FSO - ejercicios de gestión de memoria

Esta es una lista de escenarios de uso de memoria no contigua (segmentada o paginada), en la que te planteamos algunos ejercicios que manejan los espacios de direcciones lógico y físico y la arquitectura de traducción de direcciones. Los objetivos que pretendemos con estos ejercicios son ayudarte a:

1. Tomar consciencia de las diferencias entre las direcciones físicas y lógicas.
2. Entender la organización de un sistema de paginación jerárquica.
3. Conocer el impacto de la utilización de una TLB en el tiempo de acceso a memoria.

Caso 1: direcciones segmentadas

Este es un problema básico para entender las consecuencias de la organización de las direcciones de memoria (anchura de bits, cantidad de bits dedicados a segmento y a desplazamiento).

Tenemos una CPU que trabaja con segmentación. Las direcciones lógicas son de 16 bits. Los tres primeros bits se utilizan como selector de segmento.

- ¿Cuánta memoria puede direccionar un proceso en este sistema?
- ¿Cuántos segmentos puede llegar a tener un proceso?
- ¿Cuál es el tamaño máximo de un segmento?

¿Cómo variarían las respuestas si se dedicaran 2 bits para el selector de segmento?

Caso 2: espacios físico y lógico diferentes

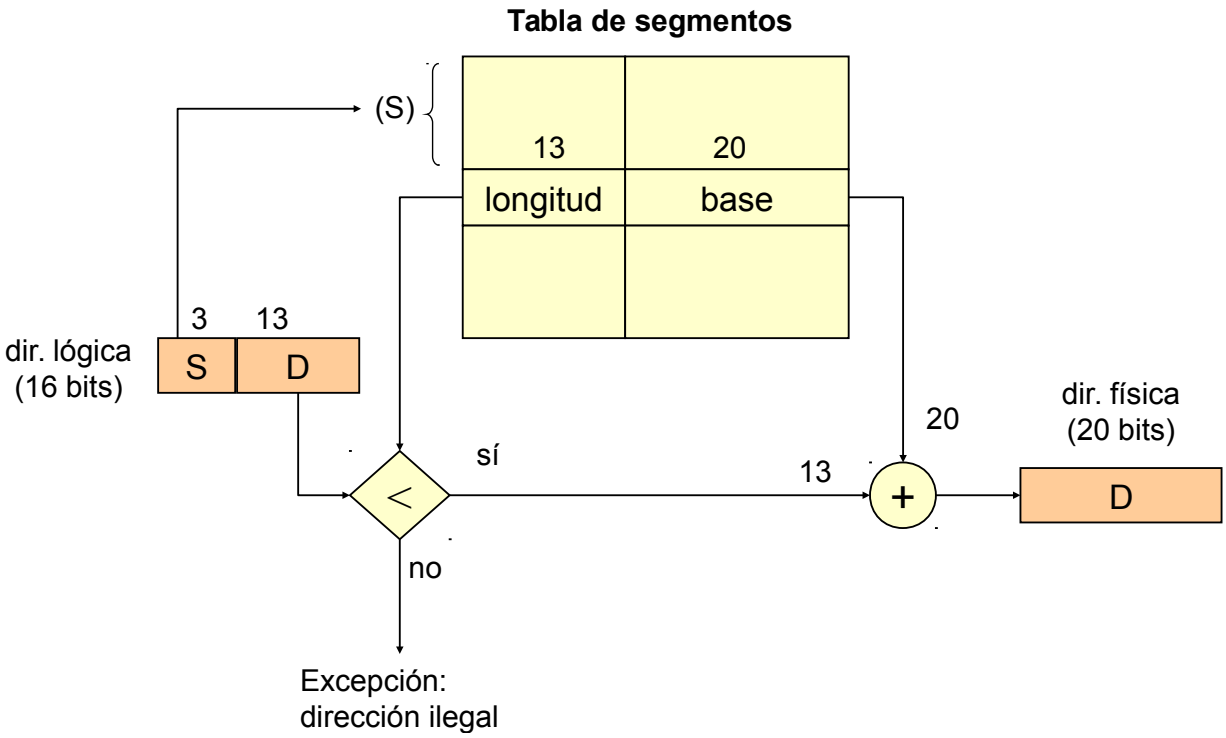
Una cuestión muy importante para entender el hardware de gestión de memoria es que el **espacio lógico** y el **espacio físico** de direcciones pueden tener tamaños diferentes. *NOTA: este tópico no está lo bastante resaltado en la bibliografía, así que lo describimos con un ejemplo a continuación, para que te sirva de ilustración.*

Supongamos la máquina descrita en el Caso 1. Maneja direcciones lógicas de 16 bits y por tanto un proceso dispone de un espacio lógico de hasta 64Ki bytes. Sin embargo, el espacio físico puede ser diferente: supongamos que en esta máquina, el bus que accede a la memoria física es de 20 bits y por ello las direcciones físicas son de 20 bits. Eso significa que el hardware puede manejar una memoria física de hasta 1MiB, pero un proceso sólo puede manejar 64KiB.

