

# Administración de la memoria

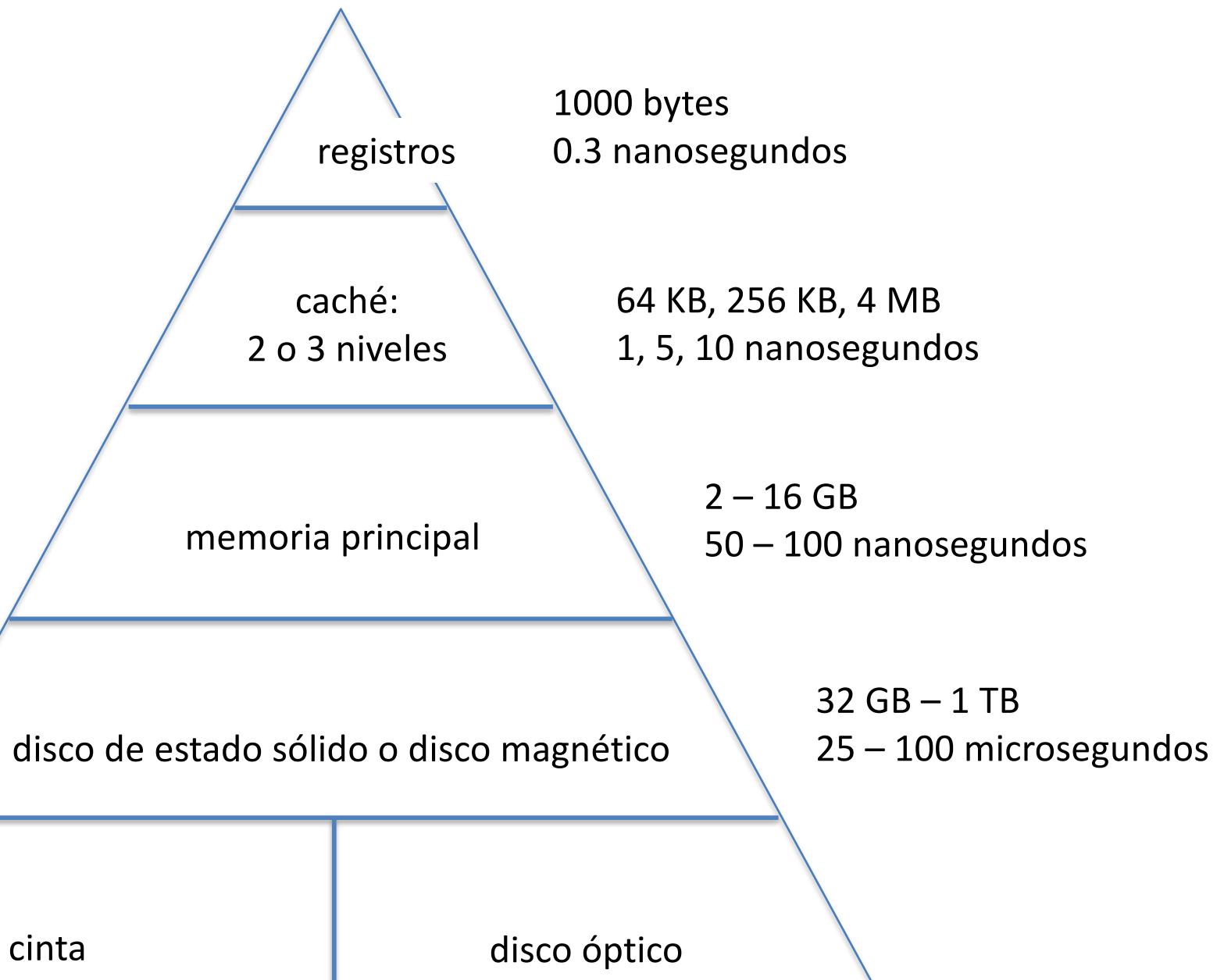
---

Sistemas Operativos – IIC2333

No es posible (aún) proveer una memoria no volátil, infinitamente grande y rápida, y además barata

En cambio, los computadores ofrecen una jerarquía de memoria:

- unos pocos megabytes de memoria caché muy rápida, pero cara y volátil
- ... más unos pocos gigabytes de memoria principal medianamente rápida, un poco más barata y también volátil
- ... y más unos pocos terabytes de almacenamiento en disco magnético o de estado sólido lento, barato y no volátil



El trabajo del sistema operativo es ofrecer un modelo abstracto útil de esta jerarquía y administrarlo:

- ofrecer al programador tanta memoria como esté disponible en la tecnología más barata
- ... pero a la velocidad de acceso de la memoria más rápida

Esta jerarquía está basada en el **principio de localidad** en el funcionamiento de los programas:

**los programas accesan una porción relativamente pequeña de su espacio de direcciones en un momento cualquiera del tiempo**

Es decir, un programa no accesa todo su código o todos sus datos al mismo tiempo con igual probabilidad:

- esto hace posible que la mayoría de los accesos a memoria sean rápidos y al mismo tiempo tengamos una memoria grande

**Localidad temporal:** si se hace referencia a un ítem —una instrucción o un dato— probablemente se hará referencia a ese mismo ítem pronto

**Localidad espacial:** si se hace referencia a un ítem, probablemente pronto se hará referencia a ítems cuyas direcciones están (numéricamente) cerca

Esta localidad se da naturalmente en los programas:

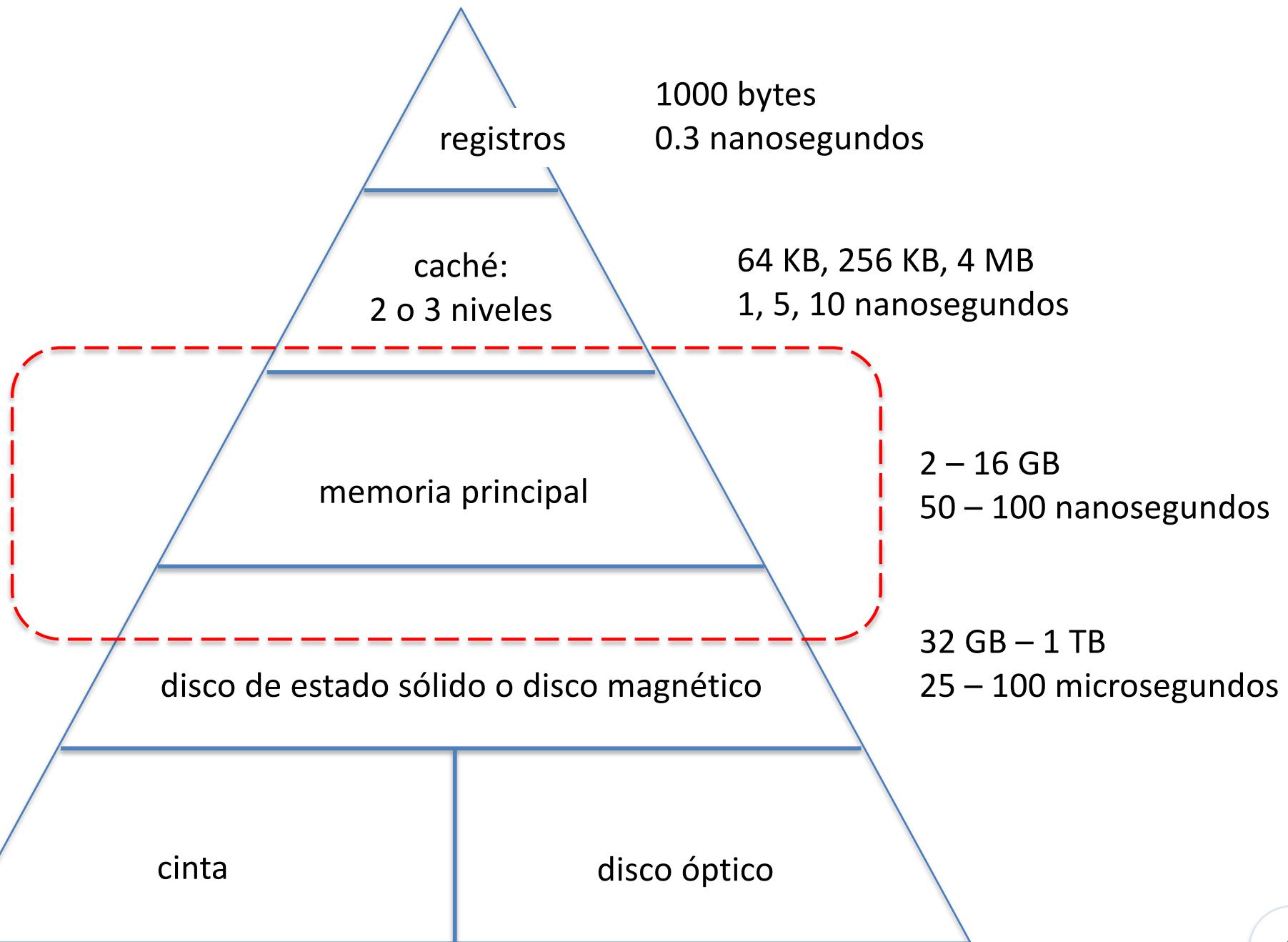
- *loops*, cuyas instrucciones y datos son usados repetidamente —localidad temporal
- ejecución secuencial de instrucciones —localidad espacial
- accesos secuenciales a los elementos de un arreglo —localidad espacial

La parte del sistema operativo que administra (parte de) la jerarquía de memoria se llama el administrador de memoria:

- seguir la pista de qué partes de la memoria están siendo usadas
- asignar memoria a los procesos cuando estos la necesitan
- ... y desasignarla cuando han terminado

¿Qué parte de la jerarquía?

