Memoria principal
9

Sistemas operativos y distribuidos

Gustavo Distel
gd@cs.uns.edu.ar

DCIC - UNS

Memoria principal

Contenido

- Generalidades.
- Asignación de memoria contigua.
- Paginación.
- Estructura de la tabla de páginas.
- *Swapping*.
- Ejemplo: arquitecturas Intel de 32 y 64 bits.
- Ejemplo: arquitectura ARMv8.

Generalidades

- La administración de la memoria es fundamental, dado que ésta se comparte entre múltiples procesos.
- La mayoría de los algoritmos de administración de memoria requieren soporte de *HW*, por lo que muchos sistemas tienen una gestión de memoria estrechamente integrada entre el *HW* y el SO.
- La memoria consta de una gran arreglo de *bytes*, cada uno con su propia dirección.
- A medida que un programa se ejecuta, la unidad de memoria solo ve un flujo de direcciones de memoria; no sabe cómo se generan o para qué sirven (instrucciones o datos).
- En consecuencia, se puede ignorar cómo se genera una dirección de memoria, solo interesa la secuencia de direcciones de memoria generadas por el programa en ejecución.

Generalidades

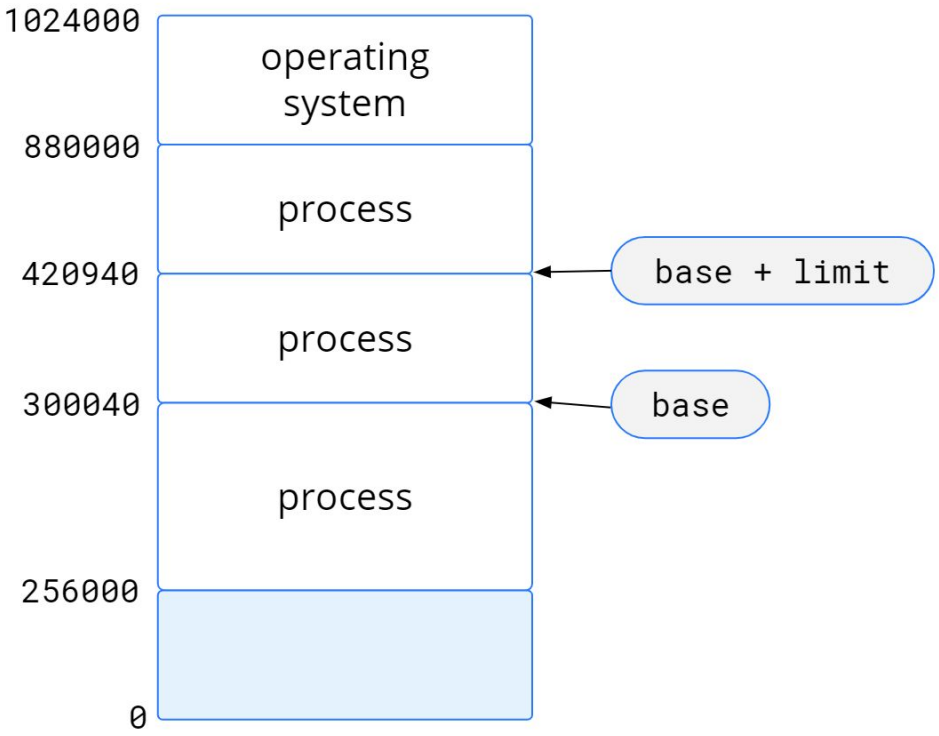
Hardware Básico

- Se debe proteger el SO de los procesos del usuario, así como también proteger los procesos de los usuarios entre sí.
- El *HW* debe proporcionar esta protección, ya que el SO no suele intervenir entre la *CPU* y sus accesos a la memoria (por razones de *performance*).
- Primero se debe asegurar que cada proceso tenga un espacio de memoria separado.
- Esta disposición protege los procesos entre sí, dado que es fundamental contar con múltiples procesos cargados en la memoria y ejecutarlos concurrentemente.
- A su vez, se necesita determinar el rango de direcciones legales a las que puede acceder el proceso y garantizar que el proceso solo pueda acceder a dichas direcciones.
- Esta protección se puede proveer mediante el uso de dos registros: un `base` y un `límite`.

Generalidades

Hardware Básico

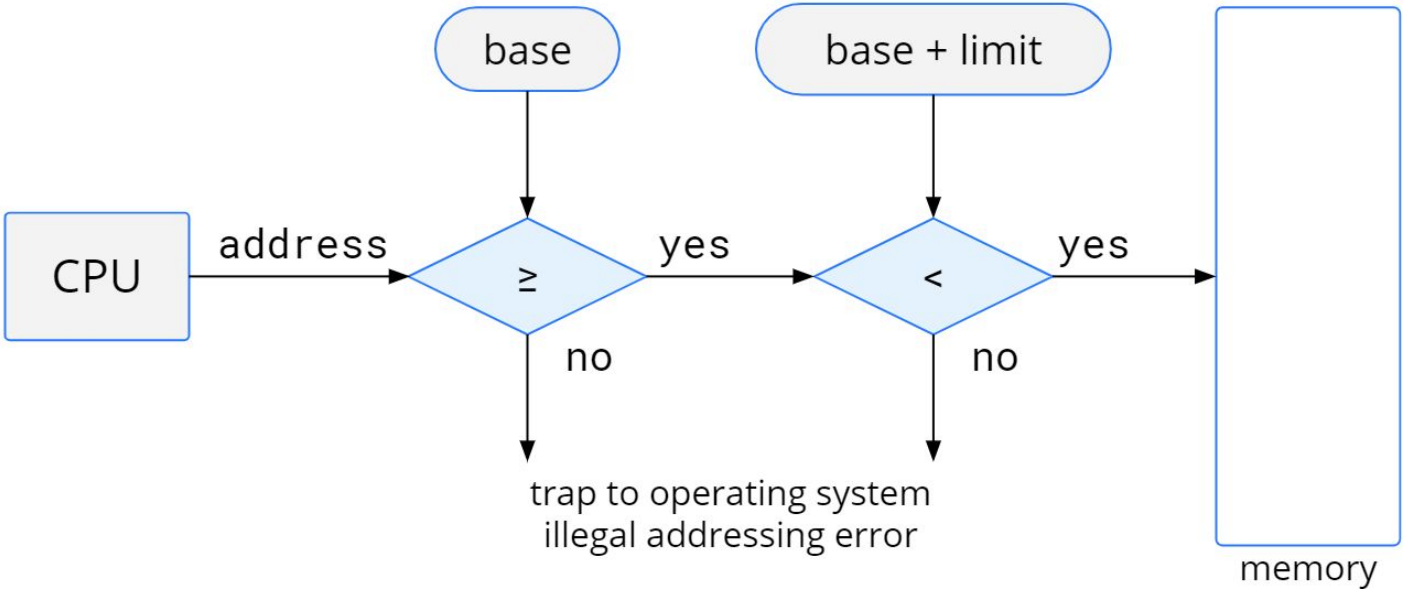
- El `registro base` contiene la dirección de memoria física legal más pequeña.
- El `registro límite` especifica el rango.
- Por ej.: si `base = 300.040` y `límite = 120.900`, entonces el programa puede acceder a todas las direcciones entre `300.040` y `420.939` (inclusive).
- La protección del espacio de memoria se logra haciendo que el *HW* de la *CPU* compare cada dirección generada en modo usuario con los registros.



Generalidades

Hardware Básico

- Cualquier intento de un programa que se ejecute en modo usuario de acceder a memoria del SO o a memoria de otros usuarios, resulta en una *trap* al SO (error fatal).
- Este esquema evita que un programa de usuario modifique (accidental o deliberadamente) el código o las estructuras de datos del SO o de otros usuarios.



Generalidades

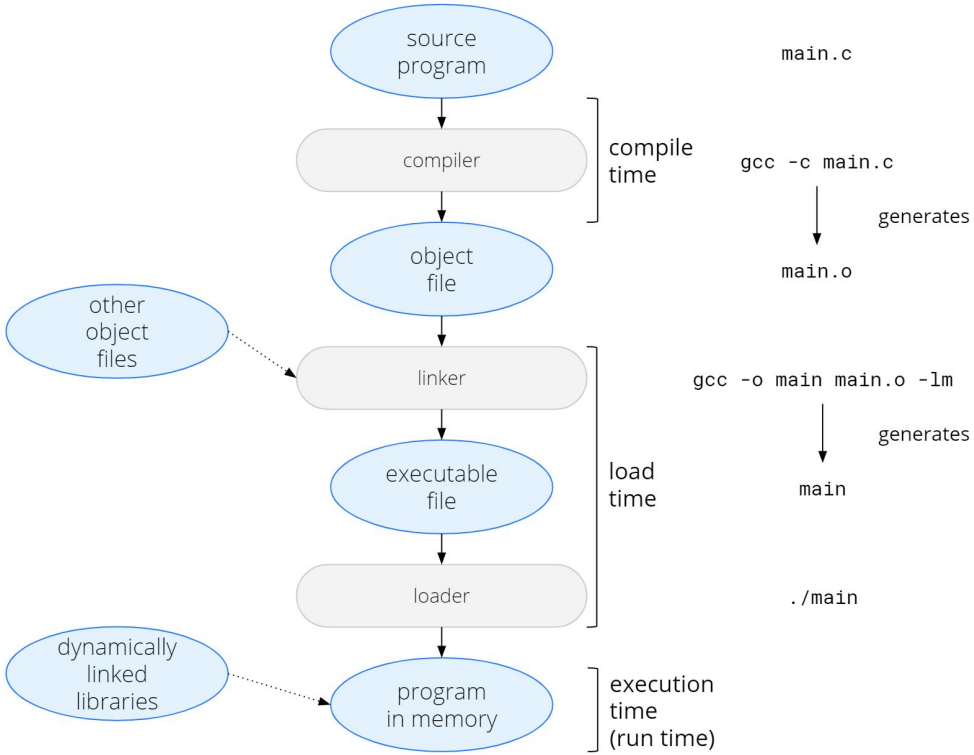
Hardware Básico

- Los registros base y límite solo pueden ser cargados por el SO con una instrucción privilegiada.
- Dado que las instrucciones privilegiadas se pueden ejecutar solo en modo *kernel*, y que el SO es el único que se ejecuta en modo *kernel*, solo este puede cargar los registros base y límite.
- Este esquema permite que el SO cambie el valor de los registros y evita que los programas de usuario cambien el contenido de los registros.
- El SO tiene acceso ilimitado tanto a su memoria como a la memoria de los usuarios.
- Esto le permite cargar los programas de los usuarios en la memoria de los usuarios, hacer *dump* en caso de errores, acceder y modificar los parámetros de las llamadas al sistema, realizar E/S desde y hacia la memoria del usuario y proporcionar otros servicios.

Generalidades

Enlace de direcciones (*Address Binding*)

- Un programa pasa por varias etapas (algunas opcionales) antes de ejecutarse.
- Las direcciones se representan de diferentes formas en estas etapas.
- Las direcciones en el programa fuente son generalmente simbólicas (ej.: una variable).
- El compilador enlaza (*binds*) estas direcciones simbólicas a direcciones reubicables (`14 bytes` desde el comienzo de este módulo).
- El enlazador (*linker*) o cargador (*loader*) a su vez enlaza las direcciones reubicables a direcciones absolutas (como `74014`).
- Cada enlace (*binding*) es una asignación de un espacio de direcciones a otro.



