6

EL NIVEL DE MÁQUINA DE SISTEMA OPERATIVO

El tema de este libro es que una computadora moderna consiste en una serie de niveles, cada uno de los cuales añade funcionalidad al nivel que está abajo. Ya vimos el nivel de lógica digital, el nivel de microarquitectura y el nivel de conjunto de instrucciones. Ha llegado el momento de ascender al siguiente nivel, el ámbito del sistema operativo.

Un **sistema operativo** es un programa que, desde el punto de vista del programador, añade varias instrucciones y funciones nuevas, más allá de lo que el nivel ISA proporciona. Normalmente, el sistema operativo se implementa casi totalmente en software, pero no existe una razón teórica para no colocarlo en hardware, como se hace normalmente con los microprogramas (cuando están presentes). Usaremos el acrónimo **OSM** (*Operating System Machine*) para referirnos al nivel que el sistema operativo implementa, el nivel de **máquina de sistema operativo**, que se muestra en la figura 6-1.

Aunque tanto el nivel OSM como el nivel ISA son abstractos (en el sentido de que no son verdadero nivel de hardware), existe una diferencia importante entre ellos. El conjunto de instrucciones del nivel OSM es el conjunto completo de instrucciones que pueden usar los programadores de aplicaciones; contiene casi todas las instrucciones del nivel ISA, así como el conjunto de instrucciones nuevas que el sistema operativo añade. Estas nuevas instrucciones se denominan **llamadas al sistema**. Una llamada al sistema invoca un servicio predefinido del sistema operativo, o sea, una de sus instrucciones. Una llamada al sistema típica solicita leer datos de un archivo. Usaremos el tipo de letra helvética en minúsculas para distinguir las llamadas al sistema.

El nivel OSM siempre se interpreta. Cuando un programa de usuario ejecuta una instrucción OSM, como leer datos de un archivo, el sistema operativo ejecuta esta instrucción paso

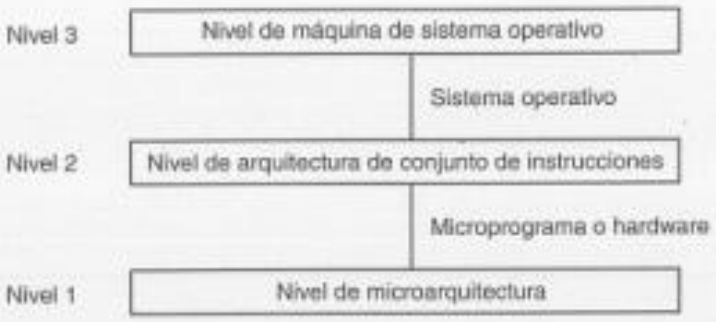

Figura 6-1. Ubicación del nivel de máquina de sistema operativo.

por paso, igual que un microprograma ejecutaría una instrucción ADD paso por paso. Sin embargo, cuando un programa ejecuta una instrucción del nivel ISA, el nivel de microarquitectura subyacente se encarga de llevarla a cabo directamente, sin ayuda del sistema operativo.

En este libro sólo podemos proporcionar una muy breve introducción al tema de los sistemas operativos. Nos concentraremos en tres importantes temas. El primero es la memoria virtual, una técnica que muchos sistemas operativos usan para aparentar que la máquina tiene más memoria de la que en realidad tiene. El segundo es la E/S de archivos, un concepto de nivel más alto que las instrucciones de E/S que estudiamos en el capítulo anterior. El tercer y último tema es el de procesamiento en paralelo: cómo varios procesos pueden ejecutarse, comunicarse y sincronizarse. El concepto de proceso es muy importante, y lo describiremos con detalle más adelante. Por ahora, podemos pensar en un proceso como un programa en ejecución y toda su información de estado (memoria, registros, contador de programa, situación de E/S, etc.). Después de explicar estos principios en general, veremos cómo se aplican a los sistemas operativos de dos de nuestras máquinas de ejemplo, el Pentium II (Windows NT) y el UltraSPARC II (UNIX). Puesto que el picoJava II normalmente se usa para sistemas incorporados, no tiene un sistema operativo con todas las de la ley.

6.1 MEMORIA VIRTUAL

En los albores de la computación, las memorias eran pequeñas y costosas. El IBM 650, la principal computadora científica de su época (fines de la década de los cincuenta), sólo tenía 2000 palabras de memoria. Uno de los primeros compiladores de ALGOL 60 se escribió para una máquina que sólo tenía 1024 palabras de memoria. Uno de los primeros sistemas de tiempo compartido trabajaba de forma muy satisfactoria en una PDP-1 con un tamaño total de memoria de 4096 palabras de 18 bits para el sistema operativo y los programas de usuario combinados. En esos días el programador dedicaba gran parte de su tiempo a lograr que sus programas cupieran en la diminuta memoria. En muchos casos era necesario usar un algoritmo que trabajaba mucho más despacio que otro algoritmo mejor, simplemente porque el algoritmo mejor era demasiado grande; es decir, un programa que usaba el algoritmo mejor no cabía en la memoria de la computadora.

La solución tradicional a este problema era usar memoria secundaria, como un disco. El programador dividía el programa en varios fragmentos, llamados **superposiciones**, cada uno de los cuales cabía en la memoria. Para ejecutar el programa, se leía la primera superposición y se ejecutaba durante un rato. Cuando terminaba, leía la siguiente superposición y la invocaba, y así. El programador tenía la responsabilidad de dividir el programa en superposiciones, decidir en qué lugar de la memoria secundaria se debía guardar cada superposición, encargarse del traslado de superposiciones entre la memoria principal y la memoria secundaria, y en general administrar el proceso de superponer sin ayuda de la computadora.

Aunque se usó ampliamente durante muchos años, esta técnica implicaba mucho trabajo invertido en la gestión de superposiciones. En 1961 un grupo de investigadores de Manchester, Inglaterra, propuso un método para realizar automáticamente el proceso de superponer, sin que el programador siquiera supiera qué estaba ocurriendo (Fotheringham, 1961). Este método, ahora conocido como **memoria virtual**, tenía la ventaja obvia de liberar al programador de una gran cantidad de molestas tareas de contabilidad, y se usó por primera vez en los años sesenta en varias computadoras, casi todas asociadas a proyectos de investigación sobre diseño de sistemas de cómputo. Para cuando comenzó la década de los setenta casi todas las computadoras contaban con memoria virtual. Ahora hasta las computadoras de un solo chip, incluidos el Pentium II y el UltraSPARC II, tienen sistemas de memoria virtual muy sofisticados, que examinaremos en una sección posterior del capítulo.

6.1.1 Paginación

La idea sugerida por el grupo de Manchester fue la de separar los conceptos de espacio de direcciones y posiciones de memoria. Consideremos, por ejemplo, una computadora representativa de esa época, con un campo de dirección de 16 bits en sus instrucciones y 4096 palabras de memoria. Un programa para esta computadora podía direccionar 65536 palabras de memoria. La razón es que existen 65536 (2^{16}) direcciones de 16 bits, cada una de las cuales corresponde a una palabra de memoria distinta. Cabe señalar que el número de palabras direccionables depende únicamente del número de bits que tiene una dirección y nada tiene que ver con el número de palabras de memoria con que se cuenta realmente. El **espacio de direcciones** de esta computadora consiste en los números 0, 1, 2, . . . , 65535, porque ése es el conjunto de posibles direcciones. Sin embargo, la computadora bien podría tener menos de 65535 palabras de memoria.

Antes de que se inventara la memoria virtual, la gente habría distinguido entre las direcciones por debajo de 4096 y aquellas iguales o mayores que 4096. Aunque pocas veces se decía explícitamente, estas dos partes se consideraban como el espacio de direcciones útiles y el espacio de direcciones inútiles, respectivamente (las direcciones mayores que 4095 eran inútiles porque no correspondían a posiciones de memoria reales). La gente no distinguía entre espacio de direcciones y direcciones de memoria, porque el hardware exigía una correspondencia uno a uno entre ellos.

La idea de separar el espacio de direcciones y las direcciones de memoria es la siguiente. En cualquier momento dado, es posible acceder directamente a 4096 palabras de memoria,

pero no es necesario que correspondan a las direcciones de memoria 0 a 4095. Podríamos, por ejemplo, "decirle" a la computadora que, en adelante, cada vez que se haga referencia a la dirección 4096, se usará la palabra de memoria que está en la dirección 0. Cada vez que se haga referencia a la dirección 4097, se usará la palabra de memoria que está en la dirección 1; cada vez que se haga referencia a la dirección 8191, se usará la palabra de memoria que está en la dirección 4095, y así. En otras palabras, habremos definido una correspondencia o "mapeo" del espacio de direcciones a las direcciones de memoria reales, como se muestra en la figura 6-2.

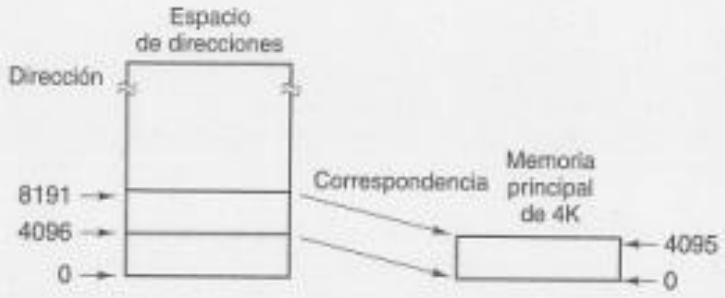

Figura 6-2. Mapeo en el que las direcciones virtuales 4096 a 8191 se hacen corresponder con las direcciones 0 a 4095 de la memoria principal.

En términos de hacer corresponder las direcciones del espacio de direcciones con las posiciones de memoria reales, una máquina de 4K sin memoria virtual simplemente tiene una correspondencia fija entre las direcciones 0 a 4095 y las 4096 palabras de memoria. Una pregunta interesante es: "¿Qué sucede si un programa ramifica a una dirección entre 8192 y 12287?" En una máquina sin memoria virtual, el programa causaría una trampa de error que imprimiría un mensaje debidamente grosero como "Referencia a memoria inexistente" y terminaría el programa. En una máquina con memoria virtual, ocurriría la siguiente sucesión de pasos:

1. El contenido de la memoria principal se guardaría en el disco.
2. Se localizarían en el disco las palabras 8192 a 12287.
3. Se cargarían en la memoria principal las palabras 8192 a 12287.
4. El mapa de direcciones se modificaría para hacer corresponder las direcciones 8192 a 12287 con las posiciones de memoria 0 a 4095.
5. La ejecución continuaría como si nada fuera de lo normal hubiera ocurrido.

