SISTEMA DE MEMORIA

M. Isabel García Clemente
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid

Febrero de 2016

Índice

1. Introducción

La memoria constituye un componente fundamental del computador, ya que es el lugar donde se almacenan las instrucciones que ejecuta y los datos con los que éstas operan. La velocidad de ejecución de los programas y, por lo tanto, el rendimiento de un computador dependen fundamentalmente de la **velocidad** a la que se transfiere la información entre el procesador y la memoria. Por otra parte, los computadores requieren cada vez mayor **capacidad** de almacenamiento, tanto por las necesidades de las aplicaciones y sistemas operativos, como por el hecho de que, en la mayor parte de los casos, la memoria es un recurso compartido, pudiendo contener en un momento dado información correspondiente a varios programas con la finalidad de obtener el máximo rendimiento del procesador.

En la práctica, el sistema de memoria de todo computador está formado por una serie de dispositivos de almacenamiento de velocidad y capacidad diferentes, organizados de forma **jerárquica**, con el objeto de conseguir que el conjunto proporcione una capacidad casi ilimitada y alta velocidad a un coste razonable.

Dado que gran parte del tiempo de ejecución de los programas se emplea en acceder a memoria, conocer la composición y funcionamiento de la jerarquía de memorias del computador en el que se ejecutan es crucial para comprender su comportamiento y, por lo tanto, para poder conseguir que se ejecuten en el menor tiempo posible.

En este documento se tratan en primer lugar los principios básicos y el funcionamiento de la organización del sistema de memoria de un computador, que en adelante denominaremos **jerarquía de memorias**. Seguidamente se estudia en detalle la memoria caché, que constituye el nivel más rápido de la jerarquía y, por lo tanto, el más cercano a la CPU, para pasar después a tratar la organización de la memoria principal. Finalmente se analiza el mecanismo de memoria virtual desde el punto de vista de la arquitectura, y en menor medida desde el punto de vista del sistema operativo, y se describe funcionamiento de un computador con memoria caché y memoria virtual.

2. Jerarquía de Memorias

La organización general de la jerarquía de memorias de un computador es la que se muestra en la figura 1, en la que se han incluido los niveles más comunes presentes en los computadores actuales. La tabla 1 recoge valores típicos de algunos de los parámetros característicos de los dispositivos que la componen.

El nivel más alto (nivel 0) está formado por los **registros** del procesador. El siguiente nivel lo constituye la **memoria caché**, construida con tecnología RAM estática y con velocidad cercana a la del procesador. La **memoria principal**, construida generalmente con tecnología RAM dinámica, constituye el nivel 2 de la jerarquía. Su capacidad es mayor que la de la memoria caché pero su velocidad es bastante menor. Por último, están los **discos magnéticos**, dispositivos más lentos, con mayor capacidad de almacenamiento y menor coste que los anteriores, tradicionalmente denominados memoria auxiliar o secundaria. En la actualidad se emplean también otro tipo de discos, denominados discos de estado sólido (SSD), que en algunos casos sustituyen a los discos magnéticos convencionales (p.e. en computadores y dispositivos portátiles), ya que son más rápidos que estos últimos, aunque de mayor coste.

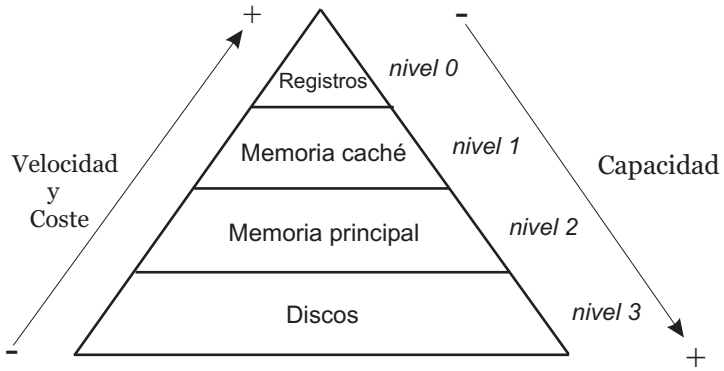


Figura 1: Organización jerárquica del sistema de memoria.

Dispositivo	**T. acceso**	**Capacidad (en bytes)**
Registros	0,25 - 0,5 ns	< 1 KB
Memorias caché	0,5 - 10 ns	8 KB - 32 MB
Memoria principal	20 - 60 ns	8 GB - 64 GB
Disco SSD	20 - 50 μs	120 GB - 1 TB
Disco magnético	5 - 15 ms	500 GB - 2 TB

Tabla 1: Parámetros característicos de los dispositivos de la jerarquía de memorias.

Otros dispositivos que se pueden englobar también bajo el nombre de memoria secundaria, y que no se tratan aquí, son las cintas magnéticas y los discos ópticos, que corresponden a niveles que estarían debajo del nivel del disco magnético, utilizados generalmente para realizar copias de seguridad la información.

Partiendo de los niveles anteriores, los computadores actuales disponen además de varios niveles de memoria caché (véase la sección 3.7.3). El tiempo y capacidad indicados en la tabla 1 para las memorias caché se refieren de hecho a los distintos niveles.

Hay que hacer notar que algunos de los niveles de la jerarquía, como la memoria principal o los discos, en general son configurables, mientras que otros, como los registros o las memorias caché, son fijos para una determinada implementación de una arquitectura. Asimismo, cada nivel es gestionado por un componente, *hardware* o *software*, del computador; los registros los gestiona el compilador, que los asigna a las variables más utilizadas dentro de un programa, la memoria caché la gestiona un *hardware* específico (el controlador de la caché), y la memoria principal el sistema operativo.

En lo sucesivo nos centraremos en el estudio de los niveles de memoria caché y memoria principal, haciendo mención los discos desde el punto de vista de su utilización como soporte para la realización del mecanismo de **memoria virtual**, que se trata en la sección 5. No consideraremos el nivel de los registros, cuyo estudio está relacionado con el procesador y su juego de instrucciones más que con la jerarquía de memorias.

2.1. Funcionamiento de la jerarquía de memorias

La jerarquía de memorias está organizada habitualmente de forma que la memoria de nivel **i** contiene un subconjunto de la información residente en la de nivel **i+1**, característica que se suele denominar **propiedad de inclusión**. Sin embargo, esta información no tiene por qué ocupar las mismas posiciones en ambos niveles, por lo que es necesario realizar una **traducción de direcciones** a la hora de gestionar la interrelación entre niveles adyacentes.

El funcionamiento básico de la jerarquía de memorias es el siguiente:

- La CPU envía la petición de acceso a la información al nivel más rápido de la jerarquía, la memoria caché. Si la información se encuentra en este nivel (se dice que se ha producido un **acierto**), se completa el acceso a la máxima velocidad que puede proporcionar el sistema de memoria.
- Si la información no está en la caché (se dice que que ha producido un **fallo**), se envía la petición al siguiente nivel (la memoria principal) y, si se encuentra en él, se transfiere al nivel inmediatamente superior (la caché). Esto puede suponer el **reemplazo** de alguna información de la caché, preferentemente la de menor utilidad.
- Si la información tampoco se encuentra en la memoria principal se busca en el siguiente nivel, procediéndose de forma análoga a la expuesta en el caso anterior.

De este modo, la información va ascendiendo hasta el nivel de la jerarquía más cercano a la CPU a medida que ésta la va necesitando, permanece allí durante el intervalo de tiempo en que la utiliza y, cuando deja de ser útil, se reemplaza por información más necesaria.

Pondremos como ejemplo un caso típico que ilustra este comportamiento: la ejecución de las instrucciones correspondientes a un bucle que inicialmente no se encuentran en la caché, pero sí en memoria principal. En este caso, durante la ejecución de la primera iteración se irán llevando a la caché las instrucciones del bucle, hasta tener allí todas ellas, si hay espacio suficiente. En las siguientes iteraciones se accederá por lo tanto a información que está en la caché, por lo que el tiempo de acceso será el correspondiente al dispositivo más rápido de la jerarquía. Por último, al acabar la ejecución de la última iteración del bucle y pasar a ejecutar otra parte del programa, la información utilizada en el bucle irá siendo reemplazada en algún momento por información correspondiente a las nuevas instrucciones.

Hay que hacer notar que la información se transfiere sólo entre dos **niveles adyacentes** y que además no se transfieren palabras individuales sino **bloques** de palabras consecutivas. Como se verá más adelante, el tamaño de estos bloques es distinto dependiendo de los niveles de que se trate.

El diseño de la organización jerárquica del sistema de memoria necesita dar respuesta a algunas cuestiones como la definición de las **políticas de extracción, ubicación y reemplazo**, todas ellas relativas al movimiento de información entre dos niveles adyacentes. La primera decide cuándo y qué información se debe llevar a un determinado nivel, la segunda determina la zona que debe ocupar la información en el nivel al que va dirigida, y la tercera selecciona la información que debe abandonarlo, para dar cabida a la procedente del siguiente nivel.

Un efecto colateral de la propiedad de inclusión es el problema de la **coherencia**. Al poder existir varias copias de la misma información en niveles diferentes de la jerarquía, puede ocurrir que la información del nivel **i** esté más actualizada que la del nivel **i+1**. Esta situación podrá dar

lugar a problemas, si existen varios componentes del computador tratando de acceder a la misma información a través de un nivel diferente de la jerarquía. Para resolver este problema, es necesario definir una **política de escritura** que determine el instante en que se actualizan las copias de la misma información cuando el procesador realiza un acceso de escritura.

Más adelante trataremos con más profundidad todas estas políticas, así como las diversas soluciones que se han propuesto y utilizado para los distintos niveles de la jerarquía.

2.2. Proximidad de referencias

El objetivo principal de la organización jerárquica de memorias, que el tiempo de acceso se acerque al del dispositivo más rápido, es posible gracias a una propiedad de los programas denominada **proximidad de referencias** (*locality*). Esta propiedad refleja el hecho de que, durante un cierto intervalo de tiempo, se tiende a acceder a las mismas zonas de código y datos, de modo que existe una alta probabilidad de que información a la que se ha accedido en un determinado instante vuelva a ser utilizada en un instante próximo, o bien que se utilice información almacenada en posiciones de memoria cercanas a aquella. Por lo tanto, si se logra tener esta información en el nivel más rápido de la jerarquía, el sistema de memoria se comportará, durante el intervalo de tiempo considerado, como si estuviera formado únicamente por el dispositivo de memoria de mayor velocidad.

Para analizar en detalle esta propiedad se parte del concepto de **traza**, que se define como la secuencia de direcciones a las que hace referencia un programa a lo largo de su ejecución. Para ilustrar este concepto consideremos el siguiente fragmento de código en ensamblador, que corresponde a la suma de los 10 elementos de un vector almacenado a partir de la dirección 800:

```
      add r3, r0, 0
      add r1, r0, 10
      add r2, r0, 800
eti:  ld r4, r2, 0
      add r3, r3, r4
      add r2, r2, 4
      sub r1, r1, 1
      bnz $eti
```

Suponiendo que al comenzar su ejecución el contador de programa tiene el valor 160, que las instrucciones y los datos ocupan 4 bytes, y que el direccionamiento se realiza a nivel de byte, la traza de este fragmento de código será la siguiente:

```
160, 164, 168, 172, 800, 176, 180, 184, 188,
               172, 804, 176, 180, 184, 188,
               172, 808, 176, 180, 184, 188,
               ......................
               172, 836, 176, 180, 184, 188
```

En este ejemplo se puede observar cómo, en lo referente a las instrucciones, tras el acceso a la dirección 160 se accede a la dirección contigua 164, tras esta a la 168 y así sucesivamente. Además, en cada iteración se repite el acceso a las direcciones 172, 176, 180, 184 y 188. En cuanto a los datos, tras el primer acceso a la dirección 800, se accede a direcciones próximas, en este caso concreto a las direcciones contiguas 804, 808, 812 ..., 836.

Existen por lo tanto dos tipos de proximidad que coexisten normalmente, en mayor o menor medida, en la ejecución de todo programa:

- **Temporal**.- Cuando hace referencia a las mismas direcciones a las que se ha accedido en un pasado reciente. En el ejemplo anterior este tipo de proximidad se manifiesta en el acceso repetido a las instrucciones que componen el bucle (172, 176, 180, 184 y 188).
- **Espacial**.- Cuando se hace referencia a direcciones cercanas a las que se ha accedido en un pasado reciente. Un caso particular es la proximidad **Secuencial**, que se da cuando se accede a direcciones consecutivas. Este último es el caso del acceso secuencial a las instrucciones y a los datos del ejemplo anterior.

La proximidad temporal se debe fundamentalmente a la utilización de estructuras de control del tipo bucle en los programas, así como al acceso a datos o la llamada a funciones de forma repetida. La proximidad espacial se debe a la ejecución secuencial de instrucciones y al uso de estructuras de datos de tipo vector, cadenas de caracteres, etc.

La jerarquía de memorias se beneficia de la proximidad temporal, manteniendo en los niveles más altos las instrucciones y datos utilizados más recientemente, y de la propiedad espacial enviando a dichos niveles no sólo la información demandada por la CPU, sino el contenido de las direcciones cercanas a ella, esto es, un **bloque** de palabras consecutivas.

El conocimiento del comportamiento de los programas típicos ejecutados en un computador con respecto a estos tipos de proximidad se puede utilizar para estimar la eficiencia de una jerarquía de memorias a la hora de tomar decisiones en su diseño. Así, la proximidad espacial permite determinar el tamaño más adecuado del bloque a transferir entre niveles adyacentes, mientras que la temporal ayuda a determinar el número de bloques de cada nivel, y la secuencial permite distribuir las direcciones en dispositivos que permitan accesos concurrentes. Este último es el caso de la memoria entrelazada, que se trata en la sección 4.1.

2.3. Rendimiento de la jerarquía de memorias

Para evaluar la eficiencia de una jerarquía de memorias se utilizan las siguientes medidas:

- **Tasa de aciertos** o *hit ratio* **(Hr).**

  Permite medir la eficiencia de un determinado nivel de la jerarquía. La **tasa de aciertos** de un nivel **i** (Hr_i) es la fracción de los accesos a dicho nivel que producen acierto (número de aciertos dividido entre número de accesos). Depende de varios aspectos del dispositivo de memoria del nivel considerado, como son los siguientes:

  - El tamaño del bloque de información que se transfiere a este nivel desde el siguiente, cuando se produce un fallo.
  - La capacidad de esta memoria.
  - La política de reemplazo utilizada.

  Además, depende de un aspecto no relacionado con el dispositivo:

  - La traza del programa que se ejecute.

- **Tiempo de acceso efectivo** ($T_{efectivo}$).

  Permite conocer la eficiencia global de la jerarquía de memorias, por lo que tiene en cuenta todos los dispositivos que la componen. Ya que el tiempo de acceso no es el mismo para todas las referencias a memoria (depende de dónde se encuentre la información), la eficiencia de la jerarquía de memorias vendrá dada por el tiempo medio empleado en un acceso. Este tiempo se suele denominar **tiempo de acceso efectivo** o **tiempo medio de acceso** y depende fundamentalmente de:

  - El tiempo de acceso del dispositivo utilizado en cada nivel.
  - La tasa de aciertos de cada nivel.

  Pongamos como ejemplo una jerarquía de dos niveles, memoria caché (Mca) y memoria principal (Mp). El tiempo de acceso efectivo se puede calcular de la forma:

$$T_{efectivo} = Hr_{Mca} \times T_{acierto} + (1 - Hr_{Mca}) \times T_{fallo}$$

  o lo que es lo mismo, promediando los tiempos de todos los accesos a memoria:

$$T_{efectivo} = \frac{N^o{}_{aciertos} \times T_{acierto} + N^o{}_{fallos} \times T_{fallo}}{N^o{}_{accesos}}$$

  donde:

  - $T_{acierto}$ es el tiempo que se tarda en completar un acceso cuando la información se encuentra en la caché. Este tiempo incluye el necesario para determinar si el acceso es un acierto.
  - **(**$1 - Hr_{Mca}$**)** es la **tasa de fallos** de la Mca o *miss ratio*.
  - T_{fallo} es el tiempo que se tarda en completar un acceso cuando la información no se encuentra en la caché. Este tiempo incluye el tiempo empleado en detectar el fallo, así como el necesario para enviar el bloque en el que se encuentra la información demandada desde la memoria principal a la caché, por lo que depende fundamentalmente del tiempo de acceso de la primera.
  - $N^o{}_{aciertos}$ es el número de accesos con acierto en la caché.
  - $N^o{}_{fallos}$ es el número de accesos que fallan en la caché ($N^o{}_{accesos} - N^o{}_{aciertos}$) y, por lo tanto, van a la memoria principal.

Ejercicio 1.- *Calcule la tasa de aciertos de la Mca y el tiempo de acceso efectivo, en la ejecución del programa expuesto en la página 4, suponiendo que los bloques que se envían a la caché son de 16 bytes, por lo que cada uno puede albergar 4 instrucciones ó 4 datos, el tiempo de la caché en el caso de acierto es el equivalente a 2 ciclos de reloj, y el tiempo adicional empleado cuando se produce fallo es el equivalente a 200 ciclos.*

El número de accesos a instrucciones es $3 + 10 \times 5 = 53$ y el de accesos a datos 10, lo que hace un total de 63 accesos a memoria.

Las ocho instrucciones están almacenadas en posiciones consecutivas de memoria, y la dirección de comienzo del código está alineada a bloque (160 es múltiplo de 16, el tamaño del bloque), por lo que el código ocupa dos bloques y se producen 2 fallos durante su ejecución:

uno en la primera referencia a cada uno de ellos. Los datos están almacenados en posiciones de memoria consecutivas a partir de una dirección también alineada a bloque, ocupando tres bloques y produciéndose en consecuencia 3 fallos adicionales.

La tasa de aciertos de la caché es:

$$Hr_{Mca} = \frac{63\ accesos - 5\ fallos}{63\ accesos} = 0,92 \rightarrow 92\,\%$$

El tiempo de fallo se calcula como:

$$T_{fallo} = 2 + 200 = 202\ ciclos$$

Por último, el tiempo de acceso efectivo:

$$T_{efectivo} = \frac{58\ aciertos \times 2\ ciclos + 5\ fallos \times 202\ ciclos}{63\ accesos} = 17,87\ ciclos$$

3. Memoria caché

Es una memoria de pequeña capacidad y alta velocidad, situada entre el procesador y la memoria principal, que contiene una copia de parte de la información residente en la memoria principal, aquella considerada de mayor utilidad en cada momento.

Desde el punto de vista de la caché, la memoria principal está compuesta por **bloques** de palabras consecutivas, todos del mismo tamaño (siempre potencia de dos, y habitualmente entre 16 y 64 bytes). En cada momento, se encuentra en la caché una copia de algunos de estos bloques, en lo que se denominan **líneas** o bloques de caché. En la figura 2, en la que se han considerado bloques de cuatro palabras, se ilustra esta idea. En ella se muestra cómo en un determinado instante de la ejecución de un programa se encuentra en la caché la información de dos bloques de memoria principal, los bloques 0 y 200, en las líneas 2 y 0 de la caché respectivamente.

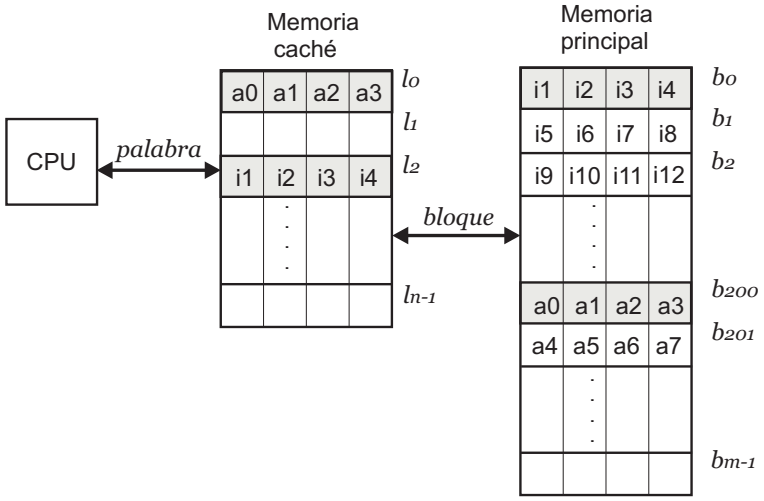


Figura 2: Relación memoria caché-memoria principal.

Dado que la memoria principal tiene mayor capacidad que la caché, hay más bloques que líneas ($m \gg n$), una determinada línea puede contener bloques diferentes en instantes diferentes, por lo que el problema que se plantea es cómo determinar si la información a la que hace referencia la CPU se encuentra o no en la caché, y si se encuentra, en qué línea. Para ello, cada línea lleva asociada una información o **etiqueta** que indica la dirección de memoria principal correspondiente al bloque almacenado en cada momento en dicha línea.

En la figura 3 se muestra cómo la memoria caché consta, por lo tanto, de dos partes: el **directorio**, compuesto por las etiquetas que identifican los bloques de memoria principal residentes en cada momento en la caché, y las **líneas**, donde se almacena una copia de su contenido. La forma en la que se establece la correspondencia entre los bloques de la memoria principal y las líneas de caché depende de la **política de ubicación** utilizada para esta última (véase la sección 3.1).

El funcionamiento básico de la caché es el que se describe a continuación, habiéndose considerado, a modo de ejemplo, un acceso de lectura. El procesador genera una dirección para acceder a la memoria principal. Esta dirección es "interceptada"por el controlador de la caché, que la interpreta como la dirección de un espacio bidimensional, compuesta por: el número de **bloque** donde se encuentra la información, y su **posición** dentro del bloque. El primer componente se utiliza para comprobar, en el directorio de la caché, si existe una etiqueta cuyo valor coincide con el bloque buscado (la forma en que se realiza esta comprobación depende de la política de ubicación utilizada). En esta comprobación puede suceder lo siguiente:

1. Que el campo **bloque** de la dirección coincida con alguna de las etiquetas del directorio. Esto significa que la información se encuentra en la caché, justamente en la línea asociada a dicha etiqueta (hay un **acierto**). Para completar el acceso, basta con acceder a la posición correspondiente dentro de la línea con el segundo componente de la dirección, e indicar al procesador que tiene la información disponible.

2. Que no coincida con ninguna de las etiquetas del directorio, lo que significa que la información no se encuentra en ese momento en la caché (se produce un **fallo**).

En el segundo caso, el controlador de la caché se encarga de iniciar la lectura del bloque de memoria principal en el que se encuentra la información, copiarlo en una línea de caché, actualizar su etiqueta asociada e indicar al procesador cuándo tiene la información disponible, para que éste pueda continuar. No olvidemos que, además, esta actualización de la caché puede llevar consigo el **reemplazo** de la información que hubiera en dicha línea, por lo que el controlador es el encargado de la gestión de dicha eventualidad.

El procesamiento de los fallos de caché provoca por lo tanto un "bloqueo"del procesador, y en consecuencia un tiempo adicional, que en adelante denominaremos **tiempo de espera** (T_{espera}). Por el momento supondremos que el procesador debe esperar los ciclos necesarios hasta que se complete la transferencia del bloque completo a la caché. En la sección 3.7 se estudian algunas alternativas para reducir este tiempo, que permiten que el procesador pueda continuar sin esperar a que se transfiera todo el bloque. Asimismo se verán algunas soluciones utilizadas en los procesadores con *pipeline* de instrucciones y ejecución fuera de orden que tienen este mismo objetivo.

Una cuestión a considerar es la posibilidad de que una línea de caché no contenga información válida. Esto ocurre por ejemplo cuando arranca el computador, en cuyo caso la

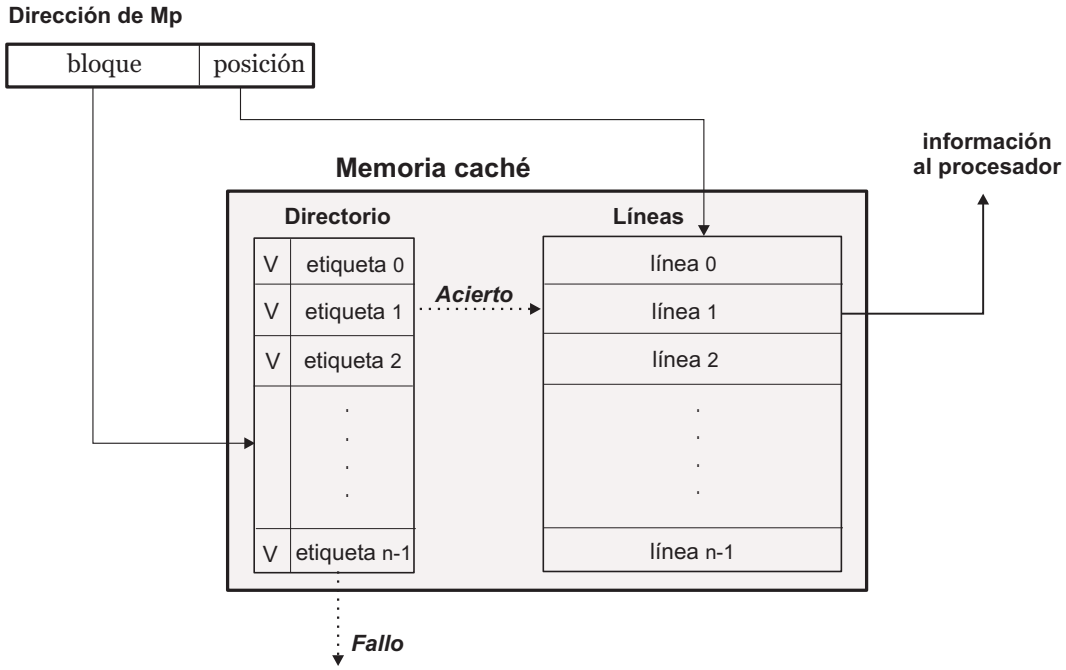


Figura 3: Esquema simplificado del acceso a una memoria caché.

caché está vacía y las etiquetas no tienen validez alguna. Incluso después de haber ejecutado algunas instrucciones puede haber líneas que aún no se han utilizado (vacías). En estos casos, hay que evitar que un acceso a caché produzca casualmente acierto, obteniéndose así información no válida. Una forma de hacerlo es asociar a cada línea un **bit de validez** para indicar si ésta contiene o no información válida (bit **V** en la figura 3), de modo que si está a cero se considera un fallo, evitándose así acceder a información no válida. Dicho bit se deberá poner a cero, generalmente por *hardware*, cuando arranca el computador, o bien para evitar problemas de coherencia en sistemas multiprocesador [Sta10]. Además, prácticamente todos los procesadores disponen de instrucciones privilegiadas para realizar esta operación de invalidación por *software*.

En las siguientes secciones se tratan las decisiones de diseño que influyen, como se verá más adelante, en el rendimiento de la memoria caché, y que son fundamentalmente las siguientes:

- La **política de ubicación**, que establece la forma en que se realiza la correspondencia entre los bloques de memoria principal y las líneas de caché.
- La **política de extracción**, que determina cuándo y qué información se envía a la caché.
- La **política de reemplazo**, que establece qué línea debe abandonar la caché para dejar espacio a un nuevo bloque.
- La **política de escritura**, que determina el momento en que se actualiza la información en memoria principal cuando se modifica su copia en caché (acceso de escritura) .
- El **tamaño** más adecuado de la caché así como de sus líneas.
- La **unicidad** y **homogeneidad** de la caché, esto es, si se emplea una sola memoria que contenga todo tipo de información o si, por el contrario, se emplean varias cachés para tipos de información distintos (p.e. instrucciones y datos).

