

# Memoria

---

Arquitectura de Computadores – IIC2343

Las CPUs siempre han sido más rápidas que las memorias

... y la diferencia solo ha ido aumentando con el tiempo

Parte de la solución es colocar la memoria en el chip de la CPU —ya que el acceso a la memoria a través del bus es muy lento

... pero esto significa CPUs más grandes —hay límites económicos y prácticos para el tamaño del chip de la CPU

Los programadores siempre hemos querido tener cantidades ilimitadas de memoria rápida:

- estudiaremos cómo crear esta ilusión

La elección parece ser entre una cantidad pequeña de memoria rápida

... o una cantidad grande de memoria lenta

Excepto que hay técnicas para combinarlas de modo de obtener simultáneamente la velocidad de una memoria rápida

... y la capacidad de una memoria de gran tamaño a un precio razonable

**El principio de localidad** subyace a la forma en que funcionan los programas en la práctica:

**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:

- p.ej., diap. #6

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

- p.ej., diap. #7

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

| Dirección | Label       | Instrucción/Dato |
|-----------|-------------|------------------|
|           |             | CODE:            |
| 0x00      | start:      | MOV CL, [var1]   |
| 0x01      | while:      | MOV AL,[res]     |
| 0x02      |             | ADD AL,[var2]    |
| 0x03      |             | MOV [res],AL     |
| 0x04      |             | SUB CL,1         |
| 0x05      |             | CMP CL,0         |
| 0x06      |             | JNE while        |
|           |             | DATA:            |
| 0x07      | var1        | 3                |
| 0x08      | <b>var2</b> | <b>2</b>         |
| 0x09      | res         | 0                |

| Dirección | Label   | Instrucción/Dato  |
|-----------|---------|-------------------|
|           |         | CODE:             |
| 0x00      | start:  | MOV SI, 0         |
| 0x01      |         | MOV AX, 0         |
| 0x02      |         | MOV BX, arreglo   |
| 0x03      |         | MOV CL, [n]       |
| 0x04      | while:  | CMP SI, CX        |
| 0x05      |         | JGE end           |
| 0x06      |         | MOV DX, [BX + SI] |
| 0x07      |         | ADD AL, DL        |
| 0x08      |         | INC SI            |
| 0x09      |         | JMP while         |
| 0x0A      | end:    | DIV CL            |
| 0x0B      |         | MOV [prom], AL    |
|           |         | DATA:             |
| 0x0C      | arreglo | 6                 |
| 0x0D      |         | 7                 |
| 0x0E      |         | 4                 |
| 0x0F      |         | 5                 |
| 0x10      |         | 3                 |
| 0x11      | n       | 5                 |
| 0x12      | prom    | 0                 |

Aprovechamos el principio de localidad implementando la memoria como una **jerarquía de memorias**:

- múltiples niveles de memoria con diferentes velocidades y tamaños

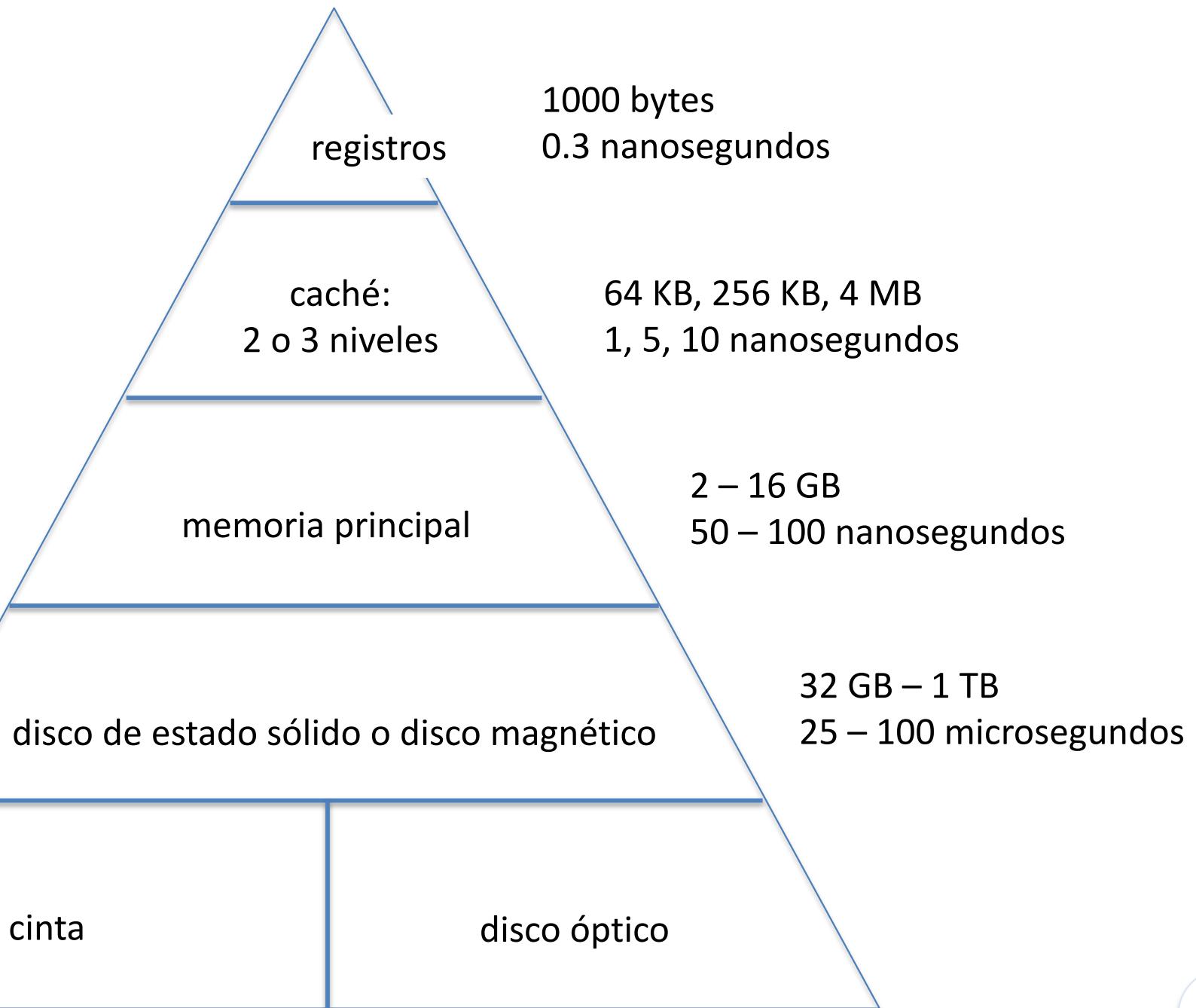
Las memorias más rápidas son más caras por bit que las memorias más lentas

... y por lo tanto son más pequeñas

También, las memorias más rápidas están más cerca del procesador

El propósito es ofrecer al programador tanta memoria como esté disponible en la tecnología más barata

... pero a la velocidad de acceso de la memoria más rápida



El contenido de las memorias también está jerarquizado

El contenido de un nivel más cerca del procesador es un subconjunto del contenido de cualquier nivel que está más lejos:

- ... y el total del contenido necesario para ejecutar un programa —tanto las instrucciones como los datos— está en el nivel más lejano

A medida que nos alejamos del procesador, los accesos a los distintos niveles de memoria toman progresivamente más tiempo

Los datos son copiados solo entre dos niveles adyacentes cada vez:

- p.ej., entre cache y memoria principal, o entre memoria principal y disco
- el nivel más cercano al procesador es más pequeño y más rápido que el nivel más lejano
  - ... ya que usa tecnología más cara

La unidad mínima de información que puede estar presente o no en una jerarquía de dos niveles y que se copia entre niveles adyacentes es un **bloque** o una **línea** (o también una **página**):

- normalmente consistente en múltiples bytes en computadores reales
  - ... aunque en nuestros ejemplos usaremos bytes individuales

Si el dato pedido por el procesador está en algún bloque del nivel más cercano → ***hit***

... de lo contrario → ***miss***:

- en este caso, se va a un nivel más abajo en el jerarquía —más lejano del procesador— para traer el bloque que contiene al dato pedido

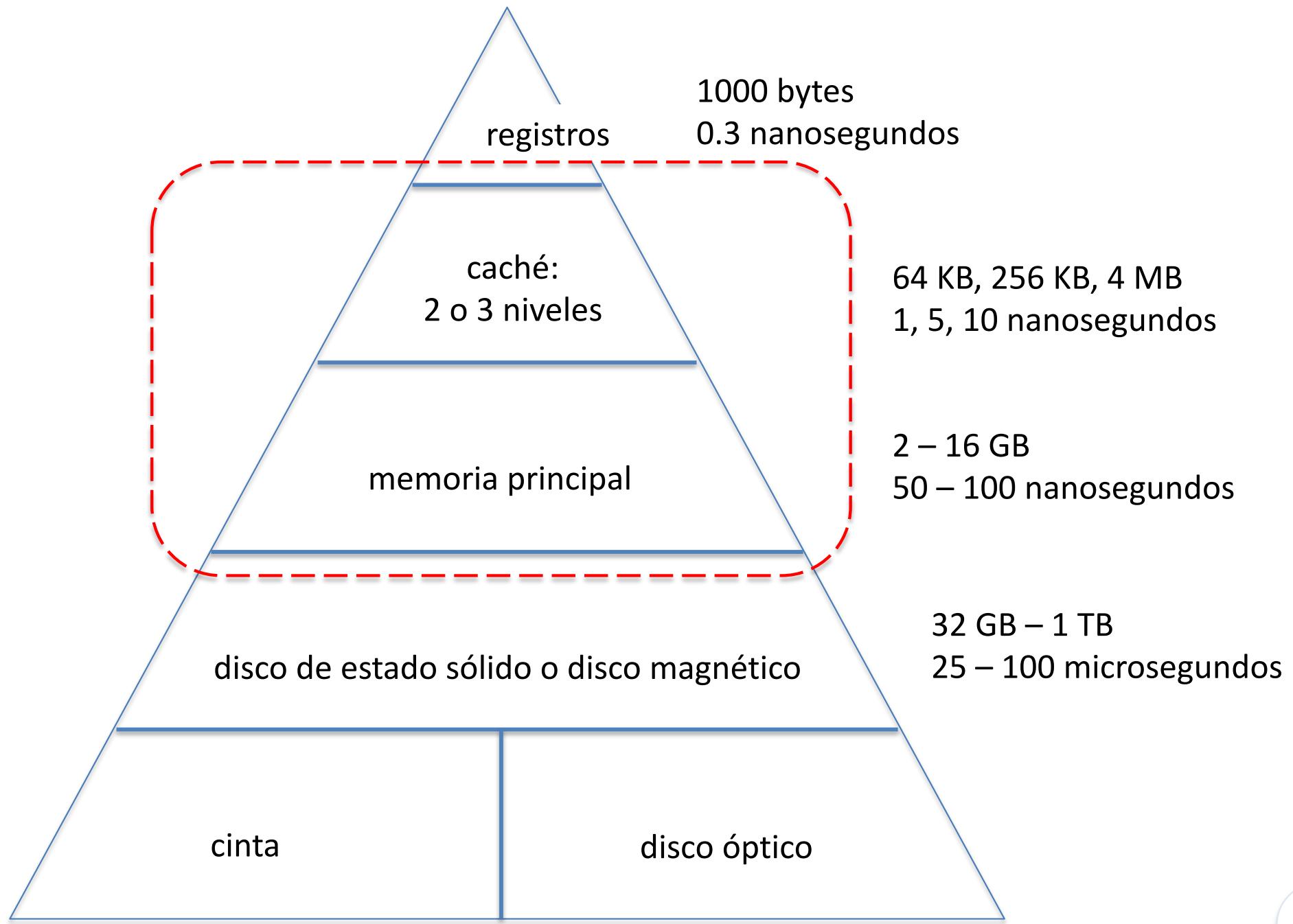
**hit rate**: fracción de los accesos a memoria que son encontrados en el nivel cercano — medida de desempeño de la jerarquía de memoria

