4 Gestión de la memoria

En este capítulo se estudian las técnicas de gestión de la memoria, partiendo de las utilizadas en los primeros sistemas operativos para llegar a la memoria virtual, a la que se dedica especial atención. Se describe el soporte de la memoria virtual y las políticas que se aplican en los diversos aspectos de su gestión. Los ejemplos introducidos pretenden aportar una perspectiva histórica.

Contenido

4.1 Introducción

En la memoria física de un computador coexisten el sistema operativo, las rutinas de enlace dinámico y los programas de usuario. En los sistemas operativos modernos la gestión de memoria resuelve aspectos como:

- La carga de programas y su ubicación. Hay que establecer la correspondencia entre las direcciones lógicas del programa y su ubicación física en memoria.
- La presencia simultánea de más de un programa en memoria.
- La posibilidad de cargar rutinas en tiempo de ejecución (rutinas de enlace dinámico[1]).
- La compartición de espacios de memoria por varios programas.
- La ejecución de programas que no caben completos en memoria.
- La gestión eficiente del espacio de memoria libre.

A lo largo de la historia, los sistemas operativos han ido introduciendo conceptos y mecanismos hasta llegar a ofrecer las características comentadas. Como el camino ha sido largo y son muchos los aspectos que se combinan hoy en día, conviene revisar las políticas de gestión de memoria teniendo en cuenta las propiedades fundamentales que pueden ofrecer:

(a) Número de programas que puede haber en memoria: un programa o varios programas.

(b) Si los programas pueden salir y entrar de memoria durante su ejecución: permanentes o no permanentes

(c) Si un programa debe ocupar posiciones consecutivas de memoria: contiguo o no contiguo

(d) Si un programa debe estar cargado entero para poder ejecutarse: entero o no entero

[1] A las librerías de rutinas de enlace dinámico se las conoce habitualmente como *Run-Time Libraries* o *Dynamic Link Libraries* (DLLs).

No todas las combinaciones de estas propiedades tienen sentido. Las políticas de gestión que se encuentran en los sistemas operativos a lo largo de la historia, a medida que se van eliminando restricciones, son las siguientes:

(1) Programa único, permanente, contiguo y entero: **monitor residente**.

(2) Varios programas, permanentes, contiguos y enteros: **particionado fijo** (***MFT***) o **variable** (***MVT***).

(3) Programas no permanentes (***swapping***).

(4) Programas no contiguos: **paginación** y **segmentación**.

(5) Programas no contiguos y no enteros: **enlace dinámico** y **memoria virtual**.

Además, es posible (y a veces conveniente) combinar políticas. Por ejemplo, la memoria virtual no excluye el enlace dinámico, se basa en la paginación y a veces se combina con el swapping de programas.

Los conceptos sobre evaluación de rendimiento introducidos en el Capítulo 1 se aplican directamente a la gestión de la memoria. Hay que recordar que se consideran básicamente dos aspectos, uno espacial y otro temporal:

- La eficiencia que consigue el mecanismo en la utilización del espacio de memoria física del computador (eficiencia espacial). Las pérdidas de eficiencia están provocadas por la fragmentación, interna o externa. La fragmentación externa, además, provoca la degradación de la memoria y la necesidad de compactar, lo que a su vez incide en pérdida de eficiencia temporal.

- La pérdida de eficiencia temporal en la CPU (*overhead*) que produce la aplicación de la política. Hay que considerar si requiere soporte hardware y mecanismos de protección adicionales, y el coste de ejecutar el código necesario para la gestión.

A continuación estudiaremos las políticas de gestión de memoria atendiendo a (1) los mecanismos hardware y de protección que requiere, (2) cómo resuelve la carga y reubicación de los programas, y (3) las necesidades de gestión que plantea al sistema operativo. Hay que incidir en que los sistemas operativos actuales utilizan memoria virtual soportada por paginación, a veces en combinación con swapping de procesos, además de proporcionar soporte para el enlace dinámico. El resto de las políticas servirán para introducir conceptos y como revisión histórica.

4.2 Sistemas primitivos

Los sistemas operativos de los primeros computadores (o, más recientemente, los sistemas operativos para los primeros ordenadores personales) almacenaban los programas de la manera más simple en cuanto a soporte hardware (enteros y en posiciones físicas contiguas), y en cuanto a necesidades de gestión (permanentes). Estas primeras políticas de gestión de memoria, que hoy en día únicamente pueden encontrarse en sistemas empotrados, son interesantes porque introducen conceptos y mecanismos a partir de los cuales se evoluciona hacia conceptos más elaborados.

4.2.1 Monitor residente

La forma más primitiva de trabajo consiste en cargar un único programa en memoria. El encargado de hacerlo era el monitor residente, que más que un sistema operativo es un precursor de estos. La memoria se divide en dos partes, sistema operativo (monitor) y programa de usuario (Figura 4.1).

Requiere un registro barrera para protección.

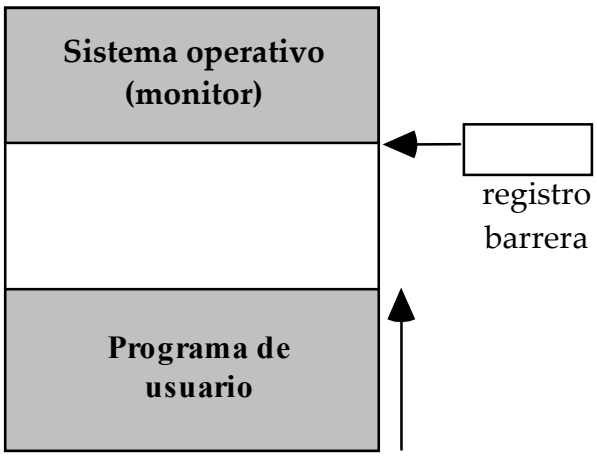

Figura 4.1. Monitor residente.

Los direccionamientos pueden ser absolutos. El cargador conoce el tamaño del programa y lo carga a partir de una dirección fija.

No puede decirse que el monitor residente realice una gestión de la memoria como tal.

4.2.2 Particiones

El sistema operativo es capaz de gestionar la coexistencia de varios programas en memoria asignando a cada uno un espacio contiguo (partición). El particionado puede ser fijo o variable.

Como soporte hardware para protección requiere dos registros:

- registro límite inferior (base).
- registro límite superior o registro longitud

Se requiere reubicación en tiempo de carga. El cargador puede establecer direcciones absolutas (reubicación estática), o establecer direcciones relativas a un registro base (reubicación dinámica). En particionado variable se requiere obligatoriamente reubicación dinámica, como veremos.

4.2.2.1 Particionado fijo

La memoria se divide en un conjunto de particiones de tamaños preestablecidos (Figura 4.2). Este mecanismo se denomina históricamente *MFT* (multiprogramación con un número fijo de tareas[2]).

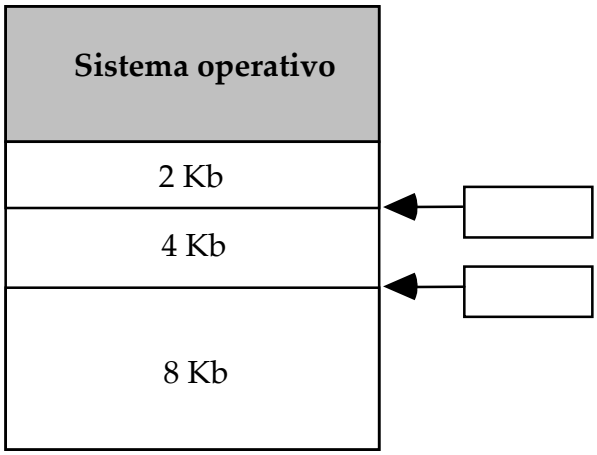

Figura 4.2. Memoria con particiones de tamaño fijo

En cada partición se ubica un único programa. Las particiones pueden ser iguales o de diferentes tamaños. En este último caso se aprovecha mejor la memoria, al poder almacenar un programa en la partición que mejor se ajuste a su tamaño. Dos consecuencias de este mecanismo son las siguientes:

- El número de programas está limitado (grado fijo de multiprogramación).
- Produce fragmentación interna en cada partición (trozo de la partición desocupada).

Los programas se encolan para poder ser cargados en memoria. Existen dos alternativas:

---

[2] Esta terminología se debe a IBM.

(a) Una cola por cada partición. Un programa se encola en la cola que le corresponde por su tamaño. Tiene el inconveniente de que puede dejar fuera a un programa habiendo memoria libre para cargarlo.

(b) Cola única. Cuando un programa encuentra ocupada la partición que corresponde a su tamaño cabe la posibilidad de asignarle una partición libre de tamaño mayor, permitiendo incrementar el grado de multiprogramación a costa de introducir fragmentación interna. El sistema puede utilizar heurísticos que maximicen el grado de multiprogramación minimizando la fragmentación, consistentes en alterar la disciplina de la cola buscando en ella los programas que se adapten mejor a los huecos disponibles, para minimizar la fragmentación interna.

Para la gestión del espacio libre es adecuado el mapa de bits.

4.2.2.2 Particionado variable

Como ya sabemos, MFT presenta problemas de fragmentación interna. La alternativa es el particionado variable de la memoria, que se denomina *MVT* (multiprogramación con un número variable de tareas).