- La reducción del **tiempo de espera** en caso de fallo.
- El uso de varios niveles de memoria caché (**memorias caché multinivel**).

3.1. Políticas de ubicación.

Para localizar una determinada información en la memoria caché es necesario tener una función que establezca la correspondencia entre su dirección en la memoria principal y la de su copia en caché. Como se vio anteriormente, esta correspondencia se establece entre bloques de memoria principal y líneas de caché, y la forma en la que se realiza viene determinada por la política de ubicación. Las políticas utilizadas habitualmente son las siguientes:

- **Directa**. Cada bloque de memoria principal puede ubicarse en una única línea de caché.
- **Asociativa**. Cada bloque de memoria principal puede ubicarse en cualquier línea de caché.
- **Asociativa por conjuntos**. Cada bloque de memoria principal puede ubicarse en un conjunto restringido de líneas de caché.

Caso de estudio. Para estudiar el funcionamiento de cada una de estas políticas o funciones de correspondencia utilizaremos, a modo de ejemplo, un sistema de memoria de dos niveles con las siguientes características:

Memoria caché:
Capacidad[1] 8KB $\Rightarrow$ 512 líneas (2^9)
Tamaño de línea 16 bytes

Memoria principal:
Accesible a nivel de byte, mediante direcciones de 32 bits $\Rightarrow 2^{28}$ bloques

El problema se centra, en este caso, en cómo establecer la correspondencia entre los 2^{28} bloques de memoria principal y las 2^9 líneas de la caché.

3.1.1. Correspondencia Directa.

Consiste en hacer corresponder a todo bloque i de memoria principal una única línea j de caché, donde $j = i$ **módulo** k, siendo k el número total de líneas de la caché.

En la figura 4 se ilustra esta correspondencia para el caso de estudio planteado, en el que un bloque i puede ubicarse **únicamente** en la línea (i **módulo** 512) de la caché. Dado que la memoria principal consta de 2^{28} bloques y la caché únicamente de 512 (2^9) líneas, 2^{19} bloques de memoria principal están asignados a cada línea de caché, por lo que se necesitan 19 bits para identificar cuál de ellos es el que se encuentra en cada momento en cada línea.

La dirección de memoria principal de 32 bits, desde el punto de vista de la caché, consta ahora de tres campos (véase la figura 5). Los 4 bits menos significativos identifican la palabra a la que se refiere el acceso dentro del bloque, y los 28 restantes identifican el bloque (2^{28} bloques posibles).

[1] La capacidad de la caché indica la cantidad de datos o instrucciones que puede contener. No incluye los bits del directorio, el de validez, y otros que pudieran ser necesarios para las políticas de escritura y reemplazo.

El controlador de la caché interpreta estos 28 bits como una **etiqueta** de 19 bits y un campo **línea** de 9 bits, donde:

- **línea** indica la línea de caché donde le corresponde estar ubicado al bloque de memoria principal al que se refiere el acceso.
- **etiqueta** identifica a cuál de los 2^{19} bloques que pueden residir en dicha línea se refiere dicho acceso.

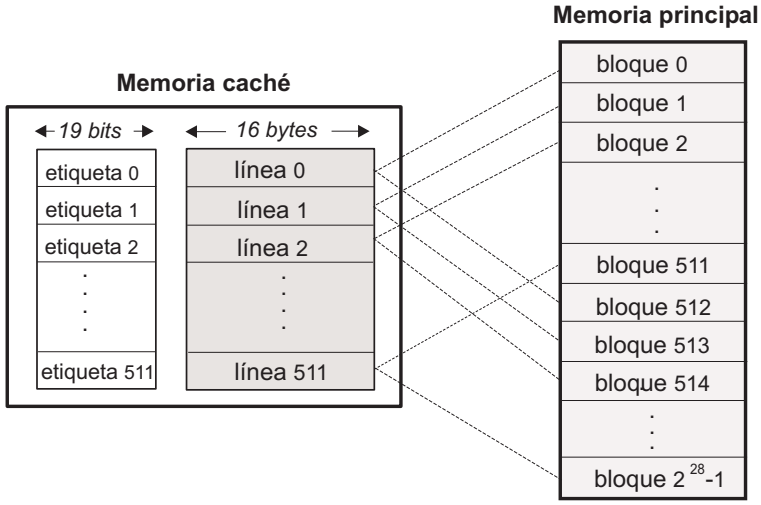


Figura 4: Ejemplo de correspondencia directa.

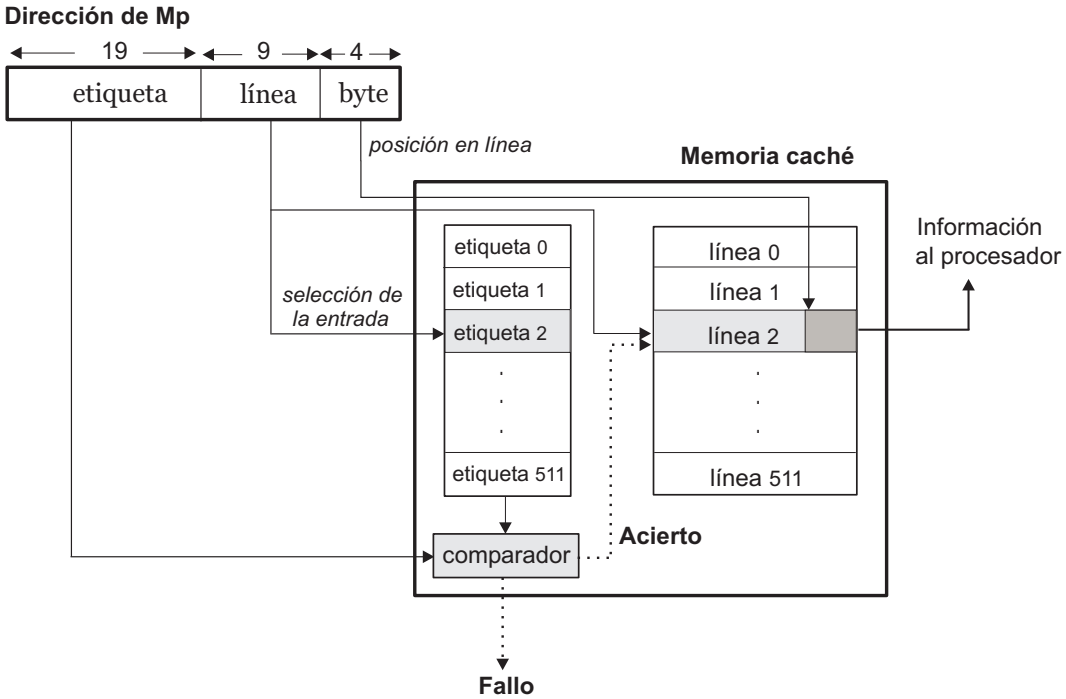


Figura 5: Acceso a la memoria caché con correspondencia directa.

La figura 5 muestra cómo se usan estos campos para acceder a la memoria caché. El campo **línea** se utiliza para seleccionar, tanto la línea donde se espera encontrar la información como

su etiqueta asociada. A continuación, si el bit de validez (que no se muestra en la figura) correspondiente a la línea seleccionada está a uno, se compara la **etiqueta** de la dirección con la seleccionada en el directorio de la caché. Si ambas coinciden significa que hay acierto y, por lo tanto, se utiliza el campo **byte** para acceder a la información. Si no coinciden, o el bit de validez está a cero, significa que la información buscada no se encuentra en la caché y por lo tanto se produce un fallo.

Entre las ventajas de esta política está el hecho de que, cuando la referencia a memoria es de **lectura** (ejemplo de la figura 5), se puede simultanear el acceso a la información con la comparación de las etiquetas. Esto supone una forma de optimizar el $T_{acierto}$ en este tipo de accesos que, por otra parte, representan un porcentaje significativo del total de los accesos a memoria. Si la comparación da como resultado un fallo, basta con ignorar la información obtenida.

Por otra parte, la política de reemplazo resulta inmediata. Ya que cada bloque de memoria principal puede ubicarse en una única línea de la caché, no habrá elección posible, por lo que el retardo introducido por la política de reemplazo será nulo.

Esta política tiene sin embargo un inconveniente. Pongamos como ejemplo un programa en el que se accede repetida y alternativamente a dos bloques que corresponden a la misma línea de caché. El acceso al segundo bloque ocasionaría el reemplazo del primero. Al acceder a este último se produciría de nuevo fallo, reemplazándose el anterior, y así sucesivamente. Todos los accesos a ambos bloques causarían fallo, además de desencadenar un tráfico inútil entre la memoria principal y la caché, lo que acarrearía un incremento del tiempo de acceso efectivo.

Los fallos que resultan del acceso a un bloque que ya estuvo en caché y fue desalojado por otro bloque que corresponde a la misma línea de caché se denominan **fallos por conflicto**. En la página 15 se plantea un ejercicio en el que se profundiza en este problema.

3.1.2. Correspondencia Asociativa.

Con esta correspondencia, cualquier bloque de memoria principal puede ubicarse en cualquier línea de la caché. En la figura 6 se ilustra esta correspondencia para el caso de estudio planteado en la sección 3.1, en el que cualquiera de los 2^{28} bloques de memoria principal puede ubicarse en cualquiera de las 512 líneas de la caché.

El controlador de la caché asociativa interpreta la dirección de memoria de 32 bits de modo que los 4 bits menos significativos (**byte**) tienen el mismo significado que en la correspondencia directa, indicando la palabra dentro del bloque, y los 28 bits restantes (**etiqueta**) identifican el bloque al que se refiere el acceso. El tamaño necesario para cada etiqueta del directorio de la caché es, en este caso, 28 bits, ya que debe identificar cuál de los 2^{28} bloques posibles se encuentra en un determinado instante en una línea.

En la figura 7 se muestra cómo se usan estos campos para acceder a la caché. El campo **etiqueta** de la dirección se compara con todas las etiquetas del directorio. Si coincide con alguna de ellas, y el bit de validez correspondiente está a uno, significa que hay acierto y se utiliza el campo **byte** para acceder a la información en la línea correspondiente. Si no coincide con ninguna, significa que la información que se busca no está en la caché, produciéndose en consecuencia un fallo.

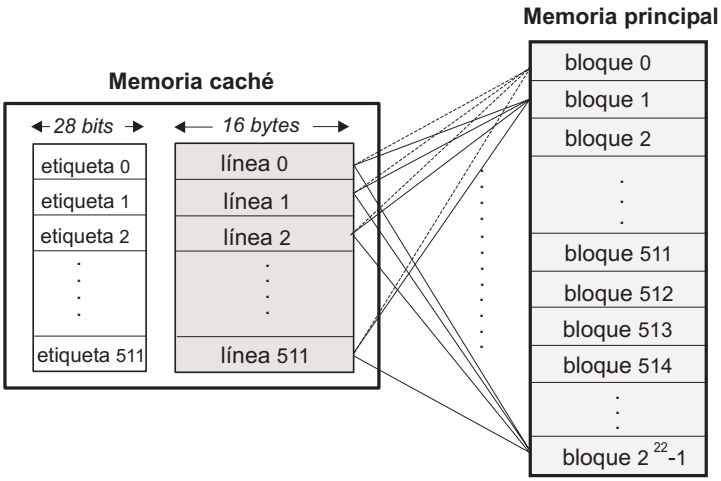


Figura 6: Ejemplo de correspondencia asociativa.

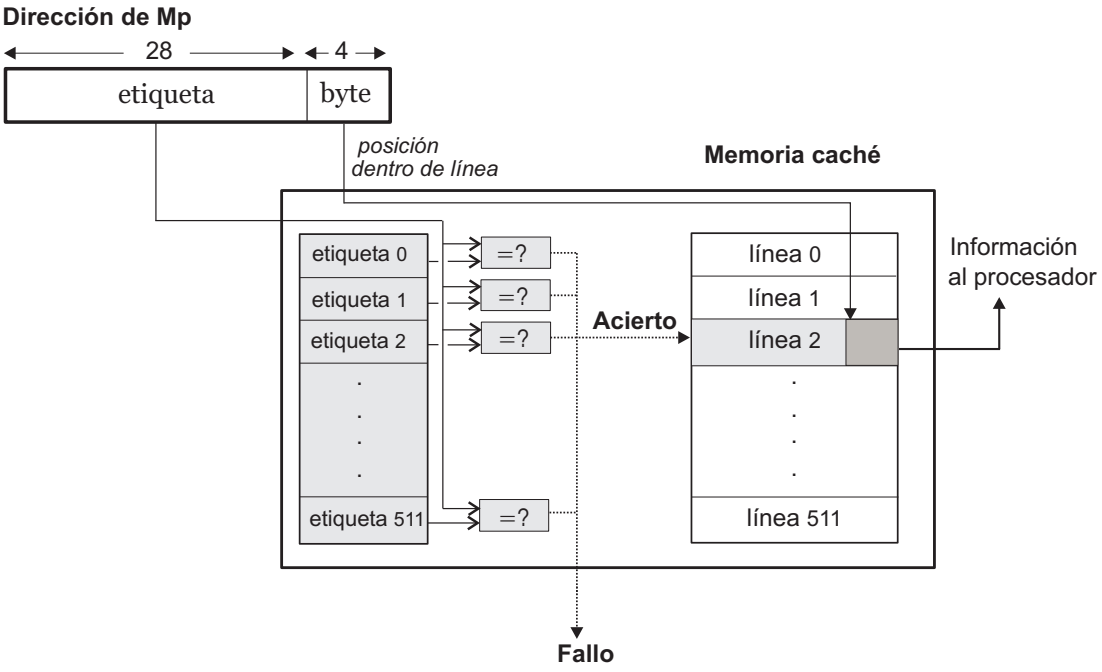


Figura 7: Acceso a una memoria caché con correspondencia asociativa.

Ahora, la búsqueda en el directorio y el acceso a la información dentro de la línea deben realizarse necesariamente de manera secuencial, ya que no es posible acceder a la línea hasta no saber en qué entrada del directorio se produce el acierto. Además, para que la búsqueda sea eficiente, debe hacerse en todas las entradas del directorio a la vez, por lo que ha utilizarse un comparador asociado a cada una de sus entradas. Todo ello supone un incremento en el coste, así como en el tiempo de acceso en caso de acierto, de modo que sólo se utiliza esta organización en memorias con un número de líneas reducido (véase la sección 5.3.2).

Por último, hay que hacer notar que la flexibilidad de esta correspondencia evita el inconveniente mencionado en la directa, y por lo tanto los fallos por conflicto, ya que los dos bloques utilizados repetidamente podrían ubicarse en dos líneas distintas de caché. Además, permite

utilizar diferentes políticas de reemplazo, pudiendo tenerse en cuenta la utilización o uso de los bloques con el fin de conseguir una alta tasa de aciertos.

3.1.3. Correspondencia Asociativa por conjuntos.

Esta correspondencia combina la sencillez de la directa y la flexibilidad de la asociativa. Consiste en dividir la caché en ***C* conjuntos**, de ***L*** líneas cada uno, y aplicar la correspondencia directa a nivel de conjunto. De este modo, un bloque ***i*** de memoria principal puede ubicarse en cualquiera de las líneas del conjunto (***i* módulo *C***) de la caché.

En el caso de estudio planteado en la sección 3.1, y considerando conjuntos de dos líneas, la caché estaría formada por 512 líneas / 2 líneas/conjunto = 256 conjuntos ($C = 256$), por lo que un bloque ***i*** de memoria principal podría ubicarse en cualquiera de las líneas del conjunto **(*i* módulo 256)** de la caché. En la figura 8 se muestra esta correspondencia para el caso planteado.

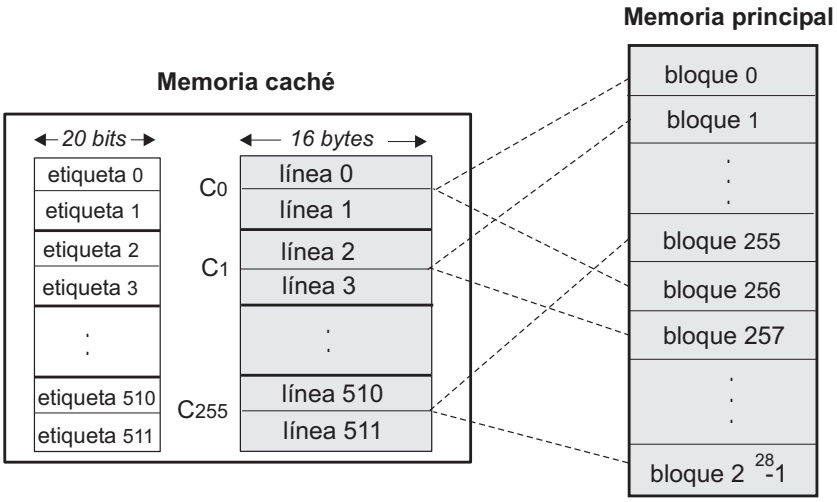


Figura 8: Ejemplo de correspondencia asociativa por conjuntos.

Ya que la memoria principal está compuesta por 2^{28} bloques y la caché por 2^8 conjuntos, 2^{20} bloques están asignados a un mismo conjunto de la caché, por lo que se necesitan 20 bits para identificar qué bloques se encuentran en las líneas de un conjunto en instante dado. Así, la dirección de memoria principal de 32 bits, desde el punto de vista de la caché asociativa por conjuntos, consta de tres campos (véase la figura 9). Los 4 bits menos significativos identifican la palabra dentro del bloque y los 28 bits restantes se interpretan como una **etiqueta** de 20 bits y un campo **conjunto** de 8 bits, donde:

- **conjunto** indica el conjunto donde le corresponde estar ubicado al bloque de memoria principal al que se refiere el acceso.
- **etiqueta** identifica a cuál de los 2^{20} bloques posibles se refiere el acceso.

La figura 9 ilustra cómo se usan estos campos para acceder a la caché. El campo **conjunto** se utiliza para seleccionar el conjunto de la caché donde se puede encontrar el bloque. A continuación se compara la **etiqueta** de la dirección con todas las etiquetas del directorio, en el conjunto seleccionado. Si coincide con alguna, y el bit de validez está a uno, significa que hay acierto y se utiliza el campo **byte** para acceder a la información dentro de la línea

correspondiente. Si no coincide con ninguna, significa que la información no está en la caché, produciéndose un fallo.

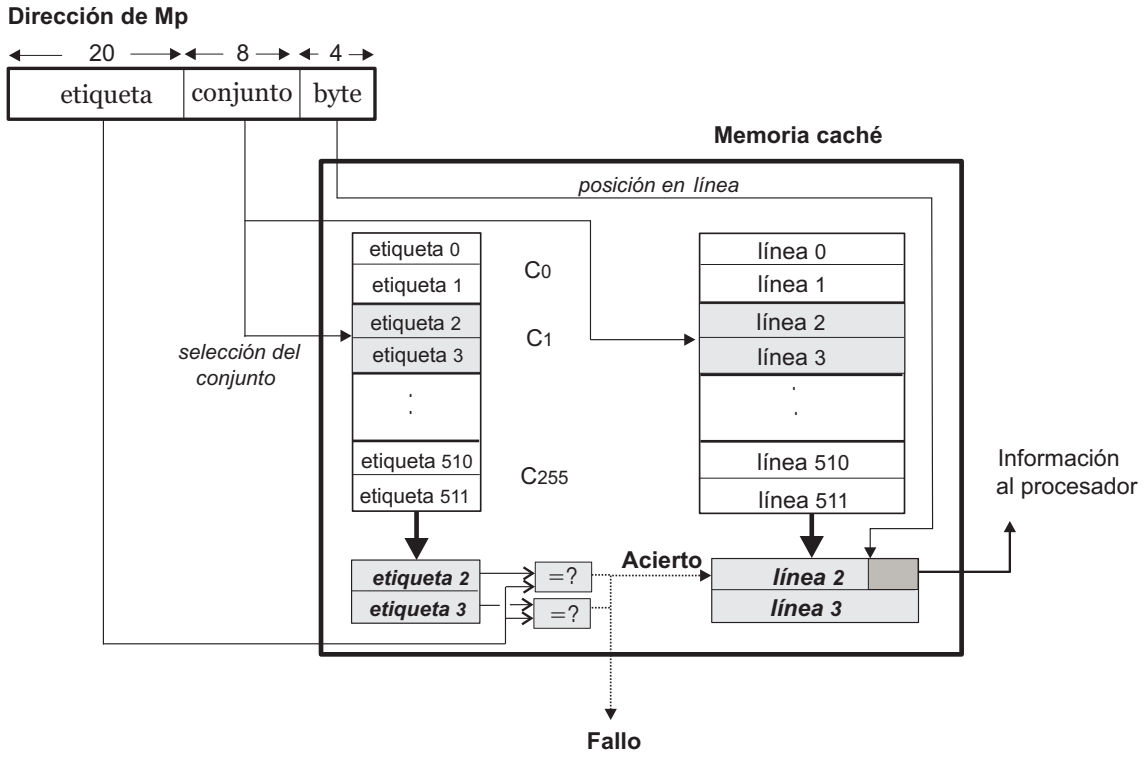


Figura 9: Acceso a una memoria caché con correspondencia asociativa por conjuntos.

En este caso, el coste de la búsqueda depende del número de etiquetas que se tengan que comparar simultáneamente, es decir del número de entradas por conjunto o **grado de asociatividad**, así como de su longitud. La correspondencia asociativa por conjuntos reduce, de este modo, el coste de la totalmente asociativa, proporcionando sin embargo una tasa de aciertos cercana a esta última. Este es uno de los motivos por los que es la política más utilizada. Resultados experimentales [HP07] muestran que, para sistemas monoprocesador, conjuntos de 2 a 8 bloques proporcionan una tasa de aciertos cercana al de la política totalmente asociativa con un coste poco mayor que el de la directa.

Por último hay que hacer notar que con esta correspondencia puede darse un problema similar al de la correspondencia directa, esto es, fallos por conflicto. En la caché de nuestro ejemplo esto sucedería si la CPU accede repetidamente a tres o más bloques que correspondan a un mismo conjunto de la caché. En cuanto a la políticas de reemplazo posibles, éstas se aplican a nivel de conjunto.

Ejercicio 2.- *Considere el siguiente fragmento de código, en el que los elementos de A, B y C son enteros de 32 bits, y están almacenados en memoria principal a partir de las direcciones 0x8000, 0xA000 y 0x7000 respectivamente.*

```
for (i=0; i<256; i=i+1)
    C[i] = A[i] + B[i];
```

Indique en qué lugar (líneas o conjuntos) de la caché se ubicará cada uno de los vectores en cada una

de las políticas de ubicación estudiadas en los apartados anteriores, teniendo en cuenta los parámetros de la jerarquía utilizada en ellos.

Las direcciones del primer bloque de los vectores se interpretan de la forma siguiente:

Directa:

31 13	12 4	3 0	
etiqueta	línea	byte	
00 .. 0100	0 0000 0000	0000	A
00 .. 0101	0 0000 0000	0000	B
00 .. 0011	1 0000 0000	0000	C

Asociativa por conjuntos de 2 líneas:

31 12	11 4	3 0	
etiqueta	conjunto	byte	
00 .. 0 1000	0000 0000	0000	A
00 .. 0 1010	0000 0000	0000	B
00 .. 0 0111	0000 0000	0000	C

Asociativa:

31 4	3 0	
Etiqueta	Byte	
00 .. 0 1000 0000 0000	0000	A
00 .. 0 1010 0000 0000	0000	B
00 .. 0 0111 0000 0000	0000	C

Cada vector ocupa:

$$\frac{256 \times 4\ bytes}{16\ bytes/bloque} = 64\ bloques$$

Según lo anterior, en la caché directa A y B ocuparán las líneas 0 .. 63, desalojándose mutuamente en cada referencia, y C las líneas 256 .. 320.

En la asociativa por conjuntos, los tres vectores se ubicarán en los conjuntos 0 .. 63. Como cada conjunto tiene únicamente dos líneas, los tres vectores se desalojarán entre sí.

En la caché asociativa se pueden ubicar en cualquier línea de caché. Como los tres vectores ocupan un total de $3 \times 64 = 192$ *bloques* y la caché tiene 512 líneas, no habrá posibilidad de que se desalojen entre sí.

3.2. Políticas de extracción.

La política de extracción determina cuándo y qué información se lleva a la memoria caché. Las más utilizadas son la extracción bajo demanda y la extracción con anticipación.

La extracción **bajo demanda** consiste en enviar a la caché únicamente la información que necesita el procesador y estrictamente cuando la necesita, es decir sólo el bloque cuyo acceso ocasiona un fallo.

La extracción **con anticipación** o *prefetching* tiene como objetivo anticiparse a las necesidades del procesador, llevando a la caché bloques a los que es previsible que se acceda en un instante próximo. La utilización de este método es de gran interés en programas con alta proximidad de referencias espacial, en cuyo caso puede suponer un incremento significativo de la tasa de aciertos de la caché respecto a la política de extracción bajo demanda. Esta técnica se puede realizar mediante *hardware* o mediante *software*, en cuyo caso es el compilador el encargado de insertar instrucciones especiales de *prebúsqueda* para acceder a la información antes de que se necesite [HP07].

Por otra parte, hay que sopesar el beneficio que se obtiene con la anticipación en cuanto a tasa de aciertos, y el coste adicional que supone en tiempo enviar a la caché más de un bloque. Por ello, la extracción con anticipación sólo tendrá sentido si se permite que el procesador pueda continuar con la ejecución del programa antes de completarse la transferencia de los bloques enviados con anticipación a la caché. En la sección 3.7 se estudian los métodos que se suelen emplear para permitir este comportamiento.

Por último, hay que mencionar que para cualquiera de las políticas anteriores se puede considerar además la extracción **selectiva**, que tiene como objetivo evitar que un determinado tipo de información sea enviado a la caché. Para ello basta con marcar dicha información para que nunca pueda ser enviada a la caché. En la sección 5.3 se estudia cómo puede establecerse esta marca.

Un ejemplo de aplicación de la extracción selectiva es el de un sistema multiprocesador con memoria compartida donde cada uno de los procesadores dispone de su propia memoria caché, y existe información compartida susceptible de ser modificada por uno o varios procesadores. En este caso, uno de los métodos para conservar la **coherencia** puede ser, mantener esta información en memoria principal y no enviarla en ningún caso a la caché de ningún procesador. De este modo, cualquier modificación se realizará directamente sobre la memoria principal y todos los procesadores accederán a la misma información.

3.3. Políticas de reemplazo.

Es el procedimiento que se utiliza para determinar qué línea debe abandonar la memoria caché para albergar un bloque procedente de la memoria principal cuando no tiene espacio libre. Evidentemente, sólo tiene sentido hablar de estas políticas para las memorias caché con correspondencia asociativa o asociativa por conjuntos, en las que hay varias líneas entre las que elegir a la hora de efectuar el reemplazo. En cachés asociativas serían candidatas todas la líneas, mientras que en la asociativa por conjuntos lo serían las líneas de un determinado conjunto.

Entre los métodos que se pueden emplear, dependiendo del criterio de selección utilizado, los más habituales en las memorias caché son: **aleatorio**, **FIFO** (*first in first out*), y **LRU** (*least recently used*).

