APUNTES DE

Introducción a los Sistemas Operativos

2º DE GRADO EN INGENIERÍA INFORMÁTICA

TEMA 5. ADMINISTRACIÓN DE MEMORIA

CURSO 2021/2022

ÍNDICE GENERAL

CAPÍTULO 5

ADMINISTRACIÓN DE MEMORIA

5.1 INTRODUCCIÓN

Como programadores, a la hora de construir un programa, nos gustaría que este se ejecutara desde un almacenamiento muy rápido, de uso exclusivo, grande y no volátil. De esta manera, entre otras cosas, nuestro programa tendría un gran rendimiento, podría leer o escribir grandes cantidades de datos a gran velocidad y no se tendría que preocupar de las posibles pérdidas de datos ante una caída del sistema. Sin embargo, con la tecnología actual, construir un almacenamiento así sería difícil (cuando no imposible) y, sobre todo, muy caro.

Para poder construir un almacenamiento similar al descrito de forma sencilla y a un precio razonable, hay que combinar distintas tecnologías de almacenamiento a través de una *jerarquía de memoria*. En ella, desde un punto de vista lógico, la memoria más rápida, pequeña y cara se encontrará cerca del procesador, y la más lenta, grande y barata estará lejos del procesador. En la actualidad, esta jerarquía está formada por unos pocos megabytes de memoria caché, unos cuantos gigabytes de memoria RAM y varios cientos de gigabytes o algunos terabytes de almacenamiento en disco. Algunas partes de la jerarquía de memoria están controladas por el hardware, como es la memoria caché, pero otras son responsabilidad del sistema operativo, como son la memoria RAM y el almacenamiento en disco.

La jerarquía de memoria y los detalles de su funcionamiento deben ocultarse al programador, por lo que el sistema operativo debe abstraerla y administrarla. A la parte del sistema operativo que se encarga de esta tarea se le llama *administrador de memoria*. Puesto que el funcionamiento de la jerarquía de memoria determina en gran medida el rendimiento de una máquina, el administrador de memoria debe gestionarla de forma eficiente: controlar qué partes de la memoria RAM están en uso, asignar memoria RAM a los procesos según la necesiten, liberar esa memoria cuando ya no esté en uso, mover información entre el disco y la memoria RAM cuando sea necesario, etc. Un aspecto importante a tener en cuenta es que, como hemos visto en el tema 2, es deseable que varios procesos compartan la CPU. Para ello será necesario que también compartan la memoria de forma segura con la ayuda del sistema operativo.

En este tema vamos a ver varias soluciones de administración de memoria con sus ventajas e inconvenientes. También veremos cómo la selección de una u otra solución depende de muchos factores, pero sobre todo del diseño del hardware. De todas las soluciones estudiadas, la memoria virtual será sin duda la más importante por ser una de las técnicas más usadas hoy en día.

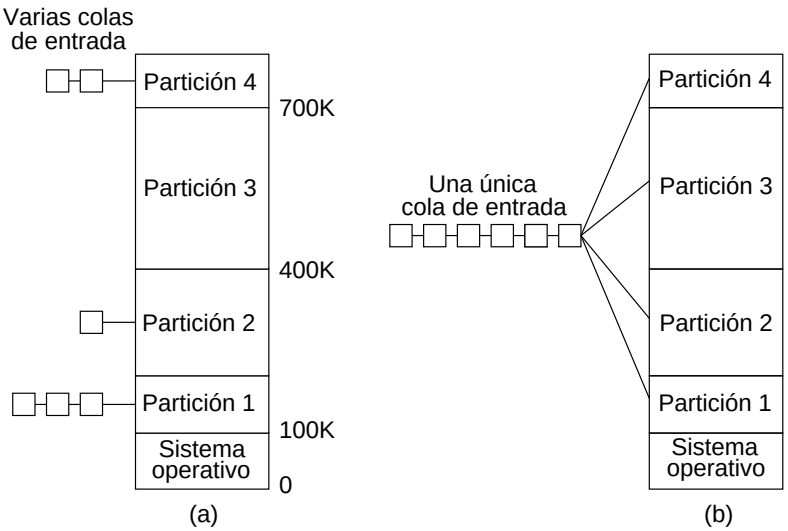


Figura 5.1. (a) Particiones fijas de memoria con colas de entrada independientes para cada partición. (b) Particiones fijas de la memoria, con una única cola de entrada.

5.2 ADMINISTRACIÓN DE MEMORIA SIN MEMORIA VIRTUAL

En primer lugar, vamos a ver esquemas de administración de memoria en los que no se utilizan técnicas propias de la memoria virtual (como la paginación), lo que hará que a cada proceso se le asigne una única zona contigua de memoria principal.

5.2.1 Multiprogramación con particiones fijas

El esquema más sencillo parar tener varios procesos a la misma vez en memoria es asignar una zona de tamaño fijo al sistema operativo y dividir el resto de la memoria principal en N partes o *particiones* de igual o diferente tamaño. Una vez creadas, las particiones no cambian de tamaño.

Las diferentes particiones se reparten entre los procesos según se van creando estos. Para realizar este reparto hay varias opciones. La primera posibilidad es tener una cola por partición y colocar cada trabajo en la cola de la partición más pequeña en la que quepa (ver figura 5.1(a)). El inconveniente de esta propuesta es que las colas de las particiones grandes casi siempre estarán vacías mientras que las colas de las particiones pequeñas serán largas, pues la gran mayoría de procesos son de pequeño tamaño. Se podría pensar en hacer todas las particiones pequeñas para repartir mejor los procesos entre ellas, pero esto haría imposible ejecutar los trabajos grandes que llegan de vez en cuando, por lo que siempre es necesario que exista alguna partición grande (al menos, tan grande como el proceso de mayor tamaño que pueda llegar).

Otra posibilidad es tener una única cola para todas las particiones (ver figura 5.1(b)). Cuando una partición queda libre, se busca la primera tarea de la cola que quepa en dicha partición. Como esta opción puede hacer que una tarea pequeña desperdicie una partición grande, también se puede buscar en toda la cola el trabajo más grande que quepa. Esto, sin embargo, discrimina a las tareas pequeñas que podrían ser tareas interactivas, las cuales requieren un procesamiento rápido. Una solución a este último problema sería tener una partición pequeña. Otra solución sería establecer un límite L. Si una tarea se excluye más de L veces, ya no se excluye más.

Este esquema de administración de memoria es sencillo, pero presenta algunos inconvenientes. Uno de ellos es que el *grado de multiprogramación*, es decir, el número

de procesos que podemos tener a la vez en memoria compartiendo la CPU, se ve limitado por el número de particiones. Otro problema es que los procesos pequeños producen mucha fragmentación interna, ya que solo aprovecharán una parte de la partición asignada. Se podría pensar en asignar el resto de la partición a otro proceso, pero esto no es posible, ya que se protegen particiones enteras (ver apartado 5.2.3), por lo que, si dos o más procesos compartieran una partición, no estarían protegidos. Finalmente, un tercer problema es que también se puede producir fragmentación externa. Esto sucede cuando hay una una o más particiones libres que no son lo suficientemente grandes para ninguno de los trabajos que están esperando.

5.2.2 Multiprogramación con particiones variables

En un sistema por lotes, la organización de memoria en particiones fijas es sencilla y eficaz: basta con tener en memoria suficientes programas en ejecución como para mantener ocupada la CPU todo el tiempo.

La situación es distinta en el tiempo compartido: por lo general, existen más procesos de usuario de los que puede albergar la memoria, por lo que el exceso de procesos debe estar en disco. Evidentemente, los procesos suspendidos en disco necesitan pasar a memoria en algún momento para reanudar su ejecución. Aunque el intercambio de procesos entre la memoria y el disco se podría hacer con particiones fijas, esta solución no sería óptima, ya que, para poder albergar muchos procesos en memoria, sería necesario crear muchas particiones pequeñas que no podrían ser usadas por procesos algo más grandes.

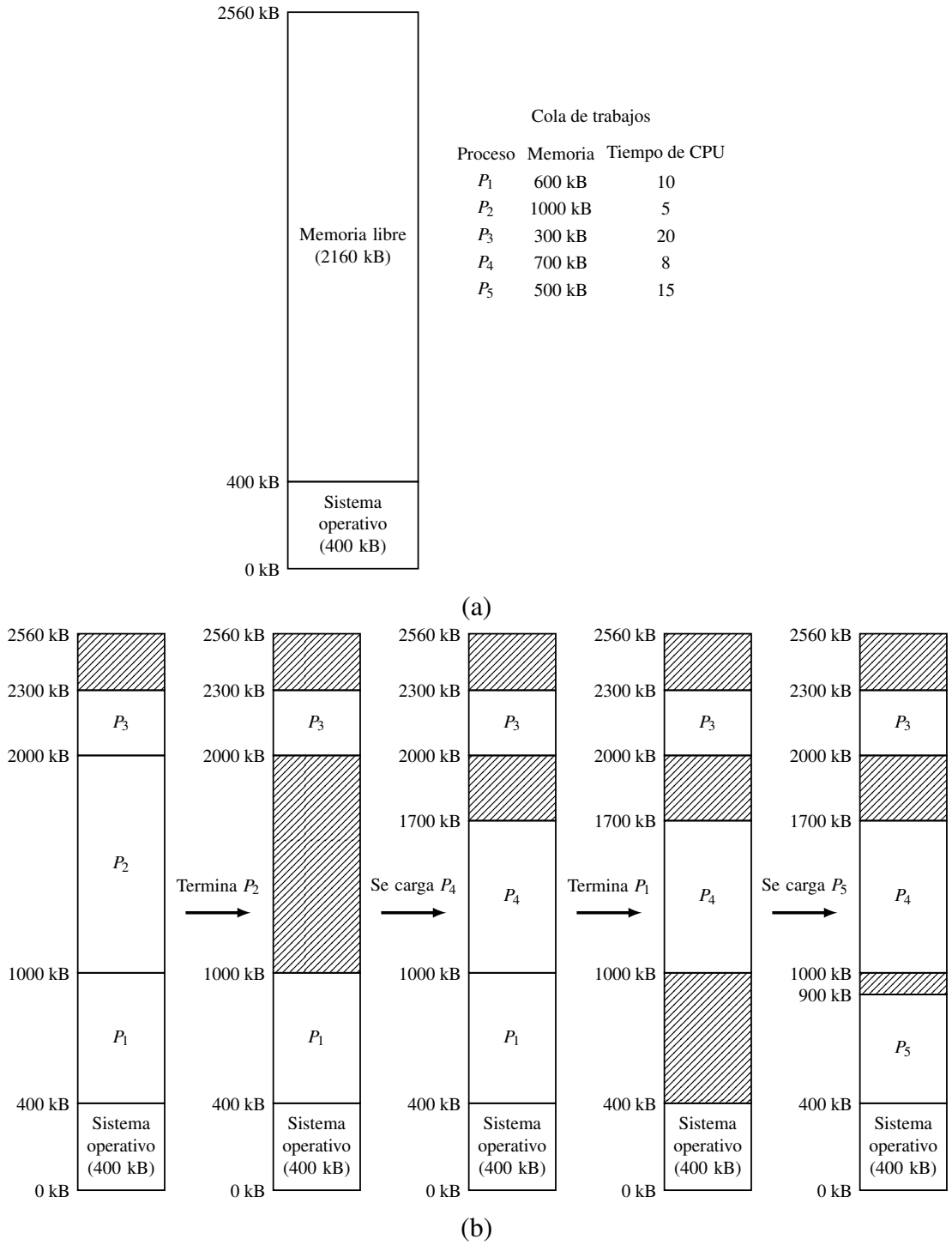
Una solución mejor es la *multiprogramación con particiones variables*, donde el número, posición y tamaño de las particiones varían dinámicamente en función del número y tamaño de los procesos en ejecución. Con este esquema, no se está sujeto a un número fijo de particiones que pudieran ser muy grandes o demasiado pequeñas. En teoría, se consigue un mejor uso de la memoria a costa de una mayor complejidad. Ahora, a cada proceso se le asigna una partición con el tamaño exacto de la memoria que necesita. Las particiones se crean, desaparecen o se intercambian a disco según las necesidades del sistema y de los procesos. En la figura 5.2 podemos ver un ejemplo de cómo se asigna la memoria a los procesos en este esquema según se van ejecutando.

Políticas de asignación de huecos

Cuando hay que asignar memoria a un nuevo proceso, se debe buscar un hueco. Hay distintas formas o políticas para buscar ese hueco libre:

- *Primero en ajustarse*: el administrador de memoria revisa los huecos desde el principio de la memoria hasta encontrar uno lo suficientemente grande. El hueco se divide en dos partes: una para el proceso y otra que pasa a ser un hueco libre (a no ser que se produzca un ajuste perfecto). Este algoritmo es rápido, pues busca lo menos posible.

- *Siguiente en ajustarse*: es una variante del anterior. El funcionamiento es el mismo, salvo que la búsqueda del hueco no se inicia siempre desde el principio de la memoria, sino desde el punto en el que se quedó la búsqueda anterior. Si se llega al final de la memoria y no se encuentra un hueco adecuado, reinicia la búsqueda desde el principio de la memoria hasta llegar al punto desde el que se comenzó. Este algoritmo también es rápido, porque, como el anterior, busca lo menos posible.



Cola de trabajos

Proceso	Memoria	Tiempo de CPU
P_1	600 kB	10
P_2	1000 kB	5
P_3	300 kB	20
P_4	700 kB	8
P_5	500 kB	15

Figura 5.2. (a) Datos iniciales para el ejemplo. (b) Asignación de memoria según se van ejecutando los trabajos.

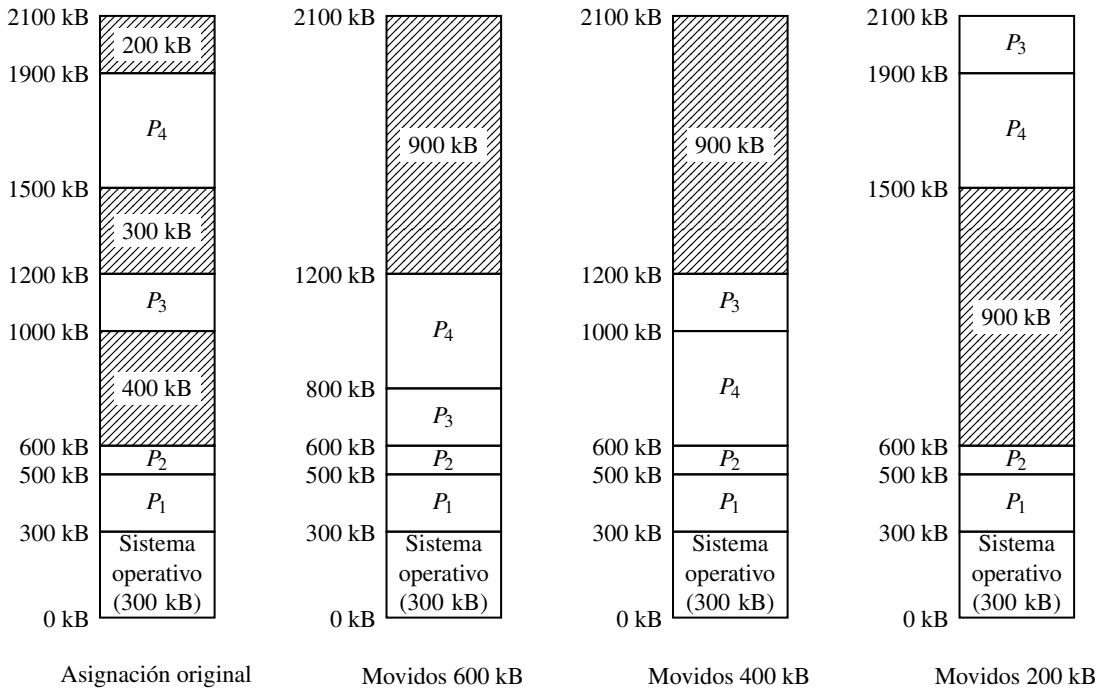


Figura 5.3. Comparación de distintas formas de compactar la memoria.

- *Mejor en ajustarse*: busca entre todos los huecos el más pequeño en el que quepa el proceso. La idea es buscar un hueco cercano al tamaño real necesario. Esta política presenta dos problemas. Por un lado, es lenta, pues tiene que buscar en todos los huecos. Por otro lado, suele desperdiciar más memoria que otras políticas, ya que tiende a producir huecos demasiado pequeños que no se pueden utilizar.

- *Peor en ajustarse*: toma siempre el hueco libre más grande. Su objetivo es que el nuevo hueco obtenido sea suficientemente grande para ser útil. Al igual que la anterior, esta política es lenta, pues busca en todos los huecos. Sin embargo, funciona bastante bien respecto al aprovechamiento de la memoria.

Compactación

En la multiprogramación con particiones variables puede producirse una gran fragmentación externa debido a la aparición de muchos huecos de pequeño tamaño que difícilmente son útiles. Debido a la flexibilidad de las particiones variables, podemos combinar todos los huecos en uno solo si movemos algunos procesos en memoria. Esta técnica se conoce como *compactación*. La figura 5.3 muestra diversas formas de compactar una memoria fragmentada. Evidentemente, la compactación será más rápida cuanto menos cantidad de información haya que mover.

Generalmente, la compactación no se suele utilizar porque consume mucho tiempo de CPU, aunque existieron ordenadores que disponían de un hardware especial para acelerar el proceso.

Obsérvese que, además de la posible fragmentación externa, siempre existe fragmentación interna, pues a los procesos se les asigna más memoria de la estrictamente necesaria para que haya hueco entre la pila y la zona de datos de cada proceso. Este hueco permite que tanto la pila como la zona de datos puedan crecer, pero rara vez se usa completamente, lo que hace que se desperdicie, aunque sea parcialmente.

