CAPÍTULO 8

Memoria principal

En el Capítulo 5, hemos mostrado cómo puede ser compartido el procesador por un conjunto de procesos. Como resultado de la planificación de la CPU, podemos mejorar tanto el grado de utilización del procesador como la velocidad de respuesta a los usuarios de la computadora. Para conseguir este incremento de las prestaciones debemos, sin embargo, ser capaces de mantener varios procesos en memoria; en otras palabras, debemos poder *compartir* la memoria.

En este capítulo, vamos a analizar diversas formas de gestionar la memoria. Como veremos, los algoritmos de gestión de memoria varían, desde técnicas primitivas sin soporte hardware específico a estrategias de paginación y segmentación. Cada una de las técnicas tiene sus propias ventajas y desventajas y la selección de un método de gestión de memoria para un sistema específico depende de muchos factores, y en especial del diseño *hardware* del sistema. Como veremos, muchos algoritmos requieren soporte hardware, aunque los diseños más recientes integran de manera estrecha el hardware y el sistema operativo.

OBJETIVOS DEL CAPÍTULO

- Proporcionar una descripción detallada de las diversas formas de organizar el hardware de memoria.
- Analizar diversas técnicas de gestión de memoria, incluyendo la paginación y la segmentación.
- Proporcionar una descripción detallada del procesador Intel Pentium, que soporta tanto un esquema de segmentación pura como un mecanismo de segmentación con paginación.

8.1 Fundamentos

Como vimos en el Capítulo 1, la memoria es un componente crucial para la operación de un sistema informático moderno. La memoria está compuesta de una gran matriz de palabras o bytes, cada uno con su propia dirección. La CPU extrae instrucciones de la memoria de acuerdo con el valor del contador de programa. Estas instrucciones pueden provocar operaciones adicionales de carga o de almacenamiento en direcciones de memoria específicas.

Un ciclo típico de ejecución de una instrucción procedería en primer lugar, por ejemplo, a extraer una instrucción de la memoria. Dicha instrucción se decodifica y puede hacer que se extraigan de memoria una serie de operandos. Después de haber ejecutado la instrucción con esos operandos, es posible que se almacenen los resultados de nuevo en memoria. La unidad de memoria tan sólo ve un flujo de direcciones de memoria y no sabe cómo se generan esas direcciones (mediante el contador de programa, mediante indexación, indirección, direcciones literales, etc.) ni tampoco para qué se utilizan (instrucciones o datos). Por tanto, podemos ignorar el *cómo* genera el programa las direcciones de memoria; lo único que nos interesa es la secuencia de direcciones de memoria generadas por el programa en ejecución.

Comenzaremos nuestras explicaciones hablando sobre diversas cuestiones relacionada con diferentes técnicas utilizadas para la gestión de la memoria. Entre estas cuestiones se incluyen u panorámica de los problemas hardware básicos, los mecanismos de acoplamiento de las direcc nes simbólicas de memoria a las direcciones físicas reales y los métodos existentes para distingu entre direcciones lógicas y físicas. Concluiremos con una exposición de los mecanismos para c gar y montar código dinámicamente y hablaremos también de las bibliotecas compartidas.

8.1.1 Hardware básico

La memoria principal y los registros integrados dentro del propio procesador son las únicas área de almacenamiento a las que la CPU puede acceder directamente. Hay instrucciones de máquin que toman como argumentos direcciones de memoria, pero no existe ninguna instrucción qu acepte direcciones de disco. Por tanto, todas las instrucciones en ejecución y los datos utilizado por esas instrucciones deberán encontrarse almacenados en uno de esos dispositivos de almace namiento de acceso directo. Si los datos no se encuentran en memoria, deberán llevarse hasta allí antes de que la CPU pueda operar con ellos.