**miss rate**:  $1 - \text{hit rate}$

**hit time**: tiempo que toma tener acceso al nivel más cercano de la jerarquía de memoria, incluyendo el tiempo necesario para determinar si el acceso es un *hit* o un *miss*

**miss penalty**: tiempo que toma reemplazar un bloque en el nivel más cercano por el bloque correspondiente del siguiente nivel, más el tiempo que toma entregar este bloque al procesador

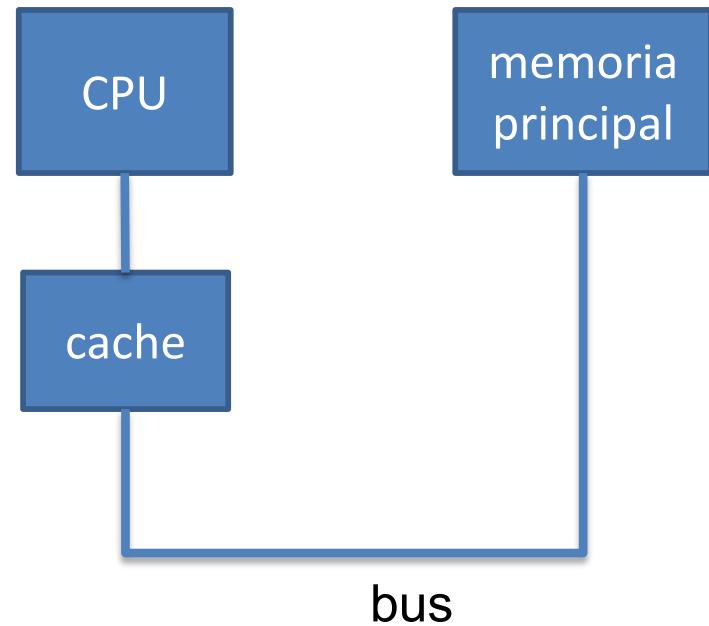
El *hit time* es mucho menor que el tiempo que toma tener acceso al siguiente nivel en la jerarquía (componente principal del *miss penalty*)



Después de los registros, la memoria cache —o simplemente la cache— ocupa el siguiente nivel de la jerarquía

La cache se encuentra en la CPU

... pero la CPU no sabe que existe



La ubicación lógica de la cache, entre la CPU y la memoria principal

La memoria principal y la cache se dividen en bloques y líneas, respectivamente —aprovechando la localidad espacial:

- **un bloque de la memoria tiene el mismo tamaño que una línea de la cache**
- cuando ocurre un *cache miss*, toda la línea de la cache (o bloque de memoria) es cargada (copiada) desde la memoria (a la cache), no solo la palabra a la que se hizo referencia —localidad espacial
- p.ej., si la línea de la cache tiene 64 bytes, una referencia a la dirección 260 va a traer los 64 bytes (un bloque) con direcciones 256 al 319

Para describir el funcionamiento, sólo es necesario entender su comunicación con la CPU y con el siguiente nivel de la jerarquía (memoria principal) —veamos

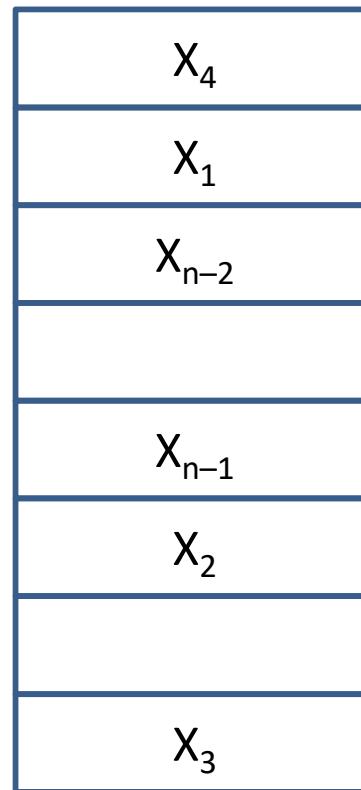
Supongamos, inicialmente, que las solicitudes del procesador son de una palabra (o de un byte)

... y que los bloques contienen también una palabra (o un byte)

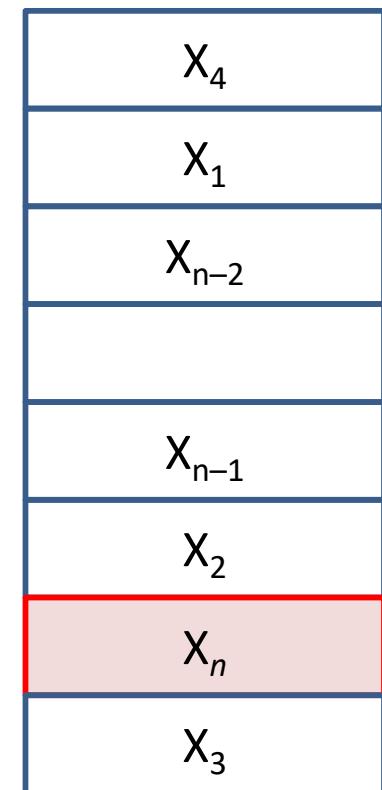
... ver figura

Dos preguntas:

- ¿cómo sabemos si un dato está en la cache?
- ... y, si está, ¿cómo lo encontramos?



Cache antes de la referencia a  $X_n$



Cache después de la referencia a  $X_n$

Si cada palabra —o cada dirección de memoria— puede ir en exactamente un lugar en la cache

... entonces es simple encontrar la palabra si ella efectivamente está en la cache

P.ej., si asignamos la ubicación en la cache, o **índice**, en base a la dirección (del bloque) de la palabra en la memoria —**direct mapping**:

- índice = (*dirección del bloque*) modulo ( $N$  = *número de líneas en la cache*)
- el cálculo se simplifica si  $N$  es una potencia de 2
  - ... p.ej., si  $N = 8$ , entonces todas las direcciones de memoria que terminan en ...001 van a parar a la línea de la cache con dirección 001,
  - ... y todas las direcciones de memoria que terminan en ...101 van a parar a la línea con dirección 101

Cada línea de la cache puede contener el contenido de varios bloques diferentes de memoria (aunque no al mismo tiempo):

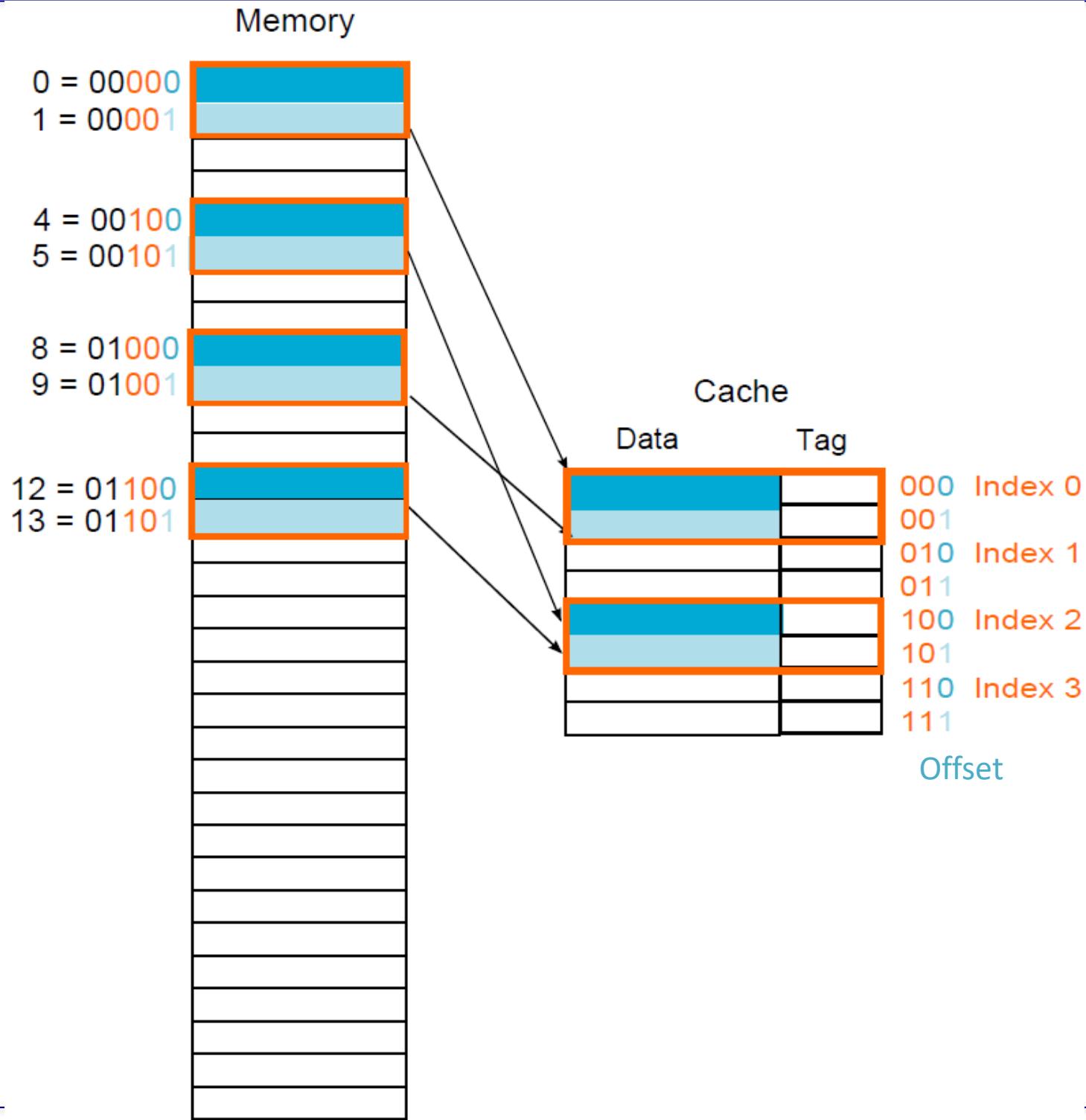
- todos aquellos bloques para los cuales el valor de ... (*dirección del bloque*) modulo ( $N = \text{número de líneas en la cache}$ ) ... es el mismo

