TEMA 5: ADMINISTRACIÓN DE MEMORIA

INTRODUCCIÓN

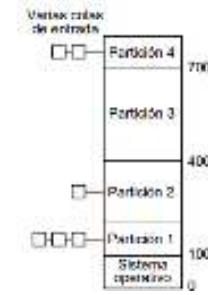
Para poder construir un almacenamiento muy rápido, de uso exclusivo, grande y no volátil, 0 similar, hay que combinar distintas tecnologías de almacenamiento a través de una *jerarquía de memoria*. En ella, la memoria más rápida, pequeña y cara se encontrará cerca del procesador, y la más lenta, grande y barata estará lejos del procesador. En la actualidad, esta jerarquía está formada por unos pocos megabytes de memoria caché, unos cuantos gigabytes de memoria RAM y varios cientos de gigabytes o algunos terabytes de almacenamiento de disco.

La jerarquía de memoria y los detalles de su funcionamiento deben ocultarse. A la parte del sistema operativo (RAM y disco) que se encarga de esta tarea se le llama administrador de memoria.

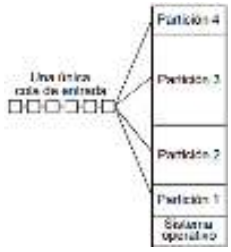
ADMINISTRACIÓN DE MEMORIA SIN MEMORIA VIRTUAL

MULTIPROGRAMACIÓN CON PARTICIONES FIJAS

El esquema más sencillo parar tener varios procesos a la misma vez en memoria es asignar una zona de tamaño fijo al sistema operativo y dividir el resto en particiones.



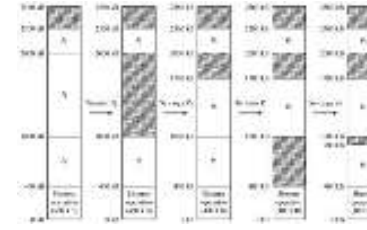
Las diferentes particiones se reparten entre los procesos según se van creando estos. Una opción, es tener **una cola por partición** y colocar cada trabajo en la cola de la partición más pequeña en la que quepa. El inconveniente de esta propuesta es que las colas de las particiones grandes casi siempre estarán vacías mientras que las colas de las particiones pequeñas serán largas; y hacer todas las particiones pequeñas haría imposible ejecutar los trabajos grandes.



Otra posibilidad es tener **una única cola para todas las particiones**. Cuando una partición queda libre, se busca la primera tarea de la cola que quepa en dicha partición, se busca en toda la cola el trabajo más grande que quepa; sin embargo, discriminaría a las tareas pequeñas. Una solución a este último problema sería tener una partición pequeña. Otra solución sería establecer un límite L.

En este esquema, el *grado de multiprogramación*, se ve limitado por el número de particiones. Otro problema es que los procesos pequeños producen mucha fragmentación interna. Finalmente, un tercer problema es que también se puede producir fragmentación externa.

MULTIPROGAMACIÓN CON PARTICIONES VARIABLES



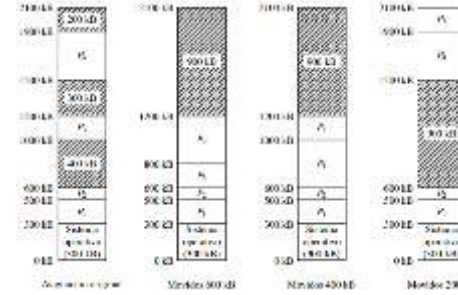
Por lo general, existen más procesos de usuario de los que puede albergar la memoria, por lo que el exceso de procesos debe estar en disco y estos necesitan pasar a memoria en algún momento; y hacerlo con particiones fijas no sería óptimo.

POLÍTICAS DE ASIGNACIÓN DE HUECOS

- **Primero en ajustarse:** el administrador de memoria revisa los huecos desde el pri-ncipio de la memoria hasta encontrar uno lo suficientemente grande. El hueco se divide en dos partes: una para el proceso y otra que pasa a ser un hueco libre. Este algoritmo es rápido.
- **Siguiente en ajustarse:** el funcionamiento es el mismo, salvo que la búsqueda del hueco no se inicia siempre desde el principio de la memoria, sino desde el punto en el que se quedó la búsqueda anterior. Este algoritmo es rápido.
- **Mejor en ajustarse:** busca entre todos los huecos el más pequeño en el que quepa el proceso. La idea es buscar un hueco cercano al tamaño real necesario. Esta política es lenta y suele desperdiciar más memoria que otras políticas.

- **Peor en ajustarse:** toma siempre el hueco libre más grande. Su objetivo es que el nuevo hueco obtenido sea suficientemente grande para ser útil. Esta política es lenta y funciona bastante bien respecto al aprovechamiento de la memoria.

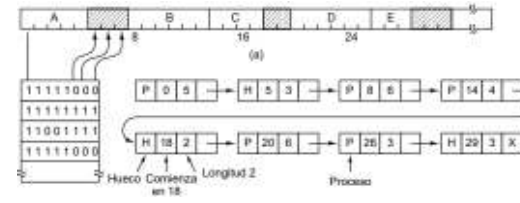
COMPACTACIÓN



En la multiprogramación con particiones variables puede producirse una gran fragmentación externa. Debido a la flexibilidad de las particiones variables, podemos combinar todos los huecos en uno solo si movemos algunos procesos en memoria.

Generalmente, la **compactación** no se suele utilizar porque consume mucho tiempo de CPU. Además, de la posible fragmentación externa, siempre existe fragmentación interna, pues a los procesos se les asigna más memoria de la estrictamente necesaria que permite que tanto la pila como la zona de datos puedan crecer.

ADMINISTRACIÓN DE LA MEMORIA LIBRE



En la **administración de memoria con mapas de bits** se divide la memoria en unidades de asignación (bloques), todas del mismo tamaño. A cada unidad de asignación le corresponde un bit en el mapa de bits.

Si la unidad de asignación es pequeña, se necesitará un mapa de bits grande, pero si el tamaño de la unidad es grande, se puede perder memoria por fragmentación interna.

En la **administración de memoria con listas ligadas** se mantiene un registro de la memoria mediante una lista ligada de los segmentos de memoria asignados o libres. Cada entrada de la lista especifica la dirección donde comienza el segmento, su longitud y un apuntador a la siguiente entrada de la lista.

