ARQUITECTURA Y SISTEMAS OPERATIVOS

CLASE 7

MEMORIA PRINCIPAL

BIBLIOGRAFIA

- Operating System Concepts. By Abraham, Silberschatz.
  - Capítulo IX

TEMAS DE LA CLASE

- Introducción
  - Hardware básico
  - Asociación de direcciones
  - Espacio de direcciones Lógicas vs Físicas
  - Carga dinámica
  - Asociación dinámica
  - Librerías compartidas
- Asignación de memoria contigua
  - Protección de la memoria
  - Asignación de la memoria
  - Fragmentación

TEMAS DE LA CLASE

- Paginado
  - Método básico
  - Soporte del hardware
  - Protección
  - Páginas compartidas
- Estructura de una Tabla de Páginas
  - Paginado Jerárquico
  - Tablas de Páginas Hash
  - Tabla de Páginas Invertida
  - Tabla de Páginas Multinivel
- Swapping
  - Estándar
  - Con Paginado

INTRODUCCION

INTRODUCCION

```
int arr[] = {5, 3, 8, 6, 2};

Index:    0     1     2     3     4
        +-----+-----+-----+-----+-----+
Value:  |  5  |  3  |  8  |  6  |  2  |
        +-----+-----+-----+-----+-----+
```

Hasta ahora vimos cómo la CPU puede ser compartida por un conjunto de procesos, y cómo al planificar su uso, se puede optimizar su utilización y mejorar la velocidad de respuesta. Para lograr este aumento de rendimiento, debemos mantener muchos procesos en memoria; es decir, es necesario compartir la memoria.

La memoria consiste en una gran arreglo de bytes, cada uno con su propia dirección. La CPU obtiene instrucciones de la memoria según el valor del *program counter*. Y estas instrucciones en sí pueden provocar carga y almacenamiento adicionales en direcciones de memoria específicas.

Por ejemplo, en un ciclo de ejecución de instrucciones, primero obtiene una instrucción de la memoria, después se decodifica y puede provocar la obtención de valores nuevamente de la memoria. Una vez ejecutada la instrucción, los resultados pueden volver a almacenarse en memoria. Vale mencionar, que desde el punto de vista de la unidad de memoria, solo ve un *stream* de direcciones de memoria, solo interesa la secuencia de direcciones generada por el programa en ejecución, y no su contenido.

INTRODUCCION

- **Hardware Básico**

Cuando la CPU ejecuta instrucciones, necesita acceder a los datos correspondientes. Si estos no se encuentran en memoria, deben ser transferidos hacia la memoria principal para que la CPU pueda operar con ellos. Los registros dentro de cada núcleo de la CPU suelen ser accesibles en un solo ciclo de reloj. Sin embargo, acceder a la memoria principal puede requerir muchos ciclos, lo que retrasa significativamente la ejecución.

En estos casos, el procesador normalmente se bloquea, ya que no dispone de los datos necesarios para completar la instrucción actual. Esta situación no es viable debido a la frecuencia con la que se realizan accesos a memoria.

INTRODUCCION

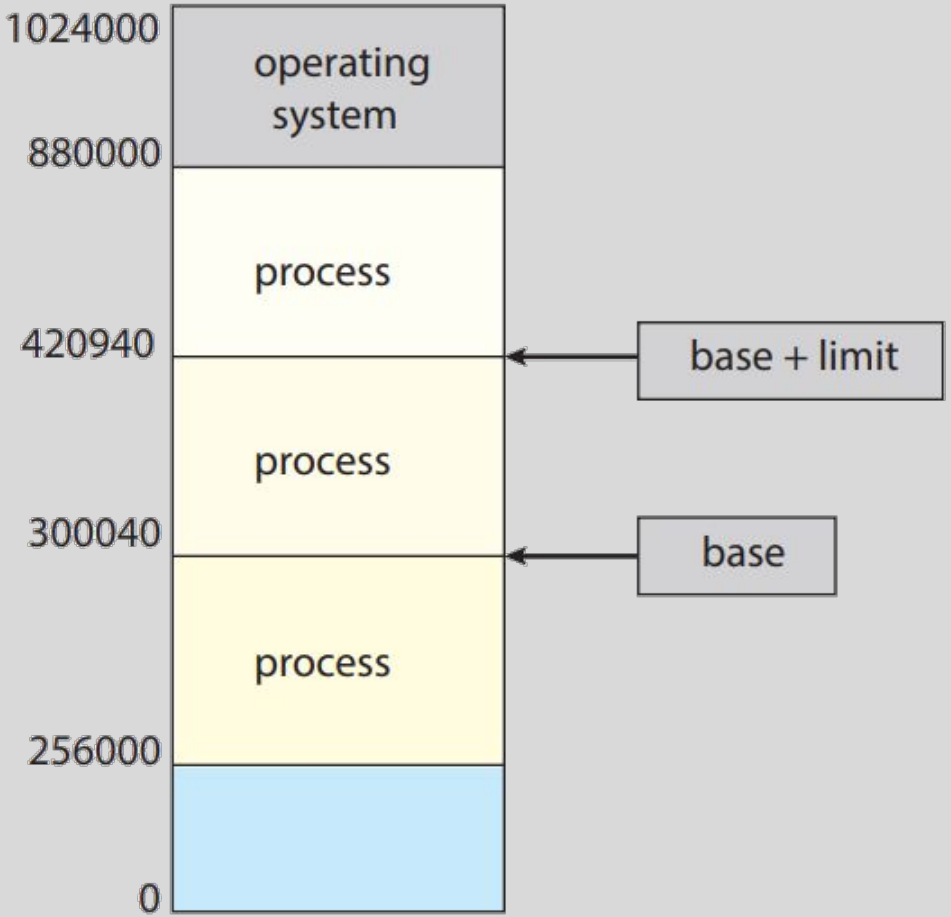
- **Hardware Básico**

Para resolver este problema, se introduce una memoria rápida intermedia entre la CPU y la memoria principal: la memoria caché. Esta caché está integrada en el propio chip de la CPU, es gestionada por hardware y permite acelerar automáticamente el acceso a la memoria, sin intervención del sistema operativo.

Pero además del rendimiento, otro aspecto fundamental del soporte de hardware es la protección de la memoria. El sistema debe evitar que un proceso acceda indebidamente a las áreas de memoria del sistema operativo o de otros procesos. Dado que el sistema operativo no interviene directamente en cada acceso de la CPU a memoria, esta protección debe ser proporcionada por el propio hardware.

EJEMPLO

Contar con un espacio de memoria independiente para cada proceso, los protege entre sí y es fundamental para la concurrencia. Para esto, es necesario poder determinar el rango de direcciones legales a las que un proceso puede acceder y asegurar que solo pueda acceder a estas direcciones. Esta protección puede lograrse mediante el uso de dos registros, uno base y uno límite. El registro base contiene la dirección de memoria legal más pequeña, y el registro límite especifica el tamaño del rango.

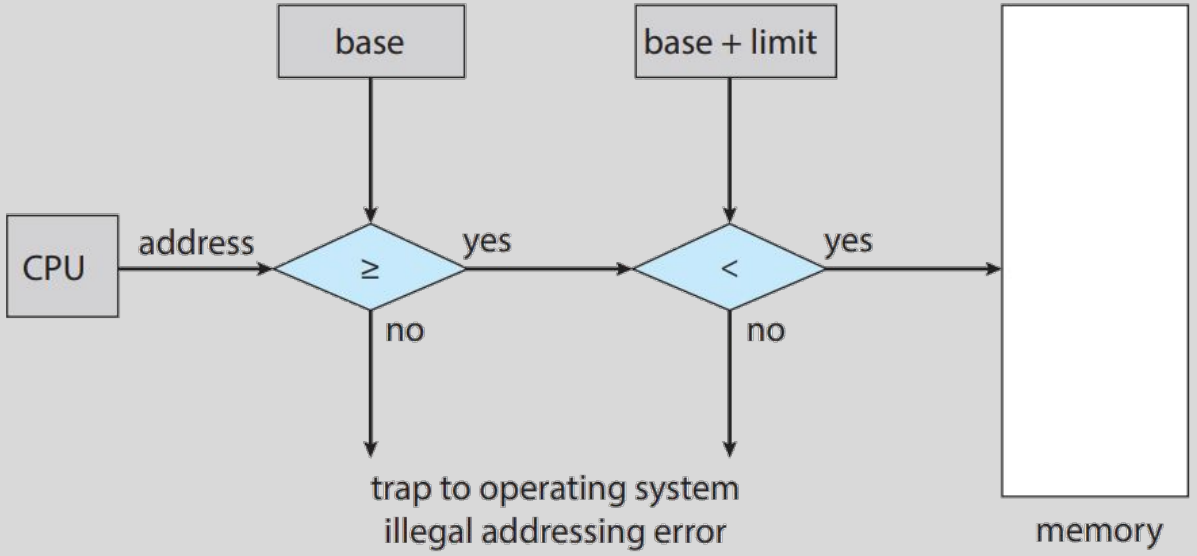


A base and a limit register define a logical address space.

EJEMPLO

De esta manera, el hardware de la CPU compara cada dirección generada con estos registros. Cualquier intento de un programa de usuario de acceder a la memoria del sistema operativo o a la de otros procesos de usuario, genera una interrupción *trap*, que se trata como un error fatal.

Estos registros sólo son accedidos por el sistema operativo a través de una instrucción privilegiada, ejecutada en modo *kernel*. Este esquema permite al sistema operativo cambiar el valor de los registros, pero impide que los programas de usuario modifiquen su contenido.



Hardware address protection with base and limit registers.

INTRODUCCION

- **Asociación de direcciones**

En la mayoría de los casos, un programa de usuario pasa por varios pasos antes de ejecutarse, algunos pueden ser opcionales. Y las direcciones de memoria pueden representarse de diferentes maneras durante estos pasos.

