# La memoria virtual

Enric Morancho Llena
Dolors Royo Vallés

PID_00215288

Universitat Oberta de Catalunya

# Índice

# Introducción

En este módulo didáctico se estudia con detalle la gestión de la memoria virtual o, más concretamente, la gestión de la memoria virtual paginada. Los contenidos del módulo se pueden dividir en tres partes bien diferenciadas:

**1)** Primero, se repasan algunos conceptos importantes que son necesarios para poder seguir los contenidos del resto del módulo.

**2)** Después, se describe el soporte hardware necesario para la implementación de la memoria virtual paginada.

**3)** Finalmente, se describen algunas de las políticas de asignación, reemplazo y carga propuestas al explicar la implementación de la memoria virtual.

Además, en un anexo del módulo se describen algunas características de la gestión de memoria en el núcleo de Linux.

# Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para alcanzar los objetivos siguientes:

1. Ampliar el concepto de memoria virtual que habéis aprendido en otras asignaturas mediante la descripción de las necesidades de hardware y de software que tiene el sistema para implementar de manera eficiente la gestión de la memoria virtual paginada.

2. Conocer las diferentes políticas de gestión de la carga, de asignación y de reemplazo de páginas y analizar sus ventajas e inconvenientes.

3. Ver de qué manera la gestión de la memoria afecta a la ejecución de código de programas.

4. Conocer los aspectos más importantes de la asignación de memoria en el núcleo de Linux.

# 1. Concepto de memoria virtual

El objetivo principal de la **memoria virtual** es ofrecer a los procesos una visión idealizada de la memoria.

**Ved también**

En el módulo "Gestión de memoria" de la asignatura *Sistemas operativos* se presentó el concepto de *memoria virtual*.

Entre otras cosas, esta idealización permite que cada proceso disponga de, como mínimo, un espacio de direcciones (espacio lógico) para él solo, independiente de los del resto de los procesos en ejecución; además, puede permitir a los procesos disponer de un espacio lógico de tamaño superior al de la memoria física instalada. Por lo tanto, la memoria virtual esconde a los procesos varias problemáticas relacionadas con la memoria, como la compartición de la memoria física entre varios procesos, la fragmentación externa, el hecho de tener toda la memoria física ocupada y tener que gestionar una jerarquía de memoria, etc. A pesar del coste hardware y software de las implementaciones de memoria virtual, las ventajas de disponer de ella son considerables.

**Ventajas de disponer de memoria virtual**

Algunas de las ventajas son, por ejemplo, simplificar la tarea del programador, reducir el tiempo de carga de los programas (para empezar a ejecutar un programa, no es necesario que esté cargado totalmente en la memoria física), permitir el aumento del grado de multiprogramación (la suma de los tamaños de los espacios lógicos de todos los procesos puede superar la capacidad de la memoria física con creces), permitir compartir datos y código entre procesos, etc.

En la asignatura *Sistemas operativos* se vio una posible implementación de memoria virtual basada en paginación. Esta implementación utiliza un hardware especializado denominado *memory management unit* (MMU) que traduce dinámicamente, es decir, en tiempo de ejecución, las direcciones lógicas generadas por los procesos en direcciones físicas. Además, dispone de un espacio en disco (área de intercambio[1]) para poder almacenar toda aquella información que, en un momento dado, no puede ser almacenada en la memoria física.

(1)En inglés, *swap area*.

Las implementaciones de memoria virtual tienen un componente hardware y otro software. Esto se debe a dos factores:

**Nota**

Aunque existen varias implementaciones posibles de memoria virtual, en este módulo también asumiremos, sin pérdida de generalidad, una implementación basada en paginación.

**a)** Los sistemas computadores están accediendo continuamente a memoria. Pensad que el paso previo a ejecutar cualquier instrucción de lenguaje máquina es leerla de memoria. Además, una cantidad muy significativa de instrucciones de lenguaje máquina (típicamente, el 40% de las instrucciones ejecutadas) también requiere leer o escribir de memoria (son instrucciones de tipo *load* o *store*). Por lo tanto, para no penalizar el rendimiento de la máquina, es aconsejable que, en el caso general, la implementación de memoria virtual se realice íntegramente por hardware.

**b)** Algunas de las tareas que debe llevar a cabo una implementación de memoria virtual pueden ser relativamente complejas y pueden variar de una versión a otra del sistema operativo. Por lo tanto, estas tareas suelen ser implementadas por código propio del sistema operativo, es decir, están implementadas por software.

En este módulo haremos un breve repaso al componente hardware de las implementaciones de memoria virtual y veremos con detalle el componente software de la implementación de memoria virtual.

**Implementación de memoria virtual**

Algunos ejemplos de las tareas que debe llevar a cabo una implementación de memoria virtual son decidir qué parte del espacio lógico de un proceso se encuentra en la memoria física y qué parte se encuentra en el área de intercambio, decidir cuántas páginas puede tener cargadas el proceso en memoria, llevar páginas desde el área de intercambio a memoria física (*swap-in*), y al revés, desde la memoria física al área de intercambio (*swap-out*), etc.

# 2. Soporte hardware

Este apartado hace un breve repaso del soporte hardware necesario para implementar memoria virtual basada en paginación. Además, se harán algunas consideraciones que se deben tener en cuenta en una implementación de memoria virtual.

## 2.1. Funcionalidades mínimas que debe ofrecer una MMU

Las **funcionalidades mínimas** que debe ofrecer la MMU[2] que implemente memoria virtual son un mecanismo de traducción y la capacidad de generar excepciones en determinados escenarios.

(2)MMU es la sigla de *memory management unit*.

**1) Mecanismo de traducción**. Traduce cada dirección lógica generada por un proceso en la dirección física equivalente. Un sistema de gestión de memoria basado en paginación divide el espacio lógico y el espacio físico en porciones de tamaño fijo llamadas páginas. La MMU descompone las direcciones lógicas en dos componentes (identificador de página y desplazamiento dentro de la página). Para hacer la traducción, consulta una tabla de páginas donde cada entrada contiene el identificador de *frame* (página física) donde se almacena la página lógica, así como una serie de bits de control –como mínimo, validez y presencia[3], pero puede tener otros como "solo lectura[4]", ejecución, modificado[5], accedido, etc.

(3)**Bit de validez**: indica si la página pertenece al espacio lógico del proceso.
**Bit de presencia**: indica si una página válida está cargada en memoria física.

(4)En inglés, *read only*.

(5)En inglés, *dirty bit*.

**Tabla de páginas**

La tabla de páginas es inicializada por el sistema operativo al crear el proceso. A medida que el proceso se ejecuta, el sistema operativo la actualiza.

**2) Generación de excepciones**. En el caso de que el mecanismo de traducción no pueda realizar la traducción, la MMU generará una excepción. La MMU puede generar varios tipos de excepciones, pero las más habituales son: dirección lógica inválida (generada cuando el proceso intenta acceder a una página inválida) y fallo de página[6] (generado cuando el proceso intenta acceder a una página válida pero que no está presente en memoria física porque se encuentra en el área de intercambio).

(6)En inglés, *page fault*.

Las rutinas de tratamiento para estas excepciones son rutinas propias del sistema operativo. Cuando se produzca una excepción, el sistema operativo tomará el control de la situación e intentará poner remedio al problema: si no es posible solucionarlo, el sistema operativo abortará el proceso; si es posible solucionarlo, el sistema operativo hará retomar la ejecución del proceso desde el acceso a memoria que ha provocado la excepción.

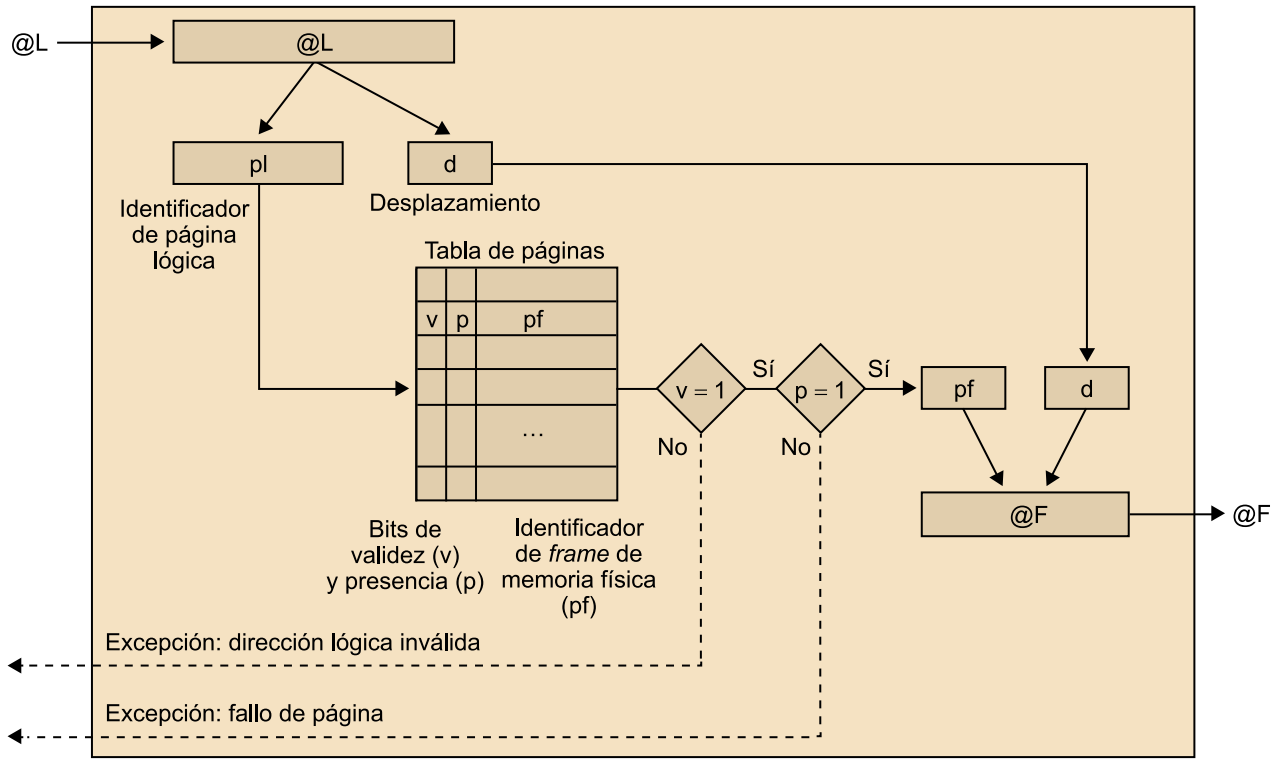
**Ejemplos de generación de excepciones**

Si un proceso intenta acceder a una página válida pero que no está presente en memoria física, la rutina de atención a la excepción de fallo de página leerá en el área de intercambio la página solicitada y la copiará en memoria física (posiblemente, también habrá que copiar la página de memoria física que se está sobrescribiendo en el área de intercambio), actualizará las estructuras de datos del sistema operativo relativas a memoria virtual y hará que el proceso continúe la ejecución desde el acceso a memoria que ha provocado la excepción.