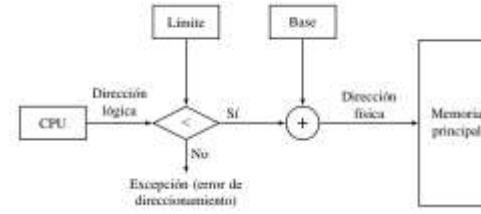
Cuando un proceso termina o se intercambia, se debe fusionar el hueco que deja con los huecos adyacentes, si los hay. Esta fusión no es necesaria si se utiliza un mapa de bits.

ASIGNACIÓN DEL HUECO DE INTERCAMBIO

Los procesos suspendidos que no caben en memoria se guardan en una zona del disco conocida como **área de intercambio**, puede ser una partición o un fichero.

Los algoritmos para la administración del área de intercambio, tanto para llevar un registro como para hacer un reparto del espacio de intercambio, pueden ser los mismos que para la administración de la memoria principal. La única diferencia es que el hueco en disco necesario para un proceso debe representarse como un número entero de bloques del disco.

REUBICACIÓN Y PROTECCIÓN



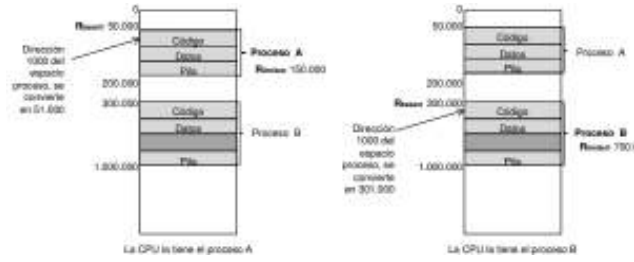
La multiprogramación permite tener varios procesos a la vez, lo que da lugar a dos problemas: reubicación y protección.

El problema de la **reubicación** surge porque, cuando un programa se ejecuta, su proceso puede ir a cualquier partición. Las particiones se encuentran en direcciones de memoria distintas y no se sabe a qué partición va. Por lo que, el programa debe utilizar código relocalizable, código que se pueda ejecutar independientemente de las direcciones.

El problema de la **protección** aparece porque hay que proteger al sistema del resto de procesos y a un proceso de los demás. Un mecanismo hardware que soluciona ambos es un *registro base y límite*. Se encuentran en la CPU. Cuando se asigna el registro base se carga con la dirección de memoria donde comienza la partición y el registro límite se carga con el tamaño de dicha partición. Estos valores se cogen del BCP del proceso. Si se asigna a otro proceso estos registros se cargan de nuevo.

El código del proceso que se está ejecutando en la CPU puede generar cualquier dirección desde *0 hasta* $T_p - 1$. Si en su ejecución el proceso genera una dirección mayor se producirá una excepción. En caso contrario, se tratará de una dirección válida a la que se le sumará el valor del registro base para obtener la dirección final.

Hay una separación entre las direcciones de memoria generadas por los procesos (***direcciones lógicas***) y las direcciones de memoria que finalmente usan (***direcciones físicas***). El conjunto de direcciones lógicas que pueden generar un proceso constituye su espacio de direcciones lógicas. Este de todos los procesos comienzan en la dirección 0. Sin embargo, como a un proceso se les suma su registro base, y este es distinto para cada proceso, dos direcciones lógicas iguales de procesos diferentes corresponderán con direcciones físicas distintas.

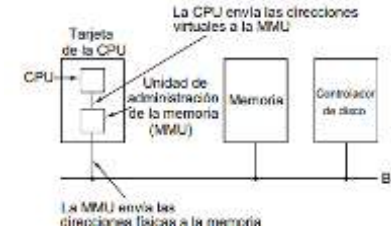


PAGINACIÓN

La **memoria virtual** es un esquema de administración de memoria que permite ejecutar programas cuyo tamaño total puede exceder la cantidad de memoria física que se les puede asignar. Para conseguir esto, el sistema mantiene en memoria principal solo las partes del proceso que se están usando y guarda el resto del proceso en disco.

FUNCIONAMIENTO DE LA PAGINACIÓN

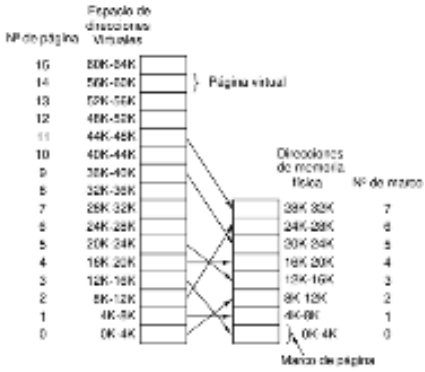
En la **paginación**, la división de un proceso en trozos y el intercambio de estos trozos entre la memoria principal y el disco son por completo responsabilidad del hardware y del sistema operativo. Un proceso no es consciente de que solo parte de su espacio de direcciones está en memoria física. Se distingue entre las direcciones que un proceso puede generar (***direcciones virtuales***), y las direcciones de memoria principal (***direcciones físicas***).



Todas las posibles direcciones virtuales que puede generar un proceso conforman su *espacio de direcciones virtuales*. El tamaño de este espacio viene limitado por las características físicas de la CPU, como el tamaño del bus de direcciones.

Al utilizar memoria virtual, las direcciones virtuales no pasan de forma directa al bus de memoria, sino que van a una **unidad de administración de memoria** (***MMU***) que asocia las direcciones virtuales con sus correspondientes direcciones de memoria física. Aunque la MMU es una unidad distinta de la CPU, se suele integrar en el mismo chip.

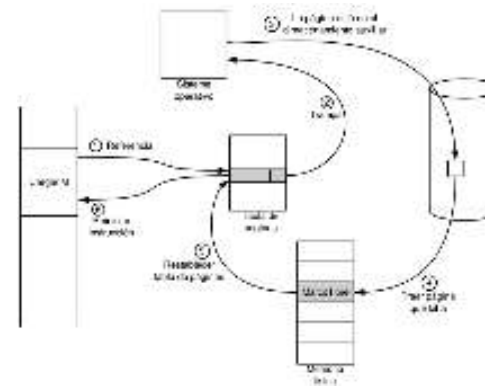
La **paginación** divide el espacio de direcciones virtuales en unidades llamadas ***páginas*** y la memoria física en unidades llamadas ***marcos de página***. Las páginas y los marcos tienen siempre el mismo tamaño. Las transferencias entre memoria y disco son siempre en unidades de página.



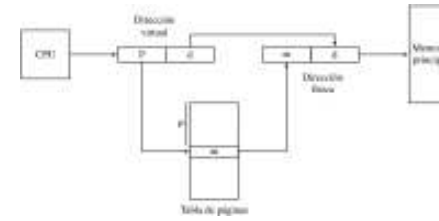
Los datos almacenados en cada página de la memoria virtual realmente se almacenan en los marcos de página de la memoria física. Por ello, cada página debe estar asociada a un marco para que las direcciones virtuales generadas por el proceso y pertenecientes a una determinada página puedan ser traducidas en direcciones físicas del marco correspondiente.

