CAPÍTULO 7

Gestión de la memoria

En un sistema monoprogramado, la memoria se divide en dos partes: una parte para el sistema operativo (monitor residente, núcleo) y una parte para el programa actualmente en ejecución. En un sistema multiprogramado, la parte de «usuario» de la memoria se debe subdividir posteriormente para acomodar múltiples procesos. El sistema operativo es el encargado de la tarea de subdivisión y a esta tarea se le denomina ***gestión de la memoria****.*

Una gestión de la memoria efectiva es vital en un sistema multiprogramado. Si sólo unos pocos procesos se encuentran en memoria, entonces durante una gran parte del tiempo todos los procesos esperarían por operaciones de E/S y el procesador estaría ocioso. Por tanto, es necesario asignar la memoria para asegurar una cantidad de procesos listos que consuman el tiempo de procesador disponible.

Comenzaremos este capítulo con una descripción de los requisitos que la gestión de la memoria pretende satisfacer. A continuación, se mostrará la tecnología de la gestión de la memoria, analizando una variedad de esquemas simples que se han utilizado. El capítulo se enfoca en el requisito de que un programa se debe cargar en memoria principal para ejecutarse. Esta discusión introduce algunos de los principios fundamentales de la gestión de la memoria.

7.1. REQUISITOS DE LA GESTIÓN DE LA MEMORIA

Mientras se analizan varios mecanismos y políticas asociados con la gestión de la memoria, es útil mantener en mente los requisitos que la gestión de la memoria debe satisfacer. [LIST93] sugiere cinco requisitos:

- Reubicación.
- Protección.
- Compartición.
- Organización lógica.
- Organización física.

REUBICACIÓN

En un sistema multiprogramado, la memoria principal disponible se comparte generalmente entre varios procesos. Normalmente, no es posible que el programador sepa anticipadamente qué programas residirán en memoria principal en tiempo de ejecución de su programa. Adicionalmente, sería bueno poder intercambiar procesos en la memoria principal para maximizar la utilización del procesador, proporcionando un gran número de procesos para la ejecución. Una vez que un programa se ha llevado al disco, sería bastante limitante tener que colocarlo en la misma región de memoria principal donde se hallaba anteriormente, cuando éste se trae de nuevo a la memoria. Por el contrario, podría ser necesario **reubicar** el proceso a un área de memoria diferente.

Por tanto, no se puede conocer de forma anticipada dónde se va a colocar un programa y se debe permitir que los programas se puedan mover en la memoria principal, debido al intercambio o *swap*. Estos hechos ponen de manifiesto algunos aspectos técnicos relacionados con el direccionamiento, como se muestra en la Figura 7.1. La figura representa la imagen de un proceso. Por razones de simplicidad, se asumirá que la imagen de un proceso ocupa una región contigua de la memoria principal. Claramente, el sistema operativo necesitará conocer la ubicación de la información de control

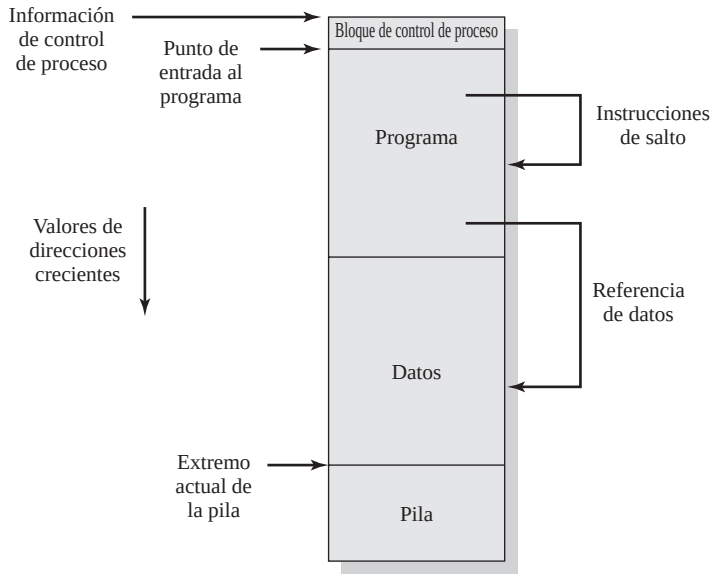


Figura 7.1. Requisitos de direccionamiento para un proceso.

del proceso y de la pila de ejecución, así como el punto de entrada que utilizará el proceso para iniciar la ejecución. Debido a que el sistema operativo se encarga de gestionar la memoria y es responsable de traer el proceso a la memoria principal, estas direcciones son fáciles de adquirir. Adicionalmente, sin embargo, el procesador debe tratar con referencias de memoria dentro del propio programa. Las instrucciones de salto contienen una dirección para referenciar la instrucción que se va a ejecutar a continuación. Las instrucciones de referencia de los datos contienen la dirección del byte o palabra de datos referenciados. De alguna forma, el hardware del procesador y el software del sistema operativo deben poder traducir las referencias de memoria encontradas en el código del programa en direcciones de memoria físicas, que reflejan la ubicación actual del programa en la memoria principal.

PROTECCIÓN

Cada proceso debe protegerse contra interferencias no deseadas por parte de otros procesos, sean accidentales o intencionadas. Por tanto, los programas de otros procesos no deben ser capaces de referenciar sin permiso posiciones de memoria de un proceso, tanto en modo lectura como escritura. Por un lado, lograr los requisitos de la reubicación incrementa la dificultad de satisfacer los requisitos de protección. Más aún, la mayoría de los lenguajes de programación permite el cálculo dinámico de direcciones en tiempo de ejecución (por ejemplo, calculando un índice de posición en un vector o un puntero a una estructura de datos). Por tanto, todas las referencias de memoria generadas por un proceso deben comprobarse en tiempo de ejecución para poder asegurar que se refieren sólo al espacio de memoria asignado a dicho proceso. Afortunadamente, se verá que los mecanismos que dan soporte a la reasignación también dan soporte al requisito de protección.

Normalmente, un proceso de usuario no puede acceder a cualquier porción del sistema operativo, ni al código ni a los datos. De nuevo, un programa de un proceso no puede saltar a una instrucción de otro proceso. Sin un trato especial, un programa de un proceso no puede acceder al área de datos de otro proceso. El procesador debe ser capaz de abortar tales instrucciones en el punto de ejecución.

Obsérvese que los requisitos de protección de memoria deben ser satisfechos por el procesador (hardware) en lugar del sistema operativo (software). Esto es debido a que el sistema operativo no puede anticipar todas las referencias de memoria que un programa hará. Incluso si tal anticipación fuera posible, llevaría demasiado tiempo calcularlo para cada programa a fin de comprobar la violación de referencias de la memoria. Por tanto, sólo es posible evaluar la permisibilidad de una referencia (acceso a datos o salto) en tiempo de ejecución de la instrucción que realiza dicha referencia. Para llevar a cabo esto, el hardware del procesador debe tener esta capacidad.

COMPARTICIÓN

Cualquier mecanismo de protección debe tener la flexibilidad de permitir a varios procesos acceder a la misma porción de memoria principal. Por ejemplo, si varios programas están ejecutando el mismo programa, es ventajoso permitir que cada proceso pueda acceder a la misma copia del programa en lugar de tener su propia copia separada. Procesos que estén cooperando en la misma tarea podrían necesitar compartir el acceso a la misma estructura de datos. Por tanto, el sistema de gestión de la memoria debe permitir el acceso controlado a áreas de memoria compartidas sin comprometer la protección esencial. De nuevo, se verá que los mecanismos utilizados para dar soporte a la reubicación soportan también capacidades para la compartición.

ORGANIZACIÓN LÓGICA

Casi invariablemente, la memoria principal de un computador se organiza como un espacio de almacenamiento lineal o unidimensional, compuesto por una secuencia de bytes o palabras. A nivel físico, la memoria secundaria está organizada de forma similar. Mientras que esta organización es similar al hardware real de la máquina, no se corresponde a la forma en la cual los programas se construyen normalmente. La mayoría de los programas se organizan en módulos, algunos de los cuales no se pueden modificar (sólo lectura, sólo ejecución) y algunos de los cuales contienen datos que se pueden modificar. Si el sistema operativo y el hardware del computador pueden tratar de forma efectiva los programas de usuarios y los datos en la forma de módulos de algún tipo, entonces se pueden lograr varias ventajas:

1. Los módulos se pueden escribir y compilar independientemente, con todas las referencias de un módulo desde otro resueltas por el sistema en tiempo de ejecución.
2. Con una sobrecarga adicional modesta, se puede proporcionar diferentes grados de protección a los módulos (sólo lectura, sólo ejecución).
3. Es posible introducir mecanismos por los cuales los módulos se pueden compartir entre los procesos. La ventaja de proporcionar compartición a nivel de módulo es que se corresponde con la forma en la que el usuario ve el problema, y por tanto es fácil para éste especificar la compartición deseada.

La herramienta que más adecuadamente satisface estos requisitos es la segmentación, que es una de las técnicas de gestión de la memoria exploradas en este capítulo.

ORGANIZACIÓN FÍSICA

Como se discute en la Sección 1.5, la memoria del computador se organiza en al menos dos niveles, conocidos como memoria principal y memoria secundaria. La memoria principal proporciona acceso

rápido a un coste relativamente alto. Adicionalmente, la memoria principal es volátil; es decir, no proporciona almacenamiento permanente. La memoria secundaria es más lenta y más barata que la memoria principal y normalmente no es volátil. Por tanto, la memoria secundaria de larga capacidad puede proporcionar almacenamiento para programas y datos a largo plazo, mientras que una memoria principal más pequeña contiene programas y datos actualmente en uso.

En este esquema de dos niveles, la organización del flujo de información entre la memoria principal y secundaria supone una de las preocupaciones principales del sistema. La responsabilidad para este flujo podría asignarse a cada programador en particular, pero no es practicable o deseable por dos motivos:

1. La memoria principal disponible para un programa más sus datos podría ser insuficiente. En este caso, el programador debería utilizar una técnica conocida como **superposición** (***overlaying***), en la cual los programas y los datos se organizan de tal forma que se puede asignar la misma región de memoria a varios módulos, con un programa principal responsable para intercambiar los módulos entre disco y memoria según las necesidades. Incluso con la ayuda de herramientas de compilación, la programación con *overlays* malgasta tiempo del programador.
2. En un entorno multiprogramado, el programador no conoce en tiempo de codificación cuánto espacio estará disponible o dónde se localizará dicho espacio.

Por tanto, está claro que la tarea de mover la información entre los dos niveles de la memoria debería ser una responsabilidad del sistema. Esta tarea es la esencia de la gestión de la memoria.

7.2. PARTICIONAMIENTO DE LA MEMORIA

La operación principal de la gestión de la memoria es traer los procesos a la memoria principal para que el procesador los pueda ejecutar. En casi todos los sistemas multiprogramados modernos, esto implica el uso de un esquema sofisticado denominado memoria virtual. Por su parte, la memoria virtual se basa en una o ambas de las siguientes técnicas básicas: segmentación y paginación. Antes de fijarse en estas técnicas de memoria virtual, se debe preparar el camino, analizando técnicas más sencillas que no utilizan memoria virtual (Tabla 7.1). Una de estas técnicas, el particionamiento, se ha utilizado en algunas variantes de ciertos sistemas operativos ahora obsoletos. Las otras dos técnicas, paginación sencilla y segmentación sencilla, no son utilizadas de forma aislada. Sin embargo, quedará más clara la discusión de la memoria virtual si se analizan primero estas dos técnicas sin tener en cuenta consideraciones de memoria virtual.

PARTICIONAMIENTO FIJO

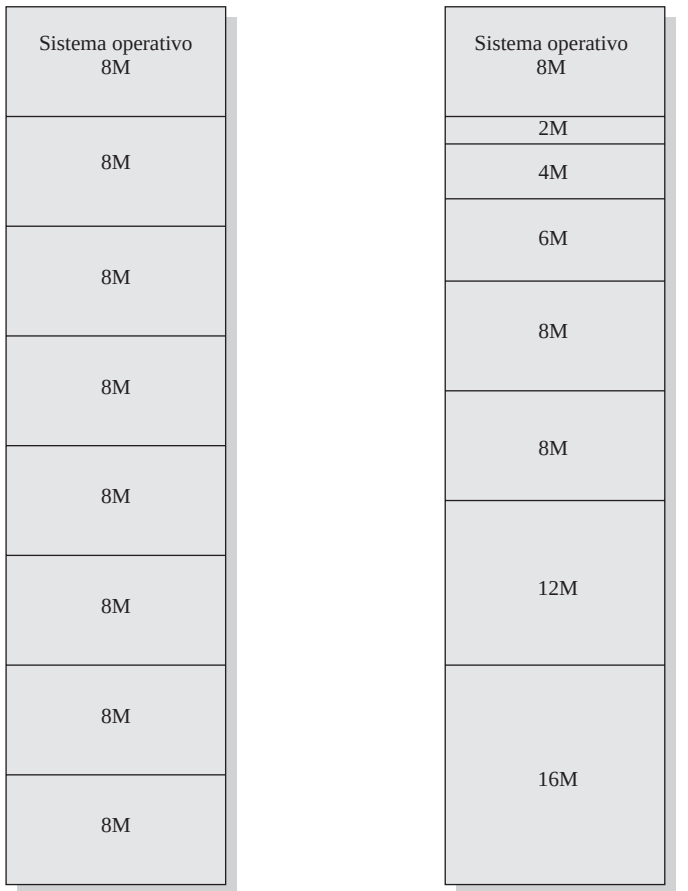
En la mayoría de los esquemas para gestión de la memoria, se puede asumir que el sistema operativo ocupa alguna porción fija de la memoria principal y que el resto de la memoria principal está disponible para múltiples procesos. El esquema más simple para gestionar la memoria disponible es repartirla en regiones con límites fijos.

Tamaños de partición

La Figura 7.2 muestra ejemplos de dos alternativas para el particionamiento fijo. Una posibilidad consiste en hacer uso de particiones del mismo tamaño. En este caso, cualquier proceso cuyo tamaño

Tabla 7.1. Técnicas de gestión de memoria.

Técnica	Descripción	Virtudes	Defectos
Particionamiento fijo	La memoria principal se divide en particiones estáticas en tiempo de generación del sistema. Un proceso se puede cargar en una partición con igual o superior tamaño.	Sencilla de implementar; poca sobrecarga para el sistema operativo.	Uso ineficiente de la memoria, debido a la fragmentación interna; debe fijarse el número máximo de procesos activos.
Particionamiento dinámico	Las particiones se crean de forma dinámica, de tal forma que cada proceso se carga en una partición del mismo tamaño que el proceso.	No existe fragmentación interna; uso más eficiente de memoria principal.	Uso ineficiente del procesador, debido a la necesidad de compactación para evitar la fragmentación externa.
Paginación sencilla	La memoria principal se divide en marcos del mismo tamaño. Cada proceso se divide en páginas del mismo tamaño que los marcos. Un proceso se carga a través de la carga de todas sus páginas en marcos disponibles, no necesariamente contiguos.	No existe fragmentación externa.	Una pequeña cantidad de fragmentación interna.
Segmentación sencilla	Cada proceso se divide en segmentos. Un proceso se carga cargando todos sus segmentos en particiones dinámicas, no necesariamente contiguas.	No existe fragmentación interna; mejora la utilización de la memoria y reduce la sobrecargada respecto al particionamiento dinámico.	Fragmentación externa.
Paginación con memoria virtual	Exactamente igual que la paginación sencilla, excepto que no es necesario cargar todas las páginas de un proceso. Las páginas no residentes se traen bajo demanda de forma automática.	No existe fragmentación externa; mayor grado de multiprogramación; gran espacio de direcciones virtuales.	Sobrecarga por la gestión compleja de la memoria.
Segmentación con memoria virtual	Exactamente igual que la segmentación, excepto que no es necesario cargar todos los segmentos de un proceso. Los segmentos no residentes se traen bajo demanda de forma automática.	No existe fragmentación interna; mayor grado de multiprogramación; gran espacio de direcciones virtuales; soporte a protección y compartición.	Sobrecarga por la gestión compleja de la memoria.



(a) Participaciones de igual tamaño (b) Participaciones de distinto tamaño

Figura 7.2. Ejemplo de particionamiento fijo de una memoria de 64 Mbytes.

es menor o igual que el tamaño de partición puede cargarse en cualquier partición disponible. Si todas las particiones están llenas y no hay ningún proceso en estado Listo o Ejecutando, el sistema operativo puede mandar a *swap* a un proceso de cualquiera de las particiones y cargar otro proceso, de forma que el procesador tenga trabajo que realizar.

Existen dos dificultades con el uso de las particiones fijas del mismo tamaño:

- Un programa podría ser demasiado grande para caber en una partición. En este caso, el programador debe diseñar el programa con el uso de *overlays*, de forma que sólo se necesite una porción del programa en memoria principal en un momento determinado. Cuando se necesita un módulo que no está presente, el programa de usuario debe cargar dicho módulo en la partición del programa, superponiéndolo (*overlaying*) a cualquier programa o datos que haya allí.
- La utilización de la memoria principal es extremadamente ineficiente. Cualquier programa, sin importar lo pequeño que sea, ocupa una partición entera. En el ejemplo, podría haber un programa cuya longitud es menor que 2 Mbytes; ocuparía una partición de 8 Mbytes cuando se lleva a la memoria. Este fenómeno, en el cual hay espacio interno malgastado debido al hecho

de que el bloque de datos cargado es menor que la partición, se conoce con el nombre de **fragmentación interna**.

Ambos problemas se pueden mejorar, aunque no resolver, utilizando particiones de tamaño diferente (Figura 7.2b). En este ejemplo, los programas de 16 Mbytes se pueden acomodar sin *overlays*. Las particiones más pequeñas de 8 Mbytes permiten que los programas más pequeños se puedan acomodar sin menor fragmentación interna.

Algoritmo de ubicación

Con particiones del mismo tamaño, la ubicación de los procesos en memoria es trivial. En cuanto haya una partición disponible, un proceso se carga en dicha partición. Debido a que todas las particiones son del mismo tamaño, no importa qué partición se utiliza. Si todas las particiones se encuentran ocupadas por procesos que no están listos para ejecutar, entonces uno de dichos procesos debe llevarse a disco para dejar espacio para un nuevo proceso. Cuál de los procesos se lleva a disco es una decisión de planificación; este tema se describe en la Parte Cuatro.

Con particiones de diferente tamaño, hay dos formas posibles de asignar los procesos a las particiones. La forma más sencilla consiste en asignar cada proceso a la partición más pequeña dentro de la cual cabe[1]. En este caso, se necesita una cola de planificación para cada partición, que mantenga procesos en disco destinados a dicha partición (Figura 7.3a). La ventaja de esta técnica es que los procesos siempre se asignan de tal forma que se minimiza la memoria malgastada dentro de una partición (fragmentación interna).

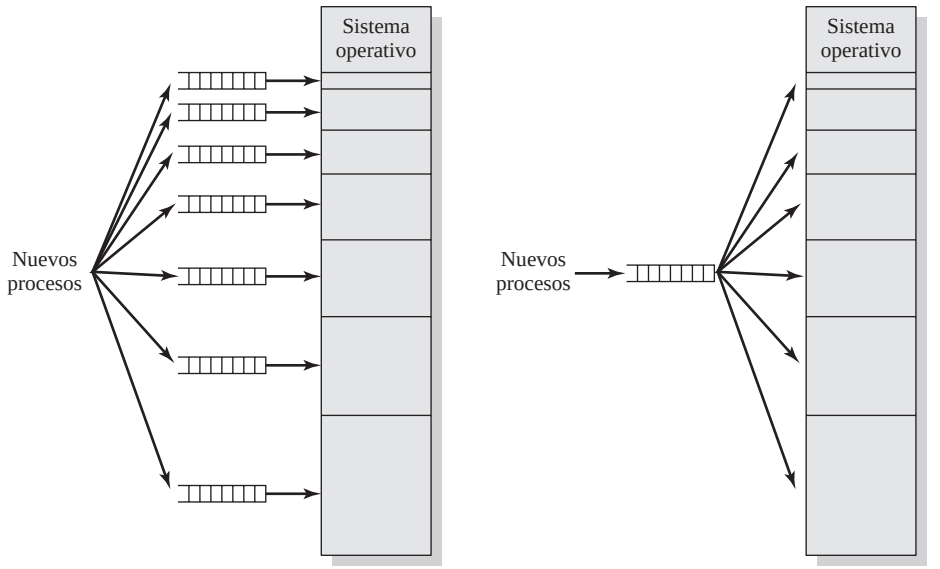
Aunque esta técnica parece óptima desde el punto de vista de una partición individual, no es óptima desde el punto de vista del sistema completo. En la Figura 7.2b, por ejemplo, se considera un caso en el que no haya procesos con un tamaño entre 12 y 16M en un determinado instante de tiempo. En este caso, la partición de 16M quedará sin utilizarse, incluso aunque se puede asignar dicha partición a algunos procesos más pequeños. Por tanto, una técnica óptima sería emplear una única cola para todos los procesos (Figura 7.3b). En el momento de cargar un proceso en la memoria principal, se selecciona la partición más pequeña disponible que puede albergar dicho proceso. Si todas las particiones están ocupadas, se debe llevar a cabo una decisión para enviar a *swap* a algún proceso. Tiene preferencia a la hora de ser expulsado a disco el proceso que ocupe la partición más pequeña que pueda albergar al proceso entrante. Es también posible considerar otros factores, como la prioridad o una preferencia por expulsar a disco procesos bloqueados frente a procesos listos.

El uso de particiones de distinto tamaño proporciona un grado de flexibilidad frente a las particiones fijas. Adicionalmente, se puede decir que los esquemas de particiones fijas son relativamente sencillos y requieren un soporte mínimo por parte del sistema operativo y una sobrecarga de procesamiento mínimo. Sin embargo, tiene una serie de desventajas:

- El número de particiones especificadas en tiempo de generación del sistema limita el número de proceso activos (no suspendidos) del sistema.
- Debido a que los tamaños de las particiones son preestablecidos en tiempo de generación del sistema, los trabajos pequeños no utilizan el espacio de las particiones eficientemente. En un entorno donde el requisito de almacenamiento principal de todos los trabajos se conoce de an-

---

[1] Se asume que se conoce el tamaño máximo de memoria que un proceso requerirá. No siempre es el caso. Si no se sabe lo que un proceso puede ocupar, la única alternativa es un esquema de *overlays* o el uso de memoria virtual.



(a) Una cola de procesos por participación (b) Un única cola

Figura 7.3. Asignación de memoria para particionamiento fijo.

temano, esta técnica puede ser razonable, pero en la mayoría de los casos, se trata de una técnica ineficiente.

El uso de particionamiento fijo es casi desconocido hoy en día. Un ejemplo de un sistema operativo exitoso que sí utilizó esta técnica fue un sistema operativo de los primeros *mainframes* de IBM, el sistema operativo OS/MFT (*Multiprogramming with a Fixed Number of Tasks*; Multiprogramado con un número fijo de tareas).

PARTICIONAMIENTO DINÁMICO

Para vencer algunas de las dificultades con particionamiento fijo, se desarrolló una técnica conocida como particionamiento dinámico. De nuevo, esta técnica se ha sustituido por técnicas de gestión de memoria más sofisticadas. Un sistema operativo importante que utilizó esta técnica fue el sistema operativo de *mainframes* de IBM, el sistema operativo OS/MVT (*Multiprogramming with a Variable Number of Tasks*; Multiprogramado con un número variable de tareas).

Con particionamiento dinámico, las particiones son de longitud y número variable. Cuando se lleva un proceso a la memoria principal, se le asigna exactamente tanta memoria como requiera y no más. Un ejemplo, que utiliza 64 Mbytes de memoria principal, se muestra en la Figura 7.4. Inicialmente, la memoria principal está vacía, excepto por el sistema operativo (a). Los primeros tres procesos se cargan justo donde el sistema operativo finaliza y ocupando el espacio justo para cada proceso (b, c, d). Esto deja un «hueco» al final de la memoria que es demasiado pequeño para un cuarto proceso. En algún momento, ninguno de los procesos que se encuentran en memoria está disponible. El sistema operativo lleva el proceso 2 al disco (e), que deja suficiente espacio para cargar un nuevo proceso, el proceso 4 (f). Debido a que el proceso 4 es más pequeño que el proceso 2, se crea otro pequeño hueco. Posteriormente, se alcanza un punto en el cual ninguno de los procesos de la memoria principal está listo, pero el proceso 2, en estado Listo-Suspendido, está disponible. Porque no hay es-

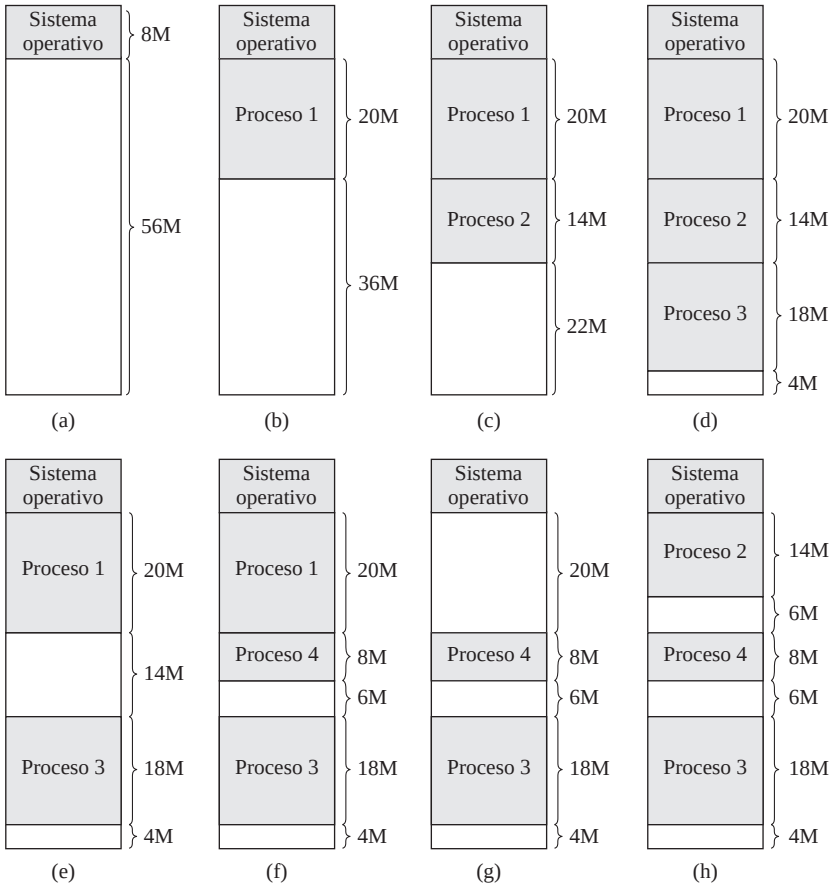


Figura 7.4. El efecto del particionamiento dinámico.

pacio suficiente en la memoria para el proceso 2, el sistema operativo lleva a disco el proceso 1 (g) y lleva a la memoria el proceso 2 (h).

Como muestra este ejemplo, el método comienza correctamente, pero finalmente lleva a una situación en la cual existen muchos huecos pequeños en la memoria. A medida que pasa el tiempo, la memoria se fragmenta cada vez más y la utilización de la memoria se decrementa. Este fenómeno se conoce como **fragmentación externa**, indicando que la memoria que es externa a todas las particiones se fragmenta de forma incremental, por contraposición a lo que ocurre con la fragmentación interna, descrita anteriormente.

Una técnica para eliminar la fragmentación externa es la **compactación**: de vez en cuando, el sistema operativo desplaza los procesos en memoria, de forma que se encuentren contiguos y de este modo toda la memoria libre se encontrará unida en un bloque. Por ejemplo, en la Figura 7.4R, la compactación permite obtener un bloque de memoria libre de longitud 16M. Esto sería suficiente para cargar un proceso adicional. La desventaja de la compactación es el hecho de que se trata de un procedimiento que consume tiempo y malgasta tiempo de procesador. Obsérvese que la compactación requiere la capacidad de reubicación dinámica. Es decir, debe ser posible mover un programa desde una región a otra en la memoria principal sin invalidar las referencias de la memoria de cada programa (*véase* Apéndice 7A).

Algoritmo de ubicación

Debido a que la compactación de memoria consume una gran cantidad de tiempo, el diseñador del sistema operativo debe ser inteligente a la hora de decidir cómo asignar la memoria a los procesos (cómo eliminar los huecos). A la hora de cargar o intercambiar un proceso a la memoria principal, y siempre que haya más de un bloque de memoria libre de suficiente tamaño, el sistema operativo debe decidir qué bloque libre asignar.

Tres algoritmos de colocación que pueden considerarse son mejor-ajuste (*best-fit*), primer-ajuste(*first-fit*) y siguiente-ajuste (*next-fit*). Todos, por supuesto, están limitados a escoger entre los bloques libres de la memoria principal que son iguales o más grandes que el proceso que va a llevarse a la memoria. **Mejor-ajuste** escoge el bloque más cercano en tamaño a la petición. **Primer-ajuste** comienza a analizar la memoria desde el principio y escoge el primer bloque disponible que sea suficientemente grande. **Siguiente-ajuste** comienza a analizar la memoria desde la última colocación y elige el siguiente bloque disponible que sea suficientemente grande.

La Figura 7.5a muestra un ejemplo de configuración de memoria después de un número de colocaciones e intercambios a disco. El último bloque que se utilizó fue un bloque de 22 Mbytes del cual se crea una partición de 14 Mbytes. La Figura 7.5b muestra la diferencia entre los algoritmos de mejor-, primer- y siguiente- ajuste a la hora de satisfacer una petición de asignación de 16 Mbytes. Me-

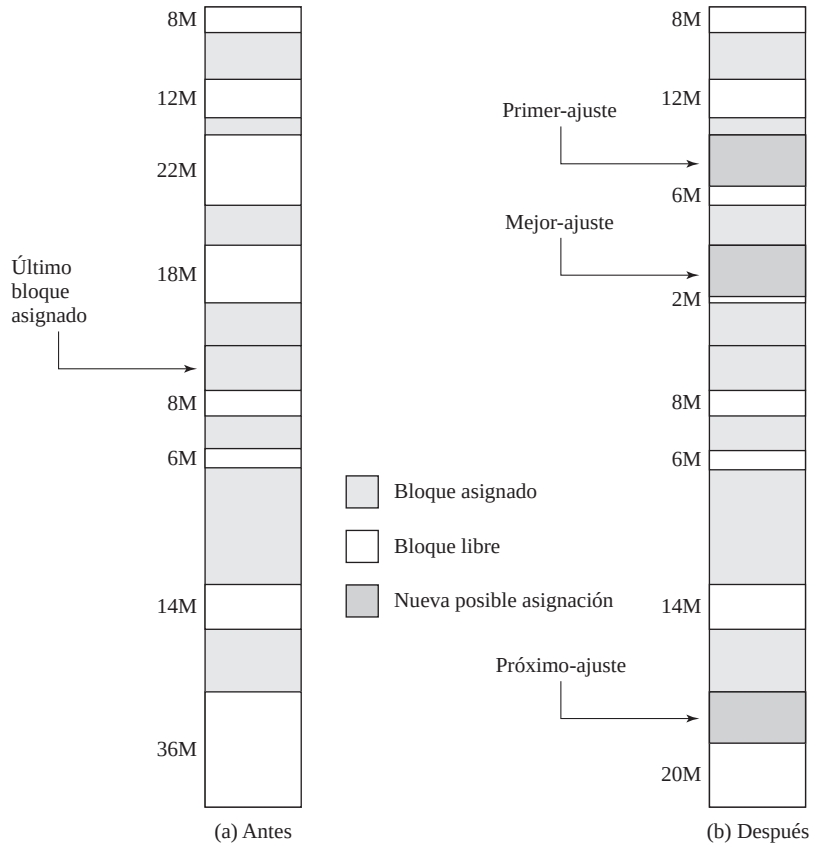


Figura 7.5. Ejemplo de configuración de la memoria antes y después de la asignación de un bloque de 16 Mbytes.

jor-ajuste busca la lista completa de bloques disponibles y hace uso del bloque de 18 Mbytes, dejando un fragmento de 2 Mbytes. *Primer-ajuste* provoca un fragmento de 6 Mbytes, y *siguiente-ajuste* provoca un fragmento de 20 Mbytes.

Cuál de estas técnicas es mejor depende de la secuencia exacta de intercambio de procesos y del tamaño de dichos procesos. Sin embargo, se pueden hacer algunos comentarios (*véase* también [BREN89], [SHOR75] y [BAYS77]). El algoritmo primer-ajuste no es sólo el más sencillo, sino que normalmente es también el mejor y más rápido. El algoritmo siguiente-ajuste tiende a producir resultados ligeramente peores que el primer-ajuste. El algoritmo siguiente-ajuste llevará más frecuentemente a una asignación de un bloque libre al final de la memoria. El resultado es que el bloque más grande de memoria libre, que normalmente aparece al final del espacio de la memoria, se divide rápidamente en pequeños fragmentos. Por tanto, en el caso del algoritmo siguiente-ajuste se puede requerir más frecuentemente la compactación. Por otro lado, el algoritmo primer-ajuste puede dejar el final del espacio de almacenamiento con pequeñas particiones libres que necesitan buscarse en cada paso del primer-ajuste siguiente. El algoritmo mejor-ajuste, a pesar de su nombre, su comportamiento normalmente es el peor. Debido a que el algoritmo busca el bloque más pequeño que satisfaga la petición, garantiza que el fragmento que quede sea lo más pequeño posible. Aunque cada petición de memoria siempre malgasta la cantidad más pequeña de la memoria, el resultado es que la memoria principal se queda rápidamente con bloques demasiado pequeños para satisfacer las peticiones de asignación de la memoria. Por tanto, la compactación de la memoria se debe realizar más frecuentemente que con el resto de los algoritmos.

Algoritmo de reemplazamiento

En un sistema multiprogramado que utiliza particionamiento dinámico, puede haber un momento en el que todos los procesos de la memoria principal estén en estado bloqueado y no haya suficiente memoria para un proceso adicional, incluso después de llevar a cabo una compactación. Para evitar malgastar tiempo de procesador esperando a que un proceso se desbloquee, el sistema operativo intercambiará alguno de los procesos entre la memoria principal y disco para hacer sitio a un nuevo proceso o para un proceso que se encuentre en estado Listo-Suspendido. Por tanto, el sistema operativo debe escoger qué proceso reemplazar. Debido a que el tema de los algoritmos de reemplazo se contemplará en detalle respecto a varios esquemas de la memoria virtual, se pospone esta discusión hasta entonces.

SISTEMA BUDDY

Ambos esquemas de particionamiento, fijo y dinámico, tienen desventajas. Un esquema de particionamiento fijo limita el número de procesos activos y puede utilizar el espacio ineficientemente si existe un mal ajuste entre los tamaños de partición disponibles y los tamaños de los procesos. Un esquema de particionamiento dinámico es más complejo de mantener e incluye la sobrecarga de la compactación. Un compromiso interesante es el sistema *buddy* ([KNUT97], [PETE77]).