Ya es conocido que el particionado variable introduce fragmentación externa al ir quedando huecos entre particiones a medida que finalizan programas. En estos huecos sólo se pueden ubicar programas de menor tamaño que el hueco, que cuando acaben dejarán un hueco aún menor, y así sucesivamente (degradación de la memoria). La necesidad de compactar implica reubicar dinámicamente el programa, requiriendo el soporte adecuado para ello.

La carga de programas se gestiona mediante una única cola. La memoria libre se representa bien con una lista de huecos, bien mediante un mapa de bits, aunque en este caso es conveniente definir una unidad de asignación de memoria mayor que el byte para mantener el tamaño del mapa en unas dimensiones razonables.

La elección del hueco donde ubicar el programa puede hacerse siguiendo las diferentes políticas de asignación introducidas en el Capítulo 1. Como se vio, first-fit y next-fit son las más adecuadas.

4.3 Swapping

Es un mecanismo para mover programas entre memoria principal y secundaria, normalmente disco (**dispositivo se *swap***). Con swapping, los programas pueden salir/entrar de/a memoria durante su tiempo de ejecución. Normalmente, un programa abandona la memoria para dejar espacio a otro. El swapping modifica el

grafo de transición de estados de los procesos en un sistema multiprogramado, desdoblando los estados de bloqueado y preparado en dentro y fuera de memoria.

La función del sistema operativo que gestiona el intercambio entre disco y memoria se denomina intercambiador o ***swapper***. La operación de escribir el programa en disco se conoce como ***swap-out***, mientras que leer el programa de disco se denomina ***swap-in***.

El swapping aporta las siguientes ventajas:

- Permite influir en la gestión de procesos para controlar el grado de multiprogramación (planificación a medio plazo).
- Proporciona flexibilidad en la gestión de la memoria, permitiendo una utilización más eficiente del espacio.

Para soportar swapping se requiere espacio para el intercambio en almacenamiento secundario, generalmente disco. Se puede utilizar un dispositivo específico independiente, una partición del disco, o incluso compartir la misma del sistema de ficheros.

El direccionamiento de los programas debe ser relativo a un registro base (reubicación dinámica). El swapper establece el nuevo valor del registro base para un proceso cada vez que lo carga en memoria.

El sacar un programa de memoria está motivado por la necesidad de obtener espacio libre, generalmente para ejecutar otro programa (quizás uno más prioritario). El swapper debe seleccionar cuidadosamente qué programas van a salir. Ya que, como hemos comentado, el swapping condiciona la planificación de procesos, algunos de los criterios a aplicar para seleccionar el proceso a sacar están relacionados con los parámetros de la planificación. Los criterios suelen ser:

- El estado del proceso. Los programas bloqueados no plantean una necesidad inmediata de proceso.
- La prioridad del proceso.
- El tamaño del programa. Los programas mayores liberan más espacio al salir de memoria, pero será más costoso cargarlos de nuevo[3].
- El tiempo que el programa lleva en memoria.

---

[3] Como veremos, en memoria virtual la paginación por demanda mitigará este problema.

El sacar procesos bloqueados plantea problemas cuando la razón del bloqueo es una operación de entrada/salida que se realice por DMA sobre un buffer de usuario (típicamente una operación de *leer* o *escribir*). Hay dos alternativas:

(a) Impedir el swap-out de los procesos con operaciones de DMA pendientes.

(b) La entrada/salida se hace siempre sobre buffers del sistema (residentes). Como se verá en la gestión de dispositivos, esta solución aportará otras ventajas adicionales.

4.4 Paginación y segmentación

Permiten la ubicación no contigua de programas para combatir la fragmentación y la degradación de la memoria. Al poder ubicarse de forma no contigua, un programa ya no necesita un hueco de su tamaño, sino que la cantidad total de memoria libre sea mayor o igual.

La ubicación no contigua requiere dividir los programas en trozos. En paginación, que es un caso particular del particionado fijo, el programa se divide en **páginas** del mismo tamaño. La segmentación, que es un caso particular de particionado variable, divide el programa en sus unidades lógicas (código, pila, datos, …), denominadas **segmentos**.

Las direcciones lógicas de los programas no contiguos presentan una gran independencia de su ubicación física (**direccionamiento virtual**), proporcionada sobre la base de tablas de traducción de direcciones. Otra ventaja de los programas no contiguos es que facilita que varios programas **compartan** trozos entre ellos, por ejemplo el código, lo que permite un ahorro importante de memoria y de tiempo de carga.

4.4.1 Soporte hardware, protección y compartición

Es necesario soporte hardware para la traducción de direcciones. Las direcciones lógicas generadas por el programa se dividen en número de página (o segmento) y desplazamiento dentro de la página. La traducción del número de página (o segmento) a **marco de página** en memoria física (o dirección de comienzo del segmento) se realiza en tiempo de ejecución mediante una **tabla de páginas** (o **tabla de segmentos**) asociada al programa, que habitualmente reside en memoria. Además, es conveniente hardware adicional para acelerar la traducción (*TLB*, *MMU*), ya que cada referencia a memoria implica dos accesos a memoria física.

Se requiere un registro apuntador a la base de la tabla de páginas (*PTBR*) del programa o a la tabla de segmentos (*STBR*). El cálculo de la dirección física es más

simple en paginación, ya que sólo requiere la concatenación del número de marco y el desplazamiento, mientras que la segmentación implica una suma de la dirección base del segmento y el desplazamiento. Las Figuras 4.3 y 4.4 representan respectivamente los esquemas de direccionamiento paginado y segmentado.

Para protección, en la paginación se puede asociar un conjunto de bits a cada entrada de la tabla de páginas:

- bit de sólo lectura,
- bits para restricciones de acceso (en sistemas multiusuario).

En segmentación, la protección es más sencilla y natural, al establecerse de acuerdo a divisiones lógicas del programa. La tabla de segmentos especifica también la longitud de cada segmento, lo que proporciona un mecanismo adicional para tratar errores de direccionamiento fuera de segmento, mediante un trap que se genera si el desplazamiento supera la longitud del segmento.

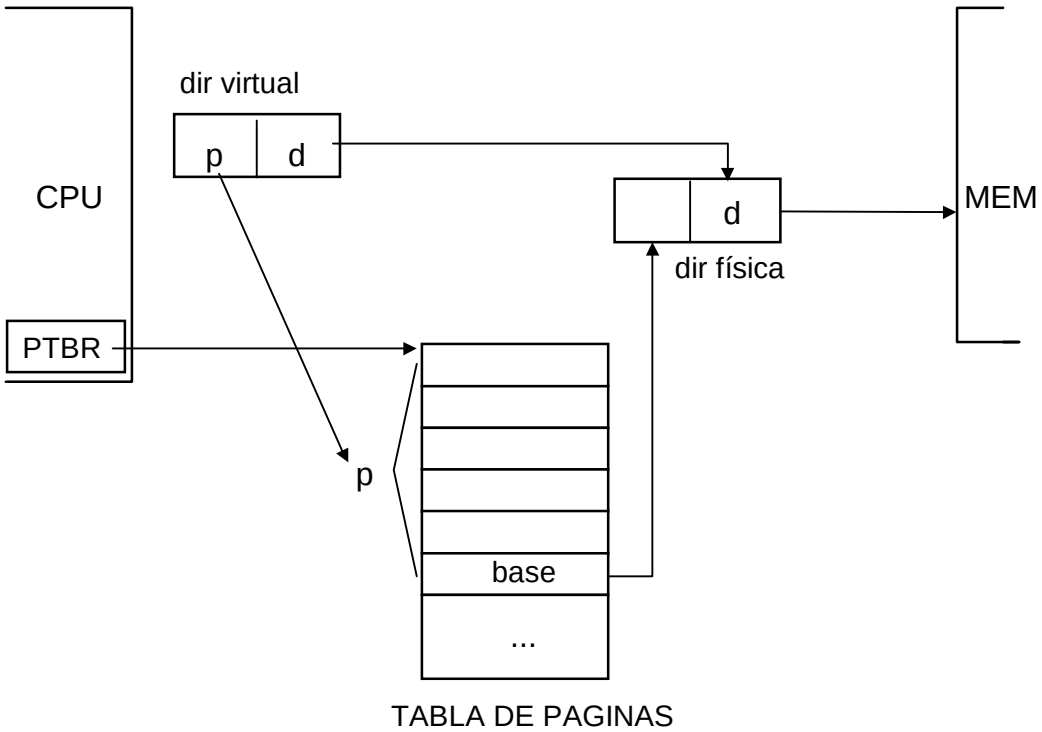

Figura 4.3 Direccionamiento con paginación

4.4.2 Carga de programas y reubicación

Las tablas de páginas o segmentos proporcionan reubicación dinámica a nivel de página o de segmento. En paginación, el cargador busca marcos de página libres en memoria, carga las páginas del programa, crea la tabla de páginas con la

correspondencia y carga su dirección en el PTBR. En segmentación el proceso es similar, salvo que requiere la asignación de huecos adecuados al tamaño de cada segmento del programa, así como la colaboración del compilador y el montador para establecer los tamaños.

