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 capitulo 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.

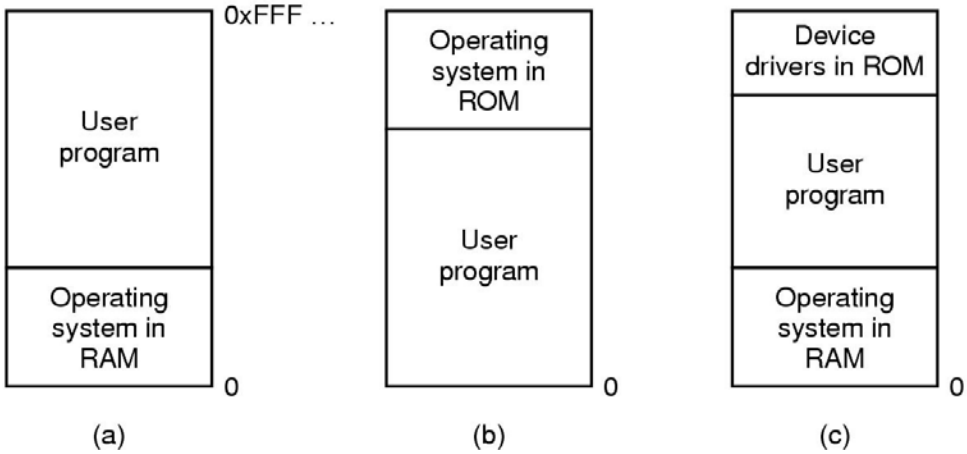

Figura 4-1. Tres formas de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

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.

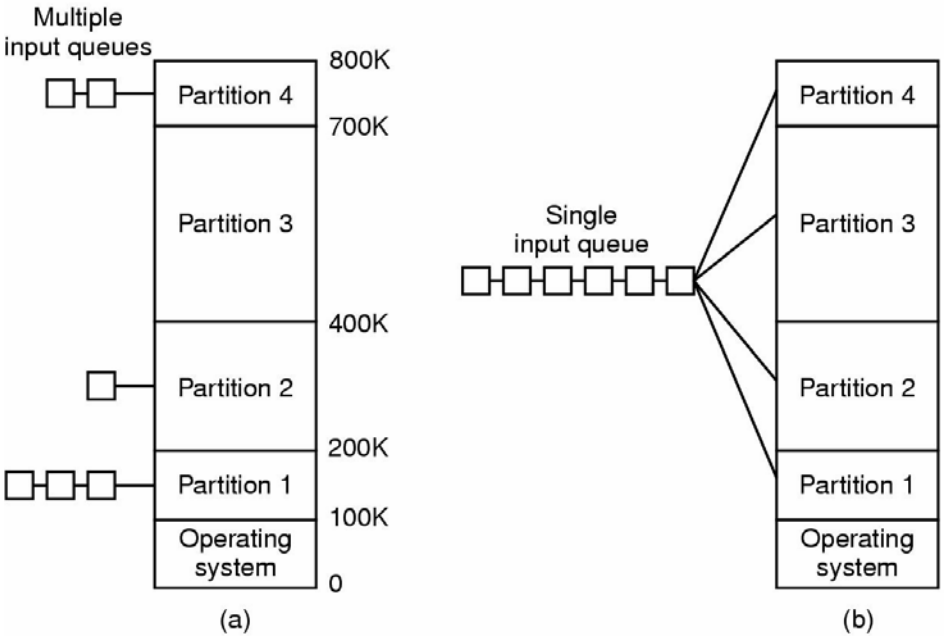

Figura 4-2. (a) Particiones de memoria fijas con colas de entrada separadas para cada partición. (b) Particiones de memoria fijas con una única cola de entrada.

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 ejecutasen 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**.

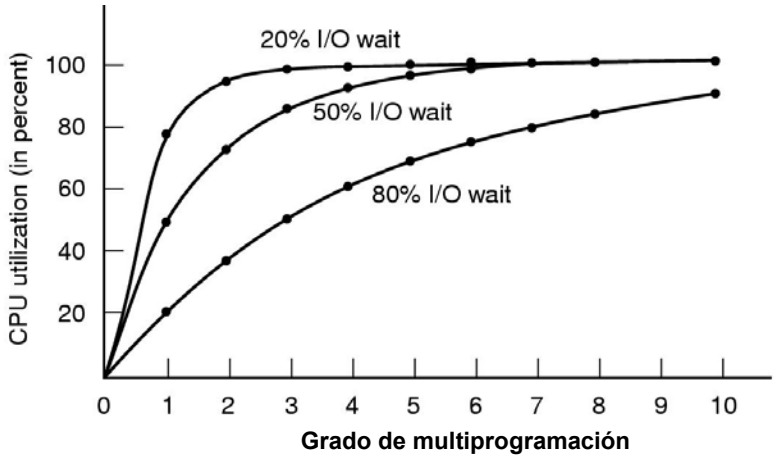

Figura 4-3. Utilización de la CPU en función del número de procesos en la memoria.

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.

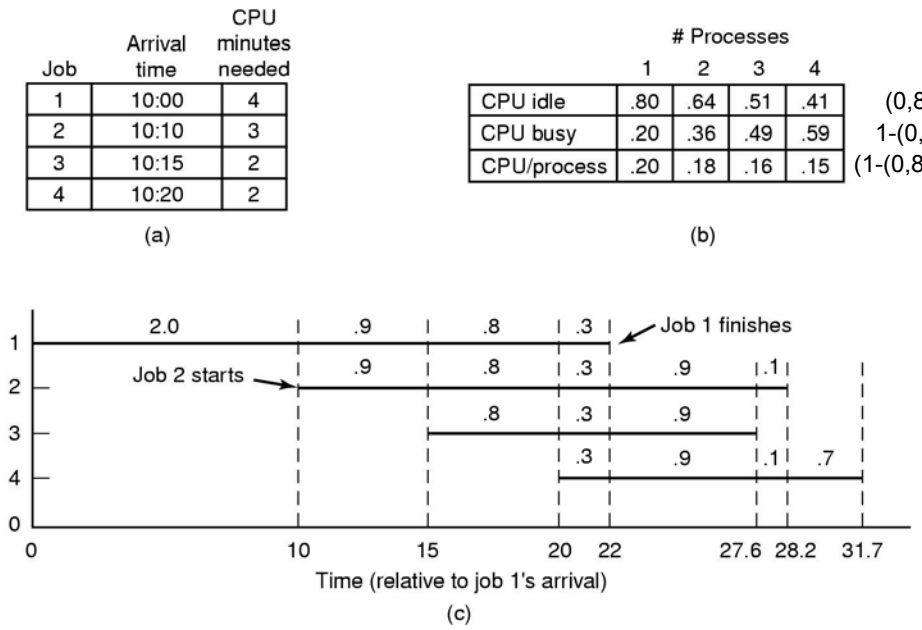

Figura 4-4. (a) Llegada y requerimientos de CPU de cuatro trabajos. (b) Utilización de la CPU para 1 a 4 procesos con el 80% de espera por E/S. (c) Secuencia de sucesos a medida que los trabajos llegan y terminan. Los números por encima de las líneas horizontales indican cuanto tiempo de CPU, en minutos,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

reubicación dinámica

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.

