La gestión de la memoria

José Ramón Herrero Zaragoza
Enric Morancho Llena
Dolors Royo Vallés

PID_00169383

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico, presentamos los principios de la gestión de la memoria principal, uno de los recursos más importantes de cualquier sistema operativo multiprogramado. La **gestión de la memoria** se encarga básicamente de asignar la memoria física del sistema computador, que es finita, a los procesos que la solicitan. Ningún programa se puede ejecutar si no se le ha asignado memoria principal para almacenar su código, sus datos y su pila de ejecución.

Primero se introducen los **espacios de direcciones**, que son el lógico y el físico. Mientras que el espacio físico es único, cada proceso en ejecución tendrá asociado un espacio lógico; cada espacio lógico será independiente de los del resto de procesos. El sistema de gestión de la memoria esconderá el espacio físico a los usuarios y la visión que éstos tendrán de la memoria será el espacio lógico. Por lo tanto, será necesario un mecanismo que traduzca, de forma transparente para el usuario, los accesos al espacio lógico en accesos al espacio físico.

La coexistencia de múltiples espacios de direcciones lógicos que pertenecen a diferentes procesos residentes en la memoria principal introduce la necesidad de proteger los espacios de direcciones. En un entorno multiprogramado, el gestor de la memoria tendría que disponer de **mecanismos de protección de la memoria**, es decir, tendría que garantizar que un proceso no pudiera acceder a la memoria física asignada a otro proceso o al sistema operativo. También es deseable que, si los procesos lo autorizan, acepte **compartir la memoria** y permita, así, espacios comunes para los procesos interesados.

Presentamos la implementación de un gestor de memoria sencillo que, con la ayuda del hardware, cumple la mayoría de los requerimientos indicados. Una vez analizadas sus limitaciones, describimos el gestor de memoria virtual. Este último sistema será capaz de independizar los procesos de la cantidad de memoria física instalada en la máquina, con lo cual permitirá ejecutar procesos que no caben en la memoria física.

Finalmente, reflexionamos sobre las posibilidades de ampliar, en tiempo de ejecución, el espacio lógico de un proceso. También comentamos algunos errores de programación típicos relacionados con el uso de la memoria.

Objetivos

Los materiales didácticos de este módulo contienen las herramientas necesarias para que el estudiante alcance los objetivos siguientes:

1. Justificar la existencia de diversos espacios de direcciones y de los mecanismos de traducción de direcciones en tiempo de ejecución.

2. Conocer las funcionalidades que tienen que ofrecer los sistemas de gestión de la memoria en un sistema operativo multiproceso de propósito general.

3. Saber cómo se puede aumentar, en tiempo de ejecución, el espacio lógico de los procesos.

1. Espacios de direcciones

La memoria física (RAM) del computador es un vector de palabras (típicamente del tamaño de un byte), cada una de las cuales se identifica mediante una dirección (@).

En los sistemas operativos multiproceso, los procesos en ejecución (y el sistema operativo mismo) comparten la memoria física. Cada proceso (y el sistema operativo) tiene asignado un conjunto de direcciones físicas para almacenar su código, sus datos y su pila de ejecución. Ahora bien, el sistema operativo tiene que garantizar que ningún otro proceso podrá acceder al conjunto de direcciones asignado al resto de procesos (y al sistema operativo).

Una posible solución pasa por considerar dos tipos de espacios de direcciones (el espacio físico y el espacio lógico) y un mecanismo de traducción de direcciones lógicas a direcciones físicas:

1) Espacio físico: está determinado por la memoria física instalada en el sistema computador. Las direcciones de memoria de este espacio se llaman direcciones físicas. Asumiremos que el rango de direcciones físicas del espacio físico es lineal y que empieza en la dirección física 0 y finaliza a la dirección Memoria_Física_Instalada - 1.

2) Espacio lógico: este espacio es la visión que el sistema operativo ofrece del espacio físico. Cada proceso tiene asociado un espacio lógico, independiente de los del resto de procesos, a través del cual puede acceder a su código, a sus datos y a su pila de ejecución. Las direcciones de memoria de este espacio se llaman direcciones lógicas. Asumiremos que este espacio es lineal. Potencialmente, el rango de direcciones lógicas del espacio lógico empieza a la dirección lógica 0 y finaliza a la dirección $2^{\text{Tamaño\_Bus\_Direcciones}}$-1, donde Tamaño_Bus_Direcciones está determinada por el tamaño del bus de direcciones del procesador (típicamente, 32 o 64 bits en los procesadores actuales). Cabe destacar que, para cada proceso, existe un rango de direcciones lógicas válidas, que dependerá de la cantidad de memoria que utilice el proceso. La traducción de dirección lógica a dirección física dependerá del proceso que la realice; por ejemplo, la dirección lógica 0x0bc00000[1] puede ser traducida a direcciones físicas diferentes para cada proceso del sistema.

(1)En este módulo, mostramos las direcciones lógicas y las direcciones físicas codificadas en hexadecimal.

Existen dos alternativas para implementar esta traducción de direcciones:

1) Estática (en tiempo de carga): al cargar un fichero ejecutable en la memoria se realiza la traducción de cada dirección lógica a la dirección física correspondiente. Por lo tanto, cada dirección lógica se traduce una única vez a lo largo de la ejecución del programa.

2) Dinámica (en tiempo de ejecución): al cargar un fichero ejecutable en la memoria no se realiza ningún tipo de traducción. La ejecución del proceso hará que el procesador genere direcciones lógicas y un hardware especializado (la MMU, *Memory Management Unit*) traducirá cada dirección lógica en la dirección física correspondiente. A diferencia del caso anterior, en el que cada dirección lógica se traducía exactamente una vez, en este caso cada dirección lógica se traduce tantas veces como sea referenciada.

Los sistemas operativos multiproceso actuales utilizan mecanismos basados en la traducción dinámica porque son más flexibles que los basados en la traducción estática.

2. Un primer mecanismo de traducción dinámica

A continuación, presentamos un mecanismo sencillo de traducción dinámica de direcciones. Aunque este mecanismo permite aislar el espacio lógico de un proceso (y del sistema operativo) de los del resto de procesos, veremos que tiene algunas limitaciones que lo convierten en inapropiado para los sistemas computadores actuales.

Cada proceso tendrá asociado un espacio lógico. El rango de direcciones válidas del espacio lógico del proceso i-ésimo será [0, tamaño_i-1], donde *tamaño_i* es el tamaño del espacio lógico del proceso i-ésimo (podemos asumir que coincide con la suma de los tamaños de sus regiones de código, datos y pila de ejecución). Cada proceso se cargará en una serie de posiciones consecutivas de la memoria física; *base_i* será la dirección de memoria física base a partir de la cual se carga el proceso i-ésimo.

La MMU de este sistema llevará a cabo dos tareas:

- Comprobará que las direcciones lógicas (@L) generadas por un proceso se encuentren dentro del rango de direcciones válidas para el proceso. En caso de que una @L no sea válida, la MMU generará una excepción del tipo "acceso a memoria inválido".

- Traducirá las direcciones lógicas en direcciones físicas (@F) mediante una suma de la dirección lógica con la dirección base.

La MMU se programará con dos registros de control:

- Base: indica la dirección física inicial asignada al proceso en ejecución.
- Tamaño: indica el tamaño en bytes del espacio lógico del proceso en ejecución.