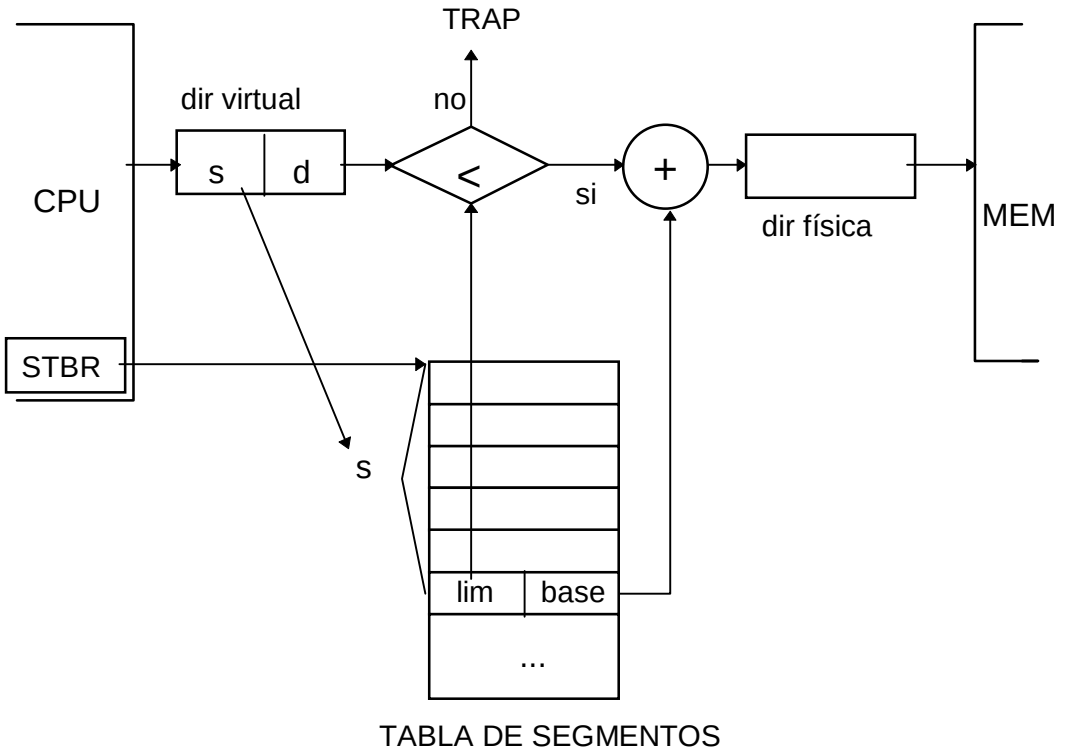

Figura 4.4. Direccionamiento con segmentación

4.4.3 Gestión

La traducción de direcciones, un mecanismo complejo y crítico para el rendimiento, ha de gestionarse a nivel hardware, genéricamente por una unidad de gestión de memoria (MMU).

La memoria libre en segmentación se gestiona bajo los criterios de las particiones de tamaño variable, mientras que en paginación es adecuado un mapa de bits.

La paginación presenta fragmentación interna en las páginas. Esto no ocurre en la segmentación, si bien ahora aparece fragmentación externa al utilizar unidades de ubicación de tamaños diferentes. Ya que el tamaño de los segmentos es típicamente mayor que el de las páginas, el problema de la fragmentación en la segmentación es más importante.

El mismo mecanismo permite que los programas compartan páginas o, más propiamente, segmentos. Por ejemplo, una única copia de código puede compartirse por varios programas simplemente haciendo que las entradas correspondientes al

segmento de código en sus tablas de segmentos (o el conjunto de entradas de sus tablas de páginas, en paginación) coincidan. Los bits de sólo lectura estarán activados, para evitar que un programa pueda corromper el código.

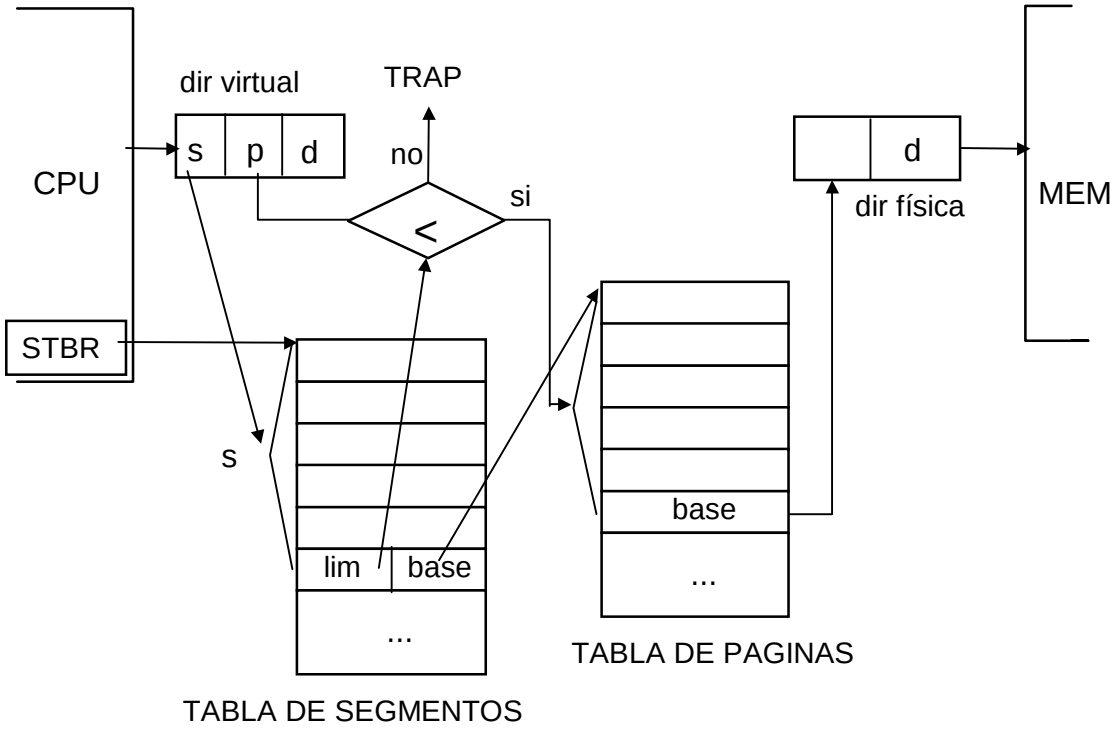

Figura 4.5. Direccionamiento en sistemas segmentado-paginados.

4.4.4 Sistemas combinados

Para evitar el problema de la fragmentación externa en segmentación pura, y para evitar el tener cargada una tabla de páginas demasiado grande en paginación, aparecen sistemas que combinan paginación y segmentación. Se denominan sistemas de **segmentación paginada** (o segmentado-paginados) o de **paginación segmentada** (o paginado-segmentados), según se entienda que los segmentos se dividen en páginas o que la tabla de páginas se segmenta en varias, respectivamente[4]. La dirección lógica se divide en tres partes: número de segmento, de página y desplazamiento dentro de la página. El número de segmento identifica en la tabla de segmentos la dirección base de una tabla de páginas, que se direcciona mediante el número de página para obtener el marco de página en memoria, según se ilustra en la Figura 4.5. El cálculo de la dirección lógica es ahora más complejo y lento,

[4] A efectos de implementación, ambos enfoques conducen al mismo esquema en el cual los segmentos de un programa se dividen en páginas. La inversa (dividir las páginas en segmentos) carece de sentido, ya que, como se desprende de un análisis inmediato, no aporta ninguna de las ventajas perseguidas.

requiriendo dos indirecciones. El soporte hardware para traducción (TLB) es ahora aún más importante.

Son también habituales los sistemas doblemente paginados (con dos niveles de paginación), que siguen un esquema parecido. Generalizando, se pueden concebir sistemas que dividen la dirección virtual en más de tres partes, especificando diferentes niveles de paginación y/o segmentación[5].

4.5 Enlace dinámico

La necesidad de ejecutar programas de tamaño mayor que la memoria física disponible ha originado a lo largo de la historia la introducción de sofisticados mecanismos de gestión de la ubicación de programas en memoria. Un primer enfoque es el de dividir en programa en varios módulos, denominados *overlays* o **solapamientos**, a partir de su estructura y cargar dinámicamente sólo aquellos que se necesiten. La carga de un solapamiento la realiza una rutina de enlace, que se encarga también de seleccionar el(los) solapamiento(s) reemplazado(s), si se requiere espacio para la carga. Tradicionalmente, el programador determinaba qué rutinas constituían los solapamientos, construía la rutina de enlace y las tablas de rutinas de los solapamientos, y elaboraba el programa de forma que las llamadas a las rutinas de los solapamientos se llamasen a través de la rutina de enlace. También eran suyos los criterios de reemplazo de los solapamientos.

El enfoque alternativo al problema es la memoria virtual, más general y completamente transparente al programador, al que dedicaremos la siguiente Sección.

Sin embargo, una evolución de los overlays ha quedado en los sistemas de hoy en día para cubrir un objetivo diferente del originalmente asignado a aquellos: las **rutinas de enlace dinámico**[6].