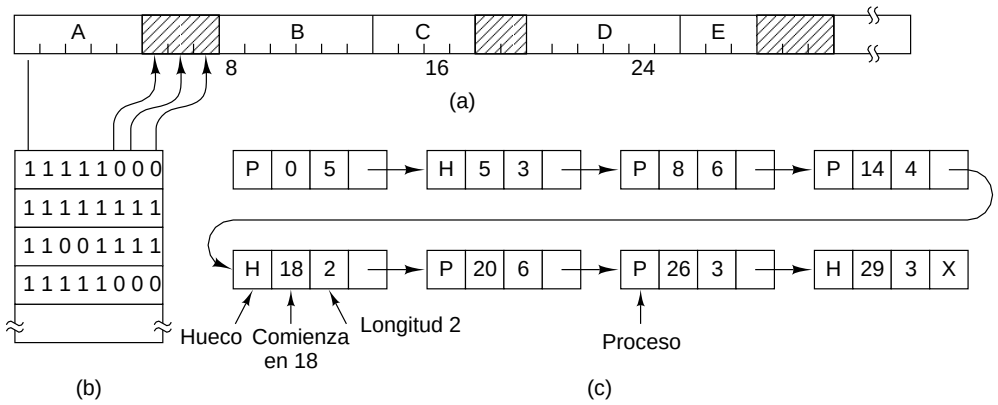


Figura 5.4. (a) Una parte de la memoria con cinco procesos y tres huecos. Las marcas muestran las unidades de asignación de la memoria. Las regiones sombreadas (0 en el mapa de bits) están libres. (b) El mapa de bits correspondiente. (c) La misma información como lista ligada[1].

Administración de la memoria libre

Hasta ahora hemos visto varias formas de asignar memoria a los distintos procesos sin preocuparnos de llevar un registro de la memoria libre y de la ocupada. Dos técnicas utilizadas para llevar este registro son los mapas de bits y las listas ligadas.

En la *administración de memoria con mapas de bits* se divide la memoria en unidades de asignación (o bloques), todas del mismo tamaño. A cada unidad de asignación le corresponde un bit en el mapa de bits, el cual toma el valor 0 si la unidad está libre y 1 si está ocupada (o viceversa). La figura 5.4(b) muestra este caso.

Un aspecto importante de esta técnica es el tamaño de la unidad de asignación. Si es pequeño, se necesitará un mapa de bits grande. Si el tamaño de la unidad es grande, se puede perder una parte importante de la memoria en la última unidad asignada a un proceso por fragmentación interna, ya que es habitual que el tamaño del proceso no sea múltiplo exacto del tamaño de la unidad de asignación.

Un posible problema de este método es que, si se quiere colocar en memoria un proceso de k unidades de tamaño, el administrador de memoria debe buscar en el mapa una cadena de k ceros consecutivos, lo cual puede ser lento si la cadena de ceros necesaria es larga debido al tamaño del proceso.

En la *administración de memoria con listas ligadas* se mantiene un registro de la memoria mediante una lista ligada de los segmentos de memoria asignados o libres. Cada segmento puede ser un proceso (P), es decir, una zona ocupada, o un hueco (H). Además del tipo de segmento (proceso o hueco), cada entrada de la lista especifica la dirección donde comienza el segmento, su longitud y un apuntador a la siguiente entrada de la lista (ver figura 5.4(c)).

Cuando un proceso termina o se intercambia, se debe fusionar el hueco que deja con los huecos adyacentes, si los hay (ver figura 5.5). Observa, sin embargo, que esta fusión no es necesaria si se utiliza un mapa de bits, pues se realiza de forma implícita.

Aunque tanto el mapa de bits como la lista ligada nos permiten conocer qué zonas de memoria están libres y qué zonas están ocupadas, ninguna de ellas nos dice qué zona ocupada corresponde a cada proceso. Generalmente, la información sobre la memoria

[1] Observa que la dirección de comienzo y el tamaño de cada segmento se dan en unidades de asignación de memoria en esta figura. Esto no es estrictamente necesario, ya que ambos datos se podrían dar en bytes perfectamente sin utilizar unidades de asignación para ello.

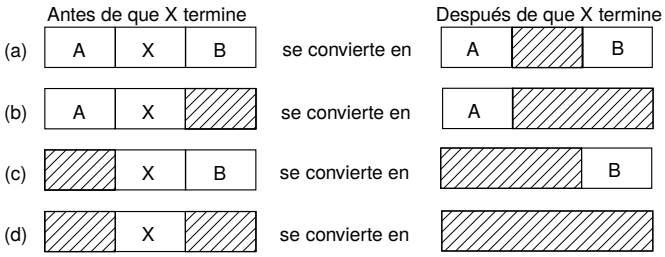


Figura 5.5. Cuatro combinaciones de vecinos para el proceso X que concluye.

asignada a un proceso se guarda en su BCP (o PCB) o en alguna estructura de datos separada a la que se puede llegar a través del BCP.

Asignación del hueco de intercambio

Los algoritmos de la sección anterior se utilizan para mantener un registro de la memoria principal, de forma que, cuando los procesos se intercambien desde el disco al ser reanudados, el sistema pueda encontrar un hueco para ellos.

Los procesos suspendidos que no caben en memoria se guardan en una zona del disco conocida como *área de intercambio*. Esta zona puede ser una partición o un fichero. En algunos sistemas, cuando un proceso se encuentra en memoria, no hay un hueco en disco asignado para él. Cuando se suspenda y deba intercambiarse a disco, se deberá buscar un hueco en el área de intercambio y asignárselo. En otros sistemas, en cambio, al crearse un proceso se le asigna un hueco de intercambio en el disco. Cuando el proceso sea intercambiado desde memoria, siempre pasará al hueco asignado, en vez de buscar un hueco en disco cada vez que se intercambie.

Los algoritmos para la administración del área de intercambio, tanto para llevar un registro como para hacer un reparto del espacio de intercambio, pueden ser los mismos que los descritos en esta sección para la administración de la memoria principal. La única diferencia es que el hueco en disco necesario para un proceso debe representarse como un número entero de bloques del disco (estos bloques son las unidades de asignación ahora).

5.2.3 Reubicación y protección

La multiprogramación, ya sea con particiones fijas o variables, permite tener varios procesos a la vez en memoria, cada uno en una partición, lo que da lugar a dos problemas que hay que solucionar: el de la *reubicación* y el de la *protección*.

El problema de la reubicación (o relocalización) surge porque, cuando un programa se ejecuta, su proceso puede ir a cualquier partición. Por tanto, como las particiones se encuentran en direcciones de memoria distintas y no se sabe *a priori* a qué partición va a ir el proceso, el programa debe utilizar *código relocalizable*, es decir, código que se pueda ejecutar correctamente independientemente de las direcciones de memoria que ocupe.

El segundo problema, el de la protección, aparece porque hay que proteger al sistema operativo del resto de procesos y también hay que proteger a un proceso de los demás.

Un mecanismo hardware que soluciona ambos problemas es tener un registro base y un registro límite. Estos registros especiales se encuentran en la CPU. Cuando la CPU se asigna a un proceso, el registro base se carga con la dirección de memoria donde comienza la partición asignada al proceso y el registro límite se carga con el tamaño de dicha partición. Estos valores se cogen del BCP del proceso. Si la CPU se asigna a otro

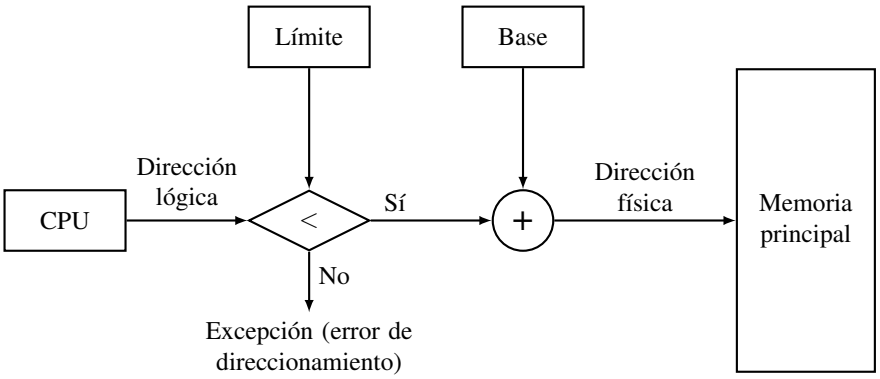


Figura 5.6. Apoyo del hardware para los registros base y límite.

proceso, estos registros se cargan de nuevo con los valores correspondientes a dicho proceso.

El código del proceso que se está ejecutando en la CPU puede generar cualquier dirección desde 0 hasta el tamaño de la partición del proceso menos 1 (porque contamos desde 0). Si en su ejecución el proceso genera una dirección mayor, se considerará una dirección no válida y se producirá una excepción que provocará la terminación del proceso. En caso contrario, se tratará de una dirección válida a la que se le sumará el valor del registro base para obtener la dirección final con la que acceder a la memoria principal. La figura 5.6 muestra el funcionamiento de estos registros.

Observa que con esta técnica hay una separación entre las direcciones de memoria generadas por los procesos y las direcciones de memoria que finalmente usan. A las primeras se les llama *direcciones lógicas*; el conjunto de direcciones lógicas que pueden generar un proceso constituye su *espacio de direcciones lógicas*. A las segundas se les llama *direcciones físicas*. Observa también que los espacios de direcciones lógicas de todos los procesos comienzan en la dirección 0. Sin embargo, como a las direcciones lógicas generadas por un proceso se les suma su registro base, y este es distinto para cada proceso, dos direcciones lógicas iguales de procesos diferentes se corresponderán con direcciones físicas distintas, por lo que los procesos no se interferirán entre sí.

La figura 5.7(a) muestra el valor de los registros base y límite cuando la CPU está ocupada con el proceso A. Como vemos, cuando el proceso A genera la dirección lógica de memoria 1 000, esta termina convirtiéndose en la dirección física 51 000 de memoria. Cuando la CPU pasa al proceso B, los registros base y límite cambian de valor, tal y como muestra la figura 5.7(b). Ahora, cuando el proceso B acceda a la su dirección lógica 1 000, se terminará accediendo a la dirección física 301 000. Por lo tanto, la misma dirección lógica en procesos distintos corresponderá a direcciones físicas diferentes, como ya hemos dicho. De hecho, este funcionamiento es el que protege al sistema operativo y a los procesos, ya que un proceso nunca podrá generar una dirección lógica válida que le permita acceder a una dirección de memoria principal asignada a otro proceso.

Como podemos ver, la figura 5.7 también muestra una zona de memoria sombreada en el proceso B. Esta zona es memoria asignada al proceso para que su pila pueda crecer hacia direcciones menores y su zona de datos pueda crecer hacia direcciones mayores de memoria, algo bastante habitual durante la ejecución de un proceso.

5.3 PAGINACIÓN

La memoria virtual es un esquema de administración de memoria que permite ejecutar programas cuyo tamaño total, incluyendo código, datos y pila, puede exceder la

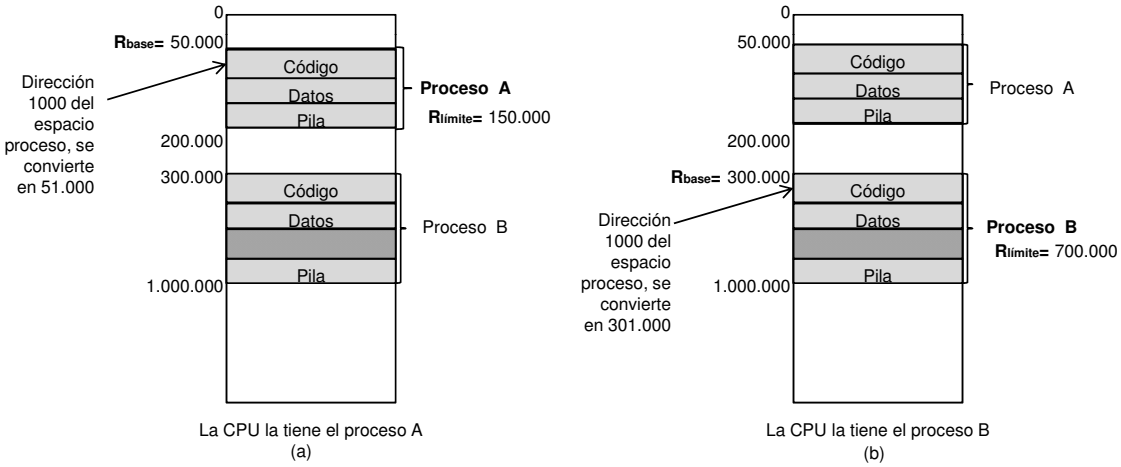


Figura 5.7. Valores de los registros base y límite cuando la CPU ejecuta (a) un proceso A o (b) un proceso B, y correspondencia entre la dirección lógica de memoria 1000 en cada caso con una dirección física de memoria diferente.

cantidad de memoria física que se les puede asignar. Para conseguir esto, el sistema mantiene en memoria principal solo las partes del proceso que se están usando y guarda el resto del proceso en disco. La decisión de qué partes se encuentran en memoria en cada momento, y qué partes en disco, es responsabilidad del sistema operativo, que transfiere información entre disco y memoria de forma automática y transparente para los procesos, por lo que el programador puede construir su programa olvidándose de estos detalles.

En lo que queda de tema analizaremos dos técnicas de memoria virtual, empezando en esta sección por la más importante de ellas: la paginación. Como veremos, todas ellas suponen la división de un proceso en trozos para, como hemos dicho, mantener en memoria principal solo las partes activas. La forma en la que hagamos esta división determinará en gran medida el funcionamiento de la técnica.

5.3.1 Funcionamiento de la paginación

En la paginación, la división de un proceso en trozos y el intercambio de estos trozos entre la memoria principal y el disco son por completo responsabilidad del hardware y del sistema operativo. Es decir, un proceso no es consciente de que solo parte de su espacio de direcciones está en memoria física, mientras que el resto se guarda en disco. Para conseguir esta transparencia, se distingue entre las direcciones que un proceso puede generar, llamadas *direcciones virtuales*, y las direcciones de memoria principal que finalmente se corresponden con esas direcciones virtuales; a las direcciones de memoria principal las llamamos *direcciones físicas*. Esta distinción es similar a la que se hace entre direcciones lógicas y físicas en la técnica de los registros base y límite (ver sección 5.2.3).

Todas las posibles direcciones virtuales que puede generar un proceso conforman su *espacio de direcciones virtuales*. Normalmente, el tamaño de este espacio viene limitado por las características físicas de la CPU, como el tamaño del bus de direcciones. Así, un procesador con un bus de direcciones de 32 bits puede permitir un tamaño de memoria virtual de hasta 4 GiB, ya que son posibles 2^{32} direcciones distintas de bytes.

Al utilizar memoria virtual, las direcciones virtuales no pasan de forma directa al bus de memoria, sino que van a una unidad de administración de memoria (*Memory Management Unit* o MMU) que asocia las direcciones virtuales con sus correspondientes

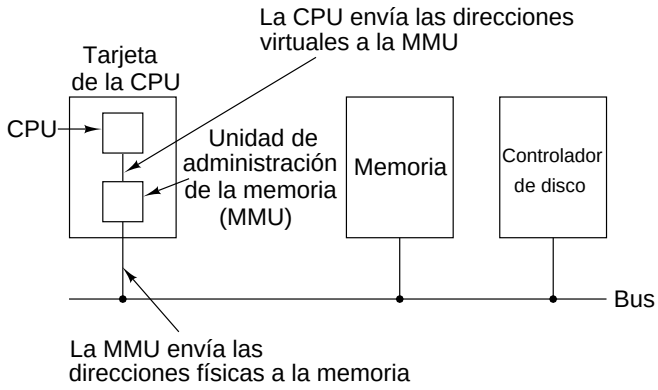


Figura 5.8. La posición y función de la MMU.

direcciones de memoria física (ver figura 5.8). Por lo tanto, las direcciones virtuales generadas por la CPU son *traducidas* en direcciones físicas por la MMU. Como muestra la figura, aunque la MMU es una unidad distinta de la CPU, se suele integrar en el mismo chip.

Para entender el funcionamiento de la traducción de direcciones, pongamos un ejemplo. Supongamos un computador que puede generar direcciones de 16 bits, que van desde 0 hasta 64 K - 1 (es decir, desde 0 hasta 65 535). Estas son las direcciones virtuales, cada una correspondiente a un byte. Supongamos también que hay 32 KiB de memoria física, por lo que serán necesarios 15 bits para especificar una dirección física. Todo esto significa que se pueden ejecutar procesos que usen hasta 64 KiB de memoria virtual (es decir, el código, los datos y la pila de un proceso pueden tener un tamaño conjunto de hasta 64 KiB), de los cuales solo 32 KiB podrán estar en memoria física. Por consiguiente, una copia completa de la memoria del proceso debe estar presente en disco, de forma que se puedan mover a memoria aquellas partes del proceso que se encuentren en disco y que sean necesarias para continuar con la ejecución del proceso.

La paginación divide el espacio de direcciones virtuales en unidades llamadas *páginas* y la memoria física en unidades llamadas *marcos de página*[2]. Las páginas y los marcos tienen siempre el mismo tamaño, normalmente entre 512 bytes y 8 KiB, aunque pueden ser bastante más grandes. En nuestro ejemplo son de 4 KiB, lo que supone que hay 16 páginas y 8 marcos. Las transferencias entre memoria y disco son siempre en unidades de página.

Los datos almacenados en cada página de la memoria virtual (la que ve un proceso) realmente se almacenan en los marcos de página de la memoria física (que es la que en verdad existe). Por ello, cada página debe estar asociada a un marco (como se puede ver en la figura 5.9) para que las direcciones virtuales generadas por el proceso y pertenecientes a una determinada página puedan ser traducidas en direcciones físicas del marco correspondiente. Con las asociaciones entre páginas y marcos mostradas, podemos establecer, por ejemplo, las siguientes correspondencias entre direcciones virtuales y físicas:

Dirección virtual	Dirección física
0	8192
8192	24576
$20500 = 5 \cdot 4096 + 20$	$12308 = 3 \cdot 4096 + 20$

[2]Se llaman marcos porque pueden contener en un momento dado cualquier página virtual, de la misma manera que un marco para fotografías puede contener fotos distintas a lo largo del tiempo. Recordemos que en la asignatura Estructura y Tecnología de Computadores los marcos se llamaban *páginas físicas*.