Generalidades

Enlace de direcciones (*Address Binding*)

- El enlace de instrucciones y datos a direcciones de memoria se puede hacer en:
  - **Tiempo de compilación**: si se sabe en tiempo de compilación dónde residirá el proceso en memoria, se puede generar código absoluto.
    - Si la ubicación de inicio cambia, entonces es necesario volver a compilar el código.
  - **Tiempo de carga**: si no se sabe en tiempo de compilación dónde residirá el proceso en memoria, entonces el compilador debe generar código reubicable.
    - El enlace (*binding*) final se retrasa hasta la carga.
    - Si la dirección de inicio cambia, solo necesitamos volver a cargar el código de usuario para incorporar este valor modificado.
  - **Tiempo de ejecución**: si el proceso se puede mover durante su ejecución de un segmento de memoria a otro, el enlace debe retrasarse hasta ejecución.
    - Se necesita *HW* especial para este esquema.
    - La mayoría de los SOs usan este método.

Generalidades

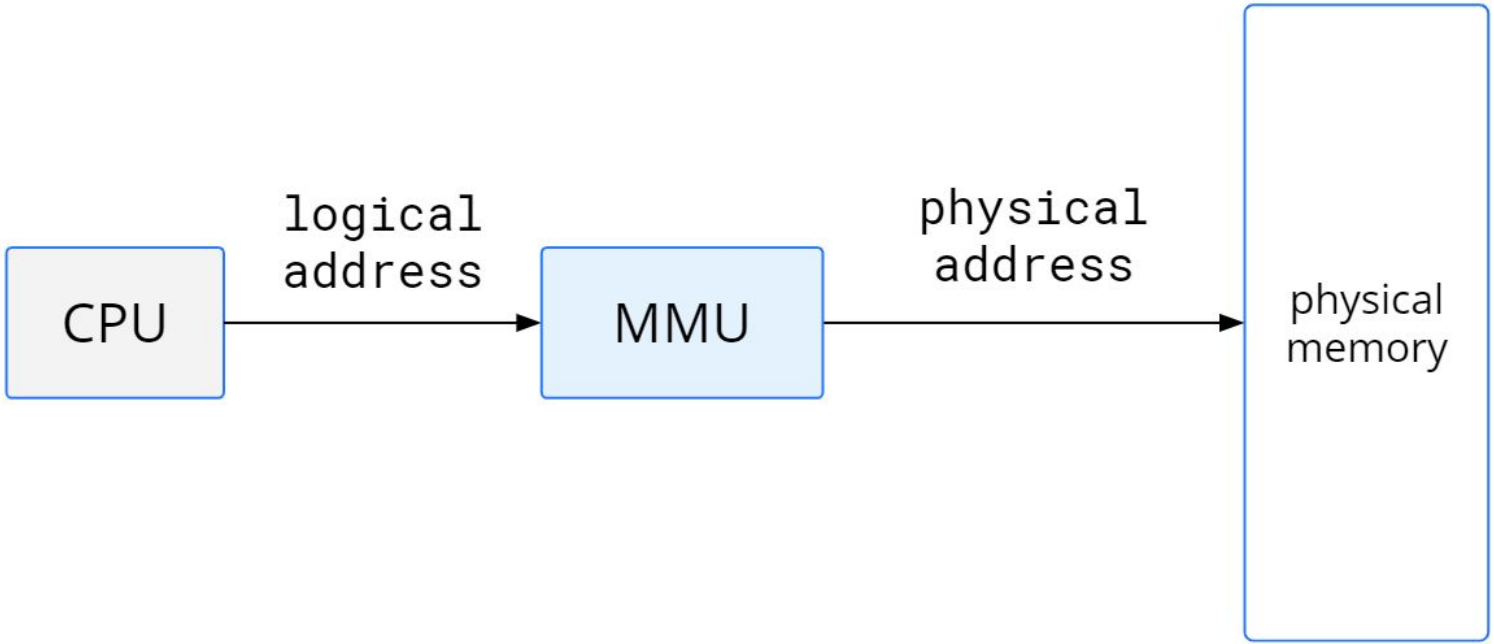
Espacio de direcciones lógico versus físico

- Una dirección generada por la *CPU* es una dirección lógica, mientras que una dirección en la unidad de memoria es una dirección física.
- El enlace de direcciones en tiempo de compilación o carga generan direcciones lógicas y físicas idénticas. Sin embargo, el esquema en tiempo de ejecución resulta en diferentes direcciones lógicas y físicas.
- Una dirección lógica también es conocida como una dirección virtual (*virtual address*).
- El conjunto de todas las direcciones lógicas generadas por un programa es el espacio de direcciones lógicas (*logical address space*).
- El conjunto de todas las direcciones físicas correspondientes a estas direcciones lógicas es el espacio de direcciones físicas (*physical address space*).
- Por lo tanto, en el esquema de enlace de direcciones en tiempo de ejecución, los espacios de direcciones lógicos y físicos difieren.

Generalidades

Espacio de direcciones lógico versus físico

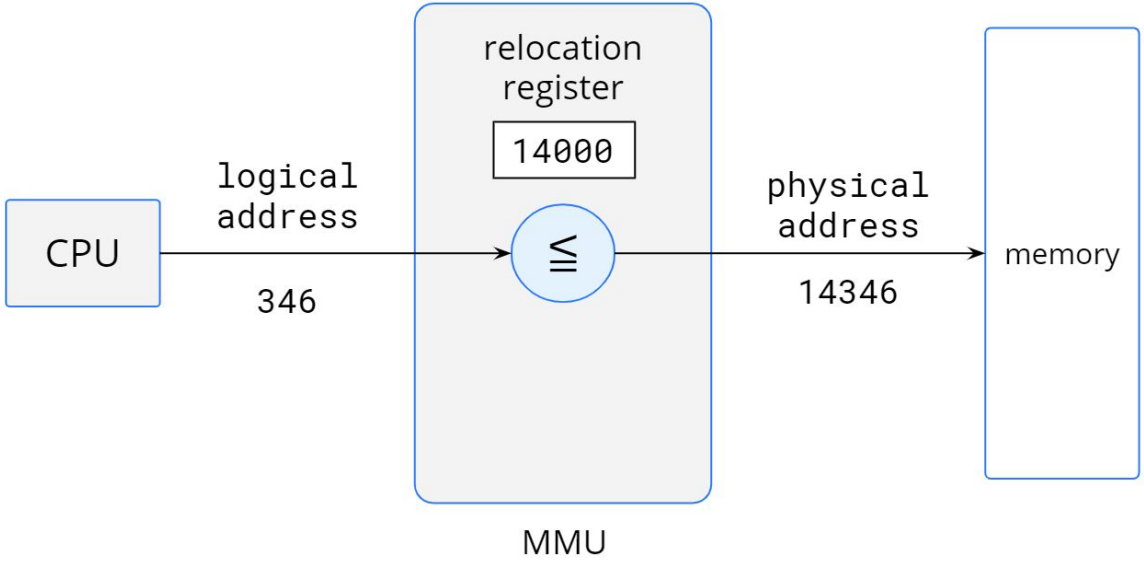
- El mapeo en tiempo de ejecución de direcciones virtuales a físicas se realiza mediante *HW* llamado unidad de administración de memoria (*memory-management unit - MMU*).



Generalidades

Espacio de direcciones lógico versus físico

- El registro base ahora se denomina registro de reubicación (*relocation register*).
- El valor en el registro de reubicación se agrega a cada dirección generada por un proceso de usuario en el momento en que la dirección se envía a memoria.
- Por ej., si base es `14000`, entonces un intento del usuario de acceder a la ubicación `0` se reubica dinámicamente en la ubicación `14000`; un acceso a la ubicación `346` se asigna a la ubicación `14346`.



Generalidades

Espacio de direcciones lógico versus físico

- El programa de usuario trata con direcciones lógicas.
- El *HW* de mapeo convierte las direcciones lógicas en direcciones físicas.
- La ubicación final de una dirección de memoria no se determina hasta que se haga la referencia.
- Ahora tenemos dos tipos diferentes de direcciones:
  - Direcciones lógicas (en el rango de `0` a `max`).
  - Direcciones físicas (en el rango de `R + 0` a `R + max` para un valor base `R`).
- El programa de usuario genera solo direcciones lógicas y considera que el proceso se ejecuta en ubicaciones de memoria de `0` a `max`.
- Sin embargo, estas direcciones lógicas deben asignarse a direcciones físicas antes de ser utilizadas.

Generalidades

Carga dinámica

- Con la carga dinámica, una rutina no se carga hasta que se invoca.
- Cuando una rutina necesita llamar a otra, primero verifica si la otra rutina se ha cargado.
  - Si no es así, se llama al cargador de enlaces reubicable para cargar la rutina deseada en la memoria y actualizar las tablas de direcciones del programa.
- La ventaja es que una rutina se carga solo cuando se necesita, ahorrando memoria física.
- Este método es útil cuando se necesitan grandes cantidades de código para manejar casos que ocurren con poca frecuencia, como las rutinas de error.
- La carga dinámica no requiere soporte del SO. Es responsabilidad de los usuarios diseñar programas para aprovecharla.
- Sin embargo, los SOs ayudan al programador al proporcionar rutinas de biblioteca para implementarla.

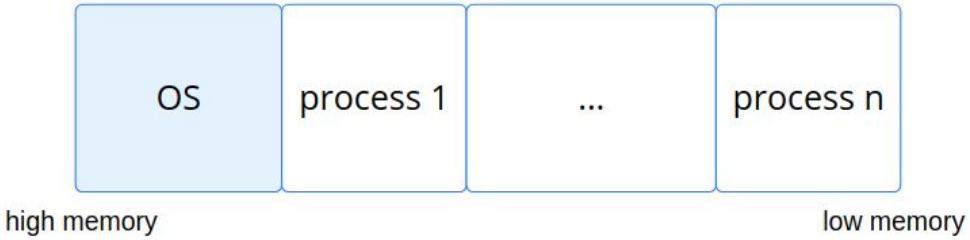
Generalidades

Enlaces dinámicos y bibliotecas compartidas

- Las bibliotecas enlazadas dinámicamente (Dynamically linked libraries - DLLs) son bibliotecas del sistema que se enlazan a los programas de usuario cuando se ejecutan.
- Aquí el enlace, en lugar de la carga, se pospone hasta el momento de ejecución; esta característica se usa generalmente con bibliotecas del sistema, como la estándar `C`.
- Sin *DLLs* cada programa debería incluir una copia de la biblioteca (o rutina/s) en la imagen ejecutable.
- A su vez las *DLLs* se pueden compartir entre múltiples procesos, de modo que solo una instancia de la *DLL* esté en memoria principal.
    - Las *DLLs* también se conocen como bibliotecas compartidas y se usan ampliamente en sistemas `Windows` y `Linux`.
- A diferencia de la carga dinámica, los enlaces dinámicos y las bibliotecas compartidas generalmente requieren la ayuda del SO.
- El SO es el único que puede verificar si la rutina está en el espacio de memoria de otro proceso o que puede permitir que múltiples procesos accedan a las mismas direcciones de memoria.