Generalmente, puede accederse a los registros integrados en la CPU en un único ciclo del reloj del procesador. La mayoría de los procesadores pueden decodificar instrucciones y realizar ope raciones simples con el contenido de los registros a la velocidad de una o más operaciones por cada tic de reloj. No podemos decir lo mismo de la memoria principal, a la que se accede mediante una transacción del bus de memoria. El acceso a memoria puede requerir muchos ciclos del reloj del procesador para poderse completar, en cuyo caso el procesador necesitará normalmente **detenerse**, ya que no dispondrá de los datos requeridos para completar la instrucción que esté ejecutando. Esta situación es intolerable, debido a la gran frecuencia con la que se accede a la memoria. El remedio consiste en añadir una memoria rápida entre la CPU y la memoria principal. En la Sección 1.8.3 se describe un búfer de memoria utilizado para resolver la diferencia de velocidad; dicho búfer de memoria se denomina **caché**.

No sólo debe preocuparnos la velocidad relativa del acceso a la memoria física, sino que también debemos garantizar una correcta operación que proteja al sistema operativo de los posibles accesos por parte de los procesos de los usuarios y que también proteja a unos procesos de usuario de otros. Esta protección debe ser proporcionada por el hardware y puede implementarse de diversas formas, como veremos a lo largo del capítulo. En esta sección, vamos a esbozar una posible implementación.

Primero tenemos que asegurarnos de que cada proceso disponga de un espacio de memoria separado. Para hacer esto, debemos poder determinar el rango de direcciones legales a las que el proceso pueda acceder y garantizar también que el proceso sólo acceda a esas direcciones legales. Podemos proporcionar esta protección utilizando dos registros, usualmente una base y un límite, como se muestra en la Figura 8.1. El **registro base** almacena la dirección de memoria física legal más pequeña, mientras que el **registro límite** especifica el tamaño del rango. Por ejemplo, si el registro base contiene el valor 300040 y el registro límite es 120900, entonces el programa podrá acceder legalmente a todas las direcciones comprendidas entre 300040 y 420940 (incluyendo los dos extremos).

La protección del espacio de memoria se consigue haciendo que el hardware de la CPU compare *todas* las direcciones generadas en modo usuario con el contenido de esos registros. Cualquier intento, por parte de un programa que se esté ejecutando en modo usuario, de acceder a la memoria del sistema operativo o a la memoria de otros usuarios hará que se produzca una interrupción hacia el sistema operativo, que tratará dicho intento como un error fatal (Figura 8.2). Este esquema evita que un programa de usuario modifique (accidental o deliberadamente) el código y las estructuras de datos del sistema operativo o de otros usuarios.

Los registros base y límite sólo pueden ser cargados por el sistema operativo, que utiliza una instrucción privilegiada especial. Puesto que las instrucciones privilegiadas sólo pueden ser ejecutadas en modo *kernel* y como sólo el sistema operativo se ejecuta en modo *kernel*, únicamente el sistema operativo podrá cargar los registros base y límite. Este esquema permite al sistema operativo modificar el valor de los registros, pero evita que los programas de usuario cambien el contenido de esos registros.

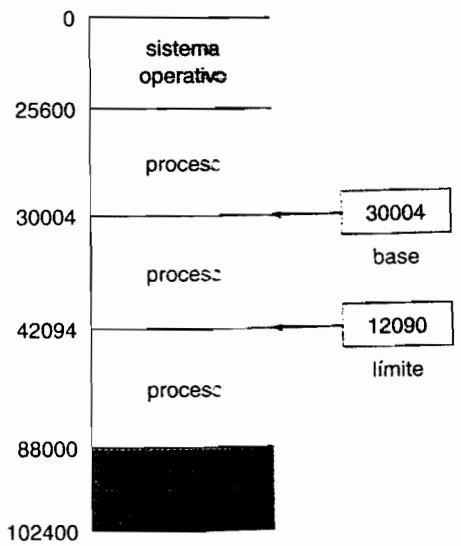

Figura 8.1 Un registro base y un registro límite definen un espacio lógico de direcciones.