- básicamente, la memoria principal (pero apoyándose en el disco)



Una posibilidad es que cada programa “vea” simplemente la memoria física (en este caso, el modelo abstracto es que ... no hay ninguno)

Igual, hay algunas posibilidades para la organización de la memoria:

- todo el sistema operativo en ROM —sistemas embebidos
- solo la BIOS en ROM —primeros computadores personales

No es posible tener dos (o más) programas corriendo en memoria al mismo tiempo:

- si un programa escribe un valor en la dirección de memoria 2000, esto borra cualquier valor que el otro programa pueda haber estado guardando allí

... especialmente si no están relacionados entre ellos:

- si están relacionados, es posible usar múltiples *threads*

La noción de un **espacio de direcciones** —una forma de memoria abstracta— permite que haya varios programas abiertos al mismo tiempo:

- es el conjunto de direcciones que un proceso puede usar
- cada proceso tiene su propio espacio de direcciones, independiente de la direcciones que pertenecen a otros procesos

P.ej., registros (especiales de hardware) *base* y *límite*:

- el programa se carga en celdas consecutivas de memoria dondequiera que haya suficiente espacio
- el registro *base* se carga con la dirección física de la primera celda ocupada por el programa
  - ... el registro *limit* se carga con el largo del programa
- cada vez que el programa hace referencia a una dirección de memoria, el hardware de la CPU automáticamente suma el valor del registro *base* a esta dirección antes de enviarla por el bus
  - ... y también verifica que la dirección así generada no sobrepase el valor en el registro *limit*

Aunque el tamaño de la memoria crece rápidamente

... el tamaño de los programas crece aún más rápidamente

Queremos correr programas que son demasiado grandes para caber en la memoria

... y queremos sistemas que permitan múltiples programas corriendo simultáneamente, tales que todos juntos exceden la memoria

En los comienzos de la computación, los programadores pasaban mucho tiempo tratando de hacer caber los programas en la memoria, que era poca y cara

La solución era usar memoria secundaria (p.ej., disco):

- *overlays*
- de responsabilidad del programador

## J. Fotheringham, 1961: automatización del proceso de *overlays* -> memoria virtual:

- cada programa tiene su propio espacio de direcciones
  - ... dividido en **páginas** —rango contiguo de direcciones
- usada hasta nuestros días
  - ... hoy, principalmente con el propósito de permitir compartir una única memoria principal entre múltiples procesos
  - ... proporcionando protección de memoria entre estos procesos y el sistema operativo

La idea está basada en separar los conceptos de **espacio de direcciones** y **localidades de memoria**

P.ej., un computador con direcciones de 16 bits y memoria de 4096 palabras:

- 16 bits pueden direccionar 65536 palabras: 0 – 65535 → este es el espacio de direcciones
- podemos decirle al computador que haga corresponder las direcciones 4096 a 8191 referenciadas en el programa con las direcciones 0 a 4095 de la memoria

Definimos una **correspondencia**, o *mapping*, desde el espacio de direcciones a las localidades reales de memoria

direcciones  
virtuales

65535



espacio de direcciones  
(direcciones virtuales)

Un *mapping* en que las direcciones virtuales 4096 a 8191 se hacen corresponder a las direcciones físicas 0 a 4095 de la memoria principal

*mapping*

direcciones  
físicas

4095

0

localidades de memoria  
en la memoria principal

¿Qué pasa si el programa hace un *jump* a una dirección entre 8192 y 12287?

Sin memoria virtual:

*“memoria referenciada inexistente”*

Con memoria virtual:

1. El contenido de la memoria principal se guarda en disco
2. Se buscan en el disco las palabras (con direcciones) 8192 a 12287
3. Las palabras 8192 a 12287 son cargadas en la memoria
4. El mapa de direcciones se cambia: las direcciones 8192 a 12287 se hacen corresponder a las localidades de memoria 0 a 4095
5. La ejecución continúa como si nada hubiera pasado

La técnica se llama **paginación**

... y los bloques de programa (o de datos) leídos desde el disco se llaman **páginas**

El mecanismo de paginación se dice que es **transparente**:

- ( excepto para quienes escriben sistemas operativos )

Hablamos del *espacio virtual de direcciones*:

- las direcciones a las que el programa puede hacer referencia

... del *espacio físico de direcciones*:

- las localidades de memoria reales, físicamente disponibles

... y del *mapa de memoria* o de la **tabla de páginas**:

- especifica para cada dirección virtual cuál es la dirección física correspondiente
- se almacena en memoria, a partir de la dirección almacenada en el **registro de la tabla de página**, o *page table register*

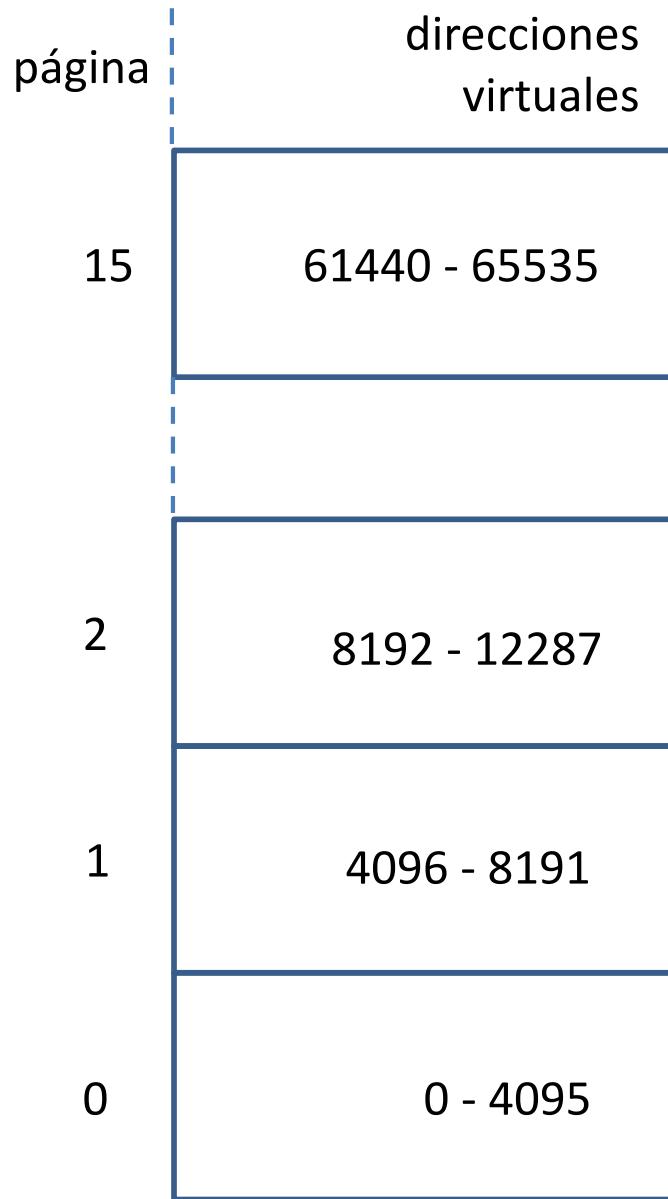
Los programas son escritos como si hubiera suficiente memoria para todo el espacio virtual de direcciones:

- un programa puede cargar desde, o almacenar en, cualquier palabra en el espacio virtual
  - ... o saltar a cualquier instrucción ubicada en cualquier parte dentro del espacio virtual
- el programador puede escribir el programa sin ni siquiera sospechar que existe la memoria virtual
- el computador se ve como si tuviera una gran memoria
- esta simulación de una gran memoria principal mediante paginación no es detectable por el programa (excepto si corre *tests* que midan el tiempo)

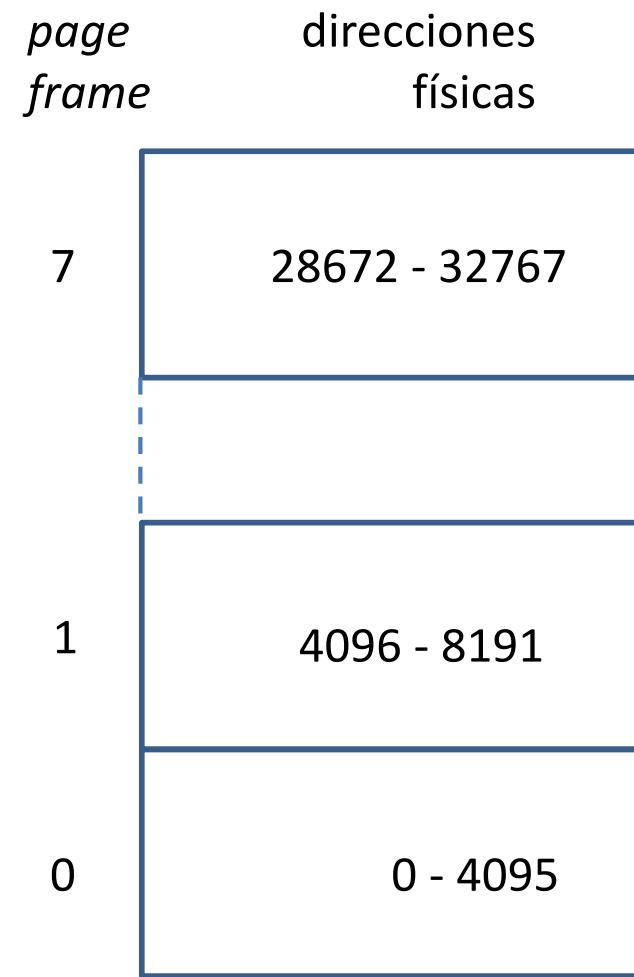
El espacio virtual de direcciones se divide en páginas de un mismo tamaño (una potencia de 2), p.ej., 512 bytes a 64 KB

El espacio físico —la memoria principal— se divide en *page frames* del mismo tamaño que las páginas:

- cada *page frame* de la memoria principal puede almacenar exactamente una página
- en la diap. #18, la memoria principal contiene solo un *page frame*
- en la próxima diap., la memoria principal contiene 8 *page frames*
- en la realidad, normalmente contiene miles (y, por otra parte, puede haber millones de páginas virtuales)



espacio virtual de direcciones (de 32 bits) dividido en páginas de 4 KB

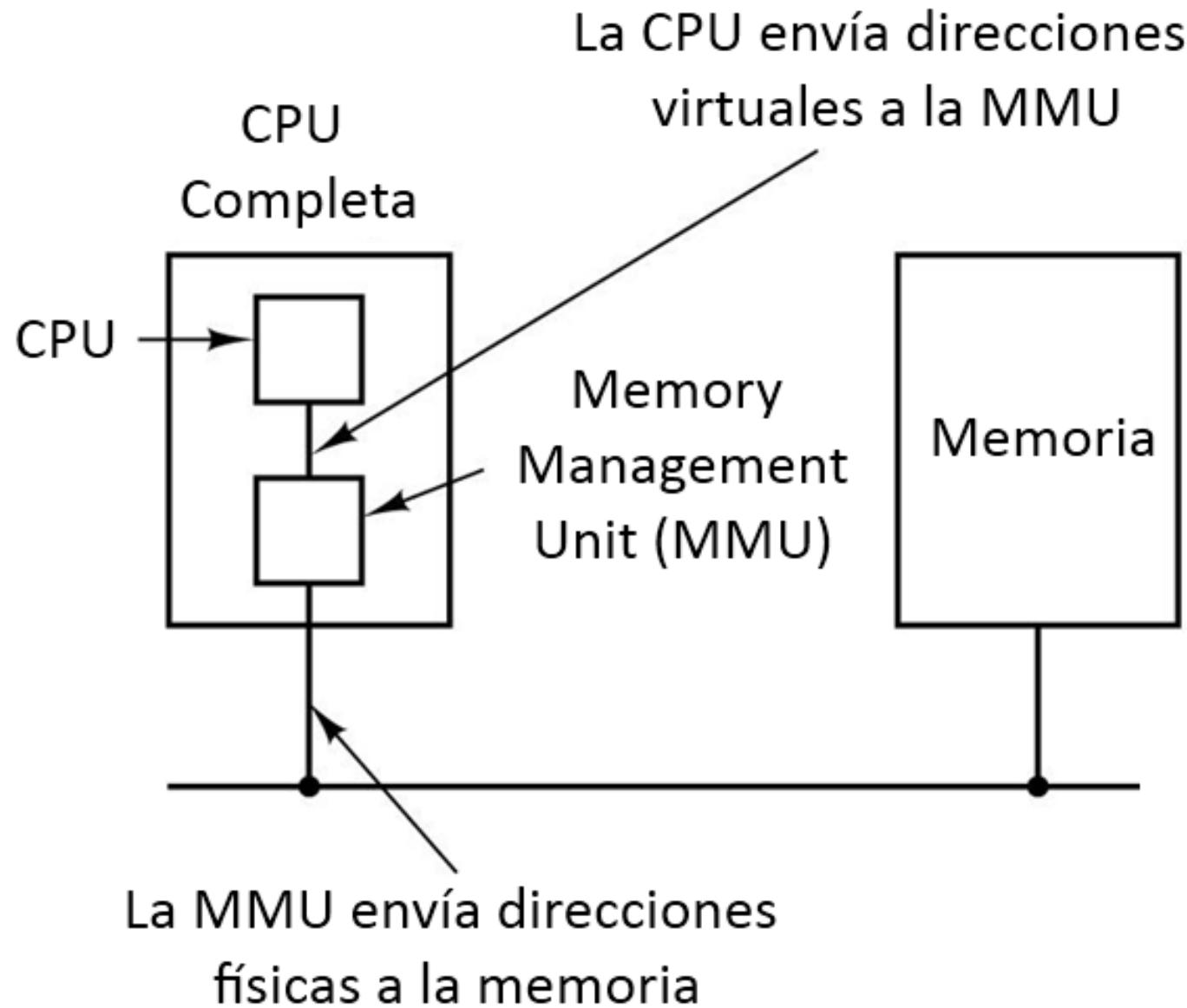


memoria principal de 32 KB  
dividida en 8 *page frames* de 4 KB

La memoria virtual es implementada mediante una *tabla de páginas*, que tiene tantas entradas (filas) como páginas hay en el espacio virtual

La **unidad de manejo de memoria (MMU)** convierte las direcciones virtuales en direcciones físicas —**traducción de direcciones**:

- puede estar en el chip de la CPU o en un chip aparte
- tiene un registro de *input* (p.ej., de 32 bits en el caso de la diap. anterior)
- ... y un registro de *output* (p.ej., de 15 bits en el caso de la diap. anterior)



Una dirección virtual de 32 bits es dividida en dos partes:

- número de página de 20 bits, como índice a la tabla de páginas
- *offset* de 12 bits dentro de la página (páginas de 4K)

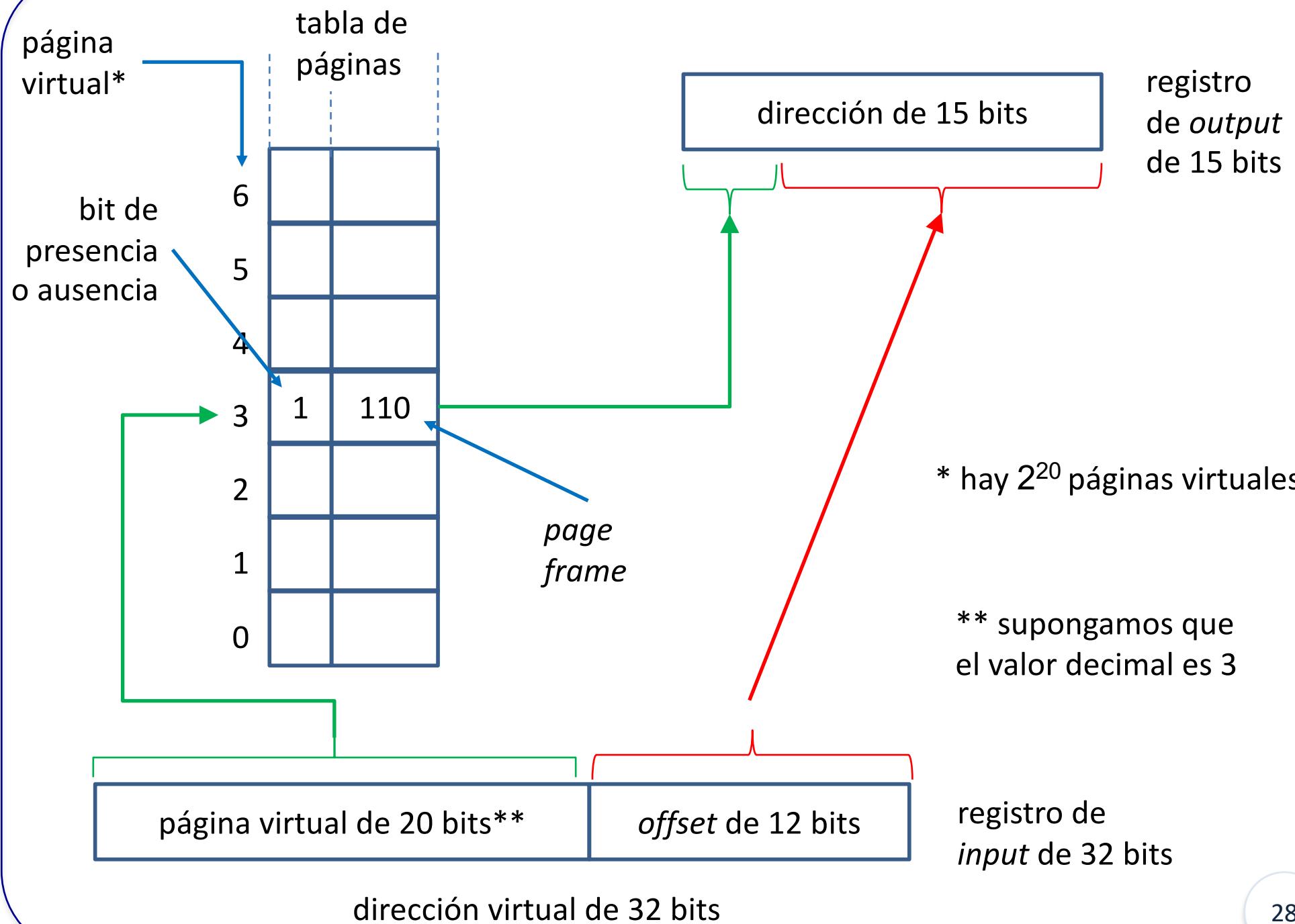
Pero, ¿está la página en memoria?

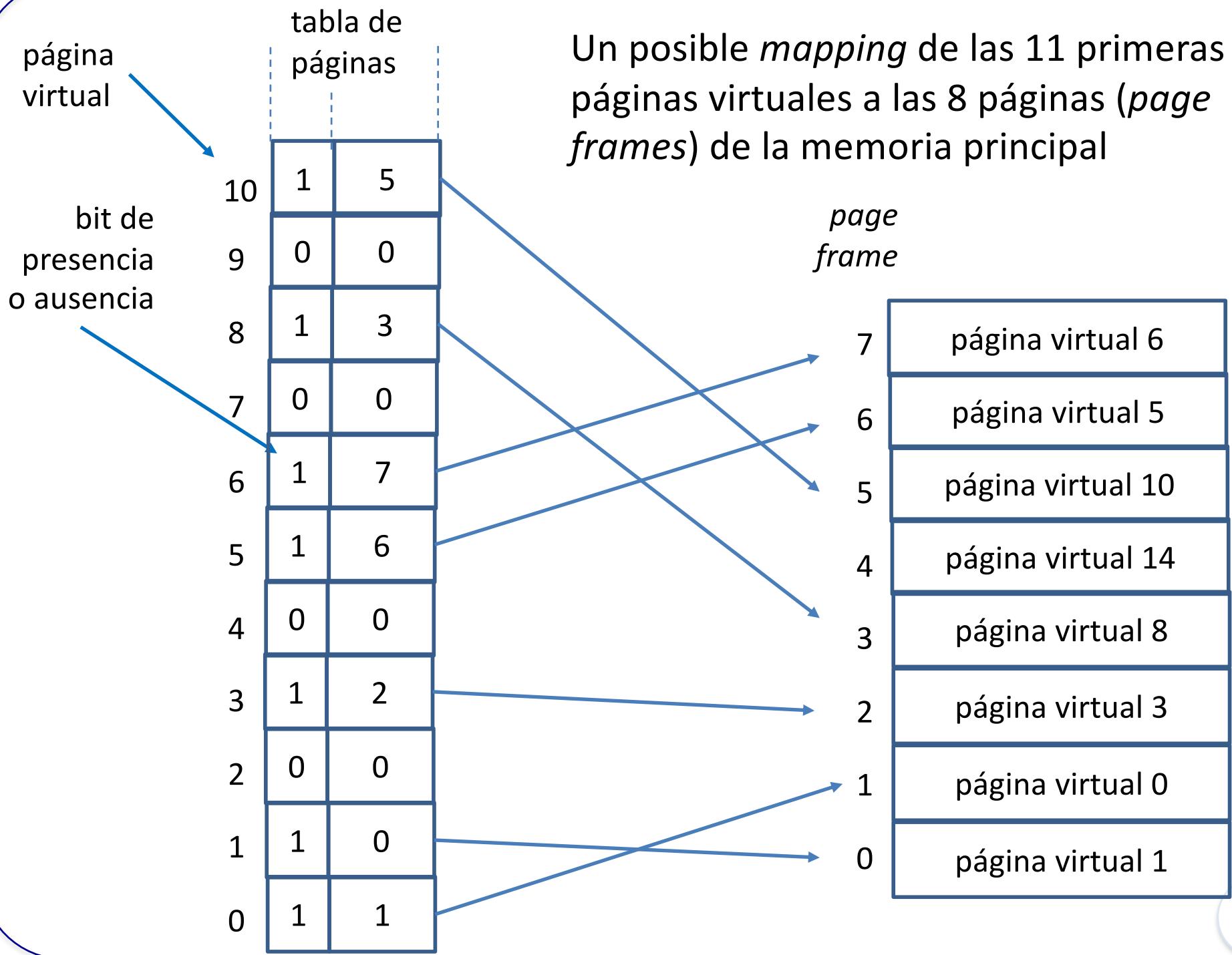
- hay  $2^{20}$  páginas virtuales y solo 8 páginas físicas (*page frames*)
- **bit de presencia(1)/ausencia(0) —o *bit de validación*—** en cada entrada en la tabla