Otro escenario que puede provocar una excepción es el desbordamiento de la pila de ejecución, es decir, cuando un proceso intenta apilar algún dato en esta pila (por ejemplo, una variable local, un parámetro de una rutina, la dirección de retorno a una función, etc.) pero resulta que esta ya está llena, pues al crear el proceso, el sistema operativo asigna un determinado número de páginas lógicas a la pila. Si este número resulta insuficiente, se acabará produciendo un desbordamiento de la pila. Este escenario provoca que el proceso intente acceder a una página marcada como inválida. La rutina del sistema operativo que atiende a esta excepción intentará aumentar el tamaño de la pila de ejecución y hará que el proceso vuelva a ejecutarse desde la instrucción que ha provocado el desbordamiento de la pila. En el caso de que el tamaño de la pila de ejecución ya haya llegado al tamaño máximo que permite el sistema operativo, la rutina de tratamiento en la excepción hará abortar el proceso.

La figura 1 muestra un diagrama con la implementación de una MMU basada en paginación. La MMU recibe una dirección lógica (@L), la separa en dos componentes (identificador de página lógica y desplazamiento) y, en función del contenido de la tabla de páginas, genera la traducción a dirección física (@F) o genera una excepción (dirección lógica inválida o fallo de página).

Figura 1. Implementación de una MMU basada en paginación que soporte memoria virtual



## 2.2. Consideraciones de eficiencia

Las implementaciones de memoria virtual topan con algunos problemas de eficiencia y de uso de recursos. El hardware da un cierto soporte para minimizar el impacto de estos problemas.

**1) Estructura de la tabla de páginas**

En las implementaciones más sencillas de paginación, la tabla de páginas debe tener tantas entradas como el número de páginas que puede llegar a tener el espacio lógico del proceso, independientemente de la validez de las páginas. Si se asume un espacio lógico de $2^{32}$ posiciones de memoria y páginas de 4 KB, esto implica que la tabla de páginas tenga $2^{20}$ entradas; si cada entrada de la tabla de páginas ocupa 4 octetos, la tabla de páginas ocupará 4 MB. Estos tamaños coinciden con los utilizados por la arquitectura IA-32.

Por lo tanto, en general, la tabla de páginas no puede almacenarse en registros, sino que hay que almacenarla en memoria. La manera más sencilla de almacenamiento sería la contigua, pero esto supone desperdiciar memoria porque los procesos no suelen utilizar todas las entradas de la tabla de páginas.

Para solucionar estos problemas, algunas arquitecturas permiten organizar las tablas de páginas en dos niveles y/o soportar varios tamaños de páginas simultáneamente (por ejemplo, la arquitectura IA-32 soporta páginas de 4 KB, 2 MB y 4 MB). La organización en dos niveles permite almacenar la tabla de páginas de manera no contigua y reducir el espacio ocupado por la tabla si muchas páginas no son válidas.

**2) TLB *(translation lookaside buffer)*.**

Conceptualmente, la traducción de direcciones basada en paginación requiere realizar dos accesos a la memoria física por cada referencia a la memoria lógica: el primero para acceder a la entrada correspondiente de la tabla de páginas y el segundo para acceder realmente al dato. Como este sobrecoste no es tolerable, los sistemas de traducción basados en paginación suelen disponer de una memoria caché que contiene las últimas entradas utilizadas de la tabla de páginas. Esta memoria recibe el nombre de TLB[7]. Antes de acceder a la tabla de páginas, la MMU accede al TLB: si la traducción se encuentra en el TLB, no habrá que acceder a la tabla de páginas, de lo contrario, se produce un fallo de TLB[8] y habrá que acceder a la tabla de páginas (como si no hubiera TLB). Los TLB son muy eficaces y ahorran una parte muy importante de los accesos a la tabla de páginas porque los accesos a memoria de los procesos exhiben localidad temporal y espacial, por lo que es habitual que un proceso acceda repetidas veces a una misma página.

(7)TLB es la sigla de *translation lookaside buffer*.

(8)En inglés, *TLB miss*.

**Ved también**

La localidad temporal y espacial se tratan en el anexo 1. "Localidad temporal y localidad espacial de un proceso".

La gestión de los *TLB misses* depende de la arquitectura. Por ejemplo, en el caso de IA-32, los *TLB misses* son gestionados directamente por el hardware; es responsabilidad del hardware buscar la traducción en la tabla de páginas[9] y actualizar el contenido del TLB. En cambio, otras arquitecturas generan una excepción y es el sistema operativo el que se encarga de acceder a la tabla de páginas y de actualizar el contenido del TLB.

[(9)]En inglés, *page walk*.

En caso de cambio de contexto, en algunas arquitecturas hay que invalidar el contenido del TLB porque las traducciones almacenadas en el TLB son válidas únicamente para el proceso que estaba en ejecución. Otras arquitecturas permiten identificar las traducciones correspondientes a accesos propios del sistema operativo; estas traducciones no se invalidan en caso de cambio de contexto. Finalmente, otras arquitecturas permiten asociar una especie de identificador de proceso a las entradas del TLB; en este caso, no es necesario invalidar el contenido del TLB cuando se produce un cambio de contexto.

**Procesador Intel Core i7**

Como referencia, el procesador Intel Core i7 dispone de un TLB organizado en dos niveles. El primer nivel contiene 64 traducciones y el segundo nivel, 512. Tanto el primer como el segundo nivel son memorias caché con asociatividad 4. Cuando hay que realizar una traducción, se consulta el primer nivel de TLB; si no se encuentra la traducción, se consulta el segundo nivel de TLB; si tampoco se encuentra (*TLB miss*), hay que acceder a la tabla de páginas almacenada en memoria.

# 3. Políticas de gestión de la memoria virtual

Cuando un proceso hace una referencia a una dirección dentro de su espacio lógico, se calcula a qué página lógica corresponde y, a continuación, la MMU comprueba si esta página está cargada en la memoria. En caso de que no esté, se produce una excepción de tipo fallo de página. La rutina de atención a esta excepción es la encargada de llevar la página a la memoria principal desde el área de intercambio. Esta tarea no es sencilla, ya que si no hay páginas libres en la memoria principal, se debe tomar una decisión: o bien se bloquea el proceso y se espera a que algún otro proceso finalice su ejecución y deje páginas de la memoria libres, o bien se elige una página de la memoria principal y se la lleva al área de intercambio para hacer sitio en la página que ha generado el fallo de página.

Un gestor de la memoria virtual paginada debe tener:

- Una **política de carga** y **descarga** de páginas para decidir en qué momento debe llevar una página a la memoria y cuándo la debe sacar.

- Una **política de emplazamiento** para saber dónde debe poner una página nueva.

- Una **política de reemplazo** para elegir, cuando el sistema no tiene ninguna página libre y necesita cargar una nueva página de algún proceso que se está ejecutando, el algoritmo que conviene aplicar para seleccionar una página de las que hay cargadas en la memoria principal y devolverla al área de intercambio.

- Una **política de asignación** que, conociendo el grado de multiprogramación, indique cuántas páginas de un mismo proceso pueden estar en la memoria de manera simultánea.

En un sistema de gestión de memoria basado en paginación, la gran mayoría de las páginas libres de la memoria física son igualmente buenas para ser elegidas a la hora de cargar una nueva página en la memoria. Por lo tanto, la política de emplazamiento no afectará a la eficiencia del gestor de la memoria virtual paginada.

**Dispositivos de entrada/salida**

Una única excepción a esta política de gestión de memoria basada en paginación serían las páginas utilizadas por dispositivos de entrada/salida que trabajan directamente con direcciones físicas de memoria. No obstante, estas páginas suelen estar asignadas al sistema operativo, con lo que nunca estarán disponibles para los procesos.

En un sistema con paginación pura (bajo demanda), solo se carga una página en la memoria principal cuando se produce un fallo de página con una dirección que hace referencia a la página en cuestión. Otra alternativa es que, aprovechando la localidad espacial de los accesos a memoria, además de cargar la

página que ha provocado el fallo también se carguen un cierto número adicional de páginas lógicamente contiguas a la que ha provocado el fallo. Esta alternativa recibe el nombre de prebúsqueda[10].

(10)En inglés, *prefetching*.

**Ved también**

Recordad que la localidad espacial se trata en el anexo 1, "Localidad temporal y localidad espacial de un proceso".

Una política trivial de descarga de páginas consiste en descargar una página (y actualizar el área de intercambio si hace falta) cuando sea necesario cargar una nueva página y no sea posible cargarla en ningún *frame* libre. Ahora bien, la mayoría de los sistemas operativos intentan disponer siempre de algunos *frames* libres antes de que sean necesarios; por ejemplo, algunas rutinas de tratamiento de interrupción pueden tener que pedir memoria y no es posible bloquear la ejecución de estas rutinas. Para garantizar que siempre haya *frames* disponibles, un proceso del sistema operativo se encarga de analizar periódicamente la utilización de *frames* de memoria física e intercambia en disco los que considera oportunos. La descarga de páginas también se realiza cuando el sistema es hibernado.

A continuación, se describen algunas políticas de reemplazo y de asignación de páginas que han sido propuestas por diferentes autores.

### 3.1. Políticas de reemplazo

Se han propuesto varias políticas de reemplazo y, muy probablemente, cada sistema operativo implementa una política diferente. En cualquier caso, una **política de reemplazo** debe tener como objetivo que el porcentaje de fallos de página (tasa de fallos de página) con respecto al total de referencias efectuadas en la memoria física sea lo más pequeño posible.

**Fallos de página**

Cada vez que se produce un fallo de página se genera una excepción que debe ser tratada por el sistema operativo, con el consiguiente cambio de modo de ejecución y transferencia de control a una rutina propia del sistema operativo. Como estas acciones tienen un coste computacional apreciable, una tasa excesivamente elevada de fallos de página puede impactar de manera apreciable en el rendimiento de la máquina.