En la práctica, para saber en qué marco se encuentra cada página se utiliza **una tabla de páginas**, una tabla con tantas entradas como páginas virtuales sean posibles y donde cada entrada indica en qué marco se almacena la página virtual asociada a la entrada. Todo esto no resuelve el problema de que el espacio de direcciones virtuales es más grande que la memoria física.

Los pasos para resolver un *fallo de página* son:



1. Durante la ejecución de un proceso, la CPU genera una dirección virtual cuya pagina no se encuentra en memoria.
2. Se produce un fallo de página. La excepción provoca un cambio de modo y el sistema operativo toma el control.
3. El sistema operativo localiza el contenido de la página que nos interesa en disco.
4. El sistema operativo carga el contenido de la página en un marco libre; si no existe ninguno, tendrá que liberar uno.
5. La información de la tabla de páginas se debe actualizar.
6. Tras resolver el fallo de página, se debe reiniciar la instrucción que lo produjo.



Para saber en qué página se encuentra una dirección virtual, podemos hacer operaciones aritméticas pero la traducción sería muy costosa.

La solución pasa por evitar cualquier tipo de operación aritmética y para ello es necesario que el tamaño de las páginas sea potencia de 2. En lugar de hacer operaciones aritméticas, una dirección virtual en binario se divide en dos campos: un primer campo de 4 bits llamado número de página y un segundo campo de 12 bits llamado ajuste o desplazamiento.

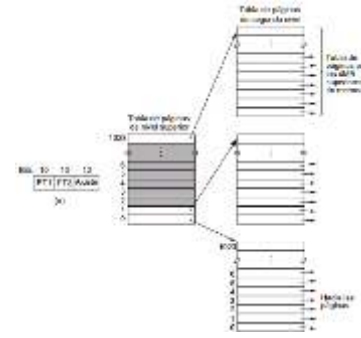
TABLA DE PÁGINAS

La finalidad de la tabla de páginas es asociar las páginas virtuales con los marcos. La tabla puede ser muy grande y debe ser rápida. Además, cada proceso necesita su propia tabla de páginas para tener su propio espacio de direcciones virtuales. Con espacios de direcciones separados, el sistema operativo está protegido de los procesos y estos entre sí.

El diseño más sencillo es tener una sola tabla de páginas consistente en un **array de registros rápidos** en hardware con una entrada por cada página virtual. Al asignar la CPU a un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tabla que se encontrará en la memoria principal. A partir de aquí, la tabla no provoca más referencias a memoria al hacer la asociación. Ventaja que las traducciones no suponen referencias a memoria; inconvenientes que es costoso en recursos y que cargar la tabla en cada cambio de proceso puede reducir el rendimiento.

El otro extremo es tener la tabla de páginas totalmente dentro de la memoria principal. En este caso, todo lo que necesita el hardware es un solo registro, r, que apunte al inicio de la tabla de páginas del proceso en ejecución. Los datos de la entrada de la página p se encontrarán en la dirección física de memoria $r + p \cdot t$, siendo t el tamaño de una entrada de la tabla de páginas. Ventaja es posible cambiar el mapa de memoria en un cambio de proceso cargando un único registro (el r); inconvenientes se necesitan una o más referencias a memoria, lo cual puede degradar seriamente el rendimiento de la memoria. Dos soluciones intermedias son las tablas de páginas de varios niveles y el TLB. También veremos una tercera solución, las tablas de páginas invertidas.

TABLA DE PÁGINAS DE VARIOS NIVELES



Para evitar el problema de tener unas tablas inmensas en la memoria es necesario dividir el número de página virtual en 2 o más partes.

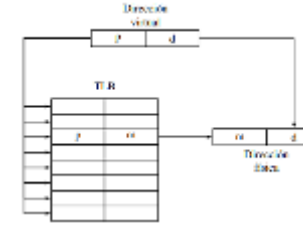
El secreto de este método consiste en no tener al mismo tiempo todas las tablas en la memoria; las que no son necesarias, no se tienen.

Esta estructura de niveles se puede aumentar a 3 niveles, 4 niveles o más. Cuantos más niveles, mayor flexibilidad, pero también, mayor complejidad.

Puesto que el objetivo de una tabla de páginas de varios niveles es reducir la cantidad de memoria necesaria, si con el transcurso del tiempo hay tablas de páginas del nivel más bajo que se quedan sin información útil, estas tablas se pueden eliminar de memoria para liberar el espacio que ocupan.

TLB

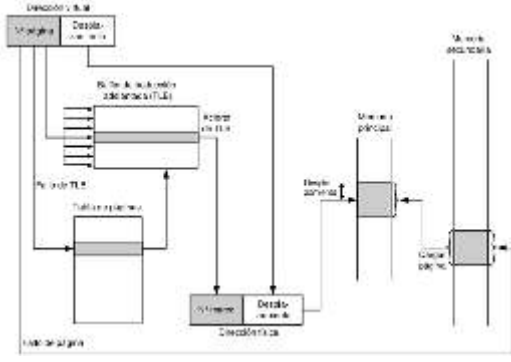
Hasta ahora, hemos supuesto que las tablas de páginas se mantienen en la memoria. Esto hace que usar la paginación suponga hacer más referencias a memoria que cuando no se utiliza, por lo que el rendimiento de la jerarquía de memoria baja.



La solución a este problema es equipar al ordenador con un pequeño dispositivo hardware para traducir las direcciones virtuales a direcciones físicas sin tener que ir a la tabla de páginas en memoria en cada traducción. Este dispositivo se llama **TLB** y es una pequeña memoria totalmente asociativa, generalmente dentro de la MMU, que contiene entradas de la tabla de páginas. La estructura de estas:

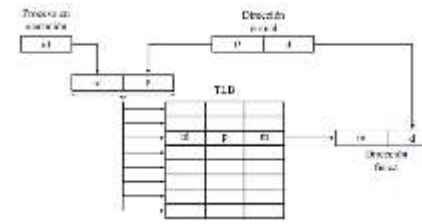
- Un bit de validez, que nos indica si la entrada del TLB tiene información útil o no.
- El número de página para la que la entrada del TLB contiene información.
- Un bit de modificación que indica si el contenido ha sido modificado.
- Diversos bits de protección.
- El número de marco.