Si está (un “*hit*”), se usa el número del *page frame* almacenado en esa entrada en la tabla:

- este número (3 bits, si hay 8 *page frames*) se une a los 12 bits del *offset* para formar la dirección que se envía a la memoria (en realidad, a la cache)

Si no está → ***page fault***





## ¿Qué se hace cuando ocurre un *page fault*?

- el sistema operativo debe traer desde el disco la página requerida
  - ... ingresar su nueva dirección física (número de *page frame*) a la tabla de páginas
  - ... y repetir la ejecución de la instrucción que causó el *page fault*

→ es posible iniciar la ejecución de un programa aun cuando nada del programa está en la memoria principal —**paginación por demanda**:

- hay que inicializar apropiadamente la tabla de páginas
- la CPU trata de leer la primera instrucción → *page fault* → la página que contiene esa instrucción es cargada en la memoria e ingresada a la tabla de páginas
- si la instrucción contiene dos direcciones en páginas diferentes, y diferentes a la página en que está la instrucción → dos *page faults* adicionales → dos nuevas páginas son traídas desde el disco antes de que la instrucción pueda finalmente ser ejecutada
- la próxima instrucción puede producir nuevos *page faults*, etc.

La paginación por demanda —se trae una página a la memoria solo cuando la página es requerida (y no por adelantado)— presenta un problema:

- si la memoria del computador es compartida por múltiples procesos (como ya veremos), y los procesos son sacados de la CPU después de ejecutar durante un cierto tiempo (p.ej., 100 ms), cada programa va a ser reanudado muchas veces
- como la tabla de páginas es única para cada programa, la pregunta de si usar o no este sistema de paginación es crítica

Recordemos el *principio de localidad*:

- los programas no referencian su espacio de direcciones uniformemente
- las referencias tienden a ser a un grupo pequeño de páginas

***Working set:***

- en todo instante de tiempo, existe un conjunto de todas las páginas usadas por las referencias de memoria más recientes
- este conjunto cambia de a poco a lo largo del tiempo
- → se puede adivinar cuáles páginas van a ser requeridas cuando el programa sea reanudado
  - ... a partir de su *working set* cuando fue suspendido la última vez

## Necesidad de reemplazo de páginas:

- idealmente, el *working set* se mantiene en memoria principal para reducir el número de *page faults*
- el *working set* debe ser “descubierto” por el sistema operativo a medida que el programa es ejecutado
- si el programa hace referencia a una página que no está en memoria, hay que ir a buscarla al disco
  - ... y cambiarla por alguna otra página que es enviada de vuelta al disco
- → se necesita un algoritmo para decidir cuál página sacar

Los sistemas operativos predicen cuál de las páginas en memoria es la menos “útil” :

- predicen cuándo va a ocurrir la próxima referencia a cada página  
... y sacan la página cuya próxima referencia está más adelante en el futuro  
(una página que no va a ser requerida en mucho tiempo)

P.ej.,

- LRU —la página que se usó por última vez hace más tiempo— aproximado
- FIFO —la página que se trajo a memoria hace más tiempo— aproximado

¿Qué pasa si una página en memoria es escrita por el programa?

La página se copia —es decir, se actualiza— en el disco solo cuando es reemplazada en la memoria (es decir, cuando se la saca de su *page frame* para poner allí otra página) —esquema *write-back*:

- para saber si una página necesita ser copiada en el disco al momento de reemplazarla —es decir, si fue escrita mientras estaba en memoria— se agrega un ***dirty bit*** de la tabla de páginas
  - ... el *dirty bit* se pone en 1 cuando cualquier palabra de la página es escrita
- si el sistema operativo reemplaza una página cuyo *dirty bit* está en 1
  - ... entonces la página tiene que ser copiada completamente en el disco antes de que su *page frame* en memoria sea entregado a otra página
- ( la escritura en disco puede tomar hasta millones de ciclos del procesador
  - el esquema *write-through* con *buffer* de escritura, que se puede usar en el caso de la cache, es impráctico para memoria virtual )

Una fila (*entry*) típica en la tabla de páginas contiene lo siguiente:

- número de *page frame*
- bit de presencia/ausencia
- bit (o bits) de protección: leer, escribir, ejecutar la página
- *dirty bit*: si la página ha sido escrita, es decir, modificada
- bit de referenciación: si la página ha sido referenciada recientemente, p.ej., leída
- bit para deshabilitar el caching, en caso de páginas que corresponden a registros de dispositivos de I/O (con *memory-mapped I/O*)

La tabla de páginas se mantiene en memoria

→ cada acceso a la memoria que hace un programa puede tomar el doble de tiempo:

- un acceso a la memoria, específicamente a la tabla de páginas, para obtener la dirección física —la traducción
- un segundo acceso, ahora a esa dirección física, para obtener el dato propiamente tal

...

...

Para mejorar, podemos aprovechar la localidad temporal/espacial de las referencias a la tabla:

- la mayoría de los programas tiende a hacer un gran número de referencias a una cantidad pequeña de páginas —solo una fracción pequeña de las filas de la tabla de páginas son muy leídas
- cuando se hace una traducción de un número de página virtual, probablemente se va a necesitar de nuevo pronto

...

...

→ Incluimos una cache especial, dentro de la MMU, que lleva un registro de las traducciones usadas recientemente —**TLB**, o *translation-lookaside buffer*:

- cada fila contiene información acerca de una página → *tag* con el número de la página virtual, número de la página física (o *page frame*), bit de validez y *dirty* bit
- 16 – 512 filas

valid	pág. virtual	modified	protection	page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

### Ejemplo de TLB:

Suponemos que el programa está ejecutando un *loop* que involucra a las páginas virtuales 19, 20 y 21

- ... los datos, p.ej., un arreglo, están en la páginas 129 y 130

- ... la página 140 contiene los índices usados en el manejo del arreglo

- ... y el *stack* está en las páginas 860 y 861

La CPU produce direcciones virtuales

Cada dirección se busca primero en el TLB:

- *hit* → el número de la página física (*page frame*) se usa para formar la dirección

... es un write → el *dirty* bit se pone en 1

- *miss* →

...

- ...
- *miss* →
  - ... la página está en memoria (caso más frecuente) → se carga la línea correspondiente desde la tabla de páginas al TLB y se vuelve a procesar la dirección virtual original
  - ... la página no está en memoria (*page fault*, caso menos frecuente) → se llama al sistema operativo, usando una excepción
  - ... en ambos casos, hay que actualizar el TLB, remplazando una de sus líneas → como la línea que sale contiene el *dirty bit* de una página, hay que actualizar este *dirty bit* en la tabla de páginas, normalmente usando *write-back*

La memoria virtual y la cache funcionan juntas como una jerarquía:

- un dato no puede estar en la cache a menos que esté presente en la memoria principal

El sistema operativo mantiene esta jerarquía:

- elimina el contenido de la página que corresponda de la cache cuando decide migrar esa página al disco  
... y actualiza la tabla de páginas y el TLB → cualquier intento de tener acceso a algún dato de la página migrada producirá un *page fault*

...

...

En el mejor caso:

- una dirección virtual es traducida por el TLB y enviada a la cache, donde el dato que se busca es encontrado, leído y finalmente enviado al procesador

En el peor caso:

- una dirección virtual no está en el TLB, ni en la tabla de páginas, ni en la cache

TLB	Tabla de páginas	Cache	¿Es posible? ¿Bajo qué circunstancia?
hit	hit	miss	Possible, pero ...
miss	hit	hit	
miss	hit	miss	
miss	miss	miss	
hit	miss	miss	Impossible: ...
hit	miss	hit	Impossible: ...
miss	miss	hit	Impossible: ...

## Revisemos

Exponer la memoria física a los procesos no es una buena idea:

- si los programas de los usuarios pueden tener acceso a cualquier celda de la memoria, fácilmente pueden dañar el sistema operativo (accidentalmente o no)
- es difícil tener múltiples programas corriendo al mismo tiempo (o turnando turnos, si hay una sola CPU), algo que, p.ej., es común en computadores personales

La noción de un espacio de direcciones —una abstracción de la memoria— ofrece protección y relocalización

Cuando ocurre un *page fault*, el sistema operativo tiene que elegir una página para sacarla de la memoria,

... para poder poner ahí la nueva página traída desde el disco:

- si la página que se va a sacar fue modificada mientras estaba en memoria, entonces hay que reescribirla en el disco primero (para que la copia en disco esté actualizada)
- si la página no ha sido modificada, entonces no es necesario reescribirla

El desempeño del sistema es mucho mejor si la página que se saca no es una que esté siendo muy usada:

- de lo contrario, seguramente va a haber que traerla de vuelta luego, produciendo *overhead* adicional

algoritmo	comentario
Óptimo	No es implementable; útil como <i>benchmark</i>
NRU	Aproximación a LRU
FIFO	Podría sacar páginas importantes
Segunda oportunidad	Mejora sobre FIFO
Reloj	Realista
LRU	Excelente, pero difícil de implementar exactamente
NFU	Aproximación a LRU
Envejecimiento	Eficiente; buena aproximación a LRU
<i>Working set</i>	Caro de implementar
<i>WSClock</i>	Bueno y eficiente

## Óptimo

Si cuando ocurre un *page fault* pudiéramos etiquetar cada página con el número de instrucciones que van a ser ejecutadas antes de que la página sea referenciada

... entonces sacamos la página con el número más grande:

- ... es decir, de todas las páginas, sacamos la que vamos a tener que traer de vuelta más lejos en el futuro

Solo que no es implementable:

- el sistema operativo no tiene cómo saber cuándo cada página va a ser referenciada nuevamente
- ( situación similar al algoritmo de scheduling *shortest-job-first* )

## **NRU** (*not recently used*)

Emplea los bits de status **R** y **M** asociados con cada página en la tabla de páginas:

- **R** es puesto en 1 cada vez que la página es referenciada (leída o escrita)  
... y es vuelto a 0 periódicamente, p.ej., en cada interrupción del reloj (algunos milisegundos)
- **M** es puesto en 1 cuando la página es escrita (modificada)
- estos bits son actualizados en cada referencia a memoria  
... por lo que es esencial que los actualice el hardware

Al ocurrir un *page fault*, el sistema operativo revisa todas las páginas y las divide en cuatro categorías:

clase 0: no referenciada ( $R=0$ ), no modificada ( $M=0$ )

clase 1: no referenciada ( $R=0$ ), modificada ( $M=1$ )

clase 2: referenciada ( $R=1$ ), no modificada ( $M=0$ )

clase 3: referenciada ( $R=1$ ), modificada ( $M=1$ )

El algoritmo NRU saca una página al azar de la clase 0

... o, sin no hay páginas en esta clase, entonces de la clase 1

... y así sucesivamente

Fácil de entender, relativamente eficiente de implementar, desempeño adecuado

## FIFO (*first-in, first-out*)

El sistema operativo mantiene una lista ligada de todas las páginas actualmente en memoria:

- la página traída más recientemente está al final de la lista
- la página traída hace más tiempo está al inicio

Al ocurrir un *page fault*, se saca la página la inicio

... y la nueva página se agrega al final

Solo que la página más antigua podría ser útil todavía:

- FIFO no se usa mucho

## Segunda oportunidad

Una modificación a FIFO: se revisa el bit **R** de la página más antigua:

- si es 0, entonces la página se saca inmediatamente
- si es 1, entonces se pone en 0, la página se pone al final de la lista, y se sigue buscando a partir de la nueva página al inicio de la lista

## Reloj

Modificación a Segunda oportunidad: en lugar de mover las páginas con el bit **R** en 1 hasta el final de la lista

... las páginas se mantienen en una lista circular (dispuestas como los números en la esfera de un reloj) con un único puntero apuntando a la página más antigua

Cuando ocurre un *page fault*, se revisa el bit **R** de la página apuntada:

- si es 0, entonces la página se saca, la página nueva se pone en ese lugar, y se avanza el puntero a la próxima página
- si es 1, entonces se pone en 0, el puntero avanza a la próxima página, y la búsqueda se reanuda desde allí

## **LRU** (*least recently used*)

Buena aproximación al algoritmo óptimo

Basado en la observación de que las páginas que han sido muy usadas recientemente probablemente van a volver a ser muy usadas pronto

... mientras que las páginas que no han sido usadas recientemente probablemente van a seguir sin ser usadas próximamente

Cuando ocurre un *page fault*, saca la página que no ha sido usada por más tiempo

## ¿Cómo implementarlo eficientemente?

P.ej., con hardware especial:

- un contador  $C$  de 64 bits automáticamente incrementado después de cada instrucción
  - ... + cada línea en la tabla de páginas tiene un campo para el contador
- después de cada referencia a memoria, el valor actual de  $C$  se almacena en la línea correspondiente a la página referenciada
- al ocurrir un *page fault*, el sistema operativo revisa todos los contadores en la tabla de páginas, buscando el de menor valor
  - ... esa página es la usada menos recientemente

Solo que los computadores reales difícilmente tienen este tipo de hardware

## NFU (*not frequently used*)

Simulación de LRU en software:

- un contador asociado con cada página, inicialmente en 0
  - ... en cada interrupción del reloj, el sistema operativo revisa todas las páginas, sumando el valor del bit **R** (0 o 1) correspondiente a cada página
    - ( los contadores llevan la cuenta aproximada de qué tan frecuentemente las páginas han sido referenciadas )
- al ocurrir un *page fault*, se saca la página con el contador con el menor valor

Solo que frecuentemente  $\neq$  recientemente

## Envejecimiento

Modificación a NFU:

- los contadores (de cada página) —p.ej., de 8 bits— son desplazados (*shifted*) un bit a la derecha antes de sumarles el valor del bit **R**
- el bit **R** es sumado al bit de más a la izquierda (y no al de más a la derecha)

Cuando ocurre un *page fault*, se saca la página con el contador con el menor valor

## ***Working set***

El *working set* es el conjunto de páginas usadas por las  $k$  referencias a memoria (o, similarmente, a páginas) más recientes:

- hay que elegir un valor de  $k$  por adelantado
- si bien hay un rango amplio de valores de  $k$  para el cual el *working set* es más o menos el mismo

... el sistema operativo debe seguir la pista a cuáles páginas están en el *working set* [próx. diap.]

Cuando ocurre un *page fault*, sacamos una página que no esté en el *working set*

En lugar de contar las últimas  $k$  referencias a memoria

... definimos el *working set* como el conjunto de páginas usadas durante los últimos  $\tau$  ms de ejecución (del proceso en cuestión) —una definición igualmente buena pero más fácil para trabajar con ella

Ahora, cada fila de la tabla de páginas contiene:

- el bit  $R$ , que es vuelto a 0 con cada tick del reloj
- la hora aproximada en que la página fue usada por última vez

Al ocurrir un *page fault*, se revisa la tabla de páginas (solo las filas correspondientes a páginas que están en memoria):

- si  $R=1$ , se actualiza la hora de último uso de la página
- si  $R=0$ , se calcula la *edad* de la página (hora actual – hora de último uso) y si es mayor que  $\tau$  ms —la página ya no está en el *working set*— esta es la página que se saca

... el resto de la tabla se sigue revisando

- si  $R=0$  pero la edad de la página es  $\leq \tau$  ms, la página aún está en el *working set* y solo se toma nota de la página de mayor edad
- si al terminar la revisión todas las páginas están en el *working set*, entonces si hay páginas con  $R=0$ , se saca la de mayor edad entre ellas

... pero si todas las páginas tienen  $R=1$ , entonces se saca una aleatoriamente (preferiblemente, una que no haya sido modificada)

Toda la tabla de páginas es revisada en cada *page fault*

## WSClock

Basado en el algoritmo del reloj más información del *working set*

De los más usados en la práctica, por simplicidad de implementación y buen desempeño

Las páginas se disponen en una lista circular, como en el algoritmo del reloj, incluyendo la hora del último uso y los bits **R** y **M**

Al ocurrir un *page fault*, se examina la página apuntada por el puntero:

- si  $R=1$ , entonces  $R$  se pone en 0, el puntero se avanza a la próxima página, y el algoritmo se repite a partir a partir de esta página
- si  $R=0$ ,  $M=0$  y edad >  $\tau$  ms, entonces ésta es la página que se saca
- si  $R=0$ ,  $M=1$  y edad >  $\tau$  ms, entonces se programa la escritura de la página al disco, pero se avanza el puntero y el algoritmo continúa con la próxima página (podría haber una página con  $R=0$ ,  $M=0$  y edad >  $\tau$  ms más adelante)

¿Qué pasa si el puntero vuelve al punto de partida?

Si el puntero vuelve al punto de partida, hay dos casos:

- si al menos se ha programado una escritura de página al disco, el puntero sigue avanzando buscando una página que no haya sido modificada

... en el peor caso, al menos la página programada va a ser escrita y su bit **M** volverá a 0

- si no se ha programado ninguna escritura de página al disco, todas las páginas están en el *working set*

... lo más simple es sacar cualquier página no modificada (con bit **M=0**)

... si no hay ninguna, se saca la página que está siendo apuntada

## ¿Qué es multiprogramación?

- forma básica de procesamiento paralelo, usando un único procesador
- como hay un solo procesador, el paralelismo generado no es 100% real
- se basa en la ejecución intercalada de (partes de) los programas, o **procesos**, en intervalos de tiempo muy pequeños
- de todas maneras, el usuario percibe la ejecución como realmente paralela

¿Qué es lo que define esencialmente a un proceso que está ejecutándose?

El estado de la CPU:

- $PC$ ,  $SP$ , registros disponibles para el programador,  $flags$  (bits de condición)
- para multiprogramación, necesitamos que la CPU maneje múltiples estados

El estado de la memoria (principal):

- las instrucciones del proceso, (los valores de) las variables del proceso, el  $stack$  de registros de activación (de la ejecución de funciones)
- para multiprogramación, tenemos que permitir múltiples procesos en memoria

¿Cómo lo hacemos para darle memoria a muchos procesos distintos?

Una solución simple es asignarle una parte de la memoria (principal) a cada proceso

... pero:

- el programador tendría que saber a priori cuál es la parte de la memoria que le va a corresponder a su proceso
- si no hay un mecanismo de protección, es posible que un proceso escriba en la parte de memoria asignada a otro
- el tamaño de la memoria asignada a cada proceso debe ser fijo

La *memoria virtual* soluciona también estos problemas:

- cada proceso trabaja sobre un espacio virtual de memoria, equivalente al espacio direccionable completo
- la MMU traduce las direcciones de ese espacio virtual de memoria a direcciones físicas (direcciones reales en la memoria principal)