En algunos sistemas operativos (como Linux), la política de reemplazo elige únicamente páginas asignadas a procesos; en ningún caso elegirá una página asignada al núcleo del sistema operativo. Esto se debe a que algunas porciones de código/datos/pila del sistema operativo en ningún caso pueden ser intercambiadas en el área de intercambio (por ejemplo, las páginas que contienen el código de atención a la excepción del fallo de página). Además, por una cuestión de eficiencia, es recomendable que todo el núcleo del sistema operativo esté presente en memoria física. En este módulo asumiremos que únicamente podrán ser reemplazadas páginas asignadas a procesos.

Los diferentes algoritmos propuestos en este apartado se evalúan ejecutándolos para una determinada cadena de referencias a las páginas de la memoria. Supongamos (si no se indica lo contrario) que tenemos la cadena de referencias siguiente: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0. Esto significa que el proceso debe acceder a determinadas direcciones de la memoria que están en las páginas lógicas indicadas en la secuencia.

A la hora de estudiar las políticas de reemplazo, debemos tener en cuenta el número de páginas de la memoria física que se pueden asignar a un proceso. En función de este número, podemos distinguir las dos políticas siguientes:

**1) Política de reemplazo global**. Si las políticas son globales, se considera que todas las páginas de la memoria física son candidatas a ser reemplazadas. Un reemplazo puede sacar de la memoria una página asignada a un proceso diferente del proceso que ha generado el fallo de página. Con las políticas de reemplazo global, un proceso puede tener un número variable de páginas cargadas en la memoria: desde cero hasta todas las páginas a las que haya hecho referencia hasta un cierto instante. Entonces, el número de fallos de página de un proceso depende de la carga del sistema y, muy probablemente, el tiempo de ejecución del proceso será variable entre ejecuciones diferentes.

**2) Políticas de reemplazo local**. Si las políticas son locales, tan solo un subconjunto de páginas de la memoria física son candidatas a ser reemplazadas. La página candidata se elige de entre el conjunto de páginas de la memoria física asignadas a un proceso. De esta manera, la ejecución de un proceso no influye en la ejecución del resto de los procesos. También se consigue un cierto control del grado de multiprogramación que debe soportar el sistema. El número de fallos de página de un proceso será muy similar cada vez que se ejecute el proceso, y el tiempo de ejecución no variará mucho entre diferentes ejecuciones del mismo proceso.

Para los ejemplos propuestos en los subapartados siguientes, supondremos que los procesos disponen de un número determinado de páginas de la memoria física (tres). Inicialmente, al empezar a ejecutar el proceso, todas están libres. En todos los ejemplos, si no se dice lo contrario, supondremos un reemplazo local.

### 3.1.1. El algoritmo de reemplazo óptimo

> Un algoritmo de reemplazo óptimo es aquel que tiene la tasa de fallo de páginas más pequeño de todos los algoritmos. Este algoritmo de reemplazo se podría describir como el algoritmo que reemplaza la página que no se utilizará durante un período de tiempo más largo.

Este algoritmo no se puede implementar, ya que para hacerlo es necesario saber en un instante determinado qué comportamiento tendrá el proceso en el futuro (qué referencias a la memoria se harán). El resultado de la aplicación del algoritmo óptimo se utiliza principalmente en estudios comparativos.

Si aplicamos el algoritmo óptimo a la cadena de referencias de nuestro ejemplo, se generan nueve fallos de página (tasa de fallos de página del 47,4%), tal como se muestra en la figura 2:

Figura 2. Algoritmo de reemplazo óptimo

| Número de referencia | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Página referenciada | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| Páginas en la memoria del proceso | 7 | 0 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 7 | 7 |
| | – | 7 | 0 | 1 | 1 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 2 |
| | – | – | 7 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| Fallo de página | F | F | F | F | – | F | – | F | – | – | F | – | – | F | – | – | – | F | – |

Las tres primeras referencias provocan tres fallos de página. La referencia a la página 2 hace que salga de la memoria la página 7, ya que no se utilizará hasta la referencia 18, mientras que la página 0 se utilizará en la referencia 5, y la página 1, en la referencia 14. La referencia 3 hace salir la página 1, y así hasta que se complete la secuencia de referencias.

### 3.1.2. El algoritmo de reemplazo FIFO

El **algoritmo de reemplazo FIFO** (*first in, first out*) necesita saber el orden en el que se cargan las páginas en la memoria principal. Cuando conviene hacer un reemplazo, la página candidata a ser reemplazada es la que hace más tiempo que se cargó en la memoria física, es decir, la más antigua.

Si aplicamos este algoritmo a la secuencia de referencias de nuestro ejemplo, se produce un total de 14 fallos de página (tasa de fallos de página del 73,7%). La figura 3 muestra un esquema del funcionamiento del algoritmo:

Figura 3. Algoritmo de reemplazo FIFO

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Número de referencia | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| | Página referenciada | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| Prioridad de reemplazo ↓ | Páginas en la memoria del proceso | 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 3 | 0 | 0 | 0 | 1 | 2 | 2 | 2 | 7 | 0 |
| | | – | 7 | 0 | 1 | 1 | 2 | 3 | 0 | 4 | 2 | 3 | 3 | 3 | 0 | 1 | 1 | 1 | 2 | 7 |
| | | – | – | 7 | 0 | 0 | 1 | 2 | 3 | 0 | 4 | 2 | 2 | 2 | 3 | 0 | 0 | 0 | 1 | 2 |
| | Fallo de página | F | F | F | F | – | F | F | F | F | F | F | – | – | F | F | – | – | F | F |

De nuevo, las tres primeras referencias producen 3 fallos de página. Cuando se referencia la página 2, se hace salir a la página 7, ya que es la primera página que se llevó a la memoria. La referencia siguiente, a la página 0, no produce fallo de página porque ya la tenemos en la memoria. La referencia siguiente, a la página 3, hace salir a la página 0 porque es la segunda página que se llevó a la memoria, y la primera ya se había sacado. Este procedimiento se sigue en el resto de las referencias.

El algoritmo FIFO[11] es fácil de implementar (solo se necesita una memoria intermedia[12] circular y un puntero), pero tiene algunas carencias, como las siguientes:

[11] FIFO es la sigla de *first in, first out*.

[12] En inglés, *buffer*.

- A la hora de elegir una página candidata no se tiene en cuenta si una página ha sido muy referenciada o no. Esto puede provocar que se saque de la memoria una página que se utiliza muchísimo y, por lo tanto, indirectamente aumentemos las posibilidades de que se produzcan más fallos de página.

- Lo más lógico es que si aumentamos el número de páginas que puede tener cargadas en la memoria un proceso, el número de fallos de página disminuya, pero está demostrado que el algoritmo de reemplazo FIFO no siempre cumple este principio.

**La anomalía de Belady**

Veamos un ejemplo concreto para ilustrar el hecho de que un aumento del número de páginas cargadas en la memoria no implica una disminución del número de fallos de página.

Supongamos que tenemos en la memoria la secuencia de referencias 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 e imaginemos que utilizamos el algoritmo de reemplazo FIFO. Podéis comprobar que si permitimos tener solamente una página en la memoria, se producen 12 fallos de página; con 2 páginas cargadas en la memoria constatamos 12 fallos de página; con 3 páginas en la memoria se producen 9 fallos de página, y con 4 páginas en la memoria tenemos 10 fallos de página. Este problema recibe el nombre de **anomalía de Belady**.

### 3.1.3. El algoritmo de reemplazo LRU

El algoritmo de reemplazo LRU (*least recently used*) consiste en registrar el momento en el que cada página ha sido referenciada por última vez. En caso de que alguna página deba salir de la memoria física, se elegirá aquella que hace más tiempo que no ha sido referenciada.

En la figura 4 mostramos el resultado de aplicar el algoritmo de reemplazo LRU[13] a la secuencia de referencias en la memoria que hemos propuesto como ejemplo:

[13] LRU es la sigla de *least recently used*.

Figura 4. Algoritmo de reemplazo LRU

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Número de referencia | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Página referenciada | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| Páginas en la memoria del proceso (Prioridad de reemplazo ↓) | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 |
| | – | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 |
| | – | – | 7 | 0 | 1 | 2 | 2 | 3 | 0 | 4 | 2 | 2 | 0 | 3 | 3 | 1 | 2 | 0 | 1 |
| Fallo de página | F | F | F | F | – | F | – | F | F | F | F | – | – | F | – | F | – | F | – |

Aquí se producen 12 fallos de página en total (tasa de fallos de página del 63,2%). Las tres primeras referencias provocan tres fallos de página. La referencia a la página 2 hace salir de la memoria física a la página 7, que es la que hace más tiempo que no ha sido referenciada. La referencia a la página 0 no provoca ningún fallo de página. La referencia a la página 4 hace salir a la página 1, que es aquella a la que hace más tiempo que se ha hecho referencia, y así para todas las referencias de la secuencia.

En general, el algoritmo LRU tiene mejor rendimiento que el FIFO, pero el coste de implementación del LRU es mucho mayor que el del FIFO.

Se pueden conseguir algunas de las implementaciones posibles del algoritmo LRU de las maneras siguientes:

**a)** Añadiendo contadores a cada entrada de la tabla de páginas que indican cuánto tiempo hace que se ha efectuado la última referencia a la página. Por ello, hay que tener un reloj lógico que lo señale. En cada referencia a la memoria se actualiza el contador de la entrada de la tabla de páginas referenciada y se saca de la memoria la página que tiene el contador más pequeño. Esto requiere hacer una búsqueda entre todas las entradas de la tabla de páginas

para encontrar la página que tiene el contador más bajo. Tenemos el problema de que el contador se puede saturar, es decir, se puede producir lo que se denomina un desbordamiento.

**b)** Utilizando una pila. La idea es mantener una estructura de tipo pila con el número de páginas que se referencian. Siempre que se hace una referencia a una página, esta pasa a la cima de la pila, de manera que las páginas que hace menos tiempo que han sido referenciadas se encuentran cerca de la cima de la pila, y las páginas que hace más tiempo que han sido referenciadas están cerca de la base de la pila. Para implementar este esquema, se necesita una lista doblemente encadenada. Efectuar una actualización tiene un coste mayor que utilizar los contadores porque se deben actualizar muchos punteros, pero así evitamos hacer una búsqueda de todas las páginas.

Ambas opciones requieren soporte de hardware para ser implementadas y hacer una implementación por software puede ser muy costoso en tiempo de ejecución a la hora de gestionar un fallo de página.