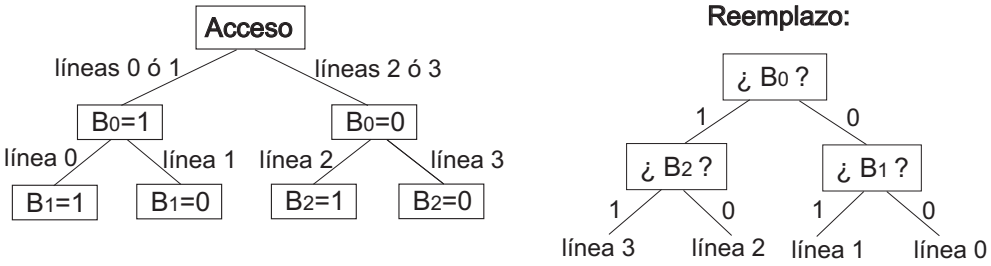
La política **aleatoria** consiste en reemplazar una de las líneas al azar, mientras que la **FIFO** reemplaza la línea que lleva más tiempo en la caché. Por último, la política **LRU** reemplaza la línea que lleva más tiempo sin ser referenciada, teniendo en cuenta de este modo el principio de **proximidad temporal**, ya que los bloques utilizados menos recientemente son, en principio, los que tienen menor probabilidad de volver a serlo.

Debido a que la aplicación de la política de reemplazo debe afectar lo menos posible a la velocidad del sistema, el mecanismo *hardware* utilizado debe ser sencillo. Desde este punto de vista, la política **aleatoria** es la más sencilla y menos costosa. La política **FIFO** es algo más compleja, ya que es necesario incluir en la caché información para seleccionar la siguiente línea a reemplazar, información que se debe consultar y actualizar en cada fallo de caché. Esta política es muy utilizada en las cachés de procesadores para sistemas empotrados (*embedded processors*), sujetos a estrictas restricciones de potencia y coste, como es el caso de las arquitecturas ARM.

La complejidad y el coste aumenta en la política **LRU**, ya que se necesita manejar información acerca de la utilización de las distintas líneas candidatas para el reemplazo, información que se

debe actualizar en cada acceso a la caché. Por todo ello, se suele emplear un mecanismo cuyo comportamiento se aproxima al de la política LRU (*pseudo LRU*), pero cuya implementación sea más sencilla. La memoria caché de algunos procesadores Intel y PowerPC utiliza esta política de reemplazo.

A continuación se muestra, a modo de ejemplo, el funcionamiento de la política de reemplazo *pseudo LRU* utilizada en la caché del i486, asociativa por conjuntos de 4 líneas. Para su implementación se utilizan 3 bits (B_0, B_1 *y* B_2), asociados a cada conjunto, que se actualizan en cada acceso con acierto o cuando se tiene que reemplazar una línea del conjunto de la siguiente forma:



Finalmente, cabe plantear la posibilidad de utilizar una política de reemplazo **óptima**, que consiste en reemplazar la línea que no se va a utilizar durante mayor periodo de tiempo. Lógicamente no es posible implementar esta política, ya que sería necesario conocer el comportamiento futuro de los accesos a memoria, por lo que se utiliza únicamente como referencia para evaluar la eficiencia de las políticas mencionadas anteriormente.

3.4. Políticas de escritura.

La política de escritura determina el instante en que se actualiza la información en el siguiente nivel de la jerarquía cuando el procesador realiza un acceso de escritura sobre su copia en la caché. Esta actualización es necesaria ya que la caché se comporta como un almacenamiento temporal, cuyas modificaciones se deben reflejar en algún momento en la memoria principal. Existen dos formas de proceder cuando el procesador realiza una escritura:

- **Escritura inmediata** (*Write through*). La escritura se realiza a la vez en la caché y en la memoria principal. De este modo, la copia de la información en la caché es siempre **coherente** con la de la memoria principal.
- **Escritura aplazada** (*Write back* o *Copy back*). La escritura se realiza únicamente en la caché y, sólo cuando la línea correspondiente debe ser reemplazada se actualiza la memoria principal.

  Para saber si la línea a reemplazar ha sido modificada, se asocia a cada línea de caché un **bit de modificación** (*dirty bit*), que debe ponerse a uno cuando el procesador realiza una escritura en alguna de las palabras de dicha línea. La comprobación del valor de este bit en el momento del reemplazo evita escribir en memoria principal líneas que no se han modificado y que, por lo tanto, contienen la misma información que la memoria principal, disminuyendo así el tráfico entre ambas memorias.

Las dos políticas tienen sus ventajas e inconvenientes. ***Copy back*** es más compleja, ya que necesita una lógica adicional para la gestión del bit de modificación. Sin embargo, produce

menos tráfico en la memoria principal, dado que sólo hay que escribir en ella cuando se reemplaza una línea de caché que ha sido modificada. Tiene también la ventaja de que los accesos de escritura, en caso de acierto, se llevan a cabo a la velocidad de la memoria caché. El inconveniente de esta política es que puede dar lugar a problemas de coherencia si, desde que se escribe la información en la caché hasta que la escritura se refleja en la memoria principal, otro componente del computador (p.e. otro procesador en un sistema *multicore*) accede a dicha información a través de la memoria principal. En estos sistemas, es necesario implementar protocolos para mantener la coherencia [Sta10].

Write through tiene la ventaja de mantener en todo momento la coherencia de la información, puesto que la memoria principal está siempre actualizada. Sin embargo conlleva un mayor tráfico en la memoria principal, ya que todas las escrituras, aciertos y fallos, pasan por ella. También implica un mayor tiempo de espera por parte del procesador, que no podrá continuar hasta que se realice la escritura en memoria principal.

Por otra parte, cabe plantear dos formas de actuar cuando se produce fallo en un acceso de escritura:

1. **Escritura con actualización** de la caché (*Write with allocate*). En este caso, se lleva a la caché el bloque que produjo el fallo y, a continuación, se realiza el acceso de escritura en la forma dictada por la política de escritura utilizada. Nótese que este comportamiento es el mismo que el descrito para los fallos de lectura. En el diagrama de flujo de la figura 10 se muestran las acciones que se llevan a cabo desde que el procesador realiza una petición de acceso a memoria hasta que ésta se completa. Este mecanismo admite no obstante una variante: realizar primero la escritura en la memoria principal y llevar a continuación el bloque, ya actualizado, a la caché.

2. **Escritura sin actualización** de la caché (*Write with no allocate*). Consiste en realizar la escritura únicamente en la memoria principal, de modo que el bloque que produjo el fallo no se lleva a la caché.

La segunda alternativa se utiliza con escritura inmediata, dando lugar a la política denominada **escritura inmediata sin actualización** (*write through with no allocate*). En este caso, dado que un posterior acceso de escritura a una palabra del bloque que produjo fallo irá obligatoriamente a memoria principal, no se ganaría nada llevándolo a la caché.

La primera alternativa se asocia a la escritura aplazada, dando lugar a la denominada política de **escritura aplazada con actualización** (*write back with allocate*). En el caso de acceder posteriormente en escritura a una palabra del mismo bloque se producirá un acierto, con el consiguiente ahorro de tiempo. A continuación veremos un ejemplo que ilustra esta idea.

Ejercicio 3.- *Suponga un fragmento de programa que copia en un vector A los elementos de otro vector B multiplicados por un determinado valor. Cada bloque tiene capacidad para almacenar 4 elementos de un vector y se conocen los siguientes tiempos:*

T_{Mca} = 1 ciclo, T_{Mp} = 100 ciclos y tiempo en transferir un bloque entre Mp y Mca = 110 ciclos.

Calcule el tiempo de acceso efectivo en los accesos de escritura para las políticas de escritura inmediata sin actualización (WTWNA) y aplazada con actualización (CBWA). En este último caso se supondrá que la probabilidad de reemplazar una línea modificada es del 10 %.

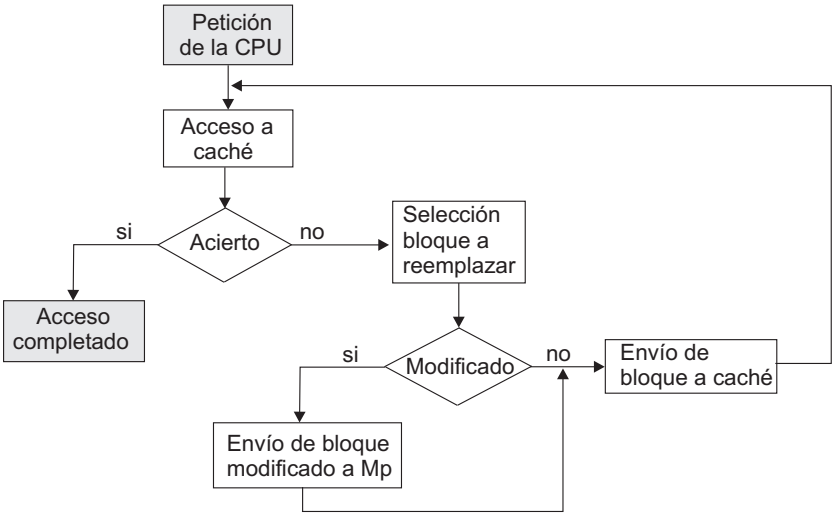


Figura 10: Diagrama de flujo del acceso a una memoria caché *copy back*.

Utilizando *CBWA*, y como cada bloque da cabida a 4 elementos del vector A, se producirá un fallo de escritura por cada 4 accesos, por lo que la tasa de fallos en escritura será del 25 %. El tiempo de acceso efectivo en escritura será el siguiente:

$$T_{efectivo} = 0,75 \times 1 + 0,25 \times (1 + (0,1 \times 110) + 110 + 1) = 31,5 \; ciclos$$

Como se puede observar, el tiempo en caso de fallo es el correspondiente al acceso a Mca, seguido de la escritura de la línea reemplazada a Mp (sólo en en caso de que dicha línea esté modificada), la transferencia del nuevo bloque de Mp a Mca y, finalmente, la escritura de la palabra en la Mca.

En el caso de *WTWNA*, la tasa de fallos en los accesos de escritura será del 100 % ya que en los fallos de escritura no se lleva el bloque correspondiente a Mca. El tiempo de acceso será el mismo para todas las escrituras, 100 ciclos (el máximo de los tiempos de Mca y Mp), por lo que el tiempo de acceso efectivo en escritura será:

$$T_{efectivo} = 100 \; ciclos$$

3.5. Tamaño de la caché y de los bloques.

La elección del tamaño de la memoria caché así como de los bloques influye directamente en su tasa de aciertos y, por lo tanto, en el tiempo de respuesta del sistema de memoria.

Para una caché de una determinada capacidad, un aumento del tamaño de los bloques favorece la proximidad espacial y es, por lo tanto, muy probable que se incremente su tasa de aciertos si la información a la que se accede tiene este tipo de comportamiento, como ocurre por ejemplo en los accesos a elementos consecutivos de un vector. Sin embargo esta mejora va en detrimento de la proximidad temporal, ya que conlleva una reducción del número total de líneas. Por otra parte, supone un incremento del tiempo empleado en la transferencia o el reemplazo de un bloque, lo que a su vez puede suponer un incremento del tiempo de espera

por parte del procesador.

Como contrapartida, la elección de bloques de tamaño reducido disminuye la cantidad de información que se transfiere entre la memoria principal y la caché y, en consecuencia, el tiempo de actualización de la caché y el tiempo de espera del procesador.

El tamaño de bloque más adecuado dependerá en gran medida del comportamiento de los programas que se ejecuten. Por ello, esta elección suele ser resultado de la observación de simulaciones basadas en las trazas de programas que constituyan una carga representativa del sistema. El tamaño de la cachés de primer nivel suele oscilar entre 16 y 64 KB, y el de los bloques entre 16 y 128 bytes (véase la tabla 2).

3.6. Unicidad y homogeneidad de las memorias caché.

Desde el punto de vista de la información que puede contener, la memoria caché puede ser unificada, lo que significa que puede dar cabida a todo tipo de información (instrucciones y datos), o bien pueden existir cachés separadas que contengan información de un único tipo.

Desde la aparición de los procesadores con *pipeline*, lo habitual es utilizar cachés separadas para instrucciones y datos (aplicación del concepto de arquitectura *Harvard* en el nivel de la memoria caché). Este diseño permite solapar los accesos a ambos tipos de información, por ejemplo leer el operando de una instrucción mientras se busca la siguiente instrucción. Con ello se consigue un mayor ancho de banda que con una caché unificada (en el mejor de los casos se duplica) en la que los accesos se deben realizar de forma secuencial, además de una mayor velocidad de ejecución por parte del procesador.

Por otra parte, el disponer de cachés separadas para instrucciones y datos permite conseguir un mejor rendimiento en cada una de ellas. Dado que el comportamiento, en cuanto a proximidad de referencias, de las instrucciones es diferente del de los datos, se puede optimizar cada caché por separado, utilizando, por ejemplo, distintas políticas de extracción, ubicación y reemplazo. De hecho, diversos estudios muestran que la tasa de aciertos en la caché de instrucciones es mayor que la que se obtiene en la de datos, al haber mayor proximidad temporal, y mayor de la que se obtendría con una caché unificada. En la tabla 2 se muestran, a modo de ejemplo, las características de las memorias caché de primer nivel de algunos procesadores actuales. En el caso de los procesadores *multicore*, Power8 e intel i7, los tamaños corresponden a cada uno de los *cores*. La política de escritura se refiere a la actualización del siguiente nivel de caché (véase la sección 3.7.3).

	Caché de instrucciones			**Caché de datos**			
procesador	Tamaño (KB)	bytes por línea	líneas por conjunto	Tamaño (KB)	bytes por línea	líneas por conjunto	Política de Escritura
ARM Cortex-A8	32	64	4	32	64	4	CBWA
intel Core i7	32	64	8	32	64	8	CBWA
Amd Opteron	64	64	2	64	64	2	CBWA
Power8	32	128	8	64	128	8	WTWNA

Tabla 2: Características de memorias caché de primer nivel.

3.7. Reducción del tiempo en los accesos con fallo.

Como hemos visto en apartados anteriores, cuando se produce un fallo en la caché es necesario emplear un tiempo adicional para localizar la información y, en la mayoría de los casos, actualizar la caché con el bloque en el que se encuentra. Por ello, en el diseño de un sistema de memoria eficiente hay que considerar dos aspectos:

- La reducción del **tiempo de espera** por parte del procesador, y en consecuencia del tiempo de acceso, para que éste pueda continuar lo antes posible.
- La reducción del tiempo empleado en actualizar la caché, acción que es necesario realizar en todos los fallos de lectura, así como en los de escritura si la política es "con actualización"(*with allocation*), ya que durante este tiempo el sistema de memoria no podrá atender otra petición (a menos que se utilicen cachés no bloqueantes, cuyo funcionamiento se trata en la sección 3.7.4).

Al tiempo que transcurre desde que se hace una petición al sistema de memoria hasta que éste puede atender la siguiente petición lo denominaremos **tiempo de ocupación**, mientras que el **tiempo de acceso** es el que transcurre desde la petición hasta que el procesador puede continuar con la ejecución de instrucciones. Hay que hacer notar que hasta ahora no se ha hecho distinción entre ambos tiempos, ya que se ha planteado un funcionamiento muy básico de la jerarquía de memorias. En los siguientes apartados se estudian algunas optimizaciones utilizadas en los computadores actuales que hacen necesario comprender la diferencia entre ambos términos.

Para reducir el tiempo de actualización de la caché y, por lo tanto, el tiempo de ocupación, se puede utilizar cualquier mecanismo que permita aumentar el ancho de banda de la memoria principal, como el que estudiaremos en la sección 4. Otra solución es utilizar niveles de caché intermedios entre la de primer nivel, considerada hasta el momento, y la memoria principal (véase la sección 3.7.3).

A continuación se describen algunas de las técnicas empleadas para reducir el **tiempo de espera** por parte del procesador, algunas de las cuales sirven asimismo para reducir el **tiempo de ocupación**.

3.7.1. Buffer de escritura.

Como se vio en la sección 3.4, la utilización de la política de escritura inmediata (*write through*) implica que, en los accesos de escritura, el procesador siempre debe esperar a que se complete la escritura en memoria principal.

Una forma de reducir esta espera es utilizar un *buffer* de escritura (*write buffer*), como se muestra en la figura 11a, con capacidad para almacenar uno o varios datos pendientes de ser escritos en memoria principal. De esta forma, cuando el procesador realiza un acceso de escritura se escribe tanto en el *buffer* como en la caché (si hubiera acierto). A partir de este momento, el procesador puede continuar su labor sin esperar a que las escrituras pendientes en el *buffer* se reflejen en la memoria principal. A medida que éstas van finalizando, se irán liberando las entradas correspondientes en el *buffer*. El procesador deberá esperar únicamente si no existe espacio libre en él.

La utilización de este *buffer* puede provocar sin embargo problemas de coherencia debido

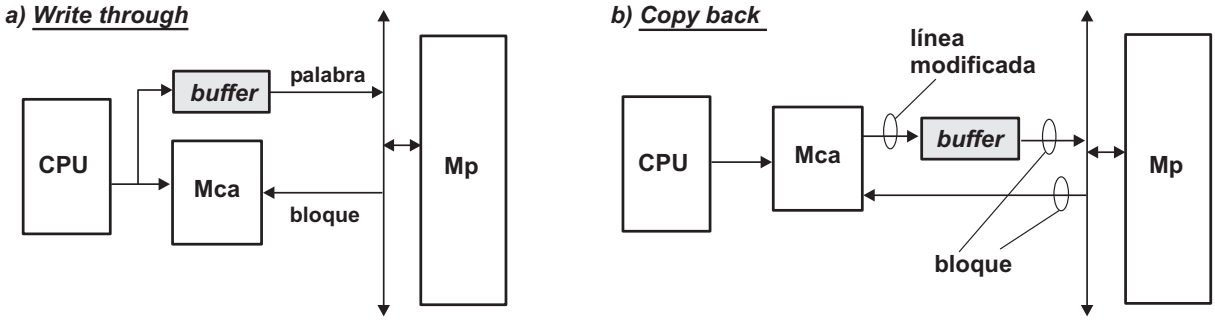


Figura 11: Utilización de un *buffer* de escritura.

a que, durante un cierto intervalo de tiempo, puede haber dos copias distintas de la misma información; el nuevo valor en el *buffer* y el valor antiguo en la memoria principal. De este modo, si tras producirse un fallo de escritura se realiza un acceso de lectura a la misma información, podría suceder que aún no se hubiera actualizado la copia de la memoria principal, leyéndose por lo tanto información incorrecta. Una posible solución es esperar siempre a que se vacíe el *buffer* cuando se produce un fallo de lectura, para asegurar que se está accediendo a la información correcta.

Muchas cachés con **escritura aplazada** también utilizan un *buffer* para reducir el tiempo de penalización cuando se produce un fallo y se reemplaza una línea modificada (figura 11b). En este caso, la optimización consiste en enviar la línea modificada de la caché al *buffer* mientras se lee de memoria principal el bloque que produjo el fallo. De este modo, el procesador no debe esperar a que se complete la actualización del bloque modificado en la memoria principal.

3.7.2. Fallos en lectura.

En los accesos de lectura, la minimización del tiempo de espera por parte del procesador es aún más importante que en los de escritura ya que, a diferencia de estos últimos, el procesador no tiene más remedio que esperar a tener la información para poder continuar. Además, hay que tener en cuenta que, en la ejecución de un programa, el porcentaje de lecturas suele ser muy superior al de escrituras.

Los mecanismos utilizados, sobre todo en las cachés de primer nivel, para reducir el tiempo de espera en los fallos de lectura consisten básicamente en evitar que el procesador espere hasta que se transfiera todo el bloque a la caché. Estos mecanismos, también denominados **políticas de lectura**, son los que se describen a continuación.

- *out of order fetch* o *critical word first*. Consiste en leer en primer lugar la palabra cuyo acceso causó el fallo, y llevarla tanto a la caché como al procesador y, a continuación, transferir el resto de las palabras del bloque a la caché.
- *early restart*. Consiste en leer las palabras del bloque en el orden en el que están almacenadas y llevar al procesador la palabra que causó el fallo tan pronto como ésta llegue a la caché, sin esperar a que se complete la transferencia de todo el bloque.

En el primer caso, el tiempo que debe esperar el procesador es fijo, el tiempo de acceso del siguiente nivel de la jerarquía, mientras que en el segundo caso depende de la posición que ocupe la palabra requerida dentro del bloque.

En ambos casos, si se produce una nueva petición a una palabra del mismo bloque, lo cual es muy probable en el caso de los accesos a instrucciones, tan pronto como la caché reciba dicha palabra, puede enviársela al procesador. Si el acceso corresponde a otro bloque, el procesador deberá esperar a que se complete la transferencia del bloque que causó el fallo.

Ejercicio 4.- *Supongamos que los tiempos de acceso de la Mca y de la Mp son 1 y 80 ciclos respectivamente, los bloques almacenan 8 palabras consecutivas, y que no se reemplazan bloques modificados. ¿Cuál es el tiempo de espera por parte del procesador si se produce un fallo de lectura al acceder a la cuarta palabra de un bloque, para cada una de las políticas de lectura expuestas, suponiendo que durante el tiempo en que se actualiza la caché no se producen nuevos accesos a dicho bloque? ¿Qué ocurriría si el siguiente acceso al que produce el fallo va dirigido a un bloque distinto? ¿Cuál es el tiempo de ocupación?*

Cuando no se accede de nuevo al bloque que produce el fallo, los tiempos de espera serán:

$$T_{espera}(out\ of\ order\ fetch) = 80\ ciclos$$

$$T_{espera}(early\ restart) = 80 \times 4 = 320\ ciclos$$

El tiempo de espera sin utilizar ninguna optimización sería el correspondiente al envío del bloque de Mp a Mca, y la posterior lectura de la palabra demandada por el procesador:

$$T_{espera}(sin\ optimizaciones) = 80 \times 8 + 1 = 641\ ciclos$$

Si el siguiente acceso va dirigido a otro bloque, el procesador deberá esperar en ambos casos a que se complete la transferencia del bloque anterior.

El tiempo de ocupación del sistema de memoria, para ambas optimizaciones, será:

$$T_{ocupacion} = 1 + 80 \times 8 = 641\ ciclos$$

esto es, el tiempo que transcurre desde que el procesador realiza un acceso con fallo en la caché hasta que el sistema de memoria puede atender otro acceso correspondiente a un bloque distinto.

Sin optimizaciones, el tiempo de ocupación sería:

$$T_{ocupacion} = 1 + 80 \times 8 + 1 = 642\ ciclos$$

Durante este tiempo, el sistema de memoria no podría servir una nueva petición, tanto si corresponde al mismo bloque como si no.

3.7.3. Memorias caché multinivel.

Una solución comúnmente utilizada para disminuir el tiempo de espera en los casos de fallo consiste en emplear varios niveles de caché. Esta solución permite a su vez reducir el tiempo de acceso efectivo, que definimos anteriormente como:

$$T_{efectivo} = Hr_{Mca} \times T_{acierto} + (1 - Hr_{Mca}) \times T_{fallo}$$

que depende de la tasa de aciertos de la caché y de la velocidad del siguiente nivel de la jerarquía, que hasta ahora se ha considerado la memoria principal.

Los procesadores actuales incorporan al menos dos niveles de memoria caché:

- Un primer nivel, compuesto por dos memorias caché (una para instrucciones y otra para datos) rápidas y relativamente pequeñas (entre 16 y 64KB cada una), con las que conseguir un $T_{acierto}$ cercano al tiempo de ciclo del procesador.
- Un segundo nivel, compuesto por una caché unificada, de mayor capacidad que la anterior (entre 256 y 512KB) cuyo tiempo de acceso es también mayor, pero aún muy inferior al de la memoria principal, cuya finalidad es que satisfaga gran parte de los fallos de la caché de primer nivel, proporcionando en consecuencia un menor tiempo de penalización en los casos de fallo en la caché de primer nivel.

  Con la inclusión de este segundo nivel se consigue además que lleguen menos peticiones a la memoria principal, aspecto muy importante en los sistemas *multicore* actuales en los que dicha memoria se comparte entre todos los *cores*.

En la tabla 3 se muestran las características de las cachés de segundo nivel de algunos procesadores utilizados actualmente, donde el tamaño corresponde a cada *core*. También se muestran sus correspondientes cachés de datos de primer nivel, para poder comparar las características de ambas.

Hay que hacer notar que generalmente las cachés de ambos niveles cumplen la propiedad de **inclusión**, esto es, la información contenida en el primer nivel lo está también en el segundo. Los procesadores de Amd son una excepción, y la caché de segundo nivel contiene los bloques desalojados de las de primer nivel (instrucciones y datos), por lo que se utiliza también para este tipo de cachés el término de **caché de víctimas** (*victim cache*).

	Caché de datos L1				**Caché unificada L2**			
procesador	tamaño (KB)	línea (bytes)	líneas cjto.	T_{acceso} (ciclos)	tamaño (KB)	línea (bytes)	líneas cjto.	T_{acceso} (ciclos)
ARM Cortex-A8	32	64	4	1	128-1MB	64	8	11
intel Core i7	32	64	8	4	256	64	8	12
Amd Opteron	64	64	2	3	512	64	16	12
Power8	64	128	8	3	512	128	8	12

Tabla 3: Características de memorias caché de primer y segundo nivel.

La incorporación de un segundo nivel de caché obliga a modificar la expresión del $T_{efectivo}$ anterior. Denominando L1 y L2 a las cachés de primer y segundo nivel respectivamente, dicha expresión quedaría de la forma siguiente:

$$T_{efectivo} = Hr_{L1} \times T_{acierto}L1 + (1 - Hr_{L1}) \times (Hr_{L2} \times T_{acierto}L2 + (1 - Hr_{L2}) \times T_{fallo}L2)$$

En los sistemas con varios niveles de caché se suele distinguir entre dos tipos de tasas de fallos [HP07]:

- La **tasa de fallos local** $(1 - Hr_L)$, que se define como el número de fallos en el nivel de caché considerado, dividido entre el número de accesos a dicho nivel.

- La **tasa de fallos global** $(1 - Hr_G)$, que se define como el número de fallos en el nivel considerado, dividido entre el número total de accesos realizados por el procesador. Para un sistema con dos niveles de caché:

$$(1 - Hr_G)L2 = (1 - Hr_L)L1 \times (1 - Hr_L)L2$$

En el caso de la caché de primer nivel ambas tasas coinciden, ya que todos los accesos emitidos por el procesador pasan por ella.

Para evaluar la eficiencia de la caché de segundo nivel se utiliza la tasa de fallos global, debido a que refleja el porcentaje de los accesos realizados por el procesador que llegan a la memoria principal.

Ejercicio 5.- *Suponiendo que de cada 100 accesos de lectura 8 producen fallo en L1 y, de estos últimos, 5 producen acierto en L2, calcule las tasas de fallos local y global de ambos niveles de caché. Suponiendo que los tiempos de acceso son 2 ciclos para la caché L1, 12 ciclos para la caché L2 y 80 ciclos para la Mp, calcule el tiempo de acceso efectivo utilizando la política de lectura "'out of order fetch"'. ¿cuál sería este tiempo si no se dispusiera de la caché de segundo nivel?*

Para la caché L1:

$$(1 - Hr_L)L1 = (1 - Hr_G)L1 = 8/100 = 8\,\% \rightarrow Hr_L L1 = 92\,\%$$

La tasa de fallos local de la caché L2 es:

$$(1 - Hr_L)L2 = (8 - 5)/8 = 37,5\,\% \rightarrow Hr_L L2 = 62,5\,\%$$

Mientras que la tasa global de L2 es:

$$(1 - Hr_G)L2 = (8 - 5)/100 = 3\,\% = (1 - Hr_L)L1 \times (1 - Hr_L)L2$$

Esto significa que sólo el 3 % de todos los accesos van a Mp, mientras que el resto se realizan, en un tiempo considerablemente menor, a través de alguna de las dos cachés.