Validez	Página	Bit de modificación	Protección	Marco
1	140	1	rw-	31



Lo ideal sería tener toda la tabla de páginas en el TLB. El problema está en que el número de entradas en el TLB suele ser muy pequeño. La solución es adoptar una solución intermedia, es decir, tener la tabla de páginas en memoria principal y almacenar en el TLB copias de aquellas entradas de la tabla de páginas que más se utilizan. Cuando la MMU recibe una dirección virtual para su traducción, el hardware verifica en primer lugar si su número de página virtual se encuentra en el TLB, comparando todas las entradas en paralelo. Si coincide con alguno y el acceso no viola los bits de protección, el marco de página se toma del TLB. Obsérvese que solo se puede producir un acierto de TLB si la página correspondiente

está en memoria, ya que al TLB solo se copian entradas de páginas que están en memoria, es decir, que no producen fallos de página.



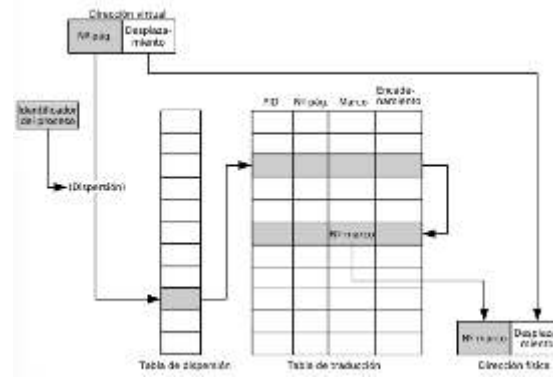
Si el número de página virtual no está en el TLB, la MMU detecta la falta y hace una búsqueda normal en la tabla de páginas. Elimina entonces una entrada del TLB y la reemplaza con la entrada leída de la tabla de páginas. Al eliminar una entrada del TLB, el bit de modificación de la página en el TLB se copia en la entrada de la tabla de páginas en la memoria. Si la página no está en memoria, entonces se producirá un fallo de página que habrá que resolver antes de tratar el fallo de TLB.

Cuando se tienen varios procesos en ejecución cada uno tiene su tabla de páginas. Al cambiar de proceso, la tabla de páginas que se debe usar es la del nuevo proceso, por lo que la MMU no debe utilizar las entradas del TLB que contienen información del proceso anterior. Para solucionar este problema:

- Invalidar el contenido del TLB borrando los bits de validez.
- Añadir un nuevo campo a cada entrada del TLB con la identificación del proceso al que pertenece.

TABLA DE PÁGINAS INVERTIDAS

En muchos procesadores actuales las tablas de páginas pueden ser enormes. Para solucionar este problema, podríamos pensar en usar páginas más grandes, pero el tamaño debería ser considerable para reducir el número de entradas de las tablas de páginas, y eso haría que se perdiera mucha memoria por fragmentación interna. Podríamos pensar también en usar tablas de páginas multinivel, pero el número de niveles necesarios sería elevado, lo que haría la gestión de estas tablas demasiado compleja.



Existen unas tablas de páginas que nos permite usar de forma eficiente direcciones virtuales de 64 bits y páginas pequeñas. Estas tablas se llaman **tablas de páginas invertidas**.

En una tabla de páginas invertida, existe una tabla de traducción donde hay una entrada por cada marco de página de la memoria física, por lo que su tamaño solo depende de la cantidad que tengamos de esta memoria. Además, esta tabla de traducción es única, por lo que todos los procesos la comparten.

Cada entrada de la tabla de traducción contiene información sobre la página virtual que se encuentra en un marco de memoria. En concreto, almacena el PID del proceso al que pertenece la página virtual, el número de dicha página virtual y el marco de memoria donde se almacena la página. Con esta información hay que buscar en toda la tabla si hay una entrada cuyo PID sea el del proceso y cuyo número de página virtual sea el correspondiente a la dirección generada. Si la entrada existe, entonces se toma de ella el número de marco y se termina la traducción. En caso contrario, se produce un fallo de página. Obsérvese que, aunque la tabla de traducción tiene tantas entradas como marcos, no hay una asociación entre entradas y marcos (la entrada N no tiene porque ser información del marco N).

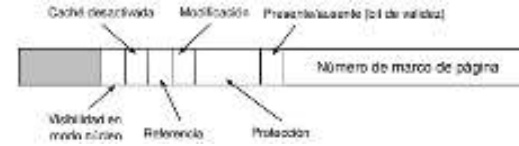
Como la búsqueda en toda la tabla haría muy lentas las traducciones, para mejorar el rendimiento se usa una **tabla de dispersión** y un **TLB**. Existe una función de dispersión que recibe como datos de entrada el identificador del proceso y el número de la página virtual. Como salida, la función devuelve

una entrada de la tabla la cual contiene la posible entrada de la tabla de páginas asociada. Puesto que se pueden dar colisiones, existe un campo en cada entrada de la tabla de traducción que permite crear una lista con las entradas a las que les corresponde el mismo valor de dispersión. Un fallo de TLB no supone recorrer toda la tabla, solo la lista de entradas correspondientes a un cierto valor.

Los procesos pueden compartir marcos. Esto hace que para un mismo marco puedan existir dos o más entradas asociadas en la tabla de páginas invertida, por lo que la tabla de traducción habitualmente tendrá más entradas que marcos.

ESTRUCTURA DE UNA ENTRADA

La estructura de una entrada de la tabla de páginas. En el caso de las tablas de páginas invertidas, la estructura es además de los campos habituales (número de marco de página y bit presente/ausente) encontramos:

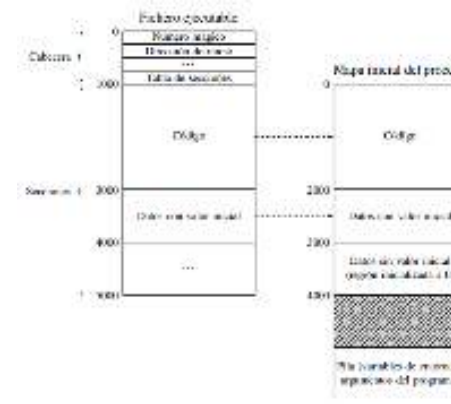


- **Los bits de protección:** indican el tipo de acceso permitido a la página.
- **El bit de modificación:** indica si se ha modificado (1) o no (0) el contenido de una página. Si el SO hace un cambio de página en un marco si el bit esta a 1 hay que guardar en memoria los cambios.
- **El bit de referencia:** toma valor 1 cuando se hace referencia (leer o escribir) a una página.
- **El bit de caching desactivado:** se utiliza en las máquinas con E/S mapeada por memoria para evitar que el contenido de la página se almacene en la memoria caché.
- **El bit de visibilidad en modo núcleo:** tiene que ver con el mapa de memoria de un proceso.