**Aproximaciones al algoritmo LRU**

Se han propuesto aproximaciones al algoritmo LRU que utilizan un bit de referencia. Se asocia a cada entrada de la tabla de páginas un bit de referencia que se actualiza (se pone a 1) cada vez que se referencia la página. A continuación, presentamos tres de estas aproximaciones:

**1) Guardar la historia de las referencias**. En este caso, además del bit de referencia, para cada página se añade un registro de *k* bits que guarda la historia de las referencias a la página (de los últimos *k* intervalos de tiempo) e indica si se ha hecho referencia a la página al menos una vez cada intervalo. Este registro histórico se actualiza a intervalos regulares, con lo que se descarta la posición menos significativa y se incorpora a la posición más significativa el bit de referencia, que se pone a cero cada vez que se actualiza el registro de historia de la página (los *k* bits).

La página que tiene el registro de historia con el valor más pequeño es la candidata a salir de la memoria. Si hay más de una página con el mismo valor del registro histórico, se puede aplicar cualquier algoritmo de selección (por ejemplo, el FIFO) que indique cuál es la página candidata a salir de la memoria, o se pueden sacar todas las posibles candidatas.

**2) Dar una segunda oportunidad**. En este caso, podríamos considerar que no tenemos constancia de las referencias a las páginas y que solo tenemos en cuenta el bit de referencia. El algoritmo de la segunda oportunidad se basa en un algoritmo FIFO. Cuando se selecciona la página que se debe sacar de la memoria miramos su bit de referencia. Si este bit es igual a 0, se reemplaza la página. En cambio, si el bit es igual a 1, la página no se reemplaza (se le da una segunda oportunidad); se vuelve a insertar en la cola (como si fuera una

página nueva) y la candidata a ser reemplazada pasa a ser la página siguiente de la cola FIFO. Cuando se da la segunda oportunidad a una página, su bit de referencia se pone a 0.

**3) Algoritmo del reloj**. Es una sofisticación del algoritmo de segunda oportunidad. Utiliza una memoria intermedia circular de páginas y mantiene un puntero en la última posición examinada. La diferencia respecto al algoritmo de segunda oportunidad es que no mueve dentro de la estructura las páginas a las que se da una segunda oportunidad. Si decide que hay que dar una segunda oportunidad a una página, lo único que hay que hacer es mover el puntero a la siguiente posición de la memoria intermedia circular.

**Algoritmo del reloj**

Linux utiliza una variante de este algoritmo como algoritmo de reemplazo.

### 3.1.4. Los algoritmos de reemplazo LFU y MFU

El algoritmo LFU (*least frequently used*) reemplaza la página que ha sido menos referenciada. Su implementación es muy sencilla: consiste en poner un contador en cada página, y cada vez que se hace referencia a una página, su contador se incrementa. Cuando hay que sacar una página, se saca la que tiene el contador más pequeño.

El problema que tenemos es la saturación del contador; y que periódicamente hay que poner a 0 los valores de todos los contadores.

Si, en cambio, sacamos de la memoria la página que tiene el contador más grande, ya que consideramos que la página con el contador más pequeño probablemente ha sido cargada hace menos tiempo y se utilizará mucho, entonces implementamos el algoritmo MFU (*most frequently used*).

### 3.1.5. Consideración final: páginas modificadas

Al sacar una página de la memoria es importante tener en cuenta que si ha sido modificada, se deberá actualizar en el área de intercambio (habrá que hacer una transferencia de la memoria al área de intercambio). Una vez que el área de intercambio ha sido actualizada, ya se puede cargar la página nueva (haciendo una transferencia del área de intercambio a la memoria).

Si una página no ha sido modificada, el coste de sacarla de la memoria se reduce a actualizar la página de la memoria con la nueva página lógica (haciendo una transferencia del área de intercambio a la memoria). Podemos considerar que esta operación, respecto al caso anterior, reduce el coste significativamente.

Por lo tanto, si tenemos diferentes páginas candidatas a salir de la memoria, interesará sacar aquellas que no hayan sido modificadas.

### 3.2. Políticas de asignación

En un sistema multiprogramado es necesario establecer alguna **regla para distribuir la memoria física** entre los distintos procesos que se ejecutan de manera concurrente en el sistema.

A la hora de distribuir la memoria entre los procesos encontramos dos grandes restricciones:

**1) El número máximo de páginas de la memoria física que se pueden asignar a un proceso**. Este número está determinado por el tamaño de la memoria física del sistema. Si permitimos a un único proceso ocupar toda la memoria del sistema, limitamos el grado de multiprogramación.

**2) El número mínimo de páginas de la memoria física que debemos asignar a un proceso**. Este número está determinado por la arquitectura del computador en el que se ejecuta el proceso. Hay que recordar que para poder ejecutar una instrucción, todas las páginas del espacio lógico al que se hace referencia durante la ejecución deben estar cargadas en la memoria.

Por ejemplo, en un lenguaje máquina en el que las instrucciones solo puedan tener un operando en la memoria y todas las referencias a memoria sean directas, es decir, si la dirección especificada en la instrucción es la dirección del operando, lo peor que puede suceder es que la instrucción esté en una página lógica y el dato esté en otra página lógica. Por lo tanto, a todo proceso, como mínimo, se le deben asignar dos páginas en la memoria principal.

En cambio, si las referencias a los operandos pueden ser direcciones indirectas, es decir, si en la instrucción se especifica una dirección de la memoria que contiene la dirección del operando buscado, entonces para asegurar que todo proceso podrá ejecutarse correctamente en el sistema se le deben asignar al menos tres páginas de la memoria: una que contenga la instrucción, otra con la dirección del operando y una tercera página que incluya el operando. En las arquitecturas actuales, de tipo RISC, en las que el lenguaje máquina es muy sencillo, las necesidades mínimas suelen ser de dos páginas, una para la instrucción y otra para el operando.

**Computadores PDP-8 o PDP-11**

En los computadores PDP-8 o los PDP-11, que disponían de una arquitectura de lenguaje máquina muy compleja (de tipo CISC), el número mínimo de páginas que se debía asignar a un proceso para asegurar una ejecución correcta podía ser considerable, por ejemplo, hasta seis en el PDP-11.

Considerando estos valores límite, el algoritmo de asignación deberá determinar el número de páginas que puede tener cargadas en memoria cada proceso.

### 3.2.1. Algoritmos de asignación

El número de páginas de la memoria principal que el sistema asigna a cada proceso determina el grado de multiprogramación y el uso que se puede hacer del procesador.

La política de asignación de páginas más sencilla consiste en dividir en partes iguales la memoria del sistema entre todos los procesos que se ejecutan y en definir un máximo y un mínimo de páginas que se pueden asignar al proceso. Entonces, todos los procesos tienen un tratamiento igualitario dentro del sistema.

Teniendo en cuenta que no todos los procesos tienen requerimientos de memoria iguales, con esta distribución hay casos en los que a un proceso se le asignan más o menos páginas de las que realmente necesita.

Otras políticas intentan asignar el número de páginas en función del tamaño del espacio lógico de cada proceso. Serían políticas proporcionales.

Otras políticas también tienen en consideración la prioridad de los procesos y asignan más páginas a aquellos procesos más prioritarios. No son más que una sofisticación de las políticas proporcionales.

También hay políticas adaptativas que intentan identificar las fases de ejecución existentes en la mayoría de los programas (inicialización, período permanente, etc.) y asignar el número de páginas adecuado para cada fase.

**Hiperpaginación (*trashing*)**

Independientemente del tipo de política aplicada, si el número de páginas asignadas a un proceso es inferior al número requerido por el proceso, la ejecución de este proceso constantemente producirá fallos de página. Entonces, el proceso pasa más tiempo "paginando" –intercambiando páginas entre la memoria y el área de intercambio– que ejecutándose. Esta situación recibe el nombre de hiperpaginación (*thrashing*).

Un sistema entra en hiperpaginación cuando, a causa de una sobrecarga del sistema o de una configuración inadecuada de los gestores de asignación y de reemplazo de páginas, el sistema está paginando continuamente y no tiene tiempo para ejecutar procesos. A consecuencia de esto, la utilización efectiva de la CPU es muy baja y el planificador podría interpretar mal este descenso en el rendimiento de la CPU e incrementar el grado de multiprogramación, decisión que agrava todavía más el mal funcionamiento del sistema, que se puede llegar a colapsar.

### 3.2.2. Conjunto de trabajo

El modelo del conjunto de trabajo[14] se intenta adaptar a la localidad de los procesos. En este algoritmo se define una ventana de trabajo en la que se tienen en cuenta un número determinado de referencias. El número de referencias determina el tamaño de la ventana (T), y el número de páginas diferentes que se referencian dentro de la ventana de trabajo es lo que forma el conjunto de

(14)En inglés, *working set*.

**Ved también**

Recordad que la localidad espacial se trata en el anexo 1, "Localidad temporal y localidad espacial de un proceso".

trabajo. Si una página se está utilizando, estará dentro del conjunto de trabajo, y si no se utiliza, saldrá del conjunto de trabajo *T* referencias después de ser referenciada por última vez.

El problema de este algoritmo es determinar el tamaño de la ventana. Si es demasiado pequeña, no se podrá sacar provecho de la localidad del proceso, y si es demasiado grande, se solaparán diferentes localidades.

El hecho de utilizar el conjunto de trabajo a la hora de asignar páginas al proceso es bastante sencillo. El sistema evalúa el conjunto de trabajo para cada proceso y asigna al proceso un número de páginas suficiente para tener el conjunto de trabajo cargado en la memoria. Si quedan páginas de la memoria libres, el sistema puede iniciar la ejecución de otro proceso. Si en algún momento la suma del conjunto de trabajo de todos los procesos que se ejecutan en el sistema supera la capacidad de la memoria física, el sistema puede suspender la ejecución de algún proceso y descargar todas las páginas asignadas a este proceso.

Con esta estrategia se consigue un buen uso de la CPU, se evita que el sistema entre en situación de hiperpaginación y, además, se consigue alcanzar un grado de multiprogramación del sistema razonablemente alto.

**Ejemplo de utilización del conjunto de trabajo**

Si tenemos la secuencia de referencias a páginas lógicas siguiente: {1, 0, 1, 5, 1, 1, 2, 8, 7, 1, 2, 3, 0}, y el tamaño de la ventana es 3, entonces el conjunto de trabajo en el momento en el que se referencia la página 5 es {0, 1, 5}, y cuando se referencia por primera vez la página 2, el conjunto de trabajo es {1, 2}.