El sistema operativo, que se ejecuta en modo kernel, tiene acceso no restringido a la memoria tanto del sistema operativo como de los usuarios. Esto permite al sistema operativo cargar los programas de los usuarios en la memoria de los usuarios, volcar dichos programas en caso de error, leer y modificar parámetros de las llamadas al sistema, etc.

8.1.2 Reasignación de direcciones

Usualmente, los programas residen en un disco en forma de archivos ejecutables binarios. Para poder ejecutarse, un programa deberá ser cargado en memoria y colocado dentro de un proceso. Dependiendo del mecanismo de gestión de memoria que se utilice, el proceso puede desplazarse entre disco y memoria durante su ejecución. Los procesos del disco que estén esperando a ser cargados en memoria para su ejecución forman lo que se denomina **cola de entrada**.

El procedimiento normal consiste en seleccionar uno de los procesos de la cola de entrada y cargar dicho proceso en memoria. A medida que se ejecuta el proceso, éste accede a las instrucciones y datos contenidos en la memoria. Eventualmente, el proceso terminará su ejecución y su espacio de memoria será declarado como disponible.

La mayoría de los sistemas permiten que un proceso de usuario resida en cualquier parte de la memoria física. Así, aunque el espacio de direcciones de la computadora comience en 00000, la

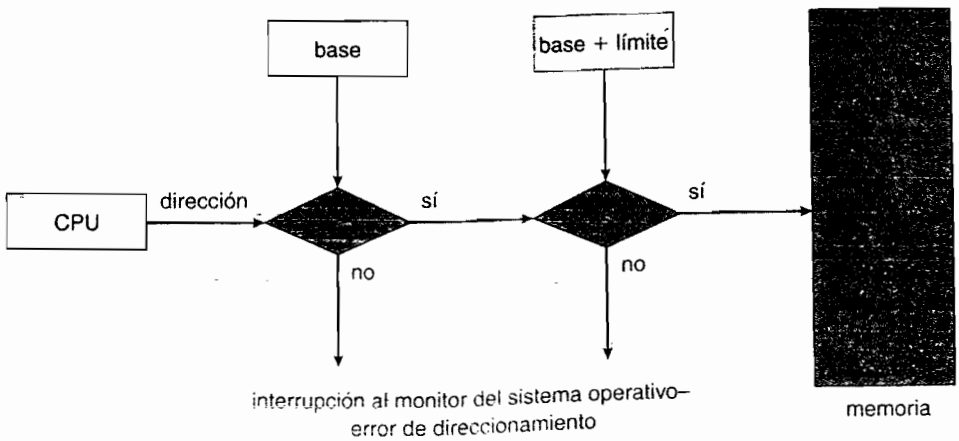

Figura 8.2 Protección hardware de las direcciones, utilizando un registro base y un registro límite.

primera dirección del proceso de usuario no tiene por qué ser 00000. Esta técnica afecta a las direcciones que el programa de usuario puede utilizar. En la mayoría de los casos, el programa de usuario tendrá que recorrer varios pasos (algunos de los cuales son opcionales) antes de ser ejecutado (Figura 8.3). A lo largo de estos pasos, las direcciones pueden representarse de diferentes formas. Las direcciones del programa fuente son generalmente simbólicas (como por ejemplo, *saldo*). Normalmente, un compilador se encargará de **reasignar** estas direcciones simbólicas a direcciones reubicables (como por ejemplo, "14 bytes a partir del comienzo de este módulo"). El editor de montaje o cargador se encargará, a su vez, de reasignar las direcciones reubicables a direcciones absolutas (como por ejemplo, 74014). Cada operación de reasignación constituye una relación de un espacio de direcciones a otro.