En el caso de las tablas de páginas invertidas, las entradas tienen básicamente la misma estructura, aunque en ellas aparecen algunos campos más, como el identificador del proceso, el número de página asociado y un puntero para el encadenamiento de entradas asociadas a una misma entrada de la tabla de dispersión.

MAPA DE MEMORIA DE UN PROCESO

Para poder ejecutarse, un proceso necesita tener en memoria su código, una zona de datos y una pila. Inicialmente, el código y los datos se obtienen desde el fichero ejecutable, mientras que la pila y la zona de datos se crean cuando el proceso también se crea. Generalmente, el código y los datos se encuentran al principio del espacio de direcciones de un proceso, mientras que la pila se encuentra al final. A la forma en la que se estructura el espacio de direcciones lógicas o virtuales se llama **mapa de memoria** del proceso.



En el sistema sin memoria virtual el mapa de memoria de un proceso suele ser sencillo: al proceso se le asigna una zona contigua de memoria física y en ella se colocan su código, datos y pila.

En un sistema con paginación, el mapa de memoria puede llegar a ser bastante más complejo, ya que el espacio de direcciones virtuales se divide en páginas y no todas ellas tienen por qué estar en memoria.

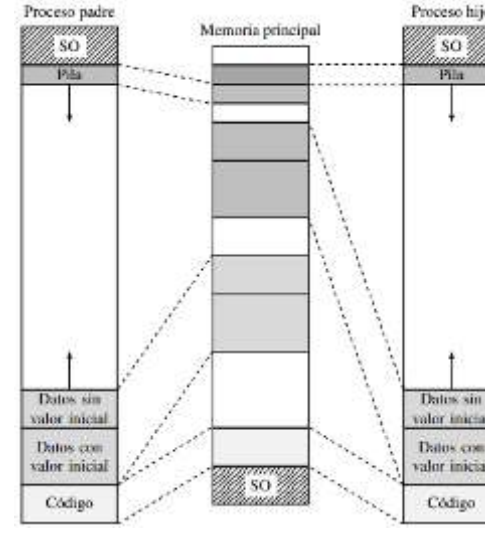
El mapa de memoria debe controlar también qué zonas del espacio de direcciones virtuales están ocupadas y cuáles no. A las zonas contiguas de memoria virtual ocupadas, dedicadas normalmente a un mismo propósito, las llamamos **regiones**. Estarán en la región código, región de datos y región pila. Cada región de memoria tiene asociada:

- **Soporte**: describe de dónde se obtienen los datos que contiene la región.
- **Tipo de compartición**: indica si las modificaciones en una región se comparten con otros procesos o no. Puede ser privada, si las modificaciones solo son visibles para el proceso que hace las modificaciones, o compartida, si son visibles por otros.
- **Protección**: nos dice si el contenido de una página se puede leer, escribir o ejecutar.
- **Tamaño fijo o variable**: indica si el tamaño de una región puede cambiar o no.

Además de las regiones vistas, los procesos suelen usar una región dedicada a **la memoria montón** o **heap** (da soporte a las reservas dinámicas de memoria), comienza tras la región de datos y crece hacia direcciones superiores. No tiene soporte asociado (esta inicialmente a 0) y va creciendo. Mediante las llamadas al sistema `brk()` y `sbrk()`. Otro ejemplo son los procesos con hilos, en los que se crea una región para la pila de cada hilo.

El sistema operativo aparece en la parte alta del espacio de direcciones virtuales de un proceso. Esto tiene diversas ventajas. Hay llamadas al sistema que requieren copiar información del sistema operativo al proceso. Con esta organización, la transferencia de datos se hace con una simple copia de bytes de una zona de memoria a otra. Para evitar que los procesos puedan leer o escribir en el sistema operativo, este solo es visible en modo núcleo. En modo usuario, la región de memoria ocupada por el sistema operativo es inválida y no puede ser accedida, produciría una violación de acceso.

La organización del mapa de memoria de un proceso en regiones y el uso de la paginación permiten implementar eficientemente la llamada al sistema `fork()`. Cuando se crea un proceso hijo, las páginas del hijo apuntarán a los mismos marcos que las correspondientes páginas del padre. El hijo hereda el mapa de memoria, entonces, las regiones del hijo estarán proyectadas en las mismas zonas de memoria física.



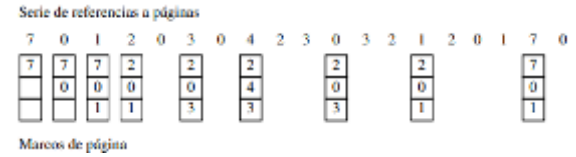
En el caso del código y de otras regiones sin permiso de escritura, la compartición de la memoria física no supone ningún problema, pues el contenido nunca se va a modificar. En las regiones de datos, pila y otras regiones modificables; la compartición se tiene que hacer cuando un proceso modifique un dato, el resto de los procesos que comparten la misma zona de memoria física no vean el cambio. Una forma de solucionar esto es utilizar la copia en escritura. Esta técnica permite que en un primer momento todas las páginas de las regiones modificables se compartan, si bien se desactiva el permiso de escritura en las páginas de esas regiones para detectar cuándo se quiere modificar una. Ahora, si uno de los dos procesos intenta escribir en una de esas páginas, saltará una excepción, el SO comprobará que la página se encuentra en una región que está compartida y es modificable, y hará una copia de la página que ha provocado la violación de acceso a un marco nuevo. Tras esto, asignará el nuevo marco a la página que se desea escribir cambiando su entrada en la tabla de páginas de su proceso. Después, desactivará la protección contra escritura de las páginas que compartían el marco, cambiando de nuevo la información almacenada en las tablas de traducción de los procesos. Finalmente, el sistema operativo reiniciará la ejecución de la instrucción que provocó la excepción.

ALGORITMOS DE REEMPLAZO DE PÁGINAS

Cuando ocurre un **fallo de página** y no hay marcos libres disponibles, el sistema operativo debe elegir una página. La forma de elegir la página que hay que eliminar de memoria da lugar a varios algoritmos de reemplazo de páginas.

ALGORITMO ÓPTIMO

Según este algoritmo, de todas las páginas que hay en memoria y que se pueden expulsar, se elimina aquella para la que pasará más tiempo antes de que sea utilizada de nuevo. Este algoritmo es irrealizable, ya que el sistema operativo no tiene forma de saber a qué página se hará referencia más tarde. No obstante, es útil, ya que, mediante simulación, se puede estudiar cómo de buenos o malos son otros algoritmos realizables.