Esta técnica de superposición automática se denomina **paginación**, y los fragmentos de programa que se leen del disco se llaman **páginas**.

Puede usarse una forma más sofisticada de hacer corresponder las direcciones del espacio de direcciones con las direcciones de memoria real. Para destacar la diferencia, llamaremos a las direcciones a las que el programa puede hacer referencia **espacio de direcciones virtual**, y a las direcciones de memoria reales, en hardware, **espacio de direcciones físico**. Un **mapa de memoria** o **tabla de páginas** relaciona las direcciones virtuales con las direc-

ciones físicas. Suponemos que hay suficiente espacio en el disco para guardar todo el espacio de direcciones virtual (o al menos la porción de ese espacio que se está usando).

Los programas se escriben como si hubiera suficiente memoria principal para todo el espacio de direcciones virtual, aunque no sea así. Los programas pueden cargar valores de, o guardarlos en, cualquier palabra del espacio de direcciones virtual, o saltar a cualquier instrucción situada en cualquier punto del espacio de direcciones virtual, sin tener en cuenta el hecho de que en realidad no hay suficiente memoria física. De hecho, el programador puede escribir programas sin siquiera saber si existe o no memoria virtual. La computadora simplemente parece tener una memoria grande.

Este punto es crucial y se contrastará posteriormente con la segmentación, en la que el programador debe tener conocimiento de la existencia de segmentos. Hacemos hincapié una vez más en que la paginación ofrece al programador la ilusión de una memoria principal grande, continua y lineal, del mismo tamaño que el espacio de direcciones virtual. En realidad, la memoria principal disponible podría ser menor (o mayor) que el espacio de direcciones virtual. La simulación de esta memoria grande por paginación no puede ser detectada por el programa (a menos que se efectúen pruebas de cronometría). Cada vez que se hace referencia a una dirección, parece estar presente la palabra de instrucción o datos correcta. Puesto que el programador puede programar como si no existiera la paginación, se dice que el mecanismo de paginación es **transparente**.

No es la primera vez que nos topamos con la idea de que un programador pueda usar alguna característica inexistente sin preocuparse por su funcionamiento. El conjunto de instrucciones del nivel ISA a menudo incluye una instrucción MUL, a pesar de que la microarquitectura subyacente no cuenta con un dispositivo de multiplicación en el hardware. La ilusión de que la máquina puede multiplicar casi siempre se mantiene con microcódigo. Así mismo, la máquina virtual que el sistema operativo proporciona puede dar la ilusión de que todas las direcciones virtuales están respaldadas por memoria real, aunque no sea así. Sólo los escritores de sistemas operativos (y estudiantes de tales sistemas) tienen que saber cómo se mantiene la ilusión.

6.1.2 Implementación de la paginación

Un requisito indispensable para una memoria virtual es un disco en el cual pueda guardarse todo el programa y todos los datos. En lo conceptual, es más sencillo pensar que la copia del programa que está en el disco es el original y que los fragmentos que se traen a la memoria principal de vez en cuando son copias, y no lo contrario. Desde luego, es importante mantener actualizado el original. Cuando se modifica la copia que está en la memoria principal, los cambios deben reflejarse en el original (tarde o temprano).

El espacio de direcciones se divide en varias páginas de tamaño uniforme. Hoy día son comunes tamaños de página entre 512 y 64K bytes por página, aunque de vez en cuando se usan tamaños de hasta 4 MB. El tamaño de página siempre es una potencia de 2. El espacio de direcciones físico se divide en fragmentos de la misma manera, y cada uno es del mismo tamaño de una página, de modo que cada fragmento de la memoria principal pueda contener exactamente una página. Estos fragmentos de la memoria principal en los que entran las

páginas se llaman **marcos de página**. En la figura 6-2 la memoria principal contiene sólo un marco de página. En los diseños prácticos por lo regular contiene miles de marcos.

La figura 6-3(a) ilustra una posible forma de dividir los primeros 64K de un espacio de direcciones virtual: en páginas de 4K. (Tome nota de que estamos hablando de 64K y 4K de direcciones aquí. Una dirección podría ser un byte pero también podría ser una palabra en una computadora en la que palabras consecutivas tienen direcciones consecutivas.) La memoria virtual de la figura 6-3 se implementaría con una tabla de páginas que tiene tantas entradas como páginas caben en el espacio de direcciones virtual. Para simplificar, sólo hemos mostrado aquí las primeras 16 entradas. Cuando el programa trata de hacer referencia a una palabra que está en los primeros 64K de su espacio de direcciones virtual, sea para traer instrucciones, traer datos o guardar datos, primero genera una dirección virtual entre 0 y 65532 (suponiendo que las direcciones de palabra deben ser divisibles entre 4). Se puede usar indexación, direccionamiento indirecto o cualquier otra de las técnicas acostumbradas para generar la dirección.

Página	Direcciones virtuales
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

Marco de página	32K bajos de la memoria principal Direcciones físicas
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

Figura 6-3. (a) Los primeros 64K del espacio de direcciones virtual divididos en 16 páginas, cada una de las cuales tiene 4K. (b) Memoria principal de 32K dividida en ocho marcos de página de 4K cada uno.

La figura 6-3(b) muestra una memoria física que consiste en ocho marcos de página de 4K. Esta memoria podría estar limitada a 32K porque (1) eso es todo lo que la máquina tiene (un procesador incorporado en una lavadora de ropa o un horno de microondas tal vez no necesite más), o (2) el resto de la memoria se asignó a otros programas.

Considere ahora cómo puede hacerse corresponder una dirección virtual de 32 bits con una dirección física en la memoria principal. Después de todo, lo único que la memoria entiende son direcciones de la memoria principal, no direcciones virtuales, así que eso es lo que debemos darle. Toda computadora con memoria virtual tiene un dispositivo para efectuar el mapeo de virtual a físico, llamado **unidad de gestión de memoria** (MMU, *Memory Management Unit*). La MMU podría estar en el chip de la CPU o en un chip aparte que funcione en estrecha colaboración con la CPU. Puesto que nuestra MMU de ejemplo establece una correspondencia entre una dirección virtual de 32 bits y una dirección física de 15 bits, necesita un registro de entrada de 32 bits y un registro de salida de 15 bits.

Para ver cómo funciona la MMU, considere el ejemplo de la figura 6-4. Cuando la MMU recibe una dirección virtual de 32 bits, la divide en un número de página virtual, de 20 bits, y una distancia dentro de la página, de 12 bits (porque en nuestro ejemplo las páginas son de 4K). El número de página virtual se usa como índice de la tabla de páginas para encontrar la entrada que corresponde a la página a la que se hizo referencia. En la figura 6-4 el número de página virtual es 3, por lo que se selecciona la entrada 3 de la tabla de páginas, como se muestra.

Lo primero que la MMU hace con la entrada de la tabla de páginas es verificar si la página a la que se hizo referencia está actualmente en la memoria principal. Después de todo, con 2^{20} páginas virtuales y sólo 8 marcos de página, no todas las páginas virtuales pueden estar en la memoria a la vez. La MMU efectúa esta verificación examinando el **bit presente/ausente** de la entrada de la tabla de páginas. En nuestro ejemplo el bit es 1, lo que implica que la página actualmente está en la memoria.

El siguiente paso es tomar el valor de marco de página de la entrada seleccionada (6 en este caso) y copiarlo en los tres bits superiores del registro de salida de 15 bits. Se requieren tres bits porque hay ocho marcos de página en la memoria física. Al mismo tiempo que se realiza esta operación, los 12 bits de orden bajo de la dirección virtual (el campo de distancia dentro de la página) se copian en los 12 bits de orden bajo del registro de salida, como se muestra. Esta dirección de 15 bits se envía entonces a caché o a la memoria para buscarla.

La figura 6-5 muestra una posible correspondencia entre páginas virtuales y marcos de página físicos. La página virtual 0 está en el marco de página 1. La página virtual 1 está en el marco de página 0. La página virtual 2 no está en la memoria principal. La página virtual 3 está en el marco de página 2. La página virtual 4 no está en la memoria principal. La página virtual 5 está en el marco de página 6, etcétera.

6.1.3 Paginación por demanda y modelo de conjunto de trabajo

En la explicación anterior se supuso que la página virtual a la que se hizo referencia estaba en la memoria principal. Sin embargo, tal supuesto no siempre se cumplirá porque no hay suficiente espacio en la memoria principal para todas las páginas virtuales. Cuando se hace una referencia a una dirección que está en una página que no está presente en la memoria principal, ocurre un **fallo de página**. En tal caso, el sistema operativo necesita leer la página requerida del disco, introducir en la tabla de páginas su nueva ubicación en la memoria física, y luego repetir la instrucción que causó el fallo.