## Resumen

En un sistema operativo es importante poder ejecutar procesos de cualquier tamaño, independientemente de la memoria física instalada. La memoria virtual es una técnica que permite establecer un enlace entre un espacio de direcciones grande (el espacio lógico del proceso) y un espacio de direcciones más pequeño (las direcciones físicas de la memoria principal), lo que posibilita ejecutar programas muy grandes e incrementar el grado de multiprogramación.

En este módulo didáctico se ha presentado una descripción detallada del funcionamiento de la gestión de la memoria virtual paginada: se han descrito las estructuras de datos necesarias para su implementación (y se ha destacado el tipo de implementación, por hardware o por software, más adecuado para cada uno) y se han identificado los diferentes tipos de acciones que debe llevar a cabo un gestor de la memoria virtual paginada: la carga inicial, la asignación y el reemplazo de páginas.

En la última parte del módulo se han descrito algunas políticas de asignación y reemplazo de páginas de la memoria, que han sido propuestas por la literatura, y se han evaluado sus ventajas e inconvenientes.

# Actividades

**1.** Una máquina ofrece a los usuarios un espacio de direcciones virtual de $2^{24}$ palabras. La máquina tiene $2^{18}$ palabras de la memoria física. La memoria virtual se implementa por paginación y el tamaño de la página es de 256 palabras. Una operación de usuario genera la dirección virtual 111234A6 (en hexadecimal). Explicad cómo establece el sistema la posición física correspondiente a esta dirección.

**2.** ¿Cómo influye en la gestión de la memoria virtual paginada el tamaño de la página?

**3.** Buscad información sobre el formato de las entradas de las tablas de páginas (PDE y PTE) en la arquitectura IA32. ¿Cuál es la diferencia entre una PDE y la PTE? ¿Cuál es la función de cada uno de los bits de control existentes?

**4.** Comparad el formato de las entradas de las tablas de páginas de los procesadores Intel de 64 bits con el formato de los procesadores Intel de 32 bits.

**5.** Cuando hay un cambio de contexto, ¿qué sucede con la información que tenemos en la TLB? Recordad que el término *cambio de contexto* hace referencia a las acciones que lleva a cabo el sistema cuando deja de ejecutar un proceso y pasa a ejecutar otro que estaba en espera para ser ejecutado.

**6.** ¿En la TLB podemos tener información de un proceso o de varios procesos a la vez? ¿Qué campos debería tener una entrada de la TLB para poder soportar varios procesos simultáneamente?

**7.** Decid si las estructuras y las técnicas de programación siguientes son útiles en un entorno en programación con prebúsqueda. Recordad que debéis tener en cuenta que cuando se produce un fallo de la página *k*, el algoritmo de prebúsqueda carga en la memoria la página *k* y la *k* + 1.

**a)** La pila.
**b)** La búsqueda secuencial.
**c)** La búsqueda dicotómica.
**d)** El código de un programa.
**e)** Operaciones con vectores.
**f)** El direccionamiento indirecto.

**8.** Considerad la secuencia siguiente de referencias a la memoria de un programa que consta de 460 palabras: 10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 458, 364.

**a)** Indicad la secuencia de referencias a la memoria suponiendo que el tamaño de la página es de 100 palabras.
**b)** Calculad la tasa de fallos de página de esta cadena de referencias suponiendo que el programa dispone de una memoria de 200 palabras para ser ejecutado, y que el algoritmo de reemplazo que se aplica es el FIFO.
**c)** ¿Qué tasa de fallos de página se generará si el algoritmo de reemplazo es el LRU?
**d)** ¿Qué tasa de fallos de página se generará si el algoritmo de reemplazo utilizado es el algoritmo óptimo?

**9.** Suponed que en un sistema con gestión de la memoria virtual paginada aplicamos una política de reemplazo que consiste en examinar cada página regularmente y descargarla de la memoria física si no se ha utilizado desde la última consulta. ¿Qué ganaríamos o perderíamos utilizando esta política en lugar de aplicar la política de reemplazo LRU o la de la segunda oportunidad?

**10.** Imaginad que queremos utilizar un algoritmo que necesita un bit de referencia (como el algoritmo de la segunda oportunidad o el que tiene en cuenta el conjunto de páginas de trabajo), pero el hardware del sistema no lo proporciona. ¿Podríais proponer alguna manera de simular el bit de referencia? En caso de que fuera posible, ¿sería muy costoso?

**11.** ¿Qué problemas pueden surgir si se sacan de la memoria páginas que intervienen en una operación de entrada/salida por DMA? ¿Cómo se podrían solucionar?

**12.** ¿Podemos tener algún problema si un algoritmo de reemplazo saca páginas de la memoria que contengan datos y/o código del núcleo del sistema operativo? ¿Por qué?

**13.** Consideremos la matriz bidimensional A declarada desde el lenguaje de alto nivel C de la manera siguiente: *int A[256][256];* podéis asumir que un entero ocupa 4 octetos, que la matriz se carga en la memoria contigua por filas, que A[0][0] se encuentra en la dirección

2048 del espacio lógico y que el sistema de gestión de memoria es paginado con un tamaño de página igual a 1 KB.

En la página 0 (que solo contiene código) tenemos un pequeño programa que inicializa la matriz a 0. El código del bucle principal es:

```
int A[256][256];
for (i=0; i<256; i++) {
  for (k=0; k<256; k++) {
    A[k][i] = 0;
  }
}
```

Contestad a las preguntas siguientes:

**a)** Suponiendo que tenemos una limitación de tres páginas físicas, ¿cuántos fallos de página se producirán si se aplica el algoritmo de reemplazo LRU? (no tengáis en cuenta el acceso al código del programa y tened en cuenta únicamente el acceso a los datos de la matriz).
**b)** ¿Cuántos fallos de página se producirán si cambiamos el orden de los bucles?
**c)** Haced el mismo análisis de los apartados anteriores a partir de un sistema con páginas de 2 KB. Suponed que la matriz está cargada en la memoria por filas.

**14.** La llamada al sistema *fork()* de Unix crea un proceso nuevo que es una copia casi idéntica del proceso que la ha invocado. Para reducir el coste computacional de esta llamada, la implementación de esta llamada utiliza una técnica denominada *copy on write* (COW). Buscad información para saber en qué consiste y cómo está implementada.

## Ejercicios de autoevaluación

**1.** Disponemos de un sistema de gestión de memoria basado en paginación con las características siguientes:

- Páginas de 4 KBytes.
- Espacio lógico de 64 KBytes.
- Espacio físico de 32 KBytes.
- Número de páginas físicas que puede llegar a ocupar un proceso: 4.

Dada la siguiente secuencia de accesos a octeto (las direcciones están codificadas en hexadecimal) realizadas por un proceso: 6b45, 2244, 3540, 6340, 4200, 5001, 2200, 6000, 2000, 4300, 5034, 6002, 2008:

**a)** Indicad qué fallos de página se producirán asumiendo:

- Que se aplica el algoritmo de reemplazo LRU.
- Que se aplica el algoritmo de reemplazo FIFO.

**b)** Indicad cuál es el contenido de la tabla de páginas del proceso una vez finalizada la quinta referencia a memoria utilizando el primero de los algoritmos.

**2.** Considerad la matriz bidimensional A declarada desde el lenguaje de alto nivel C de la manera siguiente: *int A[256][256]*. Podéis asumir que un entero ocupa 4 octetos, que la matriz se carga en la memoria contigua por filas, que A[0][0] se encuentra en la dirección 2.048 del espacio lógico y que el sistema de gestión de memoria es paginado, con un tamaño de página igual a 1 KB.

En la página 0 (que únicamente contiene código) tenemos un pequeño programa que inicializa la matriz a 0. Si el código del bucle principal es:

```
int A[256][256];
for (i=0; i<256; i++) {
  for (k=0; k<256; k++) {
    A[k][i] = 0;
  }
}
```

Contestad a las preguntas siguientes:

**a)** ¿Cuántos fallos de página se producirán si suponemos que el proceso puede tener tres páginas de la memoria física, una de las cuales siempre tiene la página que contiene el código y las otras dos son para los datos?
**b)** ¿Y si cambiamos el orden de ejecución de los bucles?
**c)** ¿Qué pasaría si se ejecutara el mismo código, pero escrito en lenguaje de alto nivel Fortran?

# Solucionario

## Ejercicios de autoevaluación

**1.** Como las páginas son de 4 KB y las direcciones lógicas son de 16 bits, los 4 bits altos de las direcciones lógicas (es decir, el primer dígito hexadecimal) indican el número de página lógica a la que estamos accediendo.

- **LRU**

```
Ref.          6 2 3 6 4 5 2 6 2 4 5 6 2
- LRU         6 2 3 6 4 5 2 6 2 4 5 6 2
               6 2 3 6 4 5 2 6 2 4 5 6
                 6 2 3 6 4 5 5 6 2 4 5
+ LRU                2 3 6 4 4 5 6 2 4
              F F F   F F F             -> 6 fallos
```

Al acabar la quinta referencia, la tabla de páginas tendrá (se ha asumido que únicamente las páginas referenciadas por el proceso son válidas y que los *frames* físicos se asignan empezando por el 0):

```
Entrada:   0 1 2 3 4 5 6 7 8 9 A B C D E F
Válido:      v v v v v
Presente:    p p p   p
Frame:       1 2 3   0
```

- **FIFO**

```
Ref.          6 2 3 6 4 5 2 6 2 4 5 6 2

- FIFO        6 2 3 3 4 5 5 6 2 2 2 2 2
               6 2 2 3 4 4 5 6 6 6 6 6
                 6 6 2 3 3 4 5 5 5 5 5
+ FIFO               6 2 2 3 4 4 4 4 4

              F F F   F F   F F         -> 7 fallos
```

**2.a)** Para calcularlo, es importante saber cómo se almacena la matriz en la memoria. En el lenguaje C, una matriz se carga en la memoria física de manera contigua por filas. Teniendo en cuenta este hecho, a partir de la dirección 2.048 (página 2) del espacio lógico del proceso encontramos los elementos A[0][0]; en la dirección 2.052, el elemento A[0][1]; en la dirección 2.056, el elemento A[0][2], y así hasta el elemento A[255][255], que está en la página 258 del espacio lógico. Las direcciones van de cuatro en cuatro, ya que los elementos de la matriz son enteros que ocupan 4 octetos.

Si miramos el patrón de acceso a la matriz del código que se ejecuta, vemos que se está accediendo a la matriz por columnas. El primer acceso, A[0][0], provoca un fallo de página y carga la página 2 en la memoria física. El segundo acceso, A[1][0], provoca otro fallo de página, dado que este elemento se mapea en la página 3 del espacio lógico del proceso. El tercer acceso, A[2][0], también provoca un fallo de página, y así en todos los accesos de la primera columna. Obtendremos un total de 256 fallos de página.