... → necesitamos indicar cuál bloque está efectivamente en la cache:

- usamos un **tag**, correspondiente al resto de los bits (los más significativos) de la dirección de memoria

En la próxima diapo., vemos el esquema cuando la memoria tiene 32 palabras y cada línea de la cache almacena dos palabras de la memoria (es decir, bloques de tamaño 2 palabras):

- p.ej., si en un momento las palabras de la memoria que ocupan la tercera línea ( $2_{10} = 10_2$ ) de la cache son las palabras con direcciones **00100** y **00101**, entonces el *tag* correspondiente va a ser **00**  
... el bit menos significativo, **0** o **1**, se usa como **offset** dentro de la línea
- ... y si más adelante las palabras que ocupan la misma tercera línea de la cache son las palabras con direcciones **01100** y **01101**, entonces el *tag* va a ser **01**



Finalmente, para saber si el contenido de la línea es válido, usamos un bit adicional —**bit de validación**:

- cuando el procesador recién parte, la información en la cache no es válida

En las próximas diapos., vemos el comportamiento de una cache de 8 bytes/línea y 4 líneas, inicialmente vacía

... bajo la secuencia de accesos a las direcciones de memoria 12, 13, 14, 4, 12 y 0

# Acceso a direcciones de memoria 12, 13, 14, 4, 12 y 0

## Cache de 8 bytes/línea y 4 líneas

| índ.<br>línea |   | valid | tag | dato |
|---------------|---|-------|-----|------|
| 00            | 0 | 0     |     |      |
|               | 1 | 0     |     |      |
| 01            | 0 | 0     |     |      |
|               | 1 | 0     |     |      |
| 10            | 0 | 0     |     |      |
|               | 1 | 0     |     |      |
| 11            | 0 | 0     |     |      |
|               | 1 | 0     |     |      |

inicial

luego del acceso a la dirección 12 = **01100**

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 01  | mem[12] |
|               | 1 | 1     | 01  | mem[13] |
| 11            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |

Luego del acceso a la dirección 13 = 01101

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 01  | mem[12] |
|               | 1 | 1     | 01  | mem[13] |
| 11            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |

Luego del acceso a la dirección 14 = 01110

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 01  | mem[12] |
|               | 1 | 1     | 01  | mem[13] |
| 11            | 0 | 1     | 01  | mem[14] |
|               | 1 | 1     | 01  | mem[15] |

Luego del acceso a la dirección 4 = 00100

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 00  | mem[4]  |
|               | 1 | 1     | 00  | mem[5]  |
| 11            | 0 | 1     | 01  | mem[14] |
|               | 1 | 1     | 01  | mem[15] |

Luego del acceso a la dirección 12 = 01100

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 01  | mem[12] |
|               | 1 | 1     | 01  | mem[13] |
| 11            | 0 | 1     | 01  | mem[14] |
|               | 1 | 1     | 01  | mem[15] |

luego del acceso a la dirección 0 = 00000

| índ.<br>línea |   | valid | tag | dato    |
|---------------|---|-------|-----|---------|
| 00            | 0 | 1     | 00  | mem[0]  |
|               | 1 | 1     | 00  | mem[1]  |
| 01            | 0 | 0     |     |         |
|               | 1 | 0     |     |         |
| 10            | 0 | 1     | 01  | mem[12] |
|               | 1 | 1     | 01  | mem[13] |
| 11            | 0 | 1     | 01  | mem[14] |
|               | 1 | 1     | 01  | mem[15] |

Supongamos una cache con 64 líneas, en que cada línea tiene 16 bytes

... ¿cuál es el índice de la línea correspondiente a la dirección de memoria 1200?

respuesta: 11

Si las líneas de la cache son de mayor capacidad (almacenan más palabras o bytes), se aprovecha mejor la localidad espacial y se reduce el *miss rate*

... excepto cuando el tamaño de las líneas llega a ser una fracción importante del tamaño de la cache:

- → va a haber pocas líneas → mucha competencia por esas líneas → cada línea va a ser reemplazada frecuentemente, antes de que haya sido posible tener acceso a varias de sus palabras

Aumentar el tamaño de las líneas también aumenta el costo de un *cache miss*:

- el tiempo que toma transferir un bloque de la memoria principal a una línea de la cache depende del tamaño del bloque (o de la línea), es decir, de cuántas palabras (o cuántos bytes) se transfieren

Manejo de un *cache miss* de una instrucción (la próxima instrucción que hay que ejecutar no está en la cache de instrucciones):

1. Enviar el valor original del PC (PC actual – 4) a la memoria
2. Ordenar a la memoria ejecutar una lectura y esperar (varios ciclos del *clock*) hasta que la memoria complete la lectura —el procesador queda "en pausa"
3. Escribir en la cache: los datos (el bloque) de la memoria en la línea, los bits más significativos de la dirección (desde la ALU) en el *tag*, y el bit de validez en 1
4. Reiniciar la ejecución de la instrucción desde el comienzo, la cual va a hacer *refetch* la instrucción, ahora va a estar en la cache

Ocurre colaborativamente entre el controlador de la cache, la unidad de control del procesador y el controlador que inicia el acceso a la memoria y re-escribe la cache

## Manejo de *writes*

En una instrucción *store*, si el dato es escrito solo en la cache y no en la memoria, entonces la cache y la memoria se vuelven *inconsistentes*

Para mantenerlas consistentes, siempre escribimos el dato tanto en la memoria como en la cache —***write-through***

Si el intento de escritura en la cache es un *miss*, entonces primero traemos el bloque desde la memoria a la cache, y luego escribimos el dato en la línea de la cache y también en la memoria

Simple ... solo que el desempeño no es muy bueno:

- supongamos un ciclo del *clock* del procesador por instrucción (sin *misses*)
- todo *write* escribe el dato en memoria, tomando p.ej., 100 ciclos
- si el 10% de las instrucciones son *stores*, entonces el número de ciclos por instrucción ahora es  $1 + 100 \times 10\% = 11$

La alternativa es efectivamente escribir el dato inicialmente solo en la línea de la cache

... y esperar hasta que la línea sea sacada de la cache (reemplazada por otra) para actualizar el bloque correspondiente en la memoria —***write-back***:

- mejora el desempeño, especialmente cuando el procesador genera *stores* frecuentemente
- ... pero es más complejo de implementar que *write-through*

En lugar de *direct-mapping*, el otro extremo en esquemas de asignación de bloques de memoria a líneas de cache es ***fully associative***:

- un bloque puede ir a parar a cualquier línea
- → la cache puede contener simultáneamente varias líneas que, bajo el esquema de *direct-mapping*, tendrían que competir por un lugar en la cache
- para encontrar el bloque (o la línea) en la cache, hay que mirar todas las líneas
- en la práctica, por desempeño, la búsqueda se hace en paralelo, encareciendo mucho el costo del hardware necesario

En una cache ***set associative***, hay un número fijo ( $> 1$ ) de ubicaciones en donde puede estar un bloque (o línea)

... si hay  $n$  ubicaciones posibles para cada dirección de memoria (un conjunto, o *set*, de  $n$  líneas), se habla de una ***cache n-way set associative***:

- en lugar de buscar la dirección en toda la cache, como en *fully-associative*, aquí se la busca solo entre  $n$  líneas
- → la cache puede contener simultáneamente más de un bloque de memoria que, bajo el esquema de *direct-mapping*, tendrían que competir por un lugar (una línea) en la cache
- 2-way y hasta 8-way funcionan bien en la práctica

set *tag* dato

|   |  |  |
|---|--|--|
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |
| 7 |  |  |

*direct-mapped*

set *tag* dato *tag* dato

|   |  |  |  |  |
|---|--|--|--|--|
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  |  |  |
| 3 |  |  |  |  |

*2-way set associative*

set *tag* dato *tag* dato *tag* dato *tag* dato

|   |  |  |  |  |  |  |  |  |
|---|--|--|--|--|--|--|--|--|
| 0 |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |

*4-way set associative*

*tag* dato *tag* dato

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

*fully associative*

## Reemplazo de bloques en esquemas *n-way associative*:

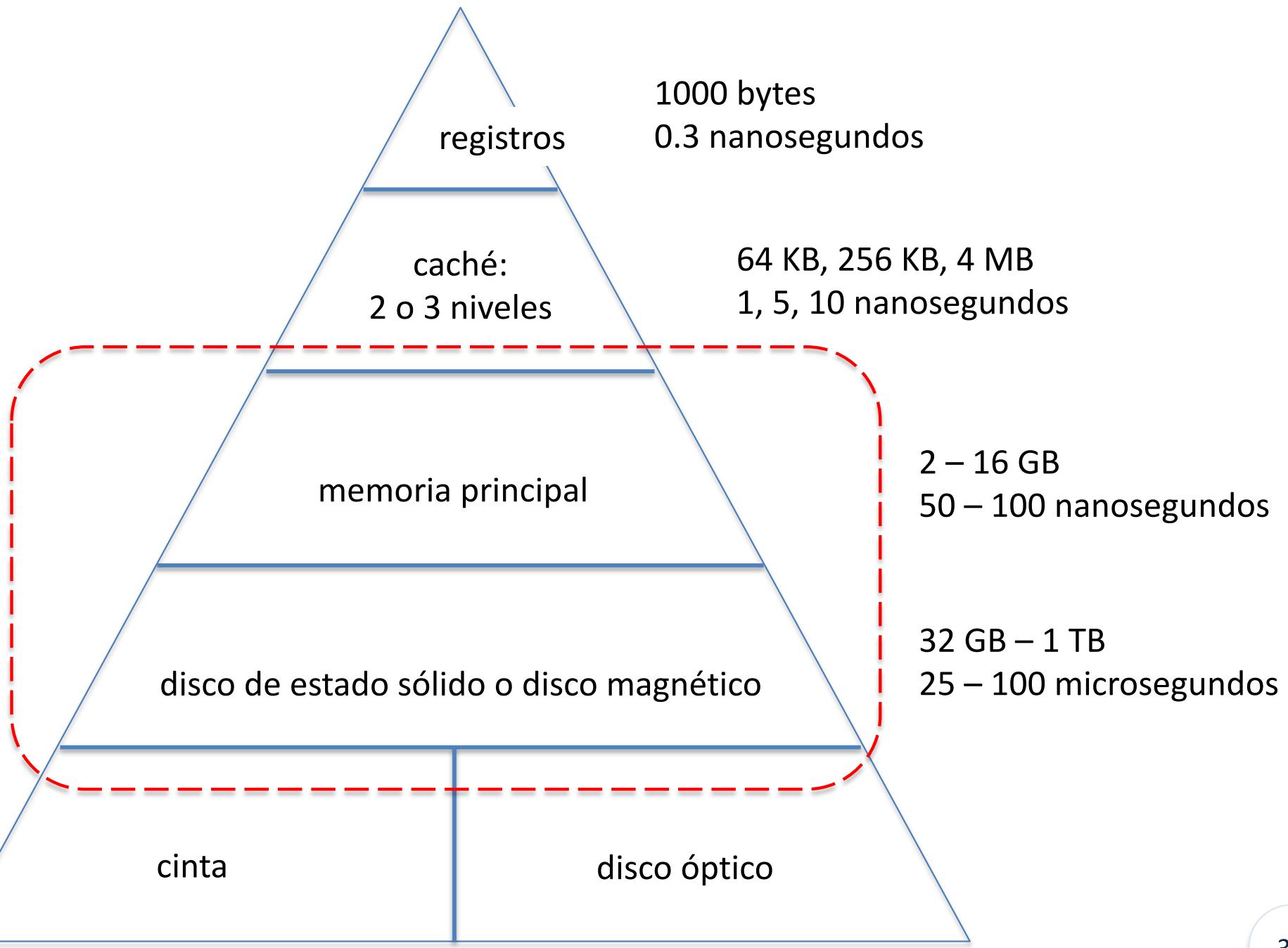
- Bélády: se saca la línea que se usará más lejos en el futuro; óptimo no alcanzable en la práctica
- *first-in first-out* (FIFO): el primero en entrar es el primero en salir; simple, pero tonto
- *least frequently used* (LFU): se saca la línea con menos accesos; mejor que FIFO, sólo un poco más complejo
- *least recently used* (LRU): se saca la línea con mayor tiempo sin accesos; complejo, requiere *timestamp*; en general el de mejor rendimiento — localidad temporal
- *random*: muy rápido y con rendimiento algo inferior a LFU y LRU