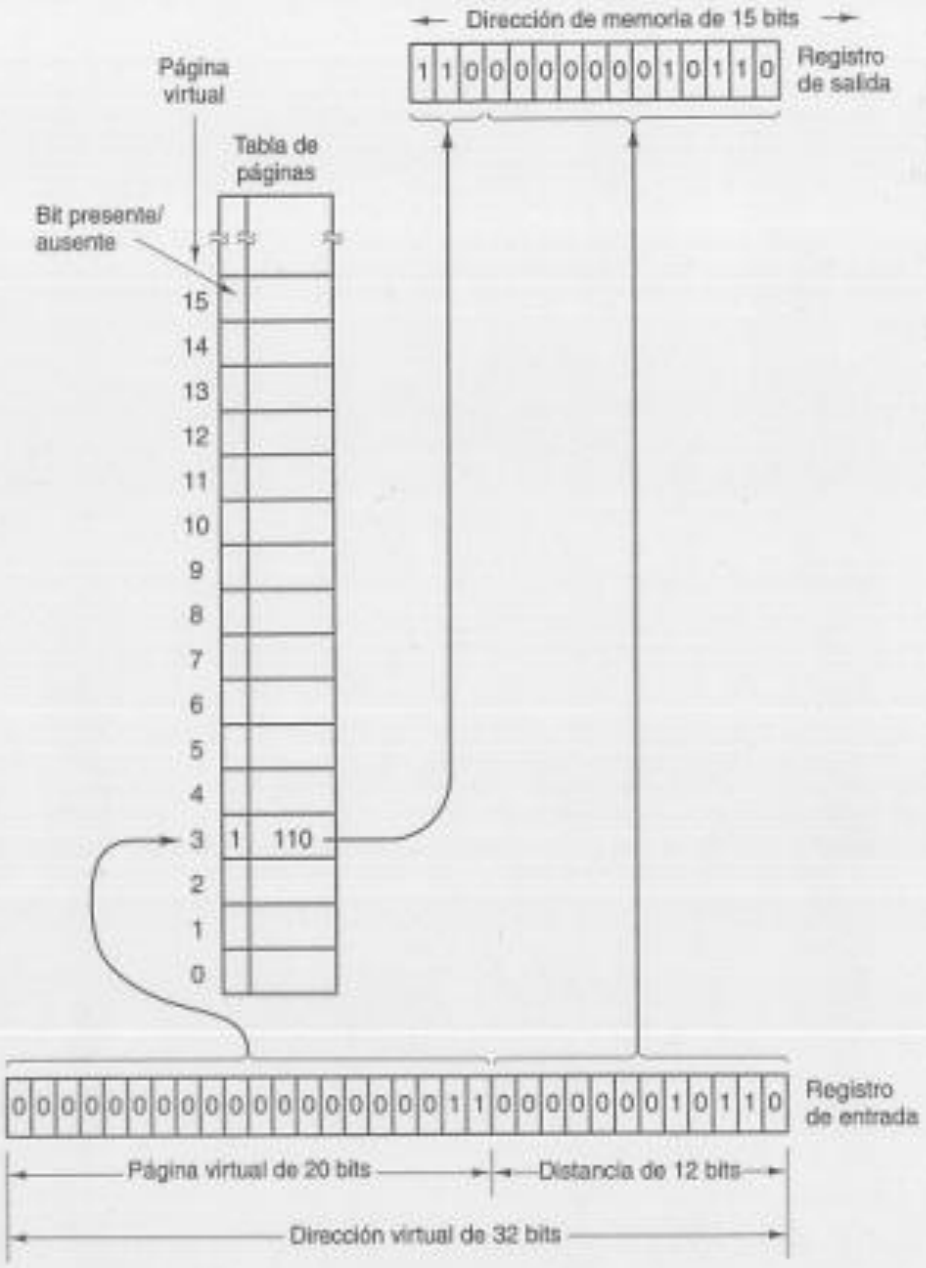

Figura 6-4. Formación de una dirección de memoria principal a partir de una dirección virtual.

Es posible iniciar la ejecución de un programa en una máquina con memoria virtual aunque ninguna parte del programa esté en la memoria principal. Simplemente hay que ajustar la tabla de páginas de modo que indique que todas y cada una de las páginas virtuales está en la memoria secundaria y no en la memoria principal. Cuando la CPU trate de traer la primera instrucción, de inmediato causará un fallo de página, lo que hará que la página que contiene la primera instrucción se cargue en la memoria y se introduzca en la tabla de páginas. Luego podrá iniciarse la primera instrucción. Si dicha instrucción tiene dos direcciones,

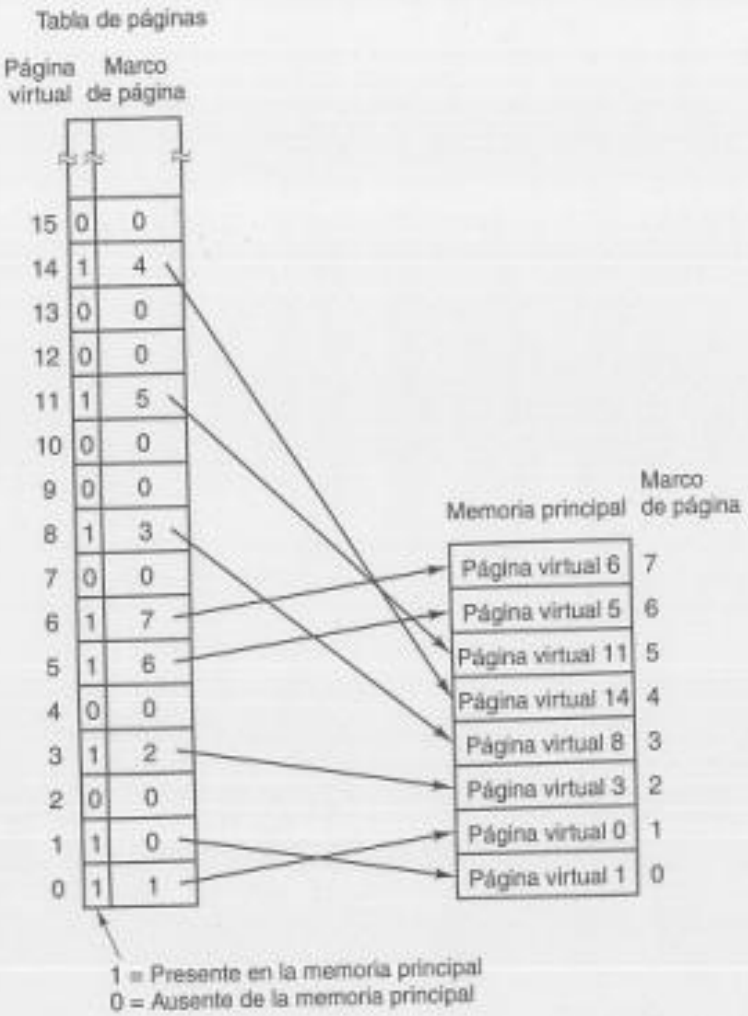

Figura 6-5. Una posible correspondencia entre las primeras 16 páginas virtuales y una memoria principal con ocho marcos de página.

las dos están en diferentes páginas y ambas páginas son diferentes de la página en la que estaba la instrucción, ocurrirán otros dos fallos de página, y se traerán dos páginas más antes de que la instrucción por fin pueda ejecutarse. La siguiente instrucción podría causar más fallos de página, y así sucesivamente.

Este método de operar una memoria virtual se llama **paginación por demanda**, por analogía con el conocido algoritmo de alimentación por demanda para los bebés: si el bebé llora, hay que alimentarlo (en lugar de alimentarlo según un programa estricto). En la paginación por demanda, sólo se traen páginas cuando se solicita realmente una página, no por adelantado.

La pregunta de si debe usarse o no paginación por demanda sólo es pertinente cuando se inicia un programa. Una vez que el programa ha estado trabajando durante cierto tiempo, las páginas requeridas ya se habrán reunido en la memoria principal. Si la computadora es de

tiempo compartido y los procesos se intercambian a disco después de ejecutarse durante 100 ms (digamos), cada programa se reiniciará muchas veces durante su ejecución. Puesto que el mapa de memoria es único para cada programa, y cambia cuando se conmutan los programas, como en un sistema de tiempo compartido, la pregunta se vuelve una cuestión crítica.

La estrategia alternativa se basa en la observación de que la mayor parte de los programas no hace referencia a su espacio de direcciones de manera uniforme, sino que las referencias tienden a concentrarse en un número reducido de páginas. Una referencia a la memoria podría traer una instrucción, podría traer datos o podría guardar datos. En cualquier instante *t* existe un conjunto formado por todas las páginas utilizadas por las *k* referencias a memoria más recientes. Denning (1968) lo llamó **conjunto de trabajo**.

Dado que el conjunto de trabajo suele variar lentamente con el tiempo, es posible hacer una conjetura razonable respecto a cuáles páginas se necesitarán cuando se reinicie el programa, con base en el conjunto de trabajo que tenía cuando se suspendió la última vez. Estas páginas podrían cargarse por adelantado antes de iniciar el programa (suponiendo que caben).

6.1.4 Política de reemplazo de páginas

Idealmente, el conjunto de páginas que un programa está usando activa e intensivamente, llamado **conjunto de trabajo**, se puede mantener en la memoria a fin de reducir los fallos de página. Sin embargo, los programadores casi nunca saben cuáles páginas están en el conjunto de trabajo, por lo que el sistema operativo debe descubrir dicho conjunto dinámicamente. Cuando un programa hace referencia a una página que no está en la memoria principal, la página requerida debe traerse del disco. Sin embargo, para que quepa generalmente es necesario enviar alguna otra página de vuelta al disco. Se necesita un algoritmo que decida cuál página debe quitarse.

Probablemente no es una buena idea escoger al azar la página que se quitará. Si la página que contiene la instrucción que causó el fallo es la que se escoge, ocurrirá otro fallo de página tan pronto como se intente traer la siguiente instrucción. Casi todos los sistemas operativos tratan de predecir cuál de las páginas de la memoria es la menos útil en el sentido de que su ausencia tendría el efecto adverso menos sensible sobre el programa en ejecución. Una forma de hacerlo es predecir cuándo ocurrirá la siguiente referencia a cada página y quitar la página cuya siguiente referencia predicha es más lejana en el futuro. En otras palabras, en lugar de expulsar una página que se necesitará en poco tiempo, se trata de seleccionar una que no se necesitará durante un buen tiempo.

Un algoritmo muy utilizado expulsa la página que se usó menos recientemente porque la probabilidad *a priori* de que no esté en el conjunto de trabajo vigente es alta. Éste es el algoritmo **LRU (menos recientemente utilizada**, *Least Recently Used*). Aunque este algoritmo normalmente funciona bien, existen situaciones patológicas, como la que se describirá a continuación, en las que falla lamentablemente.

Imagine un programa que ejecuta un ciclo grande que se extiende sobre nueve páginas virtuales en una máquina que tiene espacio sólo para ocho páginas en la memoria física. Una vez que el programa llega a la página 7, el estado de la memoria principal es el que se muestra

en la figura 6-6(a). En algún momento se intenta traer una instrucción de la página virtual 8, lo que causa un fallo de página. Se debe tomar una decisión acerca de cuál página se expulsará. El algoritmo LRU escoge la página virtual 0 porque es la que se usó menos recientemente. Se quita la página virtual 0 y se trae la página virtual 8 para sustituirla, y la situación es la que se muestra en la figura 6-6(b).