El tiempo de acceso efectivo utilizando ambos niveles de caché es:

$$T_{efectivo} = 0,92 \times 2 + 0,08 \times (0,625 \times (2 + 12) + 0,375 \times (2 + 12 + 80)) = 5,36\ ciclos$$

mientras que sin la caché de segundo nivel sería:

$$T_{efectivo} = 0,92 \times 2 + 0,08 \times (2 + 80) = 8,4\ ciclos$$

Como se puede comprobar, la utilización de la caché de segundo nivel supone una reducción de aproximadamente un 36 % en el tiempo de acceso efectivo.

Finalmente, hay que mencionar que muchos de los procesadores *multicore* actuales disponen de un tercer nivel de caché, compartida por todos los *cores*, cuyo tamaño está comprendido entre 8MB y 32 MB, y cuyo tiempo de acceso está en torno a los 35 ciclos.

3.7.4. Memorias caché no bloqueantes.

En los fallos de lectura, aún utilizando la política *out of order fetch*, el procesador debe esperar a recibir la palabra solicitada, en el peor de los casos desde la memoria principal, para poder continuar.

El objetivo de las memorias caché **no bloqueantes** [HP12] es que el procesador pueda continuar con la ejecución de instrucciones mientras se sirve un fallo de caché, aún no disponiendo del dato que causó el fallo . Este tipo de cachés la utilizan los procesadores capaces de ejecutar las instrucciones en distinto orden en el que aparecen en el programa (*out of order execution*), como es el caso de muchos de los procesadores superescalares actuales, que pueden ejecutar varias instrucciones por ciclo.

A modo de ejemplo, y para ilustrar de forma sencilla la diferencia entre este tipo de cachés y las tradicionales cachés bloqueantes, supongamos la siguiente secuencia de instrucciones:

```
1)  ld r1, #0[r10]    Fallo en caché de datos
2)  ld r2, #0[r11]    Acierto en caché de datos
3)  sub r3,r2,r4
4)  add r3,r3,r5
5)  add r6,r3,r1
    . . . . . .
```

En un procesador cuya caché fuese bloqueante, el fallo en la primera instrucción provocaría una espera por parte del procesador hasta disponer del dato, momento en el que podría continuar con la ejecución de esta instrucción y las siguientes. Utilizando una caché no bloqueante, dicho fallo no produciría esta espera sino que el procesador continuaría ejecutando las instrucciones 2, 3 y 4, que no necesitan para su ejecución el valor de dicho dato. Únicamente debería esperar si al ejecutar la instrucción 5, que sí lo necesita, dicha información aún no estuviese disponible. El uso de memorias caché no bloqueantes permite, por lo tanto, solapar la ejecución de instrucciones con el servicio de los fallos de caché.

4. Memoria principal

Constituye el nivel de la jerarquía de memorias más lento, en relación con la velocidad del procesador, exceptuando los dispositivos de almacenamiento secundario. Los principales parámetros que dan una medida de su rendimiento son los siguientes:

- Tiempo de acceso o **latencia**: Es el tiempo que la memoria emplea en completar una operación de lectura/escritura.
- **Tiempo de ciclo**: Es el tiempo que transcurre desde que la memoria comienza a atender una petición hasta que puede atender la siguiente. En las memorias dinámicas, como suele ser el caso de la memoria principal, este tiempo es algo mayor que el tiempo de acceso.
- **Ancho de banda** o *bandwidth*: Es la cantidad de información (generalmente expresada en bytes) que se puede transferir a o desde la memoria por segundo. Por ejemplo, el ancho de banda de una memoria con un tiempo de ciclo de 80 ns y organizada en palabras de 64 bits será de 100 MB/s

Para mejorar el rendimiento de la memoria principal se pueden seguir dos procedimientos: disminuir su latencia, o aumentar su ancho de banda. Generalmente, desde el punto de vista

de organización, es más sencillo aumentar el ancho de banda que disminuir la latencia, puesto que este segundo aspecto depende fundamentalmente de la tecnología, así como del tamaño de los *chips* utilizados para construir la memoria.

Existen básicamente dos formas de aumentar el ancho de banda de la memoria:

- Incrementar el ancho de palabra y, por lo tanto, del bus.
- Organizar la memoria en varios módulos a los que se pueda acceder concurrentemente y, de este modo, se puedan leer o escribir varias palabras a la vez. Esta organización se denomina **entrelazada** (*interleaved*).

En esta sección se trata únicamente este último mecanismo que, a su vez, se puede aplicar a otros niveles de la jerarquía.

4.1. Memoria entrelazada.

Consiste en dividir la memoria en un conjunto de módulos a los que se puede acceder concurrentemente. Su finalidad es conseguir acceder a tantas palabras como módulos, en el tiempo que se tardaría en acceder a una sola palabra en una organización no entrelazada.

Evidentemente, este tipo de organización sólo es útil en procesadores que pueden aprovechar este incremento de ancho de banda. Entre ellos cabe citar los procesadores superescalares, los procesadores que disponen de instrucciones y unidades funcionales vectoriales, los multiprocesadores y los sistemas con memoria caché.

Aunque, en teoría, una memoria entrelazada con K módulos permite multiplicar por K el ancho de banda, lo habitual es que no se llegue a alcanzar este máximo. Esto puede ser debido, bien a que no se generen suficientes peticiones como para tener a todos los módulos trabajando concurrentemente, o bien a los conflictos que se producen cuando varias peticiones, que se generan en un determinado instante, corresponden al mismo módulo. Es deseable, por lo tanto, que las peticiones realizadas por el procesador correspondan a módulos diferentes, para aprovechar así el paralelismo que proporciona esta organización. En consecuencia, un aspecto muy importante en el diseño de un sistema con memoria entrelazada, es la forma en que se distribuyen las direcciones entre los diferentes módulos. Trataremos seguidamente esta cuestión.

Por último, hay que hacer notar que la organización entrelazada permite conseguir el rendimiento (ancho de banda) de memorias de tecnología rápida, utilizando módulos de memoria de tecnología más lenta y, por tanto, de menor coste.

4.1.1. Tipos de entrelazado.

Existen diferentes tipos de entrelazado, dependiendo de los siguientes criterios:

a. La forma en que se distribuye el mapa de direcciones entre los distintos módulos. Según este criterio, el entrelazado puede ser de **orden inferior** o de **orden superior.**

b. El modo en que se realiza el acceso a los módulos. Según este criterio se distingue entre entrelazado **simple** o **complejo**.

Para describir el funcionamiento de los diferentes tipos de entrelazado, supondremos una memoria de $N = 2^n$ palabras, compuesta por K módulos y, para simplificar, direccionamiento a nivel de palabra. Por lo tanto, se necesitan $m = log_2 K$ bits para identificar el módulo al que se dirige la petición de acceso. El número de módulos de una memoria entrelazada es siempre una potencia de dos.

Entrelazado de orden inferior.

Consiste en asignar direcciones de memoria consecutivas a módulos consecutivos. Se denomina de orden inferior porque son los m bits menos significativos de la dirección los que se utilizan para seleccionar el módulo, mientras que los n-m bits restantes se usan para seleccionar la palabra dentro del módulo correspondiente. En la figura 12 se muestra la distribución del mapa de direcciones para cuatro módulos:

	Direcciones:
Módulo 0:	$0, 4, 8, 12, \ldots., 2^n - 4$
Módulo 1:	$1, 5, 9, 13, \ldots., 2^n - 3$
Módulo 2:	$2, 6, 10, 14, \ldots., 2^n - 2$
Módulo 3:	$3, 7, 11, 15, \ldots., 2^n - 1$

De esta forma, una determinada dirección A estará asignada al módulo (A *módulo 4*).

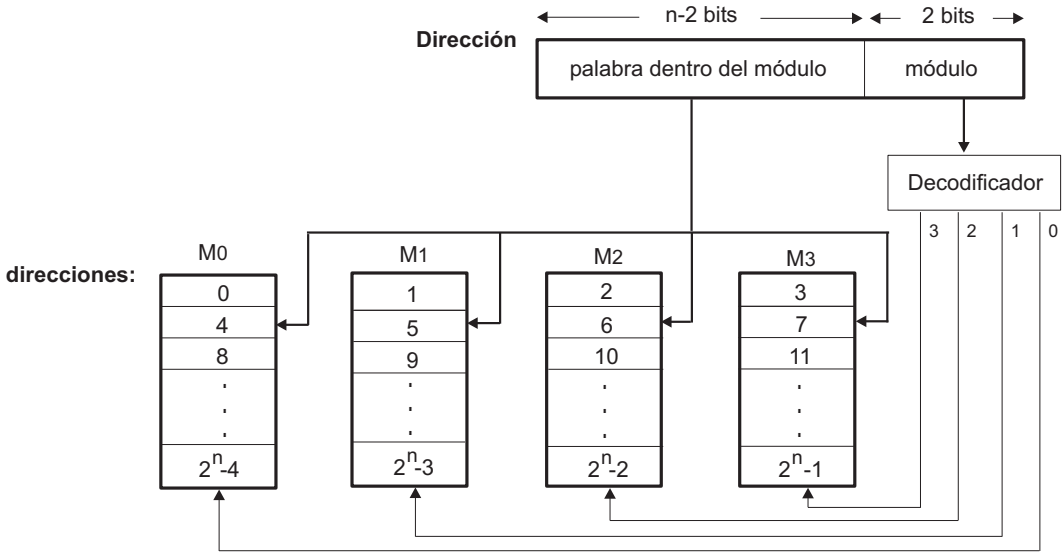


Figura 12: Entrelazado de orden inferior: distribución de direcciones con 4 módulos.

Entrelazado de orden superior.

El mapa de direcciones se distribuye entre los distintos módulos de modo que cada uno tiene asignadas N/K direcciones consecutivas. Se denomina de orden superior porque son los m bits superiores de la dirección los que se utilizan para seleccionar el módulo, mientras que los n-m bits restantes se usan para seleccionar la palabra dentro de éste. Por lo tanto, el módulo i tendrá asignadas las direcciones consecutivas $i \times 2^{n-m}$ a $(i+1) \times 2^{n-m} - 1$.

En la figura 13 se muestra un ejemplo para 4 módulos. En este caso se asignan a cada módulo 2^{n-2} direcciones consecutivas:

	Direcciones:
Módulo 0:	0 a $2^{n-2} - 1$
Módulo 1:	2^{n-2} a $2 \times 2^{n-2} - 1$
Módulo 2:	2^{n-1} a $3 \times 2^{n-2} - 1$
Módulo 3:	$3 \times 2^{n-2}$ a $4 \times 2^{n-2} - 1$

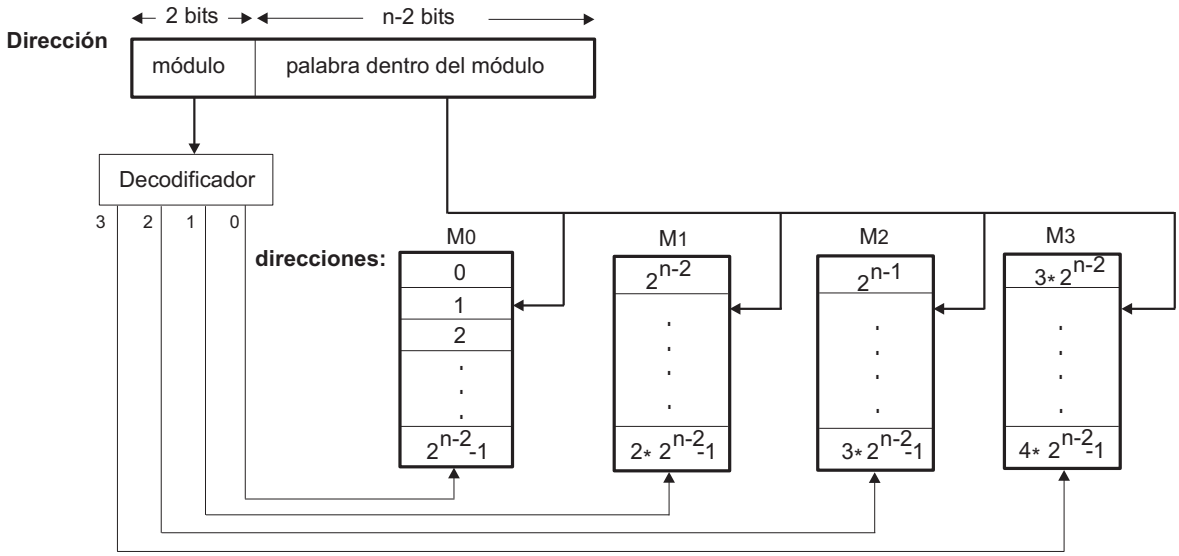


Figura 13: Entrelazado de orden superior: distribución de direcciones con 4 módulos.

La decisión de elegir entrelazado de orden inferior o superior depende la traza de ejecución. En general, el entrelazado inferior resulta muy útil cuando se accede a direcciones contiguas ya que, al estar asignadas a distintos módulos, se aprovecha el acceso en paralelo a todos ellos. Este es el caso de la ejecución secuencial de instrucciones o de las operaciones sobre vectores, ya que tanto las instrucciones de un programa como los elementos de un vector ocupan posiciones de memoria contiguas. Por ello, es también de gran utilidad en los procesadores con unidades funcionales vectoriales, en los que se necesita disponer del mayor número posible de componentes en el menor tiempo posible, dado que las operaciones se realizan sobre este tipo de estructuras de datos. También es muy útil en los sistemas con memoria caché, en los que el intercambio de información entre ambos dispositivos se realiza a nivel de bloques de palabras contiguas.

El entrelazado de orden superior se puede utilizar en sistemas multiprocesador de memoria compartida, para reducir las colisiones de los distintos procesadores en los accesos a memoria.

Por último, pueden combinarse ambos tipos de entrelazado, organizando la memoria en bancos a los que se asigna un rango de direcciones consecutivas (orden superior) y cada banco, a su vez, en módulos entre los que se distribuyen las direcciones empleando entrelazado de orden inferior. La figura 14 muestra un ejemplo de esta organización con 2 bancos de 4 módulos cada uno.

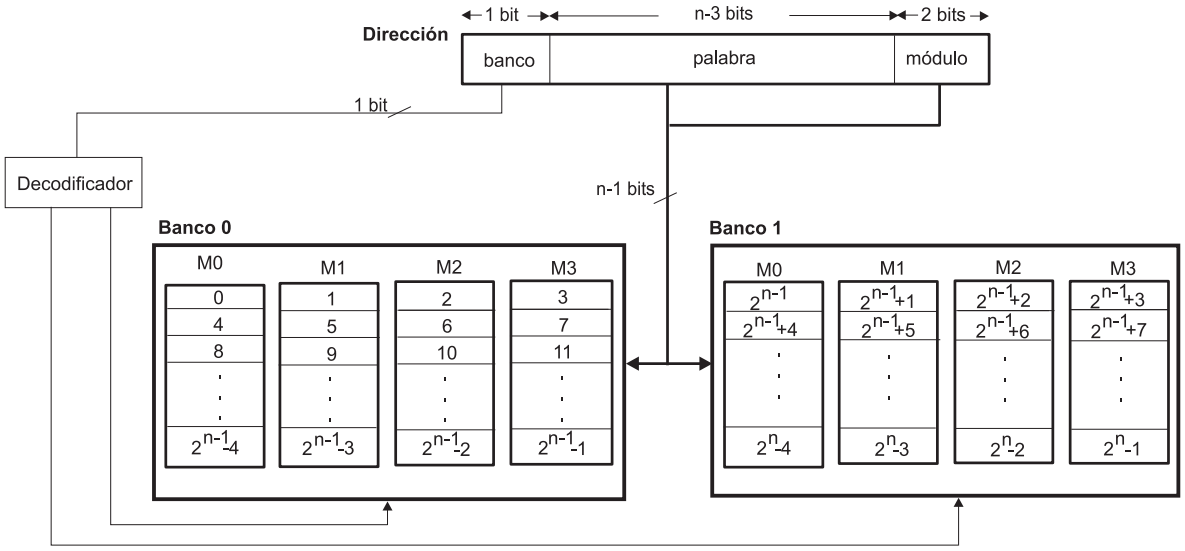


Figura 14: Combinación de entrelazado de orden superior e inferior.

Entrelazado simple.

Es aquél en el que se accede a todos los módulos simultáneamente, empleando la misma dirección, lo cual implica la utilización de entrelazado de **orden inferior**. En la figura 15 se muestra su funcionamiento. Con los *n-m* bits superiores de la dirección se accede a la misma palabra dentro de cada módulo. Si, por ejemplo, el acceso fuese de lectura, se conseguiría tener disponibles K palabras consecutivas al cabo de un tiempo T_a, que es el máximo ancho de banda que puede proporcionar esta memoria, siendo T_a el tiempo de acceso de cada módulo. Las palabras leídas en cada módulo se almacenan simultáneamente en sus correspondientes registros de datos, R_0 a R_{K-1}, utilizándose un multiplexor para gobernar la salida de las K palabras al bus, suponiendo un bus de una palabra.

Para este tipo de entrelazado es necesario utilizar un bus con capacidad de realizar **transferencias de bloque**, siendo el tamaño del bloque igual al número de módulos. La colocación de los registros de datos permite, como se puede observar en el diagrama de tiempos de la figura 16, solapar la transferencia de los datos leídos ($D_0, D_1 ..., D_{K-1}$), con un nuevo acceso a otro bloque de memoria.

Esta configuración es muy útil en los casos, anteriormente citados, en que los accesos a memoria corresponden a direcciones contiguas. Sin embargo, su utilidad disminuye en el caso de accesos no secuenciales. En la página 34 se plantea un ejemplo que ilustra este comportamiento.

Entrelazado complejo.

En este caso, los accesos a los distintos módulos se solapan en el tiempo, por lo que se utilizan direcciones distintas para acceder a cada uno. Así mismo, las direcciones no tienen por qué ser consecutivas, pudiéndose utilizar indistintamente con entrelazado de **orden superior o inferior**. En la figura 17 se muestra su estructura, en la que se ha empleado entrelazado de orden inferior. Los registros de dirección incorporados a cada módulo, R_0 a R_{K-1}, sirven para almacenar la dirección a la que se quiere acceder dentro de éste. Además, se necesita una lógica

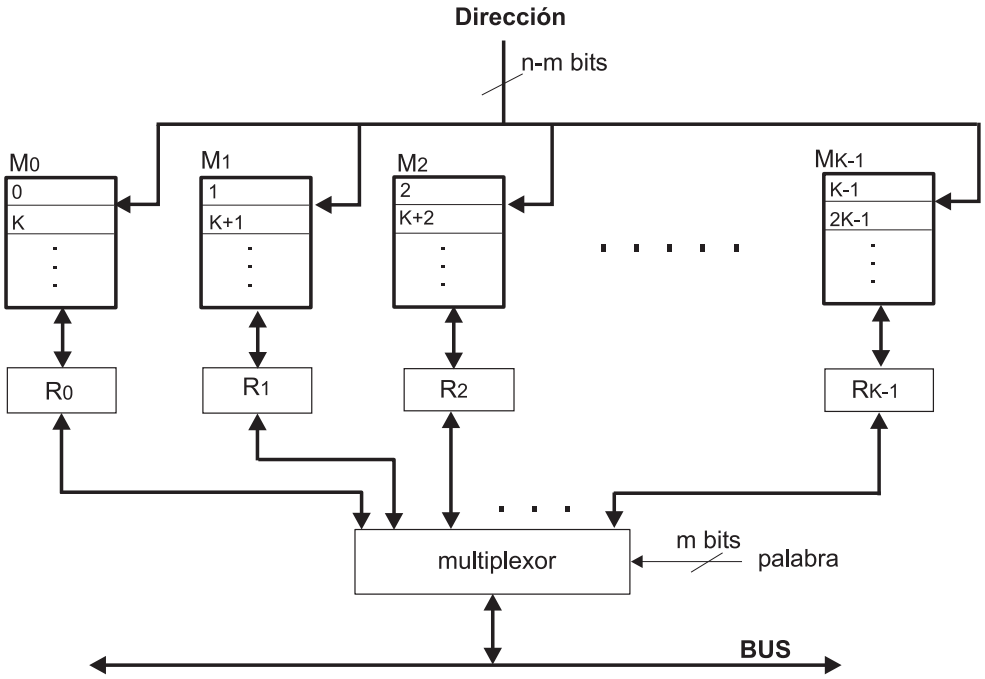


Figura 15: Modo de acceso en el entrelazado simple.

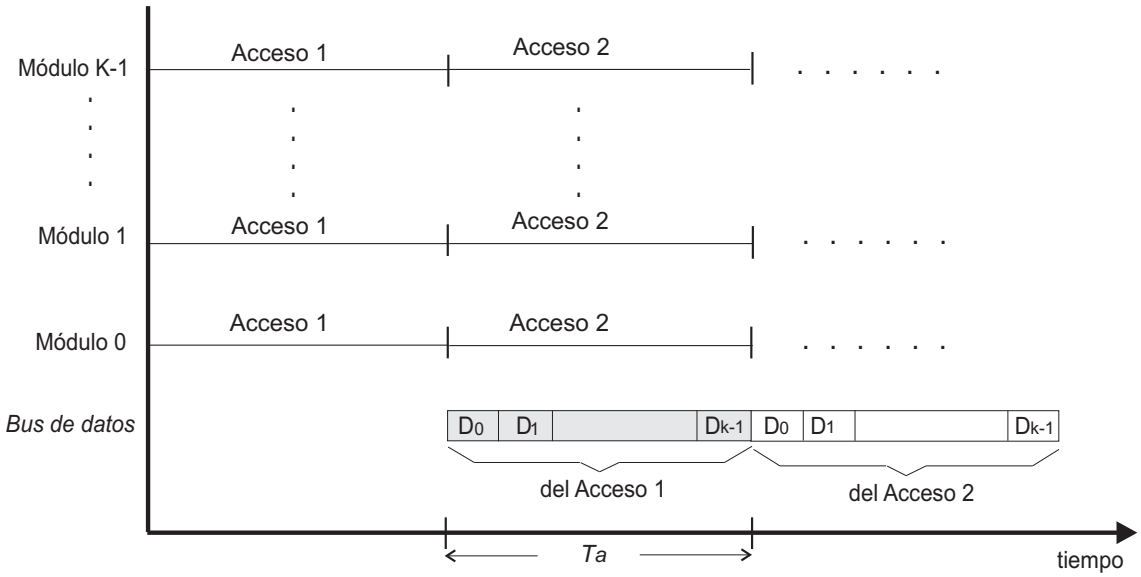


Figura 16: Diagrama de tiempos del entrelazado simple.

de control que se encargue de recibir las peticiones de acceso y de enviarlas a los módulos correspondientes, así como de encolar aquellas que vayan dirigidas a módulos ocupados con una petición anterior, y que se irán sirviendo a medida que éstas vayan siendo atendidas. También es necesario que el bus que conecta la memoria con el resto del sistema sea de **ciclo partido** (*split transaction*). De este modo se puede realizar una petición a memoria antes de haberse completado otra anterior.

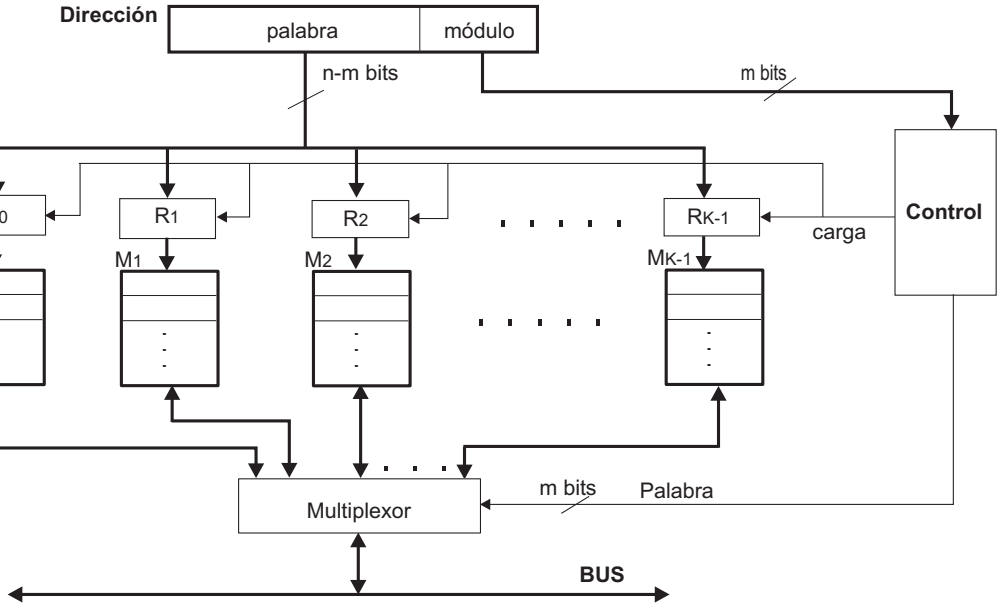


Figura 17: Modo de acceso del entrelazado complejo.

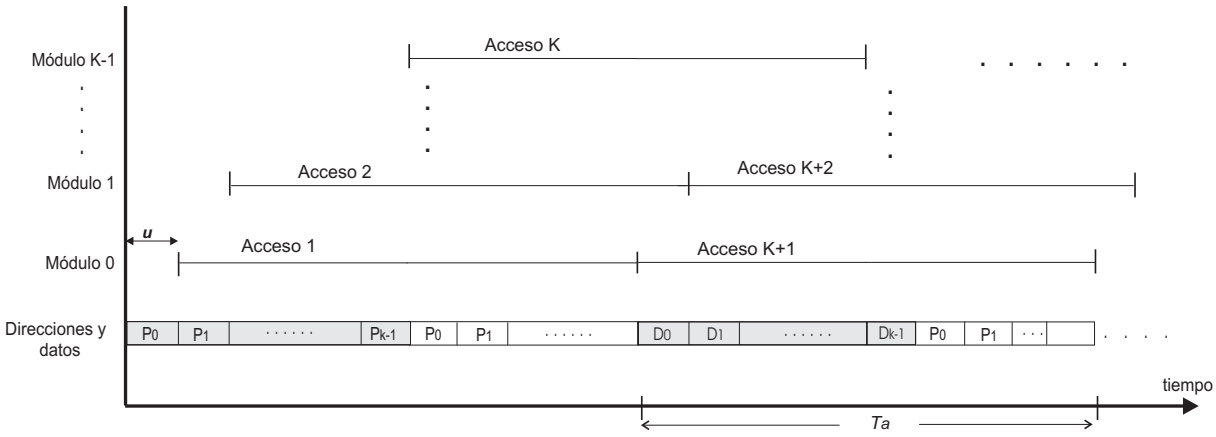


Figura 18: Diagrama de tiempos del entrelazado complejo.