En un sistema *buddy*, los bloques de memoria disponibles son de tamaño 2^k, $L \leq K \leq U$, donde

2^L = bloque de tamaño más pequeño asignado

2^U = bloque de tamaño mayor asignado; normalmente 2^U es el tamaño de la memoria completa disponible

Para comenzar, el espacio completo disponible se trata como un único bloque de tamaño 2^U. Si se realiza una petición de tamaño *s*, tal que $2^{U-1} < s \leq 2^U$, se asigna el bloque entero. En otro caso, el blo-

que se divide en dos bloques *buddy* iguales de tamaño 2^{U-1}. Si $2^{U-2} < s \leq 2^{U-1}$, entonces se asigna la petición a uno de los otros dos bloques. En otro caso, uno de ellos se divide por la mitad de nuevo. Este proceso continúa hasta que el bloque más pequeño mayor o igual que *s* se genera y se asigna a la petición. En cualquier momento, el sistema *buddy* mantiene una lista de huecos (bloques sin asignar) de cada tamaño 2^i. Un hueco puede eliminarse de la lista (i+1) dividiéndolo por la mitad para crear dos bloques de tamaño 2^i en la lista *i*. Siempre que un par de bloques de la lista *i* no se encuentren asignados, son eliminados de dicha lista y unidos en un único bloque de la lista (i+1). Si se lleva a cabo una petición de asignación de tamaño *k* tal que $2^{i-1} < k \leq 2^i$, se utiliza el siguiente algoritmo recursivo (de [LIST93]) para encontrar un hueco de tamaño 2^i:

```
void obtener_hueco (int i)
{
    if (i==(U+1))
        <fallo>;
    if (<lista_i vacía>)
    {
        obtener_hueco(i+1);
        <dividir hueco en dos buddies>;
        <colocar buddies en lista_i>;
    }
    <tomar primer hueco de la lista_i>;
}
```

La Figura 7.6 muestra un ejemplo que utiliza un bloque inicial de 1 Mbyte. La primera petición, A, es de 100 Kbytes, de tal forma que se necesita un bloque de 128K. El bloque inicial se divide en dos de 512K. El primero de éstos se divide en dos bloques de 256K, y el primero de éstos se divide en dos de 128K, uno de los cuales se asigna a A. La siguiente petición, B, solicita un bloque de 256K. Dicho bloque está disponible y es asignado. El proceso continúa, provocando divisiones y fusiones de bloques cuando se requiere. Obsérvese que cuando se libera E, se unen dos bloques de 128K en un bloque de 256K, que es inmediatamente unido con su bloque compañero (*buddy*).

La Figura 7.7 muestra una representación en forma de árbol binario de la asignación de bloques inmediatamente después de la petición «Liberar B». Los nodos hoja representan el particionamiento de la memoria actual. Si dos bloques son nodos hoja, entonces al menos uno de ellos debe estar asignado; en otro caso, se unen en un bloque mayor.

El sistema *buddy* es un compromiso razonable para eliminar las desventajas de ambos esquemas de particionamiento, fijo y variable, pero en los sistemas operativos contemporáneos, la memoria virtual basada en paginación y segmentación es superior. Sin embargo, el sistema *buddy* se ha utilizado en sistemas paralelos como una forma eficiente de asignar y liberar programas paralelos (por ejemplo, *véase* [JOHN92]). Una forma modificada del sistema *buddy* se utiliza en la asignación de memoria del núcleo UNIX (descrito en el Capítulo 8).

REUBICACIÓN

Antes de considerar formas de tratar los problemas del particionamiento, se va a aclarar un aspecto relacionado con la colocación de los procesos en la memoria. Cuando se utiliza el esquema de particionamiento fijo de la Figura 7.3a, se espera que un proceso siempre se asigne a la misma partición.

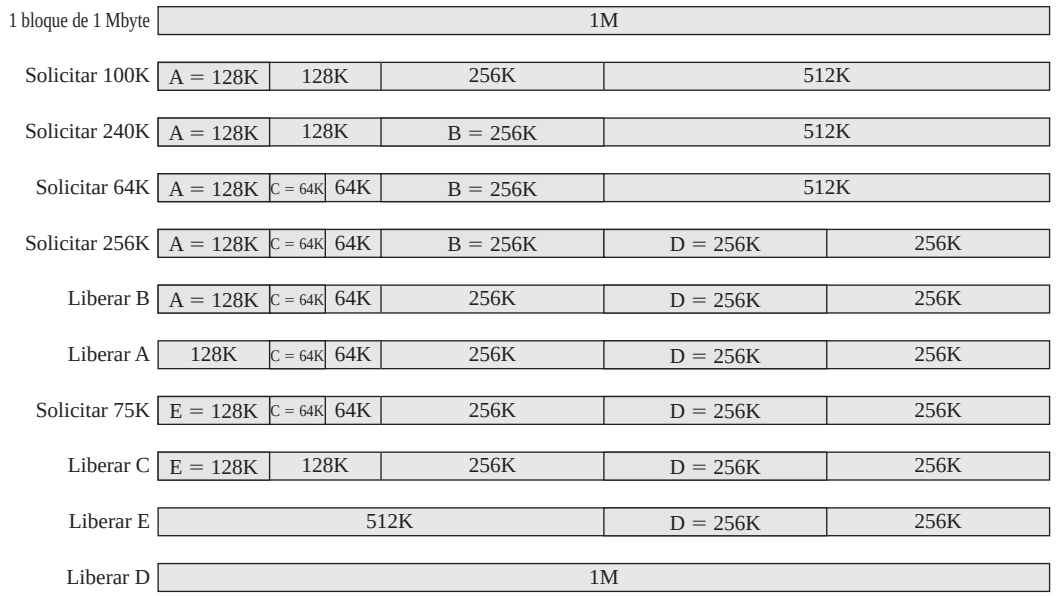


Figura 7.6. Ejemplo de sistema *Buddy.*

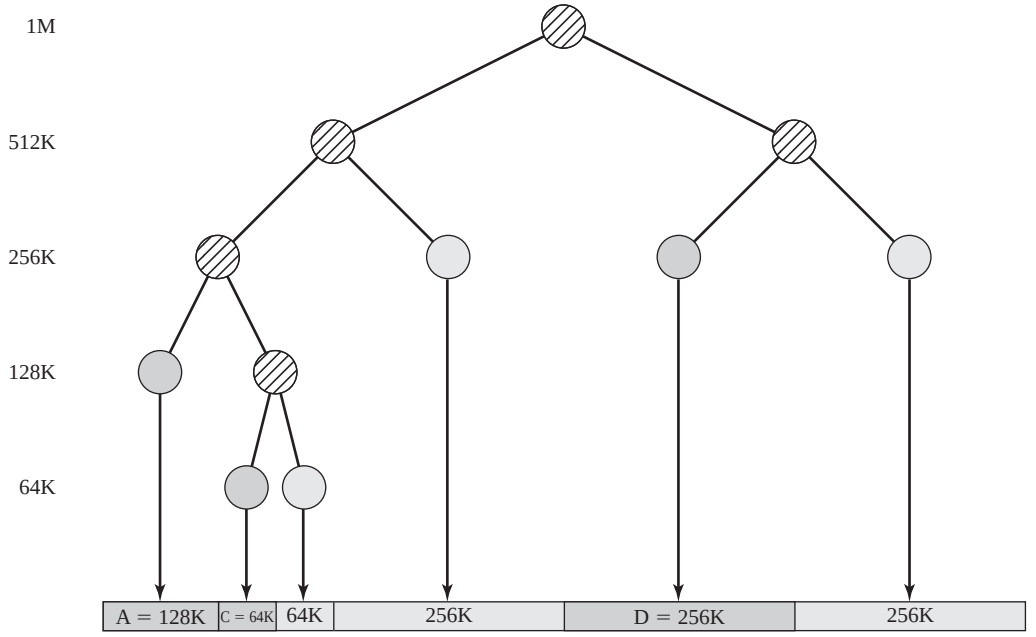


Figura 7.7. Representación en forma de árbol del sistema *Buddy.*

Es decir, sea cual sea la partición seleccionada cuando se carga un nuevo proceso, ésta será la utilizada para el intercambio del proceso entre la memoria y el área de *swap* en disco. En este caso, se utiliza un cargador sencillo, tal y como se describe en el Apéndice 7A: cuando el proceso se carga por

primera vez, todas las referencias de la memoria relativas del código se reemplazan por direcciones de la memoria principal absolutas, determinadas por la dirección base del proceso cargado.

En el caso de particiones de igual tamaño (Figura 7.2), y en el caso de una única cola de procesos para particiones de distinto tamaño (Figura 7.3b), un proceso puede ocupar diferentes particiones durante el transcurso de su ciclo de vida. Cuando la imagen de un proceso se crea por primera vez, se carga en la misma partición de memoria principal. Más adelante, el proceso podría llevarse a disco; cuando se trae a la memoria principal posteriormente, podría asignarse a una partición distinta de la última vez. Lo mismo ocurre en el caso del particionamiento dinámico. Obsérvese en las Figuras 7.4c y h que el proceso 2 ocupa dos regiones diferentes de memoria en las dos ocasiones que se trae a la memoria. Más aún, cuando se utiliza la compactación, los procesos se desplazan mientras están en la memoria principal. Por tanto, las ubicaciones (de las instrucciones y los datos) referenciadas por un proceso no son fijas. Cambiarán cada vez que un proceso se intercambia o se desplaza. Para resolver este problema, se realiza una distinción entre varios tipos de direcciones. Una **dirección lógica** es una referencia a una ubicación de memoria independiente de la asignación actual de datos a la memoria; se debe llevar a cabo una traducción a una dirección física antes de que se alcance el acceso a la memoria. Una **dirección relativa** es un ejemplo particular de dirección lógica, en el que la dirección se expresa como una ubicación relativa a algún punto conocido, normalmente un valor en un registro del procesador. Una **dirección física**, o dirección absoluta, es una ubicación real de la memoria principal.

Los programas que emplean direcciones relativas de memoria se cargan utilizando carga dinámica en tiempo de ejecución (*véase* Apéndice 7A, donde se recoge una discusión). Normalmente, todas las referencias de memoria de los procesos cargados son relativas al origen del programa. Por tanto, se necesita un mecanismo hardware para traducir las direcciones relativas a direcciones físicas de la memoria principal, en tiempo de ejecución de la instrucción que contiene dicha referencia.

La Figura 7.8 muestra la forma en la que se realiza normalmente esta traducción de direcciones. Cuando un proceso se asigna al estado Ejecutando, un registro especial del procesador, algunas veces llamado registro base, carga la dirección inicial del programa en la memoria principal. Existe también un registro «valla», que indica el final de la ubicación del programa; estos valores se establecen cuando el programa se carga en la memoria o cuando la imagen del proceso se lleva a la memoria. A lo largo de la ejecución del proceso, se encuentran direcciones relativas. Éstas incluyen los contenidos del registro de las instrucciones, las direcciones de instrucciones que ocurren en los saltos e instrucciones *call*, y direcciones de datos existentes en instrucciones de carga y almacenamiento. El procesador manipula cada dirección relativa, a través de dos pasos. Primero, el valor del registro base se suma a la dirección relativa para producir una dirección absoluta. Segundo, la dirección resultante se compara con el valor del registro «valla». Si la dirección se encuentra dentro de los límites, entonces se puede llevar a cabo la ejecución de la instrucción. En otro caso, se genera una interrupción, que debe manejar el sistema operativo de algún modo.

El esquema de la Figura 7.8 permite que se traigan a memoria los programas y que se lleven a disco, a lo largo de la ejecución. También proporciona una medida de protección: cada imagen del proceso está aislada mediante los contenidos de los registros base y valla. Además, evita accesos no autorizados por parte de otros procesos.

7.3. PAGINACIÓN

Tanto las particiones de tamaño fijo como variable son ineficientes en el uso de la memoria; los primeros provocan fragmentación interna, los últimos fragmentación externa. Supóngase, sin embargo, que la memoria principal se divide en porciones de tamaño fijo relativamente pequeños, y que cada proceso también se divide en porciones pequeñas del mismo tamaño fijo. A dichas porciones del pro-

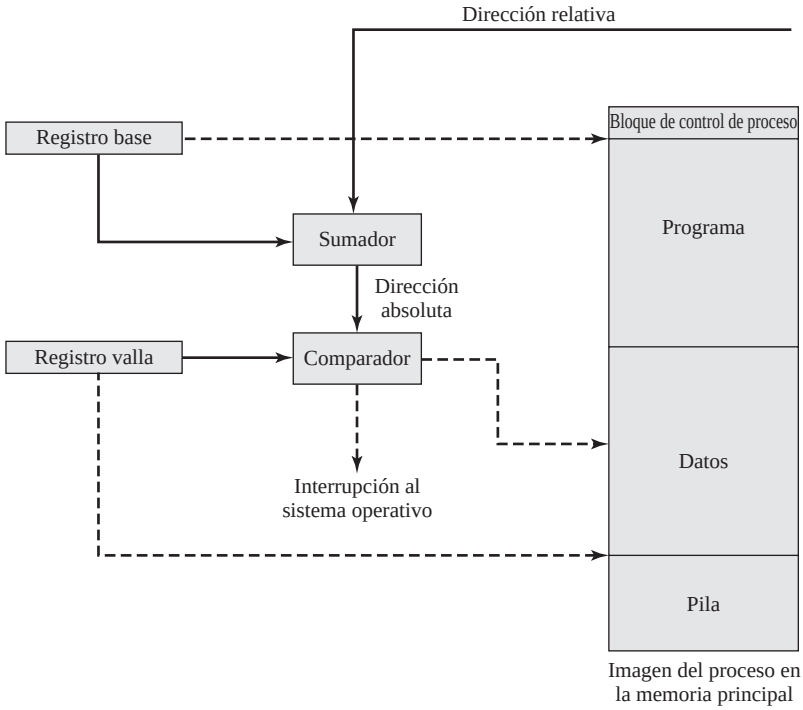


Figura 7.8. Soporte hardware para la reubicación.

ceso, conocidas como **páginas**, se les asigna porciones disponibles de memoria, conocidas como **marcos**, o marcos de páginas. Esta sección muestra que el espacio de memoria malgastado por cada proceso debido a la fragmentación interna corresponde sólo a una fracción de la última página de un proceso. No existe fragmentación externa.

La Figura 7.9 ilustra el uso de páginas y marcos. En un momento dado, algunos de los marcos de la memoria están en uso y algunos están libres. El sistema operativo mantiene una lista de marcos libres. El proceso A, almacenado en disco, está formado por cuatro páginas. En el momento de cargar este proceso, el sistema operativo encuentra cuatro marcos libres y carga las cuatro páginas del proceso A en dichos marcos (Figura 7.9b). El proceso B, formado por tres páginas, y el proceso C, formado por cuatro páginas, se cargan a continuación. En ese momento, el proceso B se suspende y deja la memoria principal. Posteriormente, todos los procesos de la memoria principal se bloquean, y el sistema operativo necesita traer un nuevo proceso, el proceso D, que está formado por cinco páginas.

Ahora supóngase, como en este ejemplo, que no hay suficientes marcos contiguos sin utilizar para ubicar un proceso. ¿Esto evitaría que el sistema operativo cargara el proceso D? La respuesta es no, porque una vez más se puede utilizar el concepto de dirección lógica. Un registro base sencillo de direcciones no basta en esta ocasión. En su lugar, el sistema operativo mantiene una **tabla de páginas** por cada proceso. La tabla de páginas muestra la ubicación del marco por cada página del proceso. Dentro del programa, cada dirección lógica está formada por un número de página y un desplazamiento dentro de la página. Es importante recordar que en el caso de una partición simple, una dirección lógica es la ubicación de una palabra relativa al comienzo del programa; el procesador la traduce en una dirección física. Con paginación, la traducción de direcciones lógicas a físicas las realiza también el hardware del procesador. Ahora el procesador debe conocer cómo acceder a la ta-

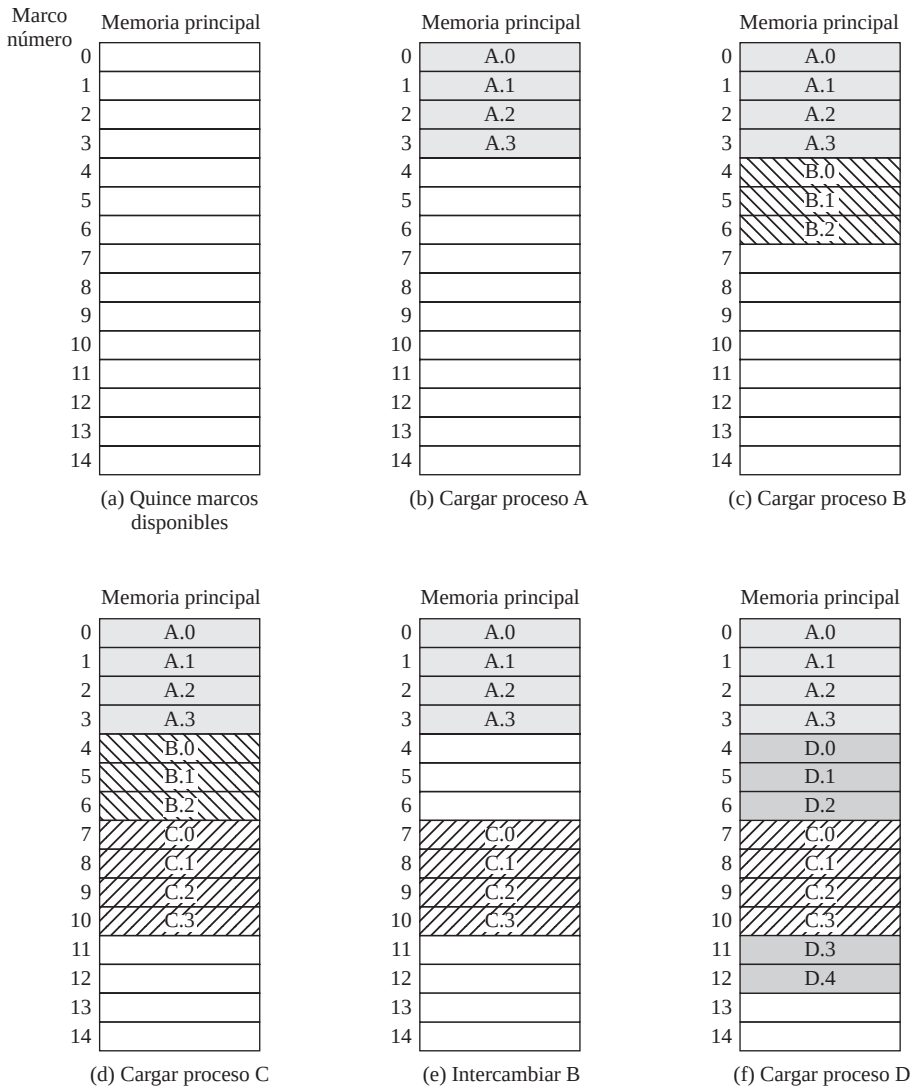


Figura 7.9. Asignación de páginas de proceso a marcos libres.

bla de páginas del proceso actual. Presentado como una dirección lógica (número de página, desplazamiento), el procesador utiliza la tabla de páginas para producir una dirección física (número de marco, desplazamiento).

Continuando con nuestro ejemplo, las cinco páginas del proceso D se cargan en los marcos 4, 5, 6, 11 y 12. La Figura 7.10 muestra las diferentes tablas de páginas en este momento. Una tabla de páginas contiene una entrada por cada página del proceso, de forma que la tabla se indexe fácilmente por el número de página (iniciando en la página 0). Cada entrada de la tabla de páginas contiene el número del marco en la memoria principal, si existe, que contiene la página correspondiente, Adicionalmente, el sistema operativo mantiene una única lista de marcos libres de todos los marcos de la memoria que se encuentran actualmente no ocupados y disponibles para las páginas.

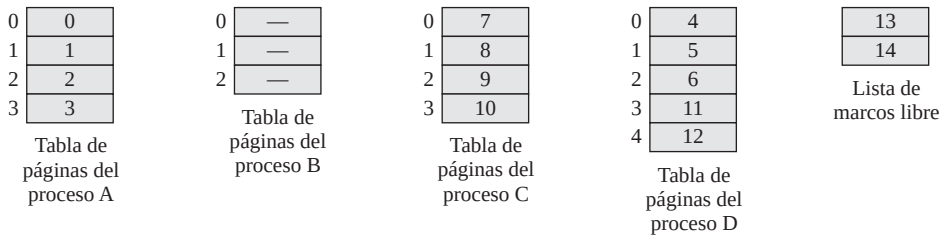


Figura 7.10. Estructuras de datos para el ejemplo de la Figura 7.9 en el instante (f).

Por tanto vemos que la paginación simple, tal y como se describe aquí, es similar al particionamiento fijo. Las diferencias son que, con la paginación, las particiones son bastante pequeñas; un programa podría ocupar más de una partición; y dichas particiones no necesitan ser contiguas.

Para hacer este esquema de paginación conveniente, el tamaño de página y por tanto el tamaño del marco debe ser una potencia de 2. Con el uso de un tamaño de página potencia de 2, es fácil demostrar que la dirección relativa, que se define con referencia al origen del programa, y la dirección lógica, expresada como un número de página y un desplazamiento, es lo mismo. Se muestra un ejemplo en la Figura 7.11. En este ejemplo, se utilizan direcciones de 16 bits, y el tamaño de la página es 1K = 1024 bytes. La dirección relativa 1502, en formato binario, es 0000010111011110. Con una página de tamaño 1K, se necesita un campo de desplazamiento de 10 bits, dejando 6 bits para el número de página. Por tanto, un programa puede estar compuesto por un máximo de 2^6=64 páginas de 1K byte cada una. Como muestra la Figura 7.11, la dirección relativa 1502 corresponde a un desplazamiento de 478 (0111011110) en la página 1 (000001), que forma el mismo número de 16 bits, 0000010111011110.

Las consecuencias de utilizar un tamaño de página que es una potencia de 2 son dobles. Primero, el esquema de direccionamiento lógico es transparente al programador, al ensamblador y al montador. Cada dirección lógica (número de página, desplazamiento) de un programa es idéntica a su dirección relativa. Segundo, es relativamente sencillo implementar una función que ejecute el hardware para llevar a cabo dinámicamente la traducción de direcciones en tiempo de ejecución. Considérese una dirección de $n+m$ bits, donde los n bits de la izquierda corresponden al número de página y los m bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11b), n = 6 y m = 10. Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de página como los n bits de la izquierda de la dirección lógica.
- Utilizar el número de página como un índice a tabla de páginas del proceso para encontrar el número de marco, k.
- La dirección física inicial del marco es $k \times 2^m$, y la dirección física del byte referenciado es dicho número más el desplazamiento. Esta dirección física no necesita calcularse; es fácilmente construida concatenando el número de marco al desplazamiento.

En el ejemplo, se parte de la dirección lógica 0000010111011110, que corresponde a la página número 1, desplazamiento 478. Supóngase que esta página reside en el marco de memoria principal 6 = número binario 000110. Por tanto, la dirección física corresponde al marco número 6, desplazamiento 478 = 0001100111011110 (Figura 7.12a).

Resumiendo, con la paginación simple, la memoria principal se divide en muchos marcos pequeños de igual tamaño. Cada proceso se divide en páginas de igual tamaño; los procesos más pequeños requieren menos páginas, los procesos mayores requieren más. Cuando un proceso se trae a la memo-

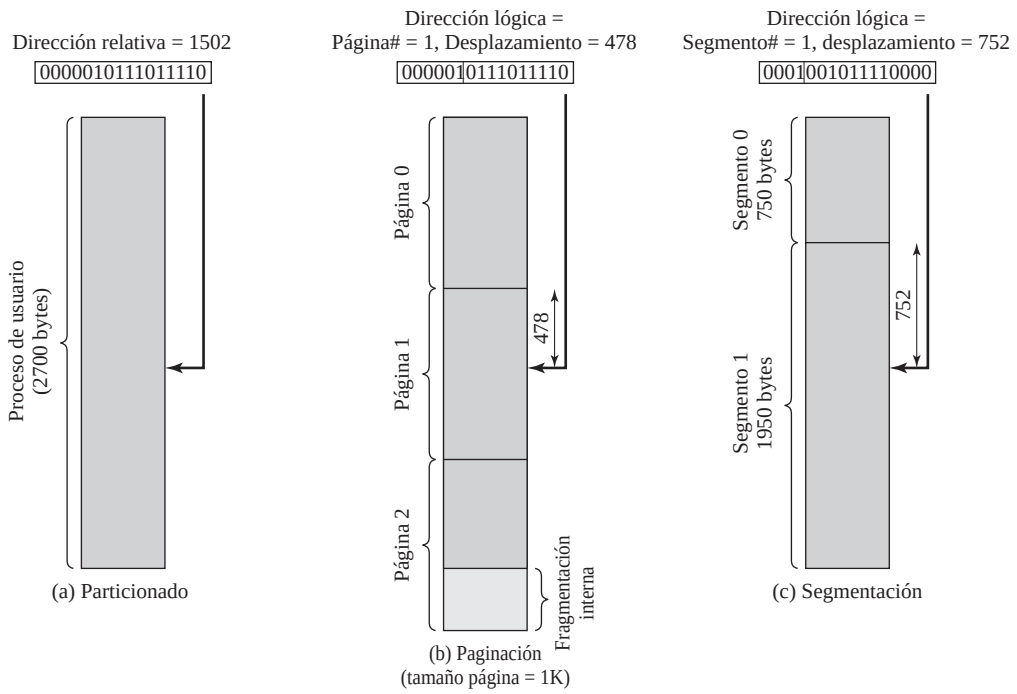


Figura 7.11. Direcciones lógicas.

ria, todas sus páginas se cargan en los marcos disponibles, y se establece una tabla de páginas. Esta técnica resuelve muchos de los problemas inherentes en el particionamiento.

7.4. SEGMENTACIÓN

Un programa de usuario se puede subdividir utilizando segmentación, en la cual el programa y sus datos asociados se dividen en un número de **segmentos**. No se requiere que todos los programas sean de la misma longitud, aunque hay una longitud máxima de segmento. Como en el caso de la paginación, una dirección lógica utilizando segmentación está compuesta por dos partes, en este caso un número de segmento y un desplazamiento.

Debido al uso de segmentos de distinto tamaño, la segmentación es similar al particionamiento dinámico. En la ausencia de un esquema de *overlays* o el uso de la memoria virtual, se necesitaría que todos los segmentos de un programa se cargaran en la memoria para su ejecución. La diferencia, comparada con el particionamiento dinámico, es que con la segmentación un programa podría ocupar más de una partición, y estas particiones no necesitan ser contiguas. La segmentación elimina la fragmentación interna pero, al igual que el particionamiento dinámico, sufre de fragmentación externa. Sin embargo, debido a que el proceso se divide en varias piezas más pequeñas, la fragmentación externa debería ser menor.

Mientras que la paginación es invisible al programador, la segmentación es normalmente visible y se proporciona como una utilidad para organizar programas y datos. Normalmente, el programador o compilador asignará programas y datos a diferentes segmentos. Para los propósitos de la programación modular, los programas o datos se pueden dividir posteriormente en múltiples segmentos. El in-

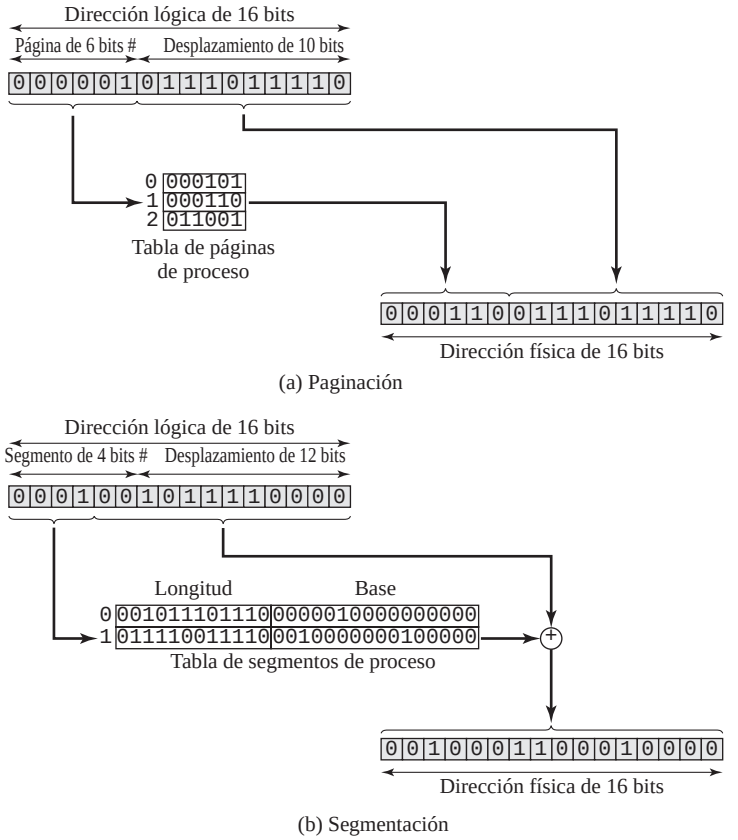


Figura 7.12. Ejemplos de traducción de direcciones lógicas a físicas.

conveniente principal de este servicio es que el programador debe ser consciente de la limitación de tamaño de segmento máximo.

Otra consecuencia de utilizar segmentos de distinto tamaño es que no hay una relación simple entre direcciones lógicas y direcciones físicas. De forma análoga a la paginación, un esquema de segmentación sencillo haría uso de una tabla de segmentos por cada proceso y una lista de bloques libre de memoria principal. Cada entrada de la tabla de segmentos tendría que proporcionar la dirección inicial de la memoria principal del correspondiente segmento. La entrada también debería proporcionar la longitud del segmento, para asegurar que no se utilizan direcciones no válidas. Cuando un proceso entra en el estado Ejecutando, la dirección de su tabla de segmentos se carga en un registro especial utilizado por el hardware de gestión de la memoria.

Considérese una dirección de $n+m$ bits, donde los n bits de la izquierda corresponden al número de segmento y los m bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11c), $n = 4$ y $m = 12$. Por tanto, el tamaño de segmento máximo es $2^{12} = 4096$. Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de segmento como los n bits de la izquierda de la dirección lógica.
- Utilizar el número de segmento como un índice a la tabla de segmentos del proceso para encontrar la dirección física inicial del segmento.

- Comparar el desplazamiento, expresado como los *m* bits de la derecha, y la longitud del segmento. Si el desplazamiento es mayor o igual que la longitud, la dirección no es válida.
- La dirección física deseada es la suma de la dirección física inicial y el desplazamiento.

En el ejemplo, se parte de la dirección lógica 0001001011110000, que corresponde al segmento número 1, desplazamiento 752. Supóngase que este segmento reside en memoria principal, comenzando en la dirección física inicial 0010000000100000. Por tanto, la dirección física es 0010000000100000 + 001011110000 = 0010001100010000 (Figura 7.12b).

Resumiendo, con la segmentación simple, un proceso se divide en un conjunto de segmentos que no tienen que ser del mismo tamaño. Cuando un proceso se trae a memoria, todos sus segmentos se cargan en regiones de memoria disponibles, y se crea la tabla de segmentos.

7.5. RESUMEN

Una de las tareas más importantes y complejas de un sistema operativo es la gestión de memoria. La gestión de memoria implica tratar la memoria principal como un recurso que debe asignarse y compartirse entre varios procesos activos. Para utilizar el procesador y las utilidades de E/S eficientemente, es deseable mantener tantos procesos en memoria principal como sea posible. Además, es deseable también liberar a los programadores de tener en cuenta las restricciones de tamaño en el desarrollo de los programas.

Las herramientas básicas de gestión de memoria son la paginación y la segmentación. Con la paginación, cada proceso se divide en un conjunto de páginas de tamaño fijo y de un tamaño relativamente pequeño. La segmentación permite el uso de piezas de tamaño variable. Es también posible combinar la segmentación y la paginación en un único esquema de gestión de memoria.

7.6. LECTURAS RECOMENDADAS

Los libros de sistema operativos recomendados en la Sección 2.9 proporcionan cobertura para la gestión de memoria.

Debido a que el sistema de particionamiento se ha suplantado por técnicas de memoria virtual, la mayoría de los libros sólo cubren superficialmente el tema. Uno de los tratamientos más completos e interesantes se encuentra en [MILE92]. Una discusión más profunda de las estrategias de particionamiento se encuentra en [KNUT97].

Los temas de enlace y carga se cubren en muchos libros de desarrollo de programas, arquitectura de computadores y sistemas operativos. Un tratamiento particularmente detallado es [BECK90]. [CLAR98] también contiene una buena discusión. Una discusión práctica en detalle de este tema, con numerosos ejemplos de sistemas operativos, es [LEVI99].

BECK90 Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1990.

CLAR98 Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

KNUT97 Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997.

LEVI99 Levine, J. *Linkers and Loaders*. New York: Elsevier Science and Technology, 1999.

MILE92 Milenkovic,M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

7.7. TÉRMINOS CLAVE, CUESTIONES DE REVISIÓN Y PROBLEMAS

TÉRMINOS CLAVE

Carga	Enlace dinámico	Particionamiento
Carga absoluta	Enlazado	Particionamiento dinámico
Carga en tiempo real dinámica	Fragmentación externa	Particionamiento fijo
Carga reubicable	Fragmentación interna	Protección
Compactación	Gestión de memoria	Reubicación
Compartición	Marca	Segmentación
Dirección física	Organización lógica	Sistema XXXX
Dirección lógica	Organización física	Tabla de páginas
Dirección relativa	Página	
Editor de enlaces	Paginación	

CUESTIONES DE REVISIÓN

7.1. ¿Qué requisitos se intenta satisfacer en gestión de la memoria?

7.2. ¿Por qué es deseable la capacidad para reubicar procesos?

7.3. ¿Por qué no es posible forzar la protección de la memoria en tiempo de compilación?

7.4. ¿Qué razones existen para permitir que dos o más procesos accedan a una misma región de la memoria?

7.5. En un esquema de particionamiento fijo, ¿cuáles son las ventajas de utilizar particiones de distinto tamaño?

7.6. ¿Cuál es la diferencia entre fragmentación interna y externa?

7.7. ¿Cuáles son las distinciones entre direcciones lógicas, relativas y físicas?

7.8. ¿Cuál es la diferencia entre una página y un marco?

7.9. ¿Cuál es la diferencia entre una página y un segmento?

PROBLEMAS

7.1. En la Sección 2.3, se listaron cinco objetivos de la gestión de la memoria y en la Sección 7.1 cinco requisitos. Discutir si cada lista incluye los aspectos tratados en la otra lista.

7.2. Considérese un esquema de particionamiento fijo con particiones de igual tamaño de 2^{16} bytes y una memoria principal total de tamaño 2^{24} bytes. Por cada proceso residente, se mantiene una tabla de procesos que incluye un puntero a una partición. ¿Cuántos bits necesita el puntero?

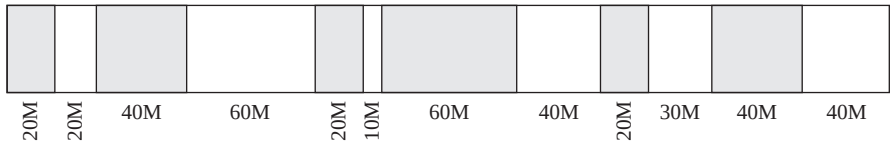
7.3. Considérese un esquema de particionamiento dinámico. Demostrar que, en media, la memoria contiene la mitad de huecos que de segmentos.

7.4. Para implementar los diferentes algoritmos de colocación discutidos para el particionamiento dinámico (Sección 7.2), se debe guardar una lista de los bloques libres de memoria.

Para cada uno de los tres métodos discutidos (mejor ajuste (*best-fit*), primer ajuste (*first-fit*) y próximo ajuste (*next-fit*)), ¿cuál es la longitud media de la búsqueda?

7.5. Otro algoritmo de colocación para el particionamiento dinámico es el de peor ajuste (*worst-fit*). En este caso, se utiliza el mayor bloque de memoria libre para un proceso. Discutir las ventajas e inconvenientes de este método comparado con el primer, próximo y mejor ajuste. ¿Cuál es la longitud media de la búsqueda para el peor ajuste?

7.6. Si se utiliza un esquema de particionamiento dinámico y en un determinado momento la configuración de memoria es la siguiente:



Las áreas sombreadas son bloques asignados; las áreas blancas son bloques libres. Las siguientes tres peticiones de memoria son de 40M, 20M y 10M. Indíquese la dirección inicial para cada uno de los tres bloques utilizando los siguientes algoritmos de colocación:

a) Primer ajuste

b) Mejor ajuste

c) Siguiente ajuste. Asúmase que el bloque añadido más recientemente se encuentra al comienzo de la memoria.

d) Peor ajuste

7.7. Un bloque de memoria de 1 Mbyte se asigna utilizando el sistema *buddy*:

a) Mostrar los resultados de la siguiente secuencia en una figura similar a la Figura 7.6: Petición 70; Petición 35; Petición 80; Respuesta A; Petición 60; Respuesta B; Respuesta D; Respuesta C.

b) Mostrar la representación de árbol binario que sigue a Respuesta B.

7.8. Considérese un sistema *buddy* en el que un determinado bloque en la asignación actual tiene la dirección 011011110000.

a) Si el bloque es de tamaño 4, ¿cuál es la dirección binaria de su bloque compañero o *buddy*?

b) Si el bloque es de tamaño 16, ¿cuál es la dirección binaria de su bloque compañero o *buddy*?