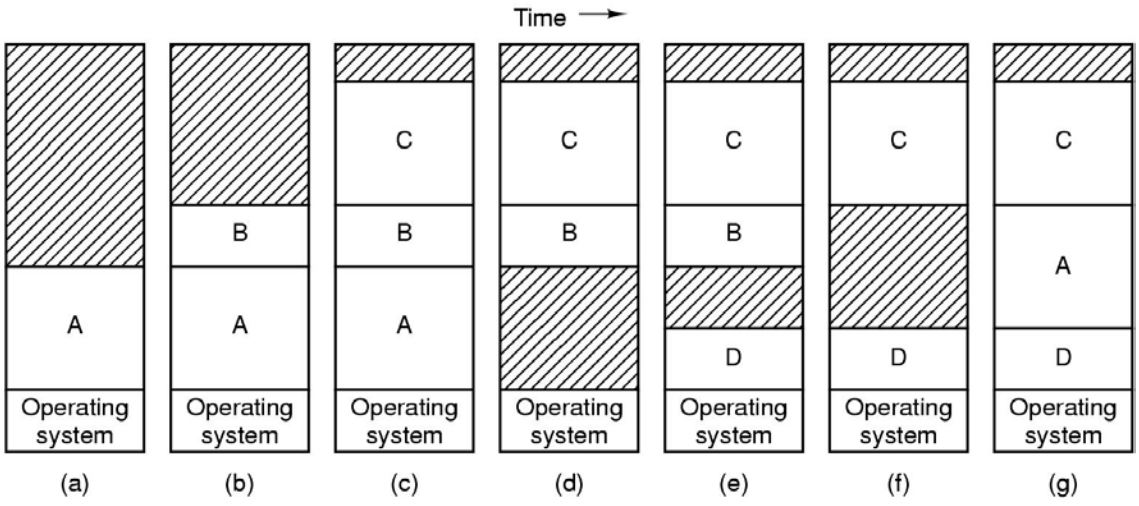

Figura 4-5. La asignación de memoria cambia a medida que los procesos entran en la memoria y salen de ella. Las regiones sombreadas representan memoria no utilizada.

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.

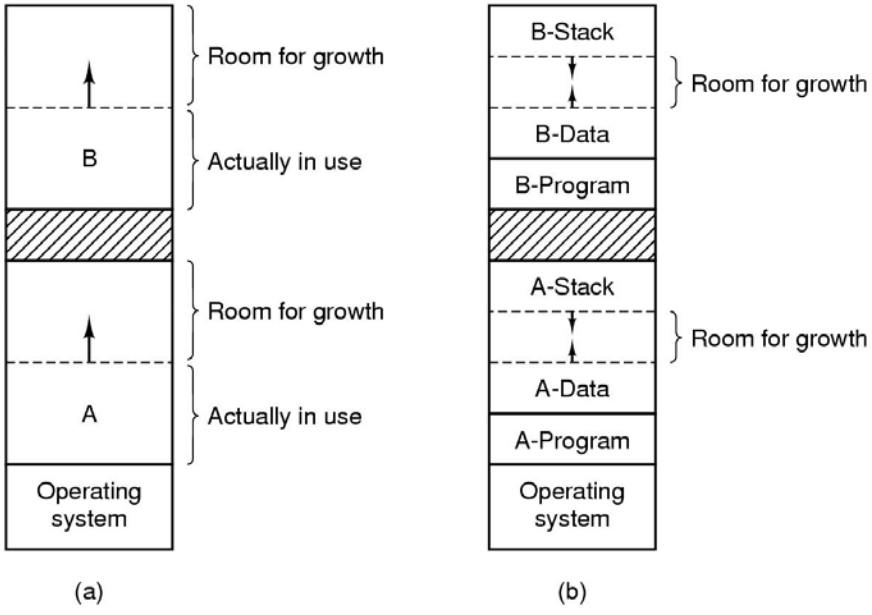

Figura 4-6. (a) Asignación de espacio para un segmento de datos que puede crecer. (b) Asignación de espacio para una pila y un segmento de datos que pueden crecer.

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.

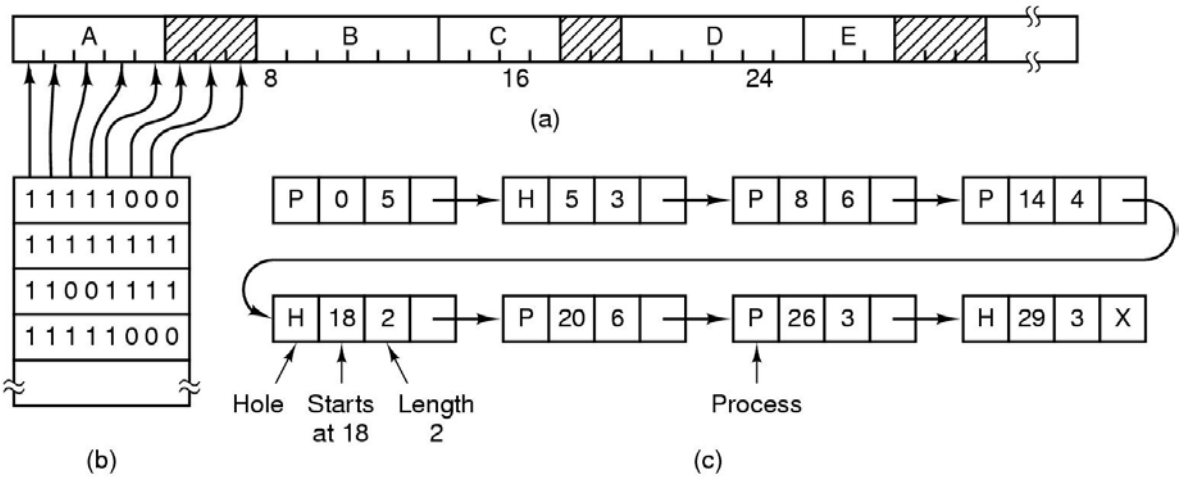

Figura 4-7. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas pequeñas corresponden a las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están desocupadas. (b) El mapa de bits correspondiente. (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una cuestión de diseño importante. Cuánto 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 32*n* 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 bloquess 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.