En el diagrama de tiempos de la figura 18 se muestra un ejemplo de cómo se produce el solapamiento de sucesivos accesos de lectura, en el que las peticiones $P_0, P_1, ..., P_{K-1}$ corresponden a los módulos 0, 1, ..., K-1. Así mismo, se puede ver cómo se pueden realizar nuevas peticiones, que se atienden una vez quedan libres los módulos a los que van dirigidas, que previamente dejan en el bus los datos leídos $D_0, D_1, ..., D_{K-1}$. El rendimiento óptimo, K palabras en un tiempo T_a, se logra únicamente cuando K accesos consecutivos corresponden a módulos distintos. En caso contrario, se producen colisiones en el acceso a determinados módulos, lo que imposibilita alcanzar el ancho de banda máximo de la memoria. Así mismo, la duración de las rodajas (*slots*) del bus deberá ser $u \leq T_a/2K$.

Ejercicio 6.- *Suponiendo una memoria con entrelazado de orden inferior y 8 módulos, en la que el tiempo de acceso de cada módulo es de 50 ns y las palabras son de 64 bits. Calcular el ancho de banda que se obtiene de esta memoria en el acceso a los elementos de un vector* `w`*, de la forma que se indica a continuación, utilizando a) entrelazado simple y b) entrelazado complejo.*

```
for (i=0; i<1000; i=i+3)
        a = a + w[i];
```

Se supondrá, para simplificar, que el vector está almacenado a partir de la dirección 0 y que sus elementos ocupan una palabra.

Como el tiempo de acceso es 50 ns, el ancho de banda de cada módulo es:

$$8\ bytes/50\ ns = 160\ MB/s$$

A continuación se muestran las direcciones de los elementos del vector a los que se accede, así como los módulos a los que corresponden dichas direcciones:

Dirección	0	3	6	9	12	15	18	21	24	27	30	33	...
Módulo	0	3	6	1	4	7	2	5	0	3	6	1	...

a) Entrelazado simple: Con la dirección de comienzo del vector se accede a los ocho módulos, leyéndose los ocho primeros elementos, que corresponden a las direcciones 0, 1, 2, 3, ..., 6 y 7, de los que se utilizan únicamente tres, los correspondientes a las direcciones 0, 3 y 6. El ancho de banda obtenido en este acceso se reduce a 3 palabras en 50 ns, en lugar de 8 que sería el máximo teórico. El siguiente acceso se realiza con la dirección 9, obteniéndose los elementos correspondientes a las direcciones 8, 9, ..., 14 y 15, de los que se utilizan de nuevo únicamente tres (direcciones 9, 12 y 15).

Este comportamiento se repite, obteniéndose en consecuencia un ancho de banda de:

$$\frac{3 \times 8\ bytes}{50\ ns} = 3 \times 160 MB/s = 480 MB/s$$

b) Con entrelazado complejo, ya que las ocho referencias sucesivas corresponden a módulos distintos, se obtendrá el máximo ancho de banda: 8 palabras cada 50 ns, esto es:

$$\frac{8 \times 8\ bytes}{50\ ns} = 8 \times 160 MB/s = 1,28 GB/s$$

5. Memoria virtual

En la sección 3 hemos visto cómo el uso de las memorias caché permite el acceso rápido a las zonas de código y datos más utilizadas. En esta sección veremos cómo la memoria principal puede actuar como una caché de la memoria secundaria, habitualmente implementada mediante discos magnéticos. Esta técnica se conoce con el nombre de memoria virtual. Históricamente los motivos principales de la aparición de la memoria virtual fueron: permitir compartir de forma eficiente y segura la memoria principal entre varios programas, y eliminar los problemas de programación debidos a su limitada capacidad, aunque en la actualidad es el primero de ellos el que justifica principalmente su utilización.

Antecedentes.- En los primeros computadores la memoria principal era cara y de escasa capacidad, de modo que si un programa y sus datos excedían su capacidad, el programador se veía forzado a dividir dicho programa, residente en una memoria secundaria de mayor capacidad, en una serie de fragmentos (*overlays*), cada uno de los cuales debía caber en memoria. El propio programa tenía que incluir las instrucciones de entrada/salida para cargar en memoria los fragmentos necesarios en cada momento y desalojarlos a la memoria secundaria cuando dejaban de serlo.

Para ejecutar el programa, se cargaba el primer fragmento en memoria y se ejecutaba. Cuando acababa este fragmento se desalojaba, se cargaba el siguiente, se le daba control y así hasta la finalización del programa. Esta técnica, denominada *Overlay*, era incómoda para el programador, que debía identificar los fragmentos en los que dividir su programa y planificar el trasiego de información entre la memoria principal y la secundaria, con el inconveniente adicional de hacer los programas dependientes de la memoria disponible, por lo que variaciones en su configuración podían obligar a revisar la división realizada.

La aparición de la **multiprogramación** planteó problemas adicionales, ya que en este caso eran varios los programas que debían residir simultáneamente en la memoria principal, por lo que era necesario **asignar** espacio de memoria a cada uno de ellos, y dotar al sistema de mecanismos que les permitiesen **compartir** información, así como **proteger** la zona de memoria utilizada por cada programa de accesos no autorizados por parte de los demás. El mecanismo de protección más sencillo consistía en dotar al procesador de dos registros, para establecer las fronteras de la zona de memoria ocupada por cada programa. De este modo, para comprobar si un acceso estaba permitido, bastaba con comprobar si la dirección estaba dentro de la zona delimitada por dichos registros y, de no ser así, notificar que dicho acceso no podía realizarse. El sistema operativo era el encargado de cambiar el contenido de estos registros cada vez que daba el control de la CPU a un nuevo programa.

Otro aspecto muy importante a tener en cuenta es el de la **reubicación dinámica** de los programas, que permite que un programa pueda ejecutarse en cualquier posición de memoria principal en la que se cargue. Antes de aparecer la memoria virtual, el procedimiento utilizado consistía en utilizar un registro del procesador (registro de base) que contenía la posición de memoria a partir de la cual estaba cargado el programa en ejecución, calculándose todas las direcciones relativas a este registro. De este modo, si por cualquier motivo dicho programa era desalojado de la memoria y posteriormente volvía a ella, podía cargarse en otra zona distinta a la que ocupaba anteriormente: bastaba con que el sistema operativo cargase en el registro de base la nueva dirección de comienzo.

En este entorno, los diseñadores del computador Atlas en la Universidad de Manchester desarrollaron en los años 60 un mecanismo que permitía realizar los *overlays* de forma automática, liberando al programador de su gestión. Este mecanismo, denominado **memoria virtual,** consistía básicamente en diseñar el sistema de memoria como una **jerarquía de dos niveles**: el nivel de la memoria principal y otro nivel, de mucha mayor capacidad y mayor tiempo de acceso, soportado por un tambor magnético, capaz de dar cabida a la totalidad del programa a ejecutar junto con sus datos (en el caso del Atlas, la capacidad de ambos niveles era de 16K y 96K palabras, respectivamente). El objetivo era que el programador pudiera utilizar este computador como si la capacidad de memoria disponible fuese la del tambor magnético.

La idea básica de partida de la memoria virtual consiste en diferenciar los conceptos de **espacio de direcciones lógicas o virtuales** y **espacio de direcciones físicas**. El espacio de direcciones lógicas es el conjunto de direcciones generadas por las instrucciones máquina que ejecuta el procesador, por lo que se puede decir que son las direcciones que "ve" el programa. El espacio de direcciones físicas es el conjunto de direcciones de que dispone realmente su memoria principal.

Para ilustrar estos conceptos, supongamos un hipotético procesador con una memoria principal de 16 GB, cuyas instrucciones generan direcciones de 64 bits. Un programa que se ejecutase en este computador podría direccionar hasta 2^{64} *bytes*, por lo que el espacio de direcciones lógicas es, en este caso, de 16 EB, mientras que el espacio de direcciones físicas disponible es de 16 GB, notablemente inferior[2].

La memoria virtual es un mecanismo transparente al programador que le permite ejecutar sus programas sin necesidad de tener en cuenta la capacidad de la memoria principal del computador, dándole así la ilusión de que dispone de un espacio de memoria prácticamente ilimitado. Esto se consigue permitiendo que el programa opere en el espacio de direcciones lógicas, aunque es necesario **traducir** estas direcciones a direcciones de memoria principal, que es donde finalmente el procesador accede a la información.

Por otra parte, resuelve automáticamente el problema de la **reubicación dinámica** de los programas y la gestión del trasiego de información entre la memoria principal y la secundaria, manteniendo en la memoria principal **sólo la información necesaria en cada momento**. Finalmente, proporciona los mecanismos para **proteger** y **compartir** información, necesarios cuando varios programas coexisten en el computador.

Para su implementación se utilizan dos niveles de la jerarquía de memorias: la memoria principal y un dispositivo de almacenamiento secundario, generalmente un disco, debiéndose abordar una serie de problemas como son:

- La traducción de las direcciones virtuales a direcciones físicas.
- La gestión de los fallos (accesos a información que no se encuentra en memoria principal) y las políticas a emplear para dicha gestión (reemplazo, ubicación, etc.).
- La asignación de memoria física a cada uno de los programas, entre los que se reparte a su vez la utilización del procesador.

En los siguientes apartados trataremos en profundidad estos problemas así como su resolución, que exige una cooperación entre el *hardware* del computador y el sistema operativo.

Por último, hay que mencionar el problema de la asignación del espacio de direcciones virtuales a los programas. Habitualmente se diferencian cuatro zonas en este espacio; código, datos estáticos, datos dinámicos (*heap*) y pila, cada una de las cuales ocupa un conjunto de direcciones virtuales consecutivas. Las dos primeras zonas son fijas y su tamaño puede calcularse en tiempo de compilación, mientras que las dos últimas son dinámicas, ya que su tamaño varía a lo largo de la ejecución del programa. La figura 19 muestra de forma genérica la asignación del espacio de direcciones virtuales a cada una de las zonas, asignación dependiente del sistema operativo, así como el sentido de crecimiento de las zonas de tamaño variable.

[2] Aunque se suele pensar que el espacio de direcciones lógicas es siempre mayor que el de direcciones físicas, puede suceder también lo contrario. Podemos plantearnos el caso en el que las direcciones generadas por las instrucciones fuesen de 32 bits.

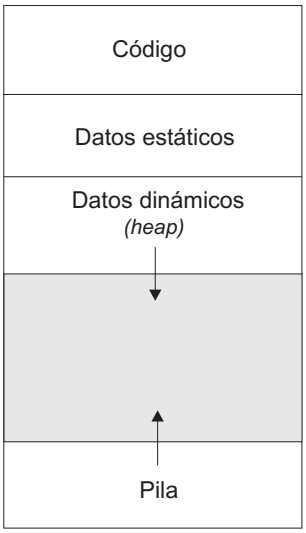


Figura 19: Asignación del espacio de direcciones virtuales.

5.1. Traducción de direcciones.

La traducción de direcciones es necesaria, ya que las direcciones que generan los programas, y por lo tanto el procesador, se refieren al espacio de direcciones virtual, mientras que el acceso a las instrucciones y datos que se necesitan en cada momento se realiza a través de la memoria principal.

Para centrar el problema de la traducción supongamos el ejemplo planteado anteriormente, de un espacio de direcciones virtuales de 16 EB (2^{64} *bytes*) y un espacio de direcciones físicas de 16 GB (2^{34} *bytes*). El procedimiento de traducción consiste aquí en convertir las direcciones lógicas de 64 bits en las correspondientes direcciones físicas de 34 bits.

Se trata, por lo tanto, de definir y realizar una función **F** tal que, para toda dirección virtual, le haga corresponder una dirección física, o bien devuelva un indicador de que la información a la que se hace referencia no está en memoria principal:

F(x) = **z**, si la información correspondiente a la dirección virtual **x** está en memoria principal en la dirección **z**.

F(x) = $\emptyset$, si la información no se encuentra en memoria principal. Se dice entonces que ha ocurrido un **fallo**.

La forma más sencilla de realizar esta función sería utilizar una tabla con tantas entradas como posibles direcciones virtuales, de forma que si **F(x) = z**, la entrada **x** de la tabla contuviese la dirección física **z**, y $\emptyset$ en caso contrario. Sin embargo, la realización de la función de traducción considerando direcciones individuales es claramente inviable, debido al excesivo espacio necesario para almacenar la información de traducción.

En la práctica, lo que se hace es dividir los espacios virtual y físico en bloques, de igual tamaño en ambos, estableciéndose la correspondencia entre bloques de direcciones virtuales y bloques de direcciones físicas consecutivas. De esta forma, se reduce la información necesaria para la traducción, ya que las entradas de la tabla se refieren a conjuntos de direcciones

consecutivas, en lugar de a direcciones individuales.

Según como sean los bloques, se distingue entre paginación y segmentación. En la **paginación** los bloques son de tamaño fijo y se denominan **páginas**, mientras que en la **segmentación** son de tamaño variable, denominándose **segmentos**. Existe además un modelo híbrido, la **segmentación paginada**, cuyo objetivo es aprovechar las ventajas de los dos anteriores. La traducción la lleva a cabo un componente *hardware* denominado **MMU** (*memory management unit*). Antes de pasar a estudiar en detalle cada una de estas técnicas, se trata en la siguiente sección el problema de la gestión de los fallos, esto es, el acceso a información que no se encuentra en memoria principal.

5.2. Gestión de los fallos

La gestión de los fallos se lleva a cabo mediante la cooperación entre la MMU y el sistema operativo. La primera, generando una excepción cuando la información solicitada no se encuentra en la memoria principal y el segundo tratando dicha excepción, lo cual consiste básicamente en realizar las siguientes acciones:

1. Suspender la ejecución del proceso en el que se produjo el fallo, ya que la información que necesita no está en memoria principal.
2. Ordenar la transferencia del bloque en el que se encuentra la información demandada, desde la memoria secundaria a la memoria principal.
3. Modificar la función de traducción para reflejar la nueva correspondencia entre la dirección lógica **x** y la dirección física **y** en la que reside ahora.

Dado que la ejecución del proceso no puede continuar hasta que el bloque en el que se encuentra la información que produjo el fallo esté en la memoria principal, el sistema operativo dará paso a otro proceso mientras se realiza su transferencia. Ello requiere salvar previamente el estado del proceso "suspendido"(toda aquella información necesaria para poder continuar posteriormente su ejecución) y cargar en los registros del procesador la información referente al nuevo proceso. Esta tarea se denomina **cambio de contexto**.

A diferencia de lo que ocurre con los fallos de caché, en este caso es necesario el cambio de contexto ya que la operación de transferencia de un bloque involucra el acceso a un dispositivo de almacenamiento secundario, cuya velocidad es mucho menor que la de la memoria principal, por lo que no sería razonable que el procesador permaneciese ocioso durante todo el tiempo que dura dicha operación.

Ejercicio 7.- *Sea un computador cuya memoria virtual se considera dividida en bloques de tamaño fijo de 4 KB y cuya CPU tiene una capacidad de procesamiento de 2000 MIPS. Además tiene un disco con velocidad de transferencia de 100 MB/s y tiempo de posicionamiento de 5 ms. ¿Cuántas instrucciones podría ejecutar la CPU durante el tiempo que tarda en transferirse un bloque de disco a memoria principal?*

El tiempo que se tardaría en llevar un bloque de 4 KB desde el disco a la memoria principal sería aproximadamente:

$$5\ ms + (4.096\ bytes/100.000.000\ bytes/s) \approx 5\ ms$$

Durante ese tiempo, la CPU podría ejecutar:

$$0,005\ s \times 2.000 \times 10^6\ inst/s = 10.000.000\ instrucciones.$$

Otro hecho a tener en cuenta es la forma de continuar la ejecución del proceso en el que se produjo el fallo, una vez tenga la información que necesita en memoria principal. Ya que el fallo puede ocurrir en cualquiera de las fases de la ejecución de una instrucción en la que se acceda a memoria, existen dos formas de continuar la ejecución: reiniciar la instrucción en la que se produjo el fallo, o continuar en el punto en el que quedó interrumpida.

El **reinicio** de la instrucción es la solución más utilizada. En este caso, el sistema debe ser capaz de reconstruir el estado del procesador tal y como estaba al comienzo de la instrucción. Si el fallo se produjo en la fase de *fetch*, la solución es sencilla: basta con recuperar el contador de programa salvado previamente, y volver a realizar la lectura de la instrucción.

En algunas ocasiones, la reconstrucción del estado puede ser bastante más compleja, como sucede con las instrucciones en las que uno de los operandos (registro, biestable de estado o posición de memoria) se utiliza a la vez como fuente y destino. Pongamos, como ejemplo, una instrucción de suma con acarreo. En este caso, el biestable de acarreo quedará modificado al final de la operación de suma para reflejar el resultado de ésta. Si el fallo se produce después de esta modificación, durante el almacenamiento del resultado en memoria principal, el reinicio de la instrucción requiere que el biestable de acarreo tenga el valor original.

Un problema similar puede darse cuando se usan modos de direccionamiento con auto-incremento y autodecremento. Si en una instrucción como `mov [r2++],[--r3]` se produce un fallo al intentar escribir en la dirección a la que apunta `r3`, para reiniciar la instrucción se deberá partir de los valores que tenían `r2` y `r3` al comienzo de la instrucción. Entre las técnicas utilizadas para posibilitar el reinicio correcto de la instrucción en la que se produce un fallo podemos citar las siguientes:

- Posponer las modificaciones hasta el final de la instrucción, cuando con seguridad no se va a producir fallo.
- Mantener una copia de todos los valores originales de los recursos utilizados por la instrucción, de forma que en caso de fallo basta con estos valores para restaurar el estado inicial del que parte la instrucción.

5.3. Paginación

Consiste en dividir el espacio de direcciones virtuales en bloques de tamaño fijo, denominados **páginas**, y el espacio de direcciones físicas en bloques del mismo tamaño que los anteriores, denominados **marcos de página**, estableciéndose la correspondencia entre páginas y marcos de página. De este modo, la asignación de memoria principal a los procesos se realiza en trozos de tamaño fijo (marcos de página).

Las direcciones virtuales generadas por los programas se interpretan como formadas por dos campos: la página donde se encuentra la información a la que se hace referencia (**PV**) y su posición relativa dentro de la página (**desp**).

El tamaño de las páginas es siempre una potencia de dos (2^k), por lo que la dirección de base de cada marco de página tendrá sus **k** bits menos significativos a cero. Por ello, para traducir una dirección virtual en su correspondiente dirección física basta con traducir la página (PV) al marco de página correspondiente (MP) y añadir los **k** bits restantes (véase la figura 20).

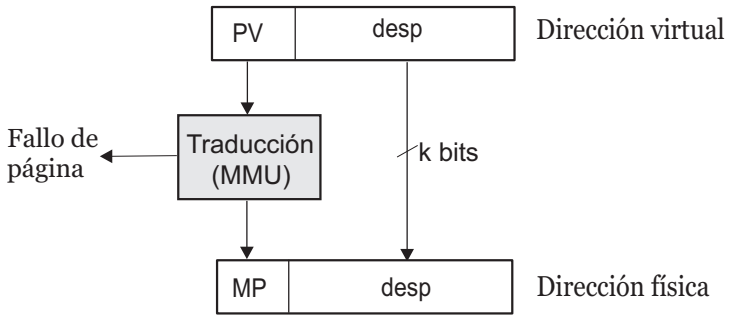


Figura 20: Traducción en la memoria virtual paginada.

Ejercicio 8.- *Suponiendo un computador con palabra de 64 bits, memoria virtual paginada de 256 TB y páginas de 4 KB, cuya memoria física es de 16 GB. ¿Cuántas páginas y marcos de página componen ambos espacios de direcciones? ¿Cuál es el formato de las direcciones virtuales generadas por los programas y cuál el de las direcciones físicas? ¿Cómo se interpreta la dirección virtual 0x000000070008?*

El espacio de direcciones virtual se compone de 2^{48} *bytes*/2^{12} *bytes/página* = 2^{36} páginas, y el de direcciones físicas de 2^{34} *bytes*/2^{12} *bytes/página* = 2^{22} marcos de página.

El formato de las direcciones virtuales, de 48 bits, se compone de dos partes: la página virtual, que corresponde a los 36 bits superiores, y la posición donde se encuentra la información, indicada en los 12 bits inferiores. En el caso de las direcciones físicas, de los 34 bits que las componen, los 22 superiores indican el marco de página y los 12 inferiores la posición dentro de él. A continuación se muestran ambos formatos:

Dirección virtual:

47 — 12	11 — 0
página	byte

Dirección física:

33 — 12	11 — 0
marco de página	byte

Según lo anterior, la dirección virtual 0x000000070008 se interpreta de la forma siguiente:

47 — 12	11 — 0
página	byte
000000070	008

Esto significa que la información requerida por el procesador está en la página 112 (7x16), y dentro de ella en la posición 8, que corresponde a la segunda palabra de la página.

El mecanismo básico para realizar la traducción consiste en utilizar una **tabla de páginas** que consta de una entrada por cada página virtual, en la que se almacena la información necesaria para realizar su traducción:

- Un bit (**R**) para indicar si la página reside o no en memoria principal.

- Si está en memoria principal (R=1), el marco de página (**MP**) donde se encuentra, y si no está (R=0), la dirección de memoria secundaria (**DMs**) donde encontrarla.

Además, cada entrada recoge otros datos relevantes acerca de cada página, que permiten proteger, compartir su información, implementar políticas de reemplazo, etc. Entre ellos cabe destacar los siguientes:

- Los derechos de acceso a su información: lectura, escritura y ejecución (**RWX**).
- Si se ha modificado o no (**M**).
- Si ha sido utilizada recientemente (**U**).
- Si es privada o contiene información compartida con otros procesos (**P**).
- Si la información que contiene debe o no llevarse a la memoria caché (**C**).

Cada proceso tiene su propia tabla de páginas, creada por el sistema operativo, que generalmente reside en la memoria principal. Asimismo, existe un registro (**RBTP**) que contiene la dirección de comienzo de la tabla del proceso en ejecución. El procedimiento básico de traducción, que se muestra en la figura 21, es el siguiente:

1. El valor del campo **PV** de la dirección virtual se utiliza como índice de la tabla, para acceder a la entrada correspondiente a la página a la que se hace referencia.
2. A continuación se comprueba, examinando el bit de residencia **R**, si la página está en memoria principal.
   a) Si está en memoria, se comprueba si el de acceso que se quiere realizar está permitido, según los derechos de acceso de la página. Si es así, se obtiene de la tabla el marco de página en el que se encuentra (**MP**) y se concatena con el campo **desp** de la dirección virtual, para formar la dirección física con la que acceder a la información en la memoria principal. De lo contrario, se genera la excepción correspondiente.
   b) Si la página no está en memoria, se genera una excepción (**fallo de página**) y el sistema operativo se encarga de ordenar su transferencia desde la memoria secundaria (**DMs**) a la memoria principal, reemplazando, si es necesario, alguna de las páginas residentes en ese momento.

La solución planteada, si bien funciona correctamente, presenta sin embargo dos problemas que se tratan en las siguientes secciones: el espacio que ocupan las tablas de páginas, y el tiempo necesario para realizar la traducción.

5.3.1. Tablas de páginas multinivel.

Como se dijo anteriormente, lo habitual es que la tabla de páginas resida en memoria principal, lo cual plantea el problema de la cantidad de memoria necesaria para su almacenamiento.

Para ilustrar el alcance de este problema tomaremos como ejemplo un sistema que maneja direcciones virtuales de 32 bits y páginas de 4 KB. La tabla de páginas de cada proceso debería tener 2^{20} entradas, una por cada posible página virtual (2^{32} *bytes*/2^{12} *bytes*/*página*). Suponiendo que cada entrada ocupase como mínimo una palabra de 4 bytes, el **tamaño de cada tabl**a sería: 2^{20} *entradas* $\times$ 2^2 *bytes*/*entrada* = 2^{22} *bytes* = 4 *MB*.[3]

[3]En algunos casos podría suceder que su tamaño excediera el de la memoria principal disponible. Plantéese qué sucedería si las direcciones virtuales fuesen de 64 bits.

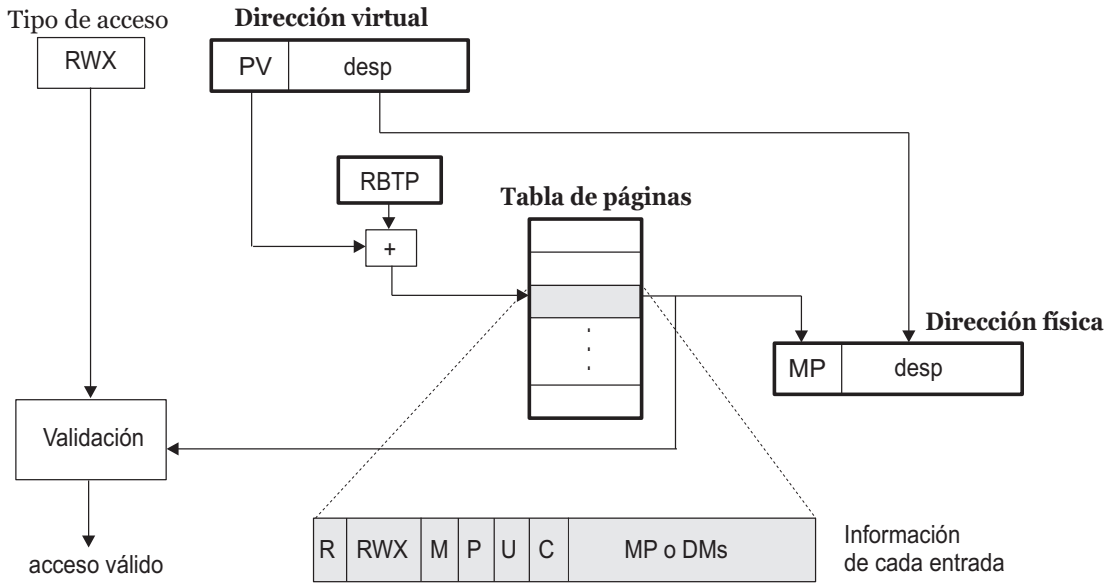


Figura 21: Mecanismo básico de traducción en un sistema con paginación.

Afortunadamente, los procesos no utilizan en general todo el espacio de direcciones virtuales de que disponen en teoría y, además en cada intervalo de tiempo hacen referencia únicamente a un subconjunto de todo el espacio utilizado a lo largo de su ejecución (recordemos la propiedad de **proximidad de referencias**). Por otra parte, el espacio que utilizan puede no ser contiguo y, además puede variar dinámicamente.

Teniendo en cuenta todo lo anterior, almacenar en memoria principal información acerca de todo el espacio virtual que un proceso puede utilizar potencialmente conllevaría un gasto inútil de ésta. La solución empleada para compactar la tabla de páginas es utilizar varios niveles de tablas, que se consultan de forma secuencial.

La figura 22 muestra un posible diseño para el sistema del ejemplo anterior, utilizando dos niveles de tablas de páginas. Conceptualmente se divide el espacio de direcciones virtual en 2^{10} zonas contiguas de 4 MB y, a su vez, cada zona en 2^{10} páginas de 4 KB. Cada entrada de la tabla del nivel 1 contiene información de cada una de las zonas, como por ejemplo si es utilizada o no por el proceso y, de ser así, la dirección de la tabla necesaria para traducir cada una de sus páginas, esto es, la tabla de nivel 2 correspondiente. En las tablas de segundo nivel estará la información de traducción propiamente dicha, así como los bits de modificación, uso de la página, etc.