Clásicamente, la reasignación de las instrucciones y los datos a direcciones de memoria puede realizarse en cualquiera de los pasos:

- **Tiempo de compilación**. Si sabemos en el momento de realizar la compilación dónde va a residir el proceso en memoria, podremos generar **código absoluto.** Por ejemplo, si sabemos que un proceso de usuario va a residir en una zona de memoria que comienza en la ubicación *R*, el código generado por el compilador comenzará en dicha ubicación y se extenderá a partir de ahí. Si la ubicación inicial cambiase en algún instante posterior, entonces sería

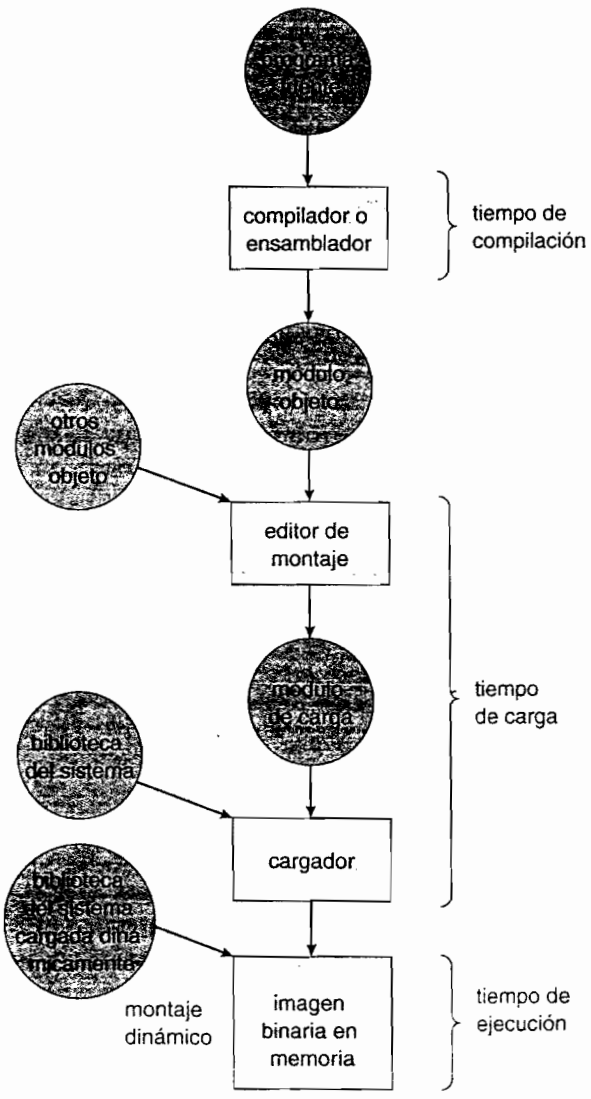

Figura 8.3 Pasos en el procesamiento de un programa de usuario.

necesario recompilar ese código. Los programas en formato .COM de MS-DOS se acoplan en tiempo de compilación.

- **Tiempo de carga.** Si no conocemos en tiempo de compilación dónde va a residir el proceso en memoria, el compilador deberá generar **código reubicable**. En este caso, se retarda la reasignación final hasta el momento de la carga. Si cambia la dirección inicial, tan sólo es necesario volver a cargar el código de usuario para incorporar el valor modificado.
- **Tiempo de ejecución.** Si el proceso puede desplazarse durante su ejecución desde un segmento de memoria a otro, entonces es necesario retardar la reasignación hasta el instante de la ejecución. Para que este esquema pueda funcionar, será preciso disponer de hardware especial, como veremos en la Sección 8.1.3. La mayoría de los sistemas operativos de propósito general utilizan este método.

Buena parte de este capítulo está dedicada a mostrar cómo pueden implementarse estos diversos esquemas de reasignación en un sistema informático de manera efectiva, y a analizar el soporte hardware adecuado para cada uno.

8.1.3 Espacios de direcciones lógico y físico

Una dirección generada por la CPU se denomina comúnmente **dirección lógica**, mientras que una dirección vista por la unidad de memoria (es decir, la que se carga en el **registro de direcciones de memoria** de la memoria) se denomina comúnmente **dirección física**.