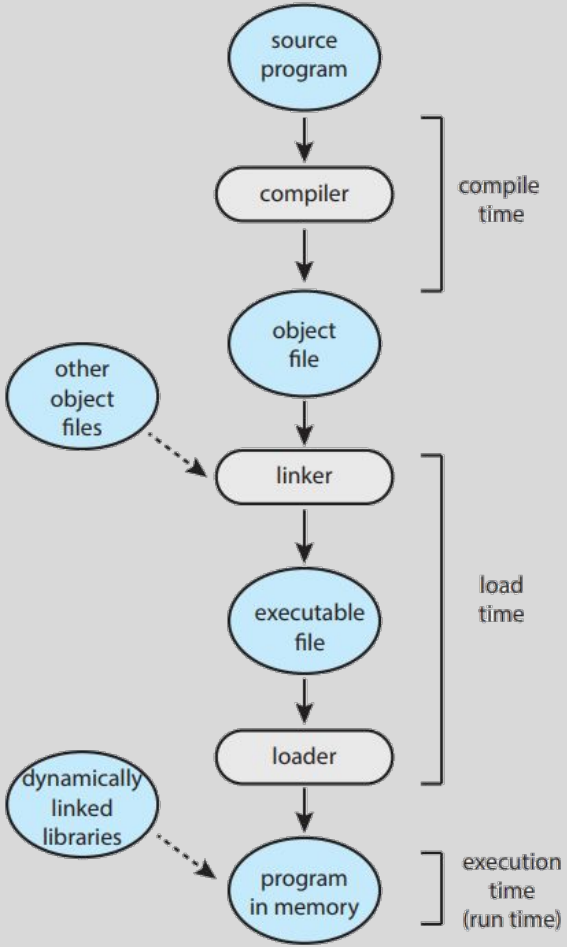
En el programa fuente, las direcciones suelen ser simbólicas (como la variable "`count`"). Un compilador suele vincular estas direcciones simbólicas a direcciones reubicables (algo como "14 bytes desde el principio de este módulo"). Y el *linker* o cargador, vincula las direcciones reubicables a direcciones absolutas (como 74014).

Cada vinculación es una asignación de un espacio de direcciones a otro.

La vinculación de instrucciones y datos a direcciones de memoria puede realizarse en:

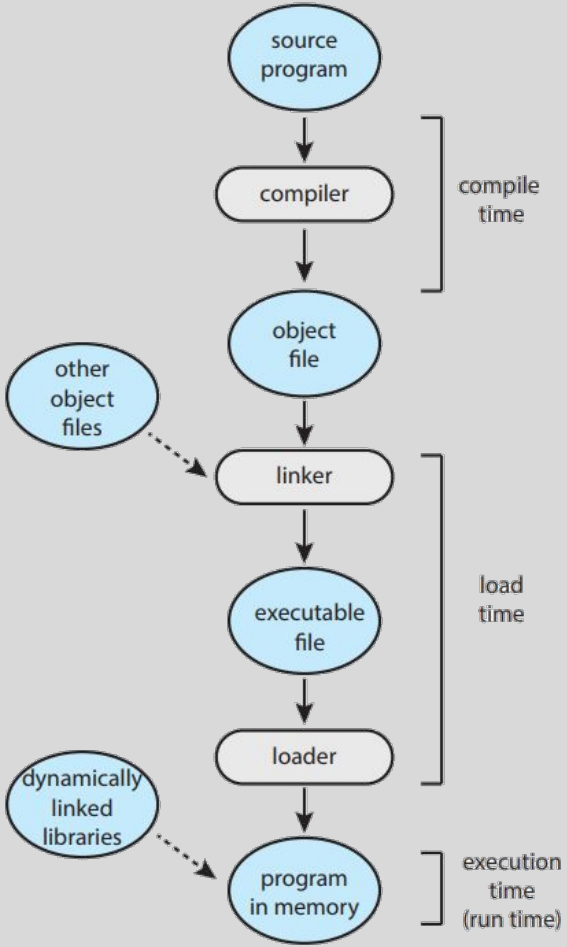
Tiempo de compilación. Si se conoce en tiempo de compilación dónde residirá el proceso en la memoria, se puede generar código absoluto. Si posteriormente cambia la ubicación de inicio, será necesario recompilar el código.

Tiempo de carga. Si en tiempo de compilación no se conoce dónde residirá el proceso, el compilador debe generar código reubicable. En este caso, la vinculación final se retrasa hasta el momento de carga. Si cambia la dirección de inicio, solo es necesario recargar el código.



Multistep processing of a user program.

Tiempo de ejecución. Si el proceso puede moverse de un segmento de memoria a otro durante su ejecución, la vinculación debe retrasarse hasta el momento de ejecución. Para que esto funcione, se requiere hardware especial (TLB), que se comenta más adelante. La mayoría de los sistemas operativos utilizan este método.



Multistep processing of a user program.

INTRODUCCION

- **Espacio de direcciones Lógicas vs Físicas**

Cuando la CPU genera una dirección, esta se denomina dirección lógica. Por otro lado, la dirección física es la que realmente utiliza la unidad de memoria, es decir, la que se carga en el registro de direcciones de memoria. El conjunto de todas las direcciones lógicas generadas por un programa forma el espacio de direcciones lógicas, mientras que el conjunto de direcciones físicas correspondientes constituye el espacio de direcciones físicas. Y dependiendo del momento en que se realice la vinculación de direcciones, estas pueden coincidir o no:

- Si la vinculación se realiza en tiempo de compilación o en tiempo de carga, las direcciones lógicas y físicas suelen ser idénticas.
- Si la vinculación ocurre en tiempo de ejecución, las direcciones lógicas y físicas son distintas, ya que las direcciones lógicas deben ser traducidas a físicas durante la ejecución del programa.

INTRODUCCION

- **Espacio de direcciones Lógicas vs Físicas**

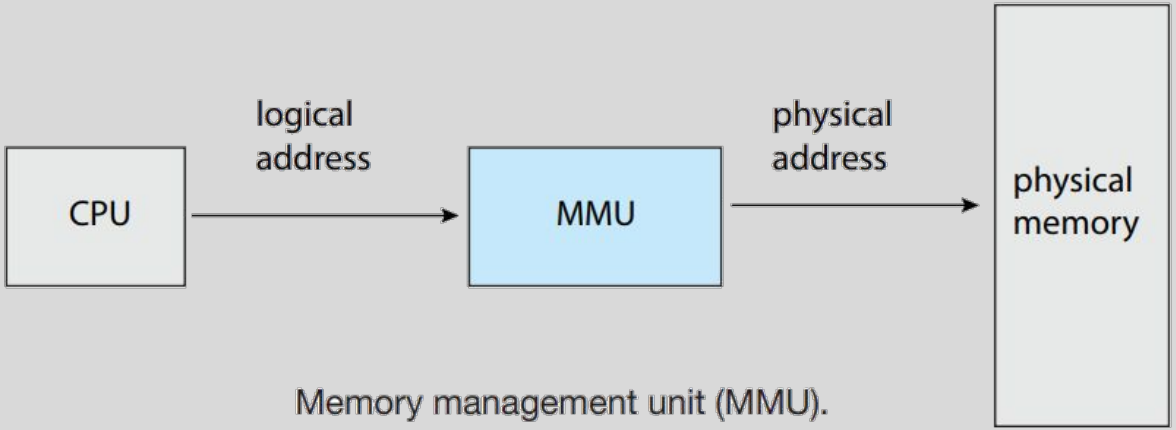
En términos prácticos, un programa de usuario opera como si se ejecutara en un espacio de memoria que va de 0 a un valor máximo (0 a máx). Sin embargo, en la realidad, estas direcciones lógicas deben mapearse a un rango de direcciones físicas, por ejemplo de R a R + máx, donde R es una dirección base definida por el sistema.

Este proceso de traducción de direcciones es fundamental para la gestión de memoria y permite implementar mecanismos de protección, asignación flexible y uso eficiente de la memoria física.

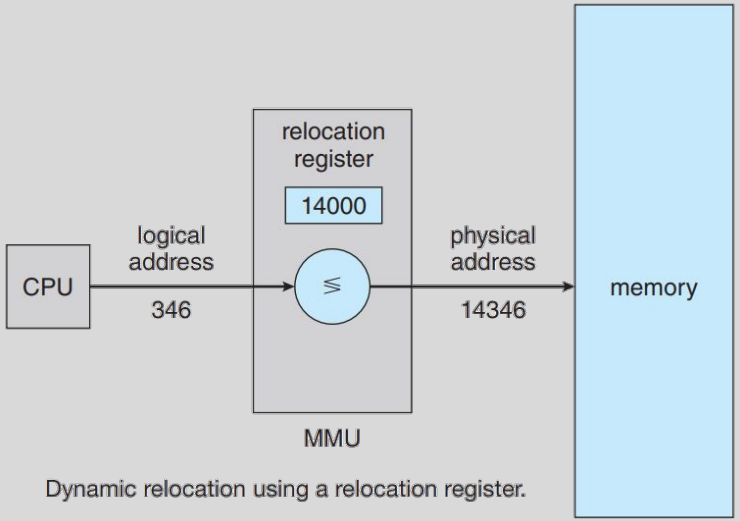
Traducción de Direcciones: Rol de la MMU.

La conversión de direcciones lógicas a direcciones físicas durante la ejecución se lleva a cabo mediante un componente de hardware llamado Unidad de Gestión de Memoria o MMU (Memory Management Unit).

En este proceso, el registro base —también conocido como registro de reubicación— contiene un valor que se suma a cada dirección lógica generada por un proceso de usuario en el momento en que se accede a memoria. De esta forma, la dirección lógica se traduce dinámicamente en una dirección física válida.