Asignación de memoria contigua

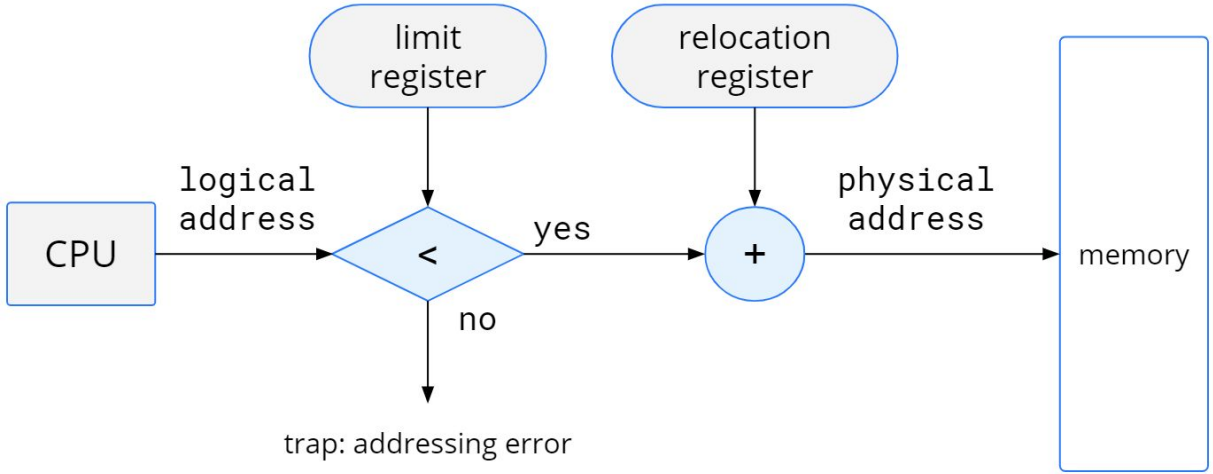
- La memoria generalmente se divide en dos particiones:
  - SO.
  - Procesos de usuarios.
- El SO puede estar en direcciones de memoria baja o en direcciones de memoria alta.
- Esta decisión depende de muchos factores, como la ubicación del vector de interrupciones.
- Sin embargo, muchos SOs (incluidos `Linux` y `Windows`) colocan el SO en memoria alta.
- En este esquema, cada proceso está contenido en una sola sección de memoria contigua a la sección que contiene el siguiente proceso.



Asignación de memoria contigua

Protección de memoria

- En un sistema con registro de reubicación y con registro límite se puede impedir que un proceso no acceda a memoria que no le pertenece.
- El registro de reubicación contiene el valor de la dirección física más pequeña; el registro límite contiene el rango de direcciones lógicas (por ej., `reubicación = 100040, límite = 74600`).
- Cada dirección lógica debe estar dentro del rango especificado por el registro de límite.
- La *MMU* mapea la dirección lógica dinámicamente sumando el valor del registro de reubicación.



Asignación de memoria contigua

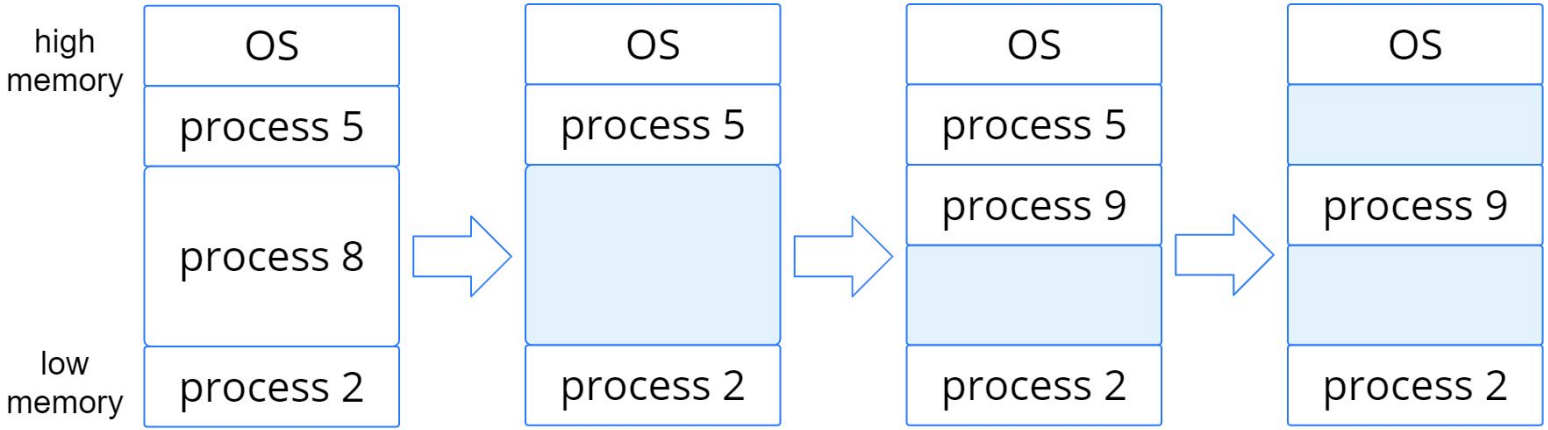
Protección de memoria

- Cuando el planificador de la *CPU* selecciona un proceso para ejecutar, el despachador carga los registros de reubicación y límite con los valores correctos en el cambio de contexto.
- Debido a que cada dirección generada por la *CPU* se compara con estos registros, podemos proteger tanto el SO como los demás programas y datos de los usuarios para que no sean modificados por este proceso en ejecución.
- El esquema de registro de reubicación proporciona una forma efectiva para permitir que el tamaño del SO cambie dinámicamente, por ej. para cargar un controlador de dispositivo.

Asignación de memoria contigua.

Asignación de memoria

- Uno de los métodos más simples para asignar memoria es asignar procesos a particiones de tamaño variable en la memoria, donde cada partición puede contener exactamente un proceso.
- En este esquema, el SO mantiene una tabla que indica qué partes de la memoria están disponibles y cuáles están ocupadas.
- Inicialmente, toda la memoria está disponible para los procesos del usuario y se considera un gran bloque de memoria disponible, un hueco.
- Finalmente, con el uso, la memoria contiene un conjunto de huecos de varios tamaños.



Asignación de memoria contigua.

Asignación de memoria

- Cuando un proceso arriba y necesita memoria, el sistema busca un hueco en el conjunto que sea lo suficientemente grande.
- Si el hueco es demasiado grande, se divide en dos partes. Una parte se asigna al proceso de llegada; el otro se devuelve al conjunto de huecos.
- Cuando un proceso termina, libera su bloque de memoria, que luego se vuelve a colocar en el conjunto de huecos.
- Si el nuevo hueco es adyacente a otros huecos, estos se fusionan para formar un hueco más grande.
- Este procedimiento es un caso particular del problema general de asignación dinámica de almacenamiento, que se refiere a cómo satisfacer una solicitud de tamaño **n** de una lista de huecos libres.
- Hay muchas soluciones a este problema. Las estrategias de primer ajuste, mejor ajuste y peor ajuste son las más utilizadas para seleccionar un hueco libre del conjunto de huecos disponibles.

Asignación de memoria contigua.

Asignación de memoria

- **Primer ajuste**: asigna el primer hueco que sea lo suficientemente grande.
  - La búsqueda puede comenzar al comienzo o en el lugar donde finalizó la búsqueda previa.
  - Se puede dejar de buscar tan pronto como se encuentre un hueco libre que sea lo suficientemente grande.
- **Mejor ajuste**: asigna el hueco más pequeño que sea lo suficientemente grande.
  - Debemos buscar en toda la lista, a menos que la lista esté ordenada por tamaño.
  - Esta estrategia produce el hueco sobrante más pequeño.
- **Peor ajuste**: asigna el hueco más grande.
  - Nuevamente, debemos buscar en toda la lista, a menos que esté ordenada por tamaño.
  - Esta estrategia produce el hueco sobrante más grande, que puede ser más útil que el hueco sobrante más pequeño del enfoque de mejor ajuste.

Asignación de memoria contigua.

Asignación de memoria

- Las simulaciones han demostrado que tanto el primer ajuste como el mejor ajuste son mejores que el peor ajuste en términos de tiempo y utilización del almacenamiento.
- Ni el primer ajuste ni el mejor ajuste son claramente mejores uno que otro en términos de utilización del almacenamiento, pero el primer ajuste es generalmente más rápido.

Asignación de memoria contigua.

Fragmentación

- **Fragmentación externa**: existe cuando hay suficiente espacio para satisfacer una solicitud, pero los espacios disponibles no son contiguos: el almacenamiento está fragmentado en una gran cantidad de pequeños huecos.
  - Este problema de fragmentación puede ser grave. En el peor de los casos, podríamos tener un bloque de memoria libre entre cada dos procesos.
  - Si todos estos pequeños fragmentos de memoria estuvieran en un gran bloque libre, se podrían ejecutar varios procesos más.
  - No importa qué algoritmo de los anteriores se use, la fragmentación externa será un problema.
- **Fragmentación interna**: memoria no utilizada que es interna a una partición.
  - Sea un esquema de asignación de particiones múltiples con un hueco de `18,464` *`bytes`*. Supongamos que el siguiente proceso solicita `18.462` *`bytes`*. Si asignamos exactamente el bloque solicitado, nos queda un hueco de `2` *`bytes`*.

Asignación de memoria contigua.

Fragmentación

- Una solución al problema de la fragmentación externa es la compactación.
- El objetivo es colocar toda la memoria libre en un único bloque grande.
- Sin embargo, la compactación no siempre es posible. Si la reubicación es estática y se realiza en el momento del montaje o la carga, no se puede realizar la compactación. Solo es posible si la reubicación es dinámica y se realiza en tiempo de ejecución.
- Otra posible solución a la fragmentación externa es permitir que el espacio de direcciones lógicas de los procesos no sea contiguo, permitiendo así que se asigne memoria física a un proceso donde esté disponible.
- Esta es la estrategia utilizada en la paginación, la técnica de administración de memoria más utilizada por los sistemas informáticos.

Paginación

- Permite que el espacio de direcciones físicas de un proceso sea no contiguo.
- La paginación evita la fragmentación externa y la compactación, dos problemas de la asignación de memoria contigua.
- Debido a que ofrece numerosas ventajas se utiliza en la mayoría de los SOs, desde servidores hasta dispositivos móviles.
- La paginación se implementa mediante la cooperación entre el SO y el *HW*.