Suponiendo de nuevo que cada entrada ocupase una palabra, para almacenar cada una de las tablas bastaría con $2^{10}\ entradas \times 2^2\ bytes/entrada = 4$ KB. Esta solución permite tener en memoria principal únicamente la información de traducción de las zonas realmente utilizadas (sombreadas en la tabla de primer nivel de la figura 22) y hace posible la protección y compartición de la información a dos niveles: zonas completas de 4 MB o páginas individuales de 4 KB.

Por otra parte, sólo es necesario tener inicialmente en memoria principal la tabla de nivel 1, ya que a partir de ella se pueden localizar y llevar a memoria principal las tablas de traducción de las zonas de memoria virtual necesarias en cada momento.

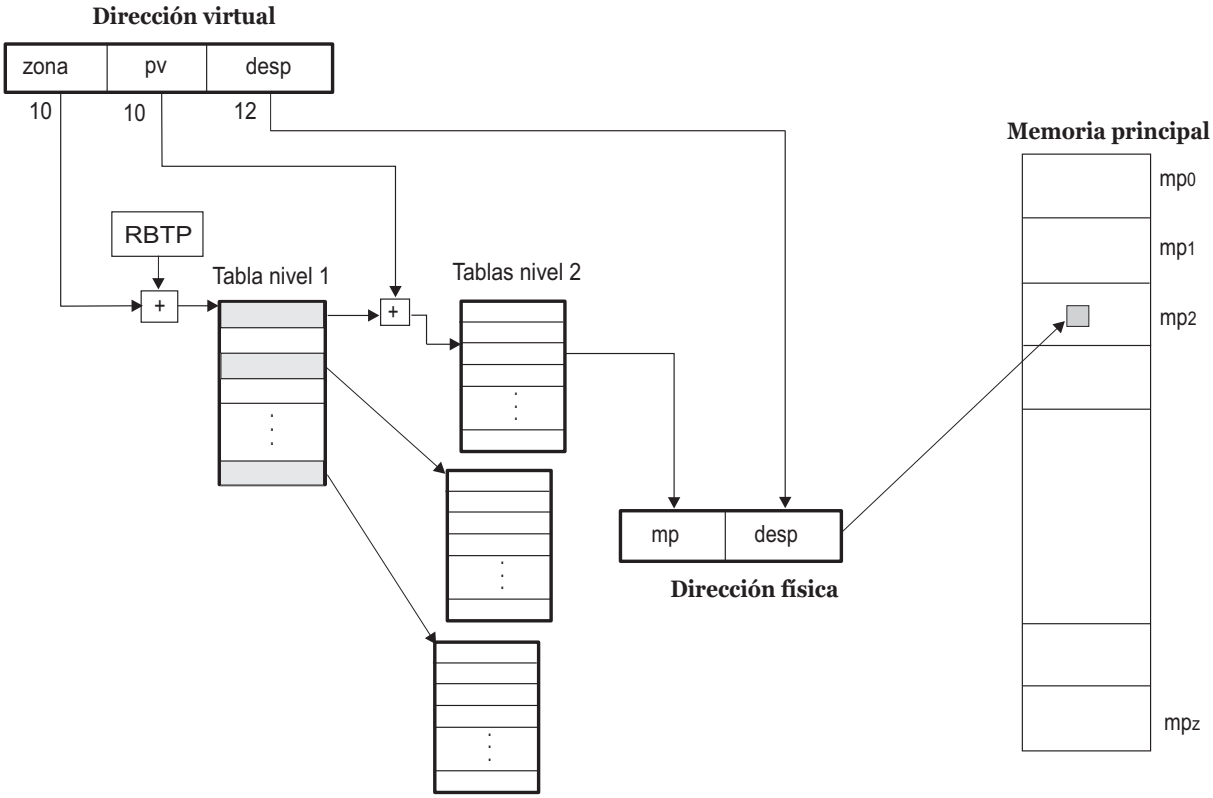


Figura 22: Ejemplo de traducción con 2 niveles de tablas de páginas.

En los sistemas actuales se utilizan entre 3 y 4 niveles de tablas, debido a la tendencia a aumentar el tamaño de las direcciones virtuales. Además, lo habitual que cada tabla ocupe una página, para el mejor aprovechamiento de la memoria principal.

Ejercicio 9.- *Suponga un computador con direcciones virtuales de 32 bits en el que se ejecuta un programa que utiliza únicamente los primeros 8 MB y los últimos 8 KB del espacio virtual de que dispone. Calcule el espacio de memoria principal necesario para almacenar las tablas para la traducción de sus direcciones, teniendo en cuenta el mecanismo de traducción de la figura 22, y que cada una de las entradas de las tablas ocupa 4 bytes. Indique finalmente cómo se interpretaría en este sistema la dirección virtual 0x00403008, así como las tablas y entradas necesarias para su traducción.*

Los primeros 8 MB utilizados por el proceso corresponderían a las dos primeras zonas de su espacio virtual, por lo que se utilizarían para su traducción las dos primeras entradas de la tabla de primer nivel y, por lo tanto, dos tablas de segundo nivel (una por cada zona). Los últimos 8 KB (dos páginas) corresponderían a la última zona de su espacio virtual, utilizándose para su traducción la última entrada de la tabla de primer nivel. De la tabla del nivel 2 correspondiente, se utilizarían únicamente las dos últimas entradas.

En total, serían necesarias la tabla de nivel 1 y tres tablas de nivel 2, que ocuparían:

$$4\ tablas \times 2^{10}\ entradas/tabla \times 2^{2}\ bytes/entrada = 16\ KB$$

frente a los 4 MB necesarios si se utilizase solo un nivel de tabla de páginas.

La dirección virtual indicada se interpreta de la forma siguiente:

31 22	21 12	11 0
zona	página	byte
0000 0000 01	00 0000 0011	0000 0000 1000

Hace referencia, por lo tanto, a la tercera palabra (byte 8) de la página 3 de la zona 1. Para su traducción se accedería a la segunda entrada de la tabla de primer nivel, de la que se obtendría la dirección de comienzo de la tabla de nivel 2 correspondiente a dicha zona. En esta última tabla se accedería a la cuarta entrada, donde se conseguiría finalmente el marco de página donde se encuentra la información (si la página está en memoria).

5.3.2. Aceleración de la traducción: la TLB

Una cuestión que surge de la ubicación de las tablas de páginas en la memoria principal es cómo afecta al tiempo de ejecución de los programas. Según el esquema básico de traducción, con un solo nivel de tabla, son necesarios dos accesos a memoria principal para acceder a un dato o instrucción: el primero a la tabla de páginas, para realizar la traducción de su dirección, y el segundo para acceder a la información propiamente dicha. El problema es aún más grave si se utilizan varios niveles de tablas, en cuyo caso son necesarios tantos accesos a memoria como niveles, únicamente para traducir las direcciones.

Si tenemos en cuenta que la ejecución de cada instrucción implica al menos un acceso a memoria para su lectura, además de los accesos a datos que se realizan en la ejecución de algunas instrucciones, esta forma de realizar la traducción aumentaría significativamente el tiempo de ejecución de los programas con respecto a la no utilización del mecanismo de memoria virtual.

La solución para acelerar la traducción se basa en el principio de **proximidad de referencias** de los programas, esto es, si las direcciones a las que se hace referencia un programa tienen esta propiedad, la información de traducción de dichas direcciones también la tendrá. Consiste en utilizar una memoria caché especial de la tabla de páginas que contenga únicamente la traducción de las páginas a las que se ha accedido recientemente. Esta memoria se denomina **TLB** (*translation lookaside buffer*).

La TLB tiene una estructura similar a la de las memorias caché. Las etiquetas contienen números de páginas y la parte de almacenamiento contiene la misma información que las entradas de las tablas de páginas (véase la sección 5.3): traducción, derechos de acceso, información de uso etc. El tipo de correspondencia utilizada, dado su reducido tamaño, suele ser totalmente asociativa o bien asociativa por conjuntos. En la figura 23 se muestra la estructura de cada una de sus entradas para uno y dos niveles de tablas de páginas.

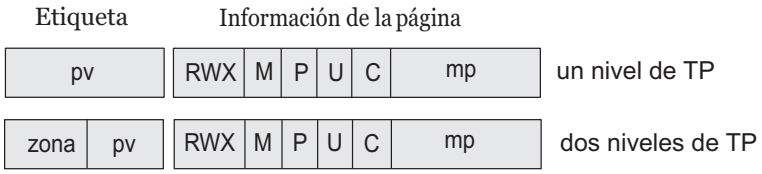


Figura 23: Formato de una entrada de la TLB para uno y dos niveles de tablas de páginas.

Con la incorporación de este dispositivo, la traducción se intenta realizar en primer lugar en la TLB y, sólo si la información de traducción no se encuentra allí, se accede a la tabla o tablas de páginas. De este modo, si la probabilidad de acierto en la TLB es suficientemente alta, el retardo medio introducido por la traducción será mínimo (véase el ejercicio 10).

En la figura 24 se muestra un esquema simplificado del funcionamiento de la traducción utilizando la TLB y un nivel de tabla de páginas. El funcionamiento, a partir de momento en que la CPU genera una dirección virtual, es el siguiente:

- **Paso 1.** Con la información de la página (PV) se accede a la TLB, y se compara con las etiquetas.
  - Si hay acierto en la TLB, a su salida estará disponible el marco de página en el que se encuentra, dándose por finalizado el proceso de traducción. Basta con concatenar el desplazamiento (desp) dentro de la página para conseguir la dirección física correspondiente.
  - Si no se encuentra (fallo en TLB), se activa el mecanismo de traducción más lento (denominado en la figura "Traductor"), que se encarga de acceder a la tabla de páginas.
- **Paso 2.** Acceso a la tabla de páginas.
  - Si la página está en memoria principal, se obtiene el marco de página en el que reside así como el resto de su información, y se actualiza la TLB. La actualización puede llevar consigo la sustitución de una de sus entradas, habitualmente la que ha sido utilizada menos recientemente (LRU) ya que será, en principio, la que tiene menor probabilidad de ser utilizada con posterioridad.
  - Si la página no se encuentra en memoria principal, se genera la excepción de **fallo de página**, que se trata de la forma expuesta en la sección 5.2.

El conjunto formado por la TLB y el Traductor conforma la MMU. En la figura 25 se muestra la interconexión entre los distintos elementos en un sistema con memoria virtual.

Ejercicio 10.- *Supongamos un sistema con memoria virtual paginada y una TLB con tiempo de acceso de 1 ns. Su tasa de aciertos en la ejecución de un cierto programa es del 90 %. Además dispone de dos niveles de tablas de páginas en memoria principal, y cada una de las entradas de las tablas ocupa una palabra. Sabiendo que el tiempo de acceso de la memoria principal es de 60 ns ¿Cuál será el tiempo medio empleado para la traducción en un acceso a memoria? ¿Y si no tuviera TLB?*

Con la TLB, el tiempo medio de traducción por acceso será:

$$Hr_{TLB} \times T_{TLB} + (1 - Hr_{TLB}) \times (T_{TLB} + T_{TP's}) = 0,9 \times 1\ ns + 0,1 \times (1 + 2 \times 60)\ ns = 13\ ns$$

Si no tuviese TLB, el tiempo de traducción sería: $2 \times 60\ ns = 120\ ns$

La utilización de la TLB tiene, sin embargo, un inconveniente cuando se produce un cambio de contexto, ya que contiene información que sólo es válida para el proceso en ejecución y carece de significado para el resto.

Para comprender el problema supongamos dos procesos, P1 y P2, que acceden a las páginas 0, 1 y 2 de su espacio virtual. Las entradas correspondientes de sus tablas de páginas se muestran

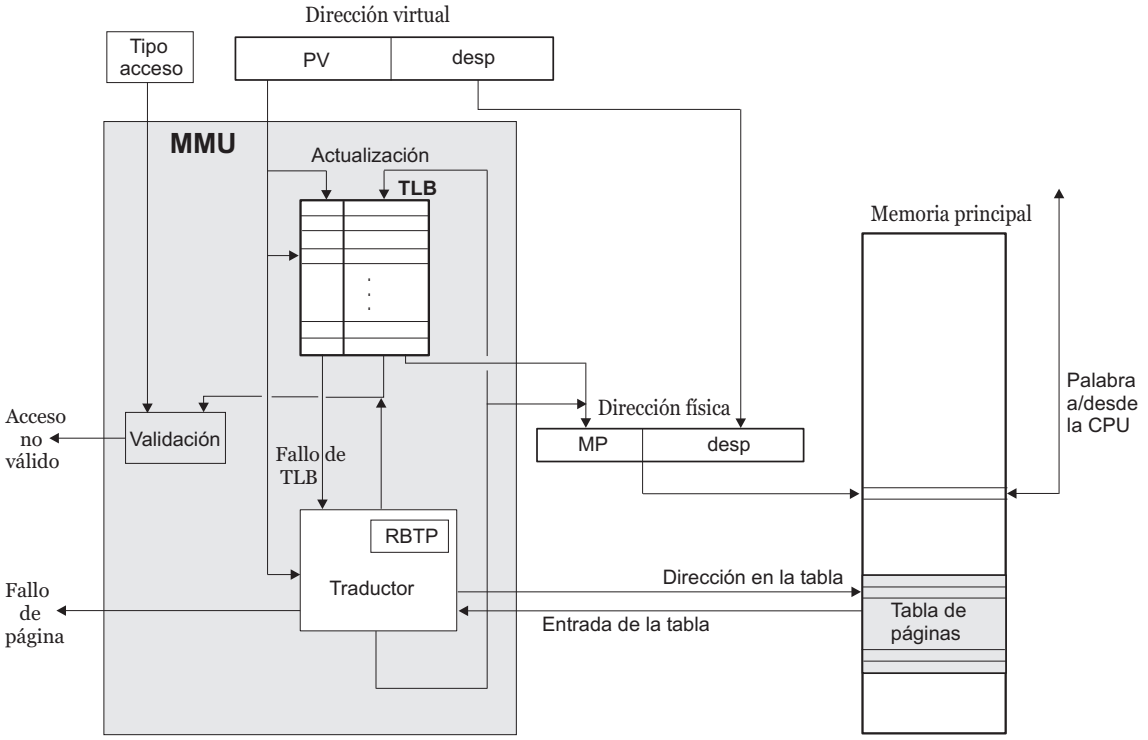


Figura 24: Traducción combinada TLB-TP en un sistema con paginación.

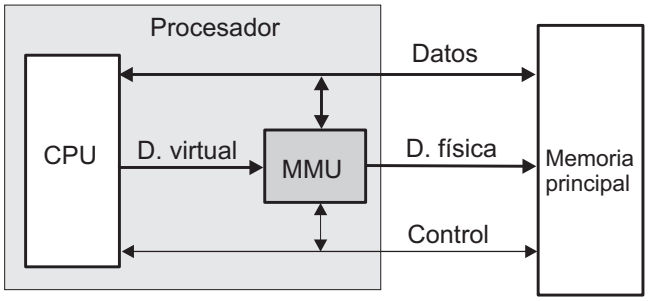


Figura 25: Conexión entre componentes en un sistema con memoria virtual.

en la figura 26, en la que se ha asumido que el proceso en ejecución es P1. La TLB (caso *a* de la figura) contiene las entradas procedentes de su tabla. Supongamos ahora que se produce un cambio de contexto, pasa a ejecutarse P2 y genera una dirección correspondiente a su página 0. Al acceder a la TLB se produciría acierto en la primera de sus entradas, obteniéndose de ella el marco de página 1, con lo que P2 no accedería a su información, que está en el marco 2, sino a la información de P1.

Para evitar este problema, la solución más sencilla es invalidar la TLB[4] cada vez que se produce un cambio de contexto. De este modo, cuando P2 pase a ejecutarse se producirá fallo al acceder a la TLB, y se irá cargando con las entradas de su tabla de páginas a medida que avance su ejecución. Si mientras se está llenando la TLB se produce un nuevo cambio de contexto, se volverá a invalidar, repitiéndose otra vez toda la operación. Si los cambios de contexto son frecuentes, el aumento de velocidad que supone el uso de la TLB podría verse contrarrestado

[4]Para invalidar la TLB basta con poner a cero los bits de validez de cada una de sus entradas.

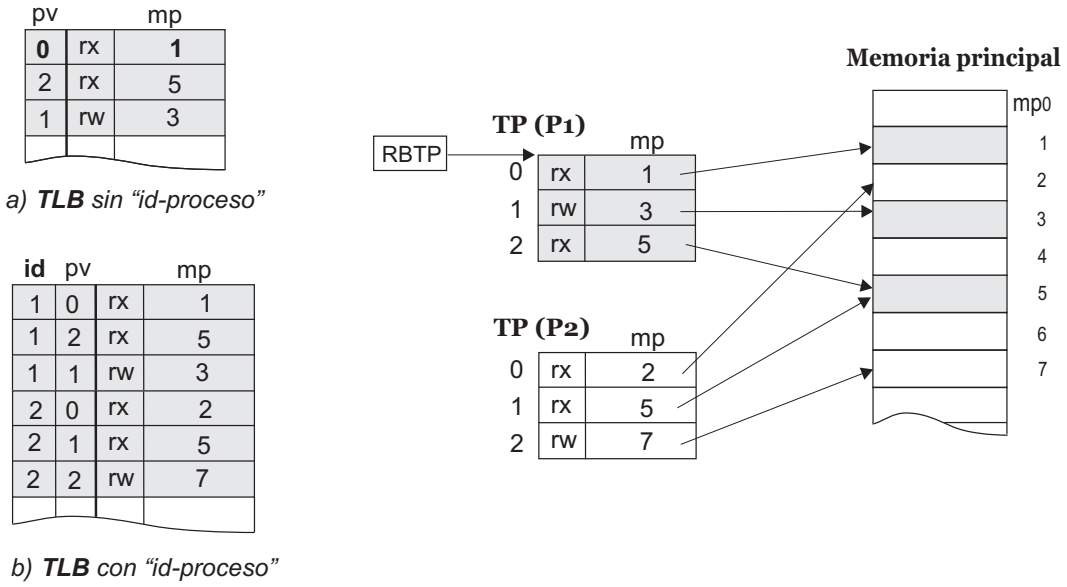


Figura 26: Cambio de contexto y uso de la TLB sin y con identificador de proceso.

por el tiempo adicional derivado de las continuas invalidaciones.

Una solución más eficiente es añadir a cada entrada de la TLB un identificador del proceso al que pertenece la dirección virtual. De este modo, la TLB puede contener información de traducción de varios procesos a la vez (caso *b* de la figura 26) sin lugar a confusión[5]. La clave de entrada a la TLB será, en este caso, la página virtual junto con el identificador correspondiente al proceso en ejecución. Así, si los cambios de contexto son frecuentes y la TLB es suficientemente grande, se reducirá su tasa de fallos, incrementándose en consecuencia la velocidad de ejecución.

En la tabla 4 se muestran algunos ejemplos de procesadores que disponen de TLBs para acelerar la traducción. En la actualidad, la mayoría dispone de dos niveles de TLB: un primer nivel rápido (un ciclo de reloj por acceso) y de pequeña capacidad, con TLBs separadas para datos e instrucciones, lo que permite realizar simultáneamente la traducción de ambos tipos de información (necesario en los procesadores con *pipeline* de instrucciones), y un segundo nivel de mayor capacidad y tiempo de acceso, que contiene la traducción de ambos tipos de información.

	TLB Instrucciones		**TLB datos**		**TLB nivel 2**	
procesador	Entradas	Líneas/cjto.	Entradas	Líneas/cjto.	Entradas	Líneas/cjto
ARM Cortex-A9	32	Asociativa	32	Asociativa	128	2
intel Core i7	128	4	64	4	512	4
Amd Opteron	40	Asociativa	40	Asociativa	512	4

Tabla 4: Ejemplos de TLBs

[5]Observe que existen dos entradas en la TLB correspondientes a distintos procesos y distintas páginas virtuales que llevan al mismo marco de página. Dichas entradas corresponden a información compartida por ambos.

5.3.3. Tamaño de página y fragmentación.

Otro aspecto importante a considerar a la hora de diseñar un sistema con paginación es la elección del tamaño de la página, puesto que de éste dependerá:

- La mejor o peor utilización del espacio de memoria principal asignado a los procesos.
- La mejor o peor utilización del dispositivo de almacenamiento secundario utilizado como soporte de la paginación.
- El tamaño de la tabla de páginas.

Ya que lo habitual es que la información utilizada por un proceso no ocupe un número entero de páginas, parte de la la última página asignada quedará sin utilizar. A este desaprovechamiento de la memoria se le denomina **fragmentación interna**. Si el tamaño de página elegido es de **p** palabras, por término medio se desaprovechará un espacio de **p/2** palabras en la última página. En consecuencia, para disminuir la cantidad de memoria desaprovechada conviene utilizar páginas pequeñas. No obstante, esto supone tener una tabla de páginas grande, al aumentar el número de páginas.

Por otro lado, la utilización de páginas pequeñas puede conducir a un uso ineficiente de los discos magnéticos, utilizados habitualmente como soporte de la paginación, ya que estos dispositivos tienen retrasos rotacionales grandes. En el ejercicio de la página 38, en el que se consideraba un disco con tiempo de posicionamiento de 5 ms y velocidad de transferencia de 100 MB/s, ya que se debe esperar como media 5 ms antes de poder comenzar una transferencia, sería más conveniente transferir un bloque de información relativamente grande, puesto que el tiempo de transferencia es más pequeño que el de posicionamiento. Se puede comprobar que existe poca diferencia en tiempo entre transferir 4 KB ó 8 KB:

$$5\ ms + (4.096\ bytes/100.000.000\ bytes/s) \approx 5,04\ ms$$

$$5\ ms + (8.192\ bytes/100.000.000\ bytes/s) \approx 5,08\ ms$$

La elección del tamaño de página más adecuado debe realizarse, por lo tanto, teniendo en cuenta de forma conjunta estos tres aspectos. En la actualidad se suelen emplear páginas de 4 u 8 KB.

5.4. Segmentación

Consiste en dividir el espacio de direcciones virtuales en varios espacios independientes, denominados **segmentos** (figura 27). Cada segmento consta de un conjunto de direcciones consecutivas, y su tamaño puede variar entre 0 y un máximo permitido. Segmentos diferentes pueden tener longitud diferente, que además puede variar durante la ejecución, como ocurre por ejemplo en un segmento de pila.

La segmentación proporciona una visión del espacio virtual más cercana a la del programador. Normalmente, el compilador construye los segmentos que reflejan la estructura del programa. Puede crear, por ejemplo, segmentos separados para las variables globales, la pila y el código de cada procedimiento o función. El montador tomará estos segmentos y les asignará un número que los identifique de forma única. Ya que cada segmento contiene información

de la misma naturaleza, la segmentación simplifica la protección y compartición de información con respecto a la paginación.

Las direcciones virtuales generadas por los programas constan del número de segmento en que se encuentra la información y la posición dentro de él. Aunque se utiliza una dirección bidimensional, la memoria principal sigue siendo un espacio de direcciones unidimensional. Por lo tanto, se debe definir la correspondencia entre las direcciones virtuales bidimensionales y las direcciones de la memoria principal.

La traducción, al igual que en la paginación, se puede realizar utilizando una tabla, denominada **tabla de segmentos**, en la que cada entrada contendrá información acerca del segmento correspondiente (figura 27), similar a la expuesta en la página 40, además de su longitud. La información de traducción propiamente dicha, en este caso, es la dirección a partir de la cual está cargado el segmento en memoria principal. Cada proceso tiene asociada una tabla de segmentos y el procesador maneja un registro de base (**RBTS**) que apunta siempre a la tabla del proceso en ejecución.

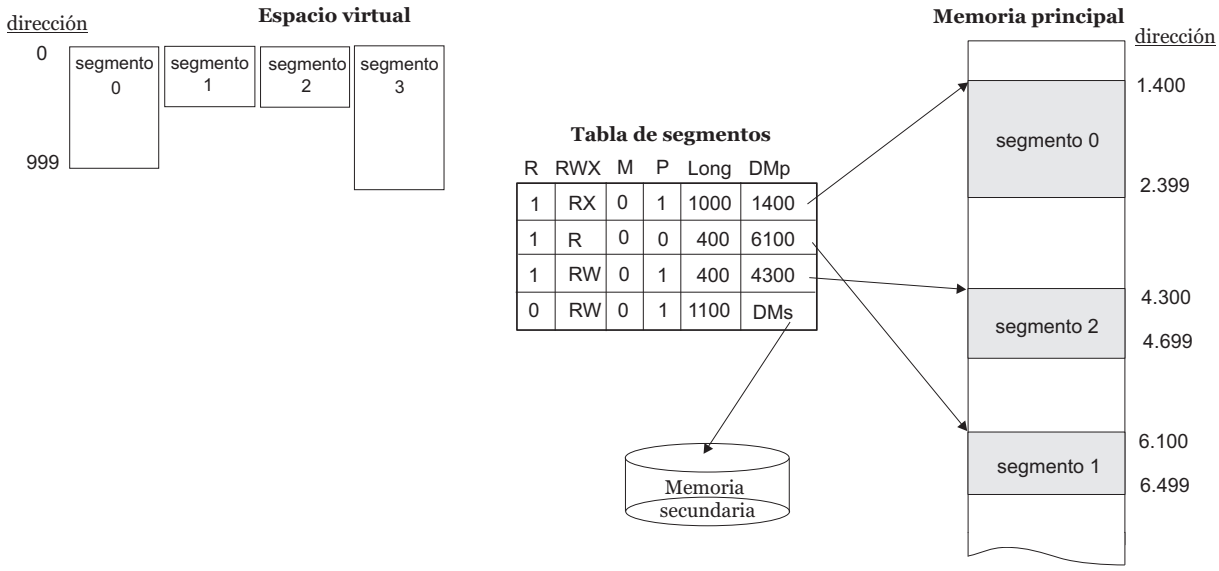


Figura 27: Correspondencia de direcciones en un sistema con segmentación.

El mecanismo de traducción básico es el que se muestra en la figura 28. Las diferencias con el mecanismo de paginación (véase la figura 21) son las siguientes:

- Hay que comprobar si se intenta acceder a una posición fuera del segmento (utilizando la información de longitud).
- Para obtener la dirección física se ha de sumar la dirección de base del segmento, obtenida de la tabla, al desplazamiento procedente de la dirección virtual.

Es importante hacer notar que, a diferencia de lo que ocurre en la paginación, la tabla de segmentos puede residir en un conjunto de registros, ya que el número de segmentos que puede utilizar un proceso suele ser pequeño. Esto supondría una mayor velocidad de traducción al evitarse accesos a memoria.