Los métodos de reasignación en tiempo de compilación y en tiempo de carga generan direcciones lógicas y físicas idénticas. Sin embargo, el esquema de reasignación de direcciones en tiempo de ejecución hace que las direcciones lógica y física difieran. En este caso, usualmente decimos que la dirección lógica es una **dirección virtual**. A lo largo de este texto, utilizaremos los términos *dirección lógica* y *dirección virtual* de manera intercambiable. El conjunto de todas las direcciones lógicas generadas por un programa es lo que se denomina un **espacio de direcciones lógicas**; el conjunto de todas las direcciones físicas correspondientes a estas direcciones lógicas es un **espacio de direcciones físicas**. Así, en el esquema de reasignación de direcciones en tiempo de ejecución, decimos que los espacios de direcciones lógicas y físicas difieren.

La correspondencia entre direcciones virtuales y físicas en tiempo de ejecución es establecida por un dispositivo hardware que se denomina **unidad de gestión de memoria** (MMU, memory-management unit). Podemos seleccionar entre varios métodos distintos para establecer esta correspondencia, como veremos en las Secciones 8.3 a 8.7. Por el momento, vamos a ilustrar esta operación de asociación mediante un esquema MMU simple, que es una generalización del esquema de registro base descrita en la Sección 8.1. El registro base se denominará ahora **registro de reubicación**. El valor contenido en el registro de reubicación *suma* a todas las direcciones generadas por un proceso de usuario en el momento de enviarlas a memoria (véase la Figura 8.4.). Por ejemplo, si la base se encuentra en la dirección 14000, cualquier intento del usuario de direccionar la posición de memoria cero se reubicará dinámicamente en la dirección 14000; un acceso a la ubicación 346 se convertirá en la ubicación 14346. El sistema operativo MS-DOS que se ejecuta sobre la familia de procesadores Intel 80x86 utiliza cuatro registros de reubicación a la hora de cargar y ejecutar procesos.

El programa de usuario nunca ve las direcciones físicas *reales*. El programa puede crear un puntero a la ubicación 346, almacenarlo en memoria, manipularlo y compararlo con otras direcciones, siempre como el número 346. Sólo cuando se lo utiliza como dirección de memoria (por ejemplo, en una operación de lectura o escritura indirecta) se producirá la reubicación en relación con el registro base. El programa de usuario maneja direcciones *lógicas* y el hardware de conversión (mapeo) de memoria convierte esas direcciones lógicas en direcciones físicas. Esta forma de acoplamiento en tiempo de ejecución ya fue expuesta en la Sección 8.1.2. La ubicación final de una dirección de memoria referenciada no se determina hasta que se realiza esa referencia.

Ahora tenemos dos tipos diferentes de direcciones: direcciones lógicas (en el rango comprendido entre 0 y *max*) y direcciones físicas (en el rango comprendido entre $R + 0$ y $R + max$ para un valor base igual a R). El usuario sólo genera direcciones lógicas y piensa que el proceso se ejecu-