7.9. Sea $buddy_k(x)$ = dirección del bloque de tamaño 2^k, cuya dirección es x. Escribir una expresión general para el bloque compañero *buddy de* $buddy_k(x)$.

7.10. La secuencia Fibonacci se define como sigue:

$$F_o=0,\ F_1=1,\ F_{n+2}= F_{n+1} + F_n,\ n \geq 0$$

a) ¿Podría utilizarse esta secuencia para establecer un sistema *buddy*?

b) ¿Cuál sería la ventaja de este sistema respecto al sistema *buddy* binario descrito en este capítulo?

7.11. Durante el curso de ejecución de un programa, el procesador incrementará en una palabra los contenidos del registro de instrucciones (contador de programa) después de que se cargue cada instrucción, pero alterará los contenidos de dicho registro si encuentra un salto o instrucción de llamada que provoque la ejecución de otra parte del programa. Ahora considérese la Figura 7.8. Hay dos alternativas respecto a las direcciones de la instrucción:

- Mantener una dirección relativa en el registro de instrucciones y hacer la traducción de direcciones dinámica utilizando el registro de instrucciones como entrada. Cuando se encuentra un salto o una llamada, la dirección relativa generada por dicho salto o llamada se carga en el registro de instrucciones.
- Mantener una dirección absoluta en el registro de instrucciones. Cuando se encuentra un salto o una llamada, se emplea la traducción de direcciones dinámica, almacenando los resultados en el registro de instrucciones.

¿Qué opción es preferible?

7.12. Considérese un sistema de paginación sencillo con los siguientes parámetros: 2^{32} bytes de memoria física; tamaño de página de 2^{10} bytes; 2^{16} páginas de espacio de direccionamiento lógico.

a) ¿Cuántos bits hay en una dirección lógica?

b) ¿Cuántos bytes hay en un marco?

c) ¿Cuántos bits en la dirección física especifica el marco?

d) ¿Cuántas entradas hay en la tabla de páginas?

e) ¿Cuántos bits hay en cada entrada de la tabla de páginas? Asúmase que cada entrada de la tabla de páginas incluye un bit válido/inválido.

7.13. Una dirección virtual *a* en un sistema de paginación es equivalente a un par *(p,w)*, en el cual *p* es un número de pagina y *w* es un número de bytes dentro de la página. Sea *z* el número de bytes de una página. Encontrar ecuaciones algebraicas que muestren *p* y *w* como funciones de *z* y *a*.

7.14. Considérese un sistema de segmentación sencillo que tiene la siguiente tabla de segmentos:

Dirección inicial	Longitud (bytes)
660	248
1752	422
222	198
996	604

Para cada una de las siguientes direcciones lógicas, determina la dirección física o indica si se produce un fallo de segmento:

a) 0,198

b) 2,156

c) 1,530

d) 3,444

e) 0,222

7.15. Considérese una memoria en la cual se colocan segmentos contiguos S_1, S_2, …, S_n en su orden de creación, desde un extremo del dispositivo al otro, como se sugiere en la siguiente figura:

Cuando se crea el segmento contiguos S_{n+1},se coloca inmediatamente después de S_n, incluso si algunos de los segmentos S_1, S_2, …, S_n ya se hubieran borrado. Cuando el límite entre segmentos (en uso o borrados) y el hueco alcanzan el otro extremo de memoria, los segmentos en uso se compactan.

a) Mostrar que la fracción de tiempo *F* utilizada para la compactación cumple la siguiente inigualdad:

$$F \geq \frac{1-f}{1+kf}, \text{ donde } k = \frac{t}{2s} - 1$$

donde

s = longitud media de un segmento, en palabras

t = tiempo de vida medio de un segmento, en referencias a memoria

f = fracción de la memoria que no se utiliza bajo condiciones de equilibrio

Sugerencia: Encontrar la velocidad media a la que los límites cruzan la memoria y

b) Encontrar *F* para *f*=0,2, *t*=1000 y s=50.

APÉNDICE 7A CARGA Y ENLACE

El primer paso en la creación de un proceso activo es cargar un programa en memoria principal y crear una imagen del proceso (Figura 7.13). Figura 7.14 muestra un escenario típico para la mayoría de los sistemas. La aplicación está formada por varios módulos compilados o ensamblados en formato de código objeto. Éstos son enlazados para resolver todas las referencias entre los módulos. Al mismo tiempo, se resuelven las referencias a rutinas de biblioteca. Las rutinas de biblioteca pueden incorporarse al programa o hacerle referencia como código compartido que el sistema operativo proporciona en tiempo de ejecución. En este apéndice, se resumen las características clave de los enlazadores y cargadores. Por motivos de claridad en la presentación, se comienza con una descripción de la tarea de carga cuando sólo se tiene un módulo de programa; en este caso no se requiere enlace.

CARGA

En la Figura 7.14, el cargador coloca el módulo de carga en la memoria principal, comenzando en la ubicación *x*. En la carga del programa, se debe satisfacer el requisito de direccionamiento mostrado en la Figura 7.1. En general, se pueden seguir tres técnicas diferentes:

- Carga absoluta
- Carga reubicable
- Carga dinámica en tiempo real

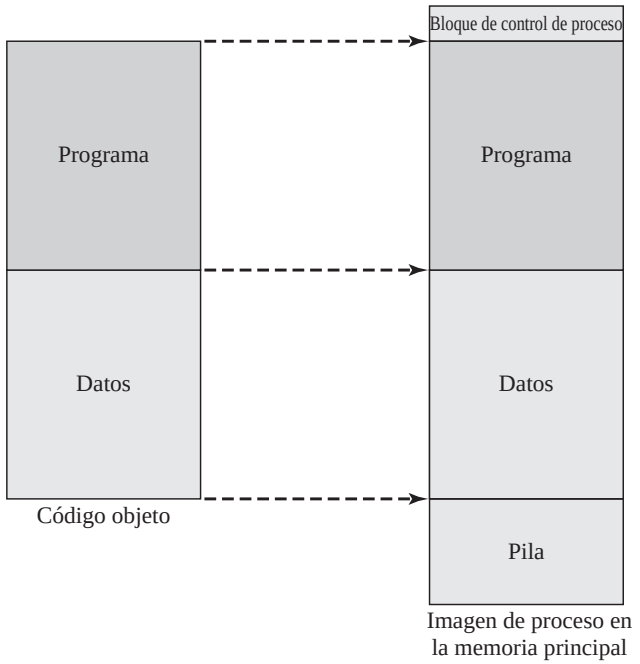


Figura 7.13. La función de carga.

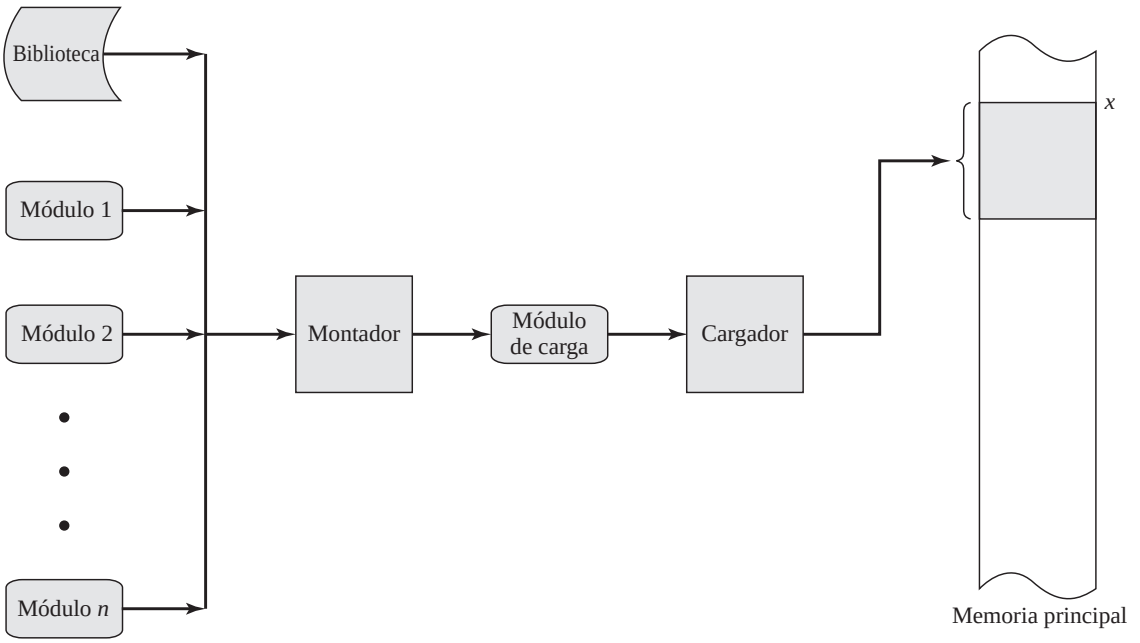


Figura 7.14. Un escenario de carga.

Carga absoluta

Un cargador absoluto requiere que un módulo de carga dado debe cargarse siempre en la misma ubicación de la memoria principal. Por tanto, en el módulo de carga presentado al cargador, todas las referencias a direcciones deben ser direcciones de memoria principal específicas o absolutas. Por ejemplo, si en la Figura 7.14 *x* es la ubicación 1024, entonces la primera palabra de un módulo de carga destinado para dicha región de memoria, tiene la dirección 1024.

La asignación de valores de direcciones específicas a referencias de programa dentro de un programa lo puede hacer el programador o se hacen en tiempo de compilación o ensamblado (Tabla 7.2a). La primera opción tiene varias desventajas. Primero, cada programador debe conocer la estrategia de asignación para colocar los módulos en memoria principal. Segundo, si se hace cualquier modificación al programa que implique inserciones o borrados en el cuerpo del módulo, entonces todas las direcciones deben alterarse. Por tanto, es preferible permitir que las referencias de memoria dentro de los programas se expresen simbólicamente, y entonces resolver dichas referencias simbólicas en tiempo de compilación o ensamblado. Esto queda reflejado en la Figura 7.15. Cada referencia a una instrucción o elemento de datos se representa inicialmente como un símbolo. A la hora de preparar el módulo para su entrada a un cargador absoluto, el ensamblador o compilador convertirá todas estas referencias a direcciones específicas (en este ejemplo, el módulo se carga en la dirección inicial de 1024), tal como se muestra en la Figura 7.15b.

Tabla 7.2. Asociación de direcciones.

(a) Cargador

Tiempo de asociación	Función
Tiempo de programación	El programador especifica directamente en el propio programa todas las direcciones físicas reales.
Tiempo de compilación o ensamblado	El programa contiene referencias a direcciones simbólicas y el compilador o ensamblador las convierte a direcciones físicas reales.
Tiempo de carga	El compilador o ensamblador produce direcciones relativas. El cargador las traduce a direcciones absolutas cuando se carga el programa.
Tiempo de ejecución	El programa cargador retiene direcciones relativas. El hardware del procesador las convierte dinámicamente a direcciones absolutas.

(b) Montador

Tiempo de montaje	Función
Tiempo de programación	No se permiten referencias a programas o datos externos. El programador debe colocar en el programa el código fuente de todos los subprogramas que invoque.
Tiempo de compilación o ensamblado	El ensamblador debe traer el código fuente de cada subrutina que se referencia y ensamblarlo como una unidad.
Creación de módulo de carga	Todos los módulos objeto se han ensamblado utilizando direcciones relativas. Estos módulos se enlazan juntos y todas las referencias se restablecen en relación al origen del módulo de carga final.
Tiempo de carga	Las referencias externas no se resuelven hasta que el módulo de carga se carga en memoria principal. En ese momento, los módulos con enlace dinámico referenciados se adjuntan al módulo de carga y el paquete completo se carga en memoria principal o virtual.
Tiempo de ejecución	Las referencias externas no se resuelven hasta que el procesador ejecuta la llamada externa. En ese momento, el proceso se interrumpe y el módulo deseado se enlaza al programa que lo invoca.

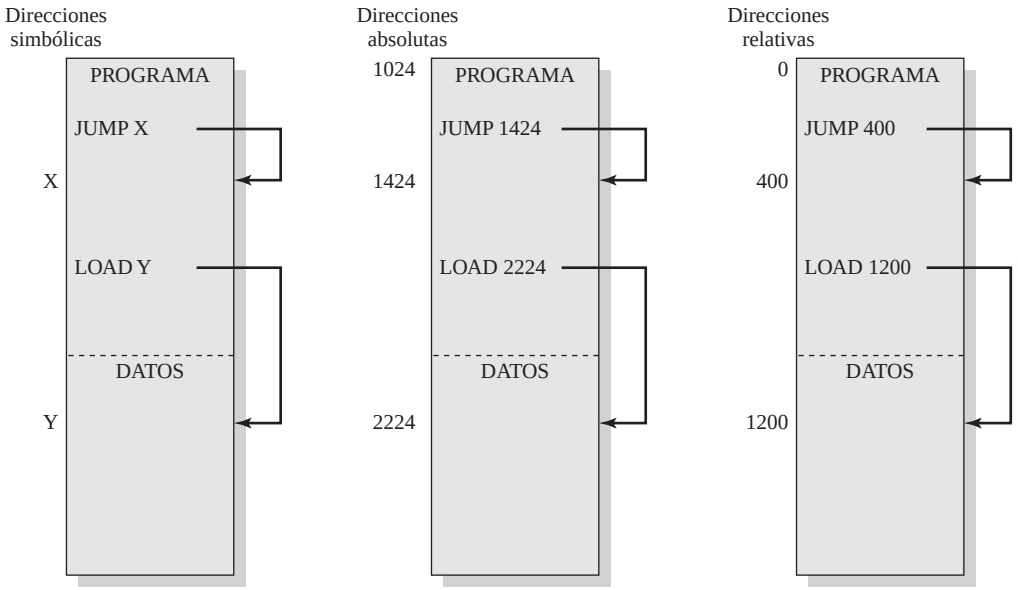


Figura 7.15. Módulos de carga absolutos y reubicables.

Carga reubicable

La desventaja de enlazar referencias de memoria a direcciones específicas antes de la carga es que el módulo de carga resultante sólo se puede colocar en una región específica de memoria principal. Sin embargo, cuando muchos programas comparten memoria principal, podría no ser deseable decidir al principio en qué región de la memoria se debe cargar un módulo particular. Es mejor tomar esta decisión en tiempo de carga. Por tanto, necesitamos un módulo de carga que pueda ubicarse en cualquier lugar de la memoria principal.

Para satisfacer este nuevo requisito, el ensamblador o compilador no produce direcciones de memoria reales (direcciones absolutas), sino direcciones relativas a algún punto conocido, tal como el inicio del programa. Esta técnica se muestra en la Figura 7.15c. El comienzo del módulo de carga se asigna a la dirección relativa 0, y el resto de las referencias de memoria dentro del módulo se expresan relativas al comienzo del módulo.

Con todas las referencias de la memoria expresadas en formato relativo, colocar el módulo en el lugar adecuado se convierte en una tarea simple para el cargador. Si el módulo se carga al comienzo de la ubicación *x*, entonces el cargador debe simplemente añadir *x* a cada referencia de la memoria cuando carga el módulo en memoria. Para asistir en esta tarea, el módulo cargado debe incluir información que dice el cargador donde están las referencias de memoria y cómo se van a interpretar (normalmente relativo al origen del programa, pero también es posible relativo a algún otro punto del programa, tal como la ubicación actual). El compilador o ensamblador prepara este conjunto de información, lo que se denomina normalmente diccionario de reubicación.

Carga dinámica en tiempo real

Los cargadores reubicables son comunes y proporcionan beneficios obvios si se comparan con los cargadores absolutos. Sin embargo, en un entorno de multiprogramación, incluso en uno que no de-

penda de la memoria virtual, el esquema de carga reubicable no es adecuado. A lo largo del libro, nos hemos referido a la necesidad de traer y quitar imágenes de procesos de la memoria principal a fin de maximizar la utilización del procesador. Para maximizar la utilización de la memoria principal, sería importante poder intercambiar las imágenes de los procesos en diferentes localizaciones en diferentes momentos. Por tanto, un programa, una vez cargado, puede intercambiarse a disco y a memoria en diferentes ubicaciones. Esto sería imposible si las referencias de la memoria se limitan a direcciones absolutas en tiempo de carga inicial.

La alternativa es posponer el cálculo de una dirección absoluta hasta que se necesite realmente en tiempo de ejecución. Para este propósito, el módulo de carga se carga en la memoria principal con todas las referencias de la memoria en formato relativo (Figura 7.15c). Hasta que una instrucción no se ejecuta realmente, no se calcula la dirección absoluta. Para asegurar que esta función no degrada el rendimiento, la realiza el hardware de procesador especial en lugar de llevarse a cabo por software. El hardware se describe en la Sección 7.2.

El cálculo dinámico de direcciones proporciona una flexibilidad total. Un programa se carga en cualquier región de la memoria principal. A continuación, la ejecución del programa se puede interrumpir y el programa se puede intercambiar entre disco y memoria, para posteriormente intercambiarse en una localización diferente.

ENLACE

La función de un montador es tomar como entrada una colección de módulos objeto y producir un módulo de carga, formado por un conjunto integrado de programa y módulos de datos, que se pasará al cargador. En cada módulo objeto, podría haber referencias a direcciones de otros módulos. Cada una de estas referencias sólo se puede expresar simbólicamente en un módulo objeto no enlazado. El montador crea un único módulo de carga que se une de forma contigua a todos los módulos objeto. Cada referencia entre módulos debe cambiarse: una dirección simbólica debe convertirse en una referencia a una ubicación dentro del módulo de carga. Por ejemplo, el módulo A en la Figura 7.16a contiene una invocación a un procedimiento del módulo B. Cuando estos módulos se combinan en el módulo de carga, esta referencia simbólica al módulo B se cambia por una referencia específica a la localización del punto de entrada de B dentro del módulo de carga.

Editor de enlace

La naturaleza de este enlace de direcciones dependerá del tipo de módulo de carga que se cree y cuando se lleve a cabo el proceso de enlace (Tabla 7.2b). Si se desea un módulo de carga reubicable, como suele ser lo habitual, el enlace se hace normalmente de la siguiente forma. Cada módulo objeto compilado o ensamblado se crea con referencias relativas al comienzo del módulo objeto. Todos estos módulos se colocan juntos en un único módulo de carga reubicable con todas las referencias relativas al origen del módulo de carga. Este módulo se puede utilizar como entrada para la carga reubicable o carga dinámica en tiempo de ejecución.

Un montador que produce un módulo de carga reubicable se denomina frecuentemente editor de enlace. La Figura 7.16 ilustra la función del editor de enlace.

Montador dinámico

Al igual que con la carga, también es posible posponer algunas funciones relativas al enlace. El término *enlace dinámico* se utiliza para denominar la práctica de posponer el enlace de algunos módulos

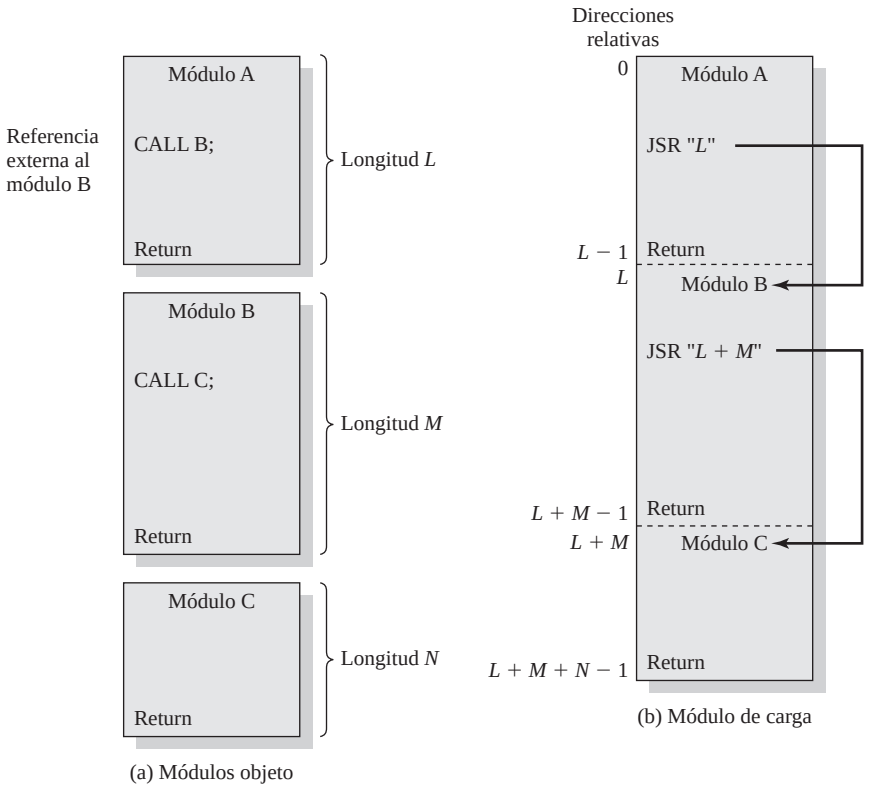


Figura 7.16. La función de montaje.

externos hasta después de que el módulo de carga se cree. Por tanto, el módulo de carga contiene referencias sin resolver a otros programas. Estas referencias se pueden resolver o bien en tiempo de carga o bien en tiempo de ejecución.

Para el **enlace dinámico en tiempo de carga**, se deben seguir los siguientes pasos. El módulo de carga (módulo de aplicación) debe llevarse a memoria para cargarlo. Cualquier referencia a un módulo externo (módulo destino) provoca que el cargador encuentre al módulo destino, lo cargue y altere la referencia a una dirección relativa a la memoria desde el comienzo del módulo de aplicación. Hay varias ventajas de esta técnica frente al enlace estático:

- Se facilita incorporar versiones modificadas o actualizadas del módulo destino, el cual puede ser una utilidad del sistema operativo o algunas otras rutinas de propósito general. Con el enlace estático, un cambio a ese módulo requeriría el reenlace del módulo de aplicación completo. No sólo es ineficiente, sino que puede ser imposible en algunas circunstancias. Por ejemplo, en el campo de los ordenadores personales, la mayoría del software comercial es entregado en el formato del módulo de carga; no se entregan las versiones fuente y objeto.
- Tener el código destino en un fichero con enlace dinámico facilita el camino para compartir código de forma automática. El sistema operativo puede reconocer que más de una aplicación utiliza el mismo código destino porque carga y enlaza dicho código. Puede utilizar esta información para cargar una única copia del código destino y enlazarlo a ambas aplicaciones, en lugar de tener que cargar una copia para cada aplicación.

- Facilita a los desarrolladores de software independientes extender la funcionalidad de un sistema operativo ampliamente utilizado, tal como Linux. Un desarrollador puede implementar una nueva función que puede ser útil a una variedad de aplicaciones y empaquetarla como un módulo de enlace dinámico.

Con **enlace dinámico en tiempo de ejecución**, algunos de los enlaces son pospuestos hasta el tiempo de ejecución. Las referencias externas a los módulos destino quedan en el programa cargado. Cuando se realiza una llamada a un módulo ausente, el sistema operativo localiza el módulo, lo carga y lo enlaza al módulo llamante.

Se ha visto que la carga dinámica permite que se pueda mover un módulo de carga entero; sin embargo, la estructura del módulo es estática, permaneciendo sin cambios durante la ejecución del proceso y de una ejecución a otra. Sin embargo, en algunos casos, no es posible determinar antes de la ejecución qué módulos objeto se necesitarán. Esta situación es tipificada por las aplicaciones de procesamiento de transacciones, como las de un sistema de reserva de vuelos o una aplicación bancaria. La naturaleza de la transacción especifica qué módulos de programa se requieren, y éstos son cargados y enlazados con el programa principal. La ventaja del uso de un montador dinámico es que no es necesario asignar memoria a unidades del programa a menos que dichas unidades se referencien. Esta capacidad se utiliza para dar soporte a sistemas de segmentación.

Una mejora adicional es posible: una aplicación no necesita conocer los nombres de todos los módulos o puntos de entrada que pueden llamarse. Por ejemplo, puede escribirse un programa de dibujo para trabajar con una gran variedad de trazadores, cada uno de los cuales se gestiona por un controlador diferente. La aplicación puede aprender el nombre del trazador que está actualmente instalado en el sistema por otro proceso o buscando en un fichero de configuración. Esto permite al usuario de la aplicación instalar un nuevo trazador que no exista en el tiempo en que la aplicación se escribió.

CAPÍTULO 8

Memoria virtual

En el Capítulo 7 se vieron los conceptos de paginación y segmentación y se analizaron sus limitaciones. Ahora vamos a entrar a discutir el concepto de memoria virtual. Un análisis de este concepto es complicado por el hecho de que la gestión de la memoria es una interacción compleja entre el hardware del procesador y el sistema operativo. Nos centraremos primero en los aspectos hardware de la memoria virtual, observando el uso de la paginación, segmentación, y combinación de paginación y segmentación. Después veremos los aspectos relacionados con el diseño de los servicios de la memoria virtual en el sistema operativo.

8.1. HARDWARE Y ESTRUCTURAS DE CONTROL

Comparando la paginación sencilla y la segmentación sencilla, por un lado, tenemos una distinción entre particionamiento estático y dinámico, y por otro, tenemos los fundamentos de comienzo de la gestión de la memoria. Las dos características de la paginación y la segmentación que son la clave de este comienzo son:

1. Todas las referencias a la memoria dentro un proceso se realizan a direcciones lógicas, que se traducen dinámicamente en direcciones físicas durante la ejecución. Esto significa que un proceso puede ser llevado y traído a memoria de forma que ocupe diferentes regiones de la memoria principal en distintos instantes de tiempo durante su ejecución.
2. Un proceso puede dividirse en varias porciones (páginas o segmentos) y estas porciones no tienen que estar localizadas en la memoria de forma contigua durante la ejecución. La combinación de la traducción de direcciones dinámicas en ejecución y el uso de una tabla de páginas o segmentos lo permite.

Ahora veamos cómo comenzar con la memoria dinámica. *Si las dos características anteriores se dan, entonces es necesario que todas las páginas o todos los segmentos de un proceso se encuentren en la memoria principal durante la ejecución.* Si la porción (segmento o página) en la que se encuentra la siguiente instrucción a buscar está y si la porción donde se encuentra la siguiente dirección de datos que se va a acceder también está, entonces al menos la siguiente instrucción se podrá ejecutar.

Consideremos ahora cómo se puede realizar esto. De momento, vamos a hablar en términos generales, y usaremos el término *porción* para referirnos o bien a una página o un segmento, dependiendo si estamos empleando paginación o segmentación. Supongamos que se tiene que traer un nuevo proceso de memoria. El sistema operativo comienza trayendo únicamente una o dos porciones, que incluye la porción inicial del programa y la porción inicial de datos sobre la cual acceden las primeras instrucciones acceden. Esta parte del proceso que se encuentra realmente en la memoria principal para, cualquier instante de tiempo, se denomina **conjunto residente** del proceso. Cuando el proceso está ejecutándose, las cosas ocurren de forma suave mientras que todas las referencias a la memoria se encuentren dentro del conjunto residente. Usando una tabla de segmentos o páginas, el procesador siempre es capaz de determinar si esto es así o no. Si el procesador encuentra una dirección lógica que no se encuentra en la memoria principal, generará una interrupción indicando un fallo de acceso a la memoria. El sistema operativo coloca al proceso interrumpido en un estado de bloqueado y toma el control. Para que la ejecución de este proceso pueda reanudarse más adelante, el sistema operativo necesita traer a la memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo de acceso. Con este fin, el sistema operativo realiza una petición de E/S, una lectura a disco. Después de realizar la petición de E/S, el sistema operativo puede activar otro proceso que se ejecute mientras el disco realiza la operación de E/S. Una vez que la porción solicitada se ha traído a

la memoria principal, una nueva interrupción de E/S se lanza, dando control de nuevo al sistema operativo, que coloca al proceso afectado de nuevo en el estado Listo.

Al lector se le puede ocurrir cuestionar la eficiencia de esta maniobra, en la cual a un proceso que se puede estar ejecutando resulta necesario interrumpirlo sin otro motivo que el hecho de que no se ha llegado a cargar todas las porciones necesarias de dicho proceso. De momento, vamos a posponer esta cuestión con la garantía de que la eficiencia es verdaderamente posible. En su lugar, vamos a ponderar las implicaciones de nuestra nueva estrategia. Existen dos implicaciones, la segunda más sorprendente que la primera, y ambas dirigidas a mejorar la utilización del sistema:

1. **Pueden mantenerse un mayor número de procesos en memoria principal**. Debido a que sólo vamos a cargar algunas de las porciones de los procesos a ejecutar, existe espacio para más procesos. Esto nos lleva a una utilización más eficiente del procesador porque es más probable que haya al menos uno o más de los numerosos procesos que se encuentre en el estado Listo, en un instante de tiempo concreto.
2. **Un proceso puede ser mayor que toda la memoria principal**. Se puede superar una de las restricciones fundamentales de la programación. Sin el esquema que hemos estado discutiendo, un programador debe estar realmente atento a cuánta memoria está disponible. Si el programa que está escribiendo es demasiado grande, el programador debe buscar el modo de estructurar el programa en fragmentos que pueden cargarse de forma separada con un tipo de estrategia de superposición (*overlay*). Con la memoria virtual basada en paginación o segmentación, este trabajo se delega al sistema operativo y al hardware. En lo que concierne al programador, él está trabajando con una memoria enorme, con un tamaño asociado al almacenamiento en disco. El sistema operativo automáticamente carga porciones de un proceso en la memoria principal cuando éstas se necesitan.

Debido a que un proceso ejecuta sólo en la memoria principal, esta memoria se denomina **memoria real**. Pero el programador o el usuario perciben una memoria potencialmente mucho más grande —la cual se encuentra localizada en disco. Esta última se denomina **memoria virtual**. La memoria virtual permite una multiprogramación muy efectiva que libera al usuario de las restricciones excesivamente fuertes de la memoria principal. La Tabla 8.1 recoge las características de la paginación y la segmentación, con y sin el uso de la memoria virtual.

PROXIMIDAD Y MEMORIA VIRTUAL

Los beneficios de la memoria virtual son atractivos, ¿pero el esquema es verdaderamente práctico? En su momento, hubo un importante debate sobre este punto, pero la experiencia de numerosos sistemas operativos ha demostrado, más allá de toda duda, que la memoria virtual realmente funciona. La memoria virtual, basada en paginación o paginación más segmentación, se ha convertido, en la actualidad, en una componente esencial de todos los sistemas operativos contemporáneos.

Para entender cuál es el aspecto clave, y por qué la memoria virtual era la causa de dicho debate, examinemos de nuevo las tareas del sistema operativo relacionadas con la memoria virtual. Se va a considerar un proceso de gran tamaño, consistente en un programa largo más un gran número de vectores de datos. A lo largo de un corto periodo de tiempo, la ejecución se puede acotar a una pequeña sección del programa (por ejemplo, una subrutina) y el acceso a uno o dos vectores de datos únicamente. Si es así, sería verdaderamente un desperdicio cargar docenas de porciones de dicho proceso cuando sólo unas pocas porciones se usarán antes de que el programa se suspenda o se mande a zona de intercambio o *swap*. Se puede hacer un mejor uso de la memoria cargando únicamente unas pocas porciones. Entonces, si el programa salta a una destrucción o hace referencia a un dato que se en-

Tabla 8.1. Características de la paginación y la segmentación.

Paginación sencilla	Paginación con memoria virtual	Segmentación sencilla	Segmentación con memoria virtual
Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal no particionada	Memoria principal no particionada
Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte el programador)	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte el programador)
Fragmentación interna dentro de los marcos	Fragmentación interna dentro de los marcos	Sin fragmentación interna	Sin fragmentación interna
Sin fragmentación externa	Sin fragmentación externa	Fragmentación externa	Fragmentación externa
El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento
El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de huecos en la memoria principal	El sistema operativo debe mantener una lista de huecos en la memoria principal
El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas
Todas las páginas del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (*overlays*)	No se necesita mantener todas las páginas del proceso en los marcos de la memoria principal para que el proceso se ejecute. Las páginas se pueden leer bajo demanda	Todos los segmentos del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (*overlays*)	No se necesitan mantener todos los segmentos del proceso en la memoria principal para que el proceso se ejecute. Los segmentos se pueden leer bajo demanda
	La lectura de una página a memoria principal puede requerir la escritura de una página a disco		La lectura de un segmento a memoria principal puede requerir la escritura de uno o más segmentos a disco

cuentra en una porción de memoria que no está en la memoria principal, entonces se dispara un fallo. Éste indica al sistema operativo que debe conseguir la porción deseada.

Así, en cualquier momento, sólo unas pocas porciones de cada proceso se encuentran en memoria, y por tanto se pueden mantener más procesos alojados en la misma. Además, se ahorra tiempo porque las porciones del proceso no usadas no se expulsarán de la memoria a *swap* y de *swap* a la memoria. Sin embargo, el sistema operativo debe ser inteligente a la hora de manejar este esquema. En estado estable, prácticamente toda la memoria principal se encontrará ocupada con porciones de procesos, de forma que el procesador y el sistema operativo tengan acceso directo al mayor número posible de procesos. Así, cuando el sistema operativo traiga una porción a la memoria, debe expulsar otra. Si elimina una porción justo antes de que vaya a ser utilizada, deberá recuperar dicha porción de nuevo casi de forma inmediata. Un abuso de esto lleva a una condición denominada **trasiego (*thrashing*)**: el sistema consume la mayor parte del tiempo enviando y trayendo porciones de *swap* en lugar de ejecutar instrucciones. Evitar el trasiego fue una de las áreas de investigación principales en la época de los años 70 que condujo una gran variedad de algoritmos complejos pero muy efectivos. En esencia, el sistema operativo trata de adivinar, en base a la historia reciente, qué porciones son menos probables de ser utilizadas en un futuro cercano.

Este razonamiento se basa en la creencia del **principio de proximidad**, que se presentó en el Capítulo 1 (*véase* especialmente el Apéndice 1A). Para resumir, el principio de proximidad indica que las referencias al programa y a los datos dentro de un proceso tienden a agruparse. Por tanto, se resume que sólo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto. También, es posible hacer suposiciones inteligentes sobre cuáles son las porciones del proceso que se necesitarán en un futuro próximo, para evitar este trasiego.

Una forma de confirmar el principio de proximidad es observar el rendimiento de los procesos en un entorno de memoria virtual. En la Figura 8.1 se muestra un famoso diagrama que ilustra de forma clara los principios de proximidad [HATF 72]. Nótese que, durante el tiempo de vida de un proceso las referencias se encuentran acotadas a un subconjunto de sus páginas.

Así pues, vemos que el principio de proximidad sugiere que el esquema de memoria virtual debe funcionar. Para que la memoria virtual resulte práctica y efectiva, se necesitan dos ingredientes. Primero, debe existir un soporte hardware para el esquema de paginación y/o segmentación. Segundo, el sistema operativo debe incluir código para gestionar el movimiento de páginas y/o segmentos entre la memoria secundaria y la memoria principal. En esta sección, examinaremos los aspectos hardware y veremos cuáles son las estructuras de control necesarias, que se crearán y mantendrán por parte del sistema operativo pero que son usadas por el hardware de gestión de la memoria. Se examinarán los aspectos correspondientes al sistema operativo en la siguiente sección.

PAGINACIÓN

El término *memoria virtual* se asocia habitualmente con sistemas que emplean paginación, a pesar de que la memoria virtual basada en segmentación también se utiliza y será tratada más adelante. El uso de paginación para conseguir memoria virtual fue utilizado por primera vez en el computador Atlas [KILB62] y pronto se convirtió en una estrategia usada en general de forma comercial.

