usarse discos para almacenar ficheros se hizo posible leer los bytes o registros de un fichero sin un orden específico, o tener acceso a los registros por clave, no por posición. Los ficheros cuyos bytes o registros pueden leerse en cualquier orden se denominan ficheros de acceso aleatorio, y muchas aplicaciones los necesitan.

Los ficheros de acceso aleatorio son indispensables en muchas aplicaciones, como los sistemas de bases de datos. Si un cliente de una línea aérea llama para reservar un asiento en un vuelo dado, el programa de reservas deberá contar con la capacidad de acceder al registro de ese vuelo sin tener que leer primero los demás miles de vuelos existentes.

Se utilizan dos métodos para especificar dónde debe comenzar la lectura. En el primero, cada operación **read** da la posición en el fichero dónde debe comenzarse a leer. En el segundo, se cuenta con una operación especial, **seek**, para establecer la posición actual. Después del **seek**, el fichero podrá leerse de forma secuencial a partir de la posición que ahora es la actual.

En algunos sistemas operativos de mainframe antiguos, los ficheros se clasifican como secuenciales o de acceso aleatorio en el momento en que se crean. Esto permite al sistema emplear técnicas de almacenamiento distintas para las dos clases. Los sistemas operativos modernos no hacen esta distinción; todos los ficheros son de acceso aleatorio de forma automática.

6.1.5 Atributos de los ficheros

Todo fichero tiene un nombre y datos. Además, todos los sistemas operativos asocian otra información a cada fichero, como la fecha y la hora en que se creó, y su tamaño. Llamaremos a esta información adicional **atributos** del fichero. La lista de atributos varía de manera considerable de un sistema a otro. La tabla de la Figura 6-4 muestra algunas de las posibilidades, pero existen otras. Ningún sistema actual maneja todos estos atributos, pero todos están presentes en algún sistema.

Los primeros cuatro atributos tienen que ver con la protección del fichero e indican quién puede tener acceso a él y quien no. Es posible usar todo tipo de esquema, algunos de los cuales estudiaremos más adelante. En algunos sistemas el usuario debe presentar una contraseña para el acceso a un fichero, en cuyo caso la contraseña deberá ser uno de los atributos.

Los indicadores son bits o campos cortos que controlan o habilitan alguna propiedad específica. Los ficheros ocultos, por ejemplo, no aparecen en los listados de todos los ficheros. El indicador de archivado es un bit que indica si el fichero ya se respaldó o no. El programa de respaldo lo establece a 0 y el sistema lo pone a 1 cada vez que se modifica el fichero. Así, el programa de respaldo sabe qué ficheros deben respaldarse. El indicador temporal permite marcar un fichero para que se borre de forma automática cuando termine el proceso que lo creó.

Los campos de longitud del registro, posición de la clave y longitud de la clave sólo están presentes en ficheros cuyos registros pueden consultarse empleando una clave. Dichos campos proporcionan la información necesaria para hallar las claves.

Las diversas horas llevan el control de cuándo se creó el fichero, cuándo fue la última vez que se tuvo acceso a él y cuando fue la última vez que se modificó. Son útiles para varias cosas. Por ejemplo, si un fichero fuente se modificó después de crear el fichero objeto correspondiente, será necesario recompilarlo. Estos campos proporcionan la información necesaria.

| Attribute | Meaning |
|---------------------|---|
| Protection | Who can access the file and in what way |
| Password | Password needed to access the file |
| Creator | ID of the person who created the file |
| Owner | Current owner |
| Read-only flag | 0 for read/write; 1 for read only |
| Hidden flag | 0 for normal; 1 for do not display in listings |
| System flag | 0 for normal files; 1 for system file |
| Archive flag | 0 for has been backed up; 1 for needs to be backed up |
| ASCII/binary flag | 0 for ASCII file; 1 for binary file |
| Random access flag | 0 for sequential access only; 1 for random access |
| Temporary flag | 0 for normal; 1 for delete file on process exit |
| Lock flags | 0 for unlocked; nonzero for locked |
| Record length | Number of bytes in a record |
| Key position | Offset of the key within each record |
| Key length | Number of bytes in the key field |
| Creation time | Date and time the file was created |
| Time of last access | Date and time the file was last accessed |
| Time of last change | Date and time the file has last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

Figura 6-4. Algunos posibles atributos de un fichero.

El tamaño actual indica lo grande que es un fichero. Algunos sistemas operativos de mainframe antiguos exigen que se especifique el tamaño máximo cuando se crea un fichero, para poder reservar por adelantado la cantidad máxima de espacio de almacenamiento. Los sistemas operativos de estaciones de trabajo y ordenadores personales son lo bastante inteligentes como para prescindir de esa información.

6.1.6 Operaciones con ficheros

Los ficheros existen para guardar información y poder recuperarla después. Los distintos sistemas ofrecen diferentes operaciones de almacenamiento y recuperación. A continuación estudiaremos las llamadas al sistema más comunes relacionadas con los ficheros.

1. **Create.** Se crea el fichero sin datos. El objetivo de la llamada es anunciar que va a haber un fichero nuevo y establecer algunos de sus atributos.
2. **Delete.** Si ya no se necesita un fichero, conviene borrarlo para desocupar el espacio en disco. Siempre hay una llamada al sistema para ese fin.
3. **Open.** Antes de usar un fichero, un proceso debe abrirlo. El propósito de la llamada open es que el sistema obtenga los atributos y la lista de direcciones de disco y los

coloque en la memoria principal para tener acceso a ellos rápidamente en llamadas posteriores.

4. **Close.** Una vez que han terminado todos los accesos, ya no se necesitarán los atributos y direcciones en disco, por lo que es recomendable cerrar el fichero para desocupar espacio en las tablas internas. Muchos sistemas fomentan esto imponiendo un límite para el número de fichero que pueden tener abiertos los procesos. Los discos se escriben en bloques, y el cierre de un fichero hace que se escriba el último bloque del fichero, aunque no esté lleno por completo.
5. **Read.** Se leen datos de un fichero. Normalmente, los bytes provienen de la posición actual. Quien efectúa la llamada debe especificar cuántos datos necesita, y el búfer donde deben colocarse.
6. **Write.** Se escriben datos en un fichero, también, normalmente, en la posición actual. Si la posición actual es el fin del fichero, aumenta el tamaño del fichero. Si la posición actual está en un punto intermedio del fichero, los datos existentes se sobrescriben y se perderán sin remedio.
7. **Append.** Esta llamada es una forma restringida de **write**; con ella sólo se puede agregar datos al final del fichero. Los sistemas que ofrecen un número mínimo de llamadas al sistema por lo general no tienen **append**, pero muchos sistemas ofrecen varias formas de hacer lo mismo, y en algunos casos cuentan con **append**.
8. **Seek.** En el caso de ficheros de acceso aleatorio, se requiere alguna forma de especificar el punto del fichero de donde se tomarán los datos. Un método común es usar una llamada al sistema, **seek**, que sitúe el puntero del fichero en un lugar específico del fichero. Una vez ejecutada esta llamada, podrán leerse datos de esa posición, o escribir en ella.
9. **Get attributes.** Muchas veces los procesos necesitan leer los atributos de un fichero para efectuar su trabajo. Por ejemplo, el programa *make* de UNIX se usa por lo común para administrar proyectos de desarrollo de software que contienen muchos ficheros fuente. Cuando se invoca a *make* se examinan los tiempos de modificación de todos los ficheros fuente y objeto y se determina el número mínimo de compilaciones necesarias para que todo esté actualizado. Para efectuar su trabajo, el sistema debe examinar atributos, a saber, las horas de modificación.
10. **Set attributes.** El usuario puede establecer algunos de los atributos, o modificarlos después de que se creó el fichero, y eso se logra con esta llamada al sistema. La información de modo de protección es un ejemplo obvio. Casi todos los indicadores pertenecen también a esa categoría.
11. **Rename.** Es común que un usuario necesite cambiar el nombre de un fichero existente. Esta llamada al sistema lo hace posible. No siempre es estrictamente necesaria, pues por lo general el fichero puede copiarse en un fichero nuevo con el nuevo nombre, borrando después el fichero viejo.

6.1.7 Ejemplo de programa que utiliza llamadas al sistema de ficheros

En esta sección examinaremos un sencillo programa en UNIX que copia un fichero de su fichero origen a su fichero de destino. El listado aparece en la Figura 6-5. El programa tiene una funcionalidad mínima y sus informes de errores son más rudimentarios todavía, pero da una

idea razonable de cómo funcionan algunas de las llamadas al sistema relacionadas con los ficheros.

El programa, *copyfile*, puede invocarse, por ejemplo, con el siguiente comando

```
copyfile abc xyz
```

para copiar el fichero *abc* en *xyz*. Si *xyz* ya existe, se sobrescribe; si no, se crea. El programa debe invocarse exactamente con dos argumentos, ambos nombres de fichero válidos.

Las cuatro directivas *#include* cerca de la parte superior del programa hacen que se incluya un gran número de definiciones y prototipos de función en el programa. Son necesarios para que el programa se ajuste a las normas internacionales pertinentes, pero no nos ocuparemos más de ellos. La siguiente línea es un prototipo de función para *main*, algo que exige el ANSI C, pero que tampoco es importante para nuestros fines.

La primera directiva *#define* es una definición de macro que especifica la cadena *BUFSIZE* como una macro que se expande al número 4096. El programa leerá y escribirá en porciones de 4096 bytes. Se considera una buena práctica de programación asignar nombres a las constantes de este tipo y usar los nombres en lugar de las constantes. Este convenio no sólo facilita la lectura de los programas, sino que facilita también su mantenimiento. La segunda directiva *#define* determina quién puede tener acceso al fichero de salida.

El programa principal se llama *main* y tiene dos argumentos, *arc* y *argv*. Éstos los proporciona el sistema operativo cuando se invoca al programa. El primero indica cuantas cadenas estaban presentes en la línea del comando introducido para ejecutar el programa, incluido el propio nombre del programa. En este caso deberá ser 3. El segundo parámetro es un array de punteros a los argumentos. En el ejemplo de llamada dado aquí, los elementos de dicho array contendrían punteros a los siguientes valores:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

Por medio de este array el programa tiene acceso a sus argumentos.

Se declaran cinco variables. Las dos primeras, *in_fd* y *out_fd*, contienen los **descriptores de fichero**, número enteros pequeños que se devuelven cuando se abre un fichero. Las dos siguientes, *rd_count* y *wt_count*, son los contadores de bytes devueltos por las llamadas al sistema *read* y *write*, respectivamente. La última, *buffer*, es el búfer empleado para contener los datos leídos y suministrar los datos a escribir.

La primera instrucción propiamente dicha verifica si *argc* es 3. Si no, termina el programa con el código de estado 1. Cualquier código de estado distinto de 0 implica que hubo un error. El código de estado es el único informe de errores que produce este programa. Una versión de producción normalmente debería imprimir también los correspondientes mensajes de error.

```
/* Programa de copia de ficheros. */
/* Con una mínima comprobación y comunicación de errores */
```

```
#include <sys/types.h>      /* se incluyen los ficheros de cabecera necesarios */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
```

```

int main ( int argc, char * argv[] ) ; /* prototipo ANSI */
#define BUF_SIZE 4096           /* tamaño de bufer de 4096 bytes */
#define OUTPUT_MODE 0700        /* bits de proteccion para el fichero de salida */
int main ( int argc, char * argv[] )
{
    int in_fd, out_fd, rd_count, wt_count ;
    char buffer[BUF_SIZE] ;

    if (argc != 3) exit(1);      /* error de sintaxis si argc no es 3 */

    /* Abre el fichero de entrada y crea el fichero de salida */
    in_fd = open(argv[1], O_RDONLY);          /* abre el fichero de origen */
    if (in_fd < 0) exit(2);                  /* si no se puede terminar */
    out_fd = creat(argv[2], OUTPUT_MODE);     /* crea el fichero de destino */
    if (out_fd < 0) exit(3);                  /* si no se puede terminar */

    /* bucle de copia */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* lee bloque de datos */
        if (rd_count <= 0) break;             /* si fin de fichero o error, salir del bucle */
            wr_count = write(out_fd, buffer, rd_count); /* escribe datos */
            if (wt_count <= 0) exit(4);         /* wt_count <= 0 es un error */
    }

    /* Cierra los ficheros */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)
        exit(0);                         /* no hubo error en la ultima lectura */
    else
        exit(5);                         /* hubo un error en la ultima lectura */
}

```

Figura 6-5. Programa sencillo para copiar un fichero.

Luego se intenta abrir el fichero de origen y crear el de destino. Si se logra abrir el fichero de origen, el sistema asigna un entero pequeño a *in_fd* para identificar el fichero. Las llamadas subsiguientes deberán incluir este entero para que el sistema sepa qué fichero quieren. De forma similar, si se logra crear el fichero de destino, *out_fd* recibe un valor que lo identifica. El segundo argumento de *creat* establece el modo de protección. Si fracasa la apertura o bien la creación, se asigna -1 al descriptor del fichero correspondiente, y el programa termina con un código de error.

Ahora viene el bucle de copiado. Lo primero que hace es tratar de leer 4 KB de datos y colocarlos en el búfer. Esto lo hace invocando el procedimiento de biblioteca *read*, que es el que emite la llamada al sistema *read*. El primer parámetro identifica el fichero, el segundo identifica el búfer y el tercero indica cuántos bytes hay que leer. El valor asignado a *rd_count* es el número de bytes que se leyeron en realidad. Comúnmente, este número será 4096, a menos que queden menos bytes en el fichero. Cuando se llegue al final del fichero, el valor será cero. Si *rd_count* llega a ser cero o negativo, la copia no podrá continuar, así que se ejecutará la instrucción *break* para salir del bucle (que de otro modo sería infinito).

La llamada a *write* envía el contenido del búfer al fichero de destino. El primer parámetro identifica el fichero, el segundo da el búfer y el tercero indica cuántos bytes hay que

escribir, de forma análoga a *read*. Cabe señalar que el contador de bytes es el número que se leyó en realidad, no *BUF_SIZE*. Este punto es importante porque la última lectura no devolverá 4096, a menos que el tamaño del fichero sea un múltiplo de 4 KB.

Una vez que se haya procesado todo el fichero, la primera llamada que rebase el fin del fichero asignará 0 a *rd_count*, con lo cual se saldrá del bucle. En este punto los dos ficheros se cierran y el programa termina con un código de estado que indica que lo hizo de forma normal.

Aunque las llamadas al sistema de Windows son diferentes de las de UNIX, la estructura general de un programa de Windows de línea de comandos para copiar un fichero es más o menos similar al de la Figura 6-5. Examinaremos las llamadas de Windows 2000 en el capítulo 11.

6.1.8 Ficheros con correspondencia en memoria

Muchos programadores opinan que tener acceso a los ficheros como acabamos de ver es un método torpe y poco recomendable, sobre todo si se le compara con el acceso a la memoria ordinaria. Por ese motivo, algunos sistemas operativos, comenzando por MULTICS, han incluido un mecanismo para establecer una correspondencia entre los ficheros y el espacio de direcciones de un proceso en ejecución. Desde el punto de vista conceptual, podemos imaginar la existencia de dos nuevas llamadas al sistema, *map* y *unmap*. La primera proporciona un nombre de fichero y una dirección virtual, y hace que el sistema operativo establezca una correspondencia del fichero con el espacio de direcciones a partir de la dirección virtual.

Por ejemplo, supongamos que se establece una correspondencia entre un fichero, *f*, cuya longitud es 64 KB, y el espacio de direcciones virtual a partir de la dirección 512 K. Entonces cualquier instrucción de máquina que lea el contenido del byte que está en 512 K obtendrá el byte 0 del fichero, y así de forma sucesiva. Del mismo modo, una escritura en la dirección 512 K + 21000 modificará el byte 21000 del fichero. Cuando termina el proceso, el fichero modificado queda en el disco, como si hubiera sido modificado por una combinación de llamadas al sistema *seek* y *write*.

Lo que sucede en verdad es que las tablas internas del sistema se modifican de modo que el fichero se convierta en el almacén de respaldo para la región de memoria que está entre 512 K y 576 K. Así, una lectura desde 512 K genera una falta de página y hace que se traiga la página 0 del fichero. De forma similar, una escritura en 512 K + 1100 provoca un fallo de página que trae a la memoria la página que contiene esa dirección, después de lo cual puede efectuarse la escritura en memoria. Si esa página llega a ser desalojada por el algoritmo de sustitución de páginas se escribe en el lugar correcto del fichero. Cuando termina el proceso, todas las páginas con correspondencia que se hayan modificado se escriben de vuelta en sus ficheros.

La correspondencia de ficheros funciona de manera óptima en un sistema que maneja segmentación. En un sistema así, cada fichero puede establecer una correspondencia con su propio segmento para que el byte *k* del fichero sea también el byte *k* del segmento. En la Figura 6-6(a) vemos un proceso que tiene dos segmentos, uno de código y otro de datos. Supongamos que este proceso copia ficheros, como el programa de la Figura 6-5. Primero establece una correspondencia entre el fichero de origen, digamos *abc*, y un segmento. Luego crea un segmento vacío y establece una correspondencia entre éste y el fichero de destino, que en nuestro caso es *xyz*. Estas operaciones producen la situación que se muestra en la Figura 6-6(b).

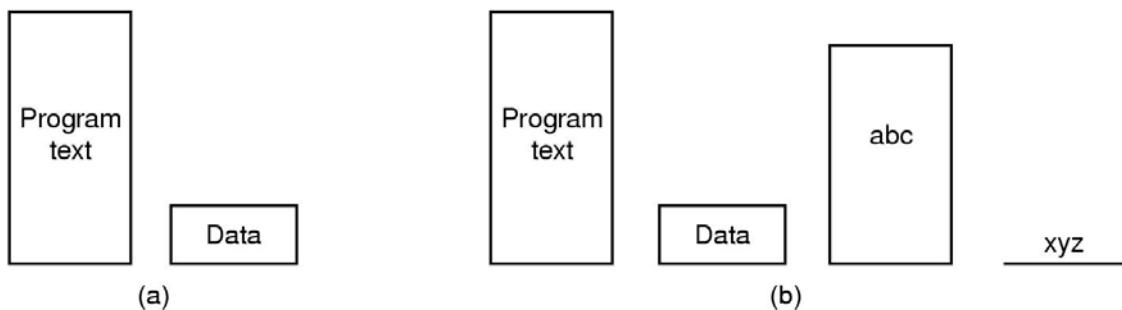


Figura 6-6. (a) Proceso segmentado antes de establecer las correspondencias entre los ficheros y su espacio de direcciones. (b) El proceso después de establecer una correspondencia entre un fichero existente *abc* y un segmento, y crear un segmento nuevo para el fichero *xyz*.

Ahora el proceso puede copiar el segmento de origen en el de destino, utilizando un bucle de copia ordinario. No se necesitan llamadas al sistema `read` ni `write`. Una vez que lo ha hecho, el proceso puede ejecutar la llamada al sistema `unmap` para eliminar los ficheros del espacio de direcciones y luego terminar. Ahora ya existe el fichero de salida, *xyz*, como si se hubiera creado de la forma convencional.

Aunque la correspondencia de ficheros elimina la necesidad de E/S y, por tanto, facilita la programación, presenta algunos problemas. En primer lugar, para el sistema es difícil conocer la longitud exacta del fichero de salida, *xyz* en nuestro ejemplo. Es fácil averiguar el número de la página más grande que se escribió, pero no hay forma de saber cuántos bytes de esa página se escribieron. Supongamos que el programa sólo utiliza la página 0, y que después de la ejecución todos los bytes siguen siendo 0 (su valor inicial). Tal vez *xyz* es un fichero que consta de 10 ceros. Tal vez es un fichero que consta de 100 ceros. Tal vez es un fichero que consta de 1000 ceros. ¿Quién sabe? El sistema operativo no. Lo único que puede hacerse es crear un fichero cuya longitud sea igual al tamaño de página.

Puede presentarse un segundo problema (potencialmente) si un proceso hace corresponder un fichero y otro lo abre para leerlo de la forma convencional. Si el primer proceso modifica una página, ese cambio no se reflejará en el fichero en disco hasta que la página sea desalojada. El sistema debe tener mucho cuidado para asegurar que los dos procesos no vean versiones incongruentes del fichero.

Un tercer problema del establecimiento de la correspondencia es que un fichero podría ser mayor que un segmento, o incluso mayor que todo el espacio de direcciones virtual. La única salida es que la llamada al sistema `map` pueda hacer corresponder sólo una porción de fichero, no todo el fichero. Aunque esto funciona, es a todas luces menos satisfactorio que hacer corresponder el fichero entero.

6.2 DIRECTORIOS

Para llevar el control de los ficheros, los sistemas de ficheros suelen tener **directorios** o **carpetas** que, en muchos sistemas son a su vez ficheros. En esta sección veremos los directorios, su organización, sus propiedades y las operaciones que pueden realizarse con ellos.

6.2.1 Sistemas de directorios a un solo nivel

La forma más sencilla de sistema de directorios es que un directorio contenga todos los ficheros. A veces se le llama **directorío raíz**, pero dado que es el único, el nombre no importa mucho. En los primeros ordenadores personales este sistema era muy común, en parte porque sólo había un usuario. Resulta interesante que el primer superordenador del mundo, el CDC 6600, también tenía un único directorio para todos los ficheros, aunque la utilizaban muchos usuarios a la vez. Esta decisión sin duda se tomó para que el diseño del software fuera lo más sencillo posible.

En la Figura 6-7 se muestra un ejemplo de sistema con un único directorio. Aquí el directorio contiene cuatro ficheros. En la figura se muestran los *propietarios* de los ficheros, no los *nombres* de los ficheros (porque los propietarios son importantes para lo que vamos a decir). Las ventajas de este esquema son su sencillez y la capacidad para localizar ficheros con rapidez; después de todo sólo pueden estar en un lugar.

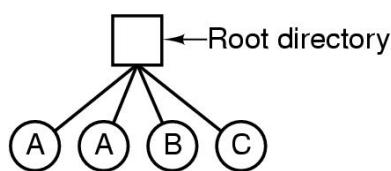


Figura 6-7. Sistema de directorios a un solo nivel que contiene cuatro ficheros, propiedad de tres personas A, B y C.

El problema de tener sólo un directorio en un sistema con múltiples usuarios es que diferentes usuarios podrían usar por accidente los mismos nombres para sus ficheros. Por ejemplo, si el usuario *A* crea un fichero llamado *correo*, y luego el usuario *B* crea también un fichero llamado *correo*, el fichero de *B* sobrescribirá al de *A*. Por ello, este esquema ya no se usa en los sistemas multiusuario, pero podría usarse en un sistema empotrado pequeño, como un sistema en un automóvil, diseñado para almacenar perfiles de un número reducido de conductores.

6.2.2 Sistemas de directorios a dos niveles

Para evitar conflictos cuando dos usuarios escogen el mismo nombre para sus propios ficheros, el siguiente escalón sería dar a cada usuario un directorio privado. Así, los nombres escogidos por un usuario no chocarán con los escogidos por otro, y no habrá problemas si el mismo nombre aparece en dos o más directorios. Este diseño lleva al sistema de la Figura 6-8. Podría usarse, por ejemplo, en un ordenador multiusuario o en una red simple de ordenadores personales que comparten un servidor de ficheros en una red local.

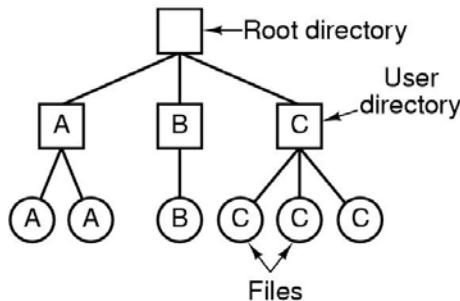


Figura 6-8. Sistema de directorios a dos niveles. Las letras indican los propietarios de los directorios y de los ficheros.

Algo implícito en este diseño es que cuando un usuario trata de abrir un fichero, el sistema necesita saber de qué usuario se trata para saber en qué directorio buscar. Por ello, se requiere algún tipo de procedimiento de inicio de sesión (*login*) en el que el usuario especifica un nombre o una identificación de inicio de sesión, algo que no era necesario en los sistemas de directorios de un nivel.

Cuando se implementa este sistema en su forma más básica, los usuarios sólo pueden tener acceso a ficheros de sus propios directorios. Sin embargo, una extensión del esquema podría permitir a los usuarios el acceso a ficheros de otros usuarios, si indican de quién es el fichero que desean abrir. Por ejemplo,

```
open("x")
```

podría ser la llamada para abrir un fichero llamado *x* en el directorio del usuario, y

```
open("nancy/x")
```

podría ser la llamada para abrir un fichero llamado *x* que está en el directorio de otro usuario, Nancy.

Una situación en la que los usuarios necesitan tener acceso a ficheros distintos de los propios es la ejecución de programas binarios del sistema. Es obvio que sería poco eficiente tener copias de todos los programas de utilidad en cada uno de los directorios. Como mínimo, se necesita un directorio del sistema que contenga los programas binarios ejecutables.

6.2.3 Sistemas de directorios jerárquicos

La jerarquía de dos niveles elimina los conflictos de nombres entre usuarios pero no es satisfactoria para usuarios que tienen un gran número de ficheros. Resulta inadecuada incluso en un ordenador personal con un único usuario. Es muy común que los usuarios quieran agrupar sus ficheros de forma lógica. Un profesor, por ejemplo, podría tener una serie de fichero que juntos integran un libro que está escribiendo para un curso, una segunda colección de ficheros formada por programas que han presentado los estudiantes para otro curso, un tercer grupo de ficheros que contienen el código de un sistema avanzado para escribir compiladores que está desarrollando, un cuarto grupo de ficheros que contienen propuestas de becas, así como otros ficheros de correo electrónico, minutos de reuniones, artículos que está escribiendo, juegos, etc. Se necesita alguna forma de agrupar estos ficheros dentro de esquemas flexibles determinados por el usuario.

Lo que se necesita es una jerarquía general (es decir, un árbol de directorios). Con este enfoque, cada usuario puede tener tantos directorios como necesite para agrupar sus ficheros en categorías naturales. El enfoque se muestra en la Figura 6-9. Aquí, los directorios *A*, *B* y *C* contenidos en el directorio raíz pertenecen cada uno a un usuario distinto, dos de los cuales han creado un subdirectorios para los proyectos en los que están trabajando.

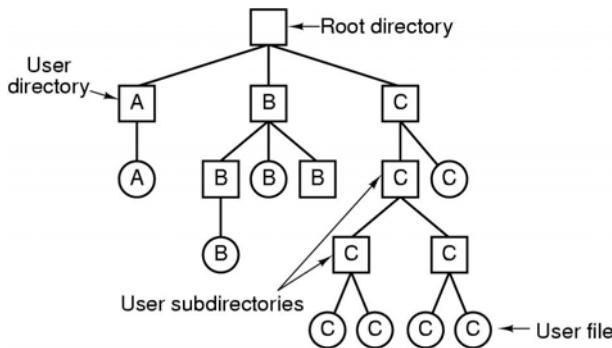


Figura 6-9. Sistema de directorios jerárquico.

6.2.4 Nombres de camino

Cuando un sistema de ficheros está organizado como un árbol de directorios, se necesita un mecanismo para especificar los nombres de fichero. Por lo común se utilizan dos métodos. En el primero, cada fichero recibe un **nombre de camino absoluto** que consiste en el camino que debe seguirse para llegar del directorio raíz hasta el fichero. Por ejemplo, el camino `/usr/ast/correo` nos indica que el directorio raíz contiene un subdirectorio, `usr`, que a su vez contiene un subdirectorio, `ast`, que contiene el fichero `correo`. Los nombres de camino absolutos siempre parten del directorio raíz y son únicos. En UNIX los componentes del camino se separan con `/`. En Windows el separador es `\`. En MULTICS era `>`. Así el mismo nombre de camino se escribiría como sigue en esos tres sistemas:

| | |
|---------|---------------------------------------|
| Windows | <code>\usr\ast\correo</code> |
| UNIX | <code>/usr/ast/correo</code> |
| MULTICS | <code>>usr>ast>correo</code> |

Sea cual sea el carácter empleado, si el primer carácter del nombre de camino es el separador, el camino será absoluto.

El otro tipo de nombre es el **nombre de camino relativo**. Éste se utiliza junto con el concepto de **directorío de trabajo** (también llamado **directorío actual**). Un usuario puede designar un directorio como su directorio de trabajo actual, en cuyo caso todos los nombres de camino que no comiencen en el directorio raíz se considerarán relativos al directorio de trabajo. Por ejemplo, si el directorio de trabajo actual es `/usr/ast`, podrá hacerse referencia al fichero cuyo camino absoluto es `/usr/ast/correo` simplemente mediante `correo`. Dicho de otro modo, en UNIX el comando

```
cp /usr/ast/correo /usr/ast/correo.bak
```

y el comando

```
cp correo correo.bak
```

hacen exactamente lo mismo si el directorio de trabajo es `/usr/ast`. La forma relativa suele ser más conveniente, pero hace lo mismo que la forma absoluta.

Algunos programas necesitan tener acceso a un fichero específico sin importar cuál sea el directorio de trabajo; en tal caso, siempre deberán utilizar nombres de camino absolutos. Por ejemplo, un corrector ortográfico podría tener que leer `/usr/lib/diccionario` para realizar su trabajo. En este caso deberá utilizar el nombre de camino absoluto porque no sabe

cuál será el directorio de trabajo en el momento en que se necesite el diccionario. El nombre de camino absoluto siempre funciona, sea cual sea el directorio de trabajo.

Desde luego, si el corrector ortográfico necesita un gran número de ficheros de */usr/lib*, una estrategia alternativa sería emitir una llamada al sistema para cambiar su directorio de trabajo a */usr/lib*, y luego utilizar simplemente diccionario como primer parámetro de *open*. Al cambiar en forma explícita el directorio de trabajo, el programa sabe con certeza en qué parte del árbol de directorios está, y puede utilizar caminos relativos.

Cada proceso tienen su propio directorio de trabajo, así que cuando un proceso cambia su directorio de trabajo y después termina, ningún otro proceso se ve afectado y no quedan rastros del cambio en el sistema de ficheros. De esta forma siempre es perfectamente seguro para un proceso cambiar su directorio de trabajo cuando le convenga. Por otra parte, si un *procedimiento de biblioteca* cambia el directorio de trabajo del programa y al terminar no regresa a donde estaba, es posible que el resto del programa no funcione porque la que se supone su ubicación podría no serlo. Por este motivo, los procedimientos de biblioteca casi nunca cambian el directorio de trabajo, y si tienen que hacerlo, siempre lo restauran antes de terminar.

La mayoría de los sistemas operativos que disponen de sistema de directorio jerárquico tienen dos entradas especiales en cada directorio, “.” Y “..” que normalmente se pronuncian “punto” y “punto punto”. Punto se refiere al directorio actual; punto punto se refiere a su padre. Para ver cómo se utilizan esas entradas, consideremos el árbol de ficheros UNIX de la Figura 6-10. Cierto proceso tiene */usr/ast* como su directorio de trabajo. Ese proceso puede utilizar “..” para subir por el árbol. Por ejemplo, el proceso puede copiar el fichero */usr/ast/diccionario* a su propio directorio emitiendo el comando

```
cp ..//lib/diccionario .
```

El primer camino le indica al sistema que suba en la jerarquía (al directorio *usr*) y luego que baje al directorio *lib* para hallar el fichero *diccionario*.

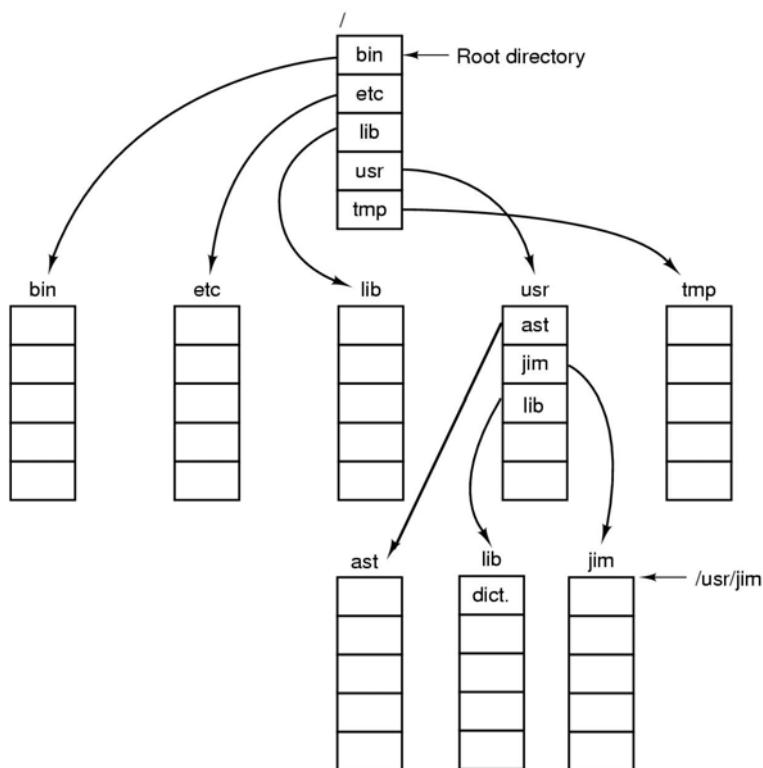


Figura 6-10. Un árbol de directorios UNIX.

El segundo argumento (punto) nombra el directorio actual. Cuando el comando *cp* recibe un nombre de directorio (que puede ser punto) como segundo argumento, copia todos los ficheros ahí. Desde luego, una forma más normal de efectuar la copia sería teclear

```
cp /usr/lib/diccionario .
```

Aquí el uso de punto ahorra al usuario el trabajo de teclear otra vez diccionario. No obstante, teclear

```
cp /usr/lib/diccionario diccionario
```

también funciona a la perfección, lo mismo que

```
cp /usr/lib/diccionario /usr/ast/diccionario
```

Todos estos comandos hacen exactamente lo mismo.

6.2.5 Operaciones con directorios

Las llamadas al sistema que pueden emitirse para administrar los directorios presentan más variaciones entre los diferentes sistemas que las llamadas para administrar los ficheros. Para dar una idea de cuáles son y cómo funcionan, damos la siguiente muestra (tomada de UNIX).

1. **Create.** Se crea un directorio, el cual está vacío excepto por punto y punto punto, que el sistema coloca ahí (o, en algunos casos, el programa *mkdir*).
2. **Delete.** Se elimina un directorio. Sólo es posible eliminar un directorio vacío. Se considera vacío un directorio que sólo contiene los directorios punto y punto punto. Por lo general no pueden borrarse esas entradas.
3. **Opendir.** Los directorios pueden leerse. Por ejemplo, si se desea visualizar la lista de todos los ficheros contenidos en un directorio, el programa que los visualiza debe abrir el directorio para leer los nombres de todos los ficheros que contiene. Para poder leer un directorio es necesario abrirlo antes, de forma análoga a como se abre y se lee un fichero.
4. **Closedir.** Una vez que se ha terminado de leer un directorio, debe cerrarse para desocupar espacio en las tablas internas.
5. **Readdir.** Esta llamada devuelve la siguiente entrada de un directorio abierto. Antes era posible leer directorios utilizando la llamada al sistema **read** normal, pero eso tenía la desventaja de que obligaba al programador a conocer y tener en cuenta la estructura interna de los directorios. En cambio, **readdir** siempre devuelve una entrada en el formato estándar, sin importar cuál de las posibles estructuras de directorio se estén utilizando.
6. **Rename.** En muchos sentidos, los directorios son como ficheros y se les puede cambiar el nombre igual que a los ficheros.
7. **Link.** El enlazado es una técnica que permite a un fichero aparecer en más de un directorio. Esta llamada al sistema especifica un fichero existente y un nombre de camino, y crea un enlace entre el fichero existente y el nombre especificado por el camino. Así el mismo fichero podría aparecer en múltiples directorios. Un enlace de este tipo, que incrementa el contador en el i-nodo del fichero (para llevar la cuenta

del número de entradas de directorio que contienen al fichero), se conoce como **enlace duro**.

8. **Unlink.** Se elimina una entrada de directorio. Si el fichero que se está desenlazando sólo está presente en un directorio (que es lo más común), se elimina del sistema de ficheros. Si está presente en varios directorios, sólo se elimina el nombre de camino especificado; los demás permanecerán. En UNIX, la llamada al sistema para borrar ficheros (que vimos antes) en realidad es `unlink`.

La lista anterior incluye las llamadas más importantes, pero hay unas pocas más, como las que administran la información de protección asociada con un directorio.

6.3 IMPLEMENTACIÓN DEL SISTEMA DE FICHEROS

Ha llegado el momento de pasar a la perspectiva que tiene el usuario acerca del sistema de ficheros, a la perspectiva del implementador. A los usuarios les preocupa qué nombres tienen los ficheros, qué operaciones pueden ejecutarse con ellos, qué aspecto tiene el árbol de directorios, y cuestiones de interfaz similares. A los implementadores les preocupa la forma en que se almacenan los ficheros y directorios, cómo se administra el espacio en disco y cómo puede hacerse que todo funcione de manera eficiente y confiable. En las secciones que siguen examinaremos varias de esas áreas para ver qué problemas y sacrificios implican.

6.3.1 Organización del sistema de ficheros

Los sistemas de ficheros se almacenan en discos. Casi todos los discos pueden dividirse en una o más particiones, con sistemas de ficheros independientes en cada partición. El sector 0 del disco se llama **registro maestro de arranque (MBR; Master Boot Record)** y sirve para arrancar el ordenador. El final del MBR contiene la tabla de particiones. Esta tabla contiene las direcciones inicial y final de cada partición. Una de las particiones de la tabla está marcada como activa. Cuando se enciende el ordenador, el BIOS lee el MBR del disco y lo ejecuta. Lo primero que hace el programa del MBR es localizar la partición activa, leer su primer bloque, llamado **bloque de arranque**, y ejecutarlo. El programa del bloque de arranque carga el sistema operativo contenido en esa partición. Por uniformidad, cada partición, comienza con un bloque de arranque aun que no contenga un sistema operativo arrancable. De cualquier modo, ese bloque podría contener uno en el futuro, por lo que es una buena idea reservarlo.

A parte de comenzar con un bloque de arranque, la organización de una partición del disco varía de forma considerable de un sistema de ficheros a otro. Es común que el sistema de ficheros contenga algunos de los elementos que se muestran en la Figura 6-11. El primero es el **superbloque**, que contiene todos los parámetros clave acerca del sistema de ficheros y se transfiere del disco a la memoria cuando se arranca el ordenador o cuando se toca por primera vez el sistema de ficheros. La información que suele contener un superbloque incluye un número mágico para identificar el tipo de sistema de ficheros, el número de bloques que hay en el sistema de ficheros y otra información administrativa crucial.

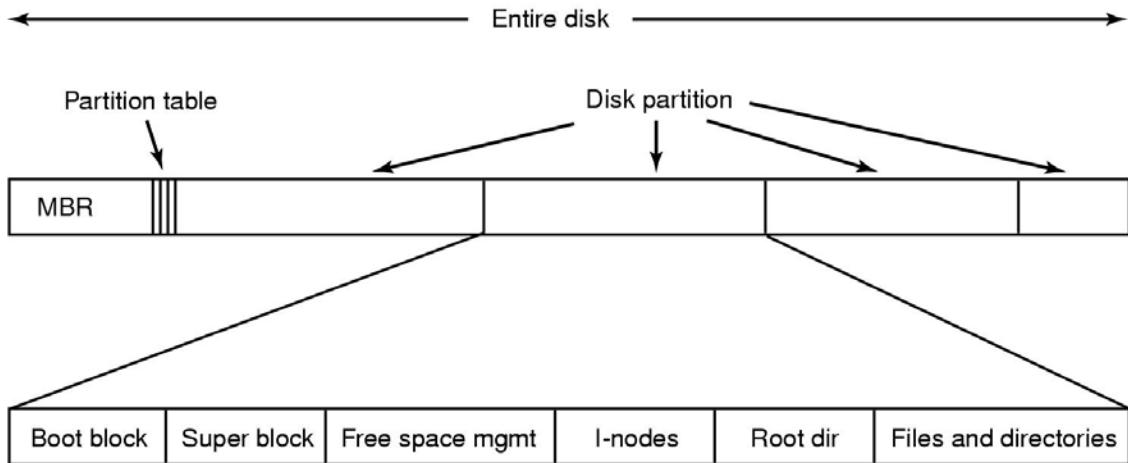


Figura 6-11. Una posible organización del sistema de ficheros.

A continuación podría haber información acerca de bloques libres en el sistema de ficheros, por ejemplo, en forma de mapa de bits o de una lista de punteros. Luego podrían estar los i-nodos, un array de estructuras, una por fichero, que proporciona todas las características del fichero. Después podría venir el directorio raíz, que contiene la parte más alta del árbol del sistema de ficheros. Por último, el resto del disco suele contener los demás directorios y ficheros.

6.3.2 Implementación de los ficheros

Tal vez el aspecto más importante de la implementación del almacenamiento de los ficheros sea llevar el control de qué bloques de disco corresponden a qué fichero. Se emplean diversos métodos en los distintos sistemas operativos. En esta sección examinaremos algunos de ellos.

Asignación contigua

El esquema de asignación más simple es almacenar cada fichero en una serie contigua de bloques de disco. Así en un disco con bloques de 1 KB, se asignarían 50 bloques consecutivos a un fichero de 50 KB. Si los bloques fueran de 2 KB, se le asignarían 25 bloques consecutivos .

Vemos un ejemplo de asignación de almacenamiento contiguo en la Figura 6-12(a). Allí se muestran los primeros 40 bloques de disco, comenzando con el bloque 0 a la izquierda . En un principio, el disco estaba vacío. Luego se escribió en el disco un fichero *A* con una longitud de cuatro bloques a partir del principio (bloque 0). Despues se escribió un fichero de seis bloques, *B*, inmediatamente después de fichero *A*. Cabe señalar que cada fichero comienza al principio de un bloque nuevo, de modo que si el fichero *A* en realidad ocupara 3,5 bloques, se desperdiciaría algo de espacio al final del último bloque. En la figura se muestra un total de siete ficheros, cada uno comenzando en el bloque que sigue al último bloque del fichero anterior. Se utiliza sombreado para que sea más fácil distinguir los ficheros.

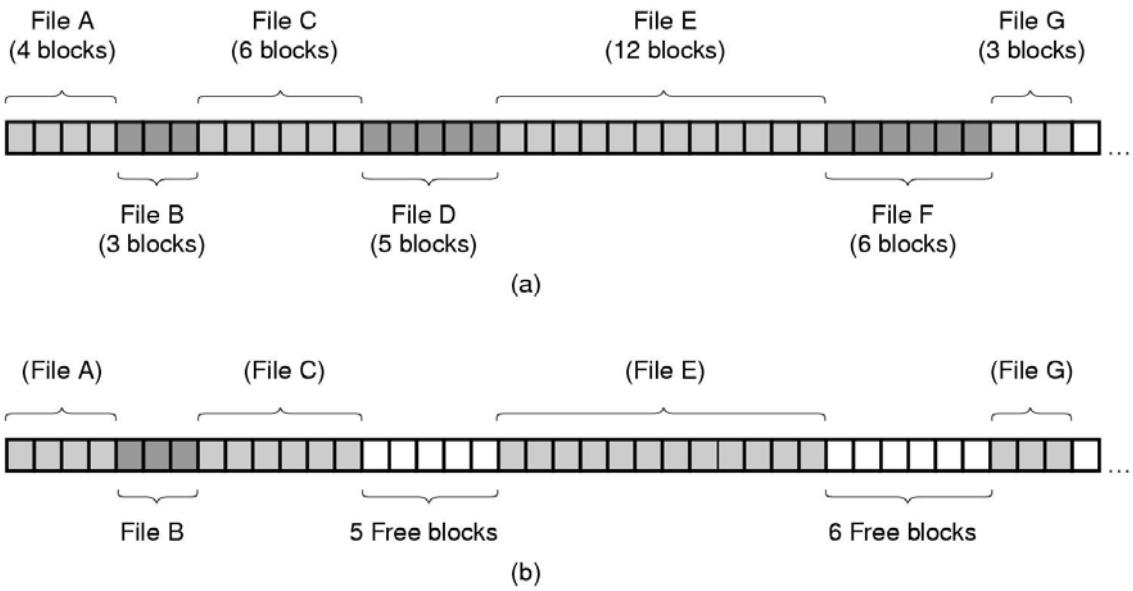


Figura 6-12. (a) Asignación contigua de espacio en disco para siete ficheros. (b) El estado del disco después de borrar los ficheros D y F.

La asignación de espacio contiguo en disco tiene dos ventajas importantes. La primera es que su implementación es sencilla porque para llevar el control de dónde están los bloques de un fichero basta con recordar dos números: la dirección en disco del primer bloque y el número de bloques del fichero. Dado el número del primer bloque, se podrá hallar el número de cualquier otro bloque mediante una simple suma.

La segunda ventaja es que la eficiencia en lectura es excelente porque puede leerse todo el fichero del disco en una sola operación. Sólo se necesita un desplazamiento del brazo (al primer bloque). Después no se requiere más desplazamientos ni retrasos rotacionales, y los datos se transfieren con el ancho de banda máximo que permite el disco. Así la asignación contigua es fácil de implementar y tiene un alto rendimiento.

Por desgracia, la asignación contigua también tiene una importante desventaja: con el tiempo, el disco se fragmenta. Para ver cómo sucede este, examinaremos la Figura 6-12(b). Aquí se han borrado dos ficheros C y F. Cuando se elimina un fichero, sus bloques se liberan, dejando una serie de bloques libres en el disco. El disco no se compacta de inmediato para tener un único hueco grande, pues eso requeriría copiar todos los bloques que están después del hueco, de los cuales podría haber millones. El resultado es que al final el disco se compone de ficheros y huecos, como se ilustra en la figura.

En un principio, esta fragmentación no es un problema porque es posible escribir cada fichero nuevo al final del disco, después del anterior. Sin embargo, tarde o temprano el disco se llenará y será necesario compactarlo, lo cual tiene un coste prohibitivo, o reutilizar el espacio desocupado (los huecos). Para ello es necesario mantener una lista de huecos, lo cual es factible. Sin embargo, cuando se va a crear un fichero nuevo se hace necesario conocer su tamaño final para escoger un hueco del tamaño correcto en el cual colocarlo.

Imaginemos las consecuencias de tal diseño. El usuario inicial un editor de texto o procesador de texto para escribir un documento. Lo primero que pregunta el programa es cuántos bytes va a tener el fichero final, negándose a continuar si no se contesta a esa pregunta. Si el número dado al final resulta demasiado pequeño, el programa tendrá que terminar de forma prematura porque el hueco en el disco está lleno y no hay lugar para colocar el resto del fichero.

Si el usuario trata de evitar este problema dando como tamaño final una cifra muy grande, poco realista, digamos 100 MB, el editor podría ser incapaz de hallar un hueco tan grande, en cuyo caso anunciaría que no puede crearse el fichero. Desde luego, el usuario podría reiniciar el programa y contestar 50 MB, y seguir así hasta hallar un hueco apropiado. De cualquier manera, es poco probable que este esquema satisfaga a los usuarios.

No obstante, existe una situación en la que la asignación contigua es factible y, de hecho, muy utilizada: en los CD-ROMs. Aquí se conoce con antelación el tamaño de todos los ficheros, y esos tamaños no cambiarán durante el uso posterior del sistema de ficheros del CD-ROM. En una sección posterior del capítulo estudiaremos el sistema de ficheros más común utilizado en los CD-ROMs.

Como mencionamos en el capítulo 1, la historia a menudo se repite en el campo de la informática, conforme surgen nuevas generaciones de tecnología. La asignación contigua se utilizó en los sistemas de ficheros de disco magnético hace años debido a su sencillez y su gran rapidez (la amabilidad para con el usuario no contaba mucho entonces). Luego se desechó la idea por la molestia de tener que especificar el tamaño final de los ficheros en el momento de crearlos. Sin embargo, con la llegada de los CD-ROMs, DVDs, y otros medios ópticos en los que se escribe una sola vez, de repente los ficheros contiguos vuelven a ser una buena idea. Por ello, es importante estudiar los sistemas e ideas antiguos que eran claros y sencillos desde el punto de vista conceptual, por que podrían ser aplicables a sistemas futuros de formas sorprendentes.

Asignación por lista enlazada

El segundo método para almacenar ficheros consiste en mantener cada uno como una lista enlazada de bloques de disco, como se muestra en la Figura 6-13. La primera palabra de cada bloque se utiliza como puntero al siguiente bloque del fichero. El resto del bloque es para datos.

A diferencia de la asignación contigua, con este método pueden utilizarse todos los bloques del disco. No se pierde espacio por fragmentación del disco (sólo por fragmentación interna en el último bloque). Además basta que la entrada del directorio correspondiente al fichero guarde la dirección de disco del primer bloque. El resto de bloques pueden localizarse a partir de ese punto.

Por otra parte, aunque la lectura secuencial de un fichero es directa, el acceso aleatorio es lento en el extremo. Para llegar al bloque n , el sistema operativo tienen que comenzar por el principio y leer los $n - 1$ bloques que lo preceden, uno por uno. Es evidente que tantas lecturas hacen demasiado lento el acceso.

Además, la cantidad de datos almacenados en un bloque ya no es una potencia de 2 porque el puntero ocupa unos cuantos bytes. Aunque no es fatal tener un tamaño peculiar, merma la eficiencia porque muchos programas leer y escriben en bloques cuyo tamaño es una potencia de 2. Si los primeros bytes de cada bloque están ocupados por un puntero al siguiente bloque, las lecturas de bloques enteros requieren obtener y concatenar información de dos bloques de disco, lo cual genera un gasto adicional debido a la copia.

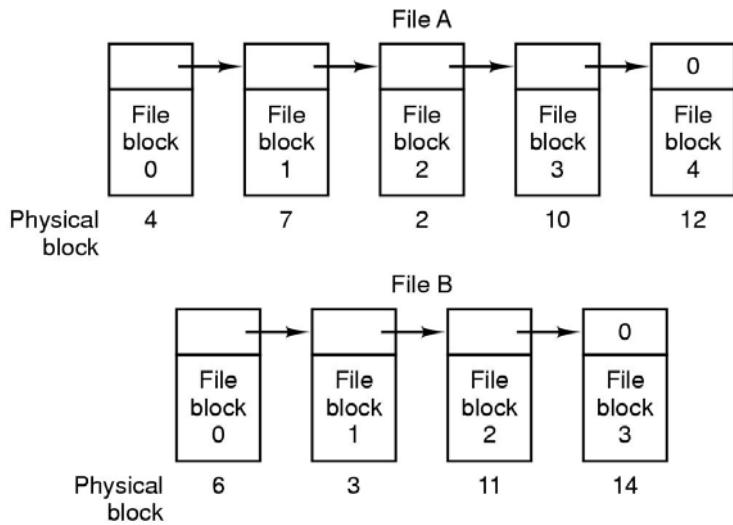


Figura 6-13. Almacenamiento de un fichero como una lista enlazada de bloques del disco.

Las dos desventajas de la asignación por lista enlazada pueden eliminarse sacando el puntero de cada bloque del disco y colocándolo en una tabla en la memoria. La Figura 6-14 muestra cómo se vería la tabla para el ejemplo de la Figura 6-13. En ambas figuras tenemos dos ficheros. El fichero *A* ocupa los bloques de disco 4, 7, 2, 10 y 12, en ese orden, y el fichero *B* ocupa los bloques 6, 3, 11 y 14, en ese orden. Con la tabla de la Figura 6-14, podemos partir del bloque 4 y seguir la cadena hasta el final. Lo mismo puede hacerse partiendo del bloque 6. Ambas cadenas terminan con un marcador especial (por ejemplo -1) que no es un número de bloque válido. Una tabla así en la memoria principal se denomina una **FAT (File Allocation Table; tabla de asignación de ficheros)**.

Con esta organización los bloques pueden llenarse ahora completamente con datos. Además el acceso aleatorio es mucho más fácil. Aunque todavía es necesario seguir la cadena para hallar un desplazamiento dado dentro del fichero, la cadena está por completo en la memoria, así que puede seguirse sin tener que leer el disco. Al igual que con el método anterior, basta con que la entrada del directorio guarde un único entero (el número del primer bloque) para poder localizar todos los bloques, sin importar qué tamaño tenga el fichero.

La desventaja primordial de este método es que, para que funcione, toda la tabla debe estar en la memoria todo el tiempo. Con un disco de 20 GB y bloques de 1 KB, la tabla necesita 20 millones de entradas, una para cada uno de los 20 millones de bloques del disco. Cada entrada debe tener un mínimo de 3 bytes, y si se desea agilizar la consulta se necesitan 4 bytes. Por tanto, la tabla ocupará 60 u 80 MB de memoria principal todo el tiempo, dependiendo de si el sistema está optimizado desde el punto de vista del espacio o del tiempo. Es concebible colocar la tabla en memoria paginable, pero de todos modos ocuparía una gran cantidad de memoria virtual y de espacio en disco, además de generar tráfico de paginación adicional.

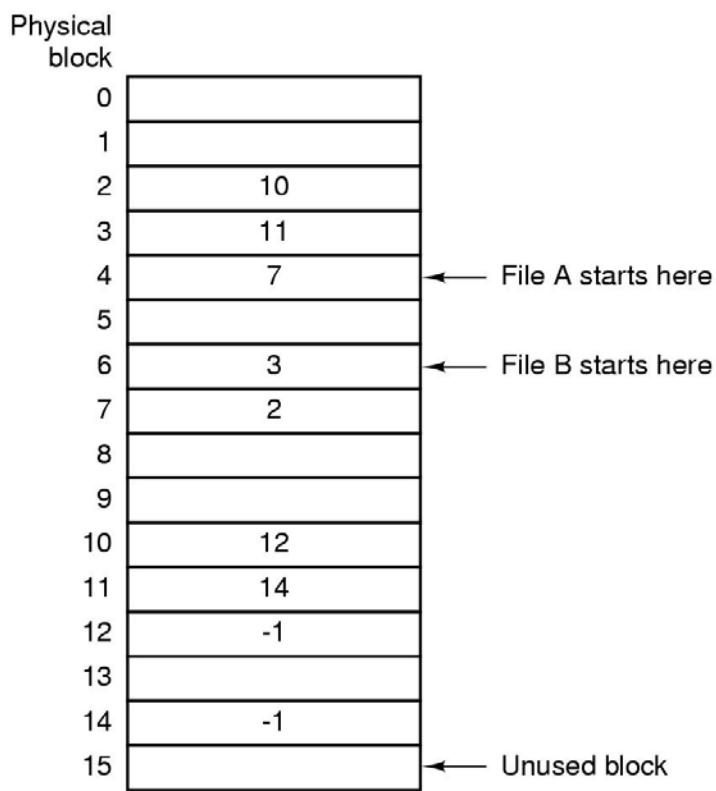


Figura 6-14. Asignación enlazada empleando una tabla de asignación de ficheros en la memoria principal.

i-nodos

Nuestro último método para llevar el control de qué bloques pertenecen a qué ficheros consiste en asociar a cada fichero una estructura de datos llamada **i-nodo (nodo de índice)**, que contiene los atributos y direcciones en disco de los bloques del fichero. En la Figura 6-15 se muestra un ejemplo sencillo. Dado el i-nodo, es posible hallar todos los bloques del fichero. La gran ventaja de este esquema respecto de las listas enlazadas empleando una tabla en la memoria es que el i-nodo sólo tiene que estar en memoria cuando el fichero correspondiente está abierto. Si cada i-nodo ocupa n bytes y no puede haber más de k ficheros abiertos al mismo tiempo, la memoria total ocupada por el array que contiene los i-nodos de los ficheros abiertos es de sólo kn bytes. Únicamente es necesario reservar esa cantidad de espacio.

Este array suele ser mucho más pequeño que el espacio ocupado por la tabla de ficheros que describimos en la sección anterior. La razón es sencilla. La tabla para contener la lista enlazada de todos los bloques del disco tiene un tamaño proporcional al disco mismo. Si el disco tiene n bloques, la tabla necesita n entradas. A medida que aumenta el tamaño de los discos, el tamaño de esta tabla crece en proporción lineal. En contraste, el esquema de i-nodos requiere un array en la memoria cuyo tamaño sea proporcional al número máximo de ficheros que pueden estar abiertos a la vez. No importa si el disco es de 1, de 10 o de 100 GB.

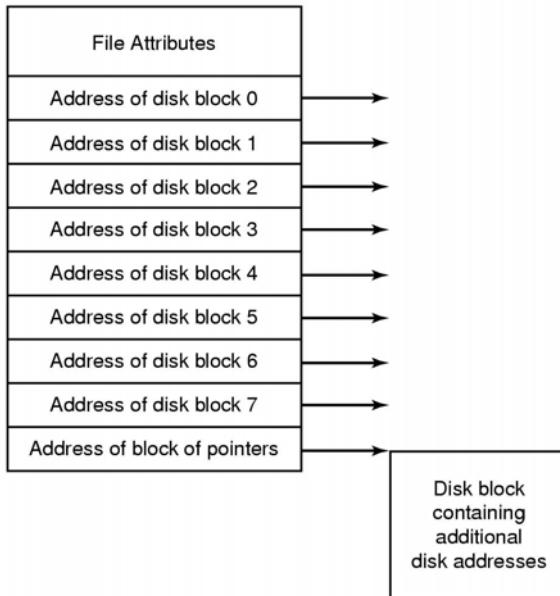


Figura 6-15. Ejemplo de i-nodo.

Un problema de los i-nodos es que si cada uno tiene espacio para un número fijo de direcciones de disco, ¿qué sucede cuando un fichero crece más allá de ese límite? Una solución es reservar la última dirección de disco no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloques de disco, como se muestra en la Figura 6-15. Algo aún más avanzado sería tener dos o más de esos bloques llenos de direcciones en disco o incluso bloques de disco que apunten a otros bloques de disco llenos de direcciones. Volveremos a ver los i-nodos cuando estudiemos UNIX más adelante.

6.3.3 Implementación de directorios

Para poder leer de un fichero, es preciso abrirlo primero. Cuando se abre un fichero, el sistema operativo utiliza el nombre de camino proporcionado por el usuario para localizar la entrada de directorio. Ésta proporciona la información necesaria para hallar los bloques de disco. Dependiendo del sistema, esta información podría ser la dirección en disco de todo el fichero (asignación contigua), el número del primer bloque (ambos esquemas de lista enlazada) o el número del i-nodo. En todos los casos. La función principal del sistema de directorios es establecer una correspondencia entre el nombre de fichero ASCII y la información necesaria para localizar los datos.

Un aspecto estrechamente relacionado es dónde deben guardarse los atributos. Todo sistema de ficheros mantiene atributos de los ficheros, como su propietario y tiempo de creación, y deben almacenarse en algún lado. Una posibilidad obvia es guardarlos directamente en la entrada de directorio. Muchos sistemas hacen precisamente esto. En la Figura 6-16(a) se muestra esta opción. En este sencillo diseño, un directorio es una lista de entradas de tamaño fijo, una por fichero, que contiene un nombre de fichero (de longitud fija), una estructura con los atributos del fichero y una o más direcciones en disco (hasta algún máximo) que indican dónde están los bloques de disco.

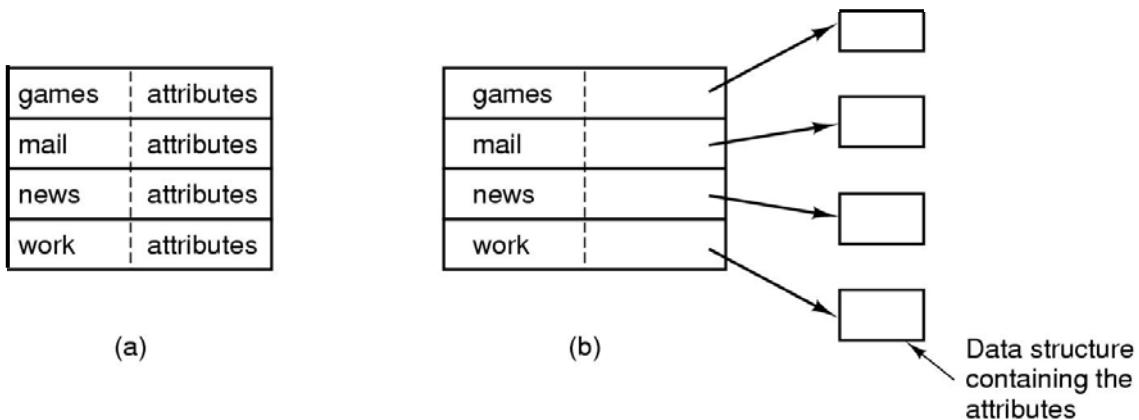


Figura 6-16. (a) Directorio sencillo que contiene entradas de tamaño fijo con las direcciones en disco y los atributos de cada fichero. (b) Directorio en el que cada entrada sólo hace referencia a un i-nodo.

En los sistemas que utilizan i-nodos, otra posibilidad para almacenar los atributos es en los i-nodos, en lugar de en las entradas de directorio. En este caso, la entrada de directorio puede ser más corta: tan sólo un nombre de fichero y un número de i-nodo. Este enfoque se ilustra en la Figura 6-16(b). Como veremos más adelante, este método tiene ciertas ventajas respecto a colocar los atributos en la entrada del directorio. Los dos enfoques que se muestran en la Figura 6-16 corresponden a MS-DOS/Windows y UNIX, respectivamente, como veremos en una sección posterior del capítulo.

Hasta ahora hemos supuesto que los ficheros tienen nombres cortos de longitud fija. En MS-DOS los ficheros tienen un nombre base de uno a ocho caracteres y una extensión opcional de uno a tres caracteres. En UNIX versión 7, los nombres de fichero tenían de uno a 14 caracteres, incluyendo cualquier extensión. Sin embargo casi todos los sistemas operativos modernos reconocen nombres de fichero más largos, de longitud variable. ¿Cómo pueden implementarse?

El método más sencillo es fijar un límite para la longitud del nombre de fichero, por lo regular 255 caracteres, y luego utilizar uno de los diseños de la Figura 6-16 con 255 caracteres reservados para cada nombre de fichero. Este método es sencillo, pero desperdicia mucho espacio de directorio, porque pocos ficheros tienen nombres tan largos. Por razones de eficiencia, conviene utilizar una estructura diferente.

Una alternativa es abandonar la idea de que todas las entradas de directorio tienen el mismo tamaño. Con este método, cada entrada de directorio contiene una porción fija, que por lo regular comienza con la longitud de la entrada, seguida de datos con un formato fijo, que normalmente incluyen al propietario, la hora en que se creó, información de protección y otros atributos. Este encabezado de longitud fija va seguido del nombre del fichero en sí, que puede tener cualquier longitud, como se muestra en la Figura 6-17(a) en formato big-endian (por ejemplo SPARC). En este ejemplo tenemos tres ficheros, *project-budget*, *personnel* y *foo*. Cada nombre de fichero termina con un carácter especial (normalmente el carácter que tiene número ASCII 0) que se representa en la figura con una cruz encerrada en un cuadrado. Para que cada entrada de directorio pueda comenzar en una frontera de palabra, cada nombre de fichero se rellena hasta un número entero de palabras, lo cual se indica con rectángulos sombreados en la figura.

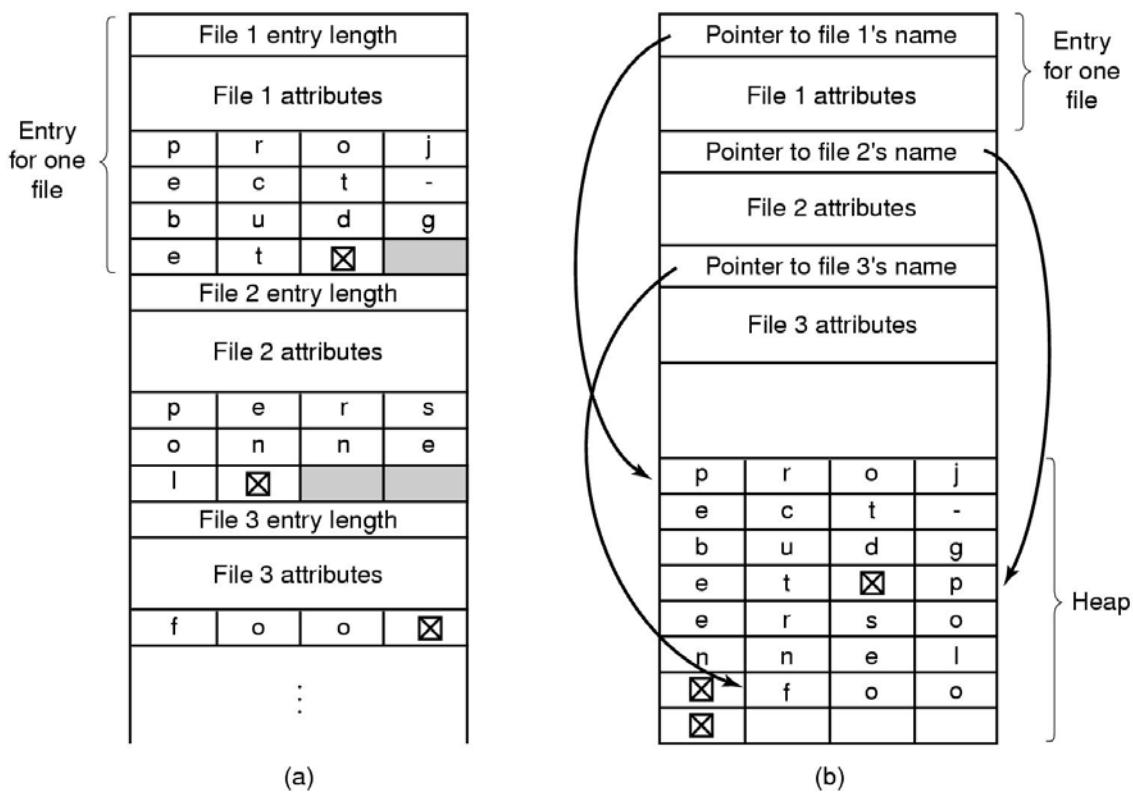


Figura 6-17. Dos formas de manejar nombres de fichero largos en un directorio. (a) Incorporados. (b) en un heap.

Una desventaja de este método es que cuando se elimina un fichero queda en el directorio un hueco de tamaño variable en el que tal vez no quiepa la entrada del siguiente fichero que se cree. Este problema es el mismo que vimos con los ficheros contiguos en disco, sólo que ahora sí es factible compactar el directorio porque está por completo en la memoria. Otro problema es que una sola entrada de directorio podría cruzar fronteras de página, por lo que podría presentarse una falta de página durante la lectura de un nombre de fichero.

Otra forma de manejar los nombres de longitud variable es hacer que todas las entradas de directorio propiamente dichas sean de longitud fija y mantener los nombres de fichero juntos en un heap al final del directorio, como se muestra en la Figura 6-17(b). Este método tienen la ventaja de que cuando se elimina una entrada siempre cabrá ahí la del siguiente fichero que se cree. Desde luego, hay que administrar el heap, y sigue existiendo la posibilidad de que se presenten fallos de página durante el procesamiento de nombres de fichero. Una ventaja menor aquí es que ya no es necesario que los nombres de fichero comiencen en fronteras de palabra, de modo que no se requieren caracteres de relleno en la Figura 6-17(b), como sucedía en la Figura 6-17(a).

En todos los diseños que hemos visto hasta ahora, los directorios se exploran mediante una búsqueda lineal, desde el principio hasta el final, cada vez que se busca un nombre de fichero. En el caso de directorios extremadamente largos, las búsquedas lineales pueden ser lentas. Una forma de acelerar la búsqueda es utilizar una tabla hash en cada directorio. Digamos que el tamaño de la tabla es n . Para introducir un nombre de fichero, el nombre se transforma en un valor entre 0 y $n - 1$, por ejemplo, dividiéndolo entre n y utilizando el resto. O bien, las palabras que integran el nombre del fichero pueden sumarse y dividir la cantidad obtenida entre n , o algo similar.

De cualquier modo, se examina la entrada de la tabla correspondiente al código hash. Si está desocupada, se coloca en ella un puntero a la entrada del fichero. Las entradas con los atributos de los ficheros se encuentran después de la tabla hash. Si ya se está utilizando esa entrada en la tabla hash, se construye una lista enlazada, poniendo como primer elemento esa entrada de la tabla, que encadena todas las entradas que tienen el mismo valor hash.

La consulta de un fichero sigue el mismo procedimiento. El nombre del fichero se transforma en un índice para seleccionar una entrada de la tabla hash. Se examinan todas las entradas de la cadena cuya primer elemento está en esa entrada, para ver si está presente el nombre del fichero. Si el nombre no está en la cadena, quiere decir que el fichero no está en el directorio.

La utilización de una tabla hash tiene la ventaja de que las búsquedas son mucho más rápidas, pero tiene la desventaja de que la administración es más complicada. En realidad sólo es lógico utilizar una tabla hash en los sistemas en los que se espera que los directorios normalmente contengan cientos o miles de ficheros.

Una forma completamente distinta de acelerar las búsquedas en directorios grandes es poner en una caché los resultados de la búsqueda. Antes de iniciar una búsqueda, se verifica si el nombre de fichero está en la caché. Si está, podrá localizarse rápido, evitando así una búsqueda larga. Claro que la utilización de caché sólo resulta útil si el número de ficheros que se buscan más es relativamente reducido.

6.3.4 Ficheros compartidos

Cuando varios usuarios colaboran en un proyecto, es normal que necesiten compartir ficheros. Por ello, en muchos casos conviene que un fichero compartido aparezca al mismo tiempo en diferentes directorios que pertenecen a usuarios distintos. La Figura 6-18 muestra otra vez el sistema de ficheros de la Figura 6-9, sólo que ahora uno de los ficheros de *C* también está presente en uno de los directorios de *B*. La conexión entre el directorio de *B* y el fichero compartido se denomina un **enlace** (*link*). El sistema de ficheros en sí es ahora un **DAG** (*Direct Acyclic Graph*; grafo acíclico dirigido), no un árbol.

Compartir ficheros es conveniente, pero también presenta problemas. Para empezar, si los directorios contienen de verdad direcciones de disco, tendrá que crearse una copia de ellas en el directorio de *B* cuando se enlace el fichero. Si después *B* o *C* hacen crecer al fichero, los nuevos bloques aparecerán sólo en el directorio del usuario que modificó el fichero. Los cambios no serán visibles para el otro usuario, frustrando así el propósito de compartir los ficheros.

Hay dos formas de resolver este problema. En la primera, los bloques de disco no se listan en los directorios, sino en una pequeña estructura de datos asociada con el fichero mismo. Los directorios apuntarían entonces sólo a la pequeña estructura de datos. Éste es el método que se utiliza en UNIX (donde la pequeña estructura de datos es el i-nodo).

En la segunda solución, *B* se enlaza con uno de los ficheros de *C* pidiendo al sistema que cree un fichero nuevo, de tipo LINK, y lo introduzca en el directorio de *B*. El nuevo fichero sólo contiene el nombre de camino del fichero con el cual está enlazado. Cuando *B* lee del fichero enlazado, el sistema operativo percibe que el fichero que se está leyendo es de tipo LINK, busca el nombre del fichero y lo lee. Este método se llama **enlace simbólico**.

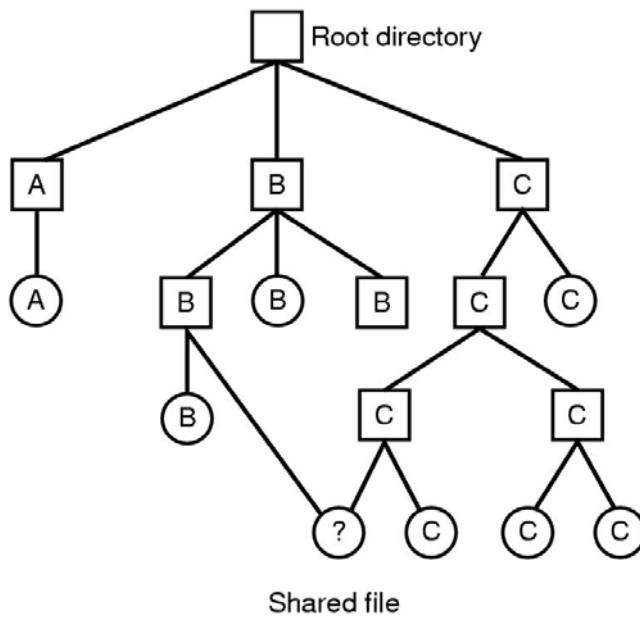


Figura 6-18. Sistema de ficheros que contiene un fichero compartido.

Todos estos métodos tienen desventajas. Con el primero, en el momento en que *B* se enlaza con el fichero compartido, el i-nodo indica que el propietario del fichero es *C*. La creación de un enlace no hace que el fichero cambie de dueño (vea la Figura 6-19), pero sí incrementa el contador de enlaces en el i-nodo para que el sistema sepa cuantas entradas de directorio están apuntando en la actualidad al fichero.

Si más adelante *C* trata de eliminar el fichero, el sistema se enfrenta a un problema. Si elimina el fichero y libera el i-nodo, *B* tendrá una entrada de directorio que apunta a un i-nodo que no es válido. Si después el i-nodo se reasigna a otro fichero, el enlace de *B* apuntará al fichero equivocado. El sistema puede ver, por la cuenta contenida en el i-nodo, que el fichero todavía está en uso, pero no tiene forma de hallar todas las entradas de directorio para ese fichero a fin de borrarlas. No es posible almacenar punteros a los directorios en el i-nodo porque podría haber un número ilimitado de directorios.

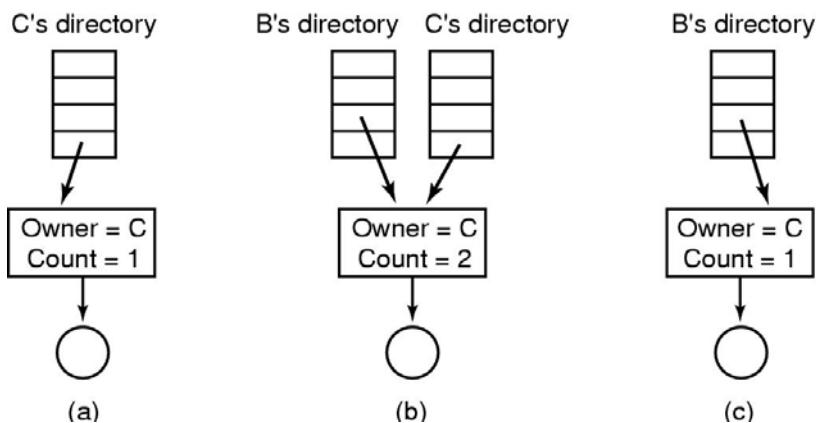


Figura 6-19. (a) La situación antes del enlace. (b) Una vez que se ha creado el enlace. (c) Después de que el propietario original elimina el fichero.

Lo único que puede hacerse es eliminar la entrada de directorio de *C*, pero dejar intacto el i-nodo, con su contador a 1, como se muestra en la Figura 6-19(c). Ahora tenemos una

situación en la que B es el único usuario que tiene una entrada de directorio para un fichero propiedad de C . Si el sistema realiza contabilidad o tiene cuotas, se seguirá cobrando a C por el fichero hasta que B decida borrarlo, si alguna vez lo hace, pues en ese momento el contador se hará cero y el fichero se borrará.

Con enlaces simbólicos no se presenta este problema porque sólo el verdadero propietario tiene un puntero al i-nodo. Los usuarios que se han enlazado con el fichero sólo tienen nombres de camino, no punteros a i-nodos. Cuando el propietario elimina el fichero, éste se destruye. Los intentos subsiguientes de utilizar el fichero por medio de un enlace simbólico fracasarán cuando el sistema no pueda localizar el fichero. La eliminación de un enlace simbólico no afecta en absoluto al fichero.

El problema con los enlaces simbólicos es el procesamiento adicional requerido. Es preciso leer el fichero que contiene el camino, el cual debe entonces analizarse y seguirse, componente por componente, hasta llegar al i-nodo. Toda esta actividad podría requerir una cantidad considerable de accesos adicionales al disco. Además, se necesita un i-nodo adicional por cada enlace simbólico, y también un bloque de disco adicional para almacenar el camino, aunque si el nombre de camino es corto, el sistema podría almacenarlo en el mismo i-nodo, como optimización. Los enlaces simbólicos tienen la ventaja de que pueden servir para enlazarse con ficheros de máquinas que puede estar en cualquier lugar del mundo, con sólo proporcionar la dirección de red de la máquina en la que reside el fichero, además de su camino en esa máquina.

Hay otro problema que provocan los enlaces, simbólicos o no. Si se permiten enlaces, los ficheros pueden tener dos o más caminos. Los programas que en el arranque se sitúan en un directorio dado y buscan todos los ficheros en ese directorio y sus subdirectorios, localizarán un fichero enlazado múltiples veces. Por ejemplo, un programa de backup en cinta de todos los ficheros de un directorio y sus subdirectorios podría crear varias copias de un fichero enlazado. Además, si la cinta se copia después en otra máquina, a menos que el programa de backup sea inteligente, el fichero enlazado se copiará dos veces en el disco, en lugar de estar enlazado.

6.3.5 Administración del espacio en disco

Los ficheros generalmente se almacenan en disco, por lo que la administración del espacio en disco es de primordial interés para los diseñadores de sistemas de ficheros. Pueden adoptarse dos estrategias generales para almacenar un fichero de n bytes: asignar n bytes consecutivos de espacio en disco, o dividir el fichero en varios bloques (no necesariamente) contiguos. Las ventajas y desventajas son las mismas que en los sistemas de administración de memoria, entre segmentación pura y paginación.

Como hemos visto, almacenar un fichero como una sucesión contigua de bytes tiene el problema obvio de que, si un fichero crece, es probable que tenga que pasarse a otro lugar del disco. El mismo problema se tiene con los segmentos en la memoria, sólo que cambiar de lugar un segmento en la memoria es una operación relativamente rápida, en comparación con cambiar un fichero de una posición a otra en el disco. Por ello, casi todos los sistemas de ficheros dividen los ficheros en bloques de tamaño fijo que no tienen que ser adyacentes.

Tamaño de bloque

Una vez que se ha decidido almacenar ficheros en bloques de tamaño fijo, surge la pregunta de qué tamaño debe de tener un bloque. Dada la forma en que están organizados los discos, el sector, la pista y el cilindro son candidatos obvios para ser la unidad de asignación (aunque todos estos tamaños dependen del dispositivo, lo cual es una desventaja). En un sistema con paginación, el tamaño de página también es un contendiente importante.

Tener una unidad de asignación grande, digamos un cilindro, implica que todos los ficheros, incluso aunque sólo tengan un byte, ocuparán cilindros enteros. Estudios efectuados (Mullender y Tanenbaum, 1984) han mostrado que la mediana del tamaño de los ficheros en los entornos UNIX es de aproximadamente 1 KB, así que asignar un bloque de 32 KB a cada fichero desperdiciaría $31/32 = 97\%$ del espacio total en el disco.

Por otra parte, el uso de una unidad de asignación pequeña implica que cada fichero va a constar de varios bloques. Leer cada bloque requiere por lo regular un desplazamiento del brazo y una latencia rotacional, por lo que resulta lenta la lectura de un fichero integrado por muchos bloques pequeños.

Por ejemplo, consideremos un disco que tiene 131072 bytes por pista, un tiempo de rotación de 8,33 ms y un tiempo medio de desplazamiento del brazo de 10 ms. El tiempo requerido en milisegundos para leer un bloque de k bytes será entonces la suma de los tiempos de desplazamiento, latencia rotacional y transferencia:

$$10 + 4,165 + (k/131072) \times 8.33$$

La curva continua de la Figura 6-20 muestra la tasa de datos de un disco así en función del tamaño del bloque. Para calcular el aprovechamiento del espacio, necesitamos suponer algo acerca del tamaño medio de los ficheros. Una medición reciente efectuada en el departamento académico del autor, que tiene 1000 usuarios y más de un millón de ficheros UNIX en disco, da una mediana del tamaño de los ficheros de 1680 bytes, lo que implica que la mitad de los ficheros tienen menos de 1680 bytes, y la otra mitad tiene un tamaño mayor. Por cierto, la mediana es una métrica mejor que la media porque un número muy pequeño de ficheros puede influir de forma considerable en la media, pero no en la mediana. (De hecho, la media es de 10845 bytes, debido en parte a unos cuantos manuales de hardware de 100 MB que por casualidad están en línea.) Por sencillez, supongamos que todos los ficheros son de 2 KB, lo cual da pie a la curva de trazos punteados de la Figura 6-20 que describe la eficiencia espacial del disco.

Las dos curvas pueden interpretarse como sigue. El tiempo para tener acceso a un bloque está dominado por completo por el tiempo de posicionamiento del brazo y la latencia rotacional. Entonces, dado que va a costar 14 ms tener acceso a un bloque, cuantos más datos se obtengan, mejor. Por tanto la tasa de datos crece al aumentar el tamaño del bloque (hasta que las transferencias tardan tanto que el tiempo de transferencia comienza a dominar). Con bloques pequeños que son potencias de 2 y ficheros de 2 KB, no se desperdicia espacio en los bloques, en cambio, con ficheros de 2 KB y bloques de 4 KB o mayores, se desperdicia algo de espacio de disco. En realidad, pocos ficheros son múltiplos del tamaño de bloque de disco, por lo que siempre se desperdicia algún espacio en el último bloque de un fichero.

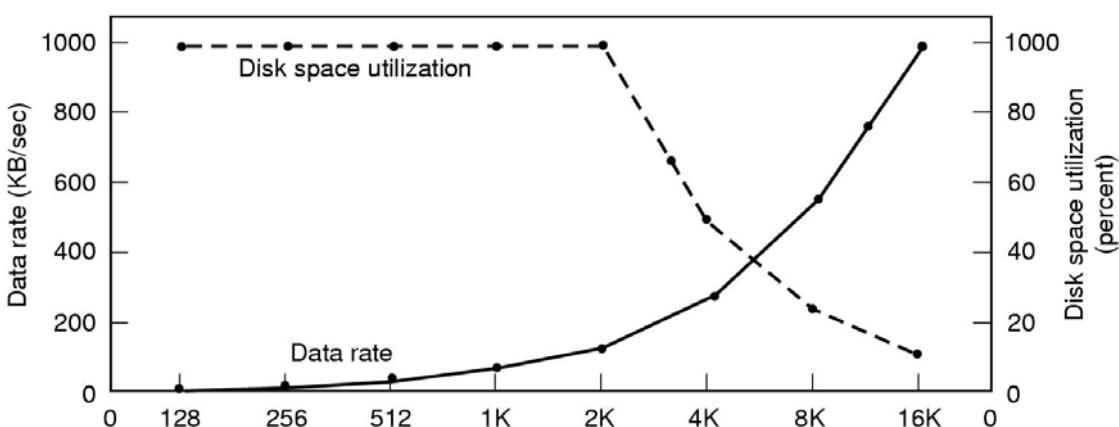


Figura 6-20. La curva (escala de la izquierda) da la tasa de datos de un disco. La curva discontinua (escala de la derecha) da la eficiencia espacial del disco. Todos los ficheros son de 2 KB.