La figura 1 muestra el hardware que lleva a cabo estas tareas. El procesador genera direcciones lógicas para acceder al código, a los datos o a la pila almacenados en la memoria física. La MMU comprueba si la dirección lógica generada es válida para el proceso (es decir, si pertenece al rango [0 ... tamaño-1]) utilizando un comparador. Si no está dentro del rango, la MMU genera una excepción del tipo "dirección lógica inválida" y no realiza ninguna traducción. Si se encuentra dentro del rango, suma la dirección lógica a la dirección física base y obtiene la dirección física.

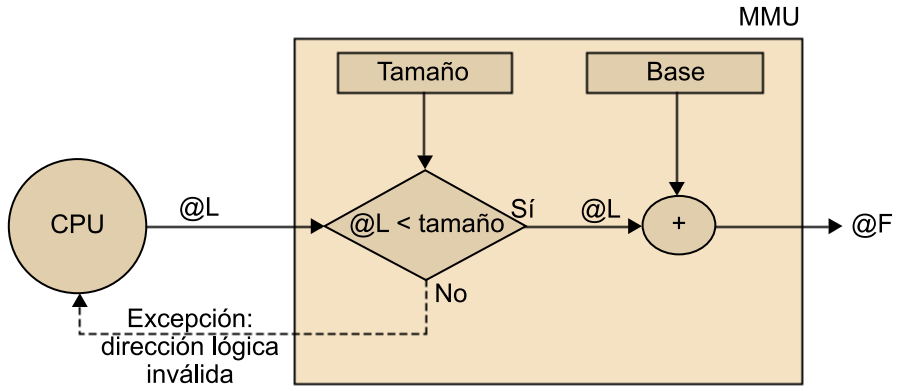

Figura 1. Esquema de una MMU sencilla que lleva a cabo traducción dinámica de direcciones y que permite aislar el espacio lógico de un proceso del resto de procesos.

Con el fin de garantizar que el mecanismo aísle el rango de direcciones de un proceso del resto de procesos, las instrucciones del lenguaje máquina que permitan modificar el valor de los registros de control de la MMU tendrán que ser privilegiadas. Cada vez que cambie de contexto será necesario modificar el valor de estos registros de control.

Ved también

En el apartado 3 del módulo 2, encontraréis más información relacionada con las instrucciones privilegiadas del lenguaje máquina.

La figura 2 presenta un ejemplo en el que asumimos que la memoria física instalada es de 1 MB (1.024 KB) y que está ocupada por el sistema operativo (256 KB a partir de la dirección 0x00000[2]), por el proceso 1 (128 KB a partir de la dirección 0x80000) y por el proceso 2 (64 KB a partir de la dirección 0xC0000).

[(2)]Como el espacio físico es de 2^{20} posiciones, cinco dígitos hexadecimales nos permiten codificar cualquier dirección del espacio.

Queremos crear un proceso (proceso 3) a partir de un ejecutable que define las regiones siguientes: código (64 KB), datos inicializados (16 KB), datos no inicializados (32 KB) y pila de ejecución (16 KB). Si se crea un proceso y se carga este fichero ejecutable en la memoria, el sistema operativo tiene que saber cuánta memoria necesita. La memoria necesaria viene dada por el tamaño de las regiones de código, datos (inicializados y no inicializados) y pila. En este caso, serían necesarios 128 KB.

Ved también

Consultad el ejemplo de compilación en el anexo del módulo didáctico 2.

El sistema operativo tiene que buscar memoria física para incluir este proceso. Para conseguirlo, necesitará alguna estructura de datos que represente el espacio libre en la memoria física. Asumimos que el sistema operativo decide cargar el proceso en el rango de direcciones físicas que empieza a partir de la dirección 0x40000. El sistema operativo tendrá que guardar en alguna estructura de datos el tamaño (128 KB) y la dirección física base (0x40000) asociados a este nuevo proceso.

Estructura de datos del SO con la información necesaria para traducir las direcciones lógicas de cada proceso (y del propio SO)

	Base	Tamaño
SO	0x00000	256 KB
Proceso 1	0x80000	128 KB
Proceso 2	0xC0000	64 KB
Proceso 3	0x40000	128 KB

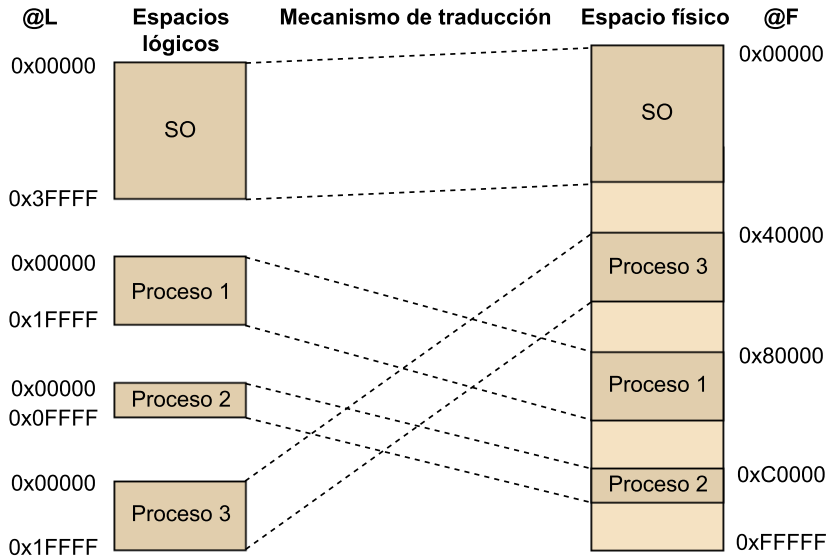

Figura 2. Ejemplo en el que el sistema operativo y tres procesos comparten el espacio físico y en el que el mecanismo de traducción dinámica presentado en la sección 2 traduce las direcciones lógicas en direcciones físicas.

Notad que cada proceso (y el sistema operativo) tiene un espacio lógico independiente del resto de procesos. Aunque aparentemente estos espacios lógicos están superpuestos (por ejemplo, en todos los espacios lógicos existe la dirección lógica 0x00FFF), el mecanismo de traducción los transforma en regiones disjuntas del espacio físico.

Notad que el código almacenado en la memoria física utilizará direcciones lógicas (dentro del espacio lógico del proceso correspondiente) para codificar las direcciones de salto y las direcciones de rutinas.

2.1. Limitaciones de este sistema

A causa de su simplicidad, este sistema de gestión de la memoria presenta una serie de problemas:

1) Contigüidad: el esquema presentado requiere que la memoria física en la que se carga un proceso sea contigua. Por ejemplo, en el ejemplo de la figura 2 no será posible cargar un proceso que necesite 200 KB porque, a pesar de disponer de 448 KB (128 KB + 128 KB + 192 KB) de memoria libre, no es posible encontrar un fragmento contiguo de 200 KB libres. Este problema se conoce con el nombre de fragmentación externa.

Fragmentación externa

El problema de la fragmentación externa también aparece en situaciones cotidianas, como el aparcamiento de coches en línea en una calle que no señalice las plazas de aparcamiento. Aunque pueden existir diversos agujeros disponibles para estacionar un vehículo, para poder aparcarlo es necesario que exista un agujero lo bastante grande para que quepa. La suma de los tamaños de los agujeros disponibles puede ser muy superior al tamaño del vehículo, pero no podremos aparcar el vehículo si el espacio libre necesario no está disponible de forma contigua.