Cuando se accede a la segunda columna vuelve a suceder a lo mismo, ya que tenemos cargadas en la memoria las páginas 257 y 258, y el primer elemento de la segunda columna está en la página 2.

Por lo tanto, se producen un total de $256^2$ fallos de página, un resultado pésimo.

**b)** Si cambiamos el orden de los bucles, la cosa va mejor. El acceso a la matriz se hace por filas y, por lo tanto, aprovechamos la localidad espacial de la matriz. El proceso funcionará de la manera siguiente: el primer acceso al elemento A[0][0] provoca un fallo de página y se carga la página 2 en la memoria principal. Los 255 accesos siguientes no originarán ningún fallo de página porque todos los elementos de la primera fila de la matriz se mapean en la página 2.

El acceso al elemento A[1][0] volverá a provocar un fallo de página y se cargará en la memoria física la página 3; los 255 accesos siguientes no provocarán ningún fallo de página. Este esquema se repite en todas las filas de la matriz.

Por lo tanto, con el nuevo código pasamos a tener 256 fallos de página, un número mucho más razonable que en el caso anterior. Si se hubiera aplicado la prebúsqueda, aún habríamos podido reducir más el número de fallos de página.

**c)** En el supuesto de que el código se hubiera escrito en Fortran, los resultados habrían sido a la inversa. El lenguaje de alto nivel Fortran introduce las matrices en la memoria por columnas. El acceso a las matrices por columnas (como en el apartado *a*) es más eficiente que el acceso por filas.

# Glosario

**bit de presencia** *m* Bit asociado a cada entrada de la tabla de páginas de un proceso que indica si una página está presente en la memoria física o no. Si el bit es 0, la página no está; si el bit es 1, la página está.

**bit de referencia** *m* Bit asociado a cada entrada de la tabla de páginas de un proceso que indica si la página ha sido referenciada.

**bit de validez** *m* Bit asociado a cada entrada de la tabla de páginas de un proceso que indica si la página pertenece al espacio lógico del proceso.

**conjunto de trabajo** *m* Conjunto de páginas que son referenciadas por un proceso en un intervalo de T referencias consecutivas.

**fallo de página** *m* Situación que se produce cuando se accede a una dirección del espacio lógico del proceso que no está cargada en la memoria. Después del fallo se genera una excepción, y la rutina asociada a esta excepción se ocupa de cargar en la memoria la página que ha provocado el fallo.

**ventana** *f* Valor (T) del intervalo de referencias consecutivas que define el conjunto de trabajo.

**memoria virtual** *f* Método de gestión de la memoria que permite la ejecución de procesos sin que sus espacios lógicos deban estar totalmente cargados en la memoria y sin que las partes de los procesos cargadas en la memoria tengan que ocupar posiciones contiguas.

**memoria virtual paginada** *f* Método de gestión de la memoria virtual que divide el espacio lógico del proceso en páginas.

**prebúsqueda** *f* Mecanismo que intenta sacar provecho de la localidad espacial de las páginas de los procesos cargando en la memoria, además de la página que ha provocado un fallo de página, las k páginas siguientes con el fin de evitar fallos de página posteriores durante la ejecución del proceso.

**tabla de traducción de direcciones (TLB)** *f* Tabla implementada en hardware que almacena las últimas entradas de la tabla de páginas que han sido utilizadas por la MMU.

# Bibliografía


### Bibliografía básica

**Silberschatz, A.; Galvin, P.; Gagne. G.** (2008). *Operating Systems Concepts* (8.ª edición). John Wiley & Sons.

**Tamenbaum, A.** (2009). *Modern Operating Systems*. Prentice Hall.

### Bibliografía complementaria

**Bovet, D.; Cesati, M.** (2006). *Understanding the Linux Kernel* (3.ª edición). O'Reilly.

**Gorman, M.** (2004). *Understanding the Linux Virtual Memory Manager*. Prentice Hall.

# Anexos

## Anexo 1. Localidad temporal y localidad espacial de un proceso

Según el principio de localidad, los programas acceden a una parte bastante reducida de su espacio lógico dentro de un período de tiempo. Podemos encontrar los dos tipos de localidad siguientes:

**a) Localidad temporal**. Si una página es referenciada una vez, hay muchas posibilidades de que vuelva a ser referenciada pronto. En cambio, si una página no ha sido referenciada desde hace bastante tiempo, seguramente no se volverá a referenciar.

**b) Localidad espacial**. Si una página es referenciada, es muy probable que las páginas siguientes (contiguas en el espacio de direcciones virtual del proceso) sean referenciadas en breve.

El principio de localidad es aplicable a la ejecución de instrucciones de procesos y al acceso a los datos de un proceso.

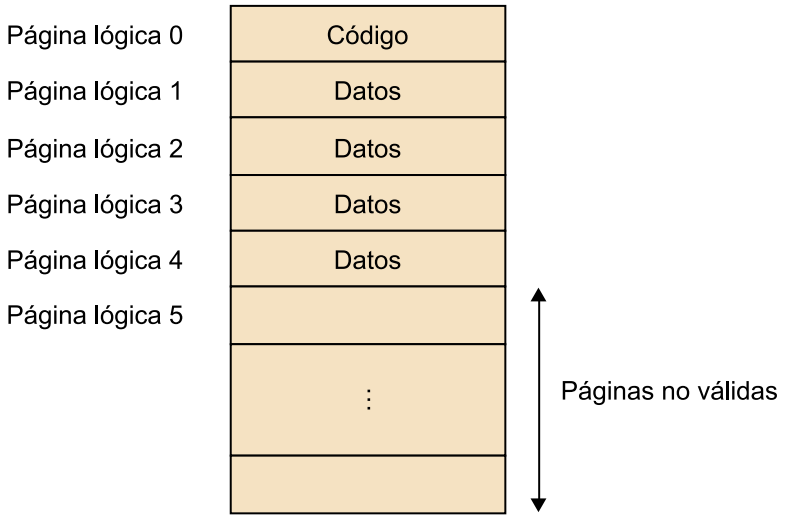
**Ejemplo. Principio de localidad**

Veamos un ejemplo que ilustra el principio de localidad. Supongamos que tenemos el proceso siguiente:

```
#define MAX 1024
int v[MAX];
main(){
  int i;
  for (i = 0; i < MAX; i++)
  v[i] = i;
}/* fin main */
```

Este proceso tiene definido un vector de 1.024 enteros. Un entero ocupa 4 octetos (32 bits) de la memoria y, por lo tanto, el espacio que ocupa el vector en la memoria es de 4.096 octetos. El proceso inicializa el contenido del vector, y por eso ejecuta el bucle *for (i = 0; i < MAX; i++)* con *MAX* igual a 1.024. Consideremos que el proceso se ejecuta en un sistema dotado de memoria virtual paginado y que el tamaño de cada página de la memoria física es igual a 1 KB. Supongamos que el espacio de direcciones virtual del proceso es el que se indica en la figura 5.

Figura 5. Espacio lógico del proceso




Cuando el proceso ejecute la primera instrucción, el sistema intentará acceder a la página virtual 0, la que contiene el código del programa. Podemos suponer perfectamente que el código del programa ocupa menos de 1 KB. Por lo tanto, todas las referencias al código para ejecutar cualquier instrucción del programa harán una referencia a la página virtual 0. Así, si miramos la secuencia de páginas referenciadas a la hora de ejecutar las instrucciones del programa, el patrón de referencias es: 0000 ... 0000. Desde el punto de vista de la localidad, las referencias al código del programa tienen localidad temporal.

Si nos fijamos en los datos, vemos que cuando se accede al primer elemento del vector, v[0], este genera la dirección lógica 1.024, que corresponde a la página 1. Las 255 referencias siguientes a los elementos v[1], v[2]..., v[255] corresponden a las direcciones lógicas 1028, 1032, 1036..., todas mapeadas en la página 1. Los 256 elementos siguientes del vector, del v[256] al v[511], generan accesos a la página 2 del espacio de direcciones virtual del proceso. El acceso a los elementos siguientes del vector, del v[512] al v[767], genera el acceso a la página 3 del espacio de direcciones virtual y, finalmente, las últimas 256 referencias acceden a la página 4.

Si miramos la secuencia de páginas referenciadas, encontramos el patrón siguiente:

1,1,1, ... 1,1,1, 2,2,2, ... 2,2,2, 3,3,3, ... 3,3,3,4,4,4, ... 4,4,4

Al analizar qué tipo de localidad aparece en este caso, vemos que, por una parte, hay localidad temporal mientras se accede a los elementos de una misma página, y, por otra, encontramos localidad espacial, ya que los elementos del vector ocupan páginas contiguas en el espacio de direcciones virtual, y estas páginas son referenciadas secuencialmente. Por lo tanto, se cumple la tendencia de que cuando referenciamos una página existe una probabilidad muy alta de que al cabo de un tiempo relativamente no muy grande se referencie la página siguiente.

Podemos sacar provecho de este comportamiento del proceso para evitar sacar páginas que son muy referenciadas. Por ejemplo, si se hace referencia continuamente a la página 0, sería muy poco eficiente llevarla al disco mientras se ejecuta el proceso, ya que se producen muchos fallos de página y habría un incremento del tiempo total de ejecución del proceso. Así pues, la localidad temporal de los procesos nos puede ayudar a definir políticas para decidir qué página es candidata a salir de la memoria.

### Cómo aprovechar localidad espacial

Hemos visto que en un sistema con paginación pura solo llevamos páginas a la memoria principal si se produce un fallo de página. Si suponemos que los procesos tienen localidad espacial, podemos adelantar trabajo y, cuando se produce un fallo de página, además de cargar en la memoria la página en

cuestión (página *k*), podemos cargar también la página siguiente, la *k* + 1 –la página contigua en el espacio de direcciones lógicas del proceso–. Esta técnica se denomina prebúsqueda.

En el caso de que se cumpla que el proceso tiene localidad espacial, cuando se hace referencia a la página *k* + 1 no se producirá un fallo de página porque ya había sido cargada en la memoria. A la hora de hacer la prebúsqueda, es posible cargar más de una página.

En el ejemplo anterior, si se aplicara la prebúsqueda de una página, a la hora de acceder al primer elemento del vector se cargarían en la memoria las páginas 1 y 2. Cuando hiciéramos la primera referencia a la página 3 cargaríamos la página 3 y la 4. El ejemplo que originalmente daba cuatro fallos de página cuando se accedía al vector solo da dos fallos de página si aplicamos la prebúsqueda.