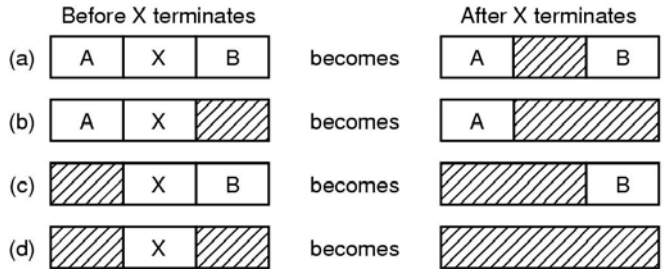

Figura 4-8. Cuatro combinaciones de vecinos para el proceso *X* que termina.

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.

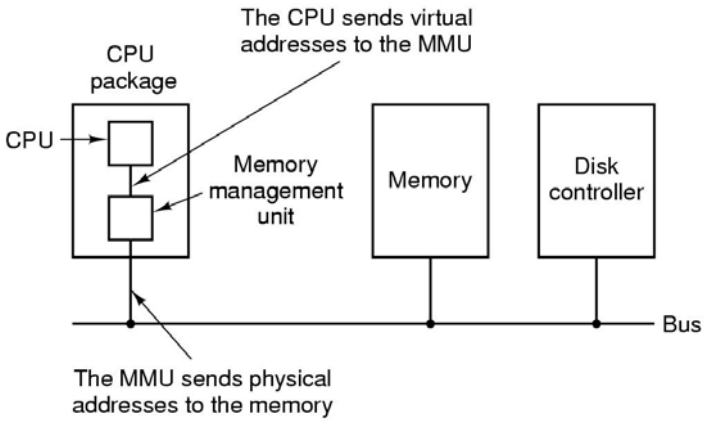

Figura 4-9. Posición y función de la MMU. Aquí se muestra la MMU formando parte del chip de la CPU porque es lo más común en la actualidad. Sin embargo, lógicamente podría ser un chip aparte y en el pasado lo era.

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.

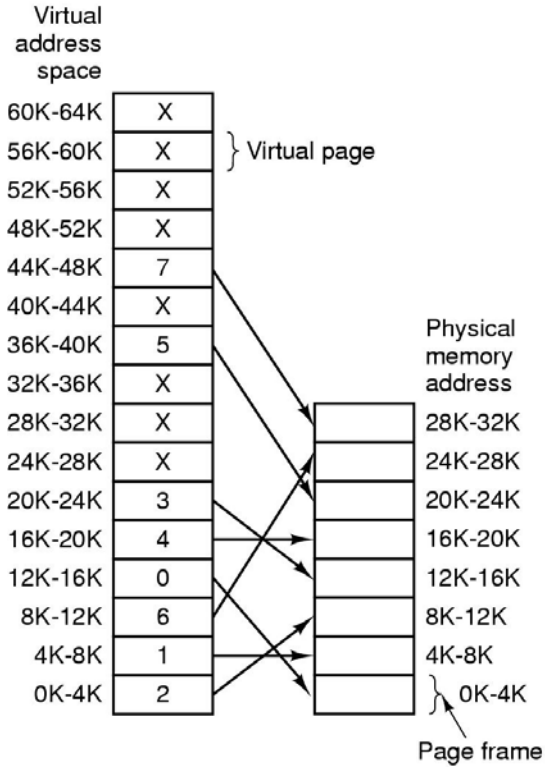

Figura 4-10. La relación entre las direcciones virtuales y las direcciones de la memoria física viene dada por la tabla de páginas.

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 vuelva 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 (0010000000000100 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.

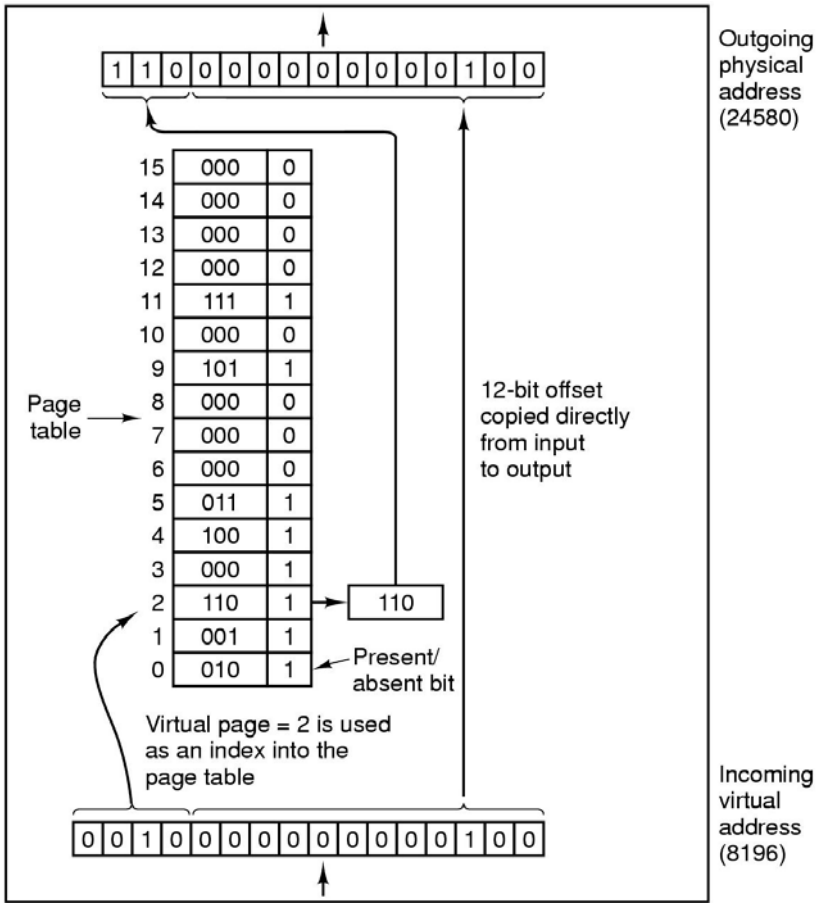

Figura 4-11. Funcionamiento interno de la MMU con 16 páginas de 4 KB.

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.