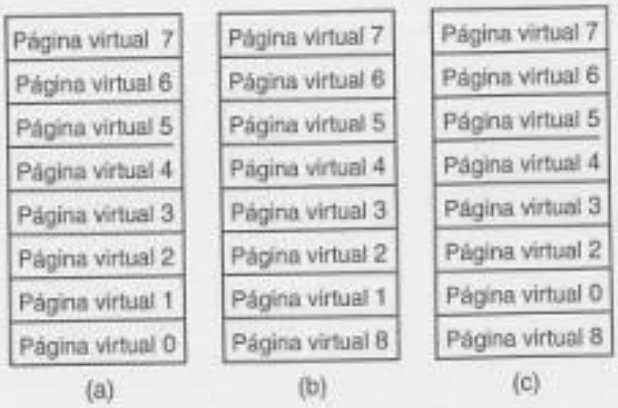

Figura 6-6. Fracaso del algoritmo LRU.

Después de ejecutar las instrucciones de la página virtual 8, el programa salta al principio del ciclo, a la página virtual 0. Este paso causa otro fallo de página. La página virtual 0, que acaba de expulsarse, se tiene que volver a traer. El algoritmo LRU escoge la página 1 para expulsión, lo cual lleva a la situación de la figura 6-6(c). El programa continuará en la página 0 durante un rato, y luego tratará de traer una instrucción de la página virtual 1, causando un fallo de página. Hay que volver a traer la página 1 y para ello se expulsará la página 2.

Ya deberá ser obvio que el algoritmo LRU está tomando la peor decisión en cada ocasión (otros algoritmos también fallan en condiciones similares). Sin embargo, si la memoria principal disponible es mayor que el tamaño del conjunto de trabajo, el algoritmo LRU sí tiende a minimizar el número de fallos de página.

Otro algoritmo de reemplazo de páginas es el de **primero en entrar, primero en salir** (**FIFO**, *First-In First-Out*). FIFO expulsa la página que se cargó menos recientemente, sin importar cuándo fue la última vez que se hizo referencia a ella. Cada marco de página tiene un contador. Inicialmente, todos los contadores contienen 0. Después de manejarse cada fallo de página, se incrementa en 1 el contador de cada página que actualmente está en la memoria, y el contador de la página que se acaba de traer se pone en 0. Cuando se hace necesario escoger una página para expulsarla, se escoge aquella cuyo contador es el más alto. Esto indica que la página correspondiente ha sido testigo del mayor número de fallos de página, y por tanto que se cargó antes que cualquiera de las otras páginas que están en la memoria y tiene (con suerte) la mayor probabilidad *a priori* de no necesitarse más.

Si el conjunto de trabajo es mayor que el número de marcos de página disponibles, ningún algoritmo que no sea un oráculo dará buenos resultados, y los fallos de página serán

frecuentes. Un programa que genera fallos de página de forma continua y frecuente está **hiperpaginando**. Sobra decir que la hiperpaginación es una condición indeseable en cualquier sistema. Si un programa usa una gran cantidad de espacio virtual de direcciones pero tiene un conjunto de trabajo pequeño que cambia lentamente y cabe en la memoria principal disponible, no causará problemas. Esta observación se cumple aunque, a lo largo de su existencia, el programa use cientos de veces más palabras de memoria virtual que las palabras de memoria principal que tiene la máquina.

Si una página que está a punto de ser expulsada no se ha modificado desde que se leyó (lo cual es muy probable si la página contiene programa, no datos), no es necesario volver a escribirla en el disco, pues ya existe ahí una copia exacta. Si la página se modificó después de leerse, la copia del disco ya no es exacta y habrá que reescribir la página.

Si hay una forma de saber si una página no ha cambiado desde que se leyó (página limpia) o si se escribió después en ella (página sucia), podrá evitarse la reescritura de páginas limpias y se ahorrará mucho tiempo. Muchas computadoras tienen un bit por página, en la MMU, que se pone en 0 cuando la página se carga, y que el microprograma o el hardware pone en 1 cada vez que se escribe en ella (es decir, se ensucia). Si el sistema operativo examina este bit, sabrá si la página está limpia o sucia, y por ende si es necesario reescribirla o no.

6.1.5 Tamaño de página y fragmentación

Si por casualidad el programa y los datos del usuario llenan exactamente un número entero de páginas, no se desperdiciará espacio cuando estén en la memoria. Pero si no sucede así, quedará cierto espacio sin utilizar en la última página. Por ejemplo, si el programa y los datos necesitan 26,000 bytes en una máquina que tiene 4096 bytes por página, las primeras seis páginas estarán llenas, para un total de $6 \times 4096 = 24{,}576$ bytes, y la última página contendrá $26{,}000 - 24{,}576 = 1424$ bytes. Puesto que en cada página caben 4096 bytes, se desperdiciarán 2672 bytes. Cada vez que la séptima página esté presente en la memoria, esos bytes ocuparán espacio en la memoria principal pero no servirán de nada. El problema de estos bytes desperdiciados se denomina **fragmentación interna** (porque el espacio desperdiciado está adentro de una página).

Si el tamaño de página es de n bytes, la cantidad media de espacio desperdiciado por fragmentación interna en la última página de un programa será de $n/2$ bytes, situación que sugiere usar páginas pequeñas para minimizar el desperdicio. Por otra parte, un tamaño de página pequeño implica muchas páginas, además de una tabla de páginas grande. Si la tabla de páginas se mantiene en hardware, y es grande, se necesitarán más registros para guardarla, y el costo de la computadora aumentará. Además, se requerirá más tiempo para cargar y guardar estos registros cada vez que se inicie o detenga un programa.

Además, las páginas pequeñas utilizan de forma ineficiente el ancho de banda del disco. Si suponemos que hay que esperar 10 ms para poder iniciar una transferencia (retraso por búsqueda + latencia rotacional), las transferencias grandes serán más eficientes que las pequeñas. Con una tasa de transferencia de 10 MB/s, transferir 8 KB tarda sólo 0.7 ms más que transferir 1 KB (10.1 ms *versus* 10.8 ms).

Sin embargo, las páginas pequeñas tienen la ventaja adicional de que si el conjunto de trabajo consiste en un gran número de regiones pequeñas discretas en el espacio de direcciones virtual podría haber menos hiperpaginación con un tamaño de página pequeño que con uno grande. Por ejemplo, considere una matriz de 10,000 × 10,000, *A* que se almacena con *A*[1, 1], *A*[2, 1], *A*[3, 1], etc., en palabras de 8 bytes consecutivas. Este almacenamiento ordenado por columna implica que los elementos de la fila 1, *A*[1, 1], *A*[1, 2], *A*[1, 3], etc., comenzarán en posiciones separadas 80,000 bytes una de la siguiente. Un programa que realice cálculos extensos con todos los elementos de esta fila usaría 10,000 regiones, cada una separada de la siguiente por 79,992 bytes. Si el tamaño de página fuera de 8 KB, se requeriría una memoria total de 80 MB para contener todas las páginas en uso.

Por otra parte, si el tamaño de página es de 1 KB, se requerirían sólo 10 MB de RAM para contener todas las páginas. Si la memoria disponible fuera de 32 MB, el programa hiperpaginaría si las páginas fueran de 8 KB, pero no si fueran de 1 KB.

6.1.6 Segmentación

La memoria virtual de la que hemos hablado aquí es unidimensional porque las direcciones virtuales van de la 0 a alguna dirección máxima, una tras otra. Para muchos problemas, tener dos o más espacios de direcciones distintos podría ser mucho mejor que tener sólo uno. Por ejemplo, un compilador podría tener muchas tablas que se van creando a medida que avanza la compilación y que incluyen:

1. La tabla de símbolos, que contiene los nombres y atributos de las variables.
2. El texto fuente que se está guardando para el listado impreso.
3. Una tabla que contiene todas las constantes enteras y de punto flotante empleadas.
4. El árbol de análisis sintáctico del programa.
5. La pila empleada para llamadas a procedimientos dentro del compilador.

Las primeras cuatro tablas crecen continuamente a medida que la compilación avanza. La última crece y se encoge de forma impredecible durante la compilación. En una memoria unidimensional, habría que asignar a estas tablas trozos contiguos del espacio de direcciones virtual, como en la figura 6-7.

Considere lo que sucede si un programa tiene un número excepcionalmente grande de variables. El trozo de espacio de direcciones asignado a la tabla de símbolos podría llenarse, aunque haya mucho espacio libre en las otras tablas. Desde luego, el compilador podría limitarse a emitir un mensaje diciendo que la compilación no puede continuar porque hay demasiadas variables, pero hacerlo no sería justo si queda espacio sin ocupar en las demás tablas.

Otra posibilidad es que el compilador actúe como Robin Hood y tome espacio libre de las tablas que tienen mucho para dárselo a las tablas que tienen poco. Esto es factible, pero sería análogo a gestionar sus propias superposiciones: una lata en el mejor de los casos y un exceso de trabajo tedioso y fútil en el peor de los casos.

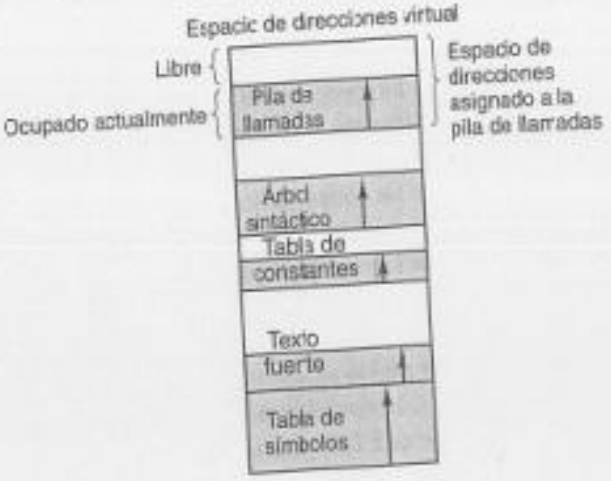

Figura 6-7. En un espacio de direcciones unidimensional con tablas que crecen, una tabla podría chocar con otra.

Lo que realmente se necesita es una forma de evitar al programador el trabajo de administrar las tablas en expansión y contracción, del mismo modo que la memoria virtual elimina la preocupación por tener que organizar el programa en superposiciones.