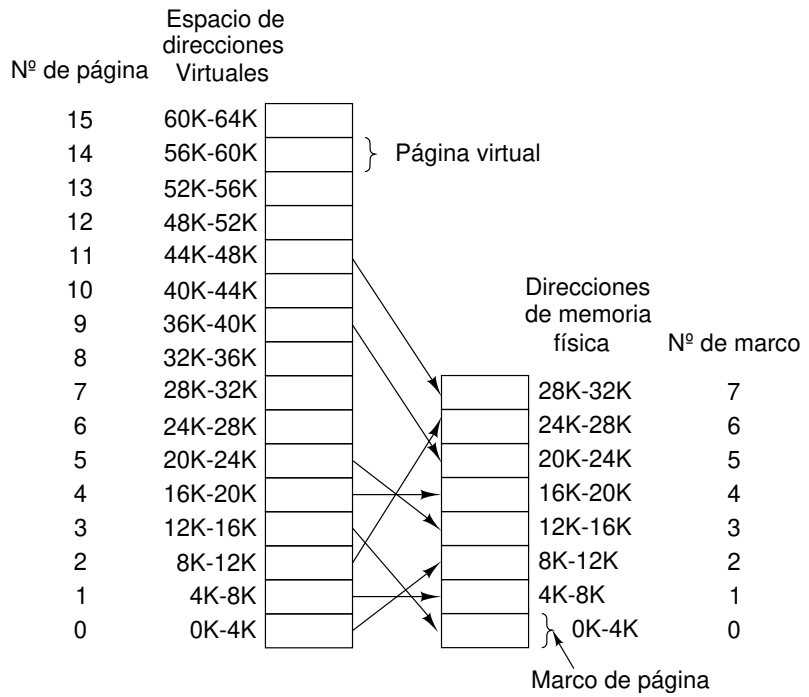


Figura 5.9. La relación entre las direcciones virtuales y las direcciones en la memoria física está dada en la tabla de páginas.

Analicemos más detenidamente la primera línea:

- La dirección virtual 0 se envía a la MMU.
- La MMU ve que esta dirección virtual 0 está dentro de la página 0 (que abarca las direcciones virtuales de la 0 a la 4095), la cual, según la figura, se corresponde con el marco 2 (que abarca las direcciones físicas de la 8192 a la 12287).
- La MMU transforma la dirección virtual 0 (la primera de la página 0) en la dirección física 8192 (la primera del marco 2) que es la que se pone en el bus.

La traducción de la dirección virtual 8192 a la dirección física 24756 funciona igual que la traducción de la primera línea: la primera dirección de la página virtual 2 (que contiene las direcciones virtuales de la 8192 a la 12287) se traduce a la primera dirección del marco 6 (que abarca las direcciones desde la 24576 hasta la 28671).

La traducción de la tercera línea es más interesante. En este caso, la dirección virtual 20500 se encuentra en la posición 20 de la página virtual 5, por lo que debe traducirse a la dirección física que se encuentra en la misma posición en el marco asociado, el 3. El resultado es la dirección física 12308. Observa que si dividimos la dirección virtual 20500 entre el tamaño de página, 4096, obtenemos el número de página como cociente, y la posición 20 dentro de la página como resto de la división.

En la práctica, para saber en qué marco se encuentra cada página se utiliza una *tabla de páginas*, una tabla con tantas entradas como páginas virtuales sean posibles y donde cada entrada indica en qué marco se almacena la página virtual asociada a la entrada.

Todo esto no resuelve el problema de que el espacio de direcciones virtuales es más grande que la memoria física. En nuestro ejemplo, puesto que solo hay ocho marcos, solo 8 de las 16 páginas virtuales de un proceso se asocian con la memoria física. Las demás, marcadas con X, no quedan asociadas. Como veremos después, en el hardware

real un bit presente/ausente[3] en cada entrada de la tabla de páginas nos dice si la página está asociada o no.

Si se intenta acceder a una dirección virtual cuya página no está asociada (por ejemplo, la 32780), la MMU hace que la CPU genere una excepción que debe ser tratada por el sistema operativo. Esta excepción se llama *fallo de página*. El sistema operativo elige entonces un marco de página y escribe su contenido en el disco (si se había escrito algo en el marco) para obtener así un marco libre. Dicho marco será ocupado por la página virtual referida (la página 8 en nuestro caso), se corregirán las asociaciones en la tabla de páginas y se reiniciará la instrucción que produjo el fallo de página.

Por ejemplo, si el sistema operativo decide liberar el marco 2 y carga la página virtual 8 a partir de la dirección física 8192 (marco 2), en la entrada de la página virtual 0 pondrá una X para indicar que ya no está asociada a ningún marco y en la página virtual 8 pondrá un 2.

En general, los pasos para resolver un fallo de página se pueden resumir en la figura 5.10:

1. Durante la ejecución de un proceso, la CPU genera una dirección virtual cuya página no se encuentra en memoria, como indica la información de la tabla de páginas.

2. Se produce un fallo de página. La trampa (o excepción) provoca un cambio de modo usuario a modo núcleo y el sistema operativo toma el control.

3. El sistema operativo localiza el contenido de la página que nos interesa en disco.

4. El sistema operativo carga el contenido de la página en un marco libre; si no existe ninguno, tendrá que liberar uno aplicando alguno de los algoritmos que veremos después.

5. La información de la tabla de páginas se debe actualizar para indicar que la página ya tiene un marco asignado.

6. Tras resolver el fallo de página, se debe reiniciar la instrucción que lo produjo. Esta vez ya no se producirá fallo al encontrarse la página en memoria.

Antes hemos visto que para saber en qué página se encuentra una dirección virtual, podemos hacer la división entera entre la dirección y el tamaño de página, y que mediante divisiones, sumas y multiplicaciones podemos traducir cualquier dirección virtual en su correspondiente dirección física. El problema de traducir direcciones mediante operaciones aritméticas es que cada traducción sería muy costosa en cuanto al tiempo necesario para llevarla a cabo. Teniendo en cuenta que hay que traducir todas las direcciones generadas por la CPU, es evidente que este tiempo no sería despreciable durante la ejecución de un proceso.

La solución pasa por evitar cualquier tipo de operación aritmética en las traducciones y para ello es necesario que el tamaño de las páginas sea potencia de 2 para que las divisiones enteras y los módulos se conviertan en simples selecciones de bits. La figura 5.11 muestra el funcionamiento interno de una MMU a la hora de hacer una traducción. Ahora, en lugar de hacer operaciones aritméticas, una dirección virtual en binario se divide en dos campos: un primer campo de 4 bits llamado *número de página* y un segundo campo de 12 bits llamado *ajuste* o *desplazamiento*. Con 4 bits para el primer campo podemos representar 16 páginas y con 12 bits para el ajuste podemos indicar la dirección

[3] Llamado *bit de validez* en Estructura y Tecnología de Computadores.

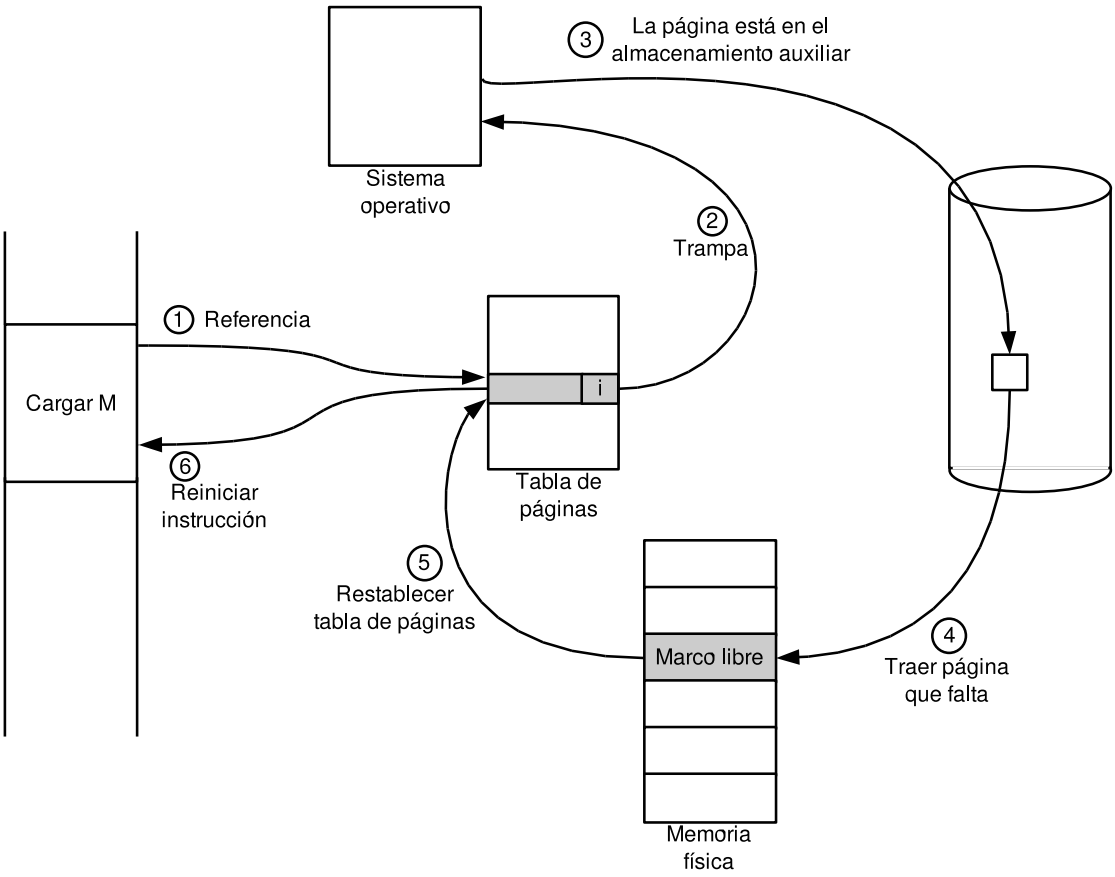


Figura 5.10. Pasos para resolver un fallo de página.

de uno de los 4096 bytes (4 KiB) de una página. Vemos, pues, que el tamaño del segundo campo viene determinado por el tamaño de página. Observa que $12 = \log_2 4096$ (o que $2^{12} = 4096$) y que el descomponer una dirección virtual de esta manera equivale a hacer una división entera por 4096, donde los 12 bits menos significativos son el módulo y los 4 bits más significativos son el cociente de la división.

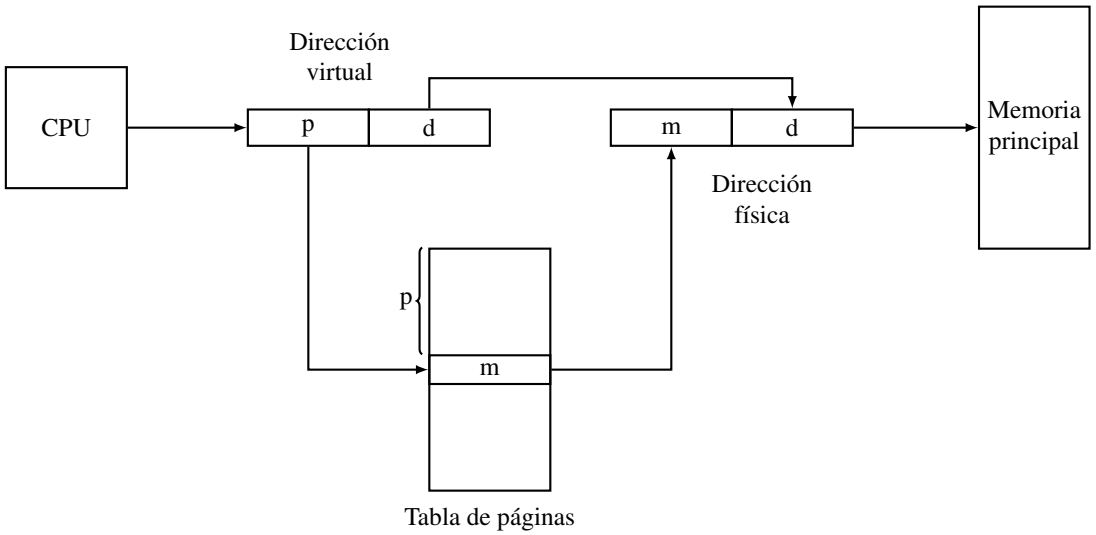
Como muestra la misma figura 5.11, el número de página se utiliza como índice en la tabla de páginas para seleccionar una de sus entradas, que contiene el número de marco correspondiente a esa página virtual. Si el bit presente/ausente es 0, se provoca una excepción hacia el sistema operativo. Si el bit es 1, el número de marco que aparece en la tabla de páginas se copia en los bits más significativos del registro de salida junto con el ajuste de 12 bits, que no sufre modificaciones. Juntos forman una dirección física de 15 bits. El contenido del registro de salida se coloca entonces en el bus de direcciones.

5.3.2 Tabla de páginas

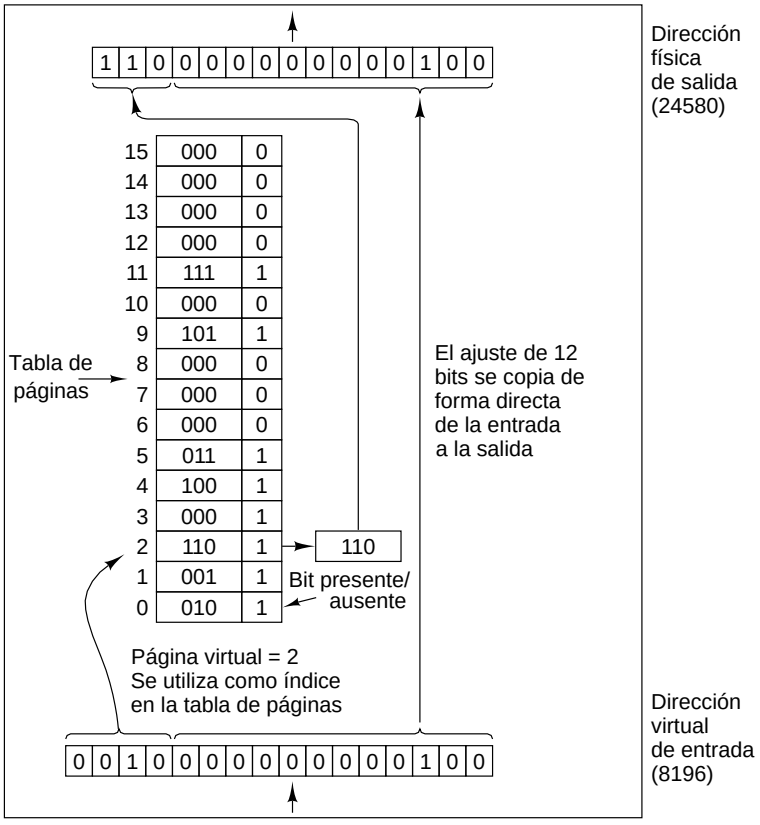
Acabamos de ver que la finalidad de la tabla de páginas es asociar las páginas virtuales con los marcos. En la implementación de dicha tabla hay dos aspectos fundamentales a tener en cuenta:

- La tabla de páginas puede ser muy grande.
- La traducción de direcciones debe ser rápida.

El primer punto se debe a que los ordenadores modernos utilizan direcciones virtuales de, al menos, 32 bits. Así, si el tamaño de página es de 4 KiB, por ejemplo, el espacio



(a)



(b)

Figura 5.11. (a) Funcionamiento lógico de la traducción de direcciones. (b) Ejemplo de operación interna de la MMU con 16 páginas de 4 KiB.

de direcciones virtuales se divide en 1 048 576 páginas, por lo que la tabla de páginas debe tener también 1 048 576 entradas. Además, debemos tener en cuenta que cada proceso necesita su propia tabla de páginas para tener su propio espacio de direcciones virtuales, independiente de los espacios de direcciones virtuales del resto de procesos. Con espacios de direcciones separados, el sistema operativo está protegido de los procesos y estos entre sí, ya que un proceso solo accederá a las zonas de memoria principal que se correspondan con sus páginas. En otras palabras, en paginación, el hecho de que cada proceso tenga su propio espacio de direcciones virtuales y su propia tabla de páginas protege al sistema operativo y al resto de procesos. Sin embargo, la existencia de una tabla por proceso agrava el problema del tamaño de la tabla de páginas.

El segundo punto es consecuencia del hecho de que la traducción de direcciones debe hacerse en cada referencia a memoria. Si una instrucción tiene operandos de memoria además del propio código de la instrucción, entonces puede ser necesario hacer varias referencias a memoria para ejecutar una única instrucción, por lo que también serán necesarias varias referencias a la tabla de páginas por instrucción para hacer las traducciones de las direcciones usadas.

El diseño más sencillo es tener una sola tabla de páginas consistente en un array de registros rápidos en hardware con una entrada por cada página virtual. Al asignar la CPU a un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tabla que se encontrará en la memoria principal. A partir de aquí, la tabla no provoca más referencias a memoria al hacer la asociación. Este diseño tiene como ventaja que las traducciones no suponen referencias a memoria, pero tiene como inconvenientes que es costoso en recursos y que cargar la tabla en cada cambio de proceso puede consumir mucho tiempo y, por ello, reducir el rendimiento.

El otro extremo es tener la tabla de páginas totalmente dentro de la memoria principal. En este caso, todo lo que necesita el hardware es un solo registro, r, que apunte al inicio de la tabla de páginas del proceso en ejecución. Los datos de la entrada de la página p se encontrarán en la dirección física de memoria $r + p \cdot t$, siendo t el tamaño de una entrada de la tabla de páginas. La ventaja de este diseño es que es posible cambiar el mapa de memoria en un cambio de proceso cargando un único registro (el r). Sin embargo, su gran inconveniente es que, además de los accesos a memoria necesarios para ejecutar una instrucción, también se necesitan una o más referencias a memoria para leer las entradas de la tabla de páginas que permiten las traducciones de las direcciones virtuales producidas por la instrucción para realizar esos accesos, lo cual puede degradar seriamente el rendimiento de la memoria. Por eso, este esquema no se usa tal cual.

A continuación vamos a ver dos soluciones intermedias a estos dos puntos extremos que tratan de solventar los inconvenientes mencionados. Estas soluciones son las *tablas de páginas de varios niveles* y el *TLB*. También veremos una tercera solución, las *tablas de páginas invertidas*, que permiten reducir considerablemente el espacio ocupado en memoria RAM por las tablas de páginas de los procesos.