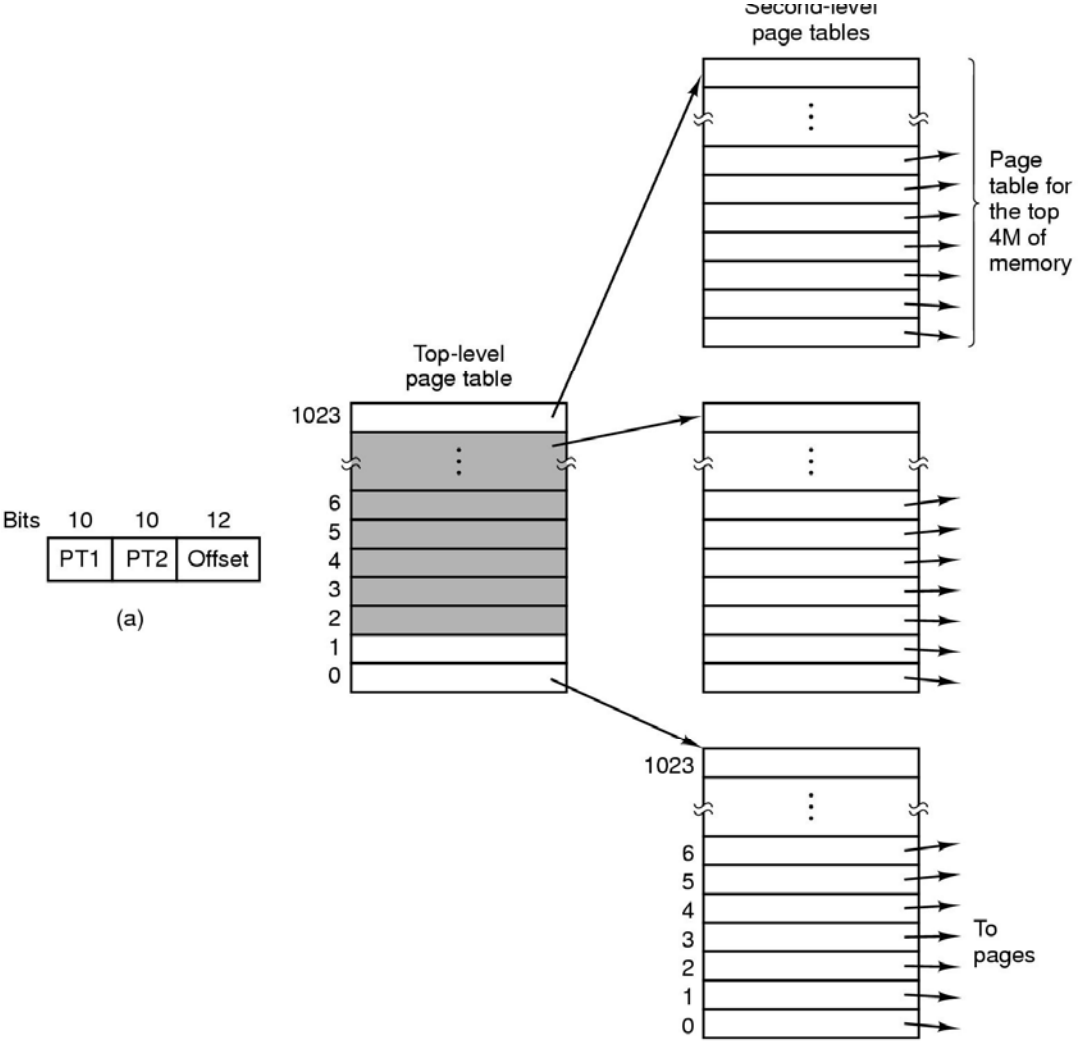

Figura 4-12. (a) Dirección de 32 bits con dos campos de tabla de páginas. (b) Tabla de páginas de dos niveles.

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 percatará 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.

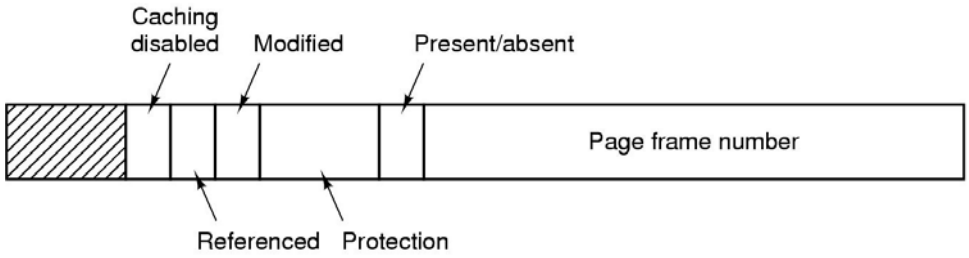

Figura 4-13. Una entrada típica de la tabla de páginas.

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 *víctima* 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.

Falta de TLB

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).

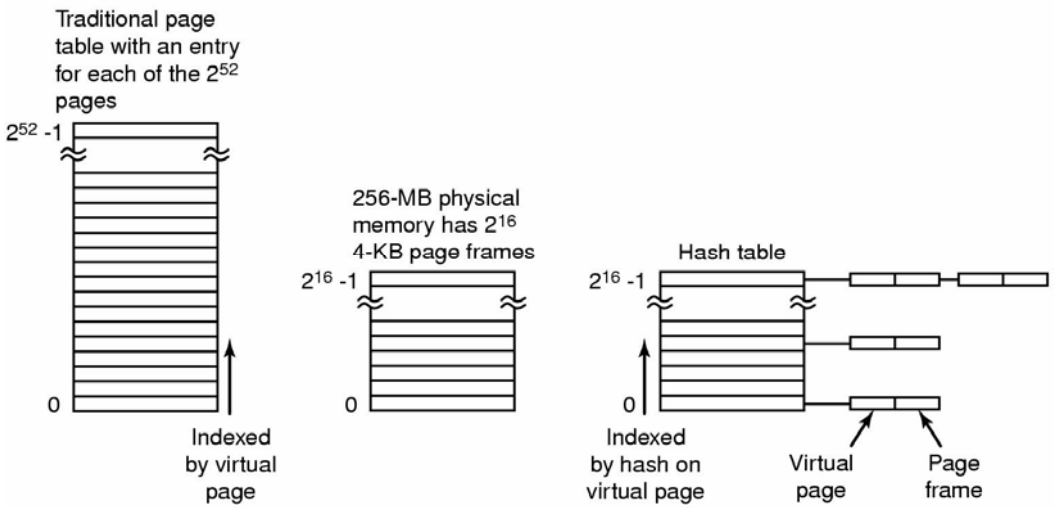

Figura 4-15. Comparación entre una tabla de páginas tradicional y una tabla de páginas invertida.

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 *víctima* 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 referencien 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*.