Paginación

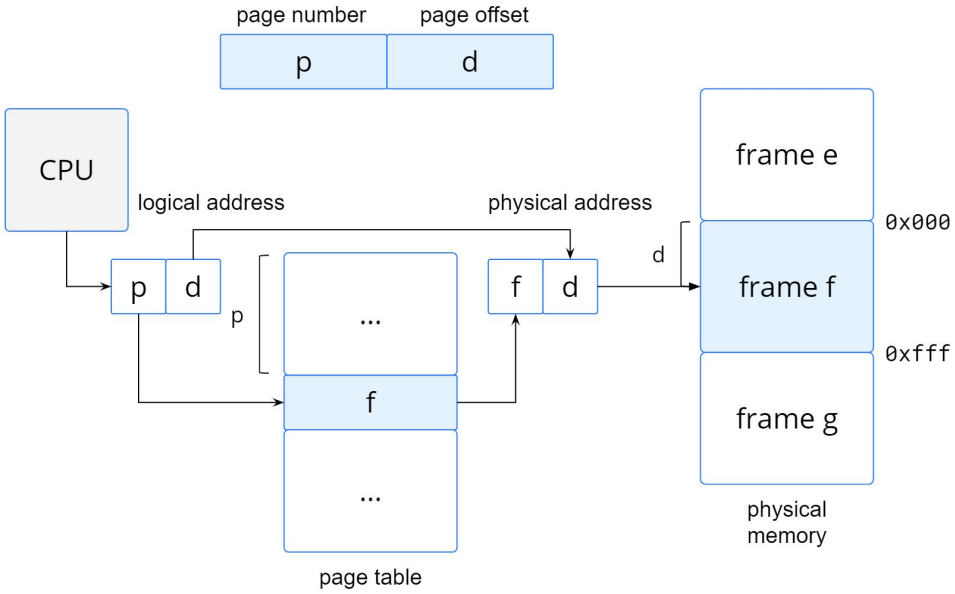
Método básico

- Consta en dividir la memoria física en bloques de tamaño fijo llamados marcos (*frames*) y dividir la memoria lógica en bloques del mismo tamaño llamados páginas (*pages*).
- Cuando un proceso se ejecuta, sus páginas se cargan en cualquier marco disponible desde la fuente (un sistema de archivos).
- El espacio de direcciones lógicas podría estar totalmente separado del espacio de direcciones físicas, por lo que un proceso puede tener un espacio de direcciones lógicas de **64 *bits*** aunque el sistema tenga menos de 2^{64} ***bytes*** de memoria física.
- Cada dirección generada por la *CPU* se divide en dos partes:
  - Un número de página (**p**) y
  - Un desplazamiento de página (**d**):

Paginación

Método básico

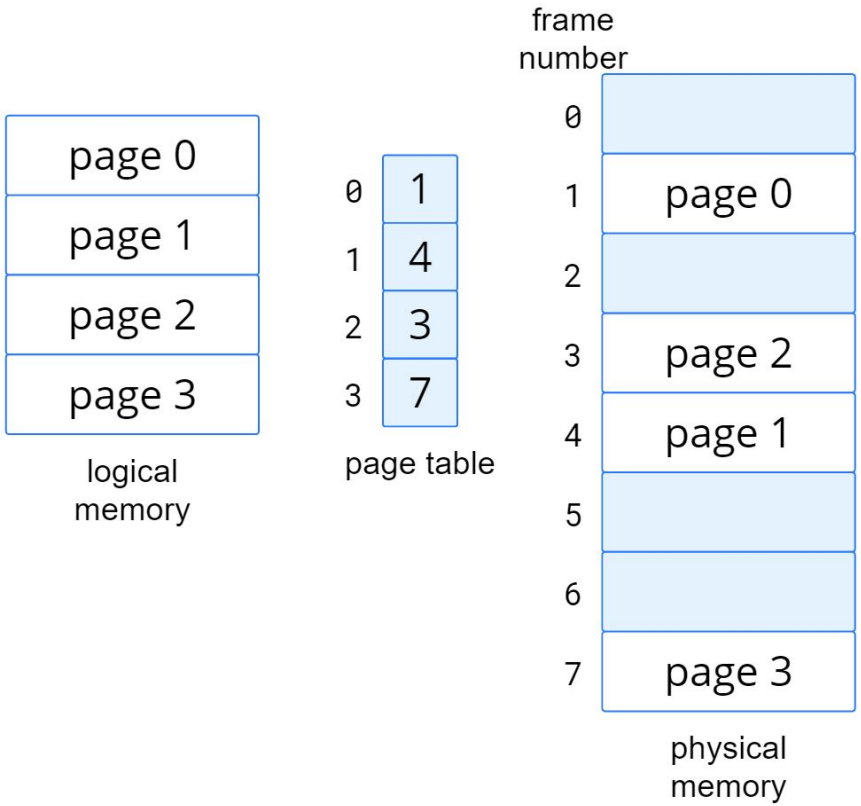
- El número de página se utiliza como índice en una tabla de página (TP - *page table*) por proceso.
- La tabla de páginas contiene la dirección base de cada marco en la memoria física, y el desplazamiento es la ubicación en el marco al que se hace referencia.
- Por lo tanto, la dirección base del marco se combina con el desplazamiento de página para definir la dirección de memoria física.



Paginación

Método básico

- El modelo de paginación de la memoria.



Paginación

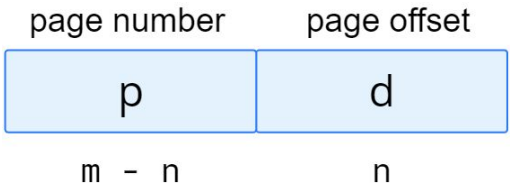
Método básico

- Pasos realizados por la *MMU* para traducir una dirección lógica generada por la *CPU* a una dirección física:
  - 1. Extraer el número de página **p** y usarlo como índice en la tabla de páginas.
  - 2. Extraer el número correspondiente de marco **f** de la tabla de páginas.
  - 3. Reemplazar el número de página **p** en la dirección lógica con el número de marco **f**.
- Como el desplazamiento **d** no cambia, no se reemplaza, y el número de marco y el desplazamiento ahora comprenden la dirección física.

Paginación

Método básico

- El tamaño de la página (como el tamaño del marco) está definido por *HW*.
- El tamaño de una página es una potencia de `2`, que generalmente varía entre `4KB` y `1GB` por página, dependiendo de la arquitectura de la computadora.
- La selección de una potencia de `2` como tamaño de página hace que la traducción de una dirección lógica a un número de página y a un desplazamiento sea fácil.
- Si el tamaño del espacio de direcciones lógicas es de 2^m, y el tamaño de una página es de 2^n *bytes*, entonces los `m-n` *bits* de orden superior de una dirección lógica designan el número de página, y los `n` *bits* de orden inferior designan el desplazamiento de página.
- Por lo tanto, la dirección lógica es la siguiente:

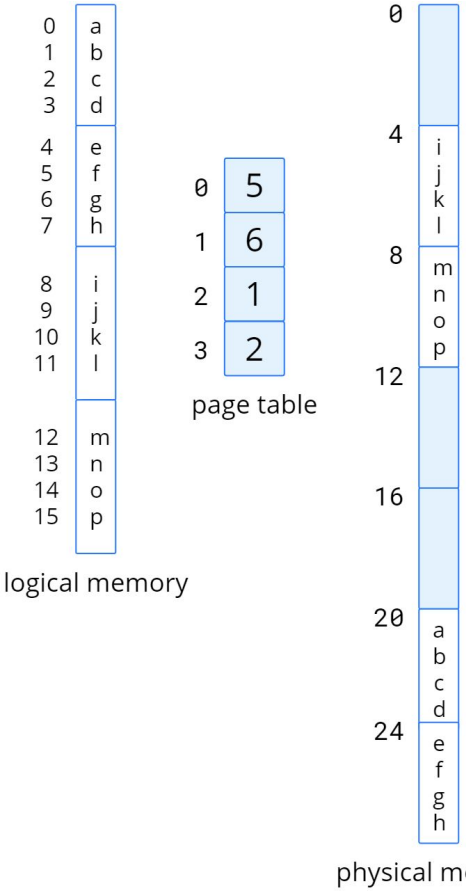


- Donde `p` es el índice en la tabla de páginas y `d` es el desplazamiento dentro de la página.

Paginación

Método básico

- Dirección lógica: `n = 2` y `m = 4`.
- Página de `4 bytes` y memoria física de `32 bytes` (`8` páginas).
- La dirección lógica `0` es la página `0`, desplazamiento `0`.
- Al indexar en TP la página `0` está en el marco `5`.
- La dirección lógica `0` se asigna a la dirección física `20[=(5×4)+0]`.
- La dirección lógica `3` (página `0`, desplazamiento `3`) se asigna a la dirección física `23[=(5×4)+3]`.
- La dirección lógica `4` es la página `1`, desplazamiento `0`; de acuerdo con la tabla de páginas, la página `1` se asigna al marco `6`.
- La dirección lógica `4` se asigna a la dirección física `24[=(6×4)+0]`.
- La dirección lógica `13` se asigna a la dirección física `9`.



Paginación

Método básico

- Cuando usamos un esquema de paginación, no tenemos fragmentación externa: cualquier marco libre puede asignarse al proceso que lo necesite.
- Sin embargo, puede haber fragmentación interna.
- Por ejemplo,
  - Si el tamaño de la página es de `2,048` *bytes*, un proceso de `72,766` *bytes* necesitará `35` páginas más `1,086` *`bytes`*.
  - Se le asignarán `36` marcos, lo que dará como resultado una fragmentación interna de:
    - `2.048 - 1.086 = 962` *`bytes`*.
- En el peor de los casos, un proceso necesitaría `n` páginas más `1` *byte*. Se asignaría `n + 1` marcos, lo que da como resultado la fragmentación interna de casi un marco completo.

Paginación

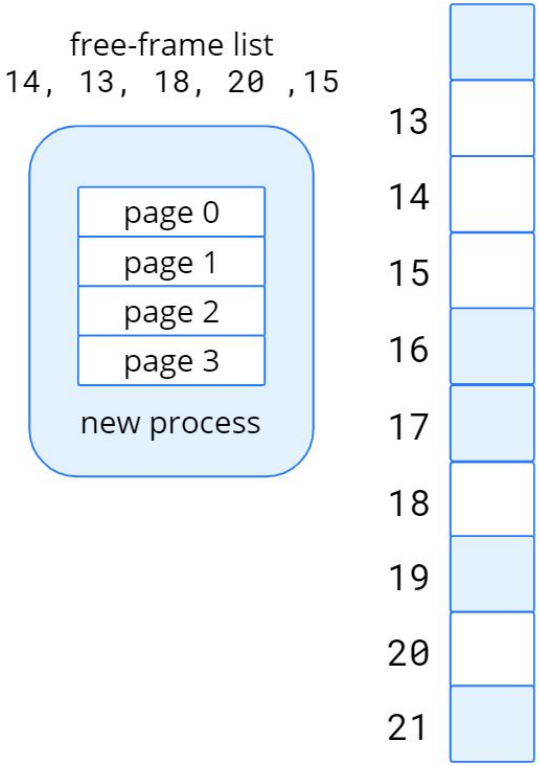
Método básico

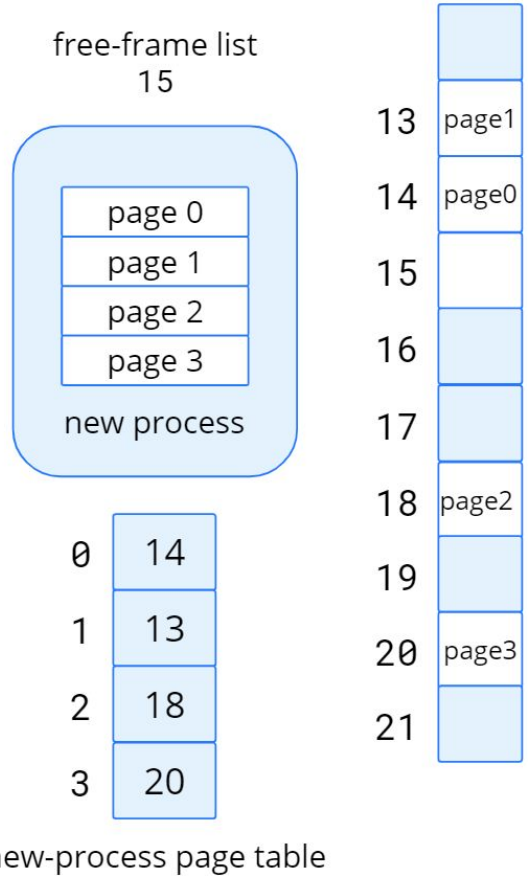
- Si el tamaño del proceso es independiente del tamaño de página, se espera que la fragmentación interna promedie media página por proceso.
  - Entonces son deseables tamaños de página pequeños.
- Sin embargo, la sobrecarga está en la tabla de páginas, y esta sobrecarga se reduce a medida que aumenta el tamaño de las páginas.
- Además, la E/S de disco es más eficiente cuando la cantidad de datos que se transfiere es mayor.
- En general, los tamaños de página han crecido con el tiempo a medida que los procesos, los datos y la memoria principal se han vuelto más grandes.
- En sistemas `x86-64`, `Windows 10` admite tamaños de página de `4 KB` y `2 MB`.
- `Linux` tiene un tamaño de página predeterminado (generalmente `4 KB`) y un tamaño de página más grande dependiente de la arquitectura llamado *huge pages*.