Tabla de páginas de varios niveles

Para evitar el problema de tener unas tablas inmensas en la memoria todo el tiempo, muchos ordenadores utilizan una tabla de páginas con varios niveles. Para ello, es necesario dividir el número de página virtual en 2 o más partes. Por ejemplo, si tenemos una dirección virtual de 32 bits, con un número de página virtual de 20 bits y un ajuste de 12 bits, podemos dividir el número de página virtual en dos campos, PT1 y PT2, de 10 bits cada uno (ver figura 5.12(a)). Observa que con 20 bits para el número de página tenemos $2^{20} = 1\,048\,576$ páginas virtuales, que en este caso serán de 4 KiB cada una

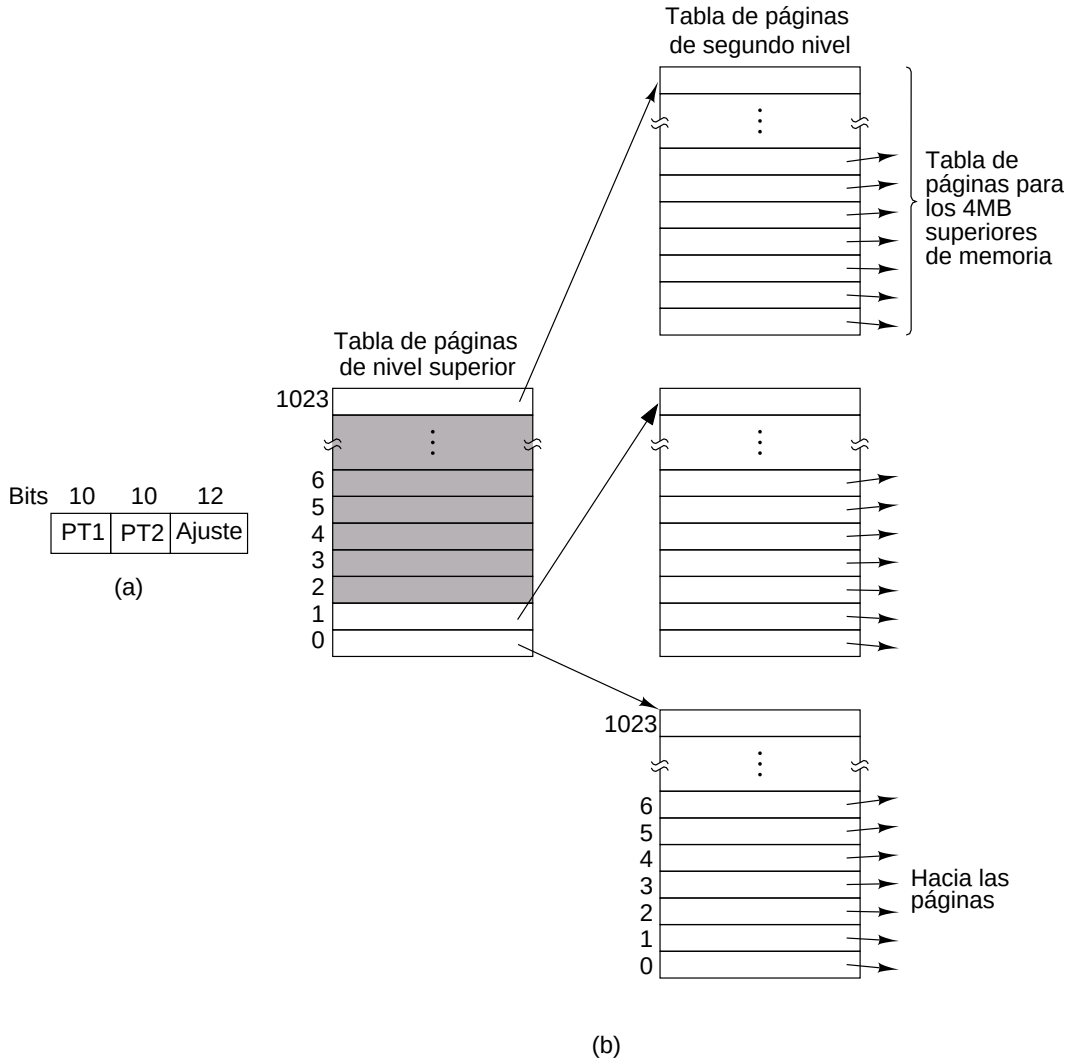


Figura 5.12. (a) Una dirección de 32 bits con dos campos para la tabla de páginas. (b) Tablas de páginas de dos niveles.

por ser el ajuste de 12 bits.

Esta división en dos del número de página da lugar a una tabla de páginas de 2 niveles. El valor de PT1 se utiliza para buscar en el primer nivel y el valor de PT2 para buscar en el segundo. En la figura 5.12(b) tenemos un ejemplo de 2 niveles. Supongamos un proceso que necesita 12 MiB: 4 MiB para el código, 4 MiB para datos y los últimos 4 MiB para la pila. Cada una de las 1024 entradas de la tabla de páginas del nivel superior (por los 10 bits de PT1) representa 4 MiB del espacio de direcciones virtuales, ya que el tamaño total de este espacio es de 4 GiB y la tabla de primer nivel divide dicho espacio en 1024 partes iguales. Cada una de las 1024 entradas de una de las tablas de páginas del segundo nivel (por los 10 bits de PT2) representa a una página de 4 KiB; esto es así por los 12 bits del ajuste, pero también porque una tabla de segundo nivel divide los 4 MiB representados por una entrada de primer nivel en 1024 partes, cada una correspondiente a una página.

El secreto de este método consiste en no tener al mismo tiempo todas las tablas en la memoria; las que no son necesarias, no se tienen. Como se puede observar en la figura, aunque el espacio de direcciones tiene 1 048 576 páginas, solo se necesitan en realidad 4 tablas de páginas de 1024 entradas cada una (en total, 4096 entradas): la tabla de nivel superior y las tablas secundarias de 0 a 4 MiB, para las páginas que contienen código,

de 4 MiB a 8 MiB, para las páginas de datos, y los 4 MiB superiores, para las páginas de la pila.

El funcionamiento de la MMU en este caso es:

- Se toman los 10 bits de PT1 y se busca en la tabla de primer nivel la entrada correspondiente.
- Si la entrada es válida, entonces contiene la dirección de memoria física de una tabla de nivel secundario. Si las tablas del segundo nivel tienen el tamaño de un marco, entonces en lugar de la dirección de memoria se puede guardar el número del marco que contiene la tabla.
- Se toman los 10 bits de PT2 y se busca la entrada correspondiente en la tabla de segundo nivel localizada en el paso anterior.
- Si esta última entrada es válida, entonces contiene el número de marco asociado a la página en la que se encuentra la dirección virtual a la que se quiere acceder.

Por ejemplo, supongamos que tenemos que traducir la dirección virtual de 32 bits 0x00403004 (o 0000 0000 0100 0000 0011 0000 0000 0100 en binario). El número de página de esta dirección nos lo dan los 20 bits más significativos, es decir, 0x00403. La traducción se haría de la siguiente manera:

- PT1 = 0000 0000 01 = 1. Se accede a la entrada 1 de la tabla de páginas del primer nivel, que corresponde a las direcciones virtuales entre 4 MiB y 8 MiB. De esa entrada obtenemos la dirección de una tabla del segundo nivel.
- PT2 = 00 0000 0011 = 3. Se accede a la entrada 3 de la tabla del segundo nivel localizada en el paso anterior. Esta entrada se corresponde con las posiciones entre 12 292 y 16 383 dentro de su bloque de 4 MiB, es decir, direcciones virtuales absolutas entre 4 206 592 (= 4 M + 12 K) y 4 210 687 (= 4 M + 16 K - 1). De esta entrada extraemos el número de marco de la página buscada (la 0x00403 en hexadecimal o 1027 en decimal).
- Si el bit presente/ausente en la entrada de la tabla de segundo nivel es 0, se provoca un fallo de página. Si no es así, se toma el número de marco que se une al ajuste para formar la dirección física, que se coloca en el bus y se envía a la memoria.

Esta estructura de niveles se puede aumentar a 3 niveles (como ocurrió en el procesador Intel Pentium Pro), 4 niveles (como ocurre actualmente en los procesadores de 64 bits de Intel y AMD) o más. De hecho, a partir de la versión 4.14, el núcleo de Linux implementa tablas de páginas de hasta 5 niveles para gestionar hasta 128 PiB de memoria virtual y hasta 4 PiB de memoria RAM en aquellos sistemas que lo requieran. Cuantos más niveles, mayor flexibilidad, pero también, mayor complejidad.

Puesto que el objetivo de una tabla de páginas de varios niveles es reducir la cantidad de memoria necesaria, si con el transcurso del tiempo hay tablas de páginas del nivel más bajo que se quedan sin información útil, estas tablas se pueden eliminar de memoria para liberar el espacio que ocupan. Evidentemente, si esto supone que una tabla del nivel superior también se quede vacía, se puede eliminar de la misma manera y así sucesivamente.

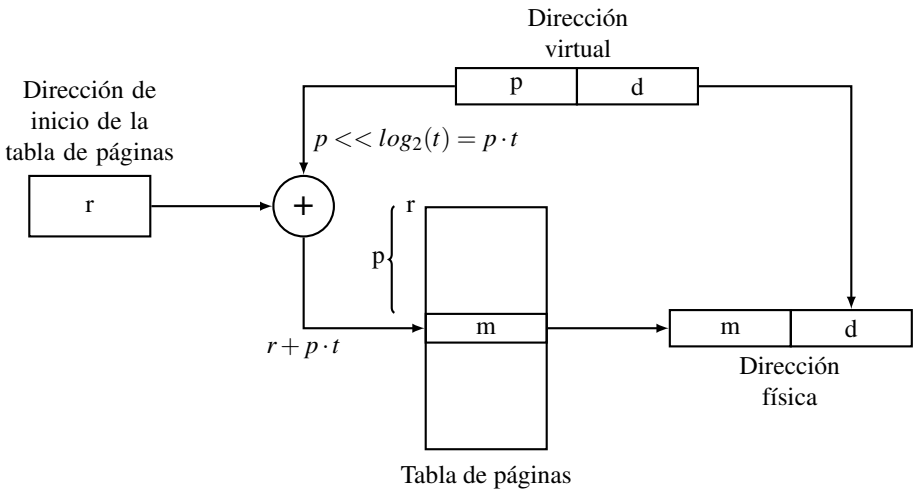


Figura 5.13. Traducción de direcciones en paginación por correspondencia directa.

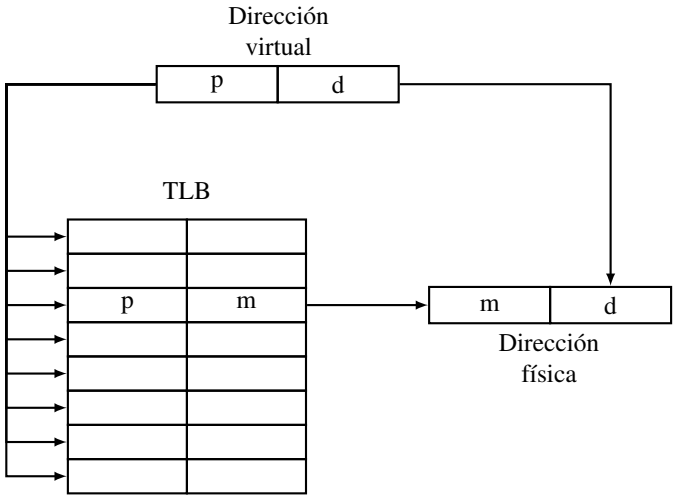


Figura 5.14. Traducción por correspondencia asociativa pura (TLB).

TLB

Hasta ahora, hemos supuesto que las tablas de páginas se mantienen en la memoria debido a su gran tamaño (ver figura 5.13). Esto hace que usar la paginación suponga hacer más referencias a memoria que cuando no se utiliza, por lo que el rendimiento de la jerarquía de memoria baja.

La solución a este problema es equipar al ordenador con un pequeño dispositivo hardware para traducir las direcciones virtuales a direcciones físicas sin tener que ir a la tabla de páginas en memoria en cada traducción. Este dispositivo se llama TLB (*Translation Look-aside Buffer*) y es una pequeña memoria totalmente asociativa, generalmente dentro de la MMU, que contiene entradas de la tabla de páginas. La figura 5.14 muestra el funcionamiento de la traducción de direcciones usando un TLB.

La figura 5.15 muestra una posible estructura de las entradas de un TLB, donde cada una tiene:

- Un bit de validez, que nos indica si la entrada del TLB tiene información útil o no.
- El número de página para la que la entrada del TLB contiene información.

Validez	Página	Bit de modificación	Protección	Marco
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 5.15. Un TLB para acelerar la paginación con un bit de validez por entrada.

- Un bit de modificación que indica si el contenido de la página ha sido modificado, es decir, si se han realizado operaciones de escritura sobre la página.
- Diversos bits de protección, que indican qué tipo de acceso podemos realizar sobre la página.
- El número de marco en el que se encuentra la página en memoria.

Lo ideal sería tener toda la tabla de páginas en el TLB, por lo que las traducciones de direcciones virtuales a físicas no supondrían ningún acceso a la memoria principal. El problema está en que el número de entradas en el TLB suele ser muy pequeño (casi nunca más de 128), ya que un TLB grande sería muy caro.

La solución es adoptar una solución intermedia, es decir, tener la tabla de páginas en memoria principal y almacenar en el TLB copias de aquellas entradas de la tabla de páginas que más se utilizan. El funcionamiento ahora se puede ver en la figura 5.16 y es el siguiente: cuando la MMU recibe una dirección virtual para su traducción, el hardware verifica en primer lugar si su número de página virtual se encuentra en el TLB, comparando todas las entradas al mismo tiempo, es decir, en paralelo. Si coincide con alguno y el acceso no viola los bits de protección, el marco de página se toma del TLB. Si coincide, pero viola los bits de protección, entonces se producirá una violación de acceso que provocará la terminación del proceso. Obsérvese que solo se puede producir un acierto de TLB si la página correspondiente está en memoria, ya que al TLB solo se copian entradas de páginas que están en memoria, es decir, que no producen fallos de página.

Si el número de página virtual no está en el TLB, la MMU detecta la falta y hace una búsqueda normal en la tabla de páginas. Elimina entonces una entrada del TLB y la reemplaza con la entrada leída de la tabla de páginas. Al eliminar una entrada del TLB, el bit de modificación de la página en el TLB (ver figura 5.15) se copia de regreso en la entrada de la tabla de páginas en la memoria. Si la página no está en memoria, entonces se producirá un fallo de página que habrá que resolver antes de tratar el fallo de TLB.

Un aspecto relacionado con el TLB surge cuando se tienen varios procesos en ejecución. Como hemos dicho, cada proceso tiene su tabla de páginas para hacer su propia traducción. Durante la ejecución de un proceso, el TLB contendrá entradas de la tabla de páginas de dicho proceso para que la MMU haga las traducciones correctas. Al cambiar de proceso, la tabla de páginas que se debe usar es la del nuevo proceso, por lo que la MMU no debe utilizar las entradas del TLB que contienen información del proceso anterior. Para solucionar este problema, dos posibles soluciones son:

- Invalidar el contenido del TLB borrando los bits de validez de todas las entradas mediante una instrucción especial del hardware (ver figura 5.15).

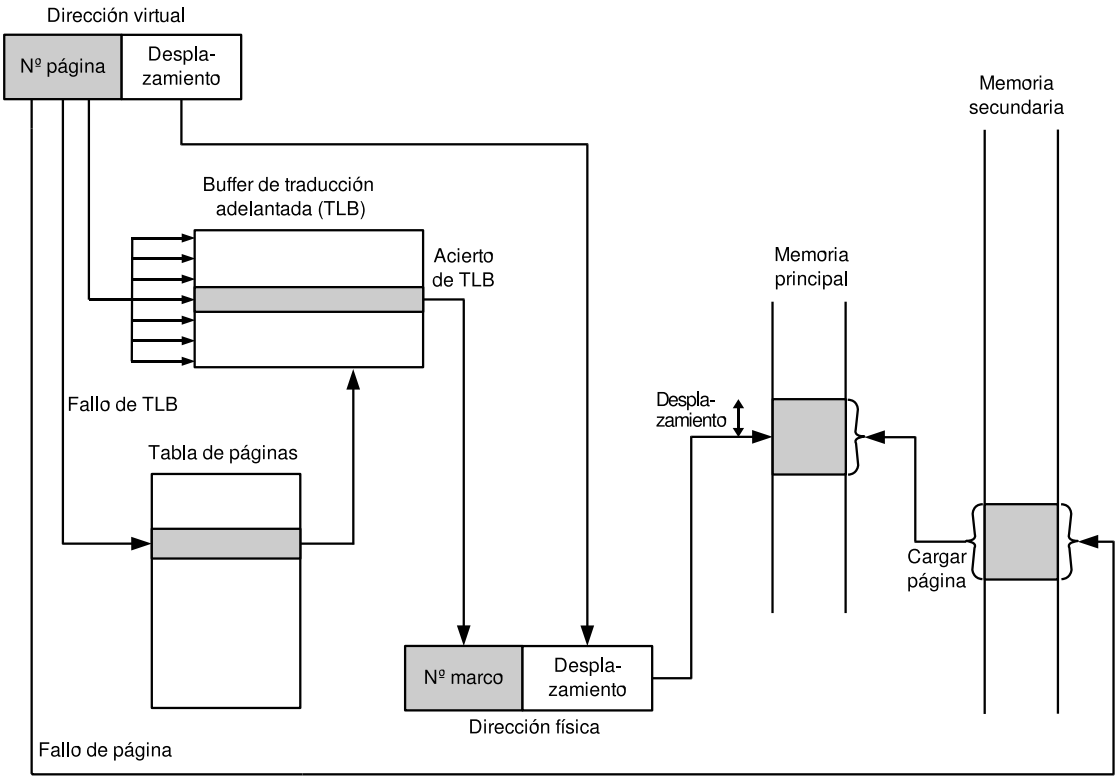


Figura 5.16. Traducción de direcciones en paginación por correspondencia asociativa/-directa.

- Añadir un nuevo campo a cada entrada del TLB con la identificación del proceso al que pertenece (por ejemplo, su PID), y añadir también un registro hardware con el identificador del proceso activo (ver figura 5.17). Se ignoran entonces las entradas de los procesos distintos al proceso activo. Aunque esta solución requiere hardware adicional, ahorra tiempo en los cambios de proceso. Además, con un poco de suerte, cuando un proceso vuelva a coger la CPU, encontrará todavía en el TLB algunas de las entradas que usó la última vez que estuvo en la CPU, por lo que el número de fallos de TLB se reducirá.

Tabla de páginas invertida

En muchos procesadores actuales, que poseen arquitecturas de 64 bits, las tablas de páginas pueden ser enormes. Así, por ejemplo, con 64 bit para una dirección virtual y páginas de 4 KiB, la tabla de páginas de cada proceso tendría $\frac{2^{64}}{2^{12}} = 2^{52}$ entradas. Si, además, el tamaño de cada entrada fuera 8 bytes (es decir, el tamaño de palabra), cada tabla de páginas ocuparía 32 PiB de memoria. Para solucionar este problema, podríamos pensar en usar páginas más grandes, pero el tamaño debería ser considerable para reducir el número de entradas de las tablas de páginas, y eso haría que se perdiera mucha memoria por fragmentación interna. Podríamos pensar también en usar tablas de páginas multinivel, pero el número de niveles necesarios sería elevado, lo que haría la gestión de estas tablas demasiado compleja (ya hemos comentado que las versiones actuales del núcleo de Linux admiten tablas de páginas de hasta 5 niveles).