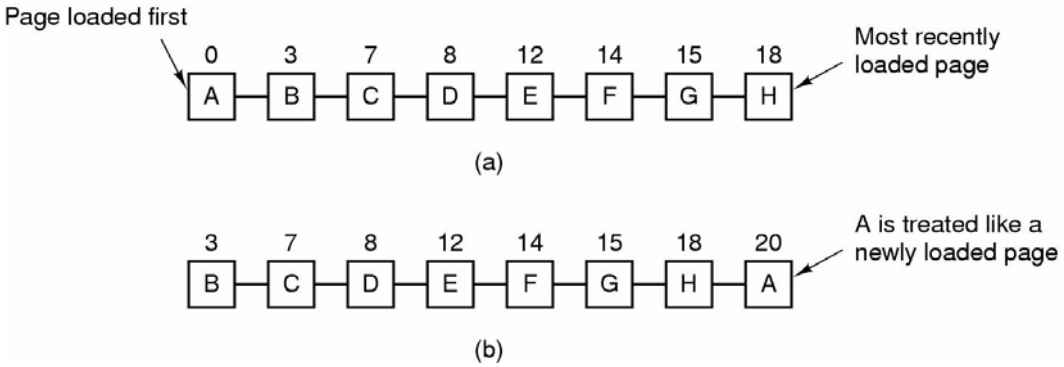

Figura 4-16. Funcionamiento del algoritmo de la segunda oportunidad. (a) Páginas en orden FIFO. (b) Lista de páginas si se produce una falta de página en el instante 20 y el bit *R* de la página *A* está activado. Los números que aparecen encima de las páginas representan sus instantes de carga.

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 ineficiente 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.

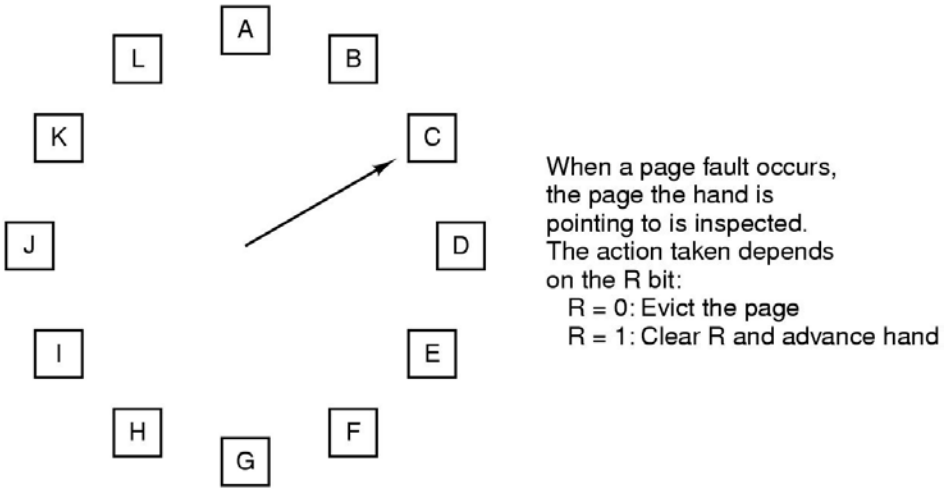

Figura 4-17. El algoritmo de sustitución de páginas del reloj.

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 reinsertarla 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.

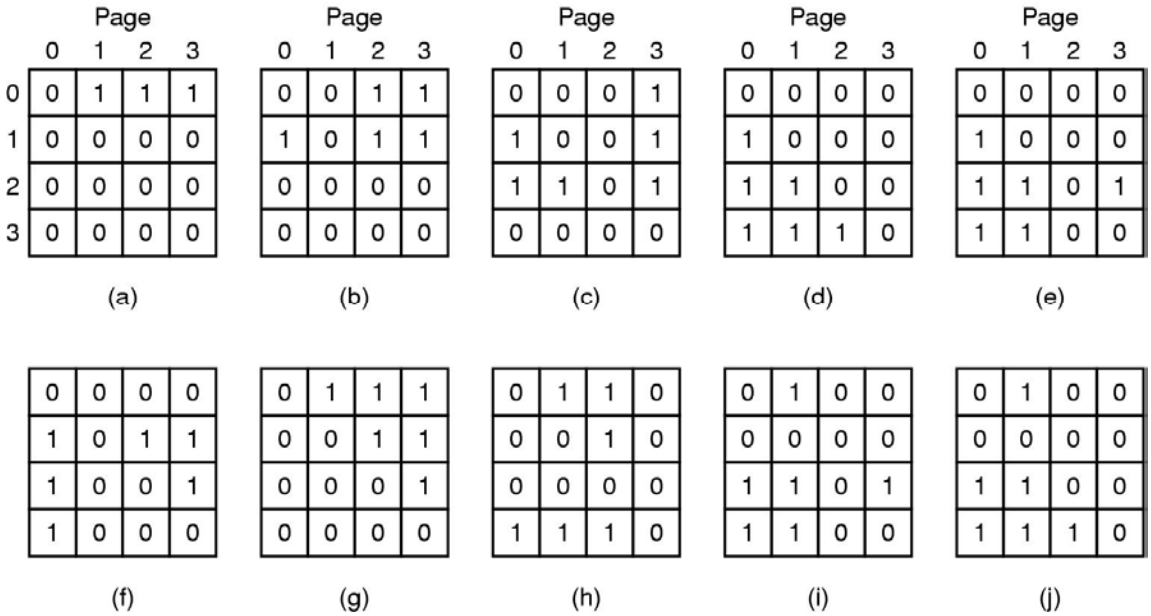

Figura 4-18. LRU utilizando una matriz cuando se hace referencia a las páginas en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

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.

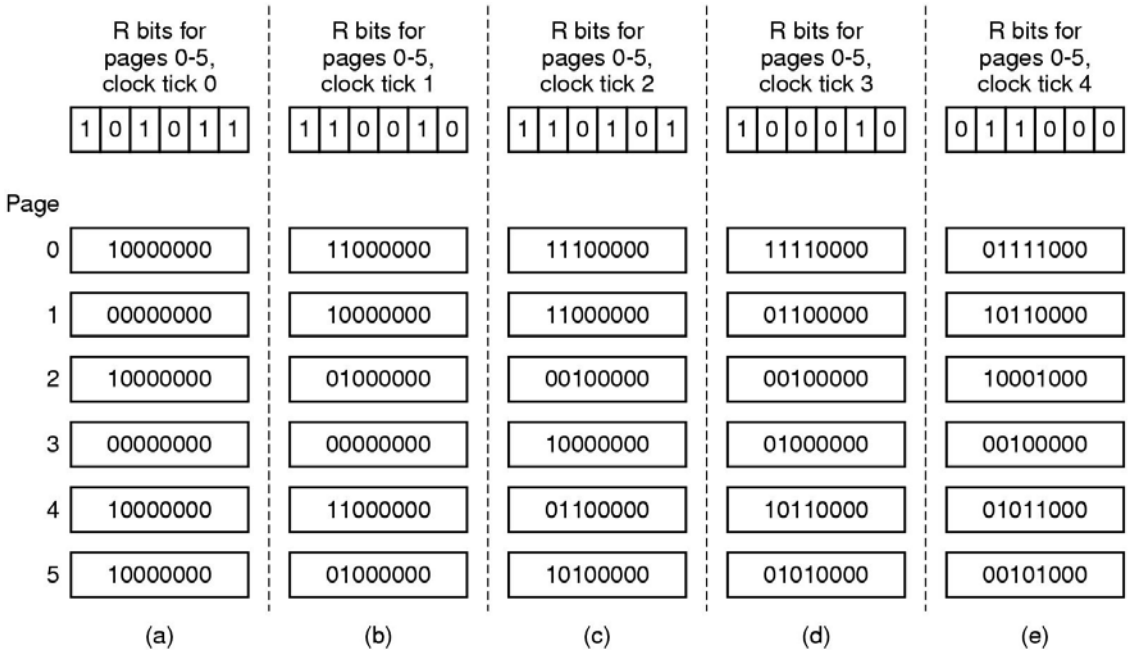

Figura 4-19. El algoritmo de envejecimiento simula por software al LRU. Se muestran seis páginas durante cinco tics de reloj. Los cinco tics se representan de la (a) a la (e).