## Memoria virtual

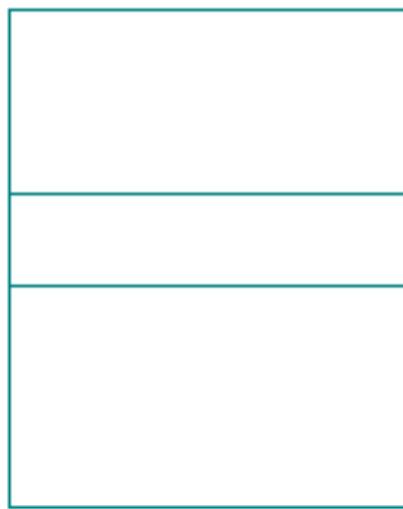
P1

100



P2

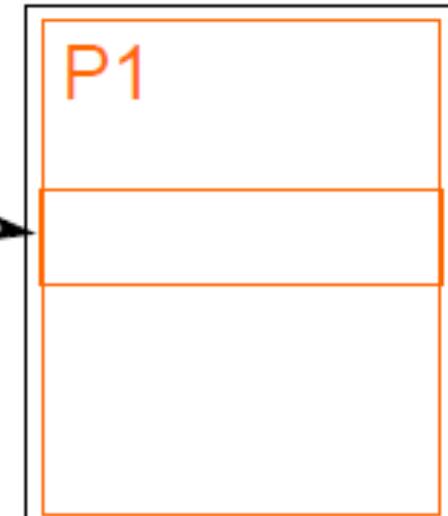
100



## Memoria física

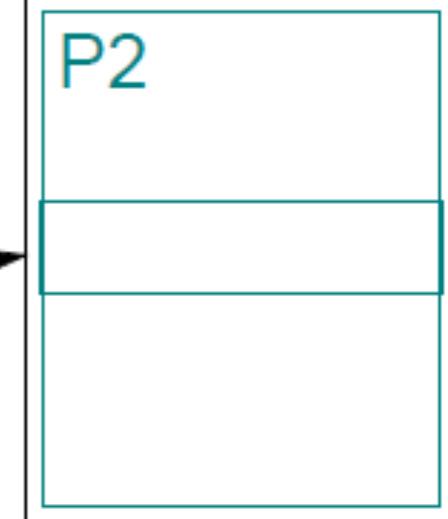
P1

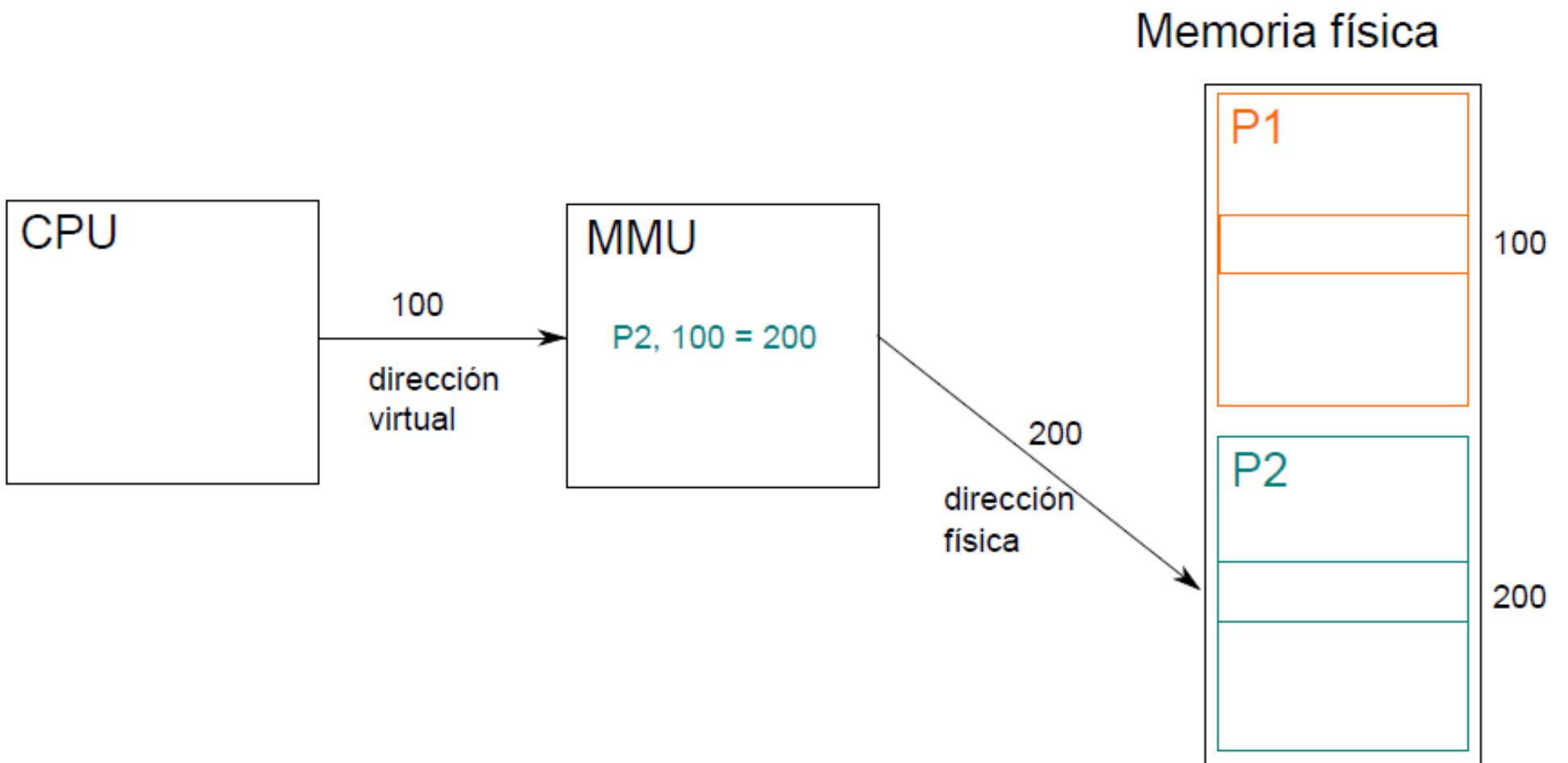
100



P2

200





En la práctica, como vimos para el caso de un programa

... la memoria virtual de cada programa se divide en *páginas* (virtuales), todas del mismo tamaño

... y la memoria principal (o física) se divide en *page frames* (o marcos físicos) del mismo tamaño que las páginas

Para cada proceso, una *tabla de páginas* específica para cada página virtual:

- si la página virtual está o no en memoria (mediante un *bit de validez*)
- ... y, en caso de estar en memoria, en cuál *page frame* está

Este esquema se conoce, como ya sabemos, como *paginación*:

- la próxima diap. muestra un ej. para dos programas,  $P_1$  y  $P_2$ , que (posiblemente) se alternan en el uso de la CPU pero conviven en memoria
- la diap. #76 muestra la tabla de páginas (resumida) correspondiente a  $P_1$

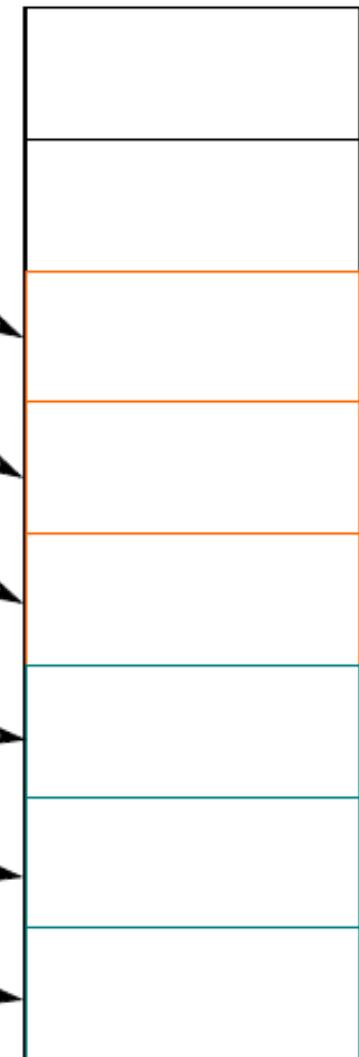
## Memoria virtual

Página virtual	Marco físico
0	2
1	3
2	4

P1



## Memoria física



P2

Página virtual	Marco físico
0	5
1	6
2	7



Página virtual	Marco físico	Validez
0	2	1
1	3	1
2	4	1
3	x	0
4	x	0
5	x	0
6	x	0
7	x	0

Las siguientes diaps. muestran lo que ocurre a medida que los procesos necesitan páginas adicionales para poder continuar su ejecución:

- inicialmente, tanto  $P_1$  como  $P_2$  tienen 4 páginas cada uno en memoria principal (#78)
- entonces  $P_1$  solicita una dirección que pertenece a una quinta página virtual, que no está en memoria → *page fault* (#79)
- una de las páginas en memoria de  $P_1$  es elegida y enviada al disco → *swap out* (#80)
- ... en donde es puesta en el ***swap space*** correspondiente a  $P_1$ , mientras que el *page frame* que ocupaba es ahora ocupado por la nueva página virtual (#81)
- la tabla de páginas de  $P_1$  refleja esta situación (#82)

...