¿Tiene esto sentido? Pues sí: históricamente ha ocurrido bastantes veces que las tecnologías de RAM se abaratan y permiten más capacidad (ej. pasar de decenas de KiB a centenares o miles de KiB), pero la arquitectura de la CPU no se puede cambiar tan fácilmente, porque hay mucho código ya escrito y no se pueden estar reescribiendo/recompilando/probando todas las aplicaciones. Por

eso, es imaginable un escenario en el que la CPU maneja un espacio lógico más pequeño que el espacio físico (ej. 16 bits contra 20 bits).

Si es así, ¿cómo podemos aprovechar que hay más memoria física, sin alterar la arquitectura de las direcciones lógicas? En nuestro sistema segmentado, las direcciones base de los segmentos podrían ser direcciones físicas de 20 bits, de manera que un segmento pueda estar ubicado en cualquier punto de la memoria física. En la figura puede verse cómo quedaría el circuito de traducción de direcciones. Los números indican la anchura en bits de cada dato.



Con el esquema descrito en la figura, un proceso sigue limitado a un espacio lógico de 64KiB, pero sus segmentos pueden residir en cualquier punto de la memoria física de 1MiB. Si nuestro sistema operativo es multiprogramado, cada proceso puede tener su espacio en zonas diferentes de la memoria, consumiendo en total mucho más de los 64KiB que tenía el sistema de 16 bits físicos. Así el sistema operativo le puede sacar partido a toda la memoria física.

Otra técnica para aprovechar el espacio físico consiste en que un proceso disponga de un servicio para trabajar con varios «bancos de memoria» dinámicos. Supongamos que desde el sistema operativo le damos al proceso 10 zonas de memoria, cada una de 8K. El proceso no puede direccionar todas esas zonas a la vez, ya que suman 80K, ni tampoco se las podemos colocar en la tabla de segmentos, que sólo tiene 8 entradas. Pero sí se podría disponer de un servicio para utilizar en cada momento una sola de las 10 zonas. Cada vez que el proceso quisiera utilizar una zona, se lo pediría al SO y este colocaría la dirección base de la zona deseada en una entrada «E» de la tabla de segmentos. Las direcciones lógicas generadas sobre el segmento «E» irían a esa zona

de memoria física. Si más tarde el proceso quiere utilizar otra zona de memoria, cambiaría el valor de la dirección base. En cada momento el proceso dispondría de un espacio lógico de 64KiB, pero a lo largo de su ejecución podría acceder a un espacio físico mucho mayor.

Esta diferencia de tamaños entre el espacio lógico y el espacio físico puede ocurrir a la inversa: sin ir más lejos, los actuales procesadores x86 manejan un espacio lógico de 64 bits, pero utilizan direcciones físicas de 48 bits. Esto se debe a que actualmente todavía no es rentable fabricar procesadores con buses físicos de 64 bits.

Caso 3: direcciones paginadas

Supongamos un sistema que utiliza memoria paginada. Las direcciones lógicas son de 24 bits. Si se dedican 10 bits para seleccionar la página y los restantes para el desplazamiento dentro de la página:

- ¿Cuántas páginas puede llegar a tener un proceso?
- ¿Qué tamaño de página utiliza el sistema?
- ¿Cuánta memoria puede direccionar un proceso?

Pongamos otro caso similar. Esta vez tenemos direcciones lógicas de 24 bits y sabemos que el tamaño de página son 4KiB. Entonces:

- ¿Cuántas páginas puede llegar a tener un proceso?
- ¿Cuántos bits hay que dedicar a página y a desplazamiento?
- ¿Cuánta memoria puede direccionar un proceso?

Otra cuestión interesante es estudiar la influencia entre el tamaño de página y el espacio ocupado por la **tabla de páginas**. ¿Cuál de los dos escenarios genera tablas de páginas más grandes?

Caso 4: espacios lógico y físico diferentes (paginación)

La situación del sistema segmentado del Caso 2 se puede dar en un sistema de memoria paginada. Por ejemplo, podríamos tener un sistema con direcciones lógicas de 24 bits que maneje direcciones físicas de 32 bits. De forma análoga al Caso 2, la estructura de traducción (la tabla de páginas) contendría números de marco *físicos* (con más bits que los lógicos) para poder ubicar las páginas lógicas del proceso en un espacio más grande. El tamaño de página es el mismo en las direcciones lógicas y físicas, porque la página es precisamente la unidad de asignación de espacio físico.

Supongamos entonces unas direcciones lógicas de 24 bits, direcciones físicas de 32 bits y un tamaño de página de 4KiB. En este caso:

- ¿Cuánta memoria puede direccionar un proceso?
- ¿Cuánto espacio físico puede direccionarse?
- ¿Cuántos marcos de página pueden llegar a existir en la memoria?
- ¿Cuántos bits dedicados a página y desplazamiento habría en una dirección física?
- ¿Cuántas entradas puede llegar a tener la tabla de páginas de un proceso?
- ¿Cuántos bits son necesarios para almacenar una entrada en la tabla de páginas?