Una librería de enlace dinámico es un conjunto de rutinas agrupadas en módulos que se cargan dinámicamente en el momento de la llamada, como ocurría en los overlays, pero de forma transparente al programa. El sistema operativo proporciona como soporte una llamada al sistema donde se especifica como parámetro la rutina de enlace dinámico a ejecutar. Cuando en un programa aparece una llamada a una rutina de este tipo, el montador, en vez de enlazar estáticamente a la rutina en el código ejecutable, enlaza a la llamada al sistema. Cuando esta llamada al sistema se

---

[5] Siempre soportados por un último nivel de paginación.

6 A las librerías que las contienen se les llama Librerías de Enlace Dinámico (*Dynamic Link Libraries, DLLs*) o *Run-Time Libraries*.

ejecuta, comprueba si el módulo que contiene la rutina especificada como parámetro está cargado en memoria. Si no lo está, lo carga y establece la referencia (indirecta) desde el programa. A esto se le llama **montaje dinámico** o **en tiempo de ejecución**.

Se requiere una tabla que especifique el estado de cada módulo de enlace dinámico (cargado o no) y la dirección de memoria donde se carga. En principio, hay que destinar un espacio de memoria específico para los módulos de enlace dinámico, y proporcionar una política de reemplazo, pero cuando se combina con memoria virtual, que es lo habitual hoy en día, el reemplazo se realiza página a página de forma integrada en el propio mecanismo de memoria virtual.

Aunque esta técnica se podría utilizar con el mismo objetivo de los overlays, actualmente, ya que todos los sistemas tienen soporte de memoria virtual, el enlace dinámico se utiliza por otros motivos:

- Los ficheros ejecutables de los programas almacenados en disco son más pequeños. Esto tiene cierta importancia si se tiene en cuenta que hoy en día los programas hacen un amplio uso de rutinas que consumen un gran volumen de espacio, como por ejemplo las rutinas de tratamiento gráfico.
- Las rutinas de enlace dinámico cargadas en memoria se comparten por varios programas, lo que permite ahorrar espacio también en memoria.
- El tiempo de carga de los programas, y por lo tanto su latencia, es menor.
- Facilita la actualización de las aplicaciones. La instalación de una nueva aplicación aporta nuevas versiones de las librerías de enlace dinámico (por ejemplo, rutinas de tratamiento gráfico más elaborado) que sustituyen a las versiones antiguas, de lo que se benefician otras aplicaciones que usan dichas rutinas

4.6 Memoria virtual

La memoria virtual es el mecanismo más general para la ejecución de programas no enteros en memoria. Se basa en un sistema de paginación (o combinado) en el que sólo un subconjunto de las páginas del programa están cargadas en memoria. El resto reside en un dispositivo de almacenamiento secundario, análogamente al de swap[7]. La memoria virtual presenta, adicionalmente a su capacidad para ejecutar programas mayores que la memoria física disponible, un conjunto de interesantes ventajas con respecto a la paginación con programas enteros:

---

[7] Como se verá más adelante, la memoria virtual se suele combinar con el swapping de procesos, por lo que el dispositivo de paginación y swap serán una misma cosa.

- Reduce la latencia en la ejecución de los programas, al no tener éstos que cargarse completamente para comenzar a ejecutarse.

- Permite gestionar más eficientemente la memoria física. Cualquier espacio libre, incluso una única página, puede ser aprovechado para cargar un nuevo programa y comenzar a ejecutarlo. Por otra parte, si una página de un programa no se referencia durante la ejecución, no habrá que cargarla.

- Al aumentar el grado de multiprogramación a costa de reducir el número de páginas cargadas de cada programa, permite incrementar la eficiencia de la CPU en sistemas multiprogramados[8].

- Ahora la independencia de los programas con respecto a la máquina es completa. Además del direccionamiento virtual que aporta la paginación, la cantidad de memoria física disponible para ejecutar el programa sólo es relevante para la velocidad de ejecución del programa.

4.6.1 Soporte hardware

Además del soporte hardware para la traducción de direcciones de los sistemas paginados, la memoria virtual requiere mecanismos hardware adicionales:

- **Espacio para paginación** en un dispositivo de almacenamiento secundario (disco).

- **Bit de validez**, *V*. Para cada entrada de la tabla de páginas es necesario un bit que indique si la página correspondiente está cargada en memoria o no.

- **Trap de fallo de página**. Cuando la página referenciada no está cargada en memoria, el mecanismo de interrupciones produce el salto a la **rutina de tratamiento del fallo de página** (que promoverá la carga de la página en memoria). A diferencia de una interrupción normal, el fallo de página puede ocurrir en cualquier referencia a memoria durante la ejecución de la instrucción, por lo que la arquitectura debe proporcionar los mecanismos adecuados para establecer un estado del procesador consistente antes de saltar a la rutina de tratamiento.

- Información adicional para la gestión del fallo de página (bit de página modificada, referenciada, …). Se verá más adelante.

---

[8] Como se verá, esta ventaja tiene su contrapartida.

4.6.2 Carga de programas y reubicación

El cargador carga la tabla de páginas del programa y una o más páginas, estableciendo las direcciones de las páginas cargadas en las entradas de la tabla de páginas y los valores de los bits de validez. Almacena también las direcciones de disco donde se encuentran las páginas no cargadas en memoria. Como en paginación, la reubicación se hace independientemente para cada página, cuando éstas se cargan tras un fallo de página.

Básicamente, hay dos estrategias para cargar las páginas de un programa:

(a) Cargar sólo la primera página de código y dejar que el resto se cargue por fallo de página (**paginación por demanda**).

(b) Cargar un cierto número de páginas o todas, dependiendo de la memoria disponible y del tamaño del programa (**prepaginación**).

4.6.3 Gestión

El proceso de direccionamiento en un sistema de memoria virtual combina el mecanismo hardware de traducción de direcciones siguiendo el mecanismo habitual de paginación pura o combinada, con la intervención del sistema operativo cuanto se produce un fallo de página. En la referencia a memoria se comprueba el estado del bit de validez. Si está activado, la página está cargada y el proceso de traducción hardware acaba proporcionando la dirección física; en caso contrario, el trap de fallo de página aborta la ejecución de la instrucción en curso[9] y provoca el salto a la rutina de tratamiento del fallo de página, que ejecuta los siguientes pasos:

(1) Si la dirección generada por el programa no es correcta[10], se aborta la ejecución del programa. Si se trata de una dirección correcta, se busca un marco de página libre en memoria donde cargar la página referenciada. Si no hay marcos libres, se consigue uno expulsando una página de memoria. La elección de ésta (**página víctima**) se realiza mediante un **algoritmo de reemplazo**. El reemplazo de una página, que se estudiará más adelante, implica escribir en disco la página víctima si ésta hubiera sido modificada.

(2) Se programa la lectura de la página referenciada en el dispositivo de paginación. En los sistemas multiprogramados se produce un cambio de

[9] O de las instrucciones en curso, en procesadores segmentados.

[10] Es decir, no existe esa dirección en el programa. No hay que confundir dirección *correcta* y dirección *válida*.

contexto: el proceso que ha provocado el fallo de página pasa a estado bloqueado y se planifica otro proceso.

(3) Mientras el programa está bloqueado, se lee por DMA en el dispositivo de paginación la página buscada y se carga en el marco libre seleccionado.

(4) Finalizada la carga, se actualiza la entrada de la tabla de páginas y el bit de validez asociado. El proceso bloqueado por fallo de página pasa a preparado para ejecución. Dependiendo de la política de planificación de procesos, el proceso puede tener la oportunidad de entrar a ejecución.

4.6.4 Evaluación del rendimiento

Como podemos deducir del apartado anterior, el tratamiento de un fallo de página es muy costoso en tiempo comparado con un acceso a memoria paginada. Hay que tener en cuenta que en la actualidad un tiempo de acceso a memoria paginada típico (t_m) es del orden de 10^{-7} segundos o menos, mientras que el tiempo de tratamiento de un fallo de página (t_f) es del orden de la decena de milisegundos (básicamente el tiempo de acceso a disco), es decir cinco órdenes de magnitud superior.

Si la probabilidad de fallo de página es p_f, el tiempo medio de acceso a memoria virtual (t_v) se calcula como:

$$t_v = t_m + p_f\, t_f$$

Para hacernos una idea de en qué órdenes debe moverse la probabilidad de fallo de página para que el rendimiento del sistema no se resienta demasiado, vamos a partir de la relación de tiempos t_v/t_m. Obsérvese que $t_v/t_m - 1$ expresa una medida de la pérdida de rendimiento relativa a t_m resultante de introducir en un sistema paginado memoria virtual, en tanto por 1. Despejando p_f en la expresión de arriba, podemos expresarla en función de esta relación:

$$p_f = (t_v/t_m - 1)\ (t_m/t_f)$$

Por ejemplo, si se pretende que la pérdida de rendimiento en el acceso a memoria con memoria virtual no supere el 10% con respecto al tiempo de acceso en el mismo sistema sin memoria virtual, la probabilidad de fallo de página debería ser:

$$p_f < 0{,}1\ (t_m / t_f)$$

que para la relación típica t_m/t_f de 5 órdenes de magnitud razonada más arriba, da una probabilidad del orden de 10^{-6}.