Paginación

Método básico

- Si el proceso requiere **n** páginas, al menos **n** marcos deben estar disponibles en la memoria.





Paginación

Método básico

- Un aspecto importante de la paginación es la separación clara entre la vista de la memoria del programador y la memoria física real.
- El programador ve la memoria como un solo espacio, que solo contiene su programa.
- El *HW* de traducción de direcciones concilia la diferencia entre la vista de la memoria del programador y la memoria física real.
- Las direcciones lógicas se traducen en direcciones físicas; esta asignación está oculta para el programador y está controlada por el SO.
- El proceso de usuario no puede acceder a memoria que no posee. No tiene forma de direccionar la memoria fuera de su tabla de páginas, y la tabla incluye solo aquellas páginas que posee el proceso.

Paginación

Método básico

- El SO administra la memoria física manteniendo en una única estructura de datos de todo el sistema llamada tabla de marcos (*frame table*).
- La tabla de marcos tiene una entrada para cada marco de página físico, que indica si está libre o asignado y, si está asignado, a qué página de qué proceso (o procesos).
- El SO mantiene una copia de la tabla de páginas por proceso, al igual que mantiene una copia del contador de instrucciones y el contenido del registro.
- Esta copia se utiliza para traducir direcciones lógicas a direcciones físicas, siempre que el SO debe asignar manualmente una dirección lógica a una dirección física.
- El despachador de la *CPU* también la utiliza para definir la tabla de páginas de *HW* cuando se asigna un proceso a la *CPU*. Por lo tanto, la paginación aumenta el tiempo de cambio de contexto.

Paginación

Soporte de *hardware*

- Dado que las tablas de páginas son estructuras de datos por proceso, un puntero a esta se almacena en el bloque de control de proceso.
- Cuando el planificador de la *CPU* selecciona un proceso para ejecutar, debe volver a cargar los registros y los valores apropiados en el *HW* de tabla de páginas.
- Debido a su tamaño, la tabla de páginas se mantiene en la memoria principal.
- Un registro base de tabla de páginas (*page-table base register - PTBR*) apunta a la tabla de páginas; cambiar de tablas de páginas requiere solo cambiar este registro, reduciendo sustancialmente el tiempo de cambio de contexto.

Paginación

Soporte de *hardware*: *Translation Look-Aside Buffer (TLB)*

- El almacenamiento de la tabla de páginas en memoria principal conlleva a tiempos de acceso a memoria más lentos.
  - Para acceder a la ubicación `i`, primero se indexa la tabla de páginas → un acceso.
  - El número de marco se combina con el desplazamiento para generar la dirección real, accediendo luego, al lugar deseado en memoria → otro acceso.
  - Se necesitan dos accesos a memoria para acceder a los datos → Solución: *TLB*.
- El *TLB* es una memoria asociativa de alta velocidad. Cada entrada en el *TLB* consta de dos partes: una `clave` (o etiqueta) y un `valor`.
- Cuando al *TLB* se le presenta un ítem, éste se compara simultáneamente con todas las claves. Si se encuentra el ítem, se devuelve el campo de valor correspondiente.
- El *TLB* es un ejemplo de la evolución de la *CPU*: los sistemas han pasado de no tener *TLB* a tener múltiples niveles de *TLB*, del mismo modo que tienen múltiples niveles de cachés.

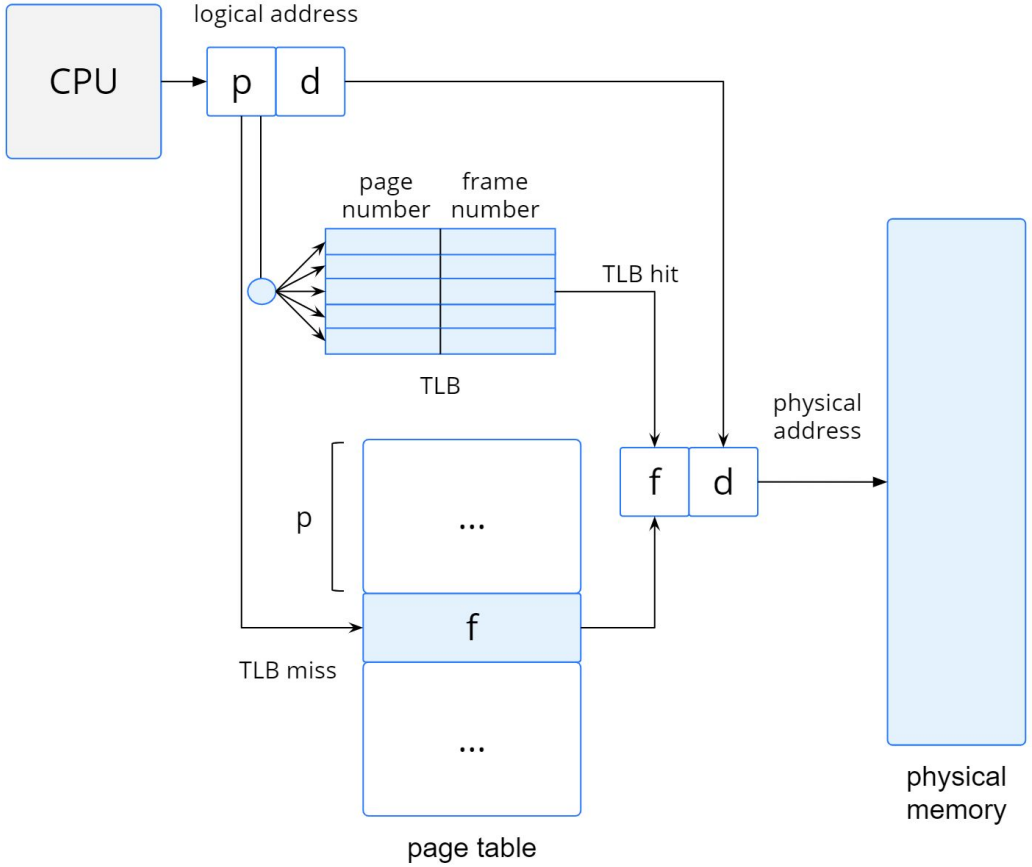
Paginación

Soporte de *hardware*: *Translation Look-Aside Buffer (TLB)*

- ***TLB* y tablas de páginas**: el *TLB* contiene solo algunas de las entradas de la tabla de páginas.
- Cuando la *CPU* genera una dirección lógica, la *MMU* primero comprueba si la página está presente en el *TLB*.
- Si se encuentra, su número de marco está disponible y se utiliza para acceder a memoria.
- Si el número de página no está en el *TLB* (*TLB miss*), se debe acceder a la tabla de páginas para obtener el número de marco y usarlo para acceder a memoria.
- Luego, se agrega el número de página y el número de marco al *TLB*, para que se encuentren rápidamente en la siguiente referencia.
    - Si el *TLB* está lleno se debe seleccionar una entrada existente para su reemplazo con una política de reemplazo, de la cual puede o no participar el SO.

Paginación

Soporte de *hardware*: *Translation Look-Aside Buffer (TLB)*



Paginación

Soporte de *hardware*: *Translation Look-Aside Buffer (TLB)*

- ***Hit ratio***: porcentaje de veces que se encuentra el nro. de página en el *TLB*.
- Un hit ratio de `80%` significa que encontramos el número de página el `80%` de las veces.
- Si toma `10ns` acceder a la memoria → tarda `10ns` cuando el nro. de página está en el *TLB*.
- Si no se encuentra el nro. de página en el *TLB*, primero se accede a la TP en memoria para obtener el nro. de marco (`10ns`) y luego acceder al *byte* deseado en la memoria (`10ns`), con un total de `20ns`.
- Para encontrar el tiempo efectivo de acceso a la memoria (*effective memory-access time*), ponderamos el caso por su probabilidad:
  - **Tiempo de acceso efectivo =** `0.80 × 10 + 0.20 × 20 = 12` nanosegundos
- Cae un `20%` el tiempo de acceso promedio a la memoria (de `10` a `12` nanosegundos).
- Para un `99%` de hit ratio, que es mucho más realista:
  - **Tiempo de acceso efectivo =** `0.99 × 10 + 0.01 × 20 = 10.1` nanosegundos
- Esta mayor tasa de aciertos produce solo una disminución del `1%` en el tiempo de acceso.

Paginación

Soporte de *hardware*: *Translation Look-Aside Buffer (TLB)*

- El *TLB* es una característica de *HW* y, por lo tanto, parecería poco importante para los SOs y sus diseñadores.
- Para un correcto funcionamiento, un diseño de SO para una plataforma determinada debe implementar la paginación de acuerdo con el diseño del *TLB* de la plataforma.
- Del mismo modo, un cambio en el diseño de *TLB* (por ej., entre diferentes generaciones de *CPU* de `Intel`) puede requerir un cambio en la implementación de paginación de los SOs que lo utilizan.

Paginación

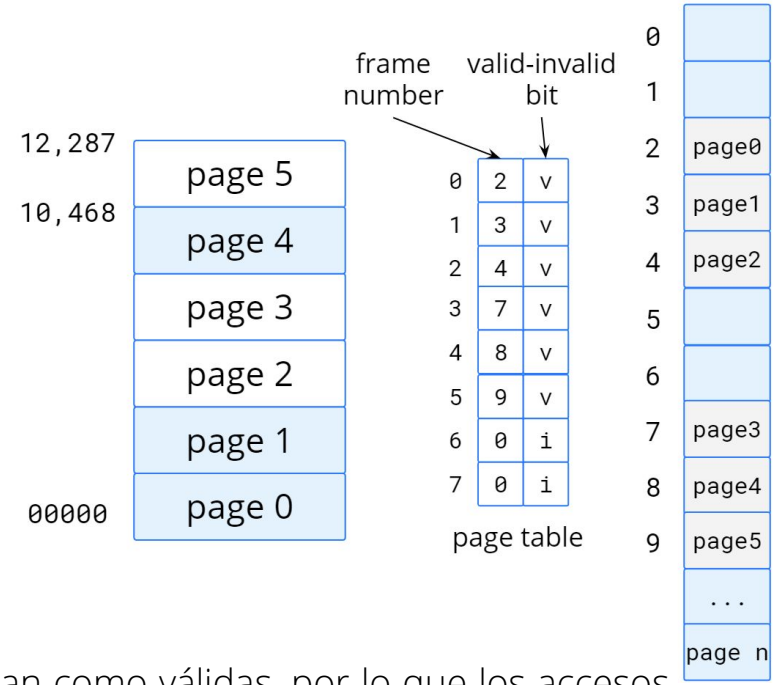
Protección

- La protección de la memoria se logra mediante *bits* asociados a cada marco, que por lo general se mantienen en la tabla de páginas.
- Un *bit* puede definir si una página es de lectura-escritura o de solo lectura.
  - Un intento de escribir en una página de solo lectura provoca una *trap* de *HW* al SO.
- Se puede extender el *HW* para proporcionar protección de ejecución; al proporcionar bits de protección separados para cada tipo de acceso, podemos permitir cualquier combinación de estos accesos.
- *Bit* válido/inválido:
  - **Válido**: la página asociada está en el espacio de direcciones lógicas del proceso y, por lo tanto, es una página legal (o válida).
  - **Inválido**: la página no está en el espacio de direcciones lógicas del proceso.
    - Las direcciones ilegales provocan *traps* mediante el uso del *bit* válido no válido.
    - El SO establece este *bit* por cada página para permitir o no el acceso a la misma.