2) Dificultad de crecimiento dinámico del espacio lógico de un proceso: algunos procesos pueden necesitar aumentar dinámicamente el tamaño de su espacio lógico para poder invocar procedimientos recursivos o para crear estructuras de datos en tiempos de ejecución. A causa de la contigüidad de este sistema de gestión de la memoria, para aumentar el espacio lógico del proceso habría que disponer de memoria física libre contigua a la que tiene asignada el proceso o bien, al cargar el proceso, reservar más memoria de la necesaria. En general, eso no siempre será factible.

3) No permite compartir porciones de código o datos entre diversos procesos: si diversos procesos están ejecutando el mismo programa (por ejemplo, el intérprete de comandos o un editor de texto), no es razonable que el código esté replicado en la memoria física para cada proceso que lo está ejecutando. Sería más efectivo, si fuera posible, que el código estuviera cargado una única vez en la memoria física; los procesos compartirían este código y cada uno trabajaría con sus regiones de datos y de pila privadas. También sería factible que diversos procesos quisieran compartir un fragmento de su región de datos. El esquema presentado en este apartado no permite la compartición de memoria entre procesos.

4) No permite la carga de un proceso que no quepa en la memoria física.

Por lo tanto, este sistema de gestión de la memoria no es adecuado para los sistemas operativos multiproceso de propósito general. En el apartado 3, daremos las pinceladas básicas de un sistema de gestión de la memoria con traducción dinámica que resuelve estos problemas.

3. Memoria virtual

Existen diversas implementaciones posibles de un sistema de gestión de la memoria que resuelva los problemas del sistema presentado en el apartado anterior. En este apartado, presentaremos un sistema de gestión de la memoria basado en paginación, aunque hay otras alternativas. Veremos que este sistema permitirá independizar los procesos de la cantidad de memoria física instalada en la máquina, con lo cual será posible ejecutar procesos que requieran más memoria de la disponible. Se dice que estos sistemas ofrecen memoria virtual.

3.1. Paginación

De forma transparente para el programador, un sistema de gestión de la memoria basado en paginación divide el espacio lógico del proceso en trozos de tamaño fijo, que llamamos páginas[3]. El espacio físico también se divide en trozos del tamaño de una página, llamados *frames*. Este sistema de traducción ubicará cada página en un *frame* de memoria física.

(3)La arquitectura IA-32 de Intel utiliza páginas de un tamaño de 4 KB (2^{12} bytes).

La figura 3 muestra un esquema del hardware que se necesita para implementar la paginación.

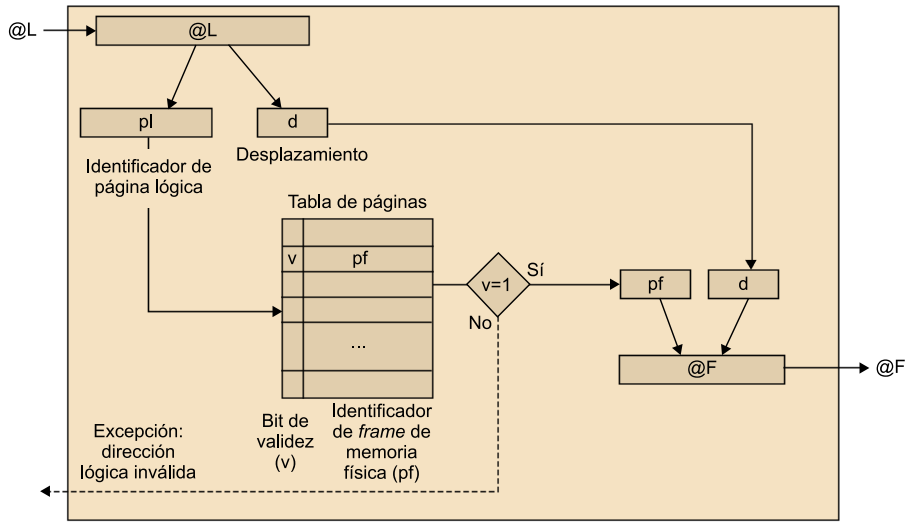

Figura 3. Esquema de una MMU que implementa paginación pura.

El mecanismo de traducción divide las direcciones lógicas en dos componentes, que son el identificador de página lógica y el desplazamiento dentro de la página. Utilizando una tabla, llamada tabla de páginas, se determina el identificador de *frame* en el espacio físico donde se encuentra la página lógica. Finalmente, se construye la dirección física concatenando el identificador de *frame* y el desplazamiento.

División de las direcciones lógicas

Como el procesador codifica las direcciones de memoria en base 2 y el tamaño de página es una potencia de 2 ($2^{log2(tamaño\_página)}$), la división de las direcciones lógicas en identificador de página lógica y el desplazamiento dentro de la página se hace de forma trivial. El desplazamiento son los log2(tamaño_página) bits bajos de la dirección lógica y el identificador de página lógica son el resto de bits de la dirección lógica.

La tabla de páginas es propia de cada proceso y tiene tantas entradas como páginas pueda llegar a tener el espacio lógico de un proceso ($2^{tamaño\_bus\_direcciones}$/tamaño_página). En cada una de las entradas se almacena un identificador de *frame* y una serie de bits de control, entre los cuales está, como mínimo, el bit de validez, que indica si esta página lógica pertenece al espacio lógico del proceso. Otros bits de control posibles serían los bits de protección (read, readwrite, execute), que indican qué operaciones es posible ejecutar en la página[4].

(4)Como la granularidad de estas protecciones es la de la página, dentro de la misma página no será posible mezclar información correspondiente a regiones diferentes.

En caso de intentar acceder a una página lógica marcada como inválida en la tabla de páginas, se provocará una excepción que notificará este hecho al sistema operativo. Normalmente, esta excepción provocará que el sistema operativo haga finalizar inmediatamente este proceso porque se considera que el proceso presenta un error de programación.

Este sistema de gestión de la memoria presenta algunos problemas prácticos que complican su implementación real:

- El tamaño de la tabla de páginas puede ser muy grande. Por ejemplo, si el tamaño del bus de direcciones es de 32 bits y el tamaño de la página es de 4 KB, la tabla de páginas tiene 2^{20} entradas (1.048.576 entradas[5]). Si cada entrada ocupa 4 bytes, representa 4 MB. Este tamaño resulta impracticable y son necesarias organizaciones que permitan reducir el tamaño de la tabla de páginas, como por ejemplo estructurar la tabla de páginas en dos niveles[6].

(5)Todos los procesos del sistema tendrán una tabla de páginas de este tamaño, independientemente de la cantidad de páginas que ocupe el proceso. El bit de validez indicará qué entradas son las que realmente utiliza el proceso.

(6)Es la solución adoptada por los procesadores que implementan la arquitectura IA-32 de Intel. No es el objetivo de este documento presentar esta solución.

- A causa del tamaño, la tabla de páginas no se puede almacenar en registros y hay que almacenarla en la memoria[7]. Por lo tanto, cada referencia a la memoria lógica implicaría dos accesos a la memoria física: uno a la tabla de páginas y otro a la dirección física calculada. Este sobrecoste es inaceptable, por lo que el hardware proporciona la TLB (*Translation Lookaside Buffer*), una memoria caché especializada en almacenar las entradas de la tabla de páginas más utilizadas recientemente.

(7)La MMU tendrá un registro de control que indicará a partir de qué posición de memoria física se encuentra la tabla de páginas del proceso en ejecución.