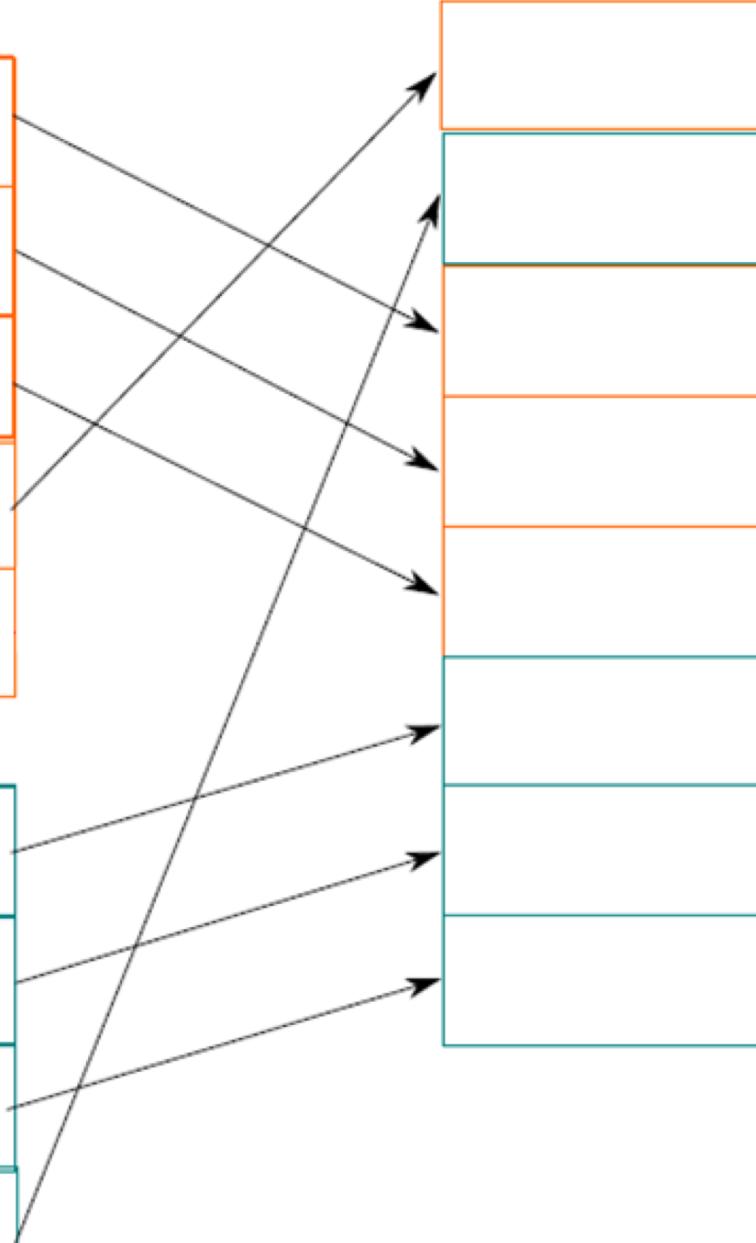
## Memoria virtual

## Memoria física

P1



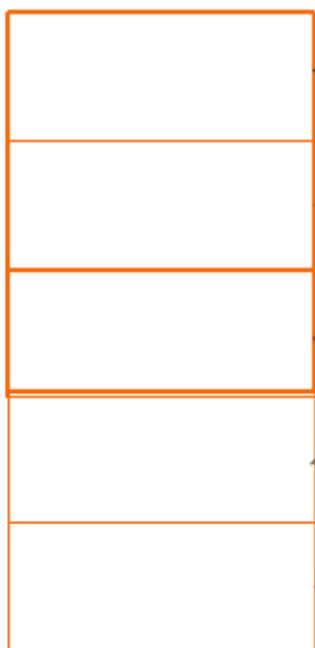
P2



## Memoria virtual

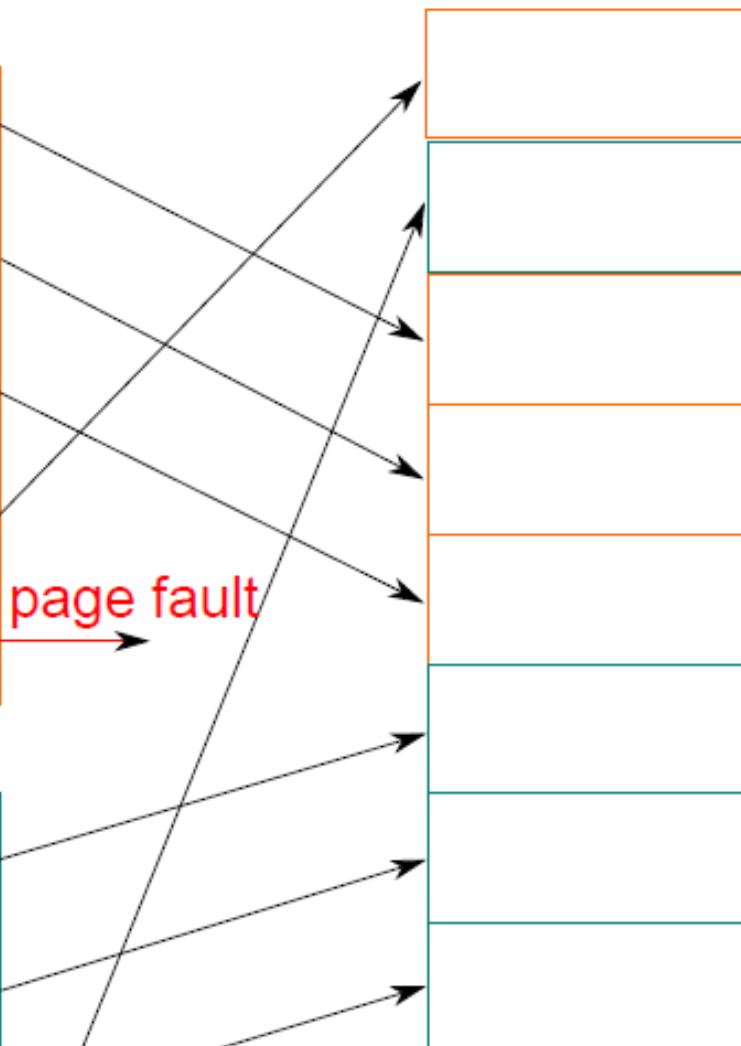
## Memoria física

P1



page fault

P2



Memoria virtual

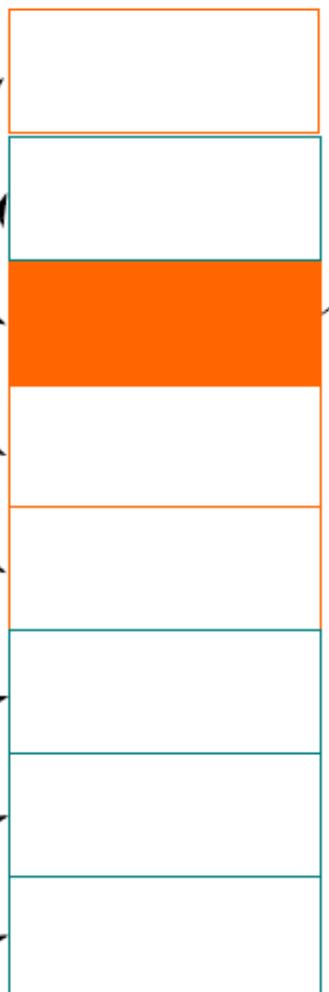
P1



P2



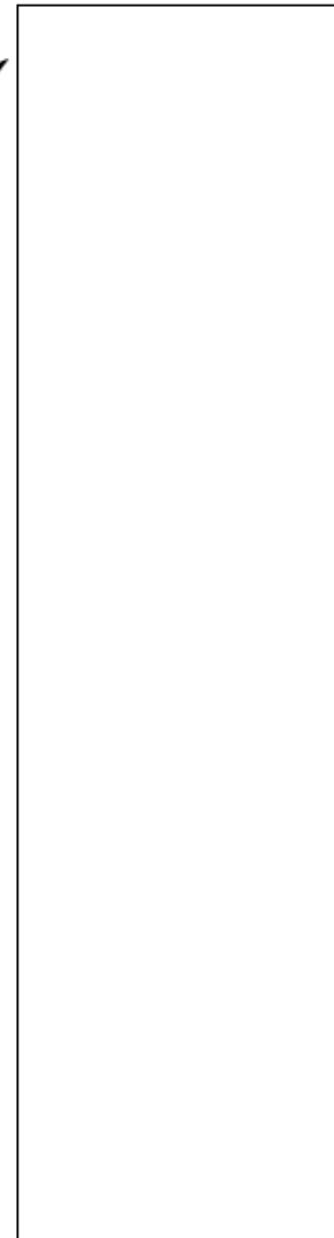
Memoria física



Disco: Swap File

7  
8  
1  
2  
3  
4  
5  
6

swap out



## Memoria virtual

P1

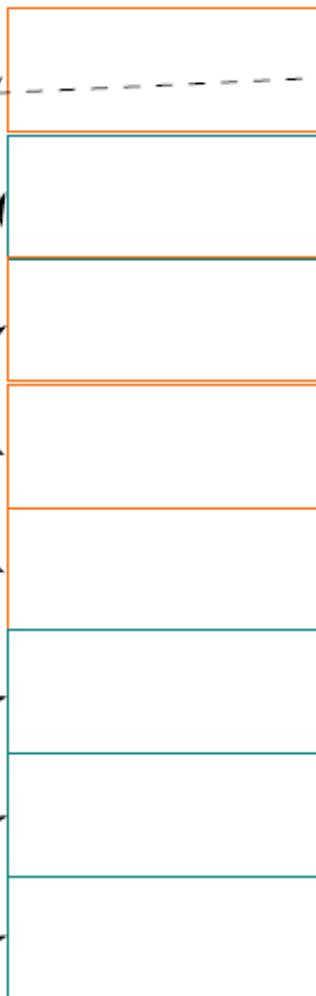


P2



## Memoria física

6  
7  
8  
1  
2  
3  
4  
5



## Disco: Swap File



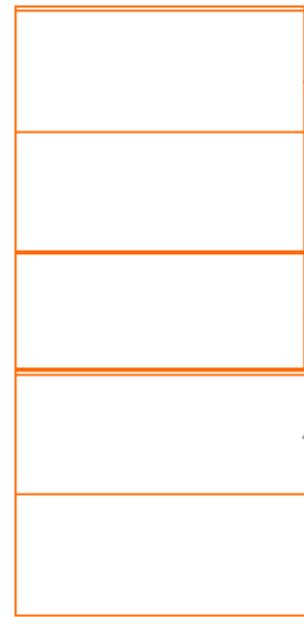
Página virtual	Marco físico	Validez	Disco
0	2	1	1
1	3	1	0
2	4	1	0
3	0	1	0
4	2	1	0
5	x	0	0
6	x	0	0
7	x	0	0

...

- si la página que recién salió de la memoria es requerida nuevamente por  $P_1$  para poder seguir su ejecución, se produce un nuevo *page fault* (#84)
- ... que obliga a que otra página de  $P_1$  en memoria sea enviada al *swap space* (#85)
- ... de modo que el *page frame* de esta última página es ahora ocupado por la página recientemente requerida por  $P_1$  (que es traída de vuelta desde el *swap space*, #86)
- lo que implica una nueva actualización de la tabla de páginas de  $P_1$  (#87 y 88)