Existen, sin embargo, unas tablas de páginas que nos permite usar de forma eficiente direcciones virtuales de 64 bits y páginas pequeñas. Estas tablas se llaman *tablas de*

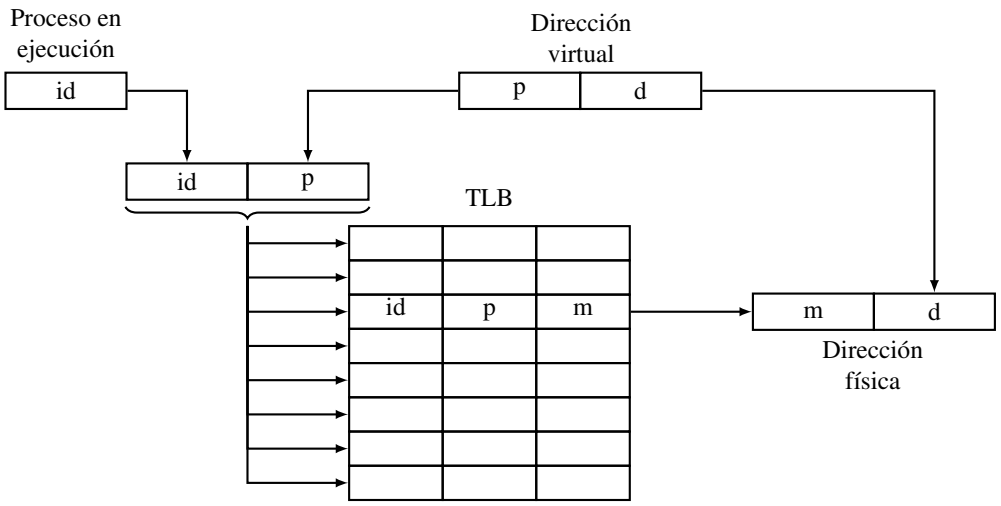


Figura 5.17. TLB donde cada entrada contiene la identificación del proceso al que pertenece. Se producirá un acierto de TLB si existe una entrada para la página *p* que pertenezca al proceso *id*.

páginas invertidas, y se utilizan en los procesadores PowerPC y UltraSPARC de 64 bits (también se utilizaron en el procesador Itanium de Intel).

En una tabla de páginas invertida, existe una *tabla de traducción* donde hay una entrada por cada marco de página de la memoria física, por lo que su tamaño solo depende de la cantidad que tengamos de esta memoria (ver figura 5.18). Por ejemplo, si tenemos 512 MiB de RAM y páginas de 4 KiB, entonces la tabla de traducción de una tabla de páginas invertida solo necesitará 131 072 entradas. Además, esta tabla de traducción es única, por lo que todos los procesos la comparten.

Cada entrada de la tabla de traducción contiene información sobre la página virtual que se encuentra en un marco de memoria. En concreto, almacena el PID del proceso al que pertenece la página virtual, el número de dicha página virtual y el marco de memoria donde se almacena la página. Con esta información, cuando hay que traducir una dirección virtual generada por un proceso, hay que buscar en toda la tabla si hay una entrada cuyo PID sea el del proceso y cuyo número de página virtual sea el correspondiente a la dirección generada. Si la entrada existe, entonces se toma de ella el número de marco y se termina la traducción. En caso contrario, se produce un fallo de página. Obsérvese que, aunque la tabla de traducción tiene tantas entradas como marcos, no hay una asociación entre entradas y marcos preestablecida en el sentido de que la entrada N de la tabla de páginas puede contener información sobre cualquier marco, no sobre el marco N.

Como la búsqueda en toda la tabla haría muy lentas las traducciones, para mejorar el rendimiento se usa una *tabla de dispersión* y un TLB. Como se puede ver en la figura 5.18, existe una función de dispersión que recibe como datos de entrada el identificador del proceso para el que se quiere hacer la traducción y el número de la página virtual de dicho proceso para la que se desea obtener el número de marco correspondiente. Como salida, la función devuelve una entrada de la tabla de dispersión la cual contiene la posible entrada de la tabla de páginas asociada a ese proceso y a esa página del proceso. Puesto que se pueden dar colisiones, existe un campo en cada entrada de la tabla de traducción que permite crear una lista con las entradas a las que les corresponde el mismo valor de dispersión. Gracias a la tabla de dispersión, un fallo de TLB no supone recorrer toda la tabla, solo la lista de entradas correspondientes a un cierto valor

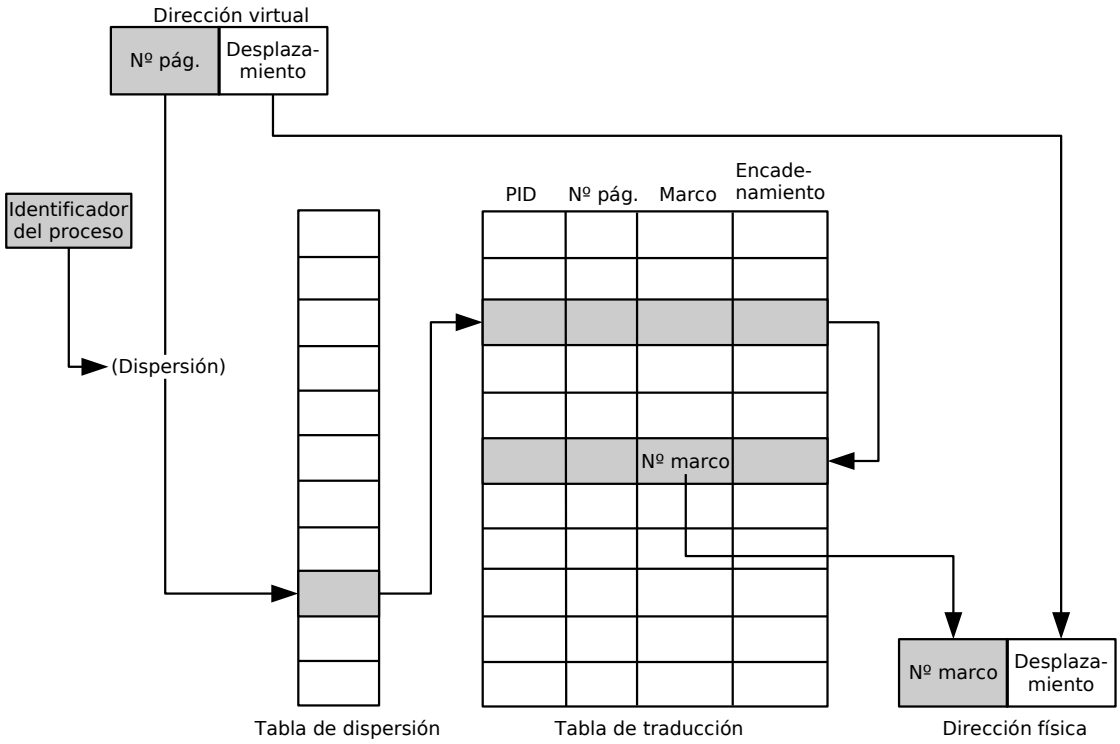


Figura 5.18. Tabla de páginas invertida.

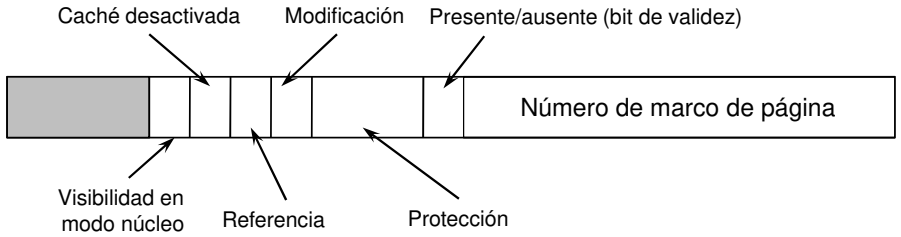


Figura 5.19. Una entrada usual en una tabla de páginas.

de dispersión.

El TLB, por su parte, funciona como antes, es decir, contiene las entradas de la tabla de traducción usadas recientemente. Ahora, dados el identificador de un proceso y un número de página, el TLB devuelve rápidamente el número de marco en el que se encuentra la página, sin necesidad de acceder a la tabla en memoria (salvo que se produzca un fallo de TLB, como hemos dicho).

Hemos dicho que, en una tabla de páginas invertida, hay al menos una entrada en la tabla de traducción por cada marco de memoria. Sin embargo, como veremos en la sección 5.3.3, los procesos pueden compartir marcos. Por ejemplo, un proceso padre puede compartir con sus procesos hijos los marcos que contienen el código del proceso. Esto hace que para un mismo marco puedan existir dos o más entradas asociadas en la tabla de traducción de una tabla de páginas invertida, por lo que la tabla de traducción habitualmente tendrá más entradas que marcos. El número de marcos en memoria principal nos dará, por tanto, el número mínimo de entradas en la tabla de traducción.

Estructura de una entrada

Para finalizar este apartado, vamos a analizar más detenidamente la estructura de una entrada de la tabla de páginas. En el caso de las tablas de páginas de correspondencia

directa (es decir, no invertidas) una posible estructura la encontramos en la figura 5.19. Esta estructura puede variar de un sistema a otro. Como vemos, además de los campos habituales (número de marco de página y bit presente/ausente) encontramos otros como:

- Los *bits de protección*: indican el tipo de acceso permitido a la página (lectura, escritura, ejecución, etc.). Así, las páginas que contienen código suelen tener permisos de lectura y ejecución, pero no de escritura, mientras que las páginas de datos suelen tener permisos de lectura y escritura, pero no de ejecución.

- El *bit de modificación*: indica si se ha modificado (1) o no (0) el contenido de una página (en realidad, del marco asociado). Si se ha modificado, y el sistema operativo decide asignar a otra página un marco modificado, la página almacenada en dicho marco debe guardarse en disco para que las modificaciones no se pierdan.

- El *bit de referencia*[4]: toma valor 1 cuando se hace referencia a una página, ya sea para leerla o para escribirla. Este bit se utiliza en varios algoritmos de reemplazo de páginas que veremos después para ayudar a decidir que página quitar de memoria cuando se necesite hueco para una nueva página. Las páginas no utilizadas suelen ser buenas candidatas.

- El *bit de caching desactivado*: se utiliza en las máquinas con E/S mapeada por memoria para evitar que el contenido de la página se almacene en la memoria caché (la que se encuentra entre la memoria principal y el procesador).

- El *bit de visibilidad en modo núcleo*: tiene que ver con el mapa de memoria de un proceso. Hablaremos del uso de este bit en la sección 5.3.3.

En el caso de las tablas de páginas invertidas, las entradas tienen básicamente la misma estructura, aunque en ellas aparecen algunos campos más, como el identificador del proceso al que pertenece la entrada, el número de página asociado y un puntero para el encadenamiento de entradas asociadas a una misma entrada de la tabla de dispersión.

5.3.3 Mapa de memoria de un proceso

Para poder ejecutarse, un proceso necesita tener en memoria tres elementos fundamentales: su código, una zona de datos y una pila. Inicialmente, el código y los datos con valor inicial, como pueden ser las variables globales a las que se les asigna un valor en el código fuente del programa, se obtienen desde el fichero ejecutable, mientras que la pila y la zona de datos sin valor inicial (por ejemplo, variables globales a las que no se les ha dado un valor en el código fuente) se crean cuando el proceso también se crea (ver figura 5.20). Generalmente, el código y los datos se encuentran al principio del espacio de direcciones de un proceso, mientras que la pila se encuentra al final. A la forma en la que se estructura el espacio de direcciones lógicas o virtuales de un proceso se le llama *mapa de memoria* del proceso.

En los sistemas sin memoria virtual, como los descritos al principio de este tema, el mapa de memoria de un proceso suele ser sencillo: al proceso se le asigna una zona contigua de memoria física y en ella se colocan su código, datos y pila de la forma que hemos comentado (código y datos al principio del espacio de direcciones, pila al final).

En un sistema con paginación, en cambio, el mapa de memoria puede llegar a ser bastante más complejo, ya que el espacio de direcciones virtuales de un proceso se divide en páginas y no todas ellas tienen por qué estar en memoria. Es más, puede

[4] En Estructura y Tecnología de Computadores se llamaba «de uso».

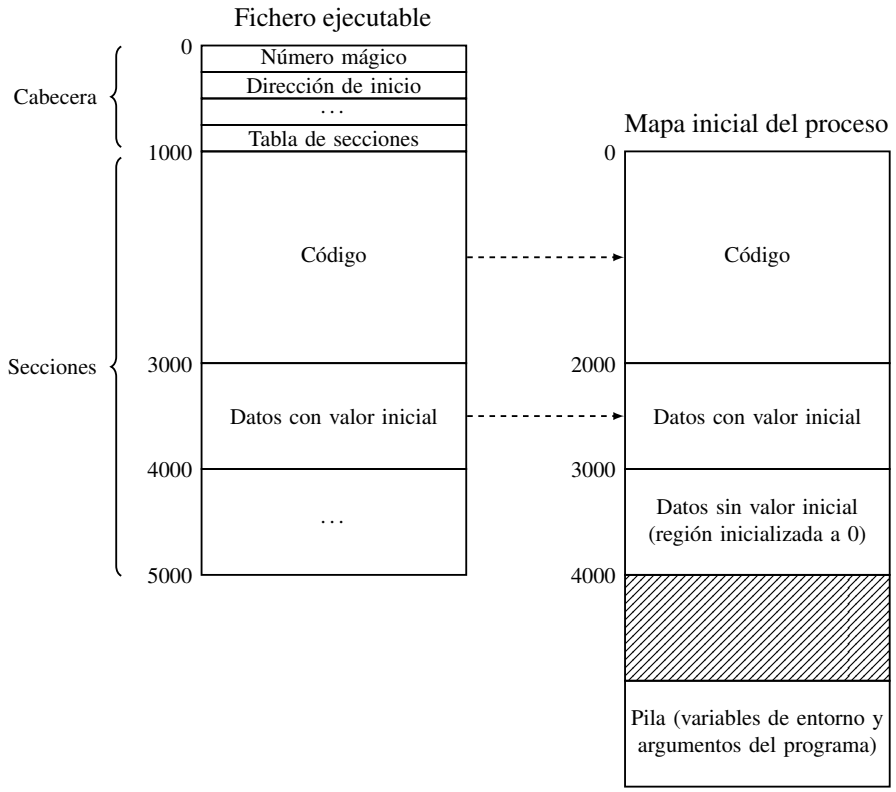


Figura 5.20. Mapa de memoria inicial de un proceso generado a partir de su ejecutable.

haber amplias zonas de memoria virtual que estén vacías y para las que cualquier acceso produzca una violación de acceso al no existir la página correspondiente a la dirección a la que se accede; observa que no se produce un fallo de página pues, al no existir la página, no se puede obtener de ningún sitio. Por lo tanto, en paginación, el mapa de memoria debe controlar también qué zonas del espacio de direcciones virtuales están ocupadas y cuáles no. A las zonas contiguas de memoria virtual ocupadas, dedicadas normalmente a un mismo propósito, las llamamos *regiones*.

Según lo que hemos comentado antes, inicialmente habrá una región dedicada al código y otra a los datos con valor inicial que se tomarán directamente del ejecutable. También se crearán una región para los datos sin valor inicial y otra para la pila.

Cada región de memoria tiene asociadas una serie de propiedades y características:

- Soporte: describe de dónde se obtienen los datos que contiene la región. Para muchas regiones, el soporte es un fichero o parte del mismo. Por ejemplo, las páginas del código se encuentran en el propio ejecutable y se obtienen de él cuando se produce un fallo de página. También hay regiones sin soporte porque no tienen un contenido inicial, como las páginas de la pila. Estas páginas, si se modifican y son expulsadas, se guardarán en la zona de intercambio de disco[5].

- Tipo de compartición: indica si las modificaciones en una región se comparten con otros procesos o no. Puede ser privada, si las modificaciones solo son visibles para el proceso que hace las modificaciones, o compartida, si las modificaciones que hace un proceso son visibles por otros.

[5]En paginación, la zona o área de intercambio (o *swap*) es aquella parte del disco donde se almacenan las páginas que han sido expulsadas de memoria. Puede ser una partición, pero también un fichero,

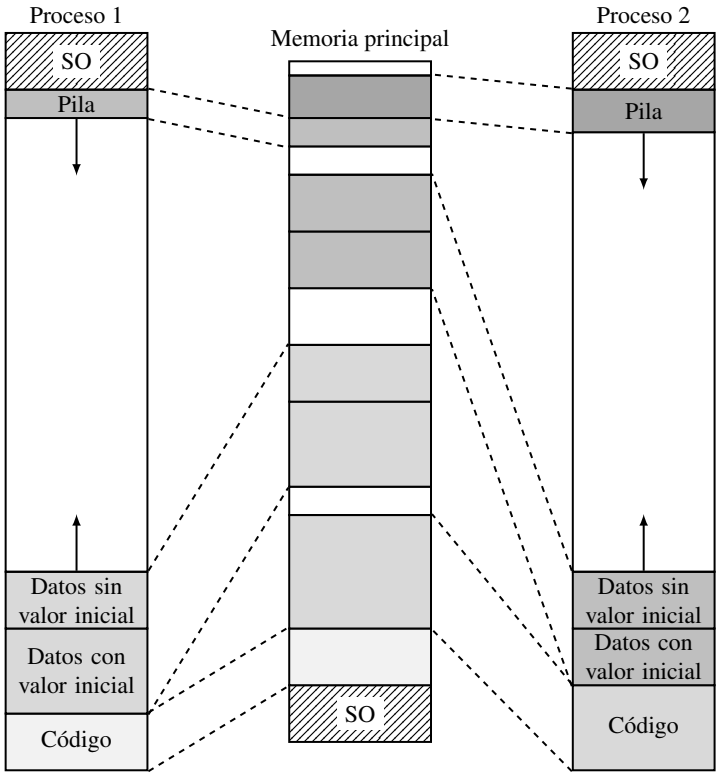


Figura 5.21. Mapas de memoria de dos procesos y correspondencia con la memoria principal en un sistema con paginación.