Una solución sencilla es proporcionar muchos espacios de direcciones totalmente independientes, llamados **segmentos**. Cada segmento consiste en una sucesión lineal de direcciones, de la 0 a alguna dirección máxima. La longitud de cada segmento puede ser cualquiera desde 0 hasta el máximo permitido. Diferentes segmentos pueden tener diferentes longitudes, y generalmente así es. Además, la longitud de los segmentos podría cambiar durante la ejecución. La longitud de un segmento de pila podría incrementarse cada vez que algo se mete en la pila y reducirse cada vez que algo se desempila.

Puesto que cada segmento constituye un espacio de direcciones individual, los diferentes segmentos pueden crecer o encogerse de manera independiente, sin afectar a los otros. Si una pila en cierto segmento necesita más espacio de direcciones para crecer, puede tenerlo, porque no hay nada más en su espacio de direcciones con lo que pueda chocar. Desde luego, un segmento podría llenarse, pero los segmentos suelen ser muy grandes, y esto casi nunca sucede. Para especificar una dirección en este memoria bidimensional segmentada, el programa debe proporcionar una dirección con dos partes: un número de segmento y una dirección dentro de ese segmento. La figura 6-8 ilustra una memoria segmentada que se usa para las tablas de compilador antes mencionadas.

Hacemos hincapié en que un segmento es una entidad lógica, de cuya existencia el programador es consciente, y que usa como una sola entidad lógica. Un segmento podría contener un procedimiento, un arreglo, una pila o una colección de variables escalares, pero por lo regular no contiene una mezcla de diferentes tipos.

Las memorias segmentadas tienen otras ventajas además de simplificar el manejo de estructuras de datos que crecen o se encogen. Si cada procedimiento ocupa un segmento distinto, y la dirección 0 es su dirección de inicio, se simplifica considerablemente la vincu-

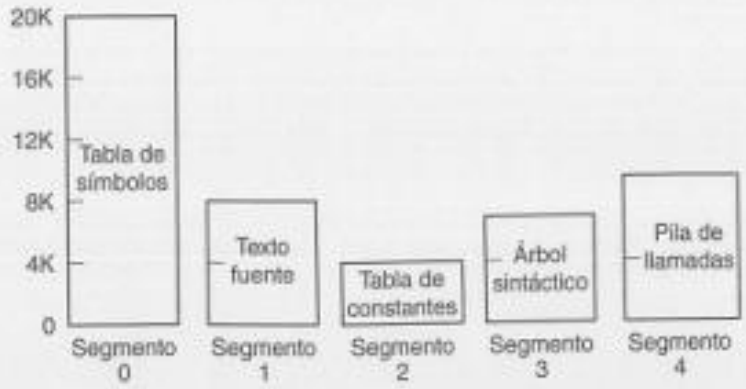

Figura 6-8. Una memoria segmentada permite a cada tabla crecer o encogerse con independencia de las otras tablas.

lación o ligado (*linking*) de procedimientos que se compilan por separado. Una vez que se han compilado y vinculado todos los procedimientos que constituyen un programa, una llamada al procedimiento en el segmento n usará la dirección de dos partes $(n, 0)$ para direccionar la palabra 0 (el punto de ingreso).

Si el procedimiento que está en el segmento n se modifica y recompila posteriormente, no será necesario modificar ningún otro procedimiento (porque no se ha cambiado ninguna dirección de inicio), aunque la nueva versión sea más grande que la antigua. Con una memoria unidimensional, los procedimientos normalmente se empaquetan muy juntos, sin espacio de direcciones entre ellos. Por tanto, la modificación del tamaño de un procedimiento puede afectar la dirección de inicio de otros procedimientos no relacionados. Esto, a su vez, requiere modificar todos los procedimientos que invocan cualquiera de los procedimientos desplazados, a fin de incorporar sus nuevas direcciones de inicio. Si un programa contiene cientos de procedimientos, este proceso puede ser costoso.

La segmentación también facilita compartir procedimientos o datos entre varios programas. Si una computadora tiene varios programas ejecutándose en paralelo (sea procesamiento en paralelo verdadero o simulado), y todos usan ciertos procedimientos de biblioteca, sería un desperdicio de memoria principal proporcionar a cada uno su copia privada. Si se hace que cada procedimiento esté en un segmento distinto, es fácil compartirlos y se elimina la necesidad de tener más de una copia física de cualquier procedimiento compartido en la memoria principal. El resultado es un ahorro de memoria.

Puesto que cada segmento forma una entidad lógica de la que el programador tiene conocimiento, como un procedimiento, un arreglo o una pila, los diferentes segmentos pueden tener diferentes tipos de protección. Un segmento de procedimiento podría especificarse como "sólo para ejecución", lo que impediría que alguien lo lea o escriba en él. Un arreglo de punto flotante podría especificarse como de lectura/escritura pero no para ejecución; con ello se detectaría cualquier intento por saltar a él. Este tipo de protección suele ser útil para detectar errores de programación.

Es importante entender por qué la protección tiene sentido en una memoria segmentada pero no en una memoria paginada unidimensional (es decir, lineal). En una memoria

segmentada el usuario sabe qué hay en cada segmento. Normalmente, un segmento no contiene un procedimiento y una pila, por ejemplo, sino sólo uno de los dos. Puesto que cada segmento contiene un solo tipo de objeto, el segmento puede tener la protección apropiada para ese tipo específico. En la figura 6-9 se comparan la paginación y la segmentación.

Consideración	Paginación	Segmentación
¿El programador necesita saber que existe?	No	Sí
¿Cuántos espacios de direcciones lineales hay?	1	Muchos
¿El espacio de direcciones virtual puede exceder el tamaño de la memoria?	Sí	Sí
¿Es fácil manejar tablas con tamaño variable?	No	Sí
¿Por qué se inventó esta técnica?	Para simular memorias grandes	Para tener muchos espacios de direcciones

Figura 6-9. Comparación de paginación y segmentación.

El contenido de una página es, en cierto sentido, accidental. El programador ni siquiera se da cuenta de que está ocurriendo paginación. Aunque sería posible incluir unos cuantos bits en cada entrada de la tabla de páginas para especificar el acceso permitido, si el programador quiere aprovechar esta capacidad tendría que saber en qué parte del espacio de direcciones están las fronteras de las páginas. Lo malo de tal idea es que la paginación se inventó precisamente para hacer innecesario ese tipo de administración. Puesto que el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos están en la memoria principal todo el tiempo, pueden direccionarse sin preocuparse por la administración de su superposición.

6.1.7 Implementación de la segmentación

Hay dos formas de implementar la segmentación: intercambio y paginación. En el primer esquema, cierto conjunto de segmentos está en la memoria en un momento dado. Si se hace referencia a un segmento que no está en la memoria, se trae ese segmento. Si no hay espacio para él, será preciso escribir primero en el disco uno o más segmentos (a menos que ya exista ahí una copia limpia, en cuyo caso puede abandonarse la copia que está en la memoria). En cierto sentido, el intercambio de segmentos es parecido a la paginación por demanda: los segmentos entran y salen conforme se van necesitando.

Sin embargo, la implementación de la segmentación difiere de la de la paginación en un aspecto fundamental: las páginas tienen un tamaño fijo y los segmentos no. La figura 6-10(a) muestra un ejemplo de memoria física que inicialmente contiene cinco segmentos. Considere ahora qué sucede si el segmento 1 se expulsa y se coloca en su lugar el segmento 7, que es más pequeño. Llegamos a la configuración de memoria de la figura 6-10(b). Entre el segmento 7 y el segmento 2 hay un área desocupada, es decir, un hueco. Luego el segmento 4 es sustituido por el segmento 5, como en la figura 6-10(c), y el segmento 3 es sustituido por el segmento 6, como en la figura 6-10(d). Después de un rato de funcionar el sistema, la memo-

ria estará dividida en varios trozos, algunos con segmentos y otros con huecos. Este fenómeno se llama **fragmentación externa** (porque se desperdicia espacio que está fuera de los segmentos, en los huecos que hay entre ellos). La fragmentación externa también se conoce como **cuadriculado**.

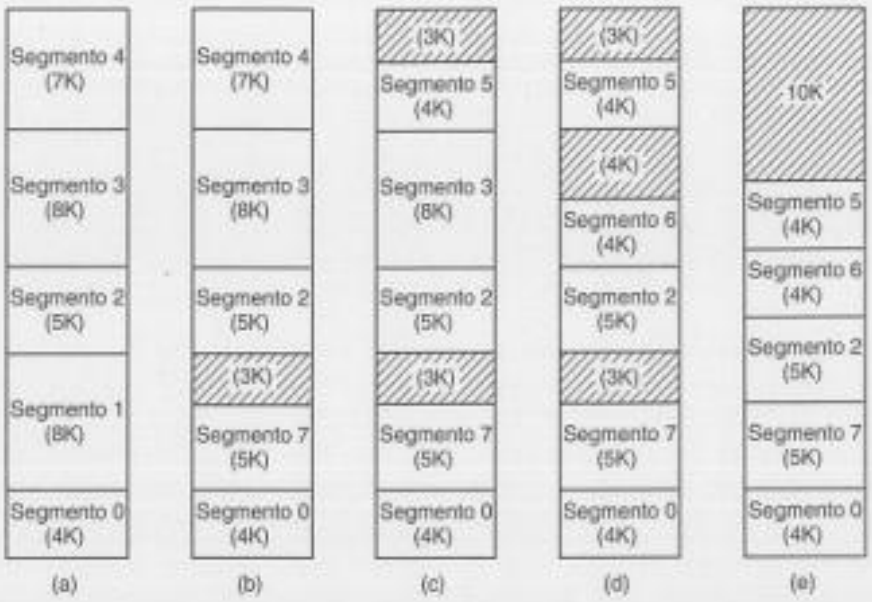

Figura 6-10. (a)-(d) Aparición de fragmentación externa. (e) Eliminación de la fragmentación externa por compactación.

Considere qué sucedería si el programa hiciera referencia al segmento 3 cuando la memoria está experimentando fragmentación externa, como en la figura 6-10(d). El espacio total que ocupan los huecos es de 10K, más que suficiente para el segmento 3, pero como el espacio está distribuido en pequeños fragmentos inútiles no podemos cargar simplemente el segmento 3; será necesario quitar primero algún otro segmento.