Un sistema operativo que utilice la paginación necesitará una serie de estructuras de datos para llevar a cabo la gestión de la memoria. Además de las tablas de páginas de cada proceso (almacenadas en el espacio de datos del sistema operativo), será necesaria alguna estructura que indique si los *frames* de la memoria física están libres u ocupados (tabla de *frames*) y algún procedimiento para obtener *frames* libres.

3.1.1. Ubicación de las regiones de código, datos y pila dentro del espacio lógico

En el primer esquema presentado en el apartado 2, ubicábamos las regiones de código, datos y pila contiguamente dentro del espacio lógico.

Utilizando la paginación, las regiones de código y de datos se ubican típicamente en un extremo del espacio lógico, mientras que la de pila se ubica en el otro; todas las páginas intermedias estarán marcadas como inválidas. Esta distribución facilitará el hecho de que las regiones de datos y de pila del proceso puedan crecer dinámicamente.

Utilizando la paginación, este agujero en el espacio lógico no implica un desperdicio de memoria física porque todas las páginas lógicas correspondientes al agujero están marcadas como inválidas, con lo cual no tendrán asociado ningún *frame* de memoria física. En cambio, en el esquema presentado en el apartado 2, esta disposición sí provocaría un desperdicio de memoria física.

La figura 4 presenta un ejemplo en el que asumimos que el espacio lógico puede llegar a tener ocho páginas y el espacio físico tiene dieciséis *frames*. Cargamos dos ejecutables, el primero con dos páginas de código, una de datos y una de pila, y el segundo con tres de código, una de datos y una de pila. Ahora bien, el segundo ejecutable está cargado dos veces (dos usuarios diferentes están ejecutando este programa) y así demostraremos que no hay que duplicar el código en la memoria física.

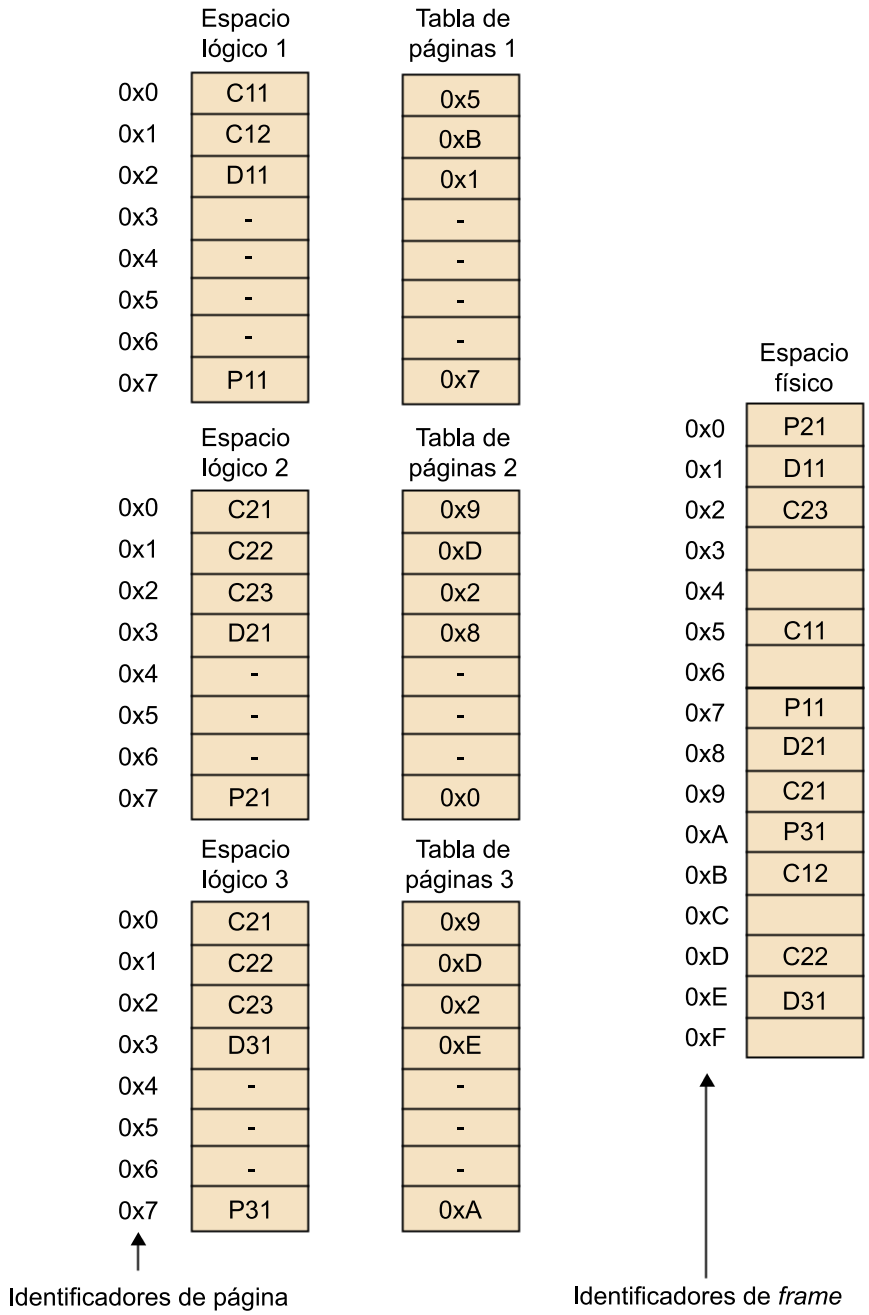

Figura 4. Ejemplo de carga de tres procesos en un sistema de gestión de la memoria paginado. Las páginas inválidas se han marcado con el símbolo -.

La figura muestra los espacios lógicos correspondientes a los tres procesos, así como una posible carga de los procesos en el espacio físico y las tablas de páginas correspondientes. Notad que la paginación permite que los dos procesos compartan las páginas de código.

3.1.2. ¿Evita la paginación las deficiencias del sistema presentado en el apartado 2?

Un mecanismo de gestión de la memoria basado en la paginación permite solucionar la mayoría de problemas descritos para el sistema anterior:

1) No es necesario que páginas contiguas en el espacio lógico tengan que serlo también en el espacio físico. Por lo tanto, la paginación no presenta el problema de la fragmentación externa; cuando se carga un ejecutable en la memoria, es posible utilizar cualquier *frame* libre de memoria física, independiente-

mente de si es contiguo o no con el resto de *frames*. En cambio, la paginación presenta otro problema, ya que la unidad mínima de asignación es la página, con lo cual las peticiones de memoria de tamaño no múltiple del tamaño de página provocarán un cierto desperdicio de memoria. Este problema se conoce como fragmentación interna y es inherente a todos los sistemas de almacenamiento (ya sea en memoria o en disco) que dividen el espacio en bloques de un tamaño fijo.

Fragmentación interna

El problema de la fragmentación interna también aparece en situaciones cotidianas, como las reservas de mesas en los restaurantes. Típicamente, en una mesa no se mezclan comensales correspondientes a reservas diferentes (es decir, la mesa es la unidad mínima de asignación de espacio). Por lo tanto, aunque el restaurante puede tener sillas libres, éstas no se pueden asignar a nuevas reservas si estas sillas libres pertenecen a mesas ya asignadas a otras reservas.

2) La paginación permite que el espacio lógico de un proceso pueda crecer y decrecer. Gracias al agujero producido al ubicar los datos y la pila a cada extremo del espacio lógico, disponemos de páginas lógicas para poder variar el tamaño tanto de la región de datos como de la pila sin desperdiciar memoria física.