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, desperdiciando 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.

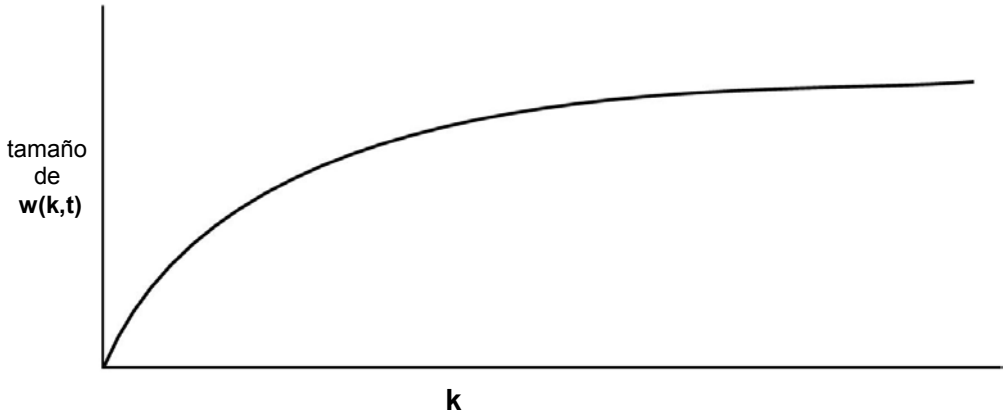

Figura 4-20. El conjunto de trabajo es el conjunto de las páginas que se han utilizado en las k últimas referencias a la memoria. La función $w(k, t)$ representa el conjunto de trabajo en el instante t.

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*).

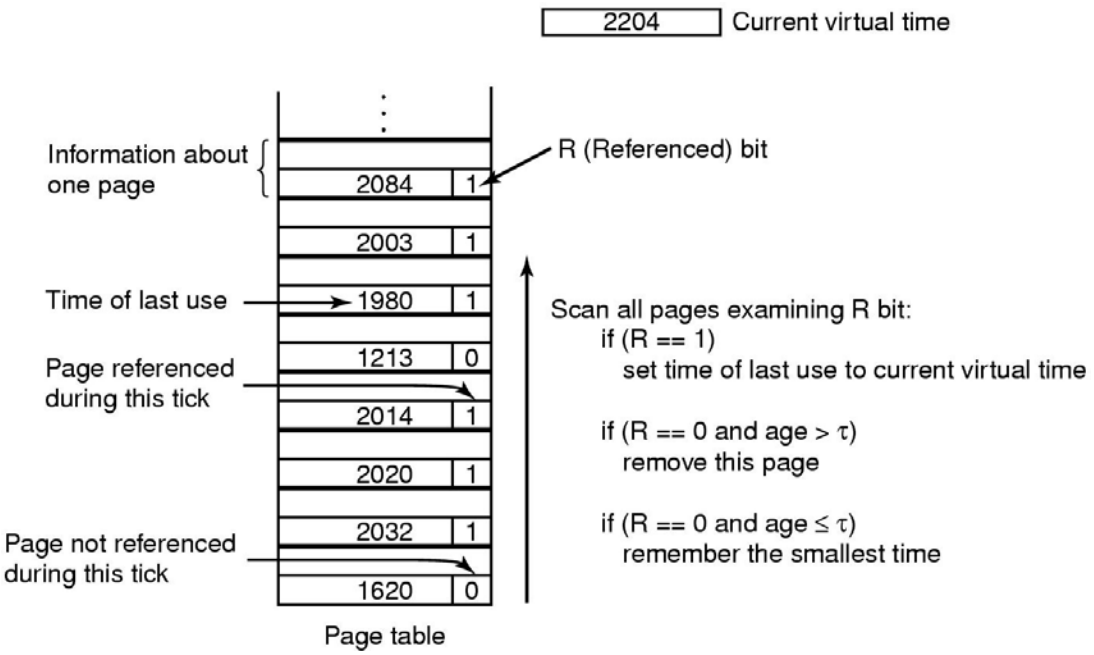

Figura 4-21. El algoritmo de sustitución del conjunto de trabajo.

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).