Una forma de evitar fragmentación externa es la siguiente: cada vez que aparezca un hueco, los segmentos que están después del hueco se desplazan para acercarlos a la posición de memoria 0 y así eliminar ese hueco y dejar un hueco grande al final. Como alternativa, podríamos esperar hasta que la fragmentación externa sea grave (por ejemplo, más de cierto porcentaje de la memoria total desperdiciado en huecos) antes de realizar la compactación (eliminación de huecos). La figura 6-10(e) muestra cómo quedaría la memoria de la figura 6-10(d) después de una compactación. La intención de compactar la memoria es reunir todos los pequeños huecos inútiles en un solo hueco grande, en el que será posible colocar uno o más segmentos. La compactación tiene la desventaja obvia de que se desperdicia cierto tiempo para llevarla a cabo. En general, no se puede compactar después de que se crea cada hueco, pues se perdería demasiado tiempo.

Si el tiempo requerido para compactar la memoria es demasiado largo, se requiere un algoritmo para determinar cuál hueco deberá usarse para un segmento dado. La gestión de

huecos requiere mantener una lista de las direcciones y tamaños de todos los huecos. Un algoritmo muy popular, llamado **de mejor ajuste**, escoge el hueco más pequeño en el que cabe el segmento requerido. La idea es equiparar los segmentos con los huecos y así evitar gastar un trozo de un agujero grande, que podría necesitarse después para un segmento grande.

Otro algoritmo muy utilizado, llamado **de primer ajuste**, explora circularmente la lista de agujeros y escoge el primer agujero que es lo bastante grande como para albergar el segmento. Esto obviamente toma menos tiempo que examinar toda la lista hasta encontrar el mejor ajuste. Resulta sorprendente que el de primer ajuste es también un mejor algoritmo en términos de desempeño global que el de mejor ajuste, porque este último tiende a generar muchos huecos pequeños totalmente inútiles (Knuth, 1997).

El primero y mejor ajuste tiende a reducir el tamaño medio de los huecos. Cada vez que un segmento se coloca en un hueco mayor que él, lo que sucede casi siempre (los ajustes exactos son raros), el agujero se divide en dos partes. Una parte la ocupa el segmento y la otra es el nuevo hueco. Éste siempre es más pequeño que el hueco viejo. A menos que exista un proceso compensador que vuelva a crear huecos grandes a partir de huecos pequeños, tanto el primer ajuste como el mejor ajuste tarde o temprano llenarán la memoria con huecos pequeños e inútiles.

Uno de esos procesos compensadores es el siguiente. Cada vez que un segmento se saca de la memoria y uno de sus vecinos inmediatos, o ambos, son huecos, no segmentos, los huecos adyacentes se pueden juntar en un hueco grande. Si el segmento 5 se quitara de la figura 6-10(d), los dos huecos que lo flanquean y los 4K utilizados por el segmento se fusionarían en un solo hueco de 11K.

Al principio de esta sección dijimos que hay dos formas de implementar la segmentación: intercambio y paginación. Nuestra explicación hasta ahora se ha centrado en el intercambio. Con este esquema, segmentos enteros se transfieren entre la memoria y el disco por demanda. La otra forma de implementar la segmentación es dividir cada segmento en páginas de tamaño fijo y paginarlas por demanda. En este esquema, algunas de las páginas de un segmento podrían estar en la memoria y otras en disco. Para paginar un segmento se requiere una tabla de páginas individual para cada segmento. Puesto que un segmento no es más que un espacio de direcciones lineal, todas las técnicas que hemos visto hasta ahora para la paginación se aplican a cada segmento. La única novedad aquí es que cada segmento tiene su propia tabla de páginas.

Uno de los primeros sistemas operativos que combinaron la segmentación con la paginación fue **MULTICS** (**Sistema de Información y Cómputo Multiplexado**, *MULtiplexed Information and Computing Service*), que inicialmente fue un proyecto conjunto de MIT, Bell Labs y General Electric (Corbató y Vyssotsky, 1995; y Organick, 1972). Las direcciones de MULTICS tenían dos partes: un número de segmento y una dirección dentro del segmento. Había un segmento de descriptores para cada proceso, que contenía un descriptor para cada segmento. Cuando se presentaba una dirección virtual al hardware, se usaba el número de segmento como índice del segmento de descriptores para localizar el descriptor del segmento accesado, como se muestra en la figura 6-11. El descriptor apuntaba a la tabla de páginas, la que permitía paginar cada segmento de la forma acostumbrada. Para mejorar el desempeño, las combinaciones segmento/página de uso más reciente se guardaban en una **memoria asociativa** de 16 entradas en hardware que permitía buscarlas rápidamente. Aun-

que hace mucho que MULTICS desapareció, su espíritu sigue vivo porque la memoria virtual de todas las CPU de Intel a partir del 386 ha seguido de cerca su modelo.

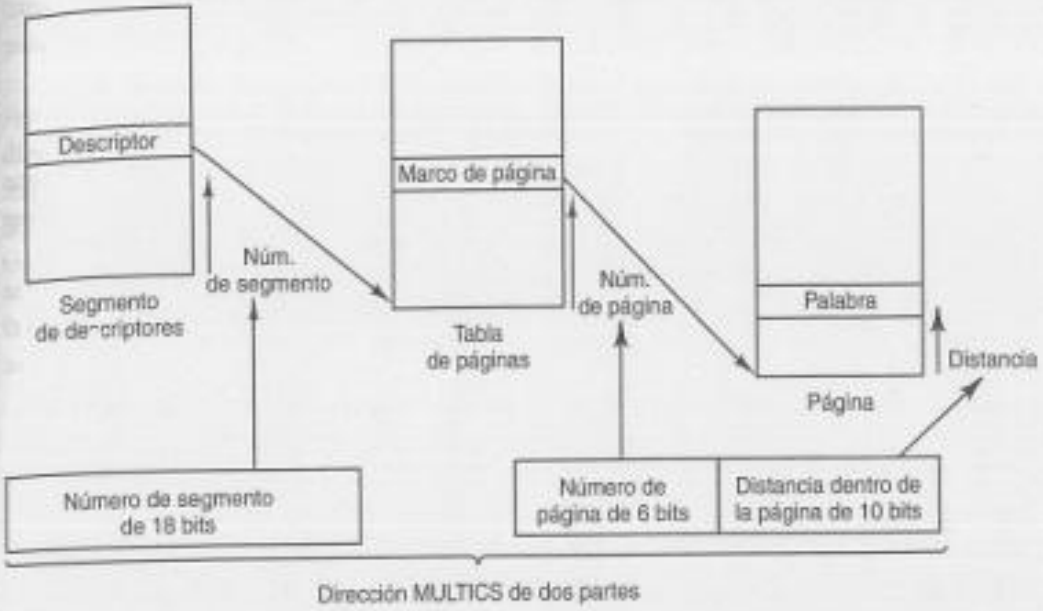

Figura 6-11. Conversión de una dirección MULTICS de dos partes en una dirección de memoria principal.

6.1.8 Memoria virtual en el Pentium II

El Pentium II tiene un sistema de memoria virtual avanzado que maneja paginación por demanda, segmentación pura y segmentación con paginación. El corazón de la memoria virtual del Pentium II consiste de dos tablas: la **tabla de descriptores locales** (**LDT**, *Local Descriptor Table*) y la **tabla de descriptores globales** (**GDT**, *Global Descriptor Table*). Cada programa tiene su propia LDT, pero hay una sola GDT compartida por todos los programas de la computadora. La LDT describe los segmentos locales de cada programa, incluidos los de código, datos, pila, etc., mientras que la GDT describe los segmentos del sistema, incluido el sistema operativo mismo.

Como explicamos en el capítulo 5, para que un programa de Pentium II accese a un segmento, primero debe cargar un selector de ese segmento en uno de sus registros de segmento. Durante la ejecución, CS contiene el selector del segmento de código, DS contiene el selector del segmento de datos, etc. Cada selector es un número de 16 bits, como se muestra en la figura 6-12.

Uno de los bits del selector indica si el segmento es local o global (es decir, si está en la LDT o en la GDT). Otros 13 bits especifican el número de entrada de la LDT o GDT, así que estas tablas sólo pueden contener 8K (2^{13}) descriptores de segmentos. Los otros dos bits

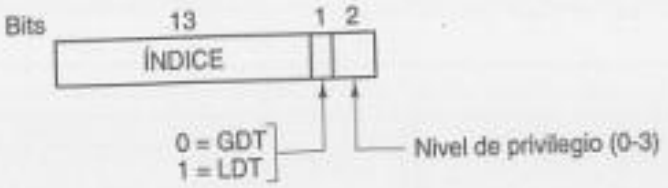

Figura 6-12. Selector de Pentium II.

tienen que ver con la protección y se describirán después. El descriptor 0 no es válido y su uso causa una trampa. Puede cargarse sin peligro en un registro de segmento para indicar que ese registro no está disponible, pero causa una trampa si se usa.

Cuando un selector se carga en un registro de segmento, el descriptor correspondiente se trae de la LDT o GDT y se guarda en registros internos de la MMU, para poder acceder a él rápidamente. Un descriptor consiste en 8 bytes e incluye la dirección base del segmento, su tamaño y otra información, como se muestra en la figura 6-13.

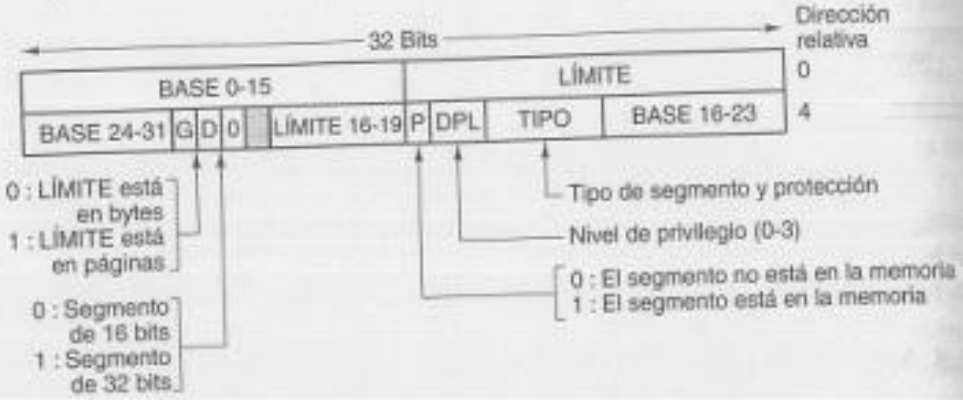

Figura 6-13. Descriptor de segmento de código de Pentium II. Los segmentos de datos son un poco diferentes.