En la presentación de la paginación sencilla, indicamos que cada proceso dispone de su propia tabla de páginas, y que todas las páginas se encuentran localizadas en la memoria principal. Cada entrada en la tabla de páginas consiste en un número de marco de la correspondiente página en la memoria principal. Para la memoria virtual basada en el esquema de paginación también se necesita una tabla de páginas. De nuevo, normalmente se asocia una única tabla de páginas a cada proceso. En este caso, sin embargo, las entradas de la tabla de páginas son más complejas (Figura 8.2a). Debido a que

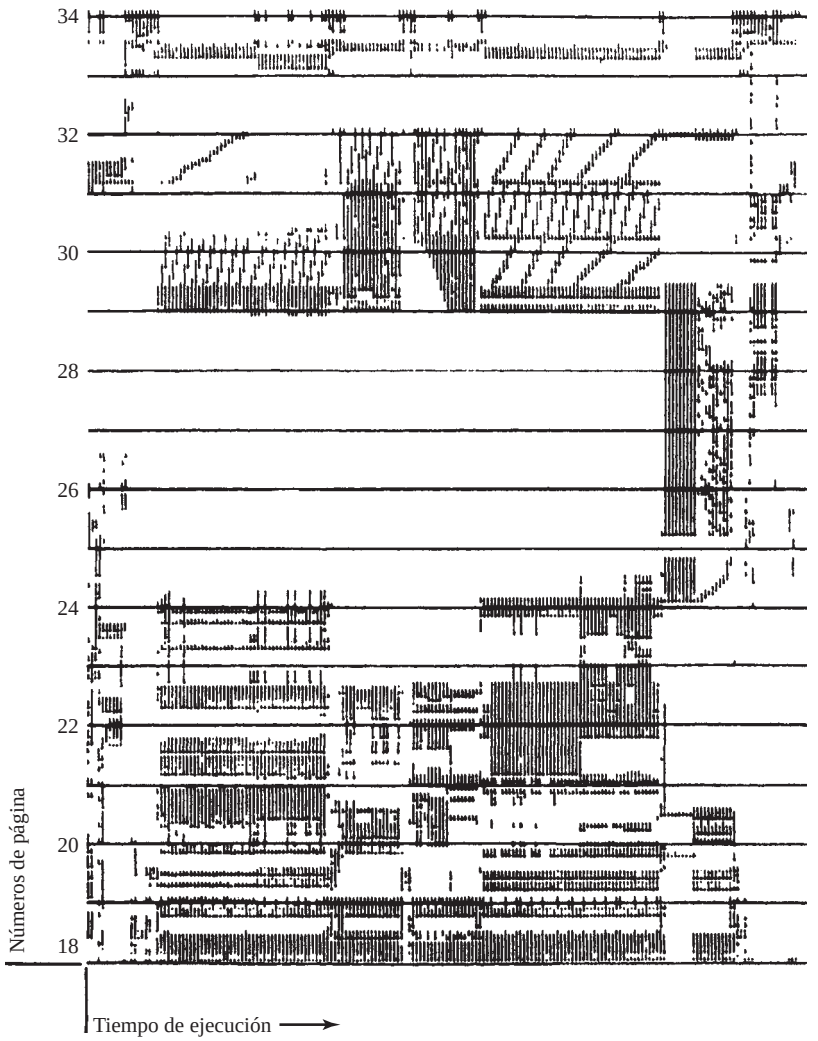


Figura 8.1. Comportamiento de la paginación.

sólo algunas de las páginas de proceso se encuentran en la memoria principal, se necesita que cada entrada de la tabla de páginas indique si la correspondiente página está presente (P) en memoria principal o no. Si el bit indica que la página está en memoria, la entrada también debe indicar el número de marco de dicha página.

La entrada de la tabla de páginas incluye un bit de modificado (M), que indica si los contenidos de la correspondiente página han sido alterados desde que la página se cargó por última vez en la memoria principal. Si no había ningún cambio, no es necesario escribir la página cuando llegue el momento de reemplazarla por otra página en el marco de página que actualmente ocupa. Pueden existir también otros bits de control en estas entradas. Por ejemplo, si la protección y compartición se gestiona a nivel de página, se necesitarán también los bits para este propósito.

Estructura de la tabla de páginas. El mecanismo básico de lectura de una palabra de la memoria implica la traducción de la dirección virtual, o lógica, consistente en un número de página y un des-

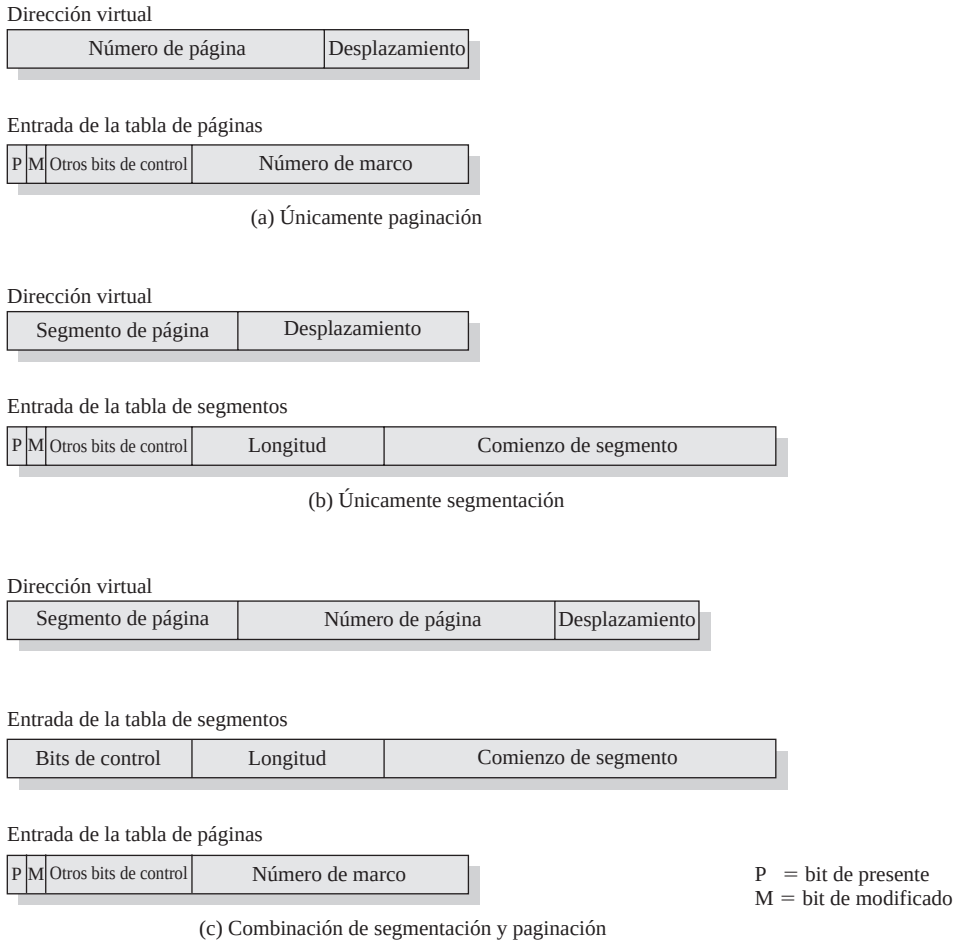


Figura 8.2. Formatos típicos de gestión de memoria.

plazamiento, a la dirección física, consistente en un número de marco y un desplazamiento, usando para ello la tabla de páginas. Debido a que la tabla de páginas es de longitud variable dependiendo del tamaño del proceso, no podemos suponer que se encuentra almacenada en los registros. En lugar de eso, debe encontrarse en la memoria principal para poder ser accedida. La Figura 8.3 sugiere una implementación hardware. Cuando un proceso en particular se encuentra ejecutando, un registro contiene la dirección de comienzo de la tabla de páginas para dicho proceso. El número de página de la dirección virtual se utiliza para indexar esa tabla y buscar el correspondiente marco de página. Éste, combinado con la parte de desplazamiento de la dirección virtual genera la dirección real deseada. Normalmente, el campo correspondiente al número de página es mayor que el campo correspondiente al número de marco de página ($n > m$).

En la mayoría de sistemas, existe una única tabla de página por proceso. Pero cada proceso puede ocupar una gran cantidad de memoria virtual. Por ejemplo, en la arquitectura VAX, cada proceso puede tener hasta $2^{31} = 2$ Gbytes de memoria virtual. Usando páginas de $2^9 = 512$ bytes, que representa un total de 2^{22} entradas de tabla de página *por cada proceso*. Evidentemente, la cantidad de memoria demandada por las tablas de página únicamente puede ser inaceptablemente grande. Para resolver este problema, la mayoría de esquemas de memoria virtual almacena las ta-

blas de páginas también en la memoria virtual, en lugar de en la memoria real. Esto representa que las tablas de páginas están sujetas a paginación igual que cualquier otra página. Cuando un proceso está en ejecución, al menos parte de su tabla de páginas debe encontrarse en memoria, incluyendo la entrada de tabla de páginas de la página actualmente en ejecución. Algunos procesadores utilizan un esquema de dos niveles para organizar las tablas de páginas de gran tamaño. En este esquema, existe un directorio de páginas, en el cual cada entrada apuntaba a una tabla de páginas. De esta forma, si la extensión del directorio de páginas es X, y si la longitud máxima de una tabla de páginas es Y, entonces un proceso consistirá en hasta $X \geq Y$ páginas. Normalmente, la longitud máxima de la tabla de páginas se restringe para que sea igual a una página. Por ejemplo, el procesador Pentium utiliza esta estrategia.

La Figura 8.4 muestra un ejemplo de un esquema típico de dos niveles que usa 32 bits para la dirección. Asumimos un direccionamiento a nivel de byte y páginas de 4 Kbytes (2^{12}), por tanto el espacio de direcciones virtuales de 4 Gbytes (2^{32}) se compone de 2^{20} páginas. Si cada una de estas páginas se referencia por medio de una entrada la tabla de páginas (ETP) de 4-bytes, podemos crear una tabla de página de usuario con 2^{20} la ETP que requiere 4 Mbytes (2^{22} bytes). Esta enorme tabla de páginas de usuario, que ocupa 2^{10} páginas, puede mantenerse en memoria virtual y hacerse referencia desde una tabla de páginas raíz con 2^{10} PTE que ocuparía 4 Kbytes (2^{12}) de memoria principal. La Figura 8. 5 muestra los pasos relacionados con la traducción de direcciones para este esquema. La página raíz siempre se mantiene en la memoria principal. Los primeros 10 bits de la dirección virtual se pueden usar para indexar en la tabla de páginas raíz para encontrar la ETP para la página en la que está la tabla de páginas de usuario. Si la página no está en la memoria principal, se produce un fallo de página. Si la página está en la memoria principal, los siguientes 10 bits de la dirección virtual se usan para indexar la tabla de páginas de usuario para encontrar la ETP de la página a la cual se hace referencia desde la dirección virtual original.

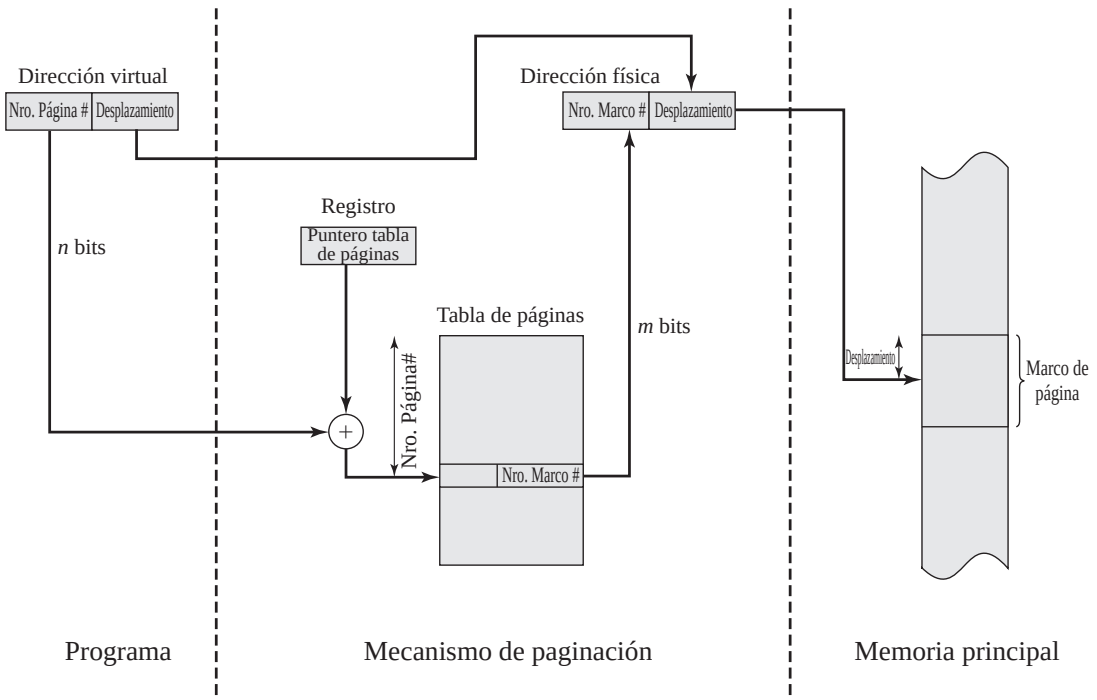


Figura 8.3. Traducción de direcciones en un sistema con paginación.

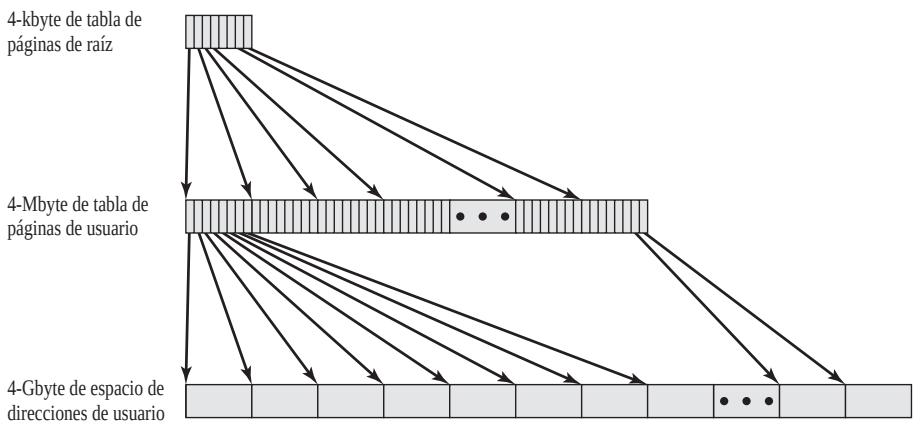


Figura 8.4. Una tabla de páginas jerárquica de dos niveles.

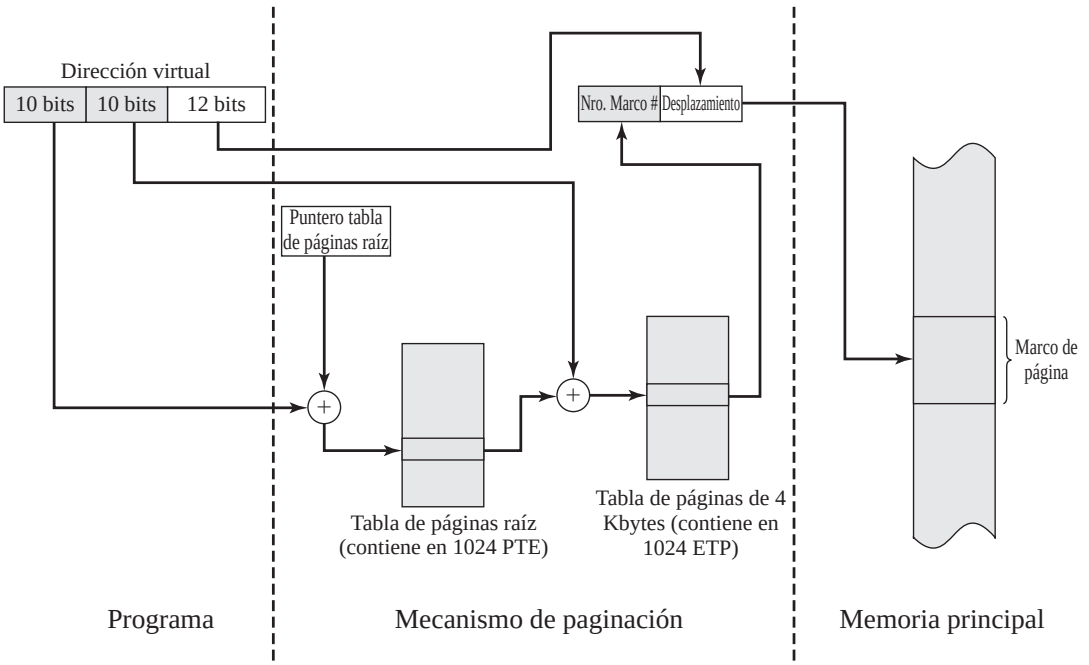


Figura 8.5. Traducción de direcciones en un sistema de paginación de dos niveles.

Tabla de páginas invertida. Una desventaja del tipo de tablas de páginas que hemos visto es que su tamaño es proporcional al espacio de direcciones virtuales.

Una estrategia alternativa al uso de tablas de páginas de uno o varios niveles es el uso de la estructura de **tabla de páginas invertida**. Variaciones de esta estrategia se han usado en arquitecturas como PowerPC, UltraSPARC, e IA-64. La implementación del sistema operativo Mach sobre RT-PC también la usa.

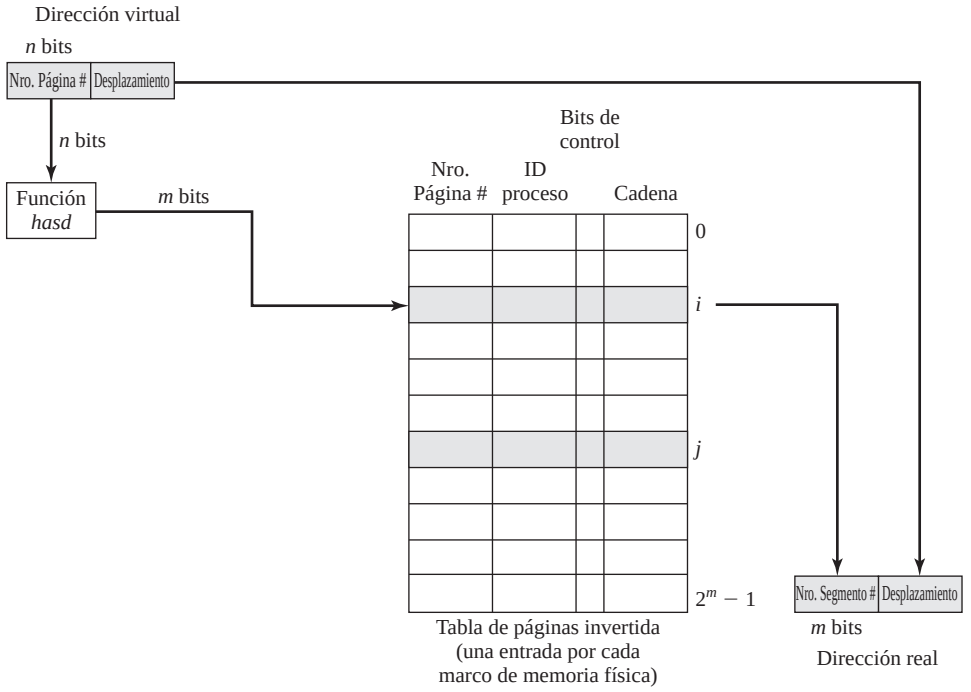


Figura 8.6. Estructura de tabla de páginas invertida.

En esta estrategia, la parte correspondiente al número de página de la dirección virtual se referencia por medio de un valor *hash* usando una función *hash* sencilla[1]. El valor *hash* es un puntero para la tabla de páginas invertida, que contiene las entradas de tablas de página. Hay una entrada en la tabla de páginas invertida por cada marco de página real en lugar de uno por cada página virtual. De esta forma, lo único que se requiere para estas tablas de página siempre es una proporción fija de la memoria real, independientemente del número de procesos o de las páginas virtuales soportadas. Debido a que más de una dirección virtual puede traducirse en la misma entrada de la tabla *hash*, una técnica de encadenamiento se utiliza para gestionar el desbordamiento. Las técnicas de *hashing* proporcionan habitualmente cadenas que no son excesivamente largas —entre una y dos entradas. La estructura de la tabla de páginas se denomina invertida debido a que se indexan sus entradas de la tabla de páginas por el número de marco en lugar de por el número de página virtual.

La Figura 8.6 muestra una implementación típica de la técnica de tabla de páginas invertida. Para un tamaño de memoria física de 2^m marcos, la tabla de páginas invertida contiene 2^m entradas, de forma que la entrada en la posición *i-esima* se refiere al marco *i*. La entrada en la tabla de páginas incluye la siguiente información:

- **Número de página.** Esta es la parte correspondiente al número de página de la dirección virtual.

[1] *Véase* Apéndice 8A para explicaciones sobre *hashing.*

- **Identificador del proceso.** El proceso que es propietario de está página. La combinación de número de página e identificador del proceso identifica a una página dentro del espacio de direcciones virtuales de un proceso en particular.
- **Bits de control.** Este campo incluye los *flags*, como por ejemplo, válido, referenciado, y modificado; e información de protección y cerrojos.
- **Puntero de la cadena.** Este campo es nulo (indicado posiblemente por un bit adicional) si no hay más entradas encadenadas en esta entrada. En otro caso, este campo contiene el valor del índice (número entre 0 y 2^{m-1}) de la siguiente entrada de la cadena.

En este ejemplo, la dirección virtual incluye un número de página de *n* bits, con $n > m$. La función *hash* traduce el número de página *n bits* en una cantidad de *m bits*, que se utiliza para indexar en la tabla de páginas invertida.

Buffer de traducción anticipada. En principio, toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada la tabla de páginas apropiada y otro para buscar los datos solicitados. De esa forma, un esquema de memoria virtual básico causaría el efecto de duplicar el tiempo de acceso a la memoria. Para solventar este problema, la mayoría de esquemas de la memoria virtual utilizan una *cache* especial de alta velocidad para las entradas de la tabla de página, habitualmente denominada *buffer* de **traducción anticipada (*translation lookaside buffer* - TLB)**[2]. Esta *cache* funciona de forma similar a una memoria *cache* general (*véase* Capítulo 1) y contiene aquellas entradas de la tabla de páginas que han sido usadas de forma más reciente. La organización del hardware de paginación resultante se ilustra en la Figura 8.7. Dada una dirección virtual, el procesador primero examina la TLB, si la entrada de la tabla de páginas solicitada está presente (*acierto en TLB*), entonces se recupera el número de marco y se construye la dirección real. Si la entrada de la tabla de páginas solicitada no se encuentra (*fallo en la TLB*), el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas. Si el bit de presente está puesto a 1, entonces la página se encuentra en memoria principal, y el procesador puede recuperar el número de marco desde la entrada de la tabla de páginas para construir la dirección real. El procesador también autorizará la TLB para incluir esta nueva entrada de tabla de páginas. Finalmente, si el bit presente no está puesto a 1, entonces la página solicitada no se encuentra en la memoria principal y se produce un fallo de acceso memoria, llamado **fallo de página**. En este punto, abandonamos el dominio del hardware para invocar al sistema operativo, el cual cargará la página necesaria y actualizada de la tabla de páginas.

La Figura 8.8 muestra un diagrama de flujo del uso de la TLB. Este diagrama de flujo muestra como si una página solicitada no se encuentra en la memoria principal, una interrupción de fallo de página hace que se invoque a la rutina de tratamiento de dicho fallo de página. Para mantener la simplicidad de este diagrama, no se ha mostrado el hecho de que el sistema operativo pueda activar otro proceso mientras la operación de E/S sobre disco se está realizando. Debido al principio de proximidad, la mayoría de referencias a la memoria virtual se encontrarán situadas en una página recientemente utiliza y por tanto, la mayoría de referencias invocarán una entrada de la tabla de páginas que se encuentra en la *cache*. Los estudios sobre la TLB de los sistemas VAX han demostrado que este esquema significa una importante mejora del rendimiento [CLAR85, SATY81].

Hay numerosos detalles adicionales relativos a la organización real de la TLB. Debido a que la TLB sólo contiene algunas de las entradas de toda la tabla de páginas, no es posible indexar simple-

---

[2] N. de T. Aunque la traducción más apropiada del *translation lookaside buffer* quizás sea la de *buffer* de traducción anticipada, en la literatura en castellano se utilizan las siglas TLB de forma generalizada para describir dicha memoria. Por ello, y para no causar confusión con otros textos, a lo largo del presente libro utilizaremos dichas siglas para referirnos a ella.

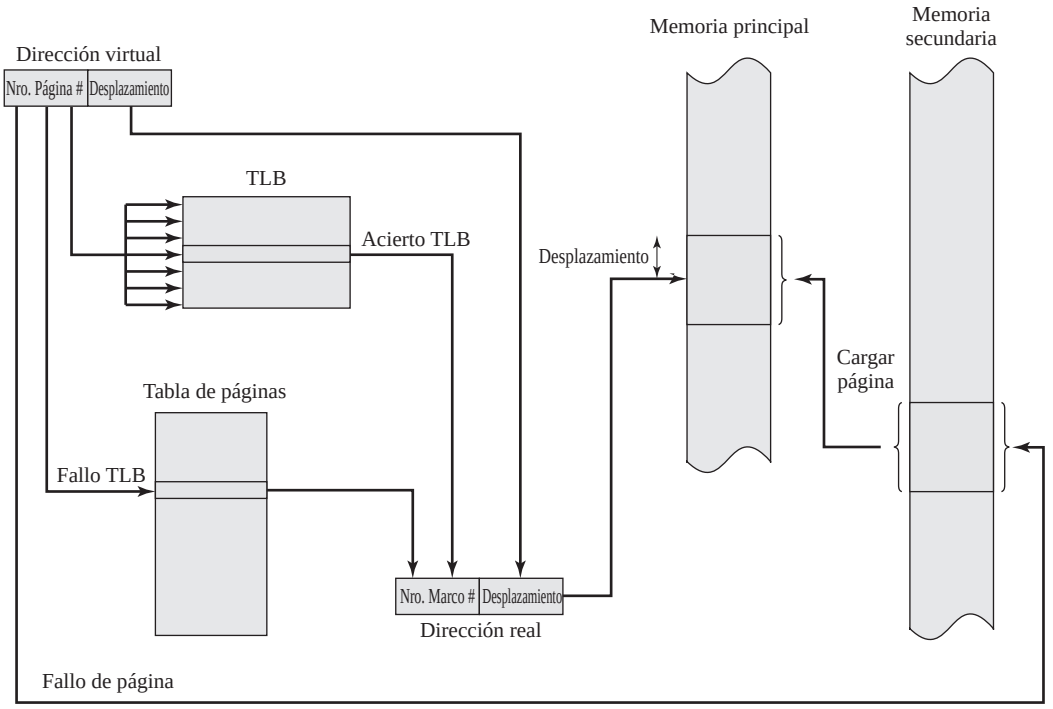


Figura 8.7. Uso de la TLB.

mente la TLB por medio de número página. En lugar de eso, cada entrada de la TLB debe incluir un número de página así como la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina **resolución asociativa (*asociative mapping*)** que contrasta con la resolución directa, o indexación, utilizada para buscar en la tabla de páginas en la Figura 8.9. El diseño de la TLB debe considerar también la forma mediante la cual las entradas se organizan en ella y qué entrada se debe reemplazar cuando se necesite traer una nueva entrada. Estos aspectos deben considerarse en el diseño de la cache hardware. Este punto no se contempla en este libro; el lector podrá consultar el funcionamiento del diseño de una cache para más detalle en, por ejemplo, [STAL03].

Para concluir, el mecanismo de memoria virtual debe interactuar con el sistema de *cache* (no la *cache* de TLB, sino la cache de la memoria principal). Esto se ilustra en la Figura 8.10. Una dirección virtual tendrá generalmente el formato número de página, desplazamiento. Primero, el sistema de memoria consulta la TLB para ver si se encuentra presente una entrada de tabla de página que coincide. Si es así, la dirección real (física) se genera combinando el número de marco con el desplazamiento. Si no, la entrada se busca en la tabla de páginas. Una vez se ha generado la dirección real, que mantiene el formato de etiqueta (*tag*)[3] y resto (*remainder*), se consulta la cache para ver si el bloque que contiene esa palabra se encuentra ahí. Si es así, se le devuelve a la CPU. Si no, la palabra se busca en la memoria principal.

---

[3] *Véase* en la Figura 1.17. Normalmente, una etiqueta son los bits situados más a la izquierda de una dirección real. Una vez más, para un estudio más detallado sobre las caches, se refiere al lector a [STAL03].

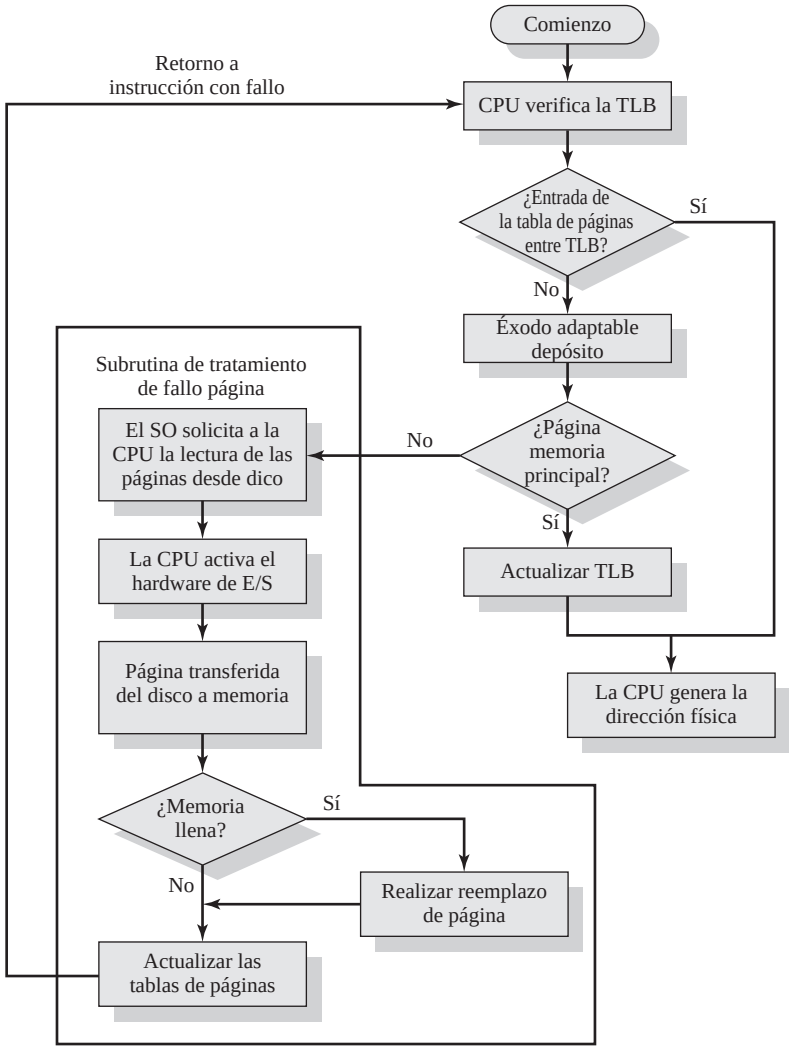


Figura 8.8. Operación de paginación y TLB [FURH87].

El lector podrá apreciar la complejidad del hardware de la CPU que participa en una referencia a memoria sencilla. La dirección virtual se traduce a una dirección real lo cual implica una referencia a la entrada de la tabla de páginas, que puede estar en la TLB, en la memoria principal, o en disco. La palabra referenciada puede estar en la cache, en la memoria principal, o en disco. Si dicha palabra referenciada se encuentra únicamente en disco, la página que contiene dicha palabra debe cargarse en la memoria principal y su bloque en la cache. Adicionalmente, la entrada en la tabla de páginas para dicha página debe actualizarse.

Tamaño de página. Una decisión de diseño hardware importante es el tamaño de página a usar. Hay varios factores a considerar. Por un lado, está la fragmentación interna. Evidentemente, cuanto mayor es el tamaño de la página, menor cantidad de fragmentación interna. Para optimizar el uso de la memoria principal, sería beneficioso reducir la fragmentación interna. Por otro lado, cuanto menor es la página, mayor número de páginas son necesarias para cada proceso. Un mayor número de páginas por proceso significa también mayores tablas de páginas. Para programas grandes en un entorno altamente multipro-

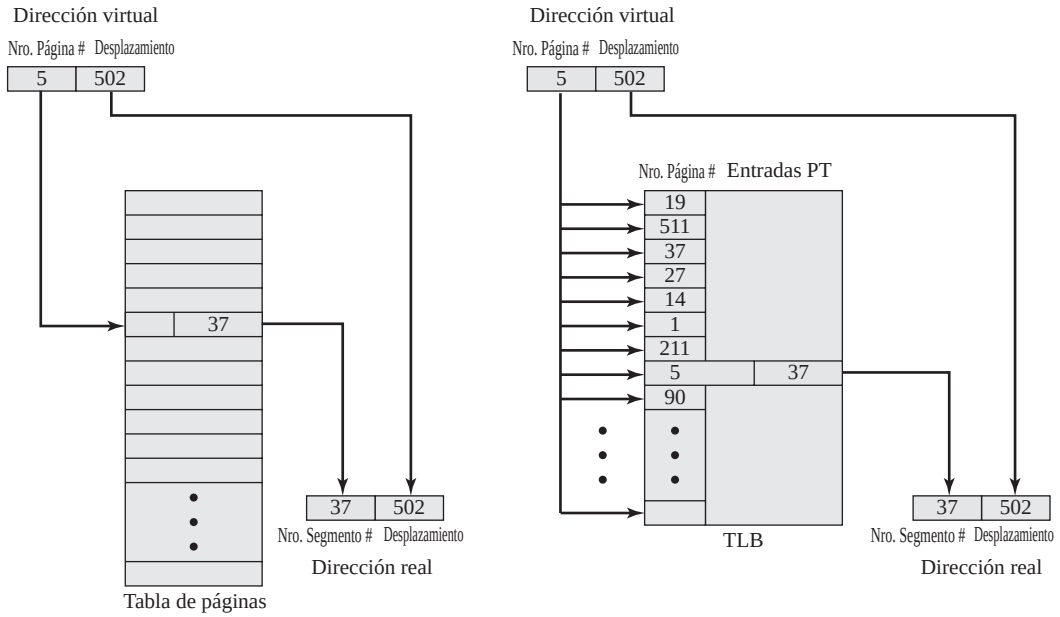


Figura 8.9. Resolución directa vs. asociativa para las entradas en la tabla de páginas.

gramado, esto significa que determinadas partes de las tablas de página de los procesos activos deben encontrarse en la memoria virtual, no en la memoria principal. Por tanto, puede haber un fallo de página doble para una referencia sencilla a memoria: el primero para atraer la tabla de página de la parte solicitada y el segundo para atraer la página del propio proceso. Otro factor importante son las características físicas de la mayoría de los dispositivos de la memoria secundaria, que son de tipo giratorio, favoreciendo tamaños de página grandes para mejorar la eficiencia de transferencia de bloques de datos.

Aumentando la complejidad de estos aspectos se encuentra el efecto que el tamaño de página tiene en relación a la posibilidad de que ocurra un fallo de página. Este comportamiento en términos generales, se encuentra recogido en la Figura 8.11a que se basa en el principio de proximidad. Si el tamaño de página es muy pequeño, de forma habitual habrá un número relativamente alto de páginas disponibles en la memoria principal para cada proceso. Después de un tiempo, las páginas en memoria contendrán las partes de los procesos a las que se ha hecho referencia de forma reciente. De esta forma, la tasa de fallos de página debería ser baja. A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada. Así pues, el efecto del principio de proximidad se debilita y la tasa de fallos de página comienza a crecer. En algún momento, sin embargo, la tasa de fallos de página comenzará a caer a medida que el tamaño de la página se aproxima al tamaño del proceso completo (punto *P* en el diagrama). Cuando una única página contiene el proceso completo, no habrá fallos de página.

Una complicación adicional es que la tasa de fallos de página también viene determinada por el número de marcos asociados a cada proceso. La Figura 8.11b muestra que, para un tamaño de página fijo, la tasa de fallos cae a medida que el número de páginas mantenidas en la memoria principal crece[4]. Por

[4] El parámetro *W* representa el conjunto de trabajo, un concepto que se analizará en la Sección 8.2.