Lo que muestran las curvas, empero, es que el rendimiento y el aprovechamiento del espacio están en conflicto de manera inherente. Los bloques pequeños son malos para el rendimiento pero buenos para el aprovechamiento del espacio en disco. Se requiere un tamaño intermedio de compromiso. Con estos datos, 4 KB podría ser una buena opción, pero algunos sistemas operativos tomaron su decisión hace mucho tiempo, cuando los parámetros de disco y los tamaños de fichero eran diferentes. En el caso de UNIX, es común utilizar 1 KB. En el caso de MS-DOS, el tamaño de bloque puede ser cualquier potencia de 2, desde 512 bytes hasta 32 KB, pero está determinado por el tamaño del disco y por factores que no tienen relación con estos argumentos. (El número máximo de bloques en una partición de disco es 2^{16} , lo que obliga a usar bloques grandes en discos grandes.)

En un experimento para ver si el uso de ficheros en Windows NT mostraba diferencias apreciables respecto al uso de ficheros en UNIX, Vogels hizo mediciones de ficheros en la Universidad de Cornell (Vogels, 1999). Observó que la utilización de los ficheros en NT es más complicada que en UNIX, y escribió:

Si tecleamos unos cuantos caracteres en el editor de texto notepad, al guardar esto en un fichero se generan 26 llamadas al sistema, incluyendo tres intentos fallidos de apertura, 1 sobrescritura de fichero y 4 secuencias de apertura y cierre adicionales.

No obstante, observó una mediana (ponderada por el uso) del tamaño de los ficheros recién escritos de 2,3 KB y de ficheros leídos y escritos de 4,2 KB. Considerando el hecho de que Cornell realiza computación científica a mucha mayor escala que la institución del autor, y la diferencia en la técnica de medición (estática frente a dinámica), los resultados son razonablemente congruentes con una mediana del tamaño de fichero alrededor de 2 KB.

Control de bloques libres

Una vez escogido un tamaño de bloque, la siguiente cuestión es cómo llevar el control de los bloques libres. Se usan dos métodos principalmente, los cuales se muestran en la Figura 2-21. El primero consiste en usar una lista enlazada de bloques de disco, en cada uno de los cuales se guardan tantos números de bloques de disco como quepan. Con bloques de 1 KB y números de bloque de 32 bits, cada bloque de la lista de bloques libres contendrá los números de 255 bloques libres (se necesita una entrada para el puntero al siguiente bloque). Un disco de 16 GB necesitará una lista de bloques libres de 16794 bloques como máximo para contener los 2^{24} números de bloque. Es común que se utilicen bloques libres para almacenar la lista libre.

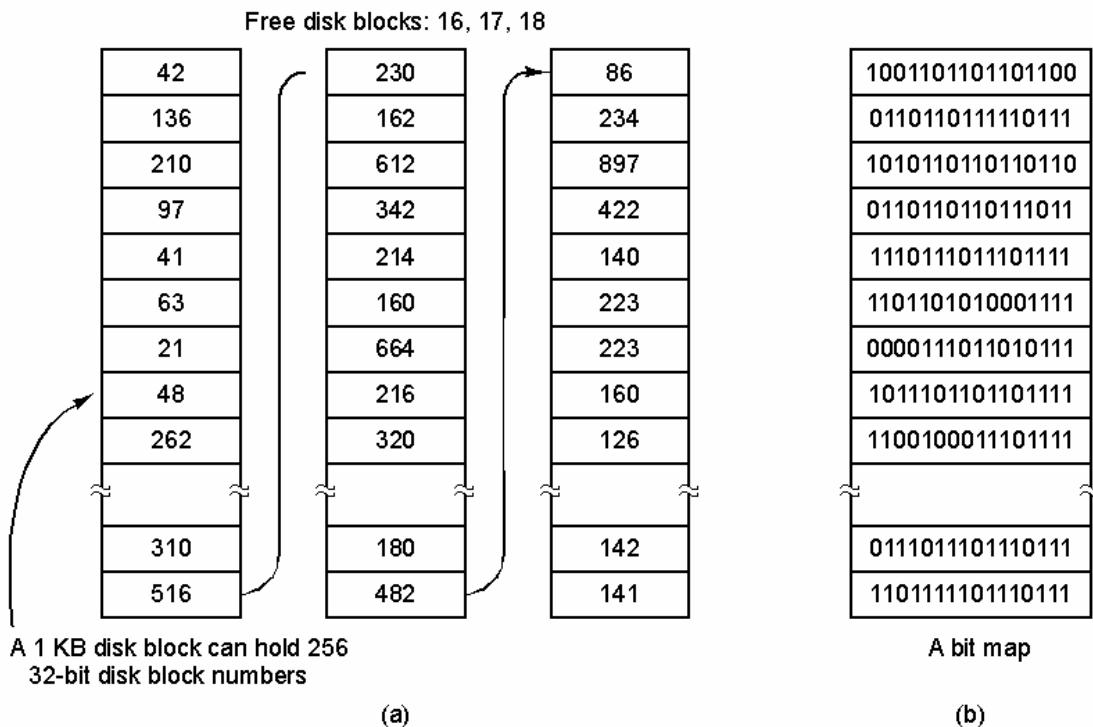


Figura 6-21. (a) Almacenamiento de la lista de bloques libres en una lista enlazada. (b) Un mapa de bits.

La otra técnica de administración del espacio libre es el mapa de bits. Un disco con n bloques requiere un mapa de bits con n bits. Los bloques libres se representan con unos en el mapa, y los bloques asignados con ceros (o viceversa). Un disco de 16 GB tiene 2^{24} bloques de 1KB y por tanto requiere 2^{24} bits para el mapa, lo cual ocupa 2048 bloques. No es sorprendente que el mapa de bits requiera menos espacio, puesto que utiliza 1 bit por bloque, en comparación con 32 bits si se utiliza el modelo de la lista enlazada. Sólo si el disco está lleno (es decir, si tiene pocos bloques libres) el esquema de lista enlazada requerirá menos bloques que el mapa de bits. Por otra parte, si hay muchos bloques libres, podrán pedirse prestados algunos de ellos para almacenar la lista libre, sin pérdida de capacidad del disco.

Si se usa el método de la lista de bloques libres, sólo es preciso mantener un bloque de punteros en la memoria principal. Cuando se crea un fichero, los bloques que necesita se toman del bloque de punteros. Cuando este bloque se agota, se lee del disco un nuevo bloque de punteros. De forma similar, cuando se borra un fichero, sus bloques se liberan y se añaden al bloque de punteros que está en la memoria principal. Si este bloque se llena, se escribe en el disco.

En ciertas circunstancias, este método da pie a una cantidad innecesaria de operaciones de E/S al disco. Consideremos la situación de la Figura 6-22(a), donde el bloque de punteros que está en la memoria sólo tiene espacio para dos entradas más. Si se libera un fichero de tres bloques, el bloque de punteros se desbordará y tendrá que escribirse en el disco, produciendo la situación de la Figura 6-22(b). Si ahora se escribe un fichero de tres bloques, será necesario traer otra vez a la memoria el bloque lleno de punteros, y volveremos a la situación de la Figura 6-22(a). Si el fichero de tres bloques recién escrito era un fichero temporal, cuando se liberen sus bloques se requerirá otra escritura en disco para guardar el bloque lleno de punteros. En síntesis, cuando el bloque de punteros está casi vacío, una serie de ficheros temporales efímeros puede generar mucha E/S de disco.

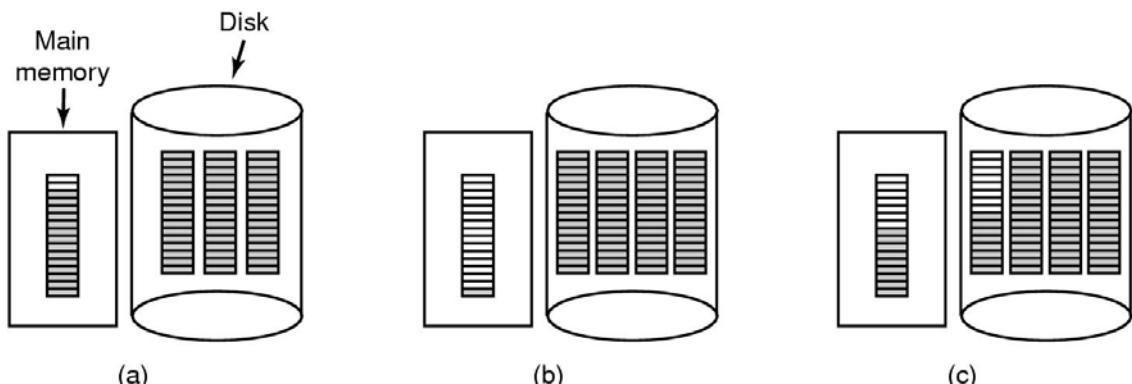


Figura 6-22. (a) Un bloque en la memoria casi lleno de punteros a bloques de disco libres y tres bloques de punteros en el disco. (b) Resultado de liberar un fichero de tres bloques. (c) Estrategia alternativa para manejar los tres bloques liberados. Las entradas sombreadas representan punteros a bloques de disco libres.

Un método alternativo que evita la mayor parte de esa E/S de disco consiste en dividir el bloque de punteros lleno. Así, en lugar de pasar de la Figura 6-22(a) a la Figura 6-22(b), pasamos de la Figura 6-22(a) a la Figura 6-22(c) cuando se liberan tres bloques. Ahora el sistema puede manejar una serie de ficheros temporales sin tener que efectuar E/S de disco. Si se llena el bloque que está en la memoria, se escribe en el disco y el bloque a medio llenar que estaba en el disco se pasa a memoria. Lo que se busca aquí es mantener llenos la mayoría de los bloques de punteros que están en el disco (para reducir al mínimo el consumo de disco), pero mantener medio lleno el que está en la memoria para poder manejar la creación y eliminación de ficheros sin que la administración de la lista libre requiera E/S de disco.

Con un mapa de bits también es posible mantener un solo bloque en la memoria, acudiendo al disco para obtener otro, sólo si el que está en la memoria se llena o se vacía. Una ventaja adicional de este método es que si se efectúa toda la asignación de bloques a ficheros a partir de un solo bloque del mapa de bits, los bloques de disco de un fichero dado estarán cercanos unos a otros, con lo que se reducirá al mínimo el movimiento del brazo del disco. Puesto que el mapa de bits es una estructura de datos fija, si el núcleo se pagina (en forma parcial), el mapa de bits podrá colocarse en la memoria virtual y sus páginas se intercambiarán a la memoria cuando se necesiten.

Cuotas de disco

Para evitar que las personas acaparen demasiado espacio de disco, los sistemas operativos multiusuario a menudo tienen un mecanismo para imponer cuotas de disco. La idea consiste en que el administrador del sistema asigne a cada usuario una porción máxima de ficheros y bloques, y que el sistema operativo cuide que los usuarios no excedan su cuota. A continuación describiremos un mecanismo típico.

Cuando un usuario abre un fichero, se localizan los atributos y direcciones de disco y se colocan en una tabla de ficheros abiertos en la memoria principal. Entre los atributos hay una entrada que indica quién es el propietario del fichero. Cualquier aumento en el tamaño del fichero se cargará a la cuota del propietario.

Una segunda tabla contiene los registros de cuota de cada uno de los usuarios que tienen un fichero abierto en ese momento, aunque alguien más haya abierto ese fichero. Esta tabla se muestra en la Figura 6-23. La tabla es un extracto de un fichero de cuotas en disco para los

usuarios cuyos ficheros están abiertos en la actualidad. Cuando se cierran todos los ficheros, el registro se escribe en el fichero de cuotas.

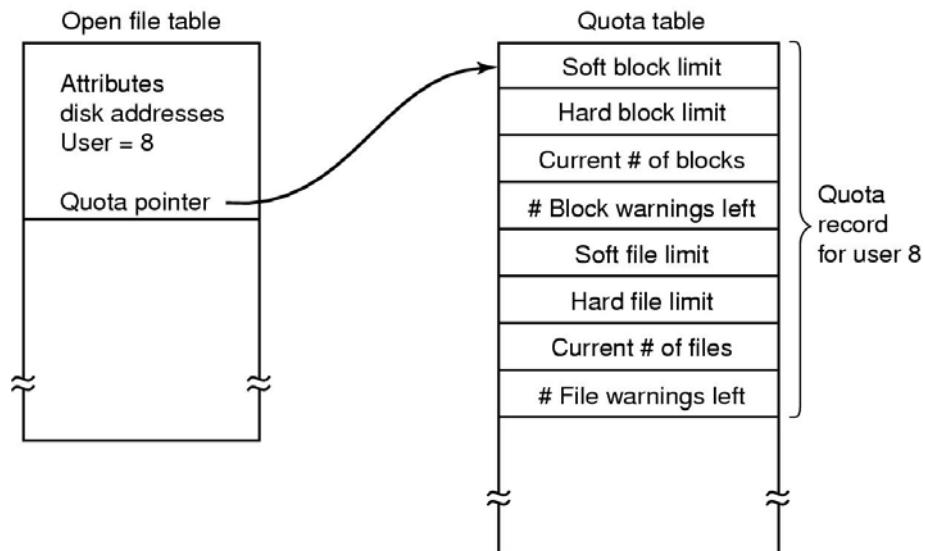


Figura 6-23. Se lleva el control de las cuotas por usuario en una tabla de cuotas.

Cuando se inserta una nueva entrada en la tabla de ficheros abiertos, se incluye en ella un puntero al registro de cuota del dueño de ese fichero, para poder hallar los diversos límites. Cada vez que se añade un bloque a un fichero, se incrementa el número total de bloques cargados al dueño, y se coteja con ambos límites, el estricto y el no estricto. El límite no estricto puede sobrepasarse, pero el estricto no. Un intento de aumentar el tamaño de un fichero cuando se ha llegado al límite de bloques estricto producirá un error. Se hacen verificaciones análogas para el número de ficheros.

Cuando un usuario intenta iniciar la sesión, el sistema examina el fichero de cuotas para ver si ese usuario ha excedido el límite no estricto de número de bloques o el de número de ficheros. Si se ha violado cualquiera de esos límites, se da una advertencia y se reduce en uno el contador de advertencias restantes. Si ese contador llega a cero, significa que el usuario ha hecho caso omiso de la advertencia demasiadas veces, y no se le permitirá iniciar la sesión. El usuario tendrá que hablar con el administrador del sistema para que se le vuelva a dar permiso para iniciar una sesión.

Este método tiene la propiedad de que los usuarios podrían rebasar sus límites no estrictos durante una sesión, siempre que se deshagan del excedente antes de terminarla. Los límites estrictos nunca pueden excederse.

6.3.6 Fiabilidad del sistema de ficheros

La destrucción de un sistema de ficheros suele ser un desastre mucho más grave que la destrucción de un ordenador. Si un ordenador queda destruido por un incendio, un rayo o una taza de café vertida en un teclado, esto es algo molesto y que nos costará mucho dinero, pero normalmente es posible comprar una máquina nueva para reemplazarlo con un mínimo de problemas. Los ordenadores personales de bajo costo pueden reemplazarse en menos de una hora con sólo acudir a una tienda (excepto en las universidades, donde la aprobación de una orden de compra requiere pasar tres comisiones, cinco firmas y 90 días).

Si se pierde de forma irremisible el sistema de ficheros de un ordenador, sea por culpa del hardware, del software o de ratas que royeron los disquetes; la recuperación de toda la información será difícil, lenta y, en muchos casos, imposible. Para las personas cuyos programas, documentos, ficheros de clientes, registros fiscales, bases de datos, planes de marketing u otros datos se han esfumado, las consecuencias pueden ser catastróficas. Aunque el sistema de ficheros no puede ofrecer protección contra la destrucción física del equipo y los medios, sí puede ayudar a proteger la información. En esta sección examinaremos algunos de los aspectos de protección del sistema de ficheros.

Los disquetes suelen ser perfectos cuando salen de la fábrica, pero algunos de sus bloques pueden estropearse durante su uso. Los discos duros a menudo tienen bloques defectuosos desde el principio: simplemente cuesta demasiado fabricarlos sin defectos. Como vimos en el capítulo 5, el controlador por lo general maneja los bloques defectuosos, sustituyendo los sectores correspondientes por sectores de repuesto que se proveen para ese fin. A pesar de esta técnica hay otros problemas de fiabilidad que consideraremos a continuación.

Copias de seguridad (Backups) (Respaldos)

La mayoría de las personas no piensa que valga la pena hacer un backup de sus ficheros debido al tiempo y al esfuerzo que requiere, hasta que un buen día su disco muere de repente y se vuelven creyentes en su lecho de muerte. En cambio, las compañías (normalmente) entienden muy bien el valor de sus datos y por lo general hacen una copia de seguridad por lo menos una vez al día, casi siempre en cinta. Las cintas modernas contienen decenas o a veces centenares de gigabytes y cuestan unos cuantos centavos de dólar por gigabyte. No obstante, hacer copias de seguridad no es tan trivial como suena, así que examinaremos ahora algunos de los problemas que ello implica.

Los backups en cinta generalmente tienen por objeto resolver uno de dos problemas potenciales:

1. Recuperarse de desastres.
2. Recuperarse de la estupidez.

Lo primero comprende hacer que el ordenador funcione otra vez después de un fallo del disco, inundación u otra catástrofe natural. En la práctica, estos sucesos no se presentan con mucha frecuencia, y es por eso por lo que muchas personas no se molestan en hacer copias de respaldo. Esas mismas personas casi nunca tienen pólizas de seguro contra incendios para sus hogares por la misma razón.

El segundo problema es que muchas veces los usuarios borran por accidente ficheros que van a necesitar después. Este problema se presenta con tanta frecuencia que cuando un fichero se “borra” en Windows, en realidad no se borra, sino que se pasa a un directorio especial, la **papelera de reciclaje**, para poder encontrarlo y restaurarlo con facilidad después. Los respaldos llevan este principio más lejos aún y permiten restaurar, a partir de cintas de respaldo viejas, ficheros que se borraron días o incluso semanas atrás.

Hacer un respaldo toma mucho tiempo y ocupa una gran cantidad de espacio, por lo que es importante hacerlo de forma eficiente y conveniente. Estas consideraciones dan pie a varias preguntas. En primer lugar hay que plantearse la pregunta: ¿debe respaldarse todo el sistema de ficheros o sólo una parte? En muchas instalaciones, los programas ejecutables (binarios) se mantienen en una parte concreta del sistema de ficheros. No es necesario respaldar esos ficheros si pueden reinstalarse utilizando los CD-ROMs del fabricante. Además, casi todos los sistemas tienen un directorio para ficheros temporales. Casi nunca tiene objeto respaldar estos ficheros. En UNIX, todos los ficheros especiales (de dispositivos de E/S) están en el directorio `/dev`. No sólo no es necesario respaldar ese directorio, sino que resultaría peligroso intentarlo porque el

programa de respaldo se bloquearía si intentase leer todos esos ficheros de principio a fin. En pocas palabras, casi siempre es recomendable respaldar sólo directorios específicos y todo su contenido, en lugar de respaldar todo el sistema de ficheros.

En segundo lugar, resulta una pérdida de tiempo respaldar ficheros que no han sufrido cambios desde la última vez en que se respaldaron, lo cual conduce a la idea de los volcados incrementales. La forma más sencilla de volcado incremental es efectuar un volcado (respaldo) completo de forma periódica, digamos cada semana o cada mes, y hacer un volcado diario sólo de los ficheros que se han modificado desde el último volcado completo. Algo mejor aún es volcar sólo los ficheros que se han modificado desde la última vez que se respaldaron. Si bien este esquema reduce al mínimo el tiempo de volcado, complica la recuperación; porque primero es preciso restaurar el último volcado completo, seguido de todos los volcados incrementales en orden inverso. Es común utilizar esquemas de volcado incremental más elaborados para facilitar la recuperación.

En tercer lugar, puesto que casi siempre se respaldan cantidades enormes de datos, podría ser una buena idea comprimirlos antes de escribirlos en cinta. Sin embargo, con muchos algoritmos de compresión un único pequeño defecto en la cinta de respaldo puede interferir con el algoritmo de descompresión e impedir la lectura de todo un fichero o incluso de toda una cinta. Por ello, hay que estudiar con detenimiento la decisión de comprimir o no el flujo de respaldo.

En cuarto lugar, es difícil respaldar un sistema de ficheros activo. Si se están añadiendo, borrando y modificando archivos y directorios durante el proceso de respaldo, la copia de seguridad resultante podría ser inconsistente. Sin embargo, como el respaldo podría tardar horas, podría ser necesario poner el sistema fuera de línea durante una buena parte de la noche para hacerlo, algo que no siempre es aceptable. Por ello, se han ideado algoritmos que toman “instantáneas” del estado del sistema de ficheros, copiando estructuras de datos cruciales, y obligando a que los cambios futuros en los ficheros y directorios copien los bloques en lugar de actualizarlos en su lugar (Hutchinson y otros, 1999). Así, el sistema de ficheros “se congela” efectivamente en el momento en que se toma la instantánea, y se puede respaldar con calma después.

En quinto y último lugar, la producción de respaldos presenta muchos problemas no técnicos en una organización. El mejor sistema de seguridad en línea del mundo podría ser inútil si el administrador del sistema guarda todas las cintas de respaldo en su oficina y la deja abierta y sin vigilancia cada vez que sale al pasillo para retirar salidas de la impresora. Lo único que tiene que hacer un espía es entrar un momento en la oficina, poner una pequeña cinta en su bolsillo y salir caminando como si nada. Adiós seguridad. Además hacer un respaldo diario no sirve de mucho si el incendio que destruye los ordenadores quema también todas las cintas de respaldo. Por ello dichas cintas deben conservarse en otro edificio, aunque esto representa más riesgos de seguridad. En Nemeth y otros (2000) se presenta un tratamiento exhaustivo de estos y otros problemas prácticos de administración. A continuación examinaremos sólo los aspectos técnicos de la generación de respaldos del sistema de ficheros.

Pueden adoptarse dos estrategias para volcar un disco en cinta: un volcado físico o un volcado lógico. Un **volcado físico** comienza por el bloque 0 del disco, escribe en orden todos los bloques de disco en la cinta de salida y se detiene cuando ha copiado el último bloque. Un programa así es tan simple que es probable que sea posible eliminar por completo los errores, algo que tal vez no pueda decirse acerca de ningún otro programa útil.

No obstante, vale la pena hacer varios comentarios acerca de los volcados físicos. Para empezar, no tiene sentido respaldar bloques de disco desocupados. Si el programa de volcado puede obtener acceso a la estructura de datos de bloques libres, podrá evitar el volcado de esos bloques. Sin embargo, el hecho de saltarse los bloques no utilizados obliga a escribir el número

de bloque (o su equivalente) antes de cada bloque, pues el bloque k de la cinta ya no es el bloque k en el disco.

Una segunda inquietud es el volcado de bloques defectuosos. Si el controlador de disco remapea todos los bloques defectuosos y los oculta del sistema operativo como describimos en la sección 5.4.4, el volcado físico funcionará sin problemas. Pero si el sistema operativo puede ver esos bloques y los mantiene en uno o más “fichero de bloques defectuosos” o mapas de bits, es absolutamente indispensable que el programa de volcado físico tenga acceso a esta información y evite vaciar esos bloques para evitar interminables errores de lectura de disco durante el proceso de vaciado.

Las ventajas principales del volcado físico son la sencillez y la rapidez (básicamente porque puede efectuarse a la velocidad del disco). Las desventajas principales son la imposibilidad de saltarse directorios específicos, de efectuar volcados incrementales y de restaurar ficheros individuales si se solicita. Por estas razones, casi todas las instalaciones efectúan volcados lógicos.

Un **volcado lógico** comienza en uno o más directorios específicos y respalda de forma recursivo todos los ficheros y directorios que se encuentran ahí y que hallan sido modificados desde alguna fecha base dada (por ejemplo, el último respaldo en el caso de un volcado incremental o la fecha de instalación del sistema en el caso de un volcado completo). Así, en un volcado lógico la cinta recibe una serie de directorios y ficheros identificados meticulosamente, lo cual facilita la restauración de un fichero o directorio específico cuando se solicite.

Puesto que el volcado lógico es la forma más común, examinaremos los pormenores de un algoritmo común utilizando el ejemplo de la Figura 6-24 como guía. Casi todos los sistemas UNIX utilizan este algoritmo. En la figura vemos un árbol de ficheros con directorios (cuadrados) y ficheros (círculos). El sombreado indica lo que se ha modificado desde la fecha base y por tanto debe volcarse. Los elementos sin sombrear no tienen que volcarse.

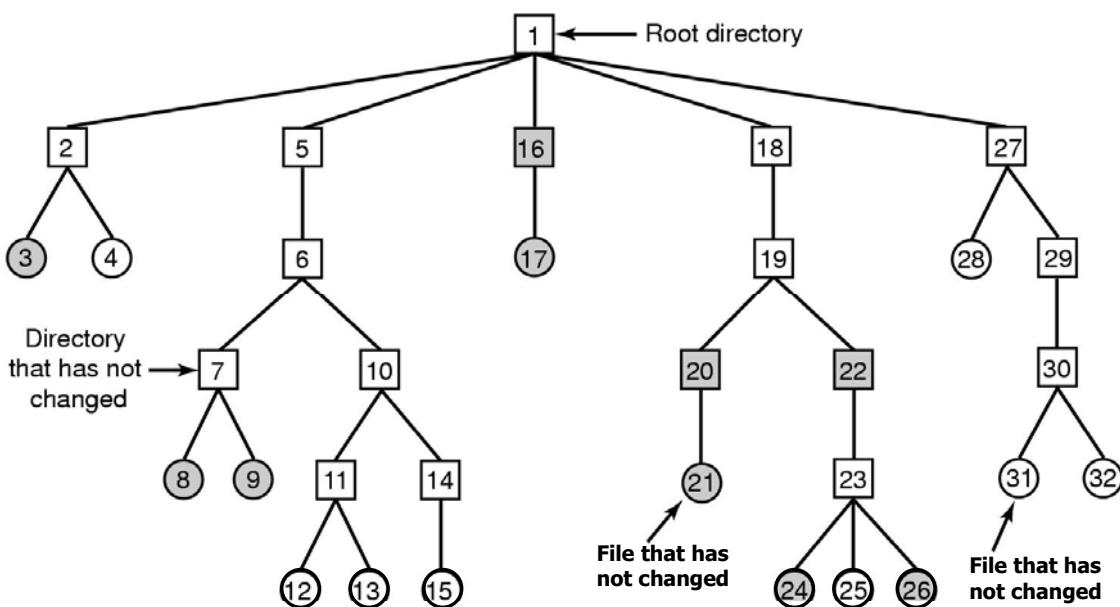


Figura 6-24. Sistema de ficheros que debe volcarse. Los cuadrados son directorios y los círculos son ficheros. Los elementos sombreados se modificaron después del último volcado. Cada directorio y fichero está rotulado con su número de i-nodo.

Este algoritmo también vuelca todos los directorios (aunque no se hayan modificado) que están en el camino que va hasta un fichero o directorio modificado, por dos razones. La primera es para que sea posible restaurar los ficheros y directorios volcados en un sistema de ficheros nuevo en otro ordenador. Así, pueden utilizarse los programas de volcado y restauración para transportar sistemas de ficheros completos de un ordenador a otro.

El segundo motivo para volcar los directorios no modificados que están por encima de ficheros modificados es hacer posible la restauración incremental de un único fichero (tal vez para recuperarse de una estupidez). Supongamos que se efectúa un volcado completo del sistema de ficheros el domingo por la noche y un volcado incremental el lunes por la noche. El martes se elimina el directorio `/usr/jhs/proa/nr3`, junto con todos los directorios y ficheros que están más abajo. El miércoles temprano el usuario quiere restaurar el fichero `/usr/jhs/proa/nr3/plans/sumary`. Sin embargo no es posible restaurar solamente el fichero resumen porque no hay donde ponerlo. Primero hay que restaurar los directorios `nr3` y `plans`. Para que sus propietarios, modos, tiempos, etcétera, sean los correctos, esos directorios deberán estar presentes en la cinta de volcado aunque no se hayan modificado desde el volcado completo anterior.

El algoritmo de volcado mantiene un mapa de bits indexado por el número de i-nodo, con varios bits por i-nodo. Se activan y desactivan bits en este mapa a medida que avanza el algoritmo. Éste último opera en cuatro fases. La fase 1 comienza en el directorio inicial (la raíz, en este ejemplo) y examina todas las entradas que contiene. Por cada fichero modificado, su i-nodo se marca en el mapa de bits. También se marca cada uno de los directorios (se haya modificado o no) y luego se inspecciona de forma recursiva.

Al término de la fase 1, todos los ficheros modificados y todos los directorios se han marcado en el mapa de bits, como se muestra (con sombreado) en la Figura 6-25(a). Conceptualmente, la fase 2 recorre el árbol otra vez de forma recursiva, quitando la marca a todos los directorios que no tengan (o en los que no haya debajo) ficheros o directorios modificados. Esta fase deja el mapa como se muestra en la Figura 6-25(b). Obsérvese que los directorios 10, 11, 14, 27, 29 y 30 ya no están marcados, porque debajo de ellos no hay nada que se haya modificado. Estos directorios no se vaciarán. En contraste, los directorios 5 y 6 sí se vacían aunque en sí no se hayan modificado, porque se necesitarán para restaurar los cambios de hoy en otra máquina. Por eficiencia, las fases 1 y 2 pueden combinarse en un único recorrido del árbol.

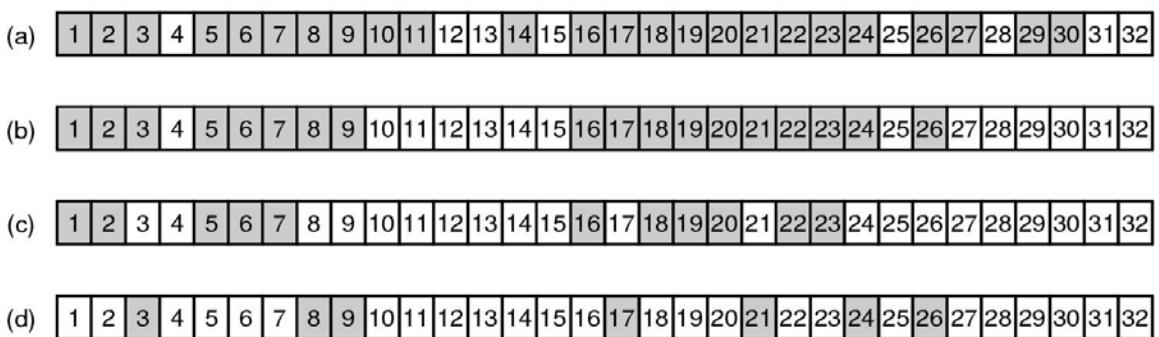


Figura 6-25. Mapas de bits utilizados por el algoritmo de volcado lógico.

Ahora se sabe qué directorios y ficheros deben volcarse. Son los que se marcaron en la Figura 6-25(b). En la fase 3 se exploran los i-nodos en orden numérico y se vuelcan todos los directorios marcados para ello. Éstos se muestran en la Figura 6-25(c). A cada directorio se le anteponen sus atributos (propietario, tiempos, etcétera) para que pueda restaurarse. Por último,

en la fase 4 se respaldan también los ficheros marcados en la Figura 6-25(d), anteponiéndoles también sus atributos. Con esto termina el volcado.

La restauración de un sistema de ficheros a partir de las cintas de volcado es directa. Lo primero que se hace es crear un sistema de ficheros vacío en el disco. Luego se restaura el volcado completo más reciente. Puesto que los directorios aparecen primero en la cinta, se restauran todos primero, para dar un esquema del sistema de ficheros. Luego se restauran los ficheros mismos. Despues se repite este proceso con el primer volcado incremental efectuado después del volcado completo, seguido del siguiente, y así en forma sucesiva.

Aunque los volcados lógicos no son complicados, deben cuidarse algunos aspectos. Uno de ellos es que la lista de bloques libres no es un fichero, de modo que no se respalda y habrá que reconstruirla desde cero una vez que se hayan restaurado todos los volcados. Esto siempre es posible porque el conjunto de bloques libres no es más que el conjunto complementario del conjunto de bloques contenidos en todos los ficheros combinados.

Otro problema es el de los enlaces. Si un fichero está enlazado a dos o más directorios, es importante que se restaure sólo una vez y que todos los directorios que deben apuntar a él lo hagan.

Un tercer problema es el hecho de que los ficheros UNIX pueden contener huecos. Está permitido abrir un fichero, escribir unos cuantos bytes, luego posicionarse dentro del fichero a una gran distancia del principio y escribir otros pocos bytes. Los bloques intermedios no forman parte del fichero y no deben vaciarse ni restaurarse. Los ficheros del núcleo suelen tener un gran hueco entre el segmento de datos y la pila. Si esto no se maneja de forma correcta, cada uno de los ficheros de núcleo que se restauren tendrá esta área llena de ceros y por tanto será del mismo tamaño que el espacio de direcciones virtual (por ejemplo 2^{32} bytes o, peor aún, 2^{64} bytes).

Por último, los ficheros especiales, las tuberías con nombre y cosas así jamás deben vaciarse, estén en el directorio que estén (no es forzoso que estén en `/dev`). Puede encontrarse más información acerca de los respaldos de sistemas de ficheros en Chervanak y otros (1998) y en Zwicky (1991).

Consistencia del sistema de ficheros

Otro área en la que la fiabilidad es un problema es en la consistencia de los sistemas de ficheros. Muchos sistemas de ficheros leen bloques, los modifican y los vuelven a escribir después. Si hay un fallo antes de que se escriban en el disco todos los bloques modificados, el sistema de ficheros podría quedar en un estado inconsistente. Este problema es grave sobre todo si algunos de los bloques que todavía no se han escrito en disco son bloques de i-nodo, bloques de directorio o bloques que contienen la lista libre.

Para resolver el problema de la inconsistencia del sistema de ficheros, casi todos los ordenadores tienen un programa de utilidad que verifica la consistencia. Por ejemplo, UNIX tiene `fsck` y Windows tiene `scandisk`. Este programa puede ejecutarse cada vez que se arranca el sistema, en especial después de una caída del sistema. La descripción que sigue es del funcionamiento de `fsck`. `Scandisk` es un tanto diferente porque opera con un sistema de ficheros distinto, pero el principio general de aprovechar la redundancia inherente del sistema de ficheros para repararlo sigue siendo válido. Todos los verificadores de sistemas de ficheros examinan cada sistema de ficheros (partición de disco) independientemente de los demás.

Es posible efectuar dos tipos de verificaciones de consistencia: de bloques y de ficheros. Para verificar la consistencia de los bloques, el programa construye dos tablas, cada una de las cuales contiene un contador para cada bloque, que en un principio se establecen a 0. Los contadores de la primera tabla cuentan las veces que cada bloque está presente en un fichero.

Los de la segunda tabla cuentan las veces que cada bloque está presente en la lista de bloques libres (o en el mapa de bits de bloques libres).

A continuación, el programa lee todos los i-nodos. Partiendo de un i-nodo, es posible construir una lista de todos los números de bloque empleados en el fichero correspondiente. A medida que se lee cada número de bloque, se incrementa su contador en la primera tabla. Luego el programa examina la lista libre o el mapa de bits para hallar todos los bloques que no se están utilizando. Cada aparición de un bloque en la lista de bloques libres hace que se incremente su contador en la segunda tabla.

Si el sistema de fichero es consistente, todos los bloques tendrán un 1 en la primera tabla o bien en la segunda, pero no en las dos, como se ilustra en la Figura 6-26(a). Sin embargo, como resultados de un fallo en las tablas podrían quedar como en la Figura 6-26(b), donde el bloque 2 está ausente de ambas tablas. Esto debe de ser indicado por el programa como un **bloque perdido** (*missing block*). Aunque los bloques perdidos realmente no representan ningún peligro, desperdician espacio y por tanto reducen la capacidad del disco. La solución en este caso es sencilla: el verificador del sistema de ficheros simplemente los añade a la lista de bloques libres.

| BLOCK NUMBER | | | | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Blocks in use | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Free blocks | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| (a) | | | | | | | | | | | | | | | |

| BLOCK NUMBER | | | | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Blocks in use | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Free blocks | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| (b) | | | | | | | | | | | | | | | |

| BLOCK NUMBER | | | | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Blocks in use | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Free blocks | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| (c) | | | | | | | | | | | | | | | |

| BLOCK NUMBER | | | | | | | | | | | | | | | |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Blocks in use | | | | | | | | | | | | | | | |
| 1 | 1 | 0 | 1 | 0 | 2 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| Free blocks | | | | | | | | | | | | | | | |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| (d) | | | | | | | | | | | | | | | |

Figura 6-26. Estados del sistema de ficheros. (a) Consistente. (b) Bloque perdido.
(c) Bloque repetido en la lista de bloques libres. (d) Bloque de datos repetido.

Otra situación que podría presentarse es la de la Figura 6-26(c). Aquí vemos un bloque, el número 4, que aparece dos veces en la lista de bloques libres. (Sólo puede haber bloques repetidos si la lista de bloques libres está realmente implementada como una lista; con un mapa de bits es imposible.) La solución en este caso es también sencilla: reconstruir la lista de bloques libres.

Lo peor que puede suceder es que el mismo bloque de datos esté presente en dos o más ficheros, como en el bloque 5 en la Figura 6-26(d). Si se borra cualquiera de estos dos ficheros, el bloque 5 se colocará en la lista de bloques libres y dará pie a una situación en la que el mismo bloque esté en uso y libre al mismo tiempo. Si se borran ambos ficheros, el bloque se colocará dos veces en la lista de bloques libres.

La acción apropiada del verificador del sistema de ficheros es asignar un bloque libre, copiar en él el contenido del bloque 5 e insertar la copia en uno de los ficheros. Así, el contenido de información de los ficheros no cambia (aunque con toda seguridad uno de ellos no será correcto), pero al menos el sistema de ficheros recupera la consistencia. Debe informarse del error, para que el usuario pueda inspeccionar el daño.

Además de comprobar que se conozca dónde están todos los bloques, el verificador del sistema de ficheros examina también el sistema de directorios. Aquí también se utiliza una tabla de contadores, pero estos son de ficheros, no de bloques. Se parte del directorio raíz, descendiendo de forma recursiva por el árbol, inspeccionando cada directorio del sistema de ficheros. Para cada fichero de cada directorio, se incrementa un contador por cada uso de ese fichero. Recordemos que, debido a los enlaces duros, un fichero podría aparecer en dos o más directorios. Los enlaces simbólicos no cuentan y no hacen que se incremente el contador del fichero en cuestión.

Al terminar el verificador, tiene una lista indexada por el número de i-nodo, que indica cuántos directorios contiene cada fichero. Luego compara estas cifras con los contadores de enlaces almacenados en los mismos i-nodos. Estos contadores son 1 cuando se crea el fichero y se incrementan cada vez que se establece un enlace (duro) con el fichero. En un sistema de ficheros consistente, ambos contadores coinciden. Sin embargo, pueden presentarse dos tipos de errores: el contador de enlaces en el i-nodo puede ser demasiado alto o demasiado bajo.

Si el contador de enlaces es mayor que el número de entradas en los directorios, entonces aunque se eliminen todos los ficheros de esos directorios el contador seguirá siendo mayor que cero y el i-nodo no se eliminará. Este error no es grave, pero desperdicia espacio en el disco con ficheros que no están en ningún directorio. Debe corregirse asignando el valor correcto al contador de enlaces en el i-nodo.

El otro error podría ser catastrófico. Si dos entradas de directorio están enlazadas en un fichero pero el i-nodo dice que sólo hay una, cuando se borre cualquiera de las dos entradas de directorio el contador del i-nodo pasará a valer cero. Cuando esto sucede, el sistema de ficheros marca el i-nodo como no utilizado y libera todos sus bloques. Esta acción hará que uno de los directorios apunte a un i-nodo que no se utiliza y cuyos bloques tal vez pronto se asignen a otros ficheros. Una vez más, la solución es simplemente asignar el número real de entradas de directorio al contador de enlaces en el i-nodo.

Estas dos operaciones, la verificación de bloques y la verificación de directorios, suelen integrarse para mejorar la eficiencia (ya que sólo habrá que dar una pasada por los i-nodos). También pueden efectuarse otras verificaciones. Por ejemplo, los directorios tienen un formato definido, con números de i-nodo y nombres ASCII. Si un número de i-nodo es mayor que el número de i-nodos que hay en disco, quiere decir que el directorio está dañado.

Además, cada i-nodo tiene un modo, algunos de los cuales son válidos pero extraños, como el 0007, que prohíbe al propietario del fichero y a su grupo todo tipo de acceso, pero permite a cualquier otra persona leer, escribir y ejecutar el fichero. Podría ser útil informar por lo menos sobre cualquier fichero que otorga a extraños más privilegios que al propietario. Los directorios con más de, digamos, 1000 entradas son también sospechosos. Los ficheros situados en directorios de usuario pero que son propiedad del superusuario y tienen activado el bit SETUID, son problemas de seguridad potenciales porque permiten a cualquier usuario que los ejecute adquiriera las facultades del superusuario. Con un poco de esfuerzo, es posible armar una lista relativamente larga de situaciones permitidas desde el punto de vista técnico, pero que no dejan de ser peculiares y de las que podría ser recomendable informar.

En los párrafos anteriores analizamos el problema de proteger al usuario contra las caídas del sistema. Algunos sistemas de ficheros se preocupan también por proteger al usuario contra sí mismo. Si el usuario quiere teclear

`rm *.o`

para eliminar todos los ficheros que terminan por `.o` (ficheros objeto generados por el compilador) pero por accidente teclea

`rm * .o`

(observe el espacio después del asterisco), *rm* borrará todos los ficheros del directorio actual y luego se quejará de que no puede encontrar *.o*. En MS-DOS y algunos otros sistemas, cuando se borra un fichero, lo único que sucede es que se activa un bit en el directorio o en el i-nodo para marcar el fichero como borrado. No se devuelven bloques libres a la lista de bloques libres mientras no se necesiten de verdad. Por tanto, si el usuario descubre el error de inmediato, es posible ejecutar un programa de utilidad especial que restaura los ficheros borrados. En Windows, los ficheros que se borran se colocan en la papelera de reciclaje, de donde se pueden recuperar sin es necesario. Por supuesto, no se libera el espacio que ocupan hasta que los ficheros no se borran realmente de la papelera de reciclaje.

6.3.7 Rendimiento del sistema de ficheros

El acceso a un disco es mucho más lento que el acceso a la memoria. La lectura de una palabra de memoria podría tardar 10 ns. La lectura de un disco duro podría efectuarse a 10 MB/s. lo cual es 40 veces más lento por palabra de 32 bits, pero a esto se le debe añadir de 5 a 10 ms para situar el brazo del disco hasta la pista y luego esperar a que el sector deseado esté debajo de la cabeza lectora. Si sólo se necesita una palabra, el acceso a la memoria es del orden de un millón de veces más rápido que el acceso al disco. En vista de esta diferencia en el tiempo de acceso, muchos sistemas de ficheros se han diseñado con diversas optimizaciones para mejorar su rendimiento. En esta sección cubriremos tres de ellas.

Uso de caché

La técnica empleada más comúnmente para reducir los accesos al disco es la **caché de bloques** o **caché de búfer**. (La palabra caché proviene del verbo francés *cacher*, que significa esconder.) En este contexto, una caché es una colección de bloques que lógicamente debían de estar en el disco pero que se están manteniendo en la memoria por cuestiones de eficiencia.

Pueden utilizarse diversos algoritmos para administrar la caché, pero uno muy común es mirar, en cada solicitud de lectura, si el bloque deseado está en la caché. Si está ahí, la solicitud de lectura podrá satisfacerse sin acceder al disco. Si el bloque no está en la caché, primero se lee del disco y se coloca en la caché y luego se copia donde se necesita. Las solicitudes posteriores que pidan el mismo bloque podrán satisfacerse desde la caché.

El funcionamiento de la caché se ilustra en la Figura 6-27. Puesto que hay muchos bloques en la caché (a menudo miles), se necesita algún mecanismo para determinar con rapidez si un bloque está presente o no. El mecanismo común consiste en dispersar (hash) el dispositivo y la dirección del disco, y buscar el resultado en una tabla hash. Todos los bloques con el mismo valor de hash están encadenados en una lista enlazada para poder seguir la cadena de colisiones.

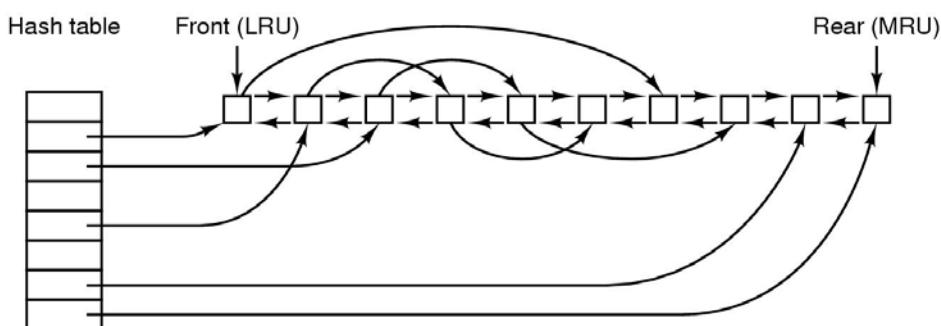


Figura 6-27. Estructuras de datos del caché de búfer.

Cuando es necesario cargar un bloque en una caché llena, hay que liberar algún bloque (y reescribirlo en el disco si ha sido modificado desde que se colocó allí). Esta situación se parece mucho a la paginación, y son aplicables todos los algoritmos de sustitución de páginas usuales que describimos en el capítulo 3, como FIFO, segunda oportunidad y LRU. Una diferencia agradable entre la paginación y el uso de caché es que las referencias a la caché no son demasiado frecuentes, por lo que es factible mantener todos los bloques en orden LRU exacto con listas enlazadas.

En la Figura 6-27 vemos que, además de las cadenas de colisiones que comienzan en la tabla hash, también hay una lista doblemente enlazada con todos los bloques ordenados según el orden de utilización, con el bloque utilizado menos recientemente al frente de la lista y el bloque utilizado más recientemente al final. Cuando se hace referencia a un bloque, se puede quitar de su posición en la lista bidireccional y colocarse en el extremo. De esta manera, puede mantenerse orden LRU exacto.

Por desgracia hay un pequeño problema. Ahora que tenemos una situación en la que es posible utilizar LRU exacto, resulta que no es recomendable. El problema tiene que ver con las caídas del sistema y la consistencia del sistema de ficheros que examinamos en la sección anterior. Si se lee del disco un bloque crucial, digamos un bloque de i-nodos, se coloca en la caché y se modifica, pero no se reescribe en el disco, un fallo dejaría al sistema de ficheros en un estado inconsistente. Si el bloque de i-nodos se coloca en el extremo de la cadena LRU, podría pasar un buen tiempo antes de que llegue al frente y se reescriba en el disco.

Además, es raro que se haga referencia a algunos bloques, como los de i-nodos, dos veces dentro de un intervalo de tiempo corto. Estas consideraciones nos llevan a un esquema LRU modificado que toma en cuenta dos factores:

1. ¿Es probable que se vuelva a necesitar pronto el bloque?
2. ¿El bloque es indispensable para la consistencia del sistema de ficheros?

Para contestar a ambas preguntas, los bloques pueden dividirse en categorías como bloques de i-nodos, bloques indirectos, bloques de directorio, bloques de datos llenos y bloques de datos parcialmente llenos. Los bloques que es probable que no se vayan a necesitar pronto se colocan al frente de la lista LRU, no al final, para que sus búferes se reciclen rápido. Los bloques que podrían volver a necesitarse pronto, como un bloque parcialmente lleno en el que se está escribiendo, se colocan al final de la lista para que permanezcan en la caché largo tiempo.

La segunda pregunta es independiente de la primera. Si el bloque es indispensable para la consistencia del sistema de ficheros (básicamente todo menos los bloques de datos) y se ha modificado, se deberá escribir en disco de inmediato, sin importar en qué extremo de la lista LRU se coloque. Al escribir rápidamente los bloques cruciales, se reduce mucho la probabilidad de que una caída del sistema estropee el sistema de ficheros. Un usuario podría molestarse si uno de sus ficheros se estropeará por un fallo, pero seguramente estaría mucho más molesto si se pierde todo el sistema de ficheros.

Incluso con esta precaución para mantener la integridad del sistema de ficheros, no es conveniente mantener los bloques de datos en la caché demasiado tiempo antes de escribirlos en el disco. Consideremos la situación de una persona que está utilizando un ordenador personal para escribir un libro. Incluso si el escritor le ordena de forma periódica al programa editor que grabe en el disco el fichero que está editando, existe la posibilidad de que todo siga en la caché y que no se escriba nada en el disco. Si el sistema falla, la estructura del sistema de ficheros no se corrompe, pero se pierde todo un día de trabajo.

Esta situación no necesita presentarse muy a manudo para que tengamos un usuario muy descontento. Los sistemas adoptan dos métodos para resolver este problema. Lo que hace UNIX es ofrecer una llamada al sistema, sync, que escribe de inmediato en disco todos los bloques modificados. Cuando se inicia el sistema, se pone en marcha un programa en segundo plano, por lo normal llamado update, que da vueltas en un bucle infinito emitiendo llamadas sync, durmiéndose durante 30 segundos entre llamadas. Gracias a esto, no se pierden más de 30 segundos de trabajo por una caída.

Lo que hace MS-DOS es escribir en disco todos los bloques modificados tan pronto como se modifican. Las cachés en las que todos los bloques modificados se escriben inmediatamente en el disco se denominan **cachés de escritura directa**, y requieren más E/S de disco que las cahés de otro tipo. La diferencia entre estos dos métodos puede observarse cuando un programa escribe un bloque de 1 KB hasta llenarlo, carácter a carácter. UNIX reúne todos los caracteres que están en la cahé y escribe el bloque en disco una vez cada 30 segundos, o cuando el bloque se desaloje de la caché. MS-DOS efectúa un acceso al disco por cada carácter que se escribe. Desde luego la mayoría de los programas utiliza búferes internos, así que normalmente no se escribe un carácter, sino una línea o unidad más grande en cada llamada al sistema write.

Una consecuencia de esta diferencia en la estrategia de almacenamiento en caché es que el simple hecho de sacar un disco (flexible) de un sistema UNIX sin emitir una llamada sync casi siempre da como resultado una pérdida de datos, y en muchos casos también un sistema de ficheros corrupto. Con MS-DOS no hay problema. Se escogieron estas diferentes estrategias porque UNIX se desarrolló en un entorno en el que los discos eran discos duros y no removibles, mientras que MS-DOS nació en el mundo de los disquetes. A medida que los discos duros se convirtieron en la norma, el método UNIX, al ser más eficiente, se convirtió también en la norma y se usa ahora en Windows con los discos duros.

Lectura adelantada de bloques

Una segunda técnica para mejorar el rendimiento aparente del sistema de ficheros es tratar de colocar bloques en la caché antes de que se necesiten, a fin de mejorar la tasa de aciertos. En particular, muchos ficheros se leen de forma secuencial. Cuando se pide al sistema de ficheros entregar el bloque k de un fichero, lo hace, pero cuando termina examina de manera subrepticia la caché para ver si ya está ahí el bloque $k + 1$. Si no está, se planifica la lectura de ese bloque con la esperanza de que, cuando se necesite, ya haya llegado a la caché o que, cuando menos, ya venga de camino.

Desde luego, tal estrategia de lectura adelantada sólo funciona en el caso de ficheros que se están leyendo de forma secuencial. Si el acceso a un fichero es aleatorio, la lectura adelantada no ayuda; de hecho, perjudica porque ocupa parte del ancho de banda del disco en la lectura de bloques que no se utilizarán y hace que se expulsen de la caché bloques que tal vez sí sean útiles (y quizás ocupa más ancho de banda de disco, aún si los bloques expulsados estaban modificados y fue necesario escribirlos primero en disco). Para ver si vale la pena la lectura adelantada, el sistema de ficheros puede determinar los patrones de acceso a los ficheros abiertos. Por ejemplo, un bit asociado con cada fichero puede indicar si está en “modo de acceso secuencial” o en “modo de acceso aleatorio”. En un principio, se da por hecho que el fichero está en modo de acceso secuencial, pero cada vez que se efectúa un posicionamiento del brazo del disco, el bit se desactiva. Si luego vuelven a presentarse lecturas secuenciales, el bit se activa de nuevo. De este modo, el sistema de ficheros puede hacer conjeturas razonables respecto a si le conviene leer por adelantado o no. Si se equivoca de vez en cuando, no será ningún desastre; sólo se desperdiciará un poco del ancho de banda del disco.

Reducción del movimiento del brazo del disco

La utilización de caché y la lectura adelantada no son las únicas formas de mejorar el rendimiento del sistema. Otra técnica importante consiste en reducir los movimientos del brazo del disco colocando juntos, preferentemente en el mismo cilindro, los bloques a los que tal vez se tendrá acceso de forma secuencial. Cuando se escribe en un fichero de salida, el sistema de ficheros tiene que asignar los bloques uno por uno, conforme se van necesitando. Si el control de los bloques libres se lleva con un mapa de bits, y todo el mapa de bits está en la memoria principal, no será difícil escoger un bloque libre lo más cercano posible al bloque anterior. Si se utiliza una lista de bloques libres, parte de la cual están en el disco, será mucho más difícil asignar bloques cercanos entre sí.

No obstante, incluso con una lista de bloques libres puede lograrse cierta agregación de bloques. El truco es llevar el control del almacenamiento en el disco no por bloques, sino por grupos de bloques consecutivos. Si los sectores constan de 512 bytes, el sistema podría usar bloques de 1 KB (2 sectores), pero asignar el almacenamiento en disco en unidades de 2 bloques (4 sectores). Esto no es lo mismo que tener bloques de disco de 2 KB porque la caché de todos modos utilizará bloques de 1 KB y las transferencias de disco seguirán siendo de 1 KB, pero la lectura secuencial de un fichero en un sistema de otra manera ocioso reducirá el número de desplazamientos del brazo del disco a la mitad, mejorando el rendimiento de manera considerable. Una variación del mismo tema es tomar en cuenta el posicionamiento rotacional. Al asignar bloques, el sistema intenta colocar los bloques consecutivos de un fichero en el mismo cilindro.

Otro cuello de botella en lo tocante al rendimiento en sistemas que utilizan i-nodos o algún equivalente es que la lectura de cualquier fichero, por pequeño que sea, requiere dos accesos al disco: uno para el i-nodo y otro para el bloque. En la Figura 6-28(a) se muestra la colocación usual de los i-nodos. Aquí todos los i-nodos están cerca del principio del disco, por lo que la distancia media entre un i-nodo y sus bloques será alrededor de la mitad del número de cilindros, y requerirá largos desplazamientos del brazo.

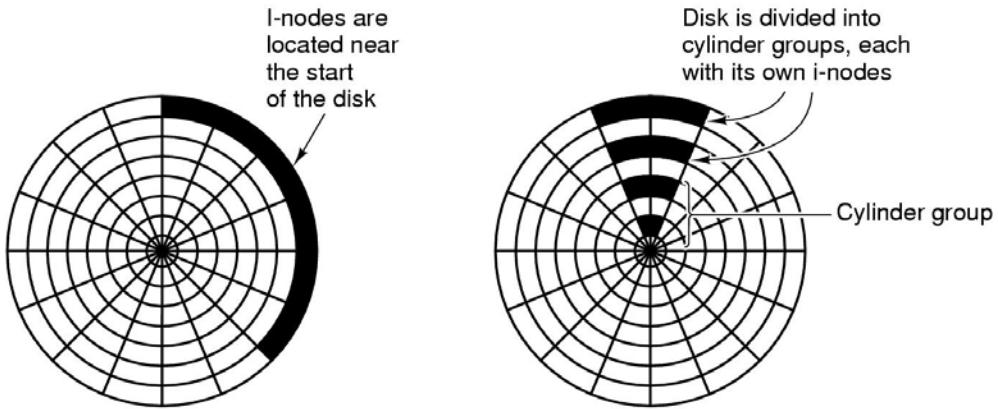


Figura 6-28. (a) i-nodos colocados al principio del disco. (b) Disco dividido en grupos de cilindros, cada uno con sus propios bloques e i-nodos.

Una forma fácil de mejorar el rendimiento es colocar los i-nodos a la mitad del disco, no al principio, con lo que se reduce a la mitad el desplazamiento medio del brazo entre el i-nodo y el primer bloque. Otra idea, que se muestra en la Figura 6-28(b), es dividir el disco en grupos de cilindros, cada uno con sus propios i-nodos, bloques y lista libre (McKusick y otros, 1984). Al crear un fichero nuevo puede escogerse cualquier i-nodo, pero se procura hallar un bloque en el mismo grupo de cilindros en el que está el i-nodo. Si no hay ninguno disponible, se utiliza un bloque de un grupo de cilindros cercano.

6.3.8 Sistemas de ficheros con estructura de registro

Los cambios tecnológicos están sometiendo a mucha presión a los sistemas de ficheros actuales. En particular, las CPUs cada día son más rápidas, los discos cada día son más grandes y de más bajo coste (pero no mucho más rápidos), y el tamaño de las memorias está creciendo de forma exponencial. El único parámetro que no está mejorando a ritmo vertiginoso es el tiempo de desplazamiento del brazo del disco. La combinación de estos factores implica que en muchos sistemas de ficheros esté apareciendo un cuello de botella en lo tocante a la eficiencia. Algunas investigaciones efectuadas en Berkeley intentaron aliviar este problema diseñando un tipo de sistemas de ficheros totalmente nuevo, el **sistema de ficheros con estructura de diálogo (LFS; Log-structured File System)**. En esta sección describiremos de forma breve el funcionamiento del LFS. Puede encontrarse un tratamiento más completo en Rosenblum y Ousterhout (1991).

La idea que impulsó el diseño del LFS es que conforme se vuelven más rápidas las CPUs y las memorias RAM aumentan de tamaño, las cachés de disco también están creciendo con rapidez. Por ello, ahora es posible satisfacer una fracción considerable de todas las solicitudes de lectura directamente de la caché del sistema de archivos, sin necesidad de accesos a disco. Una consecuencia de esta observación es que, en el futuro, casi todos los accesos al disco escrituras, así que el mecanismo de lectura adelantada empleado en algunos sistemas de ficheros para traer bloques antes de que se necesiten ya no mejora mucho el rendimiento.

Para empeorar las cosas, en la mayoría de los sistemas de ficheros las escrituras se efectúan en trozos muy pequeños. Las escrituras pequeñas son muy ineficientes, porque una escritura en disco de 50 µs muchas veces va precedida por un desplazamiento del brazo que tarda 10 ms y por un retraso rotacional de 4 ms. Con estos parámetros, la eficiencia del disco baja a una fracción de 1%.

Para ver de dónde provienen todas esas escrituras pequeñas, consideremos la creación de un fichero nuevo en un sistema UNIX. Para escribir este fichero, es preciso escribir el i-nodo del directorio, el bloque del directorio, el i-nodo del fichero y el fichero en sí. Aunque todas estas escrituras pueden aplazarse, eso expone al sistema de ficheros a problemas de consistencia graves si se presenta un fallo antes de que se efectúen las escrituras. Por ello, las escrituras de i-nodos generalmente se hacen de inmediato.

Con base en este razonamiento, los diseñadores del LFS decidieron reimplementar el sistema de ficheros de UNIX a modo de lograr el ancho de banda máximo del disco, aun cuando la carga de trabajo consista en su mayor parte en pequeñas escrituras aleatorias. La idea básica es estructurar el disco como un registro. De manera periódica, y cuando surge una necesidad especial de hacerlo, todas las escrituras pendientes que están en búferes en la memoria se reúnen en un único segmento y se escriben en el disco como un segmento contiguo único al final del registro. Así, un único segmento podría contener i-nodos, bloques de directorio y bloques de datos, todos revueltos. Al principio de cada segmento hay un resumen del segmento que indica su contenido. Si puede hacerse que el tamaño medio de tales segmentos sea cercano a 1MB, podrá aprovecharse casi todo el ancho de banda del disco.

En este diseño siguen existiendo i-nodos y tienen la misma estructura que en UNIX, pero ahora están dispersos por todo el registro, en lugar de estar en una posición fija en el disco. No obstante, cuando se localiza un i-nodo, la localización de los bloques se efectúa de la manera acostumbrada. Claro que ahora es mucho más difícil hallar un i-nodo, porque su dirección no puede calcularse simplemente a partir de su número-i, como en UNIX. Para que sea posible hallar i-nodos, se mantiene un mapa de ellos, indexado por número-i. La entrada i de este mapa apunta al i-nodo i en el disco. El mapa se mantiene en el disco, pero también en la caché, así que las partes más utilizadas estarán en la memoria la mayor parte del tiempo.

Para resumir lo que hemos dicho hasta ahora, todas las escrituras se colocan al principio en búferes en la memoria, y en forma periódica todas las escrituras almacenadas en el búfer se escriben en disco en un único segmento, al final del registro. La apertura de un fichero consiste ahora en utilizar el mapa para localizar el i-nodo del fichero. Una vez hallado ese i-nodo, proporciona las direcciones de los bloques. Todos los bloques estarán también en segmentos, en algún lugar del registro.

Si los discos fueran infinitamente grandes, la descripción anterior sería todo. Sin embargo, los discos reales son finitos, por lo que tarde o temprano el registro ocupará todo el disco, y ya no será posible escribir segmentos nuevos en él. Por fortuna, muchos segmentos existentes podrían tener bloques que ya no se necesitan. Por ejemplo, si un fichero se sobrescribió, su i-nodo apuntará ahora a los nuevos bloques, pero los viejos seguirán ocupando espacio en segmentos escritos antes.

Para resolver este problema, LFS tiene un proceso **limpiador** que dedica su tiempo a explorar el registro de forma circular para compactarla. Lo primero que hace es leer el resumen del primer segmento del registro para ver qué i-nodos y ficheros están ahí. Luego examina el mapa de i-nodos actual para ver si siguen vigentes y si los bloques de fichero todavía están en uso. Si no, esa información se desecha. Los i-nodos y bloques que todavía están en uso se pasan a la memoria para que se escriban en disco en el siguiente segmento. Ahora el segmento original se marca como libre, y el registro podrá utilizarlo para grabar datos nuevos. De este modo, el limpiador avanza por el registro, eliminando segmentos viejos de la parte de atrás y colocando los datos vigentes en la memoria para que se reescriban en el siguiente segmento. El resultado es que el disco actúa como un gran búfer circular, con el subprograma escritor añadiendo segmentos nuevos al frente, y el subprograma limpiador eliminando segmentos viejos en la parte posterior.

La compatibilización en este sistema no es trivial, pues cuando un bloque de ficheros se escribe en un nuevo segmento, es preciso localizar el i-nodo del fichero (en algún lugar del registro), actualizarlo y colocarlo en la memoria para que se escriba en el disco en el siguiente segmento. Luego hay que actualizar el mapa de i-nodos para que apunte a la nueva copia. No obstante, tal administración es factible, y las mediciones de rendimiento demuestran que toda esta complejidad vale la pena. Las mediciones presentadas en los artículos antes citados muestran que LFS mejora el rendimiento de UNIX en un orden de magnitud cuando las escrituras son pequeñas, y tiene un rendimiento tan bueno como el de UNIX, o mejor, en las lecturas y en las escrituras grandes.

6.4 EJEMPLOS DE SISTEMAS DE FICHEROS

En las secciones que siguen analizaremos varios ejemplos de sistemas de ficheros, que van desde muy sencillos hasta muy avanzados. Puesto que los sistemas de ficheros UNIX modernos y el sistema de ficheros nativo de Windows 2000 se tratan en los capítulos sobre UNIX (capítulo 10) y sobre Windows 2000 (capítulo 11), no veremos esos sistemas aquí. Lo que sí haremos a continuación será examinar a sus predecesores.

6.4.1 Sistemas de ficheros en CD-ROM

Como primer ejemplo de sistema de ficheros, consideremos los sistemas de ficheros que se utilizan en los CD-ROMs. Estos sistemas son notablemente sencillos porque se diseñaron para medios en los que sólo se escribe una vez. Por ejemplo, entre otras cosas, no consideran un control de bloques libres porque en un CD-ROM los bloques no pueden ni liberarse ni añadirse después de que se ha fabricado el disco. A continuación examinaremos el principal tipo de sistema de ficheros para CD-ROM y dos de sus extensiones.

El sistema de ficheros ISO 9660

El estándar más común para sistemas de ficheros en CD-ROM se adoptó como Estándar Internacional en 1998 bajo el nombre de **ISO 9660**. Casi todos los CD-ROMs que están en el mercado en la actualidad son compatibles con esta norma, a veces con las extensiones que analizaremos más adelante. Una de las metas de esa norma fue lograr que todo CD-ROM pudiera leerse en cualquier ordenador, independientemente de la ordenación de bytes y del sistema operativo empleados. Por ello, se impusieron algunas limitaciones al sistema de ficheros para que los sistemas operativos más débiles utilizados en ese entonces (como MS-DOS) pudieran leerlo.

Los CD-ROMs no tienen cilindros concéntricos como los discos magnéticos. En su lugar hay una única espiral continua que contienen los bits en sucesión lineal (aunque es posible hacer desplazamientos en dirección radial). Los bits a lo largo de la espiral se dividen en bloques lógicos (también llamados sectores lógicos) de 2352 bytes. Algunos de éstos son para preámbulos, corrección de errores y demás gasto adicional. La porción de “carga útil” de cada bloque lógico es de 2048 bytes. Cuando los CDs se utilizan para grabar música, tienen introducciones, terminaciones y espacios entre pistas, pero estos elementos no se usan en los CD-ROMs de datos. En muchos casos la posición de un bloque a lo largo de la espiral se especifica en minutos y segundos, pero se puede convertir en un número de bloque lineal aplicando el factor de conversión de 1 s = 75 bloques.

ISO 9660 reconoce conjuntos de CD-ROM con hasta $2^{16} - 1$ CDs en cada conjunto. Los CD-ROMs individuales también pueden dividirse en volúmenes lógicos (particiones). Sin embargo, en lo que sigue nos concentraremos en ISO 9660 para un solo CD-ROM sin particiones.

Cada CD-ROM comienza con 16 bloques cuya función no está definida por el estándar ISO 9660. Un fabricante de CD-ROMs podría utilizar esa área para grabar un programa de autoarranque que permita arrancar el ordenador desde el CD-ROM, o para algún otro fin. Luego viene un bloque que contiene el descriptor de volumen primario, el cual contiene información general acerca del CD-ROM. Dicha información incluye el identificador del sistema (32 bytes), el identificador de volumen (32 bytes), el identificador del productor (128 bytes) y el identificador del preparador de datos (128 bytes). El fabricante puede llenar esos campos como desee, pero sólo puede usar letras mayúsculas, dígitos y un número muy reducido de signos de puntuación, a fin de garantizar la compatibilidad entre plataformas.

El descriptor de volumen primario contiene también los nombres de tres ficheros, los cuales podrían contener el resumen, información sobre derechos de autor (*copyright*) e información bibliográfica, respectivamente. Además, están presentes ciertos números clave que incluyen el tamaño de bloque lógico (por lo normal 2048, pero se permiten 4096, 8192 y potencias superiores de 2 en ciertos casos), el número de bloques que tiene el CD-ROM y sus fechas de creación y expiración. Por último, el descriptor de volumen primario contiene también una entrada de directorio que indica dónde hallar el directorio raíz en el CD-ROM (es decir, en qué bloque comienza). Desde este directorio puede localizarse el resto del sistema de ficheros.

Además del descriptor de volumen primario, un CD-ROM puede contener un descriptor de volumen suplementario que contiene información similar al primario, pero no nos ocuparemos aquí de este descriptor.

El directorio raíz y todos los demás directorios, constan de un número variable de entradas, la última de las cuales contiene un bit que la marca como entrada final. Las entradas de directorio en sí también son de longitud variable. Cada una consta de 10 a 12 campos, algunos de los cuales están en ASCII mientras que otros son campos numéricos en binario. Los campos binarios están codificados dos veces, una vez en formato *little endian* (empleado por

ejemplo en el Pentium) y una en formato *big endian* (empleado por ejemplo en el SPARC). Así, un número de 16 bits ocupa 4 bytes y un número de 32 bits ocupa 8 bytes. Fue necesario utilizar esta codificación redundante para evitar lastimar los sentimientos de alguien durante el desarrollo del estándar. Si el estándar hubiera estipulado *little endian*, la gente de compañías con productos *big endian* se habría sentido menospreciada y no lo habría aceptado. Así, el contenido emocional de un CD-ROM puede cuantificarse y medirse con exactitud en kilobytes/hora de espacio desperdiciado.

El formato de una entrada de directorio ISO 9660 se ilustra en la Figura 6-29. Puesto que las entradas de directorio son de longitud variable, el primer campo es un byte que da la longitud de la entrada. Ese byte se define con el bit de orden alto a la izquierda a fin de evitar ambigüedades.

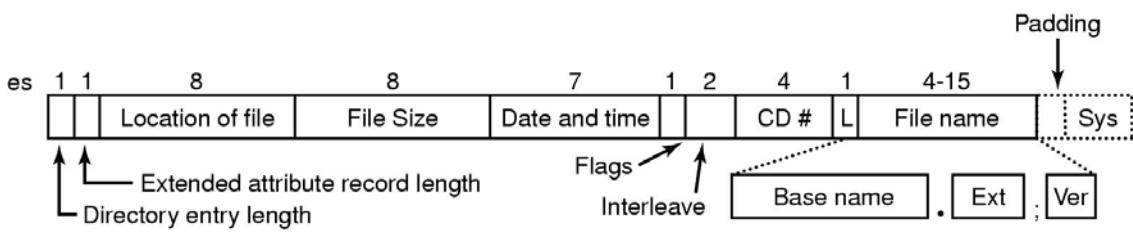


Figura 6-29. La entrada de directorio ISO 9660.

Las entradas de directorio pueden tener, de manera opcional, atributos extendidos. En tal caso, el segundo byte da la longitud del registro de atributos extendidos.

Luego viene la posición del bloque inicial del fichero mismo. Los atributos se almacenan como series contiguas de bloques, por lo que su ubicación queda especificada por completo con el bloque inicial y el tamaño, que viene en el siguiente campo.

La fecha y la hora en que se grabó el CD-ROM están en el campo que sigue, con bytes individuales para el año, mes, día, hora, minuto, segundo y huso horario. Los años se comenzaron a contar en 1900, lo que implica que los CD-ROMs van a tener un problema en el año 2156 porque el año siguiente al 2155 será el 1900. Este problema podría haberse aplazado definiendo el origen del tiempo como 1988 (el año en el que se adoptó el estándar). Si se hubiera hecho así, el problema se habría pospuesto hasta 2244. Otros 88 años de respiro son muy buenos.

El campo *Marcadores* contiene algunos bits de uso diverso, incluyendo uno para ocultar esa entrada en los listados (algo que se copió de MS-DOS), uno para distinguir una entrada de un fichero de una que es un directorio, una para habilitar el uso de los atributos extendidos y una para marcar la última entrada de un directorio. Hay otros bits en este campo pero no nos ocuparemos de ellos aquí. El siguiente campo se ocupa de la intercalación de fragmentos de ficheros en una forma que no se usa en la versión más simple de ISO 9660, así que no hablaremos más de ella.

El campo que sigue indica en qué CD-ROM está el fichero. Es válido que una entrada de directorio en un CD-ROM corresponda a un fichero situado en otro CD-ROM correspondiente a un fichero situado en otro CD-ROM del conjunto. Esto permite construir un directorio maestro en el primer CD-ROM, con una lista de todos los ficheros de todos los CD-ROMs del conjunto completo.

El campo marcado con una L en la Figura 6-29 da el tamaño del nombre de fichero en bytes, y va seguido del nombre de fichero mismo. Un nombre de fichero consta de un nombre base, un punto, una extensión, un signo de punto y coma y un número de versión binario (1 o 2 bytes). El nombre base y la extensión pueden contener letras mayúsculas, los dígitos del 0 al 9 y el carácter de subrayado. Todos los demás caracteres están prohibidos para garantizar que todos los ordenadores puedan manejar todos los nombres de fichero. El nombre base puede tener hasta ocho caracteres; la extensión puede tener hasta tres caracteres. Estas decisiones fueron obligadas por la necesidad de compatibilidad con MS-DOS. Un nombre de fichero dado puede estar presente en un directorio varias veces, mientras cada vez tenga un número de versión distinto.

Los últimos dos campos no siempre están presentes. El campo *Relleno* sirve para hacer que toda entrada de directorio tenga un número par de bytes, a fin de alinear los campos numéricos de entradas subsiguientes en fronteras de dos bytes. Si se necesita relleno, se utiliza un byte 0. Por último, tenemos el campo de *uso del sistema*. Su función y tamaño no están definidos, excepto que debe tener un número par de bytes. Los distintos sistemas lo utilizan de diferente forma. El Macintosh guarda ahí indicadores Finder, por ejemplo.

Dentro de un directorio, las entradas aparecen en orden alfabético con excepción de las dos primeras. La primera entrada es para el directorio en sí. La segunda es para su padre. En este sentido, las dos entradas son similares a las entradas de directorio . y .. de UNIX. Los ficheros no tienen que estar en orden por directorio.

No hay un límite explícito para el número de entradas de un directorio, pero sí para la profundidad de anidamiento: la profundidad máxima es ocho.

ISO 9660 define tres niveles. El nivel 1 es el más restrictivo y especifica que los nombres de fichero están limitados a 8 + 3 caracteres sin extensiones. La utilización de ese nivel ofrece la máxima garantía de que un CD-ROM podrá leerse en cualquier ordenador.

El nivel 2 relaja la restricción de longitud: permite que los ficheros y directorios tengan nombres de hasta 31 caracteres, pero el conjunto de caracteres permitidos sigue siendo el mismo.

El nivel 3 utiliza los mismos límites que el nivel 2 en cuanto a los nombres, pero relaja en parte el requisitote que los ficheros tienen que ser contiguos. Con este nivel, un fichero puede constar de varias secciones, cada una de las cuales es una serie contigua de bloques. La misma serie podría aparecer varias veces en un fichero y también podría aparecer en dos o más ficheros. Si en varios ficheros se repiten grandes porciones de datos, el nivel 3 permite optimizar un poco el espacio al no exigir que los datos estén presentes varias veces.

Extensiones Rock Ridge

Como hemos visto, ISO 9660 es muy restrictivo en varios sentidos. Poco después de que salió, algunos miembros de la comunidad UNIX comenzaron a trabajar en una extensión para poder representar sistemas de ficheros UNIX en un CD-ROM. Las extensiones se llamaron Rock Ridge, en memoria de un pueblo ficticio que aparece en la película de Gene Wilder, *Blazing Saddles*, quizá porque a uno de los miembros de la comunidad le gustaba la película.

Las extensiones aprovechan el campo *Uso del sistema* para hacer que los CD-ROMs Rock Ridge puedan leerse en cualquier ordenador. Todos los demás campos conservan el significado que tienen en ISO 9660. Si un sistema no reconoce las extensiones Rock Ridge, tan solo hará caso omiso de ellas y verá un CD-ROM normal.

Las extensiones se dividen en los campos siguientes:

1. PX – Atributos POSIX.
2. PN – Números de dispositivo principal y secundario.
3. SL – Enlace simbólico.
4. NM – Nombre alterno.
5. CL – Ubicación de hijo.
6. PL – Ubicación de padre.
7. RE – Reubicación.
8. TF – Sellos de tiempo.

El campo *PX* contiene los bits de autorización *rwxrwxrwx* estándar de UNIX para el dueño, grupo y otros. También contiene los demás bits contenidos en la palabra de modo, como los bits SETUID y SETGID, etcétera.

El campo *PN* está presente para poder representar dispositivos puros en un CD-ROM. Éste contiene los números de dispositivo principal y secundario asociados con el fichero. De este modo, podrá escribirse el contenido del directorio */dev* en un CD-ROM, pudiendo después reconstruirlo correctamente en el sistema de destino.

El campo *SL* es para enlaces simbólicos, permite que un fichero de un sistema de ficheros se refiera a un fichero en otro sistema de ficheros distinto.

Tal vez el campo más importante sea *NM*, que permite asociar un segundo nombre al fichero. Este nombre no está sujeto a las restricciones de ISO 9660 en cuanto a conjunto de caracteres y longitud, y permite expresar nombre de fichero UNIX arbitrarios en un CD-ROM.

Los tres campos que siguen se utilizan juntos para evitar el límite impuesto por ISO 9660 de solo poder anidar directorios hasta una profundidad de ocho. Si se utilizan, es posible especificar que un directorio debe reubicarse e indicar en qué lugar de la jerarquía debe ir. Esta es una manera artificial de superar la limitación en la profundidad de los directorios.

Por último, el campo *TF* contiene los tres sellos de tiempo incluidos en cada nodo-i de UNIX, a saber, la hora de creación, la hora de la última modificación y la del último acceso. Juntas estas extensiones permiten copiar un sistema de ficheros UNIX en un CD-ROM y luego restaurarlo por completo en otro sistema.

Extensiones Joliet

La comunidad UNIX no fue el único grupo que quería una forma de extender ISO 9660. A Microsoft también le pareció demasiado restrictivo (aunque fue precisamente el propio MS-DOS de Microsoft el que obligó a imponer la mayoría de las restricciones). Por tanto, Microsoft inventó ciertas extensiones que recibieron el nombre de **Joliet**. Se diseñaron para poder copiar sistemas de ficheros Windows en CD-ROM y luego restaurarlos, exactamente de la misma forma que se diseñó Rock Ridge para UNIX. Casi todos los programas que se ejecutan bajo Windows y utilizan CD-ROM reconocen Joliet, incluyendo los programas que queman CD-R. Por lo regular estos programas permiten escoger entre los distintos niveles de ISO 9660 y Joliet.

Las principales extensiones que ofrece Joliet son:

1. Nombre largos de fichero.
2. Conjunto de caracteres Unicode.
3. Anidación de directorios a más de ocho niveles.
4. Nombre de directorio con extensiones.

La primera extensión permite nombre de fichero de hasta 64 caracteres. La segunda permite utilizar el conjunto de caracteres Unicode en los nombre de fichero. Esta extensión es

importante para software destinado a utilizarse en países que no utilizan el alfabeto latino, como Japón, Israel y Grecia. Puesto que los caracteres Unicode ocupan dos bytes, la longitud máxima de un nombre de fichero en Joliet es de 128 bytes.

Al igual que Rock Ridge, Joliet elimina la limitación respecto a anidamiento de directorios. Los directorios pueden anidarse hasta cualquier profundidad que se requiera. Por último. Los nombres de directorio pueden tener extensiones. No queda claro por qué se incluyó esta extensión, pues en Windows los directorios casi nunca llevan extensiones, pero quizás algún día lo harán.

6.4.2 El sistema de ficheros de CP/M

Los primeros ordenadores personales (entonces llamados microordenadores) salieron a principios de la década de 1980. De esos primeros microordenadores, uno de los más populares utilizaba la CPU 8080 de Intel, de 8 bits, tenía 4 KB de RAM y contaba con un solo disco flexible de 8 pulgadas, con capacidad para 180 KB. Versiones posteriores utilizaron la CPU Zilog Z80, un poco más elaborada (pero todavía de 8 bits), tenían hasta 64 KB de RAM y utilizaban disquetes con la descomunal capacidad de 720 KB como dispositivo de almacenamiento masivo. A pesar de la baja velocidad y la pequeña cantidad de RAM, casi todas esas máquinas ejecutaban un sistema operativo basado en disco de una potencia sorprendente, llamado **CP/M** (Programa de Control para Microordenadores) (Goleen y Pechura, 1986). Este sistema dominó su época tanto como MS-DOS y después Windows dominaron el mundo de los PCs de IBM. Dos décadas después, ha desaparecido sin dejar huella (con la excepción de un reducido grupo de aficionados de hueso colorado), lo que permite pensar que los sistemas que ahora dominan el mundo podrían ser casi desconocidos cuando los bebés actuales se conviertan en estudiantes universitarios (¿Windows qué?).

Vale la pena echar un vistazo a CP/M por varias razones. La primera es que desde una perspectiva histórica, fue un sistema muy importante y fue el antepasado directo de MS-DOS. La segunda es que es probable que los diseñadores de sistemas operativos actuales y futuros que piensan que un ordenador necesita 32 MB tan solo para arrancar el sistema operativo podrían aprender mucho en cuanto a sencillez de un sistema que operaba de forma muy satisfactoria en 16 KB de RAM. La tercera es que, en las décadas por venir, van a ser muy comunes los sistemas empotrados. Debido a las restricciones de coste, espacio, peso y consumo de electricidad, los sistemas operativos empleados en, por ejemplo, relojes, cámaras, radios y teléfonos celulares van a tener que ser esbeltos y ágiles, como CP/M. Claro que esos sistemas no tienen disquetes de 8 pulgadas, pero bien podrían tener discos electrónicos que utilizan memoria flash, y es sencillo construir un sistema de ficheros de tipo CP/M en un dispositivo así.

La organización de CP/M en la memoria se muestra en la Figura 6-30. En la parte más alta de la memoria principal (en RAM) está el BIOS, que contiene una biblioteca básica de 17 llamadas de E/S utilizadas por CP/M (en esta sección describiremos CP/M 2.2, que fue la versión estándar cuando CP/M estaba en el céñit de su popularidad). Estas llamadas leen y escriben en el teclado, la pantalla y el disquete.