**Anexo 2. Asignación de memoria en el núcleo de Linux**

Este anexo describe algunas peculiaridades de la gestión de memoria en Linux, centrada sobre la arquitectura IA-32.

A lo largo del ciclo de vida de los procesos de usuario, estos solicitan memoria tanto en el momento de la creación (para ubicar el código, los datos y la pila del nuevo proceso) como en tiempo de ejecución (si el proceso pide memoria dinámica, si hay que hacer crecer la pila, si se carga un nuevo ejecutable utilizando alguno de los llamamientos *exec*). El núcleo utiliza la granularidad de página para asignar/desasignar memoria a los procesos de usuario.

Ahora bien, el propio núcleo del sistema operativo también tiene necesidades de memoria dinámica: memorias intermedias, estructuras de datos, etc. Hay que notar que las peticiones de memoria realizadas por el núcleo del SO tienen unas particularidades respecto a las peticiones de los procesos de usuario:

- El núcleo solicita memoria para estructuras de datos de tamaños diversos, en muchos casos inferiores al tamaño de página. Asignar una página entera a cada petición de memoria provocaría un desperdicio de la memoria debido a la fragmentación interna. Este hecho es especialmente crítico porque la memoria asignada al núcleo de Linux nunca es intercambiada en disco.

- El núcleo solicita/libera memoria frecuentemente porque la implementación de muchas llamadas al sistema requiere utilizar memorias intermedias, estructuras de datos que, en algunos casos, se liberan al finalizar la ejecución de la propia llamada al sistema o al realizar otra llamada al sistema.

- En algunos casos, el núcleo necesita garantizar que una memoria intermedia esté ubicada en posiciones contiguas de memoria física. Esto es porque

algunos elementos del hardware trabajan directamente con direcciones físicas. Además, el hecho de asignar memoria físicamente contigua permite que no haya que modificar las tablas de páginas, con la consecuente mejora de rendimiento del TLB.

Todo esto hace que el núcleo del sistema operativo utilice un gestor de memoria específico para satisfacer sus necesidades de memoria dinámica. A continuación, veremos los gestores de memoria utilizados por el núcleo de Linux para satisfacer las peticiones de memoria del propio núcleo.

**1) Espacio de direcciones lógico del núcleo de Linux en IA-32**

Sobre la arquitectura IA-32, el espacio lógico tiene un tamaño máximo de 4 GB. Linux divide este espacio lógico en dos partes:

- Desde la dirección 0 hasta la 3 GB ([0x00000000-0xbfffffff]): este rango es propio de cada proceso en ejecución. Se almacenan el código/datos/pila de cada proceso. Cada vez que se produce un cambio de contexto, se cambia la tabla de páginas utilizada por la MMU y, por lo tanto, se modifican cuáles son las páginas físicas asociadas a estas páginas lógicas.

- Desde la dirección 3 GB hasta la 4 GB ([0xc0000000-0xffffffff]): este rango es propio del núcleo de Linux. Tiene el mismo contenido para todos los procesos en ejecución.

Dentro del rango de direcciones lógico del núcleo de Linux, encontramos estas partes:

- Desde la dirección 3 GB hasta la dirección 3 GB + 896 MB: este rango de direcciones lógicas mapea directamente la memoria física instalada, es decir, la dirección lógica 3 GB + $p$ está mapeada sobre la dirección física $p$.

- Un hueco de 8 MB (marcado como páginas inválidas en la tabla de páginas) que se inicia en la dirección 3 GB + 896 MB. Este hueco tiene la función de ayudar a detectar desbordamientos en los accesos a memoria que realiza el núcleo de Linux.

- Las asignaciones de memoria físicamente no contigua. Cada asignación es de tamaño múltiple del tamaño de página (4 KB) y entre dos asignaciones consecutivas existe un hueco de una página, que tiene una función análoga al del apartado anterior.

- Otros datos que obviaremos por tratarse de casos muy particulares.

**2) Asignación de memoria al núcleo**

El núcleo considera tres escenarios:

- Asignación de páginas físicamente contiguas: satisface peticiones de tamaño múltiple del tamaño de página.

- Asignación de memoria físicamente contigua: satisface peticiones de tamaño arbitrario.

- Asignación de memoria físicamente no contigua: puede utilizarse para asignar al núcleo memoria físicamente no contigua.

A continuación, discutimos estos escenarios.

**a) Asignación de páginas físicamente contiguas: *Buddy system***

Este gestor permite obtener páginas físicamente contiguas. Ahora bien, está diseñado para intentar minimizar el impacto de la fragmentación externa que puede provocar la contigüidad. El *buddy system* se fundamenta en dos características:

- La unidad de asignación son bloques de tamaño potencia de 2 (por lo tanto, podrá tener un cierto grado de fragmentación interna). Si el núcleo solicita *K* páginas, el algoritmo satisfará la petición asignándole $2^i$ páginas, donde $2^{i-1} < K <= 2^i$. El exponente *i* se denomina la orden de la petición.

- Dispone un mecanismo de compactación de bloques libres (compacta dos bloques libres contiguos de tamaño $2^k$ en uno de tamaño $2^{k+1}$).

La implementación del mecanismo utiliza *MAX_ORDER* (típicamente 10) listas. La lista *i*-ésima contiene la lista de bloques libres de orden *i*, es decir, bloques libres de tamaño $2^i$ páginas.

El *buddy system* se inicializa asignándole un bloque de páginas libres físicamente contiguas. Este número de páginas debe ser potencia de 2. Por ejemplo, asumiendo páginas de 4 KB, un *buddy* de 512 páginas ($2^9$) estaría gestionando 2.048 KB contiguos de memoria física. Este bloque se inserta en la lista de bloques libres de orden 9.

Cuando el núcleo solicita *K* páginas, se calcula el orden correspondiente a esta petición (*i*) y se comprueba si hay algún bloque libre en la lista correspondiente a este orden.

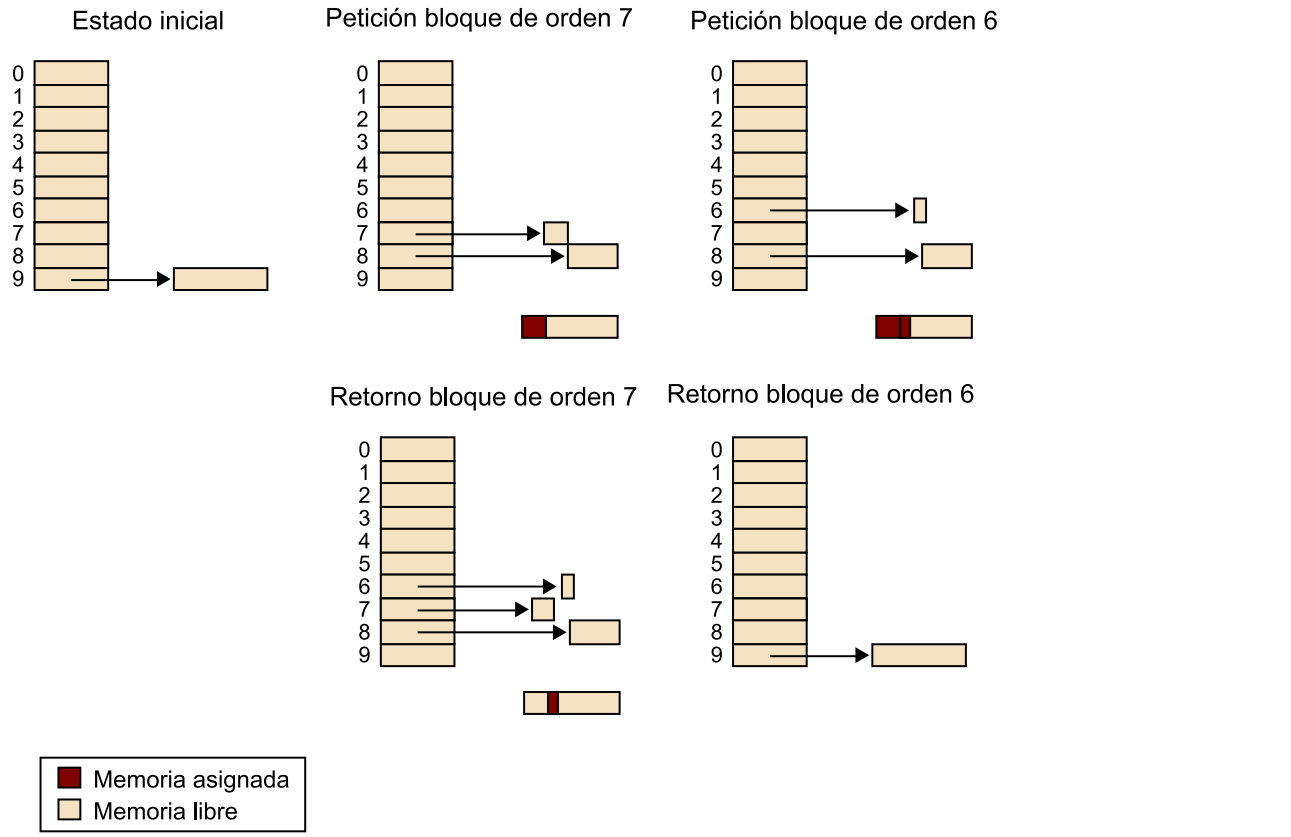
- En caso de existir, se elimina este bloque de la lista de bloques libres y se asigna al núcleo.

- En caso de no existir, se comprueban las listas de bloques libres de orden superior ($i + 1$, $i + 2$...) hasta encontrar algún bloque libre. El bloque se elimina de su lista y, asumiendo que el bloque sea de orden $k$, se divide en dos bloques de orden $i$, uno de orden $i + 1$, uno de orden $i + 2$... y uno de orden $k - 1$. Uno de los bloques de orden $i$ se asigna al núcleo y el resto de los bloques se insertan en las listas de bloques libres correspondientes.

Cuando el núcleo libera un bloque, este se incorpora a la lista de bloques libres correspondiente a su orden. Además, el sistema puede compactar dos bloques libres contiguos de orden $i$ para obtener un bloque libre de orden $i + 1$, y repetir esta operación recursivamente. La implementación del mecanismo utiliza *bitmaps* para detectar bloques libres contiguos.

La figura 6 muestra un ejemplo del funcionamiento de este gestor. Se muestra el contenido de la lista de bloques libres para cada orden y cómo evolucionan a medida que se realizan una serie de peticiones.