Paginación

Protección

- Dado un espacio de direcciones de `14 bits` (`0` a `16383`), un programa con direcciones de `0` a `10468` y un tamaño de página de `2 KB`.
- Las direcciones en las páginas `0, 1, 2, 3, 4` y `5` se asignan a través de la TP.
- Cualquier intento de generar una dirección en las páginas `6` o `7` da *bit* inválido generando un *trap* al SO (referencia de página no válida).
- Dado que el programa se extiende solo hasta la dirección `10468`, cualquier referencia que la supere es ilegal.



- Sin embargo, las referencias a la página `5` se clasifican como válidas, por lo que los accesos a direcciones de hasta `12287` son válidos. Solo las direcciones de `12288` a `16383` no son válidas.
- Este problema es el resultado del tamaño de página y refleja la fragmentación interna de la paginación.

Paginación

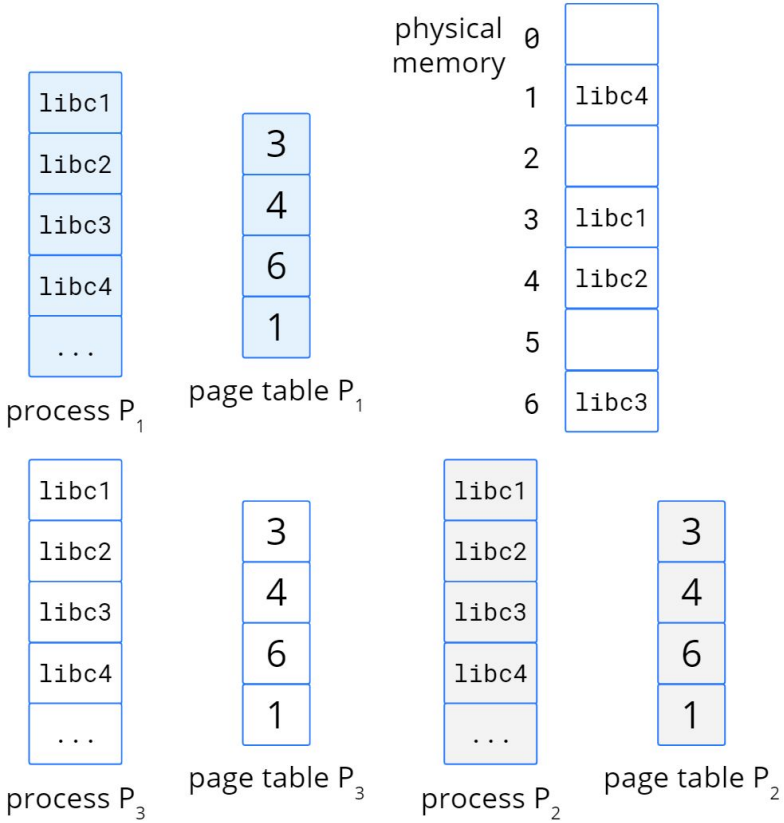
Protección

- Por lo general, los procesos usan solo una pequeña fracción del espacio de direcciones disponible.
- Sería un desperdicio en estos casos crear una tabla de páginas con entradas para cada página en el rango de direcciones.
  - La mayor parte de esta tabla no se utilizaría, pero ocuparía un valioso espacio de memoria.
- Algunos sistemas proporcionan *HW*, en forma de registro de longitud de tabla de páginas (*page-table length register - PTLR*), para indicar el tamaño de la tabla de páginas.
- Este valor se compara con cada dirección lógica para verificar que la dirección se encuentre en el rango válido del proceso.
  - Si falla provoca un *trap* al SO.

Paginación

Páginas compartidas

- Una ventaja de la paginación es la posibilidad de compartir código.
- El **código reentrante** es código que no se modifica a sí mismo: nunca cambia durante la ejecución.
- Por lo tanto, dos o más procesos pueden ejecutar el mismo código al mismo tiempo y compartirlo (por ej. la librería `libc`).
- Algunos SOs implementan memoria compartida utilizando páginas compartidas.



Estructura de la tabla de páginas

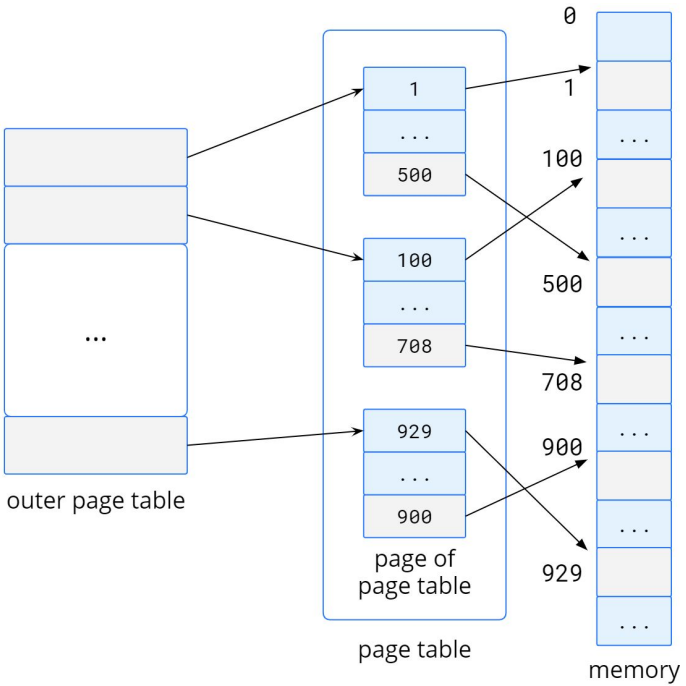
Paginación jerárquica

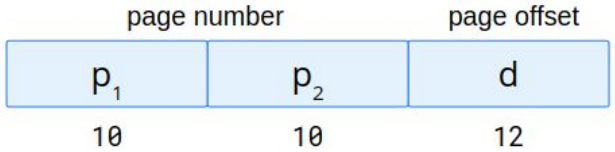
- En un sistema con espacio de direcciones lógicas de 2^{32} a 2^{64}, la tabla de páginas se vuelve excesivamente grande.
- Por ej. en un espacio de direcciones lógicas de 32 *bits*:
  - Si el tamaño de página es de 4KB (2^{12}),
  - La tabla de páginas tendrá más de 1 millón de entradas ($2^{20} = 2^{32}/2^{12}$).
  - Suponiendo que cada entrada consta de 4 *bytes*, cada proceso puede necesitar hasta 4 MB de espacio de direcciones físicas solo para la tabla de páginas.
- Una solución simple es dividir la tabla de páginas en partes más pequeñas.

Estructura de la tabla de páginas

Paginación jerárquica

- **`Paginación de dos niveles`**: la tabla de páginas también se pagina.
- Con dirección lógica de **`32-bit`** y página de **`4KB`** teníamos:
  - El número de página de **`20 bits`**.
  - El desplazamiento de página consta de **`12 bits`**.
- Como la tabla de páginas también se pagina, el número de página además se divide en:
  - El número de página de **`10 bits`**.
  - El desplazamiento de página de **`10 bits`**.
- La dirección lógica es la sig., donde **`p1`** es el índice en la TP externa, y **`p2`** es el desplazamiento dentro de la página de la TP interna.

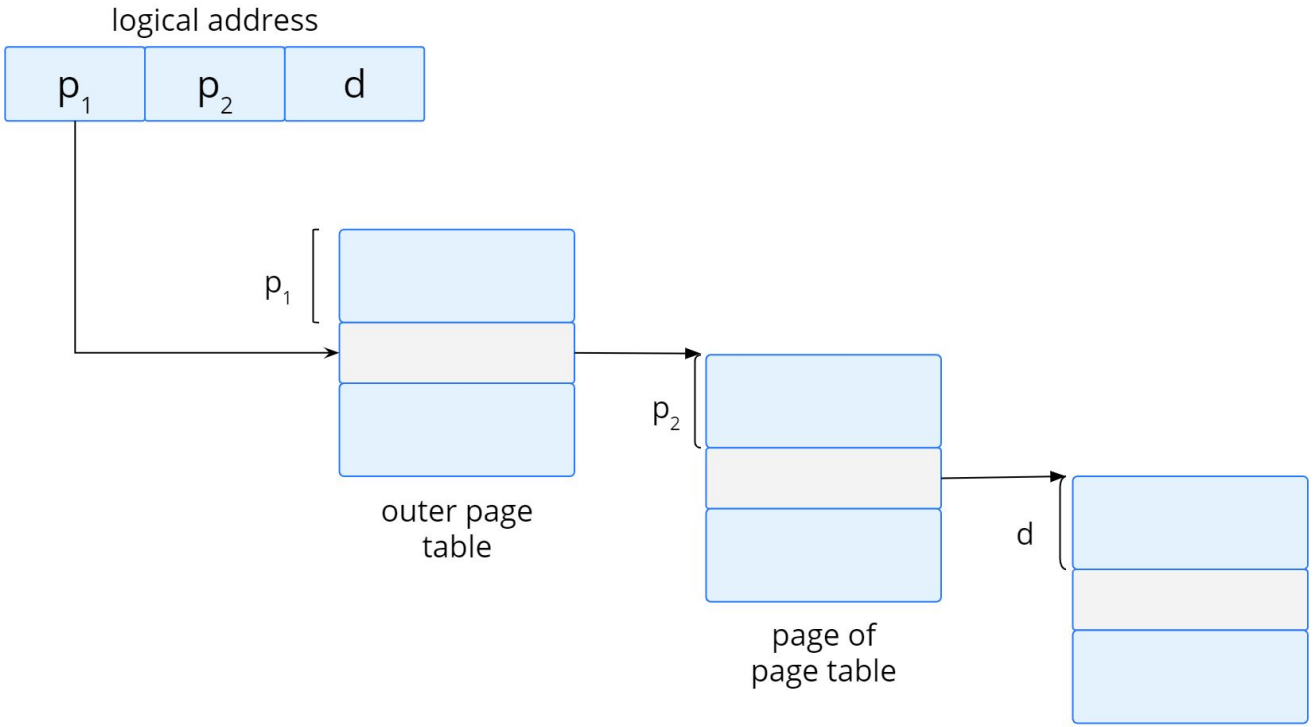




Estructura de la tabla de páginas

Paginación jerárquica

- *Forward-mapped page table*: la traducción de direcciones funciona desde la tabla de la página externa hacia adentro.



Estructura de la tabla de páginas

Paginación jerárquica

- Para un sistema con un espacio de direcciones lógicas de 64 *bits*, un esquema de paginación de dos niveles no es apropiado.
  - Si el tamaño de página es de 4 KB (2^{12}) → La tabla de páginas tiene hasta 2^{52} entradas.
  - Con un esquema de dos niveles las tablas de páginas internas pueden ser del tamaño de una página, o tener 2^{10} 4-*bytes* entradas:

outer page	inner page	offset
p_1	p_2	d
42	10	12

- La tabla de la página externa consta de 2^{42} entradas, o 2^{44} *bytes*.
- Se puede paginar la tabla de la página externa en un esquema de paginación de tres niveles, con 2^{10} entradas o 2^{12} *bytes*, aunque sigue siendo grande 2^{34} *bytes* (16 GB):
- El siguiente paso serían cuatro niveles, donde la tabla de la página externa del segundo nivel también está paginada, y así sucesivamente, lo cual puede terminar siendo inapropiado.