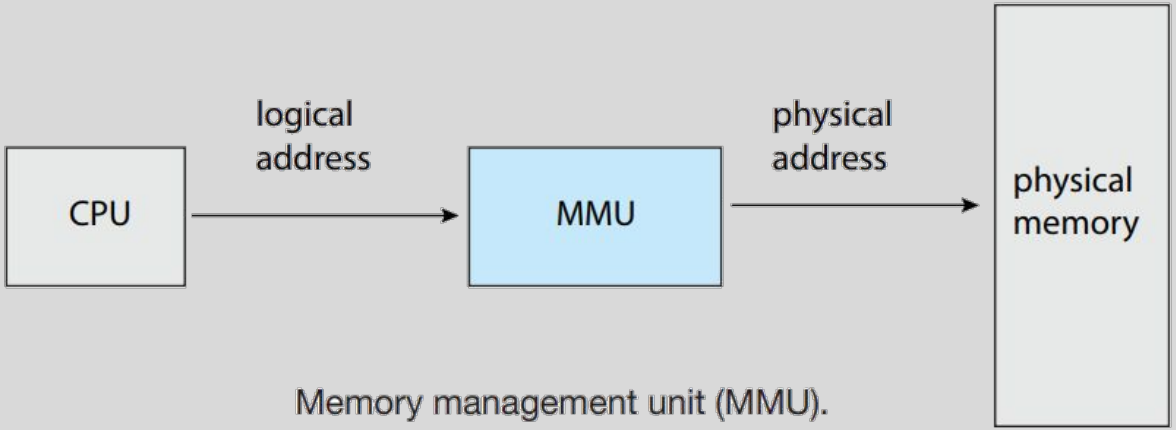
Memory management unit (MMU).



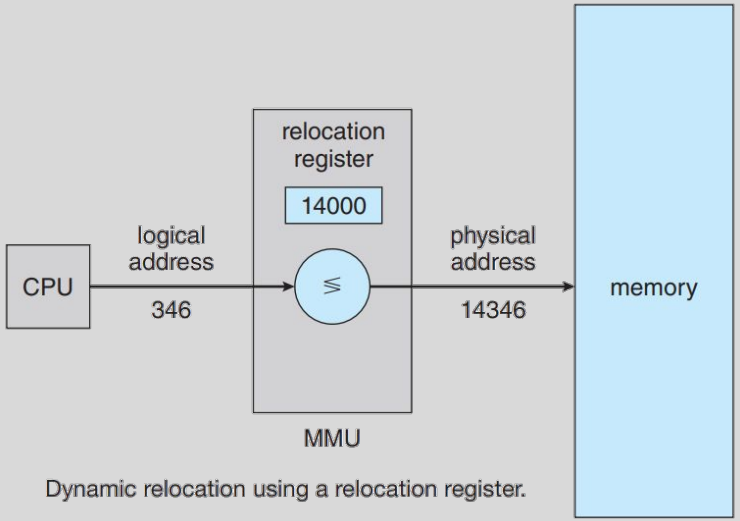
Dynamic relocation using a relocation register.

Es importante destacar que un programa de usuario nunca accede directamente a direcciones físicas. Todo el manejo de memoria desde la perspectiva del programa se realiza utilizando direcciones lógicas. La responsabilidad de traducir estas direcciones a sus equivalentes físicas recae completamente en el hardware encargado de la asignación de memoria.

En consecuencia, la ubicación física final de una dirección no se conoce hasta el momento en que se realiza el acceso a memoria, lo que permite al sistema una gran flexibilidad y seguridad en la gestión de los recursos de memoria.



Memory management unit (MMU).



Dynamic relocation using a relocation register.

INTRODUCCION

- **Carga dinámica**

La carga dinámica es una técnica en la cual una función no se carga en memoria hasta que es invocada por el programa. Es decir, cuando el programa principal se carga y comienza a ejecutarse, solo se cargan inicialmente las funciones esenciales. Si en algún momento se necesita ejecutar una función adicional, el programa primero verifica si esta ya está en memoria. Si no lo está, se procede a cargarla en ese momento.

Una vez cargada, se actualizan las tablas de direcciones del programa para reflejar la ubicación de la nueva función, y luego se transfiere el control a dicha rutina para su ejecución.

INTRODUCCION

- **Carga dinámica**

La principal ventaja de este enfoque es que las funciones solo se cargan cuando son realmente necesarias, lo que permite ahorrar memoria. Esto resulta especialmente útil en situaciones donde se requiere una gran cantidad de código para manejar casos poco frecuentes, como por ejemplo rutinas de manejo de errores.

La implementación de la carga dinámica requiere una planificación consciente por parte del programador. Aunque es el programador quien debe estructurar su código para aprovechar esta técnica, los sistemas operativos pueden colaborar ofreciendo librerías o funciones específicas que facilitan su uso.

INTRODUCCION

- **Asociación dinámica**

La asociación dinámica se refiere al uso de librerías compartidas, conocidas comúnmente como DLLs (Dynamic Link Libraries), que se vinculan a los programas en tiempo de ejecución. A diferencia de la vinculación estática, donde las funciones de una librería se integran en el ejecutable al compilar el programa, en la vinculación dinámica esta asociación se pospone hasta el momento de la ejecución.

Este mecanismo es similar a la carga dinámica, ya que ambos difieren de la carga estática tradicional. Sin embargo, mientras la carga dinámica suele ser controlada directamente por el programador, la asociación dinámica suele estar gestionada por el sistema operativo.

INTRODUCCION

- **Asociación dinámica**

Una de las principales ventajas de esta técnica es la eficiencia en el uso de memoria. Por ejemplo, sin la asociación dinámica, cada programa tendría que incluir una copia completa de librerías del sistema, como la librería estándar de C, dentro de su archivo ejecutable. Esto resultaría en ejecutables más grandes y en una mayor carga sobre la memoria principal.

Con la asociación dinámica, en cambio, varios programas pueden compartir una única copia de la librería cargada en memoria, lo que reduce el uso de recursos y mejora el rendimiento general del sistema.

INTRODUCCION

- **Librerías compartidas**

Una de las grandes ventajas de las librerías dinámicas (DLL) es que pueden ser compartidas entre varios procesos, lo que permite que haya una sola instancia de la DLL cargada en memoria, independientemente de cuántos programas la utilicen. Por esta razón, también se las conoce como librerías compartidas. Este enfoque es ampliamente utilizado en sistemas operativos como Windows y Linux.

Cuando un programa hace referencia a una función contenida en una DLL, el *loader* del sistema operativo se encarga de buscar la librería y cargarla en memoria si aún no está disponible. Una vez cargada, el *loader* ajusta las direcciones de las llamadas a funciones del programa, redirigiéndolas a la ubicación real de la DLL en memoria.

INTRODUCCION

- **Librerías compartidas**

A diferencia de la carga dinámica, que puede ser implementada por el programador, tanto la vinculación dinámica como el uso de librerías compartidas requieren soporte explícito del sistema operativo. Esto se debe a que los procesos están aislados y protegidos entre sí, por lo que solo el sistema operativo puede:

- Comprobar si una DLL ya está cargada en memoria.
- Permitir o impedir que varios procesos accedan de forma segura a la misma región de memoria compartida.

En breve se explicará con más detalle cómo se logra este tipo de compartición controlada de memoria entre procesos.

ASIGNACIÓN DE MEMORIA CONTIGUA

ASIGNACIÓN DE MEMORIA CONTIGUA

La memoria principal debe alojar tanto al sistema operativo como a los procesos de usuario. Por ello, se divide en dos particiones principales:

- Una para el sistema operativo.
- Otra para los procesos de usuario.

Como se busca que varios procesos de usuario residan en memoria al mismo tiempo, es fundamental determinar cómo asignar la memoria disponible a los procesos que están esperando ser cargados. Pero antes, veamos cómo se protege la memoria en este esquema.

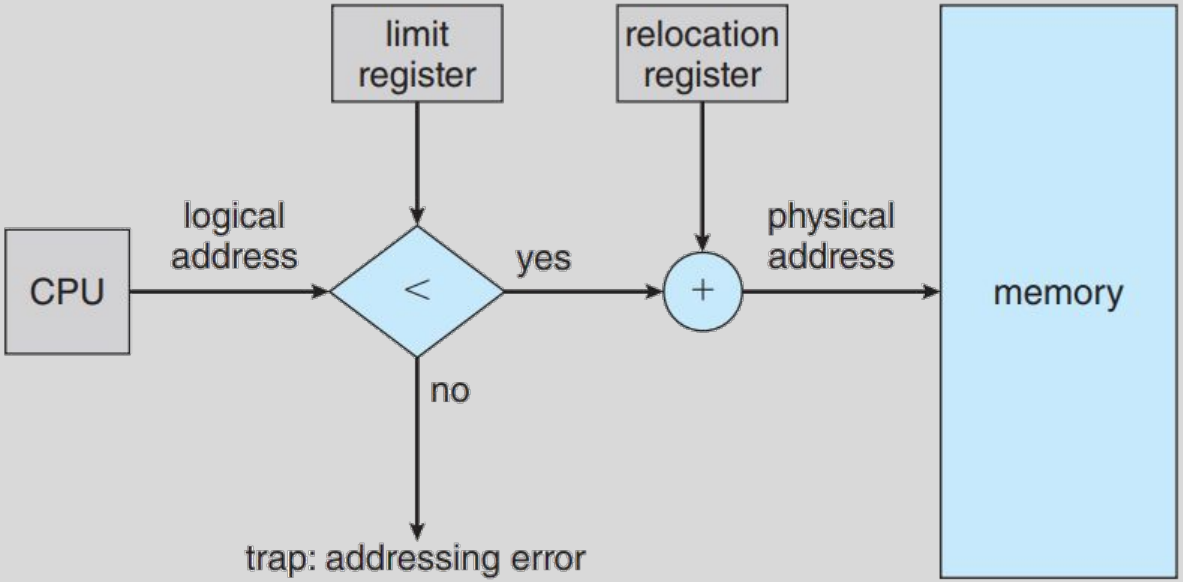
- **Protección de la Memoria**

En la asignación contigua, se utilizan dos registros importantes para proteger el acceso a la memoria:

- El registro de reubicación: contiene la dirección física más baja que puede usar el proceso.
- El registro de límite: define el rango válido de direcciones lógicas para ese proceso.