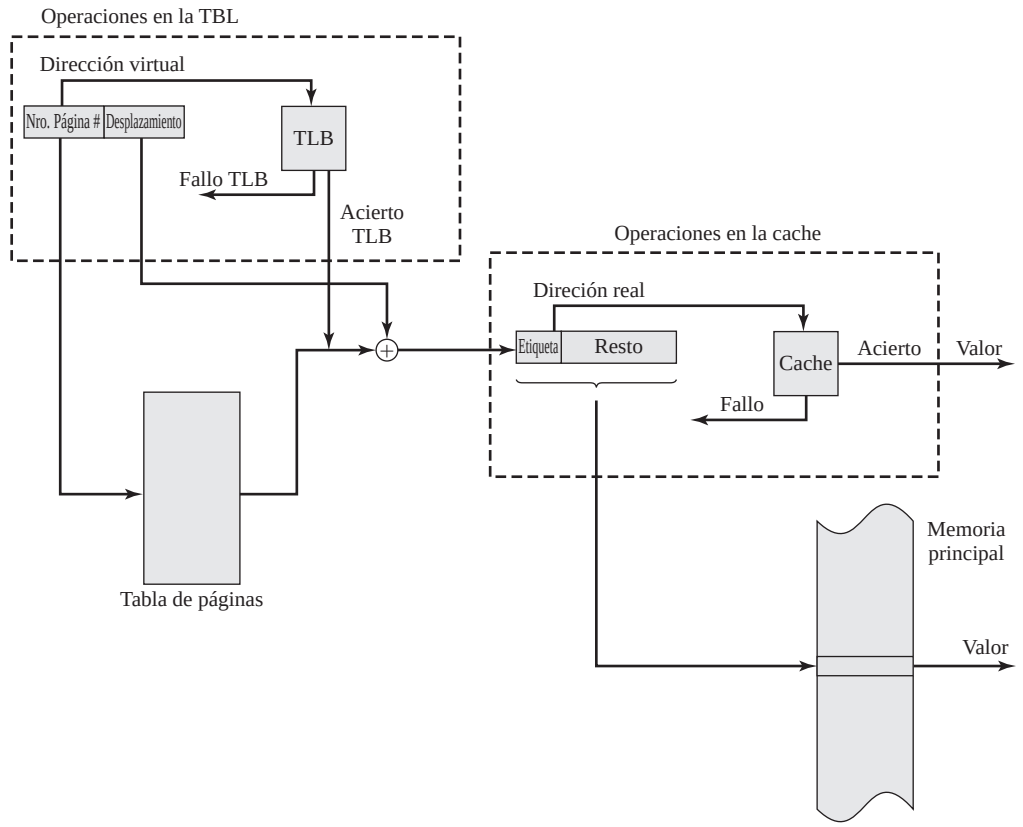
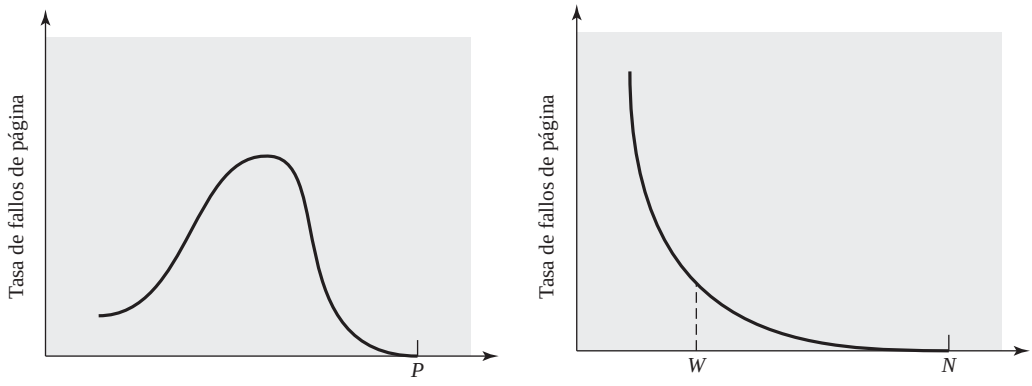


Figura 8.10. Operaciones en la TLB y en la *cache*.



(a) Tamaño de página

(b) Número de marcos de página reservados

P = tamaño del proceso entero
W = conjunto de trabajo
N = número total de páginas en proceso

Figura 8.11. Comportamiento típico de la paginación de un programa.

tanto, una política software (la cantidad de memoria reservada por cada proceso) interactúa con decisiones de diseño del propio hardware (tamaño de página).

La Tabla 8 .2 contiene un listado de los tamaños de páginas que tienen determinadas arquitecturas.

Para concluir, el aspecto de diseño del tamaño página se encuentra relacionado con el tamaño de la memoria física y el tamaño del programa. Al mismo tiempo que la memoria principal está siendo cada vez más grande, el espacio de direcciones utilizado por las aplicaciones también crece. Esta tendencia resulta más evidente en ordenadores personales y estaciones de trabajo, donde las aplicaciones tienen una complejidad creciente. Por contra, diversas técnicas de programación actuales usadas para programas de gran tamaño tienden a reducir el efecto de la proximidad de referencias dentro un proceso [HUCK93]. Por ejemplo,

- Las técnicas de programación orientada a objetos motivan el uso de muchos módulos de datos y programas de pequeño tamaño con referencias repartidas sobre un número relativamente alto de objetos en un periodo de tiempo bastante corto.
- Las aplicaciones multihilo (*multithreaded*) pueden presentar cambios abruptos en el flujo de instrucciones y referencias a la memoria fraccionadas.

Tabla 8.2. Ejemplo de tamaños de página.

Computer	Tamaño de página
Atlas	512 palabras de 48-bits
Honeywell-Multics	1024 palabras de 36-bits
IBM 370/XA y 370/ESA	4 Kbytes
Familia VAX	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes hasta 16 Mbytes
UltraSPARC	8 Kbytes hasta 4 Mbytes
Pentium	4 Kbytes o 4 Mbytes
PowerPc	4 Kbytes
Itanium	4 Kbytes hasta 256 Mbytes

Para un tamaño determinado de una TLB, a medida que el tamaño del proceso crece y la proximidad de referencias decrece, el índice de aciertos en TLB se va reduciendo. Bajo estas circunstancias, la TLB se puede convertir en el cuello de botella del rendimiento (por ejemplo, *véase* [CHEN92]).

Una forma de incrementar el rendimiento en la TLB es utilizar una TLB de gran tamaño, con más entradas. Sin embargo, el tamaño de TLB interactúa con otros aspectos del diseño hardware, por ejemplo la *cache* de memoria principal o el número de accesos a memoria por ciclo de instrucción [TALL92]. Una de las principales pegas es que el tamaño de la TLB no tiene la misma tendencia de crecimiento que el tamaño de la memoria principal, en velocidad de crecimiento. Como alternativa se encuentra el uso de tamaños de página mayores de forma que cada entrada en la tabla de páginas referenciada en la TLB apunte a un bloque de memoria relativamente grande. Pero acabamos de ver que el uso de tamaños de página muy grandes puede significar la degradación del rendimiento.

Sobre estas consideraciones, un gran número de diseñadores han investigado la posibilidad de utilizar múltiples tamaños de página [TALL92, KHAL93], y diferentes arquitecturas de microprocesadores dan soporte a diversos tamaños de página, incluyendo MIPS R4000, Alpha, UltraSPARC, Pentium, e IA-64. Los tamaños de página múltiples proporcionan la flexibilidad necesaria para el uso de la TLB de forma eficiente. Por ejemplo, regiones contiguas de memoria de gran tamaño dentro del espacio direcciones del proceso, como las instrucciones del programa, se pueden proyectar sobre un reducido número de páginas de gran tamaño en lugar de un gran número de páginas de tamaño más pequeño, mientras que las pilas de los diferentes hilos se pueden alojar utilizando tamaños de página relativamente pequeños. Sin embargo, la mayoría de sistemas operativos comerciales aún soportan únicamente un tamaño de página, independientemente de las capacidades del hardware sobre el que están ejecutando. El motivo de esto se debe a que el tamaño de página afecta a diferentes aspectos del sistema operativo; por tanto, un cambio a un modelo de diferentes tamaños de páginas representa una tarea significativamente compleja (*véase* [GANA98] para más detalle).

SEGMENTACIÓN

Las implicaciones en la memoria virtual. La segmentación permite al programador ver la memoria como si se tratase de diferentes espacios de direcciones o segmentos. Los segmentos pueden ser de tamaños diferentes, en realidad de tamaño dinámico. Una referencia a la memoria consiste en un formato de dirección del tipo (número de segmento, desplazamiento).

Esta organización tiene un gran número de ventajas para el programador sobre los espacios de direcciones no segmentados:

1. Simplifica el tratamiento de estructuras de datos que pueden crecer. Si el programador no conoce a priori el tamaño que una estructura de datos en particular puede alcanzar es necesario hacer una estimación salvo que se utilicen tamaños de segmento dinámicos. Con la memoria virtual segmentada, a una estructura de datos se le puede asignar su propio segmento, y el sistema operativo expandirá o reducirá el segmento bajo demanda. Si un segmento que necesita expandirse se encuentre en la memoria principal y no hay suficiente tamaño, el sistema operativo poder mover el segmento a un área de la memoria principal mayor, si se encuentra disponible, o enviarlo a *swap*. En este último caso el segmento al que se ha incrementado el tamaño volverá a la memoria principal en la siguiente oportunidad que tenga.
2. Permite programas que se modifican o recopilan de forma independiente, sin requerir que el conjunto completo de programas se re-enlacen y se vuelvan a cargar. De nuevo, esta posibilidad se puede articular por medio de la utilización de múltiples segmentos.
3. Da soporte a la compartición entre procesos. El programador puede situar un programa de utilidad o una tabla de datos que resulte útil en un segmento al que pueda hacerse referencia desde otros procesos.
4. Soporta los mecanismos de protección. Esto es debido a que un segmento puede definirse para contener un conjunto de programas o datos bien descritos, el programador o el administrador de sistemas puede asignar privilegios de acceso de una forma apropiada.

Organización. En la exposición de la segmentación sencilla, indicamos que cada proceso tiene su propia tabla de segmentos, y que cuando todos estos segmentos se han cargado en la memoria principal, la tabla de segmentos del proceso se crea y se carga también en la memoria principal. Cada entrada de la tabla de segmentos contiene la dirección de comienzo del correspondiente segmento en la

memoria principal, así como la longitud del mismo. El mismo mecanismo, una tabla segmentos, se necesita cuando se están tratando esquemas de memoria virtual basados en segmentación. De nuevo, lo habitual es que haya una única tabla de segmentos por cada uno de los procesos. En este caso sin embargo, las entradas en la tabla de segmentos son un poco más complejas (Figura 8.2b). Debido a que sólo algunos de los segmentos del proceso pueden encontrarse en la memoria principal, se necesita un bit en cada entrada de la tabla de segmentos para indicar si el correspondiente segmento se encuentra presente en la memoria principal o no. Si indica que el segmento está en memoria, la entrada también debe incluir la dirección de comienzo y la longitud del mismo.

Otro bit de control en la entrada de la tabla de segmentos es el bit de modificado, que indica si los contenidos del segmento correspondiente se han modificado desde que se cargó por última vez en la memoria principal. Si no hay ningún cambio, no es necesario escribir el segmento cuando se reemplace de la memoria principal. También pueden darse otros bits de control. Por ejemplo, si la gestión de protección y compartición se gestiona a nivel de segmento, se necesitarán los bits correspondientes a estos fines.

El mecanismo básico para la lectura de una palabra de memoria implica la traducción de una dirección virtual, o lógica, consistente en un número de segmento y un desplazamiento, en una dirección física, usando la tabla de segmentos. Debido a que la tabla de segmentos es de tamaño variable, dependiendo del tamaño del proceso, no se puede suponer que se encuentra almacenada en un registro. En su lugar, debe encontrarse en la memoria principal para poder accederse. La Figura 8.12 sugiere una implementacione hardware de este esquema (nótese la similitud con la Figura 8.3). Cuando un proceso en particular está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos para dicho proceso. El número de segmento de la dirección virtual se utiliza para indexar esta tabla y para buscar la dirección de la memoria principal donde comienza dicho segmento. Ésta es añadida a la parte de desplazamiento de la dirección virtual para producir la dirección real solicitada.

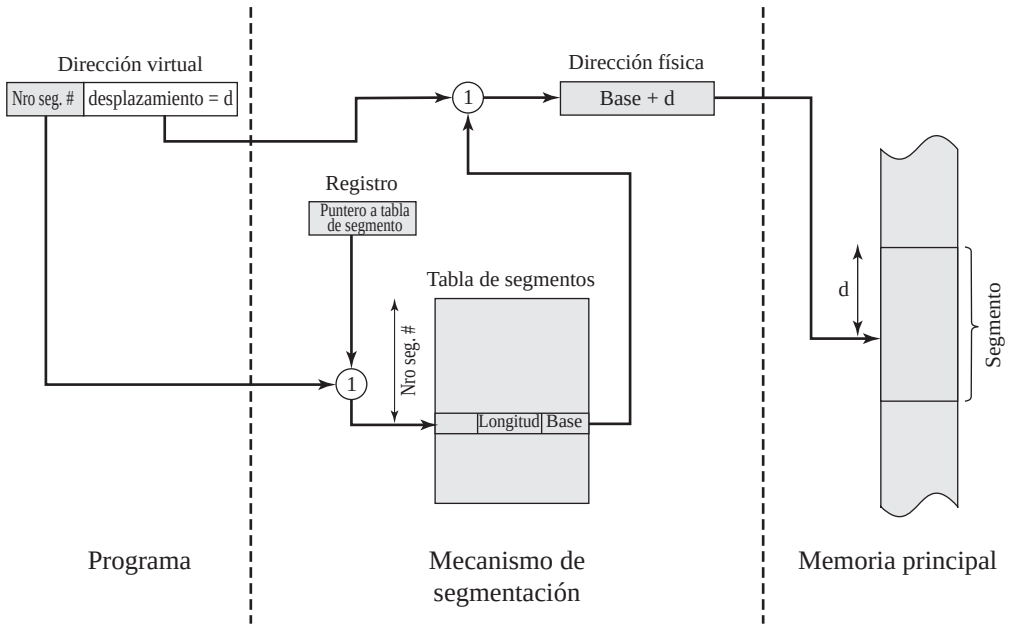


Figura 8.12. Traducción de direcciones en un sistema con segmentación.

PAGINACIÓN Y SEGMENTACIÓN COMBINADAS

Paginación y segmentación, cada una tiene sus propias ventajas. La paginación es transparente al programador y elimina la fragmentación externa, y por tanto proporciona un uso eficiente de la memoria principal. Adicionalmente, debido a que los fragmentos que se mueven entre la memoria y el disco son de un tamaño igual y prefijado, es posible desarrollar algoritmos de gestión de la memoria más sofisticados que exploten el comportamiento de los programas, como veremos más adelante. La segmentación sí es visible al programador y tiene los beneficios que hemos visto anteriormente, incluyendo la posibilidad de manejar estructuras de datos que crecen, modularidad, y dar soporte a la compartición y a la protección. Para combinar las ventajas de ambos, algunos sistemas por medio del hardware del procesador y del soporte del sistema operativo son capaces de proporcionar ambos.

En un sistema combinado de paginación/segmentación, el espacio de direcciones del usuario se divide en un número de segmentos, a discreción del programador. Cada segmento es, por su parte, dividido en un número de páginas de tamaño fijo, que son del tamaño de los marcos de la memoria principal. Si un segmento tiene longitud inferior a una página, el segmento ocupará únicamente una página. Desde el punto de vista del programador, una dirección lógica sigue conteniendo un número de segmento y un desplazamiento dentro de dicho segmento. Desde el punto de vista del sistema, el desplazamiento dentro del segmento es visto como un número de página y un desplazamiento dentro de la página incluida en el segmento.

La Figura 8.13 sugiere la estructura para proporcionar soporte o la combinación de paginación y segmentación (nótese la similitud con la Figura 8.5). Asociada a cada proceso existe una tabla de segmentos y varias tablas de páginas, una por cada uno de los segmentos. Cuando un proceso está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos de dicho proceso. A partir de la dirección virtual, el procesador utiliza la parte correspondiente al número de segmento para indexar dentro de la tabla de segmentos del proceso para encontrar la tabla de pági-

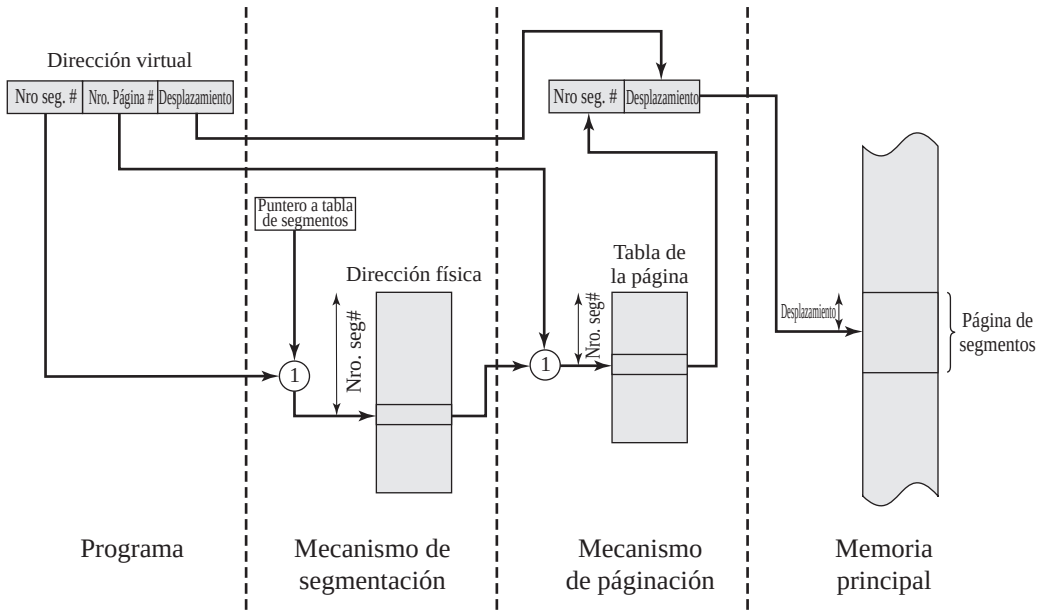


Figura 8.13. Traducción de direcciones en un sistema con segmentación/paginación.

nas de dicho segmento. Después, la parte correspondiente al número de página de la dirección virtual original se utiliza para indexar la tabla de páginas y buscar el correspondiente número de marco. Éste se combina con el desplazamiento correspondiente de la dirección virtual para generar la dirección real requerida.

En la Figura 8.2c se muestran los formatos de la entrada en la tabla de segmentos y de la entrada en la tabla de páginas. Como antes, la entrada en la tabla de segmentos contiene la longitud del segmento. También contiene el campo base, que ahora hace referencia a la tabla de páginas. Los bits de presente y modificado no se necesitan debido a que estos aspectos se gestionan a nivel de página. Otros bits de control sí pueden utilizarse, a efectos de compartición y protección. La entrada en la tabla de páginas es esencialmente la misma que para el sistema de paginación puro. El número de página se proyecta en su número de marco correspondiente si la página se encuentra presente en la memoria. El bit de modificado indica si la página necesita escribirse cuando se expulse del marco de página actual. Puede haber otros bits de control relacionados con la protección u otros aspectos de la gestión de la memoria.

PROTECCIÓN Y COMPARTICIÓN

La segmentación proporciona una vía para la implementación de las políticas de protección y compartición. Debido a que cada entrada en la tabla de segmentos incluye la longitud así como la dirección base, un programa no puede, de forma descontrolada, acceder a una posición de memoria principal más allá de los límites del segmento. Para conseguir compartición, es posible que un segmento se encuentre referenciado desde las tablas de segmentos de más de un proceso. Los mecanismos están, por supuesto, disponibles en los sistemas de paginación. Sin embargo, en este caso la estructura de páginas de un programa y los datos no son visible para el programador, haciendo que la especificación de la protección y los requisitos de compartición sean menos cómodos. La Figura 8.14 ilustra los tipos de relaciones de protección que se pueden definir en dicho sistema.

También es posible proporcionar mecanismos más sofisticados. Un esquema habitual es utilizar la estructura de protección en anillo, del tipo que indicamos en el Capítulo 3 (Problema 3.7). En este esquema, los anillos con números bajos, o interiores, disfrutan de mayores privilegios que los anillos con numeraciones más altas, o exteriores. Normalmente, el anillo 0 se reserva para funciones del núcleo del sistema operativo, con las aplicaciones en niveles superiores. Algunas utilidades o servicios de sistema operativo pueden ocupar un anillo intermedio. Los principios básicos de los sistemas en anillo son los siguientes:

- Un programa pueda acceder sólo a los datos residentes en el mismo anillo o en anillos con menos privilegios.
- Un programa puede invocar servicios residentes en el mismo anillo o anillos con más privilegios.

8.2. SOFTWARE DEL SISTEMA OPERATIVO

El diseño de la parte de la gestión de la memoria del sistema operativo depende de tres opciones fundamentales a elegir:

- Si el sistema usa o no técnicas de memoria virtual.
- El uso de paginación o segmentación o ambas.
- Los algoritmos utilizados para los diferentes aspectos de la gestión de la memoria.

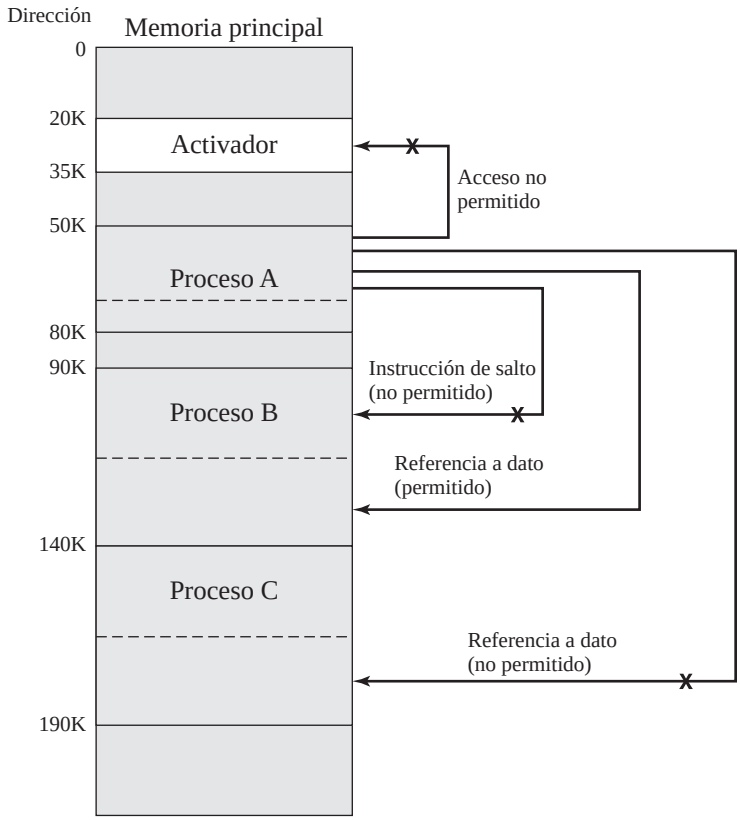


Figura 8.14. Relaciones de protección entre segmentos.

Las elecciones posibles para las dos primeras opciones dependen de la plataforma hardware disponible. Así, las primeras implantaciones de UNIX no proporcionaban memoria virtual porque los procesadores sobre los cuales ejecutaban no daban soporte para paginación o segmentación. Ninguna de estas técnicas es abordable sin una plataforma hardware para traducción de direcciones y otras funciones básicas.

Hay también dos comentarios adicionales sobre estas dos primeras opciones: primero, con la excepción de los sistemas operativos de algunas plataformas como los ordenadores personales antiguos, como MS-DOS, y de otros sistemas de carácter especializado, todos los sistemas operativos importantes proporcionan memoria virtual. Segundo, los sistemas de segmentación pura son en la actualidad realmente escasos. Cuando la segmentación se combina con paginación, la mayoría de los aspectos de la gestión de la memoria relativos al diseño sistema operativo se encuadran en el área de la paginación[5]. De esta forma, en esta sección nos concentraremos en los aspectos asociados a la paginación.

Las elecciones relativas a la tercera opción entran dentro del dominio del software del sistema operativo y son el objeto de esta sección. La Tabla 8.3 muestra los elementos de diseño clave que se

[5] La protección y la compartición, en un sistema combinado de segmentación/paginación, se suelen delegar habitualmente a nivel de segmento. Abordaremos estas cuestiones en capítulos posteriores.

Tabla 8.3. Políticas del sistema operativo para la memoria virtual.

Política de recuperación	**Gestión del conjunto residente**
Bajo demanda	Tamaño del conjunto residente
Paginación adelantada	Fijo
	Variable
Política de ubicación	Ámbito de reemplazo
	Global
Política de reemplazo	Local
Algoritmos básicos	
Óptimo	**Política de limpieza**
FIFO	Bajo demanda
Usada menos recientemente (LRU)	Limpieza adelantada
Del reloj	**Control de carga**
Buffers de página	Grado de multiprogramación

van a examinar. En cada caso, el aspecto central es el rendimiento: Se tratará de minimizar la tasa de ocurrencia de fallos de página, porque los fallos de página causan una considerable sobrecarga sobre el software. Como mínimo, esta sobrecarga incluye la decisión de qué página o páginas residentes se van a reemplazar, y la E/S del intercambio o *swap* de dichas páginas. También, el sistema operativo debe planificar la ejecución de otro proceso durante la operación de E/S de la página, causando un cambio de contexto. De la misma forma, intentaremos organizar determinados aspectos de forma que, durante el tiempo de ejecución de un proceso, la probabilidad de hacer referencia a una palabra en una página que no se encuentre presente sea mínima. En todas estas áreas indicadas en la Tabla 8.3, no existe una política que sea mejor que todas las demás. Como se verá, la tarea de gestión de la memoria en un entorno de paginación es endiabladamente compleja. Adicionalmente, el rendimiento de un conjunto de políticas en particular depende del tamaño de la memoria, de la velocidad relativa de la memoria principal y secundaria, del tamaño y del número de procesos que están compitiendo por los recursos, y del comportamiento en ejecución de los diferentes programas de forma individual. Esta última característica depende de la naturaleza de la aplicación, el lenguaje de programación y el compilador utilizado, el estilo del programador que lo escribió, y, para un programa interactivo, el comportamiento dinámico del usuario. Así pues, el lector no debe esperar de ningún modo una respuesta definitiva aquí. Para sistemas pequeños, el diseño de sistema operativo debe intentar elegir un conjunto de políticas que parezcan funcionar «bien» sobre una amplia gama de condiciones, basándose en el conocimiento sobre el estado actual del sistema. Para grandes sistemas, particularmente *mainframes*, el sistema operativo debe incluir herramientas de monitorización y control que permitan al administrador de la instalación ajustar ésta para obtener «buenos» resultados en base a las condiciones de la instalación.

POLÍTICA DE RECUPERACIÓN

La política de recuperación determina cuándo una página se trae a la memoria principal. Las dos alternativas habituales son bajo demanda y paginación adelantada (*prepaging*). Con **paginación bajo demanda**, una página se trae a memoria sólo cuando se hace referencia a una posición en dicha página. Si el resto de elementos en la política de gestión de la memoria funcionan correctamente, ocurriría

lo siguiente. Cuando un proceso se arranca inicialmente, va a haber una ráfaga de fallos de página. Según se van trayendo más y más páginas a la memoria, el principio de proximidad sugiere que las futuras referencias se encontrarán en las páginas recientemente traídas. Así, después de un tiempo, la situación se estabilizará y el número de fallos de página caerá hasta un nivel muy bajo.

Con **paginación adelantada (*prepaging*)**, se traen a memoria también otras páginas, diferentes de la que ha causado el fallo de página. La paginación adelantada tiene en cuenta las características que tienen la mayoría de dispositivos de memoria secundaria, tales como los discos, que tienen tiempos de búsqueda y latencia de rotación. Si las páginas de un proceso se encuentran almacenadas en la memoria secundaria de forma contigua, es mucho más eficiente traer a la memoria un número de páginas contiguas de una vez, en lugar de traerlas una a una a lo largo de un periodo de tiempo más amplio. Por supuesto, esta política es ineficiente si la mayoría de las páginas que se han traído no se referencian a posteriori.

La política de paginación adelantada puede emplearse bien cuando el proceso se arranca, en cuyo caso el programador tiene que designar de alguna forma las páginas necesarias, o cada vez que ocurra un fallo de página. Este último caso es el más apropiado porque resulta completamente invisible al programador. Sin embargo, la completa utilidad de la paginación adelantada no se encuentra reconocida [MAEK87].

La paginación adelantada no se debe confundir con el *swapping*. Cuando un proceso se saca de la memoria y se le coloca en estado suspendido, todas sus páginas residentes se expulsan de la memoria. Cuando el proceso se recupera, todas las páginas que estaban previamente en la memoria principal retornan a ella.

POLÍTICA DE UBICACIÓN

La política de ubicación determina en qué parte de la memoria real van a residir las porciones de la memoria de un proceso. En los sistema de segmentación puros, la política de ubicación es un aspecto de diseño muy importante; políticas del estilo mejor ajuste, primer ajuste, y similares que se discutieron en el Capítulo 7, son las diferentes alternativas. Sin embargo, para sistemas que usan o bien paginación pura o paginación combinada con segmentación, la ubicación es habitualmente irrelevante debido a que el hardware de traducción de direcciones y el hardware de acceso a la memoria principal pueden realizar sus funciones en cualquier combinación de página-marco con la misma eficiencia.

Existe otro entorno en el cual la ubicación tiene implicación importante, y es un tema de investigación y desarrollo. En aquellos sistemas llamados multiprocesadores de acceso a la memoria no uniforme (*nonuniform memory access multiprocessors*-NUMA), la memoria distribuida compartida de la máquina puede referenciarse por cualquier otro procesador dentro de la misma máquina, pero con un tiempo de acceso dependiente de la localización física y que varía con la distancia entre el procesador y el módulo de la memoria. De esta forma, el rendimiento depende significativamente de la distancia a la cual reside el dato en relación al procesador que va a utilizar [LARO92, BOLO89, COX89]. Para sistemas NUMA, una estrategia de ubicación automática aceptable es aquella que asigna las páginas al módulo de la memoria que finalmente proporcionará mejor rendimiento.

POLÍTICA DE REEMPLAZO

En la mayoría de los libros sobre sistemas operativos, el tratamiento de la gestión de la memoria incluye una sección titulada «política de reemplazo», que trata de la selección de una página en la memoria principal como candidata para reemplazarse cuando se va traer una nueva página. Este tema es, a menudo, difícil de explicar debido a que hay varios conceptos interrelacionados:

- ¿Cuántos marcos de página se van a reservar para cada uno de los procesos activos?
- Si el conjunto de páginas que se van a considerar para realizar el reemplazo se limita a aquellas del mismo proceso que ha causado el fallo de página o, si por el contrario, se consideran todos los marcos de página de la memoria principal.
- Entre el conjunto de páginas a considerar, qué página en concreto es la que se va a reemplazar.

Nos referimos a los dos primeros conceptos como la *gestión del conjunto residente*, que se trata en la siguiente subsección, y se ha reservado el término *política de reemplazo* para el tercer concepto, que se discutirá en esta misma subsección.

El área de políticas de reemplazo es probablemente el aspecto de la gestión de la memoria que ha sido más estudiado. Cuando todos los marcos de la memoria principal están ocupados y es necesario traer una nueva página para resolver un fallo de página, la política de reemplazo determina qué página de las que actualmente están en memoria va a reemplazarse. Todas las políticas tienen como objetivo que la página que va a eliminarse sea aquella que tiene menos posibilidades de volver a tener una referencia en un futuro próximo. Debido al principio de proximidad de referencia, existe a menudo una alta correlación entre el histórico de referencias recientes y los patrones de referencia en un futuro próximo. Así, la mayoría de políticas tratan de predecir el comportamiento futuro en base al comportamiento pasado. En contraprestación, se debe considerar que cuanto más elaborada y sofisticada es una política de reemplazo, mayor va a ser la sobrecarga a nivel software y hardware para implementarla.

Bloqueo de marcos. Es necesario mencionar una restricción que se aplica a las políticas de reemplazo antes de indagar en los diferentes algoritmos: algunos marcos de la memoria principal pueden encontrarse bloqueados. Cuando un marco está bloqueado, la página actualmente almacenada en dicho marco no puede reemplazarse. Gran parte del núcleo del sistema operativo se almacena en marcos que están bloqueados, así como otras estructuras de control claves. Adicionalmente, los *buffers* de E/S y otras áreas de tipo critico también se ponen en marcos bloqueados en la memoria principal. El bloqueo se puede realizar asociando un bit de bloqueo a cada uno de los marcos. Este bit se puede almacenar en la tabla de marcos o también incluirse en la tabla de páginas actual.

Algoritmos básicos. Independientemente de la estrategia de gestión del conjunto residente (que se discutirá en la siguiente subsección), existen ciertos algoritmos básicos que se utilizan para la selección de la página a reemplazar. Los algoritmos de reemplazo que se han desarrollando a lo largo de la literatura son:

- Óptimo.
- Usado menos recientemente (*least recently used*-LRU).
- FIFO (*first-in-first-out*).
- Reloj.

La **política óptima** de selección tomará como reemplazo la página para la cuál el instante de la siguiente referencia se encuentra más lejos. Se puede ver que para esta política los resultados son el menor número de posibles fallos de página [BELA66]. Evidentemente, esta política es imposible de implementar, porque requiere que el sistema operativo tenga un perfecto conocimiento de los eventos futuros. Sin embargo se utiliza como un estándar apartir del cual contrastar algoritmos reales.

La Figura 8.15 proporciona un ejemplo de la política óptima. El ejemplo asume una reserva de marcos fija (tamaño del conjunto residente fijo) para este proceso de un total de tres marcos. La eje-

cución de proceso requiere la referencia de cinco páginas diferentes. El flujo de páginas referenciadas por el programa antes citado es el siguiente:

2 3 2 1 5 2 4 5 3 2 5 2

lo cual representa que la primera página a la que se va a hacer referencia es la 2, la segunda página la 3, y así en adelante. La política óptima produce tres fallos de página después de que la reserva de marcos se haya ocupado completamente.

La política de reemplazo de la página **usada menos recientemente (LRU)** seleccionará como candidata la página de memoria que no se haya referenciado desde hace más tiempo. Debido al principio de proximidad referenciada, esta página sería la que tiene menos probabilidad de volverá a tener referencias en un futuro próximo. Y, de hecho, la política LRU proporciona unos resultados casi tan buenos como la política óptima. El problema con esta alternativa es la dificultad en su implementación. Una opción sería etiquetar cada página con el instante de tiempo de su última referencia; esto podría ser en cada una de las referencias a la memoria, bien instrucciones o datos. Incluso en el caso de que el hardware diera soporte a dicho esquema, la sobrecarga sería tremenda. De forma alternativa se puede mantener una pila de referencias a páginas, que igualmente es una opción costosa.

La Figura 8.15 muestra un ejemplo del comportamiento de LRU, utilizando el mismo flujo de referencias a páginas que en el ejemplo de la política óptima. En este ejemplo, se producen cuatro fallos de página.