## ***Split cache:***

- sectores de la memoria visitados por accesos a datos e instrucciones son distintos
- patrones de acceso también son distintos
- cachés que tratan igual a datos e instrucciones (***unified cache***) pueden sufrir grandes bajas en el *hit-rate*
- para evitarlo, se usan caches divididas internamente en datos e instrucciones

## Los tres niveles de cache del procesador Intel Haswell:

- L1: caché *split* de 64 KB por núcleo, o *core* (32 KB para instrucciones y 32 KB para datos); líneas de 64 bytes, *8-way associative* + pseudo LRU, *write-back*; en el propio chip de la CPU
- L2: caché *unified* 256 KB por núcleo; líneas de 64 bytes, *8-way associative* + pseudo LRU, *write-back*; en el paquete de la CPU
- L3: 2 MB hasta 20 MB, *unified* y compartida entre núcleos; líneas de 64 bytes, *12-way associative* + pseudo LRU, *write-back*; en el *board* del procesador (junto a la memoria y los controladores de i/o)



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

1961: automatización del proceso de *overlays* → **memoria virtual**:

- la memoria principal actúa como una “cache” para el almacenamiento secundario
- 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

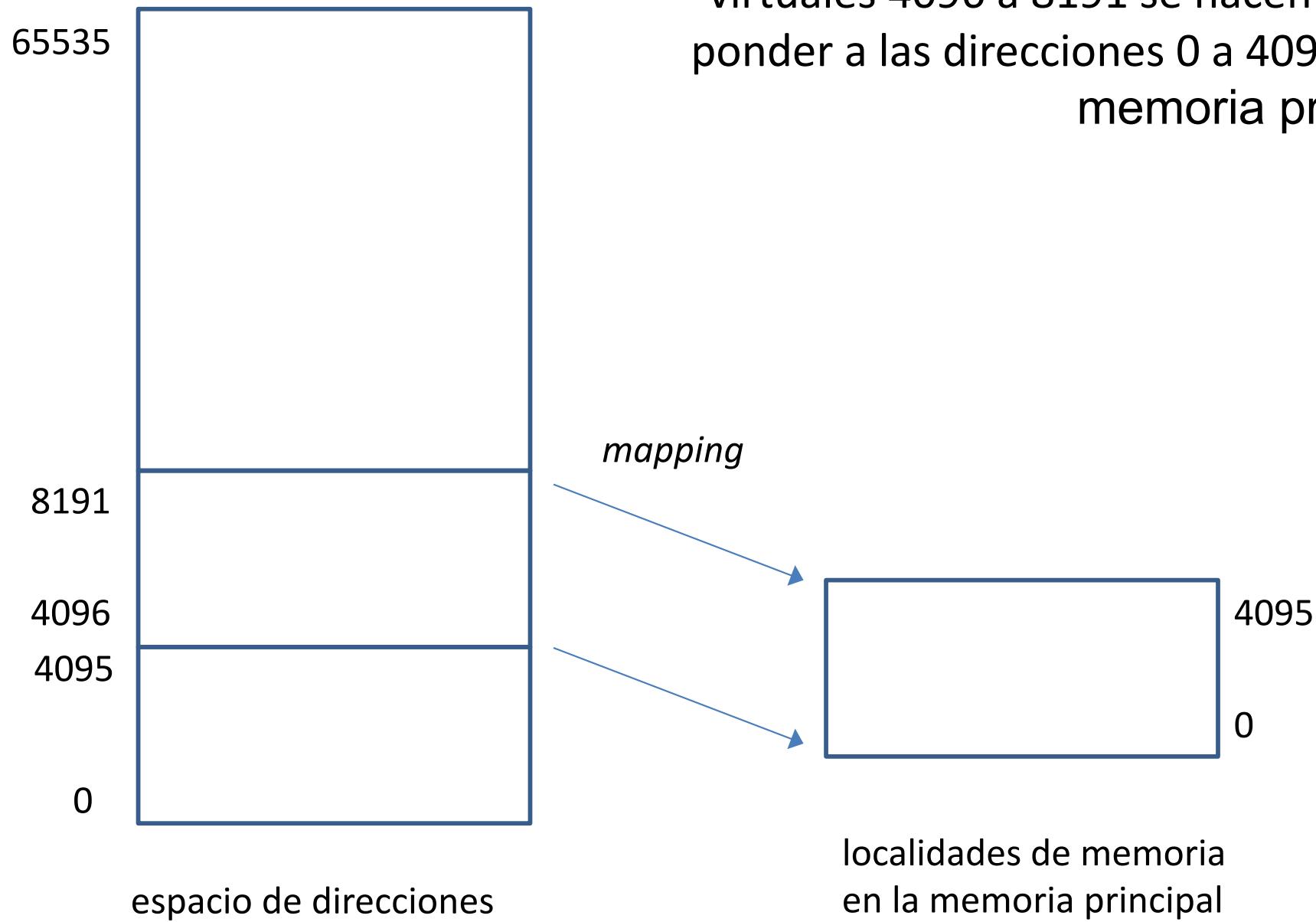
La idea está basada en separar los conceptos de **espacio de direcciones y localidades de memoria**

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

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



¿Qué pasa si un programa salta 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** (en lugar de bloques)

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)

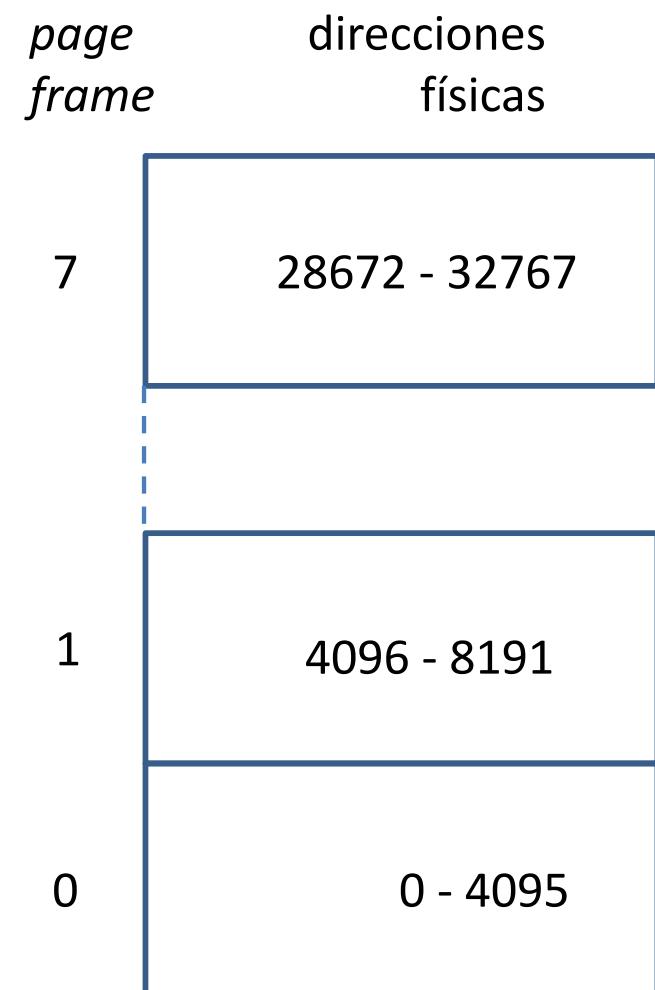
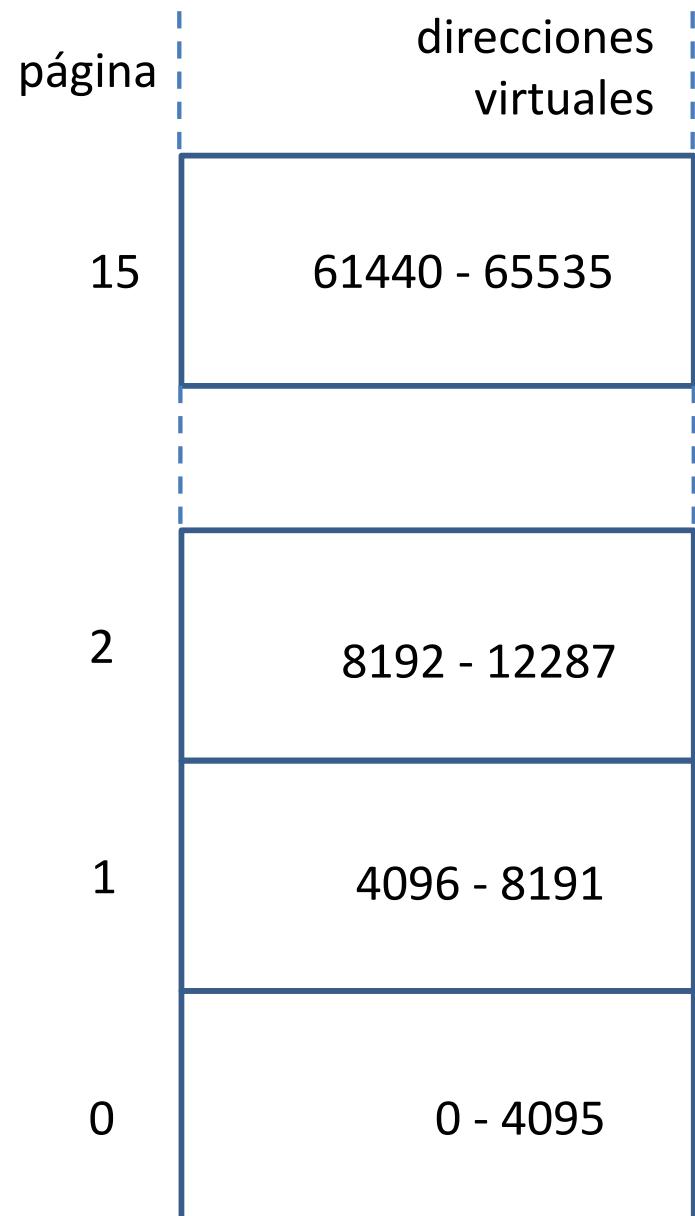
El mecanismo de paginación se dice que es **transparente**:

- ( excepto para quienes escriben sistemas operativos )

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*** (en lugar de las “líneas” de la cache) del mismo tamaño que las páginas:

- cada *page frame* de la memoria principal puede almacenar exactamente una página
- en la diap. #42, 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



La memoria virtual es implementada mediante una *tabla de páginas*, que tiene tantas entradas 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)

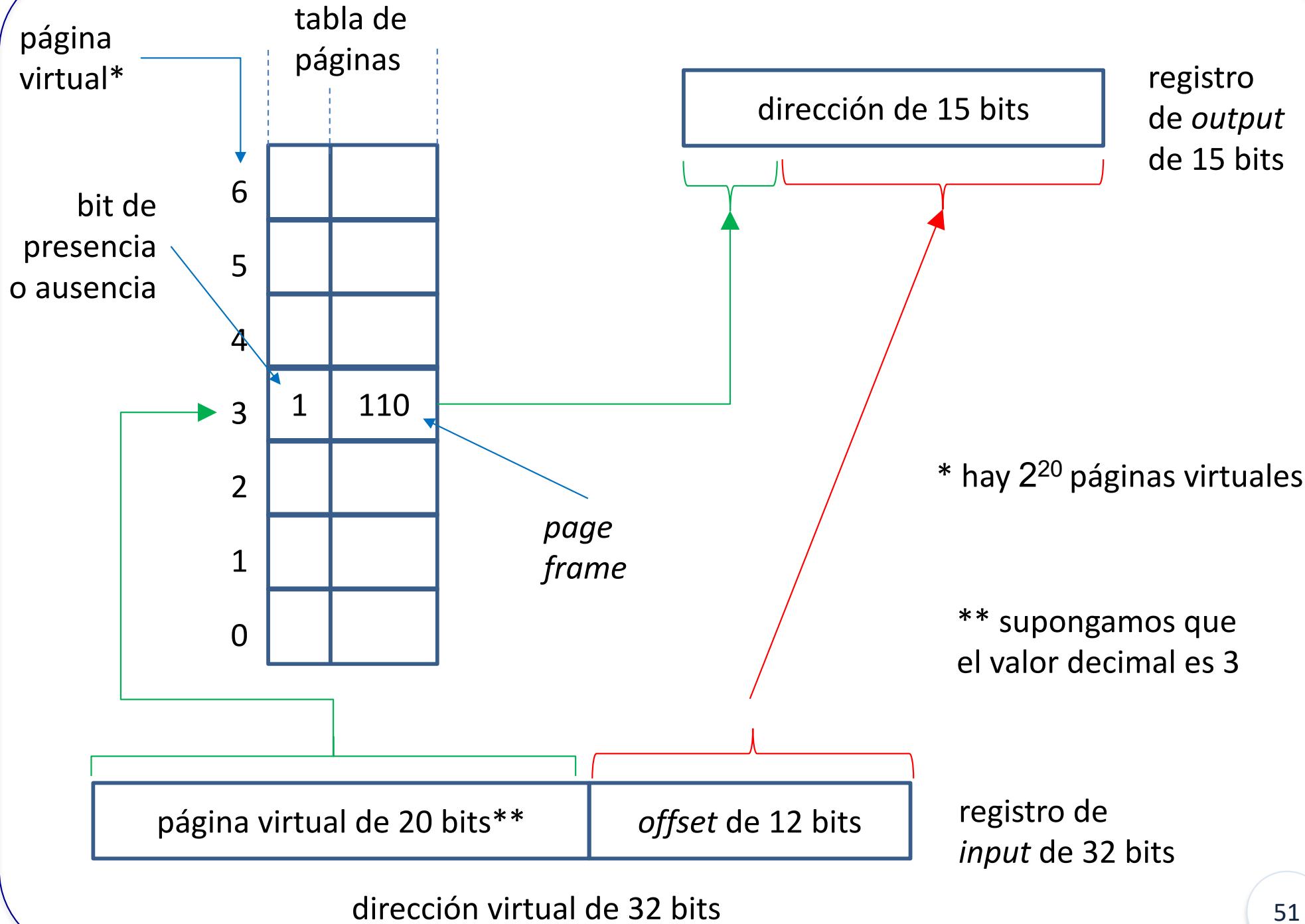
Pero, ¿está la página en memoria?

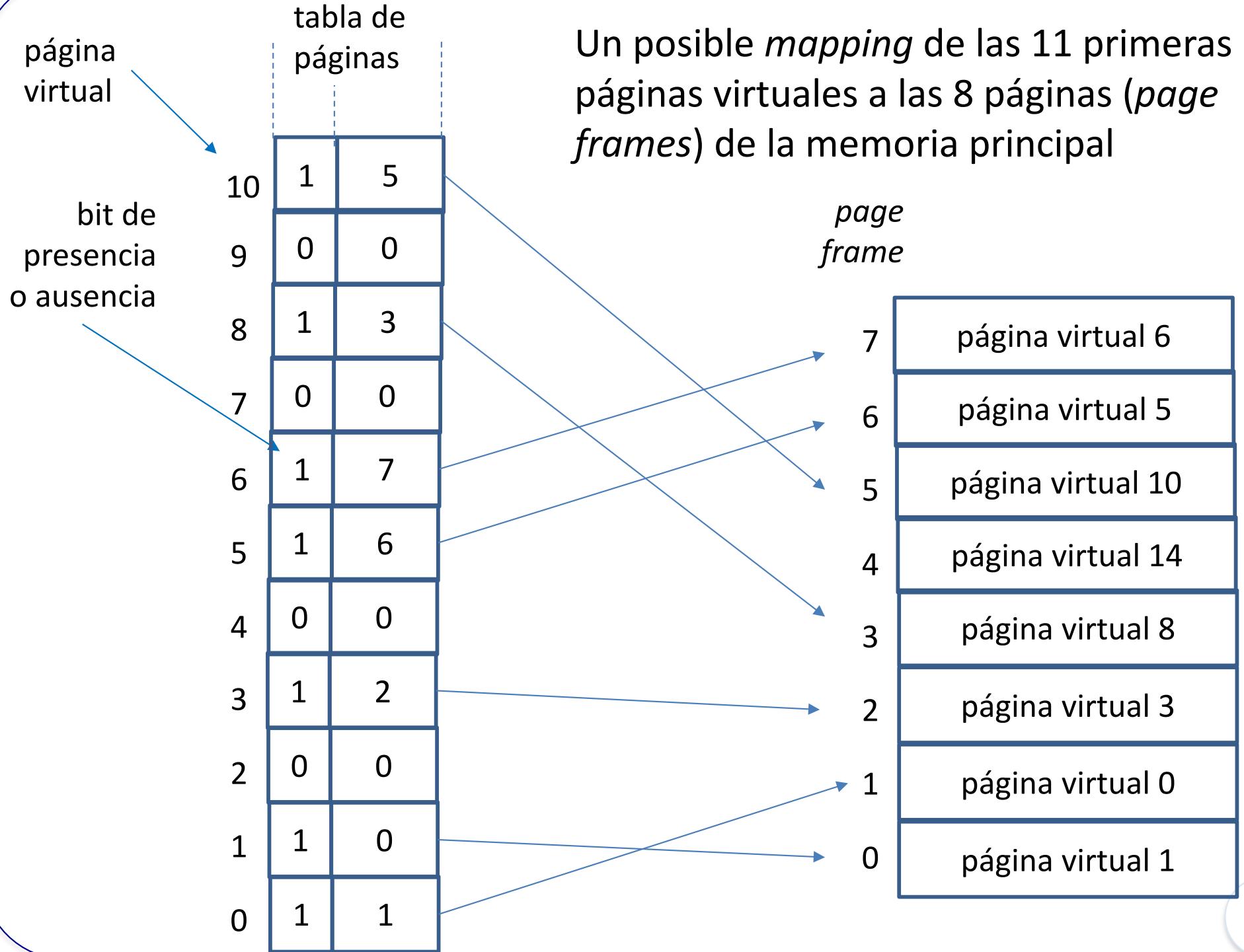
- hay  $2^{20}$  páginas virtuales y solo 8 páginas físicas (*page frames*)
- **bit de presencia(1)/ausencia(0)** —o *bit de validación*— en cada entrada en la tabla

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*** (en lugar de un “*cache miss*”)





## ¿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 de 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 (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 se vuelve crítica

## **Principio de localidad** (misma idea que en el caso de caches):

- 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

Elegir una página al azar no es una buena idea:

- si saliera la página que produjo el *page fault*  
... se va a producir otro *page fault* en cuanto se trate de leer la próxima instrucción

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*** a la tabla de páginas
  - ... el cual 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

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 para obtener la dirección física —la traducción
- un segundo acceso para obtener el dato

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

- cuando se hace una traducción de un número de página virtual, probablemente se va a necesitar de nuevo pronto

...

...