3) La paginación permite que diversos procesos compartan *frames* del espacio físico. Sólo es necesario que las entradas correspondientes de las tablas de páginas de los procesos hagan referencia a estos *frames* compartidos.

En cambio, la paginación no permite cargar procesos que no quepan en el espacio físico.

Ved también

En el subapartado 3.2, veremos cómo podemos extender la paginación para poder cargar estos procesos.

3.2. Paginación bajo demanda

Hasta ahora, todos los esquemas que hemos presentado partían de la base de que, cuando un proceso se ejecuta, es necesario que esté cargado totalmente en la memoria principal, pero eso no tiene que ser necesariamente así. La memoria virtual es un esquema de gestión de la memoria con el que un proceso puede estar sólo parcialmente cargado (residente) en la memoria principal. Con este esquema, la suma de los espacios lógicos de los procesos que se están ejecutando puede ser mayor que la memoria física disponible.

El esquema también permite ejecutar procesos que tengan un espacio lógico mayor que el espacio físico, cosa que con los esquemas anteriores era imposible. Este hecho se consigue manteniendo una imagen del espacio lógico del proceso en la memoria secundaria (disco) en una zona llamada área de intercambio (*swap area*). En cada instante, en la memoria principal sólo está cargada la parte del proceso que es necesaria para su ejecución, o bien la parte del proceso que el sistema operativo permite cargar en la memoria. La decisión de qué parte del proceso se carga en la memoria y cuándo se carga la toma el sistema operativo y lo hace en función de la carga y la disponibilidad del sistema.

La memoria virtual se puede implementar extendiendo la idea de paginación (figura 5). Esta extensión recibe el nombre de **paginación bajo demanda**. Hay que añadir un bit (bit de presencia) a cada entrada de la tabla de páginas para indicar si la página en cuestión está cargada o no en la memoria física. En caso de que no lo esté, se producirá una excepción del tipo "fallo de página" (*page fault*) y la rutina de atención tendrá que ir a buscar la página al área de intercambio, cargarla en la memoria principal, hacer que se repita el acceso en la memoria que ha provocado el fallo y hacer continuar la ejecución del proceso.

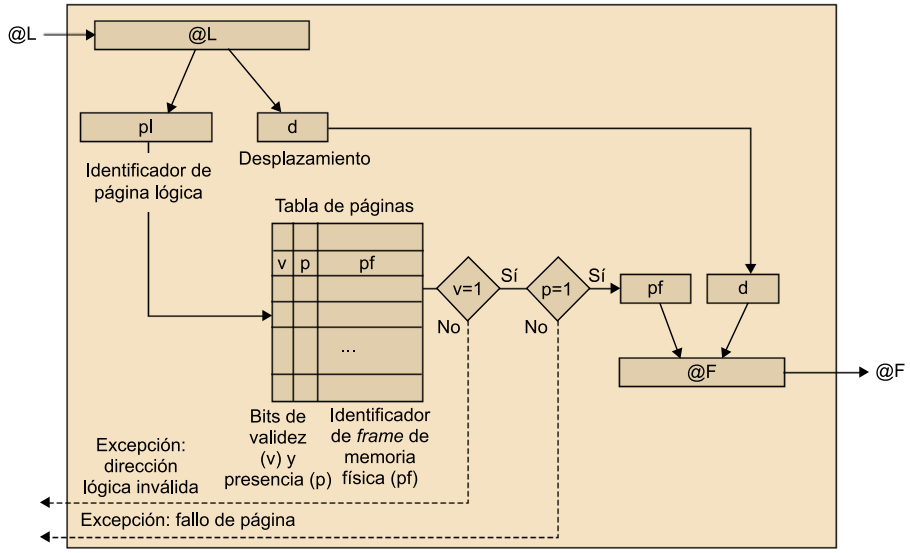

Figura 5. Extensión de la MMU de la figura 3 con el fin de permitir el uso de memoria virtual.

Además, es necesario que el sistema operativo disponga de algún tipo de estructura de datos que indique cómo está distribuido el proceso en el área de intercambio.

La gestión más habitual es que en un principio no haya ninguna página cargada en la memoria física y que las páginas del proceso se carguen a medida que se van necesitando. Cuando se ejecuta un proceso, cada referencia a la memoria lógica supone la ejecución de los pasos siguientes:

- Primero, se comprueba si la página correspondiente a la dirección es válida con el bit de validez (v).

- A continuación, se determina si la página está cargada en la memoria comprobando el bit de presencia (p). En caso de que no se encuentre en la memoria, se genera un fallo de página y el sistema operativo tiene que hacer las gestiones necesarias con el fin de ir a buscar la página en el área de intercambio y cargarla en la memoria principal en un *frame* libre. La cosa se complica todavía más si no queda ningún *frame* libre, ya que entonces el sistema de gestión de la memoria tiene que quitar alguna página de la memoria principal y llevarla al disco, siempre que haya modificaciones. Además, tiene que actualizar la tabla de páginas del proceso al que pertenece la página que se ha llevado al área de intercambio. Una vez el *frame*

ya ha quedado libre, se puede asignar al proceso que se estaba ejecutando, actualizando su tabla de páginas.

- Finalmente, si la página lógica es válida y presente, se genera la dirección física.

Los sistemas de memoria virtual tienen que gestionar las páginas de la manera más eficiente posible a fin de que el tiempo de ejecución de los procesos no sea excesivamente alto a causa de un número muy grande de fallos de página. Es importante establecer buenas políticas[8], como las siguientes:

(8)En esta asignatura, no entraremos en detalle a considerar estas políticas.

- **Políticas de asignación de páginas**: determinan cuántas páginas puede tener cargadas un proceso en la memoria física en un momento dado, es decir, determina lo que se llama *working set*.

- **Políticas de sustitución de páginas**: estas políticas determinan qué *frame* conviene llevar al disco cuando se tiene que llevar una página a la memoria física y no quedan *frames* libres. ¿Hay que escoger entre los *frames* del mismo proceso que se está ejecutando? ¿O bien hay que considerar los *frames* de otros procesos? ¿Páginas que sólo se hayan leído? (en este caso se tendría que considerar que en cada entrada de la tabla de páginas o de la tabla de *frames* hay un bit que indica que la página se ha modificado).

- **Políticas de carga de páginas**: determinan cuándo se cargan las páginas en la memoria. Existe la posibilidad de cargar páginas sólo cuando hay un fallo de página (paginación bajo demanda) o bien se puede intentar adelantar la carga de páginas aprovechando la localidad espacial de los programas (*prefetching*).

Consideraciones finales:

- Este mecanismo es totalmente transparente para el usuario.

- Se ha comprobado que las referencias a la memoria hechas por un programa no se distribuyen uniformemente por todas las páginas del espacio lógico. A causa de la localidad espacial y temporal de las referencias a la memoria, la mayoría de referencias se concentran en un conjunto reducido de páginas. Por lo tanto, la paginación bajo demanda no incrementa necesariamente el tiempo de ejecución del programa.

- La memoria virtual permite incrementar el grado de multiprogramación porque, como cada proceso necesita menos memoria física, será posible cargar más procesos. De todas formas, aumentar excesivamente este grado puede afectar al rendimiento del sistema operativo (*thrashing*).

- A lo largo de la ejecución de un proceso, una misma página se puede intercambiar del disco a la memoria y de la memoria al disco varias veces.