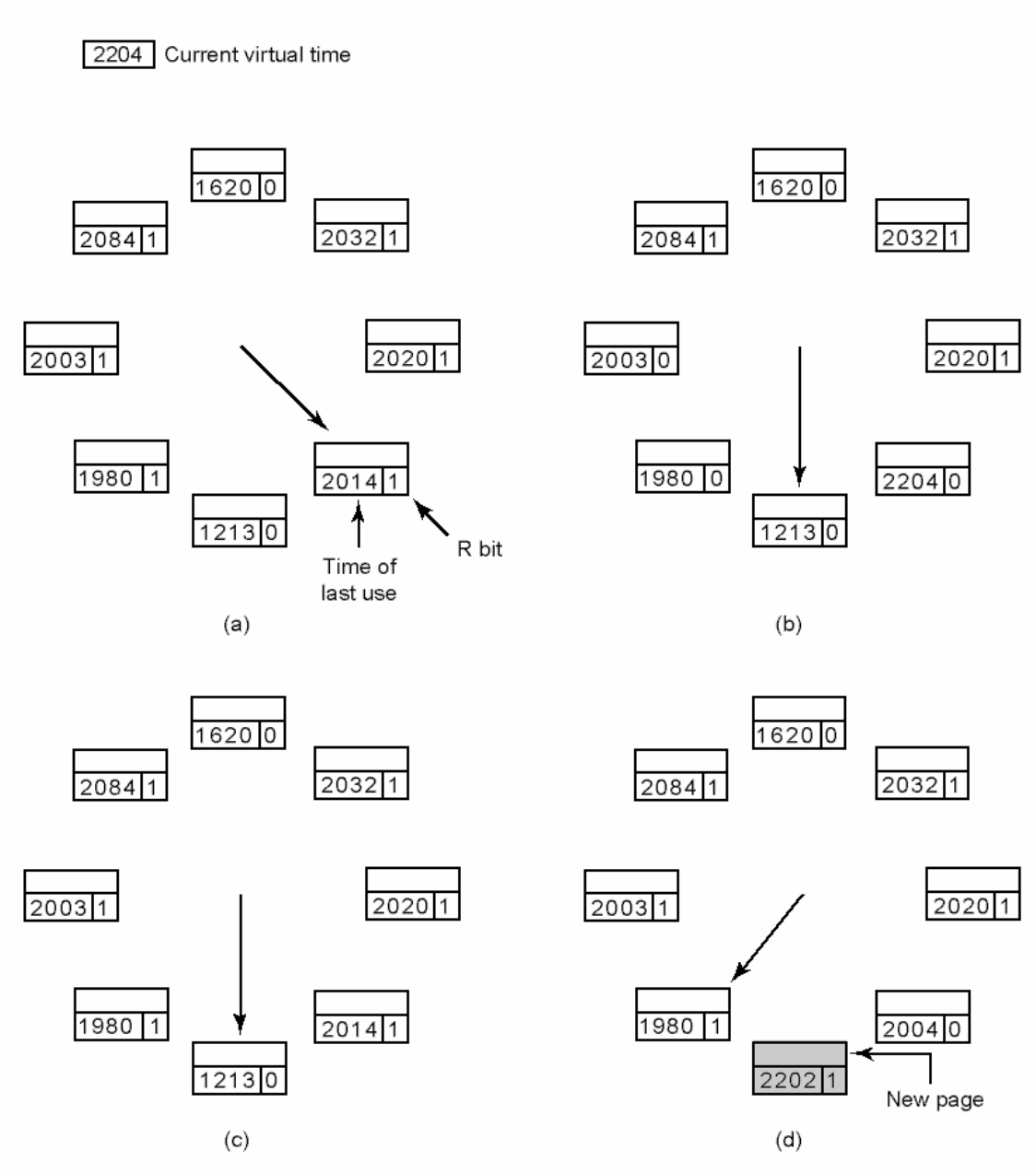

Figura 4-22. Funcionamiento del algoritmo WSClock. (a) y (b) dan un ejemplo de lo que sucede cuando $R = 1$. (c) y (d) dan un ejemplo de $R = 0$.

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.

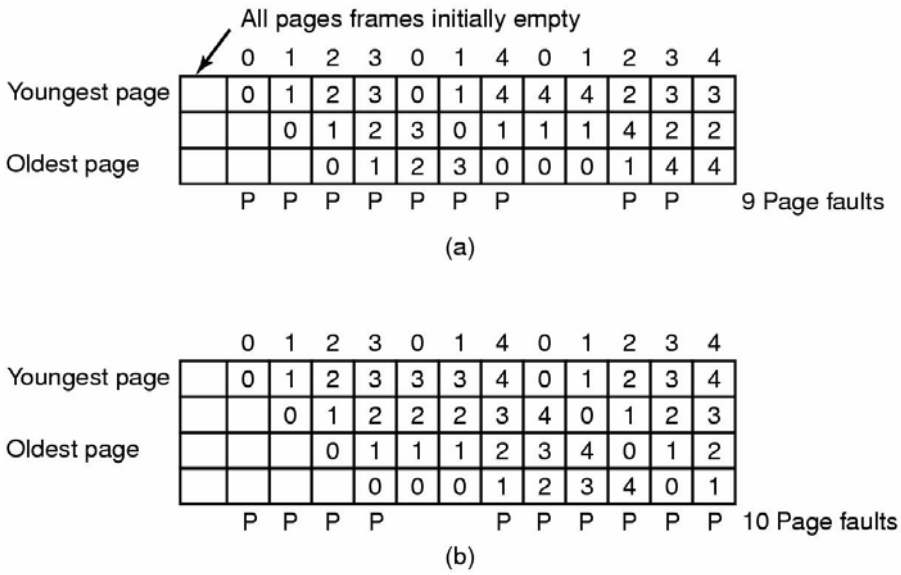

Figura 4-24. Anomalía de Belady. (a) FIFO con tres marcos de página. (b) FIFO con cuatro marcos de página. Las *P*s indican las referencias que provocan faltas de página.

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 consúltese 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.

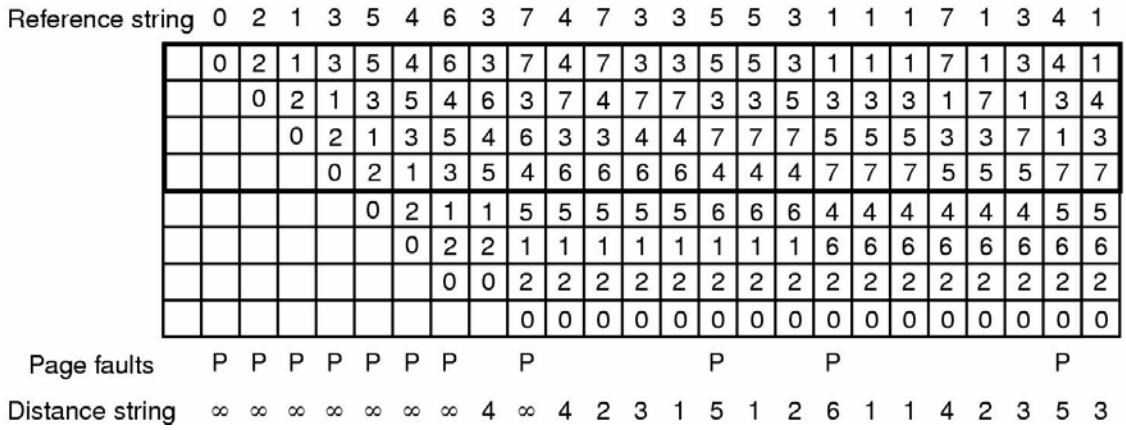

Figura 4-25. Estado del array *M*, después de procesarse cada elemento de la serie de referencias. La distancia en la serie de referencias se explicará en la siguiente sección.

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.

1

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.