El formato del selector se escogió ingeniosamente a modo de facilitar la localización del descriptor. Primero se selecciona la LDT o bien la GDT, con base en el bit 2 del selector. Luego el selector se copia en un registro de borrador de la MMU y los tres bits de orden bajo se ponen en 0, con lo que el número de selector de 13 bits se multiplica efectivamente por 8. Por último, se le suma la dirección de la tabla LDT o GDT (que se guarda en registros internos de la MMU) para dar un apuntador directo al descriptor. Por ejemplo, el selector 72 se refiere a la entrada 9 de la GDT, que se encuentra en la dirección GDT + 72.

Sigamos los pasos por los que un par (selector, distancia) se convierte en una dirección física. Tan pronto como el hardware sabe cuál registro de segmento se está usando, puede encontrar el descriptor completo que corresponde a ese selector en sus registros internos. Si el segmento no existe (selector 0) o no está en la memoria (P es 0), ocurre una trampa. El primer caso es un error de programación; el segundo obliga al sistema operativo a obtener el segmento.

Luego, el sistema verifica si la distancia rebasa el final del segmento, en cuyo caso ocurre otra trampa. Lógicamente, debería haber un campo de 32 bits en el descriptor que diera el

tamaño del segmento, pero sólo se cuenta con 20 bits, así que se emplea un esquema distinto. Si el campo G (Granularidad) es 0, el campo LÍMITE da el tamaño exacto del segmento, hasta 1 MB; si es 1, LÍMITE da el tamaño del segmento en páginas en lugar de bytes. El tamaño de página del Pentium II nunca es menor que 4 KB, por lo que 20 bits son suficientes para segmentos de hasta 2^{32} bytes.

Suponiendo que el segmento está en la memoria y la distancia no se sale del intervalo, el Pentium II suma el campo de 32 bits BASE del descriptor a la distancia para formar una **dirección lineal**, como se muestra en la figura 6-14. El campo BASE se descompone en tres fragmentos que se distribuyen por todo el descriptor por razones de compatibilidad con el 80286, en el que BASE sólo tiene 24 bits. Así pues, el campo BASE permite que un segmento inicie en cualquier punto dentro del espacio de direcciones lineal de 32 bits.

Figura 6-14. Conversión de un par (selector, distancia) en una dirección lineal.

Si se inhabilita la paginación (con un bit en un registro de control global), la dirección lineal se interpreta como la dirección física y se envía a la memoria para efectuar la lectura o escritura. Así, con la paginación desactivada, tenemos un esquema de segmentación pura, y la dirección base de cada segmento está dada en su descriptor. Por cierto, se permite el traslapo de segmentos, tal vez porque sería demasiado problemático y tardado verificar que todos sean disjuntos.

Por otra parte, si la paginación está habilitada la dirección lineal se interpreta como una dirección virtual y se transforma en una dirección física mediante tablas de páginas, prácticamente igual que en nuestros ejemplos. La única complicación es que con una dirección virtual de 32 bits y páginas de 4 K, un segmento podría contener un millón de páginas, por lo que se usa un mapeo de dos niveles para reducir el tamaño de la tabla de páginas cuando los segmentos son pequeños.

Cada programa en ejecución tiene un **directorio de páginas** que consta de 1024 entradas de 32 bits y se encuentra en una dirección a la que un registro global apunta. Cada entrada de este directorio apunta a una tabla de páginas que también contiene 1024 entradas de 32 bits. Las entradas de la tabla de páginas apuntan a marcos de página. El esquema se muestra en la figura 6-15.

En la figura 6-15(a) vemos una dirección lineal desglosada en tres campos: DIRECTORIO, PÁGINA y DISTANCIA. El campo DIRECTORIO se usa primero como índice del directorio de páginas para localizar un apuntador a la tabla de páginas apropiada. Luego se

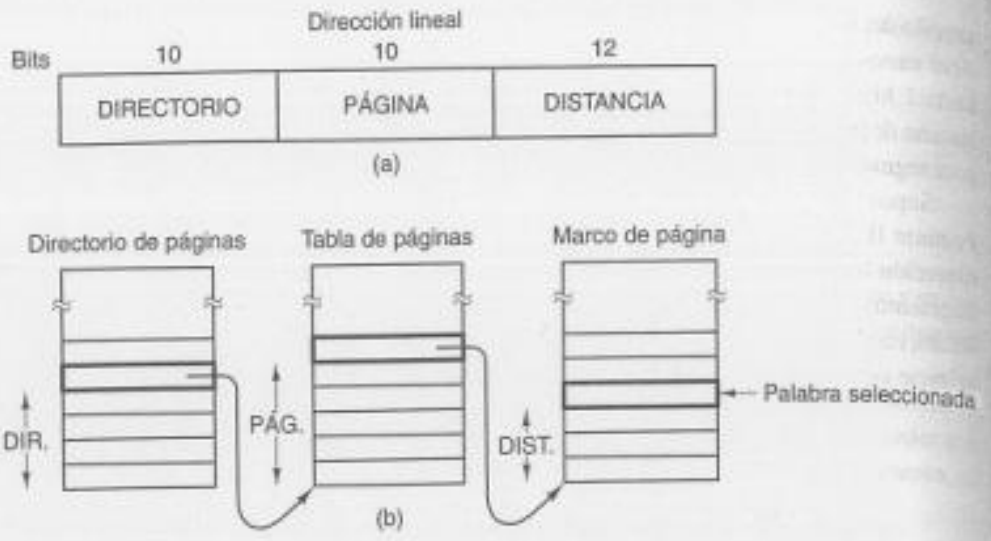

Figura 6-15. Transformación de una dirección lineal en una dirección física.

usa el campo PÁGINA como índice de la tabla de páginas para encontrar la dirección física del marco de página. Por último, DISTANCIA se suma a la dirección del marco de página para obtener la dirección física del byte o palabra direccionado.

Las entradas de la tabla de páginas tienen 32 bits cada una, de los cuales 20 contienen un número de marco de página. Los demás bits controlan el acceso e indican si la página está limpia o no. El hardware ajusta estos bits que son utilizados por el sistema operativo, los bits de protección y otros bits utilitarios.

Cada tabla de páginas tiene entradas para 1024 marcos de página de 4K, por lo que una sola tabla de páginas maneja 4 MB de memoria. Un segmento menor que 4M tendrá un directorio de páginas con una sola entrada, un apuntador a su única tabla de páginas. Así, el gasto extra en el caso de un segmento corto es de sólo dos páginas, en lugar del millón de páginas que se necesitaría en una tabla de páginas de un solo nivel.

A fin de evitar referencias repetidas a la memoria, la MMU del Pentium II cuenta con hardware especial que busca las combinaciones DIRECTORIO-PÁGINA utilizadas más recientemente y las transforma en la dirección física del marco de página correspondiente. Sólo se llevan a cabo los pasos de la figura 6-15 si la combinación actual no se usó recientemente.

Si nos ponemos a pensar un poco, veremos que si se usa paginación en realidad no tiene caso que el campo BASE del descriptor sea distinto de cero. Para lo único que sirve BASE es para desplazarse un poco dentro del directorio de páginas a fin de usar una entrada en ese punto en lugar de al principio. La verdadera razón para incluir BASE es hacer posible la segmentación pura (no paginada), y mantener la compatibilidad con el viejo 80286, que no tenía paginación.

También vale la pena señalar que si una aplicación dada no necesita segmentación y se conforma con un solo espacio de direcciones paginado de 32 bits, es fácil satisfacerla. Se asigna

a todos los registros de segmento el mismo selector, cuyo descriptor tiene BASE = 0 y LÍMITE igual al máximo. Entonces, la distancia de la instrucción será la dirección lineal, empleando un solo espacio de direcciones; esto es, de hecho, paginación tradicional.

Con esto terminamos nuestro tratamiento de la memoria virtual en el Pentium II. Sin embargo, vale la pena hablar un poco de protección, ya que este tema está íntimamente relacionado con la memoria virtual. El Pentium II maneja cuatro niveles de protección, siendo el nivel 0 el que goza de más privilegios, y el nivel 3, el que menos. Dichos niveles se muestran en la figura 6-16. En un instante dado, un programa en ejecución está en cierto nivel, indicado por un campo de dos bits en su **palabra de estado del programa** (PSW), un registro en hardware que contiene los códigos de condición y varios otros bits de estado. Además, cada segmento del sistema pertenece a un nivel dado.

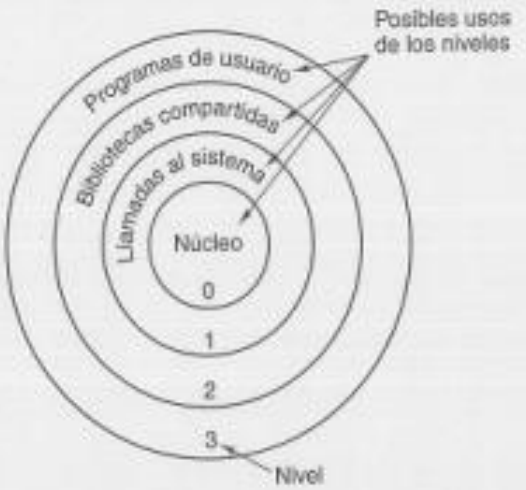

Figura 6-16. Protección en el Pentium II.

En tanto un programa se limite a usar segmentos que estén en su mismo nivel, todo funcionará de maravilla. Se permiten los intentos de accesar a datos que están en un nivel más alto, pero los intentos de accesar a datos en un nivel inferior están prohibidos y causan trampas. Los intentos por invocar procedimientos de otro nivel (más alto o más bajo) están permitidos, pero de forma muy controlada. Para efectuar una llamada internivel, la instrucción CALL debe contener un selector en lugar de una dirección. Este selector designa un descriptor llamado **puerta de llamada**, que da la dirección del procedimiento que se desea invocar. Así, no es posible saltar a la mitad de un segmento de código arbitrario en un nivel distinto; sólo pueden usarse los puntos de ingreso oficiales.