Notad que cada vez que vuelve a la memoria física puede utilizar un *frame* diferente de memoria física (el sistema de gestión permite movilidad).

4. Modificación dinámica del espacio lógico

En tiempo de ejecución, los procesos pueden tener que modificar el tamaño de su espacio lógico. Ahora bien, la forma de hacerlo depende de la región (código/datos/pila) de la que se quiera variar el tamaño.

1) Región de código: Existen diversas posibilidades para modificar dinámicamente la región de código del espacio lógico de un proceso, como *overlays*, código automodificable o bibliotecas cargadas en tiempo de ejecución[9]. Salvo las bibliotecas cargadas en tiempo de ejecución, estas opciones están en desuso o se utilizan únicamente en escenarios muy concretos. En esta asignatura, no vamos a considerar ninguna de estas opciones.

(9) En los sistemas Windows, también reciben el nombre de DLL (*dynamic-link libraries*).

2) Región de pila de ejecución: La pila es transparente para el programador de aplicaciones de alto nivel. Al crear cada proceso, el sistema operativo le asigna un tamaño inicial de pila. Si en algún momento este tamaño resulta insuficiente (el proceso ejecuta una instrucción `push` que intenta escribir más allá del tamaño asignado a la pila), la MMU generará una excepción. La rutina de atención a la excepción comprobará si el problema es debido al tamaño de la pila y, a ser posible[10], aumentará el tamaño y permitirá que el proceso repita la ejecución de la instrucción `push` y continúe su ejecución. Por lo tanto, el crecimiento de la región de pila es completamente transparente para el programador y tampoco nos ocuparemos de él.

(10) El sistema operativo puede limitar el tamaño máximo que puede llegar a alcanzar la pila de ejecución de los procesos.

3) Región de datos: el tamaño de la región de datos es responsabilidad del programador. Éste dispondrá de llamadas al sistema con el fin de poder modificar dinámicamente esta región del espacio lógico. Eso puede resultar útil porque el tamaño de algunas estructuras de datos puede depender de un valor desconocido en tiempo de compilación y que no se conocerá hasta el tiempo de ejecución del programa. En esta asignatura, nos centraremos en cómo modificar dinámicamente el tamaño de esta región.

4.1. Modificación dinámica del tamaño de la región de datos

El sistema operativo acostumbra a ofrecer llamadas al sistema para aumentar y disminuir la región de datos del espacio lógico del proceso.

En el caso del Unix, ambas tareas se pueden hacer con la llamada al sistema brk. Desgraciadamente, esta llamada tiene una interfaz poco intuitiva y presenta algunos problemas de portabilidad. Por lo tanto, no es habitual que los programas utilicen directamente esta llamada al sistema.

La biblioteca del lenguaje C ofrece diversas rutinas para modificar el tamaño de la región de datos: calloc, malloc, free y realloc. Las rutinas calloc, malloc y realloc permiten ampliar la región de datos, pero difieren en la interfaz. La rutina free permite liberar la memoria obtenida previamente con calloc/malloc/realloc.

Por ejemplo, la interfaz de la rutina malloc es void *malloc(size_t size) y recibe como parámetro la cantidad de bytes en los que queremos ampliar la región de datos. Si no es posible realizar esta operación, la rutina devuelve el puntero NULL; de lo contrario, devuelve la dirección lógica de la memoria correspondiente al primer byte que se ha añadido en la región de datos y el resto de bytes se encuentran en posiciones de memoria consecutivas. La interfaz de la rutina free es void free(void *ptr) y recibe como parámetro una dirección de memoria obtenida previamente con `calloc/malloc/realloc`.

A continuación, mostramos un par de fragmentos de código en los que se crean dinámicamente un vector (figura 6) y una matriz (figura 7).

```
int *vector;
vector = malloc(N*sizeof(int)); /* Pide espacio para N enteros */
if (vector == NULL) error("Memoria no disponible");
for (i=0; i<N; i++)
     vector[i] =. ...; /* Llena el vector */
...
free(vector);    /* Libera el espacio */
```

Figura 6. Fragmento de código que crea dinámicamente un vector de enteros de N posiciones.

```
int **matrix;

/* Pide espacio para un vector de N punteros, uno para cada fila */
matrix = malloc(N*sizeof(int *));
if (matrix == NULL) error("Memoria no disponible");
for (i=0; i<N; i++) {
   /* Pide espacio para la fila i-ésima (M enteros) */
   matrix[i] = malloc(M*sizeof(int));
   if (matrix[i] == NULL) error("Memoria no disponible");
}
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    matrix[i][j] = ...;  /* Llena la matriz */
...
for (i=0; i<N; i++)
   free(matrix[i]);     /* Libera la fila i-ésima */
free(matrix);       /* Libera el vector de punteros
```

Figura 7. Fragmento de código que crea dinámicamente una matriz de enteros de N filas y M columnas.

5. Errores de programación relacionados con el acceso a la memoria

Finalmente, haremos algunas consideraciones sobre los errores de programación relacionados con el acceso a la memoria, es decir, los errores de programación que provocan que un proceso acceda a una dirección de memoria a la que no tendría que acceder. Son especialmente típicos de programas escritos en lenguaje C o C++. La figura 8 muestra algunos ejemplos de estos errores.

Figura 8. Ejemplo de algunos errores de programación relacionados con el acceso a la memoria

```
int vector[10]; unsigned int n; char s[10]; int *k; char *p;

/* Error de programación si n >= 10 */
vector[n] = valor;

/* Error de programación porque la memoria intermedia s es demasiado pequeña (buffer overflow) */
sprintf(s, "El valor de vector[3] es %d\n", vector[3]);

/* Error de programación porque no tiene sentido referenciar variables locales de un
procedimiento una vez éste ha acabado */
int *rut()
{
 int n;
 n = valor;
 return(&n);
}
...
k = rut();
n = *k;

/* Error de programación porque se accede a memoria liberada */
p = malloc(100);
...
free(p);
p[4] = valor;
```

En general, el compilador de lenguaje C no puede detectar estos tipos de errores. Además, pueden ser muy difíciles de localizar por el programador porque:

- Es posible que, para la mayoría de datos de entrada, estos errores no afecten al comportamiento correcto del programa.

- Pueden manifestarse muchas sentencias más allá de la que ha realizado el acceso erróneo.

- Pueden manifestarse (o no) en función de las versiones del sistema operativo/compilador/bibliotecas y demás, así como de los parámetros de compilación o de los datos de entrada del proceso, entre otros.

En los apartados anteriores, hemos visto que la MMU detecta accesos a la memoria inválidos, pero tenemos que tener en cuenta que:

- La MMU no podrá detectar un error de programación que provoque el acceso a otra dirección válida del proceso. Por lo tanto, el proceso puede leer o modificar datos que no tienen nada que ver con la operación que se está realizando. Por ejemplo, un acceso a la memoria incorrecto puede estar modificando datos de control de la biblioteca de entrada/salida[11], cosa que puede provocar un error de ejecución la siguiente vez que utilicemos una rutina de esta biblioteca.

(11)Esta biblioteca guarda información en el espacio lógico de cada usuario que la utiliza.

- En un sistema basado en la paginación, si el tamaño de las regiones de código/datos/pila no es múltiplo del tamaño de página, podemos tener páginas marcadas como válidas en las que no todas las direcciones de estas páginas son correctas desde el punto de vista del programador. La MMU no podrá detectar un acceso incorrecto dentro de esta página.