Cada dirección lógica generada por el proceso debe encontrarse dentro del rango permitido. La MMU (Unidad de Gestión de Memoria) verifica esto y, si es válido, suma el valor del registro de reubicación a la dirección lógica para obtener la dirección física, que luego se envía a memoria.

Cuando el planificador de la CPU selecciona un proceso para su ejecución, el *dispatcher* carga los registros de reubicación y límite correspondientes al proceso. Así, cada dirección generada por el proceso en ejecución se verifica automáticamente, lo que protege tanto al sistema operativo como a otros procesos de accesos indebidos.



Hardware support for relocation and limit registers.

ASIGNACIÓN DE MEMORIA CONTIGUA

- **Asignación de la memoria**

El sistema operativo mantiene una tabla de asignación de memoria, indicando qué regiones están ocupadas y cuáles están libres. Cuando un proceso ingresa al sistema, se evalúan:

- Sus requisitos de memoria.
- La memoria actualmente disponible.

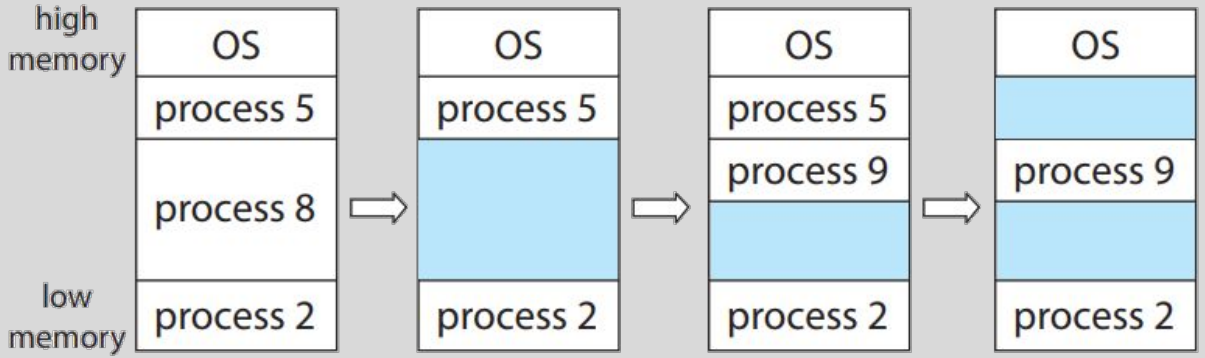
Si hay espacio suficiente, se le asigna memoria y el proceso se carga, quedando listo para competir por el uso de la CPU. Y cuando un proceso termina su ejecución, libera su memoria, que podrá ser reasignada a otros procesos. Pero, ¿qué sucede si no hay suficiente memoria para un nuevo proceso?

- Opción 1: Rechazar el proceso y mostrar un mensaje de error.
- Opción 2: Colocarlo en una cola de espera, para asignarle memoria cuando se libere espacio suficiente.

Ejemplo: supongamos que inicialmente están cargados en memoria los procesos 5, 8 y 2. Si el proceso 8 termina, queda un hueco contiguo. Si luego entra el proceso 9, puede ocupar ese espacio. Más tarde, si el proceso 5 finaliza, aparecen huecos no contiguos.

Este caso refleja un problema general en la asignación dinámica de almacenamiento, que consiste en satisfacer una solicitud de memoria de tamaño n a partir de una lista de huecos libres. Hay tres estrategias comunes:

- *First-fit*: asignar el primer hueco lo suficientemente grande.
- *Best-fit*: asignar el hueco más pequeño que sea suficiente.
- *Worst-fit*: asignar el hueco más grande disponible.



Variable partition.

ASIGNACIÓN DE MEMORIA CONTIGUA

- **Fragmentación**

Este tipo de esquema produce dos formas de fragmentación:

Fragmentación externa: ocurre cuando, tras múltiples cargas y descargas de procesos, la memoria se divide en pequeños fragmentos dispersos. Puede haber suficiente memoria total, pero no contigua, para satisfacer una solicitud.

Fragmentación interna: se da cuando se asigna un bloque mayor al necesario. Por ejemplo, si hay un hueco de 12.464 bytes y un proceso solicita 12.462 bytes, quedan 2 bytes libres. Administrar este pequeño espacio puede costar más que el valor mismo del hueco.

Posible Solución: una forma de reducir la fragmentación externa es permitir que el espacio de direcciones lógicas de un proceso no sea contiguo, es decir, que se pueda asignar memoria física de manera dispersa. Esto se explora en esquemas más avanzados de gestión de memoria, como la paginación, que se tratará en breve.

PAGINADO

PAGINADO

El paginado es un esquema de gestión de memoria que permite que el espacio de direcciones físicas de un proceso no sea contiguo. Gracias a esta característica:

- Evita la fragmentación externa.
- Elimina la necesidad de compactación.

Debido a sus ventajas, la paginación —en sus diversas variantes— se implementa en la mayoría de los sistemas operativos, desde servidores hasta dispositivos móviles. Y su implementación requiere la cooperación entre el sistema operativo y el hardware.

- **Método básico**

El paginado se basa en dividir:

- La memoria física en bloques de tamaño fijo llamados marcos (*frames*).
- La memoria lógica de cada proceso en bloques del mismo tamaño llamados páginas (*pages*).

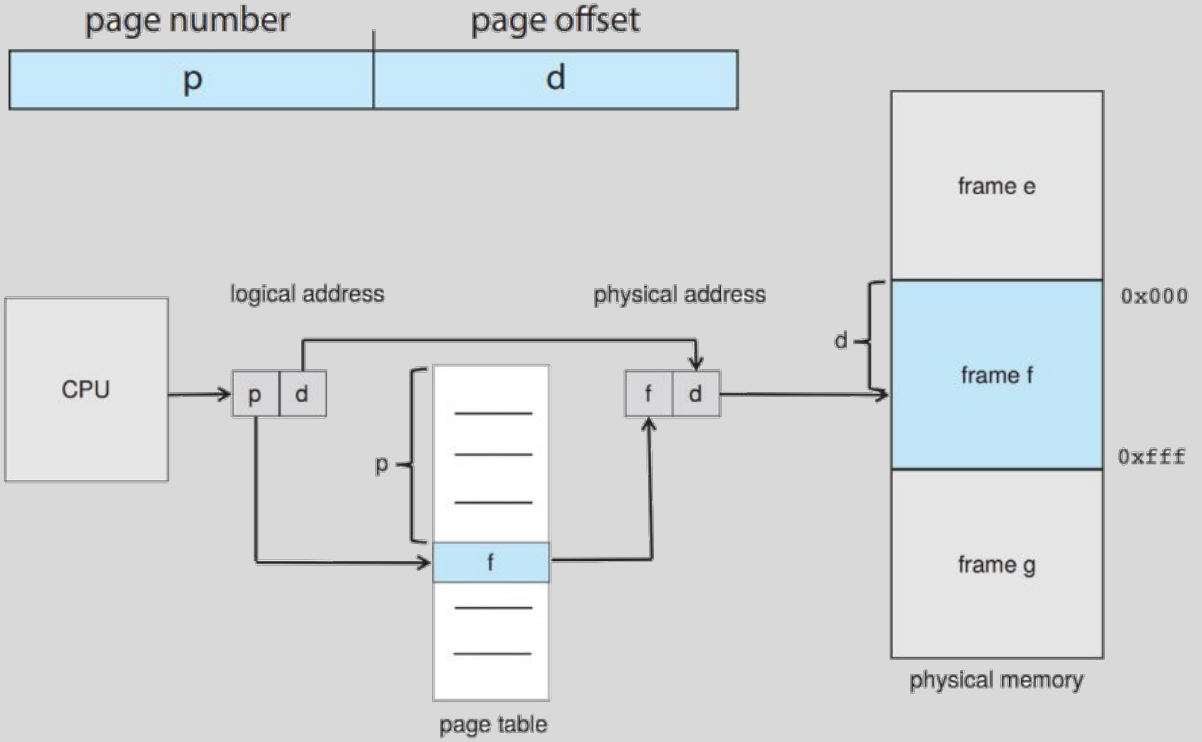
Al ejecutar un proceso, sus páginas pueden cargarse en cualquier marco disponible, sin necesidad de que sean espacios contiguos.

Cada dirección lógica generada por la CPU se divide en dos partes:

- Número de página (p): se usa como índice en la tabla de páginas del proceso.
- Desplazamiento (d): indica la posición dentro de la página.

La tabla de páginas contiene la dirección base de cada marco físico donde reside cada página.

Para todo el sistema, el sistema operativo mantiene también una tabla de marcos, que indica si un marco está libre o asignado, y en tal caso, a qué proceso y página pertenece.