Caso 5: tamaño de la tabla de páginas

En los sistemas comerciales se han manejado tamaños de páginas entre 256 bytes y 16 megabytes. Un tamaño típico y clásico son los 4KiB de los procesadores x86 de Intel, que vamos a tomar como caso para este ejercicio. En los x86 el tamaño de una entrada en la tabla de páginas es de 32 bits (4 bytes).

Bajo estas condiciones, supón que tenemos un proceso que tiene reservados 10 megabytes de memoria y calcula estos datos:

- ¿Cuántas páginas necesita este proceso?
- ¿Cuánto ocupa la tabla de páginas de este proceso?

Caso 6: paginación jerárquica en los Intel x86

En el supuesto del Caso 5, el tamaño de la tabla de páginas del proceso es *superior* al tamaño de una página. Esto origina problemas de gestión del espacio libre, si la tabla de páginas debe estar ubicada de forma contigua: habría que buscarle varios marcos libres consecutivos, con lo que estaríamos perdiendo la virtud de la paginación, que es no necesitar bloques contiguos de memoria. Este problema se acentúa cuanto más grande es el proceso y mayor es su tabla de páginas.

Una solución a este problema es la **paginación jerárquica**, que consiste en *paginar la tabla de páginas*. El Intel 80386 y demás máquinas de 32 bits utilizan un esquema de dos niveles de paginación, con un primer nivel llamado *directorio de páginas*, que apunta a los diferentes fragmentos de la tabla de páginas. Este es el modelo:

10 bits	10 bits	12 bits
1er nivel de paginación	2º nivel de paginación	desplazamiento

Las entradas de las tablas de páginas son siempre de 4 bytes (la de primer nivel y las del segundo nivel).

Sobre este modelo de paginación, contesta las siguientes preguntas:

- ¿Cuántas páginas puede llegar a tener un proceso en total?
- ¿Cuántas entradas puede llegar a tener el directorio de páginas de un proceso? *(la tabla de primer nivel)*
- ¿Qué tamaño en bytes puede alcanzar el directorio de primer nivel?
- ¿Cuántas entradas puede llegar a tener una tabla de páginas de segundo nivel?
- ¿Qué tamaño en bytes puede llegar a tener una tabla de segundo nivel?

Imagina un proceso que ocupe 100 megabytes. ¿Cuánto espacio consumirán sus tablas de páginas?

Una vez que has obtenido los resultados, podrás ver que los ingenieros de Intel seleccionaron esta organización del espacio lógico [10+10+12] de forma muy intencionada. Una forma de verlo es contestar las mismas preguntas anteriores, pero con una organización [12+8+12] o con una organización [8+12+12]. ¿Ves alguna ventaja del [10+10+12] de Intel sobre esas dos alternativas?

Caso 7: TLB y tiempo de acceso a la memoria

Un problema que tienen tanto la segmentación como la paginación es que penalizan fuertemente el tiempo de acceso a memoria, porque para traducir una dirección lógica hay que acceder a una tabla (de segmentos o de páginas). Esa tabla, en el caso de un sistema paginado, estará también en la RAM, con lo cual **se duplica** el tiempo de acceso efectivo a la memoria. Y si se utiliza un sistema de paginación jerárquica con varios niveles, el tiempo se multiplica aún más.

La solución que se ha dado a este problema es disponer de una caché dentro de la CPU con las entradas de la tabla de páginas que se estén utilizando con más frecuencia. Esta caché se llama **TLB** (*translation lookaside buffer*). Si se consiguen mantener en la TLB las entradas más probables, se pueden obtener *tasas de aciertos* superiores al 96-98%, con lo cual disminuye notablemente el tiempo efectivo de acceso a la memoria.

*NOTA: entendemos por **tiempo de acceso efectivo** a la memoria como el tiempo total que se invierte en completar un acceso, desde que la CPU genera una dirección lógica hasta que se realiza la operación en la RAM. O sea, se incluye el tiempo de traducción de la dirección más todos los accesos reales que hay que hacer a la RAM para realizar la operación.*

Para analizar el impacto del uso de una TLB, vamos a plantear este escenario: tenemos un sistema de gestión de memoria paginada, de un solo nivel, que utiliza una TLB. Se tienen los siguientes tiempos medios:

- Tiempo de acceso a la TLB: 4 nanosegundos.
- Tiempo de acceso a la RAM: 30 nanosegundos.
- Tasa de aciertos observada en la TLB: 98%.

¿Cuál sería el tiempo medio de acceso efectivo a memoria?

Ahora cambiemos el escenario y supongamos que tenemos un sistema de paginación jerárquica como el de los x86 (dos niveles de paginación). Si el resto de los valores se mantiene, ¿cuál sería ahora el tiempo medio de acceso efectivo a memoria?