- Protección: nos dice si el contenido de una página se puede leer, escribir o ejecutar.
- Tamaño fijo o variable: indica si el tamaño de una región puede cambiar o no. Si puede cambiar, entonces se especifica si crece hacia direcciones de memoria mayores, como la memoria montón (hablaremos de ella ahora), o menores, como la pila.

Además de las regiones vistas (código, datos y pila), los procesos suelen usar otras regiones. Por ejemplo, una región común es la dedicada a la memoria montón o *heap*. Esta memoria da soporte a las reservas dinámicas de memoria realizadas por los lenguajes de programación; comienza típicamente tras la región de datos sin valor inicial y crece hacia direcciones superiores. No tiene soporte asociado (está inicialmente a cero) y va creciendo conforme el proceso necesita memoria (el proceso le pide al sistema operativo que aumente el tamaño de esta región mediante las llamadas al sistema `brk()` y `sbrk()`). Otro ejemplo son los procesos con hilos, en los que se crea una región para la pila de cada hilo.

La figura 5.21 muestra los mapas de memoria de dos procesos y cómo las regiones de los mismos se corresponden con zonas de memoria. Aunque la figura muestra que cada región ocupa una serie de marcos contiguos de memoria física, realmente, no tiene por qué ser así, ya que cada región ocupa varias páginas y estas pueden estar almacenadas en cualquier marco. Es más, algunas de estas páginas podrían no estar en memoria, por lo que se produciría un fallo de página al acceder a ellas.

Como se observa en la figura, el sistema operativo aparece en la parte alta del espacio de direcciones virtuales de un proceso. El que el sistema operativo forme parte del mapa de memoria tiene diversas ventajas. Por ejemplo, hay llamadas al sistema, como las

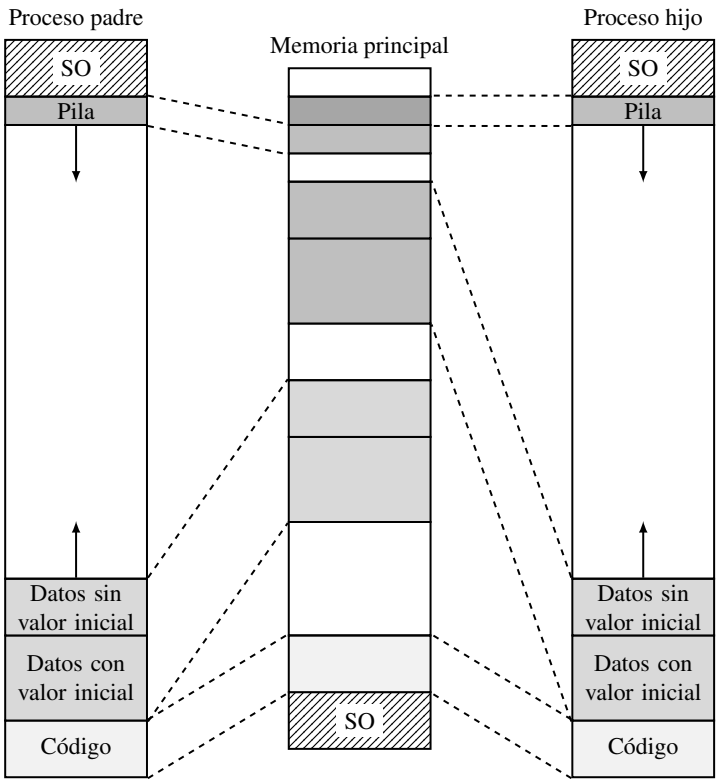


Figura 5.22. Mapas de memoria de un proceso padre y su hijo, inmediatamente después del `fork()`. En este caso, padre e hijo comparten el código.

que nos permiten leer o escribir en un fichero, que requieren copiar información del sistema operativo al proceso o viceversa. Con esta organización, la transferencia de datos se hace con una simple copia de bytes de una zona de memoria a otra. Para evitar que los procesos puedan leer o escribir en el sistema operativo, este solo es visible en modo núcleo (recordemos el bit «visible en modo núcleo» descrito en la figura 5.19). En modo usuario, la región de memoria ocupada por el sistema operativo es inválida y no puede ser accedida. Cualquier intento de lectura o escritura en esta región produciría una violación de acceso.

La organización del mapa de memoria de un proceso en regiones y el uso de la paginación permiten implementar eficientemente la llamada al sistema `fork()`. Ahora, cuando se crea un proceso hijo, este recibe una copia de la tabla de páginas del padre, por lo que las páginas del hijo apuntarán a los mismos marcos que las correspondientes páginas del padre. De esta manera, el hijo hereda del padre no solo el contenido de su memoria virtual sino también su mapa de memoria (además de otros elementos como los ficheros abiertos, variables de entorno, etc.). Al heredar el mapa de memoria, las regiones del hijo estarán proyectadas en las mismas zonas de memoria física que las correspondientes regiones del padre.

En el caso del código y de otras regiones sin permiso de escritura, la compartición de la memoria física no supone ningún problema, pues el contenido nunca se va a modificar. En las regiones de datos, pila y otras regiones modificables, sin embargo, la compartición se tiene que hacer de tal manera que, cuando un proceso modifique un dato, el resto de procesos que comparten la misma zona de memoria física no vean el cambio. Dicho de otra manera, las modificaciones solo las puede ver el proceso que las hace. Evidentemente, una forma de solucionar esto es no permitiendo que se compartan realmente las regiones modificables (como se muestra en la figura 5.22, donde solo se

comparte el código), pero una opción más eficiente es utilizar la *copia en escritura* (en inglés, *copy on write* o COW). Esta técnica permite que en un primer momento todas las páginas de las regiones modificables se compartan, si bien se desactiva el permiso de escritura en las páginas de esas regiones para detectar cuándo se quiere modificar una. Ahora, si uno de los dos procesos intenta escribir en una de esas páginas, saltará una excepción, el SO comprobará que la página se encuentra en una región que está compartida y es modificable, y hará una copia de la página que ha provocado la violación de acceso a un marco nuevo. Tras esto, asignará el nuevo marco a la página que se desea escribir cambiando su entrada en la tabla de páginas de su proceso. Después, desactivará la protección contra escritura de las páginas que compartían el marco, cambiando de nuevo la información almacenada en las tablas de traducción de los procesos[6]. Finalmente, el sistema operativo reiniciará la ejecución de la instrucción que provocó la excepción.

5.3.4 Algoritmos de reemplazo de páginas

Cuando ocurre un fallo de página y no hay marcos libres disponibles, el sistema operativo debe elegir una página (en realidad, un marco) para retirarla de memoria y hacer hueco para la página que ha provocado el fallo. Si la página a eliminar fue modificada mientras estaba en memoria, debe guardarse en disco para poder recuperar su valor actual si hiciera falta más adelante. La forma de elegir la página que hay que eliminar de memoria da lugar a varios *algoritmos de reemplazo de páginas* que pasamos a describir.

Algoritmo óptimo

Según este algoritmo, de todas las páginas que hay en memoria y que se pueden expulsar, se elimina aquella para la que pasará más tiempo antes de que sea utilizada de nuevo. En la figura 5.23 podemos ver su funcionamiento. Aquí suponemos que tenemos 8 páginas y 3 marcos. Arriba aparece la serie de referencias a las páginas y, debajo de la misma, los cambios en el estado de los 3 marcos cuando se produce un fallo de página. En cada cuadro aparece la página asignada al marco correspondiente.

Inicialmente, los tres marcos están vacíos. Las referencias a las páginas 7, 0 y 1 producen un fallo de página cada una. Tras ellas, los tres marcos están ocupados y no queda ninguno libre. Cuando se hace accede a la página 2, se produce un fallo de página. Al estar todos los marcos ocupados, el algoritmo debe seleccionar entre las páginas 0, 1 y 7 para expulsar una. De ellas, seleccionará la página 7, pues, de las tres que hay, será la que se vuelva a utilizar dentro de más tiempo.

Este algoritmo es irrealizable, ya que el sistema operativo no tiene forma de saber a qué página se hará referencia más tarde. No obstante, es útil, ya que, mediante simulación, se puede estudiar cómo de buenos o malos son otros algoritmos realizables.

Algoritmo NRU: la no usada recientemente

Ya hemos visto que en cada entrada de la tabla de páginas existe un bit de referencia (bit R) utilizado para saber si se ha hecho referencia (es decir, usado) una página o no, y un bit de modificación (bit M) para saber si una página se ha modificado o no (y así saber si se debe escribir en el disco si necesita ser eliminada de memoria). Una vez el

[6]Evidentemente, si el marco es compartido por tres o más páginas, solo se desactivará la protección contra escritura en la página que ha provocado la excepción. El resto de páginas que apuntan al mismo marco seguirán teniendo dicha protección activa para detectar nuevos intentos de escritura que serán tratados de la misma manera.

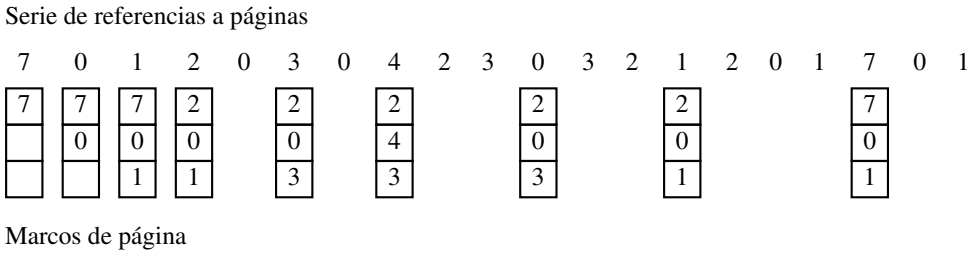


Figura 5.23. Algoritmo de reemplazo de páginas óptimo.

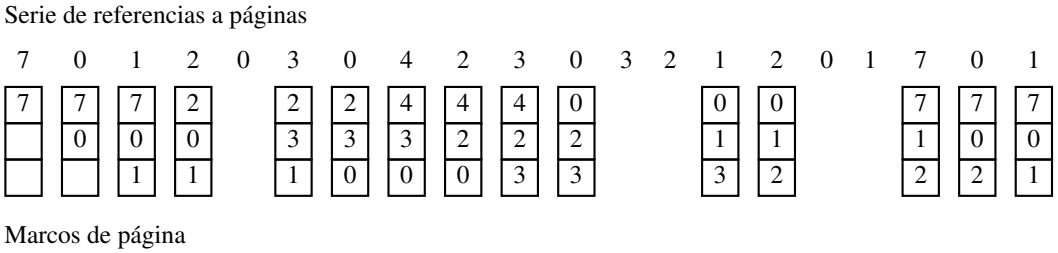


Figura 5.24. Algoritmo de reemplazo de páginas FIFO.

hardware activa cualquiera de estos 2 bits, los mismos permanece en ese estado hasta que el sistema operativo los desactiva por software.

Los bits R y M se pueden utilizar para implementar el siguiente algoritmo: durante la ejecución de un proceso, los bits R y M de cada página se activan según sea necesario. De forma periódica (por ejemplo, cada interrupción de reloj), se limpia el bit R (se pone a 0) para distinguir las páginas que no tienen referencias recientes de las que sí. Al producirse un fallo de página, si es necesario expulsar una página, el sistema operativo las inspecciona todas y las divide en cuatro categorías, según los valores de los bits R y M:

- Clase 0: R = 0, M=0.
- Clase 1: R = 0, M=1.
- Clase 2: R = 1, M=0.
- Clase 3: R = 1, M=1.

Finalmente, el algoritmo elimina una página al azar de la primera clase no vacía de número más pequeño. Es decir, si hay páginas de la clase 0, elimina cualquiera de ellas, si no, pasa a la clase 1, y así sucesivamente. Este algoritmo es sencillo, de implementación eficiente y con un rendimiento adecuado muchas veces.

Obsérvese que la existencia, que parece imposible, de las páginas de la clase 1 se produce cuando en una página de la clase 3 una interrupción de reloj limpia su bit R.

Algoritmo FIFO: primera en entrar, primera en salir

El sistema operativo tiene una lista de todas las páginas que se encuentran en memoria, siendo la primera página la más antigua y la última la más reciente. En un fallo de página, se elimina la primera página y se añade la nueva al final de la lista (ver figura 5.24).

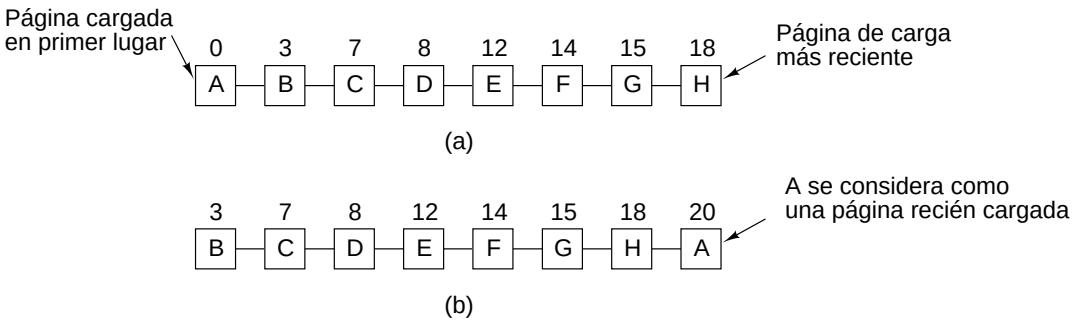


Figura 5.25. Funcionamiento de la segunda oportunidad. (a) Páginas ordenadas en orden FIFO. (b) La lista de páginas si ocurre un error de página en el instante 20 y A tiene activado su bit R.

Este algoritmo es sencillo de entender y de implementar, así como de bajo coste. Sin embargo, su gran inconveniente es que no tiene en cuenta ningún dato adicional, como la frecuencia de uso de una página, lo que hace que muchas veces produzca demasiados fallos. Por ejemplo, puede ocurrir que las primeras páginas de la lista se utilicen mucho, pero serán las que seleccione el algoritmo para reemplazar ante un fallo de página. Esto dará lugar enseguida a nuevos fallos de página, puesto que las páginas se necesitarán muy pronto.

Algoritmo de la segunda oportunidad

Es una modificación simple del algoritmo FIFO que evita deshacerse de una página de uso frecuente. Su funcionamiento es el siguiente: cuando es necesario expulsar una página, si el bit R de la primera página (la más antigua) es 0, dicha página se elimina de memoria. Si, por el contrario, el bit es 1, el bit se limpia y la página se coloca al final de la lista, como si hubiera llegado en ese momento a la memoria. Después continúa la búsqueda (ver figura 5.25). Si el bit de todas las páginas es 1, este algoritmo deriva en un simple FIFO, lo que asegura que este algoritmo siempre termina.

Algoritmo del reloj

Este algoritmo difiere del anterior solo en la implementación. Dicho de otra manera, los dos reemplazarán exactamente las mismas páginas en el mismo orden.

Aunque el algoritmo de la segunda oportunidad es razonable, es ineficiente puesto que desplaza constantemente las páginas en una lista. Un mejor enfoque es mantener las páginas en una lista circular donde una «manecilla» (que no es más que un puntero) apunta a la página más antigua (ver figura 5.26). Al ocurrir un fallo de página, se inspecciona la página a la que apunta la manecilla. Si su bit R es 0, la página se expulsa de memoria, se inserta la nueva página en su lugar en el reloj y la manecilla avanza una posición. Si R es 1, este bit se pone a 0 y la manecilla avanza a la página siguiente.

Algoritmo LRU: la usada menos recientemente

Es una buena aproximación al algoritmo óptimo, ya que se basa en una idea que se aproxima a la de este. Esta idea es que es probable que las páginas que no hayan sido utilizadas durante mucho tiempo permanezcan sin ser usadas durante bastante tiempo. Teniendo en cuenta esto, este algoritmo dice que, cuando es necesario expulsar una página, se elimina de memoria la que no ha sido utilizada desde hace más tiempo (ver figura 5.27).

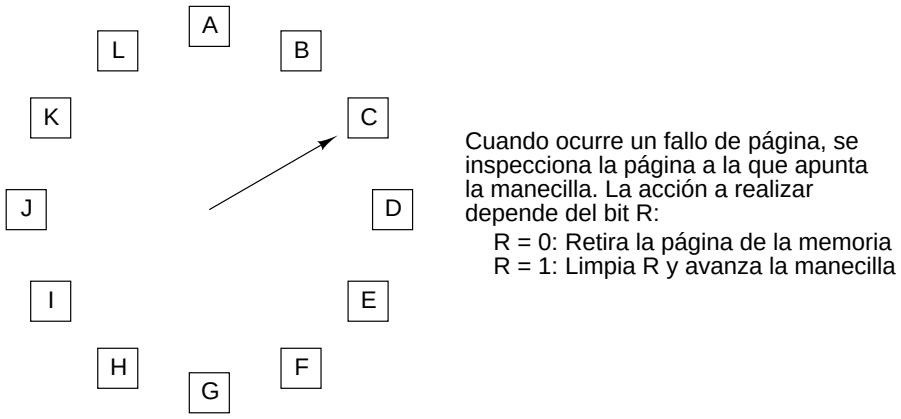


Figura 5.26. El algoritmo del reloj para el reemplazo de páginas.

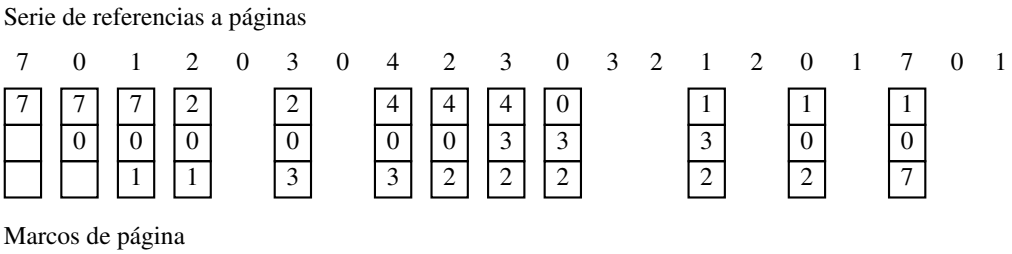


Figura 5.27. Algoritmo de reemplazo de páginas LRU.

El principal inconveniente de este algoritmo es cómo implementarlo de forma eficiente. Se puede pensar en utilizar una lista ligada de todas las páginas que hay en memoria, donde la página usada más recientemente esté al principio de la lista y la que se utilizó hace más tiempo esté al final. El problema de esta lista radica en que tendría que actualizarse en cada referencia a memoria: la página que se acaba de utilizar debería buscarse en la lista, eliminarse de su posición y trasladarse al frente. Esto haría los accesos a memoria más lentos.