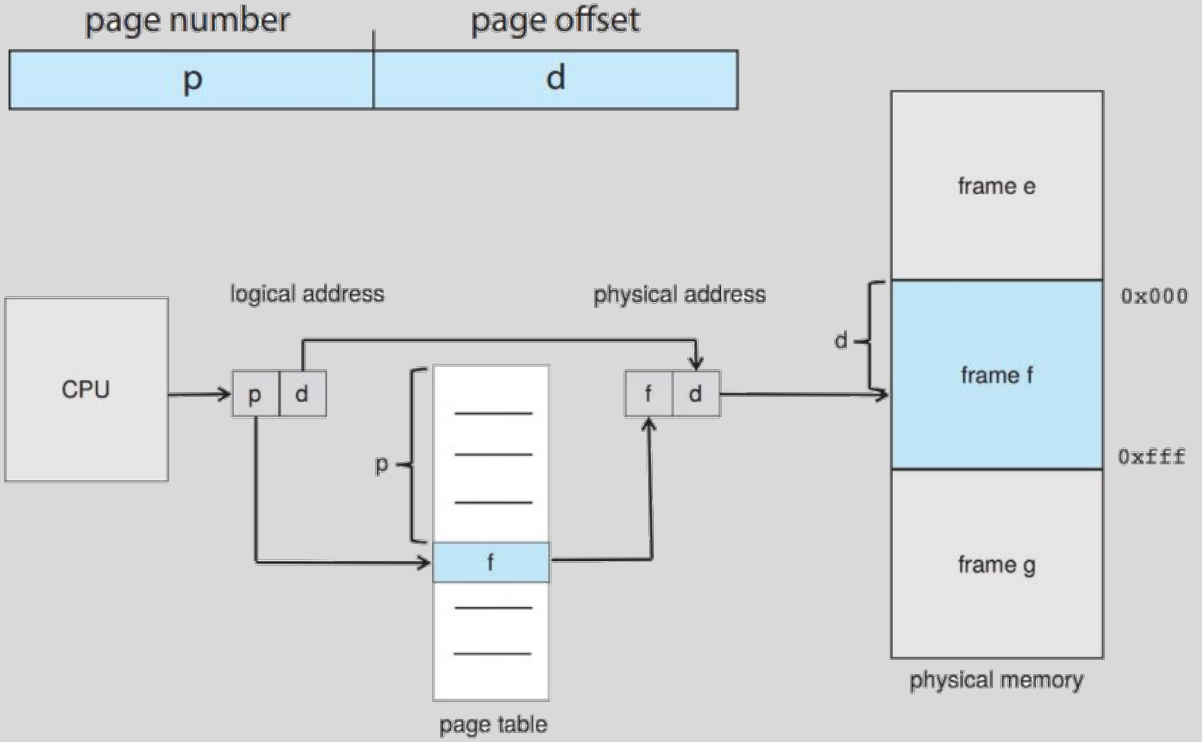


Paging hardware.

Traducción de una dirección lógica a física (por la MMU):

1. Se extrae el número de página `p`.
2. Se accede a la tabla de páginas del proceso y se obtiene el número de marco `f`.
3. Se combina el marco `f` con el desplazamiento `d` para formar la dirección física final.

Nota: El tamaño de página y de marco es fijo y está definido por el hardware.

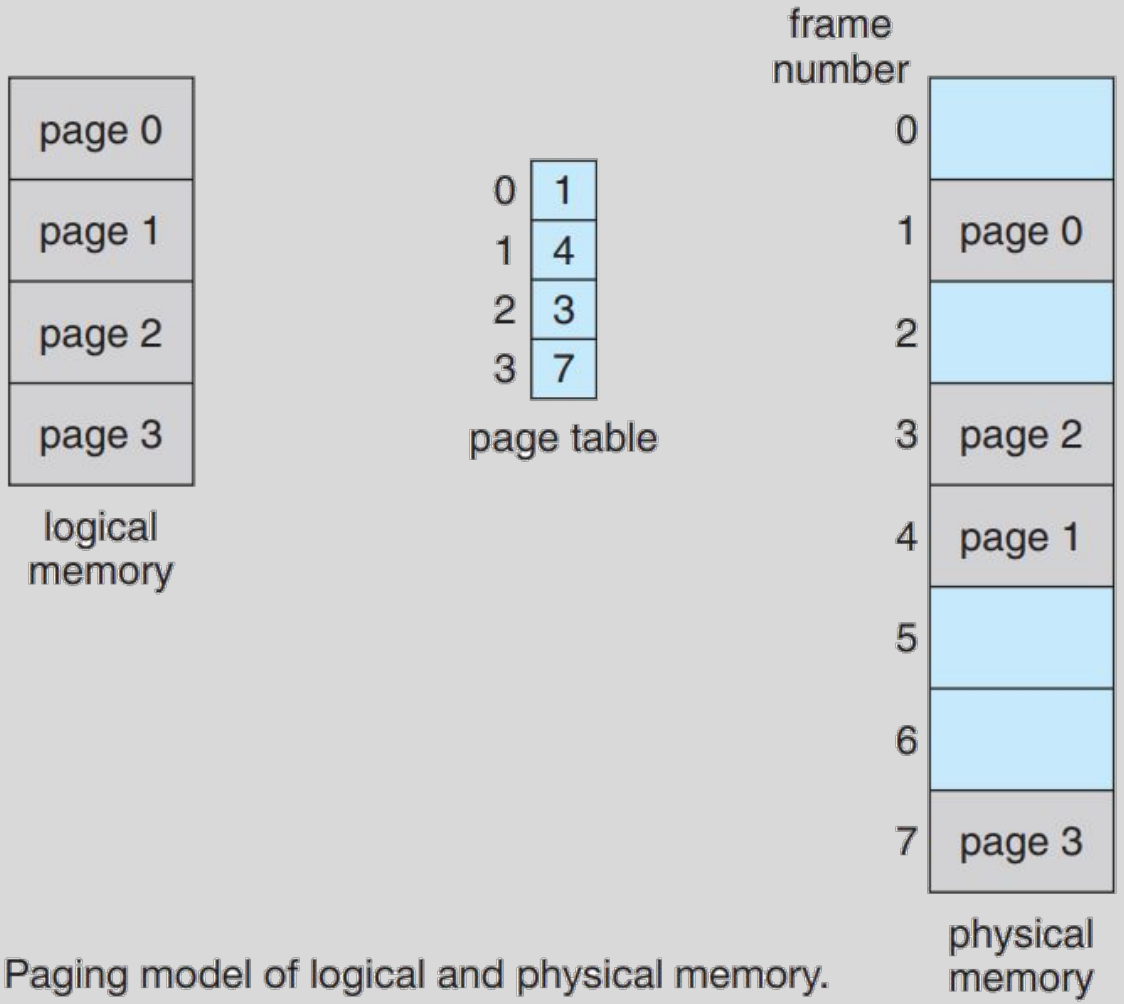


Paging hardware.

Traducción de una dirección lógica a física (por la MMU):

1. Se extrae el número de página `p`.
2. Se accede a la tabla de páginas del proceso y se obtiene el número de marco `f`.
3. Se combina el marco `f` con el desplazamiento `d` para formar la dirección física final.

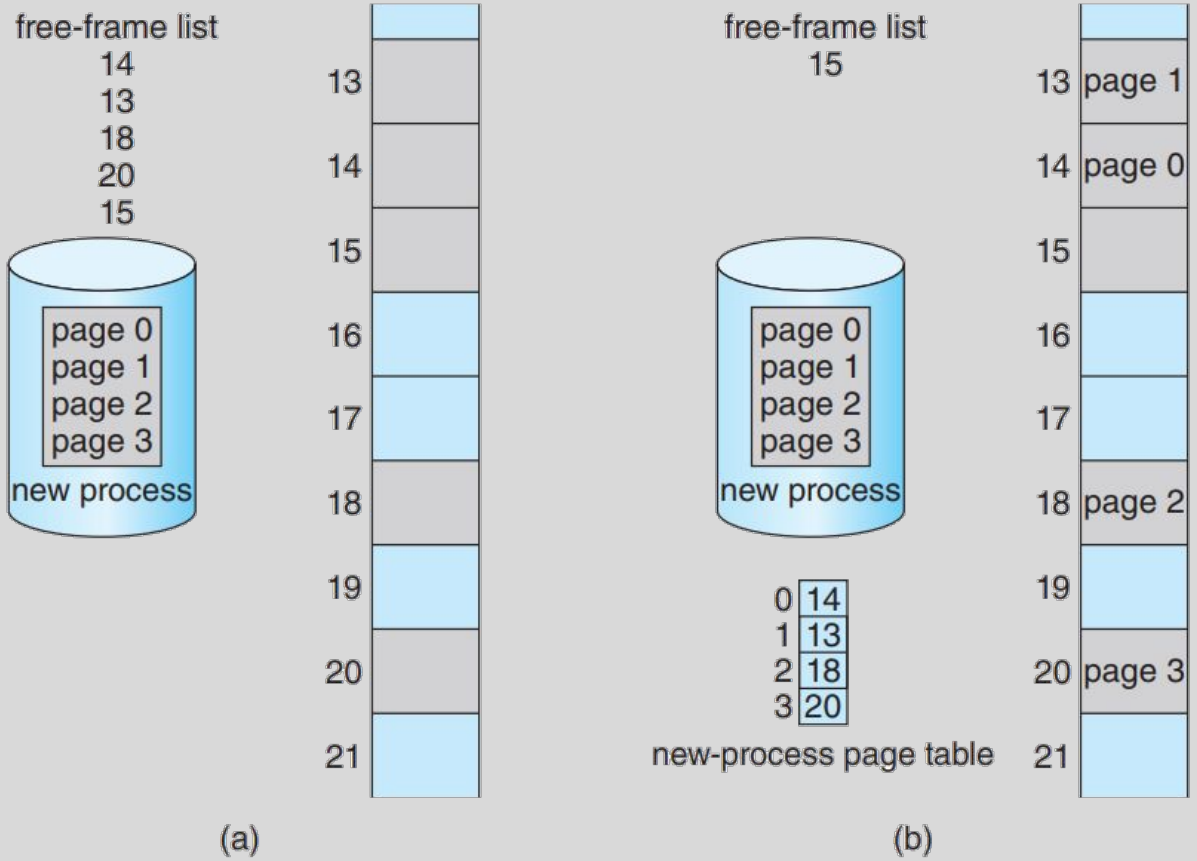
Nota: El tamaño de página y de marco es fijo y está definido por el hardware.



Paging model of logical and physical memory.

Ejemplo

Supongamos que un proceso necesita n páginas. Para cargarse en memoria, debe haber al menos n marcos libres. Cada página se carga en un marco distinto, y el número del marco se guarda en la tabla de páginas del proceso. Así, se pueden cargar las páginas en marcos no contiguos sin afectar la ejecución.



Free frames (a) before allocation and (b) after allocation.