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

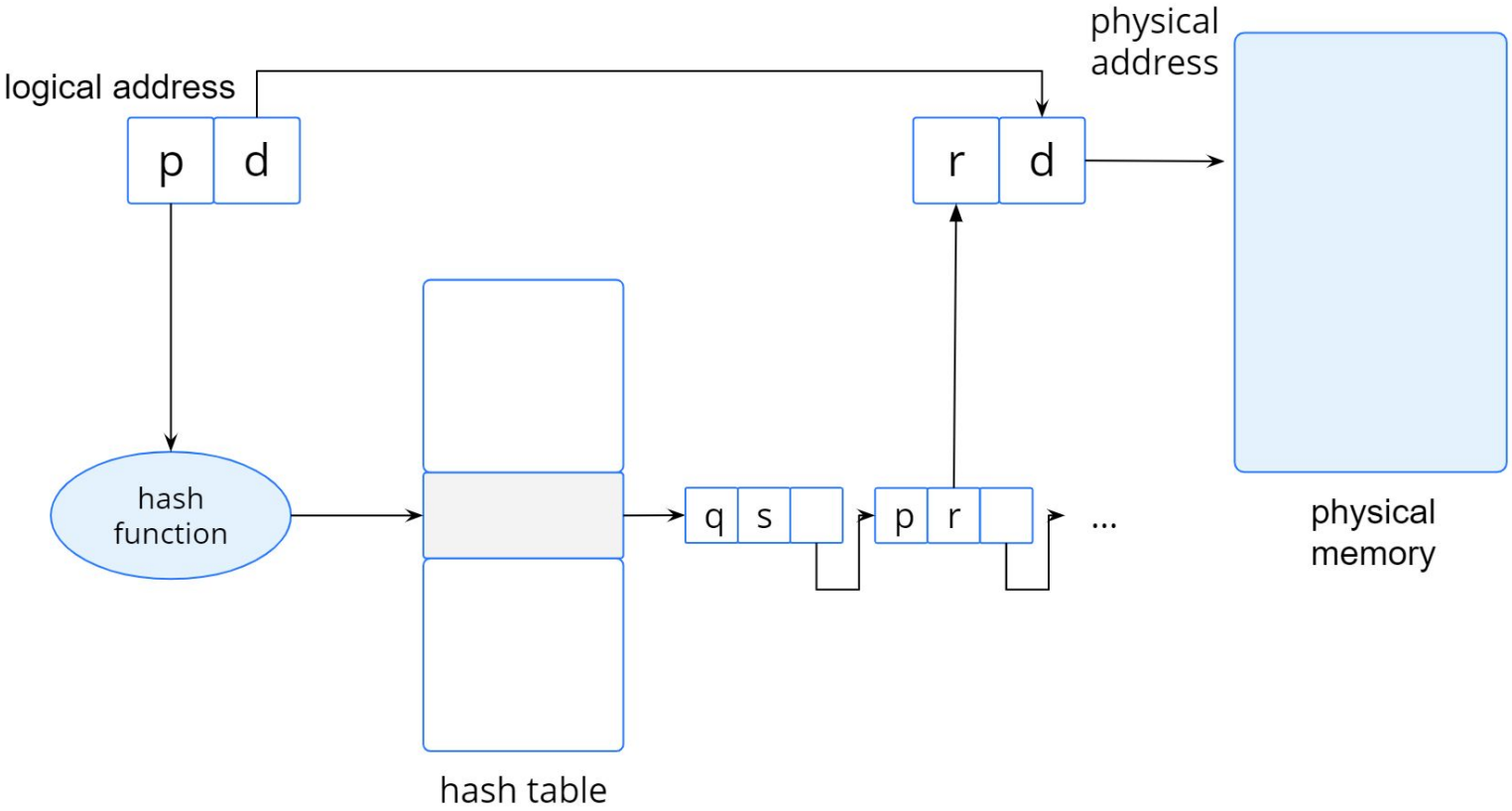
Estructura de la tabla de páginas

Tabla de páginas *hash*

- El valor *hash* es el número de página virtual.
- Cada entrada en la tabla *hash* tiene una lista enlazada de elementos que colisionan. Cada elemento consta de tres campos:
  - 1. El número de página virtual.
  - 2. El valor del marco de página asignado.
  - 3. Un puntero al siguiente elemento de la lista.
- El algoritmo funciona de la siguiente manera:
  - El número de página virtual en la dirección virtual se divide (*hashed*) en la tabla *hash*. El número de página virtual se compara con el campo 1 en el primer elemento de la lista enlazada.
  - Si hay una coincidencia, el marco de página correspondiente (campo 2) se utiliza para formar la dirección física deseada.
  - Si no hay coincidencia, se busca en las entradas posteriores de la lista enlazada un número de página virtual coincidente.

Estructura de la tabla de páginas

Tabla de páginas *hash*



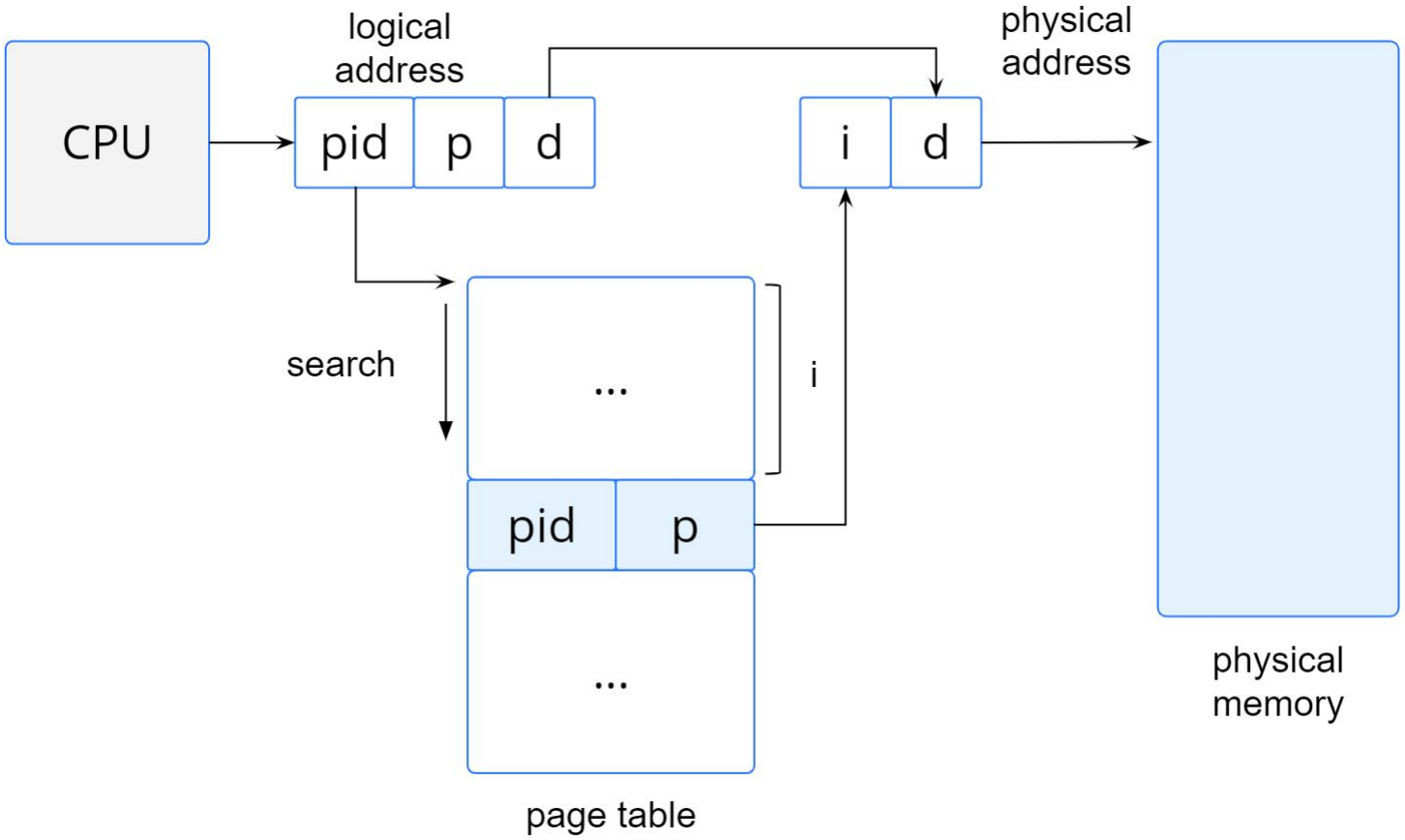
Estructura de la tabla de páginas

Tabla de páginas invertida (TPI)

- Una TPI tiene una entrada por cada página real (o marco) de memoria.
- Cada entrada consiste en la dirección virtual de la página almacenada en esa ubicación de memoria real, con información sobre el proceso que posee la página.
- Por lo tanto, solo hay una tabla de páginas en el sistema y solo tiene una entrada para cada página de memoria física.
- Las TPIs a menudo requieren que se almacene un identificador de espacio de direcciones en cada entrada de la tabla de páginas, ya que la tabla generalmente contiene varios espacios de direcciones diferentes que mapean la memoria física.
    - El *PID* por ej. puede asumir el rol de identificador de espacio de direcciones.
- El almacenamiento del identificador de espacio de direcciones garantiza que una página lógica para un proceso particular se asigne al marco de página físico correspondiente.
- Ejemplos de sistemas que utilizan tablas de páginas invertidas incluyen `Ultra SPARC` de `64 bits` y `Power PC`.

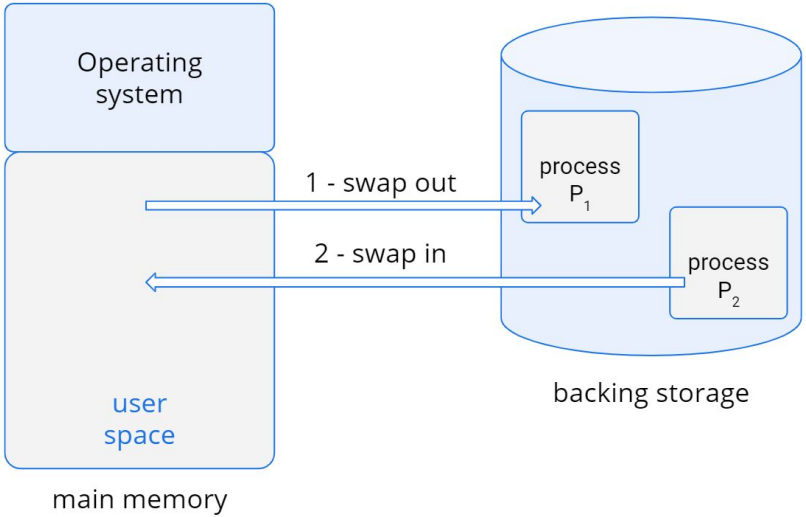
Estructura de la tabla de páginas

Tablas de páginas invertidas



Swapping

- Un proceso, o una parte de un proceso, puede sacarse temporalmente de la memoria a un almacenamiento de respaldo y luego volver a ésta para continuar con su ejecución.
- El *swapping* (intercambio) hace posible que el espacio total de direcciones físicas de todos los procesos exceda la memoria física real del sistema, aumentando así el grado de multiprogramación en un sistema.