A pesar de las ventajas de la segmentación citadas anteriormente, no es habitual encontrar sistemas con segmentación pura. Ello es debido fundamentalmente a los problemas de gestión

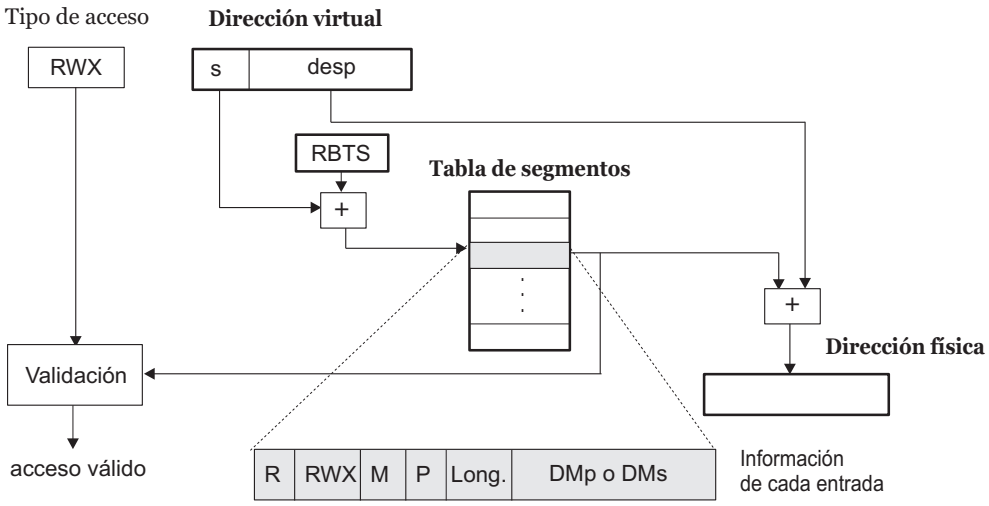


Figura 28: Mecanismo básico de traducción en un sistema con segmentación.

de memoria que hay que afrontar como consecuencia de la longitud variable de los segmentos (véase la sección 5.6.4). Por otra parte, en la segmentación se puede producir también el fenómeno de la fragmentación mencionado en la sección 5.3.3, que se denomina en este caso **fragmentación externa**. Se produce cuando en la memoria principal hay espacio suficiente para ubicar un segmento, pero este espacio no es contiguo. Una posible solución es compactar la memoria cada cierto tiempo, de forma que se agrupe todo el espacio libre en una zona contigua.

Por último, hay que hacer notar que si los segmentos son grandes, lo más eficiente es tener en memoria únicamente la parte que se esté utilizando en cada momento, por lo que, en general, los sistemas actuales combinan la segmentación y la paginación.

5.5. Segmentación paginada.

Existe también la posibilidad de combinar segmentación y paginación para aprovechar las ventajas de ambos. En la segmentación paginada, cada segmento se trata como una memoria virtual, que se divide a su vez en páginas. Se aprovechan así las ventajas de la segmentación: facilidad de protección y compartición de información, que se realiza a nivel de segmento, y las de la paginación: tamaño de página uniforme y ubicación en memoria principal de sólo aquellas partes de los segmentos que se estén utilizando.

Las direcciones virtuales generadas por los programas constan, en este caso, de tres campos: segmento (**s**), página del segmento (**p**) y posición relativa de la información dentro de la página (**desp**). Para realizar la traducción, se necesita una tabla de segmentos y una tabla de páginas para cada uno de ellos. La traducción de direcciones (figura 29) requiere dos accesos a memoria, suponiendo ambas tablas en memoria principal, aunque se suele utilizar una TLB para acelerar la traducción. La clave de acceso a dicha TLB será la información de segmento (s) junto con la de la página (p) dentro del segmento. [6]

Uno de los primeros sistemas que utilizaron este mecanismo fue el sistema operativo MULTICS [Org72], que proporcionaba una memoria virtual de hasta 2^{18} segmentos de hasta 2^{16}

[6] Obsérvese la similitud con la utilización de tablas de páginas multinivel, vista en la sección 5.3.1.

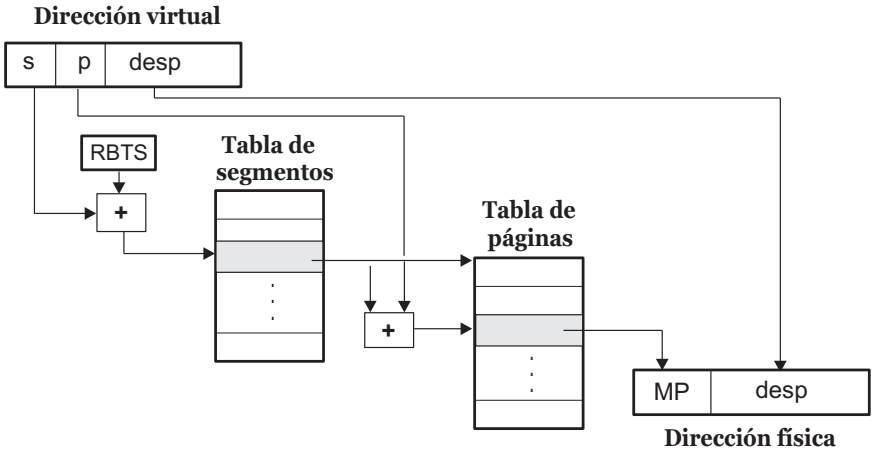


Figura 29: Mecanismo básico de traducción en un sistema con segmentación paginada.

palabras de 36 bits. Ya que podía haber potencialmente 2^{18} entradas en la tabla de segmentos, esta tabla era a su vez un segmento y estaba paginada. El tamaño normal de una página era de 1.024 palabras, aunque existía también la posibilidad de tener segmentos no paginados.

Entre las arquitecturas con soporte para segmentación paginada cabe citar PowerPC, PA-RISC e intel Pentium (IA-32). El hardware de gestión de memoria utilizado en esta última es esencialmente el mismo que en los procesadores 80386 y 80486 de Intel, que se describe a continuación.

Ejemplo.- Soporte para memoria virtual en los microprocesadores i80386/486.

Estos microprocesadores incluyen el *hardware* necesario para dar soporte tanto a paginación como a segmentación. Ambos mecanismos pueden configurarse, permitiendo implementar diferentes esquemas de memoria virtual: segmentación, paginación y segmentación paginada.

Segmentación. En este modo se pueden direccionar hasta 2^{14} segmentos de hasta 4 GB cada uno. Para realizar la traducción se dispone de 2 tablas de segmentos, denominadas aquí *tablas de descriptores*:

- la tabla de descriptores locales (**LDT**), propia de cada proceso.
- la tabla de descriptores globales (**GDT**), compartida por todos los procesos del sistema.

La dirección virtual consta de 46 bits: 14 proceden de los bits superiores de uno de los 6 registros de *selector de segmento*, cuyo formato se muestra en la figura 30, y los 32 restantes de la propia instrucción. El registro a utilizar en cada caso se deduce por el contexto; por ejemplo, si el acceso es de *fetch* se utilizará el registro CS (segmento de código), y si es a un dato, el registro DS (segmento de datos).

Uno de los 14 bits procedentes del registro selector de segmento indica si el segmento es local o global, por lo que sirve para seleccionar la tabla de segmentos a utilizar (LDT o GDT). Los otros 13 bits especifican la entrada dentro de la tabla correspondiente donde se encuentra la información de traducción. Cada tabla permite por lo tanto manejar hasta 2^{13} segmentos. La figura 30 muestra el mecanismo de traducción, así como el formato de las entradas de la tabla, en las que se recoge la siguiente información acerca de los segmentos:

- **D. base** (32 bits): dirección de comienzo del segmento en memoria principal.
- **Long** (20 bits): tamaño del segmento en bytes (si G=0) o en páginas (si G=1)
- **P** (1 bit): indica si el segmento está o no en memoria.
- **DPL** (2 bits): indica el nivel de privilegio (entre 0 y 3) asociado al segmento.

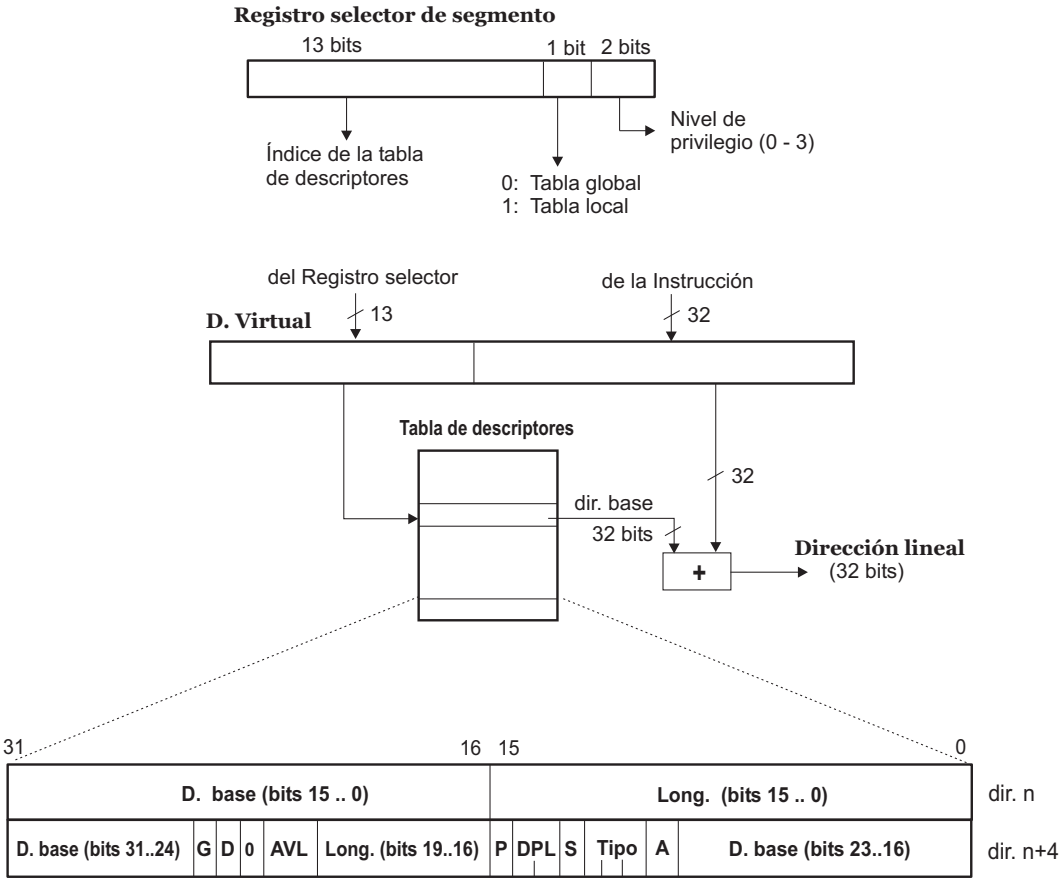


Figura 30: Formato de los descriptores de segmento y traducción.

- **S** (1 bit): indica si el segmento corresponde a información del sistema (S=0) o no (S=1).
- **A** (1 bit): se pone a "1" cuando el procesador accede al segmento. Lo utiliza el sistema operativo para la política de reemplazo.
- **D** (1 bit): longitud de los operandos y direcciones efectivas. Si D="1"se suponen de 32 bits. Si D="0"se suponen de 16 bits. Este último modo se utiliza para mantener la compatibilidad con el software para el i80286.
- **AVL**: Campo disponible para el usuario y el sistema operativo.
- **TIPO** (3 bits): Información acerca del tipo y protección de los segmentos como:
  - Si el segmento es de código o datos.
  - Si se puede leer, escribir o ejecutar.
  - Si al acceder al segmento cambia o no su nivel de privilegio.
  - El sentido de expansión del segmento (para segmentos de pila y datos).

La dirección de 32 bits que se obtiene finalmente se denomina **dirección lineal**. Si la paginación está inhibida se interpreta como la dirección física con la que se accede a memoria.

Para acelerar la traducción no se accede realmente a estas tablas en cada acceso a memoria, sino que, en el momento en que se almacena información en un registro selector de segmento, se accede a la tabla correspondiente para obtener toda la información de traducción, protección etc. Esta información se almacena en una serie de registros denominados *descriptores de segmento*. En los siguientes accesos al mismo segmento ya no se accederá a la tabla sino al registro de descriptor de segmento correspondiente.

Si la paginación está habilitada, la dirección de 32 bits se interpreta como una dirección de un espacio virtual paginado, que hay que traducir mediante el uso de tablas de páginas. Estamos entonces en el caso de segmentación paginada.

Segmentación paginada. En estos sistemas el tamaño de las páginas es de 4 KB, por lo que se pueden direccionar hasta 2^{20} páginas virtuales. El mecanismo para traducir esta dirección (figura 31) utiliza dos niveles de tablas. El primer nivel, denominado **Directorio**, consta de 2^{10} entradas de 32 bits, y su dirección de comienzo está contenida en un registro de control del microprocesador. Cada una de sus entradas contiene la dirección de comienzo de la tabla de páginas correspondiente al segundo nivel, con la información de traducción de una zona de hasta 4 MB de memoria virtual.

Para evitar el retardo que supone el tener que realizar dos accesos a memoria para completar la traducción, estos microprocesadores disponen de una pequeña memoria asociativa (véase la sección 5.3.2) con la información de traducción de las últimas páginas a las que se ha accedido. En esta memoria, la clave de acceso (etiqueta) está formada por los campos *directorio* y *página* de la dirección lineal.

Paginación. Así como el mecanismo de paginación se puede activar y desactivar, el mecanismo de segmentación siempre está presente. Si se quiere implementar en estos microprocesadores un sistema que utilice únicamente paginación habrá que recurrir a un procedimiento para simular su desactivación. Bastará con cargar los registros selectores de segmento con índices que apunten a entradas de las tablas LDT o GDT, que contendrán el valor cero en el campo *Dirección de base* y el valor máximo en el campo *Longitud*. De este modo el procesador sólo maneja un segmento que abarca todo el espacio de memoria de 4 GB.

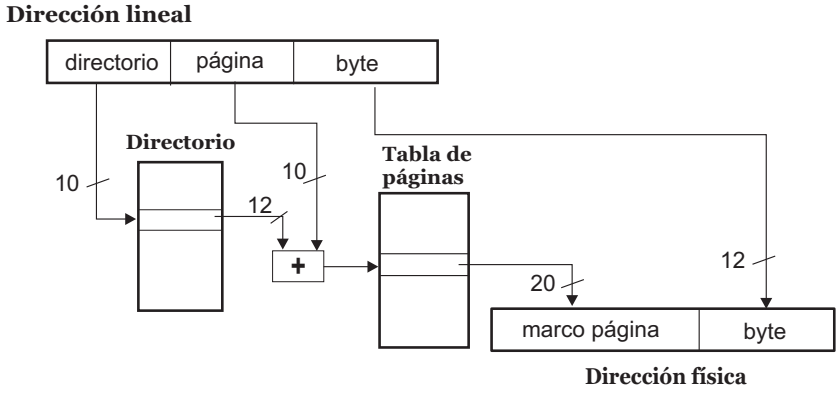


Figura 31: Traducción en segmentación paginada

5.6. Asignación y gestión de la memoria principal.

Una vez vistos los elementos *hardware* utilizados para implementar la memoria virtual, se tratan aquí algunos aspectos relativos al *software* necesario, es decir, a las labores que debe realizar el sistema operativo, en particular, la **asignación** del espacio de memoria principal a los distintos procesos y la **gestión** eficiente de dicho espacio. Aunque se trata únicamente el caso de la paginación, por ser la más utilizada, los conceptos que aquí veremos son fácilmente aplicables a la segmentación y la segmentación paginada.

Definiremos previamente los conceptos de **conjunto residente** y **conjunto de trabajo**, que utilizaremos a lo largo de esta sección.

- El **conjunto residente** de un proceso es el conjunto de páginas que dicho proceso tiene en memoria principal en un determinado instante.

- El **conjunto de trabajo** (*working set*), **W(t,k)** de un proceso en un instante **t** es el conjunto de páginas a las que ha hecho referencia en los **k** últimos accesos (k puede expresarse también en unidades de tiempo). Estas páginas serán las que con mayor probabilidad volverán a ser utilizadas en un futuro cercano, según la propiedad de proximidad de referencias.

  Consideremos por ejemplo la traza de un proceso (expresada en páginas), que se muestra en la figura 32. Suponiendo **k**=10, el conjunto de trabajo en el instante **t1** estaría compuesto por las páginas 1, 2, 5, 6 y 7, mientras que en **t2** sólo estarían las páginas 3 y 4. **k** puede verse como una ventana que se desplaza a lo largo de la traza, de modo que todas las páginas que caen dentro de ella forman el conjunto de trabajo del proceso.

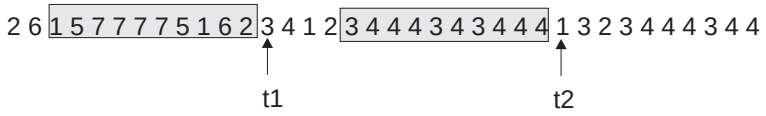


Figura 32: Ejemplo de conjunto de trabajo de un proceso.

Las funciones de asignación y gestión de memoria las realiza un componente del sistema operativo, el Gestor de memoria, que debe llevar a cabo las siguientes tareas:

- Crear y gestionar las tablas de traducción.
- Repartir la memoria principal disponible entre los procesos (Asignación). Para ello hay que establecer una estrategia para determinar cuántos marcos de página se asignan a cada proceso (**tamaño de su conjunto residente**) teniendo en cuenta las necesidades de cada proceso, y el número de procesos activos[6] (**grado de multiprogramación**).
- Gestionar el espacio de memoria asignado a cada proceso, para lo que debe determinar:
  - Qué paginas deben residir en cada momento en memoria principal (**composición de su conjunto residente**).
  - Cuándo se debe llevar la información a memoria principal (**política de extracción**).
  - El lugar donde debe colocarse la información (**política de ubicación**).
  - En caso de no existir espacio libre, qué información debe abandonar la memoria principal (**política de reemplazo**).

Trataremos en primer lugar las políticas de asignación de memoria, así como los conceptos que sirven para determinar la composición del conjunto residente en cada instante. A continuación expondremos las distintas políticas de extracción, ubicación y reemplazo. Finalmente se verán algunos aspectos relacionados con el parámetro que determina en gran parte la eficiencia del gestor de memoria: la **tasa de fallos de página**.

[6] Un proceso activo es el que tiene alguna información en memoria principal.

5.6.1. Políticas de asignación de memoria principal.

Las dos formas de distribuir el espacio de memoria principal, utilizadas en sistemas con multiprogramación son las siguientes:

- Mediante particiones, o conjuntos residentes, de **tamaño fijo**. El número de marcos de página asignados a cada proceso es fijo y se mantiene constante a lo largo de su ejecución. Entre los distintos criterios de reparto utilizados a la hora de determinar este número podemos citar los siguientes:
  - Equitativo.
  - Proporcional al tamaño de cada proceso.
  - En función de la prioridad o privilegios de cada proceso.
  - En función de la cuota de memoria asignada al usuario al que pertenece el proceso.

  Sea cual sea el criterio utilizado, existe un número mínimo de marcos de página que se debe asignar a un proceso para que sea posible su ejecución, que viene fijado por el juego de instrucciones del computador, y más concretamente por la instrucción cuya ejecución implique el mayor número de accesos a memoria y que, por lo tanto, pueda generar el mayor número de fallos de página.
- Mediante conjuntos residentes de **tamaño variable**. El número de marcos de página asignados a cada proceso varía dinámicamente en función de sus necesidades.

La ventaja de la primera alternativa es su sencillez de implementación, aunque tiene el inconveniente de no poder soportar de forma eficiente procesos cuyas necesidades de memoria presentan mucha variación a lo largo de su ejecución. Desde este punto de vista la segunda alternativa, aunque más compleja, permite un mejor aprovechamiento de la memoria principal.

5.6.2. Composición del conjunto residente.

La decisión de qué páginas deben residir en cada momento en memoria se basa en el comportamiento o **traza** de los procesos y en el concepto de **conjunto de trabajo**. Si se estudia su evolución a lo largo de la ejecución de un proceso, se puede observar que su variación con el tiempo es muy lenta, debido a la propiedad de proximidad de referencias. Esta propiedad hace que las páginas que componen el conjunto de trabajo de un proceso tiendan a situarse rápidamente en memoria principal (si hay espacio suficiente), reduciéndose en consecuencia el tiempo de acceso en las siguientes referencias a las páginas que lo componen.

En resumen, el conjunto de trabajo se puede utilizar para determinar el tamaño adecuado del conjunto residente de cada proceso, con el fin de que éste se ejecute de forma eficiente, por lo que **k** es un parámetro crítico a determinar, del que dependerá la tasa de fallos de página del proceso.

5.6.3. Políticas de extracción.

La mayor parte de los sistemas con memoria virtual utilizan la política de extracción **bajo demanda**: las páginas se envían a memoria principal únicamente cuando se produce un fallo al tratar de acceder a ellas.

A diferencia de lo que ocurre en las memorias caché, las políticas de extracción **con anticipación** no se suelen emplear aquí debido fundamentalmente al elevado tamaño de las páginas (en comparación con el tamaño de los bloques de caché) y, por lo tanto, a la menor probabilidad de que una página traída con anticipación sea realmente de utilidad.

5.6.4. Políticas de ubicación.

Para implementar las políticas de ubicación, el gestor de memoria del sistema operativo debe disponer de información acerca del espacio de memoria libre existente en cada momento. En la paginación, basta con gestionar una lista con los marcos de página disponibles. El caso de la segmentación es algo más complejo, ya que los segmentos pueden tener diferente longitud, que además puede variar, por lo que es necesario gestionar al menos una lista de huecos libres, que debe incluir su tamaño y dirección de comienzo. Estas listas son las que maneja el gestor de memoria a la hora de aplicar la **política de ubicación** a seguir cuando se produce un fallo.

En la **paginación**, la política de ubicación es relativamente sencilla: "Si existe algún marco de página libre, la página que falló se ubica en él y, si no, se reemplaza alguna de las páginas ocupadas, según la **política de reemplazo** definida".

En la **segmentación** debe buscarse un hueco de tamaño suficiente como para albergar el segmento que produjo el fallo. Existen distintos algoritmos o políticas de ubicación, en función de cómo se gestione la lista de huecos. A continuación se exponen brevemente algunos de ellos.

- **Primer encaje** (*first fit*). El gestor de memoria recorre la lista de huecos, generalmente ordenada por direcciones, hasta que encuentra uno lo suficientemente grande. El espacio sobrante, si existe, constituirá un nuevo hueco en la lista de espacio libre.
- **Mejor encaje** (*best fit*). En este caso, se consulta la lista de huecos libres hasta encontrar el que mejor se ajuste al tamaño del segmento, por lo que es conveniente que la lista esté ordenada por tamaños para disminuir el tiempo de búsqueda. Este algoritmo se comporta algo peor que anterior, ya que tiende a generar huecos libres demasiado pequeños para ser utilizados posteriormente.
- **Peor encaje** (*worst fit*). Su finalidad es intentar resolver el problema del algoritmo anterior, y consiste en utilizar el hueco libre más grande, con el objetivo de que el hueco que se genere sea a su vez grande.

Estos algoritmos tienden, antes o después, a generar huecos de tamaño pequeño y, a menos que exista algún mecanismo que atenúe este efecto creando huecos grandes a partir de los pequeños, pueden acabar llenando la memoria principal de huecos demasiado pequeños para albergar un nuevo segmento. Esta **fragmentación externa** constituye, al igual que la **fragmentación interna** que se produce en la paginación, un desaprovechamiento de la memoria.

Una solución para paliar esta situación, aparte de la compactación, es introducir un mecanismo que haga que siempre que se elimine de memoria un segmento con uno o dos huecos adyacentes, se reúna todo este espacio libre para formar un hueco único de tamaño igual a la suma de todos ellos. Esta solución es, en principio, más sencilla y consume menos tiempo, ya que basta con actualizar la lista de huecos libres cada vez que se desaloja un segmento de la memoria principal. Dependiendo de cómo esté organizada dicha lista, la actualización será más o menos rápida. Un algoritmo que sigue esta idea es el algoritmo Buddy [Tan09].

5.6.5. Políticas de reemplazo.

En la literatura sobre el tema se pueden encontrar multitud de algoritmos de reemplazo, tanto genéricos como específicos de determinados sistemas operativos. En este apartado nos limitaremos a dar una breve descripción de los más utilizados.

En general, todos los algoritmos tratan de evitar reemplazar páginas que hayan sido modificadas, ya que esto obliga a realizar dos accesos a memoria secundaria: uno para actualizar la copia en disco y otro para traer la nueva página a memoria.

Para implementar la política de reemplazo, el sistema operativo suele utilizar:

- Información contenida en las tablas de traducción. La mayoría de las unidades de gestión de memoria actuales (MMU) mantienen algún tipo de información sobre el uso de las páginas, como por ejemplo los bits M y U, mencionados en la sección 5.3:
  - **M**. La MMU activa este bit cuando se realiza un acceso de escritura sobre información de la página.
  - **U**. La MMU lo activa cuando se accede a información de la página.
- Estructuras de datos que crea y gestiona el propio sistema operativo.

Estableceremos a continuación una posible clasificación de estas políticas, basada en los siguientes criterios:

- El tiempo que han permanecido las páginas en memoria principal.
- El instante en que se hizo referencia por última vez a las páginas.
- La frecuencia de utilización de las páginas.

Además, podemos citar la política de reemplazo **aleatoria** utilizada en algunos sistemas, en la que la página a reemplazar se elige al azar.

Políticas dependientes del tiempo de permanencia en memoria.

A este grupo pertenecen las políticas **FIFO** y **LIFO**. La política **FIFO** reemplaza la página que ha permanecido en memoria más tiempo, mientras que la **LIFO** selecciona la que lleva menos tiempo en memoria.

La política FIFO es una de las más sencillas de implementar, y no necesita ningún *hardware* adicional. Basta con gestionar una cola FIFO donde se van introduciendo los números de las páginas a medida que se llevan a memoria principal. A la hora de elegir una para ser reemplazada se seleccionará la primera de la cola. Sin embargo, esta solución puede resultar ineficiente si la página elegida corresponde a información que se está utilizando con mucha frecuencia.

Una modificación sencilla, con la que se evita el problema anterior, consiste en examinar el bit de utilización (U) de la página elegida para reemplazar. Si este bit está desactivado se reemplaza, y si no lo está, se da a la página una "segunda oportunidad", poniéndola al final de la cola y desactivando su bit de utilización (es como si acabara de llegar a memoria). Este algoritmo se conoce como de la **segunda oportunidad** [Tan09].

Políticas dependientes del último instante de referencia.

En este grupo se incluyen la política **LRU** (*least recently used*) y una de sus variaciones más conocidas, la política **NRU** (*not recently used*).

La política **LRU** consiste en reemplazar la página que no ha sido utilizada durante un mayor periodo de tiempo. Por lo tanto, tiene en cuenta el principio de proximidad temporal, ya que las páginas que lleven más tiempo sin ser utilizadas serán las que, en principio, tienen menor probabilidad de volver a serlo. Su implementación es más compleja que la de la política FIFO, puesto que requiere el conocimiento de referencias pasadas. En principio, habría que asociar a cada página información acerca del instante en que se accedió a ella por última vez.

Una posible implementación consiste en utilizar un contador global, que se incremente con cada referencia a memoria, y un campo en cada una de las entradas de la tabla de páginas donde se copie el valor del contador cada vez que se haga referencia a la página correspondiente. De este modo, se reemplazará la página cuyo contador asociado tenga el valor menor. Este mecanismo, aunque factible, requiere un considerable soporte *hardware* y una búsqueda exhaustiva en la tabla de páginas para encontrar la página que lleva más tiempo sin ser utilizada, lo que puede suponer un coste significativo.