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

- cada entrada → *tag* con el número de la página virtual, número de la página física (o *page frame*), bit de validez y *dirty* bit
- 16 – 512 entradas, con líneas que contienen una o dos entradas de la tabla de páginas

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

- eliminando el contenido de la página que corresponda de la cache cuando decide migrar esa página al disco  
... y actualizando 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

| <b>TLB</b> | <b>Tabla de páginas</b> | <b>Cache</b> | <b>¿Es posible? ¿Bajo qué circunstancia?</b> |
|------------|-------------------------|--------------|--|
| 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: ...                              |

1961: automatización del proceso de *overlays* → **memoria virtual**:

- la memoria principal actúa como una “cache” para el almacenamiento secundario
- 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

## ¿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, 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 programa 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 programa, (los valores de) las variables del programa, el *stack* de registros de activación (de la ejecución de funciones)
- para multiprogramación, tenemos que permitir múltiples programas en memoria

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

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

... pero:

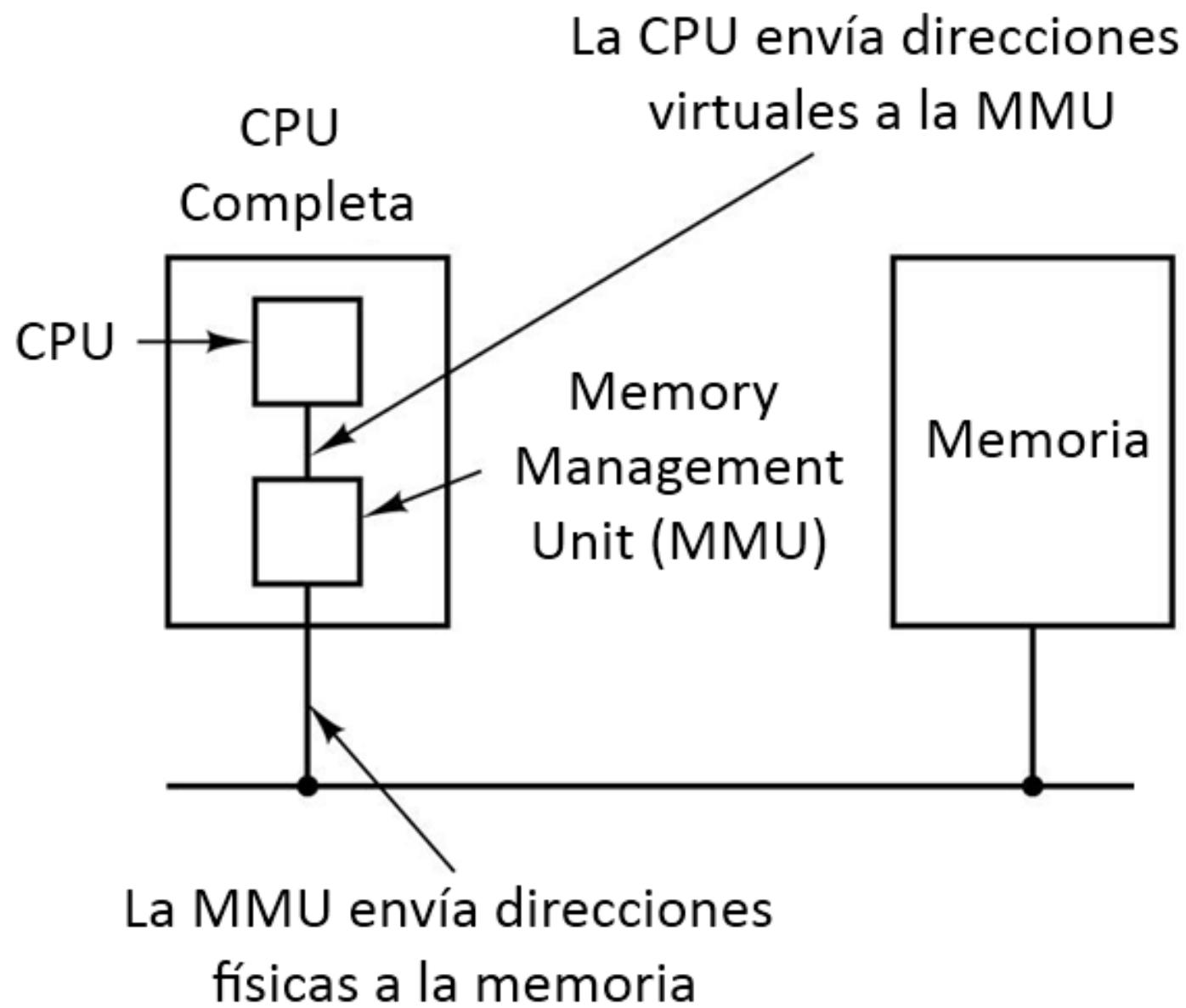
- el programador tendría que saber a priori cuál es la parte de la memoria que le va a corresponder a su programa
- si no hay un mecanismo de protección, es posible que un programa escriba en la parte de memoria asignada a otro programa
- el tamaño de la memoria asignada a cada programa debe ser fijo

## *La memoria virtual*

... cuyo funcionamiento ya conocemos como solución al problema de escasez de memoria para un programa

... soluciona también los problemas anteriores:

- cada programa 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

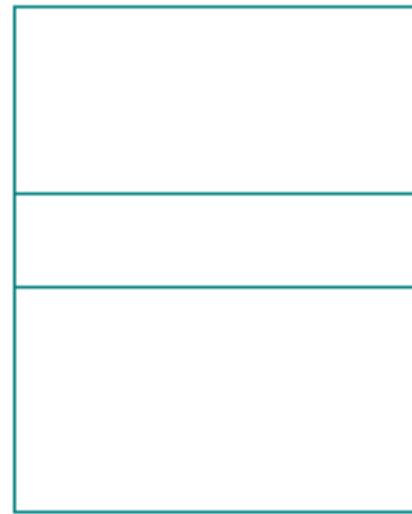
P1

100



P2

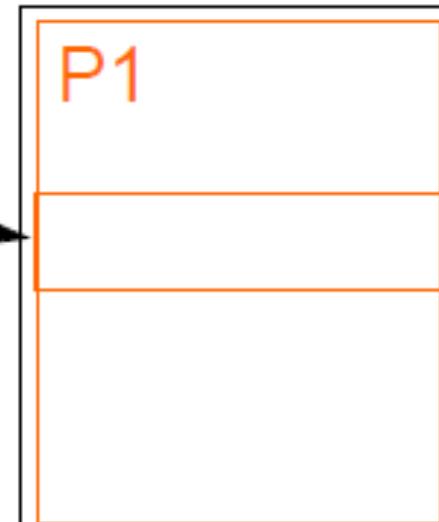
100



## Memoria física

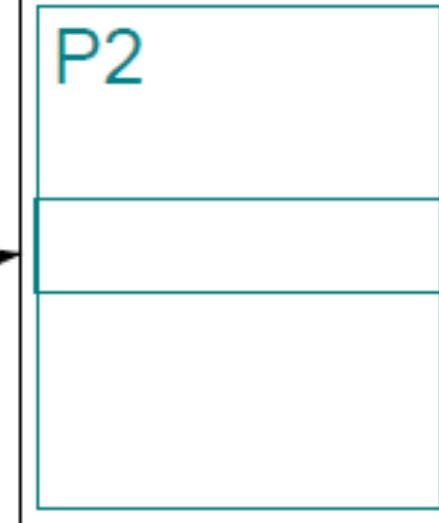
P1

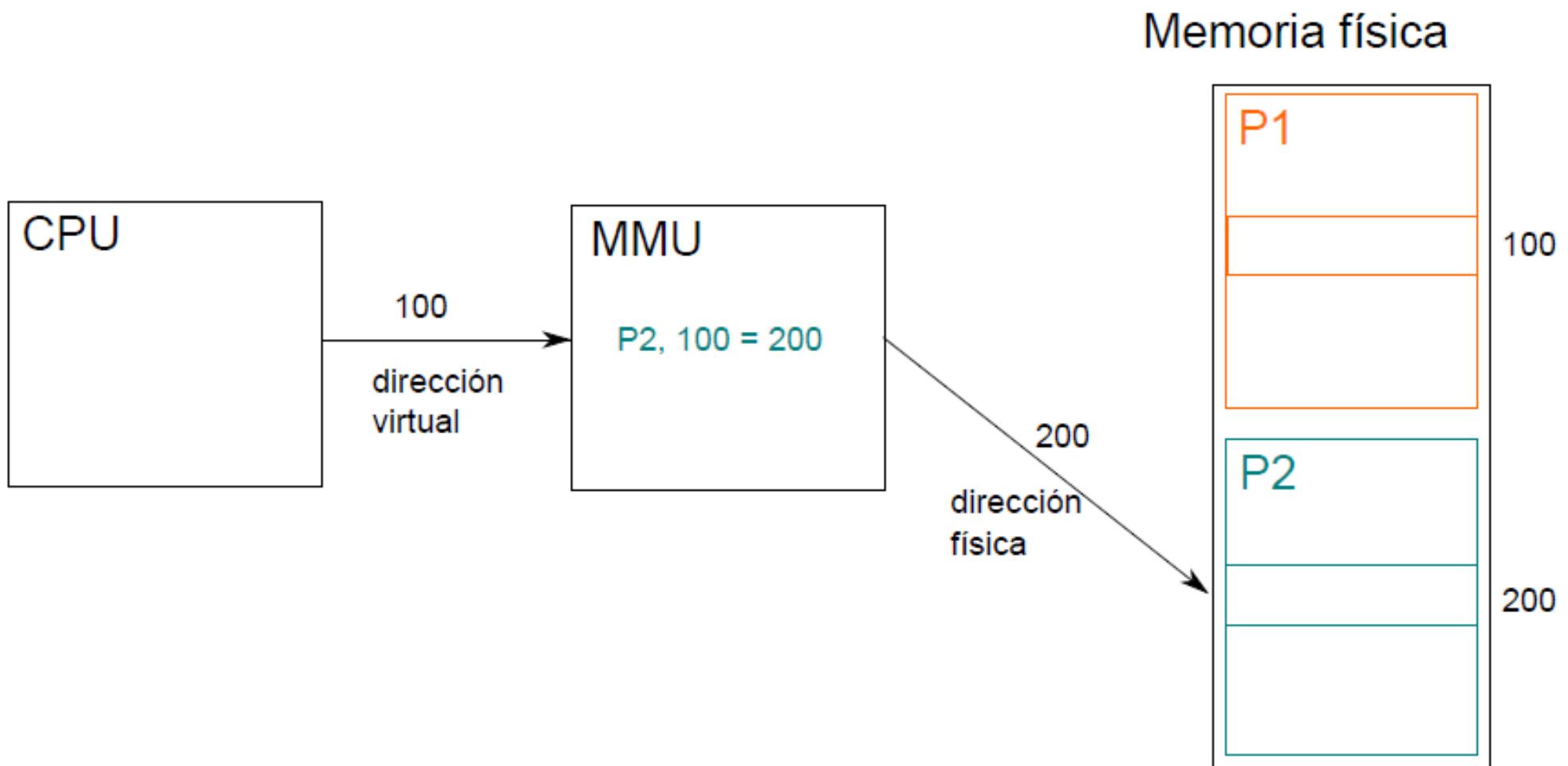
100



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 programa, una *tabla de páginas* especifica 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. subsiguiente muestra la tabla de páginas correspondiente a  $P_1$

## Memoria virtual

| Página virtual | Marco físico |
|----------------|--------------|
| 0              | 2            |
| 1              | 3            |
| 2              | 4            |

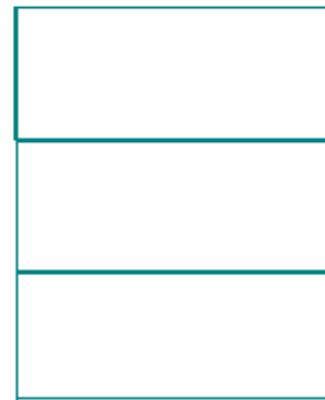
P1



## Memoria física

| Página virtual | Marco físico |
|----------------|--------------|
| 0              | 5            |
| 1              | 6            |
| 2              | 7            |

P2



| 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 programas necesitan páginas adicionales para poder continuar su ejecución:

- inicialmente, tanto  $P1$  como  $P2$  tienen 4 páginas cada uno en memoria principal (#75)
- entonces  $P1$  solicita una dirección que pertenece a una quinta página virtual, que no está en memoria → *page fault* (#76)
- una de las páginas en memoria de  $P1$  es elegida y enviada al disco → *swap out* (#77)
- ... en donde es puesta en el ***swap space*** correspondiente a  $P1$ , mientras que el *page frame* que ocupaba es ahora ocupado por la nueva página virtual (#78)
- la tabla de páginas de  $P1$  refleja esta situación (#79)

...