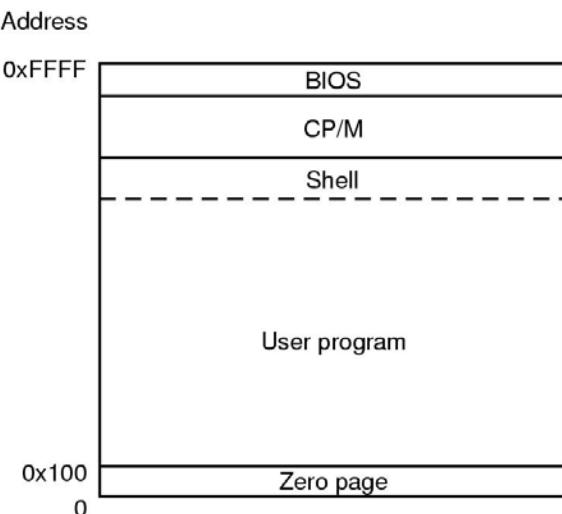


Figura 6-30. Organización de la memoria en CP/M.

Justo debajo del BIOS está el sistema operativo propiamente dicho. El tamaño del sistema operativo en CP/M 2.2 es de 3584 bytes. Increíble pero cierto: un sistema operativo completo en menos de 4 KB. Debajo del sistema operativo está el shell (procesador de la línea de comandos), que ocupa otros 2 KB. El resto de la memoria es para programas de usuario, con excepción de los 256 bytes de hasta abajo, que se reservan para los vectores de interrupción del hardware, unas cuantas variables y un búfer para la línea de comandos actual, con el fin de que los programas de usuario tengan acceso a ella.

El motivo para separar el BIOS de CP/M en sí (aunque ambos están en la RAM) fue la portabilidad. CP/M sólo interactúa con el hardware emitiendo llamadas al BIOS. Para portar CP/M a otra máquina, sólo es necesario trasladar ahí el BIOS. Una vez hecho eso, podrá instalarse CP/M sin modificación.

Un sistema CP/M sólo tiene un directorio, que contiene entradas de tamaño fijo (32 bytes). El tamaño del directorio, aunque fijo para una implementación dada, podría ser distinto en otras implementaciones de CP/M. Todos los ficheros del sistema aparecen en este directorio. Después de que CP/M arranca, lee el directorio y calcula un mapa de bits que contiene los bloques de disco libres, viendo qué bloques no están en ningún fichero. Este mapa de bits, que sólo ocupa 23 bytes para un disco de 180 KB, se mantiene en la memoria durante la ejecución. En el momento de apagar el sistema el mapa se desecha; es decir, no se escribe en el disco. Esto elimina la necesidad de un verificador de consistencia del disco (como *fsck*) y ahorra un bloque en el disco (equivalente en porcentaje al ahorro de 90 MB en un disco moderno de 16 GB).

Cuando el usuario teclea un comando, lo primero que hace el *shell* es copiarlo en un búfer en los 256 bytes más bajos de la memoria. Luego busca el programa a ejecutar, lo carga en la memoria en la dirección 256 (a continuación de los vectores de interrupción) y salta a él. Luego comienza la ejecución del programa, el cual descubre sus argumentos examinando el búfer de la línea de comandos. El programa puede sobrescribir el *shell* si necesita la memoria. Cuando termina el programa, emite una llamada al sistema CP/M para pedirle que vuelva a cargar el *shell* (si lo sobrescribió) y lo ejecute. En pocas palabras, así es como funciona CP/M.

Además de cargar programas, CP/M ofrece 38 llamadas al sistema, en su mayoría servicios de ficheros, para los programas de usuario. Las llamadas más importantes son las que leen y escriben ficheros. Para poder leer de un fichero es necesario abrirlo. Cuando CP/M recibe una llamada al sistema **open**, tiene que leer el único directorio y buscar en él el fichero. El

directorio no se mantiene en la memoria todo el tiempo, para ahorrar la escasa RAM. Cuando CP/M encuentra la entrada, tiene de inmediato los números de bloques del disco, porque están almacenados ahí mismo en la entrada, al igual que todos los atributos. En la Figura 6-31 se muestra el formato de una entrada de directorio.

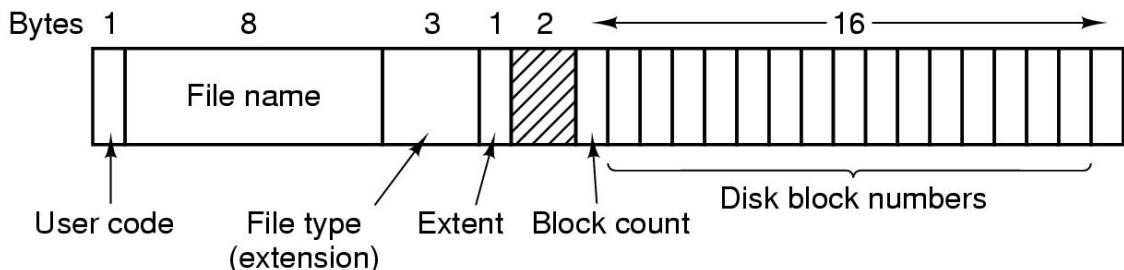


Figura 6-31. Formato de una entrada de directorio en CP/M.

Los campos de la Figura 6-31 se utilizan para lo siguiente: el campo *Código de usuario* indica quién es el dueño del fichero. Aunque sólo una persona puede trabajar con CP/M en un momento dado, el sistema reconoce múltiples usuarios que se turnan para utilizar el sistema. Al buscar un nombre de fichero, sólo se eliminan las entradas que pertenecen al usuario actual. En efecto, cada usuario tiene un directorio virtual sin el gasto adicional de administrar múltiples directorios.

Los dos campos que siguen dan el nombre y la extensión del fichero. El nombre base tiene hasta ocho caracteres; puede haber una extensión opcional de hasta tres caracteres. Sólo se permiten letras mayúsculas, dígitos y un número reducido de caracteres especiales en los nombres de fichero. Este esquema de 8 + 3 empleando sólo mayúsculas fue adoptado después por MS-DOS.

El campo *Número de bloques* indica cuántos bytes tiene el fichero medido en unidades de 128 bytes (porque la E/S se efectúa en sectores físicos de 128 bytes). El último bloque de 1 KB podría no estar lleno, así que el sistema no tiene forma de determinar el tamaño exacto de un fichero. Corresponde al usuario colocar un marcador de FIN de FICHERO si lo desea. Los últimos 16 campos contienen los números de bloques de disco en sí. Cada bloque ocupa 1 KB así que el tamaño máximo del fichero es de 16 KB. Cabe señalar que la E/S física se efectúa en sectores de 128 bytes y se lleva el control del tamaño en sectores, pero los bloques de los ficheros se asignan en unidades de 1 KB (ocho sectores a la vez) para evitar que la entrada de directorio sea demasiado grande.

No obstante, los diseñadores de CP/M se dieron cuenta de que algunos ficheros, incluso en un disquete de 180 KB, podrían exceder 16 KB, por lo que se incorporó una forma de soslayar el límite de 16 KB. Un fichero que tiene entre 16 KB y 32 KB no utiliza una entrada de directorio, sino dos. La primera contiene los primeros 16 bloques; la segunda contiene los otros 16. Más allá de 32 KB, se utiliza una tercera entrada de directorio, y así de forma sucesiva. El campo *Extensión* lleva el control del orden de las entradas de directorio para que el sistema sepa qué 16 KB vienen primero, cuáles vienen después, etcétera.

Después de una llamada `open`, se conocen las direcciones de todos los bloques de disco, así que la lectura es directa. La llamada `write` es también sencilla. Sólo se requiere asignar un bloque libre del mapa de bits que está en la memoria y luego escribir el bloque. Los bloques consecutivos de un fichero no se colocan en bloques consecutivos en el disco, porque el 8080 no puede procesar una interrupción y comenzar a leer el siguiente bloque a tiempo. Por ello, se utiliza intercalamiento para poder leer varios bloques en una única rotación.

Es evidente que CP/M no es el último grito de la moda en cuanto a los sistemas de ficheros avanzados, pero es sencillo, rápido y un programador competente puede implementarlo en menos de una semana. Para muchas aplicaciones integradas, bien podría ser todo lo que se necesita.

6.4.3 El sistema de ficheros de MS-DOS

Como primera aproximación, MS-DOS es una versión mejor y más grande de CP/M. Sólo se ejecuta en plataformas Intel, no maneja multiprogramación y sólo opera en el modo real del PC (que en un principio era el único modo). El *shell* tiene más funciones y hay más llamadas al sistema, pero la función básica del sistema operativo sigue siendo cargar programas, manejar el teclado y la pantalla, y administrar el sistema de ficheros. Esta última funcionalidad es la que nos interesa aquí.

El sistema de ficheros de MS-DOS sigue de cerca el patrón del sistema de CP/M, incluyendo la utilización de nombres de fichero de 8 + 3 caracteres (mayúsculas). La primera versión (MS-DOS 1.0) estaba limitada a un único directorio, igual que CP/M. Sin embargo, a partir de MS-DOS 2.0 se expandió de manera considerable la funcionalidad del sistema de ficheros. La principal mejora fue la inclusión de un sistema de ficheros jerárquico en el que los directorios podían anidarse hasta una profundidad arbitraria. Esto implicaba que el directorio raíz (que seguía teniendo un tamaño máximo fijo) podía contener subdirectorios, y estos podían contener otros subdirectorios, *ad infinitum*. No se permitían enlaces al estilo UNIX, así que el sistema de ficheros formaba un árbol a partir del directorio raíz.

Es común que los programas de aplicación creen un subdirectorio en el directorio raíz y coloquen allí todos sus ficheros (o en subdirectorios de ese subdirectorio), para evitar conflictos entre las distintas aplicaciones. Puesto que los directorios mismos se almacenan como ficheros, no hay límite para el número de directorios o ficheros que es posible crear. Sin embargo, a diferencia de CP/M, no existe el concepto de usuarios distintos en MS-DOS. Por ello, el usuario, que comenzó la sesión tiene acceso a todos los ficheros.

Para leer un fichero, un programa MS-DOS debe emitir primero una llamada al sistema *open* para obtener un identificador de fichero. La llamada especifica un camino, que podría ser absoluto o relativo al directorio de trabajo actual. El camino se examina componente a componente hasta que se encuentra el directorio final y se carga en la memoria. Luego se busca en él el fichero que se abrirá.

Aunque los directorios en MS-DOS tienen tamaño variable, igual que en CP/M, sus entradas son de tamaño fijo, 32 bytes. El formato de una entrada de directorio en MS-DOS se muestra en la Figura 6-32, contiene el nombre del fichero, sus atributos, la fecha y la hora en que se creó, el bloque inicial y el tamaño exacto del fichero. Los nombres de fichero de menos de 8 + 3 caracteres se ajustan a la izquierda y se llenan con espacios a la derecha, por separado para cada campo. El campo *Atributos* es nuevo y contiene bits para indicar que el fichero es de solo lectura, que necesita salvaguardarse, que está oculto o que es un fichero del sistema. Los ficheros de sólo lectura no pueden escribirse. Esto se hace para protegerlos frente a daños accidentales. El bit de salvaguarda no tiene otra función real dentro del sistema operativo (es decir, MS-DOS no lo examina ni modifica su valor). La intención es que los programas a nivel de usuario puedan ponerlo a 0 tras hacer una copia de seguridad del fichero y que otros programas lo pongan a 1 cuando modifiquen el fichero. De este modo, un programa de backup puede examinar ese bit de atributo en todos los ficheros para ver cuáles son los ficheros que hay que copiar. El bit de fichero oculto puede ponerse a 1 para evitar que el fichero aparezca en los listados de directorio. Su principal uso es evitar que los usuarios novatos se confundan con ficheros que quizás no entiendan. Por último, el bit del sistema también oculta los ficheros.

Además, los ficheros del sistema no pueden borrarse por accidente con el comando *del*. Los principales componentes de MS-DOS tienen puesto ese bit a 1.

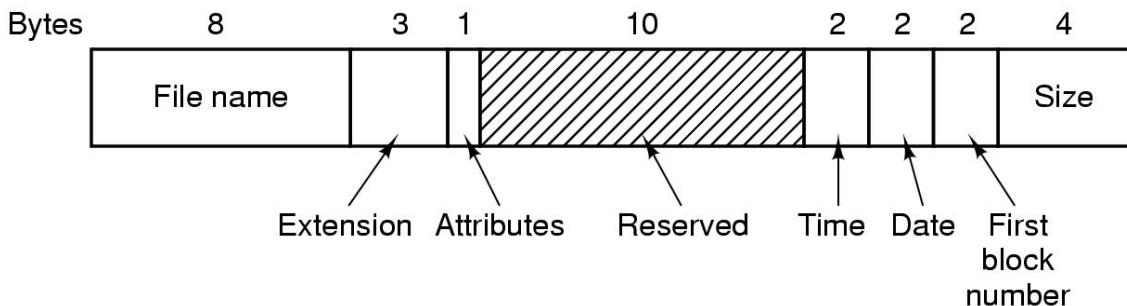


Figura 6-32. Una entrada del directorio de MS-DOS.

La entrada de directorio contiene también la hora y la fecha de creación o de última modificación del fichero. La hora tiene exactitud de ± 2 segundos porque se almacena en un campo de dos bytes, que sólo puede contener 65536 valores únicos (un día contiene 86400 segundos distintos). El campo de hora se subdivide en segundos (5 bits), minutos (6 bits) y horas (5 bits). La fecha cuenta en días, utilizando tres subcampos: día (5 bits), mes (4 bits) y año-1980 (7 bits). Con un número de siete bits para el año y el tiempo comenzando en 1980, el año más alto que se puede expresar es 2107. Por tanto, MS-DOS tiene incorporado un problema del año 2108. Para evitar una catástrofe, los usuarios de MS-DOS deben tomar medidas lo antes posible para que sus programas tomen eso en cuenta. Si MS-DOS hubiera utilizado los campos de fecha y hora combinados en forma de un contador de segundos de 32 bits, podría haber representado cada segundo con exactitud y haber aplazado la catástrofe hasta 2116.

A diferencia de CP/M, que no almacena el tamaño exacto del fichero, MS-DOS sí lo hace. Puesto que se utiliza un número de 32 bits para el tamaño, en teoría los ficheros pueden tener hasta 4 GB. Sin embargo, otros límites (que describiremos a continuación) restringen el tamaño máximo de los ficheros a 2 GB o menos. Una porción sorprendentemente grande de la entrada (10 bytes) no se utiliza.

Otra diferencia entre MS-DOS y CP/M es que MS-DOS no almacena las direcciones de disco de un fichero en su entrada de directorio, tal vez porque los diseñadores se dieron cuenta de que los discos duros grandes (comunes entonces en miniordenadores) llegarían algún día al mundo de MS-DOS. En vez de eso, MS-DOS lleva el control de los bloques de fichero mediante una tabla de asignación de ficheros (FAT) en la memoria principal. La entrada de directorio contiene el número del primer bloque del fichero. Este número se utiliza como un índice para consultar una FAT de 64K entradas en la memoria principal. Siguiendo la cadena, es posible localizar todos los bloques. El funcionamiento de la FAT se ilustra en la Figura 6-14.

El sistema de ficheros FAT viene en tres versiones para MS-DOS: FAT-12, FAT-16 y FAT 32, dependiendo del número de bits que tenga una dirección de disco. En realidad, FAT-32 es un nombre engañoso porque sólo se utilizan los 28 bits de menor peso de las direcciones de disco. Se debería haber llamado FAT-28, pero las potencias de 2 suenan mucho más elegantes.

En todas las FATs, el bloque de disco puede definirse como algún múltiplo de 512 bytes (y puede ser diferente para cada partición), y el conjunto de tamaños de bloque permitidos (llamados **tamaños de cluster** por Microsoft) es diferente para cada variante. La primera versión de MS-DOS utilizaba FAT-12 con bloques de 512 bytes, lo que daba un tamaño de partición máximo de $2^{12} \times 512$ bytes (en realidad, sólo 4086×512 , porque 10 de las direcciones de disco se utilizaban como marcadores especiales: fin de fichero, bloque defectuoso, etcétera).

Con estos parámetros, el tamaño máximo de una partición de disco era de aproximadamente 2 MB, y el tamaño de la FAT en la memoria era de 4096 entradas de dos bytes cada una. La utilización de una entrada de tabla de 12 bits habría resultado demasiado lento.

Este sistema funcionaba bien con discos flexibles, pero cuando salieron los discos duros se convirtió en un problema, que Microsoft resolvió permitiendo tamaños de bloque adicionales de 1, 2 y 4 KB. Este cambio conservó la estructura y el tamaño de la FAT-12, pero permitió particiones de disco de hasta 16 MB.

Puesto que MS-DOS reconocía cuatro particiones por unidad de disco, el nuevo sistema de ficheros FAT-12 funcionaba con discos de hasta 64 MB. Más allá de esa capacidad, algo tenía que ceder. Lo que sucedió fue la introducción de FAT-16, con punteros de disco de 16 bits. Además, se permitieron tamaños de bloque de 8, 16 y 32 KB. (32768 es la potencia de 2 más grande que puede representarse con 16 bits) La tabla FAT-16 ocupaba ahora 128 KB de la memoria principal todo el tiempo, pero como ya había memorias más grandes, entonces se empezó a utilizar más y pronto sustituyó al sistema de ficheros FAT-12. La partición de disco más grande que se puede manejar con FAT-16 es de 2 GB (64 K entradas de 32 KB cada una) y el disco más grande es de 8 GB, o sea, cuatro particiones de 2 GB cada una.

Para cartas de negocios, ese límite no representa ningún problema, pero para almacenar vídeo digital empleando el estándar DV, un fichero de 2 GB apenas contiene nueve minutos de vídeo. Una consecuencia del hecho de que un disco de PC sólo puede manejar cuatro particiones, es que el vídeo más largo que puede almacenarse en un disco tiene una duración de cerca de 38 minutos, por más grande que sea el disco. Este límite también implica que el vídeo más grande que puede editarse en línea es de menos de 19 minutos, ya que se necesita tanto un fichero de entrada como uno de salida.

A partir de la segunda versión de Windows 95, se introdujo el sistema de ficheros FAT-32, con sus direcciones de disco de 28 bits, y la versión de MS-DOS que era la base de Windows 95 se adaptó para manejar FAT-32. En este sistema, las particiones podían ser de en teoría de $2^{28} \times 2^{15}$ bytes, pero en realidad están limitadas a 2 TB (2048 GB) porque internamente el sistema lleva el control de los tamaños de las particiones en sectores de 512 bytes empleando un número de 32 bits, y $2^9 \times 2^{32}$ es 2 TB. En la Figura 6-33 se muestra el tamaño máximo de las particiones con diferentes tamaños de bloque para los tres tipos de FAT.

| Block size | FAT-12 | FAT-16 | FAT-32 |
|-------------------|---------------|---------------|---------------|
| 0.5 KB | 2 MB | | |
| 1 KB | 4 MB | | |
| 2 KB | 8 MB | 128 MB | |
| 4 KB | 16 MB | 256 MB | 1 TB |
| 8 KB | | 512 MB | 2 TB |
| 16 KB | | 1024 MB | 2 TB |
| 32 KB | | 2048 MB | 2 TB |

Figura 6-33. Tamaño máximo de las particiones con diferentes tamaños de bloque. Los cuadros vacíos representan combinaciones prohibidas.

Además de manejar discos más grandes, el sistema de ficheros FAT-32 tiene otras dos ventajas respecto a FAT-16. Primera, un disco de 8 GB que utiliza FAT-32 puede tener una sola partición. Si se utiliza FAT-16 tiene que haber cuatro particiones, las cuales se presentan al usuario de Windows como las unidades de disco lógicas C:, D:, E: y F:. Corresponde al usuario decidir qué ficheros colocará en qué discos, y llevar el control de dónde está cada cosa.

La otra ventaja de FAT-32 respecto a FAT-16 es que, para un tamaño de partición dado, puede utilizarse un tamaño de bloque más pequeño. Por ejemplo, con particiones de disco de 2 GB, FAT-16 tiene que utilizar bloques de 32 KB, pues de lo contrario no podría cubrir toda la partición con las 64K direcciones de disco de que dispone. En contraste, FAT-32 puede usar, por ejemplo, bloques de 4 KB en una partición de 2 GB. La ventaja de utilizar un tamaño de bloque más pequeño es que la mayoría de los ficheros ocupan mucho menos de 32 KB. Si el tamaño de bloque es de 32 KB, un fichero de 10 bytes ocupará 32 KB de espacio en disco. Si el fichero medio tiene, digamos, 8 KB, y se utilizan bloques de 32 KB, se desperdiciarán 3/4 partes del disco, lo cual no es una forma muy eficiente de utilizarlo. Con ficheros de 8 KB y bloques de 4 KB, no habrá desaprovechamiento del disco, pero el precio que se paga es más RAM ocupada por la FAT. Con bloques de 4 KB y particiones de disco de 2 GB, habrá 512K bloques, así que la FAT deberá tener 512K entradas en la memoria (que ocupan 2 MB de RAM).

MS-DOS utiliza la FAT para llevar el control de los bloques de disco libres. Cualquier bloque no asignado se marca con un código especial. Cuando MS-DOS necesita un nuevo bloque de disco, busca en la FAT una entrada que contenga ese código. Por tanto, no se requiere mapa de bits ni lista de bloques libres.

6.4.4 El sistema de ficheros de Windows 98

La versión original de Windows 95 utilizaba el sistema de ficheros de MS-DOS, incluyendo nombres de fichero de 8 + 3 caracteres y los sistemas de ficheros FAT-12 y FAT-16. A partir de la segunda versión de Windows 95 se permitieron nombres de fichero de más de 8 + 3 caracteres. Además, se introdujo la FAT-32, sobre todo para poder tener particiones de disco de más de 2 GB y discos de más de 8 GB, que ya habían salido a la venta. Tanto los nombres de fichero largos como la FAT-32 se utilizaron en Windows 98 de la misma forma que en la segunda versión de Windows 95. A continuación describiremos estas características del sistema de ficheros de Windows 98, que se han llevado también a Windows Me.

Puesto que los nombres de fichero largos son más emocionantes para los usuarios que la estructura de la FAT, los examinaremos primero. Una forma de introducir nombres de fichero largos habría sido inventar una nueva estructura de directorio. El problema con ese método es que, si Microsoft lo hubiera hecho, quienes todavía estaban en proceso de convertir Windows 3 a Windows 95 o Windows 98 no hubieran podido tener acceso a sus ficheros desde ambos sistemas. Se tomó una decisión política dentro de Microsoft de que los nombres creados utilizando Windows 98 debían ser también accesibles desde Windows 3 (para las máquinas de arranque doble). Esta restricción obligó a adoptar un diseño para manejar nombres de fichero largos, que fuera compatible con el viejo sistema de nombres 8 + 3 de MS-DOS. Puesto que tales restricciones de compatibilidad hacia atrás no son inusitadas en la industria de los ordenadores, vale la pena ver los pormenores de la forma en la que Microsoft logró su objetivo.

El efecto de esta decisión de ser compatible hacia atrás implicó que la estructura de directorios de Windows 98 tenía que ser compatible con la de MS-DOS. Como vimos, dicha estructura no es más que una lista de entradas de 32 bytes, como se muestra en la Figura 6-32. Este formato se tomó directamente de CP/M (que se escribió para el 8080), lo cual demuestra que las estructuras (obsoletas) pueden persistir durante mucho tiempo en el mundo de los ordenadores.

Sin embargo, ahora era posible utilizar los 10 bytes reservados de las entradas de la Figura 6-32, y eso fue lo que se hizo, como se aprecia en la Figura 6-34. Este cambio nada tiene que ver con los nombres largos, pero se utiliza en Windows 98, así que vale la pena entenderlo.

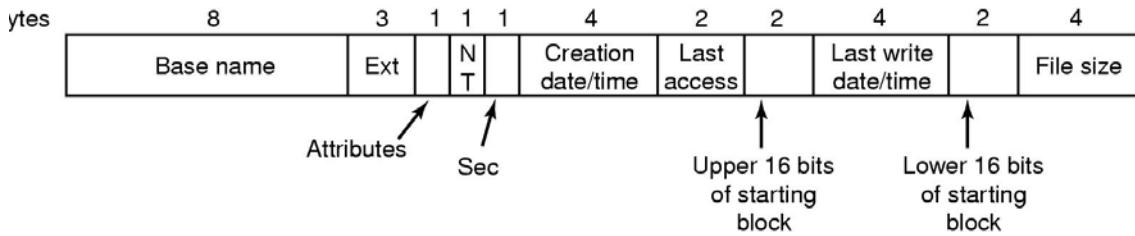


Figura 6-34. La entrada de directorio MS-DOS empleada en Windows 98.

Los cambios consisten en la adición de cinco campos nuevos donde solían estar los 10 bytes desocupados. El campo *NT* sirve sobre todo para asegurar cierta compatibilidad con Windows NT, en el sentido de desplegar los nombres de fichero en el caso correcto (en MS-DOS, todos los nombres de fichero están en mayúsculas). El campo *Seg* resuelve el problema de que no es posible almacenar la hora del día en un campo de 16 bits: proporciona bits adicionales para que el nuevo campo *Fecha/hora de creación* tenga una precisión de 10 ms. Otro campo nuevo es *Último acceso*, que almacena la fecha (pero no la hora) del último acceso al fichero. Por último, el cambio al sistema de ficheros FAT-32 implica que los números de bloque ahora son de 32 bits, así que se necesita un campo adicional de 16 bits para almacenar los 16 bits superiores del número de bloque inicial.

Ahora llegamos al corazón del sistema de ficheros Windows 98: cómo se representan los nombres de fichero largos de modo que sean compatibles con MS-DOS. La solución escogida consiste en asignar dos nombres a cada fichero: uno (potencialmente) largo (en Unicode, por compatibilidad con Windows NT) y un nombre 8 + 3 por compatibilidad con MS-DOS. Se puede tener acceso a los ficheros con cualquiera de los dos nombres. Cuando se crea un fichero cuyo nombre no obedece a las reglas de MS-DOS (longitud 8 + 3, nada de Unicode, conjunto de caracteres limitado, nada de espacios, etcétera), Windows 98 inventa un nombre MS-DOS para el fichero según cierto algoritmo. La idea básica es tomar los primeros seis caracteres del nombre convertirlos a mayúsculas, si es necesario, y añadir ~1 para formar el nombre base. Si ese nombre ya existe, se utiliza el sufijo ~2, y así de forma sucesiva. Además, se eliminan los espacios y puntos adicionales y ciertos caracteres especiales se convierten en caracteres de subrayado. Por ejemplo, a un fichero de nombre *Lista de libros comprados* se le asigna el nombre MS-DOS *LISTAD~1*. Si después se crea un fichero con el nombre *Lista de libros vendidos*, se le asignará el nombre MS-DOS *LISTAD~2*, y así en forma sucesiva.

Todo fichero tiene un nombre MS-DOS que se almacena empleando el formato de directorio de la Figura 6-34. Si el fichero también tiene un nombre largo, ese nombre se almacena en una o más entradas de directorio que preceden inmediatamente al nombre de fichero MS-DOS. Cada entrada de nombre largo contiene hasta 13 caracteres (Unicode). Las entradas se almacenan en orden inverso, con el principio del nombre justo delante de la entrada MS-DOS y los fragmentos subsiguientes antes de ella. El formato de cada entrada de nombre largo se muestra en la Figura 6-35.

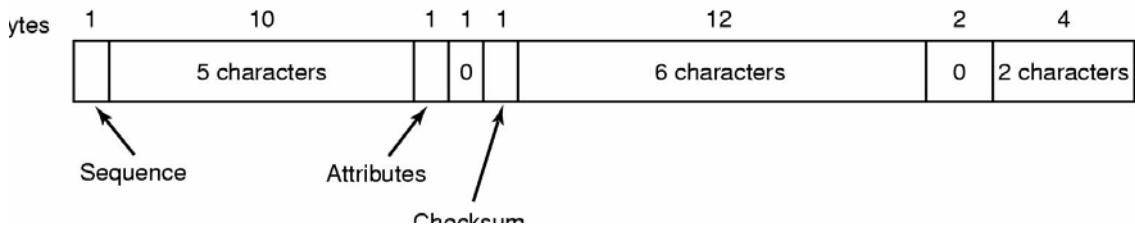


Figura 6-35. Entrada de (parte de) un nombre de fichero largo en Windows 98.

Una pregunta obvia es: “¿Cómo sabe Windows 98 si una entrada de directorio contiene un nombre de fichero MS-DOS o un (fragmento de un) nombre de fichero largo?” La respuesta está en el campo *Atributos*. En el caso de una entrada de nombre largo, este campo tiene el valor 0x0F, que representa una combinación imposible de atributos. Los programas MS-DOS antiguos que lean el directorio tan solo harán caso omiso de esa entrada, por considerarla no válida. ¡Si supieran! Los fragmentos del nombre se arman en forma consecutiva con base en el primer byte de la entrada. La última parte del nombre largo (la primera entrada de la secuencia) se marca sumando 64 al número consecutivo. Puesto que sólo se utilizan 6 bits para el número consecutivo, en teoría el tamaño máximo de los nombres de fichero es $63 \times 13 = 819$ caracteres. De hecho, los nombres están limitados a 260 caracteres por razones históricas.

Cada entrada de nombre largo contiene un campo de *Suma de comprobación* para evitar el siguiente problema. Primero, un programa de Windows 98 crea un fichero con nombre largo. Luego, se rearranca el ordenador para ejecutar Windows 3 o MS-DOS. Después, un programa viejo en ese entorno elimina el nombre de fichero MS-DOS del directorio pero no elimina el nombre largo que le precede (porque no sabe que existe). Por último, algún programa crea un fichero nuevo que reutiliza la entrada de directorio recién desocupada. Ahora tenemos una secuencia válida de entradas de nombre largo, justo antes de una entrada de fichero MS-DOS que nada tiene que ver con el nombre largo. El campo *Suma de comprobación* permite a Windows detectar esta situación, verificando que el nombre de fichero MS-DOS que sigue a un nombre largo en verdad corresponde a él. Desde luego, como sólo se utiliza un byte hay una probabilidad de 1/256 de que Windows 98 no se dé cuenta de las sustitución de ficheros.

Para ver un ejemplo de cómo funcionan los nombres largos, consideremos el ejemplo de la Figura 6-36. Aquí tenemos un fichero llamado *Carta que les escribo a mis queridos hijitos*. Con 44 caracteres, ciertamente cumple con los requisitos para ser un nombre de fichero largo. El nombre MS-DOS que se construye a partir de él es *CARTAQ~1* y se almacena en la última entrada.

La estructura de directorio incorpora cierta redundancia para ayudar a detectar problemas en caso de que un programa antiguo de Windows 3 haya hecho cosas que no debía con el directorio. El byte consecutivo en realidad no se necesita porque el byte 0x40 marca la primera entrada, pero es un ejemplo de redundancia incluida de forma deliberada. Además, el campo *Inferiores* de la Figura 6-36 (la mitad inferior del número de clúster inicial) es 0 en todas las entradas salvo la última, también para evitar que los programas antiguos lo interpreten mal y den al traste con el sistema de ficheros. El byte *NT* de la Figura 6-36 se utiliza en NT y no se toma en cuenta en Windows 98. El byte A contiene los atributos.

| | | | | | | | | | |
|-----------------|---------|-------|---------------|-----------|-----|------------|-----|------|-----|
| 68 | d o g | A 0 | C K | | | | | 0 | |
| 3 | o v e | A 0 | C K | t h e l | | | | a 0 | z y |
| 2 | w n f o | A 0 | C K | x j u m p | | | | 0 | s |
| 1 | T h e q | A 0 | C K | u i c k | | | | b 0 | r o |
| T H E Q U I ~ 1 | | A T S | Creation time | Last acc | Upp | Last write | Low | Size | |
| Bytes | | | | | | | | | |

Figura 6-36. Ejemplo de cómo se almacena un nombre largo en Windows 98.

Desde el punto de vista conceptual, la implementación del sistema de ficheros FAT-32 es similar a la del sistema de ficheros FAT-16. Sin embargo, en lugar de una tabla de 65536 entradas, hay tantas entradas como se necesiten para cubrir la parte del disco que contiene datos. Si se utiliza el primer millón de bloques, desde una perspectiva conceptual la tabla tiene un millón de entradas. Para evitar la necesidad de tenerlas todas en memoria a la vez, Windows 98 mantiene una “ventana” que ve hacia la tabla, y sólo mantiene una parte de ella en la memoria en un momento dado.

6.4.5 El sistema de ficheros de UNIX V7

Aún las primeras versiones de UNIX tenían un sistema de ficheros multiusuario relativamente elaborado, pues se derivó de MULTICS. A continuación trataremos el sistema de ficheros V7, utilizado en la PDP-11 y que hizo famoso a UNIX. Examinaremos versiones modernas en el capítulo 10.

El sistema de ficheros tiene la forma de un árbol que nace en el directorio raíz, con la adición de enlaces para formar un grafo acíclico dirigido (DAG). Los nombres de fichero tienen hasta 14 caracteres y pueden contener cualquier carácter ASCII con excepción de / (porque és es el separador de los componentes de un camino) y NUL (porque sirve para llenar los nombres de menos de 14 caracteres). NUL tiene el valor numérico 0.

Una entrada de directorio UNIX contiene una entrada para cada fichero de ese directorio. Las entradas son simples en extremo porque UNIX utiliza el esquema de i-nodos que ilustramos en la Figura 6-15. Una entrada de directorio contiene solamente dos campos: el nombre del fichero (14 bytes) y el número del i-nodo correspondiente a ese fichero (2 bytes), como se muestra en la Figura 6-37. Estos parámetros limitan el número de ficheros por sistema de ficheros a 64K.

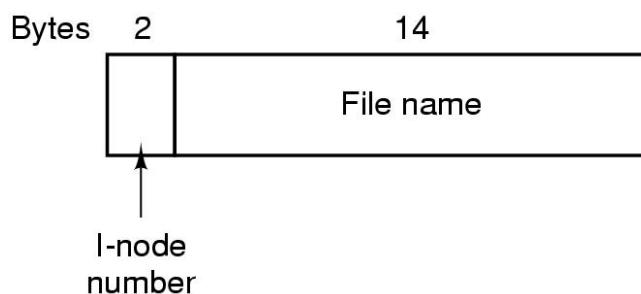


Figura 6-37. Entrada de directorio en UNIX V7.

Al igual que el i-nodo de la Figura 6-15, los i-nodos de UNIX contienen algunos atributos. Éstos incluyen el tamaño del fichero, la hora de creación, la del último acceso y la de la última modificación), propietario, grupo, información de protección y un contador del número de entradas de directorio que apuntan al i-nodo. Éste último campo es necesario para los enlaces. Cada vez que se crea un enlace nuevo con un i-nodo, se incrementa el contador en el i-nodo. Cuando se elimina un enlace, el contador se decrementa. Cuando el contador llega a 0, el i-nodo se recicla y los bloques de disco se colocan en la lista libre.

El control de los bloques de disco se lleva utilizando una generalización de la Figura 6-15 para manejar ficheros muy grandes. Las primeras 10 direcciones de disco se almacenan en el mismo i-nodo, así que en el caso de ficheros pequeños toda la información necesaria está justo en el i-nodo, que pasa del disco a la memoria principal cuando se abre el fichero. Si los ficheros son algo más grandes, una de las direcciones que están en el i-nodo es la dirección de un bloque de disco llamado **bloque indirecto simple** o **bloque indirecto** a secas. Este bloque contiene más direcciones de disco. Si todavía no son suficientes, otra dirección en el i-nodo, denominada **bloque indirecto doble**, contiene la dirección de un bloque que contiene una lista de bloques indirectos. Si ni siquiera esto es suficiente, puede utilizarse también un **bloque indirecto triple**. El panorama completo se presenta en la Figura 6-38.

Cuando se abre un fichero, el sistema de ficheros debe tomar el nombre del fichero proporcionado y localizar sus bloques de disco. Consideremos cómo se busca el nombre de camino */usr/ast/correo*. Utilizaremos UNIX como ejemplo, pero el algoritmo es básicamente el mismo en todos los sistemas de directorios jerárquicos. Primero el sistema de ficheros localiza el directorio raíz. En UNIX, el i-nodo del directorio raíz está en un lugar fijo del disco. A partir de este i-nodo, se localiza el directorio raíz, que puede estar en cualquier lugar del disco, pero digamos que en este caso está en el bloque 1.

Luego se busca el primer componente del camino, *usr*, en el directorio raíz para hallar el número de i-nodo del fichero */usr*. Localizar un i-nodo teniendo su número es fácil, porque todos tienen una posición fija en el disco. A partir de ese i-nodo, el sistema localiza el directorio de */usr* y busca en él el siguiente componente, *ast*. Al encontrar la entrada de *ast*, tendrá el i-nodo del directorio */usr/ast*. A partir de ese i-nodo se encuentra el directorio en sí y se busca *correo*. Luego se lee el i-nodo de ese fichero y se coloca en la memoria, donde se mantiene hasta que se cierre el fichero. El proceso de búsqueda se ilustra en la Figura 6-39.

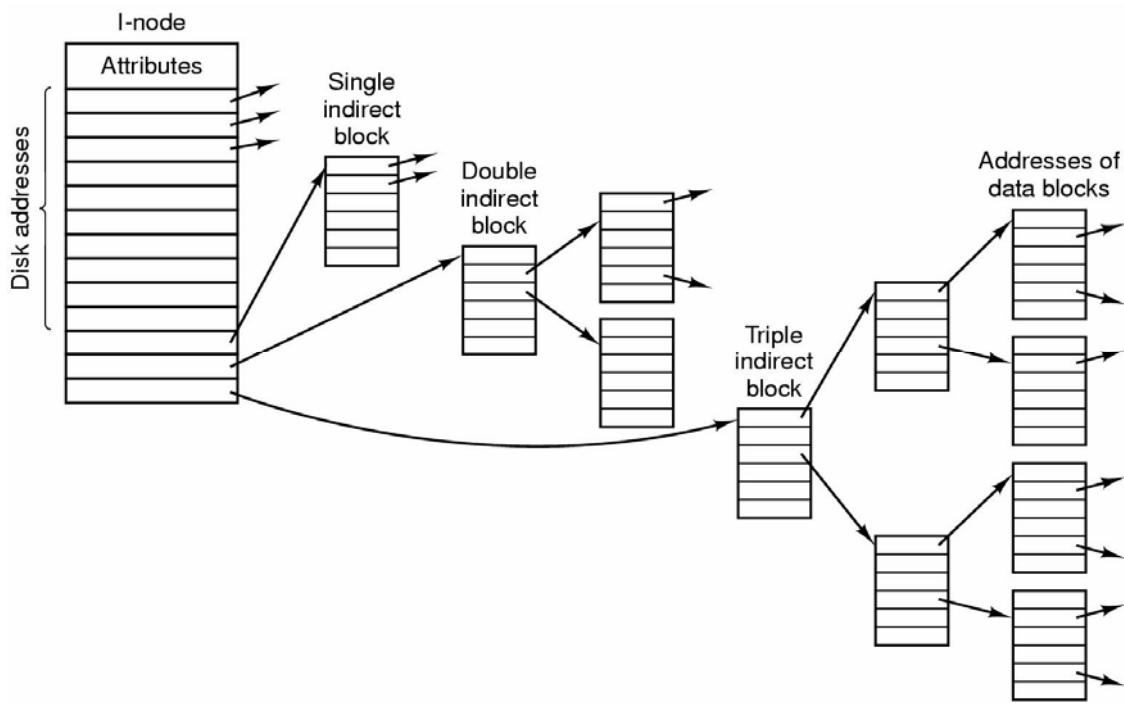


Figura 6-38. i-nodo de UNIX.

| | | | | | |
|----------------|-----|-------------------------|---|---------------------------------|--|
| Root directory | | I-node 6 is for /usr | Block 132 is /usr directory | I-node 26 is for /usr/ast | Block 406 is /usr/ast directory |
| 1 | . | Mode size times | 6 • 1 .. 19 dick 30 erik 51 jim 26 ast 45 bal | Mode size times | 26 • 6 .. 64 grants 92 books 60 mbox 81 minix 17 src |
| 1 | .. | 132 | | 406 | |
| 4 | bin | | | | |
| 7 | dev | | | | |
| 14 | lib | | | | |
| 9 | etc | | | | |
| 6 | usr | | | | |
| 8 | tmp | | | | |

Looking up
 usr yields
 i-node 6

I-node 6
 says that
 /usr is in
 block 132

/usr/ast
 is i-node
 26

I-node 26
 says that
 /usr/ast is in
 block 406

/usr/ast/mbox
 is i-node
 60

Figura 6-39. Pasos para buscar /usr/ast/correo.

Los nombres de camino relativos se buscan igual que los absolutos, sólo que partiendo del directorio de trabajo en lugar del directorio raíz. Todo directorio tiene entradas para . y .., que se colocan ahí cuando se crea el directorio. La entrada . tiene el número de i-nodo del directorio actual, y la entrada .. tiene el número de i-nodo del directorio padre. Así, un procedimiento que busca `..../luis/prog.c` simplemente consulta .. en el directorio de trabajo, halla el número de i-nodo del directorio padre y busca *luis* en ese directorio. No se necesita ningún mecanismo especial para manejar estos nombres. En lo que concierne al sistema de directorios, son sólo cadenas ASCII ordinarias, como cualquier otro nombre.

6.5 INVESTIGACIÓN SOBRE SISTEMAS DE FICHEROS

Los sistemas de ficheros han atraído siempre más investigadores que otras partes del sistema operativo, y sigue siendo así. Algunas de las investigaciones tienen que ver con la estructura de los sistemas de ficheros. Los sistemas de ficheros con estructura de registro y temas afines son populares (Matthews y otros, 1997 y Wang y otros, 1999). El disco lógico divide el sistema de ficheros en dos capas distintas: el sistema de ficheros y el sistema de disco (De Jorge y otros, 1993). La construcción de un sistema de ficheros a partir de capas apilables también es tema de investigación (Heidemann y Popek, 1994).

Los sistemas de ficheros extensibles son análogos a los kernels extensibles que vimos en el capítulo 1. Con ellos es posible añadir nuevas funciones al sistema de ficheros sin tener que rediseñarlo desde cero (Karpovich y otros, 1994 y Khalidi y Nelson, 1993).

Otro tema de investigación que ha alcanzado cierta popularidad es la medición del contenido y el uso de los sistemas de ficheros. Se ha medido la distribución de tamaños de los ficheros, la longevidad de los ficheros, el acceso equitativo a todos los ficheros, la comparación entre lecturas y escrituras, y muchos otros parámetros (Douceur y Bolosky, 1999; Gill y otros, 1994; Roselli y Lorch, 2000, y Vogels, 1999).

Otros investigadores han examinado el rendimiento de los sistemas de ficheros y la forma de mejorarlo utilizando preextracción, cachés, menos copiado y otras técnicas. Normalmente, estos investigadores realizan mediciones, averiguan dónde están los cuellos de botella, eliminan por lo menos uno de ellos y luego realizan las mediciones en el sistema mejorado para validar sus resultados (Cao y otros, 1995; Pai y otros, 2000, y Patterson y otros, 1995).

Un tema en el que pocos piensan hasta que sucede un desastre es el de los backups y la recuperación de los sistemas de ficheros. Aquí también han surgido algunas ideas nuevas acerca de cómo hacer mejor las cosas (Chen y otros, 1996; Devarakonda y otros, 1996, y Hutchinson y otros, 1999). Un tanto relacionada con este tema está la cuestión de qué hacer cuando un usuario borra un fichero: ¿eliminarlo o ocultarlo? El sistema de ficheros Elephant, por ejemplo, nunca olvida (Santry y otros, 1999a y Santry y otros, 1999b).

6.6 RESUMEN

Visto desde fuera, un sistema de ficheros es una colección de ficheros y directorios, además de operaciones con ellos. Los ficheros pueden leerse y escribirse, los directorios pueden crearse y destruirse, y los ficheros pueden cambiarse de un directorio a otro. Casi todos los sistemas de ficheros modernos manejan un sistema de directorios jerárquico en el que los directorios pueden tener subdirectorios y éstos pueden tener subdirectorios, *ad infinitum*.

Visto desde dentro, un sistema de ficheros es muy diferente. Los diseñadores del sistema de ficheros tienen que decidir cómo se asigna el espacio de almacenamiento y cómo se mantiene el sistema al tanto de qué bloque corresponde a qué fichero. Entre las posibilidades están los ficheros contiguos, las listas enlazadas, las tablas de asignación de ficheros y los i-nodos. Los distintos sistemas tienen diferentes estructuras de directorio. Los atributos pueden colocarse en los directorios o en otro lado (por ejemplo en un i-nodo). El espacio de disco puede administrarse utilizando listas libres o mapas de bits. La fiabilidad del sistema de ficheros aumenta si se realizan volcados incrementales y si se utiliza un programa para reparar sistemas de ficheros dañados. El rendimiento de los sistemas de ficheros es importante y hay varias formas de mejorarlo, entre ellas la utilización de cachés, la lectura adelantada y la colocación cuidadosa de los bloques de un fichero cercanos entre sí en el disco. Los sistemas de ficheros con estructura de registro también mejoran el rendimiento porque realizan las escrituras en unidades grandes.

Como ejemplos de sistemas de ficheros podemos citar ISO 9660, CP/M, MS-DOS, Windows 98 y UNIX. Hay muchas diferencias entre ellos, que incluyen la forma de llevar el control de qué bloques corresponden a qué ficheros, la estructura de directorios y la administración del espacio libre en disco.

5

ENTRADA/SALIDA

Una de las principales funciones de un sistema operativo es la de controlar todos los dispositivos de E/S (Entrada/Salida). El sistema operativo debe enviar comandos a los dispositivos, atender las interrupciones y gestionar los errores. También debe proporcionar una interfaz entre los dispositivos y el resto del sistema que sea sencilla y fácil de utilizar. Hasta donde sea posible, la interfaz debe ser la misma para todos los dispositivos (independencia del dispositivo). El código de E/S representa una fracción significativamente grande del sistema operativo completo. La forma en la cual el sistema operativo gestiona la E/S es el tema de este capítulo.

Este capítulo está organizado de la siguiente manera. En primer lugar vamos a ver algunos de los principios del hardware de E/S, para después fijarnos en el software de E/S en general. El software de E/S puede estructurarse en capas, cada una de las cuales tiene encomendada una tarea bien definida. Vamos a fijarnos en esas capas para ver qué es lo que hacen y cómo encajan unas con otras.

Siguiendo a esa introducción, vamos a pasar a ver varios dispositivos de E/S en detalle: discos, relojes, teclados y pantallas. Para cada dispositivo vamos a ver tanto su hardware como su software. Finalmente vamos a considerar la gestión de la energía.

5.1 PRINCIPIOS DEL HARDWARE DE E/S

Diferentes personas ven el hardware de E/S de diferentes maneras. Los ingenieros electrónicos lo ven en términos de chips, cables, fuentes de alimentación, motores y todos los demás componentes físicos que componen el hardware. Los programadores lo ven en términos de la interfaz que presenta al software – los comandos que el hardware acepta, las funciones que lleva a cabo y los informes de error que pueden ser devueltos. En este libro nos concierne lo que tenga que ver con la programación de los dispositivos de E/S, no su diseño, su construcción o su mantenimiento, por lo que nuestro interés se restringe a cómo se programa el hardware y no a cómo funciona por dentro. Sin embargo, a menudo la programación de muchos dispositivos de E/S está conectada íntimamente con su operación interna. En las siguientes tres secciones vamos a proporcionar una pequeña base general sobre el hardware de E/S en lo relativo a la programación. Puede verse como una revisión y una expansión del material introductorio de la sección 1.4.

5.1.1 Dispositivos de E/S

En términos generales, los dispositivos de E/S pueden clasificarse en dos categorías: **dispositivos de bloques** y **dispositivos de caracteres**. Un dispositivo de bloques es uno que almacena la información en bloques de tamaño fijo, cada uno con su propia dirección. El tamaño de los bloques varía desde 512 bytes a 32768 bytes. La propiedad esencial de un dispositivo de bloques es que es posible leer o escribir cada bloque independientemente de todos los demás. Los discos son los dispositivos de bloques más comunes.

Si la examinamos más de cerca, la frontera entre los dispositivos que son direccionables por bloques y aquéllos que no lo son, no está nítidamente definida. Todo el mundo está de acuerdo en que un disco es un dispositivo direccionable por bloques, debido a que, sin importar dónde se encuentre posicionado el brazo del disco, siempre es posible situarse sobre otro cilindro y esperar a que el bloque requerido rote hasta pasar por debajo de la cabeza de lectura/escritura. Consideremos ahora una unidad de cinta utilizada para hacer backups del disco. Las cintas contienen una secuencia de bloques. Si en un momento dado deseamos que la unidad de cinta lea el bloque N , siempre podemos rebobinar la cinta e ir leyendo hacia delante hasta llegar al bloque N . Esta operación es análoga a la de un disco haciendo un posicionamiento, salvo que requiere mucho más tiempo. Hay que tener en cuenta también que no siempre es posible reescribir un bloque en el medio de una cinta. Pero incluso aunque fuera posible utilizar las cintas como dispositivos de bloques de acceso directo, eso sería forzar bastante las cosas: las cintas normalmente no se usan de esa manera.

El otro tipo de dispositivos de E/S es el de los dispositivos de caracteres. Un dispositivo de caracteres proporciona o acepta un flujo de caracteres, sin tener en cuenta ninguna estructura de bloque. No es un dispositivo direccionable y no cuenta con ninguna operación de posicionamiento. Las impresoras, los interfaces de red, los ratones (para señalar en la pantalla), las ratas (para experimentar en el laboratorio de psicología) y la mayoría de los otros dispositivos que no son similares a los discos, pueden ser vistos como dispositivos de caracteres.

Este esquema de clasificación no es perfecto. Algunos dispositivos simplemente no encajan en la clasificación. Por ejemplo, los relojes (*timers*) no son dispositivos direccionables por bloques, ni tampoco generan o aceptan flujos de caracteres. Lo único que hacen es provocar interrupciones a intervalos de tiempo bien definidos. Las pantallas con RAM de vídeo mapeada en memoria tampoco encajan bien en el modelo descrito. Sin embargo, el modelo de los dispositivos de bloques y de caracteres es una base lo suficientemente general para conseguir que una buena parte del software de E/S del sistema operativo sea independiente del dispositivo. Por ejemplo, el sistema de ficheros trata sólo con dispositivos de bloques abstractos, dejando para el software de nivel inferior la parte dependiente del dispositivo.

Las velocidades de los dispositivos de E/S abarcan un rango enormemente amplio, lo que ejerce una considerable presión sobre el software para que consiga responder siempre correctamente a lo largo de varios órdenes de magnitud en las velocidades de transferencia de los datos. La Figura 5-1 muestra las velocidades de transferencia de algunos dispositivos usuales. La mayoría de estos dispositivos tienden a ser cada día más rápidos.

| Device | Data rate |
|----------------------------|---------------|
| Keyboard | 10 bytes/sec |
| Mouse | 100 bytes/sec |
| 56K modem | 7 KB/sec |
| Telephone channel | 8 KB/sec |
| Dual ISDN lines | 16 KB/sec |
| Laser printer | 100 KB/sec |
| Scanner | 400 KB/sec |
| Classic Ethernet | 1.25 MB/sec |
| USB (Universal Serial Bus) | 1.5 MB/sec |
| Digital camcorder | 4 MB/sec |
| IDE disk | 5 MB/sec |
| 40x CD-ROM | 6 MB/sec |
| Fast Ethernet | 12.5 MB/sec |
| ISA bus | 16.7 MB/sec |
| EIDE (ATA-2) disk | 16.7 MB/sec |
| FireWire (IEEE 1394) | 50 MB/sec |
| XGA Monitor | 60 MB/sec |
| SONET OC-12 network | 78 MB/sec |
| SCSI Ultra 2 disk | 80 MB/sec |
| Gigabit Ethernet | 125 MB/sec |
| Ultrium tape | 320 MB/sec |
| PCI bus | 528 MB/sec |
| Sun Gigaplane XB backplane | 20 GB/sec |

Figura 5-1. Velocidad de transferencia de datos de algunos dispositivos, redes y buses típicos.

5.1.2 Controladores de dispositivos

Las unidades de E/S constan normalmente de un componente mecánico y un componente electrónico. En muchos casos es posible separar las dos partes para tener un diseño más modular y general. El componente electrónico se denomina **controlador del dispositivo** o **adaptador**. En los ordenadores personales, este componente suele adoptar la forma de una tarjeta de circuito impreso (**tarjeta controladora**) que puede insertarse en una ranura de expansión. El componente mecánico es el dispositivo mismo. Esta organización se muestra en la Figura 1-5.

La tarjeta controladora está provista usualmente de un conector en el cual puede conectarse un cable que va al dispositivo. Muchas controladoras pueden manejar dos, cuatro o incluso ocho dispositivos idénticos. Si la interfaz entre la controladora y el dispositivo es un interfaz estándar, ya sea un estándar ANSI, IEEE o ISO oficial, o un estándar *de facto*, eso permite que cualquier fabricante de hardware pueda manufacturar controladores o dispositivos que se ajusten a esa interfaz. Por ejemplo, muchas compañías de hardware fabrican unidades de disco compatibles con la interfaz IDE o SCSI.

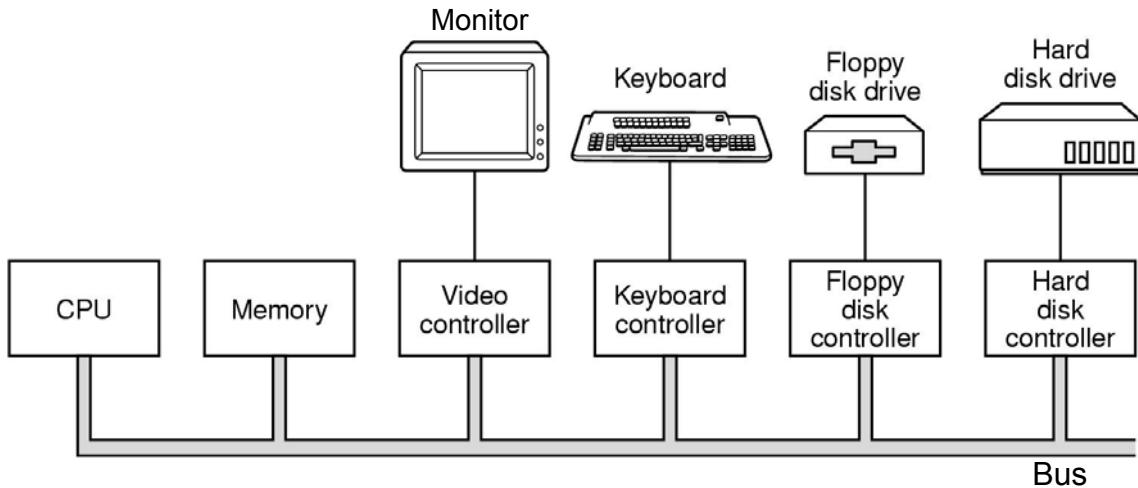


Figura 1-5. Algunos de los componentes de un ordenador personal sencillo.

La interfaz entre el controlador y el dispositivo es a menudo una interfaz de muy bajo nivel. Por ejemplo, un disco puede formatearse con 256 sectores de 512 bytes por pista. Sin embargo, lo que en realidad sale de la unidad es un flujo de bits en serie que comienza por un **preámbulo**, seguido de los 4096 bits de un sector y terminando con una suma de verificación (*checksum*), también llamada un **código de corrección de errores** (**ECC; Error-Correcting Code**). El preámbulo se escribe cuando se formatea el disco, y contiene el número de cilindro y de sector, el tamaño del sector y datos similares, así como información de sincronización.

La tarea del controlador consiste en convertir ese flujo de bits en serie en un bloque de bytes y realizar cualquier corrección de errores que sea necesaria. Normalmente primero se ensambla el bloque de bytes, bit a bit, en un búfer que está dentro del controlador. Una vez comprobado su checksum y declarado el bloque libre de errores, puede procederse a copiarlo en la memoria principal.

A un nivel igual de bajo, el controlador de un monitor también opera como un dispositivo de bits en serie: lee de la memoria bytes que contienen los caracteres a visualizar y genera las señales que sirven para modular el haz de electrones del CRT para producir la escritura en la pantalla. El controlador genera también las señales que hacen que el haz del CRT efectúe un retrazado horizontal al terminar cada barrido de una línea, así como las señales que realizan el retrazado vertical una vez que se ha barrido toda la pantalla. Si no fuera por el controlador del CRT, el programador del sistema operativo tendría que programar de forma expícita el barrido analógico del tubo de imagen del monitor. Con el controlador, el sistema operativo inicializa el controlador con unos pocos parámetros, tales como el número de caracteres o píxeles por línea y el número de líneas de la pantalla, y deja que el controlador sea realmente quien se encargue de dirigir el haz del CRT.

5.1.3 E/S mapeada en memoria

Cada controlador tiene unos cuantos registros que le sirven para comunicarse con la CPU. Escribiendo en estos registros, el sistema operativo puede ordenar al dispositivo que suministre datos, acepte datos, se encienda o apague a sí mismo, o realice alguna otra acción. Leyendo de estos registros, el sistema operativo puede averiguar en qué estado se encuentra el dispositivo, si está preparado o no para aceptar un nuevo comando, etc.

Además de los registros de control, muchos dispositivos tienen un búfer de datos que el sistema operativo puede leer y escribir. Por ejemplo, en muchos ordenadores la manera usual de visualizar píxeles en la pantalla es mediante una RAM de vídeo (que es básicamente un búfer de datos) en la que los programas o el sistema operativo pueden escribir.

Dicho lo anterior surge la pregunta de cómo se comunica la CPU con los registros de control y los búferes de datos de los dispositivos. Existen dos alternativas. Con el primer enfoque, a cada registro de control se le asigna un número de **puerto de E/S**, que es un número entero de 8 o 16 bits. Utilizando una instrucción especial de E/S como

IN REG,PUERTO

la CPU puede leer el registro de control **PUERTO** y almacenar el resultado en un registro **REG** de la CPU. Similarmente, con

OUT PUERTO,REG

La CPU puede escribir el contenido de **REG** en un registro de control. La mayoría de los primeros ordenadores – incluyendo casi todos los mainframes, como el IBM 360 y todos sus sucesores – funcionaban de esta manera.

En este esquema, los espacios de direcciones para la memoria y para E/S son distintos, como se muestra en la Figura 5-2(a). Las instrucciones

IN R0,4

y

MOV R0,4

son completamente distintas en este diseño. La primera lee el contenido del puerto de E/S 4 y lo coloca en **R0**, mientras que la segunda lee el contenido de la palabra de memoria 4 y lo coloca en **R0**. Los 4s en estos dos ejemplos se refieren a espacios de direcciones distintos que no tienen relación alguna entre sí.

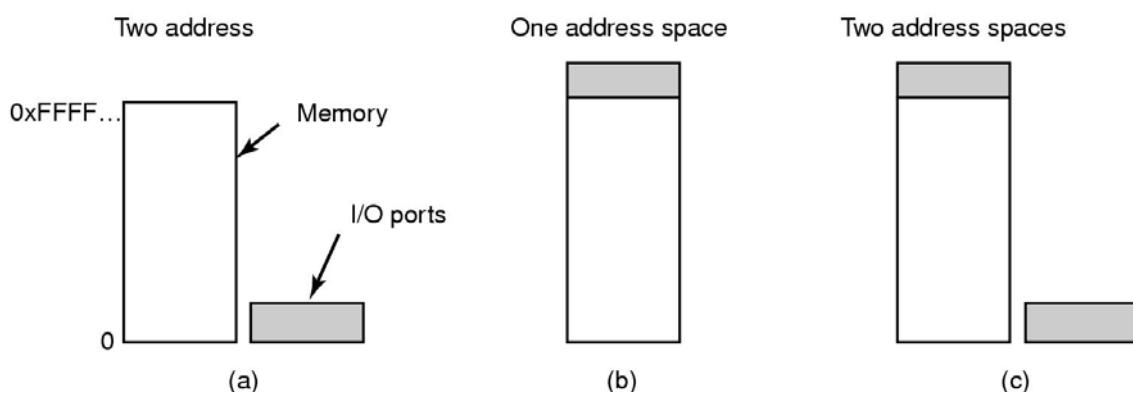


Figura 5-2. (a) Espacios de E/S y de memoria separados.
(b) E/S con mapeada en memoria. (c) Híbrido.

El segundo enfoque, introducido con el miniordenador PDP-11, consiste en mapear todos los registros de control dentro del espacio de memoria, como se muestra en la Figura 5-2(b). A cada registro de control se le asigna una dirección de memoria única a la cual no se asigna memoria. Este sistema se denomina **E/S mapeada en memoria**. Usualmente, las direcciones asignadas a los registros de control están en la parte más alta del espacio de direcciones. En la Figura 5-2(c) se muestra un esquema híbrido, con búferes de datos de E/S mapeados en memoria y puertos de E/S separados para los registros de control. El Pentium utiliza esta arquitectura en los ordenadores compatibles con el IBM PC, reservando las direcciones entre 640K y 1M para los búferes de datos de los dispositivos, además de los puertos de E/S del 0 al 64K.

¿Cómo funcionan estos dos esquemas? En todos los casos, cuando la CPU quiere leer una palabra, sea de la memoria o de un puerto de E/S, coloca la dirección que necesita en las líneas de dirección del bus y luego activa una señal READ sobre una línea de control del bus. Se usa una segunda línea de señal para indicar si la dirección se refiere al espacio de E/S o al de memoria. Si se indica el espacio de memoria, es la memoria quien responde a la solicitud; si se indica el espacio de E/S, es el dispositivo de E/S quien responde a la solicitud. Si sólo hay un espacio de memoria [como en la Figura 5-2(b)], cada módulo de memoria y cada dispositivo de E/S compara las líneas de dirección con el rango de direcciones a las que atiende. Si la dirección está dentro de ese rango, responde a la solicitud. Puesto que nunca se asigna ninguna dirección tanto a la memoria como a un dispositivo de E/S, no hay ambigüedades ni conflictos.

Los dos esquemas para direccionar los controladores tienen diferentes ventajas y desventajas. Comencemos con las ventajas de la E/S mapeada en memoria. Primera ventaja, si se necesitan instrucciones de E/S especiales para leer y escribir en los registros de control del dispositivo, el acceso a ellos requerirá el uso de código en ensamblador, porque no hay manera de ejecutar una instrucción IN o OUT en C o C++. El tener que llamar a procedimientos que realicen estas operaciones (escritos en código ensamblador) representa una sobrecarga adicional para controlar la E/S. En contraste, con la E/S mapeada en memoria, los registros de control del dispositivo no son más que variables en la memoria y pueden direccionarse en C de la misma manera que cualquier otra variable. Por lo tanto, con E/S mapeada en memoria, es posible escribir completamente en C el driver del dispositivo. Sin E/S mapeada en memoria, siempre será necesario escribir algunas partes del driver en código ensamblador.

Segunda ventaja, con E/S mapeada en memoria no se requiere ningún mecanismo de protección especial para evitar que los procesos de usuario realicen directamente la E/S. Lo único que necesita hacer el sistema operativo es cuidar de que la porción del espacio de direcciones que contiene los registros de control nunca se incluya en el espacio de direcciones virtual de ningún usuario. Mejor aún, si cada dispositivo tiene sus registros de control en una página distinta del espacio de direcciones, el sistema operativo podrá dar control a un usuario sobre algunos dispositivos específicos y no sobre otros con sólo incluir las páginas deseadas en su tabla de páginas. Tal esquema permite colocar diferentes drivers de dispositivo en diferentes espacios de direccionamiento, con lo cual no sólo se reduce el tamaño del núcleo sino que también se evita que un controlador interfiera con otros.

Tercera ventaja, con E/S mapeada en memoria cualquier instrucción que pueda hacer referencia a la memoria puede también referenciar registros de control. Por ejemplo, si en el repertorio de instrucciones hay una instrucción, TEST, que comprueba si una palabra de memoria es 0, esa misma instrucción también podrá usarse para determinar si un registro de control es 0, lo que podría indicar que el dispositivo está desocupado y puede aceptar un nuevo comando. El código en lenguaje ensamblador podría tener el aspecto que se muestra a continuación:

```

BUCLE: TEST PUERTO_4 // comprueba si el puerto 4 es 0
        BEQ LISTO    // si es 0, saltar a LISTO:
        BRANCH BUCLE // de lo contrario, continuar testeando
LISTO:

```

Si no utilizásemos E/S mapeada en memoria, primero deberíamos leer el registro de control dentro de la CPU mediante una instrucción IN, para luego realizar la comprobación, lo que significa dos instrucciones en lugar de una. En el caso del bucle anterior, la adición de una instrucción más haría un poco más lenta la detección de un dispositivo desocupado.

En el diseño de los ordenadores, prácticamente todo conduce a situaciones de compromiso, y ese es el caso también aquí. La E/S mapeada en memoria también tiene sus desventajas. Primera desventaja, la mayoría de los ordenadores actuales tienen algún tipo de caché de memoria. Cargar en la caché un registro de control de dispositivo sería desastroso. Consideremos la ejecución del bucle en código ensamblador visto anteriormente utilizando una caché. La primera referencia a PUERTO_4 haría que su valor se cargase en la caché. Todas las referencias posteriores a PUERTO_4 simplemente tomarían ese valor de la caché y no se preocuparían por consultar el dispositivo. De esa manera, cuando por fin el dispositivo esté listo, el software no tendrá forma de saberlo. El bucle no terminará nunca.

Para evitar esta situación con E/S mapeada en memoria, el hardware debe contar con la capacidad de deshabilitar de manera selectiva el uso de la caché, por ejemplo, para una página específica. Esto aumenta la complejidad tanto del hardware como del sistema operativo, que tiene que administrar el uso selectivo de la caché.

Segunda desventaja, si sólo hay un espacio de direcciones, todos los módulos de memoria y todos los dispositivos de E/S tienen que examinar todas las referencias a la memoria para determinar a cuáles deben responder. Si el ordenador tiene un único bus, como en la Figura 5-3(a), es sencillo obligar a todo el mundo a examinar cada dirección.

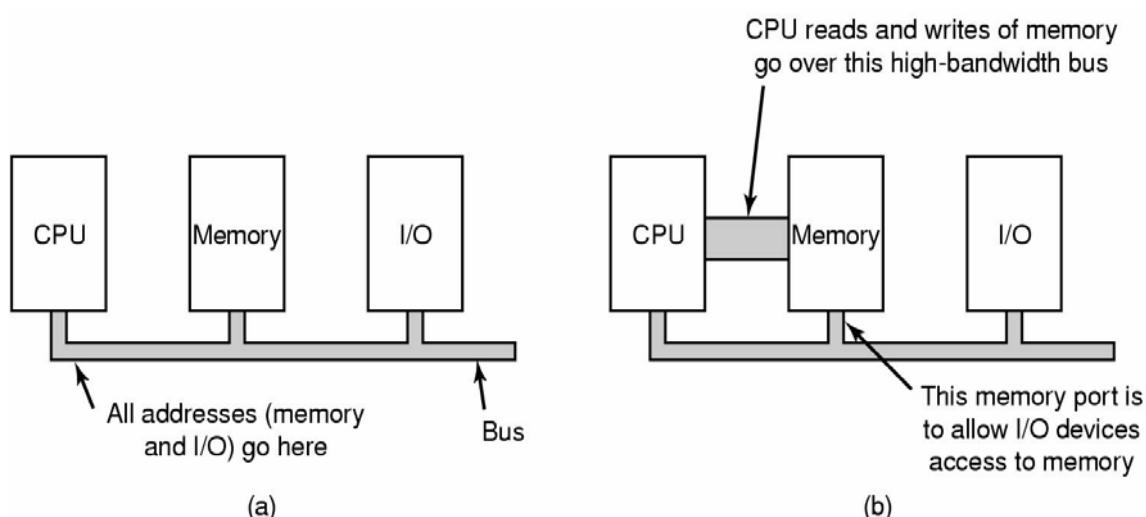


Figura 5-3. (a) Arquitectura de un solo bus. (b) Arquitectura de memoria de bus dual.

Sin embargo, en los ordenadores personales modernos la tendencia es tener un bus de alta velocidad dedicado a la memoria, como se muestra en la Figura 5-3(b), recurso con el que, por cierto, también cuentan los mainframes. Este bus está diseñado para optimizar el rendimiento de la memoria, sin verse afectado por la lentitud de los dispositivos de E/S. Los sistemas Pentium cuentan incluso con tres buses externos (memoria, PCI e ISA), como se muestra en la Figura 1-11.

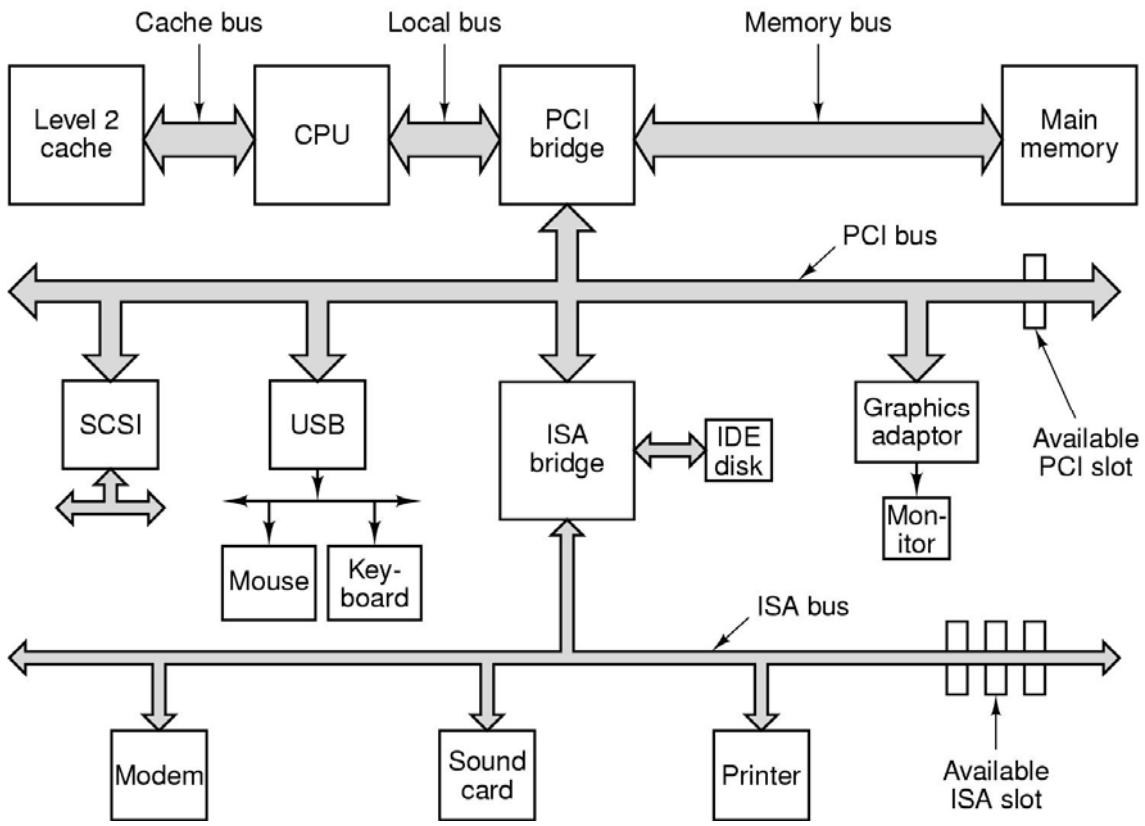


Figura 1-11. Estructura de un sistema Pentium grande.

El problema de tener un bus de memoria aparte en las máquinas con E/S mapeada en memoria es que los dispositivos de E/S no pueden ver las direcciones de memoria a su paso por el bus de memoria, por lo que no tienen la posibilidad de responder. Una vez más, es preciso tomar medidas especiales para lograr que la E/S mapeada en memoria funcione en un sistema con múltiples buses. Una posibilidad es enviar primero todas las referencias a la memoria. Si la memoria no responde a una referencia, la CPU prueba con los otros buses. Este diseño puede funcionar pero requiere un aumento de la complejidad del hardware.

Una segunda posibilidad de diseño es colocar un dispositivo fisiognomónico (*snooping*) en el bus de memoria para que pase todas las direcciones presentadas a dispositivos de E/S potencialmente interesados. El problema aquí es que los dispositivos de E/S seguramente no podrán procesar las peticiones con la misma rapidez que puede hacerlo la memoria.

Una tercera posibilidad, que es la que se usa en la configuración Pentium de la Figura 1-11, es filtrar las direcciones en el chip puente del PCI. Este chip contiene registros de rango que se cargan en el momento del arranque de la máquina. Por ejemplo, el rango de 640K a 1M podría marcarse como *no de memoria*. Las direcciones que caen en alguno de los intervalos marcados como *no de memoria* se envían por el bus PCI, en vez de enviarse a la memoria. La desventaja de este esquema es que durante el arranque de la máquina es preciso determinar qué direcciones de memoria no son realmente tales. Así pues, cada esquema tiene argumentos a favor y en contra, por lo que es inevitable hacer compromisos y concesiones.

5.1.4 Acceso directo a memoria

Independientemente de que tenga o no E/S mapeada en memoria, la CPU necesita dirigir los controladores de dispositivo para intercambiar datos con ellos. La CPU puede solicitar datos del controlador de E/S byte a byte, pero haciendo así estaría desperdiçándose mucho tiempo de CPU. Por ese motivo normalmente se utiliza un esquema diferente, denominado **acceso directo a memoria (DMA; Direct Memory Access)**. El sistema operativo sólo puede utilizar DMA si el hardware dispone de un controlador de DMA, por lo que la mayoría de los sistemas cuentan con él. A veces ese controlador está integrado en los controladores de disco o en otros controladores, pero tal diseño significa tener un controlador de DMA por cada dispositivo. Más comúnmente, se tiene un único controlador de DMA (por ejemplo en la placa madre) para regular las transferencias con múltiples dispositivos, a menudo de forma concurrente.

Sea cual sea su ubicación física, el controlador de DMA tiene acceso al bus del sistema independientemente de la CPU, como se muestra en la Figura 5-4. El controlador contiene varios registros en los que la CPU puede leer y escribir. Éstos incluyen un registro de dirección de memoria, un registro contador de bytes y uno o más registros de control. Los registros de control especifican el puerto de E/S que se utilizará, la dirección de la transferencia (leyendo del dispositivo de E/S o escribiendo en el dispositivo de E/S), la unidad de transferencia (un byte a la vez o una palabra a la vez) y el número de bytes que se transferirán en cada ráfaga.

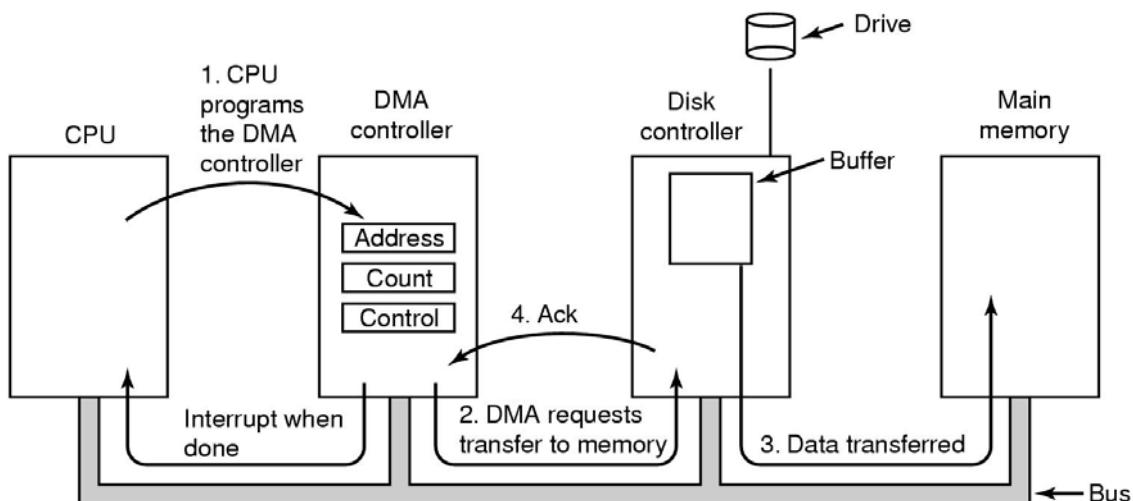


Figura 5-4. Funcionamiento de una transferencia con DMA.

Para explicar cómo funciona el DMA, veamos primero cómo se realizan las lecturas del disco cuando no se utiliza DMA. Primero, el controlador lee el bloque (uno o más sectores) de la unidad, bit a bit en serie, hasta que esté todo el bloque en el búfer interno del controlador. Luego, calcula el checksum para comprobar que no se produjeron errores al leer, y provoca una interrupción. Cuando el sistema operativo comienza a ejecutarse, puede leer ya el bloque de disco del búfer de la controladora byte a byte (o palabra a palabra), ejecutando un bucle, en el que en cada iteración lee un byte (o una palabra) de un registro de control del controlador y lo almacena en la memoria principal.

Cuando se utiliza DMA, el procedimiento es diferente. Primero la CPU programa el controlador de DMA, escribiendo los valores apropiados en sus registros para que sepa qué debe transferir y a dónde debe transferirse (paso 1 en la Figura 5-4). La CPU envía también un

comando al controlador del disco para indicarle que lea los datos del disco en su búfer interno y compruebe el checksum. Cuando haya datos válidos en el búfer del controlador del disco, el DMA puede comenzar.

El controlador de DMA inicia la transferencia enviando por el bus una petición de lectura al controlador de disco (paso 2). Esta petición de lectura es similar a cualquier otra petición de lectura, y el controlador del disco no sabe ni le importa si proviene de la CPU o de una controladora de DMA. Típicamente, la dirección de memoria en la que va a escribirse está ya en las líneas de dirección del bus, así que cuando el controlador del disco toma la siguiente palabra de su búfer interno, sabe donde escribirla. La escritura en memoria es otro ciclo de bus estándar (paso 3). Cuando termina la escritura, el controlador del disco envía una señal de acuse al controlador de DMA, también por el bus (paso 4). Después, el controlador de DMA incrementa la dirección de memoria a utilizar y decrementa el contador de bytes. Si el contador de bytes sigue siendo todavía mayor que 0, se repiten los pasos 2 a 4 hasta que el contador llega a valer 0. En ese momento, el controlador de DMA interrumpe a la CPU para avisarle de que se ha completado la transferencia. Cuando el sistema operativo tome el control, no tendrá que copiar el bloque del disco en la memoria; el bloque ya está ahí.

Los controladores de DMA varían mucho en cuanto a su sofisticación. Los más sencillos realizan una transferencia de cada vez, como acabamos de describir. Otros más complejos pueden programarse para manejar varias transferencias simultáneas. Tales controladores cuentan internamente con múltiples conjuntos de registros, un conjunto por cada canal. La CPU comienza cargando cada conjunto de registros con los parámetros pertinentes para su transferencia. Cada transferencia debe utilizar un controlador de dispositivo diferente. Después de transferir cada palabra (pasos 2 a 4 de la Figura 5-4), el controlador de DMA decide a qué dispositivo atenderá a continuación. Podría configurarse para aplicar un algoritmo de tipo round-robin (asignando turnos de forma circular) o un esquema de prioridades para dar preferencia a algunos dispositivos sobre otros. Es posible tener pendientes al mismo tiempo varias solicitudes a diferentes controladores de dispositivo, supuesto que exista una forma inequívoca de distinguir los diferentes acuses provenientes de los controladores. Por esa razón es frecuente la utilización de una línea de acuse del bus diferente para cada canal de DMA.

Muchos buses pueden operar de dos modos: modo palabra a palabra y modo bloque. Algunos controladores de DMA pueden operar también en cualquiera de los dos modos. En el primer modo el funcionamiento es el que acabamos de describir: el controlador de DMA solicita la transferencia de una palabra y la obtiene. Si la CPU también quiere el bus, tendrá que esperar. Este mecanismo se denomina **robo de ciclo** (*cycle stealing*) porque el controlador del dispositivo de vez en cuando quita furtivamente un ciclo de bus a la CPU, retrasándola ligeramente. En el modo bloque, el controlador de DMA solicita al controlador del dispositivo que adquiera el bus, realice una serie de transferencias y libere el bus. Esta forma de funcionamiento se denomina **modo ráfaga** (*burst mode*). Es más eficiente que el robo de ciclo porque la adquisición del bus requiere cierto tiempo y en el modo ráfaga se transfieren varias palabras al precio de una única adquisición del bus. La desventaja del modo ráfaga es que puede bloquear a la CPU y a otros dispositivos durante un periodo de tiempo considerable si se está transfiriendo una ráfaga muy larga.

En el modelo que hemos estado analizando, conocido a veces como **modo fly-by**, el controlador de DMA pide al controlador de dispositivo que transfiera el dato directamente a la memoria principal. Un modo alternativo que utilizan algunos controladores de DMA consiste en pedir al controlador del dispositivo que envíe la palabra al controlador de DMA, el cual realiza entonces una segunda petición de bus para escribir la palabra en el lugar de destino. Este esquema requiere un ciclo de bus extra por cada palabra transferida, pero es más flexible ya que permite realizar también transferencias de dispositivo a dispositivo e, incluso, de memoria a memoria (realizando primero una solicitud de lectura de la memoria y luego una solicitud de escritura en memoria sobre una dirección diferente).

La mayoría de los controladores de DMA utilizan direcciones físicas de memoria para sus transferencias. Utilizar direcciones físicas requiere que el sistema operativo convierta la dirección virtual del búfer de memoria deseado en una dirección física, y escriba esa dirección física en el registro de dirección del controlador de DMA. Un esquema alternativo que se utiliza en algunos (más bien pocos) controladores de DMA consiste en escribir direcciones virtuales en el controlador. En ese caso el controlador de DMA debe utilizar la MMU para efectuar la traducción de la dirección virtual a física. Sólo tiene sentido poner las direcciones virtuales en el bus si la MMU forma parte de la memoria (lo cual es posible pero poco común), en vez de estar integrada con la CPU.

Mencionamos anteriormente que el disco primero lee los datos en su búfer interno antes de que pueda comenzar el DMA. El lector seguramente se estará preguntando por qué motivo el controlador del disco no almacena simplemente los bytes en la memoria principal tan pronto como los recibe del disco. En otras palabras, ¿por qué necesita el controlador tener un búfer interno? Hay dos razones. La primera es que al utilizar un búfer interno, el controlador puede comprobar el *checksum* (suma de verificación) antes de iniciar una transferencia. Si el checksum es incorrecto, se comunicará el error y no se realizará la transferencia.