En la figura 6-16 se sugiere un posible uso de este mecanismo. En el nivel 0 está el núcleo del sistema operativo, que se encarga de la E/S, la gestión de memoria y otras tareas cruciales. En el nivel 1 está el manejador de llamadas al sistema. Los programas de usuario pueden invocar procedimientos de este nivel para solicitar que se efectúen llamadas al sistema, pero sólo es posible invocar una lista específica y protegida de procedimientos. El nivel 2 contiene procedimientos de biblioteca, posiblemente compartidos entre muchos programas

en ejecución. Los programas de usuario pueden invocar estos procedimientos, pero no modificarlos. Por último, los programas de usuario se ejecutan en el nivel 3, que es el que menos protección tiene. Al igual que el esquema de gestión de memoria del Pentium II, el sistema de protección se basa en el de MULTICS.

Las trampas e interrupciones usan un mecanismo similar a las puertas de llamada; también hacen referencia a descriptores, no a direcciones absolutas, y dichos descriptores apuntan a procedimientos específicos que deben ejecutarse. El campo TIPO de la figura 6-13 distingue entre segmentos de código, segmentos de datos y los diversos tipos de puertas.

6.1.9 Memoria virtual en el UltraSPARC II

El UltraSPARC II es una máquina de 64 bits y maneja una memoria virtual paginada basada en una dirección virtual de 64 bits. Sin embargo, por razones de ingeniería y costos, los programas no pueden usar todo el espacio de direcciones virtual de 64 bits. Sólo se reconocen 44 bits, por lo que los programas no pueden exceder 1.8×10^{13} bytes. La memoria virtual permitida se divide en dos zonas de 2^{43} bytes cada una, una en la parte superior del espacio de direcciones virtual y una en la parte inferior. En medio hay un hueco que contiene direcciones que no pueden usarse; un intento por usarlas causa un fallo de página.

La memoria física máxima en un UltraSPARC II es de 2^{41} bytes, que equivale a unos 2200 GB, lo bastante grande para casi todas las aplicaciones ordinarias. Se reconocen cuatro tamaños de página: 8 KB, 64 KB, 512 KB y 4 MB. En la figura 6-17 se ilustran las correspondencias implícitas en estos cuatro tamaños de página.

Figura 6-17. Transformaciones virtual-física en el UltraSPARC II.

Por lo extremadamente grande de espacio de direcciones virtual, una tabla de páginas sencilla como la del Pentium II no sería práctica. En vez de ello, la MMU del UltraSPARC II adopta una estrategia muy diferente: contiene una tabla en hardware llamada **buffer de consulta para traducción** (**TLB**, *Translation Lookaside Buffer*) que transforma números de página virtual en números de marco de página físico. Para el tamaño de página de 8K hay 2^{31} números de página virtual, o sea, más de dos millones. Es evidente que no todos pueden estar en el mapa.

En vez de ello, el TLB sólo contiene los números de página virtual más recientemente usados. Se sigue la pista por separado a las páginas de instrucciones y a las de datos, y el TLB guarda los 64 números de página virtual más recientemente usados de cada categoría. Cada entrada del TLB contiene un número de página virtual y el número de marco de página físico correspondiente. Cuando se presentan a la MMU un número de proceso, llamado **contexto**, y

una dirección virtual dentro de ese contexto, la MMU emplea circuitos especiales para comparar simultáneamente el número de página virtual contenido en la dirección virtual con todas las entradas del TLB. Si se encuentra una concordancia, el número de marco de página contenido en esa entrada del TLB se combina con la distancia tomada de la dirección virtual para formar una dirección física de 41 bits y producir algunas banderas, como los bits de protección. El TLB se ilustra en la figura 6-18(a).

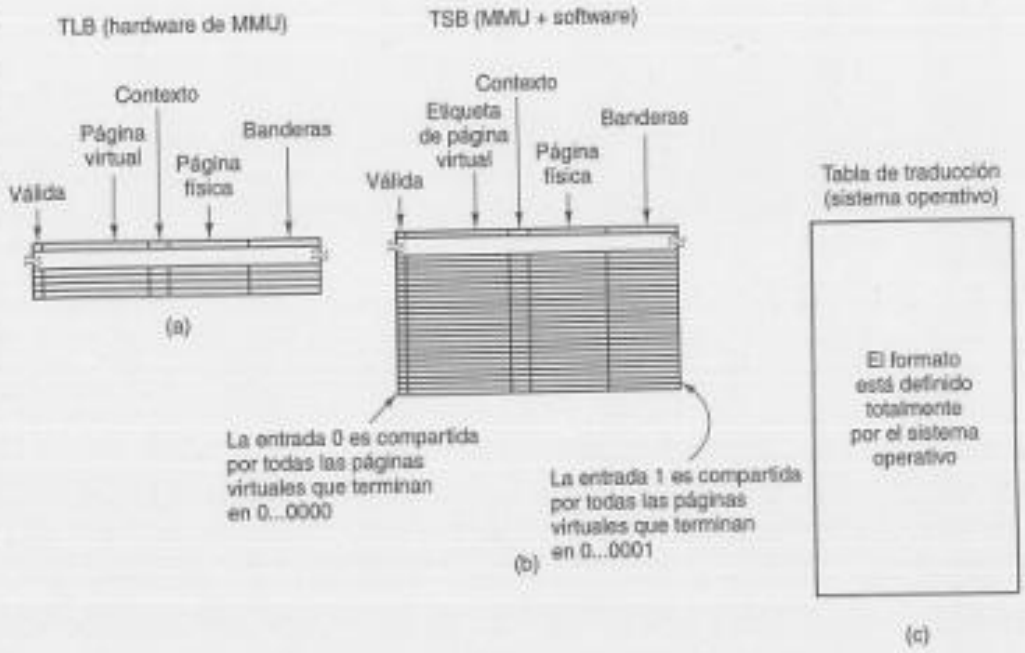

Figura 6-18. Estructuras de datos empleadas para traducir direcciones virtuales en el UltraSPARC. (a) TLB. (b) TSB. (c) Tabla de traducción.

Si ninguna de las entradas de la TLB concuerda, ocurre un **fallo de TLB**, que causa un salto al sistema operativo a través de una trampa. El manejo del fallo corresponde al sistema operativo. Cabe señalar que un fallo de TLB es muy diferente de un fallo de página. Puede ocurrir un fallo de TLB aunque la página a la que se hizo referencia esté en la memoria. En teoría, el sistema operativo puede hacer lo que quiera para cargar una nueva entrada de TLB para la página virtual requerida. Sin embargo, para acelerar esta operación crucial, se puede contar con cierta ayuda del hardware si es que el software coopera.

En particular, se espera que el sistema operativo mantenga una caché en software con las entradas de TLB más utilizadas en una tabla llamada **buffer de almacenamiento para traducción** (**TSB**, *Translation Storage Buffer*). Esta tabla se organiza como caché de páginas virtuales con correspondencia directa. Cada entrada de TSB de 16 bytes se refiere a una página virtual y contiene un bit de validez, el número de contexto, la etiqueta de dirección virtual, el número de página física y algunos bits que sirven como banderas. Si el tamaño del caché es de, digamos, 8192 entradas, todas las páginas virtuales cuyos 13 bits de orden bajo sean 0000000000000 competirán por la entrada 0 del TSB. Así mismo, todas las páginas virtuales cuyos bits de orden bajo sean 0000000000001 competirán por la entrada 1 del TSB,

como se muestra en la figura 6-18(b). El tamaño del TSB lo determina el software y se comunica a la MMU por medio de registros especiales a los que sólo el sistema operativo tiene acceso.

Cuando ocurre un fallo de TLB, el sistema operativo verifica si la entrada correspondiente en el TSB contiene la página virtual requerida. La MMU ayuda aquí calculando la dirección de esta entrada y colocándola en un registro interno MMU accesible para el sistema operativo. Si ocurre un acierto de caché en el TSB, se borrará alguna entrada del TLB y la entrada de TSB requerida se copiará en el TLB. El hardware también ayuda a escoger cuál entrada del TLB se borrará utilizando un algoritmo LRU de un bit.

Si la búsqueda en el TSB no corre con suerte y la página virtual a la que se hizo referencia no está en la caché, el sistema operativo usa otra tabla para localizar la información acerca de la página, que podría o no estar en la memoria principal. La tabla que se usa para esta consulta de último recurso se llama **tabla de traducción**. Puesto que el hardware no ayuda con las consultas a esta tabla, el sistema operativo está en libertad de usar el formato que quiera. Por ejemplo, puede usar *hashing* dividiendo el número de página virtual entre algún número primo p, y usar el residuo como índice de una tabla de apuntadores, cada uno de los cuales apunta a una lista enlazada de entradas de página virtual que dan p por *hashing*. Cabe señalar que estas entradas no son las páginas, sino entradas de TSB. El resultado de buscar una página en la tabla de traducción podría ser localizar la página en la memoria, en cuyo caso se actualizará la entrada de TSB de la caché en software, pero también podría ser el descubrimiento de que la página no está en la memoria, en cuyo caso se inicia la acción estándar de fallo de página.

Resulta interesante comparar los esquemas de paginación del Pentium II y el UltraSPARC II. El Pentium II maneja segmentación pura, paginación pura y segmentos paginados. El UltraSPARC II sólo tiene paginación. El Pentium II también usa hardware para recorrer la tabla de páginas y volver a cargar el TLB en caso de un fallo de TLB. El UltraSPARC II simplemente transfiere el control al sistema operativo cuando hay un fallo de TLB.

La razón primordial de esta diferencia es que el Pentium II usa segmentos de 32 bits, y tales segmentos tan pequeños (sólo un millón de páginas) pueden manejarse con tablas de páginas convencionales. En teoría, el Pentium II tendría problemas si un programa usara miles de segmentos, pero dado que ninguna versión de Windows ni de UNIX reconoce más de un segmento por proceso, el problema no se presenta. En contraste, el UltraSPARC II es una máquina de 64 bits y puede tener hasta 2000 millones de páginas, por lo que las tablas de páginas convencionales no funcionan. En el futuro, todas las máquinas tendrán espacios de direcciones virtuales de 64 bits, por lo que esquemas como el del UltraSPARC II se convertirán en la norma. En (Jacob y Mudge, 1998b) se hace una comparación del Pentium II, el UltraSPARC II y otros cuatro esquemas de memoria virtual.

6.1.10 Memoria virtual y uso de cachés

Aunque a primera vista no parece haber relación entre la memoria virtual (paginada por demanda) y el uso de cachés, en lo conceptual son muy similares. Con memoria virtual todo el programa se mantiene en disco y se divide en páginas de tamaño fijo. Un subconjunto de