## Memoria virtual

P1



P2

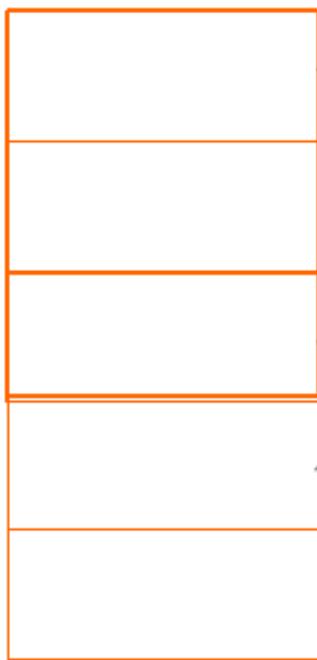


## Memoria física

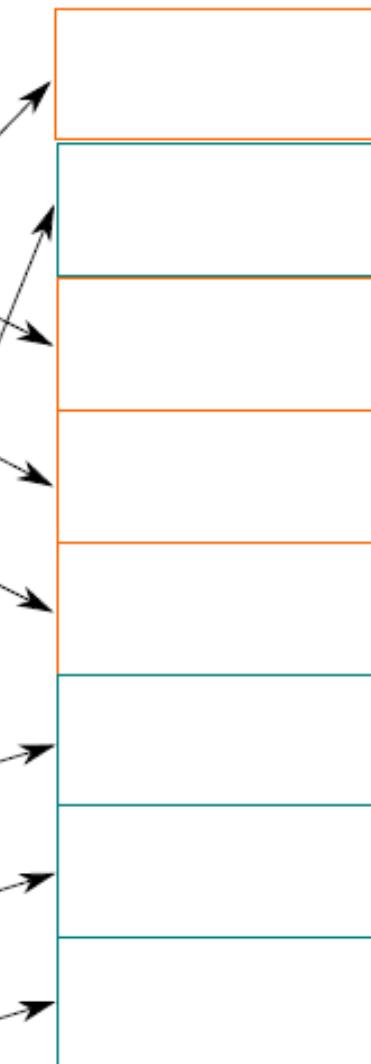


## Memoria virtual

P1



## Memoria física



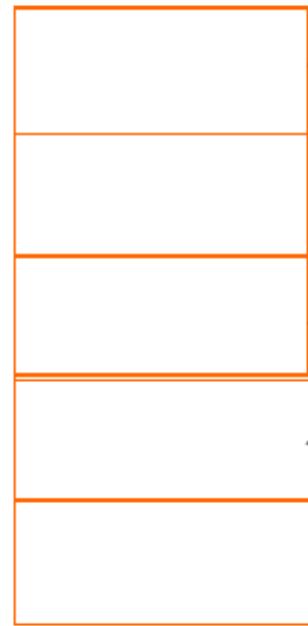
P2



page fault

Memoria virtual

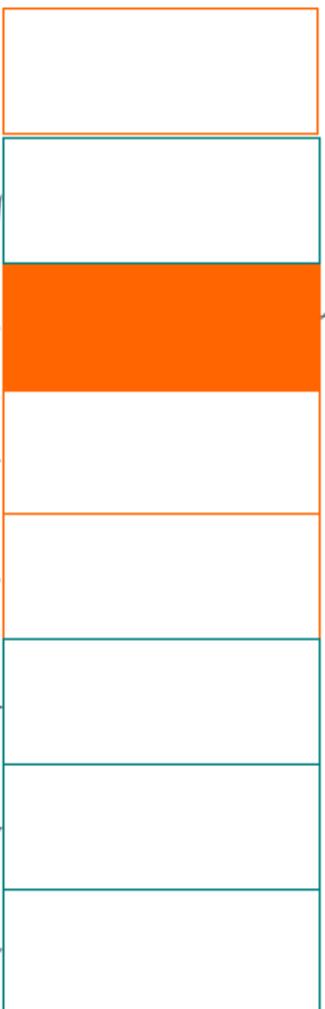
P1



P2



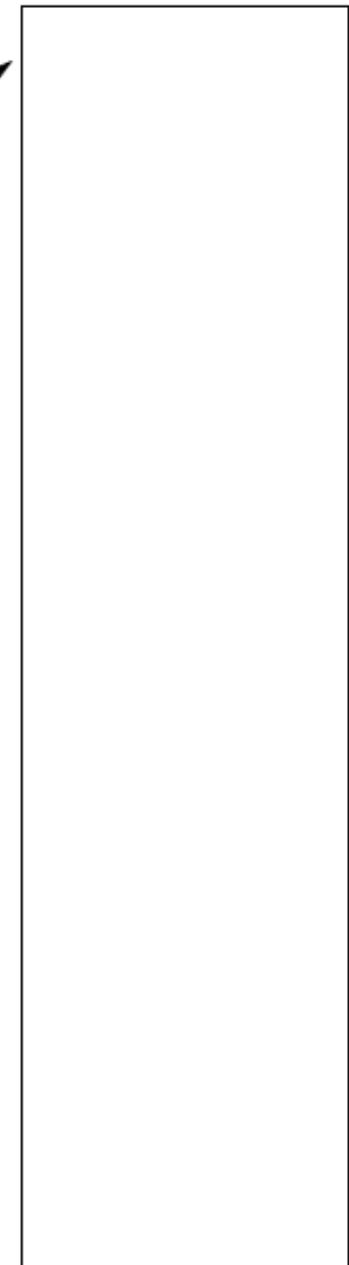
Memoria física



Disco: Swap File

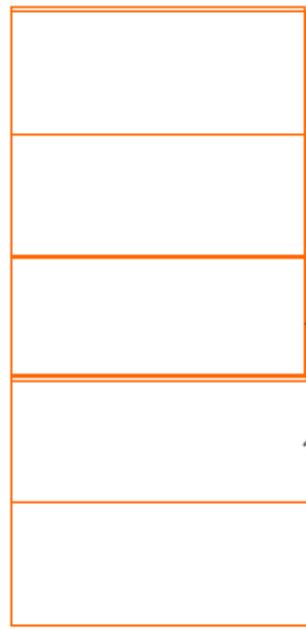
7  
8  
1  
2  
3  
4  
5  
6

swap out



Memoria virtual

P1



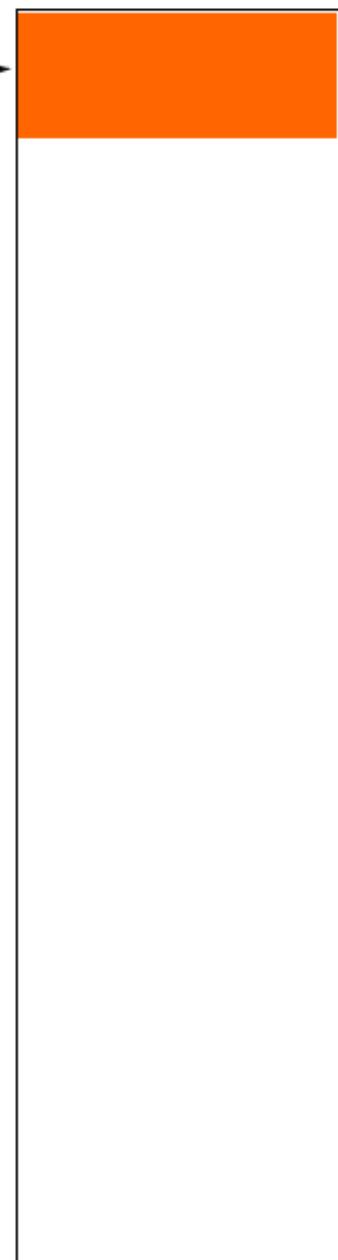
P2



Memoria física



Disco: Swap File



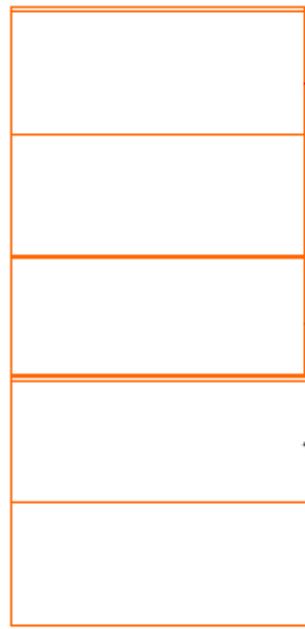
| Página virtual | Marco físico | Validez | Disco |
|----------------|--------------|---------|-------|
| 0              | 2            | 1       | 1     |
| 1              | 3            | 1       | 0     |
| 2              | 4            | 1       | 0     |
| 3              | 0            | 1       | 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* (#81)
- ... que obliga a que otra página de  $P_1$  en memoria sea enviada al *swap space* (#82)
- ... 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*, #83)
- lo que implica una nueva actualización de la tabla de páginas de  $P_1$  (#84 y 85)