PAGINADO

- **Soporte del Hardware**

Cada proceso posee su propia tabla de páginas, por lo que es necesario mantener una referencia a esta estructura. Esta referencia puede guardarse en un registro especial del PCB (bloque de control del proceso). Opciones de implementación:

- Registros rápidos de hardware dedicados: Permiten traducción muy eficiente, pero implican mayor sobrecarga al cambiar de contexto, ya que deben recargarse.
- Tabla de páginas en memoria principal + registro base: Solo se modifica el registro base al cambiar de contexto, reduciendo la sobrecarga. Sin embargo, cada acceso a memoria implica 2 accesos reales:
  - Uno a la tabla de páginas.
  - Otro al dato deseado.

Este retraso puede reducir a la mitad el rendimiento, algo inaceptable en la mayoría de los casos.

Solución: TLB (Translation Lookaside Buffer)

PAGINADO

- **Soporte del Hardware**

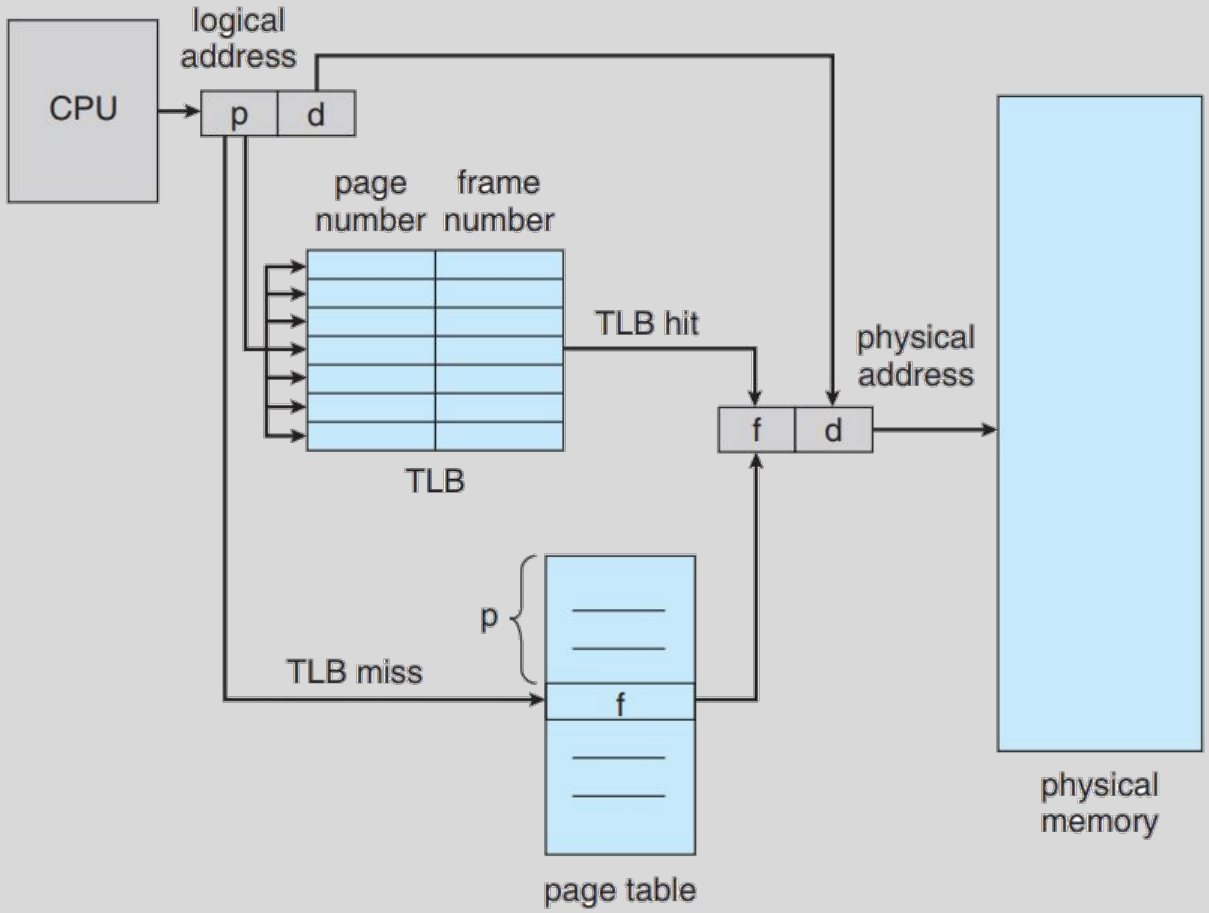
Para acelerar este proceso, se usa una caché de hardware especializada llamada TLB, que es:

- Pequeña y muy rápida.
- Memoria asociativa: compara todas sus claves simultáneamente.

El TLB contiene algunas entradas de la tabla de páginas. Cuando se genera una dirección lógica:

1. La MMU primero busca el número de página en el TLB.
2. Si lo encuentra, obtiene directamente el número de marco → acceso inmediato.
3. Si no está:
   a. Se consulta la tabla de páginas en memoria.
   b. Se accede al marco correspondiente.
   c. La entrada se agrega al TLB (reemplazando alguna si es necesario).

Aunque es una característica de hardware, el sistema operativo debe comprender su funcionamiento, ya que impacta directamente en el rendimiento.



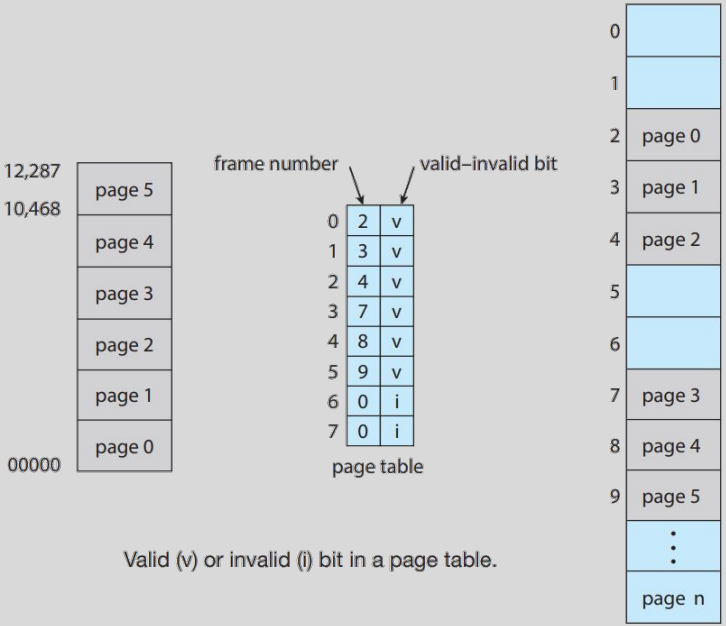
Paging hardware with TLB.

PAGINADO

- **Protección**

La paginación permite implementar mecanismos de protección por página, mediante bits de control asociados a cada entrada en la tabla de páginas:

- Bit de protección: indica si la página es de solo lectura o lectura-escritura. Se revisa en cada acceso para evitar modificaciones indebidas.
- Bit válido / inválido:
  - Válido: la página forma parte del espacio de direcciones del proceso.
  - Inválido: la página no pertenece al proceso, y cualquier intento de acceso genera una excepción.



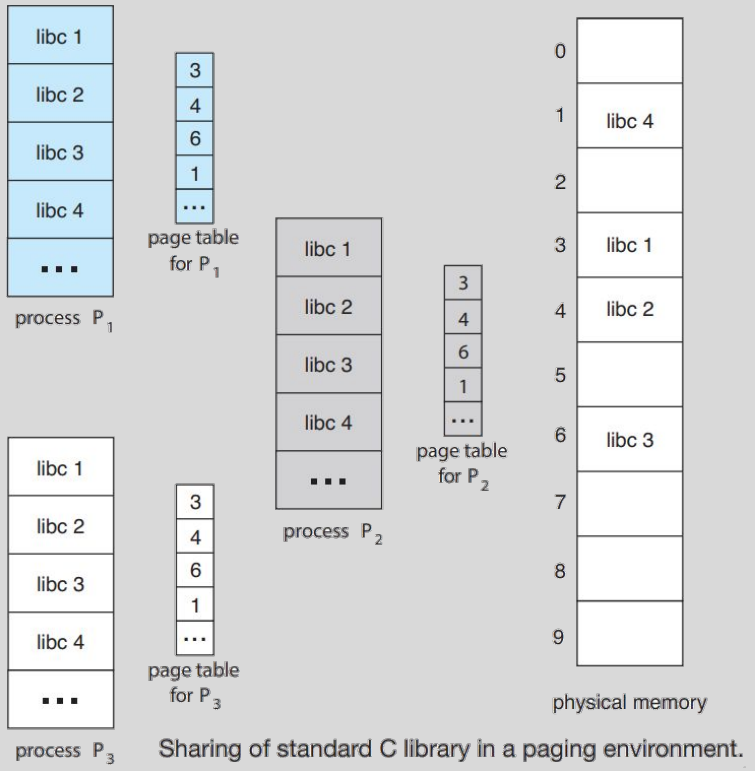
Valid (v) or invalid (i) bit in a page table.

PAGINADO

- **Páginas compartidas**

Una gran ventaja del paginado es la posibilidad de compartir código entre procesos, como por ejemplo, la librería estándar de C (`libc`).

Supongamos que hay 40 procesos en ejecución y cada uno necesita usar libc, que ocupa 2 MB. Si cada proceso carga su propia copia, se consumirán 80 MB de memoria.



Sharing of standard C library in a paging environment.