Afortunadamente, hay algunos factores que permiten matizar estos resultados:

- Si el sistema no fuese de memoria virtual, debería cargar inicialmente todo el programa, lo que supone un tiempo equivalente al de tratar tantos fallos de página como páginas tiene el programa.

- La localidad espacial de los programas y el tamaño de las páginas (típicamente de varios Kb en la actualidad) aseguran que, cargadas las primeras páginas, habrá gran cantidad de referencias a ellas antes de producir un fallo.

- En sistemas multiprogramados, la CPU puede ejecutar otros procesos mientras se atiende un fallo de página.

- Con relación a otros niveles de la jerarquía de memoria (por ejemplo la memoria cache), en memoria virtual el espacio destinado en disco para área de paginación ha de configurarse de forma que no sea mucho mayor que el espacio la memoria física disponible[11].

Una política de reemplazo de páginas adecuada explotará ventajosamente la localidad temporal de los programas, haciendo posible un rendimiento razonable.

4.6.5 Reemplazo de páginas

Como hemos visto, cuando se produce un fallo de página y no hay marcos libres es necesario liberar uno de los marcos de página ocupados para poder cargar la página referenciada. Los pasos a seguir son los siguientes:

(1) Se selecciona la página víctima mediante un **algoritmo de reemplazo** que ejecute una política de reemplazo determinada.

(2) Si la página víctima había sido modificada durante su estancia en memoria, hay que escribirla en el dispositivo de paginación (***page-out***). Si no, esta operación no es necesaria. Para la gestión de páginas modificadas se asocia un **bit de página modificada** para cada marco de página, que se activa cada vez que se accede a memoria para escritura.

(3) Se pone a cero el bit de validez correspondiente a la página víctima en su tabla de páginas.

A continuación se sigue con el tratamiento del fallo de página como se describió anteriormente, leyendo del dispositivo de paginación la página que provocó el fallo y cargándola en el marco libre (***page-in***).

---

[11] Empíricamente se estima que es adecuado destinar como área de paginación un espacio del orden del doble de la memoria física de la máquina.

4.6.6 Políticas de reemplazo de páginas

Los criterios a seguir para implementar un algoritmo de reemplazo de páginas son fundamentalmente dos:

- Minimizar el número de fallos de página, como se ha razonado en el apartado anterior. Explotar la localidad temporal de los programas será fundamental.
- Sencillez de implementación. Un algoritmo complejo, como veremos, puede requerir intervención adicional en los accesos a memoria, lo que implicará o pérdida de rendimiento, o un hardware costoso, lo que a su vez redundará probablemente en pérdida de rendimiento[12].

A continuación se describen las políticas de reemplazo de páginas. Para probar los algoritmos de reemplazo y así poder evaluar su rendimiento comparativo se utiliza una **secuencia de referencias** a páginas (finita y determinista[13]), que permite determinar la tasa de fallos de página para cada algoritmo. Para simplificar, se considera que las referencias son producidas por un único programa. El Ejercicio 7 incluye una secuencia de referencias que permitirá probar el comportamiento de los algoritmos. Como se verá más adelante, en multiprogramación es necesario considerar que un conjunto de secuencias de referencias se intercalan en el tiempo.

4.6.6.1 Política óptima

En un algoritmo de reemplazo óptimo en cuanto al número de fallos de páginas, la página víctima a seleccionar será aquélla que más tiempo va a tardar en ser referenciada. Desafortunadamente, determinar las referencias a memoria futuras es irresoluble en la práctica, por lo que los algoritmos de reemplazo reales se basan habitualmente en el comportamiento pasado de las referencias a memoria, confiando en la propiedad de localidad temporal de los programas.

4.6.6.2 Política FIFO

Se elige como página víctima la que más tiempo lleva cargada.

---

[12] Los recursos hardware y software necesarios para soportar una política de reemplazo sofisticada podrían haberse dedicado más provechosamente en otras funciones, como, por ejemplo, memoria cache adicional.

[13] Si bien las secuencias de referencias generadas por la ejecución real de los programas no son deterministas.

La implementación puede hacerse mediante una cola FIFO de marcos de página, que se actualiza cada vez que se carga una página. Una alternativa es elegir la víctima sobre la base de un registro que almacena el tick en que se produjo la carga, lo que proporciona una aproximación a FIFO.

Al no basarse en criterios de localidad, FIFO no obtiene buenos resultados en cuanto a probabilidad de fallo. Además, presenta un problema conocido como **anomalía de Belady**, un fenómeno de pérdida de rendimiento (mayor número de fallos de página), que presentan algunos algoritmos de reemplazo con determinadas secuencias de referencias cuando se incrementa el número de marcos de página en memoria. Esto ocurre, por ejemplo, con la siguiente secuencia:

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

En concreto, con esta secuencia se producen más fallos de página con cuatro marcos que con tres.

4.6.6.3 Política LRU

De acuerdo al principio de localidad temporal en las referencias (es más probable la referencia a una página cuanto más recientemente se haya referenciado), se selecciona como página víctima la *menos recientemente referenciada*. Nótese que ello implica modificar en cada referencia la información acerca de las páginas cargadas, a diferencia del algoritmo FIFO, que sólo requiere hacerlo en la carga de cada página. Esta característica hace de LRU un **algoritmo de pila**. Los algoritmos de pila presentan la propiedad de que, dada una secuencia de referencias, una memoria con más marcos de página, tras un número determinado de referencias, mantiene el conjunto de páginas que contendría con un tamaño menor en la misma referencia. En otras palabras, la tasa de fallos nunca aumenta si se incrementa el tamaño de memoria. Los algoritmos de pila evitan la anomalía de Belady.

La implementación hardware de la política LRU es compleja. Hay dos mecanismos:

(a) Mantener una lista encadenada de páginas ordenadas por el tiempo de su última referencia.

(b) Mantener un contador de referencias y un registro (de al menos 64 bits) asociado a cada marco de página, que se actualiza en una referencia con el valor del contador.

En ambos casos la implementación es muy costosa, por lo que, como alternativa al algoritmo LRU puro se utilizan aproximaciones. Un mecanismo de aproximación consiste en almacenar el tick de reloj en el que se produce la referencia en vez de la propia referencia, para lo que se requiere un **bit de referencia** asociado a cada marco en base al cual la rutina de atención al reloj puede determinar si la página ha sido

referenciada desde el tick anterior y actualizar el registro de ticks de última referencia a la página, que puede implementarse por software. Con esta aproximación se pierde precisión, al convertir la relación de orden total entre las referencias en una relación de orden parcial (las referencias dentro de un mismo tick no están ordenadas).

Algunos de los siguientes algoritmos también son aproximaciones a LRU.

4.6.6.4 Segunda Oportunidad

Se distingue entre páginas que han sido referenciadas y páginas que no lo han sido. Si, siguiendo un orden determinado, una página debería ser seleccionada como víctima y ha sido referenciada, se deja en memoria, pero se marca como no referenciada. Será candidata a página víctima en el próximo fallo si no vuelve a ser referenciada antes.

Se requiere un bit de referencia, R, por marco de página, que se activa cuando se referencia la página. La política de la segunda oportunidad se puede implementar mediante el **algoritmo del reloj**. Los marcos se organizan en orden circular con un puntero a uno de ellos. Cuando ocurre un fallo, si el bit R correspondiente está a cero se elige esa página como víctima. Si no, R se pone a cero y se apunta al siguiente.

4.6.6.5 NRU

Se elige como página víctima una *no usada recientemente*.

Se mantienen dos bits por página:

R referenciada/no referenciada

M modificada/no modificada

Cuando una página se carga, sus bits R y M se ponen a cero. El bit R de una página se activa cuando ésta se referencia. El bit M se activa si la referencia es para escritura. Periódicamente (por ejemplo, con cada tick de reloj), todos los bits R se ponen a cero.

De acuerdo a estos bits, se pueden establecer cuatro categorías de páginas, según su grado de candidatura para ser elegidas páginas víctimas:

(1) no referenciadas y no modificadas

(2) no referenciadas y modificadas

(3) referenciadas y no modificadas

(4) referenciadas y modificadas

Entre páginas de una misma categoría puede aplicarse cualquier criterio de elección.

Presenta el problema de que en los momentos inmediatamente posteriores a la puesta a cero del bit R incluso páginas que están siendo actualmente referenciadas son susceptibles de ser elegidas como víctimas.

4.6.6.6 NFU

En el algoritmo NFU (página *no referenciada frecuentemente*), se establece si en un intervalo de tiempo (por ejemplo, un tick de reloj) una página ha sido o no referenciada. Se lleva la cuenta del número de intervalos que cada página ha sido referenciada, eligiendo como víctima la de cuenta más baja.

Para cada página se requiere, además del bit R que se activa en cada referencia, un contador. En cada tick de reloj, para cada página, el bit R se acumula en el contador. Comparado con LRU puro, en NFU los contadores se incrementan con mucha menor frecuencia, por lo que es implementable en software.

NFU, al recordar todas las referencias, no prima la localidad temporal de los programas, por lo que páginas que se han usado mucho en un pasado lejano permanecen en memoria por delante de páginas que están comenzando a usarse intensamente en el momento del fallo.