Otra forma de implementar la política LRU consiste en utilizar una lista con los números de las páginas residentes en memoria, ordenada por el instante de acceso, de modo que cuando se haga referencia a una página se moverá su identificador al principio de la lista. De esta forma, se reemplazará la página cuyo número se encuentre al final de la lista. Hay que hacer notar que la actualización de dicha lista debería realizarse en cada acceso a memoria, por lo que sería muy costosa aunque se implementase por *hardware*

La política **NRU** consiste en reemplazar la página que no ha sido utilizada recientemente. Este algoritmo requiere, por lo tanto, cuantificar qué se considera "pasado reciente".

Políticas dependientes de la frecuencia de utilización.

En este grupo se incluyen las políticas **LFU** (*least frequently used*) y **MFU** (*most frequently used*). La política LFU consiste en reemplazar la página utilizada menos frecuentemente, mientras que la MFU reemplaza la que se ha utilizado más frecuentemente.

La primera se basa en la suposición de que una página utilizada con mucha frecuencia será la que con mayor probabilidad volverá a serlo en un futuro cercano. Una posible implementación simplificada de **LFU** consiste en asociar a cada página un contador que lleve una cuenta aproximada del número de veces que se ha accedido a ella. Esta cuenta podría llevarse a cabo examinando el bit de utilización (U) y borrándolo cada cierto tiempo. Si cuando se examina, dicho bit se encuentra a 1 se incrementará el contador correspondiente, de modo que la página cuyo contador sea menor será la que se reemplace. El problema de esta política aparece cuando se tiene una página a la que se ha accedido mucho hasta el momento, pero que no se va a volver a hacer referencia a ella. En este caso, su contador tendrá un valor grande y por lo tanto tenderá a permanecer mucho tiempo en memoria. En [Tan09] se puede ver cómo una pequeña modificación de este algoritmo elimina este problema: el algoritmo de **envejecimiento**.

La política **MFU** se basa en la suposición contraria, es decir, que la página utilizada menos frecuentemente, probablemente porque lleva poco tiempo en memoria, será la que con mayor probabilidad se utilice en el futuro.

Política de reemplazo óptima.

Cabe plantearse la posibilidad de utilizar una política de reemplazo óptima. Esta política, propuesta por Bélády, consistiría en reemplazar aquella página que no va a ser utilizada durante mayor periodo de tiempo. Para poder implementarla sería necesario conocer previamente las referencias del proceso a partir de cualquier instante, en el que pudiese producirse un fallo de página. Dado que este conocimiento solo es posible una vez analizada la traza de referencias que ha generado el proceso, esta política se utiliza únicamente como referencia para evaluar la eficiencia de los algoritmos reales.

5.6.6. Tasa de fallos e hiperpaginación.

La **tasa de fallos** es uno de los parámetros más importantes a la hora de evaluar la eficiencia de la política de gestión de memoria utilizada. Se denomina tasa de fallos de un proceso al número de fallos de página que genera por unidad de tiempo. Este parámetro dependerá fundamentalmente del tamaño de su conjunto residente. El inverso de la tasa de fallos, denominado el **tiempo de vida**, se define como el tiempo medio transcurrido entre fallos sucesivos generados por dicho proceso. El objetivo de toda política de gestión de memoria es minimizar la tasa de fallos, maximizando así el tiempo de vida de los procesos (si se tienen todas las páginas del proceso en memoria, su tiempo de vida será el que dure su ejecución).

Por otra parte, el hecho de intentar conseguir una tasa de fallos mínima para un determinado proceso puede suponer que su conjunto residente sea demasiado grande, ya que la memoria debe ser compartida entre varios procesos. Por ello, se deberá disminuir probablemente este número, óptimo desde el punto de vista de la tasa de fallos, para favorecer al resto de procesos. La elección de la tasa de fallos más adecuada se debe realizar teniendo en cuenta el número de procesos activos en el sistema y sus respectivas necesidades de memoria, de modo que los cambios de contexto no produzcan un aumento excesivo de la tasa de fallos de página.

En los sistemas con **multiprogramación**, otro recurso compartido, además de la memoria, es el propio procesador. Dado que, en general, el número de procesos en un sistema es mayor que el número de procesadores de que dispone, en un instante determinado uno o varios procesos estarán ejecutándose (dependiendo de si el sistema es monoprocesador o multiprocesador), mientras que otros estarán esperando a ser ejecutados, y otros esperando a que se realice una operación de entrada/salida. En esta situación, es necesario:

- Que la memoria principal contenga algunas páginas de varios procesos (su conjunto de trabajo) para que el cambio de contexto no produzca muchos fallos, ya que esto supondría un incremento de la tasa de fallos global del sistema.
- Que exista un grado de multiprogramación lo suficientemente alto como para que se aprovechen otros recursos como el procesador y los dispositivos de entrada/salida.

Estas necesidades son de algún modo conflictivas, ya que puede ocurrir lo siguiente:

- Si disminuye el grado de multiprogramación, aumentará el tamaño de los conjuntos residentes, lo cual puede producir una disminución de la tasa de fallos pero también puede conllevar una infrautilización del procesador.
- Si aumenta el grado de multiprogramación, disminuirá el tamaño de los conjuntos residentes, lo que puede suponer un incremento en la tasa de fallos global del sistema. Un fallo de página producido por un tamaño reducido del conjunto residente de un proceso, provocará un cambio de contexto a otro proceso con un conjunto residente también reducido, por lo que existirá mucha probabilidad de que se produzca un nuevo fallo. Esto provocaría un nuevo cambio de contexto a otro proceso, repitiéndose de nuevo el ciclo. Si esto ocurre continuamente, aparece el fenómeno denominado **hiperpaginación** (*thrashing*), en cuyo caso el rendimiento del sistema se degrada, debido a que emplea casi todo su tiempo en atender fallos de página y realizar cambios de contexto.

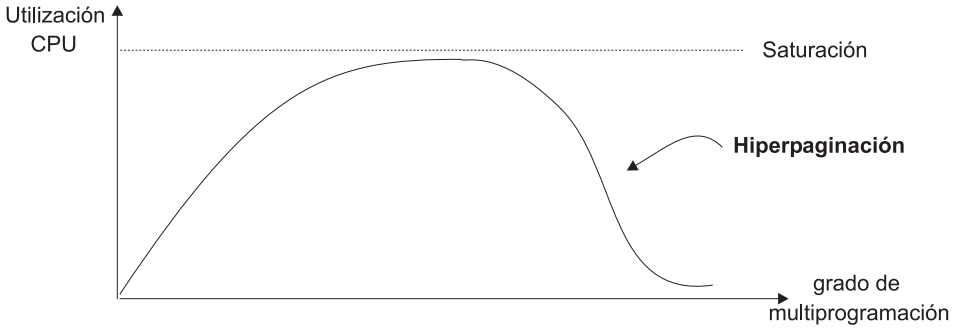


Figura 33: Utilización de la CPU en función del número de procesos activos.

La figura 33 muestra gráficamente la utilización de la CPU en función del grado de multiprogramación. A medida que aumenta el número de procesos activos, aumenta también la utilización de la CPU hasta llegar a un máximo, a partir del cual incrementar el grado de multiprogramación produce un descenso rápido en la utilización de la CPU. Es entonces cuando aparece el fenómeno de hiperpaginación.

Una manera de evitar la hiperpaginación es adaptar dinámicamente el grado de multiprogramación a las necesidades de memoria de los procesos. El sistema operativo realiza esta labor monitorizando la utilización de la CPU. Si ésta es muy baja, aumenta el grado de multiprogramación introduciendo un nuevo proceso en memoria, mientras que si se llega al punto de hiperpaginación deberá desalojar algún proceso de memoria, disminuyendo el grado de multiprogramación. No obstante, si llega un momento en el que el sistema está continuamente en hiperpaginación puede deberse a que la memoria de que se dispone es insuficiente para las aplicaciones que está soportando.

El ajuste del grado de multiprogramación de un computador lo realiza uno de los componentes del sistema operativo, denominado **Controlador de carga**. Este componente, en combinación con el **Planificador** (*Scheduler*) y el **Gestor de memoria**, determina el conjunto de procesos con información en memoria en un momento dado, así como el conjunto residente de cada uno.

6. Combinación de memoria caché y memoria virtual.

Debido a que la mayor parte de los sistemas actuales disponen tanto de memoria virtual como de memorias caché, es importante tener una visión conjunta de su funcionamiento, lo que permitirá tener constancia de los componentes del procesador involucrados en un acceso a memoria. En la figura 34 se muestran los componentes básicos de un sistema típico, en el que se ha supuesto que la caché tiene política de ubicación directa y se utiliza paginación.

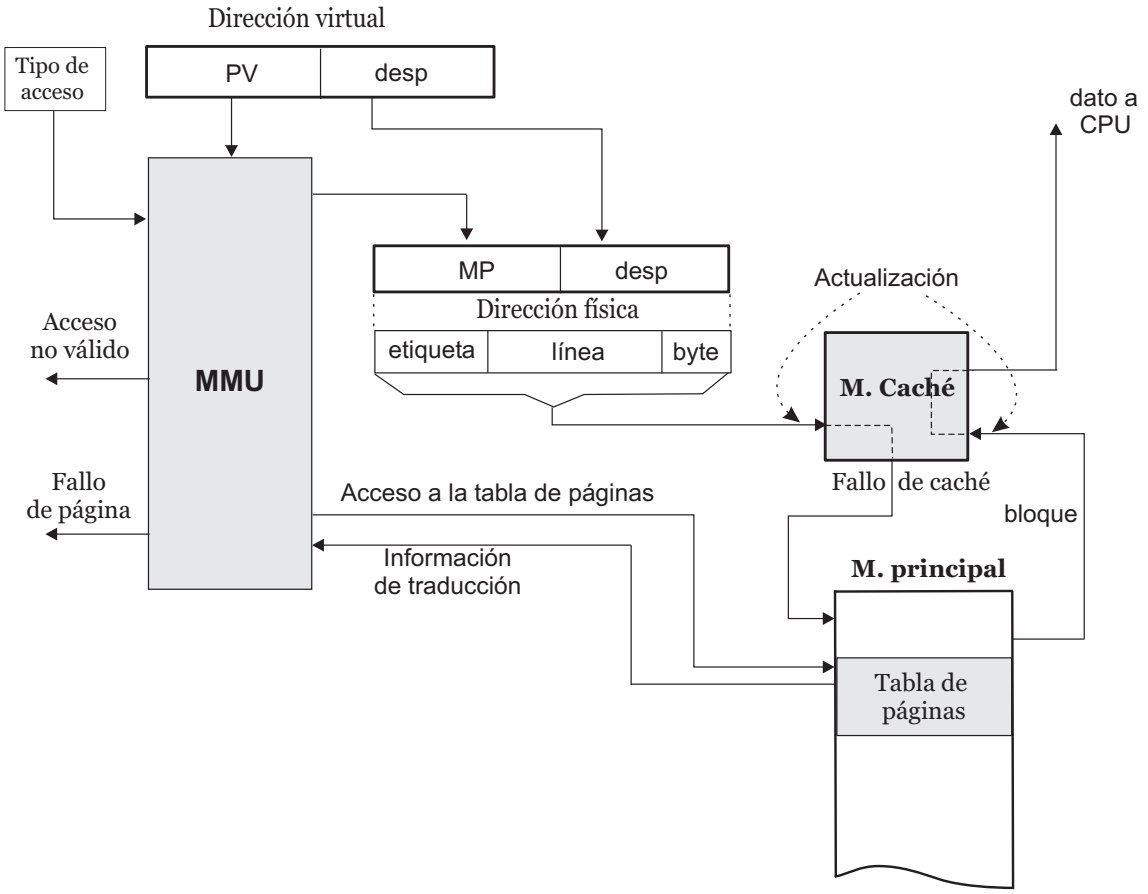


Figura 34: Esquema simplificado de un sistema con memoria virtual y caché.

En el diagrama de flujo de la figura 35 se muestran las acciones que se llevan a cabo desde que el procesador realiza una petición de acceso a memoria proporcionando la **dirección virtual** donde se encuentra la información hasta que se completa dicho acceso.

Ya que para realizar el acceso es necesaria la **dirección física** donde está la información, lo primero que se hace es traducir la dirección virtual a través de la MMU, proceso que llevará mas o menos tiempo dependiendo de si la traducción se obtiene de la TLB o, por el contrario, de la tabla o tablas de páginas residentes en la memoria principal. En este punto puede ocurrir:

1. Que la información buscada esté en memoria principal.
2. Que no esté en memoria principal (fallo de página).

En el segundo caso se realizará, como ya se vio en la sección 5.3, un cambio de contexto a otro proceso mientras se transfiere la página desde la memoria secundaria a la principal.

En el primer caso, se dispone ya de la **dirección física**, con la que se accede en primer lugar a la memoria caché. Desde el punto de vista de la caché, esta dirección consta de tres campos: etiqueta, línea donde le corresponde estar ubicada la información a la que se hace referencia y posición de dicha información dentro de la línea. Para acceder a la caché se selecciona la línea y se comprueba si su etiqueta asociada coincide con la de la dirección, pudiendo suceder:

- Que coincida (acierto en caché), dándose por finalizado el acceso.
- Que no coincida (fallo en caché), en cuyo caso se accede a memoria principal y se actualiza la caché (a no ser que el acceso sea de escritura y se utilice una política de escritura *write with no allocate*). En las figuras 34 y 35 se ha supuesto una caché con política de escritura *copy back with allocate* y sin política de lectura *out of order fetch.*

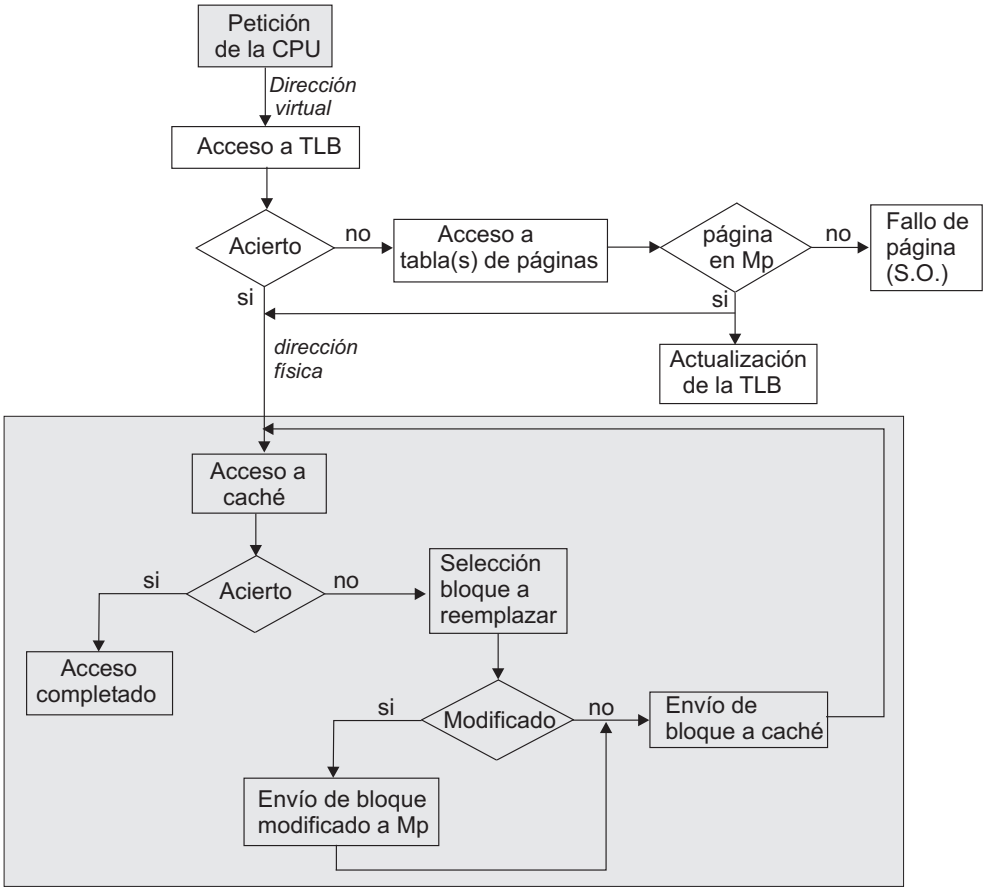


Figura 35: Diagrama de flujo de las acciones realizadas en un acceso a memoria.

En la figura 36 se muestra la interconexión entre los distintos elementos involucrados en el proceso de acceso al sistema de memoria.

Puesto que al utilizar memoria virtual se introduce un retardo en el acceso debido a la traducción, es de vital importancia tratar de reducir al mínimo este tiempo, sobre todo en el caso de acierto en caché, que por otra parte es el más probable. En el siguiente ejercicio se trata esta cuestión.

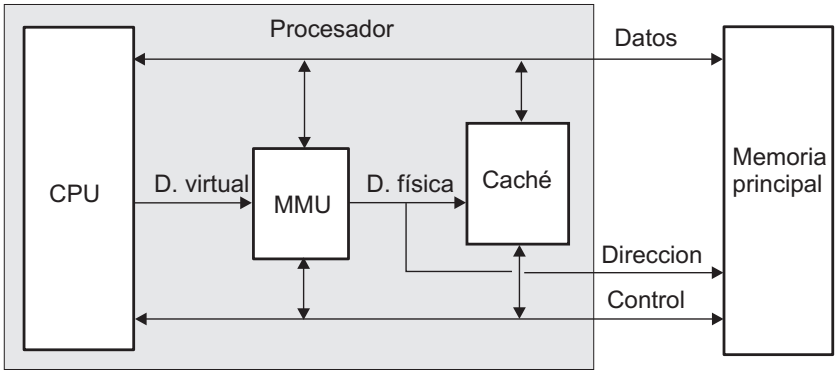


Figura 36: Interconexión entre los elementos de un sistema con memoria virtual y caché.

Ejercicio 11.- *Supongamos un sistema con memoria virtual paginada, que dispone de una TLB con tiempo de acceso de 1 ciclo y cuya tasa de aciertos, para un determinado programa, es del 95 %. Además dispone de dos niveles de tablas de páginas, ocupando cada entrada de dichas tablas una palabra, y de una memoria caché cuyo tiempo de acceso es de 2 ciclos. ¿Cuáles serán los tiempos de acceso mínimo, máximo y medio, en caso de acierto en la caché, sabiendo que el tiempo de acceso de la memoria principal es de 80 ciclos ?*

El tiempo de acceso mínimo es:

$$T_{TLB} + T_{Mca} = 1 + 2 = 3 \; ciclos.$$

El tiempo de acceso máximo:

$$T_{TLB} + T_{TPs} + T_{Mca} = 1 + 2 \times 80 + 2 = 163 \; ciclos$$

Y el tiempo medio de acceso:

$$Hr_{TLB} \times T_{TLB} + (1 - Hr_{TLB}) \times (T_{TLB} + T_{TPs}) + T_{Mca} = 0,95 \times 1 + 0,05 \times (1 + 2 \times 80) + 2 = 11 \; ciclos$$

Como se puede observar en el ejercicio anterior, en el mejor de los casos se añade el tiempo de acceso a la TLB al de la caché y, aunque el retardo introducido por la TLB es pequeño, si tenemos en cuenta que se produce en cada acceso, puede suponer un impacto importante en las prestaciones del sistema de memoria.

Para reducir el tiempo de acceso en los casos de acierto en la caché se pueden considerar dos alternativas, que se exponen brevemente a continuación:

1. Realizar la traducción en la TLB y el acceso a la caché en paralelo, en lugar de hacerlo de forma secuencial.
2. Utilizar directamente las direcciones virtuales para acceder a la caché, eliminando de este modo el tiempo adicional debido a la traducción.

6.1. Acceso simultáneo a la TLB y a la caché.

Como es sabido, la dirección virtual consta de dos tipos de información: uno que debe pasar el proceso de traducción (la página virtual) y otro que no interviene en dicho proceso, la posición dentro de la página, por lo que se puede utilizar esta última para acceder a la caché a la vez que se traduce la página en la TLB. Para dar por finalizado el acceso faltaría únicamente comprobar la coincidencia de las etiquetas.

Las memorias caché que utilizan esta técnica suelen ser las de primer nivel, y se conocen bajo el nombre de "cachés indexadas virtualmente y etiquetadas físicamente"(*virtually indexed and physically tagged caches*), a diferencia de las de segundo nivel en adelante, a las que se accede con la dirección física resultante de la traducción, denominadas "cachés indexadas y etiquetadas físicamente"(*physically indexed and physically tagged caches*).

Esta técnica impone, sin embargo, una limitación en el caso de cachés directas, y es que la capacidad de la caché no puede ser mayor que el tamaño de las páginas. Supongamos que las páginas son de 8 KB (2^{13} bytes) y las líneas de la caché de 16 bytes (2^4). Para poder acceder a la caché sin esperar a que finalice la traducción, es necesario que los restantes 13-4 = 9 bits de la dirección virtual que no se traducen sean suficientes para seleccionar todas las líneas de la caché (figura 37a), por lo que ésta debería ser como máximo de 8 KB (2^9 líneas $\times 2^4$ bytes/línea). Si el número de líneas de la caché fuese mayor de 2^9, habría que esperar a finalizar la traducción para poder seleccionar cualquiera de ellas.

Una forma de utilizar este método con cachés de tamaño sea mayor al de las páginas es utilizar cachés asociativas por conjuntos. En este caso, es necesario poder seleccionar el conjunto y el byte dentro de una línea sin tener que esperar a que finalice la traducción (figura 37b). En el ejemplo anterior, la caché debería tener 2^9 conjuntos, y si los conjuntos fuesen de dos líneas se podría utilizar una caché de 16 KB, el doble de capacidad que en el caso de la directa.

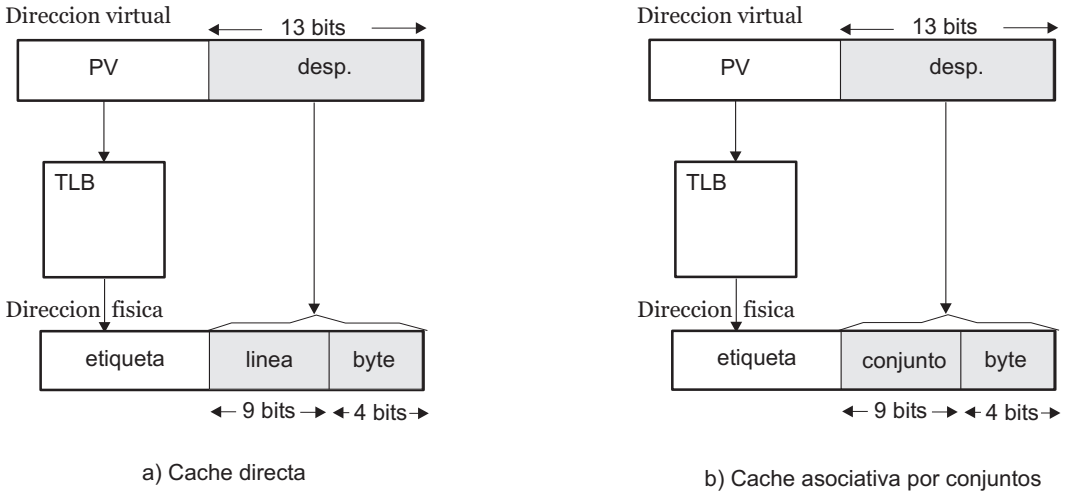


Figura 37: Acceso simultáneo a la TLB y a la caché.

6.2. Memorias caché virtuales.

El objetivo de esta solución es eliminar la etapa de traducción en el caso de acierto en la caché, lo que se consigue accediendo directamente a la caché mediante direcciones virtuales en lugar de físicas. A estas cachés "indexadas y etiquetadas virtualmente"(*virtually indexed and virtually tagged*) se les suele denominar cachés virtuales. La traducción de la dirección virtual es necesaria únicamente cuando se produce un fallo en el acceso a la caché de primer nivel, en cuyo caso hay que acceder al resto de niveles o a la memoria principal, que utilizan siempre direcciones físicas.

Aunque, en principio, puede resultar interesante el empleo de este tipo de cachés, lo cierto es que se utilizan raramente. El primer motivo es la protección de la información. Dado que en el proceso de traducción se comprueba además si el tipo de acceso que se quiere realizar está permitido, la utilización de cachés virtuales obliga a incluir esta información en su directorio y a hacer dicha comprobación cada vez que se accede a la caché.

Por otra parte, hay que tener en cuenta que lo normal es trabajar en un entorno en el que existen varios procesos, en cuyo caso podría suceder:

1. Que dos **direcciones virtuales iguales**, correspondientes a procesos diferentes, correspondan a **distinta dirección física** y, por lo tanto, a distinta información. En este caso, ante un cambio de contexto es necesario invalidar la memoria caché. Una solución para evitar la invalidación es incluir en ella el identificador de proceso o el identificador del espacio de direcciones, como se hace en el caso de la TLB (véase la sección 5.3.2).

2. Que dos **direcciones virtuales diferentes**, correspondientes a procesos diferentes, correspondan a una **misma dirección física** (**sinónimos**) y por lo tanto a la misma información en memoria principal (esto sucede con la información compartida por varios procesos). En este caso, podría ocurrir que existieran varias copias de la misma información en la caché, pudiendo dar lugar a un problema de coherencia si se modifique una de ellas. Este problema no se puede dar en las cachés a las que se accede mediante direcciones físicas, ya que la traducción de ambas direcciones daría como resultado el mismo bloque de memoria principal.

Entre los ejemplos de procesadores con cachés virtuales cabe citar las familias ARM7 hasta ARM10, incluidos los procesadores Intel StrongARM e Intel XScale.

Referencias

[BJ98] T. Mudge B. Jacob. Virtual memory in contemporary microprocessors. *IEEE Micro*, pages 60–75, August 1998.

[BO16] Randal E. Bryant and David R. O´Hallaron. *Computer Systems. A programmer´s perspective*. Pearson, third edition, 2016.

[Han93] J. Handy. *The cache memory book*. Academic Press, 1993.

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2007.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fifth edition, 2012.

[Hwa84] K. Hwang. *Computer Architecture and Parallel processing*. Mac Graw Hill, 1984.

[i4889] Intel Corporation. *i486 Microprocessor*, December 1989.

[MC97] M. Dubois M. Cekleov. Virtual-address caches. part1: Problems and solutions in uniprocessores. *IEEE Micro*, pages 64–71, September 1997.

[Mil90] M. Milenkovic. Microprocessor memory management units. *IEEE Micro*, pages 70–85, April 1990.

[Org72] E.I. Organick. *The Multics System: An Examination of Its Structures*. MIT Press, 1972.

[Pen01] *The microarchitecture of the Pentium 4 Processor*, 2001. Intel Technology Journal Q1.

[PH12] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., fourth edition, 2012.

[Smi82] A.J. Smith. Cache memories. *Computing Surveys*, pages 473–530, September 1982.

[Sta10] William Stallings. *Computer Organization and Architecture*. Pearson Prentice Hall, eigth edition, 2010.

[Sto93] Harold S. Stone. *High Performance Computer Architecture*. Addison Wesley, Reading, MA, third edition, 1993.

[Tan09] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, third edition, 2009.

[www16] www.cpu-world.com, 2016.

[García00] M. Isabel García y otros. *Estructura de Computadores. Problemas y soluciones*. RA-MA, 2000.

[García06] M. Isabel García y otros. *Estructura de Computadores. Problemas resueltos*. RA-MA, 2006.

Índice alfabético