La **política FIFO** trata los marcos de página ocupados como si se tratase de un *buffer* circular, y las páginas se remplazan mediante una estrategia cíclica de tipo round-robin. Todo lo que se necesita

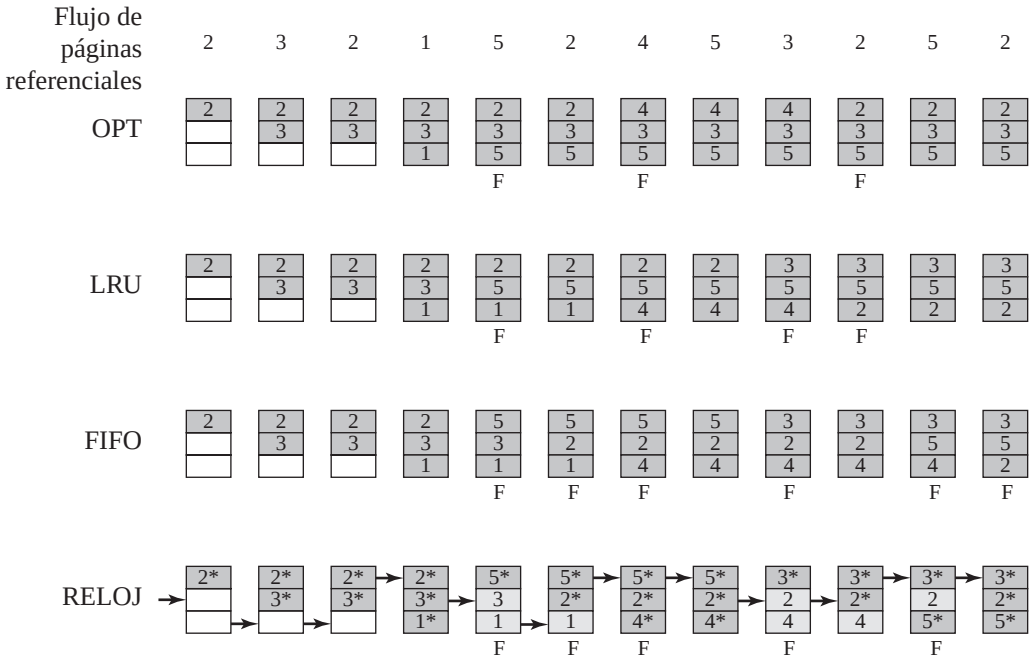


Figura 8.15. Comportamiento de cuatro algoritmos de reemplazo de páginas.

es un puntero que recorra de forma circular los marcos de página del proceso. Por tanto, se trata de una de las políticas de reemplazo más sencilla de implementar. El razonamiento tras este modelo, además de su simplicidad, es el reemplazo de la página que lleva en memoria más tiempo: una página traída a la memoria hace mucho tiempo puede haber dejado de utilizarse. Este razonamiento a menudo es erróneo, debido a que es habitual que en los programas haya una zona del mismo o regiones de datos que son utilizados de forma intensiva durante todo el tiempo de vida del proceso. Esas páginas son expulsadas de la memoria y traídas de nuevo de forma repetida por un algoritmo de tipo FIFO.

Continuando con el mismo ejemplo de la Figura 8.15, la política FIFO genera un total de seis fallos de página. Nótese que el algoritmo LRU reconoce que a las páginas 2 y 5 se hace referencia con mayor frecuencia que a cualquier otra página, mientras que FIFO no lo hace.

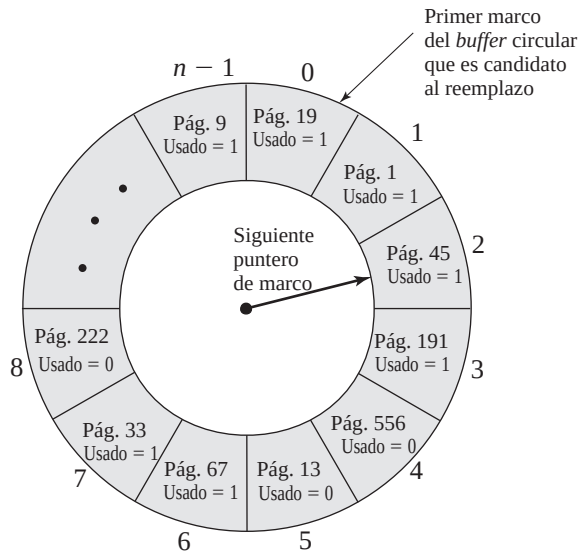
Mientras que la política LRU alcanza unos resultados similares a la política óptima, es difícil de implementar e impone una sobrecarga significativa. Por otro lado, la política FIFO es muy sencilla de implementar pero su rendimiento es relativamente pobre. A lo largo de los años, los diseñadores de sistemas operativos han intentado un gran número de algoritmos diferentes para aproximarse a los resultados obtenidos por LRU e intentando imponer una sobrecarga más reducida. Muchos de estos algoritmos son variantes del esquema denominado **política del reloj**.

En su forma más sencilla la política del reloj requiere la inclusión de un bit adicional en cada uno de los marcos de página, denominado bit de usado. Cuando una página se trae por primera vez a la memoria, el bit de usado de dicho marco se pone a 1. En cualquier momento que la página vuelva a utilizarse (después de la referencia generada con el fallo de página inicial) su bit de usado se pone a 1. Para el algoritmo de reemplazo de páginas, el conjunto de todas las páginas que son candidatas para reemplazo (de este proceso: ámbito local; toda la memoria principal: ámbito global[6]) se disponen como si se tratase de un *buffer* circular, al cual se asocia un puntero. Cuando se reemplaza una página, el puntero indica el siguiente marco del *buffer* justo después del marco que acaba de actualizarse. Cuando llega el momento de reemplazar una página, el sistema operativo recorre el *buffer* para encontrar un marco con su bit de usado a 0. Cada vez que encuentra un marco con el bit de usado a 1, se reinicia este bit a 0 y se continúa. Si alguno de los marcos del *buffer* tiene el bit de usado a 0 al comienzo de este proceso, el primero de estos marcos que se encuentre se seleccionará para reemplazo. Si todos los marcos tienen el bit a 1, el puntero va a completar un ciclo completo a lo largo del *buffer*, poniendo todo los bits de usado a 0, parándose en la posición original, reemplazando la página en dicho marco. *Véase* que esta política es similar a FIFO, excepto que, en la política del reloj, el algoritmo saltará todo marco con el bit de usado a 1. La política se domina política del reloj debido a que se pueden visualizar los marcos de página como si estuviesen distribuidos a lo largo del círculo. Un gran número de sistemas operativos han empleado alguna variante de esta política sencilla del reloj (por ejemplo, Multics [CORB68]).

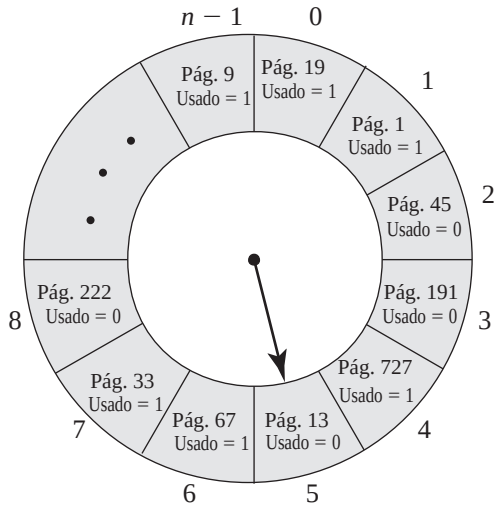
La Figura 8.16 plantea un ejemplo del mecanismo de la política del reloj. Un *buffer* circular con *n* marcos de memoria principal que se encuentran disponibles para reemplazo de la página. Antes del comienzo del reemplazo de una página del *buffer* por la página entrante 727, el puntero al siguiente marco apunta al marco número 2, que contiene la página 45. En este momento la política del reloj comienza a ejecutarse. Debido a que el bit usado de la página 45 del marco 2 es igual a 1, esta página no se reemplaza. En vez de eso, el bit de usado se pone a 0 y el puntero avanza. De forma similar la página 191 en el marco 3 tampoco se remplazará; y su bit de usado se pondrá a 0, avanzando de nuevo el puntero. En el siguiente marco, el marco número 4, el bit de usado está a 0. Por tanto, la página 556 se reemplazará por la página 727. El bit de usado se pone a 1 para este marco y el puntero avanza hasta el marco 5, completando el procedimiento de reemplazo de página.

---

[6] El concepto de ámbito se discute en la subsección «Ámbito de reemplazo», más adelante.



(a) Estado del *buffer* justo ante del reemplazo de una página



(b) Estado del *buffer* justo después del siguiente reemplazo de página

Figura 8.16. Ejemplo de operación de la política de reemplazo del reloj.

El comportamiento de la política del reloj se encuentra ilustrado en la Figura 8.15. La presencia de un asterisco indica que el correspondiente bit de usado es igual a 1, y la flecha indica cuál es la posición actual del puntero. Nótese que la política del reloj intenta proteger los marcos 2 y 5 de un posible reemplazo.

La Figura 8.17 muestra los resultados del experimento realizado por [BAER80], que compara los cuatro algoritmos que se han comentado; se asume que el número de marcos de página asignados a cada proceso es fijo. El resultado se basa en la ejecución de 0,25 x 10^6 referencias en un programa FORTRAN, utilizando un tamaño de página de 256 palabras. Baer ejecutó el experimento con unas reservas de 6, 8, 10, 12, y 14 marcos. Las diferencias entre las cuatro políticas son más palpables

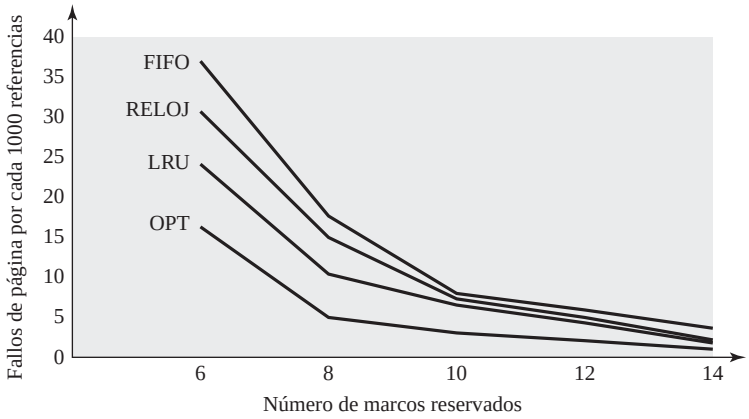


Figura 8.17. Comparativa de algoritmos de reemplazo local con reserva de marcos fija.

cuando el número de marcos reservados es pequeño, estando FIFO por encima en un factor de 2 veces peor que el óptimo. Las cuatro curvas mantienen la misma forma de comportamiento que el ideal mostrado en la Figura 8.11b. Con intención de ejecutar de forma eficiente, sería deseable encontrarse en el lado derecho de la curva (con una tasa de fallos de página pequeña) mientras que al mismo tiempo se mantiene una necesidad de reservar relativamente pocos marcos (hacia el lado izquierdo de la curva). Estas dos restricciones indican que el modo deseable de operación estaría aproximadamente en la mitad de la curva.

[FINK88] tanbién reporta unos resultados prácticamente idénticos, de nuevo mostrando una desviación máxima en torno a un factor de 2. La estrategia de Finkel consistía en simular el efecto de varias políticas en una cadena de referencias a páginas generada sintéticamente de un total de 10.000 referencias seleccionadas dentro del espacio virtual de 100 páginas. Para aproximarse a los efectos del principio de próximidad de referencia, se impuso el uso de una distribución exponencial de probabilidad para hacer referencia a una página en concreto. Finkel indica que se podría concluir que no tiene mucho sentido elaborar algoritmos de reemplazo de páginas cuando sólo hay un factor de 2 en juego. Pero remarca que esta diferencia puede tener un efecto considerable en los requisitos de memoria principal (si se quiere evitar que el rendimiento del sistema operativo se degrade) o para el propio rendimiento del sistema operativo (si se quiere evitar el requisito de una memoria principal mucho mayor).

El algoritmo del reloj también se ha comparado con estos otros algoritmos cuando la reserva de marcos es variable y se aplican ámbitos de reemplazamiento tanto global como local (*véase* la siguiente explicación relativa a las políticas de reemplazo) [CARR81, CARR 84]. El algoritmo del reloj se encuentra muy próximo en rendimiento al LRU.

El algoritmo del reloj puede hacerse más potente incrementando el número de bits que utiliza[7]. En todos los procesadores que soportan paginación, se asocia un bit de modificado a cada una de las páginas de la memoria principal y por tanto con cada marco de la memoria principal. Este bit es necesario debido a que, cuando una página se ha modificado, no se la puede reemplazar hasta que se haya escrito de nuevo a la memoria secundaria. Podemos sacar provecho de este bit en el algoritmo del reloj de la siguiente manera. Si tenemos en cuenta los bits de usado y modificado, cada marco de página cae en una de las cuatro categorías siguientes:

[7] Por otro lado, si se reduce el número de bits utilizados a 0, el algoritmo del reloj degenera a uno de tipo FIFO.

- No se ha accedido recientemente, no modificada ($u = 0$; $m = 0$)
- Accedida recientemente, no modificada ($u = 1$; $m = 0$)
- No se ha accedido recientemente, modificada ($u = 0$; $m = 1$)
- Accedida recientemente, modificada ($u = 1$; $m = 1$)

Con esta clasificación, el algoritmo del reloj puede actuar de la siguiente manera:

1. Comenzando por la posición actual del puntero, recorremos el *buffer* de marcos. Durante el recorrido, no se hace ningún cambio en el bit de usado. El primer marco que se encuentre con ($u = 0$; $m = 0$) se selecciona para reemplazo.
2. Si el paso 1 falla, se recorre el *buffer* de nuevo, buscando un marco con ($u = 0$; $m = 1$). El primer marco que se encuentre se seleccionará para reemplazo. Durante el recorrido, se pondrá el bit de usado a 0 en cada uno de los marcos que se vayan saltando.
3. Si el paso 2 también falla, el puntero debe haber vuelto a la posición original y todo los marcos en el conjunto tendrán el bit de usado a 0. Se repite el paso 1 y, si resulta necesario el paso 2. Esta vez, se encontrará un marco para reemplazo.

En resumen, el algoritmo de reemplazo de páginas da vueltas a través de todas las páginas del *buffer* buscando una que no se haya modificado desde que se ha traído y que no haya sido accedida recientemente. Esta página es una buena opción para reemplazo y tiene la ventaja que, debido a que no se ha modificado, no necesita escribirse de nuevo en la memoria secundaria. Si no se encuentra una página candidata en la primera vuelta, el algoritmo da una segunda vuelta al *buffer*, buscando una página modificada que no se haya accedido recientemente. Incluso aunque esta página tenga que escribirse antes de ser remplazada, debido al principio de proximidad de referencia, puede no necesitarse de nuevo en el futuro próximo. Si esta segunda pasada falla, todos los marcos en el *buffer* se encuentran marcados como si no hubiesen sido accedidos recientemente y se realiza una tercera pasada.

Esta estrategia se ha utilizado en el esquema de memoria virtual de las versiones antiguas de los Macintosh [GOLD89], mostrados en la Figura 8.18. La ventaja de este algoritmo sobre el algoritmo del reloj básico es que se les otorga preferencia para el reemplazo a las páginas que no se han modificado. Debido a que la página que se ha modificado debe escribirse antes del reemplazo, hay un ahorro de tiempo inmediato.

***Buffering* páginas**. A pesar de que las políticas LRU y del reloj son superiores a FIFO, ambas incluyen una complejidad y una sobrecarga que FIFO no sufre. Adicionalmente, existe el aspecto relativo a que el coste de reemplazo de una página que se ha modificado es superior al de una que no lo ha sido, debido a que la primera debe escribirse en la memoria secundaria.

Una estrategia interesante que puede mejorar el rendimiento de la paginación y que permite el uso de una política de reemplazo de páginas sencilla es el *buffering* de páginas. La estrategia más representativa de este tipo es la usaba por VAX VMS. El algoritmo de reemplazo de páginas es el FIFO sencillo. Para mejorar el rendimiento, una página remplazada no se pierde sino que se asigna a una de las dos siguientes listas: la lista de páginas libres si la página no se ha modificado, o la lista de páginas modificadas si lo ha sido. *Véase* que la página no se mueve físicamente de la memoria; al contrario, la entrada en la tabla de páginas para esta página se elimina y se coloca bien en la lista de páginas libres o bien en la lista de páginas modificadas.

La lista de páginas libres es una lista de marcos de páginas disponibles para lectura de nuevas páginas. VMS intenta mantener un pequeño número de marcos libres en todo momento. Cuando una

página se va a leer, se utiliza el marco de página en la cabeza de esta lista, eliminando la página que estaba. Cuando se va a reemplazar una página que no se ha modificado, se mantiene en la memoria ese marco de página y se añade al final de la lista de páginas libres. De forma similar, cuando una página modificada se va a escribir y reemplazar, su marco de página se añade al final de la lista de páginas modificadas.

El aspecto más importante de estas maniobras es que la página que se va a reemplazar se mantiene en la memoria. De forma que si el proceso hace referencia a esa página, se devuelve al conjunto residente del proceso con un bajo coste. En efecto, la lista de páginas modificadas y libres actúa como una *cache* de páginas. La lista de páginas modificadas tiene también otra función útil: las páginas modificadas se escriben en grupos en lugar de una a una. Esto reduce de forma significativa el número de operaciones de E/S y por tanto el tiempo de acceso disco.

Una versión simple de esta estrategia de *buffering* de páginas la implementa el sistema operativo Mach [RASH88]. En este caso, no realiza distinción entre las páginas modificadas y no modificadas.

Política de reemplazo y tamaño de la *cache*. Como se ha comentado anteriormente, cuando el tamaño de la memoria principal crece, la proximidad de referencia de las aplicaciones va a decrecer. En compensación, los tamaños de las caches pueden ir aumentando. Actualmente, grandes tamaños de *caches*, incluso de varios megabytes, son alternativas de diseño abordables [BORG90]. Con una

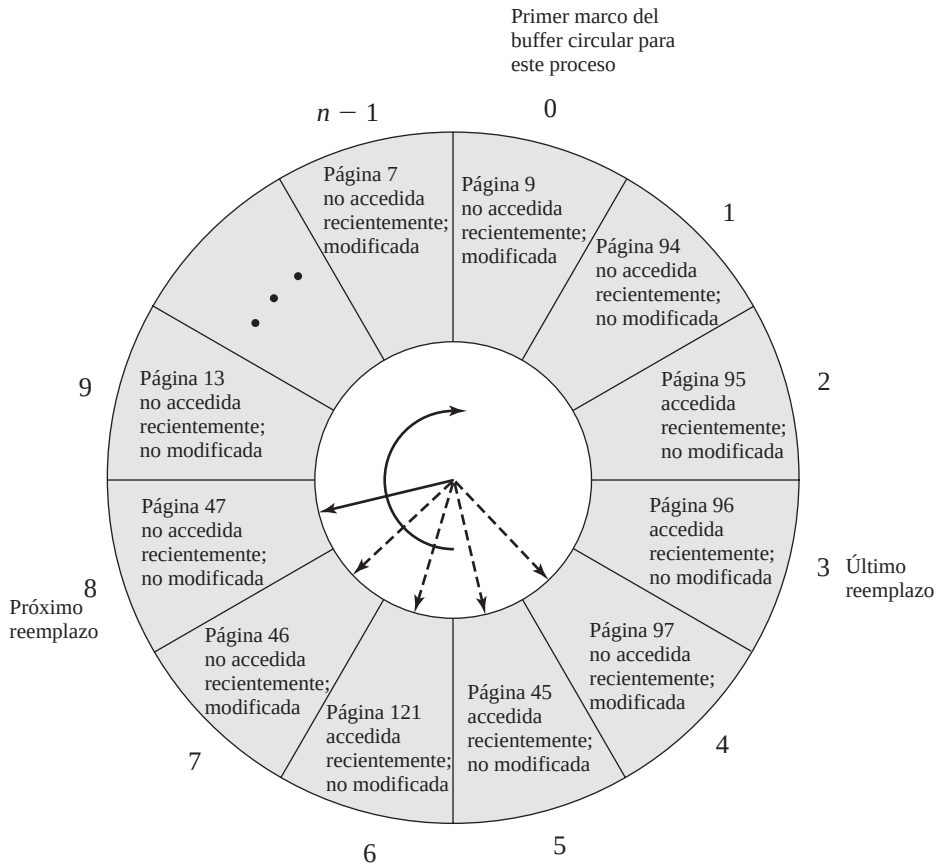


Figura 8.18. El algoritmo de reemplazo de páginas del reloj [GOLD89].

cache de gran tamaño, el reemplazo de páginas de la memoria virtual puede tener un impacto importante en el rendimiento. Si el marco de página destinado al reemplazo está en *cache*, entonces el bloque de *cache* se pierde al mismo tiempo que la página que lo contiene.

En sistemas que utilizan algún tipo de *buffering* de páginas, se puede mejorar el rendimiento de la cache añadiendo a la política de reemplazo de páginas una política de ubicación de páginas en el *buffer* de páginas. La mayoría de sistemas operativos ubican las páginas seleccionando un marco procedente del *buffer* de páginas, de forma arbitraria; utilizando habitualmente una disciplina de tipo FIFO. El estudio reportado en [KESS92] muestra que una estrategia de reemplazo de páginas cuidadosa puede significar entre un 10 y un 20% menos de fallos de *cache* si se compara con un reemplazo simple.

En [KESS92] se examinan diferentes algoritmos de reemplazo de páginas. Estos detalles están más allá del ámbito de este libro, debido a que dependen de detalles de la estructura de las *caches* y sus políticas. La esencia de estas estrategias consiste en traer páginas consecutivas a la memoria principal de forma que se minimice el número de marcos de página que se encuentran proyectados en las mismas ranuras de la cache.

GESTIÓN DEL CONJUNTO RESIDENTE

Tamaño del conjunto residente. Con la memoria virtual paginada, no es necesario, y en algunos casos no es ni siquiera posible, traer todas las páginas de un proceso a la memoria principal para preparar su ejecución. Debido a que el sistema operativo debería saber cuántas páginas traerse, esto es, cuánta memoria principal debería reservar para un proceso en particular. Diferentes factores entran en juego:

- Cuanto menor es la cantidad de memoria reservada para un proceso, mayor es el número de procesos que pueden residir en la memoria principal a la vez. Esto aumenta la probabilidad de que el sistema operativo pueda encontrar al menos un proceso listo para ejecutar en un instante dado, así por tanto, reduce el tiempo perdido debido al *swapping*.
- Si el conjunto de páginas de un proceso que están en memoria es relativamente pequeño, entonces, en virtud del principio de proximidad de referencia, la posibilidad de un fallo de página es mayor (*véase* Figura 8.11.b).
- Más allá de un determinado tamaño, la reserva de más memoria principal para un determinado proceso no tendrá un efecto apreciable sobre la tasa de fallos de página de dicho proceso, debido al principio de proximidad de referencia.

Teniendo en cuenta estos factores, se pueden encontrar dos tipos de políticas existentes en los sistemas operativos contemporáneos. La política de **asignación fija** proporciona un número fijo de marcos de memoria principal disponibles para ejecución. Este número se decide en el momento de la carga inicial de proceso (instante de creación del proceso) y se puede determinar en base al tipo de proceso (interactivo, por lotes, tipo de aplicación) o se puede basar en las guías proporcionadas por el programador o el administrador del sistema. Con la política de asignación fija, siempre que se produzca un fallo de página del proceso en ejecución, la página que se necesite reemplazará una de las páginas del proceso.

Una política de **asignación variable** permite que se reserven un número de marcos por proceso que puede variar a lo largo del tiempo de vida del mismo. De forma ideal, a un proceso que esté causando una tasa de fallos de página relativamente alta de forma continua, indicativo de que el principio proximidad de referencia sólo se aplica de una forma relativamente débil para este proceso, se le otorgarán marcos de página adicionales para reducir esta tasa de fallos; mientras tanto, a un proceso

con una tasa de fallos de páginas excepcionalmente baja, indicativo de que el proceso sigue un comportamiento bien ajustado al principio de proximidad de referencia, se reducirán los marcos reservados, con esperanza de que esto no incremente de forma apreciable la tasa de fallos. El uso de políticas de asignación variable se basa en el concepto de ámbito de reemplazo, como se explicará en la siguiente subsección.

La política de asignación variable podría parecer más potente. Sin embargo, las dificultades de esta estrategia se deben a que el sistema operativo debe saber cuál es el comportamiento del proceso activo. Esto requiere, de forma inevitable, una sobrecarga software por parte del sistema operativo y depende de los mecanismos hardware proporcionados por la propia plataforma del procesador.

Ámbito de reemplazo. La estrategia del ámbito de reemplazo se puede clasificar en global y local. Ambos tipos de políticas se activan por medio de un fallo de página cuando no existen marcos de página libres. Una política de **reemplazo local** selecciona únicamente entre las páginas residentes del proceso que ha generado el fallo de página. Para la identificación de la página a reemplazar en una política de **reemplazo global** se consideran todas las páginas en la memoria principal que no se encuentren bloqueadas como candidatos para el reemplazo, independientemente de a qué proceso pertenezca cada página en particular. Mientras que las políticas locales son más fáciles de analizar, no existe ninguna evidencia convincente de que proporcionen un rendimiento mejor que las políticas globales, que son más atractivas debido a la simplicidad de su implementación con una sobrecarga mínima [CARR84, MAEK87].

Existe una correlación entre el ámbito de reemplazo y el tamaño del conjunto residente (Tabla 8.4). Un conjunto residente fijo implica automáticamente una política de reemplazo local: para mantener el tamaño de conjunto residente, al reemplazar una página se debe eliminar de la memoria principal otra del mismo proceso. Una política de asignación variable puede emplear claramente la política de reemplazo global: el reemplazo de una página de un proceso en la memoria principal por la de otro causa que la asignación de memoria para un proceso crezca en una página mientras que disminuye la del otro. Se verá también que la asignación variable y el reemplazo local son una combinación válida. Se examinan ahora estas tres combinaciones.

Asignación fija, ámbito local. En este caso, se parte de un proceso que se encuentra en ejecución en la memoria principal con un número de marcos fijo. Cuando se da un fallo de página, el sistema operativo debe elegir una página entre las residentes del proceso actual para realizar el reemplazo. Se utilizarían los algoritmos de reemplazo que se han visto en la sección precedente.

Con la política de asignación fija, es necesario decidir por adelantado la cantidad de espacio reservado para un proceso determinado. Esto se puede hacer en base al tipo de aplicación y al tamaño del programa. Las desventajas de esta estrategia son de dos tipos: si las reservas resultan ser demasiado pequeñas, va a haber una alta tasa de fallos de página, haciendo que el sistema multiprogramado completo se ralentice. Si las reservas, por contra, resultan demasiado grandes, habrá muy pocos programas en la memoria principal y habrá mucho tiempo del procesador ocioso o mucho tiempo perdido en *swapping*.

Asignación variable, ámbito global. Esta combinación es, probablemente, la más sencilla de implementar y ha sido adoptada por un gran número de sistemas operativos. En un momento determinado, existen un número de procesos determinado en la memoria principal, cada uno de los cuales tiene una serie de marcos asignados. Normalmente, el sistema operativo también mantiene una lista de marcos libres. Cuando sucede un fallo de página, se añade un marco libre al conjunto residente de un proceso y se trae la página a dicho marco. De esta forma, un proceso que sufra diversos fallos de página crecerá gradualmente en tamaño, lo cual debería reducir la tasa de fallos de página global del sistema.

Table 8.4. Gestión del conjunto residente.

	Reemplazo Local	Reemplazo Global
Asignación Fija	• El número de marcos asignados a un proceso es fijo. • Las páginas que se van a reemplazar se eligen entre los marcos asignados al proceso.	• No es posible
Asignación Variable	• El número de marcos asignados a un proceso pueden variarse de cuando en cuando. • Las páginas que se van a reemplazar se eligen entre los marcos asignados al proceso.	• Las páginas que se van a reemplazar se eligen entre todos los marcos de la memoria principal esto hace que el tamaño del conjunto residente de los procesos varíe.

La dificultad de esta estrategia se encuentra en la elección de los reemplazos cuando no existen marcos libres disponibles, el sistema operativo debe elegir una página que actualmente se encuentra en la memoria para reemplazarla. Esta selección se lleva a cabo entre todos los marcos que se encuentran en la memoria principal, a excepción de los marcos bloqueados como son los usados por el núcleo. Utilizando cualquiera de las políticas vistas en la sección anterior, se tiene que la página seleccionada para reemplazo puede pertenecer a cualquiera de los procesos residentes; no existe ninguna disciplina predeterminada que indique qué proceso debe perder una página de su conjunto residente. Así pues, el proceso que sufre la reducción del tamaño de su conjunto residente no tiene porqué ser el óptimo.

Una forma de encontrar una contraprestación a los problemas de rendimiento potenciales de la asignación variable con reemplazo de ámbito global, se centran en el uso de *buffering* de páginas. De esta forma, la selección de una página para que se reemplace no es tan significativa, debido a que la página se puede reclamar si se hace referencia antes de que un nuevo bloque de páginas se sobrescriba.

Asignación variable, ámbito local. La asignación variable con reemplazo de ámbito local intenta resolver los problemas de la estrategia de ámbito global. Se puede resumir en lo siguiente:

1. Cuando se carga un nuevo proceso en la memoria principal, se le asignan un cierto número de marcos de página a su conjunto residente, basando en el tipo de aplicación, solicitudes del programa, u otros criterios. Para cubrir esta reserva se utilizará la paginación adelantada o la paginación por demanda.
2. Cuando ocurra un fallo página, la página que se seleccionará para reemplazar pertenecerá al conjunto residente del proceso que causó el fallo.
3. De vez en cuando, se reevaluará la asignación proporcionada a cada proceso, incrementándose o reduciéndose para mejorar al rendimiento.

Con esta estrategia, las decisiones relativas a aumentar o disminuir el tamaño del conjunto residente se toman de forma más meditada y se harán contando con los indicios sobre posibles demandas futuras de los procesos que se encuentran activos. Debido a la forma en la que se realiza esta valoración, esta estrategia es mucho más compleja que la política de reemplazo global simple. Sin embargo, puede llevar a un mejor rendimiento.

Los elementos clave en la estrategia de asignación variable con ámbito local son los criterios que se utilizan para determinar el tamaño del conjunto residente y la periodicidad de estos cambios. Una

estrategia específica que ha atraído mucha atención en la literatura es la denominada **estrategia del conjunto de trabajo**. A pesar de que la estrategia del conjunto de trabajo pura sería difícil implementar, es muy útil examinarla como referencia para las comparativas.

El conjunto de trabajo es un concepto acuñado y popularizado por Denning [DENN68, DENN70, DENN80b]; y ha tenido un profundo impacto en el diseño de la gestión de la memoria virtual. El conjunto de trabajo con parámetro Δ para un proceso en el tiempo virtual t, $W(t, \Delta)$ es el conjunto de páginas del proceso a las que se ha referenciado en las últimas Δ unidades de tiempo virtual.

El tiempo virtual se define de la siguiente manera. Considérese la secuencia de referencias a memoria, $r(1), r(2), \ldots$, en las cuales $r(i)$ es la página que contiene la i-esima dirección virtual generada por dicho proceso. El tiempo se mide en referencias a memoria; así $t=1,2,3\ldots$ mide el tiempo virtual interno del proceso.

Se considera que cada una de las dos variables de W. La variable Δ es la ventana de tiempo virtual a través de la cual se observa al proceso. El tamaño del conjunto trabajo será una función nunca decreciente del tamaño de ventana. El resultado que se muestra en la Figura 8.19 (en base a [BACH86]), demuestra la secuencia de referencias a páginas para un proceso. Los puntos indican las unidades de tiempo en las cuales el conjunto de trabajo no cambia. Nótese que para mayor tamaño ventana, el tamaño del conjunto trabajo también es mayor. Esto se puede expresar en la siguiente relación:

$$W(t, \Delta+1) \supseteq W(t, \Delta)$$

El conjunto trabajo es también una función del tiempo. Si un proceso ejecuta durante Δ unidades de tiempo, y terminando el tiempo utiliza una única página, entonces $|W(t, \Delta)|=1$. Un conjunto trabajo también puede crecer hasta llegar a las N páginas del proceso si se accede rápidamente a muchas páginas diferentes y si el tamaño de la ventana lo permite. De esta forma,

$$1 \leq |W(t, \Delta)| \leq min(\Delta,N)$$

La Figura 8.20 indica cómo puede variar el tamaño del conjunto de trabajo a lo largo del tiempo para un valor determinado de Δ. Para muchos programas, los periodos relativamente estables del tamaño de su conjunto de trabajo se alternan con periodos de cambio rápido. Cuando un proceso comienza a ejecutar, de forma gradual construye su conjunto de trabajo a medida que hace referencias a nuevas páginas. Esporádicamente, debido al principio de proximidad de referencia, el proceso deberá estabilizarse sobre un conjunto determinado de páginas. Los periodos transitorios posteriores reflejan el cambio del programa a una nueva región de referencia. Durante la fase de transición algunas páginas del antiguo conjunto de referencia permanecerán dentro de la ventana, Δ, causando un rápido incremento del tamaño del conjunto de trabajo a medida que se van referenciando nuevas páginas. A medida que la ventana se desplaza de estas referencias, el tamaño del conjunto de trabajo se reduce hasta que contiene únicamente aquellas páginas de la nueva región de referencia.

Este concepto del conjunto de trabajo se puede usar para crear la estrategia del tamaño conjunto residente:

1. Monitorizando el conjunto de trabajo de cada proceso.
2. Eliminando periódicamente del conjunto residente aquellas páginas que no se encuentran en el conjunto de trabajo, esto en esencia es una política LRU.
3. Un proceso puede ejecutar sólo si su conjunto trabajo se encuentra en la memoria principal (por ejemplo, si su conjunto residente incluye su conjunto de trabajo).

Secuencia de referencias a páginas	Tamaño de ventana, Δ			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	15 18 23	15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

Figura 8.19. Conjunto de trabajo de un proceso definido por el tamaño de ventana.

Esta estrategia funciona debido a que parte de un principio aceptado, el principio de proximidad de referencia, y lo explota para conseguir una estrategia de la gestión de la memoria que minimice los fallos de página. Desgraciadamente, persisten varios problemas en la estrategia del conjunto trabajo:

1. El pasado no siempre predice el futuro. Tanto el tamaño como la pertenencia al conjunto de trabajo cambian a lo largo del tiempo (por ejemplo, *véase* la Figura 8.20).
2. Una medición verdadera del conjunto de trabajo para cada proceso no es practicable. Sería necesario, por cada proceso, asignar un sello de tiempo a cada referencia a una página que asigne el tiempo virtual del proceso y que mantenga una lista ordenada por tiempo de las páginas de cada uno de ellos.
3. El valor óptimo de Δ es desconocido y en cualquier caso puede variar.

Sin embargo el espíritu de esta estrategia sí es válido, el gran número de sistemas operativos intentan aproximarse a la estrategia del conjunto de trabajo. La forma de hacer esto es centrarse no en las referencias exactas a páginas y sí en la tasa de fallos de página. Como se ilustra en la Figura 8.11b, la tasa de fallos de páginas cae a medida que se incrementa el tamaño del conjunto residente de un proceso. El tamaño del conjunto de trabajo debe caer en un punto de esta curva, indicado por *W* en la figura. Por tanto, en lugar de monitorizar el tamaño del conjunto trabajo de forma directa se pueden alcanzar resultados comparables monitorizando la tasa de fallos de página. El razonamiento es el siguiente: si la tasa

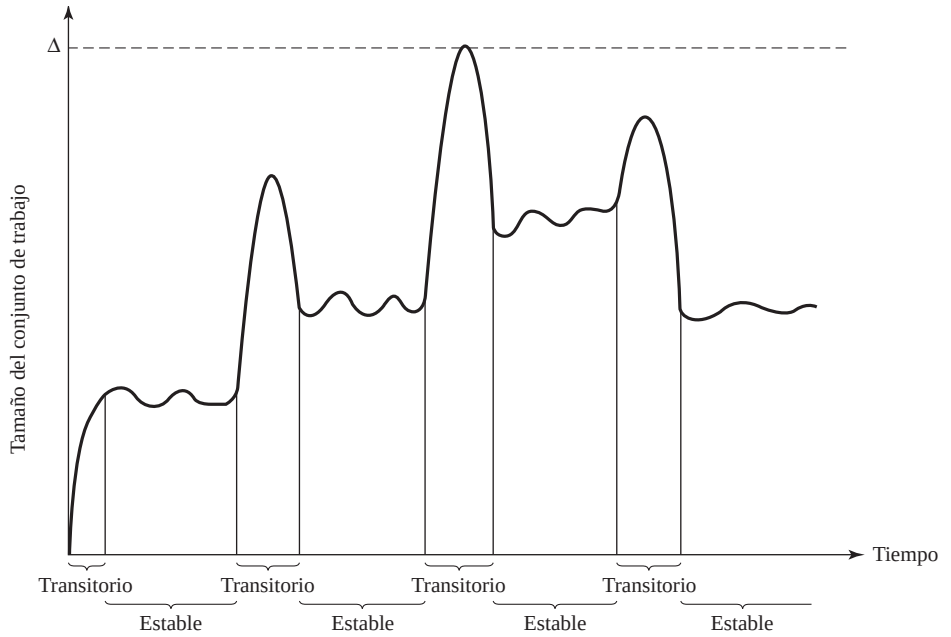


Figura 8.20. Gráfico típico del tamaño del conjunto de trabajo [MAEK87].

de fallos de páginas de un proceso está por debajo de un determinado límite, el sistema de forma global se puede beneficiar de asignar un tamaño de conjunto residente menor para este proceso (debido a que se dispone de un mayor número de marcos de páginas libres para otros procesos) sin dañar al proceso en cuestión (causando un incremento de sus fallos de página). Si la tasa de fallos de página entonces está por encima de un umbral máximo, el proceso puede beneficiarse de un incremento en el tamaño de su conjunto residente (y que así se produzca un menor número de fallos) sin degradar el sistema.