4.6.6.7 Envejecimiento

Como una mejora de NFU, se puede introducir un parámetro de olvido (o envejecimiento) sobre las referencias pasadas.

Un mecanismo para implementar el envejecimiento consiste en desplazar los contadores un bit a la derecha en cada tick de reloj, introduciendo el bit R por la izquierda. De esta forma, una página referenciada durante el último tick permanecerá en memoria sobre cualquiera que no lo haya sido.

4.6.7 Memoria virtual y multiprogramación

Hasta aquí hemos descrito la paginación en sistemas de memoria virtual de manera simplificada. Se ha supuesto que la gestión de páginas se hace de acuerdo a una secuencia de referencias única, como si fuese obtenida por un único programa. Esto es perfectamente válido en monoprogramación. Como veremos, en sistemas multiprogramados es muy interesante considerar qué programa genera cada referencia, lo que equivale a tener una secuencia de referencias para cada programa.

Para una secuencia de referencias única, la elección de la página víctima sólo tiene sentido entre todas las páginas de memoria. En cambio, en sistemas

multiprogramados, donde cada programa genera su propia secuencia de referencias, la localidad se mantiene entre las páginas de un mismo programa, pero no globalmente. Por lo tanto, parece una alternativa razonable considerar de manera independiente la secuencia de referencias de cada programa para seleccionar la página víctima. El subconjunto de páginas al que se aplica el algoritmo de reemplazo determina lo que se conoce como **rango de asignación**.

En sistemas multiprogramados, el rango de asignación y otros criterios de la gestión de la memoria virtual determinan en cierta medida el grado de multiprogramación del sistema, y por lo tanto influyen decisivamente en la planificación de procesos. En los siguientes apartados estudiaremos los aspectos relacionados con la asignación de páginas en sistemas operativos multiprogramados.

4.6.7.1 Rango de asignación

La gestión de la memoria virtual para un sistema operativo multiprogramado debe determinar entre qué páginas se aplica el algoritmo de reemplazo. Hay dos políticas alternativas:

(a) **Asignación global**. Las referencias generadas por todos los programas se consideran como una secuencia de referencias única, de forma que un fallo de página producido por un programa puede provocar que salga de memoria una página de otro programa (esta será la elección más probable para la mayor parte de los algoritmos de reemplazo, ya que las páginas del proceso en ejecución son precisamente las más recientemente referenciadas).

(b) **Asignación local**. El algoritmo de reemplazo se aplica exclusivamente entre las páginas del proceso. Hay que establecer los criterios para el reparto de los marcos de página entre los procesos. Fundamentalmente hay dos posibilidades:

(b1) Igual número de marcos para cada proceso. Si hay M marcos de página y N programas, cada programa dispone de M/N marcos de página. Este criterio penaliza excesivamente a los programas de gran tamaño.

(b2) Asignación proporcional. Si un proceso i ocupa s_i páginas, definiendo $S=\Sigma s_i$ como la suma del tamaño de los N procesos ($S>M$), la asignación de páginas para el proceso i, a_i, será:

$$a_i = (s_i / S\ )\ M$$

Este criterio se puede combinar con la prioridad del proceso.

Las alternativas descritas son extremos entre los que caben soluciones intermedias. Por ejemplo, la asignación global se puede suavizar introduciendo límites (máximos y mínimos) en el número de marcos de página que tiene asignado cada proceso.

4.6.7.2 Sobrepaginación

Debido a que la memoria virtual permite la existencia de programas no enteros en memoria, la capacidad de la memoria física deja de ser un factor limitante del grado de multiprogramación. De hecho, en principio, el grado de multiprogramación tiene como límite el número de marcos de página que caben en memoria.

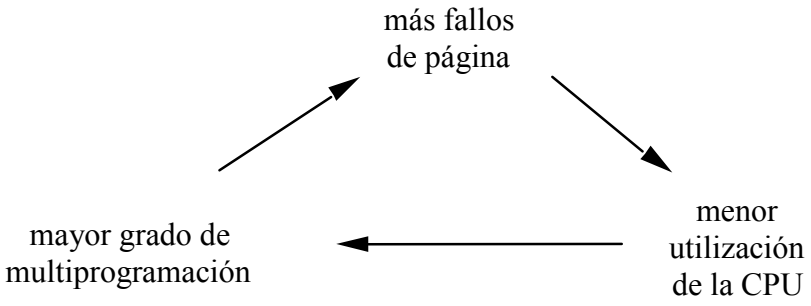

Figura 4.6. Realimentación que conduce a la sobrepaginación

En principio, un grado de multiprogramación alto incrementa la eficiencia de la CPU. Sin embargo, en sistemas con una fuerte carga, un número excesivo de programas en memoria conduce a tener muy pocas páginas de cada proceso y, por tanto, una probabilidad muy alta de fallo de página. Como el proceso que comete el fallo de página deja libre la CPU y pasa a estado bloqueado, se produce un cambio de contexto que pronto provocará un nuevo fallo de página al cambiar la localidad de las referencias. Esta situación se realimenta (Figura 4.6) hasta que la mayoría de los procesos estarán bloqueados por fallo de página, y la CPU tendrá una utilización muy baja. Esta caída drástica de la eficiencia (Figura 4.7) es lo que se conoce como **sobrepaginación** o ***thrashing***.

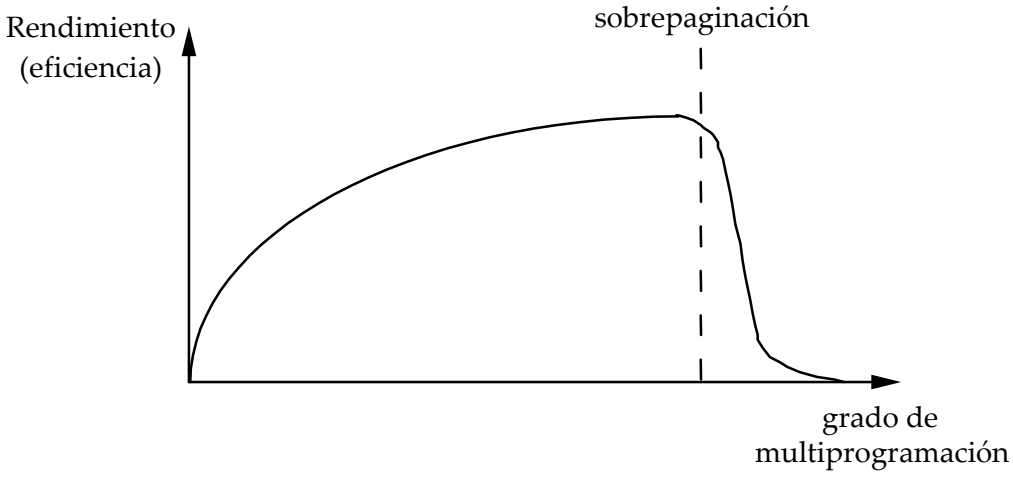

Figura 4.7. Caída del rendimiento por sobrepaginación.

4.6.7.3 Modelo de *working-set*

Se puede intentar evitar la sobrepaginación explotando hasta el límite la propiedad de localidad de los programas. Para un programa y en un instante de tiempo dado, es posible definir con bastante precisión un subconjunto de sus páginas que va a recibir la práctica totalidad de las referencias que genera el programa (***working-set*** o **conjunto de trabajo** del programa, ***WS***). Esto nos garantiza que dicho programa va a provocar muy pocos fallos de página. Si mantenemos en memoria únicamente conjuntos de trabajo completos, aseguraremos tasas de fallo de página bajas.

Implícitamente, la utilización de conjuntos de trabajo establece una nueva política de asignación que limita el grado de multiprogramación del sistema, *N*, ya que debe cumplirse:

$$\sum_{i=1}^{N} tamaño(WS_i) \leq M$$

Para definir conjuntos de trabajo (sobre la base de la propiedad de localidad temporal) es necesario llevar una traza de las últimas referencias de cada programa. Dicha traza puede verse como una ventana de tamaño Δ sobre la secuencia de referencias del programa, desde la referencia actual hacia atrás. El conjunto de páginas referenciadas en Δ es el WS del programa.

La elección de un Δ adecuado es muy importante. Una ventana demasiado pequeña no asumiría bien la localidad del programa. Una ventana demasiado grande abarcaría localidades de instantes de tiempo anteriores. Se ha mostrado experimentalmente que un valor razonable de Δ es del orden de 10.000 referencias.

La implementación de la ventana del WS sobre la base de una cuenta de referencias es costosa. Una aproximación razonable es utilizar el concepto de envejecimiento con la implementación que se vio en la sección dedicada a los algoritmos de reemplazo. En el caso más simple puede usarse únicamente el bit R y un Δ de un tick de reloj.

En el modelo de conjuntos de trabajo la asignación de marcos de página es local. Usualmente se definen tamaños máximos de WS.

Si para la carga del conjunto de trabajo de un programa se usa paginación por demanda, existirá un periodo transitorio hasta que se cargue el WS del proceso durante el cual el número de fallos de página puede ser muy elevado, lo que conduciría a sobrepaginación. Esto se evita cargando todo el conjunto de trabajo antes de planificar el proceso (prepaginación). De esta forma, el transitorio sólo se produce cuando se crea el proceso, cuando todavía no tiene establecido su WS.