Memoria virtual

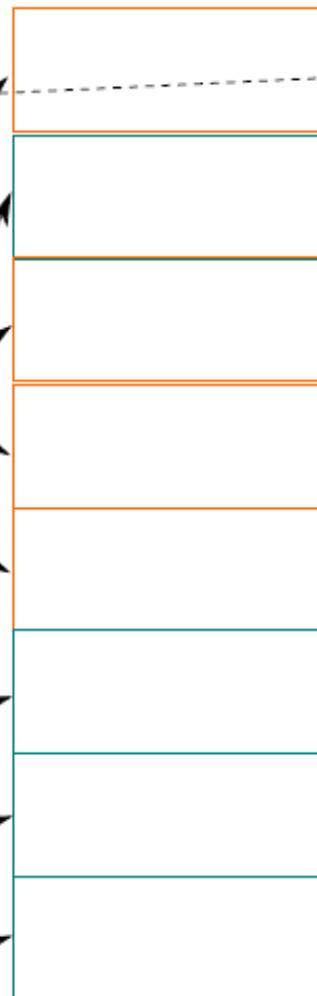
P1



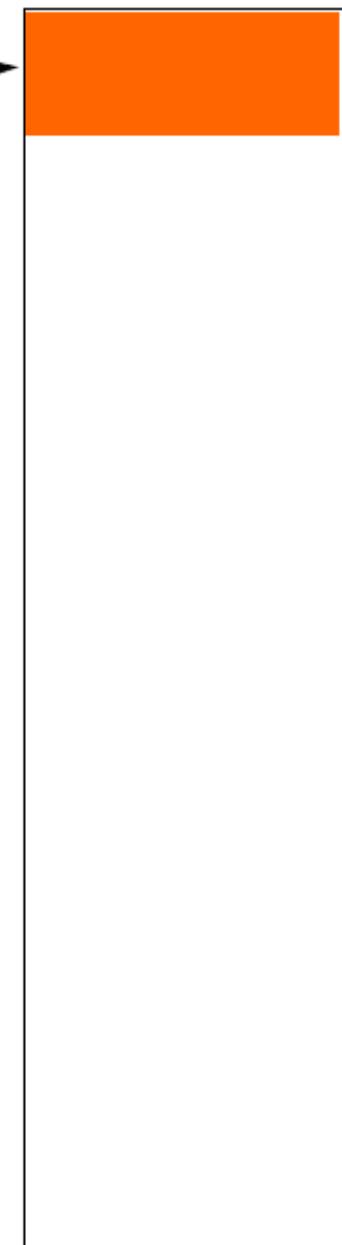
P2



Memoria física

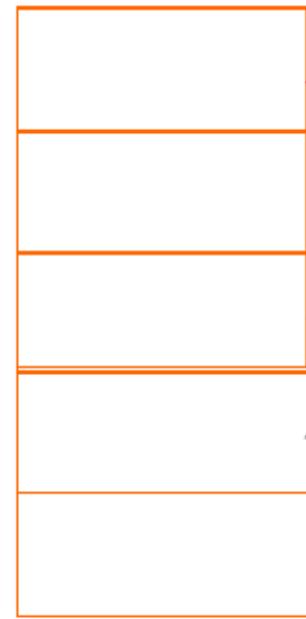


Disco: Swap File



Memoria virtual

P1



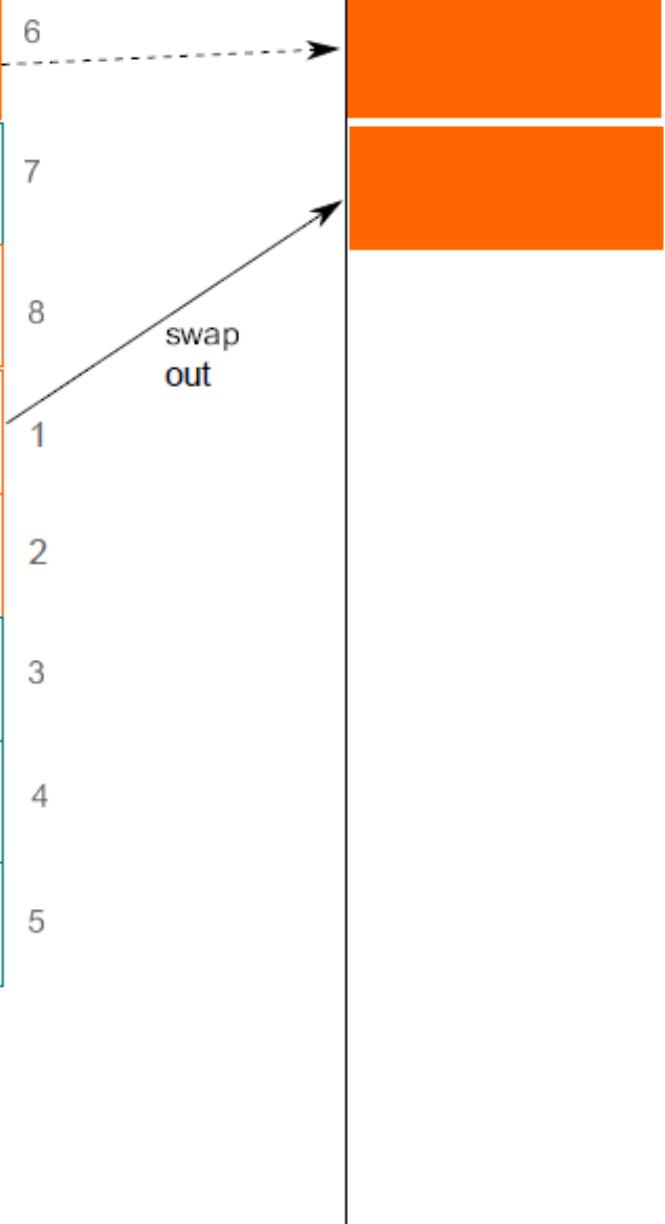
P2



Memoria física

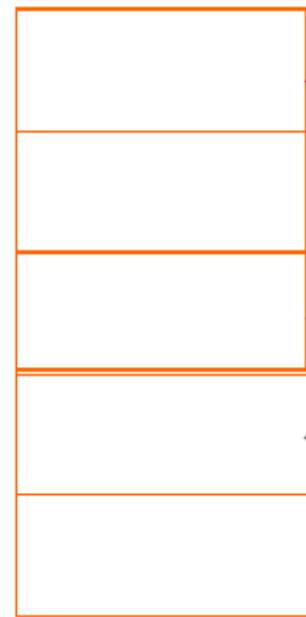


Disco: Swap File



Memoria virtual

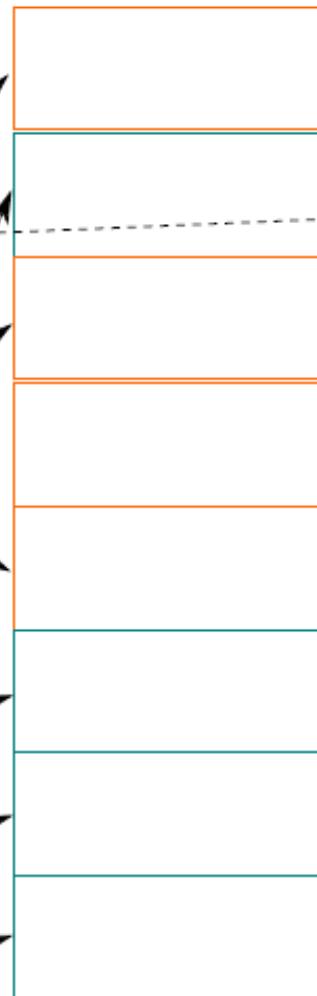
P1



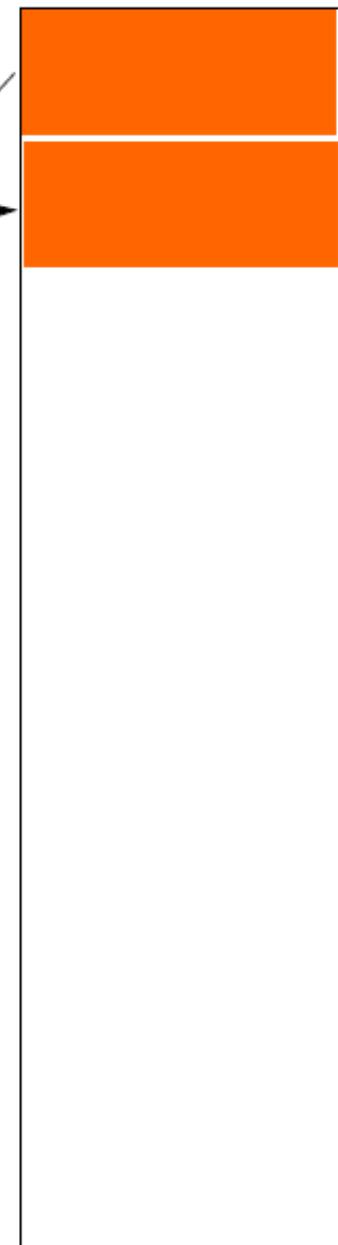
P2



Memoria física

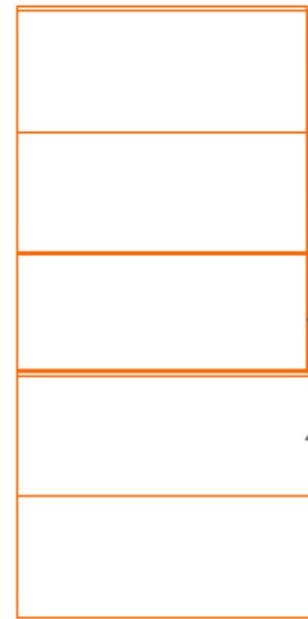


Disco: Swap File



Memoria virtual

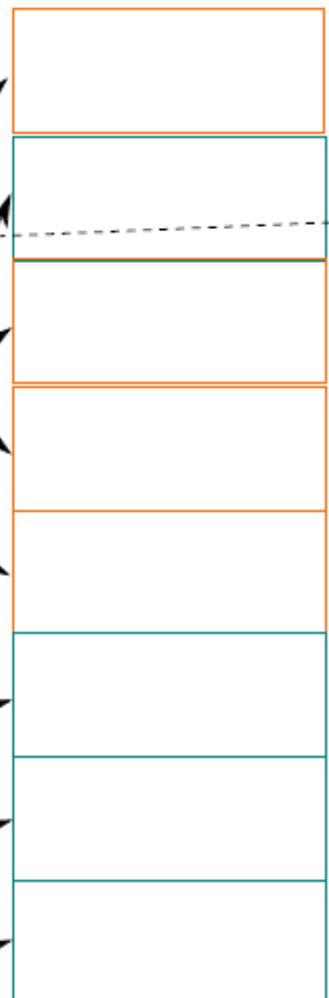
P1



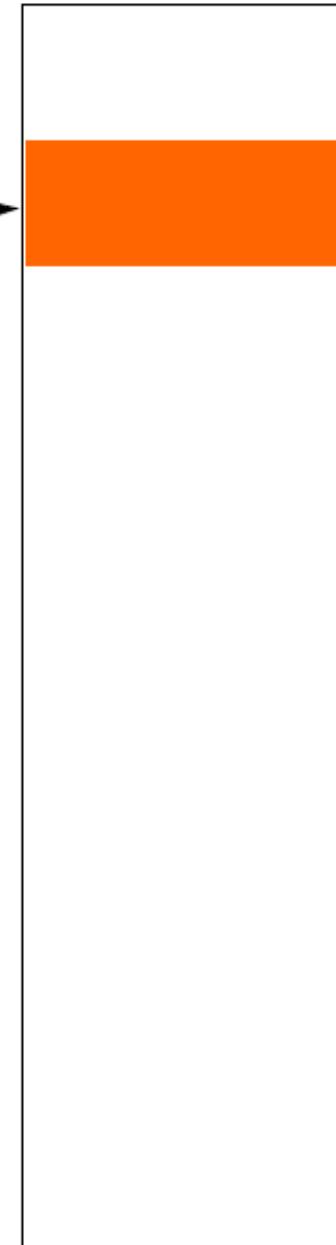
P2



Memoria física



Disco: Swap File



Página virtual	Marco físico	Validez	Disco
0	3	1	0
1	3	1	1
2	4	1	0
3	0	1	0
4	2	1	0
5	x	0	0
6	x	0	0
7	x	0	0