La segunda razón es que una vez que ha comenzado una transferencia de disco, los bits van llegando inexorablemente del disco a un ritmo constante que no tiene en cuenta si el controlador está listo para recibirlas o no. Si se encomienda al controlador la escritura de los datos directamente en la memoria, dicho controlador necesitará poder utilizar el bus del sistema una vez por cada palabra transferida. Si alguna de esas veces el bus resulta estar ocupado porque otro dispositivo lo está utilizando (por ejemplo, en modo ráfaga), el controlador tendrá que esperar, y si la siguiente palabra de disco llega antes de que la primera se escriba en la memoria, el controlador tendrá que ponerla en algún lado. Si el bus está muy ocupado, el controlador tendrá que guardar provisionalmente un buen número de palabras y realizar muchas tareas administrativas adicionales. En cambio, si el bloque se coloca en un búfer interno, el bus sólo se necesita cuando comienza el DMA, por lo que el diseño del controlador es mucho más sencillo al no tener que transferir cada palabra en un tiempo crítico. (Algunas controladoras antiguas sí que realizaban la transferencia directamente a la memoria con tan solo un pequeño búfer interno, pero cuando el bus estaba muy ocupado podía llegar a suceder que la transferencia se viera suspendida por un error de desbordamiento del búfer.)

No todos los ordenadores utilizan DMA. El argumento en contra es que la CPU principal a menudo es mucho más rápida que el controlador de DMA y puede realizar el trabajo mucho más rápidamente (cuando el factor limitante no es la rapidez del dispositivo de E/S). Si la CPU (rápida) no tiene otra cosa que hacer, no tiene objeto hacer que espere a que termine el controlador de DMA (lento). Además, prescindir del controlador de DMA y dejar que la CPU realice todo el trabajo por software ahorra dinero, lo cual es importante en los ordenadores de gama baja.

5.1.5 Repaso de las interrupciones

En la sección 1.4.3 dimos una breve introducción sobre el tema de las interrupciones, pero es necesario conocer más cosas sobre ellas. En un ordenador personal típico, la estructura de las interrupciones es como se muestra en la Figura 5-5. A nivel de hardware, las interrupciones funcionan como sigue. Cuando un dispositivo de E/S termina el trabajo que se le encomendó, provoca una interrupción (suponiendo que el sistema operativo ha habilitado las interrupciones). Esto lo hace aplicando una señal a una línea del bus que se le ha asignado. El chip controlador de interrupciones situado en la placa madre detecta esa señal y decide lo que se va a hacer a continuación.

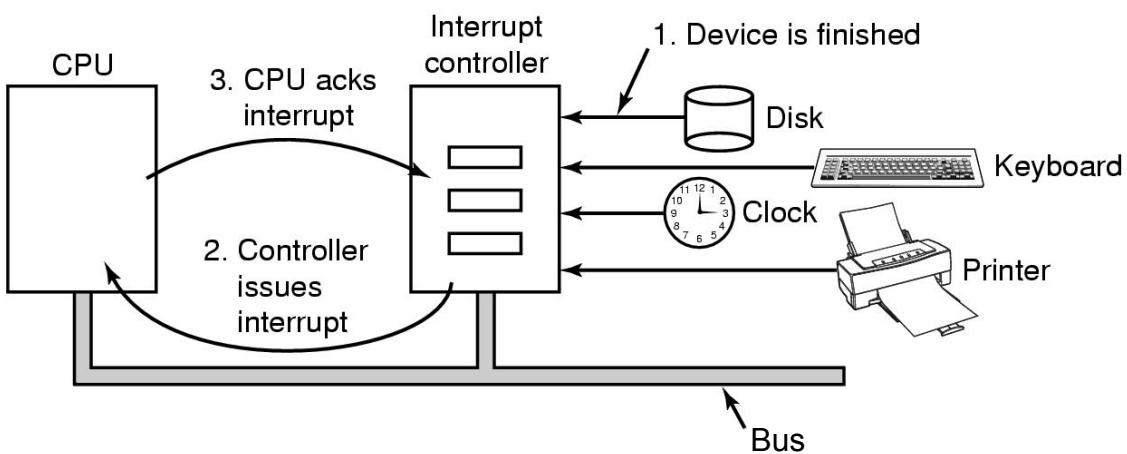


Figura 5-5. Forma en la que se produce una interrupción. En la realidad las conexiones entre los dispositivos y el controlador de interrupciones utilizan líneas del bus en vez de cables dedicados.

Si no hay otras interrupciones pendientes, el controlador de interrupciones procesa la interrupción inmediatamente. Si está atendiendo alguna otra interrupción en ese momento, o si otro dispositivo ha realizado una petición simultánea sobre una línea de petición de interrupción de mayor prioridad, el primer dispositivo será ignorado momentáneamente. En este caso, el dispositivo seguirá aplicando la señal de interrupción al bus hasta que reciba de la CPU el servicio deseado.

Para gestionar la interrupción, el controlador vuelca un número en las líneas de dirección del bus especificando qué dispositivo requiere atención y aplica una señal que interrumpe a la CPU.

La señal de interrupción provoca que la CPU deje lo que estaba haciendo y comience a hacer alguna otra cosa. El número que está en las líneas de dirección se usa como un índice de una tabla llamada la **tabla de vectores de interrupción** para extraer un nuevo contador de programa, el cual apunta al comienzo del correspondiente procedimiento de servicio de la interrupción. Normalmente tanto los traps, como las interrupciones utilizan el mismo mecanismo a partir de este punto, y a menudo comparten la misma tabla de vectores de interrupción. La ubicación de la tabla de vectores de interrupción puede estar cableada en la máquina o puede estar en cualquier lugar de la memoria, en cuyo caso habrá un registro en la CPU (cargado por el sistema operativo) que apuntará a su comienzo.

Poco después de comenzar a ejecutarse, el procedimiento de servicio de la interrupción efectúa un ciclo de reconocimiento de la interrupción, escribiendo cierto valor en uno de los puertos de E/S del controlador de interrupciones. Este reconocimiento (o acuse de la recepción) de la interrupción le dice al controlador que, habiéndose aceptado ya su petición de interrupción, queda libre para poder solicitar una nueva interrupción. La demora por parte de la CPU del reconocimiento de la interrupción hasta que esté lista para gestionar la siguiente interrupción permite evitar muchas condiciones de carrera involucrando a múltiples interrupciones casi simultáneas. Como comentario, algunos ordenadores (antiguos) no cuentan con un controlador de interrupciones centralizado, por lo que cada controlador de dispositivo debe solicitar directamente sus propias interrupciones.

El hardware siempre guarda cierta información antes de comenzar la ejecución del procedimiento de servicio de la interrupción. La información que se guarda y el lugar donde se guarda varían mucho de unas CPUs a otras. Como mínimo debe guardarse el contador de programa para poder retomar la ejecución del proceso interrumpido. En el otro extremo, podrían guardarse todos los registros visibles y un gran número de registros internos.

El problema es dónde guardar esa información. Una opción sería ponerla en registros internos que el sistema operativo pueda leer cuando lo necesite. Este enfoque tiene la desventaja de que no puede enviarse el acuse de la aceptación de la interrupción al controlador de interrupciones hasta no haber leído toda la información potencialmente relevante almacenada, ya que una segunda interrupción podría sobrescribir los registros internos donde se guarda el estado previo a la interrupción. Esta estrategia provoca largos tiempos muertos, durante los que las interrupciones están inhibidas, con el riesgo de pérdida de nuevas interrupciones y datos.

Consecuentemente la mayoría de las CPUs guardan la información en la pila. Sin embargo, este enfoque también tiene sus problemas. Para empezar: ¿la pila de quién? Si se utiliza la pila actual, posiblemente sería la pila de un proceso de usuario. En ese caso cabe la posibilidad de que el puntero de pila tenga un valor incorrecto, lo cual provocaría un error fatal cuando el hardware intente escribir palabras en ella. Pero incluso siendo correcto el puntero de pila, podría apuntar al final de una página, de manera que tras varias escrituras en la memoria podría atravesarse la frontera de la página, generándose una falta de página. El tener una falta de página mientras el núcleo está procesando una interrupción crea un problema todavía mayor: ¿dónde se guarda el estado actual mientras se resuelve la falta de página?

Si se utiliza la pila del supervisor (la pila del núcleo), aumenta mucho la probabilidad de que el puntero de pila sea válido y apunte a una página que no pueda causar problemas. Sin embargo, el cambio a modo supervisor podría requerir un cambio de contexto de la MMU y probablemente anularía la validez de la mayor parte o la totalidad de la caché y la TLB. La recarga de toda esa información, estática o dinámicamente incrementa el tiempo necesario para procesar una interrupción y por lo tanto desperdicia tiempo de CPU.

Otro problema se debe al hecho de que casi todas las CPUs modernas están altamente segmentadas (tienen un *pipeline* con muchas etapas) y muchas veces son superescalares (paralelas internamente). En los sistemas más antiguos, una vez que terminaba de ejecutarse una instrucción, el microprograma o el hardware comprobaba si había una interrupción pendiente. En tal caso, el contador de programa y la PSW se guardaban en la pila y comenzaba la secuencia de interrupción. Después de ejecutar el manejador de la interrupción, se realizaba el proceso inverso, sacando de la pila la PSW y el contador de programa anteriores y continuando con el proceso interrumpido.

Este modelo supone implícitamente que si se produce una interrupción inmediatamente después de alguna instrucción, eso significa que todas las instrucciones anteriores junto con la última se han ejecutado por completo, y que las instrucciones siguientes no han comenzado a

ejecutarse. En las máquinas más antiguas esa suposición era siempre válida. En las modernas podría no serlo.

Para empezar, consideremos el modelo de pipeline de la Figura 1-6(a). ¿Qué sucede si se presenta una interrupción cuando el pipeline está lleno (que es lo normal)? En ese caso hay varias instrucciones que están en diversas etapas intermedias de su ejecución. Cuando se presenta la interrupción, es muy probable que el valor del contador de programa no refleje la frontera correcta entre las instrucciones ejecutadas y las no ejecutadas. Lo más probable es que refleje la dirección de la siguiente instrucción que se extraerá de la memoria (y se introducirá en el pipeline) en vez de la dirección de la siguiente instrucción que la unidad de ejecución va a procesar.

Como consecuencia, aun en el caso (dudoso) de que exista una frontera bien definida entre las instrucciones que se han ejecutado y las que no lo han hecho, el hardware puede ser incapaz de saber dónde está esa frontera. Por tanto, cuando el sistema operativo deba retornar de una interrupción, no puede simplemente comenzar a llenar el pipeline a partir de la dirección contenida en el contador de programa, sino que debe determinar de alguna manera cuál fue la última instrucción que se ejecutó, lo que supone a menudo realizar una tarea de software compleja como es el análisis del estado de la máquina.

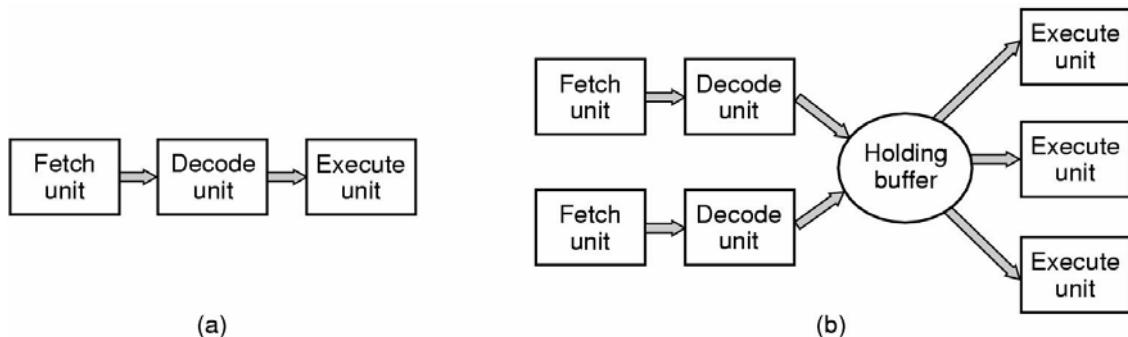


Figura 1-6. (a) Pipeline de tres etapas. (b) CPU superescalar.

Aunque esta situación es mala, las interrupciones en una máquina superescalar, como la de la Figura 1-6(b), son incluso peores. Dado que las instrucciones podrían ejecutarse desordenadas, la probabilidad de que no exista una frontera bien definida entre las instrucciones ejecutadas y las no ejecutadas es mucho mayor. Así podría suceder perfectamente que ya se hubieran ejecutado las instrucciones 1, 2, 3, 5 y 8, pero no todavía las instrucciones 4, 6, 7, 9, 10 y siguientes. Además, puede ser que el contador de programa esté apuntado actualmente a cualquiera de las instrucciones 9, 10 u 11.

Una interrupción que deja a la máquina en un estado bien definido se denomina una **interrupción precisa** (Walter y Cragon, 1995). Una interrupción de ese tipo tiene cuatro propiedades:

1. El contador de programa (PC) se guarda en un lugar conocido.
2. Todas las instrucciones anteriores a aquella a la que apunta el PC se han ejecutado ya por completo.
3. No ha sido ejecutada ninguna instrucción posterior a aquella a la que apunta el PC.
4. Se conoce el estado de ejecución de la instrucción a la que apunta el PC.

En cuanto a la propiedad 3 hay que aclarar que no se prohíbe el inicio de la ejecución de instrucciones posteriores a aquella a la que apunta el PC, pero cualquier cambio que produzcan tales instrucciones en los registros o en la memoria deberá deshacerse antes de que la

interrupción ceda el control a su rutina de tratamiento de la interrupción. Se permite que la instrucción a la que apunta el PC haya terminado ya su ejecución, pero también se permite que todavía no lo haya hecho. Sin embargo, debe quedar claro de qué caso se trata. Si la interrupción es de E/S, es frecuente que todavía no se haya iniciado la instrucción; pero si la interrupción es en realidad una excepción o una falta de página, el PC por lo general apunta a la instrucción que provocó el fallo, para poder continuar con su ejecución después.

Una interrupción que no cumple estos requisitos se denomina **una interrupción imprecisa** y complica extremadamente la vida del escritor del sistema operativo, quien ahora no sólo tiene que determinar lo que ha sucedido, sino también lo que no ha sucedido todavía. Las máquinas con interrupciones imprecisas suelen vomitar a la pila una gran cantidad de información sobre el estado interno para que el sistema operativo tenga la posibilidad de dilucidar lo que estaba sucediendo. El tener que guardar una gran cantidad de información en la memoria en cada interrupción hace que las interrupciones sean lentas y peor aún la recuperación tras ellas. Esto conduce a la irónica situación de que a veces las rapidísimas CPUs superescalares no son apropiadas para las aplicaciones en tiempo real debido a la lentitud de las interrupciones.

Algunos ordenadores se diseñan de modo que algunas interrupciones de E/S (y excepciones) sean precisas y otras no. Por ejemplo, si las interrupciones de E/S son precisas, el que las excepciones por errores de programación fatales sean imprecisas no representa ningún problema, ya que en ese caso el proceso que se estaba ejecutando se aborta directamente sin que haya ninguna necesidad de continuar con su ejecución. Algunas máquinas tienen un bit que puede activar el sistema operativo para forzar a que todas las interrupciones sean precisas. El inconveniente de activar ese bit es que fuerza a la CPU a llevar un registro minucioso de todo lo que está haciendo y a mantener copias sombra de los registros de manera que pueda generar una interrupción precisa en cualquier instante. Toda esta sobrecarga de trabajo afecta negativamente al rendimiento de la máquina.

Algunas máquinas superescalares, como el Pentium Pro y todas sus sucesoras, tienen interrupciones precisas para permitir que los programas antiguos escritos para el 386, 486 y Pentium I funcionen correctamente (la superescalaridad se introdujo en el Pentium Pro; el Pentium I sólo tenía dos pipelines). El precio que se paga por tener interrupciones precisas es una lógica de interrupciones extremadamente compleja dentro de la CPU para asegurar que cuando el controlador de interrupciones indique que desea provocar una interrupción, se permita terminar a todas las instrucciones que hayan llegado hasta cierto punto y no se permita a ninguna instrucción posterior tener ningún efecto perceptible sobre el estado de la máquina. Aquí el precio no se paga en tiempo sino en área de chip y en complejidad del diseño. Si no se requiriesen interrupciones precisas para garantizar la compatibilidad hacia atrás, este área del chip podría aprovecharse para hacer más grandes las cachés internas del chip, haciendo así más rápida a la CPU. Por otra parte, las interrupciones imprecisas hacen que el sistema operativo sea mucho más complicado y lento, por lo que es difícil determinar qué enfoque es realmente mejor.

5.2 PRINCIPIOS DEL SOFTWARE DE E/S

Vamos a dejar ahora a un lado el hardware de E/S pasando a echar un vistazo al software de E/S. Trataremos primero los objetivos del software de E/S y luego las distintas formas en las que puede llevarse a cabo la E/S desde el punto de vista del sistema operativo.

5.2.1 Objetivos del software de E/S

Un concepto clave en el diseño del software de E/S es lo que se conoce como la **independencia del dispositivo**, lo que significa que debe ser posible escribir programas capaces de acceder a cualquier dispositivo de E/S sin tener que especificar por adelantado de qué dispositivo se trata. Por ejemplo, un programa que tome su entrada de un fichero debe poder leerlo tanto de un disquete, como de un disco duro, como de un CD-ROM sin tener que modificar el programa para cada dispositivo diferente. Similarmente, debe ser posible que un comando del shell como

```
sort < entrada > salida
```

funcione con la entrada proveniente de un disquete, un disco IDE, un disco SCSI o el teclado, y enviando la salida a cualquier tipo de disco o a la pantalla. Corresponde al sistema operativo resolver los problemas causados por el hecho de que todos esos dispositivos son en realidad diferentes y requieren secuencias de comandos muy distintas para leer o escribir.

El objetivo de **denominación uniforme de ficheros y dispositivos** está estrechamente relacionado con la independencia del dispositivo. El nombre de un fichero o dispositivo debe ser simplemente una cadena de caracteres o un entero y no depender en absoluto del dispositivo. En UNIX, todos los discos pueden integrarse en la jerarquía del sistema de ficheros con total libertad, de manera que el usuario no necesita saber qué nombre corresponde a qué dispositivo. Por ejemplo, un disquete puede **montarse** en el directorio `/usr/ast/backup` de manera que copiando cualquier fichero en el directorio `/usr/ast/backup/lunes`, estamos realmente copiando dicho fichero en el disquete. De esta manera todos los ficheros y dispositivos se direccionan del mismo modo: mediante su nombre de camino (absoluto o relativo).

Otro aspecto importante del software de E/S es el **manejo de errores**. En general, los errores deben tratarse tan cerca del hardware como sea posible. Si el controlador descubre un error de lectura, él mismo debe tratar de corregirlo en un primer momento. Si no puede, será el driver del dispositivo quien deberá tratar de corregirlo, por ejemplo repitiendo el intento de lectura del bloque. Muchos errores son transitorios, tales como los errores de lectura provocados por una partícula de polvo en la cabeza de lectura, y desaparecen si se repite la operación. Sólo debe informarse del problema a las capas superiores cuando las capas inferiores no puedan solucionar el problema por sí mismas. En muchos casos, la recuperación de los errores puede realizarse de forma transparente en los niveles más bajos, sin necesidad de que los niveles superiores se enteren siquiera de que tuvo lugar el error.

También otro aspecto clave son las transferencias **síncronas** (bloqueantes) frente a las **asíncronas** (dirigidas por interrupciones). Casi toda la E/S física es asíncrona – la CPU pone en marcha la transferencia y se pone a hacer alguna otra cosa hasta que llega la interrupción. Los programas de usuario son mucho más fáciles de escribir si las operaciones de E/S son bloqueantes – después de una llamada al sistema `read`, el programa se suspende

automáticamente hasta que los datos estén disponibles en el búfer. Corresponde al sistema operativo hacer que las operaciones que realmente están controladas por interrupciones parezcan bloqueantes desde la perspectiva de los programas de usuario.

Otra cuestión que corresponde al software de E/S es el **almacenamiento intermedio de los datos** (*buffering*). A menudo los datos provenientes de un dispositivo no pueden almacenarse directamente en su destino final. Por ejemplo, cuando llega un paquete por la red, el sistema operativo no sabe a donde dirigirlo hasta que no lo guarda en algún sitio y lo examina. Además, algunos dispositivos están sujetos a severas restricciones de tiempo real (por ejemplo, los dispositivos de audio digital o las grabadoras de CDs), por lo que los datos deben colocarse primeramente en un búfer de salida para desacoplar la velocidad a la que se llena el búfer con la velocidad a la que se vacía, de manera que el dispositivo de destino nunca se encuentre con el búfer vacío (*buffer underruns*). El uso de búferes requiere numerosas operaciones de copiado y tiene a menudo un importante impacto sobre el rendimiento de la E/S.

El último concepto que mencionaremos aquí es el de dispositivos compatibles frente a dispositivos dedicados. Algunos dispositivos de E/S, como los discos, pueden ser utilizados simultáneamente por muchos usuarios. No hay ningún problema porque varios usuarios tengan ficheros abiertos en el mismo disco simultáneamente. Otros dispositivos, como las unidades de cinta, tienen que estar dedicados a un único usuario hasta que ese usuario termine. Más tarde podrá asignarse la unidad de cinta a otro usuario. Ciertamente no puede funcionar bien que dos o más usuarios escriban bloques entremezclados de forma aleatoria en la misma cinta. La introducción de dispositivos dedicados (no compartidos) introduce también diversos problemas, como los interbloqueos. Una vez más, el sistema operativo debe ser capaz de manejar tanto dispositivos compartidos como dedicados de forma que no se produzcan problemas.

5.2.2 E/S programada

Hay tres formas fundamentalmente distintas de hacer E/S. En esta sección vamos a echar un vistazo a la primera (E/S programada). En las dos secciones siguientes examinaremos las otras (E/S dirigida por interrupciones y E/S utilizando DMA). La forma más sencilla de E/S consiste en dejar que la CPU haga todo el trabajo. Este método se denomina **E/S programada**.

Resulta más sencillo ilustrar la E/S programada mediante un ejemplo. Consideremos un proceso de usuario que desea imprimir la cadena de ocho caracteres “ABCDEFGHI” por la impresora. Lo primero que hace es formar la cadena de caracteres en un búfer en el espacio de usuario, como se muestra en la Figura 5-6(a).

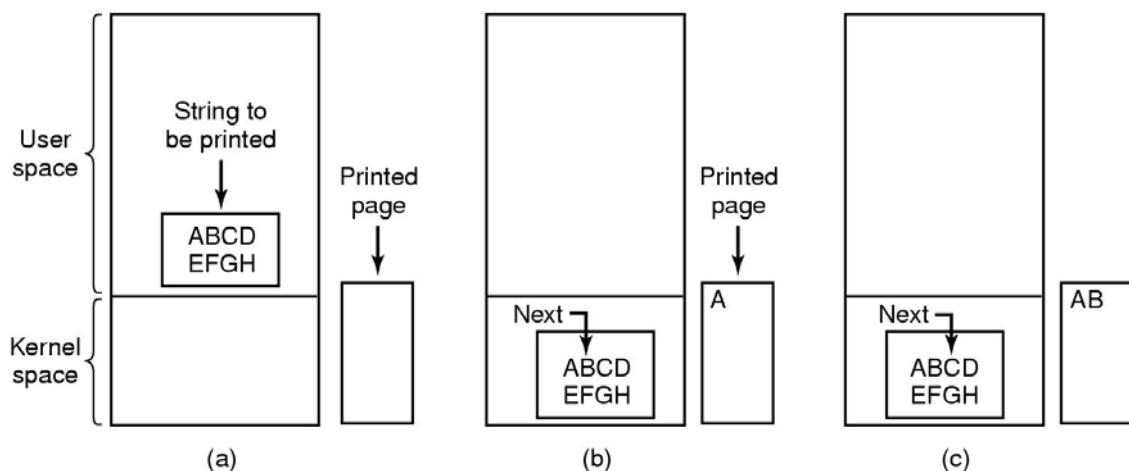


Figura 5-6. Pasos en la impresión de una cadena.

A continuación el proceso de usuario debe solicitar poder utilizar la impresora para escribir, haciendo una llamada al sistema para abrirla. Si actualmente la impresora está siendo utilizada por otro proceso, esta llamada no tendrá éxito y, dependiendo del sistema operativo y de los parámetros de la llamada, devolverá un código de error o se bloqueará hasta que esté disponible la impresora. Una vez que el proceso de usuario consiga la impresora, realizará una llamada al sistema diciéndole al sistema operativo que imprima la cadena.

Seguidamente lo más normal es que el sistema operativo copie el búfer que contiene la cadena en una tabla, llamémosla p , en el espacio del núcleo, donde puede acceder más fácilmente a la cadena (porque el núcleo podría tener que cambiar el mapa de memoria para poder acceder al espacio de usuario). Luego comprueba si la impresora está disponible actualmente. Si no lo está, esperará hasta que lo esté. Tan pronto como la impresora esté disponible, el sistema operativo copia el primer carácter en el registro de datos de la impresora, en este ejemplo utilizando E/S mapeada en memoria. Esta acción activa la impresora. Es posible que el carácter no aparezca todavía porque algunas impresoras esperan a que se complete una línea o toda una página antes de imprimir nada, manteniendo en el búfer los caracteres recibidos. Sin embargo en la Figura 5-6(b) vemos que se ha imprimido el primer carácter y que el sistema considera la “B” como el siguiente carácter a imprimir.

Tan pronto como el sistema operativo termina de copiar el primer carácter a la impresora, comprueba si está lista para aceptar el siguiente. Generalmente, la impresora tiene un segundo registro, que informa de su estado. El propio acto de escribir en el registro de datos provoca que el estado sea el de “no preparada”. Cuando el controlador de la impresora termina de procesar el carácter actual, indica su disponibilidad activando algún bit de su registro de estado o colocando algún valor especial en él.

En este momento el sistema operativo espera a que la impresora esté lista otra vez. Cuando eso sucede, imprime el siguiente carácter, como se muestra en la Figura 5-6(c). Este ciclo continúa hasta que se imprime toda la cadena. Finalmente se devuelve el control al proceso de usuario.

Las acciones realizadas por el sistema operativo se resumen en la Figura 5-7. Primero se copian los datos en el núcleo. Luego el sistema operativo entra en un bucle enviando los caracteres de uno en uno. El funcionamiento esencial de la E/S programada, que ilustra claramente la figura, es que después de enviar un carácter a la impresora, la CPU muestrea continuamente el estado del dispositivo para ver si está lista para aceptar otro. Este comportamiento se denomina a menudo *polling* o **espera activa** (*busy waiting*).

```
copy_from_user(buffer, p, count);           /* p es el bufer del nucleo */
for (i=0; i < count; i++) {
    while (*printer_status_reg != READY);   /* repetir para cada caracter */
    *printer_data_register = p[i];          /* repetir hasta que este lista */
}                                            /* envia el caracter */
return_to_user();
```

Figura 5-7. Escritura de una cadena en la impresora utilizando E/S programada.

La E/S programada es sencilla pero tiene la desventaja de mantener a la CPU ocupada todo el tiempo hasta que termina la E/S. Si el tiempo requerido para “imprimir” un carácter es muy corto (porque lo único que hace la impresora es copiar el nuevo carácter a un búfer interno), la espera activa resulta apropiada. En un sistema empotrado donde la CPU no tiene ninguna otra cosa que hacer, la espera activa resulta igualmente razonable. Sin embargo, en los sistemas más complejos, donde la CPU tiene otras tareas que realizar mientras dura la E/S, la espera activa es ineficiente. Es necesario un método mejor para realizar la E/S.

5.2.3 E/S por interrupciones

Consideremos ahora el caso en el cual tenemos que imprimir la cadena de caracteres utilizando una impresora que no guarda temporalmente los caracteres en un búfer, sino que imprime inmediatamente cada carácter según le llega. Si suponemos que la impresora puede imprimir 100 caracteres/segundo, tenemos que cada carácter se imprime en unos 10 milisegundos. Eso significa que después de escribir un carácter en el registro de datos de la impresora, la CPU permanecerá dando vueltas en un bucle ocioso (que no realiza trabajo útil) durante 10 milisegundos a la espera de que se le permita enviar el siguiente carácter. Ese tiempo es más que suficiente para realizar un cambio de contexto y ejecutar algún otro proceso durante esos 10 milisegundos que se desperdiciarían de otra manera.

La forma de permitir que la CPU haga otra cosa mientras espera a que la impresora esté lista es utilizar interrupciones. Cuando se efectúa la llamada al sistema para imprimir la cadena, el búfer se copia al espacio del núcleo, como vimos antes, y el primer carácter se copia a la impresora tan pronto como esté dispuesta a aceptar uno. En ese momento la CPU llama al planificador y pasa a ejecutarse algún otro proceso. El proceso que pidió que se imprimiera la cadena quedará bloqueado hasta que termine de imprimirse toda la cadena. En la Figura 5-8(a) se muestra el trabajo que se realiza durante la llamada al sistema.

```

copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*pwriter_data_register = p[0];
scheduler();

if (count == 0) {
    unblock_user();
} else {
    *pwriter_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();

```

(a)

(b)

Figura 5-8. Escritura de una cadena de caracteres por la impresora utilizando E/S dirigida por interrupciones. (a) Código ejecutado cuando se realiza la llamada al sistema para imprimir. (b) Rutina de tratamiento de la interrupción.

Cuando la impresora termina de imprimir el carácter y está preparada para aceptar el siguiente, genera una interrupción que detiene el proceso actual y guarda su estado. A continuación se ejecuta la rutina de tratamiento de la interrupción (también llamada a veces procedimiento de servicio de la interrupción o manejador de la interrupción). En la Figura 5-8(b) se muestra una versión poco elaborada de este código. Si ya no hay más caracteres que imprimir, el manejador de la interrupción realiza alguna acción para desbloquear al usuario. En caso contrario, envía a la impresora el siguiente carácter, acusa recibo de la interrupción al controlador y retorna al proceso que se estaba ejecutando justo antes de la interrupción, que continúa desde el mismo punto donde se había quedado.

5.2.4 E/S por DMA

Una desventaja obvia de la E/S controlada por interrupciones es que tiene lugar una interrupción por cada carácter. Las interrupciones ocupan su tiempo, por lo que este esquema desperdicia cierta cantidad de tiempo de CPU. Una solución es utilizar DMA. Aquí la idea consiste en dejar que el controlador de DMA envíe los caracteres a la impresora uno a uno, sin que la CPU tenga que intervenir. En esencia, el DMA es E/S programada, sólo que el controlador de DMA es el que realiza todo el trabajo y no la CPU principal. En la Figura 5-9 se presenta un bosquejo del código.

| | |
|---|--|
| <pre>copy_from_user(buffer, p, count); set_up_DMA_controller(); sheduler();</pre> | <pre>acknowledge_interrupt(); unlock_user(); return_from_interrupt();</pre> |
| (a) | (b) |

Figura 5-9. Impresión de una cadena utilizando DMA. (a) Código que se ejecuta cuando se efectúa la llamada al sistema para imprimir. (b) Rutina de tratamiento de la interrupción.

La gran ventaja del DMA es que se reduce el número de interrupciones, pasando de una interrupción por carácter, a una única interrupción por búfer impreso. Si hay muchos caracteres para imprimir y las interrupciones son lentas, el ahorro de tiempo de CPU puede suponer una mejora considerable del sistema. Por otra parte, el controlador de DMA suele ser mucho más lento que la CPU principal. Si el controlador de DMA no puede operar el dispositivo a su máxima velocidad, o si la CPU de todas maneras no tiene nada que hacer mientras espera la interrupción del DMA, puede ser preferible utilizar E/S dirigida por interrupciones o incluso E/S programada.

5.3 CAPAS DEL SOFTWARE DE E/S

El software de E/S está organizado típicamente en cuatro capas, como se muestra en la Figura 5-10. Cada capa tiene encomendada una función bien definida y ofrece a las capas adyacentes una interfaz igualmente bien definida. La funcionalidad y las interfaces difieren de un sistema a otro, motivo por el cual nuestro análisis siguiente, que examina todas las capas comenzando por la más baja, no es específico de ninguna máquina concreta.

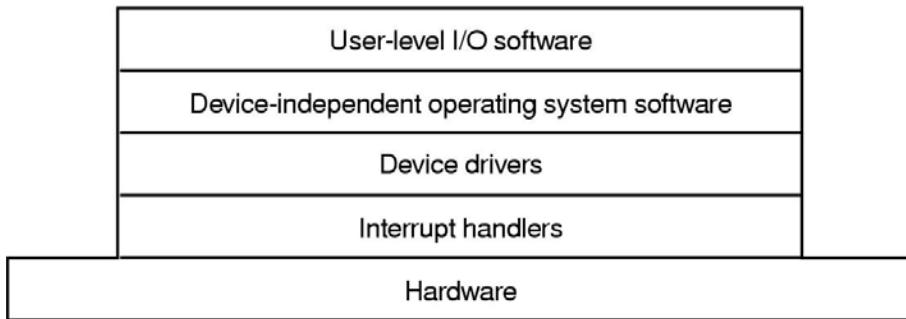


Figura 5-10. Capas del software del sistema de E/S.

5.3.1 Rutinas de tratamiento de las interrupciones

Aunque a veces es útil la E/S programada, en la mayoría de los sistemas de E/S las interrupciones son de esas cosas inevitables que tiene la vida. No obstante es preciso mantener ocultas las interrupciones en las profundidades del sistema operativo, reduciendo al mínimo la parte del sistema operativo que tiene conocimiento de ellas. La mejor manera de ocultar las interrupciones es hacer que el driver que pone en marcha una operación de E/S se bloquee hasta que se complete la E/S y se produzca la interrupción. El driver puede bloquearse a sí mismo por ejemplo ejecutando una operación **bajar** sobre un semáforo, o un **wait** sobre una variable de condición o un **receive** sobre un mensaje, o algo similar.

Cuando llega la interrupción, la rutina de tratamiento hace lo necesario para atender a la interrupción, tras lo cual puede desbloquear el driver que programó esa interrupción. En algunos casos la rutina de tratamiento simplemente ejecutará un **subir** sobre un semáforo. En otros casos, ejecutará un **signal** sobre una variable de condición de un monitor. En otros casos diferentes, enviará un mensaje al driver bloqueado. En absolutamente todos los casos, el efecto neto de la interrupción será que un driver que antes estaba bloqueado pasará ya a poder ejecutarse. Este modelo funciona mejor si los drivers se estructuran como procesos del núcleo, con sus propios estados, pilas y contadores de programa.

Por supuesto, la realidad no es tan sencilla. Procesar una interrupción no consiste tan solo en tomar la interrupción, ejecutar un **subir** sobre algún semáforo y ejecutar una instrucción **IRET** para retornar de la interrupción al proceso anterior. Se requiere por parte del sistema operativo la realización de mucho más trabajo que vamos a esbozar como una serie de pasos que deben realizarse por software después de que el hardware acepte la interrupción. Los detalles dependen mucho del sistema concreto, por lo que algunos de los pasos siguientes podrían no ser necesarios en una máquina dada, pudiendo ser necesarios otros pasos que no están incluidos en la lista. Además, en algunas máquinas los pasos podrían tener lugar en un orden muy diferente.

1. Guardar los registros (incluida la PSW) que no haya guardado aún el hardware de interrupciones.
2. Establecer el contexto adecuado para la ejecución de la rutina de tratamiento de la interrupción. Esto podría implicar establecer la TLB, la MMU y una tabla de páginas.

3. Disponer una pila para su uso por parte de la rutina de tratamiento de la interrupción.
4. Enviar el acuse de la recepción de la interrupción al controlador de interrupciones. Si no hay un controlador de interrupciones centralizado, volver a habilitar las interrupciones.
5. Copiar los registros de donde se guardaron (posiblemente de alguna pila) a la tabla de procesos.
6. Ejecutar la rutina de tratamiento de la interrupción, la cual necesitará leer la información contenida en los registros del controlador de dispositivo que interrumpió.
7. Escoger el proceso que se ejecutará a continuación. Si la interrupción provocó que algún proceso de alta prioridad que estaba bloqueado pasara a estar listo, podría suceder que ese proceso fuera escogido para ejecutarse ahora.
8. Establecer el contexto de la MMU para el proceso que se ejecutará a continuación. También podría ser necesario preparar la TLB.
9. Cargar los registros del nuevo proceso, incluida su PSW.
10. Comenzar a ejecutar el nuevo proceso.

Como puede apreciarse, el procesamiento de las interrupciones dista mucho de ser trivial y requiere un número considerable de instrucciones de la CPU, sobre todo en las máquinas en las que se utiliza memoria virtual siendo preciso preparar tablas de páginas o guardar el estado de la MMU (por ejemplo los bits R y M). En algunas máquinas podría ser necesario reajustar también la TLB y la caché de la CPU al conmutar de modo usuario a modo supervisor, lo que puede suponer muchos ciclos de máquina adicionales.

5.3.2 Drivers de dispositivo

Anteriormente en este capítulo hemos echado un vistazo a lo que hacen los controladores de dispositivo. Vimos que cada controlador tiene algunos registros de dispositivo utilizados para enviar comandos al dispositivo o algunos registros de dispositivo utilizados para conocer su estado, o ambos. El número de registros de dispositivo y la naturaleza de los comandos varía radicalmente de un dispositivo a otro. Por ejemplo, un controlador de ratón tiene que aceptar información del ratón indicándole la distancia que se ha desplazado y qué botones están presionados. En contraste, un controlador de disco necesita información sobre los sectores, las pistas, los cilindros, las cabezas, la dirección de movimiento del brazo, los motores, el tiempo de estabilización de las cabezas y todos los demás aspectos mecánicos que se requieren para que funcione correctamente. Obviamente estos dos controladores tienen que ser muy diferentes.

Como consecuencia, cada dispositivo de E/S conectado a un ordenador necesita algún código específico de ese dispositivo que lo controle. Ese código, denominado el **driver del dispositivo**, está normalmente escrito por el fabricante del dispositivo que lo proporciona junto con el dispositivo. Dado que cada sistema operativo necesita sus propios drivers, los fabricantes suelen proporcionar sus drivers para varios de los sistemas operativos más utilizados.

Cada driver de dispositivo maneja normalmente un tipo de dispositivo o, cuando más, una clase de dispositivos estrechamente relacionados. Por ejemplo, usualmente un driver de disco SCSI puede manejar varios discos SCSI de diferentes tamaños y diferentes velocidades, y quizás también un CD-ROM SCSI. Por otra parte, un ratón y un joystick son tan diferentes que casi siempre necesitan drivers diferentes. Sin embargo no existe ninguna restricción técnica que

impida que un driver de dispositivo controle varios dispositivos que no tengan relación entre sí; simplemente resulta que no es una buena idea.

Para acceder al hardware del dispositivo, es decir a los registros del controlador, normalmente es necesario que el driver forme parte del núcleo del sistema operativo, al menos con las arquitecturas actuales. Realmente, es posible construir drivers que se ejecuten en el espacio del usuario, con llamadas al sistema para leer y escribir en los registros del dispositivo. De hecho, ese diseño podría ser una buena idea, ya que conseguiría aislar al núcleo de los drivers y a los drivers entre sí. Haciendo las cosas así podría eliminarse una de las principales causas de caídas del sistema: drivers con errores que interfieren con el núcleo de una manera u otra. Sin embargo, dado que los sistemas operativos actuales esperan que los drivers se ejecuten en el núcleo, ése es el modelo que vamos a considerar aquí.

Puesto que los diseñadores de cualquier sistema operativo saben que se instalarán en él fragmentos de código (drivers) escritos por otras personas, es preciso que utilicen en su diseño una arquitectura que permita tal instalación. Eso significa tener un modelo bien definido de lo que hace un driver y de cómo interactúa con el resto del sistema operativo. Los drivers de dispositivo se ubican comúnmente por debajo del resto del sistema operativo, como se ilustra en la Figura 5-11.

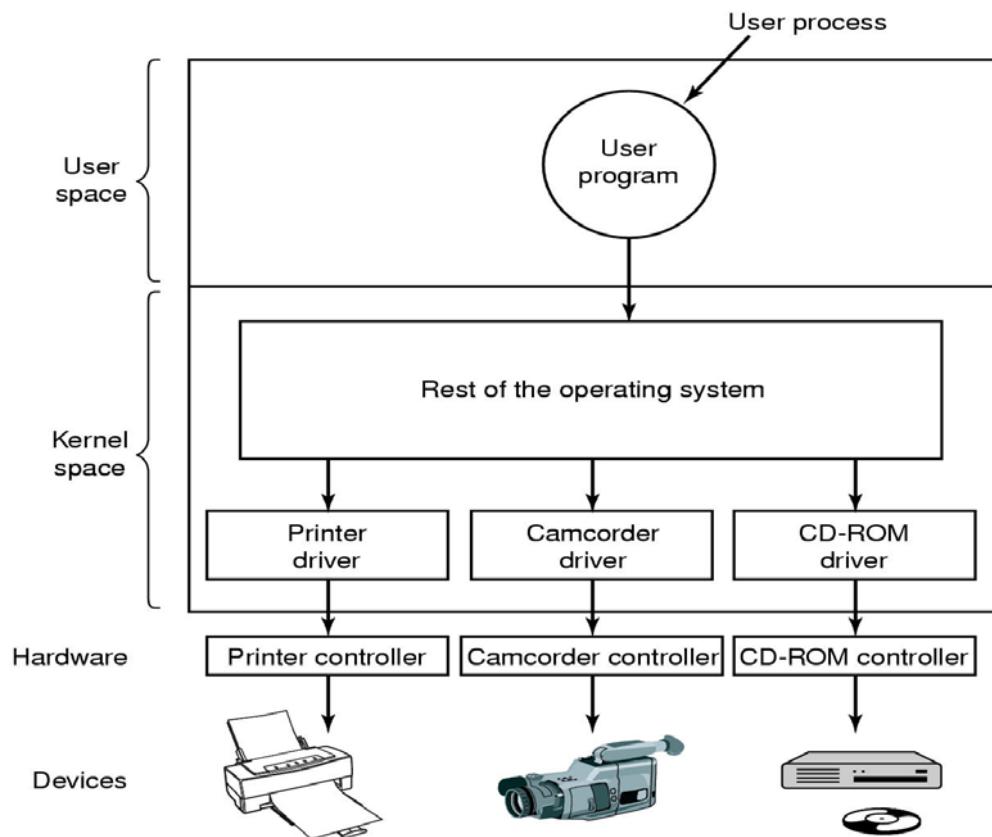


Figura 5-11. Ubicación lógica de los drivers de dispositivo. En realidad, toda la comunicación entre los drivers y los controladores de dispositivo se realiza a través del bus.

Usualmente los sistemas operativos clasifican los drivers en unas cuantas categorías. Las más comunes son los **dispositivos de bloques**, como los discos, que contienen múltiples bloques de datos susceptibles de direccionarse independientemente, y los **dispositivos de caracteres**, como los teclados e impresoras, que generan o aceptan un flujo de caracteres.

La mayoría de los sistemas operativos definen una interfaz estándar que todos los drivers de bloques deben soportar y una segunda interfaz estándar que todos los drivers de caracteres deben soportar. Tales interfaces consisten en varios procedimientos que el resto del sistema operativo puede invocar para pedir al driver que realice algún trabajo. Los procedimientos más comunes son los que leen un bloque (dispositivo de bloques) o los que escriben una cadena de caracteres (dispositivo de caracteres).

En algunos sistemas, el sistema operativo es un único programa binario que contiene todos los drivers que pueda necesitar compilados dentro de él. Este esquema fue la norma durante años en los sistemas UNIX debido a que se ejecutaban en centros de cálculo en los que los dispositivos de E/S raramente cambiaban. Si se añadía un nuevo dispositivo el administrador del sistema simplemente recompilaba el núcleo con el nuevo controlador para obtener un nuevo binario.

Con la llegada de los ordenadores personales, con su miríada de dispositivos de E/S, este modelo dejó de funcionar bien. Pocos usuarios son capaces de recompilar o reenlazar el núcleo incluso si cuentan con el código fuente o con los módulos objeto, lo cual no siempre es el caso. Por ese motivo, comenzando con MS-DOS los sistemas operativos cambiaron a un modelo en el que los drivers se cargan dinámicamente en el sistema durante la ejecución. Cada sistema diferente maneja la carga de los drivers de manera diferente.

Un driver de dispositivo tiene varias funciones. La más obvia es la de aceptar peticiones de lectura o escritura abstractas enviadas por el software independiente del dispositivo y controlar que se lleven a cabo, pero existen también otras funciones que debe realizar. Por ejemplo, el driver debe inicializar el dispositivo, en caso de ser necesario. También puede tener que controlar su consumo de energía eléctrica y mantener un registro de eventos.

Muchos drivers de dispositivo tienen una estructura general similar. Un driver típico comienza comprobando los parámetros de entrada para ver si son válidos. Si no lo son, se devuelve un error. Si los parámetros son válidos, puede ser necesaria una traducción de términos abstractos a concretos. En el caso de un driver de disco, esto puede significar convertir un número de bloque lineal en los números de cabeza, pista, sector y cilindro correspondientes de acuerdo con la geometría del disco.

Luego el driver puede comprobar si el dispositivo está actualmente en uso. En tal caso, la petición deberá encolarse para su procesamiento posterior. Si el dispositivo está desocupado, debe examinarse el estado del hardware para ver si puede atenderse ya la petición. Puede ser necesario encender el dispositivo o poner en marcha un motor antes de comenzar las transferencias. Una vez que está encendido el dispositivo y listo para trabajar, es cuando comienza el control propiamente dicho del dispositivo.

Controlar el dispositivo significa enviarle una secuencia de comandos. El driver es el lugar donde se determina la secuencia de comandos, dependiendo de la tarea a realizar. Después de que el driver determina qué comandos tiene que enviar, comienza a escribirlos en los registros de dispositivo del controlador. Después de escribir cada comando en el controlador, puede ser necesario comprobar si el controlador ha aceptado el comando y está preparado para aceptar el siguiente. Esto se repite hasta terminar de enviar toda la secuencia de comandos. Algunos controladores son capaces de leer y procesar por sí mismos toda una lista enlazada de comandos (en la memoria) sin ninguna ayuda del sistema operativo posterior a la petición de procesamiento de la lista.

Una vez que se envían los comandos pueden darse dos situaciones. En muchos casos el driver del dispositivo debe esperar hasta que el controlador realice algún trabajo, por lo que se bloquea a sí mismo hasta que llega la interrupción que lo desbloquea. Sin embargo, en otros casos, la operación termina de inmediato, por lo que el driver del dispositivo no necesita

bloquearse. Un ejemplo de la segunda situación es el desplazamiento vertical (*scroll*) de la pantalla en modo texto, el cual sólo requiere escribir unos cuantos bytes en los registros del controlador. No se requiere ningún movimiento mecánico, por lo que toda la operación puede completarse en nanosegundos.

En el primer caso, el driver bloqueado debe ser despertado por la interrupción. En el segundo caso, el driver nunca se duerme. En ambos casos, una vez que se completa la operación, el driver del dispositivo debe comprobar si se ha producido algún error. Si todo fue bien, el controlador dispone ya de los datos que debe comunicar al software independiente del dispositivo (por ejemplo, un bloque que acaba de leerse). Finalmente, el driver devuelve cierta información de estado a quien lo invocó para informarle de si todo salió bien o de si hubo errores, y cuáles fueron. Si hay más peticiones pendientes en la cola, se selecciona una de ellas y se arranca. Si por el contrario la cola está vacía, el driver se bloquea a la espera de la siguiente petición.

Este sencillo modelo no es más que una burda aproximación a la realidad. Hay muchos factores que hacen que el código sea mucho más complicado. Por ejemplo, un dispositivo de E/S podría terminar mientras un driver está ejecutándose, interrumpiendo por tanto al driver. La interrupción puede provocar que se ejecute un driver de dispositivo. De hecho, podría provocar que se ejecutase nuevamente el driver actual. Por ejemplo, mientras el driver de red está procesando un paquete entrante, podría llegar otro paquete. Consecuentemente, los drivers deben ser **reentrantes**, lo que significa que un driver que está ejecutándose, debe tener prevista la posibilidad de que se le invoque una segunda vez antes de que haya terminado la primera llamada.

En un sistema que permite la conexión en caliente es posible añadir o quitar dispositivos mientras el ordenador está funcionando. Como resultado, puede suceder que mientras un driver está ocupado leyendo de algún dispositivo, el sistema operativo informe de que el usuario ha quitado repentinamente el dispositivo del sistema. En ese caso no solamente es necesario abortar la transferencia de E/S actual sin dañar ninguna estructura de datos del núcleo, sino que deberán eliminarse con sumo cuidado del sistema todas las peticiones pendientes para el dispositivo ahora desaparecido, comunicando la mala noticia a quienes hicieron esas peticiones. Además, la adición inesperada de nuevos dispositivos podría obligar al núcleo a reasignar recursos (por ejemplo, líneas de solicitud de interrupción), quitándole algunos recursos al driver y dándole otros nuevos a cambio.

Los drivers a menudo necesitan interactuar con el resto del núcleo. Aunque desde los drivers no pueden realizarse llamadas al sistema, usualmente sí que se les permite que realicen llamadas a ciertos procedimientos del núcleo. Por ejemplo, usualmente hay llamadas para asignar y liberar páginas de memoria para utilizarlas como búferes. Otras llamadas útiles son necesarias para gestionar la MMU, los timers, el controlador de DMA, el controlador de interrupciones, etc.

5.3.3 Software de E/S independiente del dispositivo

Aunque una parte del software de E/S es específica para los dispositivos concretos existentes en el sistema, otras partes son independientes del dispositivo. La frontera exacta entre los drivers y el software independiente del dispositivo depende del sistema (y del dispositivo), porque algunas funciones que podrían realizarse con independencia del dispositivo en realidad se llevan a cabo en los drivers por cuestiones de eficiencia u otras razones. Las funciones que se muestran en la Figura 5-12 se realizan típicamente en el software independiente del dispositivo.

| |
|--|
| Interfaz uniforme con los drivers de los dispositivos |
| Buffering |
| Informar de los errores |
| Asignación y liberación de dispositivos dedicados |
| Proporcionar un tamaño de bloque independiente del dispositivo |

Figura 5-12. Funciones del software de E/S independiente del dispositivo.

La función básica del software independiente del dispositivo es realizar las operaciones de E/S que son comunes a todos los dispositivos y presentar una interfaz uniforme al software a nivel de usuario. A continuación examinaremos los aspectos anteriores con más detalle.

Interfaz uniforme con los drivers de los dispositivos

Una cuestión principal en un sistema operativo es cómo conseguir que todos los dispositivos de E/S y sus drivers tengan un aspecto más o menos similar. Si la interfaz con los discos, impresoras, teclados, etc., es muy diferente para cada caso, cada vez que se añade un nuevo dispositivo al sistema será preciso modificar el sistema operativo para ese nuevo dispositivo. No es una buena idea el que haya que hacer este tipo de retoques técnicos en el sistema operativo cada vez que se añade un nuevo dispositivo.

Un aspecto de esta cuestión es la interfaz entre los drivers de dispositivo y el resto del sistema operativo. En la Figura 5-13(a) se ilustra una situación en la que cada driver de dispositivo tiene una interfaz diferente con el sistema operativo, lo que significa que las funciones del driver que el sistema puede invocar difieren de un driver a otro. También podría significar que las funciones del núcleo que necesita el driver difieren también de un controlador a otro. En conjunto, todo esto significa que la interfaz con cada nuevo driver requiere un elevado nuevo esfuerzo de programación.

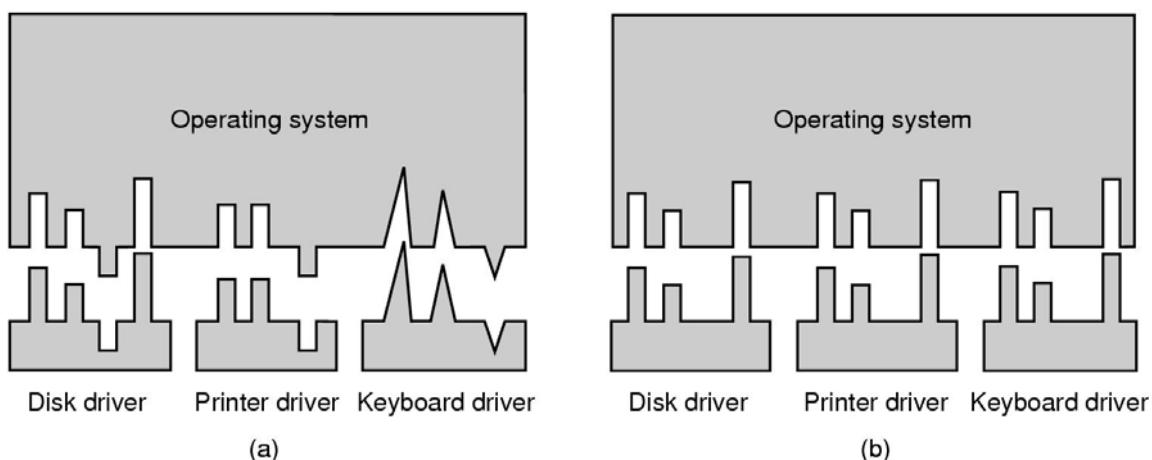


Figura 5-13. (a) Sin una interfaz estándar con los drivers.
(b) Con una interfaz estándar con los drivers.

En contraste, en la Figura 5-13(b) se muestra un diseño diferente en el cual todos los drivers tienen la misma interfaz. En este caso resulta mucho más fácil añadir un nuevo driver,

siempre y cuando se ajuste a la interfaz con los drivers existentes. Además en este caso los escritores de drivers conocen perfectamente lo que se espera de ellos (es decir, qué funciones debe proporcionar el driver y qué funciones del núcleo puede invocar el driver). En la práctica, no todos los dispositivos son absolutamente idénticos, pero usualmente sólo hay un pequeño número de tipos de dispositivos e incluso esos tipos son generalmente casi el mismo. Por ejemplo, incluso los dispositivos de bloques y de caracteres tienen muchas funciones en común.

Otro aspecto de tener una interfaz uniforme es la forma en la que se nombran los dispositivos de E/S. El software independiente del dispositivo se encarga de mapear los nombres simbólicos de los dispositivos sobre el driver correcto. Por ejemplo, en UNIX un nombre de dispositivo, como `/dev/disk0`, especifica únicamente el i-nodo de un fichero especial, y ese i-nodo contiene el **número de dispositivo mayor**, que se utiliza para localizar el driver apropiado. El i-nodo contiene también el **número de dispositivo menor**, que se pasa como parámetro al driver en orden a especificar la unidad de la que se va a leer o a escribir. Todos los dispositivos tienen número de dispositivo mayor y menor, y el acceso a los drivers se realiza utilizando el número de dispositivo mayor para seleccionar el driver.

La protección es algo que está íntimamente relacionado con la forma de nombrar a los dispositivos. ¿Cómo puede impedir el sistema que los usuarios puedan acceder a dispositivos para los que no tienen derecho a acceder? Tanto en UNIX como en Windows 2000 los dispositivos aparecen en el sistema de ficheros como objetos con nombre, lo que significa que las reglas de protección habituales para los ficheros son también aplicables a los dispositivos de E/S. El administrador del sistema puede establecer así los permisos correctos para cada dispositivo.

Buffering

Por varias razones la utilización de búferes es otra cuestión importante, tanto para los dispositivos de bloques como para los de caracteres. Para entender una de ellas consideremos un proceso que quiere leer datos desde un módem. Una posible estrategia para tratar el flujo de caracteres que llegan es hacer que el proceso de usuario realice una llamada al sistema `read` y se bloquee a la espera del siguiente carácter. La llegada de cada carácter provoca una interrupción. La rutina de tratamiento de esa interrupción entrega el carácter al proceso de usuario, desbloqueándolo a continuación. Después de dejar el carácter en algún lado, el proceso lee otro carácter y se bloquea de nuevo. Este modelo se ilustra en la Figura 5-14(a).

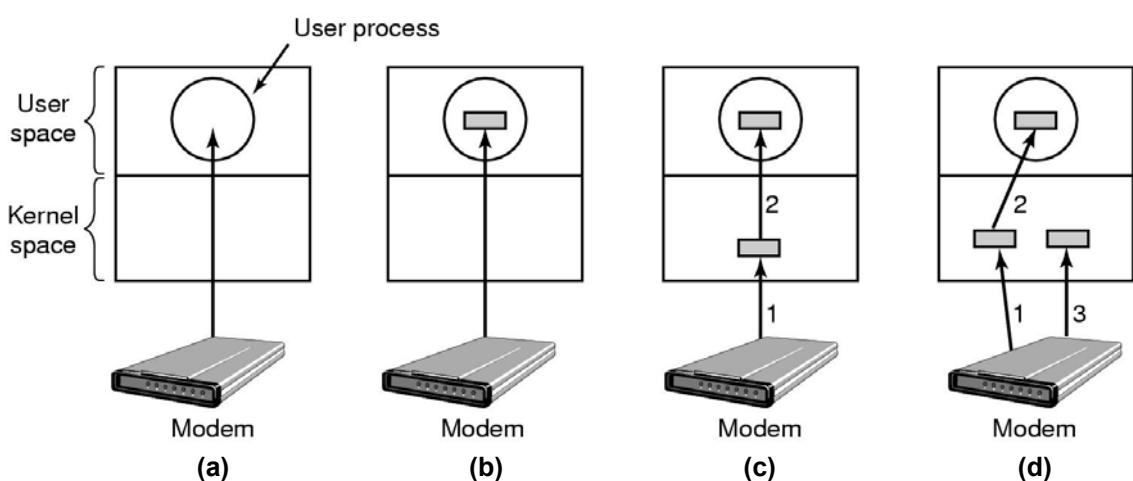


Figura 5-14. (a) Entrada sin búfer. (b) Buffering en el espacio del usuario. (c) Buffering en el núcleo seguido de copia en el espacio de usuario. (d) Doble buffering en el núcleo.

El problema con esa forma de hacer negocios es que el proceso de usuario tiene que ponerse en marcha y bloquearse por cada carácter que llega. Poner en marcha un proceso muchas veces durante cortos lapsos de tiempo resulta ineficiente, por lo que este no es un buen diseño.

En la Figura 5-14(b) se ilustra una mejora que consiste en que el proceso de usuario proporciona un búfer de n caracteres en el espacio de usuario y realiza una lectura de n caracteres. La rutina de tratamiento de la interrupción deja los caracteres que llegan en ese búfer hasta que se llena, momento en el que despierta al proceso de usuario. Este esquema es mucho más eficiente que el anterior, pero también tiene un inconveniente: ¿qué sucede si en el momento en que llega un carácter las páginas donde está el búfer se encuentran en el disco al haber sido víctimas de los algoritmos de sustitución? Podríamos bloquear las páginas del búfer en la memoria, pero si muchos procesos comienzan a bloquear páginas en la memoria, tendremos una reducción en la reserva de páginas disponibles con una consiguiente degradación del rendimiento del sistema.

Un tercer enfoque consiste en crear un búfer dentro del núcleo y hacer que el manejador de la interrupción deje allí los caracteres, como se muestra en la Figura 5-14(c). Cuando el búfer se llena, se carga del disco la página que contiene el búfer del usuario, si es necesario, y el búfer del núcleo se copia allí en una única operación. Este esquema es mucho más eficiente.

Sin embargo, incluso este esquema tiene un problema: ¿qué sucede con los caracteres que llegan mientras se está cargando del disco la página que contiene el búfer del usuario? Puesto que el búfer está lleno, no tenemos ningún sitio dónde dejarlos. Una salida es tener un segundo búfer en el núcleo. Después de que se llene el primer búfer, y mientras se copia al espacio del usuario, se utiliza el segundo búfer, como se muestra en la Figura 5-14(d). Cuando se llene el segundo búfer, podrá copiarse en el búfer de usuario (suponiendo que el usuario lo pidió). Mientras el segundo búfer se está copiando en el espacio de usuario, el primero puede utilizarse para guardar los nuevos caracteres. De esta manera los dos búferes van alternándose: mientras uno se está copiando en el espacio de usuario, el otro está acumulando las nuevas entradas. Este esquema de utilización de búferes se denomina **doble buffering**.

Los búferes son también importantes en las operaciones de salida. Consideremos, por ejemplo, la forma en la que realiza la salida de datos hacia el módem siguiendo el modelo de la Figura 5-14(b). El proceso de usuario ejecuta una llamada al sistema `write` para enviar a la salida n caracteres. En ese momento, el sistema tiene dos opciones. Puede bloquear al usuario hasta que todos los caracteres se hayan escrito, pero eso podría significar mucho tiempo sobre una lenta línea de teléfono. Por otro lado, el sistema también podría liberar al usuario inmediatamente y realizar la E/S mientras el proceso de usuario sigue con sus cálculos, pero esto nos conduce a un problema todavía peor: ¿cómo puede saber el proceso de usuario cuándo termina la operación de salida y cuándo puede volver a utilizar el búfer? El sistema podría generar una señal o una interrupción software, pero ese estilo de programación es difícil y muy propenso a sufrir condiciones de carrera. Una solución mucho mejor es que el núcleo copie los datos en un búfer del núcleo, de forma análoga a como se hace en la Figura 5-14(c) (pero en el otro sentido) y desbloquee inmediatamente al proceso. Ahora no importa cuándo termina la E/S real; el usuario puede volver a utilizar el búfer en el instante mismo en que se le desbloquea.

El uso de búferes es una técnica extensamente utilizada, pero que también tiene sus desventajas. Si los datos van pasando sucesivamente por demasiados búferes, se produce una merma en el rendimiento. Consideremos por ejemplo la red de la Figura 5-15 y supongamos que un usuario realiza una llamada al sistema para escribir en la red. El núcleo copia el paquete a un búfer del núcleo para que el usuario pueda seguir trabajando de inmediato (paso 1).

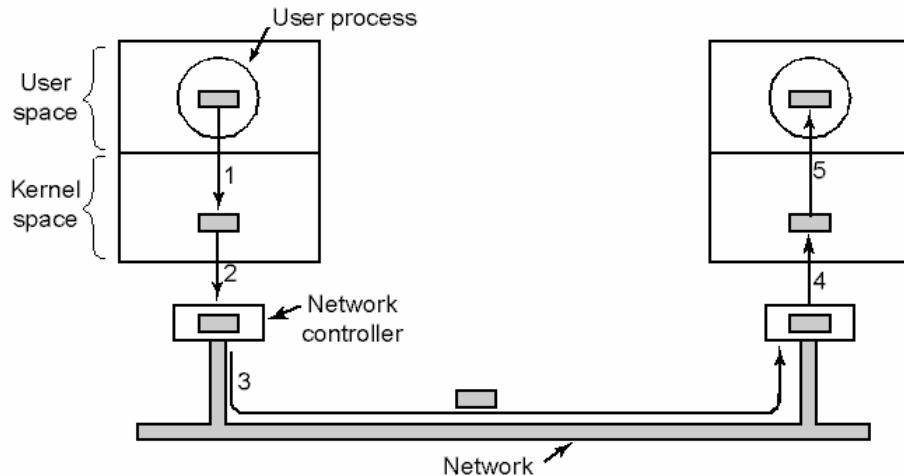


Figura 5-15. Múltiples copias sucesivas de un paquete enviado a través de una red.

Cuando se llama al driver, éste se encarga de copiar el paquete al controlador para proceder a su envío (paso 2). La razón por la cual el driver no envía directamente el paquete a la línea desde la memoria del núcleo es que una vez que se inicia la transmisión de un paquete ésta debe continuar a una velocidad uniforme. El driver no puede garantizar que vaya a conseguir acceder a la memoria a una velocidad uniforme, ya que los canales de DMA y otros dispositivos de E/S podrían estar robando a la CPU muchos ciclos de memoria. Cualquier retraso excesivo en la lectura de una palabra, echará a perder todo el paquete. Este problema se evita almacenando previamente a su transmisión todo el paquete dentro del controlador.

Después de copiar el paquete en el búfer interno del controlador, se copia a través de la red (paso 3). Los bits llegan al receptor poco después de haber sido enviados, así que, casi inmediatamente después de enviarse el último bit, ese bit llega al receptor, donde el paquete queda almacenado en un búfer dentro del controlador. A continuación el paquete se copia al búfer del núcleo del receptor (paso 4). Finalmente, se copia al búfer del proceso receptor (paso 5). Usualmente, el receptor envía entonces un acuse de recibo. Cuando el transmisor recibe el acuse de recibo, puede iniciar la transmisión del siguiente paquete. Sin embargo debe quedar claro que todas esas copias van a reducir considerablemente la velocidad de transmisión porque todos los pasos deben proceder secuencialmente unos después de otros.

Informe de errores

Los errores son mucho más comunes en el contexto de E/S que en otros contextos. Cuando se presentan, el sistema operativo debe solventarlos lo mejor que pueda. Muchos errores son específicos del dispositivo por lo que su tratamiento sólo puede llevarlo a cabo el correspondiente driver. No obstante el marco general del tratamiento de los errores es independiente del dispositivo.

Una clase particular de errores de E/S es la de los errores de programación. Éstos se presentan cuando un proceso solicita algo imposible, como escribir en un dispositivo de entrada (teclado, ratón, escáner, etc.) o leer de un dispositivo de salida (impresora, ploter, etc.). Otros errores frecuentes consisten en proporcionar una dirección de búfer u otro parámetro inválido, o en especificar un dispositivo inválido (por ejemplo, el disco 3 cuando el sistema sólo tiene dos discos). La medida a tomar ante tales errores es sencilla: devolver un código de error al proceso que efectuó la llamada al sistema para hacer E/S.

Otra clase de errores es la clase de los verdaderos errores de E/S, como tratar de escribir en un sector del disco que está dañado o tratar de leer de una cámara de vídeo que acaba de apagarse. En esas circunstancias, corresponde al driver determinar qué es lo que hay que hacer. Si el driver no sabe lo que hacer, tendrá que comunicar el problema al software independiente del dispositivo.

Lo que haga ese software dependerá del entorno y de la naturaleza del error. Si se trata de un simple error de lectura y la operación ha sido solicitada por un usuario interactivo, podría desplegarse una ventana de diálogo para preguntarle qué es lo que desea que se haga en respuesta al error. Las opciones podrían incluir reintentar el acceso cierto número de veces, ignorar el error o terminar el proceso que solicitó la E/S. Si no hay ningún usuario al que poder consultar, probablemente la única opción viable sea dar por fallida la llamada al sistema devolviendo un código de error.

Sin embargo, algunos errores no pueden tratarse de esta manera. Por ejemplo, podría haberse destruido una estructura de datos crucial, como el directorio raíz o la lista de bloques libres, y en tal caso, el sistema tendrá que mostrar un mensaje de error y detenerse completamente para evitar que el daño se propague.

Asignar y liberar dispositivos dedicados

Algunos dispositivos, como las grabadoras de CD-ROM, sólo pueden ser utilizados por un proceso a la vez. Corresponde al sistema operativo examinar las solicitudes de uso de los dispositivos y aceptarlas o rechazarlas, dependiendo de si el dispositivo solicitado está disponible o no. Una forma sencilla de gestionar estas solicitudes es requerir a los procesos que realicen directamente llamadas `open` sobre los ficheros especiales de los dispositivos. Si el dispositivo no está disponible el `open` falla. El cierre del fichero especial libera el dispositivo.

Un enfoque alternativo sería contar con mecanismos especiales para solicitar y liberar dispositivos dedicados. Un intento de adquirir un dispositivo que no está disponible, bloqueará al proceso, en lugar de fallar. Los procesos bloqueados esperan en una cola asociada al dispositivo. tarde o temprano, el dispositivo solicitado estará disponible y se permitirá que el primer proceso de la cola lo adquiera y continúe con su ejecución.

Tamaño del bloque independiente del dispositivo

Los diferentes discos podrían tener diferentes tamaños de sector. Corresponde al software independiente del dispositivo ocultar estas diferencias y proporcionar un tamaño de bloque uniforme a las capas superiores, por ejemplo, tratando varios sectores como un único bloque lógico. De esta manera, las capas superiores sólo tratarán con dispositivos abstractos, todos los cuales utilizan el mismo tamaño de bloque lógico, independiente del tamaño del sector físico. Similarmente, algunos dispositivos de caracteres suministran sus datos byte a byte (como los módems), mientras que otros suministran los suyos en unidades más grandes (como las interfaces de red). Por supuesto, es posible ocultar también estas diferencias.

5.3.4 Software de E/S en el espacio de usuario

Aunque casi todo el software de E/S está dentro del sistema operativo, una pequeña porción consiste en bibliotecas enlazadas junto con los programas de usuario, e incluso en programas enteros que se ejecutan fuera del núcleo. Los procedimientos de biblioteca normalmente realizan llamadas al sistema, incluyendo las correspondientes a la E/S. Cuando un programa en C contiene la llamada

```
contador = write(fd, buffer, nbytes) ;
```

el procedimiento de biblioteca *write* debe enlazarse con el programa por lo que estará contenido en el programa binario presente en la memoria en tiempo de ejecución. Es obvio que el conjunto de todos estos procedimientos de biblioteca forma parte del sistema de E/S.

Si bien estos procedimientos no hacen mucho más que colocar sus parámetros en el lugar apropiado para la llamada al sistema, hay otros procedimientos de E/S que sí efectúan un trabajo real. En particular, son estos procedimientos de biblioteca quienes realizan el formateo de las entradas y salidas. Un ejemplo en C es *printf*, que toma como entrada un string de formato y posiblemente algunas variables, construye un string ASCII y finalmente llama a *write* para enviar ese string a la salida. Como un ejemplo de *printf*, consideremos la instrucción

```
printf(" El cuadrado de %3d es %6d\n", i, i*i) ;
```

Esta instrucción formatea un string consistente en el string de 16 caracteres “ El cuadrado de “ seguido del valor de *i* como un string de 3 caracteres, del string de 4 caracteres “ es “, de i^2 como un string de 6 caracteres y de finalmente un carácter de salto de línea.

Un ejemplo de un procedimiento similar para la entrada es *scanf*, que lee la entrada y la almacena en variables descritas en un string de formato que utiliza la misma sintaxis que *printf*. La biblioteca estándar de E/S contiene varios procedimientos que implican E/S, y todos se ejecutan como parte de los programas de usuario.

No todo el software de E/S en el nivel de usuario consiste en procedimientos de biblioteca. Otra categoría importante es el sistema de spooling. El **spooling** es una forma de tratar los dispositivos dedicados en un sistema con multiprogramación. Consideremos un dispositivo que típicamente se gestiona mediante spooling: una impresora. Aunque desde el punto de vista técnico sería fácil permitir que el proceso de usuario abriese el fichero especial de caracteres correspondiente a la impresora, eso tendría el peligro de que un proceso podría abrir el fichero sin hacer nada con él durante horas. Durante ese tiempo ningún otro proceso podría imprimir nada a pesar de que la impresora no está realmente siendo utilizada.

En vez de eso, lo que se hace es crear un proceso especial, llamado **demonio** (*daemon*), y un directorio especial, llamado **directorio de spooling**. Para imprimir un fichero, lo primero que hace un proceso es generar el fichero completo que desea imprimir, colocándolo a continuación en el directorio de spooling. Corresponde al demonio, que es el único proceso autorizado para usar el fichero especial de la impresora, imprimir los ficheros que están en el directorio. Al proteger el fichero especial contra el uso directo por parte de los usuarios, se elimina el problema de que uno de ellos lo mantenga abierto durante un tiempo innecesariamente largo.

El spooling no sólo se utiliza con las impresoras, sino que también se utiliza en otras situaciones. Por ejemplo a menudo se utiliza un demonio de red para transferir ficheros por una red. Si un usuario quiere enviar un fichero a través de la red, debe colocarlo en un directorio de spooling de la red. Más tarde, el demonio de la red cogerá el fichero y lo transmitirá. Un uso particular de la transmisión de ficheros mediante spooling es el sistema de news USENET. Esta

red consta de millones de máquinas de todo el mundo comunicándose a través de Internet. Existen miles de grupos de news sobre numerosos temas. Para publicar un mensaje de news, el usuario invoca un programa de news, el cual acepta el mensaje a publicar y lo deposita en un directorio de spooling para transmitirlo más tarde a otras máquinas. Todo el sistema de news se ejecuta fuera del sistema operativo.

La Figura 5-16 resume el sistema de E/S, mostrando todas las capas y las principales funciones de cada capa. Comenzando por la base, las capas son: el hardware, las rutinas de tratamiento de las interrupciones, los drivers de dispositivo, el software independiente del dispositivo y finalmente, los procesos de usuario.

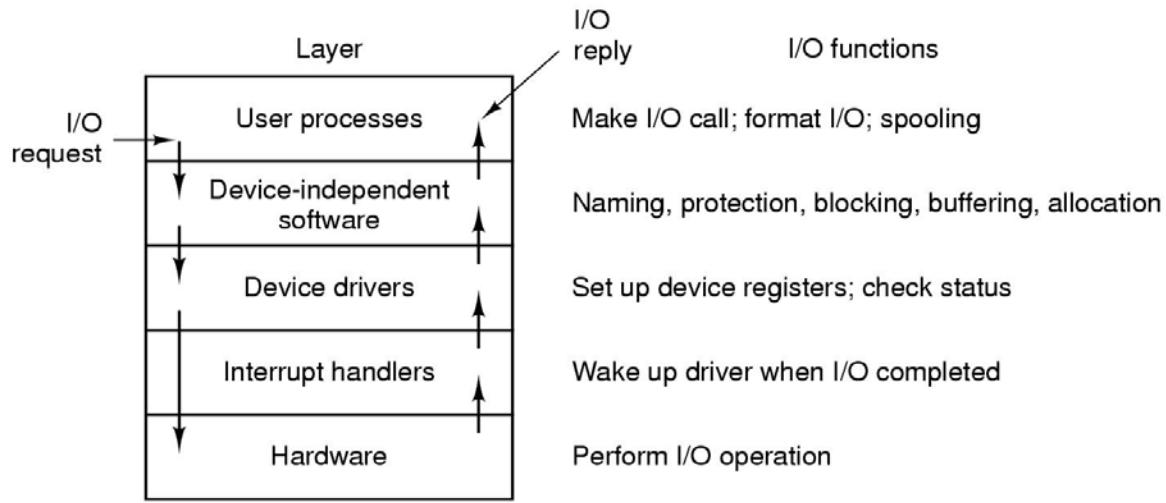


Figura 5-16. Capas del sistema de E/S y funciones principales de cada capa.

Las flechas de la Figura 5-16 muestran el flujo de control. Por ejemplo, cuando un programa de usuario trata de leer un bloque de un fichero se invoca al sistema operativo para que lleve a cabo la llamada. El software independiente del dispositivo busca el bloque en el búfer de la caché, por ejemplo. Si el bloque requerido no está allí, se invoca al driver del dispositivo para que envíe la solicitud al hardware para que obtenga el bloque del disco. Luego el proceso se bloquea hasta que se completa la operación del disco.

Cuando el disco termina, el hardware genera una interrupción. El manejador de la interrupción se ejecuta para descubrir qué es lo que ha sucedido, es decir, qué dispositivo quiere que lo atiendan en ese momento. Luego revisa el estado del dispositivo y despierta al proceso dormido para que finalice la solicitud de E/S y permita que pueda continuar el proceso de usuario.

5.4 DISCOS

Vamos a comenzar a estudiar algunos dispositivos de E/S reales. Empezaremos con los discos para continuar después con los relojes, teclados y pantallas.

5.4.1 Hardware del disco

Existe un enorme variedad de tipos de discos. Los más comunes son los discos magnéticos (discos duros y disquetes), caracterizándose por el hecho de que las lecturas y escrituras son igual de rápidas, lo que los hace ideales como memoria secundaria (paginación, sistemas de ficheros, etc.). A veces se utilizan conjuntamente varios de esos discos para obtener dispositivos de almacenamiento altamente fiables. Son también importantes de cara a la distribución de programas, datos y películas diversos tipos de discos ópticos (CD-ROMs, CD-Regrabables y DVDs). En las siguientes secciones describiremos primero el hardware y luego el software de estos dispositivos.

Discos magnéticos

Los discos magnéticos están organizados en cilindros, cada uno de los cuales contiene tantas pistas como cabezas tenga apiladas verticalmente. Las pistas se dividen en sectores, siendo el número de sectores alrededor de cada circunferencia típicamente de 8 a 32 en los disquetes, y de hasta varios cientos en los discos duros. El número de cabezas varía entre 1 y 16.

Algunos discos magnéticos tienen tan poca electrónica que tan solo pueden suministrar un simple flujo de bits en serie. En estos discos, el controlador realiza la mayor parte del trabajo. En otros discos, en particular en los discos **IDE** (*Integrated Drive Electronics*; Electrónica Integrada en la Unidad), la propia unidad de disco contiene un microcontrolador que realiza una parte del trabajo, permitiendo al controlador real del dispositivo trabajar con un conjunto de comandos de más alto nivel.

Una característica del dispositivo que tiene importantes implicaciones para el driver del disco es la posibilidad de que el controlador realice posicionamientos de las cabezas lectoras de dos o más unidades de disco al mismo tiempo, lo que se denomina **posicionamiento de cabezas solapado** (*overlapped seeks*). Mientras el controlador y el software están esperando a que se complete un posicionamiento en una unidad, el controlador puede iniciar otro posicionamiento en otra unidad. Muchos controladores también pueden leer o escribir en una unidad mientras realizan posicionamientos en otras unidades, pero un controlador de disquete no puede leer o escribir en dos unidades al mismo tiempo. (Leer o escribir requiere que el controlador transfiera bits en una escala de tiempo de microsegundos, por lo que una transferencia requiere casi toda su potencia de cómputo.) La situación es diferente en el caso de los discos duros que tienen controladores integrados, y en un sistema con varios de esos discos duros todos ellos pueden operar simultáneamente, al menos en cuanto a realizar transferencias entre el disco y el búfer del controlador. Sin embargo, sólo puede realizarse una transferencia entre el controlador y la memoria principal a la vez. La capacidad de realizar dos o más operaciones al mismo tiempo puede reducir considerablemente el tiempo de acceso medio.

En la Figura 5-17 se comparan los parámetros del medio de almacenamiento estándar en el PC original de IBM con los parámetros de un disco duro moderno para poner de manifiesto lo mucho que han cambiado los discos en las dos últimas décadas. Es interesante destacar que no todos los parámetros han mejorado en la misma medida. El tiempo de posicionamiento medio es siete veces mejor, la velocidad de transferencia es 1300 veces mejor, mientras que la capacidad ha mejorado en un factor de 50.000. Esta diferencia se debe a que las mejoras en las partes móviles han sido relativamente graduales, mientras que la densidad de bits de las superficies de grabación ha experimentado un aumento muy superior.

| Parameter | IBM 360-KB floppy disk | WD 18300 hard disk |
|--------------------------------|------------------------|--------------------|
| Number of cylinders | 40 | 10601 |
| Tracks per cylinder | 2 | 12 |
| Sectors per track | 9 | 281 (avg) |
| Sectors per disk | 720 | 35742000 |
| Bytes per sector | 512 | 512 |
| Disk capacity | 360 KB | 18.3 GB |
| Seek time (adjacent cylinders) | 6 msec | 0.8 msec |
| Seek time (average case) | 77 msec | 6.9 msec |
| Rotation time | 200 msec | 8.33 msec |
| Motor stop/start time | 250 msec | 20 sec |
| Time to transfer 1 sector | 22 msec | 17 μ sec |

Figura 5-17. Parámetros de disco correspondientes al disquete original de 360 KB del IBM PC y al disco duro de Western Digital WD 18300.

Algo que debe tenerse muy presente al estudiar las especificaciones de los discos duros modernos es que la geometría que se especifica, que es la que utiliza el software del driver, podría ser diferente del formato físico. En los discos más antiguos el número de sectores por pista era el mismo para todos los cilindros. Los discos modernos están divididos en zonas, teniendo las zonas exteriores más sectores que las interiores. La Figura 5-18(a) ilustra un disco pequeño con dos zonas. La zona exterior tiene 32 sectores por pista; la interior, 16. Un disco real, como el WD 18300, suele tener 16 zonas, aumentando el número de sectores aproximadamente un 4% de una zona a la siguiente, a medida que nos movemos desde la zona más interna a la más externa.

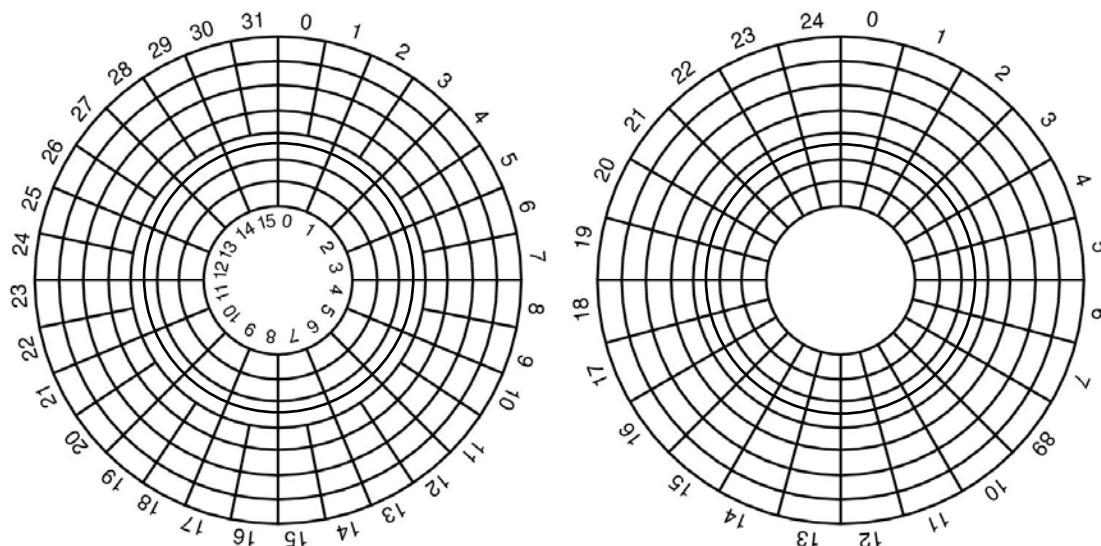


Figura 5-18. (a) Geometría física de un disco con dos zonas.
(b) Una posible geometría virtual para este disco.