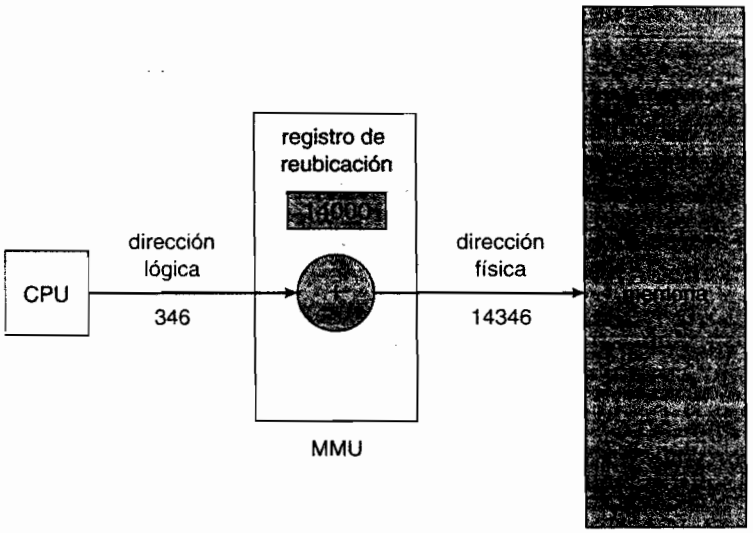

Figura 8.4 Reubicación dinámica mediante un registro de reubicación.

ta en las ubicaciones comprendidas entre 0 y *max*. El programa de usuario suministra direcciones lógicas y estas direcciones lógicas deben ser convertidas en direcciones físicas antes de utilizarlas.

El concepto de un *espacio de direcciones lógicas* que se acopla a un *espacio de direcciones físicas* separado resulta crucial para una adecuada gestión de la memoria.

8.1.4 Carga dinámica

En las explicaciones que hemos dado hasta el momento, todo el programa y todos los datos de un proceso deben encontrase en memoria física para que ese proceso pueda ejecutarse. En consecuencia, el tamaño de un proceso está limitado por el tamaño de la memoria física. Para obtener una mejor utilización del espacio de memoria, podemos utilizar un mecanismo de **carga dinámica.** Con la carga dinámica, una rutina no se carga hasta que se la invoca; todas las rutinas se mantienen en disco en un formato de carga reubicable. Según este método, el programa principal se carga en la memoria y se ejecuta. Cuando una rutina necesita llamar a otra rutina, la rutina que realiza la invocación comprueba primero si la otra ya ha sido cargada, si no es así, se invoca el cargador de montaje reubicable para que cargue en memoria la rutina deseada y para que actualice las tablas de direcciones del programa con el fin de reflejar este cambio. Después, se pasa el control a la rutina recién cargada.

La ventaja del mecanismo de carga dinámica es que una rutina no utilizada no se cargará nunca en memoria. Este método resulta particularmente útil cuando se necesitan grandes cantidades de código para gestionar casos que sólo ocurren de manera infrecuente, como por ejemplo rutinas de error. En este caso, aunque el tamaño total del programa pueda ser grande, la porción que se utilice (y que por tanto se cargue) puede ser mucho más pequeña.

El mecanismo de carga dinámica no requiere de ningún soporte especial por parte del sistema operativo. Es responsabilidad de los usuarios diseñar sus programas para poder aprovechar dicho método. Sin embargo, los sistemas operativos pueden ayudar al programador proporcionándole rutinas de biblioteca que implementen el mecanismo de carga dinámica.

8.1.5 Montaje dinámico y bibliotecas compartidas

La Figura 8.3 muestra también **bibliotecas de montaje dinámico**. Algunos sistemas operativos sólo permiten el **montaje estático**, mediante el cual las bibliotecas de lenguaje del sistema se tratan como cualquier otro módulo objeto y son integradas por el cargador dentro de la imagen binaria del programa. El concepto de montaje binario es similar al de carga dinámica, aunque en este caso lo que se pospone hasta el momento de la ejecución es el montaje, en lugar de la carga. Esta

funcionalidad suele emplearse con las bibliotecas del sistema, como por ejemplo las bibliotecas de subrutinas del lenguaje. Utilizando este mecanismo, cada programa de un sistema deberá incluir una copia de su biblioteca de lenguaje (o al menos de las rutinas a las que haga referencia el programa) dentro de la imagen ejecutable. Este requisito hace que se desperdicie tanto espacio de disco como memoria principal.