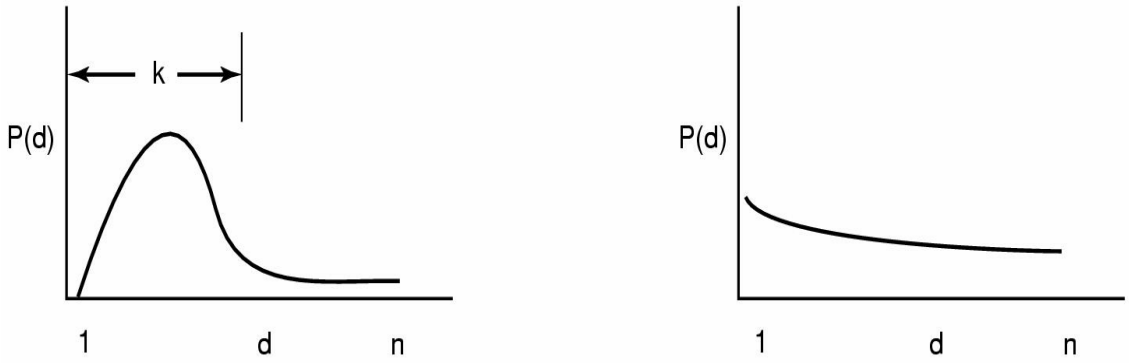

Figura 4-26. Funciones de densidad de probabilidad para dos distancias en series de referencias hipotéticas.

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.

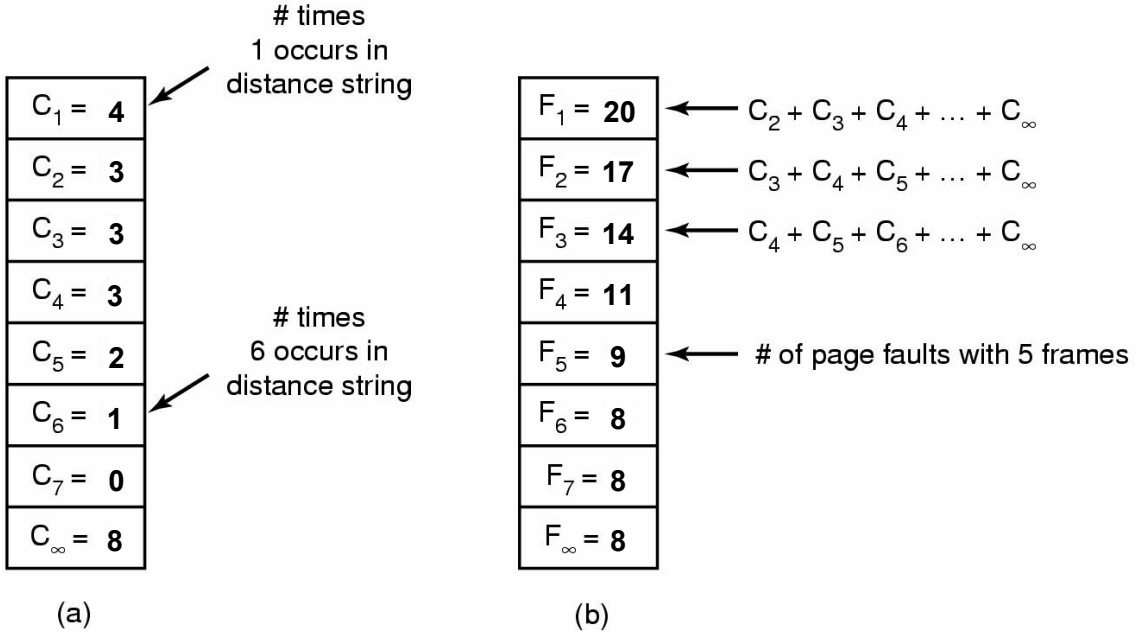

Figura 4-27. Cálculo de la tasa de faltas de página a partir de la serie de distancias. (a) El vector *C*. (b) El vector *F*.

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 *A5*, 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 *B3* 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.

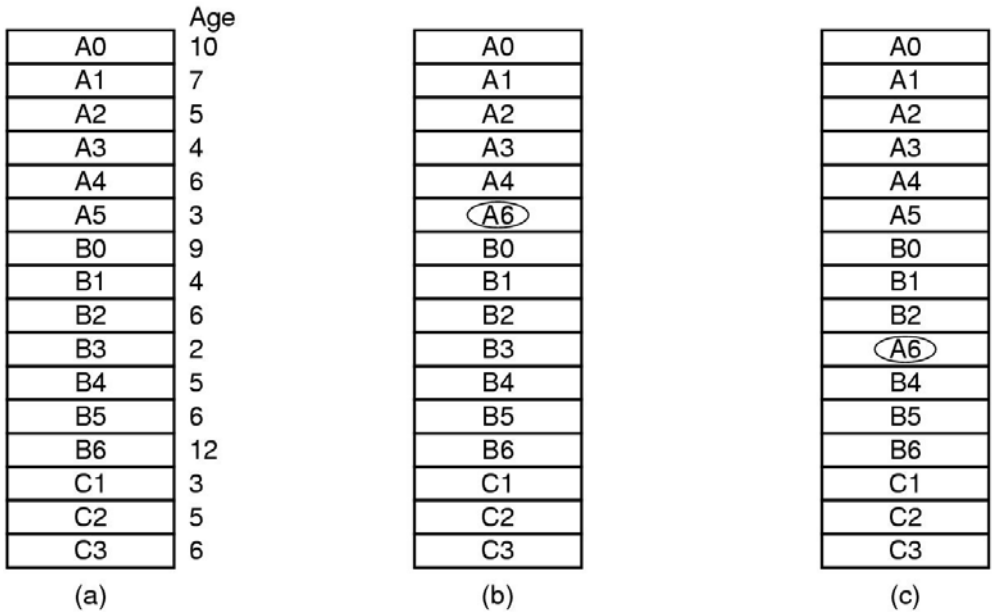

Figura 4-28. Sustitución de páginas local y global. (a) Configuración original. (b) Sustitución de páginas local. (c) Sustitución de páginas global.

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.

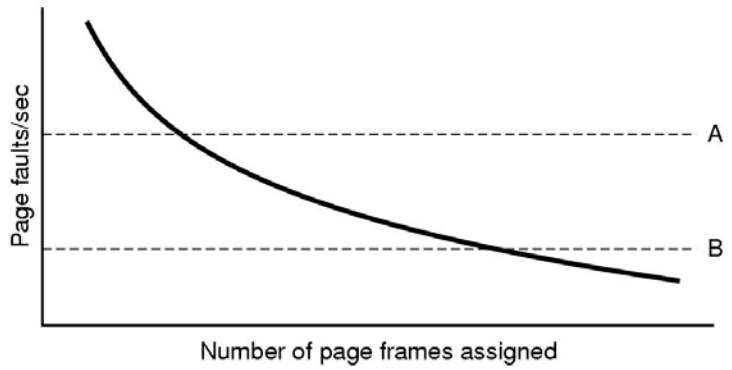

Figura 4-29. La tasa de faltas de página como una función del número de marcos de página asignados.

Medir la tasa de faltas de página es sencillo: basta con contar el número de faltas de página por segundo, calculando quizá 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 repartiéndose 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 periodos 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 $\times$ 10 milisegundos para cargar 64 páginas de 512 bytes, pero sólo 4 $\times$ 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, quizá 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.