ALGORITMO NRU: LA NO USADA RECIENTEMENTE

En cada entrada a de la tabla de páginas existe un bit de referencia (**R**) utilizado para saber si se ha hecho referencia y un bit de modificación (**M**) para saber si una página se ha modificado o no. Una vez el hardware activa cualquiera de estos 2 bits, los mismos permanece en ese estado hasta que el sistema operativo los desactiva por software.

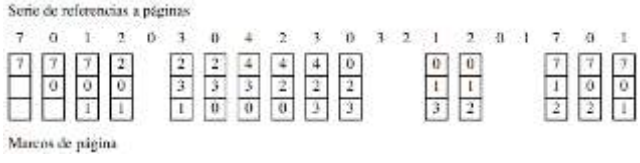
Durante la ejecución de un proceso, los bits R y M de cada página se activan según sea necesario. De forma periódica se limpia el bit R para distinguir las páginas que no tienen referencias recientes de las que sí. Al producirse un fallo de página, si es necesario expulsar una página, el sistema operativo las inspecciona todas y las divide en cuatro categorías, según los valores de los bits R y M.

Clase	R	M
0	0	0
1	0	1
2	1	0
3	1	1

Finalmente, el algoritmo elimina una página al azar de la primera clase no vacía de número más pequeño. Es decir, si hay páginas de la clase 0, elimina cualquiera de ellas, si no, pasa a la clase 1, y así sucesivamente. Este algoritmo es sencillo, de implementación eficiente y con un rendimiento adecuado muchas veces.

ALGORITMO FIFO: PRIMERA EN ENTRAR, PRIMERA EN SALIR

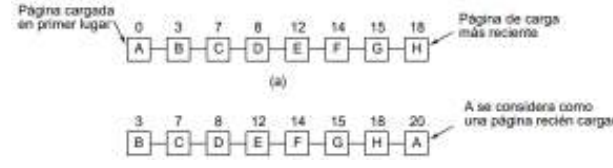
El sistema operativo tiene una lista de todas las páginas que se encuentran en memoria, siendo la primera página la más antigua y la última la más reciente. En un fallo de página, se elimina la primera página y se añade la nueva al final de la lista.



Este algoritmo es sencillo de entender y de implementar, así como de bajo coste. Sin embargo, su gran inconveniente es que no tiene en cuenta ningún dato adicional, como la frecuencia de uso de una página, lo que hace que muchas veces produzca demasiados fallos.

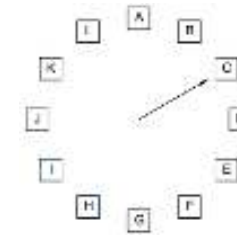
ALGORITMO DE LA SEGUNDA OPORTUNIDAD

Es una modificación simple del algoritmo FIFO que evita deshacerse de una página de uso frecuente. Cuando es necesario expulsar una página, si el bit R de la primera página es 0, dicha página se elimina de memoria. Si, por el contrario, el bit es 1, el bit se limpia y la página se coloca al final de la lista, como si hubiera llegado



en ese momento a la memoria. Después continúa la búsqueda. Si el bit de todas las páginas es 1, este algoritmo deriva en un simple FIFO, lo que asegura que este algoritmo siempre termina.

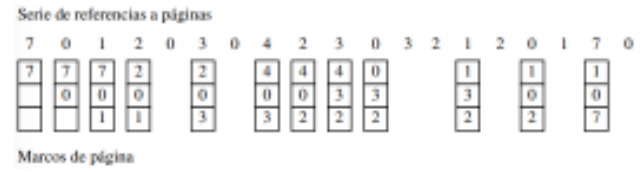
ALGORITMO DEL RELOJ



Este algoritmo difiere del anterior solo en la implementación. El algoritmo de la segunda oportunidad es razonable, es ineficiente. Un mejor enfoque es mantener las páginas en una lista circular donde una «*manecilla*» (puntero) apunta a la página más antigua. Al ocurrir un fallo de página, se inspecciona la página a la que apunta la manecilla. Si su bit R es 0, la página se expulsa de memoria, se inserta la nueva página en su lugar en el reloj y la manecilla avanza una posición. Si R es 1, este bit se pone a 0 y la manecilla avanza a la página siguiente

ALGORITMO LRU: LA USADA MENOS RECIENTE

Es una buena aproximación al algoritmo óptimo, ya que se basa en una idea que se aproxima a la de este. Esta idea es que es probable que las páginas que no hayan sido utilizadas durante mucho tiempo permanezcan sin ser usadas durante bastante tiempo. Teniendo en cuenta esto, este algoritmo dice que, cuando es necesario expulsar una página, se elimina de memoria la que no ha sido utilizada desde hace más tiempo.



El principal inconveniente de este algoritmo es cómo implementarlo de forma eficiente. Se puede pensar en utilizar una lista ligada de todas las páginas que hay en memoria. El problema de esta lista radica en que tendría que actualizarse en cada referencia a memoria: la página que se acaba de utilizar debería buscarse en la lista, eliminarse de su posición y trasladarse al frente. Esto haría los accesos a memoria más lentos.

Existen otras formas más eficientes de implementar el algoritmo LRU, siempre que el hardware proporcione el soporte adecuado. Una opción es tener en hardware un contador especial C de 64 bits que se incremente automáticamente tras cada referencia a memoria. También, en cada entrada de la tabla de páginas, tendremos un campo de tamaño adecuado donde copiar C. Las entradas del TLB también dispondrán de este campo. Tras cada referencia a memoria, se incrementa el valor de C y se almacena en el TLB, en la entrada correspondiente a la página a la que se hizo referencia; al no acceder a memoria principal, esta operación es muy rápida. Los valores del contador almacenados en las entradas del TLB se guardan en la tabla de páginas según se van reemplazando entradas por fallos de TLB. Al producirse un fallo de página, el sistema operativo examina todos los contadores de todas las tablas de páginas y elige el mínimo. Esa es la página que se utilizó hace más tiempo.

Otra opción es, si se tienen N marcos en memoria, utilizar una matriz de N × N bits implementada en hardware, cuyos datos iniciales son todos 0. En una referencia al marco K, el hardware primero activa todos los bits de la fila K y después desactiva todos los bits de la columna K. En cualquier instante, la fila cuyo valor en binario sea mínimo corresponderá al marco que se utilizó hace más tiempo.