Con el fin de ocultar los detalles sobre el número de sectores que hay en cada pista, la mayoría de los discos modernos tiene una geometría virtual que se presenta al sistema operativo. El software actúa como si hubiera x cilindros, y cabezas y z sectores por pista. Luego, el controlador establece la correspondencia entre una petición del sector (x, y, z) y el cilindro, cabeza y sector reales. En la Figura 5-18(b) se muestra una posible geometría virtual para el disco físico de la Figura 5-18(a). En ambos casos el disco tiene 192 sectores ($4 \times 32 + 4 \times 16 + 8 \times 24$), sólo que la disposición publicada es diferente de la real.

En los ordenadores basados en el Pentium, los valores máximos para esos tres parámetros suelen ser (65535, 16 y 63), debido a la necesidad de mantener la compatibilidad hacia atrás con las limitaciones del IBM PC original. Sobre esa máquina se utilizaron campos de 16, 4 y 6 bits respectivamente para especificar dichos números, con cilindros y sectores numerados a partir de 1 y cabezas numeradas a partir de 0. Con estos parámetros y 512 bytes por sector, el disco más grande posible tiene una capacidad de 31,5 GB. Para poder superar ese límite, muchos discos modernos soportan ahora un sistema denominado **direcciónamiento de bloque lógico** (LBA), en el cual los sectores del disco se numeran de forma consecutiva a partir de 0, sin tener en cuenta para nada la geometría física del disco.

RAID

El rendimiento de las CPUs ha estado aumentando de manera exponencial durante la última década, duplicándose prácticamente cada 18 meses. No ha sucedido lo mismo con el rendimiento de los discos. En los años setenta, los tiempos de posicionamiento de las cabezas eran de 50 a 100 ms. Ahora dichos tiempos son de poco menos de 10 ms. En la mayoría de las industrias tecnológicas (como la automovilística o la de la aviación), una mejora en el rendimiento en un factor de 5 a 10 en dos décadas sería digno de celebrar, pero en la industria de los ordenadores resulta poco menos que vergonzoso. Así, la brecha entre el rendimiento de la CPU y el de los discos se ha ido agrandando con el paso del tiempo.

Como hemos visto, cada vez se está usando más el procesamiento paralelo para acelerar el rendimiento de la CPU. Desde hace años, a muchas personas se les ha ocurrido que la E/S en paralelo también podría ser una buena idea. En su artículo de 1988, Patterson y otros sugirieron seis organizaciones específicas de disco que podrían servir para mejorar el rendimiento de los discos, su fiabilidad o ambas cosas (Patterson y otros, 1988). Estas ideas no tardaron en ser adoptadas por la industria y han dado pie a una nueva clase de dispositivo de E/S denominado **RAID**. Patterson y otros definieron un RAID como un **array redundante de discos baratos** (*Redundant Array of Inexpensive Disks*), pero la industria redefinió la I de RAID de modo que significara “independientes” en lugar de “baratos” (¿tal vez para poder utilizar discos caros?). Puesto que también se necesitaba un villano (como en RISC contra CISC, que también se debe a Patterson), el malo de la película fue el **SLED** (*Single Large Expensive Disk*; un único disco grande y caro).

La idea básica que hay tras el RAID es la de instalar una caja llena de discos junto a un ordenador (típicamente un servidor grande), sustituir la tarjeta controladora de disco por una controladora RAID, copiar los datos al RAID y luego continuar con la operación normal. En otras palabras, un RAID debe verse igual que un SLED a los ojos del sistema operativo, pero tiene un mayor rendimiento y una mayor fiabilidad. Puesto que los discos SCSI tienen buen rendimiento, bajo precio y capacidad para que una única controladora opere hasta 7 unidades (15 en el caso del SCSI ancho), es natural que la mayoría de los RAID consistan de una tarjeta controladora RAID SCSI y una caja de discos SCSI que el sistema operativo ve como un único disco grande. De esta manera, no se requieren cambios en el software para utilizar el RAID, lo cual es un punto importante a tener en cuenta en el momento de la compra para muchos administradores de sistemas.

Los RAID se presentan al software como un único disco, pero además, todos los RAID tienen la propiedad de que los datos se distribuyen entre las unidades de disco para permitir realizar operaciones en paralelo. Patterson y otros definieron varios esquemas diferentes para hacer esto, los cuales se conocen ahora como RAID nivel 0 hasta RAID nivel 5. Existen además unos cuantos niveles más que no trataremos. El término “nivel” no es muy apropiado, ya que no se trata de una jerarquía; simplemente hay seis organizaciones diferentes posibles.

En la Figura 5-19(a) se ilustra el RAID nivel 0. En este caso el disco virtual simulado por el RAID se considera dividido en tiras (*strips*) de k sectores cada una, con los sectores de 0 a $k - 1$ en la tira 0, los sectores de k a $2k - 1$ en la tira 1, y así de forma sucesiva. Para $k = 1$, cada tira es un único sector; para $k = 2$, cada tira tiene dos sectores, etc. La organización RAID de nivel 0 escribe las tiras consecutivas repartiéndolas entre los discos por turno circular, como se muestra en la Figura 5-19(a) para un RAID con cuatro unidades de disco. Esta distribución de los datos entre varias unidades de disco se denomina **grabación en tiras** (*striping*). Por ejemplo, si el software envía un comando para leer un bloque de datos que consta de cuatro tiras consecutivas, comenzando al principio de una tira, el controlador RAID descompondrá este comando en cuatro comandos, uno para cada uno de los cuatro discos, haciéndolos operar en paralelo. Tenemos por tanto E/S paralela sin que el software sea consciente de ello.

El RAID de nivel 0 funciona mejor si las peticiones son grandes; cuanto más grandes, mejor. Si se pide un bloque mayor que el número de unidades de disco multiplicado por el tamaño de la tira, algunas unidades recibirán múltiples peticiones de sectores, por lo que cuando terminen de atender la primera petición, iniciarán la segunda. Corresponde al controlador descomponer la petición inicial y enviar los comandos adecuados a los discos adecuados en el orden correcto, ensamblando luego correctamente los resultados en la memoria. El rendimiento es excelente y la implementación es directa.

El RAID de nivel 0 funciona peor con sistemas operativos que habitualmente piden datos un sector de cada vez. Los resultados son correctos, pero no hay ningún paralelismo y, por lo tanto, tampoco ninguna mejora del rendimiento. Otra desventaja de esta organización es que la fiabilidad es potencialmente peor que la de un SLED. Si un RAID consiste en cuatro discos, cada uno con un tiempo medio de fallo de 20.000 horas, fallará alguna unidad aproximadamente una vez cada 5000 horas perdiéndose irremisiblemente los datos. Un SLED con un tiempo medio de fallo de 20.000 horas sería cuatro veces más fiable. Puesto que no hay redundancia en este diseño, en realidad no se trata de un verdadero RAID.

La siguiente opción, RAID de nivel 1, mostrada en la Figura 5-19(b), es ya un verdadero RAID. Todos los discos están duplicados, de modo que hay cuatro discos primarios y cuatro de respaldo (backup). Al escribir, todas las tiras se escriben dos veces. Al leer, puede utilizarse cualquiera de las copias, distribuyendo la carga entre más unidades de disco. Consecuentemente, el rendimiento de las escrituras no es mejor que con una sola unidad de disco, pero el rendimiento de las lecturas puede ser hasta dos veces mejor. La tolerancia a fallos es excelente: Si una unidad deja de funcionar, simplemente se utiliza la copia en su lugar. La recuperación consiste simplemente en instalar una nueva unidad de disco y copiar en ella la unidad de respaldo entera.

A diferencia de los niveles 0 y 1, que trabajan con tiras de sectores, el RAID de nivel 2 trabaja sobre una base de palabra, o incluso sobre una base de byte. Imaginemos que dividimos cada byte del disco virtual en un par de *nibbles* de 4 bytes cada uno, añadiendo un código de Hamming a cada nibble para formar una palabra de 7 bits, de la cual los bits 1, 2 y 4 son bits de paridad. Imaginemos también que las siete unidades de disco de la Figura 5-19(c) se sincronizan en términos de la posición de sus brazos y de su posición rotacional. Entonces sería posible escribir estas palabras Hamming de 7 bits repartiéndolas entre las siete unidades, un bit por unidad.

El ordenador CM-2 de Thinking Machines utilizaba este esquema, tomando palabras de datos de 32 bits y añadiéndoles 6 bits de paridad para formar una palabra Hamming de 38 bits, más un bit extra para la paridad de las palabras, repartiendo cada palabra entre 39 unidades de disco. La capacidad de transferencia de datos total era enorme, porque en el tiempo normal de escritura de un sector podían escribirse 32 sectores de datos. Además, la pérdida de una unidad no provoca ningún problema grave, porque equivale a perder un bit de cada palabra de 39 bits leída, algo que el código Hamming puede resolver sobre la marcha.

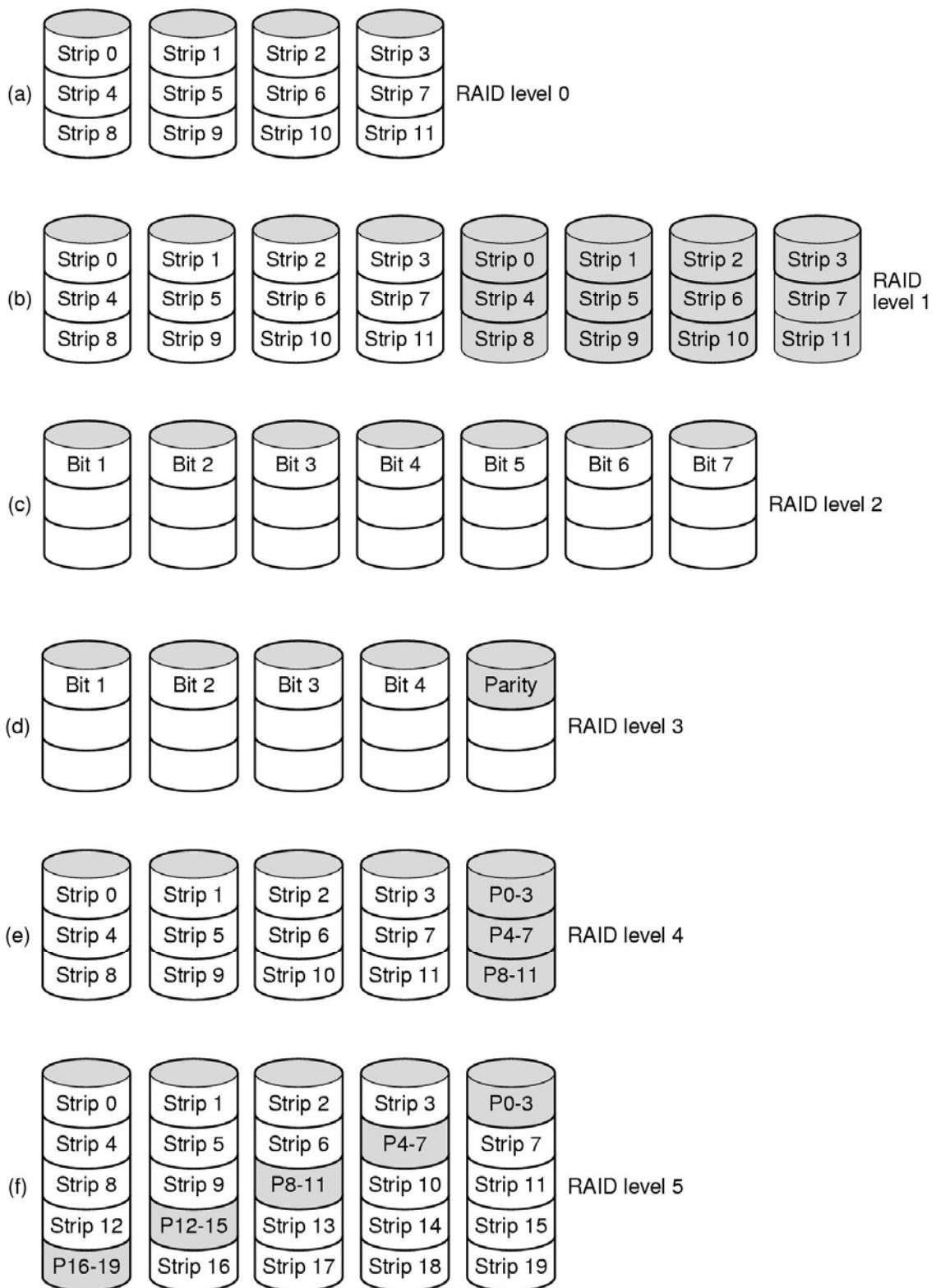


Figura 5-19. Niveles RAID 0 a 5. Las unidades de respaldo y paridad se muestran sombreadas.

En el lado de los inconvenientes, este esquema requiere que todas las unidades de disco estén sincronizadas rotacionalmente, y sólo tiene sentido si el número de unidades es considerable (incluso con 32 unidades de datos y seis unidades de paridad, la sobrecarga adicional es del 19%). También exige mucho de la controladora, que debe calcular una suma de verificación Hamming en el tiempo de lectura normal de cada bit.

El RAID de nivel 3 es una versión simplificada del RAID de nivel 2, y se ilustra en la Figura 5-19(d). Aquí se calcula un único bit de paridad por cada palabra de datos y se escribe en un disco de paridad. Al igual que en RAID nivel 2, las unidades deben estar perfectamente sincronizadas, porque las palabras de datos individuales están repartidas sobre varios discos.

A primera vista, podría parecer que utilizar un único bit de paridad sólo ofrece detección de errores, sin posibilidad de corregirlos. En el caso de errores aleatorios no detectados, esta observación es cierta. Sin embargo, en el caso de que deje de funcionar una unidad de disco, sí que es posible corregir completamente los errores de un bit correspondientes al conocerse la posición del bit erróneo. Si una unidad de disco deja de funcionar, el controlador simplemente hace como si todos sus bits fueran ceros. Si una palabra tiene un error de paridad, eso quiere decir que el bit de la unidad inoperante era realmente un 1, por lo que se corrige. Aunque los niveles RAID 2 y 3, ofrecen tasas de transferencia de datos muy altas, el número de peticiones individuales de E/S que pueden manejar por segundo no es mejor que con una única unidad de disco.

Los RAIDs de nivel 4 y 5 trabajan de nuevo con tiras, no con palabras individuales con paridad, y no requieren unidades de disco sincronizadas. RAID nivel 4 [vea la Figura 5-19(e)] se parece a RAID nivel 0 pero con una paridad tira a tira escrita en una unidad de disco adicional. Por ejemplo, si cada tira tiene una longitud de k bytes, se calcula el OR EXCLUSIVO de todas las tiras, y el resultado es una tira de paridad de k bytes. Si una unidad de disco deja de funcionar, los bytes perdidos pueden recalcularse a partir de la unidad de paridad.

Este diseño protege contra la pérdida de una unidad de disco, pero tiene un rendimiento muy pobre cuando lo que se realizan son pequeñas actualizaciones. Si se modifica un sector, es necesario leer todas las unidades para poder recalcular la paridad, que tendrá que volver a escribirse. Alternativamente, pueden leerse los datos de usuario antiguos y los datos de paridad viejos y calcular una nueva paridad a partir de ellos. Incluso con esta optimización, una pequeña actualización requiere dos lecturas y dos escrituras.

Como consecuencia de la pesada carga que asume la unidad de paridad, dicha unidad puede convertirse en un verdadero cuello de botella. Este cuello de botella se elimina en el RAID nivel 5 distribuyendo los bits de paridad de manera uniforme entre todas las unidades, por turno circular, como se muestra en la Figura 5-19(f). Sin embargo, en el caso de que un disco deje de funcionar, la reconstrucción de sus contenidos es un proceso complejo.

CD-ROMs

Recientemente han comenzado a estar disponibles los discos ópticos (en oposición a los magnéticos), teniendo densidades de grabación mucho más altas que los discos magnéticos convencionales. Los discos ópticos se desarrollaron originalmente para grabar programas de televisión, pero puede dárseles un uso más estético como dispositivos de almacenamiento para ordenadores. Debido a su capacidad, potencialmente enorme, los discos ópticos han sido el tema de un gran número de investigaciones y han experimentado una evolución increíblemente rápida.

Los discos ópticos de primera generación fueron inventados por el conglomerado de electrónica Philips, de los Países Bajos, para almacenar películas en ellos. Su diámetro era de 30 cm y se vendían con el nombre LaserVision, pero no tuvieron éxito, salvo en Japón.

En 1980, Philips, junto con Sony, desarrolló el CD (*Compact Disc*) que rápidamente sustituyó al disco de vinilo de 33 1/3 rpm para grabar música (excepto entre los conociédores, quienes todavía prefieren el vinilo). Los detalles técnicos precisos del CD se publicaron como un estándar internacional oficial (ISO 10149), conocido popularmente como el **Libro Rojo**, por el color de su portada. Los estándares internacionales son emitidos por la organización internacional para la estandarización, que es la contrapartida internacional de los organismos nacionales de normas como ANSI, DIN, etc. Cada estándar tiene su propio número ISO. Lo que se busca al publicar las especificaciones del disco y de la unidad como un estándar internacional es hacer posible la compatibilidad entre los CDs de diferentes productores de música y los reproductores de diferentes fabricantes de aparatos electrónicos. Todos los CDs tienen un diámetro de 120 mm y un espesor de 1,2 mm, con un agujero de 15 mm en el centro. El CD de audio fue el primer medio de almacenamiento digital que tuvo éxito en el mercado de masas. Se supone que duran 100 años. Estamos esperando que alguien compruebe esto en el año 2080 y nos informe de cómo le ha ido al primer lote de CDs.

Un CD se prepara utilizando un láser infrarrojo de alta potencia para quemar agujeros de 0,8 micras de diámetro en un disco maestro recubierto de vidrio. A partir de ese disco maestro, se fabrica un molde, con pequeñas protuberancias donde estaban los agujeros realizados con el láser. En este molde se inyecta resina de policarbonato fundida que toma la forma de un CD con el mismo patrón de agujeros que el disco maestro de vidrio. Luego se deposita sobre el policarbonato una capa muy fina de aluminio reflectante que se cubre con una laca protectora y finalmente una etiqueta. Las depresiones en el sustrato de policarbonato se denominan **fosos** (*pits*); las áreas no quemadas entre los fosos se denominan **llanos** (*lands*).

Cuando se lee un CD en un reproductor, un diodo láser de baja potencia ilumina los fosos y llanos a medida que el disco gira bajo el láser, utilizando una luz infrarroja con una longitud de onda de 0,78 micras. La luz del láser incide por el lado del policarbonato, de modo que los fosos sobresalen hacia el láser como protuberancias en la superficie por lo demás plana. Puesto que los fosos tienen una altura igual a una cuarta parte de la longitud de onda de la luz del láser, la luz que se refleja de un foso está desfasada media longitud de onda respecto a la que refleja la superficie circundante. Como resultado las dos partes se interfieren de forma destructiva y devuelven menos luz al fotodetector del reproductor que la luz que se refleja de un llano. Así es como el reproductor distingue un foso de un llano. Aunque podría parecer más sencillo utilizar un foso para registrar un 0 y un llano para registrar un 1 (o viceversa), resulta más fiable utilizar una transición foso/llano o llano/foso para un 1 y su ausencia para un 0, por lo que éste es el esquema que se usa.

Los fosos y llanos se graban en una única espiral continua que comienza cerca del agujero y avanza una distancia de 32 mm hacia el borde. La espiral describe 22.188 revoluciones alrededor del disco (aproximadamente 600 por mm). Si se desenrollara, tendría una longitud de 5,6 km. La espiral se ilustra en la Figura 5-20.

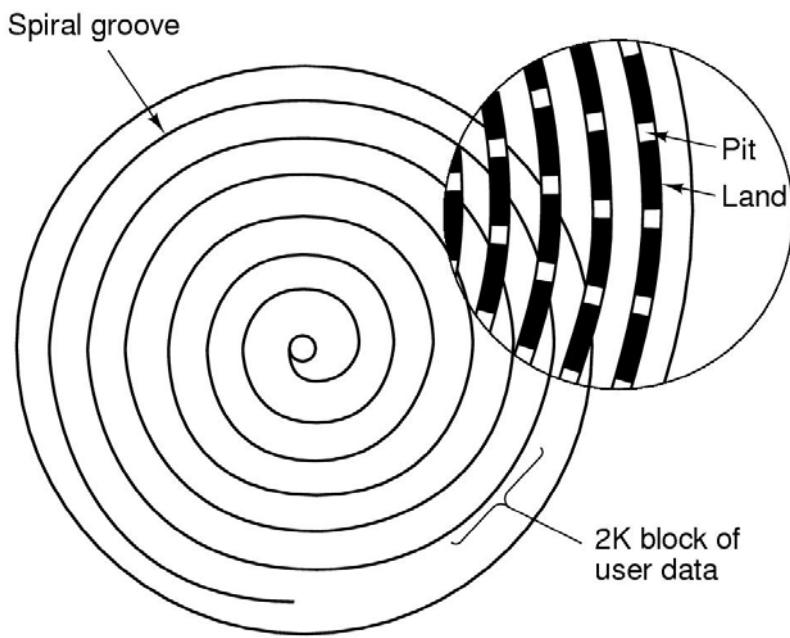


Figura 5-20. Estructura de grabación de un disco compacto o CD-ROM.

Para conseguir que la música se reproduzca a un ritmo uniforme, es necesario que los fosos y llanos fluyan a una velocidad lineal constante. Consecuentemente la velocidad de rotación del CD debe reducirse continuamente a medida que la cabeza lectora se aleja del centro del CD hacia el exterior. En la parte más interna, la velocidad de rotación es de 530 rpm para conseguir una velocidad lineal de 120 cm/s; en la parte más externa la velocidad de rotación debe bajar a 200 rpm para mantener la misma velocidad lineal bajo la cabeza. Una unidad de disco con velocidad lineal constante es muy distinta de una unidad de disco magnético, que opera a una velocidad angular constante, independiente del lugar donde la cabeza esté actualmente posicionada. Además, 530 rpm no es comparable con las velocidades de 3600 a 7200 rpm a las que giran la mayoría de los discos magnéticos.

En 1984, Philips y Sony se percataron de la posibilidad de utilizar los discos compactos para almacenar datos destinados a un ordenador, por lo que publicaron el **Libro Amarillo**, que define un estándar preciso para lo que ahora se conoce como **CD-ROM** (*Compact Disk - Read Only Memory*). A fin de aprovechar el ya entonces considerable mercado de los CDs de audio, los CD-ROMs debían tener el mismo tamaño físico que los CDs de audio, ser mecánica y ópticamente compatibles con ellos, y producirse utilizando las mismas máquinas de estampado por inyección de policarbonato. La consecuencia de esta decisión no sólo fue la necesidad de contar con motores lentos de velocidad variable, sino también que el costo de fabricación de un CD-ROM enseguida se hizo mucho menor de un dólar en volúmenes de producción moderados.

Lo que el Libro Amarillo definió fue el formateo de los datos del ordenador. También se mejoró la capacidad de corrección de errores del sistema, lo que era un paso esencial porque, si bien a los amantes de la música no les importa perder algún que otro bit, los amantes de los ordenadores tienden a ser muy quisquillosos al respecto. El formato básico de un CD-ROM consiste en codificar cada byte con un símbolo de 14 bits. Como vimos antes, son suficientes 14 bits para codificar mediante Hamming un byte de 8 bits, sobrando dos bits. De hecho se emplea un sistema de codificación aún más potente. La transformación de 14 a 8 durante la lectura se realiza por hardware mediante la consulta de unas tablas.

En el siguiente nivel hacia arriba, un grupo de 42 símbolos consecutivos forma una **trama** de 588 bits. Cada trama contiene 192 bits de datos (24 bytes). Los 396 bits restantes se utilizan para corrección de errores y control. Hasta aquí el esquema es idéntico para los CDs de audio y los CD-ROMs.

Lo que el Libro Amarillo añade es el agrupamiento de 98 tramas en un **sector de CD-ROM**, como se muestra en la Figura 5-21. Todo sector de CD-ROM comienza con un preámbulo de 16 bytes, de los cuales los primeros 12 son 00FFFFFFFFFFFFFFFFFF00 (hexadecimal) para hacer posible que el reproductor reconozca el principio de un sector de CD-ROM. Los tres bytes siguientes contienen el número de sector, que se necesita porque el posicionamiento de la cabeza lectora en un CD-ROM con su espiral de datos única es mucho más difícil que un disco magnético con sus pistas concéntricas uniformes. Para posicionarse, el software de la unidad calcula aproximadamente el lugar al que debe ir, mueve la cabeza allí y luego espera hasta detectar un preámbulo que le dirá lo cerca o lejos que está del punto de destino. El último byte del preámbulo es el modo.

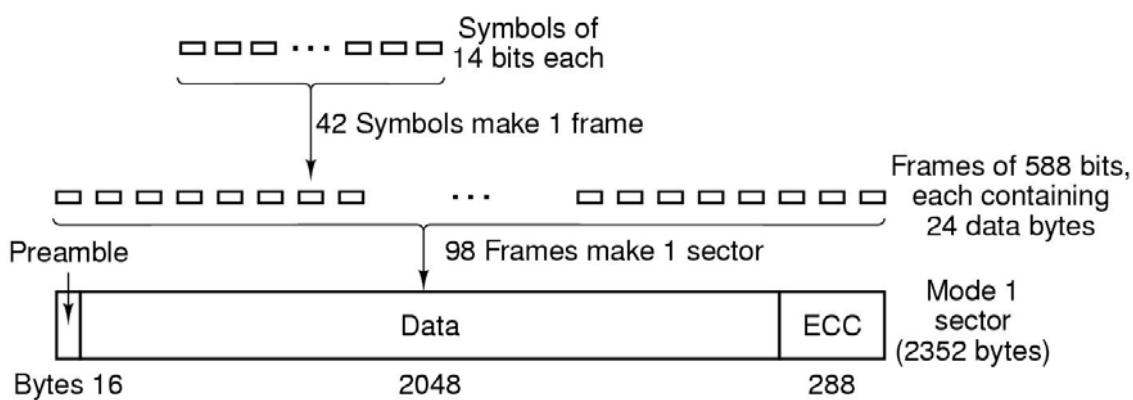


Figura 5-21. Disposición lógica de los datos en un CD-ROM.

El Libro Amarillo define dos modos. El modo 1 utiliza la disposición de la Figura 5-21, con un preámbulo de 16 bytes, 2048 bytes de datos y un código de 288 bytes para corrección de errores (un código Reed-Solomon intercalado). El modo 2 combina los campos de datos y de ECC en un campo de datos de 2336 bytes para aquellas aplicaciones que no necesitan corrección de errores (o no disponen de tiempo para realizarla), tales como audio y vídeo. Cabe señalar que para mantener una excelente fiabilidad, se utilizan tres esquemas separados de corrección de errores: dentro de un símbolo, dentro de una trama y dentro de un sector de CD-ROM. Los errores de un único bit se corrigen en el nivel más bajo; los errores de ráfaga cortos se corrigen en el nivel de trama y los errores residuales se capturan en el nivel de sector. El precio que se paga por esa fiabilidad es que se requieren 98 tramas de 588 bits (7203 bytes) para una carga útil de datos de sólo 2048 bytes; una eficiencia de tan solo el 28%.

Las unidades de CD-ROM de una sola velocidad (1x) operan a 75 sectores/s, lo que significa una tasa de datos de 153.600 bytes/s en el modo 1 y 175.200 bytes/s en el modo 2. Las unidades de doble velocidad (2x) son dos veces más rápidas, y así sucesivamente hasta la velocidad más alta. Una unidad de 40x puede entregar datos a razón de 40×153.600 bytes/s, suponiendo que tanto la interfaz de la unidad como el bus y el sistema operativo pueden manejar esa tasa de datos. Un CD de audio estándar tiene espacio para 74 minutos de música que, si se utiliza para datos en modo 1, da una capacidad de 681.984.000 bytes. Esta cifra suele expresarse como 650 MB ya que 1 MB equivale a 2^{20} bytes (1.048.576 bytes), no 1.000.000 bytes.

Una unidad de CD-ROM de 32x (4.915.200 bytes/s) no es rival para una unidad de disco magnético Fast SCSI-2, que opera a 10 MB/s. Aunque muchas unidades de CD-ROM utilizan la interfaz SCSI son muy frecuentes las unidades de CD-ROM IDE. Si pensamos que el tiempo de posicionamiento de la cabeza lectora suele ser de varios cientos de milisegundos, resulta obvio que las unidades de CD-ROM no están en la misma categoría de rendimiento que las de disco magnético, a pesar de su gran capacidad.

En 1986, Philips dio un nuevo golpe con el **Libro Verde**, añadiendo gráficos y capacidad para intercalar audio, vídeo y datos en el mismo sector, característica indispensable para los CD-ROMs multimedia.

La última pieza del rompecabezas de los CD-ROMs es el sistema de ficheros. Para poder utilizar el mismo CD-ROM en diferentes ordenadores, era preciso llegar a un acuerdo en lo tocante a los sistemas de ficheros en CD-ROM. A fin de lograr este acuerdo, los representantes de muchas compañías de ordenadores se reunieron en Lake Tahoe en lo alto de High Sierras, en la frontera entre los estados de California y Nevada, e idearon un sistema de ficheros al que denominaron **High Sierra** y que más adelante se convirtió en un nuevo estándar internacional (ISO 9660). Hay tres niveles. El nivel 1 utiliza nombres de fichero de hasta 8 caracteres seguidos opcionalmente por una extensión de hasta tres caracteres (el convenio de MS-DOS para nombrar ficheros). Los nombres de fichero sólo pueden contener letras mayúsculas, dígitos y el trazo de subrayado. Los directorios pueden anidarse hasta una profundidad de ocho, pero los nombres de directorio no pueden contener extensiones. El nivel 1 exige que todos los ficheros sean contiguos, lo que no resulta un problema en un medio que se graba una sola vez. Cualquier CD-ROM que se ajuste a ISO 9660 nivel 1 puede leerse utilizando MS-DOS, un ordenador Apple, un ordenador UNIX o casi cualquier otro ordenador. Los productores de CD-ROMs ven este hecho como una gran ventaja.

ISO 9660 nivel 2 permite nombres de hasta 32 caracteres, y el nivel 3 permite ficheros no contiguos. Las extensiones Rock Ridge (denominadas así caprichosamente por el pueblo que figura en la película de Gene Wilder, *Blazing Saddles*) permiten nombres muy largos (para UNIX), UIDs, GIDs y enlaces simbólicos, pero los CD-ROMs que no se ajustan al nivel 1 no pueden leerse en todos los ordenadores.

Los CD-ROMs se han vuelto extremadamente populares para publicar juegos, películas, enciclopedias, atlas y trabajos de referencia de todo tipo. En la actualidad, casi todo el software comercial se vende en CD-ROM. Su combinación de alta capacidad y bajo coste de fabricación los hace apropiados para innumerables aplicaciones.

CDs grabables

Inicialmente, el equipamiento necesario para producir un CD-ROM maestro (o un CD de audio) era extremadamente costoso. Pero como suele suceder en la industria de los ordenadores, nada permanece caro durante mucho tiempo. A mediados de los años noventa, las grabadoras de CD con un tamaño no mayor que un reproductor de CD eran periféricos comunes disponibles en la mayoría de las tiendas de ordenadores. Estos dispositivos seguían siendo diferentes de los discos magnéticos ya que una vez grabados, los CD-ROMs no podían borrarse. Sin embargo, rápidamente encontraron un nicho propio como medio de backup para grandes discos duros y también permitiendo a individuos o a empresas nacientes fabricar sus propios CD-ROMs en lotes pequeños o crear discos maestros para entregarlos a plantas comerciales de duplicación de CDs a gran escala. Estas unidades se conocen como CD-R (*CD-Recordable*).

Físicamente, los CD-Rs comienzan siendo discos vírgenes de policarbonato de 120 mm parecidos a los CD-ROMs, excepto en que contienen un surco de 0,6 mm de anchura para guiar el láser durante la escritura. El surco tiene una excusión sinusoidal de 0,3 mm a una frecuencia de exactamente 22,05 kHz para proporcionar una retroalimentación continua de forma que la

velocidad de rotación pueda ser precisamente monitorizada y ajustada si es necesario. Los CD-Rs tienen una apariencia similar a los CD-ROMs, excepto que son de color dorado en la parte de arriba, en lugar de ser plateados. El color dorado se debe a la utilización de oro en lugar de aluminio en la capa reflectante. A diferencia de los CDs plateados, que tienen depresiones físicas, en los CD-Rs la diferencia de reflectividad de los fosos y llanos debe simularse. Esto se consigue añadiendo una capa de colorante entre el policarbonato y la capa de oro reflectante, como se muestra en la Figura 5-22. Se utilizan dos tipos de colorante: cianina, que es verde, y ftalocianina, que es de color naranja amarillento. Los químicos pueden enfascarse en argumentaciones interminables acerca de cuál es mejor. Estos colorantes son similares a los empleados en fotografía, lo cual explica por qué Kodak y Fuji son fabricantes destacados de discos CD-R en blanco.

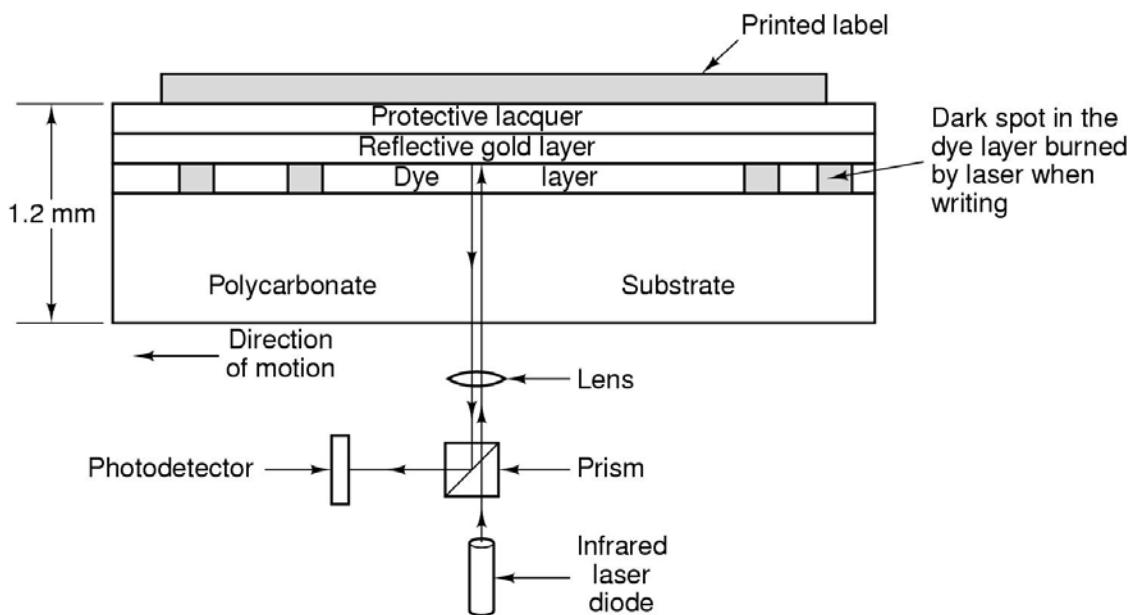


Figura 5-22. Sección de un disco CD-R y su láser (no está a escala). Un CD-ROM plateado tiene una estructura similar, sin la capa de colorante y con una capa agujereada de aluminio en lugar de la capa de oro.

En su estado inicial, la capa de colorante es transparente y permite que la luz láser pase y se refleje en la capa de oro. Para escribir, la potencia del láser del CD-R se aumenta a 8-16 mW. Cuando el haz incide sobre un punto del colorante, lo calienta y rompe un enlace químico. Este cambio en la estructura molecular crea una mancha oscura. Cuando se lee (a 0,5 mW), el fotodetector nota la diferencia entre las manchas oscuras donde se calentó el colorante y las áreas transparentes donde está intacto. Esta diferencia se interpreta como la diferencia entre fosos y llanos, incluso cuando se lee con un lector de CD-ROM normal o incluso con un reproductor de CDs de audio.

Ninguna nueva especie de CDs podría enorgullecerse de su origen sin un libro de color, por lo que al CD-R le corresponde el **Libro Naranja**, publicado en 1989. Este documento define el CD-R y también un formato nuevo, el **CD-ROM XA**, que permite escribir los CD-Rs de forma incremental, unos cuantos sectores hoy, unos cuantos mañana y otros pocos el mes que viene. Un grupo de sectores consecutivos escritos de una vez se denomina una **pista de CD-ROM**.

Uno de los primeros usos del CD-R fue para el Kodak PhotoCD. En este sistema, el cliente lleva un rollo de película fotográfica expuesta y su viejo PhotoCD al procesador fotográfico, recibiendo más tarde el mismo PhotoCD con las nuevas fotografías añadidas después de las antiguas. El nuevo lote de fotografías, que se crea digitalizando los negativos, se

escribe en el PhotoCD como una pista de CD-ROM separada. La escritura incremental era necesaria porque, cuando se introdujo ese producto, los discos CD-R vírgenes eran demasiado caros como para utilizar uno nuevo para cada rollo de película fotográfica.

Sin embargo la escritura incremental crea un nuevo problema. Antes del Libro Naranja, todos los CD-ROMs tenían una única **tabla de contenidos del volumen (VTOC; Volume Table Of Contents)** al principio. Ese esquema no funciona con las escrituras incrementales (es decir, multipista). La solución ofrecida por el Libro Naranja fue dar a cada pista de CD-ROM su propia VTOC. Los ficheros listados en la VTOC pueden incluir algunos de los ficheros de pistas anteriores, o todos. Cuando se inserta el CD-R en la unidad, el sistema operativo busca en todas las pistas del CD-ROM hasta encontrar la VTOC más reciente, que muestra el estado actual del disco. Al incluirse en la VTOC algunos de los ficheros de pistas anteriores, pero no todos, es posible crear la ilusión de que se han borrado algunos ficheros. Las pistas pueden agruparse en **sesiones**, dando pie a CD-ROMs **multisesión**. Los reproductores de CD de audio estándar no pueden reproducir CDs multisesión porque esperan una única VTOC al principio.

Cada pista tiene que escribirse de una vez en una única operación continua sin paradas. Como consecuencia, el disco duro del que provienen los datos debe ser lo suficientemente rápido como para suministrarlos a tiempo. Si los ficheros que van a copiarse están dispersos por todo el disco, los tiempos de posicionamiento del brazo pueden provocar que cese temporalmente el flujo de datos al CD-R ocasionando el vaciado del búfer (*buffer underrun*). El resultado del vaciado del búfer en medio de la grabación de la pista es un bonito y brillante (pero bastante caro) posavasos, o un disco volador dorado de 120 mm. El software del CD-R normalmente ofrece la opción de juntar todos los ficheros a escribir en una única imagen de CD-ROM contigua, de 650 MB, antes de grabar el CD-R, pero ese proceso normalmente duplica el tiempo de escritura real, requiere 650 MB de espacio libre en el disco, y tampoco protege contra los discos duros que, habiéndose calentado demasiado, empiezan a fallar hasta el punto de que el pánico les obliga a realizar una laboriosa recalibración térmica.

Los CD-Rs hacen posible que las personas individuales y las pequeñas compañías puedan copiar fácilmente CD-ROMs (y CDs de audio), generalmente violando los derechos de autor de quien los produjo. Se han ideado varios esquemas para obstaculizar tal piratería e impedir que un CD-ROM pueda leerse utilizando cualquier software que no sea del productor. Uno de ellos consiste en grabar todas las longitudes de los ficheros en el CD-ROM como siendo de varios gigabytes, frustrando cualquier intento de copiar los ficheros en un disco duro utilizando el software de copia estándar. Las longitudes verdaderas están implícitas en el software del productor u ocultas (posiblemente cifradas) en el CD-ROM en un lugar inesperado. Otro esquema utiliza ECCs intencionalmente incorrectos en sectores seleccionados, con la expectativa de que el software de copia de CDs “corrija” los errores. El software de la aplicación verifica los ECCs negándose a trabajar si los encuentra corregidos. Otras posibilidades son el uso de huecos no estándar entre las pistas y otros “defectos” físicos.

CDs regrabables

Aunque la gente está acostumbrada a otros medios en los que sólo puede escribirse una vez como el papel y la película fotográfica, existe demanda de un CD-ROM reescribible. Una tecnología que ya está disponible es el **CD-RW (CD-ReWritable)**, que utiliza discos del mismo tamaño que los CD-Rs. Sin embargo, en lugar de un colorante como cianina o ftalocianina, los CD-RW utilizan una aleación de plata, indio, antimonio y teluro para la capa de grabación. Esta aleación tiene dos estados estables: cristalino y amorfo, con diferentes reflectividades.

Las unidades de CD-RW utilizan láseres con tres niveles de potencia. En la potencia más alta, el láser funde la aleación, convirtiéndola del estado cristalino, altamente reflectante, al estado amorfo, de baja reflectividad, para simular un foso. En la potencia media, la aleación se

funde y vuelve a solidificarse en su estado cristalino natural, para convertirse otra vez en un llano. En la potencia más baja el estado del material se detecta (para su lectura) pero no tiene lugar ninguna transición de fase.

La razón por la que los CD-RWs no han sustituido a los CD-Rs es que los discos CD-RW vírgenes son mucho más caros que los CD-Rs. Además, para las aplicaciones de backup de los discos duros resulta una gran ventaja el hecho de que, una vez grabado, un CD-R no pueda borrarse accidentalmente.

DVD

El formato de CD/CD-ROM básico se ha estado utilizando desde alrededor de 1980. La tecnología ha mejorado desde entonces, por lo que ahora son económicamente factibles discos ópticos de mayor capacidad, existiendo una gran demanda de ellos. A Hollywood le encantaría sustituir las cintas de vídeo analógico por discos digitales, ya que ofrecen una calidad superior, su fabricación es más económica, duran más, ocupan menos espacio en los anaquellos de las tiendas de vídeo y no tienen que rebobinarse. Las compañías de electrónica de consumo estaban buscando un nuevo producto que arrasase, y muchas compañías de ordenadores querían añadir características multimedia a su software.

Esta combinación de tecnología y demanda por parte de tres industrias inmensamente ricas y poderosas ha dado origen al **DVD**, que originalmente era el acrónimo de **disco de vídeo digital** pero que ahora oficialmente es **Disco Digital Versátil**. Los CDs utilizan el mismo diseño general que los CDs, con discos de policarbonato de 120 mm moldeados por inyección que contienen fosos y llanos, se iluminan con un diodo láser y se leen con un fotodetector. Lo nuevo es el uso de:

1. Fosos más pequeños (de 0,4 micras en lugar de 0,8 micras en los CDs).
2. Una espiral más apretada (0,74 micras entre pistas frente a las 1,6 de los CDs).
3. Un láser rojo (a 0,65 micras en lugar de 0,78 micras en los CDs).

En conjunto, estas mejoras aumentan la capacidad siete veces, hasta los 4,7 GB. Una unidad DVD a 1x opera a 1,4 MB/s (compárese con los 150 KB/s de los CDs). Desafortunadamente el cambio a los láseres rojos empleados en los supermercados implica que los reproductores de DVD requieren un segundo láser o un sistema óptico de conversión complicado para poder leer los CDs y CD-ROMs existentes, algo que quizás no todas las unidades puedan incluir. Además, algunas unidades de DVD no pueden leer CD-Rs o CD-RWs.

¿Es suficiente con 4,7 GB? Quizás. Utilizando compresión MPEG-2 (estandarizada en el ISO 13346), un disco DVD de 4,7 GB puede contener 133 minutos de vídeo en pantalla completa y con alta definición (720×480), así como pistas sonoras en hasta ocho idiomas y subtítulos en 32 idiomas más. Cerca del 92% de todas las películas que se han hecho en Hollywood duran menos de 133 minutos. No obstante, algunas aplicaciones como juegos multimedia o trabajos de referencia podrían necesitar más, y a Hollywood le gustaría poder poner varias películas en el mismo disco, por lo que se han definido cuatro formatos:

1. Un solo lado, una sola capa (4,7 GB).
2. Un solo lado, doble capa (8,5 GB).
3. Dos lados, una sola capa (9,4 GB).
4. Dos lados, doble capa (17 GB).

¿Por qué tantos formatos? En una sola palabra: política. Philips y Sony querían discos de un solo lado y doble capa para la versión de alta capacidad, pero Toshiba y Time Warner querían discos de dos lados y una sola capa. Philips y Sony no creían que la gente estaría dispuesta a dar la vuelta a los discos, y Time Warner no creía factible eso de colocar dos capas de un mismo lado. El compromiso: todas las combinaciones, pero el mercado determinará cuáles sobrevivirán.

La tecnología de capa dual tiene una capa reflectante en el fondo, y más arriba una capa semirreflectante. Dependiendo del lugar donde se enfoque el láser, el haz rebotará en una capa o en la otra. La capa inferior necesita fosos y llanos un poco más grandes para que pueda leerse de forma fiable, por lo que su capacidad es un poco más baja que la de la capa superior.

Los discos de dos lados se fabrican tomando dos discos de un solo lado, de 0,6 mm de espesor, y pegándolos reverso con reverso. Para que el espesor de todas las versiones sea el mismo, un disco de un solo lado consta de un disco de 0,6 mm pegado a un sustrato en blanco (o tal vez en el futuro uno que tenga grabados 133 minutos de anuncios con la esperanza de que la gente sienta curiosidad por ver qué es lo que hay ahí abajo). La estructura del disco de dos lados y doble capa se ilustra en la Figura 5-23.

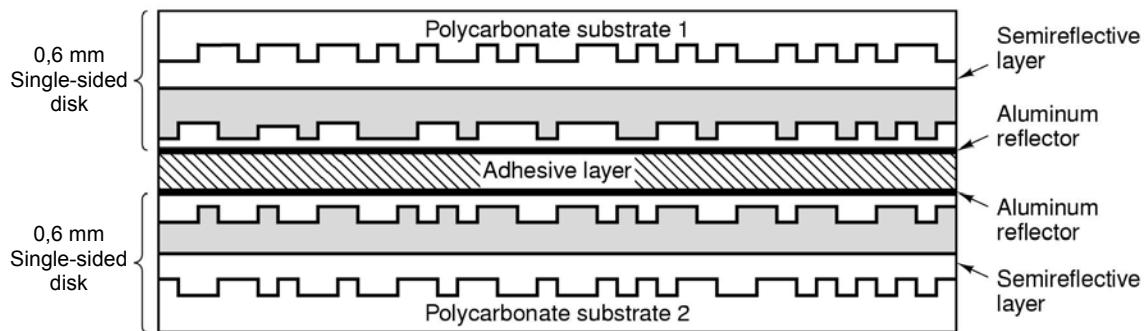


Figura 5-23. Disco DVD de dos lados y doble capa.

El DVD fue ideado por un consorcio de 10 compañías de electrónica de consumo, siete de ellas japonesas, en estrecha cooperación con los principales estudios hollywoodienses (algunos de los cuales son propiedad de las compañías japonesas de electrónica del consorcio). Las industrias de los ordenadores y de telecomunicaciones no fueron invitadas a la fiesta, y el enfoque resultante fue el uso del DVD para el alquiler de películas y para presentaciones de productos. Por ejemplo, entre las funciones estándar está la de saltarse en tiempo real las escenas escabrosas (de modo que los padres puedan convertir una película clasificada para adultos en una que los niños pequeños puedan ver sin peligro), sonido de seis canales y manejo de *Pan-and-Scan*. Esta última función permite al reproductor de DVD decidir de forma dinámica cómo recortar los bordes izquierdo y derecho de las películas (cuya proporción de aspecto es de 4:3).

Otra cosa que quizás no se le habría ocurrido a la industria de los ordenadores es una premeditada incompatibilidad entre los discos destinados a Estados Unidos y los destinados a Europa, además de estándares para otros continentes. Hollywood exigió esta “función” porque las películas siempre se exhiben primero en Estados Unidos y luego se envían a Europa una vez que los videos salen a la venta en Estados Unidos. Lo que se buscaba era garantizar que las tiendas de video europeas no pudieran comprar videos en Estados Unidos demasiado pronto, pues de lo contrario, se reducirían las entradas de taquilla por la exhibición de películas nuevas en las salas de cine europeas. Si Hollywood controlara el destino de la industria de los ordenadores, habríamos tenido disquetes de 3,5 pulgadas en Estados Unidos y de 9 cm en Europa.

Formateo del disco

Un disco duro consiste en una pila de platos de aluminio, aleación o vidrio de 5,25 o 3,5 pulgadas de diámetro (o incluso más pequeños en los ordenadores portátiles). En cada plato está depositada una delgada capa de óxido metálico magnetizable. Después de la fabricación, no hay ninguna información en ninguna parte del disco.

Para que el disco pueda utilizarse es necesario que cada plato reciba un **formato de bajo nivel** realizado por software. El formato consiste de una serie de pistas concéntricas, cada una de las cuales contiene cierto número de sectores, con cortos espacios vacíos entre ellos. En la Figura 5-24 se muestra el formato de un sector.

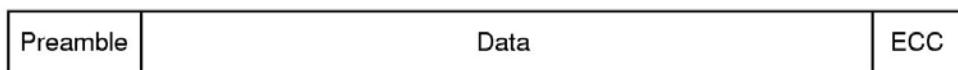


Figura 5-24. Un sector de disco.

El preámbulo comienza con cierto patrón de bits que permite al hardware reconocer el principio del sector. También contiene los números de cilindro y de sector y alguna otra información. El programa de formateo de bajo nivel determina el tamaño de la porción de datos. La mayoría de los discos utilizan sectores de 512 bytes. El campo ECC contiene información redundante que puede servir para corregir errores de lectura. El tamaño y el contenido de este campo varía de fabricante a fabricante, dependiendo de cuanto espacio de disco está dispuesto a sacrificar el diseñador a cambio de una mayor fiabilidad, y del grado de complejidad del código ECC que puede manejar la controladora. Un campo ECC de 16 bytes no es inusual. Además, todos los discos duros tienen asignados un cierto número de sectores de reserva que sirven para sustituir los sectores con defectos de fabricación que puedan aparecer.

Cuando se aplica el formato de bajo nivel el sector 0 de cada pista se desplaza respecto del sector 0 de la pista anterior. Este desplazamiento, denominado **sesgo del cilindro** (*cylinder skew*), tiene por objeto mejorar el rendimiento. La idea es permitir que el disco lea varias pistas en una sola operación continua sin perder datos. La naturaleza del problema puede apreciarse observando la Figura 5-18(a). Supongamos que una petición necesita que se lean 18 sectores comenzando por el sector 0 de la pista más interna. La lectura de los primeros 16 sectores tarda una rotación del disco, pero a continuación es necesario mover el brazo una pista hacia fuera para leer el sector 17. Para cuando la cabeza termina de moverse una pista, el sector 0 ya ha pasado de largo, de modo que se necesita esperar toda una rotación para leer ese sector. El problema se elimina desplazando los sectores como se muestra en la Figura 5-25.

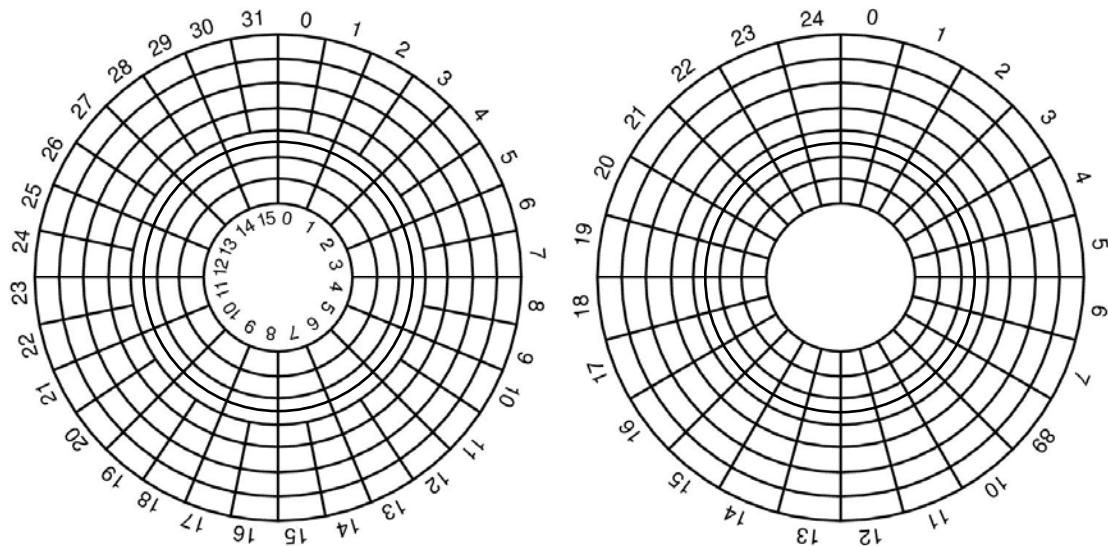


Figura 5-18. (a) Geometría física de un disco con dos zonas.
 (b) Una posible geometría virtual para este disco.

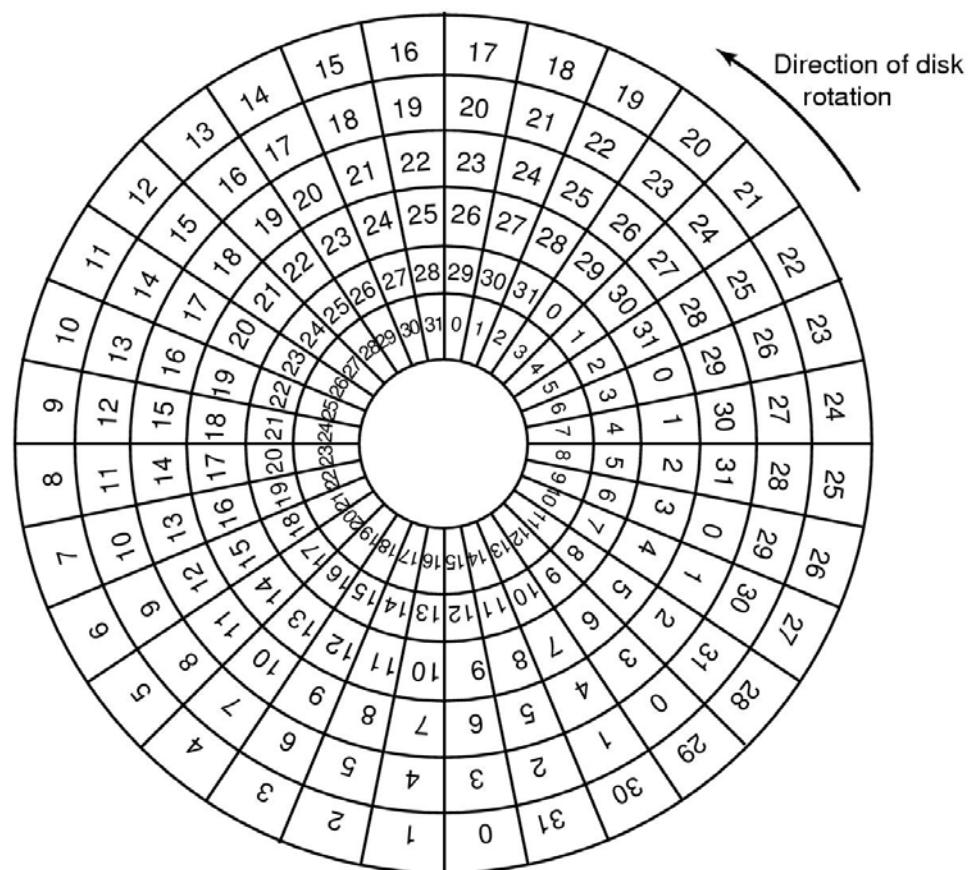


Figura 5-25. Una ilustración del sesgo del cilindro.

La cantidad de sesgo del cilindro depende de la geometría de la unidad. Por ejemplo, una unidad de 10.000 rpm realiza una rotación completa cada 6 milisegundos. Si una pista contiene 300 sectores, pasará un sector bajo la cabeza cada 20 microsegundos. Si el tiempo de posicionamiento desde una pista a la siguiente es de 800 microsegundos, eso significa que pasarán 40 sectores durante el posicionamiento, por lo que el sesgo del cilindro debe ser de 40 sectores, en vez de los tres sectores que se muestran en la Figura 5-25. Vale la pena mencionar que la comutación entre cabezas tarda también un tiempo finito, de manera que hay un **sesgo de la cabeza** además de un sesgo del cilindro, pero el sesgo de la cabeza no es muy grande.

Como resultado del formateo de bajo nivel, la capacidad del disco se reduce, dependiendo de los tamaños del preámbulo, del espacio vacío entre sectores y del ECC, así como del número de sectores reservados para sustituir los sectores defectuosos. A menudo la capacidad tras ese formateo es un 20% menor que la capacidad inicial. Los sectores de reserva no cuentan de cara a la capacidad tras el formateo, por lo que todos los discos de un tipo dado tienen exactamente la misma capacidad cuando salen de la fábrica, independientemente de cuántos sectores defectuosos tengan en realidad (si el número de sectores defectuosos excede el número de sectores de reserva, la unidad se rechaza y no sale de la fábrica).

Existe una considerable confusión en lo que respecta a la capacidad de los discos, porque algunos fabricantes anuncian la capacidad no formateada para aparentar que sus discos son más grandes de lo que en realidad lo son. Por ejemplo, consideremos una unidad de disco cuya capacidad no formateada es de 20×10^9 bytes. Esta unidad podría venderse como un disco de 20 GB. Sin embargo, tras el formateo, quizás sólo queden $2^{34} \cong 17,2 \times 10^9$ bytes disponibles para guardar datos. Por si no hubiera poca confusión, es probable que el sistema operativo informe de esa capacidad como 16,0 GB, no 17,2 GB, porque el software considera que 1 GB es 2^{30} (1.073.741.824) bytes, y no 10^9 (1.000.000.000) bytes.

Para empeorar todavía más las cosas, en el mundo de la comunicación de datos 1 Gbps significa 1.000.000.000 bits/segundo porque el prefijo *giga* realmente significa 10^9 (después de todo, un kilómetro son 1000 metros, no 1024 metros). Sólo al hablar de tamaños de memorias y de discos los prefijos *kilo*, *mega*, *giga* y *tera* significan 2^{10} , 2^{20} , 2^{30} y 2^{40} , respectivamente.

El formateo afecta también al rendimiento. Si un disco de 10.000 rpm tiene 300 sectores de 512 bytes por pista, tardará 6 milisegundos en leer los 153.600 bytes de una pista, lo que significa una velocidad de transferencia de datos de 25.600.000 bytes/segundo, o 24,4 MB/s. Será imposible alcanzar una velocidad mayor, sea cual sea el tipo de interfaz presente, incluso aunque sea una interfaz SCSI que opere a 80 MB/segundo o a 160 MB/segundo.

En realidad, para poder leer de forma continua a esas velocidades se requiere un gran búfer en el controlador. Por ejemplo, consideremos un controlador cuyo búfer tiene capacidad para un sector y al que se ha enviado un comando para leer dos sectores consecutivos. Después de leer el primer sector del disco y realizar el cálculo de su ECC, hay que transferir los datos a la memoria principal. Mientras tiene lugar esa transferencia, el siguiente sector está pasando velozmente bajo la cabeza. Cuando se completa la copia a la memoria, el controlador tiene que esperar casi el tiempo de una rotación entera hasta que el segundo sector vuelva a pasar debajo de la cabeza.

Este problema puede eliminarse numerando los sectores de una forma intercalada cuando se formatea el disco. En la Figura 5-26(a) vemos el patrón de numeración usual (ignorando aquí el sesgo del cilindro). En la Figura 5-26(b) vemos un **intercalamiento sencillo** (*single interleaving*), que da a la controladora un respiro entre sectores consecutivos para que le dé tiempo a copiar el búfer en la memoria principal.

Si el proceso de copiado es muy lento, podría ser necesario utilizar **doble intercalamiento** como en la Figura 5-26(c). Si el controlador tiene un búfer con capacidad para un solo sector, el intercalamiento es una técnica necesaria, independientemente de si la copia del búfer a la memoria principal la realiza el controlador, la CPU principal o un chip de DMA, ya que de todos modos la copia tardará algún tiempo. Para evitar la necesidad del intercalamiento de los sectores el controlador tendría que ser capaz de guardar en el búfer toda una pista. Muchos controladores modernos pueden hacerlo.

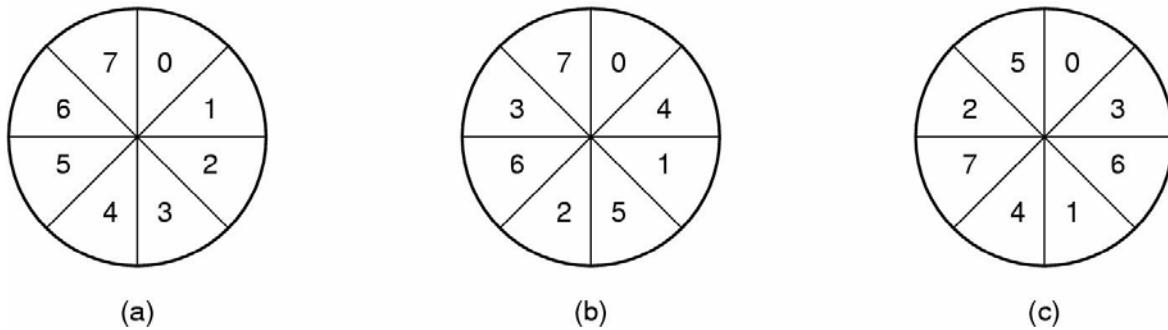


Figura 5-26. (a) Sin intercalamiento. (b) Intercalamiento sencillo. (c) Intercalamiento doble.

Una vez completado el formateo de bajo nivel, el disco se divide en particiones. Desde el punto de vista lógico, cada partición es como si fuera un disco separado. En el Pentium y en la mayoría de los demás ordenadores, el sector 0 contiene el **registro de arranque maestro** (*master boot record*), que contiene el programa de arranque junto con la tabla de particiones al final del sector. La tabla de particiones contiene para cada partición su sector de comienzo y su tamaño. En el Pentium la tabla de particiones tiene espacio para sólo cuatro particiones. Si todas ellas son para Windows, recibirán los nombres C:, D:, E: y F:, y se tratan como unidades separadas. Si tres de ellas son para Windows y una es para UNIX, Windows llamará a sus particiones C:, D: y E:. El primer CD-ROM será entonces F:. Para poder arrancar desde el disco duro, una partición debe marcarse como activa en la tabla de particiones.

El paso final en la preparación de un disco para su uso consiste en realizar un **formateo de alto nivel** a cada partición (de forma separada). Esta operación establece el bloque de arranque, la gestión de la memoria libre (lista de bloques libres o mapa de bits), el directorio raíz y un sistema de ficheros vacío. También se introduce un código en la entrada apropiada de la tabla de particiones para indicar qué sistema de ficheros se está utilizando en la partición, ya que cada sistema operativo utiliza normalmente un sistemas de ficheros incompatible (por razones históricas). En este momento es posible ya por fin arrancar el sistema.

Cuando se enciende el ordenador, comienza ejecutándose el BIOS, el cual lee del disco el registro de arranque maestro y le cede el control a su programa de arranque. Este programa de arranque averigua entonces qué partición está activa, lee el sector de arranque de esa partición y lo ejecuta. El sector de arranque contiene un pequeño programa que busca en el directorio raíz de la partición un cierto programa (el sistema operativo o bien un cargador de autoarranque más grande). Ese programa se carga en la memoria y se ejecuta.

5.4.2 Algoritmos de planificación del brazo del disco

En esta sección echaremos un vistazo a algunas cuestiones relacionadas con los drivers de disco en general. Comenzaremos considerando el tiempo que se requiere para leer o escribir un bloque de disco. El tiempo requerido viene determinado por tres factores:

1. Tiempo de posicionamiento (el tiempo que se tarda en mover el brazo hasta el cilindro correcto).
2. Retraso (latencia) rotacional (el tiempo que tarda el sector correcto en girar hasta pasar por debajo de la cabeza de lectura).
3. Tiempo de transferencia de datos real.

Para la mayoría de los discos, el tiempo de posicionamiento (*seek time*) domina los otros dos tiempos, por lo que una reducción en el tiempo de posicionamiento medio puede mejorar substancialmente el rendimiento del sistema.

Si el driver del disco sólo acepta una petición de cada vez y las atiende en el orden de llegada, es decir, First-Come First-Served (FCFS), poco puede hacerse para optimizar el tiempo de posicionamiento. Sin embargo, si el disco está sometido a una carga intensa de peticiones, podría utilizarse otra estrategia. Es muy probable que mientras el brazo esté moviéndose para atender una petición, otros procesos generen nuevas peticiones de disco. Muchos drivers de disco mantienen una tabla, indexada por número de cilindro, con todas las peticiones pendientes para cada cilindro encadenadas en una lista enlazada encabezada por las entradas de la tabla.

Dado este tipo de estructura de datos, podemos mejorar el algoritmo de planificación primera en llegar, primera en ser servida. Para ver cómo, consideremos un disco imaginario con 40 cilindros. Supongamos que llega una petición para leer un bloque en el cilindro 11, y que mientras el brazo está moviéndose hacia el cilindro 11, llegan nuevas peticiones para los cilindros 1, 36, 16, 34, 9 y 12, en ese orden. Esas peticiones se colocan en la tabla de peticiones pendientes, con una lista enlazada distinta para cada cilindro. Las peticiones se muestran en la Figura 5-27.

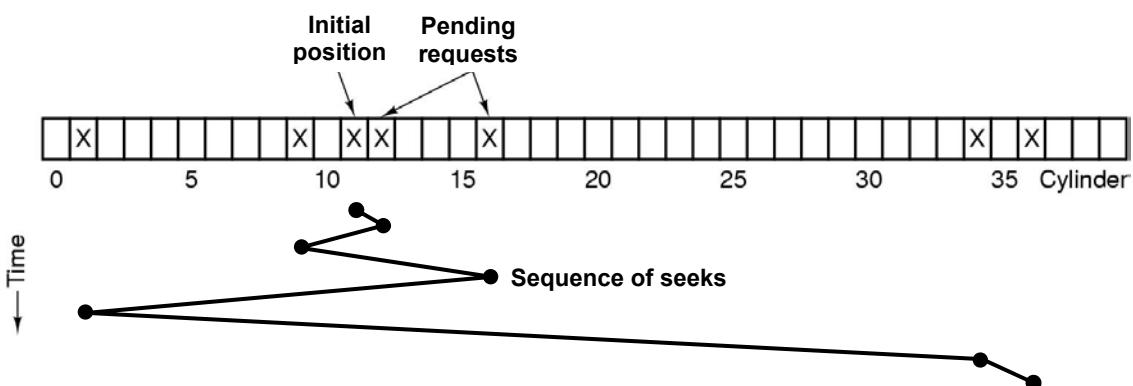


Figura 5-27. Algoritmo de planificación del disco Shortest Seek First (SSF).

Cuando termina de atenderse la petición actual (correspondiente al cilindro 11), el driver del disco tiene que elegir la petición que atenderá a continuación. Si utiliza FCFS, irá a continuación al cilindro 1, luego al 36, y así todas las demás. Este algoritmo requerirá movimientos del brazo de 10, 35, 20, 18, 25 y 3 cilindros respectivamente, para un total de 111 cilindros.

Alternativamente, el driver del disco podría atender siempre a continuación la petición más cercana, con el fin de minimizar el tiempo de posicionamiento. Dadas las peticiones de la Figura 5-27, la secuencia es 12, 9, 16, 1, 34 y 36, como indica la línea en zigzag de la parte baja de la Figura 5-27. Con esta sucesión, los movimientos del brazo son de 1, 3, 7, 15, 33 y 2, para un total de 61 cilindros. Este algoritmo, **Shortest Seek First (SSF)**, reduce el movimiento total del brazo en casi la mitad comparado con FCFS.

Desafortunadamente SSF tiene un problema. Supongamos que siguen llegando más peticiones mientras se están procesando las peticiones de la Figura 5-27. Por ejemplo, si después de posicionarnos sobre el cilindro 16 está presente una nueva petición para el cilindro 8, esa solicitud tendrá prioridad sobre la del cilindro 1. Si luego llega una petición para el cilindro 13, el brazo se dirigirá entonces al cilindro 13, no al 1. Si el disco está sometido a mucha carga de peticiones, el brazo tenderá a permanecer en el medio del disco la mayor parte del tiempo, mientras que las peticiones para ambos extremos tendrán que esperar hasta que una fluctuación estadística en la carga de peticiones provoque que no haya ninguna petición de cilindros de la zona central. En consecuencia las peticiones alejadas del centro del disco podrían recibir un mal servicio con este algoritmo. En este caso se produce un conflicto entre el objetivo de maximizar el rendimiento y el objetivo de ser justos evitando discriminar algún tipo de peticiones.

Los rascacielos también tienen que enfrentarse con esta situación comprometida. El problema de planificar el ascensor de un rascacielos es similar al de planificar el brazo de un disco. Continuamente llegan peticiones llamando al ascensor para que acuda a los pisos (cilindros) al azar. El ordenador que controla el ascensor fácilmente puede seguir la pista de la secuencia en la cual los usuarios oprimieron el botón de llamada, atendiéndolos utilizando FCFS o SSF.

Sin embargo, la mayoría de los ascensores utilizan un algoritmo diferente para conciliar los objetivos en conflicto de justicia y eficiencia. El ascensor siempre continúa su avance en el mismo sentido (hacia arriba o hacia abajo) hasta que no queden peticiones pendientes en ese sentido, momento en el cual el ascensor cambia de sentido. Este algoritmo conocido tanto en el mundo de los discos como en el de los ascensores como el **algoritmo del ascensor**, requiere que el software mantenga un bit: el bit del sentido actual del movimiento, ARRIBA o ABAJO. Cuando termina de atenderse una petición, el driver del disco o del ascensor comprueba el bit. Si es ARRIBA, el brazo o la cabina se mueve hasta la siguiente petición pendiente hacia arriba. Si no hay peticiones pendientes en posiciones más altas, se invierte el bit del sentido del movimiento. Una vez que el bit se establece a ABAJO, el movimiento es hasta la siguiente posición hacia abajo solicitada, si la hay.

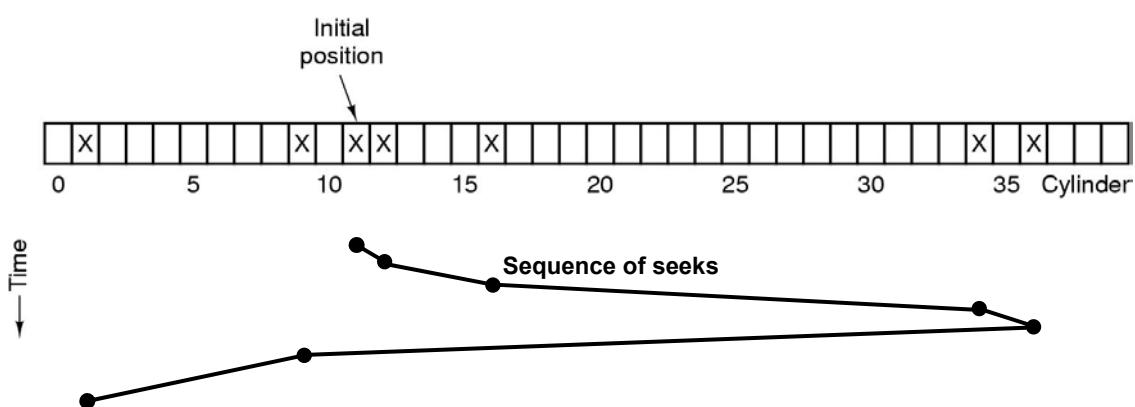


Figura 5-28. Algoritmo del ascensor para la planificación de peticiones de disco.

La Figura 5-28 muestra el algoritmo del ascensor aplicado a las mismas siete peticiones que en la Figura 5-27, suponiendo que el bit de sentido del movimiento es inicialmente ARRIBA. El orden en el que se atienden las peticiones es 12, 16, 34, 36, 9 y 1, lo que significa movimientos del brazo de 1, 4, 18, 2, 27 y 8, para un total de 60 cilindros. En este caso el algoritmo del ascensor es ligeramente mejor que el SSF, aunque normalmente es peor. Una propiedad agradable que tiene el algoritmo del ascensor es que, dado cualquier conjunto de peticiones, existe una cota superior fija sobre el movimiento total, que es exactamente el doble del número de cilindros.

Una pequeña modificación de ese algoritmo que tiene menor varianza en los tiempos de respuesta (Teory, 1972) consiste en explorar siempre en el mismo sentido. Una vez que se ha servido el cilindro de número más alto con una petición pendiente, el brazo se dirige al cilindro de número más bajo que tenga una petición pendiente, para luego continuar moviéndose hacia arriba. En definitiva es como si el cilindro de número más bajo estuviera justo a continuación del cilindro de número más alto.

Algunos controladores de disco permiten que el software conozca el número del sector que está actualmente pasando bajo la cabeza. Con un controlador así, es posible realizar otra optimización. Si hay dos o más peticiones pendientes para el mismo cilindro, el driver puede enviar una petición para el sector que pasará primero bajo la cabeza. Obsérvese que cuando un cilindro comprende varias pistas, dos peticiones consecutivas pueden corresponder a pistas distintas sin que eso represente ninguna penalización; el controlador puede seleccionar cualquiera de sus cabezas instantáneamente, ya que la selección de cabeza lectora no requiere ni movimiento del brazo ni retraso rotacional.

Si el disco tiene la propiedad de que su tiempo de posicionamiento es mucho más corto que su latencia rotacional, deberá utilizarse una estrategia de optimización diferente. Las peticiones pendientes deberán ordenarse por número de sector, y tan pronto como el siguiente sector esté a punto de pasar bajo la cabeza, el brazo deberá posicionarse sobre la pista correcta para leer o escribir en él.

En los discos duros modernos, los retrasos por posicionamiento y rotación dominan de tal manera el rendimiento que resulta ineficiente leer tan solo uno o dos sectores de cada vez. Por esa razón, muchos controladores de disco siempre leen y guardan en la caché varios sectores, incluso aunque sólo se haya solicitado uno. Normalmente, cualquier petición de lectura de un sector provocará que se lea ese sector y buena parte de la pista que lo contiene, o toda, dependiendo del espacio que esté disponible en la memoria caché del controlador. Por ejemplo, el disco descrito en la Figura 5-17 tiene una caché de 2 o 4 MB. El controlador determina dinámicamente el uso que se dará a la caché. En su modo más sencillo, la caché se divide en dos secciones, una para las lecturas y otra para las escrituras. Si puede satisfacerse una lectura posterior con el contenido de la caché de la controladora, será posible remitir de inmediato los datos solicitados.

Vale la pena señalar que la caché del controlador de disco es completamente independiente de la caché del sistema operativo. La caché del controlador normalmente contiene bloques que realmente no se han solicitado, pero que resultaba conveniente leer porque dio la casualidad de que pasaron bajo la cabeza lectora como un efecto secundario de alguna otra lectura. En contraste, cualquier caché mantenida por el sistema operativo contendrá bloques que se leyeron explícitamente y que el sistema operativo considera que podrían volverse a necesitar en un futuro cercano (por ejemplo, un bloque de disco que contiene un bloque de directorio).

Cuando un mismo controlador controla varias unidades de disco, el sistema operativo debe mantener una tabla de peticiones pendientes distinta para cada unidad. Siempre que cualquier unidad esté ociosa, el driver deberá ordenarle que vaya moviendo su brazo hacia el cilindro que se va a necesitar a continuación (supuesto que el controlador permita el

posicionamiento solapado). Cuando termina la transferencia en curso, puede comprobarse si alguna unidad de disco está ya posicionada sobre el cilindro correcto. Si una o más lo están, puede iniciarse la siguiente transferencia en una unidad que ya esté posicionada sobre el cilindro correcto. Si ninguno de los brazos está todavía en el lugar correcto, el driver deberá ordenar un nuevo posicionamiento en la unidad que acaba de terminar su transferencia quedándose a la espera de la siguiente interrupción para ver cual de los brazos llega primero a su destino.

Es importante darse cuenta de que todos los algoritmos de planificación de disco anteriores suponen tácitamente que la geometría real del disco es idéntica a la geometría virtual. Si no lo fuese, no tendría sentido planificar las peticiones de disco porque el sistema operativo realmente no puede saber si el cilindro 40 está más cerca del cilindro 39 que el cilindro 200. Por otra parte, si el controlador de disco puede aceptar múltiples solicitudes pendientes, podrá utilizar internamente esos algoritmos de planificación. En ese caso, los algoritmos siguen siendo válidos, pero a un nivel más bajo, dentro del propio controlador.

5.4.3 Tratamiento de los errores

Los fabricantes de discos siempre están empujando los límites de la tecnología, aumentando las densidades lineales de grabación de bits. Una pista situada en el medio de un disco de 5,25 pulgadas tiene una circunferencia de aproximadamente 300 milímetros. Si la pista contiene 300 sectores de 512 bytes, la densidad lineal de grabación puede ser de unos 5000 bits/milímetro tomando en cuenta el hecho de que se pierde algo de espacio debido a los preámbulos, los ECCs y los huecos entre sectores. La grabación de 5000 bits/milímetro requiere un sustrato extremadamente uniforme y un recubrimiento de óxido muy fino. Desafortunadamente, no es posible fabricar un disco con tales especificaciones que no tenga defectos. Tan pronto como la tecnología de fabricación mejora hasta el punto en el que es posible operar impecablemente con tales densidades, los diseñadores del disco se pasan a densidades más altas para aumentar la capacidad. En consecuencia vuelven a aparecer los defectos.

Los defectos de fabricación introducen sectores defectuosos, es decir, sectores que no permiten leer correctamente el valor que supone que se acaba de escribir en ellos. Si el defecto es muy pequeño, digamos de unos cuantos bits, es posible seguir utilizando el sector defectuoso dejando que el ECC corrija los errores continuamente. Si el defecto es mayor, no será posible enmascarar el error.

Hay dos enfoques generales para tratar los bloques defectuosos: que se ocupe de ellos el controlador o que se ocupe de ellos el sistema operativo. En el primer enfoque, antes de que el disco salga de la fábrica, se testea escribiéndose en el disco una lista conteniendo los sectores defectuosos. Cada sector defectuoso se sustituye por uno de los de repuesto.

Hay dos formas de realizar esa sustitución. En la Figura 5-29(a) vemos una pista de disco con 30 sectores de datos y dos de repuesto, donde el sector 7 tiene un defecto. El controlador puede reasignar el número 7 a uno de los sectores de repuesto, como se muestra en la Figura 5-29(b). La otra posibilidad es desplazar todos los sectores una posición hacia arriba, como se muestra en la Figura 5-29(c). En ambos casos el controlador tiene que saber qué sector es cada uno. Puede mantenerse al tanto de esa información con la ayuda de tablas internas (una por pista) o reescribiendo los preámbulos de modo que proporcionen los números de sector reajustados. Si se reescriben los preámbulos, el método de la Figura 5-29(c) implica más trabajo (porque hay que reescribir 23 preámbulos) pero en última instancia produce un mejor rendimiento porque sigue siendo posible leer toda una pista en una única rotación del disco.

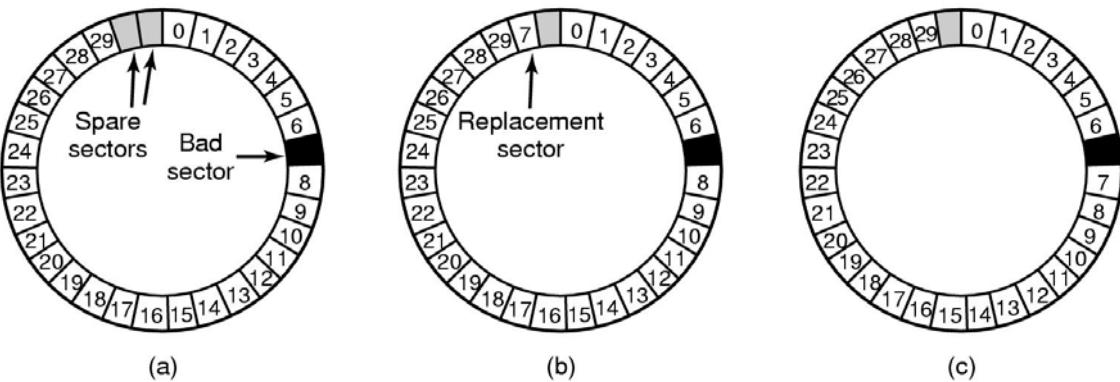


Figura 5-29. (a) Una pista de un disco con un sector defectuoso. (b) Sustitución del sector defectuoso por uno de repuesto. (c) Desplazamiento de todos los sectores para saltarse el sector defectuoso.

Los errores también pueden surgir durante el funcionamiento normal tiempo después de que se haya instalado la unidad de disco. La primera línea de defensa al presentarse un error que no puede resolver el ECC consiste en reintentar la operación. Algunos errores de lectura son transitorios, es decir, están causados por partículas de polvo que se sitúan temporalmente bajo la cabeza y desaparecen en un segundo intento. Si el controlador detecta la aparición de errores persistentes en el acceso a cierto sector, puede sustituirlo por un sector de repuesto antes de que el sector original quede completamente inutilizable. De esa manera se evita la pérdida de datos sin que ni el sistema operativo ni el usuario lleguen siquiera a enterarse del problema. Normalmente es necesario utilizar el método de la Figura 5-29(b) al encontrarse todos los demás sectores ya ocupados con datos. En ese caso la utilización del método de la Figura 5-29(c) podría requerir no sólo la reescritura de los preámbulos, sino también la copia de todos los datos.

Ya dijimos anteriormente que hay dos enfoques a la hora de tratar los errores: tratarlos en el controlador o tratarlos en el sistema operativo. Si el controlador no tiene capacidad para remapear de forma transparente los sectores como hemos visto anteriormente, deberá ser el sistema operativo quien lo haga por software. Eso significa que el sistema operativo deberá crear una lista de sectores defectuosos, bien sea leyéndola del disco o simplemente testeando él mismo todo el disco. Una vez que el sistema operativo sabe qué sectores son defectuosos, puede proceder a construir las tablas de remapeo de los bloques. En el caso de que el sistema operativo quiera utilizar el enfoque de la Figura 5-29(c) deberá desplazar un sector hacia arriba los datos de los sectores 7 a 29.

Cuando el sistema operativo realiza el remapeo de los sectores deber asegurarse de que no haya sectores defectuosos en los ficheros y tampoco en la lista o en el mapa de bits de bloques libres. Una forma de hacerlo es crear un fichero secreto integrado por todos los sectores defectuosos. Si este fichero no se incorpora al sistema de ficheros, los usuarios no podrán leerlo accidentalmente (o, lo que sería peor, liberar sus bloques).