Con el montaje dinámico, se incluye un *stub* dentro de la imagen binaria para cada referencia a una rutina de biblioteca. El stub es un pequeño fragmento de código que indica cómo localizar la rutina adecuada de biblioteca residente en memoria o cómo cargar la biblioteca si esa rutina no está todavía presente. Cuando se ejecuta el stub, éste comprueba si la rutina necesaria ya se encuentra en memoria; si no es así, el programa carga en memoria la rutina. En cualquiera de los casos, el stub se sustituye así mismo por la dirección de la rutina y ejecuta la rutina. Así, la siguiente vez que se ejecute ese segmento de código concreto, se ejecutará directamente la rutina de biblioteca, sin tener que realizar de nuevo el montaje dinámico. Con este mecanismo, todos los procesos que utilicen una determinada biblioteca de lenguaje sólo necesitan ejecutar una copia del código de la biblioteca.

Esta funcionalidad puede ampliarse a las actualizaciones de las bibliotecas de código (como por ejemplo las destinadas a corregir errores). Puede sustituirse una biblioteca por una nueva versión y todos los programas que hagan referencia a la biblioteca emplearán automáticamente la versión más reciente. Sin el mecanismo de montaje dinámico, sería necesario volver a montar todos esos programas para poder acceder a la nueva biblioteca. Para que los programas no ejecuten accidentalmente versiones nuevas e incompatibles de las bibliotecas, suele incluirse información de versión tanto en el programa como en la biblioteca. Puede haber más de una versión de una biblioteca cargada en memoria y cada programa utilizará su información de versión para decidir qué copia de la biblioteca hay que utilizar. Los cambios de menor entidad retendrán el mismo número de versión, mientras que para los cambios de mayor entidad se incrementará ese número. De este modo, sólo los programas que se compilen con la nueva versión de la biblioteca se verán afectados por los cambios incompatibles incorporados en ella. Otros programas montados antes de que se instalara la nueva biblioteca continuarán utilizando la antigua. Este sistema se conoce también con el nombre de mecanismo de **bibliotecas compartidas**.

A diferencia de la carga dinámica, el montaje dinámico suele requerir algo de ayuda por parte del sistema operativo. Si los procesos de la memoria están protegidos unos de otros, entonces el sistema operativo será la única entidad que pueda comprobar si la rutina necesaria se encuentra dentro del espacio de memoria de otro proceso y será también la única entidad que pueda permitir a múltiples procesos acceder a las mismas direcciones de memoria. Hablaremos más en detalle de este concepto cuando analicemos el mecanismo de paginación en la Sección 8.4.

8.2 Intercambio

Un proceso debe estar en memoria para ser ejecutado. Sin embargo, los procesos pueden ser **intercambiados** temporalmente, sacándolos de la memoria y almacenándolos en un **almacén de respaldo** y volviéndolos a llevar luego a memoria para continuar su ejecución. Por ejemplo, suponga que estamos utilizando un entorno de multiprogramación con un algoritmo de planificación de CPU basado en turnos. Cuando termina un cuanto de tiempo, el gestor de memoria comienza a sacar de ésta el proceso que acaba de terminar y a cargar en el espacio de memoria liberado por otro proceso (Figura 8.5). Mientras tanto, el planificador de la CPU asignará un cuanto de tiempo a algún otro proceso que ya se encuentre en memoria. Cada vez que un proceso termine su cuanto asignado, se intercambiará por otro proceso. Idealmente, el gestor de memoria puede intercambiar los procesos con la suficiente rapidez como para que haya siempre algunos procesos en memoria, listos para ejecutarse, cuando el planificador de la CPU quiera asignar el procesador a otra tarea. Además, el cuanto debe ser lo suficientemente grande como para que pueda realizarse una cantidad razonable de cálculos entre una operación de intercambios y la siguiente.

Para los algoritmos de planificación con prioridad se utiliza una variante de esta política de intercambio. Si llega un proceso de mayor prioridad y ese proceso desea ser servido, el gestor de memoria puede descargar el proceso de menor prioridad y, a continuación, cargar y ejecutar