Swapping

Estándar *swapping*

- Mueve procesos completos entre la memoria principal y un almacenamiento de respaldo.
- El almacenamiento debe ser lo suficientemente grande (y rápido) como para acomodar cualquier parte del proceso que deba almacenarse y recuperarse.
- Cuando un proceso o parte de él se intercambia al almacén de respaldo, las estructuras de datos asociadas con el proceso deben escribirse en el almacén de respaldo.
- Para un proceso multihilado, todas las estructuras por hilo también deben intercambiarse.
- El SO también debe mantener metadatos para los procesos que se han intercambiado, para que puedan restaurarse cuando se vuelven a intercambiar a memoria.
- La ventaja de este esquema es que permite que la memoria física se use por encima de su capacidad, de modo que el sistema pueda acomodar más procesos que la memoria física real.
- Los procesos inactivos son buenos candidatos al *swapping*;
    - La memoria que se haya asignado a estos procesos se puede dedicar a procesos activos.

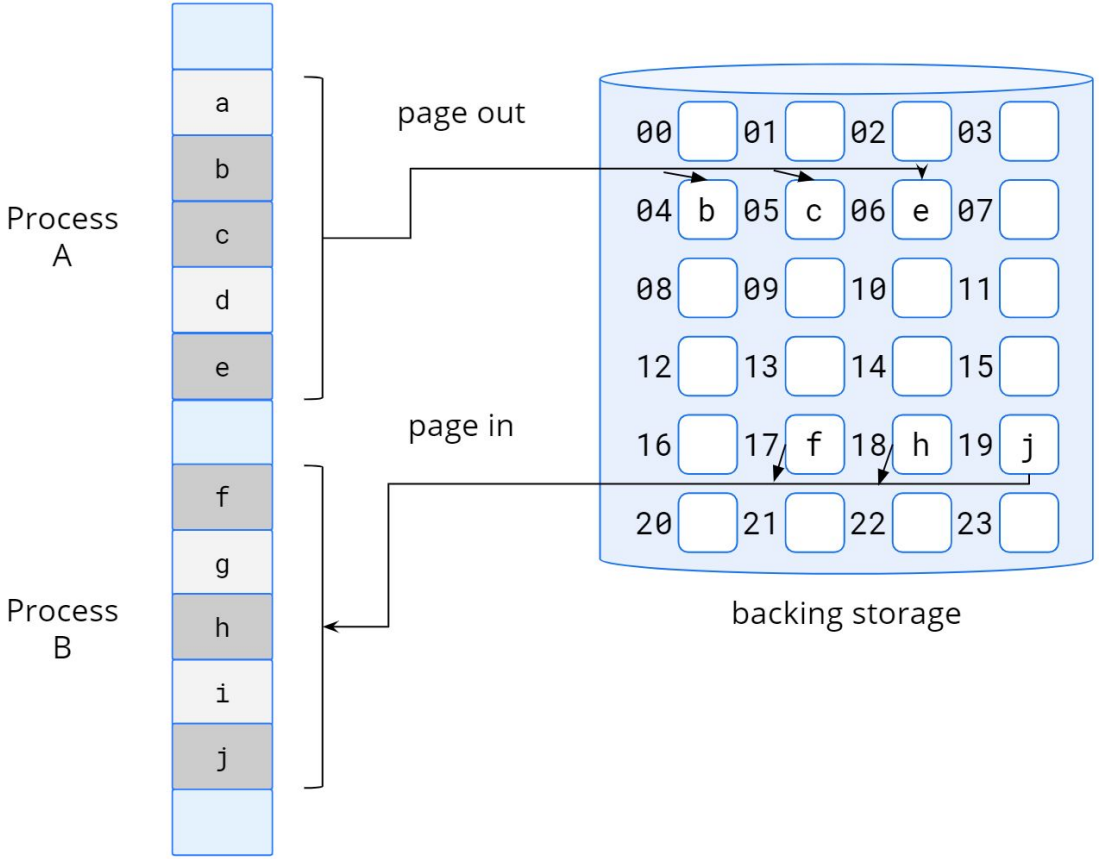
Swapping

Swapping con paginación

- El estándar *swapping* se usó en los sistemas `UNIX` tradicionales, aunque ya no se usa en los SO contemporáneos porque la cantidad de tiempo requerida para mover procesos completos entre la memoria y el almacén de respaldo es prohibitiva.
- La mayoría de los sistemas, incluidos `Linux` y `Windows`, ahora usan una variación de *swapping* en el que se pueden intercambiar páginas de un proceso, en lugar de un proceso completo.
- Esta estrategia también permite que la memoria física se use por encima de su capacidad sin incurrir en el costo de intercambiar procesos completos, ya que presumiblemente solo un pequeño número de páginas participará en el *swapping*.
- Una operación de *page out* mueve una página de la memoria al almacenamiento secundario; el proceso inverso se conoce como *page in*.
- Como se verá, *swapping* con paginación funciona bien con memoria virtual.

Swapping

Swapping con paginación



Swapping

Swapping en sistemas móviles

- Los sistemas móviles generalmente no admiten *swapping* de ninguna manera.
- Los dispositivos móviles usan memoria *flash* en lugar de discos duros para almacenamiento no volátil.
- La restricción de espacio es una de las razones por las cuales los diseñadores de SOs móviles evitan el *swapping*.
- Otras razones incluyen el número limitado de escrituras que la memoria *flash* puede tolerar antes de que no sea confiable y el bajo rendimiento (*throughput*) entre la memoria principal y la memoria *flash*.
- En lugar de utilizar *swapping*, cuando la memoria libre cae por debajo de un cierto umbral, `iOS` de `Apple` solicita a las aplicaciones que renuncien voluntariamente a la memoria asignada.
    - Los datos de solo lectura (como el código) se eliminan de la memoria principal y luego se vuelven a cargar de la memoria *flash* si es necesario. Los datos que se han modificado (como la pila) nunca se eliminan.
    - Sin embargo, el SO puede finalizar cualquier aplicación que no libere suficiente memoria.

Swapping

Swapping en sistemas móviles

- `Android` adopta una estrategia similar a la utilizada por `iOS`.
  - Puede terminar un proceso si no hay suficiente memoria libre disponible.
  - Sin embargo, antes de finalizar un proceso, `Android` escribe el estado de su aplicación en la memoria *flash* para que pueda reiniciarse rápidamente.
- Debido a estas restricciones, los desarrolladores de sistemas móviles deben asignar y liberar cuidadosamente la memoria para garantizar que sus aplicaciones no utilicen demasiada memoria o sufran pérdidas.

Ejemplos arquitecturas

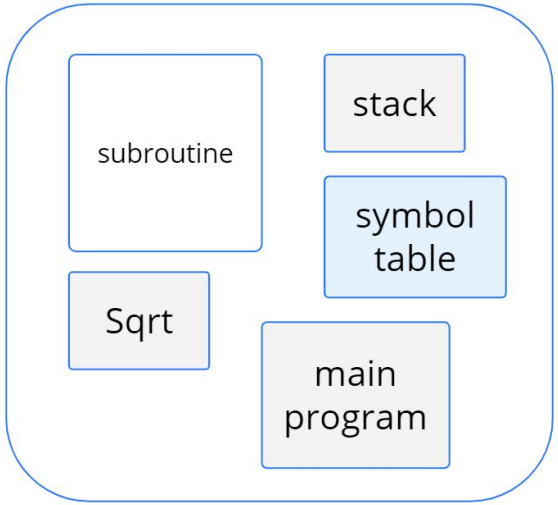
- Ejemplo: arquitecturas `Intel` de `32` y `64` ***`bits`***.
- Ejemplo: arquitectura `ARMv8`.

Segmentación

Método básico

- *Corresponde a la versión **9na** del libro "Operating System Concepts" Silberschatz, A., Gagne G., y Galvin, P.B.*

- Los programadores prefieren ver a la memoria como una colección de segmentos de tamaño variable sin ningún orden.
- Un programador considera un programa principal con un conjunto de métodos, procedimientos o funciones.
- Cada uno de estos módulos o elementos de datos se conoce por su nombre, sin importar qué direcciones en memoria ocupen.
- Los segmentos varían en longitud, y la longitud de cada uno está intrínsecamente definida por su propósito en el programa.



- Los elementos dentro de un segmento se identifican por su desplazamiento desde el principio del segmento: la primera instrucción del programa, la quinta instrucción de `Sqrt()`, y así sucesivamente.

Segmentación

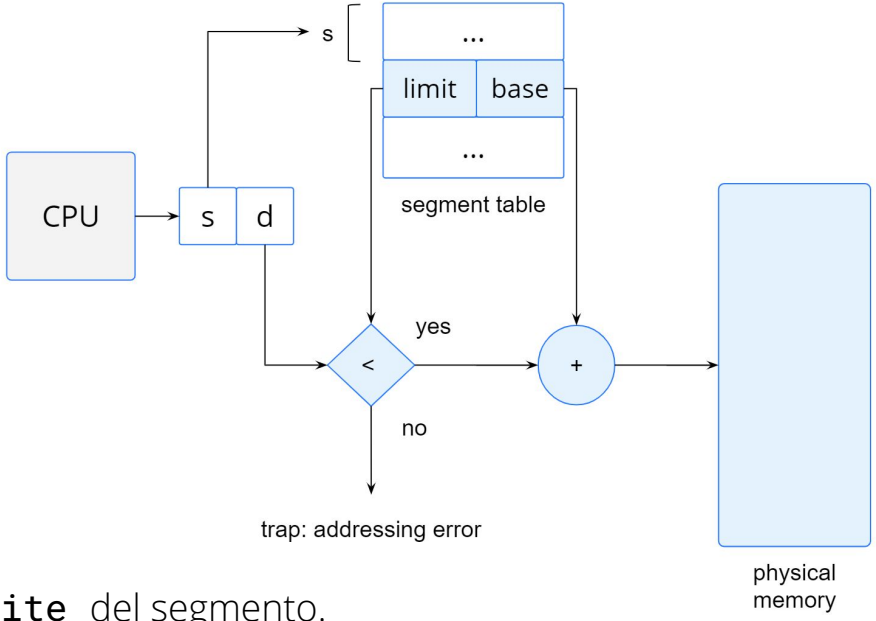
Método básico

- La segmentación es un esquema de administración de memoria donde un espacio de direcciones lógicas es una colección de segmentos.
- Cada segmento tiene un nombre (número de segmento) y una longitud (desplazamiento).
- Una dirección lógica consta de:
  - `<número de segmento, desplazamiento>.`
- Por lo general, el compilador crea segmentos que reflejan el programa de entrada; un compilador de `C` podría crear segmentos separados para:
  - 1. El código.
  - 2. Variables globales.
  - 3. El *heap*, desde el cual se asigna la memoria.
  - 4. Las pilas utilizadas por cada hilo.
  - 5. La biblioteca estándar `C`.

Segmentación

Hardware de segmentación

- Cada entrada en la tabla de segmentos tiene la base del segmento y un límite de segmento.
  - **Base**: contiene la dirección física inicial donde reside en memoria.
  - **`Límite`**: especifica la longitud.
- Dirección lógica:
  - **`s`**: número de segmento.
  - **`d`**: desplazamiento en ese segmento.
- El número de segmento se utiliza como índice de la tabla de segmentos.



- El desplazamiento **`d`** debe estar entre **`0`** y el **`límite`** del segmento.
  - Si no es así, *trap* al SO. (intento de direccionamiento lógico más allá del final del segmento).
  - Si es legal, se agrega a la base del segmento para generar la dirección en la memoria física del *byte* deseado.