Sin embargo, todavía queda el problema de los backups (copias de respaldo). Si se hace una copia de seguridad de un disco fichero a fichero, es importante que el programa de backup no trate de copiar el fichero de bloques defectuosos. Para evitarlo, el sistema operativo tiene que ocultar dicho el fichero de bloques defectuosos tan bien que ni siquiera un programa de backup pueda encontrarlo. Si el disco se copia sector a sector en vez de fichero a fichero, será muy difícil, si no imposible, evitar errores de lectura durante el backup. La única esperanza es que el programa de backup sea lo bastante inteligente como para darse por vencido después de 10 lecturas fallidas, continuando con el siguiente sector.

Los sectores defectuosos no son la única fuente de errores. También pueden presentarse errores de posicionamiento del brazo provocados por problemas mecánicos. El controlador sigue la pista de la posición del brazo internamente. Para realizar un posicionamiento, el controlador envía una serie de pulsos al motor del brazo, un pulso por cilindro, a fin de mover el brazo al nuevo cilindro. Una vez que el brazo llega a su destino, el controlador lee el número de cilindro actual del preámbulo del siguiente sector. Si el brazo no está en el lugar correcto, es que ha ocurrido un error de posicionamiento.

La mayoría de los controladores de disco duro corrigen automáticamente los errores de posicionamiento, pero la mayoría de los controladores de disquete (incluido el del Pentium) tan solo activan un bit de error y dejan el resto como responsabilidad del driver. El driver trata el error enviando un comando **recalibrate** para mover el brazo tan lejos como pueda ir y resetea el estado interno del controlador para que considere como cilindro actual el cilindro 0. Normalmente eso resuelve el problema. En otro caso será necesario reparar la unidad.

Como hemos visto, el controlador es en realidad un pequeño ordenador especializado, incluyendo software, variables, búferes y, de vez en cuando, errores (*bugs*). A veces sucede que una secuencia inusual de sucesos tales como una interrupción en una unidad, exactamente en el momento en que se envía un comando **recalibrate** a otra unidad puede disparar el error provocando que la controladora entre en un bucle infinito u olvide lo que estaba haciendo en ese momento. Los diseñadores de los controladores normalmente tienen previsto lo peor y proporcionan un pin en el chip que, cuando se activa, fuerza al controlador a olvidarse de todo lo que estaba haciendo y a resetearse a sí mismo. Si todo lo demás falla, el driver del disco puede establecer un bit para activar esta señal y resetear el controlador. Si eso no resuelve el problema, lo único que el driver del dispositivo puede hacer es mostrar un mensaje y darse por vencido.

La recalibración de un disco produce un ruidillo raro pero por lo demás normalmente no causa ningún problema. Sin embargo, hay una situación en la que la recalibración representa un serio problema: los sistemas con restricciones de tiempo real. Cuando se está reproduciendo un vídeo almacenado en un disco duro, o cuando se están grabando ficheros de un disco duro en un CD-ROM es esencial que los bits lleguen desde el disco duro a un ritmo uniforme. Bajo esas circunstancias, las recalibraciones insertan huecos en el flujo de bits y son por lo tanto inaceptables. Para tales aplicaciones existen unidades de disco especiales, llamadas **discos AV (discos audiovisuales)**, que nunca necesitan recalibrarse.

5.4.4 Almacenamiento Estable

Como hemos visto, en los discos a veces aparecen errores que convierten repentinamente sectores buenos en sectores defectuosos, o que destruyen unidades completas de forma inesperada. Los RAIDs protegen frente a la pérdida de unos cuantos sectores o incluso de toda una unidad, pero no nos protegen frente a errores de escritura que consigan grabar datos inicialmente erróneos. Tampoco nos protegen frente a caídas del sistema durante las escrituras, las cuales corrompen los datos originales sin reemplazarlos por nuevos datos.

En algunas aplicaciones es esencial que los datos nunca se pierdan ni corrompan, aunque se presenten errores de disco o de la CPU. Idealmente, un disco debería trabajar todo el tiempo sin errores. Desafortunadamente, la realidad no es así. Lo que sí es factible es tener un subsistema de disco que tenga la siguiente propiedad: cuando se pone en marcha una operación de escritura, o bien termina habiéndose escrito correctamente los datos, o bien termina sin efecto comunicando un fallo y dejando los datos existentes en el disco intactos. Un sistema así se denomina un **sistema de almacenamiento estable** y se implementa por software (Lamson y Sturgis, 1979). A continuación describimos la idea original con ligeras variaciones.

Antes de describir el algoritmo, es importante tener un modelo claro de los posibles errores. El modelo supone que cuando se escribe un bloque en un disco (uno o más sectores), la escritura es correcta o incorrecta, y que ese error puede detectarse en una lectura subsiguiente examinando los valores de los campos ECC. En principio, nunca es posible garantizar la detección de errores porque con un campo ECC de, digamos, 16 bytes para proteger un sector de 512 bytes existen 2^{4096} posibles valores de los datos y sólo 2^{128} posibles valores de ECC. Por tanto, si un bloque sufre alteraciones durante la escritura pero el ECC no, existen billones de billones de combinaciones incorrectas que producen el mismo ECC. Si por casualidad se presenta cualquiera de ellas, no será posible detectar el error. En general, la probabilidad de que unos datos al azar tengan el ECC de 16 bytes correcto es de 2^{-128} , una cifra lo bastante pequeña como para considerarla cero, aunque en realidad no lo sea.

El modelo considera posible que un sector escrito correctamente pueda estropearse espontáneamente, dejando de poder leerse. Sin embargo se supone que tales sucesos son tan raros que la probabilidad de que el mismo sector se estropee en una segunda unidad (independiente) durante un intervalo de tiempo razonable (por ejemplo, un día) es lo suficientemente pequeña como para poder ignorarla.

El modelo también supone que la CPU puede fallar, en cuyo caso simplemente para. Cualquier escritura en disco que esté en progreso en el momento del fallo también se detendrá, dejando datos incorrectos en un sector y un ECC incorrecto que más tarde podrá detectarse. Bajo todas esas condiciones puede conseguirse un sistema de almacenamiento estable 100% fiable en el sentido de que las escrituras o bien funcionan de forma correcta, o dejan intactos los datos que había anteriormente. Por supuesto que esto no nos protege frente a desastres físicos, tales como que se produzca un terremoto que provoque que el ordenador caiga por una grieta de 100 metros y se hunda en un río de lava hirviente. Mediante software resulta imposible recuperarse de una situación así.

El almacenamiento estable utiliza un par de discos idénticos en los que los bloques correspondientes colaboran para formar un bloque libre de errores. En ausencia de errores, los bloques correspondientes en ambas unidades son iguales. Leyendo cualquiera de los dos bloques se obtiene el mismo resultado. Para conseguir el almacenamiento estable, se definen las siguientes tres operaciones:

1. **Escrituras estables.** Una escritura estable consiste en escribir primero el bloque en la unidad 1, y luego leerlo para verificar que se escribió correctamente. Si eso no fue así, se repite la escritura y la posterior lectura n veces hasta que funcionen. Tras n fracasos consecutivos, el bloque se remapea en un bloque de repuesto, repitiéndose la operación hasta que se tenga éxito, sin importar cuántos bloques de repuesto sea preciso utilizar. Una vez que se ha logrado escribir correctamente en la unidad 1, se escribe y se relee el bloque correspondiente en la unidad 2, reintentándolo varias veces si es necesario, hasta lograrlo. En ausencia de fallos de la CPU, al terminarse la escritura estable se habrá escrito correctamente el bloque y se habrá verificado en ambas unidades.
2. **Lecturas estables.** Una lectura estable lee primero el bloque de la unidad 1. Si eso produce un ECC incorrecto, la lectura vuelve a intentarse hasta n veces. Si en todas esas lecturas se obtiene un ECC incorrecto, se lee el bloque correspondiente de la unidad 2. Dado el hecho de que una escritura estable con éxito deja dos copias correctas del bloque, y dada nuestra suposición de que es insignificante la probabilidad de que el mismo bloque se estropee espontáneamente en ambas unidades dentro de un intervalo de tiempo razonable, una lectura estable siempre tiene éxito.

3. **Recuperación después de caídas.** Después de una caída del sistema, un programa de recuperación explora ambos discos comparando bloques correspondientes. Si un par de bloques son correctos e iguales, no se hace nada. Si uno de ellos tiene un error de ECC, el bloque erróneo se sobrescribe con el bloque correcto correspondiente. Si un par de bloques son aparentemente correctos pero diferentes, el bloque de la unidad 1 se escribe en la unidad 2.

En ausencia de fallos de la CPU, este esquema siempre funciona ya que las escrituras estables siempre escriben dos copias válidas de cada bloque y se supone que los errores espontáneos nunca ocurren en ambos bloques correspondientes al mismo tiempo. Pero, ¿qué sucede si la CPU falla durante una escritura estable? Todo depende del momento preciso en que haya ocurrido el fallo. Hay cinco posibilidades, que se ilustran en la Figura 5-30.

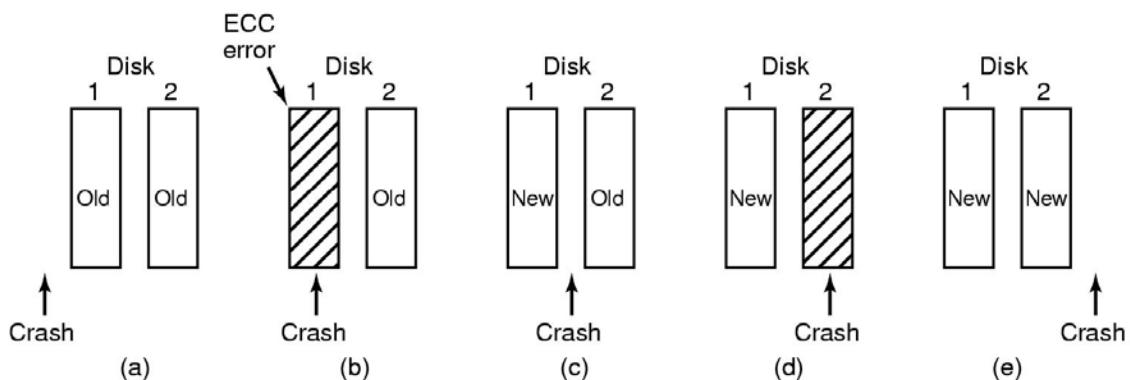


Figura 5-30. Análisis de la influencia de las caídas del sistema sobre las escrituras estables.

En la Figura 5-30(a), el fallo de la CPU se presenta antes de que se escriba cualquiera de las copias del bloque. Durante la recuperación no se modifica ninguna de ellas por lo que seguirá existiendo el antiguo valor, lo cual es correcto.

En la Figura 5-30(b), la CPU falla durante la escritura en la unidad 1, destruyendo los contenidos del bloque. Sin embargo, el programa de recuperación detecta este error y restaura el bloque de la unidad 1 a partir del bloque correspondiente de la unidad 2. Por tanto el efecto del fallo se anula, restaurándose completamente el estado anterior a la primera escritura.

En la Figura 5-30(c), el fallo de la CPU se presenta después de que se ha escrito en la unidad 1 pero antes de escribir en la unidad 2. En ese momento se ha rebasado ya el punto de no retorno: el programa de recuperación copia el bloque de la unidad 1 a la unidad 2, y la escritura se completa con éxito.

La Figura 5-30(d) es similar a la Figura 5-30(b): durante la recuperación, el bloque defectuoso se sobrescribe con el bloque correcto de la unidad 1. Otra vez, el valor final de ambos bloques es el nuevo.

Finalmente, en la Figura 5-30(e) el programa de recuperación ve que ambos bloques son iguales, por lo que no modifica ninguno, completándose con éxito la escritura.

Son posibles varias optimizaciones y mejoras sobre este esquema. Para empezar, es factible, pero costoso, comparar todos los bloques por pares después de un fallo. Una mejora enorme consiste en mantenerse al tanto de qué bloque se está escribiendo durante una escritura

estable, de modo que sólo sea necesario verificar un bloque durante la recuperación. Algunos ordenadores cuentan con una pequeña cantidad de **RAM no volátil** que es una memoria CMOS especial alimentada por una batería de litio. Tales baterías duran años, posiblemente toda la vida útil del ordenador. A diferencia de la memoria principal, cuyo contenido se pierde después de un fallo, el contenido de la RAM no volátil no se pierde en esos casos. Normalmente se guarda en ella la hora del día (incrementándose automáticamente mediante un circuito especial), y gracias a esto es por lo que los ordenadores siguen manteniendo la hora correcta incluso después de apagarse.

Supongamos que se cuenta con unos cuantos bytes de RAM no volátil que el sistema operativo puede usar para sus fines. La escritura estable puede colocar en la RAM no volátil el número del bloque que está a punto de actualizar, antes de iniciar la escritura. Una vez terminada con éxito la escritura estable, el número de bloque en la RAM no volátil se sobrescribe con un número de bloque inválido, por ejemplo -1. Bajo estas condiciones, después de una caída del sistema el programa de recuperación podrá consultar la RAM no volátil para saber si se estaba en medio de una escritura estable durante el fallo y, en tal caso, saber exactamente qué bloque se estaba escribiendo cuando se cayó el sistema. A continuación puede verificarse ya la corrección y consistencia de las dos copias del bloque.

Si no se cuenta con RAM no volátil, de todas maneras puede simularse como sigue. Al principio de una escritura estable, se sobrescribe un bloque de disco fijado de la unidad 1 con el número del bloque que se quiere escribir de manera estable. Luego se lee ese bloque para verificarlo. Una vez que se comprueba que es correcto se escribe y verifica el bloque correspondiente en la unidad 2. Cuando termine correctamente la escritura estable, ambos bloques se sobrescriben con un número de bloque inválido y se verifican. Después de un fallo, también será fácil determinar con este esquema si se estaba en medio de una escritura estable durante la caída. Por supuesto, esta técnica requiere ocho operaciones de disco adicionales para escribir un bloque estable, por lo que sólo deberá utilizarse cuando de verdad sea indispensable.

Vale la pena destacar un último punto. Supusimos que sólo uno de los bloques de un par de bloques dado puede estropearse espontáneamente en un mismo día. Si pasan suficientes días, podría estropearse también el otro bloque. Por tanto, debe efectuarse una exploración completa de ambos discos una vez al día, reparando cualquier daño que se detecte. De esta manera, al comienzo de cada mañana ambos discos siempre serán idénticos. Incluso si ambos bloques de un par se estropean dentro de un periodo de unos pocos días, se conseguirá reparar todos los errores correctamente.

5.10 INVESTIGACIÓN SOBRE ENTRADA/SALIDA

Se están realizando numerosas investigaciones sobre la entrada/salida, pero la mayoría de ellas se concentran sobre dispositivos específicos, y no sobre la E/S en general. A menudo el objetivo es mejorar el rendimiento de una forma u otra.

Los sistemas de disco son un ejemplo a destacar. Los algoritmos de planificación del brazo del disco más antiguos utilizan un modelo de disco que realmente ya no es aplicable, así Worthington y otros (1994) echaron un vistazo a los modelos que corresponden a los discos modernos. RAID es un tema de moda del que se están ocupando muchos investigadores en relación con diversos aspectos de estos sistemas. Álvarez y otros (1997) estudiaron la forma de mejorar la tolerancia a fallos, como hizo Blaum y otros (1994). Cao y otros (1994) examinaron la idea de tener un controlador paralelo en un RAID. Wilkes y otros (1996) describieron un sistema RAID avanzado que construyeron para HP. Tener múltiples unidades de disco requiere una buena planificación paralela, por lo que también se está investigando este tema (Chen y Towsley, 1996; y Kallahalla y Varman, 1999). Lumb y otros (2000) han argumentado a favor de precargar datos durante el tiempo que transcurre desde que el brazo termina su posicionamiento hasta que el sector requerido pasa por debajo de la cabeza. Incluso mejor que aprovechar el tiempo de latencia rotacional para hacer trabajo útil es eliminar completamente la rotación utilizando un dispositivo de almacenamiento microelectromecánico de estado sólido (Griffin y otros., 2000; y Carley y otros, 2000) o almacenamiento holográfico (Orlov, 2000). Otra tecnología nueva que hay que tener muy presente es el almacenamiento magneto óptico (McDaniel, 2000).

El terminal SLIM ofrece una versión moderna del antiguo sistema de tiempo compartido, con toda la computación realizándose de forma centralizada y poniendo a disposición de los usuarios terminales que simplemente gestionan la pantalla, el ratón, el teclado, y nada más (Schmidt y otros, 1999). La diferencia principal con respecto al tiempo compartido de los viejos tiempos es que en lugar de conectar el terminal al ordenador mediante un modém de 9600 bps, se utiliza una Ethernet a 10 Mbps, que proporciona suficiente ancho de banda para una interfaz gráfica completa a disposición del usuario.

Las GUIs se han estandarizado en gran medida, pero todavía se está trabajando mucho en ese campo, por ejemplo en el uso de entradas de voz (Malkewitz, 1998; Manaris y Harkreader, 1998; Slaughter y otros, 1998; y Van Buskirk y LaLomia, 1995). La estructura interna de la GUI es también tema de investigación (Taylor y otros, 1995).

Dado el gran número de científicos informáticos que tienen ordenadores portátiles y dada la microscópica duración de la batería en la mayoría de ellos, no debe sorprendernos el gran interés que hay en utilizar técnicas de software para administrar y ahorrar la energía de las baterías (Ellis, 1999; Flinn y Satyanarayanan, 1999; Kravets y Krishnan, 1998; Lebeck y otros, 2000; Lorch y Smith, 1996, y Lu y otros, 1999).

5.11 RESUMEN

La entrada/salida es un tema que se descuida a menudo, a pesar de su importancia. Una gran parte de cualquier sistema operativo corresponde a la E/S. Hay tres formas de realizar E/S. En primer lugar está la E/S programada, en la cual la CPU principal transmite o recibe cada byte o palabra, debiendo a continuación esperar dando vueltas dentro de un pequeño bucle hasta que sea posible enviar o recibir el siguiente byte. En segundo lugar está la E/S dirigida por interrupciones, en la cual la CPU inicia una transferencia de E/S para un carácter o palabra, pasando a hacer otra cosa hasta que llega una interrupción que le avisa de que ya terminó la E/S. En tercer lugar está el DMA, en el cual un chip aparte gestiona la transferencia completa de todo un bloque de datos, produciendo una única interrupción cuando termina la transferencia de todo el bloque.

La E/S puede estructurarse en cuatro niveles: las rutinas de tratamiento de las interrupciones, los drivers de los dispositivos, el software de E/S independiente del dispositivo y las bibliotecas de E/S y los programas de *spooling* que se ejecutan en el espacio de usuario. Los drivers de los dispositivos se encargan de los detalles de operación de los dispositivos, así como de presentar interfaces uniformes al resto del sistema operativo. El software de E/S independiente del dispositivo se encarga de cosas como informar de los errores o disponer los búferes que resulten convenientes.

Hay diversos tipos de discos, incluyendo discos magnéticos, RAIDs y varios tipos de discos ópticos. En muchos casos es posible utilizar algoritmos de planificación del brazo del disco para mejorar el rendimiento, pero la presencia de geometrías virtuales complica mucho las cosas. Si se emparejan dos discos, puede construirse un medio de almacenamiento estable con ciertas propiedades útiles.

Los relojes sirven para llevar el control del tiempo real, limitar el tiempo de ejecución de los procesos, implementar temporizadores vigilantes y llevar la contabilidad de los recursos utilizados.

Los terminales orientados a caracteres tienen varios aspectos relacionados con los caracteres especiales que pueden introducirse y las secuencias de escape especiales que pueden generarse. Las entradas pueden estar en modo crudo o en modo elaborado, dependiendo del grado de control que quiera tener el programa sobre ellas. Las secuencias de escape en la salida controlan el movimiento del cursor y permiten insertar y borrar texto en la pantalla.

Muchos ordenadores personales utilizan GUIs para visualizar sus salidas. Estas interfaces se basan en el paradigma WIMP: ventanas, iconos, menús y dispositivo señalador. Los programas basados en GUI están controlados normalmente por eventos, siendo los eventos de teclado, de ratón y de otro tipo enviados al programa para que los proceсе tan pronto como ocurran.

Hay varios tipos de terminales de red. Uno de los más populares es el que ejecuta X, un sistema sofisticado que puede utilizarse para construir diversas GUIs. Una alternativa a X Windows es una interfaz de bajo nivel que simplemente envía píxeles crudos a través de la red. Los experimentos realizados con el terminal SLIM muestran que esta técnica produce un rendimiento sorprendentemente bueno.

Por último, la administración de la energía es una cuestión principal para los ordenadores portátiles debido a la limitada duración de las baterías. El sistema operativo puede utilizar diversas técnicas para reducir el consumo de energía. Los programas también pueden ayudar sacrificando algo en la calidad del servicio a cambio de que las baterías duren más.

El propósito de la ruta de datos es ejecutar algún repertorio de instrucciones. Algunas de esas instrucciones pueden completarse en un único ciclo de ruta de datos; otras pueden requerir varios ciclos de ruta de datos. Las instrucciones pueden utilizar registros u otros recursos del hardware. Juntos, el hardware y las instrucciones visibles para el programador en lenguaje ensamblador constituyen el nivel **ISA** (*Instruction Set Architecture*; Arquitectura del Repertorio de Instrucciones). A este nivel se le denomina a menudo el nivel del **lenguaje máquina**.

El lenguaje máquina tiene típicamente entre 50 y 300 instrucciones, la mayoría de las cuales son para mover datos dentro de la máquina, hacer operaciones aritméticas y comparar valores. En este nivel, los dispositivos de entrada/salida se controlan cargando valores en **registros especiales de los dispositivos**. Por ejemplo, puede encargarse la lectura de un sector del disco cargando los valores de la dirección del sector en el disco, la dirección de memoria principal, el número de bytes y la direccionalidad (lectura o escritura) en sus registros. En la práctica, se necesita especificar muchos más parámetros, y la información de estado retornada por la unidad después de una operación es enormemente compleja. Además, en la programación de muchos dispositivos de E/S (Entrada/Salida) juega un papel muy importante una adecuada temporización.

Para ocultar esa complejidad se proporciona un sistema operativo, el cual consiste en una capa de software que oculta (parcialmente) el hardware y da al programador un repertorio de instrucciones más conveniente con el que trabajar. Por ejemplo, `read block from file` es más simple conceptualmente que tener que preocuparse sobre los detalles de cómo mover las cabezas lectoras, esperar a que se estabilicen, etcétera.

Por encima del sistema operativo está el resto del software del sistema. Aquí encontramos el intérprete de comandos (*shell*), los sistemas de ventanas, los compiladores, los editores y los demás programas independientes de la aplicación. Es importante darse cuenta de que ciertamente esos programas no son parte del sistema operativo, incluso a pesar de que generalmente sea el fabricante del ordenador quien nos los proporcione. Éste es un punto crucial, pero sutil. El sistema operativo es (usualmente) la porción del software que se ejecuta en **modo núcleo** (*kernel*) o **modo supervisor**, de forma que está protegido frente a la manipulación por parte del usuario (ignorando por el momento algunos antiguos microprocesadores de gama baja que no cuentan absolutamente con ningún hardware de protección). Los compiladores y los editores se ejecutan en **modo usuario**. Si a un usuario no le agrada un compilador particular, es muy libre de escribir su propio compilador si así lo desea; sin embargo no es libre para escribir su propia rutina de tratamiento de la interrupción del reloj, la cual es parte del sistema operativo, y por tanto normalmente estará protegida por el hardware frente a cualquier intento por parte del usuario de modificarla.

Sin embargo esta distinción queda a veces muy difuminada en los sistemas empotrados (los cuales pueden no disponer de modo supervisor) o en los sistemas interpretados (tales como los sistemas operativos basados en Java, los cuales utilizan la interpretación en vez del hardware para separar los componentes). No obstante para los ordenadores tradicionales sí se cumple que el sistema operativo es todo aquello que se ejecuta en modo supervisor.

Dicho esto, en muchos sistemas hay programas que se ejecutan en modo usuario pero que ayudan al sistema operativo o realizan funciones privilegiadas. Por ejemplo es frecuente contar con un programa que permite a los usuarios cambiar su contraseña (*password*). Este programa no es parte del sistema operativo y no se ejecuta en modo supervisor, pero claramente lleva a cabo una función comprometida y tiene que ser protegida de una forma especial.

En algunos sistemas esta idea se lleva al extremo, y fragmentos de lo que tradicionalmente se considera que es el sistema operativo (tales como el sistema de ficheros) se ejecutan en el espacio del usuario. En tales sistemas es difícil trazar claramente una frontera.

Todo aquello que se ejecuta en modo supervisor es claramente parte del sistema operativo, pero puede argumentarse que algunos programas ejecutándose fuera son también parte de él, o al menos están estrechamente asociados con él.

Finalmente, por encima de los programas del sistema están los programas de aplicación. Estos programas los compran o los escriben los usuarios para resolver sus problemas particulares, tales como el procesamiento de textos, la gestión de hojas de cálculo, los cálculos de ingeniería o el almacenamiento de información en una base de datos.

1.1 ¿QUÉ ES UN SISTEMA OPERATIVO?

La mayoría de los usuarios de un ordenador han tenido alguna experiencia con un sistema operativo, pero es difícil atrapar una definición precisa de lo que es realmente un sistema operativo. Parte del problema reside en que los sistemas operativos realizan dos funciones básicamente no relacionadas, extendiendo la máquina y gestionando los recursos, y dependiendo de quien esté hablando, uno oye más sobre una función que sobre la otra. Vamos a ver ahora esas dos funciones.

1.1.1 El Sistema Operativo como una Máquina Extendida

Como se mencionó anteriormente, la **arquitectura** (repertorio de instrucciones, organización de la memoria, E/S y estructura del bus) de la mayoría de los ordenadores al nivel del lenguaje máquina es primitiva y muy difícil de programar, especialmente en lo que respecta a la entrada/salida. Para hacer este punto más concreto, veamos brevemente cómo se realiza la E/S desde la disquetera utilizando un chip controlador compatible con el NEC PD765 como el que se utiliza en la mayoría de los ordenadores personales basados en Intel. El controlador PD765 tiene 16 instrucciones, cada una de las cuales se especifica cargando entre uno y nueve bytes en un registro de dispositivo. Estas instrucciones son para leer y escribir datos, mover el brazo del disco y formatear pistas, así como para inicializar, detectar, resetear y recalibrar el controlador y las unidades de disco.

Las instrucciones más básicas son **read** y **write**, cada una de las cuales requiere 13 parámetros, comprimidos en 9 bytes. Estos parámetros especifican la dirección del bloque de disco a leer, el número de sectores por pista, el modo de grabación empleado sobre el medio físico, la separación entre sectores, qué hacer con una marca de dirección de datos borrada, y cosas por el estilo. Si el lector no entiende esta jerga, no debe preocuparse; de hecho, eso es precisamente lo que se quiere poner de manifiesto: que todo el asunto es un tanto esotérico. Una vez que se lleva a cabo la operación, el controlador devuelve 23 campos de estado y error comprimidos en 7 bytes. Por si no fuera suficiente, en todo momento el programador de la disquetera debe preocuparse también de saber si el motor está encendido o apagado. Si está apagado, habrá que encenderlo (con un largo retraso hasta que el disquete adquiera la velocidad adecuada) antes de poder leer o escribir los datos. Pero el motor no puede quedarse encendido demasiado tiempo, ya que en ese caso el disquete se desgastaría rápidamente. Así, el programador se ve forzado a encontrar una solución de compromiso entre tener largos retrasos de arranque o permitir peligrosos desgastes del disquete (con la posibilidad de perder datos grabados).

Sin entrar en los detalles *reales*, debe quedar claro que es probable que el programador medio no quiera involucrarse demasiado íntimamente con los pormenores de la programación de los disquetes (o discos duros, que son igual de complejos aunque muy diferentes). En vez de eso, lo que el programador quiere es trabajar con una abstracción de alto nivel sencilla. En el caso de los discos, una abstracción típica sería que el disco contiene una colección de ficheros con nombre. Cada fichero puede abrirse para lectura o escritura, para luego leer o escribir en él, debiendo finalmente cerrarse. Los detalles de si la grabación debe realizarse por modulación de

frecuencia modificada o si el motor está encendido o apagado, no deben aparecer en la abstracción que se presenta al usuario.

El programa que oculta al programador la verdad acerca del hardware y presenta una visión bonita y sencilla de ficheros con nombre que se pueden leer y en los que se puede escribir, es por supuesto, el sistema operativo. Así como el sistema operativo separa al programador del hardware del disco y presenta una interfaz sencilla orientada hacia los ficheros, también oculta muchos otros asuntos desagradables relacionados con las interrupciones, timers, gestión de memoria y otras características de bajo nivel. En cada caso la abstracción que ofrece el sistema operativo es más sencilla y más fácil de usar que la que ofrece el hardware subyacente.

Desde esta perspectiva, la función del sistema operativo es presentar al usuario el equivalente de una **máquina extendida** o **máquina virtual** que es más fácil de programar que el hardware subyacente. La forma en la que el sistema operativo logra este objetivo es una larga historia, que estudiaremos en detalle a lo largo del libro. En pocas palabras, el sistema operativo presta una variedad de servicios que los programas pueden obtener empleando instrucciones especiales que se conocen como llamadas al sistema. Examinaremos algunas de las más comunes en una sección posterior de este capítulo.

1.1.2 El Sistema Operativo como un Gestor de Recursos

El concepto de sistema operativo como algo que proporciona primordialmente a sus usuarios una interfaz cómoda es un enfoque descendente (top-down). Un enfoque alternativo, ascendente (bottom-up), diría que el sistema operativo está ahí para administrar todos los elementos de un sistema complejo. Los ordenadores modernos constan de procesadores, memorias, timers, discos, ratones, interfaces de red, impresoras y una amplia gama de otros dispositivos. Según esta perspectiva alternativa, la tarea del sistema operativo consiste en asegurar un reparto ordenado y controlado de los procesadores, memorias y dispositivos de E/S, entre los diversos programas que compiten por obtenerlos.

Imaginemos qué sucedería si tres programas que se ejecutan en algún ordenador trataran de imprimir sus salidas al mismo tiempo por la misma impresora. Las primeras líneas del listado podrían provenir del programa 1, dos o tres siguientes del programa 2, luego algunas del programa 3, y así. El resultado sería un caos. El sistema operativo puede imponer orden en el caos potencial colocando en búferes de disco todas las salidas dirigidas a la impresora. Al terminar un programa, el sistema operativo podrá copiar sus salidas del fichero en disco donde las almacenó, a la impresora, y mientras tanto otro programa puede seguir generando más salidas, sin ser consciente de que no se están enviando (todavía) a la impresora.

Cuando un ordenador (o red de ordenadores) tiene múltiples usuarios, la necesidad de administrar y proteger la memoria, los dispositivos de E/S y los demás recursos es aún mayor, ya que en otro caso los usuarios podrían interferirse entre sí. Es común que los usuarios tengan que compartir no solo el hardware, sino también la información (ficheros, bases de datos, etcétera). En pocas palabras, esta perspectiva del sistema operativo dice que su tarea primordial es mantenerse al tanto de quién está utilizando cada recurso, conceder recursos solicitados, contabilizar el uso de los recursos y resolver los conflictos que se presenten entre las solicitudes de los diferentes programas y usuarios.

La administración de los recursos incluye la multiplexación de los recursos de dos formas: en el tiempo y en el espacio. Cuando un recurso se multiplexa en el tiempo, eso significa que varios programas o usuarios se turnan para usarlo. Primero uno de ellos usa el recurso, luego otro, y así. Por ejemplo, si sólo hay una CPU y varios programas quieren ejecutarse, el sistema operativo asignará primero la CPU a un programa; luego, cuando considere que ya se ha ejecutado durante suficiente tiempo, le quitará la CPU y se la asignará a

otro programa, luego a otro, y en algún momento al primero otra vez. La determinación de cómo se multiplexa el recurso en el tiempo – quién sigue y durante cuánto tiempo – es tarea del sistema operativo. Otro ejemplo de multiplexación en el tiempo es una impresora compartida. Cuando hay varios trabajos de impresión esperando para imprimirse en una misma impresora, es preciso decidir qué trabajo se imprimirá a continuación.

El otro tipo de multiplexación es en el espacio. En lugar de que los clientes se turnen, cada uno recibe una parte del recurso. Por ejemplo, la memoria principal normalmente se reparte entre los programas que están en ejecución, de forma que todos estén residentes al mismo tiempo (por ejemplo para poder turnarse en el uso de la CPU). Suponiendo que haya suficiente memoria para contener varios programas, suele ser más eficiente tener varios programas en la memoria a la vez, que asignarle toda la memoria a uno de ellos, sobre todo si cada programa sólo necesita una pequeña fracción del total de la memoria. Desde luego, esto hace surgir problemas de equidad, protección, etcétera, y corresponde al sistema operativo resolverlos. Otro recurso que se multiplexa en el espacio es el disco (duro). En muchos sistemas, un único disco puede contener ficheros de muchos usuarios al mismo tiempo. Repartir el espacio de disco y mantenerse al tanto de quién está usando cada bloque del disco es una tarea de administración de recursos típica del sistema operativo.

1.2 HISTORIA DE LOS SISTEMAS OPERATIVOS

Los sistemas operativos han evolucionado enormemente con el paso de los años. En las siguientes secciones mencionaremos brevemente algunos de los momentos históricos más sobresalientes. Puesto que históricamente los sistemas operativos han estado íntimamente ligados con la arquitectura de los ordenadores en los que se han ejecutado, examinaremos las generaciones sucesivas de ordenadores para ver cómo eran sus sistemas operativos. Esta correspondencia entre las generaciones de sistemas operativos y las generaciones de ordenadores es bastante cruda, pero proporciona alguna estructura donde de otra manera no habría ninguna.

El primer ordenador digital verdadero fue diseñado por el matemático inglés Charles Babbage (1792-1871). Aunque Babbage gastó la mayor parte de su vida y de su fortuna intentando construir su “máquina analítica”, nunca logró que funcionara adecuadamente debido a que era una máquina puramente mecánica, y la tecnología de su época no era capaz de producir las ruedas, engranajes y levas necesarias con la suficiente precisión que se necesitaba. Es innecesario decir que la máquina analítica carecía por completo de un sistema operativo.

Como nota histórica interesante, Babbage se dio cuenta de que necesitaría software para su máquina analítica, por lo que contrató a una joven mujer llamada Ada Lovelace, hija del famoso poeta inglés Lord Byron, como la primera programadora de la historia. El lenguaje de programación Ada® se llama así en su honor.

1.2.1 La Primera Generación (1945-1955): Tubos de Vacío y Tableros de Conexiones

Tras los infructuosos esfuerzos de Babbage, hubo pocos avances en la construcción de ordenadores digitales hasta la Segunda Guerra Mundial. En torno a mediados de la década de 1940, Howard Aiken en Harvard; John von Neumann en el Instituto de Estudios Avanzados de Princeton; J. Presper Eckert y William Mauchley en la Universidad de Pensilvania, y Konrad Zuse en Alemania, entre otros, tuvieron todos éxito en la construcción de máquinas de calcular. Las primeras utilizaban relés mecánicos por lo que eran muy lentas, con tiempos de ciclo medidos en términos de segundos. Posteriormente los relés fueron reemplazados por tubos de vacío. Estas máquinas eran enormes, llenando habitaciones enteras de decenas de miles de tubos

de vacío, pero eran todavía millones de veces más lentas que incluso los ordenadores personales más baratos disponibles hoy en día.

En esos primeros tiempos, un único grupo de personas diseñaba, construía, programaba, operaba y mantenía cada máquina. Toda la programación se efectuaba en lenguaje máquina absoluto, a menudo conectando cables en tableros de conexiones (*plugboards*) para controlar las funciones básicas de la máquina. Se desconocían los lenguajes de programación (incluso se desconocía el lenguaje ensamblador). Los sistemas operativos eran algo inaudito. El modo de operación usual era que el programador reservase su turno de varias horas firmando en una hoja pegada en la pared. Cuando le llegaba el turno reservado podía ya bajar al cuarto donde estaba la máquina, insertar su tablero de conexiones en el ordenador, y pasar las siguientes escasas horas reservadas rezando para que ninguno de los cerca de 20.000 tubos de vacío se quemara durante la ejecución de su programa. Virtualmente todos los problemas eran cálculos numéricos triviales, como la preparación de tablas de senos, cosenos y logaritmos.

A principios de la década de 1950, la rutina cotidiana había mejorado un poco con la introducción de las tarjetas perforadas. Ahora era posible escribir los programas en tarjetas que la máquina podía leer, en lugar de utilizar tableros de conexiones; por lo demás, el procedimiento de operación era el mismo.

1.2.2 La Segunda Generación (1955-1965): Transistores y Sistemas por Lotes

La introducción del transistor a mediados de la década de 1950 cambió radicalmente el panorama. Los ordenadores se volvieron lo bastante fiables como para fabricarse y venderse a clientes dispuestos a pagar por ellos con la confianza de que les seguirían funcionando el tiempo suficiente como para completar algún trabajo útil. Por primera vez hubo una separación clara entre diseñadores, constructores, operadores, programadores y personal de mantenimiento.

Aquellas máquinas, que ahora se denominan **mainframes**, fueron alojadas en salas de ordenador especialmente aire-acondicionadas, con equipos de operadores profesionales para manejarlas. Sólo las grandes corporaciones, las principales agencias del gobierno o las universidades podían permitirse pagar los millones de dólares que costaban. Para ejecutar un **trabajo** (*job*), es decir un programa o conjunto de programas, un programador debía escribir primero el programa en papel (en FORTRAN o en ensamblador) y luego grabarlo en tarjetas utilizando una perforadora. Despues debía bajar el paquete de tarjetas al cuarto de entrada de los trabajos y entregárselo a uno de los operadores, pudiéndose luego ir a tomar café hasta que la salida del programa estuviese lista.

Cuando en un momento dado el ordenador terminaba de ejecutar cualquier trabajo, el operador debía acudir a la impresora, cortar las hojas de papel continuo impresas y llevarlas al cuarto de salida, donde luego el programador podía recogerlas. A continuación el operador debía tomar uno de los paquetes de tarjetas traídos desde el cuarto de entrada y colocarlo en la lectora de tarjetas conectada al ordenador. Si se necesitaba el compilador de FORTRAN, el operador tenía que coger del armario el paquete de tarjetas correspondiente y colocarlo también en la lectora. El caso es que, mientras los operadores iban de un lado para otro en el cuarto de máquinas, estaba desperdiándose la mayor parte del tiempo de ordenador.

Dado el elevado coste de los equipos, no es sorprendente que pronto se buscaran formas de reducir el tiempo desperdiciado. La solución generalmente adoptada fue el **sistema de procesamiento por lotes** (*batch systems*). La idea que había detrás era la de llenar completamente una bandeja de trabajos procedentes del cuarto de entrada para luego pasarlo a una cinta magnética, empleando un ordenador pequeño y (relativamente) barato, como el IBM 1401, el cual era muy bueno para leer tarjetas, copiar cintas e imprimir salidas, pero realmente penoso para realizar cálculos numéricos. Otras máquinas mucho más caras, como el IBM 7094, realizaban los cálculos propiamente dichos. Esta situación se muestra en la Figura 1-2.

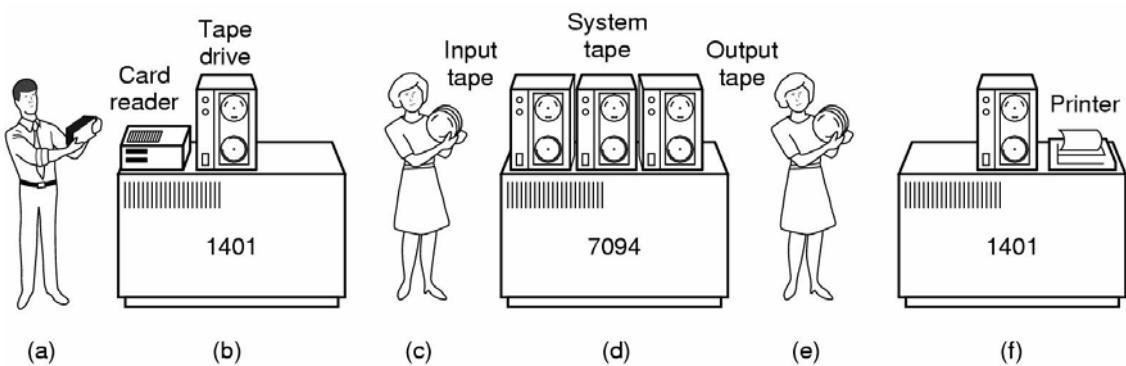


Figura 1-2. Un sistema por lotes. (a) Los programadores traen las tarjetas al 1401. (b) El 1401 lee un lote de trabajos y lo graba en cinta. (c) Un operador lleva la cinta de entrada al 7094. (d) El 7094 realiza los cálculos. (e) Un operador lleva la cinta de salida a un 1401. (f) El 1401 imprime la salida.

Después de cerca de una hora para completar un lote de trabajos, la cinta se rebobinaba y se traía al cuarto de máquinas, donde se montaba en una unidad de cinta. Luego el operador cargaba un programa especial (el antecesor del sistema operativo actual) que leía el primer trabajo de la cinta y lo ejecutaba. Las salidas se grababan en una segunda cinta, en vez de imprimirse directamente. Cada vez que se terminaba un trabajo, automáticamente el sistema operativo leía el siguiente trabajo de la cinta y comenzaba su ejecución. Cuando finalmente se terminaba de ejecutar el último trabajo del lote, el operador desmontaba las cintas de entrada y de salida, reemplazaba la cinta de entrada con el siguiente lote y llevaba la cinta de salida a un 1401 para imprimir las salidas **off line** (es decir, sin estar conectada la impresora al ordenador principal).

En la Figura 1-3 se muestra la estructura de un trabajo de entrada típico. Lo primero era una tarjeta \$JOB, que especificaba el tiempo de ejecución máximo expresado en minutos, el número de cuenta al cual cargar el coste del procesamiento y el nombre del programador. Luego venía una tarjeta \$FORTRAN, que indicaba al sistema operativo que debía cargar el compilador de FORTRAN de la cinta del sistema. Despues venía el programa a compilar y luego una tarjeta \$LOAD, que indicaba al sistema operativo que debía cargar en memoria el programa objeto recién compilado. (Los programas compilados a menudo se grababan por defecto en cintas provisionales, por lo que su carga en memoria debía solicitarse de manera explícita). A continuación venía la tarjeta \$RUN, que indicaba al sistema operativo que debía ejecutar el programa con los datos que venían a continuación de esa tarjeta. Por último, la tarjeta \$END marcaba el final del trabajo. Estas primitivas tarjetas de control fueron las precursoras de los lenguajes de control de trabajos e intérpretes de comandos modernos.

Los grandes ordenadores de la segunda generación se utilizaron principalmente para realizar cálculos científicos y de ingeniería, tales como resolver las ecuaciones en derivadas parciales que se presentan en la física y la ingeniería. Esos cálculos fueron programados principalmente en FORTRAN y en lenguaje ensamblador. Como sistemas operativos típicos de esta etapa podemos citar FMS (*Fortran Monitor System*) e IBSYS, el sistema operativo de IBM para el 7094.

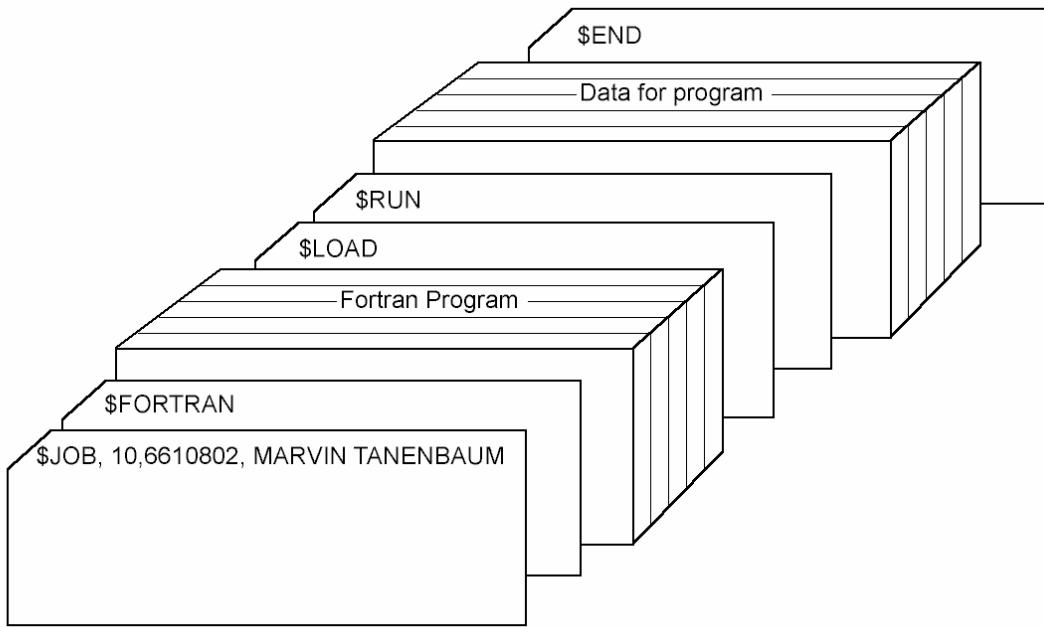


Figura 1-3. Estructura de un trabajo típico en el sistema operativo FMS.

1.2.3 La Tercera Generación (1965-1980): Circuitos Integrados y Multiprogramación

A principios de la década de 1960, la mayoría de los fabricantes de ordenadores tenían dos líneas de productos distintas, y completamente incompatibles. De un lado estaban los ordenadores científicos a gran escala, tales como el 7094, orientados a palabras y que se utilizaban para cálculos numéricos en ciencias e ingeniería. Del otro lado estaban los ordenadores comerciales orientados a caracteres, tales como el 1401, cuyo uso se había generalizado en bancos y compañías de seguros para ordenar cintas e imprimir.

Para los fabricantes de ordenadores resultaba muy caro tener que desarrollar y mantener dos líneas de productos completamente diferentes. Además, los clientes nuevos, que inicialmente compraban una máquina pequeña, deseaban tener la posibilidad de adquirir posteriormente una máquina más grande que pudiera seguir ejecutando todos sus programas anteriores, pero en menos tiempo.

IBM intentó resolver ambos problemas de un solo golpe, introduciendo el System/360. El 360 era realmente una serie de máquinas compatibles a nivel de software que iban desde ordenadores del tamaño del 1401, hasta otras máquinas mucho más potentes que la 7094. Las máquinas sólo diferían en su precio y rendimiento (máximo de memoria, velocidad del procesador, número de dispositivos de E/S permitidos, etcétera). Puesto que todas las máquinas tenían la misma arquitectura y repertorio de instrucciones, los programas escritos para una máquina podían ejecutarse en todas las demás, al menos en teoría. Además, el 360 se diseñó de modo que pudiera realizar computación tanto científica (es decir numérica) como comercial. Así una única familia de máquinas podía satisfacer las necesidades de todos los clientes. En los años siguientes, IBM sacó al mercado sucesores compatibles con la línea 360, fabricados con tecnología más moderna: las series 370, 4300, 3080 y 3090.

La línea 360 fue la primera línea de ordenadores importante que utilizó circuitos integrados (a pequeña escala), ofreciendo por ello una notable ventaja en precio y potencia respecto a las máquinas de la segunda generación, que se construían con transistores individuales. Su éxito fue inmediato, y todos los demás grandes fabricantes de ordenadores enseguida adoptaron también la idea de una familia de ordenadores compatibles. Los descendientes de estas máquinas se siguen utilizando hoy en día en los centros de cálculo, para administrar bases de datos enormes (por ejemplo, en sistemas de reserva de pasajes aéreos) o como servidores de sitios web que deben procesar miles de peticiones por segundo.

La mayor fortaleza de la idea de “una familia” es al mismo tiempo su punto más débil. La intención era que todo el software, incluido el sistema operativo **OS/360**, tenía que funcionar en todos los modelos. Tenía que hacerlo en sistemas pequeños, que a menudo eran simples sustitutos de los 1401 para copiar tarjetas en cinta, y también en sistemas muy grandes, que a menudo sustituían a los 7094 para hacer predicciones meteorológicas y efectuar otros cálculos pesados. Tenía que ser bueno en sistemas con pocos periféricos y en sistemas con muchos periféricos. Tenía que operar en entornos comerciales y en entornos científicos. Y sobre todo, tenía que ser eficiente para todos esos usos tan diferentes.

Era imposible que IBM (o cualquier otro) pudiera escribir un fragmento de software capaz de satisfacer todos esos requisitos en conflicto. El resultado fue un sistema operativo enorme y extraordinariamente complejo, quizás de dos a tres órdenes de magnitud más grande que el FMS. Estaba compuesto por millones de líneas en lenguaje ensamblador escritas por miles de programadores, y contenía miles y miles de errores, por lo que necesitaba un flujo continuo de nuevas versiones en un intento por corregirlos. Cada versión nueva corregía algunos errores e introducía otros nuevos, por lo que es probable que el número de errores haya permanecido constante a lo largo del tiempo.

Uno de los diseñadores del OS/360, Fred Brooks, escribió después un simpático y mordaz libro (Brooks, 1996) en el que describía sus experiencias con el OS/360. Aunque sería imposible resumir aquí ese libro, baste con decir que la portada muestra una manada de bestias prehistóricas atascadas en un pozo de brea. La portada de Silberschatz y otros (2000) sugiere también que los sistemas operativos son como dinosaurios.

A pesar de lo enorme de su tamaño y sus problemas, el OS/360 y los sistemas operativos de la tercera generación similares producidos por otros fabricantes de ordenadores, en realidad satisficieron de manera razonable a la mayoría de sus clientes. Además, popularizaron varias técnicas clave que no se utilizaban en los sistemas operativos de la segunda generación. Tal vez la más importante de ellas fue la **multiprogramación**. En el 7094, cuando el trabajo en curso hacía una pausa para esperar a que terminara una operación de cinta u otra operación de E/S, la CPU permanecía inactiva hasta que la E/S terminaba. En el caso de programas de cálculo científico, que hacen un uso intensivo de la CPU, la E/S es poco frecuente, así que el tiempo desperdiciado no es importante. En el procesamiento de datos comerciales, el tiempo de espera por la E/S puede ser del 80% o 90% del tiempo total, así que era urgente hacer algo para evitar que la (costosa) CPU estuviera inactiva tanto tiempo.

La solución que se desarrolló fue dividir la memoria en varias partes, con un trabajo distinto en cada partición, como se muestra en la Figura 1-4. Mientras un trabajo estaba esperando a que terminara la E/S, otro podía estar usando la CPU. De esta manera la CPU podría mantenerse ocupada casi el 100% del tiempo siempre y cuando pudieran mantenerse suficientes trabajos en la memoria principal a la vez. Tener varios trabajos en la memoria al mismo tiempo, sin causar problemas, requiere un hardware especial para proteger cada trabajo frente al fisingo y las travesuras de los demás trabajos, pero el 360 y otros sistemas de la tercera generación estaban equipados con ese hardware.

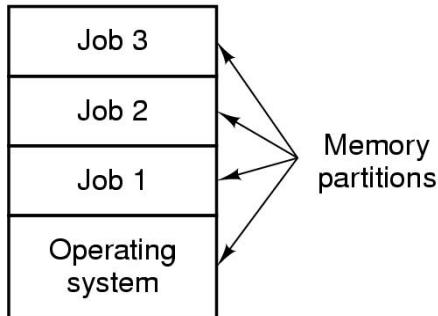


Figura 1-4. Sistema de multiprogramación con tres trabajos en la memoria

Otra característica importante de los sistemas operativos de la tercera generación era que podían leer los trabajos de las tarjetas y grabarlos en el disco tan pronto como se llevaban al cuarto de ordenadores. Así cada vez que terminaba de ejecutarse un trabajo, el sistema operativo podía cargar un trabajo nuevo del disco y colocarlo en la partición recién desocupada para ejecutarlo. Esta técnica se llama **spooling** (de *Simultaneous Peripheral Operation On Line*) y se utilizaba también para la salida de los programas. Con el spooling, dejaron de ser necesarios los 1401, y desapareció por completo el acarreo de cintas.

Aunque los sistemas operativos de la tercera generación eran muy apropiados para realizar grandes cálculos científicos y para procesar volúmenes enormes de datos comerciales, seguían siendo básicamente sistemas por lotes. Muchos programadores añoraban los tiempos de la primera generación en los que la máquina era toda para ellos durante unas cuantas horas, lo que les permitía depurar sus programas con rapidez. Con los sistemas de la tercera generación, el tiempo entre la entrada de un trabajo y la obtención de su salida solía ser de varias horas, por lo que una sola coma fuera de lugar podría hacer que fallara una compilación y que el programador perdiera medio día.

Este deseo de obtener respuestas rápidas preparó el camino para el **tiempo compartido** (*timesharing*), una variante de la multiprogramación en la que cada usuario tiene un terminal en línea. En un sistema de tiempo compartido, si 20 usuarios están trabajando y 17 de ellos están pensando o hablando o bebiendo café, la CPU puede asignarse por turno a los tres trabajos que efectivamente necesitan ser atendidos. Puesto que en la depuración de los programas por lo regular se generan trabajos cortos (por ejemplo, compilar un procedimiento de cinco páginas), en vez de trabajos largos (por ejemplo, ordenar un fichero con un millón de registros), el ordenador puede prestar un servicio rápido e interactivo a cierto número de usuarios, y quizás también ir procesando en segundo plano trabajos por lotes grandes cuando la CPU no tenga ningún trabajo interactivo que ejecutar. El primer sistema de tiempo compartido serio, **CTSS** (*Compatible Time Sharing System*), se desarrolló en el MIT en un 7094 con modificaciones especiales (Corbató y otros, 1962). Sin embargo, el tiempo compartido no se popularizó en realidad sino hasta que se generalizó el uso del hardware de protección necesario durante la tercera generación.

Después del éxito del sistema CTSS, el MIT, los Laboratorios Bell y General Electric (que era entonces un importante fabricante de ordenadores) decidieron emprender el desarrollo de un “servicio de ordenador”, una máquina que debía dar servicio simultáneamente a cientos de usuarios de tiempo compartido. Su modelo fue el sistema de distribución de la electricidad: cuando alguien necesita energía eléctrica, sólo tiene que conectar un cable en el enchufe de la pared para disponer allí mismo de toda la electricidad que necesite (dentro de lo razonable). Los diseñadores de este sistema, llamado **MULTICS** (*MULTIplexed Information and Computing Service*), imaginaron una máquina enorme capaz de proporcionar potencia de cálculo a todos los habitantes del área de Boston. La idea de que millones de máquinas mucho más potentes que su mainframe GE-645 se venderían a mil dólares cada una en apenas 30 años después era pura ciencia ficción, algo así como la idea de trenes supersónicos bajo el Atlántico.

MULTICS fue un éxito con matices. Se le diseñó para dar soporte a cientos de usuarios con una máquina apenas más potente que un PC basado en un 386 de Intel, aunque con mucha mayor capacidad de E/S. Esto no es tan absurdo como parece, porque en aquella época la gente sabía escribir programas pequeños y eficientes, habilidad que actualmente se ha perdido. Hubo muchas razones por las que MULTICS no se adueñó del mundo; una de las principales fue que estaba escrito en PL/I, y el compilador de PL/I se retrasó varios años y apenas funcionaba cuando por fin apareció. Además MULTICS era demasiado ambicioso para su época, algo así como la máquina analítica de Charles Babbage en el siglo XIX.

Resumiendo la que es una larga historia, MULTICS introdujo numerosas ideas fértiles en la literatura de los ordenadores, pero el convertirlo en un producto serio y con un éxito comercial importante resultó mucho más difícil de lo que nadie hubiera esperado. Los Laboratorios Bell se separaron del proyecto y General Electric abandonó del todo el negocio de los ordenadores. No obstante, el MIT persistió y al fin logró hacer funcionar a MULTICS. En última instancia, fue vendido como producto comercial por la compañía que adquirió la rama de ordenadores de GE (Honeywell) y se instaló en cerca de 80 compañías y universidades importantes de todo el mundo. Aunque los usuarios de MULTICS fueron pocos, mostraron una lealtad a toda prueba. General Motors, Ford y la Agencia de Seguridad Nacional de Estados Unidos, por ejemplo, no retiraron sus sistemas MULTICS hasta finales de la década de 1990, 30 años después de la primera versión de MULTICS.

Por el momento, el concepto de un servicio de ordenador ha quedado en el olvido, pero bien podría reaparecer en forma de servidores gigantes de Internet centralizados a los que se conectarían máquinas de usuario relativamente tontas, realizándose la mayor parte del trabajo en los servidores. La motivación podría ser que la mayoría de las personas no desea administrar en su ordenador un sistema cada vez más complejo y quisquilloso, y preferiría que ese trabajo lo realizará un equipo de profesionales trabajando para la compañía propietaria del servidor. El comercio electrónico está evolucionando ya en esa dirección, y varias compañías operan centros comerciales electrónicos sobre servidores multiprocesador a los que se conectan máquinas cliente sencillas, algo muy parecido en espíritu al diseño de MULTICS.

A pesar de su falta de éxito comercial, MULTICS tuvo una enorme influencia en los sistemas operativos subsiguientes. El sistema se describe en Corbató y otros (1972), Corbató y Vyssotsky (1965), Daley y Dennos (1968), Organick (1972) y Saltzer (1974). También tiene un sitio web que sigue activo, www.multicians.org, con abundante información acerca del sistema, sus diseñadores y sus usuarios.

Otro adelanto importante durante la tercera generación fue el fenomenal crecimiento de los miniordenadores, comenzando con el DEC PDP-1 en 1961. El PDP-1 sólo tenía 4K palabras de 18 bits, pero su precio de 120.000 dólares por máquina (menos del 5% del precio del 7094) hizo que se vendiera como rosquillas. Para ciertos tipos de trabajo no numérico, era casi tan rápido como el 7094, y dio origen a una industria totalmente nueva. Pronto le siguieron una serie de otros PDPs (todos incompatibles a diferencia de los miembros de la familia IBM) que culminaron en el PDP-11.

Uno de los científicos de los Laboratorios Bell que había trabajado en el proyecto MULTICS, Ken Thompson, encontró tiempo más tarde un pequeño miniordenador, un PDP-7, que nadie estaba usando y se puso a escribir una versión de MULTICS para un solo usuario recortando partes del sistema original. Esa labor dio pie más adelante al sistema operativo **UNIX®**, que se popularizó en el mundo académico, en las agencias gubernamentales y en muchas compañías.

La historia de UNIX se cuenta en otras obras (por ejemplo, Salus, 1994). Parte de esa historia se presentará en el capítulo 10. Por ahora, es suficiente con decir que, gracias a que el código fuente podía conseguirse fácilmente, varias organizaciones desarrollaron sus propias versiones (incompatibles), lo cual llevó al caos. Surgieron dos versiones principales, **UNIX System V**, de AT&T, y **UNIX BSD** (*Berkeley Software Distribution*), de la Universidad de California en Berkeley. Estas versiones tuvieron a su vez otras variantes menores. Para hacer posible la escritura de programas susceptibles de ejecutarse en cualquier sistema UNIX, el IEEE creó un estándar para UNIX, llamado **POSIX**, reconocido por la mayoría de las versiones actuales de UNIX. POSIX define una interfaz mínima de llamadas al sistema que deben entender los sistemas UNIX compatibles. De hecho, algunos otros sistemas operativos soportan también ya el interfaz POSIX.

Como nota interesante, vale la pena mencionar que en 1987, el autor publicó un clon pequeño de UNIX, llamado **MINIX**, con fines educativos. Desde el punto de vista funcional, MINIX es muy similar a UNIX, y es compatible con POSIX. También hay un libro que describe su funcionamiento interno y presenta el código fuente en un apéndice (Tanenbaum y Woodhull, 1997). Puede obtenerse MINIX de forma gratuita (incluido todo el código fuente) en Internet en la dirección www.cs.vu.nl/~ast/minix.html.

El deseo de contar con una versión de producción libre (no meramente educativa) de MINIX llevó a un estudiante finlandés, Linus Torvalds, a escribir **Linux**. Este sistema se desarrolló bajo MINIX y originalmente soportaba varios elementos característicos de MINIX (como el sistema de ficheros). Desde entonces Linux ha sido extendido en muchas direcciones pero sigue conservando una buena parte de la estructura subyacente de MINIX y UNIX (en el que se basó el primero). Por tanto, casi todo lo que se diga en este libro sobre UNIX será válido también para System V, BSD, MINIX, Linux y otras versiones y clones de UNIX.

1.2.4 La Cuarta Generación (de 1980 hasta el presente): Ordenadores Personales

Con el desarrollo de los circuitos integrados a gran escala (LSI; *Large Scale Integration*), es decir chips que contienen miles de transistores en un centímetro cuadrado de silicio, surgió la era del ordenador personal. Desde el punto de vista de la arquitectura, los ordenadores personales (denominados al principio **microordenadores**) no eran muy diferentes de los miniordenadores de la clase PDP-11, pero desde el punto de vista del precio, sí que eran muy distintos. Mientras que el miniordenador hizo posible que un departamento de una compañía o universidad tuviera su propio ordenador, el chip microprocesador hizo posible que cualquier persona pudiera tener su propio ordenador personal.

En 1974, cuando Intel presentó el 8080, la primera CPU de ocho bits de propósito general, buscó un sistema operativo para ese procesador, en parte para poder probarlo. Intel pidió a uno de sus consultores, Gary Kildall, que escribiera uno. Kildall y un amigo construyeron primero un controlador para la unidad de disco flexible de ocho pulgadas recién salida de Shugart Associates, enganchando así el disquete al 8080, y dando lugar al primer microordenador con disco. Luego Kildall escribió un sistema operativo basado en disco llamado **CP/M (Control Program for Microcomputers)**. Intel no pensó que los microordenadores basados en disco fueran a tener mucho futuro, de manera que cuando Kildall pidió quedarse con los derechos del CP/M, Intel se lo concedió. Kildall formó entonces una compañía, Digital Research, para seguir desarrollando y vender el sistema operativo CP/M.

En 1977, Digital Research reescribió CP/M para que pudiera ejecutarse en los numerosos microordenadores que utilizaban el 8080, el Zilog Z80 u otros microprocesadores. Se escribieron muchos programas de aplicación para ejecutarse sobre CP/M, lo que permitió a este sistema operativo dominar por completo el mundo de los microordenadores durante unos cinco años.

A principios de la década de 1980, IBM diseñó el PC y buscó software que se ejecutara en él. Gente de IBM se puso en contacto con Bill Gates para utilizar bajo licencia su intérprete de BASIC. También le preguntaron si sabía de algún sistema operativo que funcionara sobre el PC. Gates sugirió a IBM que se pusiera en contacto con Digital Research, que entonces era la compañía de sistemas operativos dominante. Kildall rechazó reunirse con IBM y envió a un subordinado en su lugar, tomando la que seguro fue la peor decisión en los negocios de toda la historia. Para empeorar las cosas, su abogado se negó incluso a firmar el convenio de confidencialidad de IBM que incluía al todavía no anunciado PC. Consecuentemente, IBM fue de vuelta con Gates para preguntarle si podría ofrecerle un sistema operativo.

Cuando IBM vino a verle, Gates se percató de que un fabricante de ordenadores local, *Seattle Computer Products*, tenía un sistema operativo apropiado, **DOS** (*Disk Operating System*). Gates se reunió con el fabricante y se ofreció a comprarle el sistema (supuestamente por 50.000 dólares), lo que aceptó de buena gana. Luego Gates ofreció a IBM un paquete DOS/BASIC, que IBM aceptó. IBM pidió que se hicieran ciertas modificaciones en el sistema, por lo que Gates contrató a la persona que había escrito DOS, Tim Paterson, como empleado de su naciente compañía, Microsoft, para que las llevara a cabo. El sistema revisado se rebautizó con el nombre de **MS-DOS** (*Microsoft Disk Operating System*) y pronto dominó el mercado del IBM PC. Un factor clave aquí fue la decisión (en retrospectiva, extremadamente sabia) de Gates de vender MS-DOS a compañías de ordenadores para incluirlo con su hardware, en comparación con el intento de Kildall de vender CP/M a los usuarios finales uno por uno (al menos inicialmente).

Para cuando el PC/AT de IBM salió a la venta el 1983 con la CPU 80286 de Intel, MS-DOS estaba firmemente afianzado mientras que CP/M agonizaba. Más tarde se utilizó ampliamente MS-DOS en el 80386 y el 80486. Aunque la versión inicial de MS-DOS era más bien primitiva, sus versiones posteriores incluyeron funciones más avanzadas, incluyendo muchas procedentes de UNIX. (Microsoft tenía perfecto conocimiento de UNIX, e incluso vendió durante los primeros años de la compañía una versión para microordenador a la que llamó XENIX.)

CP/M, MS-DOS y otros sistemas operativos para los primeros microordenadores obligaban al usuario a introducir comandos a través del teclado. En un momento dado la cosa cambió, gracias a las investigaciones hechas por Doug Engelbart en el Stanford Research Institute, en los años sesenta. Engelbart inventó la **GUI** (*Graphical User Interface*; Interfaz Gráfica de Usuario), provista de ventanas, iconos, menús y ratón. Los investigadores de Xerox PARC adoptaron estas ideas, incorporándolas a las máquinas que fabricaban.

Cierto día Steve Jobs, uno de los dos jóvenes que inventaron el ordenador Apple en el garaje de su casa, visitó PARC, vio una GUI, y de inmediato se dio cuenta de su valor potencial, algo que, de forma asombrosa, no hizo la gerencia de Xerox (Smith y Alexander, 1988). Jobs se dedicó entonces a construir un Apple provisto de una GUI. Este proyecto condujo a la construcción del ordenador Lisa, que resultó demasiado caro y fracasó comercialmente. El segundo intento de Jobs, el Apple Macintosh, fue un enorme éxito, no sólo porque era mucho más económico que Lisa, sino también porque era **amigable con el usuario** (*user friendly*), lo que significa que iba dirigido a usuarios que no sólo carecían de conocimientos sobre ordenadores, sino que además no tenían ni la más mínima intención de aprender.

Cuando Microsoft decidió desarrollar un sucesor para MS-DOS, estuvo fuertemente influenciado por el éxito del Macintosh. La compañía produjo un sistema basado en una GUI al que llamó Windows, y que originalmente se ejecutaba por encima de MS-DOS (es decir, era más un *shell* que un verdadero sistema operativo). Durante cerca de 10 años, de 1985 a 1995, Windows no fue más que un entorno gráfico por encima de MS-DOS. Sin embargo, en 1995 salió a la circulación una versión autónoma de Windows, Windows 95, que incluía muchas funciones del sistema operativo y sólo utilizaba el sistema MS-DOS subyacente para arrancar y

ejecutar programas antiguos de MS-DOS. En 1998 salió una versión ligeramente modificada de este sistema, llamada Windows 98. No obstante tanto Windows 95 como Windows 98 contienen todavía una buena cantidad de lenguaje ensamblador Intel de 16 bits.

Otro sistema operativo de Microsoft es **Windows NT** (NT significa Nueva Tecnología), que es compatible hasta cierto punto con Windows 95, pero que internamente está reescrito desde cero. Se trata de un sistema completamente de 32 bits. El diseñador en jefe de Windows NT fue David Cutler, quien también fue uno de los diseñadores del sistema operativo VAX VMS, así que algunas de las ideas de VMS están presentes en NT. Microsoft confiaba en que la primera versión de NT exterminaría a MS-DOS y todas las demás versiones anteriores de Windows porque se trata de un sistema enormemente superior, pero no fue así. Sólo con Windows NT 4.0 comenzó a adoptarse ampliamente el sistema, sobre todo en redes corporativas. La versión 5 de Windows NT se rebautizó como Windows 2000 a principios de 1999, con la intención de que fuera el sucesor tanto de Windows 98 como de Windows NT 4.0. Eso tampoco funcionó como se pensaba, así que Microsoft sacó una versión más de Windows 98 llamada **Windows Me** (*Millennium edition*).

El otro contendiente importante en el mundo de los ordenadores personales es UNIX (y sus diversos derivados). UNIX predomina en estaciones de trabajo y otros ordenadores potentes, como los servidores de red. Es especialmente popular en máquinas basadas en chips RISC de alto rendimiento. En los ordenadores basados en Pentium, Linux se está convirtiendo en una alternativa popular a Windows, para estudiantes y cada vez más para usuarios corporativos. (En todo este libro utilizaremos el término “Pentium” para referirnos al Pentium I, II, III y 4.)

Aunque muchos usuarios de UNIX, sobre todo programadores experimentados, prefieren un interfaz basado en comandos en vez de una GUI, casi todos los sistemas UNIX reconocen un sistema de ventanas llamado **X Windows** producido en el MIT. Este sistema se encarga de la gestión básica de las ventanas y permite a los usuarios crear, borrar, trasladar y redimensionar ventanas con un ratón. En muchos casos se cuenta con una GUI completa, como **Motif**, que opera encima del sistema X Windows para conferir a UNIX un aspecto y manejo parecido a Macintosh o Microsoft Windows, para aquellos usuarios de UNIX que lo deseen.

Una tendencia interesante que surgió a mediados de la década de 1980 es el crecimiento de las redes de ordenadores personales que ejecutan **sistemas operativos en red** y **sistemas operativos distribuidos** (Tanembaum y Van Oteen, 2002). En un sistema operativo en red, los usuarios son conscientes de la existencia de múltiples ordenadores y pueden iniciar una sesión en las máquinas remotas, así como copiar ficheros de una máquina a otra. Cada máquina ejecuta su propio sistema operativo local y tiene su propio usuario (o usuarios) local(es).

Los sistemas operativos en red no son distintos en lo fundamental de los diseñados para un solo procesador. Por supuesto, necesitan un controlador del interfaz de red y cierto software de bajo nivel para operar, así como programas para iniciar la sesión y tener acceso a los ficheros de una máquina remota, pero estos añadidos no alteran la naturaleza fundamental del sistema operativo.

En contraste, un sistema operativo distribuido se presenta a los usuarios como un sistema monoprocesador tradicional, aunque en realidad se compone de múltiples procesadores. Los usuarios no deben preocuparse por saber dónde se están ejecutando sus programas o dónde están almacenados sus ficheros; de eso debe encargarse el sistema operativo de forma automática y eficiente.

Los verdaderos sistemas operativos distribuidos requieren algo más que añadir un poco de código a un sistema operativo monoprocesador, porque los sistemas distribuidos y los centralizados presentan diferencias cruciales. Por ejemplo, a menudo los sistemas distribuidos permiten que las aplicaciones se ejecuten en varios procesadores al mismo tiempo, lo que

requiere algoritmos de planificación del procesador más complejos a fin de optimizar el grado de paralelismo.

Los retrasos de comunicación por la red a menudo implican que estos algoritmos (y otros) deben ejecutarse con información incompleta, caducada o incluso incorrecta. Esta situación es radicalmente distinta de la que se da en un sistema con un único procesador, donde el sistema operativo cuenta con información precisa sobre el estado del todo el sistema.

1.2.5 La Ontogenia Recapitula la Filogenia

Después de publicarse el libro de Charles Darwin, *El origen de las especies*, el zoólogo alemán Ernst Haeckel estableció que “la ontogenia recapitula la filogenia”. Con esto quiso decir que el desarrollo de un embrión (ontogenia) repite (es decir, recapitula) la evolución de la especie (filogenia). En otras palabras, después de la fertilización, un embrión humano pasa por las etapas de pez, cerdo, etcétera, antes de convertirse en un bebé humano. Los biólogos modernos consideran esto una burda simplificación, pero no deja de tener su núcleo de verdad.

Algo análogo ha sucedido en la industria de los ordenadores. Parece como que cada nueva especie (mainframe, miniordenador, ordenador personal, ordenador empotrado, tarjeta inteligente, etcétera) pasa por el mismo desarrollo que sus antepasados. Los primeros mainframes se programaban por completo en lenguaje ensamblador. Incluso algunos programas complejos como los compiladores y los sistemas operativos se escribían en ensamblador. Para cuando entraron en escena los miniordenadores, FORTRAN, COBOL y otros lenguajes de alto nivel ya eran comunes en los mainframes, pero los nuevos miniordenadores se programaban en ensamblador (por la escasez de memoria). Cuando se inventaron los microordenadores (los primeros ordenadores personales), también se programaron en ensamblador, aunque para entonces los miniordenadores se programaban ya en lenguajes de alto nivel. Los ordenadores de bolsillo (palmtop) también comenzaron programándose en código ensamblador pero pronto cambiaron a lenguajes de alto nivel (sobre todo porque el trabajo de desarrollo de los programas se efectuaba en máquinas más grandes), y lo mismo ha sucedido con las tarjetas inteligentes.

Veamos ahora los sistemas operativos. Los primeros mainframes no tenían hardware de protección ni soportaban multiprogramación, por lo que ejecutaban sistemas operativos sencillos que manejaban un programa a la vez, el cual se cargaba de forma manual. Más adelante, esos ordenadores adquirieron primero el hardware y el soporte del sistema operativo necesario para manejar varios programas a la vez, y luego funciones completas de tiempo compartido.

Cuando aparecieron los miniordenadores, tampoco tenían hardware de protección y ejecutaban un único programa a la vez, también cargado de forma manual, aunque para entonces la multiprogramación ya estaba bien asentada en el mundo de los mainframes. De manera gradual, estas máquinas adquirieron hardware de protección y la capacidad de ejecutar dos o más programas a la vez. Los primeros microordenadores tampoco podían ejecutar más de un programa a la vez, pero luego adquirieron la capacidad de multiprogramar. Los ordenadores de bolsillo y las tarjetas inteligentes siguieron ese mismo camino.

Los discos aparecieron primero en los mainframes, luego en los miniordenadores, microordenadores, etc. Incluso ahora, las tarjetas inteligentes no tienen disco duro, pero con la llegada de las ROM de tipo flash, pronto tendrán algo equivalente. Cuando aparecieron por primera vez los discos, nacieron sistemas de ficheros primitivos. En el CDC 6600, el mainframe más potente del mundo durante gran parte de la década de 1960, su sistema de ficheros consistía en usuarios que podían crear un fichero y luego declararlo permanente, lo que significaba que seguiría en el disco aunque el programa que lo había creado ya hubiera terminado. Para tener acceso más adelante a tal fichero, un programa tenía que abrirlo con una instrucción especial y

proporcionar su contraseña (que se asignaba cuando el archivo se hacía permanente). En consecuencia, había un único directorio que compartían todos los usuarios, dejándose a la responsabilidad de los propios usuarios el evitar conflictos de nombres de fichero. Los primeros sistemas de ficheros de los miniordenadores tenían un único directorio compartido por todos los usuarios, y lo mismo sucedió con los sistemas de ficheros de los primeros microordenadores.

La memoria virtual (la capacidad de ejecutar programas más grandes que la memoria física) tuvo un desarrollo similar. Apareció primero en los mainframes y luego en los miniordenadores, microordenadores y así de forma gradual, hasta sistemas cada vez más y más pequeños. Las redes tuvieron una historia similar.

En todos los casos, el desarrollo del software estuvo dictado por la tecnología. Los primeros microordenadores, por ejemplo, tenían 4KB de memoria y carecían de hardware de protección. Los lenguajes de alto nivel y la multiprogramación simplemente eran demasiado para que un sistema tan diminuto pudiera soportarlos. A medida que evolucionaron los microordenadores para convertirse en los ordenadores personales modernos, adquirieron el hardware y luego el software necesarios para manejar funciones más avanzadas. Es probable que este desarrollo continúe durante varios años, y que otros campos cuenten también con este ciclo de reencarnación, pero al parecer en la industria de los ordenadores este ciclo se repite a mucha mayor velocidad.

1.3 TIPOS DE SISTEMAS OPERATIVOS

Toda esta historia y desarrollo nos ha dejado con una amplia variedad de sistemas operativos, de los cuales no todos son ampliamente conocidos. En esta sección describiremos de manera breve siete de ellos. Volveremos a algunos de estos tipos de sistemas en capítulos posteriores del libro.

1.3.1 Sistemas Operativos de Mainframe

En el extremo superior están los sistemas operativos para los mainframes, esos ordenadores gigantes que todavía se encuentran en importantes centros de cálculo corporativos. Tales máquinas se distinguen de los ordenadores personales por su capacidad de E/S. No es raro encontrar mainframes con 1000 discos y miles de gigabytes de datos; sin embargo resultaría verdaderamente extraño encontrar un ordenador personal con esas especificaciones. Los mainframes están renaciendo ahora como servidores web avanzados, servidores para sitios de comercio electrónico a gran escala y servidores para transacciones de negocio a negocio.

Los sistemas operativos para mainframes están claramente orientados al procesamiento de varios trabajos a la vez, necesitando la mayoría de esos trabajos prodigiosas cantidades de E/S. Los servicios que ofrecen suelen ser de tres tipos: procesamiento por lotes, procesamiento de transacciones y tiempo compartido. Un sistema por lotes procesa datos rutinarios sin que haya un usuario interactivo presente. El procesamiento de reclamaciones en una compañía de seguros o los informes de ventas de una cadena de tiendas generalmente se realizan por lotes. Los sistemas de procesamiento de transacciones atienden gran número de pequeñas peticiones, como por ejemplo, en el procesamiento de cheques en un banco o en la reserva de pasajes aéreos. Cada unidad de trabajo es pequeña, pero el sistema debe atender cientos o miles de ellas por segundo. Los sistemas de tiempo compartido permiten a múltiples usuarios remotos ejecutar trabajos en el ordenador de forma simultánea, tales como la consulta de una gran base de datos. Estas funciones están íntimamente relacionadas; muchos sistemas operativos de mainframe las realizan todas. Un ejemplo de sistema operativo de mainframe es el OS/390, un descendiente del OS/360.

1.3.2 Sistemas Operativos de Servidor

Un nivel más abajo están los sistemas operativos de servidor. Éstos se ejecutan en servidores, que son ordenadores personales muy grandes, o estaciones de trabajo o incluso mainframes. Dan servicio a múltiples usuarios a través de una red, permitiéndoles compartir recursos de hardware y software. Los servidores pueden prestar servicios de impresión, servicios de ficheros o servicios web. Los proveedores de Internet tienen en funcionamiento muchas máquinas servidoras para dar soporte a sus clientes, y los sitios web utilizan esos servidores para almacenar las páginas web y atender las peticiones que les llegan. Entre los sistemas operativos de servidor típicos están UNIX y Windows 2000. Linux también está ganando terreno en los servidores.

1.3.3 Sistemas Operativos Multiprocesador

Una forma cada vez más común de obtener potencia de computación de primera línea es conectar varias CPUs en un mismo sistema. Dependiendo de la forma exacta de la conexión y de qué recursos se compartan, estos sistemas se llaman ordenadores paralelos, multicomputadores o multiprocesadores. Necesitan sistemas operativos especiales, pero con frecuencia éstos son variaciones de los sistemas operativos de servidor, con características especiales para la comunicación y su conectividad.

1.3.4 Sistemas Operativos de Ordenador Personal

La siguiente categoría es el sistema operativo de ordenador personal. Su cometido consiste en presentar una buena interfaz a un único usuario. Se les utiliza ampliamente para procesamiento de texto, hojas de cálculo y acceso a Internet. Ejemplos comunes son Windows 98, Windows 2000, el sistema operativo Macintosh y Linux. Los sistemas operativos de ordenador personal son tan conocidos que con toda seguridad no necesitan mucha presentación. De hecho, muchas personas ni siquiera saben que existen otros tipos de sistemas operativos.

1.3.5 Sistemas Operativos de Tiempo Real

Otro tipo de sistema operativo es el sistema de tiempo real. Estos sistemas se caracterizan por tener al tiempo como su principal parámetro. Por ejemplo, en los sistemas de control de procesos industriales, los ordenadores de tiempo real tienen que recoger datos acerca del proceso de producción y utilizarlos para controlar las máquinas de la fábrica. Con frecuencia existen ciertos plazos que deben cumplirse estrictamente. Por ejemplo, si un automóvil avanza en una línea de montaje, deben efectuarse ciertas acciones en ciertos instantes precisos. Si un robot soldador suelda demasiado pronto o demasiado tarde, el automóvil puede quedar arruinado. Si es absolutamente indispensable que la acción se efectúe en cierto momento (o dentro de cierto intervalo), tenemos un **sistema de tiempo real riguroso** (*hard real-time system*).

Otro tipo de sistema de tiempo real es el **sistema de tiempo real moderado** (*soft real-time system*), en el cual es aceptable dejar de cumplir ocasionalmente algún plazo. Los sistemas de audio digital o multimedia pertenecen a esta categoría. VxWorks y QNX son sistemas operativos de tiempo real muy conocidos.

1.3.6 Sistemas Operativos Empotrados

Continuando en nuestro descenso a sistemas cada vez más pequeños, llegamos a los ordenadores de bolsillo (*palmtop*) y sistemas empotrados. Un ordenador de bolsillo o **PDA** (*Personal Digital Assistant*; Asistente Personal Digital) es un pequeño ordenador que cabe en el bolsillo de la camisa y realiza unas cuantas funciones tales como agenda de direcciones

electrónica y bloc de notas. Los sistemas empotrados operan en los ordenadores que controlan dispositivos que por lo general no se consideran ordenadores, como televisores, hornos microondas y teléfonos móviles. Estos sistemas suelen tener algunas características de los sistemas de tiempo real, pero tienen además limitaciones de tamaño, memoria y consumo de electricidad que los hacen especiales. Algunos ejemplos de tales sistemas operativos son PalmOS y Windows CE (*Consumer Electronics*; Electrónica de Consumo).

1.3.7 Sistemas Operativos de Tarjeta Inteligente

Los sistemas operativos más pequeños se ejecutan en tarjetas inteligentes, que son dispositivos del tamaño de una tarjeta de crédito que contienen un chip de CPU. Sus limitaciones son muy severas en cuanto a potencia de procesamiento y memoria. Algunos de ellos sólo pueden desempeñar una función, como el pago electrónico, pero otros pueden realizar varias funciones en la misma tarjeta inteligente. A menudo se trata de sistemas patentados.