Memoria virtual

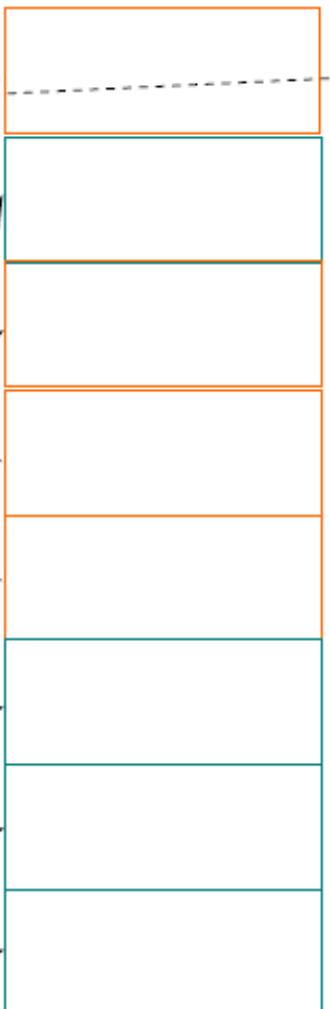
P1



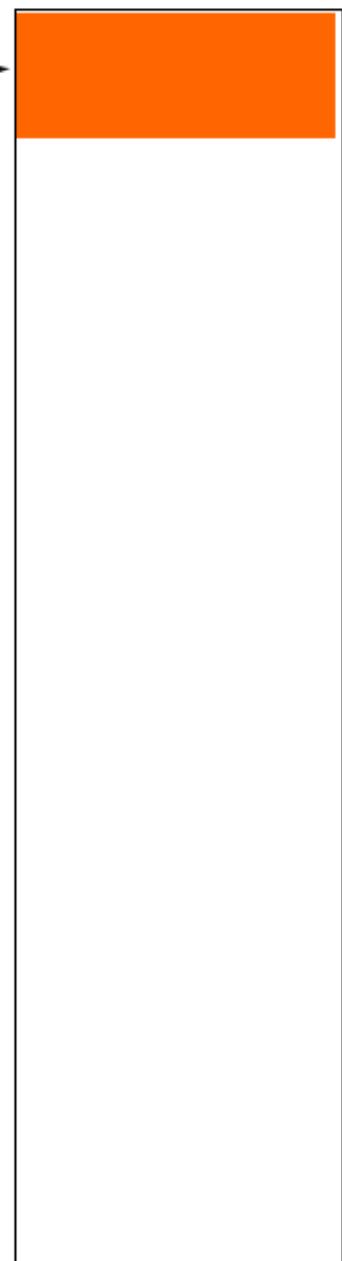
P2



Memoria física



Disco: Swap File



Memoria virtual

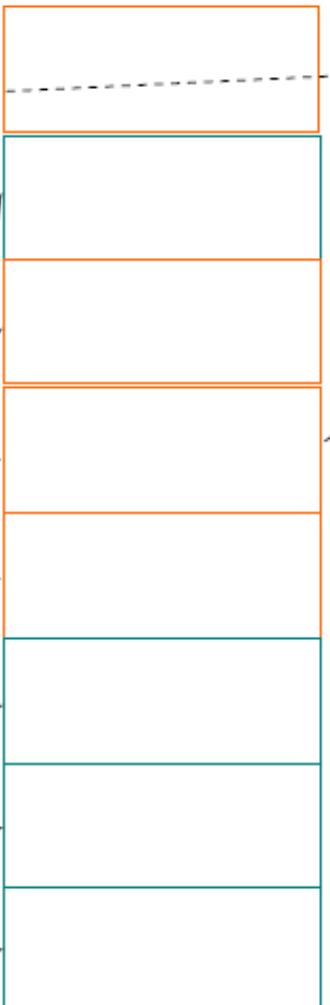
P1



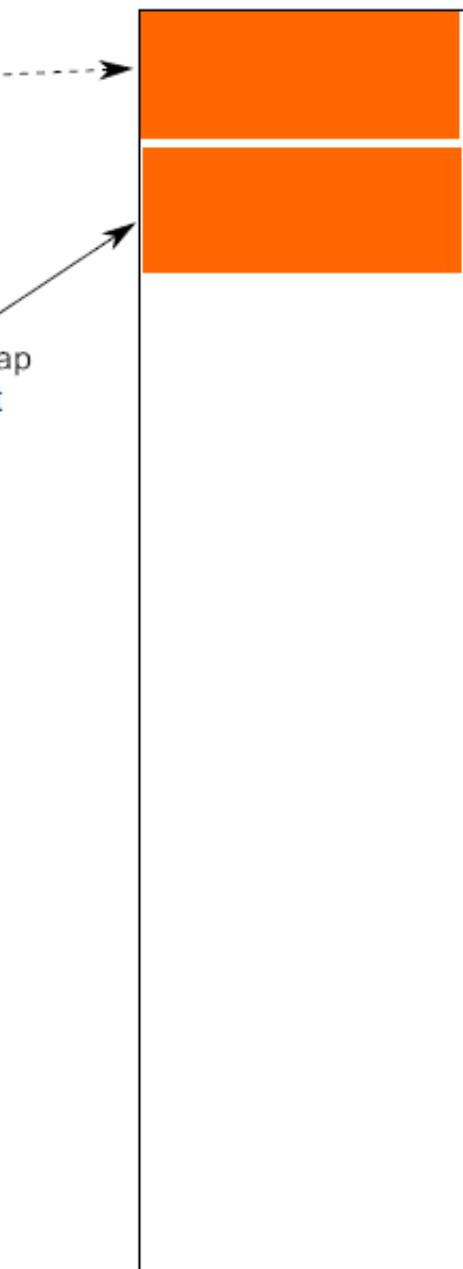
P2



Memoria física

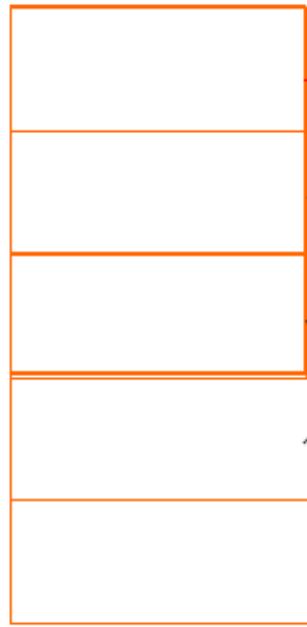


Disco: Swap File

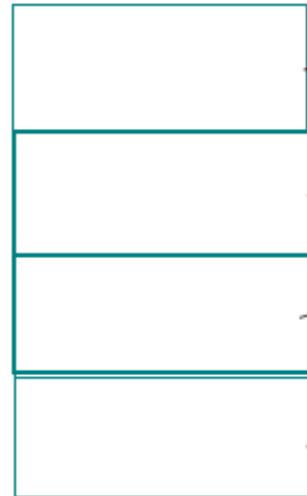


Memoria virtual

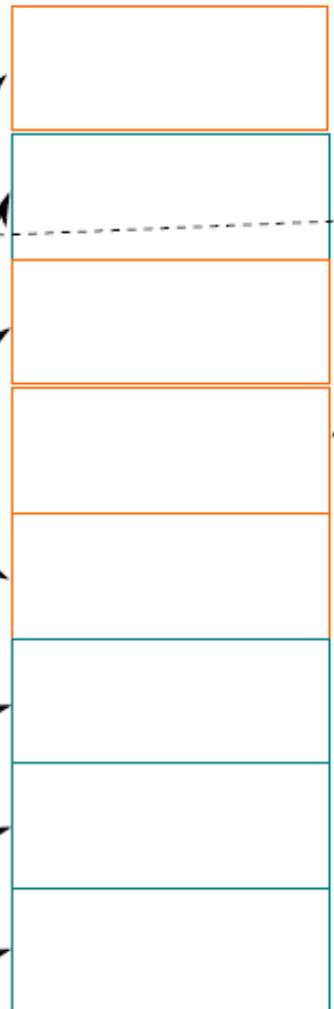
P1



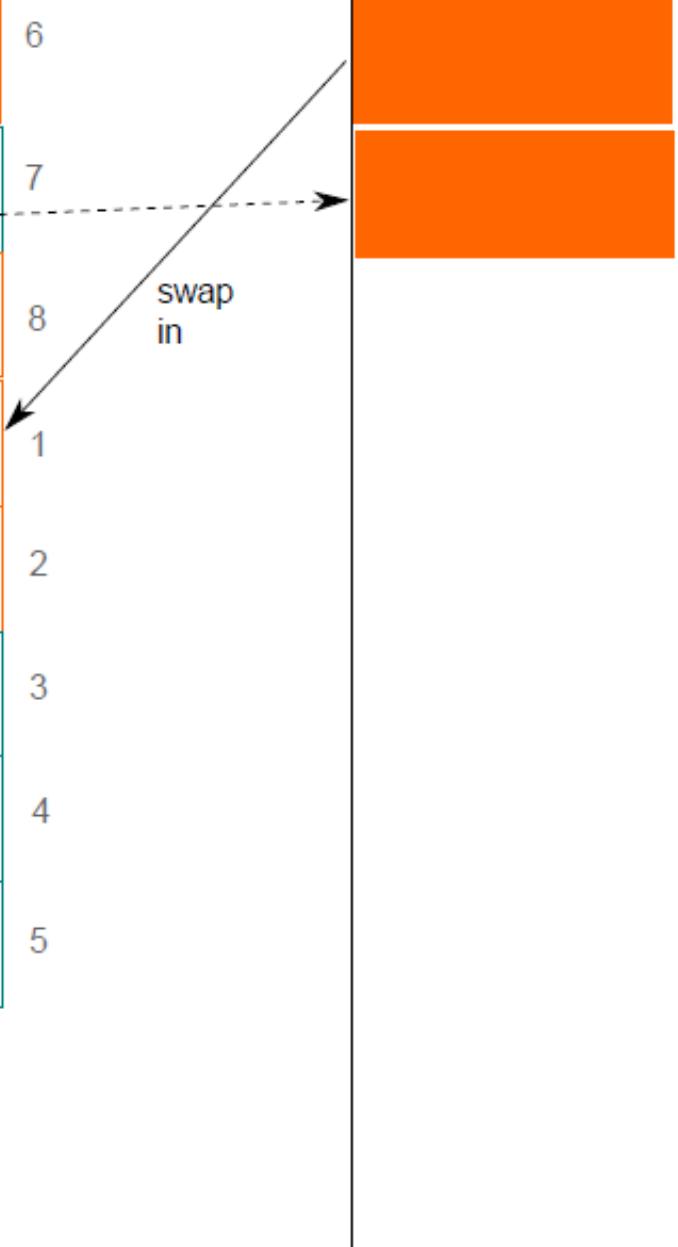
P2



Memoria física



Disco: Swap File



Memoria virtual

P1



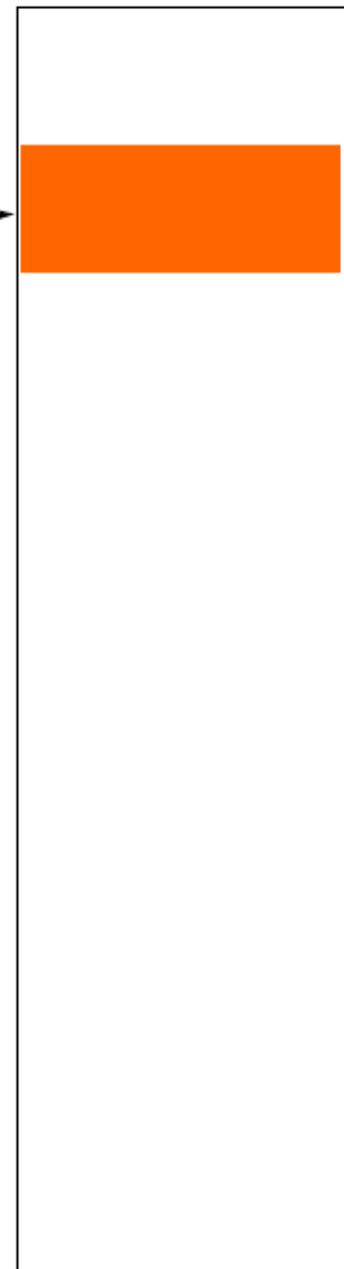
P2



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     |