Existen, no obstante, otras formas más eficientes de implementar el algoritmo LRU, siempre que el hardware proporcione el soporte adecuado. Por ejemplo, una opción es tener en hardware un contador especial C de 64 bits que se incremente automáticamente tras cada referencia a memoria. También, en cada entrada de la tabla de páginas, tendremos un campo de tamaño adecuado donde copiar C. Como el TLB contiene copias de algunas entradas de la tabla de páginas, las entradas del TLB también dispondrán de este campo. Tras cada referencia a memoria, se incrementa el valor de C y se almacena en el TLB, en la entrada correspondiente a la página a la que se hizo referencia; al no acceder a memoria principal, esta operación es muy rápida. Los valores del contador almacenados en las entradas del TLB se guardan en la tabla de páginas según se van reemplazando entradas por fallos de TLB. Al producirse un fallo de página, el sistema operativo examina todos los contadores de todas las tablas de páginas y elige el mínimo. Esa es la página que se utilizó hace más tiempo.

Otra opción es, si se tienen N marcos en memoria, utilizar una matriz de $N \times N$ bits implementada en hardware, cuyos datos iniciales son todos 0. En una referencia al marco K, el hardware primero activa todos los bits de la fila K y después desactiva todos los bits de la columna K. En cualquier instante, la fila cuyo valor en binario sea mínimo corresponderá al marco que se utilizó hace más tiempo. Esta segunda implementación la podemos ver en la figura 5.28 para 4 marcos. Nótese que, en un sistema actual con

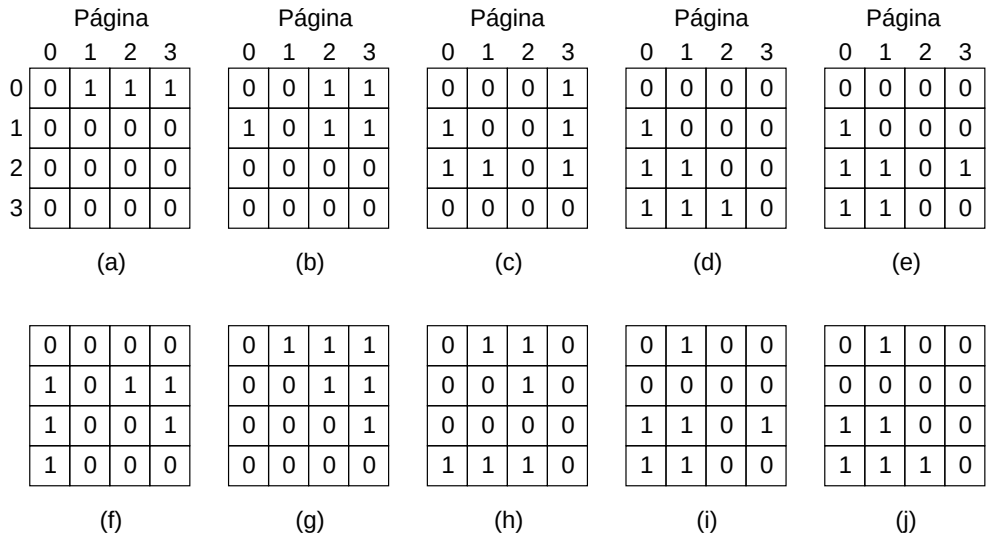


Figura 5.28. LRU usando una matriz. Los marcos se usan en el orden 0, 1, 2, 3, 2, 1, 0, 3, 2 y 3.

varios gigabytes de memoria RAM, esta implementación es inviable. Por ejemplo, con 2 GiB de RAM y marcos de 4 KiB, tendríamos $2^{19} = 524\,288$ marcos y una matriz de $2^{19} \times 2^{19} = 2^{38}$ bits o 2^{35} bytes, por lo que el tamaño de la matriz sería mayor que el tamaño de la memoria RAM disponible.

Algoritmo de maduración

Es un intento de simular el algoritmo LRU en software. Cada entrada de la tabla de páginas contiene un contador, por ejemplo, de 8 bits. En cada interrupción de reloj (llamada *marca*), se desplaza cada contador un bit a la derecha, el valor del bit R se añade al bit del extremo izquierdo del contador correspondiente y se limpia el bit R (ver figura 5.29). Al ocurrir un fallo de página, se elimina aquella cuyo contador tenga el valor más pequeño.

Hay dos diferencias importantes de este algoritmo con el LRU. La primera es que puede eliminarse una página de memoria sin ser exactamente la que se utilizó hace más tiempo. Por ejemplo, en el instante (e) de la figura 5.29 se elegiría la página 3 (contador más pequeño), aunque puede ocurrir que en realidad fuera la página 5 la que se utilizó hace más tiempo. El problema es que nosotros estudiamos las referencias entre interrupciones de reloj sin saber qué es lo que ocurre por medio. En nuestro ejemplo, las páginas 3 y 5 recibieron sus últimas referencias entre la segunda y tercera interrupción, pero no sabemos en qué orden.

La segunda diferencia es que los contadores tienen un número finito de bits. Aunque dos páginas tengan sus contadores a 0 no podemos saber si es porque a una se accedió hace 9 marcas de reloj y a la otra hace 1000.

5.3.5 Algunos aspectos de diseño para los sistemas de paginación

La descripción de la paginación que hemos realizado en las secciones anteriores es, en muchos casos, una simplificación de lo que ocurre en un sistema real. A continuación, describimos brevemente otros aspectos de un sistema de paginación que consideramos interesantes.

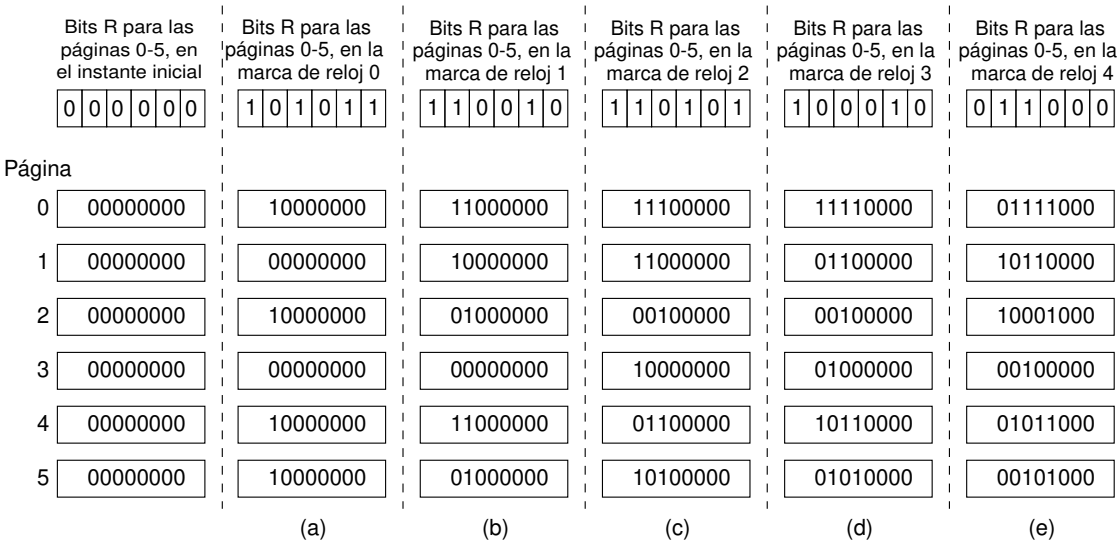


Figura 5.29. El algoritmo de maduración simula el LRU en software. En la figura se muestra la evolución de los contadores de seis páginas durante cinco marcas de reloj, desde el instante (a) hasta el (e).

	Reemplazo local	**Reemplazo global**
Asignación fija	El nº de marcos asignados a un proceso es fijo. La página a reemplazar se elige de entre los marcos asignados al proceso.	No es posible.
Asignación dinámica	El nº de marcos asignados a un proceso puede cambiar de un momento a otro. La página a reemplazar se elige de entre los marcos asignados al proceso.	La página a reemplazar se elige de entre todos los marcos disponibles en la memoria principal; esto hace que cambie el número de marcos asignados a cada proceso.

Tabla 5.1. Combinación de las políticas de reemplazo con las políticas de asignación.

Políticas de reemplazo y políticas de asignación

Aunque cada proceso tiene su propia tabla de páginas, todos comparten los marcos que hay en memoria física. Cuando un proceso produce un fallo de página y es necesario reemplazar una página, ¿la página a reemplazar se busca solo en las páginas del proceso (*reemplazo local*) o en todas las páginas que hay en memoria (*reemplazo global*)? En la figura 5.30 podemos ver una comparación entre estas dos políticas. En ambos casos, el criterio que se sigue es el de eliminar la página más antigua.

El tipo de reemplazo utilizado está estrechamente relacionado con el tipo de asignación de marcos, es decir, con el número de marcos asignados a cada proceso. La asignación puede ser *estática*, cuando a un proceso se le asigna una cantidad de marcos que no cambia durante su ejecución, o *dinámica*, cuando el número de marcos asignados varía.

La combinación de las políticas de asignación y de reemplazo dan lugar a tres posibles combinaciones, como muestra la tabla 5.1. Como se ve, los algoritmos locales se utilizan cuando a cada proceso se le asigna una cantidad fija de memoria (un número fijo

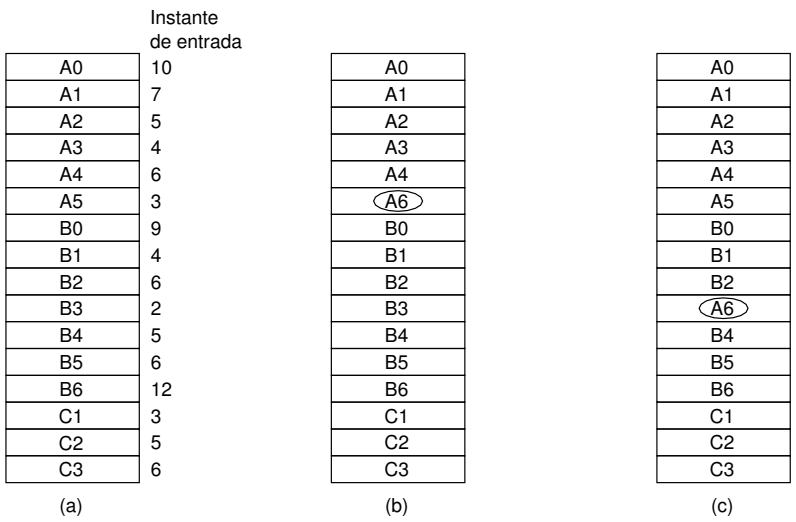


Figura 5.30. Reemplazo global *vs* reemplazo local de páginas. (a) Configuración inicial. (b) Reemplazo local de páginas. (c) Reemplazo global de páginas.

de marcos) o una cantidad variable, siempre que la cantidad no cambie continuamente, sino periódicamente tras comprobar si el proceso puede necesitar más o menos páginas. Los algoritmos globales, en cambio, se utilizan cuando el número de marcos asignados a un proceso puede cambiar en cualquier momento.

De las tres combinaciones posibles, el reemplazo local con asignación dinámica es bastante interesante. Por un lado, evita que un proceso que de repente empieza a producir fallos de página (porque, por ejemplo, necesita nuevas páginas) le quite páginas a otros procesos, como ocurriría en un reemplazo global. Por otro lado, es más flexible que una asignación fija, ya que podemos aumentar el número de marcos de un proceso si se detecta que este los necesita. Un algoritmo de asignación que funciona bien con esta combinación es el *algoritmo de frecuencia de fallos de página*: si un proceso produce muchos fallos de página, le asigna más marcos, y si produce pocos fallos, sus marcos se asignan a otros procesos. La idea es tratar de que las tasas de fallos de página de todos los procesos se mantengan dentro de unos límites razonables.

Dos últimos aspectos relacionados con la asignación de marcos a los procesos son el número mínimo de marcos que debe tener un proceso y la cantidad inicial de marcos asignados a un proceso. El primer aspecto depende del hardware, en concreto, del número máximo de páginas que se pueden utilizar en una única instrucción. Para el segundo se pueden diseñar varias políticas: la misma cantidad inicial para todos los procesos, proporcional al tamaño de cada proceso, etc.

Tamaño de página

Elegir un tamaño de página pequeño o grande tiene sus ventajas e inconvenientes que dependen de varios factores. Por un lado, las páginas pequeñas producen menos fragmentación interna en aquellas páginas que no se utilizan totalmente. Sin embargo, dan lugar a tablas de páginas más grandes que hacen que: (a) el cambio de proceso sea más lento si hay entradas de la tabla de páginas del nuevo proceso que deben cargarse en, por ejemplo, el TLB, antes de cederle la CPU y (b), si el proceso es pequeño, utilizará pocas entradas de la tabla y el espacio ocupado por el resto de entradas se desperdiciará.

Por otro lado, si las páginas son grandes, entonces las ventajas/inconvenientes de las páginas pequeñas se convierten en inconvenientes/ventajas de las páginas grandes. De este modo, las páginas grandes producen una mayor fragmentación interna (por lo tanto,

se desperdicia más memoria), si bien las tablas de páginas son más pequeñas y ocupan menos memoria.

Además de la fragmentación interna y del tamaño de las tablas de páginas, otro factor importante a tener en cuenta es que las transferencias entre memoria y disco son, por lo general, de una página. Como veremos en el siguiente tema, en el caso de los discos duros, la mayor parte del tiempo de una transferencia de disco se debe al retraso por búsqueda y rotación, por lo que hay poca diferencia entre el tiempo de transferencia de una página pequeña y el de una página grande. Además, si las páginas son pequeñas, leer X bytes de disco puede suponer leer muchas páginas mientras que, si las páginas son grandes, se deben leer pocas, lo cual puede compensar. Por ejemplo, si el tiempo de lectura de una página de 512 bytes es de 15 ms y el de una página de 8 KiB es de 25 ms, entonces leer 32 KiB en el primer caso supone 64 lecturas con un tiempo total que puede llegar a los 960 ms, mientras que en el segundo caso supone 4 lecturas con un tiempo total de 100 ms. En general, los tamaños de página de 4 y 8 KiB son frecuentes.

Hiperpaginación

Si un proceso da lugar a muchos fallos de página puede entrar en *hiperpaginación*. Esta se produce si el proceso emplea más tiempo esperando a que se resuelvan sus fallos de página que ejecutando código. Hay muchas causas. En una política local el proceso puede necesitar muchos más marcos de los que tiene. En una política global puede que se quiten marcos a un proceso para dárselos a otro que ha empezado a producir fallos de página. Si el primer proceso necesita también muchos marcos, se los quitará de nuevo al segundo, etc. En definitiva, la hiperpaginación se produce porque se necesitan muchos más marcos de los que se dispone.

Se pueden adoptar distintas soluciones para paliar el problema. La más inmediata es aumentar la capacidad de la memoria principal. Otra, que puede ser llevada a cabo por el sistema operativo si tiene mecanismos para detectar la hiperpaginación, es suspender temporalmente algunos procesos para liberar memoria y reanudarlos cuando el porcentaje de fallos de página baje hasta un nivel aceptable.

Políticas de lectura y escritura de páginas

Un fallo de página supone leer una página y, si se expulsa una página modificada, escribir otra página en disco. Ahora bien, cuando se lee la página que ha provocado el fallo, ¿se lee solo esa página o se lee alguna página más (por ejemplo, de las que se encuentran a continuación), por si se necesita poco después? Y en el caso de las páginas modificadas, ¿esperamos a que sean expulsadas para escribirlas en disco o las revisamos periódicamente y las escribimos para que cuando sean eliminadas de memoria ya no haya que esperar a que se escriban? Las respuestas a estas preguntas dan lugar a dos políticas de lectura y dos de escritura:

- *Paginación por demanda*: en un fallo de página, solo se lee la página que lo produce.
- *Prepaginación* o *paginación anticipada*: en un fallo de página, se leen la página que lo produce y varias páginas más (generalmente, las que se encuentran después en el espacio de direcciones virtuales).
- *Escritura por demanda*: una página se escribe en disco cuando se expulsa. El problema de esta política es que incrementa el tiempo de resolver muchos fallos de

página cuando hay que desalojar páginas de memoria, pues supone dos operaciones de disco: una lectura (por la página que produce el fallo) y una escritura (por la página modificada que se expulsa).

- *Escritura anticipada*: existe un hilo del núcleo llamado *demonio de paginación* que cada *X* segundos se despierta y escribe en disco las páginas modificadas. Las ventajas de esta política son que se escriben varias páginas a la vez, lo cual es más eficiente en dispositivos como los discos duros, y que muchos fallos de página se resuelven en menos tiempo al no tener que esperar una posible escritura de la página expulsada.

  Un problema que puede surgir con esta política es que muchas escrituras pueden ser inútiles si las páginas que se guardan en disco se modifican de nuevo poco después. Para evitar este problema, el demonio de paginación puede, en lugar de escribir en disco todas las páginas modificadas, buscar unas cuantas páginas que podrían ser expulsadas próximamente siguiendo el algoritmo de reemplazo usado por el sistema y, de esas páginas, escribir en disco todas las que estén modificadas.

  Además de escribir en disco posibles páginas modificadas, el demonio de paginación también puede liberar las páginas que ha revisado (modificadas o no) eliminándolas de las tablas de páginas de sus correspondientes procesos. De este modo, tendremos un conjunto de marcos libres que se podrán usar para solucionar rápidamente cualquier fallo de página que se produzca. El demonio de paginación podría revisar periódicamente la cantidad de marcos libres y liberar más si fuera necesario. A esta técnica se le conoce como *caché de páginas*[7].

  Observa que el contenido de las páginas que se liberan no tiene por qué eliminarse de memoria principal. Solo cuando el marco que ocupa una de estas páginas se necesite, su contenido se podrá descartar. De esta manera, una página liberada que se necesita poco después, podría encontrar su contenido todavía en memoria, por lo que se podría recuperar de ahí sin necesidad de leerla de disco.

5.4 SEGMENTACIÓN

La segmentación trata de aproximarse a la perspectiva que tiene el usuario de la memoria. El programador o usuario no considera la memoria como un array lineal de bytes, como ocurre en la paginación, sino que prefiere pensar en ella como un conjunto de *segmentos* de tamaño variable sin ningún orden especial. El programador ve arrays, tablas, pilas, funciones, etc. repartidos en varios segmentos (ver figura 5.31). Los elementos dentro de un segmento se identifican por su desplazamiento a partir del inicio del segmento (la primera dirección dentro de todo segmento es siempre la 0).