Los accesos incorrectos pueden producirse en cualquier modo de ejecución (privilegiado y no privilegiado), pero el tratamiento es diferente.

1) Modo de ejecución no privilegiado: si la MMU determina que un acceso es inválido, la MMU provocará una excepción. La rutina del sistema operativo que atiende esta excepción hace abortar el proceso. En el caso del Unix, la rutina de atención acostumbra a crear un fichero[12] llamado `core` en el que se almacena el estado completo del proceso (contenido de los registros y de la memoria) en el momento que se provoca la excepción. Este fichero se podrá examinar posteriormente con el depurador (*debugger*). Cuando un proceso creado desde el intérprete de comandos ha sido abortado a causa de un acceso a memoria inválido, los intérpretes de comandos del Unix suelen mostrar el mensaje *Segmentation fault. Core dumped* (error de segmentación).

(12)El usuario puede indicar si quiere que se generen estos ficheros y qué tamaño máximo pueden tener.

2) Modo de ejecución privilegiado: siempre que alguna llamada al sistema tiene que acceder a algún rango de direcciones lógicas del proceso[13], antes el sistema operativo comprueba por software que este rango de direcciones sea válido para el proceso. El código del sistema operativo accederá a estas direcciones únicamente si la comprobación indica que todo el rango de direcciones es válido; ahora bien, aunque el rango de direcciones sea válido, no quiere decir que el acceso a la memoria sea correcto desde el punto de vista del programador. En el caso de que alguna dirección del rango sea inválida, la llamada al

(13)Por ejemplo, si un proceso invoca la llamada al sistema `write` del Unix, la rutina de atención tendrá que leer datos de un fichero y escribirlos en el espacio lógico del usuario, en el rango de direcciones especificado por una dirección lógica inicial y un tamaño.

sistema no se realizará y devolverá el código de error adecuado. Si, a pesar de todo, la MMU detecta un acceso a la memoria inválido en modo de ejecución privilegiado, por precaución el sistema operativo suele dejar de prestar servicio y aparece un *Kernel panic* (Linux) o la *Blue screen of death* (Windows).

Para ayudar a los programadores a detectar o corregir estos tipos de errores, existen herramientas como:

- depuradores (*debuggers*), que permiten analizar los ficheros `core`,
- versiones sofisticadas de las bibliotecas de asignación de memoria dinámica.

Es importante que los programas no presenten estos tipos de errores. Hay que pensar que una cantidad significativa de ataques a sistemas informáticos aprovechan un tipo particular de estos errores (los *buffer overflows*).

Resumen

En este módulo didáctico, hemos visto la manera como el sistema operativo, con la ayuda del hardware, puede ofrecer una visión de la memoria idealizada a los procesos en ejecución. Mientras que la memoria física está limitada y es compartida por todos los procesos, la visión que tendrá cada proceso es que dispone de un espacio de direcciones no compartido con ningún otro proceso, independientemente de la ocupación actual de la memoria física y que puede tener un tamaño mayor que la memoria física instalada. El sistema operativo realiza esta tarea de forma transparente para los procesos utilizando mecanismos de traducción dinámicos (en tiempo de ejecución) implementados dentro de un hardware especializado llamado MMU (*Memory Management Unit*).

Actividades

1. Explicad la diferencia que existe entre fragmentación interna y fragmentación externa.

2. Se quiere reducir la fragmentación interna en un sistema de gestión de la memoria basado en la paginación reduciendo el tamaño de las páginas (por ejemplo, de 4 KB a 1 KB). ¿Qué impacto tiene esta reducción en el tamaño de la tabla de páginas de los procesos?

3. En la figura 3 no se especifica el número de bits que circula por cada línea del circuito de traducción. Determinadlo asumiendo que el tamaño de página es de 4 KB, que el bus de direcciones es de 32 bits y que la cantidad máxima de memoria física instalada es de 2 GB.

4. El operador & del lenguaje C se aplica a una variable y nos devuelve la dirección de memoria donde se almacena. ¿Esta dirección es lógica o física?

5. Buscad información sobre la manera como Linux estructura el espacio lógico en los procesadores IA-32 de Intel. ¿Dónde ubica Linux las regiones de código, datos y pila de un proceso? ¿Por qué creéis que, a partir de la dirección lógica 0xC0000000, todos los espacios lógicos tienen el mismo contenido (código, datos y pila del sistema operativo)? ¿Por qué creéis que el código del programa no se empieza a cargar a partir de la dirección lógica 0x00000000?

6. Buscad información sobre la estructura de dos niveles de la tabla de páginas aplicada por los procesadores que implementan la arquitectura IA-32 de Intel.

7. Buscad información sobre cuáles son los bits de control presentes en las entradas de las tablas de página de los procesadores que implementan la arquitectura IA-32 de Intel.

8. En un sistema paginado con memoria virtual queremos controlar mediante un bit (*dirty bit*) si una página ha sido modificada. ¿En qué tabla situaríais este bit: en la tabla de páginas del proceso, en la tabla de *frames* o en la tabla de páginas del disco?

9. ¿En un sistema paginado podemos afirmar que la MMU detectará todos los accesos a la memoria en una posición de memoria incorrecta (por ejemplo, un acceso más allá del tamaño de un vector)?

10. Buscad información sobre los *stack overflows*. ¿En qué consisten? ¿Por qué son peligrosos? Buscad información sobre el *Execute-Disable (NX) bit* que algunos procesadores Intel incorporan en la tabla de páginas.

Ejercicios de autoevaluación

1. Disponemos de una máquina con un espacio lógico de procesador de 1.024 Kbytes y una memoria física instalada de 128 Kbytes. El sistema de gestión de la memoria del sistema operativo instalado en esta máquina se basa en la técnica de la paginación bajo demanda, en la que las páginas tienen un tamaño de 2 Kbytes. Inicialmente, el proceso no tiene ninguna página cargada en la memoria. Suponemos que en una página sólo habrá o bien código, o bien datos del proceso, o bien la pila de ejecución del proceso.

Nota

Por *espacio lógico de procesador* entendemos el tamaño máximo que puede llegar a tener el espacio lógico de un proceso.

En esta máquina queremos ejecutar un programa. La cabecera del ejecutable de este programa nos indica que el código del programa ocupa 10 Kbytes, que tenemos 6 Kbytes de datos inicializados, que los datos no inicializados ocuparán 15 Kbytes y que la pila tiene 12 Kbytes. La cabecera del ejecutable ocupa 128 bytes. Con esta información, contestad a las preguntas siguientes:

a) ¿Podéis decir aproximadamente el tamaño que tendrá el fichero ejecutable del programa que hemos descrito? Si el tamaño de las páginas de la memoria fuera de 1 Kbyte, ¿cuál sería el tamaño del fichero ejecutable?

b) Una vez cargado este programa y creado el espacio lógico del proceso correspondiente y suponiendo que todas las páginas que ocupa el proceso estuvieran presentes en la memoria física, ¿cuál sería la cantidad de memoria física que ocuparía nuestro proceso? Responded a la misma pregunta considerando que las páginas fueran de 1 Kbyte.

c) ¿Cuál es la fragmentación interna que produciría este proceso en caso de que las páginas fueran de 2 Kbytes? ¿Y si fueran de 1 Kbyte?

d) ¿Qué efecto tendría un cambio en la cantidad de memoria física instalada que tiene esta máquina sobre los espacios de direcciones siguientes: fichero ejecutable, espacio lógico del procesador y espacio lógico del proceso?