Un algoritmo que sigue esta estrategia es el algoritmo de **frecuencia de fallos de página** (*page fault frequency* – PFF) [CHU72, GUPT78]. El algoritmo necesita un bit de usado que se encuentre asociado a cada página de memoria. Este bit se pondrá a 1 cuando se haya accedido a la página. Cuando se produce un fallo de página, el sistema operativo anotará el tiempo virtual desde el último fallo de página para dicho proceso; esto se puede realizar manteniendo un contador de las referencias a páginas. Se fija un umbral *F*. Si la diferencia de tiempo con el último fallo de página es menor que éste, entonces se añade una página al conjunto residente del proceso. En otro caso, se descartan todas los páginas con el bit de usado a 0, y se reduce el tamaño del conjunto residente de forma acorde. Mientras tanto, se pone a 0 el bit de usado del resto de las páginas. Esta estrategia se puede refinar usando dos umbrales: un umbral máximo que se utiliza para disparar el crecimiento del conjunto residente, y un límite inferior que se utiliza para disparar la reducción de tamaño del conjunto residente.

El tiempo entre fallos de página es recíproco a la tasa de fallos de página. A pesar de ello parecería mejor mantener una medida a lo largo de la ejecución de la tasa de fallos de página, sin embargo la utilización de una medida de tiempo sencilla es un compromiso razonable que permite que las decisiones relativas al tamaño del conjunto residente se basen en la tasa de fallos de página. Si una estrategia de este estilo se complementa con el *buffering* de páginas, se puede conseguir como resultado un rendimiento bastante bueno.

Sin embargo, existe un fallo grave en la estrategia adoptada por PFF, que hace que su comportamiento no sea bueno durante los periodos transitorios cuando se produce un desplazamiento hacia

una nueva región de referencia. Con PFF ninguna página sale del conjunto residente antes de que hayan pasado *F* unidades de tiempo virtual desde su última referencia. Durante estos periodos entre dos regiones de referencia, la rápida sucesión de fallos de página hace que el conjunto residente del proceso crezca antes de que las páginas de la antigua región de referencia se expulsen; los súbitos picos en las solicitudes de memoria pueden producir desactivaciones y reactivaciones de procesos innecesarias, que se corresponden con cambios de proceso y sobrecargas de *swapping* no deseables.

Una estrategia que intenta manejar este fenómeno de transición entre regiones de referencia con una sobrecarga relativamente baja comparado con PFF es la política de **conjunto de trabajo con muestreo sobre intervalos variables** (*variable-interval sampled working set* – VSWS) [FERR83]. La política VSWS evalúa el conjunto de trabajo del proceso en instantes de muestreo basados en el tiempo virtual transcurrido. Al comienzo del intervalo de muestreo, los bits de usado de las páginas residentes de procesos se ponen a 0; al final, sólo las páginas a las que se ha hecho referencia durante el intervalo mantendrán dicho bit a 1; estas páginas se mantienen dentro del conjunto residente del proceso a lo largo del siguiente intervalo, mientras que las otras se descartan. De esta forma el tamaño del conjunto residente solamente decrecerá al final del intervalo. Durante cada intervalo, todas las páginas que han causado fallo se añaden al conjunto residente; de esta forma el conjunto residente mantiene un tamaño fijo o crece durante el intervalo.

La política VSWS toma tres diferentes parámetros:

M: la duración mínima del intervalo de muestreo

L: la duración máxima del intervalo de muestreo

Q: el número de fallos de página que se permite que ocurran entre dos instantes de muestreo

La política VSWS es la siguiente:

1. Si el tiempo virtual entre el último muestreo alcanza *L*, se suspende el proceso y se analizan los bits de usado.
2. Si, antes de que el tiempo virtual transcurrido llegue a *L*, ocurren *Q* fallos de página,
   a) Si el tiempo virtual desde el último muestreo es menor que *M*, se espera hasta que el tiempo virtual transcurrido alcance dicho valor para suspender el proceso y analizar los bits de usado.
   b) Si el tiempo virtual desde el último muestreo es mayor o igual a *M*, se suspende el proceso y se analizan sus bits de usado.

Los valores de los parámetros se toman de forma que el muestreo se dispare habitualmente cuando ocurre el fallo de página *Q* después del último muestreo (caso 2b). Los otros dos parámetros (*M* y *L*) proporcionan fronteras de protección para condiciones excepcionales. La política VSWS intenta reducir el pico de solicitudes de memoria causadas por una transición abrupta entre dos áreas de referencia, incrementando para ello la frecuencia de muestreo, y por tanto la tasa a la cual las páginas no utilizadas se descartan del conjunto residente, cuando la tasa de fallos de página, se incrementa. La experiencia con esta técnica en el sistema operativo del *mainframe* de Bull, GCOS 8, indica que este mecanismo es tan sencillo de implementar como PFF y mucho más efectivo [PIZZ89].

POLÍTICA DE LIMPIEZA

La política de limpieza es la opuesta a la política de recuperación; se encarga de determinar cuándo una página que está modificada se debe escribir en memoria secundaria. Las dos alternativas más co-

munes son la limpieza bajo demanda y la limpieza adelantada. Con la **limpieza bajo demanda**, una página se escribe a memoria secundaria sólo cuando se ha seleccionado para un reemplazo. En la política de la **limpieza adelantada** se escribe las páginas modificadas antes de que sus marcos de páginas se necesiten, de forma que las páginas se puedan escribir en lotes.

Existe un peligro en perseguir cualquiera de estas dos políticas hasta el extremo. Con la limpieza adelantada, una página se escribe aunque continúe en memoria principal hasta que el algoritmo de reemplazo páginas indique que debe eliminarse. La limpieza adelantada permite que las páginas se escriban en lotes, pero tiene poco sentido escribir cientos o miles de páginas para darnos cuenta de que la mayoría de ellas van a modificarse de nuevo antes de que sean reemplazadas. La capacidad de transferencia de la memoria secundaria es limitada y no se debe malgastar con operaciones de limpieza innecesarias.

Por otro lado, con la limpieza bajo demanda, la escritura de una página modificada colisiona con, y precede a, la lectura de una nueva página. Esta técnica puede minimizar las escrituras de páginas, pero implica que el proceso que ha sufrido un fallo página debe esperar a que se completen dos transferencias de páginas antes de poder desbloquearse. Esto implica una reducción en la utilización del procesador.

Una estrategia más apropiada incorpora *buffering* páginas. Esto permite adoptar la siguiente política: limpiar sólo las páginas que son reemplazables, pero desacoplar las operaciones de limpieza y reemplazo. Con *buffering* de páginas, las páginas reemplazadas pueden ubicarse en dos listas: modificadas y no modificadas. Las páginas en la lista de modificadas pueden escribirse periódicamente por lotes y moverse después a la lista de no modificadas. Una página en la lista de no modificadas puede, o bien ser reclamaba si se referencia, o perderse cuando su marco se asigna a otra página.

CONTROL DE CARGA

El control de carga determina el número de procesos que residirán en la memoria principal, eso se denomina el grado de multiprogramación. La política de control de carga es crítica para una gestión de memoria efectiva. Si hay muy pocos procesos residentes a la vez, habrá muchas ocasiones en las cuales todos los procesos se encuentren bloqueados, y gran parte del tiempo se gastarán realizando *swapping*. Por otro lado, si hay demasiados procesos residentes, entonces, de media, el tamaño de conjunto residente de cada proceso será poco adecuado y se producirán frecuentes fallos de página. El resultado es el trasiego (*thrashing*).

Grado de multiprogramación. El trasiego se muestra en la Figura 8.21. A medida que el nivel de multiprogramación aumenta desde valores muy pequeños, cabría esperar que la utilización del procesador aumente, debido a que hay menos posibilidades de que los procesos residentes se encuentren bloqueados. Sin embargo, se alcanza un punto en el cual el tamaño de conjunto residente promedio no es adecuado. En este punto, el número de fallos de páginas se incrementa de forma dramática, y la utilización del procesador se colapsa.

Existen numerosas formas de abordar este problema. Un algoritmo del conjunto de trabajo o de frecuencia de fallos de página incorporan, de forma implícita, control de carga. Sólo aquellos procesos cuyo conjunto residente es suficientemente grande se les permite ejecutar. En la forma de proporcionar el tamaño de conjunto residente necesario para cada proceso activo, se encuentra la política que automática y dinámicamente determina el número de programas activos.

Otra estrategia, sugerida por Denning [DENN80b], se conoce como el *criterio de L = S*, que ajusta el nivel de multiprogramación de forma que el tiempo medio entre fallos de página se iguala al tiempo medio necesario para procesar un fallo de página. Los estudios sobre el rendimiento indican

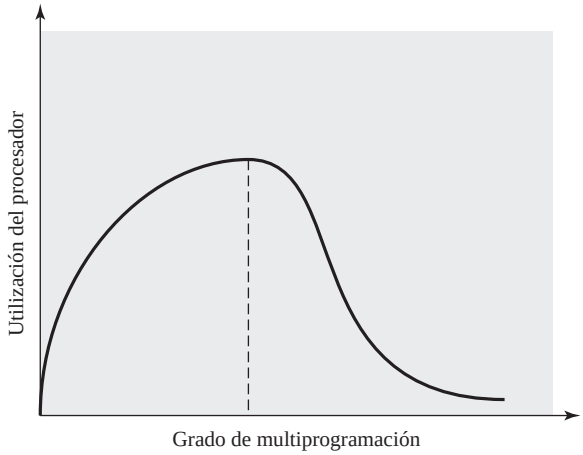


Figura 8.21. Efectos de la multiprogramación.

que éste es el punto en el cual la utilización del procesador es máxima. Una política que presenta un efecto similar, propuesta en [LERO76], es el *criterio del 50%*, que intenta mantener la utilización del dispositivo de paginación aproximadamente al 50%. De nuevo, los estudios sobre el rendimiento indican que éste es el punto para el cual la utilización del procesador es máxima.

Otra alternativa es adaptar el algoritmo de reemplazo de páginas del reloj descrito anteriormente (Figura 8.16). [CARR84] describe la técnica, usando un ámbito global, que implica monitorizar la tasa a la cual el puntero recorre el *buffer* circular de marcos. Si la tasa está por debajo de un nivel de umbral determinado, éste indica alguna (o las dos) circunstancias siguientes:

1. Si están ocurriendo pocos fallos de página, que implican pocas peticiones para avanzar este puntero.
2. Por cada solicitud, el número de marcos promedio que se recorren por el puntero es pequeño, que indica que hay muchas páginas residentes a las que no se hace referencia y que son realmente reemplazables.

En ambos casos, el grado de multiprogramación puede incrementarse con seguridad. Por otro lado, si la tasa de recorrido circular del puntero supera un umbral máximo, indica que la tasa de fallos de página es alta o que hay dificultad para encontrar páginas reemplazables, lo cual implica que el grado de multiprogramación es demasiado alto.

Suspensión de procesos. Si se va a reducir el grado de multiprogramación, uno o más de los procesos actualmente residentes deben suspenderse (enviarlos a *swap*). [CARR84] proporciona seis posibilidades:

- **Procesos con baja prioridad.** Esto implementa una decisión de la política de activación y no se encuentra relacionada con cuestiones de rendimiento.
- **Procesos que provoca muchos fallos.** La razón es que hay una gran probabilidad de que la tarea que causa los fallos no tenga su conjunto de trabajo residente, y el rendimiento sufrirá menos si dicha tarea se suspende. Adicionalmente, esta elección trae una ventaja asociada debido que se bloquea un proceso que estaría a punto de bloquearse de cualquier manera, y que si se elimina se evita por tanto la sobrecarga del reemplazo de páginas y la operación de E/S.

- **Proceso activado hace más tiempo.** Éste es el proceso que tiene menor probabilidad de tener su conjunto de trabajo residente.
- **Proceso con el conjunto residente de menor tamaño.** Éste requerirá un menor esfuerzo para cargarse de nuevo. Sin embargo, penaliza a aquellos programas con una proximidad de referencias muy fuerte.
- **Proceso mayor.** Éste proporciona un mayor número de marcos libres en una memoria que se encuentra sobrecarga, haciendo que futuras desactivaciones sean poco probables a corto plazo.
- **Proceso con la mayor ventana de ejecución restante.** En la mayoría de esquemas de activación de procesos, un proceso sólo puede ejecutarse durante una determinada rodaja de tiempo antes de recibir la interrupción y situarse al final de la lista de Listos. Esta estrategia se aproxima a la disciplina de activación de primero el proceso con menor tiempo en ejecución.

Es como en otras muchas áreas del diseño de sistemas operativos, la selección de la política más apropiada es una cuestión a considerar y depende de muchos otros factores de diseño en el sistema operativo así como de las características de los programas que se ejecutarán.

8.3. GESTIÓN DE MEMORIA DE UNIX Y SOLARIS

Debido a que UNIX pretende ser un sistema independiente de la máquina, su esquema de gestión de memoria variará de un sistema a otro. En las primeras versiones, UNIX utilizaba particionamiento variable sin ningún esquema de memoria virtual. Las implantaciones actuales de UNIX y Solaris utilizan la memoria virtual paginada.

En SVR4 y Solaris, existen dos esquemas de gestión de memoria separados. El **sistema de paginación** proporciona las funcionalidades de la memoria virtual para asignar marcos de página en la memoria principal a los diferentes procesos y también asignar marcos de página a *buffers* de bloques de disco. A pesar de que éste es un esquema de gestión de memoria efectivo para los procesos de usuario y la E/S de disco, un esquema de la memoria virtual paginada es menos apropiado para gestionar la asignación de memoria del núcleo. Por esas cuestiones, se utiliza un **asignador de memoria del núcleo**. Se van a examinar estos dos mecanismos en orden.

SISTEMA DE PAGINACIÓN

Estructuras de datos. Para la memoria virtual paginada, UNIX utiliza varias estructuras de datos que, con pequeñas diferencias, son independientes de la máquina (Figura 8.22 y Tabla 8.5):

- **Tabla de páginas.** Habitualmente, habrá una tabla de páginas por proceso, con una entrada por cada página de memoria virtual de dicho proceso.
- **Descriptor de bloques de disco.** Asociado a cada página del proceso hay una entrada en esta tabla que indica la copia en disco de la página virtual.
- **Tabla de datos de los marcos de página.** Describe cada marco de memoria real y se indexa por medio de un número marco. El algoritmo de reemplazo usa esta tabla.
- **Tabla de utilización de *swap*.** Existe una tabla de uso de *swap* por cada dispositivo de intercambio, con una entrada por cada página de dicho dispositivo.

La mayoría de los campos definidos en la Tabla 8.5 proporcionan su propia descripción. Unos pocos de ellos requieren una explicación adicional. El campo Edad en la entrada de la tabla de pági-

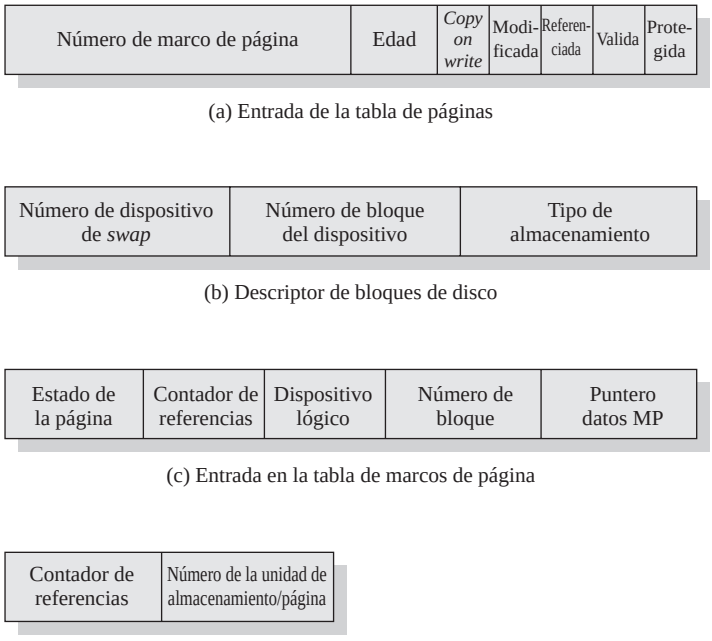


Figura 8.22. Formatos de gestión de memoria de UNIX SVR4.

nas es un indicador de cuánto tiempo hace que el programa no ha hecho referencia a este marco. Sin embargo, el número de bits y la frecuencia de actualización de este campo son dependientes de la implementación. Por tanto, no hay un uso universal de este campo por parte de UNIX para las políticas de reemplazo de páginas.

El campo de Tipo de Almacenamiento en el descriptor de bloques de disco se necesita por la siguiente razón: cuando un fichero ejecutable se usa por primera vez para crear un nuevo proceso, sólo parte del programa y de los datos de dicho fichero se cargan en la memoria real. Más tarde, según se van dando fallos de página, nuevas partes del programa de los datos se irán cargando. Es sólo en el momento de la carga inicial cuando se crean las páginas de memoria virtual y se las asocia con posiciones en uno de los dispositivos que se utiliza para ello. En ese momento, el sistema operativo indica si es necesario limpiar (poner a 0) las posiciones en el marco de página antes de la primera carga de un bloque de programa o datos.

Reemplazo de páginas. La tabla de marcos de página se utiliza para el reemplazo de las páginas. Se usan diferentes punteros para crear listas dentro esta tabla. Todos los marcos disponibles se enlazan en una lista de marcos libres disponibles para traer páginas. Cuando el número de marcos disponibles cae por debajo un determinado nivel, el núcleo quitará varios marcos para compensar.

El algoritmo de reemplazo páginas usado en SVR4 es un refinamiento del algoritmo del reloj (Figura 8.16) conocido como el algoritmo del reloj con dos manecillas (Figura 8.23), el algoritmo utiliza el bit de referencia en la entrada de la tabla de páginas por cada página en memoria que sea susceptible de selección (no bloqueada) para un reemplazo. Este bit se pone a 0 cuando la página se trae por primera vez y se pone a 1 cuando se ha hecho referencia a la página para lectura o escritura. Una de las manecillas del algoritmo del reloj, la manecilla delantera, recorre las páginas de la lista de páginas seleccionables y pone el bit de referencia a 0 para cada una de ellas. Un instante después, la manecilla

trasera recorre la misma lista y verifica el bit de referencia. Si el bit está puesto a 1, entonces se ha hecho referencia a la página desde el momento en que pasó la manecilla delantera por ahí, estos marcos se saltan. Si el bit está a 0, entonces no se ha hecho referencia a dicha página en el intervalo de tiempo de la visita de las dos manecillas; estas páginas se colocan en la lista de páginas expulsables.

Dos parámetros determinan la operación del algoritmo:

- **La tasa de recorrido.** La tasa a la cual las dos manecillas recorren la lista de páginas, en páginas por segundo.
- **La separación entre manecillas.** El espacio entre la manecillas delantera y la trasera.

Estos dos parámetros vienen fijados a unos valores por omisión en el momento de arranque, dependiendo de la cantidad de memoria física. El parámetro tasa de recorrido puede modificarse para responder a cambios en las diferentes condiciones del sistema. El parámetro puede variar linealmente entre los valores de recorrido lento (*slowscan*) y recorrido rápido (*fastscan*) (fijados en el momento de la configuración) dependiendo de que la cantidad de memoria libre variando entre los valores *lotsfree* y *minfree*. En otras palabras, a medida que la memoria libre se reduce, las manecillas del reloj se mueven más rápidamente para liberar más páginas. El parámetro de separación entre manecillas indicado por el espacio entre la manecilla delantera y la trasera, por tanto, junto con la tasa de recorrido, indica la ventana de oportunidad para usar una página antes de que sea descartable por falta de uso.

Asignador de memoria de núcleo. El núcleo, a lo largo de su ejecución, genera y destruye pequeñas tablas y *buffers* con mucha frecuencia, cada uno de los cuales requiere la reserva de memoria dinámica. [VAHA96] muestra varios ejemplos:

- El encaminamiento para traducción de una ruta puede reservar un *buffer* para copiar la ruta desde el espacio usuario.

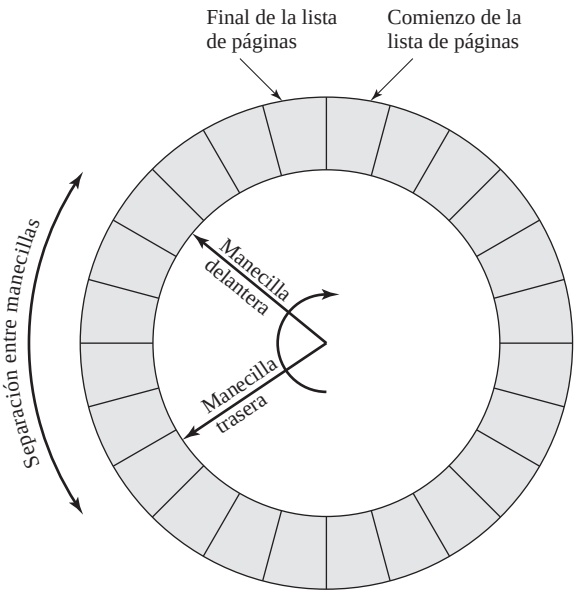


Figura 8.23. Algoritmo de reemplazo de paginas del reloj de dos manecillas.

Tabla 8.5. Parámetros de gestión de memoria de UNIX SVR4.

Entrada de la tabla de páginas
Número de marco de página Indica el marco de la memoria real.
Edad Indica cuánto tiempo ha pasado la página en la memoria sin haberse referenciado. La longitud y contenidos de este campo dependen del procesador.
Copy on write[8] Puesto a 1 cuando más de un proceso comparte esta página. Si uno de los procesos escribe en esta página, se debe hacer primero una copia aparte de la misma para todos los procesos que aún comparten la página original. Esta funcionalidad permite demorar la operación de copia hasta que sea completamente necesaria y evitar los casos en los cuales no llega a serlo.
Modificada Indica si la página se ha modificado.
Referenciada Indica que se ha hecho referencia a esta página. Este bit se pone a 0 cuando la página se carga por primera vez y se puede reiniciar periódicamente por parte del algoritmo de reemplazo de páginas.
Válida Indica si la página se encuentra en la memoria principal.
Protegida Indica si se permite la operación escritura o no.
Descriptor de bloques de disco
Número de dispositivo de *swap* Número de dispositivo lógico de almacenamiento secundario que almacena la página correspondiente. Se permite que existan más de un dispositivo para ser utilizados como *swap*.
Número de bloque del dispositivo Posición del bloque de la página en el dispositivo de *swap*.
Tipo de almacenamiento El dispositivo de almacenamiento puede ser una unidad de *swap* o un fichero ejecutable. En el último caso, existe una indicación que denota si la memoria virtual asignada debe borrarse o no.
Entrada en la tabla de marcos de página
Estado de la página Indica si este marco se encuentra disponible o si está asociado a una página. En este último caso, especifica si la página se encuentra en dispositivo de *swap*, en fichero ejecutable, o en una operación de E/S.
Contador de referencias Número de procesos que hacen referencia a esta página.
Dispositivo lógico Dispositivo lógico que contiene una copia de la página.
Número de bloque Posición del bloque de la copia de la página sobre el dispositivo lógico.
Punteros datos MP Puntero a otros datos que apuntan a otras entradas en la tabla de marcos de página. Con este puntero se puede construir la lista de páginas libres o una lista *hash* de páginas.
Entrada en la tabla de uso de *swap*
Contador de referencias Número de entradas en la tabla de página que apunta a esta página en el dispositivo de *swap*.
Número de la unidad de almacenamiento/página Identificador de la página en la unidad almacenamiento

[8] N. del T. El término *copy on write* se ha dejado en inglés puesto que es la forma más habitual de denotarlo, incluso en la literatura en castellano.

- La rutina `allocb()` reserva un *buffer* para flujos de tamaño arbitrario.
- Muchas implementaciones de UNIX reservan estructuras *zombie* para recoger el estado de salida e información de la utilización de recursos sobre procesos finalizados.
- En SVR4 y en Solaris, el núcleo reserva muchos objetos (como estructuras del *proc*, v-nodos, y bloques de descripción de fichero) de forma dinámica y bajo demanda.

La mayoría de sus bloques son significativamente más pequeños que el tamaño típico de una página de la máquina, y por tanto el mecanismo de paginación sería muy deficiente a la hora de reserva de la memoria dinámica del núcleo. Para el caso de SVR4, se utiliza una modificación del sistema *buddy*, descrito en la Sección 7.2.

En los sistemas *buddy*, el coste de reservar y liberar un bloque de la memoria es bajo comparado con las políticas de mejor ajuste y primer ajuste [KNUT97]. De esta forma, en el caso de la gestión de la memoria del núcleo, las operaciones de reserva y liberación se deben realizar lo más rápido posible. La desventaja de los sistemas *buddy* es el tiempo necesario para fragmentar y reagrupar bloques.

Barkley y Lee de AT&T propusieron una variación conocida como sistema *buddy* perezoso [BARK89], y ésta es la técnica adoptada por SVR4. Los autores observaron que UNIX a menudo muestra un comportamiento estable en las solicitudes de memoria de núcleo; esto es, la cantidad de peticiones para bloques de un determinado tamaño varía ligeramente a lo largo del tiempo. Por tanto, si un bloque del tamaño 2^i se libera e inmediatamente se reagrupa con su vecino en un bloque del tamaño 2^{i+1}, la próxima vez que el núcleo solicite un bloque de tamaño 2^i, implicará la necesidad de dividir el bloque de tamaño mayor de nuevo. Para evitar esta agrupación innecesaria y su posterior división, el sistema *buddy* perezoso pospone la reagrupación hasta el momento en el que parezca que resulta necesaria, y en ese momento intenta reagrupar el mayor número de bloques posibles.

El sistema *buddy* perezoso usa los siguientes parámetros:

N_i = número actual de bloques de tamaño 2^i.

A_i = número actual de bloques de tamaño 2^i que se encuentran reservados (ocupados).

G_i = número actual de bloques de tamaño 2^i que están libres globalmente; estos son los bloques que se pueden seleccionar para ser reagrupados; si el vecino de uno de estos bloques se encuentra libre, entonces ambos bloques se pueden agrupar en un bloque libre global de tamaño 2^{i+1}. Los bloques libres (huecos) en un sistema *buddy* estándar pueden considerarse libres globales.

L_i = número actual de bloques de tamaño 2^i que se encuentran libres localmente; estos son los bloques que no se encuentran como seleccionables para su grabación. Incluso si el vecino de dicho bloque se encuentra libre, los dos bloques no se reagrupan. En lugar de eso, los bloques libres locales se mantienen a la espera de una petición futura para un bloque de dicho tamaño.

La siguiente relación se mantiene:

$$N_i = A_i + G_i + L_i$$

En general, el sistema *buddy* perezoso intenta mantener un caudal de bloques libres locales y únicamente solicita la reagrupación si el número de bloques libres locales supera un determinado límite. Si hay muchos bloques libres locales, entonces existe la posibilidad de que falten bloques libres en el siguiente nivel. La mayoría del tiempo, cuando se libera un bloque, la reagrupación no se realiza, de

forma que se minimizan los costes de gestión y de operaciones. Cuando se va a reservar un bloque, no se hace distinción alguna entre bloques libres locales y globales; de la misma forma, esto minimiza la gestión.

El criterio que se utiliza para la reagrupación es que el número de bloques libres locales del tamaño determinado no puede exceder el número de bloques reservados para ese tamaño (por ejemplo, se debe tener $L_i \geq A_i$). Ésta es una guía razonable para evitar el crecimiento de bloques libres locales, en los experimentos llevados a cabo por [BARK89] se confirman los resultados de que este esquema proporciona unas mejoras considerables.

Para implementar este esquema, los autores de definen una variable de demora de la siguiente forma:

$$D_i = A_i - L_i = N_i - 2L_i - G_i$$

La Figura 8.24 muestra este algoritmo.

Valor inicial de D_i es igual a 0
después de la operación, el valor de D_i se actualiza de la siguiente manera:

(I) si la siguiente operación es una solicitud de reservar un bloque:
si hay bloques libres, se selecciona uno a reserva
si el bloque seleccionado se encuentra libre de forma local
entonces $D_i := D_i + 2$
sino $D_i := D_i + 1$
en otro caso
primero se toman dos bloques dividiendo un bloque mayor en dos (operación recursiva)
se reserva uno y se marca el otro como libre de forma local
D_i permanece sin cambios (pero se puede cambiar D para otros tamaños de bloque debido a la llamada recursiva)

(II) si la siguiente operación es una liberación de un bloque:
Caso $D_i \geq 2$
se marca como libre de forma local y se libera localmente
$D_i := D_i - 2$
Caso $D_i = 1$
se marca como libre de forma global y se libera globalmente; reagrupación si es posible
$D_i := 0$
Caso $D_i = 0$
se marca como libre de forma global y se libera globalmente; reagrupación si es posible
se selecciona un bloque libre local de tamaño $2i$ y se libera de forma global; reagrupación si es posible
$D_i := 0$

Figura 8.24. Algoritmo del sistema *buddy* perezoso.

8.4. GESTIÓN DE MEMORIA EN LINUX

Linux comparte muchas de las características de los esquemas de gestión de la memoria de otras implementaciones de UNIX pero incorpora sus propias características. Por encima de todo, el esquema de gestión de la memoria de Linux es bastante complejo [DUBE98].

En esta sección, vamos a dar una ligera visión general de dos de los principales aspectos de la gestión de la memoria en Linux: la memoria virtual de los procesos y la asignación de la memoria del núcleo.

MEMORIA VIRTUAL EN LINUX

Direccionamiento de la memoria virtual. Linux usa una estructura de tablas de páginas de tres niveles, consistente en los siguientes tipos de tablas (cada tabla en particular tiene el tamaño de una página):

- **Directorio de páginas.** Un proceso activo tiene un directorio de páginas único que tiene el tamaño de una página. Cada entrada en el directorio de páginas apunta a una página en el directorio intermedio de páginas. El directorio de páginas debe residir en la memoria principal para todo proceso activo.
- **Directorio intermedio de páginas.** El directorio intermedio de páginas se expande a múltiples páginas. Cada entrada en el directorio intermedio páginas apunta a una página que contiene una tabla de páginas.
- **Tabla de páginas.** La tabla de páginas también puede expandirse a múltiples páginas. Cada entrada en la tabla de páginas hace referencia a una página virtual del proceso.

Para utilizar esta estructura de tabla de páginas de tres niveles, una dirección virtual en Linux se puede ver cómo consistente en cuatro campos (Figura 8.25) el campo más a la izquierda (el más significativo) se utiliza como índice en el directorio de páginas. El siguiente campo sirve como índice en el directorio intermedio de páginas. El tercer campo se utiliza para indexar en la tabla de páginas. Y con el cuarto campo se proporciona el desplazamiento dentro de la página de la memoria seleccionada.

La estructura de la tabla de páginas en Linux es independiente de plataforma y se diseñó para acomodarse al procesador Alpha de 64 bits, que proporciona soporte hardware para tres niveles de páginas. Con direcciones de 64 bits, la utilización de únicamente dos niveles de páginas en una arquitectura Alpha resultaría unas tablas de páginas y unos directorios de gran tamaño. La arquitectura Pentium/x86 de 32 bits tiene un sistema hardware de paginación de dos niveles. El software de Linux se acomoda al esquema de dos niveles definiendo el tamaño del directorio intermedio de páginas como 1. Hay que resaltar que todas las referencias a ese nivel extra de indirección se eliminan en la optimización realizada en el momento de la compilación, no durante la ejecución. Por tanto, no hay ninguna sobrecarga de rendimiento por la utilización de un diseño genérico de tres niveles en plataformas que sólo soportan dos niveles.

Reserva de páginas. Para mejorar la eficiencia de la lectura y escritura de páginas en la memoria principal, Linux define un mecanismo para manejar bloques de páginas contiguas que se proyectarán sobre bloques de marcos de página también contiguos. Con este fin, también se utiliza el sistema *buddy*. El núcleo mantiene una lista de marcos de página contiguos por grupos de un tamaño fijo; un grupo puede consistir en 1, 2, 4, 8, 16, o 32 marcos de páginas. A lo largo del uso, las páginas se asignan y liberan de la memoria principal, los grupos disponibles se dividen y se juntan utilizando el algoritmo del sistema *buddy*.

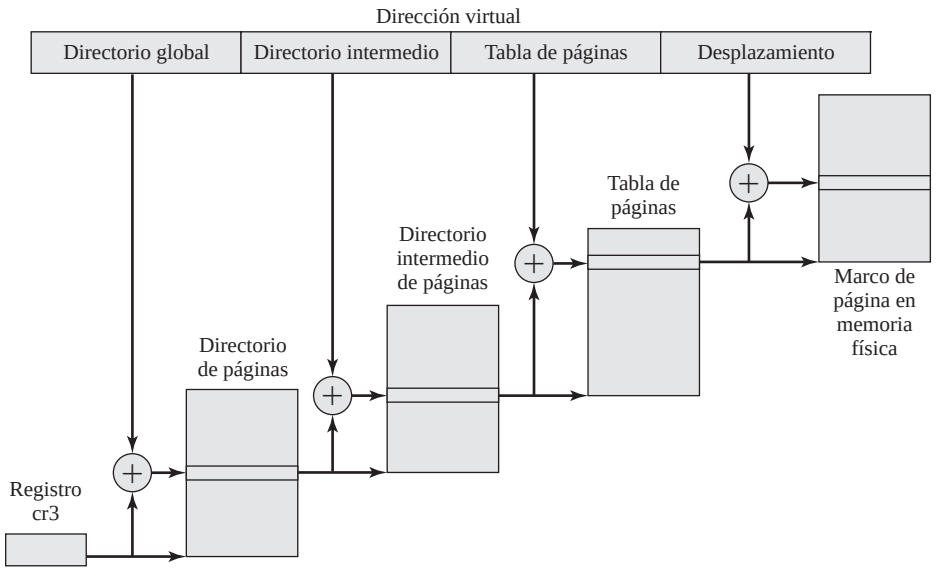


Figura 8.25. Traducción de direcciones en el esquema de memoria virtual de Linux.

Algoritmo de reemplazo de páginas. El algoritmo de reemplazo de páginas de Linux se basaba en el algoritmo del reloj descrito en la Sección 8.2 (*véase* Figura 8.16). En el algoritmo del reloj sencillo, se asocia un bit de usado y otro bit de modificado con cada una de las páginas de memoria principal. En el esquema de Linux, el de usado se reemplaza por una variable de 8 bits. Cada vez que se accede a una página, la variable se incrementa. En segundo plano, Linux recorre periódicamente la lista completa de páginas y decrementa la variable de edad de cada página a medida que va rotando por todas ellas en la memoria principal. Una página con una edad de 0 es una página «vieja» a la que no se ha hecho referencia desde hace algún tiempo y es el mejor candidato para un reemplazo. Cuando el valor de edad es más alto, la frecuencia con la que se ha accedido a la página recientemente es mayor y, por tanto, tiene menor posibilidad de elegirse para un reemplazo de esta forma, el algoritmo de Linux es una variante de la política LRU.

RESERVA DE MEMORIA DEL NÚCLEO

La gestión de la memoria del núcleo se realiza en base a los marcos de página de la memoria principal. Su función básica es asignar y liberar marcos para los diferentes usos. Los posibles propietarios de un marco incluyen procesos en espacio de usuario (por ejemplo, el marco es parte de la memoria virtual de un proceso que se encuentra actualmente residiendo en la memoria real), datos del núcleo reservados dinámicamente, código estático del núcleo, y la cache de páginas[9].