Nótese que la gestión de la memoria mediante conjuntos de trabajo es una forma de combinar memoria virtual y swapping.

4.6.8 Otros aspectos de la memoria virtual

4.6.8.1 Fijación de páginas en memoria

Considérense los siguientes casos en un sistema con asignación global:

Caso A:

(1) Un proceso P_1 se bloquea para leer de disco en un buffer de memoria.
(2) Entra a ejecución otro proceso, P_2.
(3) P_2 comete fallo de página.
(4) La página de P_1 que contiene el buffer de E/S es elegida como víctima.

Caso B:

(1) Un proceso P_1 se bloquea por fallo de página.
(2) Entra a ejecución otro proceso, P_2.
(3) La página pedida por P_1 se carga en memoria y P_1 a preparado para ejecución.
(4) P_2 comete fallo de página.
(5) La página recién cargada de P_1 es elegida como víctima.

El Caso A se introdujo con el swapping de procesos, y se vio entonces que una de las alternativas es definir los buffers de E/S en el espacio del sistema operativo. El Caso B es una nueva consecuencia de la memoria virtual. Una solución general aplicable a ambas situaciones es evitar que dichas páginas salgan de memoria mediante un **bit de fijación** de la página en memoria que la hace no elegible como víctima.

4.6.8.2 Páginas iniciadas a cero

Las páginas iniciadas a cero en tiempo de compilación (por ejemplo, arrays de gran tamaño) no se almacenan en el fichero ejecutable. El montador crea un segmento específico y las entradas correspondientes de la tabla de páginas se marcan como inválidas y se señalan como páginas iniciadas a cero mediante un bit específico. Al ser referenciada una de estas páginas, la rutina de fallo de páginas asigna un marco y lo inicia a ceros.

4.6.8.3 Páginas copy-on-reference

Este bit permite la gestión de páginas compartidas modificables, de utilidad en sistemas donde los procesos hijos heredan las características de padre, como es el caso de UNIX (llamada al sistema *fork*). Las páginas del nuevo proceso, en vez de copiarse en la creación del proceso, se comparten y marcan con un bit de *copy-on-*

reference, para duplicarse bajo demanda sólo si fuese necesario. Cuando un proceso referencie para escritura una página de este tipo, se le proporciona una copia privada de la página, desactivándose entonces el bit.

4.7 Ejemplos

4.7.1 VAX/VMS

El sistema operativo VMS, desarrollado para la familia VAX-11 de Digital, introducida a finales de los 70, proporciona un entorno unificado para hardware con diferentes niveles de rendimiento. La gestión de memoria se encontraba muchas veces con un soporte hardware muy limitado. Aunque prácticamente en desuso, su estudio tiene interés por razones históricas, como veremos.

El VAX-11 proporcionaba un espacio de direccionamiento virtual de 32 bits, repartidos entre usuario y sistema. Las direcciones eran paginadas, con un tamaño de página de 512 bytes. Un programa de usuario disponía de tres *regiones* (segmentos), cada uno con una tabla de páginas con registros de base y de longitud asociados[14]. Como soporte para memoria virtual, cada entrada de las tablas de páginas contaba, al menos, con un bit de validez y un conjunto de bits para protección de acceso.

Las principales características de la gestión de la memoria virtual en VMS son las siguientes:

Asignación de páginas local. Existe un límite máximo en el número de páginas *residentes* por proceso (su *working-set*, en terminología VAX/VMS). Los procesos pueden ser *expulsados* de memoria para evitar la sobrepaginación. Cuando un proceso vuelve a memoria lo hace con el mismo conjunto de páginas residentes con que fue expulsado.

Paginación con reemplazo FIFO y buffer de marcos global. El paginador mantiene un número adicional de marcos de página que no están asignados a los conjuntos residentes de cada proceso. Este buffer de marcos se compone de dos listas: lista de marcos modificados y lista de marcos libres. Un marco elegido como víctima se añade a una de estas listas dependiendo de si la página ha sido o no modificada. Cuando un proceso hace referencia a una página que no está en su conjunto residente, la rutina de fallo de página busca primero en estas listas; si lo encuentra, lo añade a su conjunto residente sin necesidad de acceder a memoria secundaria. Sólo cuando el número de marcos libres alcanza un límite mínimo se produce realmente

[14] En la dirección virtual, la región se especificaba con 2 bits y la página con 21.

una expulsión de página al eliminarse elementos de la lista con disciplina FIFO[15]. En el caso de la lista de marcos modificados, éstos se escriben conjuntamente en disco.

Agrupamiento de páginas. El pequeño tamaño de la página del VAX causaría un rendimiento muy pobre si éstas se transfirieran individualmente entre memoria y disco. Para evitarlo, las páginas de un proceso se agrupan para transferirse conjuntamente. En el caso de lectura, por ejemplo, se transfieren las páginas del conjunto residente de un proceso. En el caso de escritura, se transfieren las de la lista de modificadas.[16]

Swapping. En determinadas circunstancias, la única forma de recuperar marcos libres es sacando de memoria el working-set completo de uno o más procesos, para lo que se tiene en cuenta la prioridad y un *quantum* de memoria[17] de los procesos.

4.7.2 UNIX

UNIX es un sistema operativo diseñado para ser instalado sobre una gran variedad de máquinas. Ya que la gestión de la memoria es muy dependiente del soporte hardware disponible, nos encontraremos con que existen diferentes formas de gestión de memoria en UNIX. Así, las primeras implementaciones de UNIX, sobre máquinas PDP-11, no trabajaban con memoria virtual. Las implementaciones posteriores proporcionan memoria virtual con paginación por demanda combinado con swapping de procesos. A continuación describiremos la gestión de la memoria en el System V.

Para cada entrada de la tabla de páginas se considera la dirección física del marco de página, bits de protección (r, w, x), bit de validez, bit de referencia, bit de página modificada, y un contador de antigüedad de la página. Estrictamente, sólo la dirección física y el bit de validez deben estar obligatoriamente soportados por hardware. El resto de la información puede simularse en software (como ocurría en

---

15 Esta estrategia equivale a una implementación software de un algoritmo de reemplazo NRU. Al no poder contar siempre con un bit de referencia hardware, es el bit de validez el que hace su papel; la rutina de atención al fallo de página representa implícitamente las páginas no referenciadas recientemente dentro de las listas de marcos libres y modificados.

16 El paginador organiza la ubicación de las páginas en disco de manera que se almacenen en lo posible en bloques contiguos.

17 El quantum de memoria se refiere a la cantidad de tiempo que un proceso puede estar residente en memoria, y no coincide con el concepto de *quantum* que habitualmente se usa para describir modelos Round-Robin, referido al tiempo que un proceso puede estar *consecutivamente* en la CPU. Al contrario, en el VAX/VMS se utiliza el término quantum para referirse al tiempo *acumulado* de CPU de un proceso en memoria.

el VAX-11). Además, para cada entrada de la tabla de páginas se almacena la información para acceder a la página en memoria secundaria (descriptor de bloque del disco).

Otra estructura de información que se mantiene es una *lista de marcos de página,* que guarda información acerca del estado de la página (asignada, en fichero ejecutable, libre, cargándose por DMA), el número de procesos que referencian el marco (para compartición), y el dispositivo que contiene una copia de la página y el número de bloque.

Algunas características de la gestión de la memoria virtual en el Sistema V son los siguientes:

Asignación global y working-set. UNIX define el working-set de un proceso como el conjunto de páginas asignadas a un proceso. Los marcos no asignados a ningún proceso son marcos libres asignables a cualquier proceso. El sistema define el número mínimo de marcos libres, que comprueba periódicamente. Si en un instante de tiempo no se alcanza dicho límite, un proceso paginador (*page-stealer*) entra para *envejecer* las páginas. Toda referencia a una página pone a cero la edad de la página. Si la edad alcanza un valor *n*, establecido por la instalación, la página pasa a estado libre. El diagrama de transición de estados que ilustra este mecanismo se muestra en la Figura 4.8. La página de un marco libre puede ser rescatada en una referencia si el marco no hubiera sido asignado antes. Este mecanismo se ha implementado en los Sistemas V hasta la versión 3, y es utilizado también por Linux. El SVR4 ha simplificado el reemplazo de páginas prescindiendo del contador de edad. El paginador se limita a poner a cero el bit de referencia (lo que equivale a envejecer hasta n=1) y a liberar las páginas con R=0 siguiendo el algoritmo del reloj.

Escritura de páginas en disco por grupos. Si el paginador decide liberar una página y la página se ha modificado o no existe una copia en disco (por ejemplo, página iniciada a ceros), el paginador pone la página en una lista de páginas a escribir. Si el número de elementos de esta lista alcanza un cierto límite, las páginas se escriben físicamente en el disco.

Swapping. En situaciones de mucha demanda de memoria por parte de los procesos, el paginador puede no ser capaz de conseguir marcos libres a la velocidad necesaria. Cuando la rutina de fallo de página no es capaz de mantener el nivel mínimo de marcos libres, entra el proceso *swapper* para sacar algún proceso de memoria. Los criterios de elección dependen de la versión de UNIX, pero fundamentalmente se basan en el estado del proceso y su prioridad.