2. Suponemos que una máquina tiene un sistema de gestión de la memoria en el que la traducción de direcciones se efectúa dinámicamente (en ejecución) y que ofrece memoria virtual. Contestad a las preguntas siguientes justificando brevemente vuestra respuesta.

a) ¿Puede ser mayor el tamaño del espacio lógico de un proceso que la cantidad total de la memoria física instalada en la máquina?

b) ¿Qué efecto tendría un aumento de la cantidad de memoria física instalada sobre el tamaño del espacio lógico del procesador?

c) ¿Es posible que la misma dirección lógica de dos procesos diferentes sea traducida en la misma dirección física en un mismo momento?

d) ¿Es posible que una dirección lógica de un programa en ejecución sea traducida en direcciones físicas diferentes en diferentes instantes de la misma ejecución?

e) ¿Es necesario que la pila de ejecución de un proceso se almacene en direcciones lógicas consecutivas? ¿Y en direcciones físicas consecutivas?

f) Compilamos un mismo programa de dos maneras diferentes; en la primera, indicamos que el tamaño inicial de la pila será de 10 K y en la segunda indicamos que será de 20 K. ¿Cuál creéis que será la diferencia de tamaño entre los dos ficheros ejecutables generados?

g) ¿Permite ampliar el grado de multiprogramación el hecho de disponer de la memoria virtual?

3. Contestad a las preguntas siguientes:

a) ¿Puede el espacio lógico del procesador ser mayor que la suma de los tamaños de la memoria física instalada y del dispositivo de almacenamiento (área de intercambio) de páginas sacadas de la memoria física?

b) ¿Puede el espacio lógico del proceso ser mayor que la suma de los tamaños de la memoria física instalada y del tamaño del dispositivo de almacenamiento (disco) de páginas sacadas de la memoria física?

c) ¿Tiene cualquier proceso de usuario partes de su espacio lógico de proceso que tienen que encontrarse permanentemente en la memoria física durante la ejecución del proceso?

d) Si durante la ejecución concurrente de dos procesos ambos intentan acceder a la misma dirección lógica, ¿quiere decir que están compartiendo la memoria?

Solucionario

Ejercicios de autoevaluación

1.a) Un ejecutable está formado por el código y los datos inicializados. La pila y los datos no inicializados se crean en tiempo de ejecución. Así pues, el tamaño del ejecutable es igual al tamaño de la cabecera + el tamaño del código + el tamaño de los datos inicializados (128 bytes + 10 Kbytes + 6 Kbytes). En el caso de páginas de memoria de 1 Kbyte, el tamaño del ejecutable sería el mismo porque es independiente del tamaño de la página.

b) Con páginas de 2 Kbytes, el tamaño de la memoria física ocupado sería igual en 22 páginas (44 Kbytes) y, con páginas de 1 Kbyte, el tamaño sería igual a 43 páginas (43 Kbytes).

c) Con páginas de 2 Kbytes, tendríamos una fragmentación interna de 1 Kbyte y con páginas de 1 Kbyte no habría fragmentación interna.

d) No les afectaría de ningún modo.

2.a) Sí, ya que el sistema de gestión de la memoria permite ejecutar procesos sin que todo el espacio lógico esté en la memoria física simultáneamente.

b) Ninguno, ya que el espacio lógico del procesador está determinado por los bits de dirección de memoria de los que dispone el procesador.

c) Sí, en el caso de que los dos procesos estén compartiendo código o datos.

d) Sí, ya que el sistema de gestión permite que una página se pueda mover dentro de la memoria física.

e) Es necesario que se encuentre en direcciones lógicas consecutivas, pero no hace falta que esté en direcciones físicas consecutivas.

f) Los ejecutables tendrán el mismo tamaño, ya que en el ejecutable se guarda información del tamaño inicial de la pila, pero no se reserva espacio.

g) Sí, ya que en un sistema no entero no hay que tener todo el espacio lógico de los procesos simultáneamente en la memoria física y, por lo tanto, permite que entren más.

3.a) Puede ser mayor porque el espacio lógico del procesador sólo depende del número de bits que tiene el bus de direcciones del procesador.

b) El espacio lógico de un proceso no puede ser mayor que la suma de los tamaños de estas dos zonas.

c) Un sistema de gestión de la memoria que sea no entero permite que todo el espacio lógico de un proceso (código, datos y pila) esté en el disco durante la ejecución. A medida que se van referenciando páginas, la rutina de atención en el fallo de página ya las irá llevando del disco a la memoria física. Las estructuras de datos del sistema operativo (tablas de páginas y otras informaciones relativas al proceso) no forman parte del espacio lógico de proceso.

d) No necesariamente, porque procesos diferentes tienen programaciones de gestores de la memoria diferentes; por lo tanto, los dos pueden generar la misma dirección lógica, pero la traducción dinámica puede hacer que esta dirección lógica vaya a parar a direcciones físicas diferentes.

Glosario

bit de presencia *m* Bit utilizado en los sistemas de gestión de la memoria virtual para indicar si el fragmento del espacio lógico del proceso al que se hace referencia está cargado en la memoria.

bit de validez *m* Bit utilizado en los sistemas de gestión de la memoria para indicar si el fragmento del espacio lógico al que se hace referencia pertenece al espacio lógico del proceso.

fallo de página *f* Excepción generada por la MMU en un sistema de gestión de la memoria que ofrece memoria virtual cuando el procesador genera una dirección lógica que se traduce en una página válida, pero que no está cargada (no está presente) en la memoria física.

fragmentación externa *f* Espacios de memoria no utilizables que se crean en sistemas de gestión de la memoria en los que los procesos se cargan en posiciones de la memoria física contiguas. Tiene lugar cuando no hay un espacio de memoria contiguo libre lo bastante grande para cargar un proceso, aunque haya trozos libres más pequeños cuya suma de las capacidades sería suficiente para cargarlo.

fragmentación interna *f* Espacios de memoria no utilizables que se crean cuando la memoria se divide en partes de tamaño fijo (por ejemplo, *frames* de memoria principal o bloques de disco) y una de estas partes se asigna a un conjunto de datos más pequeño que la capacidad de la parte en cuestión.

hiperpaginación *m* Situación que se da cuando la memoria disponible en un momento dado para la ejecución de un proceso es inferior a la mínima necesaria (*working set*) para su ejecución, lo que provoca una elevada actividad de intercambio. Un sistema se encuentra en *thrashing* si se pasa más tiempo intercambiando páginas que ejecutando código de las aplicaciones. Una causa habitual es que el grado de multiprogramación es excesivamente elevado para la configuración del hardware.
en *thrashing*

intercambio *m* Acción de llevar páginas de un proceso de la memoria secundaria (disco) a la memoria principal o *swap in*. El proceso inverso, es decir, el hecho de llevar páginas de un proceso desde la memoria principal hasta la secundaria, se llama *swap out*.
en *swapping*

memoria virtual *f* Sistema de gestión de la memoria que permite ejecutar un proceso a pesar de tener sólo parte del proceso cargada en la memoria. Mediante técnicas como la paginación bajo demanda, el sistema irá cargando otras partes del proceso a medida que se necesiten.

paginación *f* Sistema de gestión de la memoria que divide el espacio lógico y el espacio físico en bloques de tamaño fijo y asigna a cada bloque del espacio lógico (página) un bloque del espacio físico (*frame*).

Bibliografía

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

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