En la segmentación, el espacio de direcciones lógicas de un proceso se compone de un conjunto de segmentos, cada uno de los cuales tiene una posición en memoria principal y un tamaño. Cada segmento es una sucesión lineal de direcciones, desde 0 hasta su *tamaño - 1*, que puede crecer o disminuir independientemente del resto y puede tener su propia protección (lectura, ejecución, etc.).

Las direcciones generadas por los procesos especifican el número de segmento y el desplazamiento dentro de él. Para hacer corresponder las direcciones bidimensionales de los procesos (segmento, desplazamiento) con las direcciones físicas unidimensionales de la memoria principal, se utiliza una *tabla de segmentos*. Cada entrada de la tabla

[7] En realidad, el concepto de caché de páginas es más amplio que el descrito aquí, pero para una asignatura de introducción a los sistemas operativos es suficiente.

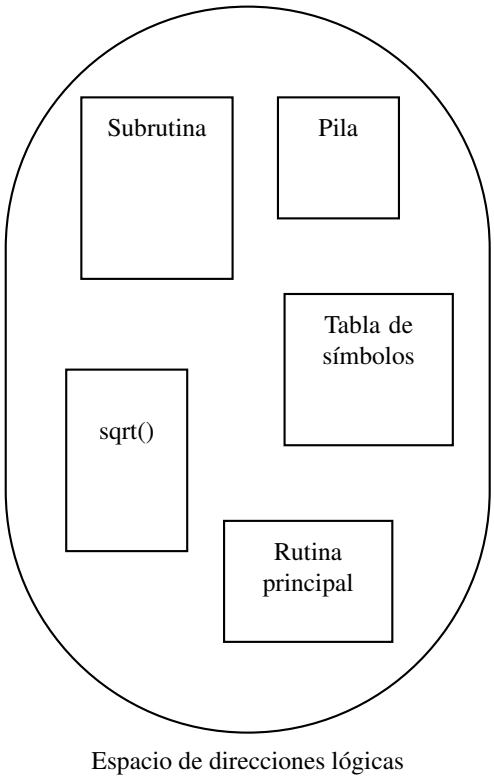


Figura 5.31. Perspectiva de un programa para un usuario.

de segmentos corresponde a un segmento y contiene una *base* (dirección de la memoria principal a partir de la cual se almacena el segmento) y un *límite* (tamaño del segmento). La figura 5.32 muestra cómo se realiza la traducción de direcciones en segmentación. En ella podemos ver que el desplazamiento se compara con el tamaño del segmento. Si es menor, se trata de una referencia válida dentro del segmento, en cuyo caso se suma la base al desplazamiento para obtener la dirección física final. Si es mayor o igual, se produce un error de direccionamiento que provoca la terminación del proceso.

En la figura 5.33 podemos ver un ejemplo de segmentación. Según la tabla de segmentos, las siguientes direcciones virtuales (segmento, desplazamiento) se traducen en estas direcciones físicas:

Dirección virtual	Dirección física
(2, 53)	4353 = 4300 + 53
(3, 852)	4052 = 3200 + 852
(0, 1222)	Excepción: error de direccionamiento

Al igual que ocurre en la paginación, cada proceso tendrá su tabla de segmentos. Habrá un registro hardware que apuntará al inicio de la tabla de segmentos del proceso que se está ejecutando. Asimismo, se puede utilizar un TLB para acelerar la traducción de direcciones.

La segmentación facilita también la protección y compartición de información al proteger y compartir segmentos enteros y no varias páginas, como en el esquema de paginación. En la figura 5.34 se muestra una comparación entre segmentación y paginación. Como se ve, en la segmentación hay muchos espacios lineales de direcciones, uno por segmento, y el espacio total de direcciones puede exceder el tamaño de la memoria principal, ya que no todos los segmentos de un proceso tienen por qué estar a la vez en

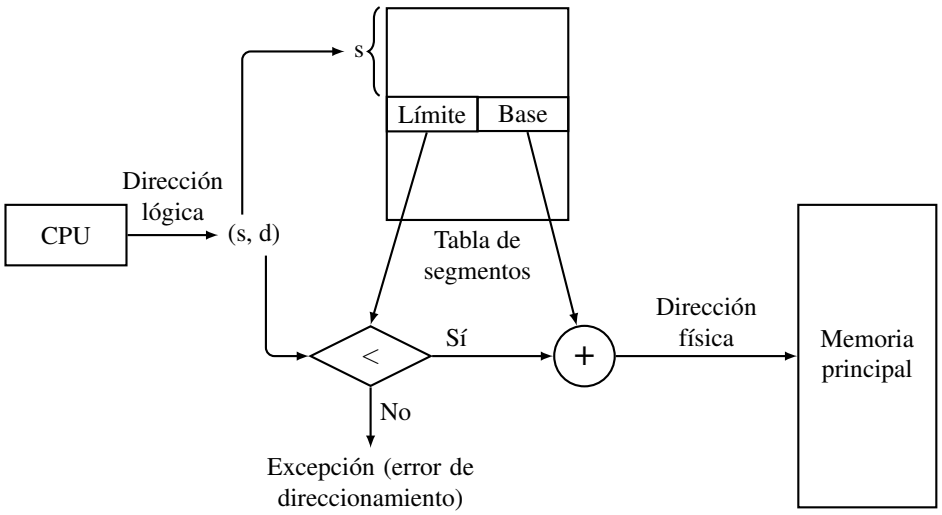


Figura 5.32. Hardware de segmentación.

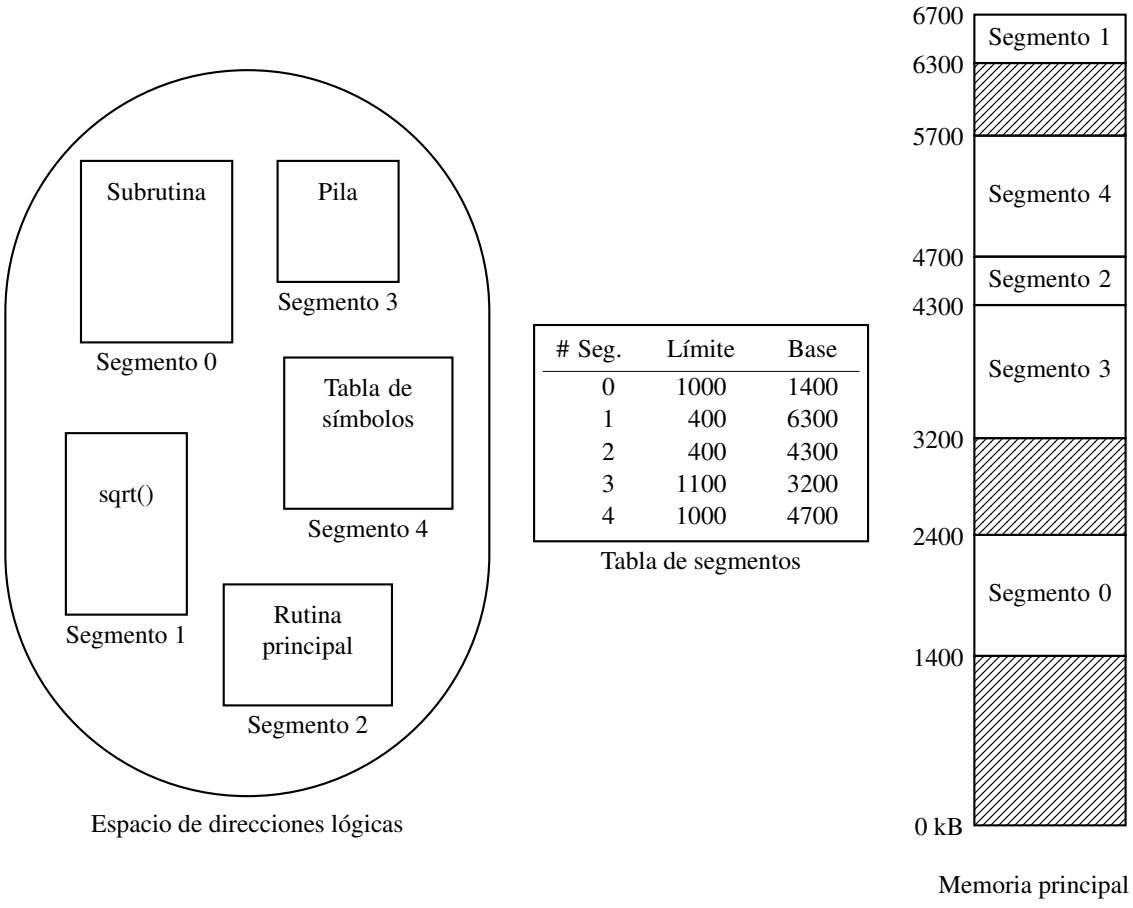


Figura 5.33. Ejemplo de segmentación.

Considerando	Paginación	Segmentación
¿Necesita saber el programador si está utilizando esta técnica?	No	Sí
¿Cuántos espacios lineales de direcciones existen?	1	Muchos
¿Puede el espacio total de direcciones exceder el tamaño de la memoria física?	Sí	Sí
¿Pueden distinguirse los procedimientos y los datos, además de protegerse individualmente?	No	Sí
¿Pueden adecuarse con facilidad las tablas de tamaños fluctuantes?	No	Sí
¿Se facilita el uso de procedimientos compartidos entre los usuarios?	No	Sí
¿Para qué se inventó esta técnica?	Para obtener un gran espacio lineal de direcciones sin tener que adquirir más memoria física	Para permitir que los programas y datos fueran separados en espacios independientes de direcciones y poder proporcionar la protección y uso de objetos compartidos

Figura 5.34. Comparación de paginación y segmentación.

memoria. Sin embargo, ningún segmento puede superar el tamaño de la memoria física y los intercambios con el disco siempre se hacen moviendo segmentos enteros.

Como se puede apreciar, la segmentación está estrechamente relacionada con los modelos de administración de memoria de multiprogramación con particiones variables vistos al principio de este tema y, como en ellos, un problema importante es el de la fragmentación externa. Por eso, la segmentación pura que hemos descrito aquí no se usa tal cual hoy en día.

5.5 SEGMENTACIÓN PAGINADA

En la segmentación paginada se paginan los segmentos, es decir, los segmentos se dividen en un número determinado de páginas. Esto hace que desaparezca la fragmentación externa, aunque no la interna, y que la asignación de memoria a los segmentos sea algo trivial, ya que no hay que buscar un hueco adecuado para cada segmento.

La figura 5.35 muestra de forma esquemática el funcionamiento de la segmentación paginada. Al igual que en la segmentación pura, lo primero que se hace es comprobar si el desplazamiento dado en la dirección lógica es menor que el tamaño del segmento a usar. Si no es así, se producirá una excepción. En caso contrario, se tratará de una referencia válida a memoria y se continuará con la traducción. Ahora, en lugar de sumar el desplazamiento a la dirección de comienzo del segmento para terminar la traducción, se toma el desplazamiento como una dirección virtual que hay que traducir mediante la

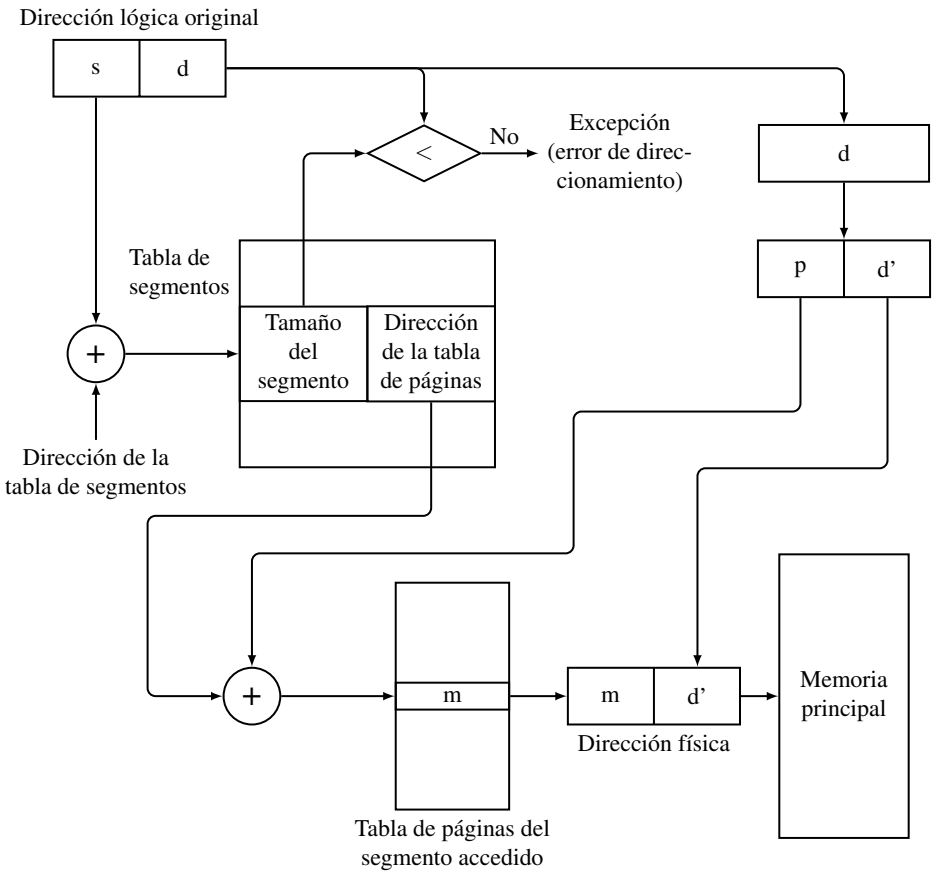


Figura 5.35. Funcionamiento de la segmentación paginada.

tabla de páginas asociada al segmento. Para ello, el desplazamiento dentro del segmento se divide en un número de página y en un desplazamiento dentro de esa página, y la traducción se termina como ya hemos visto en paginación. Observa que, puesto que cada segmento tiene un tamaño distinto, el tamaño de la tabla de páginas de cada uno, en número de entradas, también cambia, ya que depende del tamaño del segmento.

Al igual que hemos hecho en las técnicas anteriores, podemos utilizar un TLB para acelerar la traducción. La figura 5.36 muestra este caso. Como podemos ver, el TLB es direccionable por el número del segmento a usar y por el número de la página dentro de ese segmento a la que se va a acceder (el número de página se obtiene del desplazamiento dentro del segmento, como hemos explicado). Si se produce un acierto de TLB, este nos devuelve el número del marco en el que se encuentra la página y se termina la traducción. Si, por el contrario, se produce un fallo, a través de la tabla de segmentos del proceso accedemos a la tabla de páginas del segmento indicado en la dirección lógica y obtenemos de ella el marco en el que se encuentra la página a la que se va a acceder. Esta información se copia en el TLB y se reinicia la instrucción.

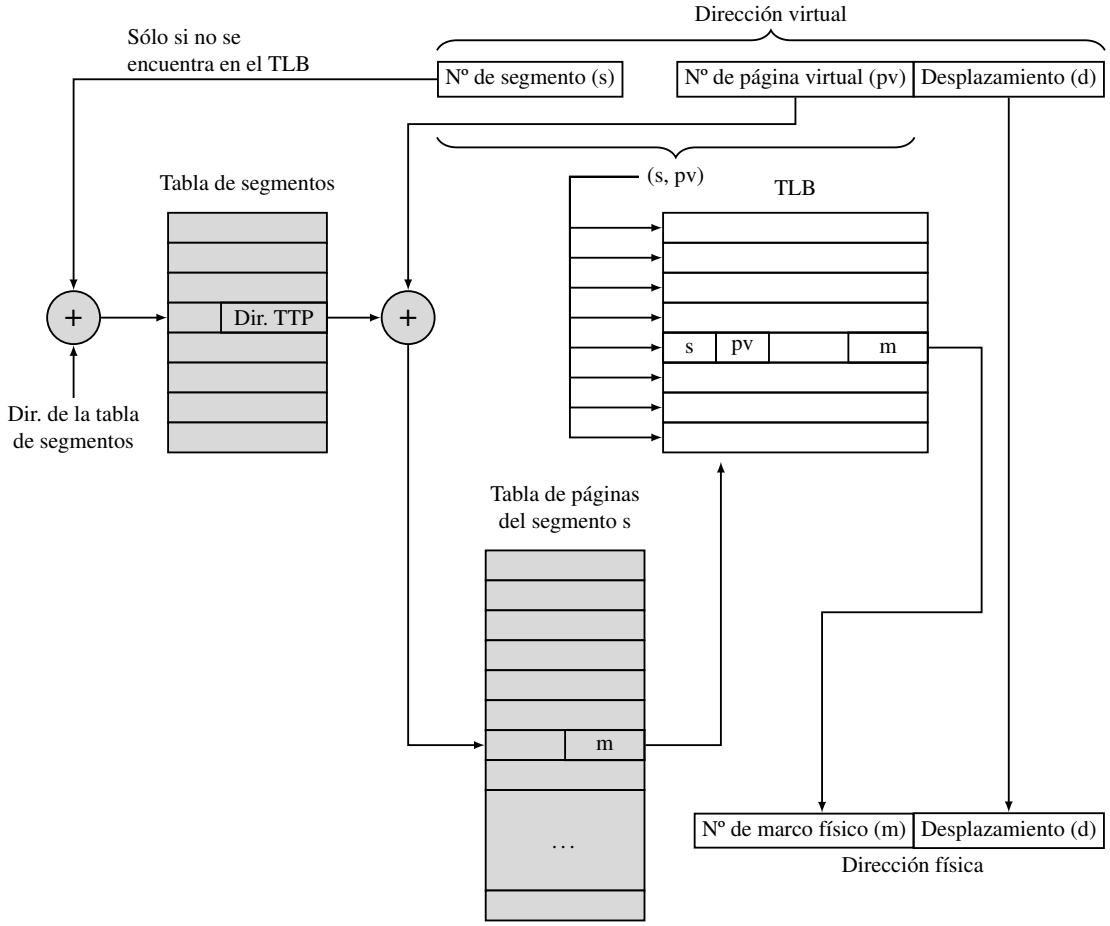


Figura 5.36. Traducción de direcciones virtuales en segmentación paginada con TLB. La parte sombreada (tabla de segmentos y tabla de páginas de un segmento) solo entra en funcionamiento en los fallos de TLB.