Los fundamentos de la reserva de memoria de núcleo para Linux son los mecanismos de reserva de páginas ya usados para la gestión de la memoria virtual de usuario. Como en el caso del esquema de la memoria virtual, se utiliza el algoritmo *buddy* de forma que la memoria del núcleo

[9] La cache de páginas tiene propiedades similares a un *buffer* de disco, descrito en este capítulo, así como de cache disco, que se verá en el Capítulo 11. Se pospone la discusión sobre la cache de páginas de Linux a dicho Capítulo 11.

se pueda reservar y liberar en unidades de una o más páginas. Debido a que el tamaño mínimo de memoria que se pueda reservar de esta forma es una página, la reserva de páginas únicamente sería insuficiente debido a que el núcleo requiere pequeños fragmentos que se utilizarán durante un corto periodo de tiempo y que son de diferentes tamaños. Para ajustarse a estos pequeños tamaños, Linux utiliza un esquema conocido como *asignación por láminas* (*slab allocation*) [BONW94] dentro una página ya reservada. En una máquina Pentium/x86, el tamaño página es de 4 Kbytes y los fragmentos dentro una página se pueden asignar en tamaños de 32, 64, 128, 252, 508, 2040, y 4080 bytes.

La asignación por láminas es relativamente compleja y no se va a examinar en detalle aquí; una buena descripción se pueda encontrar en [VAHA96]. En esencia, Linux mantiene un conjunto de listas enlazadas, una para cada tamaño del fragmento. Todos los fragmentos se pueden dividir y agregar de una manera similar a la indicada por el algoritmo *buddy*, y también se pueden mover entre las diferentes listas de la forma correspondiente.

8.5. GESTIÓN DE MEMORIA EN WINDOWS

El gestor de memoria virtual en Windows controla la forma en la que se reserva la memoria y cómo se realiza la paginación. El gestor de memoria se ha diseñado para funcionar sobre una variada gama de plataformas y para utilizar tamaños de páginas que van desde los 4 Kbytes hasta los 64 Kbytes. Las plataformas Intel, PowerPC, y MIPS tienen 4096 bytes por página y las plataformas DEC Alpha tienen 8192 bytes por página.

MAPA DE DIRECCIONES VIRTUALES EN WINDOWS

Cada proceso de usuario en Windows puede ver un espacio de direcciones independiente de 32 bits, permitiendo 4 Gbytes de memoria por proceso. Por omisión, una parte de esta memoria se encuentra reservada para el sistema operativo, de forma que en realidad cada usuario dispone de 2 Gbytes de espacio virtual de direcciones disponibles y todos los procesos comparten los mismos 2 Gbytes de espacio de sistema. Existe una opción que permite que el espacio de direcciones crezca hasta los 3 Gbytes, dejando un espacio de sistema de únicamente 1 Gbytes. En la documentación de Windows se indica que esta característica se incluyó para dar soporte a aplicaciones que requieren un uso intensivo de grandes cantidades de memoria en servidores con memorias de varios gigabytes, en los cuales un espacio de direcciones mayor puede mejorar drásticamente el rendimiento de aplicaciones de soporte como la decisión o *data mining*.

La Figura 8.26 muestra el espacio de direcciones virtuales que, por efecto, ve un usuario. Consiste en cuatro regiones:

- 0x00000000 a 0x0000FFFF. Reservada para ayudar a los programadores a capturar asignaciones de punteros nulos.
- 0x00010000 a 0x7FFEFFFF. Espacio de direcciones disponible para el usuario. Este espacio se encuentra dividido en páginas que se pueden cargar de la memoria principal.
- 0x7FFF0000 a 0x7FFFFFFF. Una página de guarda, no accesible por el usuario. Esta página hace que al sistema operativo le resulte más fácil verificar referencias a punteros fuera de rango.
- 0x80000000 a 0xFFFFFFFF. Espacio de direcciones de sistema. Este área de 2 Gbytes se utiliza por parte del Ejecutivo de Windows, el micronúcleo y los manejadores de dispositivos.

PAGINACIÓN EN WINDOWS

Cuando se crea un proceso, puede, en principio, utilizar todo el espacio de usuario de 2 Gbytes (menos 128 Kbytes). Este espacio se encuentra dividido en páginas de tamaño fijo, cualquiera de las cuales se puede cargar en la memoria principal. En la práctica, una página se puede encontrar, a efectos de gestión, en los presentes estados:

- **Disponible.** Páginas que no están actualmente usadas por este proceso.
- **Reservada.** Conjunto de páginas contiguas que el gestor de memoria virtual separa para un proceso pero que no se cuentan para la cuota de memoria usada por dicho proceso. Cuando un proceso necesite escribir en la memoria, parte de esta memoria reservada se asigna al proceso.
- **Asignada.** Las páginas para las cuales el gestor de la memoria virtual ha reservado espacio en el fichero de paginación (por ejemplo, el fichero de disco donde se escribirían las páginas cuando se eliminen de la memoria principal).

La distinción entre la memoria reservada y asignada es muy útil debido a que (1) minimiza la cantidad de espacio de disco que debe guardarse para un proceso en particular, manteniendo espacio libre en disco para otros procesos; y (2) permite que un hilo o un proceso declare una petición de una cantidad de memoria que puede proporcionarse rápidamente si se necesita.

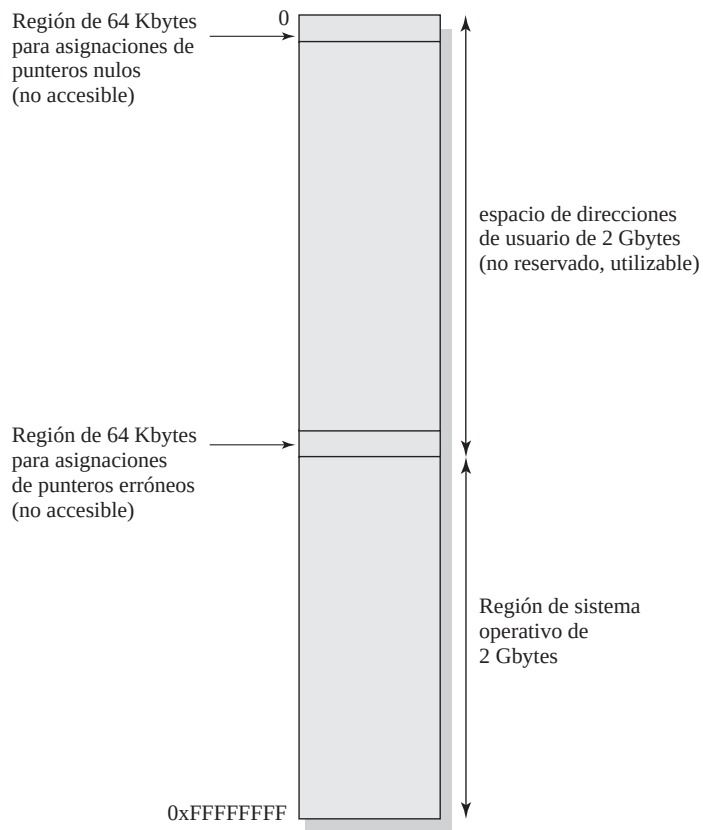


Figura 8.26. Espacio de direcciones virtuales habitual de Windows.

El esquema de gestión del conjunto residente que utiliza Windows es de asignación variable, con ámbito local (*véase* Tabla 8.4). Cuando se activa un proceso por primera vez, se le asigna un cierto número de marcos de página de la memoria principal como conjunto de trabajo. Cuando un proceso hace referencia a una página que no está en la memoria, una de las páginas residentes de dicho proceso se expulsa, trayéndose la nueva página. Los conjuntos de trabajo de los procesos activos se ajustan usando las siguientes condiciones generales:

Cuando hay memoria principal disponible, el gestor de la memoria virtual permite que los conjuntos residentes de los procesos activos crezcan. Para hacer esto, cuando ocurre un fallo página, se trae la nueva página a la memoria sin expulsar una página antigua, haciendo que se incremente el conjunto residente de proceso en una página.

Cuando la memoria empieza escasear, el gestor de la memoria virtual recupera la memoria de sistema moviendo las páginas que se han utilizado hace más tiempo de cada uno de los procesos hacia *swap*, reduciendo el tamaño de esos conjuntos residentes.

8.6. RESUMEN

Para poder usar de forma eficiente el procesador y las funciones de E/S, es aconsejable mantener el mayor número de procesos posibles en la memoria principal. Además, es deseable liberar al programador de las restricciones de tamaño en el desarrollo de programas.

La vía para llevar a cabo ambas recomendaciones es por medio de la memoria virtual. Con la memoria virtual, todas las referencias a direcciones son lógicas y se traducen en tiempo de ejecución a direcciones reales. Esto permite que un proceso se pueda ubicar en cualquier parte de la memoria principal e incluso que dicha ubicación cambie a lo largo del tiempo. La memoria virtual también permite que un proceso se dividida en fragmentos. Estos fragmentos no tienen porqué estar ubicados de forma contigua en la memoria principal durante su ejecución y, en realidad, tampoco se necesita que todos los fragmentos del proceso se encuentren en la memoria principal durante dicha ejecución.

Las dos estrategias básicas para proporcionar memoria virtual son paginación y segmentación. Por medio de la paginación, cada proceso se divide en páginas de tamaño fijo, relativamente pequeñas. La segmentación proporciona la posibilidad de definir fragmentos de tamaño variable. Se pueden combinar segmentación y paginación en un único esquema de la gestión de la memoria.

Un esquema de gestión de la memoria virtual requiere soporte hardware y software. El soporte hardware lo proporciona el procesador. Dicho soporte incluye traducción dinámica de las direcciones virtuales en direcciones físicas y la generación de interrupciones cuando se hace referencia a una página o segmento que no se encuentra en la memoria principal. Dichas interrupciones disparan el software de la gestión de la memoria del sistema operativo.

En este capítulo hemos tratado diversas consideraciones de diseño relativas al soporte del sistema operativo:

- **Política de recuperación.** Las páginas se pueden traer bajo demanda, o con una política de paginación adelantada, que agrupa la carga de varias páginas a la vez.
- **Política de ubicación.** En un sistema de segmentación pura, cuando un segmento se trae a memoria debe ajustarse al espacio en memoria que se encuentra disponible.
- **Política de reemplazo.** Cuando la memoria se encuentra llena, se debe decidir qué página o páginas van a reemplazarse para traer una nueva.
- **Gestión del conjunto reciente.** El sistema operativo debe decidir cuánta memoria asigna a un proceso en particular cuando dicho proceso se trae a la memoria. Esto se puede hacer me-

diante asignación estática, realizada en el momento de crear un proceso, o se puede cambiar dinámicamente.

- **Política de limpieza.** Las páginas del proceso que se han modificado se pueden escribir en el momento de reemplazo, o se puede articular una política de limpieza adelantada, la cual agrupa la actividad de escritura de páginas en varias de ellas a la vez.
- **Control de carga.** El control de carga es el responsable de determinar el número de procesos que van a residir en la memoria principal en un instante determinado.

8.7. LECTURA RECOMENDADA Y PÁGINAS WEB

Como era previsible, la memoria virtual se encuentra tratada en la mayoría de libros sobre sistemas operativos. [MILE92] proporciona un buen resumen de varias áreas de investigación. [CARR84] proporciona un excelente examen en profundidad de las cuestiones de rendimiento. El artículo clásico, [DENN70] es aún una lectura muy recomendada. [DOWD93] proporciona un análisis de rendimiento profundo de varios organismos de reemplazo de páginas. [JACO98a] es una buena revisión de los aspectos de diseño de la memoria virtual; incluye una explicación sobre las tablas de páginas invertidas. [JACO98b] examina la organización del hardware de la memoria virtual para varios microprocesadores.

Una lectura más sobria, [IBM86], que cuenta de forma detallada las herramientas y opciones disponibles para la administración de la optimización de las políticas de la memoria virtual de los sistemas MVS. Este documento ilustra perfectamente la complejidad del problema.

[VAHA96] es uno de los mejores tratamientos de los esquemas de la gestión de la memoria utilizados en varios tipos de sistemas UNIX. [GORM04] es un tratamiento profundo de la gestión de la memoria en Linux.

CARR84 Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.

DENN70 Denning, P. «Virtual Memory.» *Computing Surveys.* Septiembre, 1970.

DOWD93 Dowdy, L. y Lowery, C. *P.S. to Operating Systems.* Upper Saddle River, NJ: Prentice Hall, 1993.

GORM04 Gorman, M. *Understanding the Linux Virtual Memory Management*. Upper Saddle River, NJ: Prentice Hall, 2004.

IBM86 IBM Nacional Technical Support, Large Systems. *Multiple Virtual Storage (MVS) Virtual Storage Tuning Cookbook*. Dallas Systems CenterTechnical Bulletin G320-0597, Junio 1986.

JACO98a Jaboc, B. y Mudge, T. «Virtual Memory: Issues of Implementation.» *Computer*, Junio 1998.

JACO98b Jaboc, B. y Mudge, T. «Virtual Memory in Contemprany Microprocessors» *IEEE Micro*, Agosto 1998.

MILE92 Milenkovic, M. *Operating Systems: Concepts and Design*. New Cork: McGraw-Hill, 1992.

VAHA96 Vahalia, U. *UNIX Internals: The New Frontiers*. Upper Saddle River, NJ: Prentice Hall, 1996.

Páginas Web recomendadas:

- **The Memory Management Reference.** Una buena fuente de documentos y enlaces sobre todo los aspectos de la gestión de la memoria.

8.8. TÉRMINOS CLAVE, CUESTIONES DE REPASO, Y PROBLEMAS

TÉRMINOS CLAVE

asignación por láminas	memoria real	proximidad de referencia
conjunto de trabajo	memoria virtual	segmentación
conjunto reciente	página	segmento
fallo de página	paginación	tabla de páginas
fragmentación externa	paginación adelantada	tabla de segmentos
fragmentación interna	paginación por demanda	tabla *hash*
gestión del conjunto residente	política de recuperación	TLB
hashing	política de reemplazo	traducción asociativa
marco	política de ubicación	trasiego

CUESTIONES DE REPASO

8.1. ¿Cuál es la diferencia entre la paginación sencilla y la paginación con memoria virtual?

8.2. Explique el trasiego o *thrashing*.

8.3. ¿Por qué el principio de proximidad de referencia es crucial para el uso de la memoria virtual?

8.4. ¿Qué elementos se encuentran típicamente en la entrada de tabla de páginas? Defina brevemente cada elemento.

8.5. ¿Cuál es el propósito de la TLB?

8.6. Defina brevemente las alternativas para la política de recuperación.

8.7. ¿Cuál es la diferencia entre la gestión del conjunto residente y la política de reemplazo de páginas?

8.8. ¿Cuál es la relación entre los algoritmos de reemplazo de páginas FIFO y del reloj?

8.9. ¿Cuál es el cometido del *buffering* de páginas?

8.10. ¿Por qué no es posible combinar a la política de reemplazo global y la política de asignación fija?

8.11. ¿Cuál es la diferencia entre el conjunto reciente y el conjunto de trabajo?

8.12. ¿Cuál es la diferencia entre la limpieza por demanda y la limpieza adelantada?

PROBLEMAS

8.1. Suponga que la tabla de páginas del proceso actualmente en ejecución es la siguiente. Todos los números están en formato decimal, todos se encuentran numerados comenzando desde cero, y todas las direcciones son direcciones de memoria a nivel de byte. El tamaño de página es de 1024 bytes.

Número de página virtual	Bit de valida	Bit de referenciada	Bit de modificada	Número de marco de página
0	1	1	0	4
1	1	1	1	7
2	0	0	0	—
3	1	0	0	2
4	0	0	0	—
5	1	0	1	0

a) Describa exactamente cómo, en términos reales, se generaría a partir de la dirección virtual la dirección de memoria física, por medio de la traducción realizada por la CPU.

b) ¿Qué dirección física, si hay alguna, se corresponderá con las siguientes direcciones virtuales? (No hace falta que trate ningún fallo de página, se produce).

i) 1052

ii) 2221

iii) 5499

8.2. Considere un sistema de memoria virtual paginada con direcciones virtuales de 32 bits y páginas de 1 Kbyte. Cada entrada en la tabla de páginas requiere 32 bits. Y se desea limitar el tamaño de las tablas de páginas a una única página.

a) ¿Cuántos niveles de tablas de páginas se necesitan?

b) ¿Cuál es el tamaño de la tabla de páginas de cada nivel? *Sugerencia*: uno de los tamaños de tabla de página es menor.

c) La tabla de páginas menor se puede usar en la parte más alta o la parte más baja de la jerarquía de tablas de páginas. ¿Cuál de las estrategias implican menor consumo de páginas?

8.3. a) ¿Cuánto espacio de memoria se necesita para las tablas de páginas de usuario de la Figura 8.4?

b) Asumiendo que se quiere implementar una tabla de páginas invertida de tipo *hash* para el mismo esquema de direccionamiento mostrado en la Figura 8.4, por medio de una función *hash* que proyecta el número página de 20 bits en un valor *hash* es de 6 bits. La entrada en la tabla contiene el número de página, el número de marco, y un puntero a la cadena. Si la tabla de páginas dispone de hasta tres entradas adicionales por colisiones por cada entrada *hash*, ¿cuánto espacio de memoria estaría ocupando la tabla de páginas invertida?

8.4. Un proceso tiene cuatro marcos reservados para el uso (los siguientes números están en formato decimal, y todas las numeraciones comienzan desde 0). La siguiente tabla muestra el instante de tiempo en el que se cargó la última página en cada marco, el instante de tiempo del último acceso a cada página, el número de página virtual, los bits de referenciada (R) y modificada (M) para cada uno de los marcos de página (los instantes de tiempo están expresados en *ticks* de reloj desde el comienzo del proceso).

Número de página virtual	Marco de página	Instante de carga	Instante de referencia	Bit R	Bit M
2	0	60	161	0	1
1	1	130	160	1	0
0	2	26	162	1	0
3	3	20	163	1	1

Se produce un fallo de página para la página virtual 4 en el instante 164. ¿Qué marco de página reemplazará su contenido para cada una de las siguientes políticas de la gestión de la memoria? Explique por qué en cada uno de los casos.

a) FIFO

b) LRU

c) Reloj

d) Óptimo (usando la siguiente cadena de referencias)

Tomando el estado de la memoria antes mencionado, justo en el instante en el que se produce el fallo de página, y teniendo en cuenta la siguiente cadena de referencias a páginas virtuales:

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

e) ¿Cuántos fallos de página ocurrirían si se usase la política del conjunto de trabajo con LRU tomando una ventana de tamaño 4 en lugar de asignación fija? Indique claramente cuándo se produce cada fallo de página.

8.5. Un proceso hace referencia a cinco páginas, A, B, C, D, y E, en el siguiente orden:

A; B; C; D; A; B; E; A; B; C; D; E

Asuma que el algoritmo de reemplazo es FIFO, y encuentre el número de transferencias de páginas durante la anterior secuencia de referencias que comienza con la memoria principal vacía con una limitación de tres marcos de página. Repita lo mismo para el caso de cuatro marcos de página.

8.6. Un proceso contiene ocho páginas virtuales en disco y se asigna de forma fija cuatro marcos de página de memoria principal. La traza de las páginas es la siguiente:

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

a) Muestre las sucesivas páginas residentes en los cuatro marcos utilizando la política de reemplazo LRU. Calcule el índice de acierto de la memoria principal. Asúmase que los marcos están inicialmente vacíos.

b) Repita el apartado (a) para política de reemplazo FIFO.

c) Compare ambos índices de acierto y comente la efectividad de utilizar FIFO como aproximación a LRU en lo referente a esta traza en particular.

8.7. En los sistemas VAX, las tablas de página de los usuarios se colocan en direcciones virtuales en el espacio de sistema. ¿Cuál es la ventaja de tener las tablas de página de usuarios en la memoria virtual en lugar de la memoria principal? ¿Cuáles son las desventajas?

8.8. Supóngase las siguiente instrucciones de programa

```
for(i=1;i<=n;i++)
        a[i]=b[i]+c[i];
```

Si se ejecutan en una memoria con tamaño de página de 1000 palabras. Siendo *n = 1000*. Usando una máquina que tiene un conjunto completo de instrucciones registro-registro y que utiliza registros índice, escriba un programa hipotético que implemente las siguientes instrucciones. Posteriormente, muestre la secuencia de referencias a páginas durante su ejecución.

8.9. Los sistemas IBM/370 utilizan una estructura de memoria de dos niveles y denominan a cada uno de los niveles como segmentos y páginas, a pesar de que la parte segmentación carece de muchas de las características descritas anteriormente en este capítulo. Para arquitectura básica 370, el tamaño de página puede ser o bien 2 Kbytes o 4 Kbytes, y el tamaño segmento se fija entre 64 Kbytes o 1 Mbyte. Para las arquitecturas 370/XA y 370/ESA, el tamaño de página es de 4 Kbytes y el tamaño de segmentos de 1 Mbyte. ¿Qué beneficios de la segmentación no se encuentran disponibles en este esquema? ¿Cuál es el beneficio del modelo segmentación de los 370?

8.10. Asumiendo un tamaño página de 4 Kbytes y que una entrada de la tabla páginas requiere 4 bytes, ¿Cuántos niveles de tablas de páginas se necesitarán para proyectar un espacio de direcciones de 64 bits si la tabla de páginas principal entra en una única página?

8.11. Considere un sistema con gestión de memoria basada en páginas y que utiliza un único nivel de páginas. Asumiendo que la tabla de páginas necesaria se encuentra siempre en la memoria.

a) Si una referencia a la memoria consume 200 ns, ¿cuánto puede tardar una referencia a una página de memoria?

b) Ahora añadimos una MMU que añade una sobrecarga de 20 ns, tanto si hay acierto como si no. Si se asume que el 85% de todas las referencias a memoria son aciertos en la TLB de la MMU, ¿cuál es el tiempo efectivo de acceso a la memoria?

c) Explíquese cómo la tasa de aciertos en la TLB afecta al tiempo de acceso a la memoria efectivo.

8.12. Considere una cadena de referencias a páginas de un proceso con un conjunto de trabajo de *M* marcos, inicialmente vacíos. La cadena de referencias a páginas es una longitud *P* con *N* números de páginas diferentes. Para cualquier algoritmo de reemplazo de páginas,

a) ¿Cuál es el límite inferior en el número de fallos de página?

b) ¿Cuál es el límite superior en el número de fallos de página?

8.13. En explicaciones sobre los diferentes algoritmos de reemplazo de páginas, un autor realiza la analogía con un quita-nieves moviéndose a lo largo de una pista circular. La nieve cae de forma uniforme a lo largo de toda la pista y el único quita-nieves se mueve a una velocidad constante a lo largo de todo el recorrido. La nieve una vez que ha pasado el quita-nieves por la pista desaparece del sistema.

a) ¿A cuál de los algoritmo de reemplazo de páginas presentados en la Sección 8.2 se puede asociar esta analogía?

b) ¿Qué sugiere esta analogía sobre el comportamiento del algoritmo de reemplazo de páginas en cuestión?

8.14. En la arquitectura S/370, una clave de almacenamiento es un campo de control asociado a cada marco de página de la memoria real. Dos de los bits de esa clave son importantes para el reemplazo de páginas y son el bit de referencia y el bit de cambio. El bit de referencia se pone a 1 cuando se ha accedido a una dirección dentro del marco para lectura o para escritura y se pone a 0 cuando se carga una nueva página en el marco. El bit de cambio se pone a 1 cuando se realiza una operación escritura sobre cualquier posición dentro del marco. Sugiera una estrategia para determinar cuál de los marcos de página se ha utilizado hace más tiempo, usando únicamente el bit de referencia.

8.15. Considere la siguiente secuencia de referencias a páginas (cada elemento de la secuencia representa un número de página):

1 2 3 4 5 2 1 3 3 2 3 4 5 4 5 1 1 3 2 5

Se refiere el *tamaño del conjunto de trabajo medio* después de la referencia *k*-ésima como $s_k(\Delta) = \sum_{t=1}^{k} W(t, \Delta)$ y se define la *probabilidad de página ausente* como

$m_k(\Delta) = \frac{1}{k} \sum_{t=1}^{k} F(t, \Delta)$, donde F(t, Δ)=1 si ocurre un fallo de página en el tiempo virtual *t*, y 0 en otro caso.

a) Dibuje un diagrama similar al mostrado la Figura 8.19 para secuencia referencias, restringiéndose a los valores de Δ = 1, 2, 3, 4, 5, 6.

b) Dibuje s_{20} (Δ) en function de Δ.

c) Dibuje m_{20} (Δ) en function de Δ.

8.16. Una clave para el rendimiento de la política de gestión del conjunto reciente VSWS es el valor de *Q*. La experiencia ha demostrado que, para un valor fijo de *Q* en un determinado proceso, hay diferencias considerables en la secuencia de fallos de página a lo largo de las distintas etapas de la ejecución. Además, si se toma un único valor de *Q* para diferentes procesos, las frecuencias de fallos de página son considerablemente diferentes. Estas diferencias indican claramente que un mecanismo que ajuste dinámicamente el valor de *Q* durante el tiempo de vida de un proceso incrementaría considerablemente el rendimiento del algoritmo. Sugiera un mecanismo sencillo para este propósito.

8.17. Considere que una tarea se encuentra dividida en cuatro segmentos de igual tamaño y que el sistema construye una tabla de descriptores de páginas de ocho entradas por cada segmento. De esta forma, el sistema implementa una combinación de segmentación y paginación. Se asume que el tamaño de la página es de 2 Kbytes.

a) ¿Cuál es el tamaño máximo de los segmentos?

b) ¿Cuál es el tamaño máximo del espacio de direcciones lógicas de una tarea?

c) Supongamos que un elemento de la ubicación física 00021ABC va a ser accedido por la tarea, ¿cuál sería el formato de la dirección lógica que la tarea generaría para él? ¿Cuál es el espacio de direcciones físicas máximo para el sistema?

8.18. Considere un espacio de direcciones lógicas paginadas (compuesto por 32 páginas de 2 Kbytes cada una) que se proyecta en un espacio de memoria física de 1 Mbyte.

a) ¿Cuál es el formato de las direcciones lógicas del procesador?

b) ¿Cuál es la longitud y anchura de la tabla de páginas (no tenga en cuenta los bits de «derechos de acceso»)?

c) ¿Cuál sería el efecto en la tabla de páginas si el espacio de la memoria física sólo fuese la mitad?

8.19. El núcleo de UNIX proporciona crecimiento dinámico de la pila de proceso en la memoria virtual, pero nunca intentará reducirla. Considere el caso de un programa que llama a una subrutina escrita en C que pide que se reserve un vector local en la pila que consume 10 Kbytes. El núcleo expandirá el segmento de pila para poder ubicarlo. Cuando se regresa de la subrutina, el puntero de pila se ajusta y dicho espacio podría liberarse por parte del núcleo, pero no lo hace. Explique por qué sería posible reducir la pila en ese instante y por qué el núcleo de UNIX no lo hace.

APÉNDICE 8A TABLAS HASH

Consideremos el siguiente problema: un conjunto de *N* objetos que se van a almacenar en una tabla. Cada objeto consiste en una etiqueta y la información adicional, a la que nos referiremos como valor del objeto. Adicionalmente, sería deseable disponer de una serie de operaciones ordinarias sobre esta tabla, inserción, borrado y búsqueda de un valor por una determinada clave.

Si las etiquetas de los objetos son numéricas, en el rango de *0* a *M – 1*, la solución sencilla sería utilizar la tabla de longitud *M*. Un objeto con etiqueta *i* se insertaría en la tabla en la posición *i*, siempre y cuando los objetos sean de tamaño fijo, la búsqueda en la tabla es trivial e implica la indexación de la misma en base a la etiqueta numérica del objeto. Además, no sería necesario almacenar dicha etiqueta en la tabla, debido a que la propia posición del objeto implica su valor. Estas tablas se denominan **tablas de acceso directo**.

Si las etiquetas no son numéricas, aún así existe la posibilidad de utilizar la estrategia de acceso directo. Si nos referimos a los objetos como *A[1]* ... *A[N]*, cada objeto *A[i]* consiste en una etiqueta, k_i, y una clave, v_i. Si se define una función de proyección *I(k)*, de forma que *I(k)* tome valores entre *1* y *M* para todas las claves y que $I(k_i) \neq (k_j)$ para cualquier par de valores *i* y *j* diferentes. En este caso, también se pueden utilizar tablas de acceso directo, con una tabla de longitud *M*.

La principal dificultad de estos esquemas ocurre cuando *M* es mucho más grande que *N*. En este caso, la proporción de entradas sin usar en la tabla es muy grande, y esto implica un uso poco eficiente de la memoria. La alternativa sería la utilización de una tabla de longitud *M* para almacenar esos objetos (etiqueta y valor) en cada una de las entradas. En este esquema, la cantidad de memoria se minimiza pero hace muchísimo más pesado el proceso de consulta de la tabla. Las diferentes posibilidades son:

- **Búsqueda secuencial.** Esta es una estrategia de fuerza bruta que consume mucho tiempo para tablas de gran tamaño.
- **Búsqueda asociativa.** Con el soporte apropiado del hardware, todos los elementos de la tabla se pueden buscar de forma simultánea. Esa estrategia no es de propósito general y no se puede aplicar a todos los tipos de tablas.
- **Búsqueda binaria.** Si las etiquetas o las proyecciones numérica de las etiquetas se ordenan de forma ascendente en la tabla, la búsqueda binaria es mucho más rápida que la secuencia (Tabla 8.6) y no requiere hardware especial.

La búsqueda binaria parece prometedora para la consulta de estas tablas. La principal desventaja de este método es que la inserción del objeto no es un proceso simple y habitualmente requerirá la reordenación de todas las entradas. Por tanto, la búsqueda binaria se usa habitualmente en tablas que son razonablemente estáticas y que sufren pocos cambios.

TABLA 8.6. Tiempo de búsqueda medio para uno de los *N* objetos de una tabla de tamaño *M*.

Técnica	Longitud de la búsqueda
Directa	1
Secuencial	$\frac{M+1}{2}$
Binaria	$\log_2(M)$
Hashing lineal	$\frac{2-N/M}{2-2N/M}$
Hash (desbordamiento encadenado)	$1+\frac{N-1}{2M}$

Queremos evitar las penalizaciones de memoria de una estrategia de acceso directo sencillo y las penalizaciones de proceso de las alternativas vistas anteriormente. El método utilizado más habitualmente para conseguir esto se denomina ***hashing***. El *hashing*, que fue desarrollado en los años 50, es muy sencillo de implementar y tiene dos ventajas. En primer lugar, puede consultar la mayoría de objetos en un acceso directo, como el caso que hemos visto antes, en segundo lugar, las inserciones y los borrados se pueden manejar sin ninguna complejidad adicional.

La función de *hashing* se puede definir de la siguiente manera. Si asumimos que tenemos *N* objetos que van a ser almacenados en una **tabla *hash*** de longitud *M*, con $M \geq N$, pero no mucho mayor. Para insertar un objeto en la tabla,

I1. Se convierte la etiqueta del objeto en un valor pseudo-aleatorio *n* que se encuentra entre 0 y *M – 1*. Por ejemplo, una función de proyección habitual es dividir la etiqueta entre *M* y quedarse con el resto del valor como *n*.

I2. Se usa *n* como índice en la tabla *hash*.

   a) Su entrada correspondiente a la tabla esta vacía, almacenar el objeto (etiqueta y valor) en dicha entrada.

   b) Si la entrada está ocupada, el almacenamiento del objeto ha causado una colisión, y se almacenará el objeto en un área de desbordamiento, como veremos más adelante.

Para la operación de consulta del objeto del cual conocemos la etiqueta,

C1. Se convierte la etiqueta del objeto en un valor pseudo-aleatorio que se encuentra entre 0 y *M – 1*, usando la misma función de proyección que en el caso de la inserción.

C2. Se usa *n* como índice en la tabla *hash*.

   a) Si la entrada correspondiente en la tabla se encuentra vacía, entonces el objeto no se encontraba almacenado previamente en la tabla.

   b) Si la entrada se encuentra ocupada y las etiquetas coinciden, entonces se puede recuperar el valor.

   c) Si la entrada se encuentra ocupada y las etiquetas no coinciden, entonces se continuará la consulta en el área de desbordamiento.

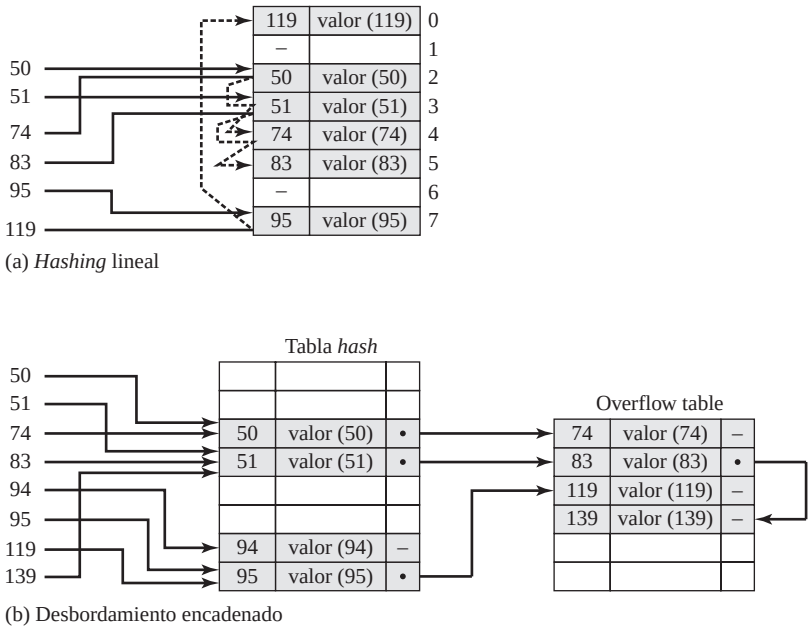


Figura 8.27. *Hashing.*

Los esquemas de *hashing* difieren en la forma la en que se manejan las colisiones y los desbordamientos. Una técnica habitual se denomina ***hashing* lineal** y se usa habitualmente en compiladores. En esta técnica, la regla I2.b se convierte en

I2.b Si la entrada está ocupada, $n = n+1$ (mod M) y se vuelve al paso I2.a.

La regla C2.c se modifica consecuentemente.

La Figura 8.27a es un ejemplo. En este caso, las etiquetas de los objetos que se almacenan son numéricas, y la tabla *hash* tiene ocho posiciones ($M=8$). La función de proyección consiste en quedarse con el resto de la división entre 8. La figura que asume los objetos fueron insertados en orden ascendente, sin que esto sea necesario. De esta forma, los objetos 50 y 51 se proyectan en las posiciones 2 y 3, respectivamente, como están vacías se insertan en ellas. El objeto 74 también se proyecta en la posición 2, pero ésta no se encuentra vacía, se intentará en la posición 3. Como ésta está también ocupada, se intentará en la posición 4, la cual está libre.

No es fácil determinar la longitud media de la operación de consulta para un determinado objeto en una tabla así debido a este efecto de agrupamiento. Una fórmula aproximada fue obtenida por Schay y Spruth [SCHA62]:

$$\text{Longitud media de búsqueda } \frac{2-r}{2-2r}$$

donde $r = N/M$. Nótese que el resultado es independiente del tamaño de la tabla y depende sólo de cómo se encuentre de llena. El resultado sorprendente es que para una tabla que se encuentra al 80% de ocupación, la longitud media de una búsqueda se encuentra en torno a 3.

Incluso así, una búsqueda de longitud 3 se puede considerar larga, y las tablas de *hashing* lineal tienen la desventaja adicional que no resulta fácil borrar elementos de ellas. Una estrategia alter-

nativa, que proporciona longitud de búsqueda más corta (Tabla 8.6) y permite borrados de la misma forma que las inserciones, es el **desbordamiento encadenado**. Esta técnica se muestra en la Figura 8.27b. En este caso, hay una tabla separada a la cual se redirigen las operaciones de colisión. Esta tabla incluye punteros entre las entradas asociándolas a una cadena con cualquier posición de la tabla *hash*. En este caso la longitud media de búsqueda, asumiendo datos distribuidos de forma aleatoria, es:

$$\text{Longitud media de búsqueda} = 1 + \frac{N-1}{2M}$$

Para valores grandes N y M, este valor se aproxima a 1,5 para el caso de $M = N$. De esta forma, esta técnica proporciona un almacenamiento compacto con una operación de consulta rápida.