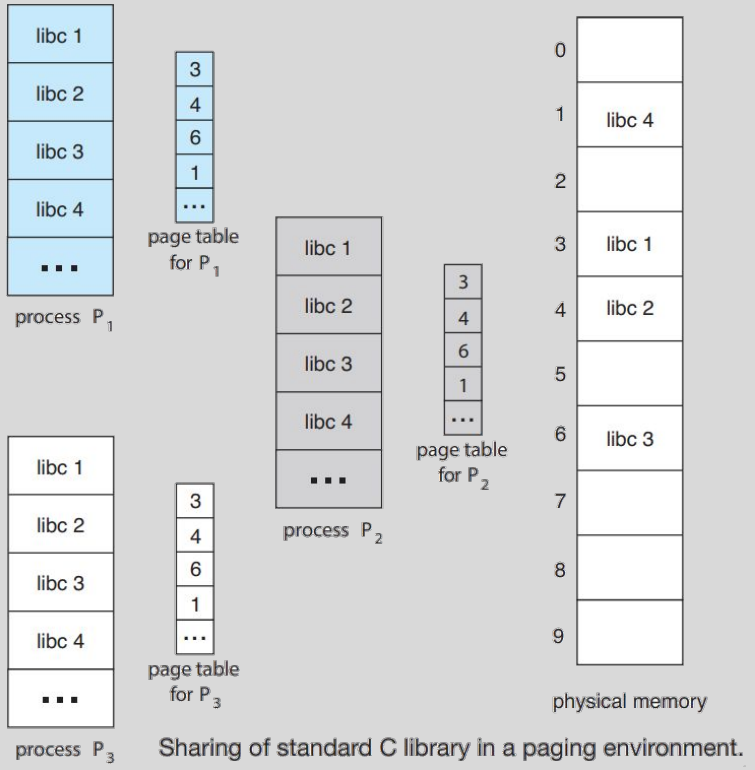
PAGINADO

- **Páginas compartidas**

Pero si el código de la librería es reentrante (no se modifica durante la ejecución), entonces puede ser compartido por todos los procesos:

- Se carga una sola vez en memoria.
- Todos los procesos que lo necesiten comparten las mismas páginas.

Resultado: Solo se utilizan 2 MB en lugar de 80 MB, lo que supone un ahorro significativo.



Sharing of standard C library in a paging environment.

ESTRUCTURA DE UNA TABLA DE PÁGINAS

ESTRUCTURA DE UNA TABLA DE PÁGINAS

Los sistemas operativos modernos deben manejar espacios de direcciones lógicas muy amplios, que van desde 2^{32} hasta 2^{64}. En estos entornos, una tabla de páginas tradicional puede volverse extremadamente grande. Por ejemplo, en un sistema con:

- Espacio de direcciones lógicas de 32 bits (2^{32} direcciones).
- Tamaño de página de 4 KB (2^{12} bytes por página).

Se requieren:

- 2^{20} entradas en la tabla de páginas (2^{32} / 2^{12}).
- Si cada entrada ocupa 4 bytes → la tabla total ocupa 4 MB por proceso solo para su tabla de páginas.

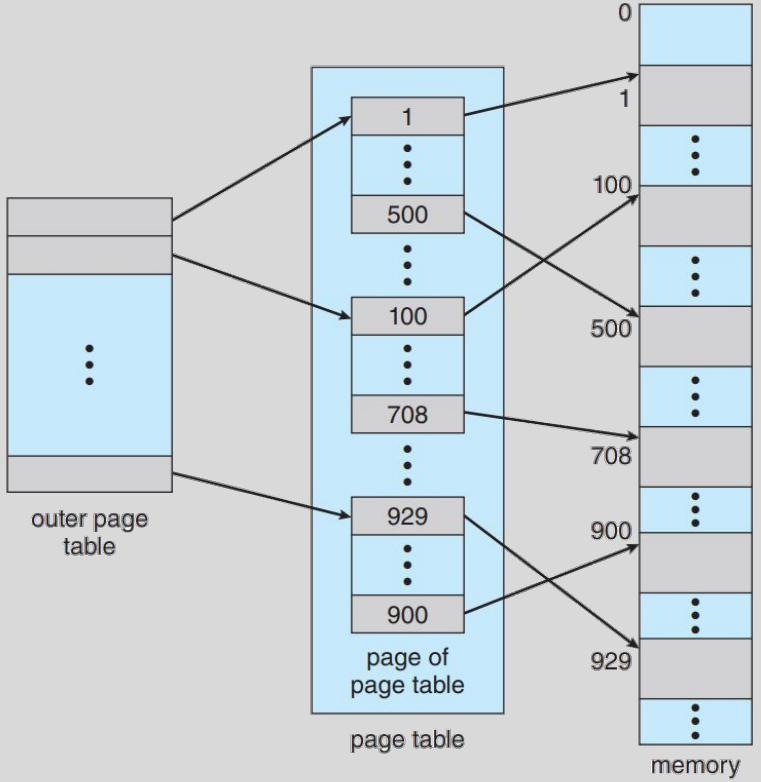
ESTRUCTURA DE UNA TABLA DE PÁGINAS

- **Paginado Jerárquico**

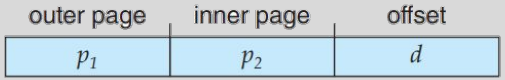
Una solución a este problema es dividir la tabla de páginas en estructuras más pequeñas, reduciendo el uso de memoria. Una estrategia común es el paginado en dos niveles, en el cual la tabla de páginas también se página.

Este esquema permite:

- Mantener solo las partes necesarias de la tabla en memoria.
- Evitar asignar una estructura completa cuando muchas páginas no se usan.

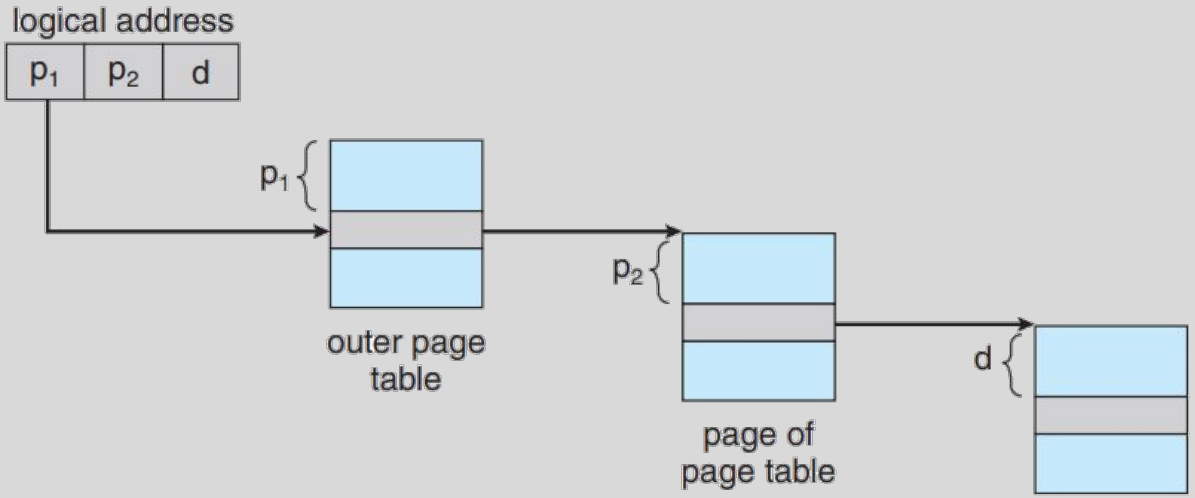


A two-level page-table scheme.



Así queda la traducción de direcciones para esta arquitectura.

Para sistemas con espacios de direcciones de 64 bits, esta técnica se vuelve insuficiente, y se requieren estructuras más avanzadas (como paginación multinivel con más niveles o técnicas alternativas).



Address translation for a two-level 32-bit paging architecture.

ESTRUCTURA DE UNA TABLA DE PÁGINAS

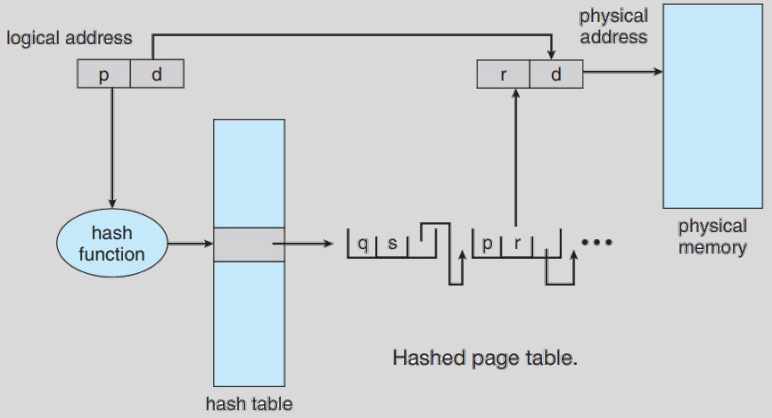
- **Tablas de Páginas Hash**

Una alternativa para gestionar grandes espacios de direcciones (más allá de 32 bits) es utilizar una tabla de páginas con hashing. En este enfoque:

- Se utiliza el número de página virtual como entrada a una función hash.
- El resultado de la función indica el índice en la tabla hash.

Cada entrada de la tabla hash contiene una lista enlazada (para resolver colisiones), y cada nodo en la lista contiene:

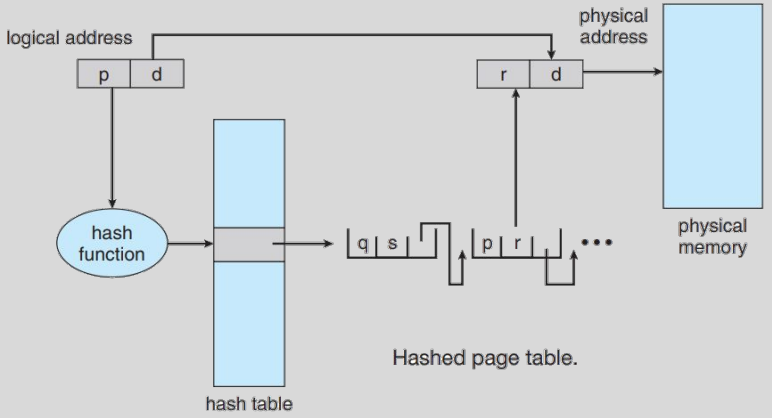
1. El número de página virtual (q).
2. El número de marco físico correspondiente (s).
3. Un puntero al siguiente elemento de la lista.