Figura 6. Ejemplo de funcionamiento del *buddy system*




Las versiones 2.6 del núcleo de Linux utilizan este gestor.

En el fichero /proc/buddyinfo podemos obtener información sobre su utilización actual. Cada línea corresponde a una zona de memoria y cada columna corresponde al número de páginas libres de aquel orden. A continuación, tenéis un ejemplo de la información que podemos obtener si consultamos este fichero. El concepto de *nodo* hace referencia a máquinas NUMA *(non-uniform*

*memory access*) donde un rango de direcciones físicas puede tener un tiempo de acceso diferente a otro rango de direcciones físicas. El concepto de *zone* responde a peculiaridades de la arquitectura IA-32, donde no toda la memoria física instalada tiene el mismo tratamiento: *DMA* corresponde a las páginas por debajo de la dirección 16 MB (por una limitación de los buses ISA antiguos, había que hacer las transferencias por DMA en este rango), *normal* corresponde a páginas entre la dirección 16 MB y la dirección 896 MB, y la *high mem* corresponde a las páginas por encima de la dirección 896 MB (como se ha comentado en el punto 1 de este anexo, el núcleo de Linux instalado sobre un procesador IA-32 no puede acceder directamente a todo el espacio físico de memoria por encima de esta dirección. En procesadores de 64 bits esta zona está vacía). Cuando el núcleo solicita memoria al utilizar el *buddy system*, debe especificar de qué zona la quiere.

```
user@host:~$ more /proc/buddyinfo
Node 0, zone      DMA     4     3     4     4     3     3     3     0     0     1     1
Node 0, zone   Normal   482   689   508   253    75    43    36     8     1     0     1
Node 0, zone HighMem     1     2    23    16     6     1     1     0     0     0     0
```

**b) Asignación de memoria físicamente contigua (tamaño arbitrario)**

En este escenario estamos pensando en peticiones de memoria de tamaño arbitrario y, en muchos casos, de tamaño inferior al tamaño de página. Aunque la idea del *buddy system* puede ser extendida para satisfacer peticiones de tamaño inferior al tamaño de página, la fragmentación que puede causar y la gestión de los *bitmaps* necesarios para poder hacer las compactaciones provocan que no sea la mejor opción.

El núcleo de Linux ha utilizado diferentes gestores para este escenario. A continuación, se efectúa una descripción de las principales características de algunos de ellos. Al compilar el núcleo de Linux es posible elegir qué gestor queremos utilizar. Los tres gestores presentados tienen la misma interfaz.

Estos gestores están implementados sobre el gestor anterior (*buddy system*).

**SLAB allocator**

Este gestor está disponible en el núcleo de Linux desde la versión 2.2. Sus características principales son:

- Un *slab* (losa) es un conjunto de una o más páginas físicas contiguas.
- Una *slab-cache* está formada por una o más losas. Todas las *slabs* de una *slab-cache* son del mismo tamaño.
- Existe una *slab-cache* para cada objeto o estructura de datos que el núcleo necesite crear dinámicamente (por ejemplo, el *task_struct* de los procesos

o los *inodes* de los ficheros). Cada *slab-cache* puede tener su propio tamaño de *slab*.

La figura 7 muestra un ejemplo que relaciona estos conceptos:

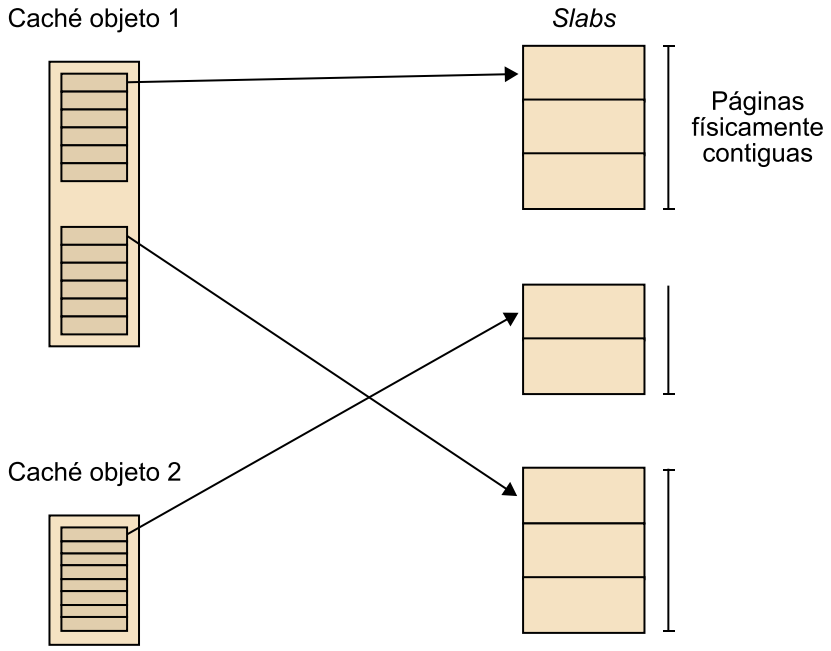



Figura 7. Ejemplo donde se muestran dos *slab-cache*, una con dos losas (de 3 páginas) y la otra con una losa (de 2 páginas)

Al crear una *slab-cache*, se asigna un *slab* a la *slab-cache* y la *slab-cache* pasa a contener un cierto número de objetos que inicialmente están marcados como libres. Por ejemplo, si consideramos una *slab-cache* para objetos de tamaño 504 octetos, las páginas son de 4.096 octetos y cada *slab* de esta *slab-cache* contiene 2 páginas, la *slab-cache* podría llegar a contener tantos objetos como la parte entera de 4.096 * 2/504, es decir, 16 objetos. En función del tamaño de los objetos, el gestor almacena el estado de los objetos y las colas que permiten gestionarles en el propio *slab* o en una estructura externa a la losa.

Cada losa de una *slab-cache* tiene uno de estos tres estados: *empty* (todos sus objetos están marcados como libres), *partial* (algunos objetos están marcados como libres y otros, como utilizados) y *full* (todos los objetos están marcados como utilizados). Inicialmente, la losa está *empty*.

Cuando el núcleo solicita memoria para un objeto a la *slab-cache* correspondiente, el gestor actúa del modo siguiente:

- Si la *slab-cache* tiene alguna losa marcada como *partial*, obtiene memoria para el objeto de esta losa.
- En cambio, si la *slab-cache* tiene alguna losa marcada como *empty*, obtiene memoria para el objeto de esta losa.

- De otro modo, el gestor asigna una nueva losa a la *slab-cache* (utilizando el *buddy system*) y obtiene memoria para el objeto de esta losa.

Cuando el núcleo devuelve la memoria de un objeto, solo hay que marcar el objeto como libre y, si es necesario, actualizar el estado de la losa correspondiente.

Las ventajas de este gestor son que minimiza la fragmentación interna y que es muy rápido asignando/desasignando objetos al núcleo (notad que la memoria ha sido preasignada al gestor al crear cada *slab-cache*). El inconveniente principal es la falta de escalabilidad en sistemas en muchos procesadores.

En el fichero /proc/slabinfo podemos obtener información sobre todas las *slab-caches* gestionadas por este gestor. A continuación, tenéis un ejemplo de la información que podemos obtener si consultamos este fichero. Si escribimos sobre este fichero, es posible modificar la configuración del gestor. La instrucción *slabtop* también nos da información sobre la utilización de estas *slab-caches*. Cada línea corresponde a una *slab-cache*. Significado de las seis primeras columnas: nombre de la *slab-cache*, número de objetos de la *slab-cache* asignados al núcleo, número total de objetos en la *slab-cache*, tamaño del objeto (en octetos), número de objetos en cada losa, número de páginas en cada *slab*. La penúltima columna indica el número de losas en la *slab-cache*.

```
user@host:~$ more /proc/slabinfo
slabinfo - version: 2.1
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab>
: tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs>
<num_slabs> <s haredavail>
fat_inode_cache     342   342  416  19 2 : tunables 0 0 0 : slabdata   18   18 0
fat_cache           170   170   24 170 1 : tunables 0 0 0 : slabdata    1    1 0
UDPLITEv6             0     0  704  23 4 : tunables 0 0 0 : slabdata    0    0 0
UDPv6                46    46  704  23 4 : tunables 0 0 0 : slabdata    2    2 0
tw_sock_TCPv6         0     0  256  16 1 : tunables 0 0 0 : slabdata    0    0 0
TCPv6                46    46 1408  23 8 : tunables 0 0 0 : slabdata    2    2 0
kcopyd_job            0     0  272  15 1 : tunables 0 0 0 : slabdata    0    0 0
dm_uevent             0     0 2464  13 8 : tunables 0 0 0 : slabdata    0    0 0
dm_rq_target_io       0     0  232  17 1 : tunables 0 0 0 : slabdata    0    0 0
kmalloc_dma-512      16    16  512  16 2 : tunables 0 0 0 : slabdata    1    1 0
ext2_inode_cache  81537 85904  504  16 2 : tunables 0 0 0 : slabdata 5369 5369 0
ext3_inode_cache  19140 20048  512  16 2 : tunables 0 0 0 : slabdata 1253 1253 0
```

**SLUB allocator**

SLUB es un gestor que intenta mejorar el rendimiento y la escalabilidad del gestor SLAB mediante la reducción de la cantidad de metainformación almacenada (de hecho, elimina la metainformación almacenada en los *slabs*) y la

simplificación de la estructura general del gestor. Así, la *U* del nombre SLUB indica *unqueued* porque SLUB elimina la mayoría de las colas de objetos existentes en SLAB.

**SLOB allocator**

SLOB (*simply list of blocks*) es un gestor que está implementado pensando en dispositivos pequeños y empotrados. Por lo tanto, su coste computacional es muy inferior al de los gestores SLAB y SLUB. La desventaja de este gestor es que la fragmentación interna puede hacerle desperdiciar la memoria disponible.

**c) Asignación no contigua**

El núcleo de Linux también puede solicitar memoria sin imponer la restricción de que las páginas de memoria física asignadas deban ser físicamente contiguas. El núcleo de Linux utiliza este tipo de asignación cuando el usuario instala algún módulo, al realizar entrada/salida sobre algunos dispositivos y cuando necesita acceder a memoria física por encima de la dirección física 896 MB.

Las asignaciones deben tener tamaño múltiple del tamaño de página. Además, el gestor separa las asignaciones consecutivas mediante una página marcada como inválida. Esto permite detectar desbordamientos dentro de cada asignación de memoria de este tipo.