Algunas tarjetas inteligentes están orientadas a Java. Eso quiere decir que la ROM de la tarjeta inteligente contiene un intérprete de la Máquina Virtual de Java (JVM). Los *applets* (pequeños programas) de Java se descargan a la tarjeta y son interpretados por el intérprete JVM. Algunas de estas tarjetas pueden tratar varios applets al mismo tiempo, lo que conduce a la multiprogramación y a la necesidad de planificarlos. La gestión de los recursos y su protección es también una cuestión importante cuando dos o más applets están presentes al mismo tiempo. El sistema operativo (por lo regular muy primitivo) presente en la tarjeta debe tratar de resolver estas cuestiones.

1.4 REVISIÓN DE ASPECTOS HARDWARE

Un sistema operativo está íntimamente relacionado con el hardware del ordenador sobre el que se ejecuta pues extiende el conjunto de instrucciones del ordenador y administra sus recursos. Para poder realizar su trabajo debe conocer muy bien el hardware, o al menos la apariencia que el hardware presenta al programador.

Conceptualmente, un ordenador personal sencillo puede ser abstraído mediante un modelo parecido al de la Figura 1-5. La CPU, la memoria y los dispositivos de E/S están todos conectados por el bus del sistema y se comunican entre sí a través de él. Los ordenadores personales modernos tienen una estructura más complicada en la que intervienen varios buses, los cuales examinaremos más adelante. Por ahora, este modelo será suficiente. En las secciones que siguen analizaremos de forma somera estos componentes y examinaremos algunos de los aspectos del hardware que interesan a los diseñadores de los sistemas operativos.

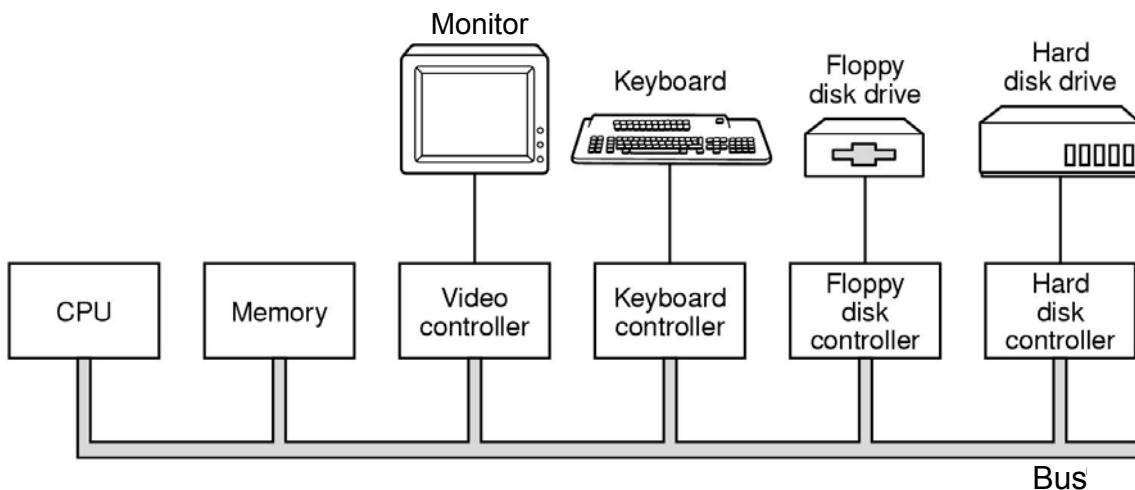


Figura 1-5. Algunos de los componentes de un ordenador personal sencillo.

1.4.1 Procesadores

El “cerebro” del ordenador es la CPU, la cual toma instrucciones de la memoria y las ejecuta. El ciclo básico de toda CPU consiste en tomar la primera instrucción de la memoria, decodificarla para determinar su tipo y operandos, ejecutarla, y luego tomar, decodificar y ejecutar las instrucciones subsiguientes. Es así como se ejecutan los programas.

Cada CPU ejecuta un repertorio de instrucciones específico. Por lo tanto, un Pentium no puede ejecutar programas para un SPARC, y un SPARC no puede ejecutar programas para un Pentium. Puesto que acceder a la memoria para extraer una instrucción o una palabra de datos tarda mucho más que la ejecución de una instrucción, todas las CPUs contienen algunos registros internos para guardar variables importantes y resultados temporales. El repertorio de instrucciones incluye por lo general instrucciones para cargar una palabra de la memoria en un registro, y para almacenar en la memoria una palabra que está en un registro. Otras instrucciones combinan dos operandos tomados de los registros, de la memoria o de ambos, para producir un resultado; por ejemplo, sumar dos palabras y almacenar el resultado en un registro o en la memoria.

Además de los registros generales que se utilizan para guardar variables y resultados temporales, casi todos los ordenadores tienen varios registros especiales que puede ver el programador. Uno de ellos es el **contador de programa**, que contiene la dirección de memoria

en la que está la siguiente instrucción que se va a extraer. Una vez extraída esa instrucción, el contador del programa se actualiza automáticamente para apuntar a la siguiente instrucción.

Otro registro es el **puntero de pila**, que apunta a la parte superior (cima) de la pila actual en la memoria. La pila contiene una trama (o registro de activación) por cada procedimiento al que se ha llamado pero del cual no se ha retornado todavía. La trama de pila de un procedimiento contiene los parámetros de entrada, las variables locales y variables temporales que no se guardan en registros.

Otro registro más es la **PSW (Program Status Word; palabra de estado del programa)** (también se le llama el **registro de estado** del procesador). Este registro contiene los bits de código de condición (también denominados indicadores o flags), que se activan cuando se ejecutan instrucciones de comparación, junto con la prioridad de ejecución de la CPU, el modo (usuario o supervisor (núcleo)) y otros bits de control. Los programas de usuario por lo general pueden leer la PSW entera, pero sólo pueden escribir en algunos de sus campos. La PSW desempeña un papel muy importante en las llamadas al sistema y la E/S.

El sistema operativo debe conocer todos los registros. Al multiplexar en el tiempo la CPU, es común que el sistema operativo tenga que detener el programa en ejecución para iniciar o continuar la ejecución de otro. Cada vez que el sistema operativo detiene un programa en ejecución, debe guardar todos los registros para que puedan restablecerse cuando el programa continúe su ejecución.

Con el fin de mejorar el rendimiento, los diseñadores de las CPUs abandonaron desde hace ya mucho tiempo el modelo según el cual simplemente se extrae, decodifica y ejecuta una instrucción a la vez. Muchas CPUs modernas cuentan con los recursos necesarios para ejecutar más de una instrucción al mismo tiempo. Por ejemplo, una CPU podría tener unidades individuales para extraer, decodificar y ejecutar, de manera que mientras esté ejecutando la instrucción n , también puede estar decodificando la instrucción $n+1$ y extrayendo la instrucción $n+2$. Tal organización se denomina **pipeline** (o segmentación encauzada) y se ilustra en la Figura 1-6(a) con un pipeline de tres etapas, aunque son comunes pipelines más largos. En casi todos los diseños de pipelines, una vez que una instrucción entra en el pipeline, debe ejecutarse necesariamente, aunque la instrucción anterior haya sido un salto condicional que haya dado lugar a una ruptura de secuencia. Los pipelines provocan grandes dolores de cabeza a quienes escriben compiladores y sistemas operativos, porque les obligan a tener en cuenta aspectos muy complejos de la máquina en cuestión.

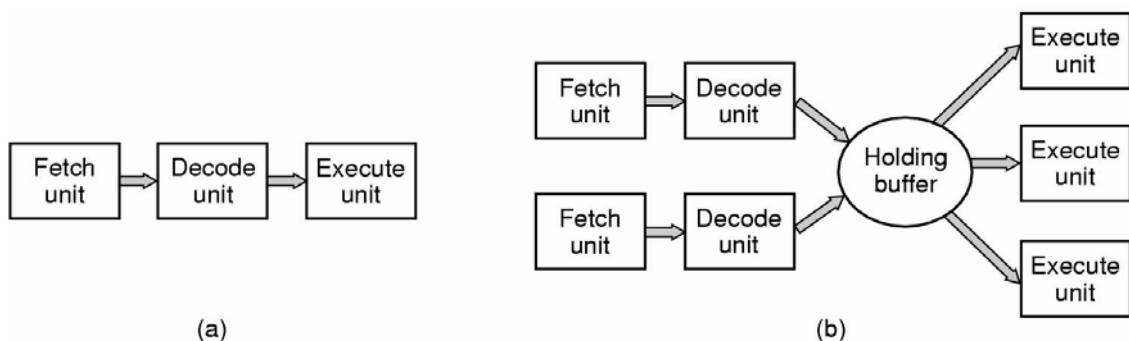


Figura 1-6. (a) Pipeline de tres etapas. (b) CPU superescalar.

Un diseño todavía más avanzado que el pipeline es una CPU **superescalar**, que se muestra en la Figura 1-6(b). Cuenta con varias unidades de ejecución, por ejemplo, una para aritmética de enteros, una para aritmética de punto flotante y una para operaciones booleanas.

Se extraen dos o más instrucciones a la vez, se decodifican y se dejan en un búfer de retención hasta que puedan ejecutarse. Cuando queda libre una unidad de ejecución, se busca en el búfer de retención una instrucción que pueda ejecutarse en ella y, si la hay, se la saca del búfer y se ejecuta. Una implicación de este diseño es que las instrucciones del programa a menudo se ejecutan desordenadas. En su mayor parte, corresponde al hardware asegurarse de que el resultado producido sea el mismo que se habría obtenido con una implementación secuencial, pero una buena parte de la complejidad se endosa al sistema operativo, como veremos.

La mayoría de las CPUs, salvo las más simples que se utilizan en los sistemas empotrados, tienen dos modos de operación: modo núcleo y modo usuario, como se mencionó antes. Por lo regular, un bit de la PSW controla el modo. Cuando la CPU opera en modo núcleo puede ejecutar cualquiera de las instrucciones que componen su repertorio de instrucciones y realizar todas las funciones del hardware. El sistema operativo se ejecuta en modo núcleo, y eso le permite acceder a todo el hardware.

En contraste, los programas de usuario se ejecutan en modo usuario, que sólo permite ejecutar un subconjunto del repertorio de instrucciones y tener acceso a un subconjunto de las funciones del hardware. En general, todas las instrucciones que implican E/S y protección de memoria están deshabilitadas en modo usuario. Desde luego, también está prohibido cambiar el bit de modo de la PSW para pasar de modo usuario a modo núcleo.

Para obtener algún servicio del sistema operativo, el programa de usuario debe hacer una **llamada al sistema**, la cual realiza un trap dentro del núcleo e invoca al sistema operativo. La instrucción TRAP cambia de modo usuario a modo núcleo y cede el control al sistema operativo. Una vez completado el trabajo solicitado al sistema operativo, se devuelve el control al programa de usuario justo en la instrucción inmediatamente siguiente a la llamada al sistema. Explicaremos los detalles del proceso de llamada al sistema más adelante en este capítulo. Como nota tipográfica, utilizaremos el tipo de letra Arial minúscula para indicar llamadas al sistema en el texto normal, como por ejemplo: `read`.

Vale la pena señalar que los ordenadores tienen otros traps (interrupciones y excepciones) además de las instrucciones para ejecutar una llamada al sistema (denominadas a veces interrupciones software). La mayoría de los demás traps están provocados por el hardware para advertir de una situación excepcional, tales como un intento de división por cero o un underflow de coma flotante. En todos los casos, el sistema operativo toma el control y decide lo que hay que hacer a continuación. A veces es preciso abortar el programa retornando un código de error. En otras ocasiones puede ignorarse el error (por ejemplo, ante un underflow de una variable puede simplemente asignársele un 0). Finalmente, si el programa ha anunciado con antelación que quiere manejar ciertos tipos de condiciones, puede devolvérsele el control permitiéndole que intente resolver el problema por sí mismo.

1.4.2 Memoria

El segundo componente importante de cualquier ordenador es la memoria. Lo ideal sería que la memoria de un ordenador fuese extremadamente rápida (más rápida que la CPU ejecutando una instrucción, para que la CPU nunca se viese frenada en los accesos a memoria), abundantemente grande y muy barata. Ninguna tecnología actual satisface todos esos objetivos, por lo que se adopta un enfoque diferente. El sistema de memoria se construye mediante una jerarquía de capas, como se muestra en la Figura 1-7.

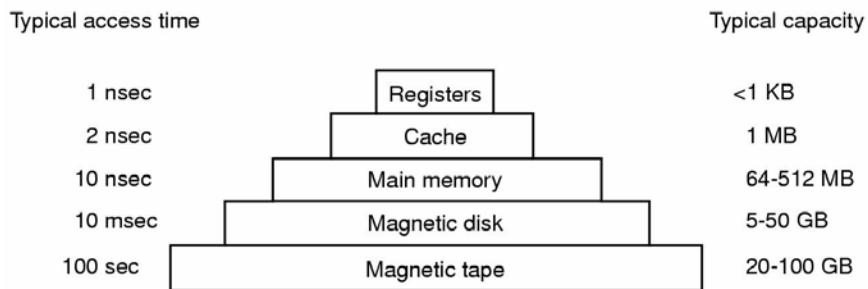


Figura 1-7. Una jerarquía usual de memoria. Las cifras son aproximaciones muy burdas.

La capa superior consiste en los registros internos de la CPU. Éstos se componen del mismo material que la CPU por lo que son tan rápidos como ella. Consecuentemente no se produce ningún retraso al acceder a ellos. La capacidad de almacenamiento de estos registros suele ser de 32×32 bits en una CPU de 32 bits, y de 64×64 bits en una CPU de 64 bits. En ambos casos, menos de 1 KB. Los programas deben administrar los registros (es decir, decidir qué se coloca en ellos) por su cuenta, mediante el software.

Luego viene la memoria caché, que en su mayor parte está bajo el control del hardware. La memoria principal se divide en **líneas de caché**, normalmente de 64 bytes, con las direcciones de 0 a 63 en la línea de caché 0, las direcciones 64 a 127 en la línea 1, etc. Las líneas de la caché de uso más frecuente se mantienen en una caché de alta velocidad situada dentro de la CPU o muy cerca de ella. Cuando el programa necesita leer una palabra de memoria, el hardware de la caché determina si la línea necesaria está o no en la caché. Si está, lo que constituye un **acuerdo de caché**, se atiende la petición desde la caché y no se envía ninguna petición por el bus a la memoria principal. Normalmente los aciertos de caché tardan en completarse alrededor de dos ciclos de reloj. Los fallos de caché implican acceder a la memoria, con una considerable pérdida de tiempo. El tamaño de la memoria caché está limitado por su elevado coste. Algunas máquinas tienen dos o incluso tres niveles de caché, cada uno más lento y más grande que el anterior.

A continuación viene la memoria principal. Éste es el caballo de batalla del sistema de memoria. La memoria principal se conoce también como la **RAM** (*Random Access Memory*; memoria de acceso aleatorio). Los veteranos de la informática a veces se refieren a ella como la memoria de núcleos (*core memory*) porque los ordenadores de las décadas de 1950 y 1960 utilizaban diminutos núcleos magnetizables de ferrita como memoria principal. Actualmente las memorias tienen decenas o cientos de megabytes y siguen creciendo con rapidez. Todas las peticiones de la CPU que no pueden atenderse desde la caché se dirigen a la memoria principal.

En el siguiente escalón de la jerarquía está el disco magnético (disco duro). El almacenamiento en disco es dos órdenes de magnitud más barato por bit que la RAM y también suele ser dos órdenes de magnitud más grande. El único problema es que el tiempo necesario para acceder aleatoriamente a los datos que contiene es casi tres órdenes de magnitud más grande. Esta velocidad tan baja se debe al hecho de que un disco es un dispositivo mecánico, como el mostrado en la Figura 1-8.

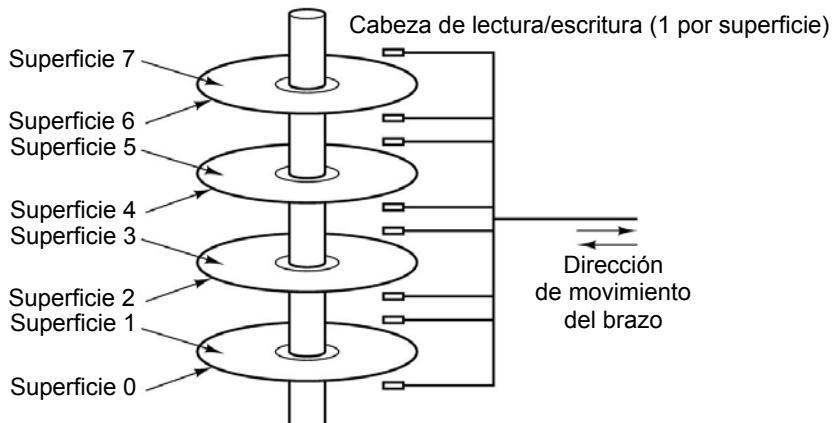


Figura 1-8. Estructura de una unidad de disco.

Un disco consta de uno o más platos de metal que giran continuamente a 5.400, 7.200 o 10.000 rpm. Un brazo mecánico pivota sobre los platos desde una esquina, como lo haría el brazo de un tocadiscos de 33 rpm para reproducir discos musicales de vinilo. La información se graba en el disco en una serie de circunferencias concéntricas. En cualquier posición del brazo, cada una de sus cabezas puede leer una región anular llamada **pista** (track). Juntas, todas las pistas que quedan bajo una posición dada del brazo constituyen lo que se denomina un **cilindro**.

Cada pista se divide en cierto número de sectores, que por lo general tienen 512 bytes cada uno. En los discos modernos, los sectores exteriores contienen más sectores que los interiores. Desplazar el brazo de un cilindro al siguiente tarda aproximadamente 1 ms; desplazarlo a un cilindro al azar suele tardar entre 5 y 10 ms, dependiendo de la unidad. Una vez que el brazo está en la pista correcta, la unidad deberá esperar hasta que la rotación del disco deje el sector requerido bajo la cabeza, lo que implica un retraso adicional de 5 a 10 ms, dependiendo de la velocidad de rotación de la unidad. Una vez que el sector está bajo la cabeza, la lectura o escritura se efectúa a razón de 5 MB/s en los discos más económicos, y hasta 160 MB/s en los más rápidos.

La última capa de la jerarquía de memoria corresponde a la cinta magnética. Éste medio suele utilizarse como un *backup* (respaldo o copia de seguridad) de la memoria de disco y para guardar conjuntos de datos muy grandes. Para tener acceso a una cinta, primero hay que colocarla en un lector de cintas, acción que puede realizar una persona o un robot (el manejo automatizado de las cintas es común en instalaciones que tienen bases de datos enormes). Luego podría ser necesario hacer avanzar la cinta hasta llegar al bloque solicitado. En total, esto podría tardar minutos. La gran ventaja de la cinta es que es extremadamente barata y removible, lo cual es importante en el caso de cintas de *backup* que deben guardarse en otro lugar para que sobrevivan a incendios, inundaciones, terremotos, etc.

La jerarquía de memoria que hemos descrito es típica pero algunas instalaciones no tienen todas las capas o tienen algunas capas distintas (como el disco óptico). No obstante, en todas ellas, conforme se baja en la jerarquía aumenta de forma drástica el tiempo de acceso aleatorio, la capacidad se incrementa de forma igual de drástica y el coste por bit baja enormemente. Por ello, es probable que las jerarquías de memoria persistan aún durante muchos años.

Además de los tipos de memoria mencionados, muchos ordenadores tienen una pequeña cantidad de memoria de acceso aleatorio no volátil. A diferencia de la RAM, la memoria no volátil no pierde su contenido cuando se corta el suministro de electricidad. La **ROM** (*Read Only Memory*; memoria de sólo lectura) se programa en la fábrica y no puede modificarse

posteriormente. La ROM es rápida y económica. En algunos ordenadores el programa de arranque del ordenador está almacenado en ROM. Además, algunas tarjetas de E/S llevan incorporada su propia ROM con rutinas que se encargan del control del dispositivo a bajo nivel.

La **EEPROM** (*Electrically Erasable Programmable ROM*; ROM borrable y programable eléctricamente) y la **flash RAM** tampoco son volátiles, pero en contraste con la ROM, su contenido puede borrarse y volver a escribirse. Sin embargo, su escritura tarda varios órdenes de magnitud más que la escritura en RAM, por lo que se usan de la misma manera que la ROM, con la única diferencia de que en su caso es posible corregir errores en los programas que contienen, y reescribirlos en el mismo lugar donde se encuentran.

Un tipo más de memoria es la CMOS, que es volátil. Muchos ordenadores emplean memoria CMOS para guardar la fecha y hora actuales. La memoria CMOS y el circuito de reloj que incrementa sus contenidos se alimentan con una pequeña batería para que la hora se siga actualizando de forma correcta aunque el ordenador esté apagado. La memoria CMOS también puede guardar los parámetros de configuración, entre los que está la unidad de disco desde la que se debe arrancar. Se utiliza CMOS porque consume tan poca electricidad que la batería original instalada en la fábrica puede durar varios años. No obstante, cuando la batería empieza a fallar, el ordenador comenzará a comportarse como si padeciese de Alzheimer, olvidando cosas que ha conocido desde hace años, como la unidad desde la cual debe arrancar.

Por el momento vamos a concentrarnos en la memoria principal. A menudo es conveniente mantener varios programas en la memoria al mismo tiempo. Si un programa se bloquea mientras espera a que termine una lectura del disco, otro programa podría pasar a utilizar la CPU, mejorando así su utilización. Sin embargo, con dos o más programas simultáneamente cargados en la memoria, deben resolverse los siguientes dos problemas:

1. Cómo proteger a un programa de los otros, y cómo proteger al núcleo del sistema operativo frente a todos ellos.
2. Cómo llevar a cabo la reubicación de los programas.

Son posibles muchas soluciones, pero todas exigen equipar a la CPU con hardware especial.

El primer problema es obvio, pero el segundo es un poco más sutil. Cuando se compila y enlaza un programa, el compilador y el enlazador no saben en qué parte de la memoria física se cargarán cuando se ejecute. Por esta razón, usualmente suponen en principio que el programa comenzará en la dirección 0 y colocan allí su primera instrucción. Supongamos que la primera instrucción toma de la memoria la palabra que está en la dirección 10.000. Supongamos ahora que el programa entero y los datos se cargan a partir de la dirección 50.000. Cuando se ejecute la primera instrucción, funcionará mal porque hará referencia a la palabra que está en la dirección 10.000 en vez de a la que está en la 60.000. Para resolver este problema, hay que reubicar el programa en el momento de su carga, localizando todas las direcciones y modificándolas (lo cual es factible, pero costoso), o bien efectuar la reubicación sobre la marcha (*on-the-fly*), en el momento de la ejecución.

La solución más sencilla se muestra en la Figura 1-9(a). En esta figura vemos un ordenador equipado con dos registros especiales, el **registro de base** y el **registro de límite**. (En este libro, los números que comienzan con 0x están en hexadecimal, siguiendo el convenio empleado en el lenguaje C. Similarmente, los números que comienzan por un cero a la izquierda están en octal.) Cuando se ejecuta un programa, el registro de base se ajusta de modo que apunte al principio del código del programa, mientras que el registro de límite indica cuánto espacio ocupa en total el código del programa y sus datos. Cuando hay que extraer una instrucción, el hardware comprueba si el contador de programa tiene un valor menor que el registro de límite y, de ser así, suma el valor del contador de programa al del registro base y envía la suma a la

memoria. De forma similar, cuando el programa quiere tomar una palabra de datos (digamos, de la dirección 10.000), el hardware suma de manera automática el contenido del registro de base (en nuestro caso 50.000) a esa dirección y envía la suma (60.000) a la memoria. El registro de base impide que un programa haga referencia a cualquier parte de la memoria por debajo de la posición donde está almacenado. Además, el registro de límite impide referenciar cualquier parte de la memoria que esté por encima del programa. Así, este esquema resuelve tanto el problema de protección como el de reubicación a costa de añadir dos nuevos registros a la CPU y de aumentar ligeramente el tiempo de ciclo (para efectuar la comprobación del límite y la suma del contenido del registro de base).

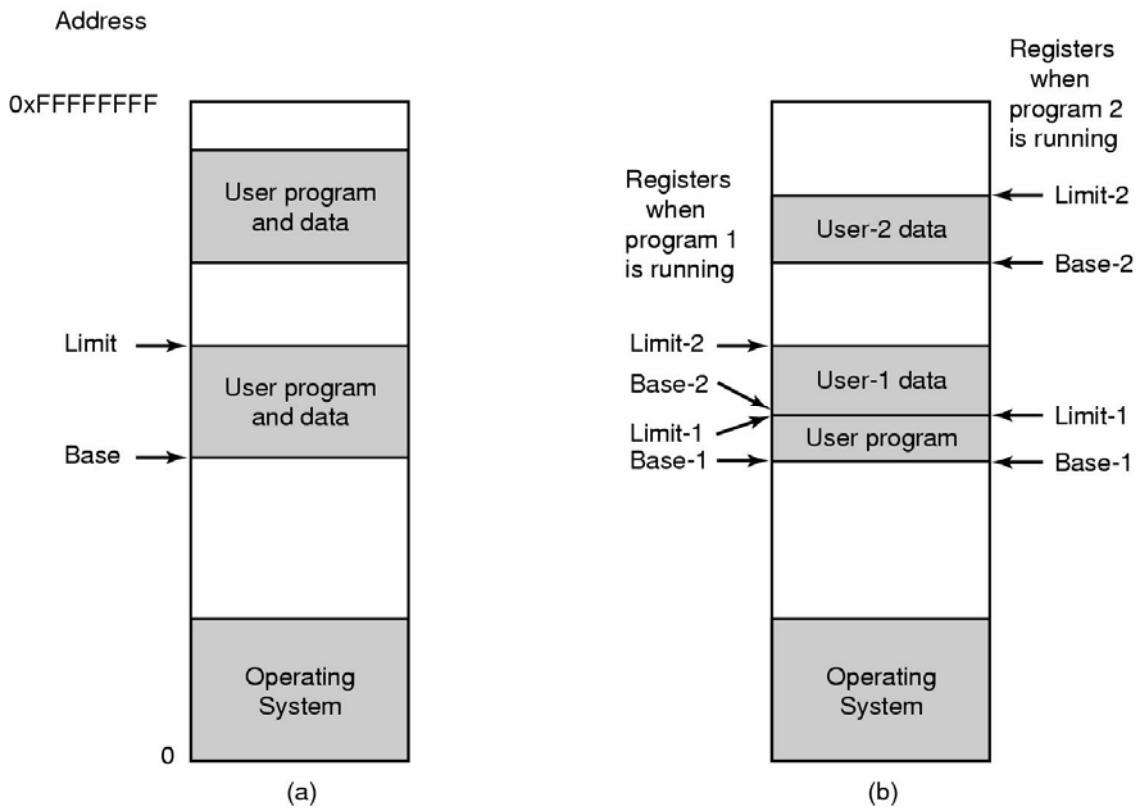


Figura 1-9. (a) Uso de un par base-límite. El programa puede acceder a la memoria entre la base y el límite. (b) Uso de dos pares base-límite. El código del programa está entre Base-1 y Límite-1, mientras que los datos están entre Base-2 y Límite-2.

La comprobación y transformación anteriores convierten la dirección generada por el programa, llamada **dirección virtual**, en una dirección utilizada por la memoria, llamada **dirección física**. El dispositivo que se encarga de la comprobación y transformación es la **MMU (Memory Management Unit; unidad de gestión de memoria)**, que está situada dentro del chip de la CPU o cerca de él, pero que desde un punto de vista lógico está entre la CPU y la memoria.

En la Figura 1-9(b) se ilustra una MMU más sofisticada. Dicha MMU cuenta con dos pares de registros de base y de límite, un par para el código del programa y otro par para los datos. El contador de programa y todas las demás referencias al código del programa utilizan el par 1, y las referencias a datos utilizan el par 2. En consecuencia, ahora es posible que varios usuarios compartan el mismo programa, manteniendo una única copia de él en la memoria, algo que era imposible con el primer esquema. Cuando se está ejecutando el programa 1, los cuatro registros se establecen como indican las flechas a la izquierda de la Figura 1-9(b). Cuando se está ejecutando el programa 2, se ajustan como indican las flechas a la derecha de esa figura.

Existen MMUs mucho más sofisticadas. Estudiaremos algunas de ellas en capítulos posteriores. Lo que debe quedar claro aquí es que la gestión de la MMU es una función exclusiva del sistema operativo, ya que no puede esperarse que los usuarios hagan por sí mismos la gestión de forma correcta.

Hay dos aspectos del sistema de memoria que tienen un mayor impacto sobre el rendimiento. En primer lugar, las cachés ocultan la velocidad relativamente baja de la memoria. Cuando un programa ha estado ejecutándose durante cierto tiempo, la caché está llena de líneas de caché de ese programa, obteniéndose un buen rendimiento. Sin embargo, cuando el sistema operativo conmuta de un programa a otro, la caché sigue estando llena de las líneas de caché del primer programa. Las líneas de caché que el nuevo programa necesita tendrán que cargarse una a una desde la memoria física. Si ocurre con demasiada frecuencia, esta operación puede afectar muy negativamente al rendimiento.

En segundo lugar, cuando se conmuta de un programa a otro, es preciso volver a establecer los registros de la MMU. En la Figura 1-9(b) sólo hay que restablecer cuatro registros, lo cual no significa ningún problema, pero en las MMUs reales es preciso volver a cargar muchos más registros, bien de forma explícita o dinámicamente, según sea necesario. De cualquier modo, todo eso requiere su tiempo. La moraleja es que conmutar de un programa a otro, lo que se conoce como **cambio de contexto**, resulta una operación muy costosa.

1.4.3 Dispositivos de E/S

La memoria no es el único recurso que debe administrar el sistema operativo. Los dispositivos de E/S también interactúan intensamente con él. Como vimos en la Figura 1-5, los dispositivos de E/S constan generalmente de dos partes: un controlador de dispositivo y el dispositivo en sí. El **controlador del dispositivo** es un chip o un conjunto de chips montados en una tarjeta insertable (denominada **tarjeta controladora**) que controla físicamente el dispositivo. Dicho controlador acepta comandos del sistema operativo, como por ejemplo leer datos del dispositivo, y los ejecuta.

En muchos casos, el control real del dispositivo es muy complicado y detallado, por lo que es tarea del controlador presentar una interfaz más sencilla al sistema operativo. Por ejemplo, un controlador de disco podría aceptar un comando para leer el sector 11.206 del disco 2. El controlador tiene entonces que convertir este número de sector lineal en un sector, cilindro y cabeza. Esta conversión podría complicarse por el hecho de que los cilindros exteriores tienen más sectores que los interiores, y que algunos sectores defectuosos se han redirigido hacia otros sectores. A continuación el controlador tiene que determinar en qué cilindro está actualmente el brazo y enviarle una secuencia de pulsos para desplazarlo hacia adentro o hacia afuera el número de cilindros requerido. Luego el controlador tiene que esperar a que el sector correcto haya rotado hasta quedar debajo de la cabeza, antes de comenzar a leerlo y almacenar los bits a medida que salen de la unidad, a la vez que elimina el preámbulo del sector y calcula su suma de verificación (*checksum*). Finalmente, el controlador ensambla los bits que le llegan, para formar palabras y guardarlas en la memoria. Para hacer todo ese trabajo, es común que los controladores contengan pequeños ordenadores empotrados programados convenientemente para realizar su trabajo.

El otro componente es el dispositivo en sí. Los dispositivos tienen interfaces relativamente simples, debido a que no hacen cosas complicadas y para poder estandarizarse. Lo último es necesario para que cualquier controladora de disco IDE pueda controlar cualquier disco IDE, por ejemplo. **IDE** son las siglas en inglés de **Integrated Drive Electronics** (electrónica integrada en la unidad) y es el tipo de disco estándar en el Pentium y en algunos otros ordenadores. Puesto que la interfaz real con el dispositivo está oculta tras el controlador, lo

único que ve el sistema operativo es la interfaz con el controlador, que podría ser muy diferente de la interfaz con el dispositivo.

Puesto que cada tipo de controlador es distinto, se necesita diferente software para controlar cada uno. El software que se comunica con un controlador, enviándole comandos y aceptando sus respuestas, se denomina **controlador** (software) **del dispositivo** o **driver del dispositivo**. Para evitar confusiones entre el controlador (software) y el controlador (hardware) del dispositivo, nos referiremos en lo sucesivo al controlador (software) del dispositivo como el driver del dispositivo. En otros libros se evita esa ambigüedad simplemente refiriéndose (en femenino) al controlador (hardware) como la (tarjeta) controladora del dispositivo. Los fabricantes de controladores de dispositivos tienen que proporcionar los drivers del dispositivo para cada sistema operativo que lo soporte. De esta manera un escáner podría venir con drivers para Windows 98, Windows 2000 y UNIX, por ejemplo.

Antes de poder utilizar el dispositivo, es necesario incluir su driver en el sistema operativo para que pueda ejecutarse en modo núcleo. Teóricamente los drivers también podrían ejecutarse fuera del núcleo, pero pocos sistemas operativos actuales soportan esta posibilidad debido a que requiere la capacidad para permitir que un driver en el espacio de usuario pueda tener acceso al dispositivo de forma controlada, característica raramente soportada. Hay tres maneras de situar el driver dentro del núcleo. La primera consiste en volver a enlazar el núcleo con el nuevo driver y luego reiniciar el sistema. Muchos sistemas UNIX funcionan así. La segunda manera consiste en incluir una entrada en un fichero del sistema operativo para indicarle a éste que necesita el driver del dispositivo, y luego reiniciar el sistema. En el momento del arranque, el sistema operativo procede a buscar los drivers que necesita y los carga. Windows funciona así. La tercera manera es que el sistema operativo pueda aceptar nuevos drivers mientras se está ejecutando y los instale sobre la marcha sin tener que reiniciar el ordenador. Esta manera de integrar el driver se desarrolló para situaciones raras pero actualmente se ha convertido en algo habitual. Los dispositivos que permiten su conexión en caliente, tales como los dispositivos USB e IEEE 1394 (que trataremos más adelante) siempre necesitan drivers que se cargan dinámicamente de esta manera.

Todo controlador cuenta con un pequeño número de registros que sirven para comunicarse con él. Por ejemplo, un controlador de disco en su versión más sencilla podría tener registros para especificar la dirección en disco, la dirección en memoria, el número de sectores y el sentido de la transferencia (lectura o escritura). Para activar el controlador, el driver recibe un comando del sistema operativo y lo traduce a los valores apropiados que debe escribir en los registros del dispositivo.

En algunos ordenadores, los registros del dispositivo están mapeados en el espacio de direcciones del sistema operativo, de modo que pueden leerse y escribirse como si fueran palabras de memoria ordinarias. En tales ordenadores no se necesitan instrucciones de E/S especiales y es posible proteger el hardware del acceso indiscriminado por parte de los programas de usuario simplemente colocando fuera de su alcance esas direcciones de memoria (por ejemplo, utilizando registros de base y de límite). En otros ordenadores, los registros de dispositivo se colocan en un espacio de puertos de E/S especial, con cada registro teniendo una dirección de puerto. En estas máquinas se dispone en modo núcleo de instrucciones especiales IN y OUT que permiten a los drivers leer y escribir en los registros. El primer esquema elimina la necesidad de instrucciones de E/S especiales pero mantiene permanentemente ocupada alguna parte del espacio de direcciones. El segundo esquema no ocupa el espacio de direcciones para nada pero requiere instrucciones especiales en el repertorio de instrucciones del lenguaje máquina. Ambos esquemas se utilizan ampliamente.

Las operaciones de entrada y salida pueden realizarse de tres maneras distintas. En el método más sencillo, un programa de usuario realiza una llamada al sistema, que el núcleo traduce en una llamada a un procedimiento del driver apropiado. El driver pone en marcha

entonces la E/S y entra en un bucle de espera que consulta continuamente el dispositivo para ver si ya terminó (es usual que haya un bit que indique si el dispositivo sigue ocupado o no). Una vez terminada la E/S, el driver coloca los datos (si los hay) donde se necesitan y retorna. El sistema operativo devuelve entonces el control al programa que lo invocó a través de la llamada al sistema. Este método se denomina **espera activa** (*busy waiting* o *polling*) y tiene la desventaja de mantener ocupada a la CPU consultando el estado del dispositivo hasta que termina la E/S.

El segundo método consiste en que el driver pone en marcha el dispositivo y lo programa para que genere una interrupción cuando haya terminado. En ese momento el driver retorna devolviendo el control al sistema operativo. Entonces el sistema operativo si es necesario bloquea al programa que hizo la llamada al sistema y busca otras cosas útiles que hacer. Cuando el controlador (hardware) del dispositivo detecta el final de la transferencia, genera una **interrupción** para avisar de su terminación.

Las interrupciones son muy importantes en los sistemas operativos, por lo que vamos a examinar la idea con más detenimiento. En la Figura 1-10(a) vemos los tres pasos para realizar una E/S. En el paso 1, el driver del dispositivo le dice al controlador del disco lo que debe hacer escribiendo en sus registros de dispositivo. A continuación, el controlador pone en marcha el dispositivo. Cuando el controlador termina de leer o escribir el número de bytes que se le pidió transferir, avisa al controlador de interrupciones utilizando ciertas líneas del bus (paso 2). Si el controlador de interrupciones está preparado para aceptar la interrupción (pues podría no estarlo si está ocupado con una interrupción de mayor prioridad), activa una línea de la CPU para informarle de la interrupción (paso 3). En el paso 4, el controlador de interrupciones vuelca el número del dispositivo al bus para que la CPU pueda leerlo y sepa qué dispositivo acaba de terminar (podrían estar operando muchos dispositivos al mismo tiempo).

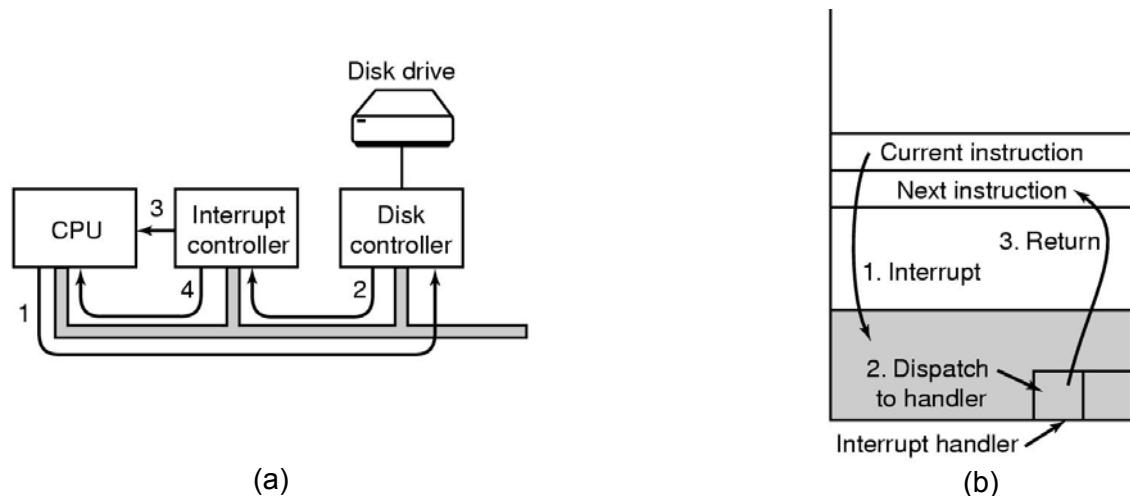


Figura 1-10. (a) Pasos desde que se pone en marcha un dispositivo de E/S hasta que llega la interrupción. (b) El tratamiento de las interrupciones implica aceptar la interrupción, ejecutar la rutina de tratamiento de la interrupción y retornar al programa de usuario.

Cuando la CPU decide aceptar la interrupción, el contador de programa y la PSW se salvan normalmente en la pila actual y la CPU pasa a modo núcleo. El número del dispositivo que interrumpe se lee del bus y puede utilizarse como índice de una parte de la memoria para encontrar la dirección de la rutina de tratamiento (o manejador) de la interrupción correspondiente a ese dispositivo. Esa parte de la memoria se denomina **la tabla de vectores de interrupción**. Una vez que comienza la rutina de tratamiento de la interrupción (que es parte del driver del dispositivo que interrumpió), saca el contador de programa y la PSW de la pila, los

guarda y consulta el dispositivo para saber cuál es su estado. Cuando termina la rutina de tratamiento, devuelve el control al programa de usuario que se estaba ejecutando, justo en la siguiente instrucción a la que había ejecutado anteriormente. Estos pasos se muestran en la Figura 1-10(b).

El tercer método para realizar la E/S utiliza un chip especial de **DMA** (*Direct Memory Access; acceso directo a memoria*) que puede controlar el flujo de bits entre la memoria y algún controlador de dispositivo sin que la CPU tenga que intervenir constantemente. La CPU programa el chip de DMA, indicándole cuántos bytes hay que transferir, el dispositivo, las direcciones de memoria en cuestión y el sentido, para a continuación desentenderse de él. Cuando el chip de DMA termina, provoca una interrupción que se trata como acabamos de describir. El hardware del DMA y de E/S, en general, se tratarán con mayor detalle en el capítulo 5.

Con frecuencia las interrupciones se presentan en momentos en que sería desastroso atenderlas, por ejemplo, mientras se está ejecutando una rutina de tratamiento de otra interrupción. Por ese motivo, la CPU cuenta con un mecanismo para inhibir las interrupciones y para volver a habilitarlas después. Mientras que las interrupciones están inhibidas, cualquier dispositivo que termine seguirá aplicando su señal de interrupción, pero la CPU no se interrumpirá en tanto no se habiliten nuevamente las interrupciones. Si varios dispositivos terminaron mientras las interrupciones estaban inhibidas, el controlador de interrupciones decidirá a cuál se atenderá primero, basándose normalmente en prioridades estáticas asignadas a cada dispositivo. El dispositivo de mayor prioridad gana.

1.4.4 Buses

La organización de la Figura 1-5 se utilizó durante años en los miniordenadores y también en el PC original de IBM. Sin embargo, a medida que aumentó la rapidez de los procesadores y de las memorias, la capacidad de un único bus (y ciertamente la del bus del PC de IBM) para manejar todo el tráfico se exprimió hasta el límite. Tenía que hacerse algo. Como resultado, se añadieron buses adicionales, tanto para los dispositivos de E/S más rápidos como para el tráfico entre la CPU y la memoria. Como consecuencia de esa evolución, un sistema Pentium grande tiene en la actualidad un aspecto parecido al que se muestra en la Figura 1-11.

Este sistema tiene ocho buses (caché, local, memoria, PCI, SCSI, USB, IDE e ISA), cada uno con una velocidad de transferencia y función diferente. El sistema operativo debe estar al tanto de todos ellos para configurarlos y gestionarlos. Los dos principales buses son el bus **ISA** (*Industry Standard Architecture; Arquitectura Estándar de la Industria*) original del PC de IBM y su sucesor, el bus **PCI** (*Peripheral Component Interface; Interfaz de Componentes Periféricos*). El bus ISA, que fue originalmente el bus del PC/AT de IBM, opera a 8,33 MHz y puede transferir dos bytes a la vez, para dar una velocidad máxima de 16,67 MB/s. Se incluyó para mantener la compatibilidad con las tarjetas de E/S antiguas y lentas. El bus PCI fue inventado por Intel como un sucesor del bus ISA. Puede operar a 66 MHz y transferir 8 bytes a la vez, para dar una tasa de datos de 528 MB/s. La mayoría de los dispositivos de E/S de alta velocidad utilizan ahora el bus PCI. Incluso algunos ordenadores que no son de Intel utilizan el bus PCI, debido al gran número de tarjetas de E/S disponibles para ese bus.

En esta configuración, la CPU se comunica por el bus local con el chip puente (*bridge*) PCI, y el chip puente PCI se comunica con la memoria a través de un bus de memoria dedicado, que a menudo opera a 100 MHz. Los sistemas Pentium tienen una caché de nivel 1 en el chip, y una caché de nivel 2 mucho más grande fuera del chip, conectada a la CPU por el bus de caché.

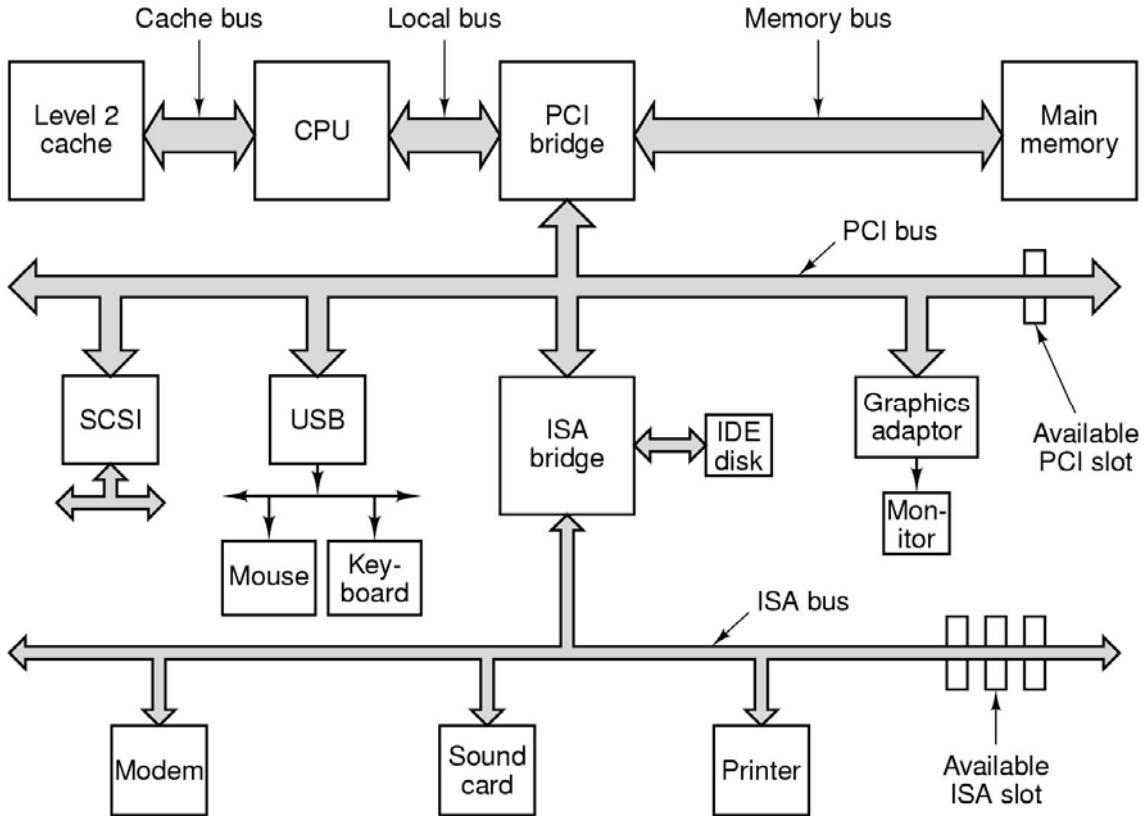


Figura 1-11. Estructura de un sistema Pentium grande.

Además, este sistema contiene tres buses especializados: IDE, USB y SCSI. El bus IDE permite conectar dispositivos periféricos tales como discos y unidades de CD-ROM al sistema. El bus IDE es una expansión de la interfaz controladora de disco en el PC/AT y es ahora estándar en casi todos los sistemas basados en el Pentium para el disco duro y a menudo para el CD-ROM.

El **USB** (*Universal Serial Bus*; Bus Serie Universal) se inventó para conectar al ordenador todos los dispositivos de E/S lentos, tales como el teclado y el ratón. Este bus utiliza un conector pequeño con cuatro contactos, dos de los cuales suministran energía eléctrica a los dispositivos USB. El USB es un bus centralizado en el que un dispositivo raíz consulta a los dispositivos de E/S cada milisegundo para ver si tienen algún tráfico. El bus puede manejar una carga agregada de 1,5 MB/s. Todos los dispositivos USB comparten un único driver de dispositivo USB, lo que hace innecesario instalar un nuevo driver para cada dispositivo USB nuevo. Consecuentemente, es posible añadir al sistema nuevos dispositivos USB sin tener que reiniciar el ordenador.

El bus **SCSI** (*Small Computer System Interface*; pequeña interfaz para sistemas de ordenador) es un bus de alto rendimiento diseñado para discos rápidos, escáneres y otros dispositivos que necesitan un considerable ancho de banda. Puede operar hasta a 160 MB/s. Ha estado presente en los sistemas Macintosh desde que se inventaron, siendo también populares en sistemas UNIX y en algunos sistemas basados en Intel.

Otro bus más (que no se muestra en la Figura 1-11) es el **IEEE 1394**, también conocido como FireWire, aunque estrictamente hablando FireWire es el nombre que Apple utiliza para su implementación del 1394. Al igual que el USB, el bus IEEE 1394 transmite bits en serie pero está diseñado para transferir paquetes a velocidades de hasta 50 MB/s, lo que lo hace muy útil para conectar al ordenador cámaras de vídeo digitales y dispositivos multimedia similares. A

diferencia del USB, el IEEE 1394 no tiene un controlador central. SCSI e IEEE 1394 actualmente se enfrentan a la competencia de una versión más rápida del USB que está siendo desarrollada (USB 2.0).

Para operar en un entorno como el de la Figura 1-11, el sistema operativo tiene que saber qué dispositivos hay y configurarlos. Esta necesidad llevó a Intel y a Macintosh a diseñar un sistema para el PC llamado **plug and play** (conectar y usar), basado en un concepto similar que se implementó por primera vez en el Apple Macintosh. Antes del plug and play, cada tarjeta de E/S tenía un nivel de petición de interrupción fijo y direcciones fijas para sus registros de E/S. Por ejemplo, el teclado tenía la interrupción 1 y utilizaba las direcciones de E/S de 0x60 a 0x64, el controlador de la disquetera tenía la interrupción 6 y utilizaba las direcciones de E/S de 0x3F0 a 0x3F7, la impresora tenía la interrupción 7 y utilizaba las direcciones de E/S de 0x378 a 0x37A, y lo mismo sucedía para otros dispositivos.

Hasta ahí, todo iba bien. Los problemas se presentaban cuando el usuario compraba una tarjeta de sonido y una tarjeta módem y resultaba que ambas utilizaban, digamos, la interrupción 4. Por lo tanto se presentaba un conflicto entre las dos tarjetas que les impedía operar juntas. La solución fue incluir microinterruptores DIP o puentes (jumpers) en cada tarjeta de E/S y explicar al usuario cómo ajustarlos para seleccionar un nivel de interrupción y unas direcciones de los dispositivos de E/S que no entraran en conflicto con cualesquiera otras del sistema del usuario. Los adolescentes que dedicaban su vida a los vericuetos del hardware del PC podían a veces hacer eso sin cometer errores. Desafortunadamente, nadie más era capaz, por lo que el resultado final era el caos.

El plug and play hace posible que el sistema pueda recoger automáticamente información sobre los dispositivos de E/S, asignando de forma centralizada los niveles de interrupción y las direcciones de E/S, para luego comunicar a cada tarjeta los valores concretos que le corresponden. Muy brevemente, esto funciona como sigue en el Pentium. Todo Pentium tiene una placa madre (*motherboard*) con un programa llamado el sistema **BIOS** (*Basic Input Output System*; Sistema Básico de Entrada/Salida). El BIOS contiene software de E/S de bajo nivel, incluyendo procedimientos para leer del teclado, escribir en la pantalla y realizar E/S de disco, entre otras cosas. Hoy en día, el BIOS reside en RAM de tipo flash, que no es volátil pero que puede ser actualizada por el sistema operativo cuando se detecten errores en el BIOS.

Cuando arranca el ordenador, comienza a ejecutarse el BIOS. Lo primero que hace es determinar cuanta RAM está instalada, y comprobar si el teclado y otros dispositivos básicos están instalados y responden correctamente. A continuación comienza a explorar los buses ISA y PCI para detectar todos los dispositivos conectados a ellos. Algunos de esos dispositivos suelen ser **heredados** (*legacy*, es decir, diseñados antes de inventarse el plug and play) y tienen niveles de interrupción y direcciones de E/S fijos (posiblemente establecidos por interruptores o puentes en la tarjeta de E/S, pero no por el sistema operativo). Estos dispositivos se registran, así como los de tipo plug and play. Si los dispositivos presentes no son los mismos que había la última vez que se arrancó el sistema, se configuran los nuevos dispositivos encontrados.

Luego el BIOS determina el dispositivo de arranque probando con una lista de dispositivos almacenados en la memoria CMOS. El usuario puede alterar esa lista entrando en el programa de configuración (setup) del BIOS inmediatamente después del arranque. Normalmente, se intenta arrancar desde un disquete metido en la disquetera. Si eso falla, se prueba con el CD-ROM. Si no hay un disquete ni un CD-ROM, el sistema se arranca desde el disco duro. Se lee el primer sector del dispositivo de arranque, se almacena en la memoria y se ejecuta. Este sector contiene un programa que normalmente examina la tabla de particiones al final del sector de arranque para determinar cuál es la partición que está activa. Luego se lee un programa cargador de arranque secundario de esa partición. El cargador lee el sistema operativo de la partición activa y lo pone en marcha.

Después, el sistema operativo consulta el BIOS para obtener la información de configuración. Luego comprueba para cada dispositivo si dispone del driver correspondiente. Si falta algún driver, pide al usuario que inserte un disquete o un CD-ROM con el driver del dispositivo (proporcionado por el fabricante del dispositivo). Una vez que el sistema operativo tiene todos los drivers de dispositivo, los carga en el núcleo. A continuación inicializa sus tablas internas, crea todos los procesos de fondo (background) necesarios y arranca un programa de inicio de sesión (*login*) o GUI en cada terminal. Al menos, esa es la forma en que se supone que funciona. En la vida real, plug and play es tan poco fiable que muchos prefieren llamarlo plug and pray (conectar y rezar).

1.5 CONCEPTOS DE LOS SISTEMAS OPERATIVOS

Todos los sistemas operativos tienen ciertos conceptos básicos, tales como procesos, memoria y ficheros, que son fundamentales para entenderlos. En las siguientes secciones examinaremos algunos de estos conceptos básicos de forma breve a modo de introducción. Volveremos a tratar cada uno de ellos con mayor detalle posteriormente en este libro. Para ilustrar estos conceptos, utilizaremos de vez en cuando algunos ejemplos, casi siempre tomados de UNIX. Sin embargo, normalmente es posible encontrar ejemplos similares en otros sistemas.

1.5.1 Procesos

Un concepto clave en todos los sistemas operativos es el de **proceso**. Un proceso es básicamente un programa en ejecución. Todo proceso tiene asociado un **espacio de direcciones**, es decir una lista de posiciones de memoria desde algún mínimo (normalmente 0) hasta algún máximo, que el proceso puede leer y en las que puede escribir. El espacio de direcciones contiene el programa ejecutable, sus datos y su pila. Cada proceso tiene asociado también algún conjunto de registros, incluido el contador de programa, el puntero de pila y otros registros hardware, así como toda la demás información necesaria para ejecutar el programa.

Trataremos con mucho mayor detalle el concepto de proceso en el capítulo 2, pero por ahora la mejor forma de conseguir una buena percepción intuitiva de lo qué es un proceso es pensar en los sistemas de tiempo compartido. Periódicamente, el sistema operativo decide dejar de ejecutar un proceso y comenzar a ejecutar otro, por ejemplo, porque el primero ya ha recibido más de su porción de tiempo de CPU en el último segundo.

Cuando a un proceso se le suspende temporalmente como al anterior, posteriormente es necesario poder proseguir con su ejecución a partir de exactamente el mismo estado que tenía cuando se le suspendió. Eso significa que toda la información acerca del proceso debe guardarse de forma explícita en algún lado durante su suspensión. Por ejemplo, el proceso podría tener varios ficheros abiertos para su lectura de forma simultánea. Cada uno de esos ficheros tiene asociado un puntero que indica la posición actual (es decir, el número del byte o registro que se leerá a continuación). Cuando un proceso se suspende de manera temporal, todos esos punteros tienen que guardarse de forma que una llamada **read** que se ejecute después de reiniciado el proceso lea los datos correctos. En muchos sistemas operativos toda la información acerca de cada proceso, salvo el contenido de su propio espacio de direcciones, se guarda en una tabla del sistema operativo denominada la **tabla de procesos**, que es un array (o una lista enlazada) de registros (estructuras, según la terminología de C), uno por cada uno de los procesos existentes en ese momento.

Así pues, un proceso (suspendido) consiste en su espacio de direcciones, usualmente denominado como la **imagen del núcleo** (*core image*, recordando las memorias de núcleos magnéticos de ferrita utilizadas antaño), y su entrada en la tabla de procesos (denominada también **descriptor de proceso** o **bloque de control del proceso**), que contiene entre algunas otras cosas sus registros.

Las llamadas al sistema más importantes para la gestión de los procesos son las que se ocupan de la creación y terminación de los procesos. Consideremos un ejemplo típico en el que un proceso llamado el **intérprete de comandos** o **shell** lee comandos desde un terminal. Si el usuario teclea un comando solicitando la compilación de un programa, el shell debe crear un nuevo proceso que ejecute el compilador. Cuando ese proceso termine la compilación, ejecutará una llamada al sistema para terminarse a sí mismo.

Si un proceso puede crear uno o más procesos (llamados **procesos hijos**), y éstos a su vez pueden crear también procesos hijos, pronto llegaremos a una estructura de árbol de procesos como la de la Figura 1-12. Los procesos relacionados que están cooperando para llevar a cabo alguna tarea necesitan a menudo comunicarse entre sí y sincronizar sus actividades. Esta comunicación se denomina **comunicación entre procesos** (*interprocess communication*) y se tratará con detalle en el capítulo 2.

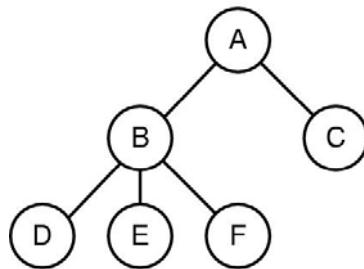


Figura 1-12. Un árbol de procesos. El proceso *A* creó dos procesos hijos, *B* y *C*. El proceso *B* creó tres procesos hijos, *D*, *E* y *F*.

Existen otras llamadas al sistema para solicitar más memoria (o liberar memoria que ya no se utiliza), para esperar a que un proceso hijo termine, y para sobrescribir (recubrir) el programa actual del proceso con un programa diferente.

En ocasiones surge la necesidad de comunicar información a un proceso en ejecución que no la está esperando. Por ejemplo, supongamos un proceso que está comunicándose con otro situado en un ordenador distinto mediante el envío de mensajes al proceso remoto a través de la red. Para protegerse contra la posibilidad de que se pierda un mensaje o su respuesta, el remitente podría solicitar que su propio sistema operativo le avise después de un cierto número de segundos, de forma que pueda retransmitir el mensaje si todavía no ha recibido acuse de su recepción por parte del destinatario. Una vez establecido ese temporizador (*timer*), el programa puede continuar realizando otras tareas.

Cuando transcurre el número de segundos especificado, el sistema operativo envía una **señal de alarma** al proceso. La señal provoca que el proceso suspenda temporalmente cualquier cosa que esté haciendo, guarde sus registros en la pila y comience a ejecutar un procedimiento especial de tratamiento de la señal, por ejemplo, para retransmitir un mensaje presumiblemente perdido. Cuando termina el tratamiento de la señal, el proceso en ejecución prosigue su ejecución desde el estado en el que estaba justo antes de recibir la señal. Las señales son el análogo software de las interrupciones hardware, y pueden generarse por diversas causas además de la expiración de los temporizadores. Muchas excepciones del sistema detectadas por el hardware, tales como la ejecución de una instrucción no permitida o el acceso a una dirección inválida, se convierten también en señales que se envían al proceso culpable.

El administrador del sistema asigna un **UID** (*User IDentification*; identificador de usuario) a cada persona autorizada para utilizar el sistema. Cada proceso que se crea tiene el UID de la persona que lo creó. Un proceso hijo tiene el mismo UID que su padre. Los usuarios pueden ser miembros de grupos, cada uno de los cuales tiene un **GID** (*Group IDentification*; identificador de grupo).

Un UID, denominado el **superusuario** (en UNIX), tiene derechos especiales y podría violar muchas de las reglas de protección. En las grandes instalaciones, sólo el administrador del sistema conoce la contraseña (*password*) necesaria para convertirse en superusuario, pero muchos de los usuarios ordinarios (especialmente estudiantes) dedican un considerable esfuerzo

Señales

a tratar de encontrar errores en el sistema que les permitan convertirse en superusuarios sin necesidad de la contraseña.

En el capítulo 2 estudiaremos los procesos, la comunicación entre procesos y cuestiones relacionadas con estos temas.

1.5.2 Interbloqueos

Cuando dos o más procesos están interactuando, a veces pueden llegar a una situación de estancamiento de la que no pueden salir. Tal situación se denomina un **interbloqueo** (*deadlock*).

La mejor manera de presentar los interbloqueos es con un ejemplo del mundo real que todos conocemos, un atasco de tráfico. Consideremos la situación de la Figura 1-13(a). Vemos cuatro autobuses que se están acercando a una intersección de dos calles. Detrás de cada uno vienen más (que no se muestran). Con un poco de mala suerte, los primeros cuatro podrían llegar a la intersección al mismo tiempo, dando lugar a la situación de la Figura 1-13(b) en la que se bloquean mutuamente (interbloquean) y ninguno de ellos puede avanzar. Cada uno está bloqueando a uno de los otros, y no pueden retroceder debido a que se lo impiden los autobuses que están detrás de ellos. No es fácil salir de esta situación.

Los procesos de un ordenador pueden experimentar una situación análoga en la cual ninguno de ellos pueda hacer ningún progreso. Por ejemplo, imaginemos un ordenador con una unidad de cinta y una grabadora de CD. Ahora imaginemos que dos procesos necesitan producir un CD-ROM cada uno a partir de datos que están en una cinta. El proceso 1 solicita la unidad de cinta y se le concede. Luego el proceso 2 solicita la grabadora de CD y se le concede. Ahora el proceso 1 solicita la grabadora de CD y se le suspende hasta que el proceso 2 termine de utilizarla. Por último el proceso 2 solicita la unidad de cinta y también queda suspendido porque el proceso 1 la está utilizando. Aquí tenemos ya un interbloqueo del cual no hay escapatoria. En el capítulo 3 estudiaremos en detalle los interbloqueos y qué puede hacerse con ellos.

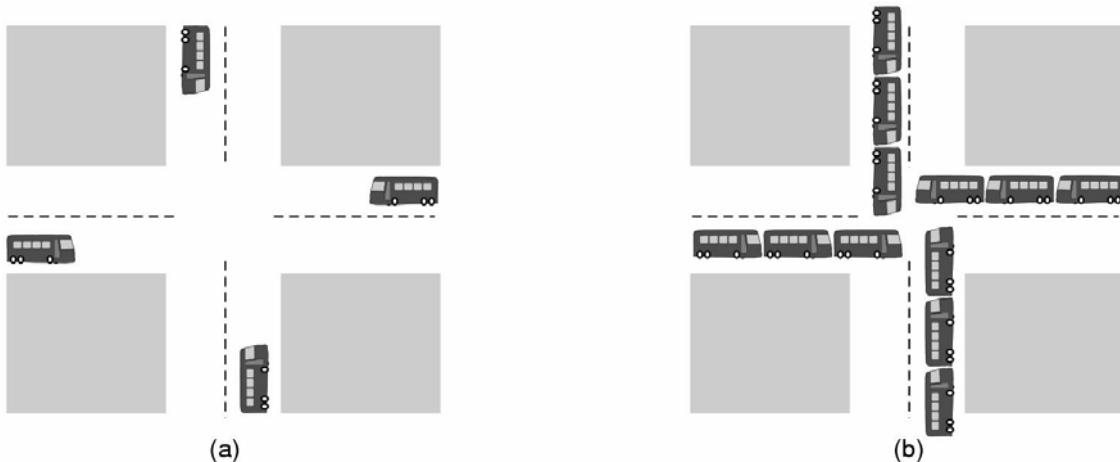


Figura 1-13. (a) Un interbloqueo potencial. (b) un interbloqueo real.

1.5.3 Gestión de Memoria

Todo ordenador tiene una memoria principal que utiliza para albergar los programas en ejecución. En los sistemas operativos más sencillos, sólo hay un programa a la vez en la memoria. Para ejecutar un segundo programa, es preciso desalojar el primero y colocar el segundo en la memoria.

Los sistemas operativos algo más sofisticados permiten que haya varios programas en la memoria al mismo tiempo. Para evitar que se interfieran (y que interfieran con el sistema operativo), es necesario algún tipo de mecanismo de protección. Aunque este mecanismo tiene que estar en el hardware, es controlado por el sistema operativo.

El punto de vista anterior tiene que ver con la gestión y la protección de la memoria principal del ordenador. Un aspecto distinto, pero igualmente importante, relacionado con la memoria es la gestión del espacio de direcciones de los procesos. Normalmente, cada proceso tiene algún conjunto de direcciones que puede usar y que normalmente va desde 0 hasta algún máximo. En el caso más sencillo, la cantidad máxima de espacio de direcciones que tiene un proceso es menor que la memoria principal. De esa manera, un proceso puede llenar su espacio de direcciones habiendo suficiente espacio en la memoria principal para contenerlo.

Sin embargo, en muchos ordenadores las direcciones son de 32 o 64 bits, lo que significa espacios de direcciones de 2^{32} o 2^{64} bytes, respectivamente. ¿Qué sucede si el espacio de direcciones de un proceso es mayor que la memoria principal del ordenador y el proceso quiere hacer uso de todo su espacio? En los primeros ordenadores no se podía ejecutar ese desafortunado proceso. Actualmente existe una técnica denominada memoria virtual, en la cual el sistema operativo mantiene una parte de su espacio de direcciones en la memoria principal y otra parte en el disco, y transfiere fragmentos entre ambos lugares según sea necesario. Esta importante función del sistema operativo, y otras relacionadas con la administración de memoria, se tratarán en el capítulo 4.

1.5.4 Entrada/Salida

Todos los ordenadores tienen dispositivos físicos para admitir entradas y producir salidas. Después de todo, ¿de qué serviría un ordenador si los usuarios no pudieran decirle qué quieren que haga y no pudieran recibir los resultados una vez realizado el trabajo requerido? Existen muchos tipos de dispositivos de entrada/salida, incluyendo teclados, monitores, impresoras, etc., y corresponde al sistema operativo gestionar esos dispositivos.

Consecuentemente, todo sistema operativo cuenta con un subsistema de E/S para gestionar sus dispositivos de E/S. Parte del software de E/S es independiente del dispositivo, es decir, es igualmente válida para muchos o todos los dispositivos de E/S. Otras partes, como los drivers de los dispositivos, son específicos de dispositivos particulares de E/S. En el capítulo 5 examinaremos el software de E/S.

1.5.5 Ficheros

Otro concepto clave que soportan virtualmente todos los sistemas operativos es el sistema de ficheros. Como señalamos anteriormente, una de las funciones más importantes del sistema operativo consiste en ocultar las peculiaridades de los discos y demás dispositivos de E/S, y presentar al programador un bonito y claro modelo abstracto de ficheros independientes del dispositivo. Es obvio que se requieren llamadas al sistema para crear ficheros, borrar ficheros, leer ficheros y escribir ficheros. Antes de poder leer un fichero es preciso localizarlo en el disco y abrirlo, y después de leer el fichero es necesario cerrarlo, por lo que el sistema operativo debe proporcionar llamadas al sistema para realizar esas tareas.

Para proporcionar un lugar donde guardar los ficheros, casi todos los sistemas operativos incorporan el concepto de **directorio** como una forma de agrupar los ficheros. Por ejemplo, un estudiante podría tener un directorio para cada asignatura que está recibiendo (para los programas que necesita en esa asignatura), otro directorio para su correo electrónico y otro más para su página principal World Wide Web. Por tanto, se necesitan llamadas al sistema para

crear y borrar directorios. También se proporcionan llamadas para colocar un fichero existente en un directorio y para borrar un fichero de un directorio. Una entrada de directorio puede corresponder a un fichero o a otro directorio. Este modelo da lugar también a una jerarquía –el sistema de ficheros– como se muestra en la Figura 1-14.

Tanto la jerarquía de procesos como la de ficheros están organizadas en forma de árbol, pero las similitudes sólo llegan hasta ahí. Las jerarquías de procesos no suelen ser muy profundas (es inusual que haya más de tres niveles), mientras que las jerarquías de ficheros suelen tener cuatro, cinco o incluso más niveles de profundidad. Las jerarquías de procesos son por normalmente muy efímeras, existiendo a lo más durante unos pocos minutos, mientras que la jerarquía de directorios puede existir durante años. La propiedad y la protección también son diferentes para procesos y ficheros. Normalmente, sólo un proceso padre puede controlar o siquiera tener acceso a los procesos hijos, pero casi siempre existen mecanismos que permiten leer ficheros y directorios a un grupo de usuarios mayor que sólo su dueño.

Todo fichero dentro de la jerarquía de directorios puede especificarse mediante su **camino** (*path name*) desde lo alto de la jerarquía (el **directorio raíz**). Tales caminos absolutos consisten en la lista de los directorios que hay que atravesar para ir desde el directorio raíz hasta el fichero, separando los componentes con *slashes*. En la Figura 1-14, el camino del fichero *CS101* es */Faculty/Prof.Brown/Courses/CS101*. El slash inicial indica que el camino es absoluto, es decir, que comienza en el directorio raíz. Como comentario, en MS-DOS y Windows se utiliza el carácter *backslash* (\) como separador en lugar del carácter slash (/), de modo que el camino del fichero anterior se escribiría *\Faculty\Prof.Brown\Courses\CS101*. En todo este libro usaremos por lo general el convenio de UNIX para los caminos.

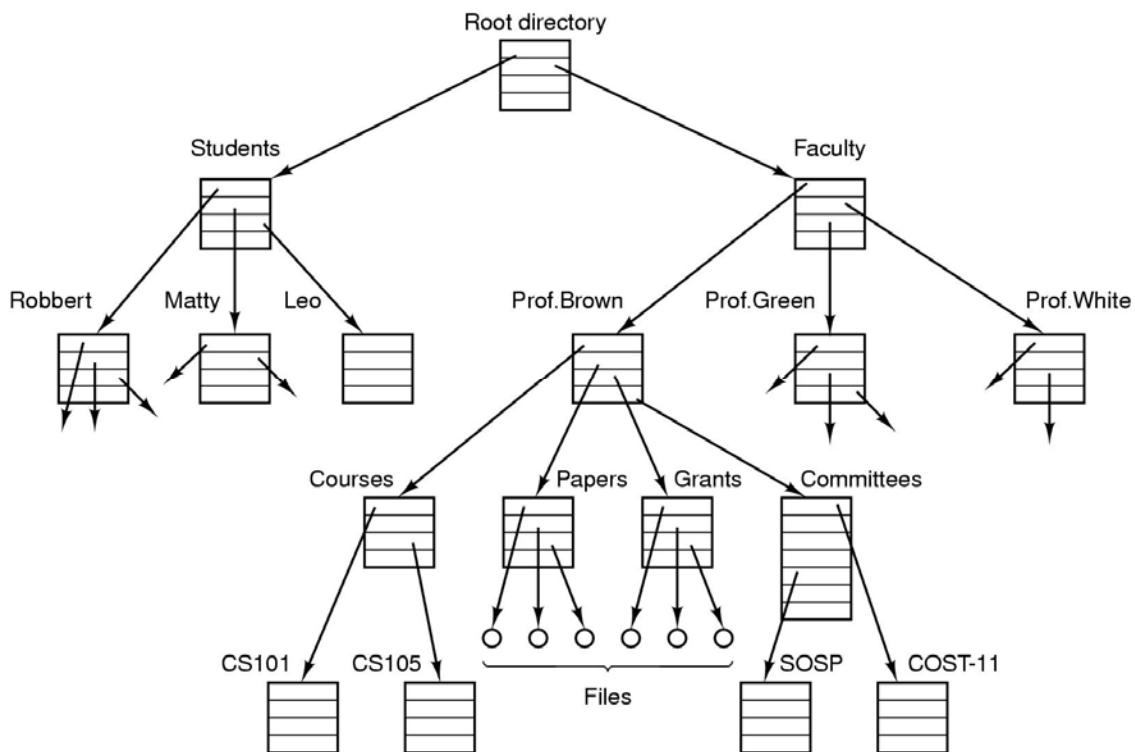


Figura 1-14. Sistema de ficheros para un departamento universitario.

En cualquier instante dado, cada proceso tiene un **directorio de trabajo** actual, en el cual se buscan los caminos (*path names*) que no comienzan con un slash. Por ejemplo, en la Figura 1-14, si el directorio de trabajo fuera *|Faculty|Prof.Brown*, la utilización del nombre de camino *Courses\CS101* haría referencia al mismo fichero que el camino absoluto que dimos antes. Los procesos pueden cambiar su directorio de trabajo haciendo una llamada al sistema especificando el nuevo directorio de trabajo.

Para poder leer o escribir en un fichero es preciso abrirlo, comprobándose en ese momento los permisos de acceso. Si está permitido el acceso, el sistema devuelve un entero corto denominado un **descriptor de fichero** para su utilización en las operaciones subsiguientes. Si el acceso está prohibido, se retorna un código de error.

Otro concepto importante en UNIX es el de sistema de ficheros montado. Casi todos los ordenadores personales tienen una o más unidades de disquete en las que pueden meterse y sacarse discos. Con el fin de proporcionar una manera elegante para operar con medios removibles (incluidos los CD-ROMs), UNIX permite unir el sistema de ficheros de un disquete al árbol principal. Consideremos la situación de la Figura 1-15(a). Antes de la llamada a *mount*, el **sistema de ficheros raíz**, en el disco duro, y un segundo sistema de ficheros, en un disquete, están separados sin ninguna relación.

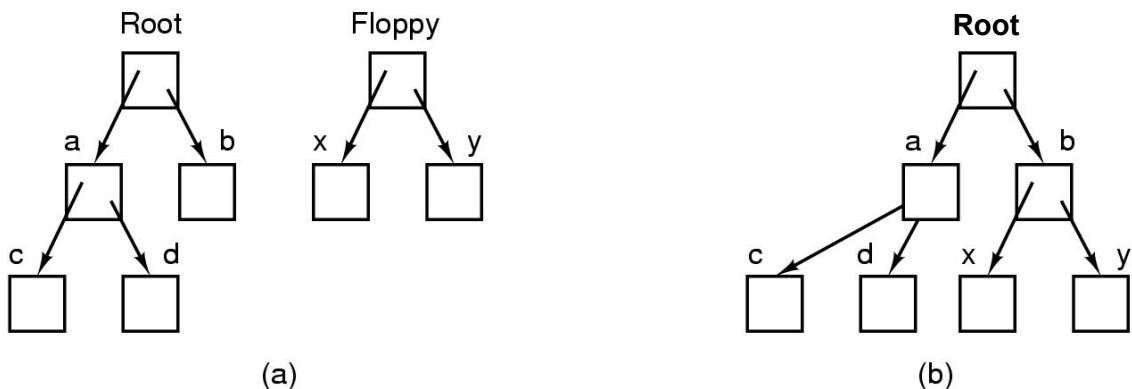


Figura 1-15. (a) No se tiene acceso a los ficheros del disquete antes de montarlo. (b) Despues de montarlo forman parte de la jerarquía de ficheros.

Sin embargo, el sistema de ficheros del disquete no puede utilizarse porque no hay forma de especificar caminos (*path names*) en él. UNIX no permite poner como prefijo a los caminos un nombre o número de unidad; esa sería precisamente el tipo de dependencia del dispositivo que los sistemas operativos deben eliminar. En vez de eso, la llamada al sistema *mount* permite añadir el sistema de ficheros del disquete al sistema de ficheros raíz, cuando el programa quiera. En la Figura 1-15(b), el sistema de ficheros del disquete se ha montado ya en el directorio *b*, lo que permite el acceso a los ficheros */b/x* y */b/y*. Si el directorio *b* hubiera contenido previamente algún fichero, dicho fichero quedaría inaccesible mientras el disquete permanezca montado, ya que */b* se refiere actualmente al directorio raíz del disquete. (No tener acceso a esos ficheros no es tan grave como en principio parece: los sistemas de ficheros se montan casi siempre en directorios vacíos). Si un sistema contiene varios discos duros, pueden montarse todos también en un único árbol.

Otro concepto importante en UNIX es el de **fichero especial**. Los ficheros especiales se proporcionan con el objetivo de lograr que los dispositivos de E/S parezcan ficheros. De esa manera, los dispositivos de E/S pueden leerse y escribirse utilizando las mismas llamadas al sistema que se utilizan para leer y escribir ficheros. Existen dos tipos de ficheros especiales: **ficheros especiales de bloques** y **ficheros especiales de caracteres**. Los ficheros especiales de

bloques se utilizan para modelar dispositivos que consisten en una colección de bloques direccionables aleatoriamente, tales como los discos. Abriendo un fichero especial de bloques y leyendo, digamos el bloque 4, un programa puede acceder directamente al cuarto bloque del dispositivo, sin tener en cuenta la estructura del sistema de ficheros que contenga. Similarmente, los ficheros especiales de caracteres se utilizan para modelar impresoras, módems y otros dispositivos que aceptan o producen un flujo de caracteres. Por convenio, los ficheros especiales se guardan en el directorio `/dev`. Por ejemplo, `/dev/lp` podría ser la impresora.

La última característica que trataremos en esta visión de conjunto es una que se relaciona tanto con los procesos como con los ficheros: las tuberías. Una **tubería** (*pipe*) es una especie de pseudofichero que puede utilizarse para conectar dos procesos, como se muestra en la Figura 1-16. Si los procesos *A* y *B* desean comunicarse mediante una tubería, deberán establecerla con antelación. Cuando el proceso *A* quiere enviar datos al proceso *B*, escribe en la tubería como si fuera un fichero de salida. El proceso *B* puede leer los datos de la tubería como si fuera un fichero de entrada. De esta manera, la comunicación entre los procesos en UNIX se parece mucho a la lectura y escritura ordinaria de ficheros. Aún más fuerte, la única forma en que un proceso puede descubrir que el fichero de salida sobre el que está escribiendo no es realmente un fichero, sino una tubería, es haciendo una llamada al sistema especial. Los sistemas de ficheros son muy importantes. Tendremos que decir muchas más cosas de ellos en el capítulo 6 y también en los capítulos 10 y 11.

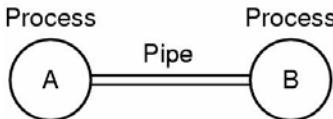


Figura 1-16. Dos procesos conectados por una tubería.

1.5.6 Seguridad

Los ordenadores contienen grandes cantidades de información que los usuarios a menudo desean que sea confidencial. Tal información podría incluir correo electrónico, planes de negocios, declaraciones de la renta y muchas otras cosas más. Corresponde al sistema operativo gestionar la seguridad del sistema de forma que los ficheros, por ejemplo, sólo sean accesibles para los usuarios autorizados.

Como un ejemplo sencillo con el único fin de dar una idea de cómo podría funcionar la seguridad, consideremos UNIX. En UNIX los ficheros se protegen asignándoles un código de protección binario de 9 bits. Dicho código de protección consiste en tres campos de tres bits, uno para el propietario, otro para el resto de miembros del grupo del propietario (el administrador del sistema divide a los usuarios en grupos) y otro para todos los demás usuarios. Cada campo tiene un bit para el acceso de lectura, un bit para el acceso de escritura y un bit para el acceso de ejecución. Estos tres bits se conocen como **bits rwx**. Por ejemplo, el código de protección `rwxr-x--x` significa que el propietario puede leer (*read*), escribir (*write*) y ejecutar (*execute*) el fichero, el resto de miembros del grupo pueden leer o ejecutar (pero no escribir) el fichero, y el resto de los usuarios pueden ejecutar (pero no leer ni escribir) el fichero. En el caso de un directorio, *x* indica permiso de búsqueda. Un guion indica que no se ha dado el permiso correspondiente.

Además de la protección de los ficheros, hay muchos otros aspectos sobre la seguridad. Uno de ellos es la protección del sistema contra intrusos no deseados, tanto humanos como no humanos (por ejemplo, los virus). En el capítulo 9 examinaremos diversas cuestiones sobre la seguridad.

1.5.7 El shell

El sistema operativo es el código que lleva a cabo las llamadas al sistema. Por tanto, los editores, compiladores, ensambladores, enlazadores e intérpretes de comandos por tanto no forman parte del sistema operativo, aunque sean muy importantes y útiles. Con riesgo de confundir un poco las cosas, en esta sección examinaremos brevemente el intérprete de comandos de UNIX, denominado el **shell**. Aunque no es parte del sistema operativo, hace un uso intensivo de muchas de sus características y por tanto sirve como un buen ejemplo de cómo pueden usarse las llamadas al sistema. También constituye la principal interfaz entre un usuario sentado frente a su terminal y el sistema operativo, a menos que el usuario esté utilizando una interfaz gráfica de usuario (GUI). Existen muchos shells, incluyéndose entre ellos *sh*, *csh*, *ksh* y *bash*. Todos soportan la funcionalidad que describimos a continuación, que se deriva del shell original (*sh*).

Cuando un usuario inicia su sesión, se arranca un shell que tiene al terminal como entrada estándar y salida estándar. Lo primero que hace el shell es mostrar un **prompt**, un carácter tal como el signo de dólar (\$), que indica al usuario que el shell está esperando recibir un comando. Si ahora el usuario teclea, por ejemplo,

```
date
```

el shell creará un proceso hijo que ejecutará el programa *date*. Mientras se está ejecutando el proceso hijo, el shell espera a que termine. Cuando el hijo termina, el shell muestra de nuevo el prompt y trata de leer la siguiente línea de entrada.

El usuario puede especificar que la salida estándar se redirija a un fichero, por ejemplo:

```
date > fichero
```

Similarmente, puede redirigirse la entrada estándar, como en:

```
sort < fichero1 > fichero2
```

que invoca el programa de ordenación *sort* tomando su entrada de *fichero1* y enviando su salida a *fichero2*.

La salida de un programa puede utilizarse como entrada de otro programa mediante su conexión a través de una tubería. De este modo,

```
cat fichero1 fichero2 fichero3 | sort > /dev/lp
```

invoca el programa *cat* para concatenar tres ficheros y enviar la salida a *sort* para que ordene todas las líneas por orden alfabetico. La salida de *sort* se redirige al fichero */dev/lp*, que normalmente corresponde a la impresora.