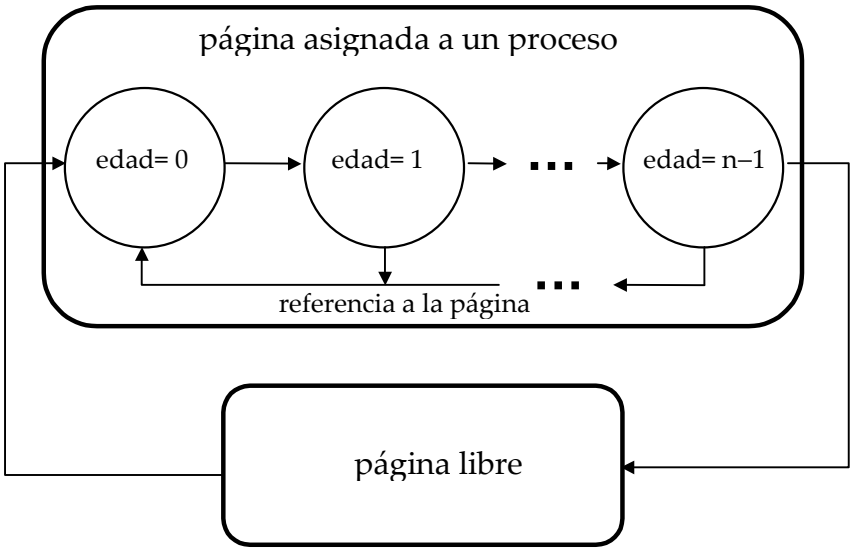

Figura 4.8. Transición de estados y envejecimiento de una página en UNIX.

4.7.3 Windows

Windows NT, como sistema operativo moderno, presupone un potente soporte hardware para la gestión de memoria, como es el caso de los procesadores actuales. Los procesos manejan espacios de direcciones de 32 bits. El tamaño de página es de 4 Kbytes, aunque en algunas máquinas se puede configurar. Utiliza memoria virtual con paginación a dos niveles y proporciona los mecanismos de protección y compartición habituales.

En lo referente a la política de gestión de la memoria virtual, básicamente copia la filosofía de VMS. Las capacidades de la gestión de memoria las proporciona un módulo del núcleo denominado *VM manager*. Cabe destacar las siguientes características.

Paginación por demanda con clustering. A partir de un fallo de página no sólo se carga la página que ha producido el fallo sino también un conjunto de páginas adyacentes.

Asignación de páginas local. Aplica una política FIFO entre las páginas cargadas del proceso (su *working-set*).

Tamaño de working-set ajustable dinámicamente. El VM manager ajusta el tamaño de los working-sets de los procesos en función de sus comportamientos, sus cuotas de memoria y de las necesidades globales de memoria. Cuando necesita memoria, el VM manager recupera marcos de los working-sets de los procesos, respetándoles un tamaño mínimo. Un proceso puede aumentar su working-set a medida que comete fallos de página si hay marcos libres.

4.8 Bibliografía

Los textos clásicos de sistemas operativos vuelven a ser útiles en este tema. Al ser la gestión de memoria un tema muy asentado académicamente, todos ellos siguen un enfoque similar y son prácticamente equivalentes como texto de referencia. El orden en que se relacionan a continuación es meramente alfabético: [SIL08], [STA05], [TAN08].

[BAH86] (capítulo 9) describe la implementación de la gestión de memoria en UNIX System V, mientras que [VAH96] (capítulos 13, 14 y 15) lo hace para los SVR4 y BSD (éste último no tratado en estos apuntes). [CUS93, SOL98, SOL00] describen la gestión en Windows.

4.9 Ejercicios

1 En un sistema con memoria física de 64 Mbytes gestionada por segmentación, comparar las alternativas de gestión del espacio libre (mapa de bits y lista de huecos libres) en cuanto a eficiencia espacial. ¿Qué cambiaría si el sistema fuera segmentado-paginado, con páginas de 4 Kbytes?

2 Un sistema de memoria paginada tiene un espacio de direccionamiento lógico de 1 Gbyte y páginas de 4 Kbytes. Para una memoria física de 64 Mbytes, calcular el tamaño de la tabla de páginas, teniendo en cuenta que la memoria es direccionable a nivel de byte.

3 El siguiente código forma parte de un sistema de gestión de memoria paginado como el del Ejercicio 2. Se define una función `ubicar()`, que será llamada desde *ejecutar_programa*, que reserva espacio en memoria para el proceso creado:

```
int tablas_pag[NUM_PROCS][MAX_ENT_TP];
int marcos_libres;

int ubicar(int num_paginas, int num_proc)
{
      int i, marco;

      entrar_SC(MUTEX_MEM);
      if (num_paginas > marcos_libres) { /*no hay espacio*/
            dejar_SC(MUTEX_MEM);
            return(-1);
      }
      marcos_libres= marcos_libres - num_paginas;
      dejar_SC(MUTEX_MEM);
      for (i=0; i<num_paginas; i++) {
            entrar_SC(MUTEX_MEM);
            marco= buscar_en_mapa_bits();
            poner_a_1_mapa_bits(marco);
            dejar_SC(MUTEX_MEM);
            tablas_pag[num_proc][i]= marco;
```

```
        }
        for (i=num_paginas; i<MAX_ENT_TP; i++)
              tablas_pag[num_proc][i]= -1; /*páginas no usadas*/
        return(0);
}
```

(a) Definir MAX_ENT_TP. Si el sistema dedica 4 Mbytes de memoria para ubicar las tablas de páginas, ¿cuál debe ser el valor de NUM_PROCS?

(b) Programar la rutina **liberar**(num_proc), que libera los bloques ocupados por el proceso num_proc. Utilizar funciones para acceso al mapa de bits.

4 Se pretende segmentar el sistema de memoria del Ejercicio 2 para que las tablas de páginas no ocupen más de un marco de página.

(a) Dibujar el esquema de la traducción de direcciones, mostrando la estructura de las direcciones virtual y física.

(b) ¿Cuántas entradas tendrá la tabla de segmentos? Calcular el tamaño mínimo de cada entrada, teniendo en cuenta que las tablas de páginas pueden estar ubicadas en cualquier dirección.

(c) Tenemos cargados 100 procesos que ocupan toda la memoria, y cada proceso tiene como media 3 segmentos. Calcular (c1) la fragmentación interna media por proceso, (c2) la fragmentación externa total en el sistema, y (c3) la pérdida de eficiencia espacial en el sistema por fragmentación interna y externa.

(d) Si la gestión de marcos libres se hace por mapa de bits, ¿cuántos bytes ocupará el mapa?

(e) ¿Qué parámetros se modificarían si se ampliase la memoria a 128 Mbytes?

5 El sistema del ejercicio anterior se gestiona con memoria virtual. A partir de las siguientes definiciones,

```
struct ent_seg {
            int longitud;
            int *ptabla_pag;
} tablas_seg[NUM_PROCS][NUM_SEG];
```

(a) Introducir las definiciones necesarias y/o modificar las existentes.

(b) Programar una función **`ubicar`**`(num_proc)` con prepaginación, que rellena las tablas de páginas a partir de las entradas de longitud de la tabla de segmentos correspondiente, que se supone que ya ha sido cargada. Nótese que `ubicar()` no carga las páginas en memoria.

(c) Programar una nueva versión de **ubicar**(num_proc) que establece la carga de páginas por demanda.

6 Se quiere introducir memoria virtual en un sistema paginado donde el tiempo de acceso a memoria paginada es de 0,1 μs. El tiempo medio para tratar un fallo de página en el nuevo sistema con memoria virtual se estima que será de 5 ms. Calcular la probabilidad máxima de fallo de página para que el tiempo medio de acceso a memoria en el sistema de memoria virtual no se incremente en más de 20% con respecto al del sistema paginado original.

7 En un sistema paginado de memoria virtual con 3 marcos de memoria física, considerar la siguiente secuencia de referencias a páginas:

1 2 3 4 2 3 2 3 1 5 1 2 6 2 5 1 5 2 5 5 1 6 3 5 6

Indicar la sucesión de fallos de página y las páginas víctimas elegidas según se siga un algoritmo de reemplazo óptimo, FIFO, LRU o Segunda Oportunidad.

8 A partir de las siguientes definiciones:

```
struct ent_tab_pag {
      char bit_V;  /* bit de validez */
      int marco;
} tab_pag[NUM_PROC][NUM_PAG];

struct ent_marco {
      char bits;  /* MASK_R:referencia, MASK_M:modificado */
      int tick_carga;
      int proceso;
} marco[NUM_MARCOS];
```

escribir una rutina int **pag_victima**(p) que devuelva el número de marco de la página elegida como víctima para el proceso p (asignación local) siguiendo las políticas (a) de la segunda oportunidad, (b) FIFO. Modificar (con ambos algoritmos) la rutina pag_victima() para que la asignación de páginas sea global.

9 Según se ha observado en un sistema como el del Ejercicio 5 (basado en el del Ejercicio 2), aparece thrashing cuando el grado de multiprogramación supera los 100 procesos. Calcular el tamaño mínimo del working set para evitar esta situación.