ESTRUCTURA DE UNA TABLA DE PÁGINAS

El algoritmo funciona de la siguiente manera:

- Se aplica la función hash al número de página virtual p → índice en la tabla.
- Se compara p con q (número de página almacenado).
- Si coinciden → se usa el marco s y se accede a memoria.
- Si no coinciden → se sigue recorriendo la lista hasta encontrar coincidencia o confirmar ausencia.



ESTRUCTURA DE UNA TABLA DE PÁGINAS

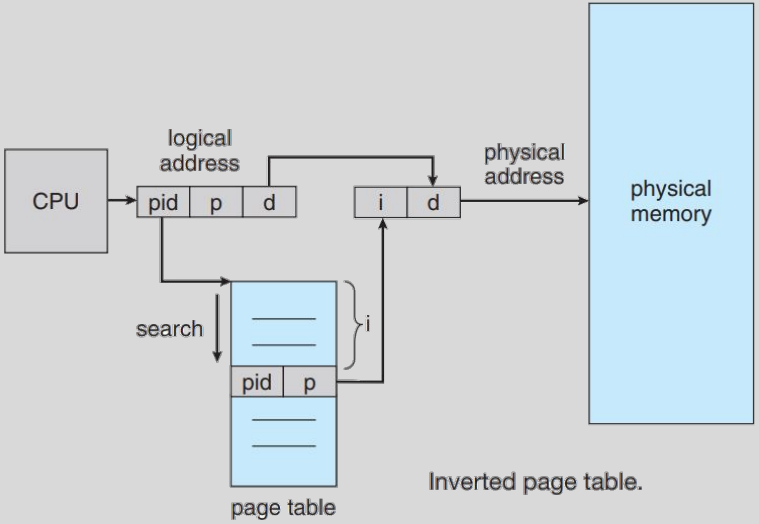
- **Tabla de Páginas Invertida**

En lugar de tener una tabla por proceso, se mantiene una tabla global para todo el sistema. Donde cada entrada representa un marco físico. Y para cada marco físico, almacena:

- Qué proceso lo posee
- Qué página virtual le asignó ese proceso

Por eso se llama invertida: en lugar de buscar desde una página hacia un marco, se parte desde el marco para conocer su contenido.

`[ frame #i ] → [ pid, p ]`



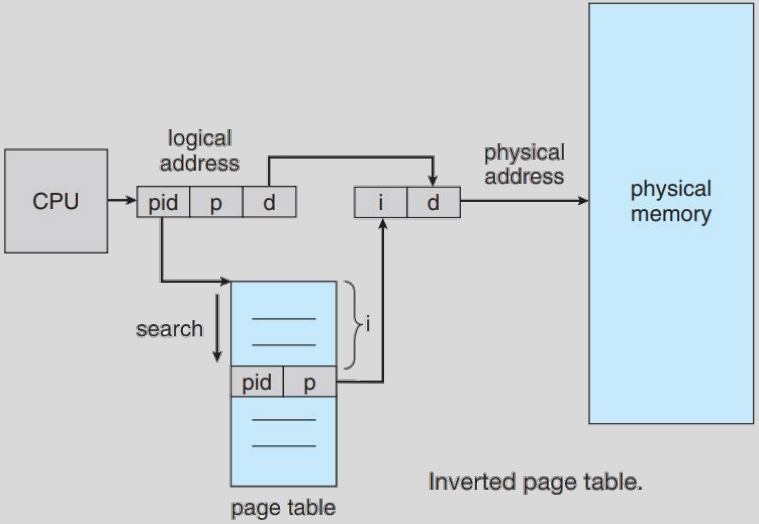
ESTRUCTURA DE UNA TABLA DE PÁGINAS

Supongamos que un proceso quiere acceder a su página virtual `p`. El sistema operativo busca en la tabla de páginas invertidas si alguna entrada coincide con:

- `pid == id proceso actual`
- `Número de página virtual == p`

Si se encuentra: Se obtiene el número de marco físico (índice de entrada). Si no: Es un fallo de página, y por lo tanto se carga la página en un marco.

Ventaja: usa mucha menos memoria para las tablas de páginas (especialmente con muchos procesos).
Contra: la búsqueda es más lenta porque es posible que tengamos que buscar en toda la tabla.



ESTRUCTURA DE UNA TABLA DE PÁGINAS

Por ejemplo, Linux no utiliza tablas de páginas invertidas ni tablas de páginas hash para la gestión de memoria de propósito general. En su lugar, el Linux moderno utiliza:

- **Tablas de Páginas Multinivel**
- https://github.com/JereFassi/utn-so/blob/main/docs/MultiLevelPageTable.md

SWAPPING

SWAPPING

Como sabemos, para que un proceso se ejecute, sus instrucciones y datos deben estar en memoria principal. Sin embargo, no siempre es necesario que el proceso completo permanezca en RAM todo el tiempo. El sistema operativo puede intercambiar (*swap*) un proceso o parte de él entre la memoria principal (RAM) y un almacenamiento secundario (como el disco).

Este mecanismo se conoce como swapping, y ocurre cuando el sistema:

- Mueve un proceso fuera de la memoria (a disco) para liberar espacio.
- Vuelve a cargarlo en memoria cuando debe continuar su ejecución.

¿Por qué es útil el swapping?

- Porque permite ejecutar más procesos de los que físicamente caben en la memoria.
- Aumenta el grado de multiprogramación.
- Libera espacio para tareas más urgentes o de mayor prioridad.

El sistema operativo gestiona este movimiento para mantener un equilibrio entre disponibilidad de memoria y rendimiento del sistema.

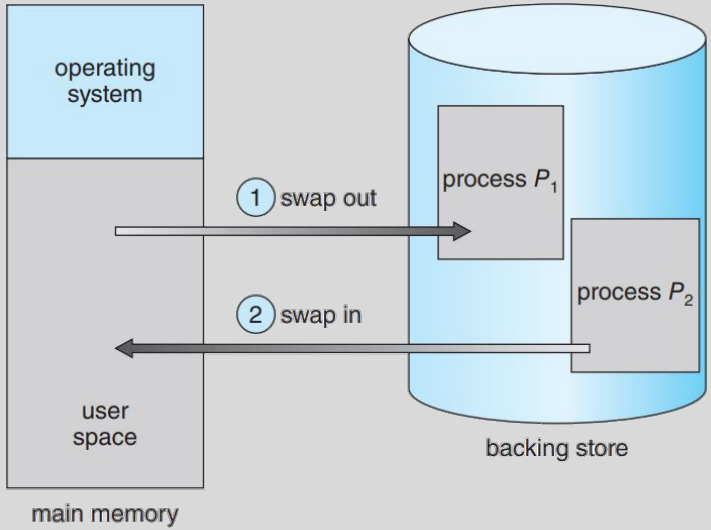
ESTRUCTURA DE UNA TABLA DE PÁGINAS

- **Swap Estándar**

En el swap tradicional, se transfiere el proceso completo entre RAM y disco. Para que esto sea posible, se requieren:

- Espacio suficiente en disco para almacenar las imágenes de memoria de los procesos.
- Acceso directo al contenido de los procesos intercambiados.
- Copia de todas las estructuras del proceso, incluyendo:
  - Registros.
  - Pila.
  - Segmento de datos.

En caso de procesos multihilo: esto se expande a las estructuras de cada hilo. Además, el sistema debe mantener metadatos asociados a cada proceso swap-eado, para poder restaurarlo correctamente en la memoria cuando sea necesario.



Standard swapping of two processes using a disk as a backing store.

ESTRUCTURA DE UNA TABLA DE PÁGINAS

- **Swap con Paginado**

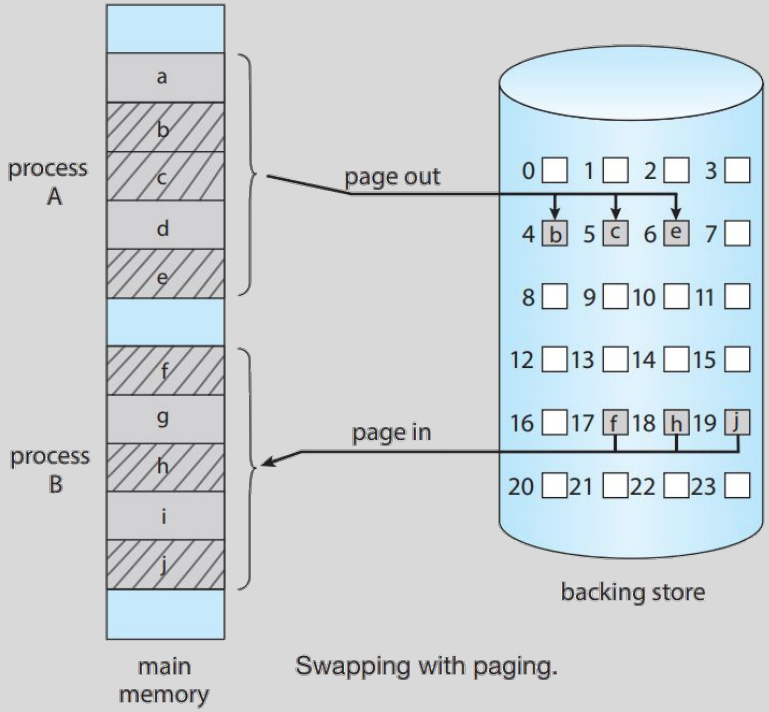
Mover procesos completos puede ser ineficiente y lento, especialmente en sistemas con muchos procesos o con procesos grandes. Por eso, los sistemas modernos (como Linux y Windows) adoptan un enfoque más flexible: el swapping a nivel de página.
En este modelo:

- Solo se intercambian páginas individuales del proceso.
- Esto reduce el tiempo y recursos necesarios para el intercambio.

Operaciones:

- *Page-out*: mueve una página de memoria a disco.
- *Page-in*: recupera una página del disco a memoria.

Este modelo combina la eficiencia del paginado con la flexibilidad del swapping, permitiendo una gestión dinámica y granular de la memoria.



Swapping with paging.

Muchas Gracias

Jeremías Fassi

Javier E. Kinter