Si un usuario añade un *ampersand* (&) después de un comando, el shell no esperará a que termine su ejecución, y presentará el prompt inmediatamente. Consecuentemente,

```
cat fichero1 fichero2 fichero3 | sort > /dev/lp &
```

realiza la ordenación como un trabajo en segundo plano (*background*) o de fondo, permitiendo al usuario seguir trabajando normalmente mientras se efectúa la ordenación. El shell tiene otras funciones interesantes, las cuales no podemos entrar a comentar aquí. La mayoría de los libros sobre UNIX tratan ampliamente el shell (por ejemplo, Kernighan y Pike, 1984; Kochan y Wood, 1990; Medinets, 1999; Newhan y Rosenblatt, 1998, y Robbins, 1999).

1.5.8 Reciclaje de conceptos

La informática, al igual que otros muchos campos, ha avanzado desde siempre gracias al empuje de la tecnología. La razón por la que los antiguos romanos no tenían automóviles no es que les gustase mucho caminar, sino que no sabían cómo construirlos. Los ordenadores personales existen no porque millones de personas tuvieran un deseo largamente contenido de poseer un ordenador, sino porque ahora es posible fabricarlos de manera económica. A menudo olvidamos lo mucho que la tecnología afecta a nuestra perspectiva de los sistemas y vale la pena meditar al respecto de vez en cuando.

En particular, frecuentemente sucede que un cambio en la tecnología deja obsoleta alguna idea que inmediatamente a continuación desaparece. Sin embargo, otro cambio en la tecnología podría hacer que esa idea repentinamente volviera a estar vigente. Esto es especialmente cierto cuando el cambio tiene que ver con el rendimiento relativo de diversas partes del sistema. Por ejemplo cuando las CPUs se volvieron mucho más rápidas que las memorias, las cachés adquirieron una importancia crucial porque aceleraban el “lento” acceso a la memoria. Si algún día una nueva tecnología de memoria consigue que las memorias sean mucho más rápidas que las CPUs, desaparecerán inmediatamente las cachés. Y si luego una nueva tecnología de la CPU vuelve a hacer que éstas sean más rápidas que las memorias, las cachés reaparecerán al instante. En biología, la extinción es definitiva, pero en informática la extinción es a veces sólo por unos pocos años.

Como consecuencia de esta falta de permanencia, en este libro examinaremos de vez en cuando “conceptos obsoletos”, esto es, ideas que no son óptimas con la tecnología actual. Sin embargo, ciertos cambios en la tecnología podrían hacer que volvieran algunos de estos supuestos “conceptos obsoletos”. Por ese motivo, es importante entender por qué está obsoleto un concepto y qué cambios en el entorno podrían hacer que resurgiera.

A fin de aclarar este punto, consideremos unos cuantos ejemplos. Los primeros ordenadores tenían repertorios de instrucciones implementados físicamente mediante circuitos. Las instrucciones eran ejecutadas directamente por el hardware y no podían modificarse de ninguna manera. Luego llegó la microprogramación, en la que un intérprete subyacente ejecutaba las instrucciones por software. La ejecución cableada se quedó obsoleta. Luego se inventaron los ordenadores RISC y la microprogramación (es decir, la ejecución interpretada) se quedó a su vez obsoleta porque la ejecución directa era mucho más rápida. Ahora estamos viendo un resurgir de la interpretación en la forma de applets de Java que se envían a través de Internet y se interpretan al llegar. La velocidad de ejecución no siempre es crucial porque los retrasos a través de la red son ya tan grandes que tienden a dominar. Pero eso también podría cambiar algún día.

Los primeros sistemas operativos asignaban espacio en el disco a los ficheros colocando todo su contenido en sectores contiguos, uno después de otro. Aunque este esquema era muy fácil de implementar, no era flexible porque cuando un fichero crecía, podía no haber ya suficiente espacio contiguo para almacenarlo. De este modo, el concepto de ficheros almacenados en sectores contiguos se desechó por ser obsoleto. Pero eso fue hasta que llegaron los CD-ROMs. En su caso no existe el problema de que los ficheros puedan crecer. De repente, la sencillez de la asignación contigua de espacio para los ficheros se vio como una brillante idea por lo que los sistemas de ficheros en CD-ROM se basan ahora en ella.

Como idea final, consideremos el enlace dinámico. El sistema MULTICS se diseñó de modo que funcionara día y noche sin detenerse nunca. Para corregir errores en el software, era necesario tener una forma de reemplazar procedimientos de biblioteca mientras se estaban usando. Con ese fin se inventó el concepto de enlace dinámico. Cuando MULTICS pasó a mejor vida, el concepto cayó en el olvido durante algún tiempo. Sin embargo, ese concepto fue

redescubierto cuando los sistemas operativos modernos necesitaron una forma de permitir que muchos programas compartiesen los mismos procedimientos de biblioteca sin tener que hacer sus propias copias privadas (debido a que las bibliotecas de gráficos habían crecido de forma desmesurada). La mayoría de los sistemas soportan ahora alguna forma de enlace dinámico. La lista sigue, pero estos ejemplos deben dejar ya bien claro que una idea que hoy está obsoleta podría ser mañana la estrella de la fiesta.

La tecnología no es el único factor que hace evolucionar a los sistemas y al software. La economía desempeña también un papel importante. En las décadas de 1960 y 1970, la mayoría de los terminales eran terminales de impresión mecánica (teletipos) o tubos de rayos catódicos (CRTs) de 25×80 caracteres, y no terminales gráficos de mapas de bits. Esta elección no era una cuestión de tecnología. Los terminales gráficos de mapas de bits se utilizaban ya antes de 1960. El problema era que costaban decenas de miles de dólares cada una. Sólo cuando su precio se desplomó pudo pensarse (fuera de ámbito militar) en dedicar un terminal gráfico a un usuario individual.

1.6 LLAMADAS AL SISTEMA

La interfaz entre el sistema operativo y los programas de usuario está definida por el conjunto de llamadas al sistema ofrecidas por el sistema operativo. Para entender realmente lo que hacen los sistemas operativos, debemos examinar de cerca esa interfaz. Las llamadas al sistema disponibles en la interfaz varían de un sistema operativo a otro (aunque los conceptos subyacentes tienden a ser similares).

Por lo tanto estamos obligados a hacer una elección entre (1) vagas generalidades (“los sistemas operativos tienen llamadas al sistema para leer ficheros”) y (2) algún sistema específico (“UNIX tiene una llamada al sistema `read` con tres parámetros: uno para especificar el fichero, otro para indicar dónde deben colocarse los datos y otro para indicar el número de bytes a leer”).

Hemos elegido el segundo enfoque. De esta manera hay que trabajar más, pero se comprende mejor lo que los sistemas operativos hacen en realidad. Aunque esta discusión se refiere específicamente a POSIX (Estándar Internacional 9945-1) y por lo tanto también a UNIX, System V, BSD, Linux, MINIX, etc., la mayoría de los demás sistemas operativos modernos tienen llamadas al sistema que realizan las mismas funciones, aunque difieran los detalles. Puesto que el mecanismo real para hacer una llamada al sistema depende mucho de la máquina y a menudo debe expresarse en código ensamblador, es usual que se proporcionen procedimientos de biblioteca para hacer posible realizar llamadas al sistema desde programas en C y a menudo también desde otros lenguajes de programación.

Resulta útil tener en mente lo siguiente. Cualquier ordenador monoprocesador (es decir, que tenga una única CPU) sólo puede ejecutar una instrucción a la vez. Si un proceso está ejecutando un programa de usuario en modo usuario y necesita un servicio del sistema, tal como leer datos de un fichero, tendrá que ejecutar una instrucción de *trap* o de llamada al sistema para transferir el control al sistema operativo. El sistema operativo determina entonces lo que quiere el proceso invocador examinando los parámetros. A continuación llevará a cabo la llamada al sistema y devolverá el control a la instrucción que está justo después de la llamada al sistema. En cierto sentido, efectuar una llamada al sistema es como efectuar una llamada a un procedimiento de tipo especial, sólo que las llamadas al sistema entran en el núcleo (*kernel*) y las llamadas a procedimiento no.

Para dejar más claro el mecanismo de llamada al sistema, vamos a echar un vistazo a la llamada al sistema `read`. Como ya mencionamos, tiene tres parámetros: el primero especifica el fichero, el segundo apunta al búfer y el tercero indica el número de bytes a leer. Igual que casi todas las llamadas al sistema, puede invocarse desde programas en C llamando a un procedimiento de biblioteca que tiene el mismo nombre que la llamada al sistema: `read`. Una llamada desde un programa en C podría tener el siguiente aspecto:

```
contador = read(fd, &buffer, nbytes);
```

La llamada al sistema (y el procedimiento de biblioteca) devuelve el número de bytes que realmente se leyeron, dejándolo en la variable `contador`. Normalmente ese valor es igual a `nbytes`, pero podría ser menor si, por ejemplo, se llega al final del fichero durante la lectura.

Si no puede llevarse a cabo la llamada al sistema, sea por un parámetro no válido o por un error de disco, se asigna -1 a `contador` y el número de error se coloca en una variable global, `errno`. Los programas siempre deben comprobar los resultados de una llamada al sistema para ver si se produjo algún error.

Las llamadas al sistema se ejecutan en una serie de pasos. Para aclarar más este concepto, vamos a examinar la llamada `read` discutida anteriormente. En preparación de la llamada al procedimiento de biblioteca `read`, que es el que realmente hace la llamada al sistema `read`, el programa que invoca la llamada apila los parámetros en la pila, como se muestra en los pasos 1, 2 y 3 de la Figura 1-17. Los compiladores de C y C++ apilan los parámetros en la pila en orden inverso por razones históricas (que tienen que ver con conseguir que el primer parámetro de `printf`, la cadena de caracteres de formato, quede en la cima de la pila, independientemente del número de parámetros reales de la llamada). El primer y tercer parámetros se pasan por valor, pero el segundo parámetro se pasa por referencia, lo que significa que se pasa la dirección del búfer (indicada por `&`), no su contenido. Luego viene la llamada al procedimiento de biblioteca propiamente dicha (paso 4). Ésta instrucción es la instrucción de llamada a procedimiento que se utiliza normalmente para llamar a cualquier procedimiento.

El procedimiento de biblioteca, posiblemente escrito en lenguaje ensamblador, normalmente pone el número de llamada al sistema en un lugar donde el sistema operativo espera encontrarlo, como por ejemplo en un registro (paso 5). Luego ejecuta una instrucción TRAP para pasar de modo usuario a modo núcleo (supervisor) e inicia la ejecución a partir de una dirección fija dentro del núcleo (paso 6). El código del núcleo iniciado examina el número de llamada al sistema y bifurca al manejador de la llamada al sistema correcto, utilizando usualmente una tabla de punteros a manejadores de llamadas al sistema indexada por el número de llamada al sistema (paso 7). En este punto se ejecuta el manejador de la llamada al sistema (paso 8). Una vez que el manejador de la llamada al sistema termina su trabajo, puede devolver el control al procedimiento de biblioteca en el espacio de usuario, en la instrucción que sigue a la instrucción TRAP (paso 9). Este procedimiento retorna entonces al programa de usuario de la forma usual que lo hacen las llamadas a los procedimientos (paso 10).

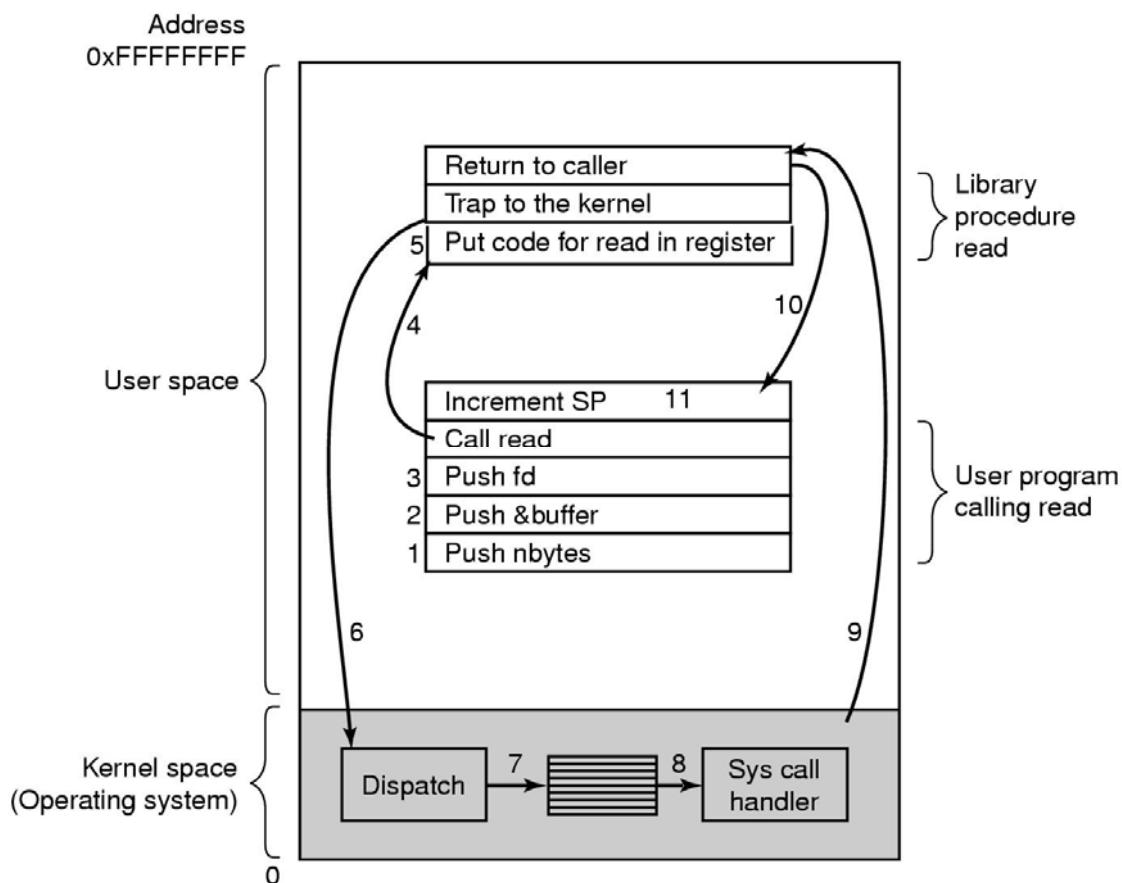


Figura 1-17. Los 11 pasos para hacer la llamada al sistema `read(fd, &buffer, nbytes)`.

Para terminar, el programa de usuario tiene que limpiar la pila, como se hace después de cualquier llamada a un procedimiento (paso 11). Suponiendo que la pila crece hacia abajo, como es frecuente, el código compilado incrementará el puntero de pila exactamente lo justo para eliminar los parámetros que se metieron en la pila antes de la llamada a `read`. Ahora el programa queda en libertad para hacer a continuación lo que quiera.

En el paso 9, teníamos una buena razón para decir que el manejador de la llamada al sistema “puede devolver el control al procedimiento de biblioteca en el espacio de usuario ...”. La llamada al sistema podría bloquear al proceso que hizo la llamada, impidiéndole continuar. Por ejemplo, si se está tratando de leer del teclado y no se ha tecleado nada todavía, el proceso que hace la llamada al sistema tendrá que bloquearse. En este caso, el sistema operativo buscará algún otro proceso que pueda ejecutarse a continuación. Después, cuando esté disponible la entrada deseada, el sistema atenderá al primer proceso continuándose con los pasos 9, 10 y 11.

Process management

| Call | Description |
|--|--|
| <code>pid = fork()</code> | Create a child process identical to the parent |
| <code>s = waitpid(pid, &statloc, options)</code> | Wait for a child to terminate |
| <code>s = execve(name, argv, environp)</code> | Replace a process' core image |
| <code>exit(status)</code> | Terminate process execution and return status |

File management

| Call | Description |
|---|--|
| <code>fd = open(file, how, ...)</code> | Open a file for reading, writing or both |
| <code>s = close(fd)</code> | Close an open file |
| <code>n = read(fd, buffer, nbytes)</code> | Read data from a file into a buffer |
| <code>n = write(fd, buffer, nbytes)</code> | Write data from a buffer into a file |
| <code>position = lseek(fd, offset, whence)</code> | Move the file pointer |
| <code>s = stat(name, &buf)</code> | Get a file's status information |

Directory and file system management

| Call | Description |
|---|--|
| <code>s = mkdir(name, mode)</code> | Create a new directory |
| <code>s = rmdir(name)</code> | Remove an empty directory |
| <code>s = link(name1, name2)</code> | Create a new entry, name2, pointing to name1 |
| <code>s = unlink(name)</code> | Remove a directory entry |
| <code>s = mount(special, name, flag)</code> | Mount a file system |
| <code>s = umount(special)</code> | Unmount a file system |

Miscellaneous

| Call | Description |
|---|---|
| <code>s = chdir(dirname)</code> | Change the working directory |
| <code>s = chmod(name, mode)</code> | Change a file's protection bits |
| <code>s = kill(pid, signal)</code> | Send a signal to a process |
| <code>seconds = time(&seconds)</code> | Get the elapsed time since Jan. 1, 1970 |

Figura 1-18. Algunas de las principales llamadas al sistema POSIX. El código de retorno `s` es `-1` si ocurrió algún error. Los demás códigos de retorno son como sigue: `pid` es un identificador de proceso, `fd` es un descriptor de fichero, `n` es un contador de bytes, `position` es un desplazamiento dentro del fichero y `seconds` es el tiempo transcurrido. Los parámetros se explican en el texto.

En las secciones siguientes examinaremos algunas de las llamadas al sistema de POSIX que más se usan, o más exactamente, los procedimientos de biblioteca que nos facilitan hacer esas llamadas al sistema. POSIX tienen cerca de 100 llamadas a procedimientos de ese tipo. Algunas de las más importantes se listan en la Figura 1-18 agrupadas por conveniencia en cuatro categorías. En el texto examinaremos brevemente cada llamada para ver lo qué hace. En gran medida, los servicios ofrecidos por estas llamadas determinan casi todo lo que el sistema operativo puede hacer, ya que la gestión de recursos en los ordenadores personales es mínima (al menos en comparación con las máquinas más grandes que tienen múltiples usuarios). Los servicios incluyen cosas como crear y terminar procesos, crear, borrar, leer y escribir ficheros, gestionar directorios y realizar entrada/salida.

Como comentario, merece la pena señalar que la correspondencia entre las llamadas a procedimientos POSIX y las llamadas al sistema no es uno a uno. El estándar POSIX especifica cierto número de procedimientos que un sistema conforme a POSIX debe proporcionar, pero no especifica si son llamadas al sistema, llamadas a procedimientos de biblioteca, o cualquier otra cosa. Si un procedimiento puede llevarse a cabo sin invocar una llamada al sistema (es decir, sin entrar en el núcleo), casi siempre se ejecutará en el espacio del usuario por cuestiones de eficiencia. Sin embargo, la mayoría de los procedimientos POSIX sí invocan llamadas al sistema, correspondiéndose usualmente cada procedimiento directamente con una llamada al sistema. En unos pocos casos, especialmente donde se requieren varios procedimientos que no son más que pequeñas variaciones unos de otros, una llamada al sistema se encarga de más de una llamada a un procedimiento de biblioteca.

1.6.1 Llamadas al sistema para la gestión de los procesos

El primer grupo de llamadas de la Figura 1-18 se ocupa de la gestión de procesos. **Fork** (bifurcación) es un buen lugar para comenzar la explicación. **Fork** es la única manera de crear un nuevo proceso en UNIX. Lo que se crea es un duplicado exacto del proceso original, incluyendo todos los descriptores de fichero, registros, ... en una palabra todo. Después de **fork**, el proceso original y la copia (el padre y el hijo) siguen cada cual su camino. Todas las variables tienen valores idénticos en el momento de ejecutar el **fork**, pero, dado que los datos del padre se copian para crear al hijo, los cambios posteriores en uno de ellos no afectan al otro. (El padre y el hijo comparten el código del programa, que no puede modificarse.) La llamada **fork** devuelve un valor que es cero en el hijo e igual al identificador del proceso hijo o **PID** en el padre. Utilizando el **PID** devuelto, los dos procesos pueden saber cuál de ellos es el proceso padre y cuál de ellos es el proceso hijo.

En la mayoría de los casos, después de un **fork**, el hijo tendrá que ejecutar diferente código que el padre. Consideremos el caso del shell, que lee un comando del terminal, produce un proceso hijo con un **fork**, espera a que el hijo ejecute el comando, y cuando el hijo termina lee el siguiente comando. Para esperar a que el hijo termine, el padre ejecuta una llamada al sistema **waitpid**, que simplemente espera hasta que el hijo termina (cualquier hijo si es que hay más de uno). **Waitpid** puede esperar a que termine un hijo específico, o a que termine cualquier hijo asignando **-1** a su primer parámetro. Cuando **waitpid** termina, la dirección a la que apunta el segundo parámetro, **statloc**, contendrá el estado de terminación del hijo (normal o anormal, y el valor de terminación indicado en una llamada **exit**). Se dispone de varias opciones, que se especifican con el tercer parámetro.

Consideremos ahora la manera en la que el shell utiliza **fork**. Cuando se teclea un comando, el shell produce un nuevo proceso mediante un **fork**. Ese proceso hijo deberá ejecutar el comando del usuario, cosa que hace por medio de la llamada al sistema **execve**, la cual provoca que toda su imagen del núcleo sea reemplazada por el fichero que se indica en su primer parámetro. (En realidad, la llamada al sistema propiamente dicha es **exec**, pero varios procedimientos de biblioteca distintos la invocan con diferentes parámetros y nombres

ligeramente distintos. Aquí trataremos a todos estos procedimientos como llamadas al sistema.) En la Figura 1-19 se muestra un shell muy simplificado que ilustra el uso de `fork`, `waitpid` y `execve`.

```
#define TRUE 1

while (TRUE) {                                /* repite indefinidamente */
    type_prompt();                            /* muestra el prompt en pantalla */
    read_command(command, parameters); /* lee un comando del terminal */

    if (fork() != 0) {                         /* crea el proceso hijo */
        /* codigo del padre */
        waitpid(-1, &status, 0);           /* espera a que el hijo termine */
    } else {
        /* codigo del hijo */
        execve(command, parameters, 0); /* ejecuta el comando */
    }
}
```

Figura 1-19. Un shell reducido. A lo largo de este libro supondremos que `TRUE` está definido como 1.

En el caso más general, `execve` tiene tres parámetros: el nombre del fichero a ejecutar, un puntero al array de argumentos y un puntero al array del entorno. Describiremos estos parámetros en breve. Con el fin de poder omitir parámetros o especificarlos de distintas maneras se proporcionan diferentes rutinas de biblioteca, entre ellas `execl`, `execv`, `execle` y `execve`. A lo largo del libro utilizaremos el nombre `exec` para referirnos a la llamada al sistema que invocan todos estos procedimientos.

Vamos a considerar el caso de un comando como

```
cp fichero1 fichero2
```

que copia el `fichero1` en el `fichero2`. Una vez que el shell ha hecho el `fork`, el proceso hijo localiza y ejecuta el fichero `cp` y le pasa los nombres de los ficheros origen y destino.

El programa principal de `cp` (y el de la mayoría de los programas en C) contiene la declaración

```
main(argc, argv, envp)
```

donde `argc` es el número de elementos que hay en la línea de comandos, incluyendo el nombre del programa. En el ejemplo anterior `argc` es 3.

El segundo parámetro, `argv`, es un puntero a un array. El elemento *i* de ese array es un puntero a la *i*-esima cadena de la línea de comandos. En nuestro ejemplo, `argv[0]` apuntaría a la cadena “`cp`”, `argv[1]` apuntaría a la cadena “`fichero1`” y `argv[2]` apuntaría a la cadena “`fichero2`”.

El tercer parámetro de `main`, `envp`, es un puntero al entorno, que es un array de cadenas conteniendo asignaciones de la forma *nombre* = *valor*, utilizadas para pasar información al programa, tal como el tipo de terminal y el nombre del directorio de partida del usuario (`$HOME`). En la Figura 1-19 no se pasa ningún entorno al hijo, por lo que el tercer parámetro de `execve` es un cero.

Si al lector le parece que `exec` es complicada, no debe desesperarse; se trata de la más compleja (desde el punto de vista semántico) de todas las llamadas al sistema de POSIX. Todas las demás son mucho más sencillas. Como ejemplo de llamada sencilla tenemos a `exit`, la cual deben utilizar todos los procesos cuando terminan su ejecución. Sólo tiene un parámetro que es el estado de terminación (del 0 al 255), que se devuelve al padre mediante `statloc` en la llamada al sistema `waitpid`.

En UNIX la memoria de un proceso se divide en tres segmentos: el **segmento de código** (es decir, el código del programa), el **segmento de datos** (o sea, las variables) y el **segmento de pila**. El segmento de datos crece hacia arriba y el de pila crece hacia abajo, como se muestra en la Figura 1-20. Entre ellos hay un hueco de espacio de direcciones desocupado. La pila crece dentro de ese hueco automáticamente, a medida que se va necesitando, pero la expansión del segmento de datos se efectúa de manera explícita utilizando una llamada al sistema, `brk`, que especifica la nueva dirección donde termina el segmento de datos. Esta llamada, sin embargo, no está definida por el estándar POSIX, ya que se recomienda a los programadores que utilicen el procedimiento de biblioteca `malloc` para asignar memoria dinámicamente, y no se consideró conveniente estandarizar la implementación de `malloc` porque pocos programadores la utilizan de forma directa.

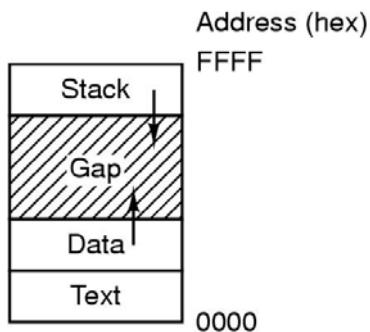


Figura 1-20. Los procesos tienen tres segmentos: código, datos y pila.

1.6.2 Llamadas al sistema para la gestión de ficheros

Muchas llamadas al sistema están relacionadas con el sistema de ficheros. En esta sección examinaremos llamadas que operan sobre ficheros individuales; en la siguiente nos ocuparemos de las que trabajan con directorios o con el sistema de ficheros en su totalidad.

Para leer o escribir en un fichero, primero debe abrirse el fichero utilizando `open`. Esta llamada especifica el nombre del fichero a abrir, como un nombre de camino absoluto o relativo al directorio de trabajo, y uno de los códigos `O_RDONLY`, `O_WRONLY` u `O_RDWR`, indicando apertura para leer, para escribir o para ambas cosas. Para crear un fichero nuevo se utiliza `O_CREAT`. Una vez abierto el fichero, el descriptor de fichero devuelto puede utilizarse para leer o escribir. Después el fichero puede cerrarse con `close`, lo que hace que el descriptor de fichero quede libre para reutilizarse en un `open` posterior.

Las llamadas que más se utilizan son indudablemente `read` y `write`. Ya vimos `read` antes. `Write` tiene los mismos parámetros.

Aunque la mayoría de los programas lean y escriben ficheros secuencialmente, algunos programas de aplicación necesitan ser capaces de tener acceso a cualquier parte de un fichero de forma aleatoria. Cada fichero tiene asociado un puntero que indica la posición actual en el fichero. Al leer (escribir) secuencialmente, dicho puntero normalmente apunta al siguiente byte a leer (escribir). La llamada `lseek` modifica el valor del puntero de posición, de forma que las llamadas posteriores a `read` o `write` puedan comenzar en cualquier punto del fichero.

La llamada al sistema **Iseek** tiene tres parámetros: el primero es el descriptor de fichero, el segundo es una posición en el fichero y el tercero indica si dicha posición es relativa al principio del fichero, a la posición actual o al fin del fichero. El valor devuelto por **Iseek** es la posición absoluta en el fichero después de haber desplazado el puntero.

Para cada fichero, UNIX se mantiene al tanto del tipo de fichero (fichero regular, fichero especial, directorio, etc.), su tamaño, cuando se modificó por última vez y otra información. Los programas pueden pedir conocer esa información con la llamada al sistema **stat**. El primer parámetro especifica el fichero a inspeccionar; el segundo es un puntero a una estructura donde se colocará la información correspondiente al fichero.

1.6.3 Llamadas al sistema para la gestión de los directorios

En esta sección veremos algunas llamadas al sistema que se relacionan más con directorios o con el sistema de ficheros en su totalidad, que con un fichero particular, como en la sección previa. Las primeras dos llamadas, **mkdir** y **rmdir**, crean y borran directorios vacíos, respectivamente. La siguiente llamada es **link**. Su propósito es permitir que el mismo fichero aparezca con dos o más nombres, a menudo en directorios diferentes. Una utilización típica sería permitir que varios miembros de un equipo de programación comparten un fichero común, que aparecerá en el directorio de cada uno, posiblemente bajo diferentes nombres. Compartir un fichero no es lo mismo que proporcionar a cada miembro del equipo una copia privada, debido a que tener un fichero compartido significa que los cambios que haga cualquier miembro del equipo deben ser visibles instantáneamente para los demás miembros—sólo existe un fichero. Cuando se hacen copias de un fichero, los cambios posteriores realizados sobre una copia no afectan a las demás copias.

Para ver cómo funciona **link**, consideremos la situación de la Figura 1-21(a). Tenemos dos usuarios, *ast* y *jim*, que poseen cada uno de ellos su propio directorio con algunos ficheros. Si ahora *ast* ejecuta un programa contenido la llamada al sistema

```
link("/usr/jim/memo", "/usr/ast/note") ;
```

el fichero *memo* del directorio de *jim* aparecerá ahora también en el directorio de *ast* bajo el nombre *note*. En adelante, */usr/jim/memo* y */usr/ast/note* se referirán al mismo fichero. Por cierto, el que los directorios de usuario se guarden en */usr*, */user*, */home* o algún otro lado es puramente una decisión tomada por el administrador del sistema local.

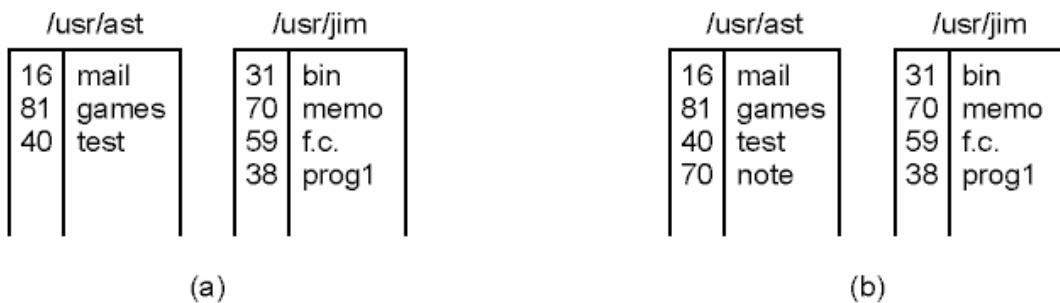


Figura 1-21. (a) Dos directorios antes de enlazar */usr/jim/memo* al directorio de *ast*. (b) Los mismos directorios después del enlace.

Para dejar más claro lo que hace **link** lo mejor es que entendamos cómo funciona. Todo fichero en UNIX tiene un número único, su número de i-nodo, que lo identifica. Este número es un índice para buscar en una tabla de **i-nodos**, con un i-nodo por cada fichero, que indica quién es el propietario del fichero, dónde están sus bloques de disco, etc. Un directorio es simplemente un fichero que contiene un conjunto de pares (número de i-nodo, nombre ASCII). En las primeras versiones de UNIX, cada entrada de directorio tenía 16 bytes: 2 bytes para el número de i-nodo y 14 bytes para el nombre. Actualmente se necesita una estructura más complicada para soportar nombres de fichero más largos, pero conceptualmente un directorio sigue siendo un conjunto de pares (número de i-nodo, nombre ASCII). Por ejemplo, en la Figura 1-21(a), *mail* tiene el número de i-nodo 16. Lo que hace **link** es simplemente crear una nueva entrada de directorio con un nombre (quizá nuevo) y el número de i-nodo de un fichero que ya existe. En la Figura 1-21(b), dos entradas tienen el mismo número de i-nodo (70) y por tanto se refieren al mismo fichero. Si cualquiera de ellas se elimina posteriormente, utilizando la llamada al sistema **unlink**, la otra permanecerá. Si ambas se eliminan, UNIX verá que ya no hay entradas que hagan referencia al fichero (un campo del i-nodo lleva la cuenta del número de entradas de directorio que apuntan al fichero), por lo que el fichero se borrará del disco.

Como ya mencionamos anteriormente, la llamada al sistema **mount** permite fusionar dos sistemas de ficheros en uno solo. Una situación común es tener en un disco duro el sistema de ficheros raíz que contiene las versiones binarias (ejecutables) de los comandos comunes y otros ficheros muy utilizados. El usuario podría insertar entonces en la disquetera un disquete con los ficheros a leer.

Mediante la ejecución de la llamada al sistema **mount**, el sistema de ficheros del disquete puede añadirse al sistema de ficheros raíz, como se muestra en la Figura 1-22. Una instrucción en C representativa para efectuar la operación sería

```
mount("/dev/fd0", "/mnt", 0);
```

donde el primer parámetro es el nombre de un fichero especial de bloques para la unidad de disquete 0, el segundo parámetro es el punto donde se montará en el árbol, y el tercero indica si el sistema de ficheros se montará para lectura-escritura o sólo-lectura.

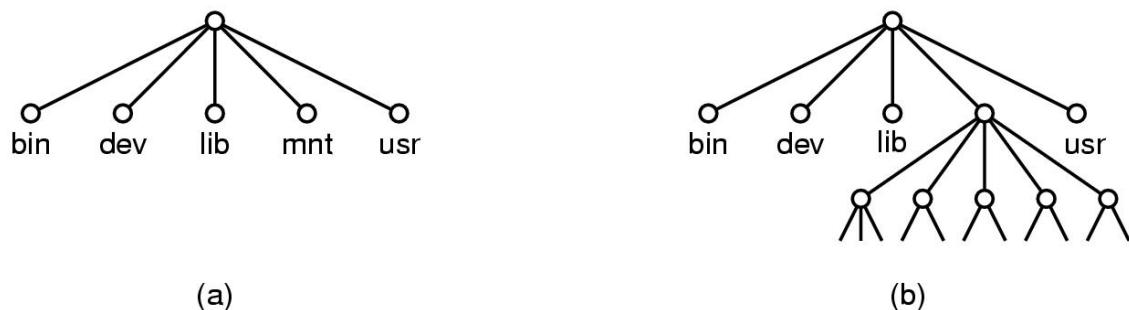


Figura 1-22. (a) Sistemas de ficheros antes del montaje. (b) Sistema de ficheros después del montaje.

Después de la llamada **mount**, se podrá tener acceso a un fichero en la unidad 0 con sólo dar su camino desde el directorio raíz o desde el directorio de trabajo, sin tener que especificar en qué unidad se encuentra. De hecho, es posible montar una segunda, tercera o cuarta unidad en cualquier parte del árbol. La llamada al sistema **mount** permite integrar medios removibles en una única jerarquía de ficheros integrada, sin tener que preocuparse de en qué dispositivo está un fichero. Aunque este ejemplo se refiere a disquetes, también pueden montarse con esta técnica discos duros o porciones de discos duros (a menudo denominadas **particiones** o **dispositivos menores**). Cuando ya no se necesita un sistema de ficheros, puede desmontarse mediante la llamada al sistema **umount**.

1.6.4 Otras llamadas al sistema

Existe una gran variedad de otras llamadas al sistema, pero aquí nos fijaremos en sólo cuatro de ellas. La llamada **chdir** cambia el directorio de trabajo actual. Después de la llamada

```
chdir("/usr/ast/test");
```

si abrimos con **open** el fichero *xyz*, estaremos abriendo exactamente el fichero */usr/ast/test/xyz*. El concepto de directorio de trabajo elimina la necesidad de teclear todo el tiempo nombres de camino absolutos (que suelen ser muy largos).

En UNIX, todo fichero tiene asignado un modo utilizado con fines de protección. El modo incluye los bits rwx (read-write-execute) para el propietario, para su grupo y para los demás usuarios. La llamada al sistema **chmod** permite cambiar el modo de un fichero. Por ejemplo, si queremos que un fichero sea de sólo lectura para todo el mundo con excepción de su propietario, podríamos ejecutar

```
chmod("fichero", 0644);
```

La llamada al sistema **kill** es la manera con la cual los usuarios y los procesos de usuario envían señales. Si un proceso está preparado para capturar una señal dada, cuando ésta llegue se ejecutará un manejador de la señal. Si el proceso no está preparado para manejar una señal, la llegada de la señal simplemente “mata” (de ahí el nombre de la llamada) al proceso.

POSIX define varios procedimientos para controlar el tiempo. Por ejemplo, **time** devuelve el tiempo actual en segundos, contados a partir del 1 de enero de 1970 a media noche (en el momento que se inicia el día, no al terminar). En los ordenadores con tamaño de palabra de 32 bits, el valor máximo que puede devolver **time** es $2^{32}-1$ segundos (suponiendo que se utiliza un entero sin signo). Este valor corresponde a un poco más de 136 años. Por tanto, en el año 2106 los sistemas UNIX de 32 bits se volverán locos, al estilo del famoso “efecto 2000”. Si el lector tiene un sistema UNIX de 32 bits, le recomendamos cambiarlo por uno de 64 bits antes del año 2106.

1.6.5 La API Win32 de Windows

Hasta aquí nos hemos concentrado primordialmente en UNIX. Ahora llegó el momento de echar un vistazo a Windows. Windows y UNIX difieren fundamentalmente en sus modelos de programación. Un programa UNIX consiste en código que hace alguna u otra cosa, realizando llamadas al sistema para que se lleven a cabo ciertos servicios. En contraste, un programa de Windows normalmente está controlado por eventos. El programa principal espera a que ocurra algún evento, y luego llama a un procedimiento para tratar el evento. Son eventos típicos la pulsación de una tecla, el movimiento del ratón, la pulsación de un botón del ratón o la inserción de un disquete. Se llama a los manejadores para procesar el evento, actualizar la pantalla y actualizar el estado interno del programa. Esto da pie a un estilo de programación un tanto distinto del que se utiliza en UNIX, pero dado que el tema de este libro es el funcionamiento y la estructura de los sistemas operativos, no nos ocuparemos mucho más de estos diferentes modelos de programación.

Por supuesto, Windows también tiene llamadas al sistema. En UNIX existe casi una relación uno a uno entre las llamadas al sistema (como `read`) y los procedimientos de biblioteca (como `read`) que se utilizan para invocar las llamadas al sistema. En otras palabras, por cada llamada al sistema suele haber un procedimiento de biblioteca que se llama para invocarla, como se indica en la Figura 1-17. Además, POSIX tiene apenas cerca de 100 llamadas a procedimientos de biblioteca.

Con Windows, la situación es radicalmente distinta. Para empezar, las llamadas a procedimientos de biblioteca y las llamadas al sistema reales están altamente desacopladas. Microsoft ha definido un conjunto de procedimientos, llamado el API **Win32** (*API: Application Program Interface; Interfaz del Programa de Aplicación*) que supuestamente deben utilizar los programadores para obtener los servicios del sistema operativo. Esta interfaz está (parcialmente) soportada en todas las versiones de Windows, desde Windows 95. Al desacoplar la interfaz de las llamadas al sistema propiamente dichas, Microsoft se reserva la posibilidad de modificar con el tiempo las llamadas al sistema (incluso de una versión a la siguiente) sin invalidar los programas existentes. También hay cierta ambigüedad en cuanto a lo qué constituye realmente Win32, ya que Windows 2000 tiene muchas llamadas nuevas que antes no estaban disponibles. En esta sección, Win32 se refiere a la interfaz soportada por todas las versiones de Windows.

El número de llamadas de la API Win32 es extremadamente grande, llegando a ser miles. Además, aunque muchas de ellas sí invocan llamadas al sistema, un número substancial de ellas se ejecutan enteramente en el espacio de usuario. Como consecuencia, en Windows es imposible saber qué es una llamada al sistema (ejecutada por el núcleo) y qué es simplemente una llamada a un procedimiento de biblioteca en el espacio de usuario. De hecho, lo que es una llamada al sistema en una versión de Windows, podría ejecutarse en el espacio de usuario en otra, y viceversa. Cuando analicemos las llamadas al sistema de Windows en este libro, utilizaremos los procedimientos de Win32 (donde sea apropiado), pues Microsoft garantiza su estabilidad en el tiempo. Pero es necesario recordar que no todos ellos son verdaderas llamadas al sistema (es decir traps al núcleo).

Otra complicación es que, en UNIX, la GUI (como X-Windows y Motif) se ejecuta en su totalidad en el espacio de usuario, por lo que las únicas llamadas al sistema que se necesitan para escribir en la pantalla son `write` y otras pocas de menor importancia. Por supuesto, hay un gran número de llamadas a X Windows y la GUI, pero no son de ninguna manera llamadas al sistema.

En contraste, la API de Win32 tiene un enorme número de llamadas para manejar ventanas, figuras geométricas, texto, tipos de letra, barras de desplazamiento, cuadros de diálogo, menús y otras características de la GUI. En la medida en que el subsistema de gráficos se ejecuta en el núcleo (lo cual es cierto en algunas versiones de Windows, pero no en todas),

éstas son llamadas al sistema; de otra manera, sólo son llamadas a procedimientos de biblioteca. ¿Debemos tratar tales llamadas en este libro o no? Puesto que en realidad no están relacionadas con la función de un sistema operativo, hemos decidido no hacerlo, incluso aunque podrían ser ejecutadas por el núcleo. Los lectores interesados en el API Win32 pueden consultar uno de los muchos libros sobre el tema, como Hart (1997); Rector y Newcomer (1997), y Simon (1997).

Es imposible mencionar aquí todas las llamadas de la API Win32, por lo que nos limitaremos a las que corresponden aproximadamente a la funcionalidad de las llamadas de UNIX enumeradas en la Figura 1-18. Las presentamos en la Figura 1-23.

| UNIX | Win32 | Description |
|-------------|---------------------|--|
| fork | CreateProcess | Create a new process |
| waitpid | WaitForSingleObject | Can wait for a process to exit |
| execve | (none) | CreateProcess = fork + execve |
| exit | ExitProcess | Terminate execution |
| open | CreateFile | Create a file or open an existing file |
| close | CloseHandle | Close a file |
| read | ReadFile | Read data from a file |
| write | WriteFile | Write data to a file |
| lseek | SetFilePointer | Move the file pointer |
| stat | GetFileAttributesEx | Get various file attributes |
| mkdir | CreateDirectory | Create a new directory |
| rmdir | RemoveDirectory | Remove an empty directory |
| link | (none) | Win32 does not support links |
| unlink | DeleteFile | Destroy an existing file |
| mount | (none) | Win32 does not support mount |
| umount | (none) | Win32 does not support mount |
| chdir | SetCurrentDirectory | Change the current working directory |
| chmod | (none) | Win32 does not support security (although NT does) |
| kill | (none) | Win32 does not support signals |
| time | GetLocalTime | Get the current time |

Figura 1-23. Llamadas de la API Win32 que corresponden aproximadamente a las llamadas UNIX de la Figura 1-18.

Repasemos brevemente la lista de la Figura 1-23. `CreateProcess` crea un nuevo proceso, realizando la labor combinada de `fork` y `execve` en UNIX. Tiene muchos parámetros que especifican las propiedades del nuevo proceso creado. Windows no tiene una jerarquía de procesos como UNIX, así que no existe ningún concepto de proceso padre y proceso hijo. Una vez creado un proceso, el creador y el creado son iguales. `WaitForSingleObject` sirve para esperar un evento, y son muchos los eventos que pueden esperarse. Si el parámetro especifica un proceso, el proceso que invocó la llamada espera hasta que el proceso especificado termina, lo cual se hace utilizando `ExitProcess`.

Las seis llamadas que siguen operan con ficheros y son funcionalmente similares a sus contrapartidas en UNIX aunque difieren en sus parámetros y detalles. De cualquier modo, es posible abrir, cerrar, leer y escribir ficheros de forma muy similar a como se hace en UNIX. Las llamadas `SetFilePointer` y `GetFileAttributesEx` establecen la posición del fichero y obtienen algunos de los atributos del fichero.

Windows tiene directorios que se crean con `CreateDirectory` y `RemoveDirectory`. También existe el concepto de directorio actual, que se establece con `SetCurrentDirectory`. La hora actual se obtiene con `GetLocalTime`.

La interfaz Win32 no maneja ni enlaces entre ficheros, ni sistemas de ficheros montados, ni seguridad, ni señales, por lo que no existen las correspondientes llamadas a las de UNIX. Por supuesto, Win32 tiene muchas otras llamadas que UNIX no tiene, especialmente las que gestionan la GUI, y Windows 2000 tiene un sistema de seguridad complejo que también maneja enlaces entre ficheros.

Quizás es necesario hacer un último comentario sobre Win32. Win32 no es una interfaz terriblemente uniforme o consistente que digamos. La principal culpable en este sentido fue la necesidad de mantener la compatibilidad hacia atrás con la anterior interfaz de 16 bits empleada en Windows 3.x.

1.7 ESTRUCTURA DEL SISTEMA OPERATIVO

Ahora que hemos visto cual es el aspecto externo de los sistemas operativos (o sea, la interfaz del programador), es el momento de echar un vistazo a su interior. En las siguientes secciones examinaremos cinco estructuras diferentes que se han probado, a fin de tener una idea del espectro de posibilidades. Desde luego, no es una muestra exhaustiva, pero da una idea de algunos diseños que se han probado en la práctica. Los cinco diseños son: sistemas monolíticos, sistemas en capas, máquinas virtuales, exokernels y sistemas cliente-servidor.

1.7.1 Sistemas monolíticos

Esta organización, que con mucho es la más común, bien podría subtitularse como “El Gran Lío”. La estructura consiste en que no hay estructura. El sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales puede llamar a cualquiera de los otros siempre que lo necesite. Cuando se utiliza esta técnica, cada procedimiento del sistema tiene una interfaz bien definida desde el punto de vista de los parámetros y resultados, y cada uno está en libertad de llamar a cualquier otro, si ese otro realiza alguna operación útil que el primero necesita.

Cuando se adopta este enfoque, el programa objeto del sistema operativo se construye compilando primero todos los procedimientos individuales, o ficheros que contienen los procedimientos, para a continuación enlazarlos todos en un único fichero objeto, utilizando el enlazador del sistema. En cuanto a la ocultación de la información, esencialmente no hay ninguna ya que cualquier procedimiento puede ver a cualquier otro (en contraposición con una estructura que contiene módulos o paquetes, en la que gran parte de la información queda oculta dentro de los módulos, y desde afuera sólo pueden invocarse los puntos de entrada oficialmente declarados).

Sin embargo, incluso en los sistemas monolíticos es posible tener al menos un poco de estructura. Los servicios (llamadas al sistema) proporcionados por el sistema operativo se solicitan colocando los parámetros en un lugar bien definido (la pila), y ejecutando después una instrucción TRAP. Esta instrucción hace que el procesador cambie de modo usuario a modo núcleo y transfiera el control al sistema operativo, lo cual se muestra como el paso 6 en la Figura 1-17. Entonces, el sistema operativo obtiene los parámetros y determina qué llamada al sistema debe ejecutarse. Después de eso, utiliza el número de llamada al sistema k como un índice para una tabla que contiene en su entrada k un puntero al procedimiento que lleva a cabo esa llamada (paso 7 de la Figura 1-17).

Esta organización sugiere una estructura básica para el sistema operativo:

1. Un programa principal que invoca el procedimiento de servicio solicitado.
2. Un conjunto de procedimientos de servicio que llevan a cabo las llamadas al sistema.
3. Un conjunto de procedimientos de utilidad que sirven de ayuda a los procedimientos de servicio.

En este modelo, por cada llamada al sistema hay un procedimiento de servicio que se encarga de ella. Los procedimientos de utilidad hacen cosas que son necesarias para varios procedimientos de servicio, como obtener datos de los programas de usuario. Esta división de los procedimientos en tres capas se muestra en la Figura 1-24.

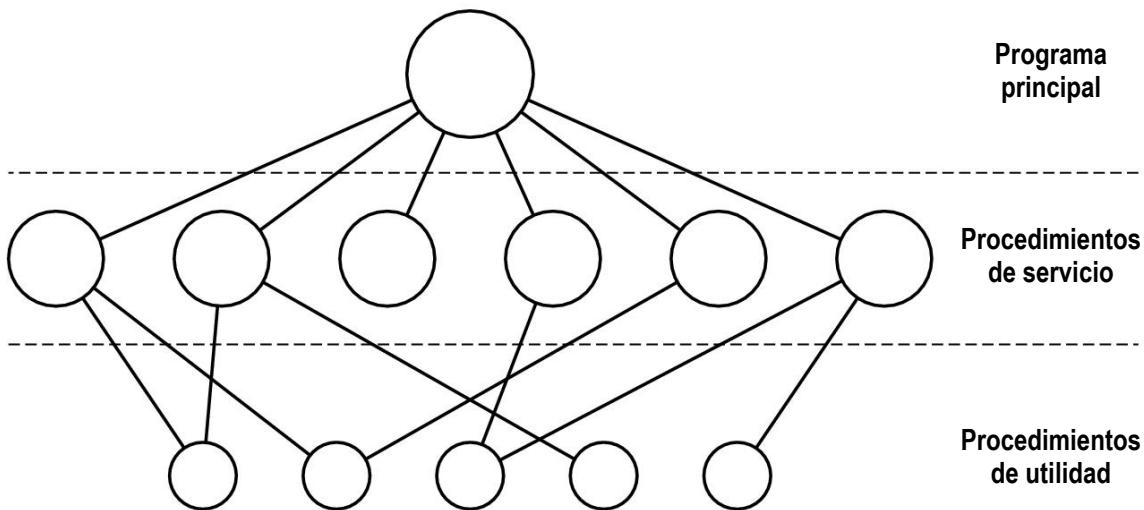


Figura 1-24. Modelo de estructuración simple para un sistema monolítico.

1.7.2 Sistemas estructurados en capas

Una generalización del enfoque de la Figura 1-24 consiste en organizar el sistema operativo en una jerarquía de capas, cada una construida sobre la que está debajo. El primer sistema construido de esta manera fue el THE construido en la Technische Hogescholl Eindhoven en los Países Bajos por E.W. Dijkstra (1968) y sus estudiantes. El sistema THE era un sencillo sistema por lotes para un ordenador holandés, la Electrologica X8, que tenía 32K palabras de 27 bits (los bits eran costosos en aquel entonces).

El sistema tenía seis capas, como se muestra en la Figura 1-25. La capa 0 se ocupaba de la asignación del procesador, conmutando entre procesos según tenían lugar las interrupciones o expiraban los timers. Por encima de la capa 0, el sistema consistía de procesos secuenciales, cada uno de los cuales podía programarse sin tener que preocuparse por el hecho de que varios procesos estuvieran ejecutándose en un único procesador. En otras palabras, la capa 0 hacía posible la multiprogramación básica de la CPU.

| Capa | Función |
|------|---|
| 5 | El operador |
| 4 | Programas de usuario |
| 3 | Gestión de Entrada/Salida |
| 2 | Comunicación operador-proceso |
| 1 | Gestión de memoria y tambor |
| 0 | Asignación del procesador y multiprogramación |

Figura 1-25. Estructura del sistema operativo THE.

La capa 1 se encargaba de la administración de la memoria. Asignaba memoria a los procesos en la memoria principal y sobre un tambor de 512K palabras en el que se guardaban las partes de los procesos (páginas) que no cabían en la memoria principal. Por encima de la capa 1, los procesos no tenían que preocuparse de saber si estaban en la memoria o en el tambor; el software de la capa 1 se encargaba de que las páginas se transfirieran a la memoria cuando se necesitaban.

La capa 2 manejaba la comunicación entre cada proceso y la consola del operador. Por encima de esta capa cada proceso tenía efectivamente su propia consola de operador. La capa 3 se encargaba de gestionar los dispositivos de E/S y de colocar búferes intermedios en los flujos de información hacia y desde los dispositivos de E/S. Encima de la capa 3 cada proceso podía tratar con dispositivos de E/S abstractos con bonitas propiedades, en lugar de dispositivos reales con muchas peculiaridades. En la capa 4 estaban los programas de usuario, que no tenían que preocuparse por la gestión de los procesos, la memoria, la consola o la E/S. El proceso del operador del sistema se localizaba en la capa 5.

En el sistema MULTICS se hizo presente una generalización adicional del concepto de estructuración por capas. En lugar de capas, el sistema MULTICS se describió como si estuviera formado por una serie de anillos concéntricos, teniendo los anillos interiores más privilegios que los exteriores (lo que es efectivamente lo mismo). Cuando un procedimiento de un anillo exterior quería llamar a un procedimiento de un anillo interior, tenía que hacer el equivalente a una llamada al sistema, es decir, una instrucción TRAP cuyos parámetros se verificaban cuidadosamente para comprobar que fueran válidos, antes de permitir que se efectuara la llamada. Aunque en MULTICS todo el sistema operativo formaba parte del espacio de direcciones de cada proceso de usuario, el hardware permitía designar procedimientos individuales (en realidad, segmentos de memoria) como protegidos frente a lectura, escritura o ejecución.

Si bien el esquema de capas del sistema THE no era más que una ayuda para el diseño, porque en última instancia todas las partes del sistema se enlazaban en un único programa objeto, en MULTICS el mecanismo de anillos sí que estaba muy presente en tiempo de ejecución, estando reforzado por el hardware de protección. La ventaja del mecanismo de anillos es que puede extenderse con facilidad para estructurar los subsistemas de usuario. Por ejemplo, un profesor puede escribir un programa para testear y evaluar los programas de los estudiantes y ejecutarlo en el anillo n , mientras que los programas de usuario se ejecutarían en el anillo $n + 1$ para que de ninguna manera pudieran alterar sus calificaciones.

1.7.3 Máquinas virtuales

Las primeras versiones de OS/360 fueron estrictamente sistemas por lotes. No obstante, muchos usuarios de las 360 deseaban disponer de tiempo compartido, por lo que diversos grupos, tanto dentro como fuera de IBM, decidieron escribir sistemas de tiempo compartido para esa máquina. El sistema de tiempo compartido oficial de IBM, el TSS/360, tardó mucho en entregarse, y cuando por fin llegó era tan grande y lento que pocos sitios adoptaron el nuevo sistema. Eventualmente el sistema se abandonó después de que su desarrollo hubiera consumido alrededor de 50 millones de dólares (Graham, 1970). No obstante, un grupo del Centro Científico de IBM, en Cambridge, Massachusetts, produjo un sistema radicalmente distinto, que IBM aceptó al final como producto, y que ahora se utiliza ampliamente en los mainframes que subsisten.

Este sistema, denominado originalmente CP/CMS y rebautizado más adelante como VM/370 (Seawright y MacKinnon, 1979), se basaba en una astuta observación: un sistema de tiempo compartido proporciona: (1) multiprogramación y (2) una máquina extendida con una interfaz más conveniente que el hardware desnudo. La esencia del VM/370 consiste en separar por completo estas dos funciones.

El corazón del sistema, conocido como **monitor de máquina virtual**, se ejecuta sobre el hardware desnudo y realiza la multiprogramación, proporcionando no una, sino varias máquinas virtuales a la siguiente capa inmediatamente superior, como se muestra en la Figura 1-26. Sin embargo, a diferencia de todos los demás sistemas operativos, estas máquinas virtuales no son máquinas extendidas, con ficheros y otras características bonitas. En vez de eso, son

copias *exactas* del hardware desnudo que incluyen el modo dual de ejecución usuario/supervisor, E/S, interrupciones y todo lo demás que tiene la máquina real.

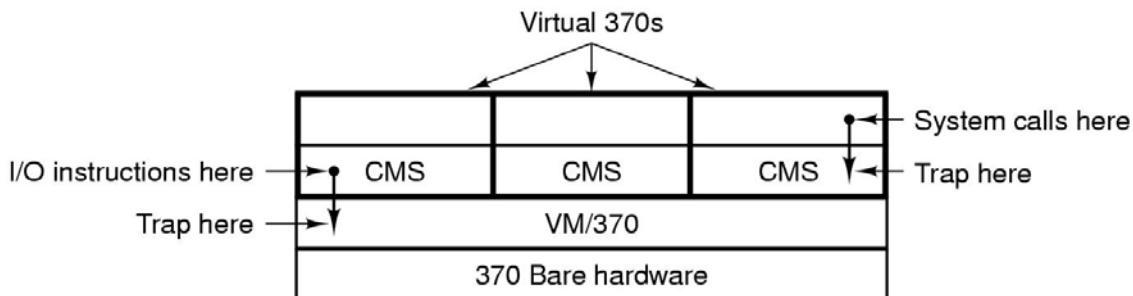


Figura 1-26. Estructura de VM/370 con CMS.

Dado que cada máquina virtual es idéntica al hardware verdadero, cada una puede ejecutar cualquier sistema operativo ejecutable directamente sobre el hardware desnudo. Diferentes máquinas virtuales pueden ejecutar sistemas operativos distintos, y a menudo lo hacen. Algunas ejecutan uno de los descendientes del OS/360 para el procesamiento por lotes o de transacciones, mientras que otras ejecutan un sistema interactivo monousuario llamado **CMS** (*Conversational Monitor System*; Sistema Monitor Conversacional) para usuarios interactivos de tiempo compartido.

Cuando un programa CMS ejecuta una llamada al sistema, ésta salta (mediante un TRAP) al sistema operativo en su propia máquina virtual, no al VM/370, como haría si se estuviera ejecutando sobre una máquina real, no virtual. Luego el CMS ejecuta las instrucciones de E/S normales para leer de su disco virtual, o lo que sea que se necesite para llevar a cabo la llamada. VM/370 atrapa estas instrucciones de E/S y luego las ejecuta como parte de su simulación del hardware real. Al separar por completo las funciones de multiprogramación y de proporcionar una máquina extendida, cada una de las partes puede ser mucho más sencilla, más flexible y más fácil de mantener.

El concepto de máquina virtual se utiliza mucho hoy en día en un contexto diferente: la ejecución de programas MS-DOS antiguos en un Pentium (u otra CPU Intel de 32 bits). Al diseñar el Pentium y su software, tanto Intel como Microsoft se percataron de que podría haber una gran demanda de gente queriendo ejecutar su software antiguo sobre el nuevo hardware. Por ese motivo, Intel incluyó un modo 8086 virtual en el Pentium. De este modo, la máquina actúa como un 8086 (que es idéntico a un 8088 desde el punto de vista del software), incluyendo el direccionamiento de 16 bits con un límite de 1 MB.

Windows y otros sistemas operativos utilizan este modo para ejecutar programas de MS-DOS. Estos programas se inician en el modo 8086 virtual. En tanto que ejecuten instrucciones normales, se ejecutan sobre el hardware desnudo, pero cuando un programa trate de saltar al sistema operativo para hacer una llamada al sistema, o intente realizar E/S protegida directamente, entonces tendrá lugar un salto (TRAP) al monitor de máquina virtual.

Este diseño puede tener dos variantes. En la primera, MS-DOS se carga en el espacio de direcciones del 8086 virtual, de modo que lo único que hace el monitor de máquina virtual es rebotar el salto a MS-DOS, como sucedería en un 8086 real. Cuando luego MS-DOS intente realizar la llamada él mismo, la operación será capturada y llevada a cabo por el monitor de la máquina virtual.

En la otra variante, el monitor de máquina virtual se limita a atrapar el primer trap y a efectuar él mismo la E/S, pues ya conoce todas las llamadas al sistema de MS-DOS y, por tanto, sabe qué se supone que debe hacer cada trap. Esta variante es menos pura que la primera, puesto que sólo emula correctamente a MS-DOS, y no a otros sistemas operativos, como hace la primera. Por otra parte, es mucho más rápida, pues ahorra el trabajo de poner en marcha al MS-DOS para que realice la E/S. Una desventaja adicional de ejecutar realmente MS-DOS en modo 8086 virtual es que MS-DOS se mete mucho con el bit que habilita/inhibe las interrupciones, y la emulación de esto es muy costosa.

Es necesario resaltar que ninguno de estos enfoques es en realidad igual al del VM/370, ya que la máquina emulada no es un Pentium completo, sino sólo un 8086. Con el sistema VM/370 es posible ejecutar el propio sistema VM/370 en la máquina virtual. Con el Pentium no es posible ejecutar por ejemplo Windows en el 8086 virtual, debido a que ninguna versión de Windows se ejecuta sobre un 8086; un 286 es lo mínimo que se necesita incluso para la versión más antigua, y no se proporciona la emulación del 286 (y mucho menos del Pentium). No obstante, basta modificar un poco el binario de Windows para hacer posible esta emulación, y de hecho incluso está disponible en algunos productos comerciales.

Otro área donde se utilizan las máquinas virtuales, pero de forma un tanto diferente, es en la ejecución de programas en Java. Cuando Sun Microsystems inventó el lenguaje de programación Java, también inventó una máquina virtual (es decir, una arquitectura de ordenador) llamada **JVM** (*Java Virtual Machine*; Máquina Virtual de Java). El compilador de Java produce código para la JVM, que normalmente es ejecutado por un intérprete software de JVM. La ventaja de este enfoque es que el código JVM puede enviarse por Internet a cualquier ordenador que tenga un intérprete de JVM y ejecutarse allí. Si el compilador hubiera producido programas binarios para SPARC o Pentium, por ejemplo, no se podrían haber enviado y ejecutado en cualquier lugar tan fácilmente. (Desde luego, Sun podría haber producido un compilador que produjera binarios para SPARC y luego distribuir un intérprete de SPARC, pero JVM es una arquitectura mucho más sencilla que se presta muy bien a la interpretación.) Otra ventaja de usar JVM es que si el intérprete se implementa como es debido, lo cual no es del todo trivial, es posible verificar que los programas JVM que lleguen sean seguros y luego ejecutarlos bajo un entorno protegido de forma que no puedan robar datos ni causar ningún perjuicio.

1.7.4 Exokernels

Con el VM/370, cada proceso de usuario obtiene una copia exacta del ordenador real. Con el modo 8086 virtual del Pentium, cada proceso de usuario obtiene una copia exacta de un ordenador diferente. Yendo un paso más lejos, algunos investigadores del M.I.T. construyeron un sistema que proporciona a cada usuario un clon del ordenador real, pero con un subconjunto de los recursos (Engler y otros, 1995). Así, una máquina virtual podría obtener los bloques de disco del 0 al 1023, la siguiente podría recibir los bloques del 1024 al 2047, y así de forma sucesiva.

En la capa más baja, ejecutándose en modo núcleo, está un programa llamado **exokernel**. Su labor consiste en asignar recursos a las máquinas virtuales y luego comprobar cualquier intento de utilizarlos para garantizar que ninguna máquina trate de utilizar los recursos de cualquier otra. Cada máquina virtual a nivel de usuario puede ejecutar su propio sistema operativo, como sobre el VM/370 y los 8086 virtuales del Pentium, sólo que cada una está limitada a los recursos que solicitó y que le fueron asignados.

La ventaja del esquema de exokernel es que ahorra una capa de conversión. En los otros diseños, cada máquina virtual cree que tiene su propio disco, cuyos bloques van desde 0 hasta algún máximo, lo que obliga al monitor de la máquina virtual a mantener tablas para convertir las direcciones de disco (y todos los demás recursos). Con el exokernel no es necesario efectuar

esa conversión, pues lo único que tiene que hacer es mantenerse al tanto de qué recursos se han asignado a qué máquinas virtuales. Este método sigue teniendo la ventaja de separar la multiprogramación (en el exokernel) y el código del sistema operativo del usuario (en el espacio del usuario), pero con menos sobrecarga porque la única tarea del exokernel es evitar que las máquinas virtuales se interfieran mutuamente.

1.7.5 Modelo cliente-servidor

El VM/370 gana mucho en simplicidad al mover una gran parte del código del sistema operativo tradicional (la implementación de la máquina extendida) a una capa superior, CMS. No obstante, VM/370 sigue siendo él mismo un programa complejo porque la simulación de varias 370 virtuales en su totalidad no es tan sencilla (sobre todo si se quiere hacer con una eficiencia razonable).

Una tendencia en los sistemas operativos modernos consiste en llevar más lejos aún la idea de subir código a las capas superiores y quitar tanto como sea posible del modo núcleo, dejando un **microkernel** mínimo. El enfoque usual es implementar la mayor parte del sistema operativo en procesos de usuario. Para solicitar un servicio, tal como la lectura de un bloque de un fichero, un proceso de usuario (que ahora se denomina **proceso cliente**) envía una solicitud a un **proceso servidor**, que realiza el trabajo y devuelve la respuesta.

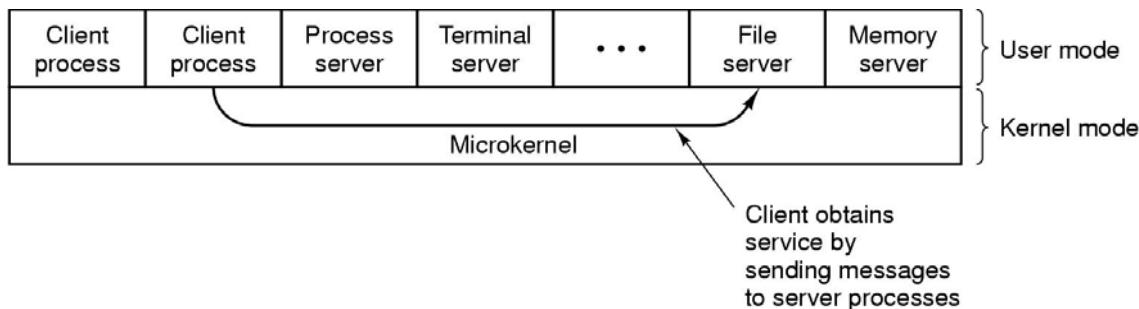


Figura 1-27. El modelo cliente-servidor.

En este modelo, que se muestra en la Figura 1-27, lo único que hace el núcleo es manejar la comunicación entre clientes y servidores. Al dividir el sistema operativo en partes, cada una de las cuales sólo se encarga de una faceta del sistema, tales como el servicio de ficheros, el servicio de procesos, el servicio de terminal o el servicio de memoria, cada parte se vuelve más pequeña y manejable. Además, dado que todos los servidores se ejecutan como procesos en modo usuario, no en modo núcleo, no tienen acceso directo al hardware. En consecuencia, si se produce un error en el servidor de ficheros, podría fallar el servicio de ficheros, pero usualmente ese error no llega a provocar que se detenga toda la máquina.

Otra ventaja del modelo cliente-servidor es su adaptabilidad para usarse en sistemas distribuidos (vea la Figura 1-28). Si un cliente se comunica con el servidor enviándole mensajes, el cliente no necesita saber si el mensaje se maneja de forma local en su propia máquina, o si se envió a través de la red a un servidor situado en una máquina remota. En lo que concierne al cliente, en ambos casos sucede lo mismo: se envió una solicitud y se recibió una respuesta.

La descripción presentada anteriormente de un núcleo que tan solo se encarga de transportar mensajes desde los clientes a los servidores y al revés, no es del todo realista. Algunas funciones del sistema operativo (como la carga de comandos en los registros de los dispositivos de E/S físicos) son difíciles, sino imposibles, de llevar a cabo desde programas en

el espacio del usuario. Hay dos formas de resolver este problema. Una es hacer que algunos procesos servidores cruciales (por ejemplo, los drivers de dispositivo) se ejecuten realmente en modo núcleo, con un acceso sin restricciones a todo el hardware, pero manteniendo todavía su comunicación con los otros procesos, empleando el mecanismo normal de mensajes.

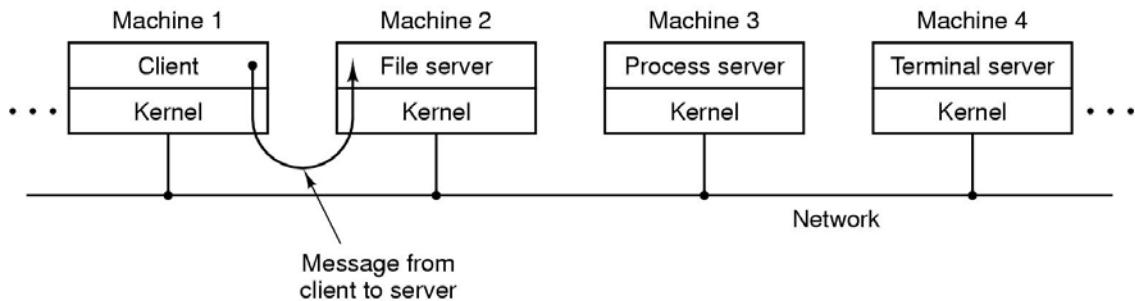


Figura 1-28. El modelo cliente-servidor en un sistema distribuido.

La otra forma es incorporar un mínimo de **mecanismo** en el núcleo pero dejar las decisiones de **política** a los servidores en el espacio de usuario (Levin y otros, 1975). Por ejemplo, el núcleo podría reconocer que un mensaje enviado a una cierta dirección especial implica tomar el contenido de ese mensaje y cargarlo en los registros de dispositivo de E/S de algún disco, a fin de iniciar una lectura del disco. En este ejemplo, el núcleo ni siquiera examinaría los bytes del mensaje para ver si son válidos o lógicos; tan solo los copiaría a ciegas en los registros de dispositivo del disco. (Obviamente tendría que utilizarse algún esquema para limitar tales mensajes a los procesos autorizados.) La división entre mecanismo y política es un concepto importante; se presenta una y otra vez en los sistemas operativos en diversos contextos.

1.8 INVESTIGACIÓN SOBRE SISTEMAS OPERATIVOS

La informática es un campo que avanza con rapidez y es difícil predecir hacia donde se dirige. A los investigadores de las universidades y laboratorios industriales continuamente se les ocurren nuevas ideas, algunas de las cuales no llegan a ningún lado, mientras que otras se convierten en la piedra angular de futuros productos y tienen un impacto enorme en la industria y los usuarios. Distinguir entre ambos tipos de ideas es más fácil en retrospectiva que en tiempo real. Separar el trigo de la paja es especialmente difícil porque a menudo pasan 20 o 30 años entre la idea y su impacto.

Por ejemplo, cuando el presidente Eisenhower creó la Agencia de Proyectos de Investigación Avanzada (ARPA, *Advanced Research Projects Agency*) del Departamento de Defensa, en 1958, estaba tratando de evitar que el ejército, la armada y la fuerza aérea lucharan a muerte entre sí por el presupuesto de investigación del Pentágono. No estaba tratando de inventar Internet. No obstante una de las cosas que ARPA hizo fue financiar algunas investigaciones universitarias relacionadas con el entonces poco conocido concepto de commutación de paquetes, que pronto condujo a la primera red experimental de commutación de paquetes, ARPANET. Esta red comenzó a existir en 1969. Poco después, otras redes de investigación financiadas por ARPA se conectaron a ARPANET, y así nació Internet. Entonces los investigadores académicos pudieron utilizar felizmente Internet para enviarse mensajes de correo electrónico durante 20 años. A principios de los años noventa, Tim Berners-Lee inventó la World Wide Web en el laboratorio de investigación CERN en Ginebra, y Marc Andreessen escribió un navegador gráfico para ella en la Universidad de Illinois. De repente, Internet se llenó de chats entre adolescentes. Probablemente el presidente Eisenhower esté revolcándose en su tumba.

Las investigaciones en el campo de los sistemas operativos han conducido también a drásticos cambios en los sistemas prácticos. Como mencionamos antes, los primeros ordenadores comerciales eran sistemas por lotes, hasta que el M.I.T. inventó el tiempo compartido interactivo, a principios de la década de 1960. Todos los ordenadores se basaban en modo texto hasta que Doug Engelbart inventó el ratón y la interfaz de usuario gráfica, en el Instituto de Investigación de Stanford, a finales de la década de 1960. ¿Quién sabe qué nos deparará el futuro?

En esta sección y en secciones comparables en todo el libro daremos un vistazo a algunas de las investigaciones sobre sistemas operativos que se han realizado durante los últimos 5 a 10 años, con el fin de tener una idea de lo que podría haber en el horizonte. Esta introducción no va a ser exhaustiva, y se basa, en gran medida, en artículos que aparecen en las principales publicaciones y en conferencias sobre investigación, lo que asegura al menos que esas ideas han sobrevivido a un riguroso proceso de revisión antes de su publicación. La mayoría de los artículos citados en las secciones de investigación han sido publicados por la ACM, la IEEE Computer Society o USENIX, y están disponibles por Internet para los (estudiantes) miembros de esas organizaciones. Para más información sobre esas organizaciones y sus bibliotecas digitales, puede visitarse

| | |
|-----------------------|---|
| ACM | http://www.acm.org |
| IEEE Computer Society | http://www.computer.org |
| USENIX | http://www.usenix.org |

Virtualmente todos los investigadores en el campo de los sistemas operativos aceptan que los sistemas operativos actuales son voluminosos, inflexibles, poco fiables, inseguros y que están repletos de errores, algunos más que otros (*no se dan nombres para proteger a los culpables*). Consecuentemente, se ha investigado intensamente la manera de construir sistemas flexibles y confiables. Gran parte de las investigaciones se ocupan de sistemas de microkernel.

Estos sistemas tienen un núcleo mínimo, por lo que existe la posibilidad razonable de que puedan llegar a ser fiables y estar completamente depurados de errores. También son flexibles porque gran parte del sistema operativo real se ejecuta como una serie de procesos en modo usuario que pueden sustituirse o adaptarse con facilidad, quizás incluso durante su ejecución. Típicamente, lo único que hace el microkernel es manejar la administración de recursos de bajo nivel y el paso de mensajes entre los procesos de usuario.

La primera generación de microkernels, como Amoeba (Tanenbaum y otros, 1990), Chorus (Rozier y otros, 1988), Mach (Acceta y otros, 1986) y V (Cheriton, 1988), demostró que era posible construir tales sistemas y lograr que funcionaran. La segunda generación está tratando de demostrar que no sólo pueden funcionar, sino que pueden hacerlo proporcionando un alto rendimiento (Ford y otros, 1996; Hartig y otros, 1997; Liedtke, 1995, 1996; Rawson, 1997, y Zuberi y otros, 1999). Con base en las mediciones publicadas, parece ser que se ha alcanzado ese objetivo.

Gran parte de las investigaciones actuales sobre el núcleo se concentran en la construcción de sistemas operativos extensibles. Éstos son normalmente sistemas de microkernel con la capacidad de extenderse o adaptarse en algún sentido. Como ejemplos podemos citar Fluke (Ford y otros, 1997), Paramecium (Van Doorn y otros, 1995), SPIN (Bershad y otros, 1995b) y Vino (Seltzer y otros, 1996). Algunos investigadores están buscando también la forma de extender sistemas existentes (Ghormley y otros, 1998). Muchos de estos sistemas permiten a los usuarios añadir su propio código al núcleo, lo que hace surgir obviamente el problema de cómo permitir las extensiones de usuario sin comprometer la seguridad. Entre las posibles técnicas están interpretar las extensiones, restringirlas a "cajas de arena" de código, utilizar lenguajes con verificación de tipos y utilizar firmas de código (Grimm y Bershad, 1997, y Small y Seltzer, 1998). Druschel y otros (1997) presentan una opinión contraria, alegando que se está invirtiendo demasiado esfuerzo en la seguridad de los sistemas extensibles por el usuario. Según ellos, los investigadores deben determinar qué extensiones son útiles y simplemente incorporarlas al núcleo de la forma usual, sin permitir a los usuarios extender el núcleo sobre la marcha.

Aunque una estrategia para eliminar los sistemas operativos inflados, plagados de errores y poco confiables es hacerlos más pequeños, una estrategia más radical consiste en eliminar el sistema operativo por completo. El grupo de Kaashoek en el M.I.T. está adoptando este enfoque en sus investigaciones sobre exokernels. Se trata de tener una delgada capa de software que se ejecuta sobre el hardware desnudo y cuya única misión es asignar de manera segura los recursos de hardware a los usuarios. Por ejemplo, el exokernel debe decidir quién puede usar qué parte del disco y dónde deben entregarse los paquetes de red que lleguen. Todo lo demás se deja en manos de los procesos de usuario, haciendo posible construir sistemas operativos tanto de propósito general como altamente especializados (Engler y Kaashoek, 1995; Engler y otros., 1995, y Kaashoek y otros, 1997).

1.9 ESBOZO DEL RESTO DEL LIBRO

Hemos terminado nuestra introducción y recorrido a vista de pájaro el sistema operativo. Ha llegado el momento de descender sobre los detalles. El capítulo 2 trata sobre los procesos. En él se discuten las propiedades de los procesos y la forma en que se comunican entre sí. También se proporcionan varios ejemplos detallados de cómo funciona la comunicación entre procesos y de cómo evitar algunos escollos.

El capítulo 3 trata los interbloqueos. En este capítulo de introducción ya explicamos brevemente lo que son, pero hay mucho más que decir al respecto. Se discutirán diferentes maneras de prevenirlos o de evitarlos.

En el capítulo 4 estudiaremos en detalle la administración de la memoria. Se examinará el importante tema de la memoria virtual, junto con otros conceptos íntimamente relacionados, tales como la paginación y la segmentación.

La entrada/salida es el tema del capítulo 5. Veremos los conceptos de dependencia e independencia del dispositivo. Utilizaremos como ejemplo varios dispositivos importantes, incluyendo discos, teclados y pantallas.

Luego en el capítulo 6, entraremos en el tema (superimportante) de los sistemas de ficheros. En gran medida, lo que el usuario ve, por encima de cualquier otro aspecto del sistema operativo, es el sistema de ficheros. Nos fijaremos tanto en la interfaz del sistema de ficheros como en la implementación del sistema de ficheros.

En este punto habremos terminado nuestro estudio de los principios básicos de los sistemas operativos con una única CPU. Sin embargo, hay mucho más que decir, especialmente sobre temas avanzados. En el capítulo 7 examinaremos los sistemas multimedia, que tienen varias propiedades y requisitos que difieren de los sistemas operativos convencionales. Entre otras cosas, la naturaleza del software multimedia afecta a la planificación y al sistema de ficheros. Otro tema avanzado es el de los sistemas con varios procesadores, que incluyen multiprocesadores, ordenadores paralelos y sistemas distribuidos. Estos temas se tratan en el capítulo 8.

Un tema enormemente importante es la seguridad de los sistemas operativos, que se cubre en el capítulo 9. Entre los temas que se discuten en ese capítulo están las amenazas (por ejemplo virus y gusanos), los mecanismos de protección y los modelos de seguridad.

A continuación realizamos un estudio de algunos sistemas operativos reales. Éstos son UNIX (capítulo 10) y Windows 2000 (capítulo 11). El libro concluye con algunas ideas sobre el diseño de sistemas operativos en el capítulo 12.

1.10 UNIDADES MÉTRICAS

Para evitar cualquier confusión, es necesario establecer explícitamente que en este libro, como en informática en general, se utilizan las unidades del sistema métrico decimal (metro-kilo-segundo) en lugar de las unidades inglesas tradicionales (el sistema estadio-piedra-quincena). En la Figura 1-29 se presentan los principales prefijos métricos. Esos prefijos suelen abreviarse con su inicial (en mayúscula si la unidad es mayor que 1). Así una base de datos de 1 TB ocupa 10^{12} bytes de memoria y un reloj de 100 ps da un tic cada 10^{-10} segundos. Ya que tanto mili como micro comienzan con la letra “m”, hubo que hacer una elección. Normalmente, “m” significa mili y “u” (la letra griega mu) significa micro.

Figura 1-29. Los principales prefijos métricos.

También es necesario señalar que, para medir tamaños de memoria, en la práctica común de la industria, las unidades tienen significados ligeramente diferentes. Por ejemplo kilo significa 2^{10} (1024) en vez de 10^3 (1000) porque las memorias tienen siempre una capacidad que es potencia de 2. Así una memoria de 1 KB contiene 1024 bytes, no 1000 bytes. De forma similar, una memoria de 1 MB contiene 2^{20} (1.048.576) bytes y una memoria de 1 GB contiene 2^{30} (1.073.741.824) bytes. Sin embargo, una línea de comunicación de 1 Kbps transmite 1000 bits por segundo y una LAN de 10 Mbps opera a 10.000.000 bits/s porque estas velocidades no son potencias de 2. Lamentablemente, mucha gente tiende a mezclar estos dos sistemas, especialmente en lo tocante al tamaño de los discos duros. Para evitar ambigüedades, en este libro usaremos los símbolos KB, MB y GB para indicar 2^{10} , 2^{20} y 2^{30} bytes, respectivamente, y los símbolos Kbps, Mbps y Gbps para indicar 10^3 , 10^6 , y 10^9 bits por segundo, respectivamente.

1.11 RESUMEN

Los sistemas operativos pueden verse desde dos perspectivas: como gestores de recursos y como máquinas extendidas. Desde la perspectiva de gestor de recursos, la labor del sistema operativo consiste en administrar eficientemente las distintas partes del sistema. Desde la perspectiva de máquina extendida, la labor del sistema es proporcionar a los usuarios una máquina virtual que sea más cómoda de usar que la máquina real.

Los sistemas operativos tienen una larga historia, que va desde los días en que reemplazaron al operador, hasta los sistemas de multiprogramación modernos. Algunos puntos sobresalientes son los primeros sistemas por lotes, los sistemas de multiprogramación y los sistemas de ordenador personal.

Puesto que los sistemas operativos interactúan estrechamente con el hardware, es necesario tener ciertos conocimientos del hardware del ordenador para entenderlos. Los ordenadores están compuestos de procesadores, memorias y dispositivos de E/S. Esas partes están conectadas mediante buses.

Los conceptos básicos sobre los que se construyen todos los sistemas operativos son los conceptos de proceso, gestión de memoria, gestión de E/S, sistema de ficheros y seguridad. Trataremos cada uno de ellos en un capítulo posterior.

El corazón de cualquier sistema operativo es el conjunto de llamadas al sistema que puede llevar a cabo. Éstas nos dicen qué es lo que hace el sistema operativo en realidad. Hemos examinado cuatro grupos de llamadas al sistema para el caso de UNIX. El primer grupo de llamadas al sistema se relaciona con la creación y terminación de los procesos. El segundo grupo es para leer y escribir en ficheros. El tercer grupo es para la gestión de los directorios. El cuarto grupo contiene otras llamadas al sistema misceláneas.

Hay varias formas de estructurar los sistemas operativos. Las más comunes son como un sistema monolítico, como una jerarquía de capas, como un sistema de máquina virtual, como un exokernel o siguiendo el modelo cliente-servidor.