ALGORITMO DE MADURACIÓN

Es un intento de simular el algoritmo LRU en software. Cada entrada de la tabla de páginas contiene un contador. En cada interrupción de reloj (llamada marca), se desplaza cada contador un bit a la derecha, el valor del bit R se añade al bit del extremo izquierdo del contador correspondiente y se limpia el bit R. Al ocurrir un fallo de página, se elimina aquella cuyo contador tenga el valor más pequeño.

Hay dos diferencias importantes de este algoritmo con el LRU. La primera es que puede eliminarse una página de memoria sin ser exactamente la que se utilizó hace más tiempo. La segunda diferencia es que los contadores tienen un número finito de bits.

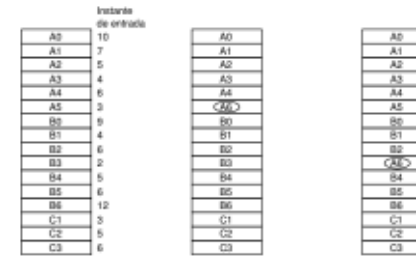
ALGUNOS ASPECTOS DE DISEÑO PARA LOS SISTEMAS DE PAGINACIÓN

La descripción de la paginación que hemos realizado es una simplificación de lo que ocurre en un sistema real.

POLÍTICAS DE REEMPLAZO Y POLÍTICAS DE ASIGNACIÓN

Aunque cada proceso tiene su propia tabla de páginas, todos comparten los marcos que hay en memoria física. Cuando un proceso produce un fallo de página y es necesario reemplazar una página. Y el tipo de reemplazo utilizado está estrechamente relacionado con el tipo de asignación de marcos.

	Remplazo local	**Remplazo global**
Asignación fija	El nº de marcos asignados a un proceso es fijo. La página a reemplazar se elige de entre los marcos asignados al proceso.	No es posible
Asignación dinámica	El nº de marcos asignados a un proceso puede cambiar de un momento a otro. La página a reemplazar se elige de entre los marcos asignados al proceso.	La página a reemplazar se elige de entre todos los marcos disponibles en la memoria principal; esto hace que cambie el número de marcos asignados a cada proceso.



De las tres combinaciones posibles, el reemplazo local con asignación dinámica es bastante interesante. Por un lado, evita que un proceso que de repente empieza a producir fallos de página les quite páginas a otros procesos. Por otro lado, es más flexible que una asignación fija, ya que podemos aumentar el número de marcos de un proceso si se detecta que este los necesita. Un algoritmo de asignación que funciona bien con esta combinación es el algoritmo de frecuencia de fallos de página: si un proceso produce muchos fallos de página, le asigna más marcos, y si produce pocos fallos, sus marcos se asignan a otros procesos. La idea es tratar de que las tasas de fallos de página de todos los procesos se mantengan dentro de unos límites razonables.

Dos últimos aspectos relacionados con la asignación de marcos a los procesos son el número mínimo de marcos que debe tener un proceso que depende del hardware, en concreto, del número máximo de páginas que se pueden utilizar en una única instrucción. Y la cantidad inicial de marcos asignados a un proceso que se pueden diseñar varias políticas: la misma cantidad inicial para todos los procesos, proporcional al tamaño de cada proceso, etc.

TAMAÑO DE PÁGINA

Las paginas *pequeñas* producen menos fragmentación interna. Sin embargo, dan lugar a tablas de páginas más *grandes* que hacen que: el cambio de proceso sea más lento si hay entradas de la tabla de páginas del nuevo proceso que deben cargarse en el TLB y, si el proceso es pequeño, utilizará pocas entradas de la tabla y el espacio ocupado por el resto de las entradas se desperdiciará. Por otro lado, si las páginas son grandes, entonces las ventajas/inconvenientes de las páginas pequeñas se convierten en inconvenientes/ventajas de las páginas grandes.

Otro factor importante a tener en cuenta es que las transferencias entre memoria y disco son de una página. Si las páginas son pequeñas, leer X bytes de disco puede suponer leer muchas páginas mientras que, si las páginas son grandes, se deben leer pocas, lo cual puede compensar. En general, los tamaños de página de 4 y 8 KiB son frecuentes.

HIPERPAGINACIÓN

Si un proceso da lugar a muchos fallos de página puede entrar en **hiperpaginación**. Esta se produce si el proceso emplea más tiempo esperando a que se resuelvan sus fallos de página que ejecutando código. En definitiva, la hiperpaginación se produce porque se necesitan muchos más marcos de los que se dispone.

Se pueden adoptar distintas soluciones para paliar el problema. La más inmediata es aumentar la capacidad de la memoria principal. Otra, que puede ser llevada a cabo por el sistema operativo es suspender temporalmente algunos procesos para liberar memoria y reanudarlos cuando el porcentaje de fallos de página baje hasta un nivel aceptable.

POLÍTICAS DE LECTURA Y ESCRITURA DE PÁGINAS

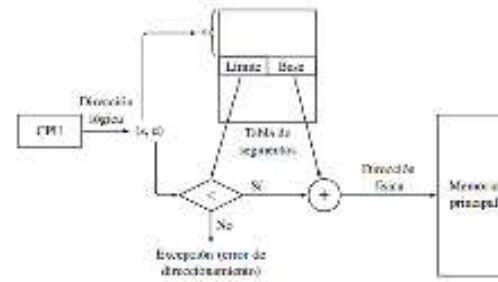
Un fallo de página supone leer una página y, si se expulsa una página modificada, escribir otra página en disco. Hay dos políticas de lectura y dos de escritura:

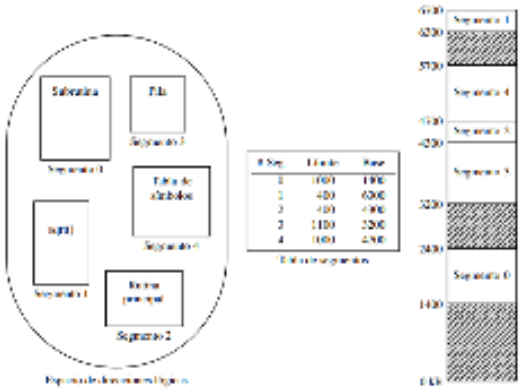
- **Paginación por demanda**: en un fallo de página, solo se lee la página que lo produce.
- **Prepaginación o paginación anticipada**: en un fallo de página, se leen la página que lo produce y varias páginas más.
- **Escritura por demanda**: una página se escribe en disco cuando se expulsa. Incrementa el tiempo de resolver muchos fallos de página, pues supone dos operaciones, una lectura y una escritura.
- **Escritura anticipada**: existe un hilo del núcleo llamado ***demonio de paginación*** que cada X segundos se despierta y escribe en disco las páginas modificadas. Las ventajas de esta política son que se escriben varias páginas a la vez, y que muchos fallos de página se resuelven en menos tiempo al no tener que esperar una posible escritura de la página expulsada.
  Un problema que puede surgir con esta política es que muchas escrituras pueden ser inútiles si las páginas que se guardan en disco se modifican de nuevo poco después. Para evitar este problema, el demonio de paginación puede buscar unas cuantas páginas que podrían ser expulsadas próximamente siguiendo el algoritmo de reemplazo usado por el sistema y, de esas páginas, escribir en disco todas las que estén modificadas.
  Además, el demonio de paginación también puede liberar las páginas que ha revisado eliminándolas de las tablas de páginas de sus correspondientes procesos. De este modo, tendremos un conjunto de marcos libres que se podrán usar para solucionar rápidamente cualquier fallo de página que se produzca. A esta técnica se le conoce como caché de páginas.
  El contenido de las páginas que se liberan no tiene por qué eliminarse de memoria principal. Solo cuando el marco que ocupa una de estas páginas se necesite, su contenido se podrá descartar. De esta manera, una página liberada que se necesita podría encontrar su contenido todavía en memoria, por lo que se podría recuperar de ahí sin necesidad de leerla de disco.

SEGMENTACIÓN

La **segmentación** trata de aproximarse a la perspectiva que tiene el usuario de la memoria. El programador o usuario prefiere pensar en ella como un conjunto de segmentos de tamaño variable sin ningún orden especial. Los elementos dentro de un segmento se identifican por su desplazamiento a partir del inicio del segmento.

En la segmentación, el espacio de direcciones lógicas de un proceso se compone de un conjunto de segmentos, cada uno de los cuales tiene una posición en memoria principal y un tamaño. Cada segmento es una sucesión lineal de direcciones, *desde 0 hasta su tamaño - 1*, que puede crecer o disminuir independientemente del resto y puede tener su propia protección.





Las direcciones generadas por los procesos especifican el *número de segmento* y el *desplazamiento* dentro de él. Para hacer corresponder las direcciones bidimensionales (segmento, desplazamiento) con las direcciones físicas unidimensionales se utiliza una tabla de segmentos. Cada entrada de la tabla de segmentos corresponde a un segmento y contiene una base y un límite. El desplazamiento se compara con el tamaño del segmento. Si es menor, se trata de una referencia válida, en cuyo caso se suma la base al desplazamiento para obtener la dirección física final. Si es mayor o igual, se produce un error de direccionamiento.

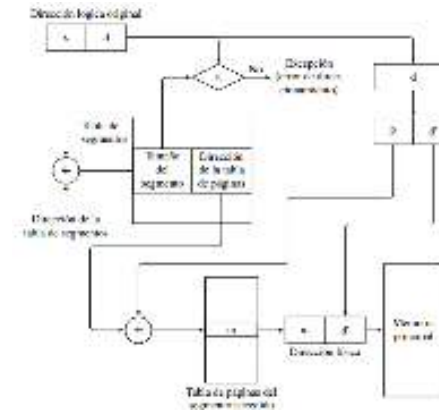
Cada proceso tendrá su tabla de segmentos. Habrá un registro hardware que apuntará al inicio de la tabla de segmentos del proceso que se está ejecutando. Asimismo, se puede utilizar un TLB para acelerar la traducción de direcciones.

La segmentación facilita también la protección y compartición de información al proteger y compartir segmentos enteros y no varias páginas. En la segmentación hay muchos espacios lineales de direcciones, uno por segmento, y el espacio total de direcciones puede exceder el tamaño de la memoria principal, ya que no todos los segmentos de un proceso tienen por qué estar a la vez en memoria. Sin embargo, ningún segmento puede superar el tamaño de la memoria física y los intercambios con el disco siempre se hacen moviendo segmentos enteros.

La segmentación está estrechamente relacionada con los modelos de administración de memoria de multiprogramación con particiones variables y, como en ellos, un problema importante es el de la fragmentación externa.

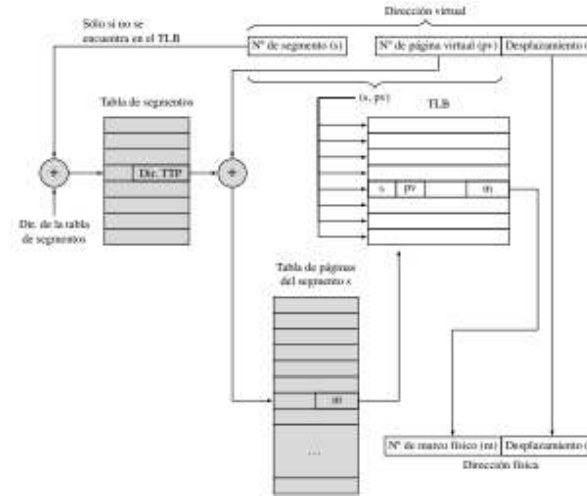
SEGMENTACIÓN PAGINADA

En la **segmentación paginada** se paginan los segmentos, es decir, los segmentos se dividen en un número determinado de páginas. Esto hace que desaparezca la fragmentación externa, aunque no la interna, y que la asignación de memoria a los segmentos sea algo trivial.



Al igual que en la segmentación pura, lo primero que se hace es comprobar si el desplazamiento dado en la dirección lógica es menor que el tamaño del segmento a usar. Si no es así, se producirá una excepción. En caso contrario, se tratará de una referencia válida a memoria y se continuará con la traducción. Ahora se toma el desplazamiento como una dirección virtual que hay que traducir mediante la tabla de páginas asociada al segmento. Para ello, el desplazamiento dentro del segmento se divide en un número de página y en un desplazamiento dentro de esa página, y la traducción se termina como ya hemos visto en paginación. Cada segmento tiene un tamaño

distinto, el tamaño de la tabla de páginas de cada uno, en número de entradas, también cambia, ya que depende del tamaño del segmento.



Podemos utilizar un TLB para acelerar la traducción. El TLB es direccionable por el número del segmento a usar y por el número de la página dentro del segmento al que se va a acceder. Si se produce un acierto de TLB, este nos devuelve el número del marco en el que se encuentra la página y se termina. Si se produce un fallo, a través de la tabla de segmentos del proceso accedemos a la tabla de páginas del segmento indicado en la dirección lógica y obtenemos de ella el marco en el que se encuentra la página. Esta información se copia en el TLB y se reinicia la instrucción.