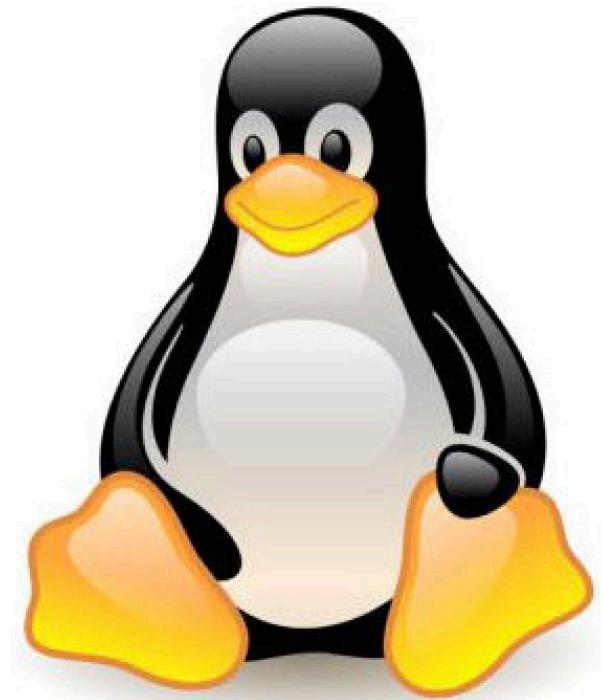


Tema 3: Gestión de memoria

Sistemas Operativos

ETSI Informática. Universidad de Málaga



Manuel Ujaldón

Catedrático de Arquitectura de Computadores
Departamento de Arquitectura de Computadores
Universidad de Málaga

Índice [66 diapositivas]

I. Introducción. [5]

1. Fundamentos.
2. Tecnologías.
3. Evolución de la DRAM.
4. Librerías de enlace dinámico (DLLs).
5. Ubicación de un proceso en DRAM.

II. Swapping. [7]

1. Conceptos.
2. Visión global.
3. Asignación del espacio en memoria: First-fit, best-fit y worst-fit. [3]
4. Fragmentación externa e interna. [2]

III. Paginación. [19]

1. Páginas y marcos. [2]
2. Traducción de direcciones lógicas a físicas. [4]
3. La TLB. [2]
4. Compartición y protección de páginas. [2]
5. Estructura de la tabla de páginas. [9]
 1. Conceptos. [1]
 2. Tabla de páginas jerarquizada en niveles. [5]
 3. Tabla de páginas invertida. [3]

Índice (cont.)

IV. Segmentación. [4]

1. Concepto.
2. Implementación.
3. Traducción de dirección lógica a física.
4. Segmentación paginada.

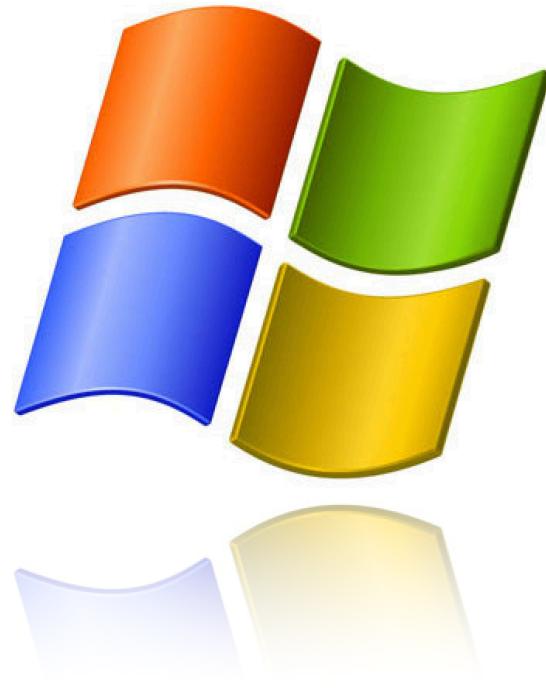
V. Memoria virtual. [21]

1. La idea de memoria virtual.
2. Ejemplo de memoria virtual segmentada.
3. Paginación bajo demanda.
4. Tabla de páginas de la memoria virtual.
5. Traducción de dirección virtual a física y gestión de las faltas de página.
6. Rendimiento de la paginación bajo demanda y thrashing. [4]
7. Algoritmos de reemplazo de páginas: FIFO, LRU, óptimo. Implementaciones. [12]

VI. Optimizaciones [8]

1. Alojamiento global frente a local.
2. El modelo del conjunto de trabajo.
3. Frecuencia óptima de faltas de página.
4. Prebúsqueda de páginas en disco.
5. Anclaje de páginas en memoria física.
6. El tamaño ideal de página.
7. Geometría de los accesos a memoria. [2]

VII. Resumen y bibliografía [2]



I. Introducción

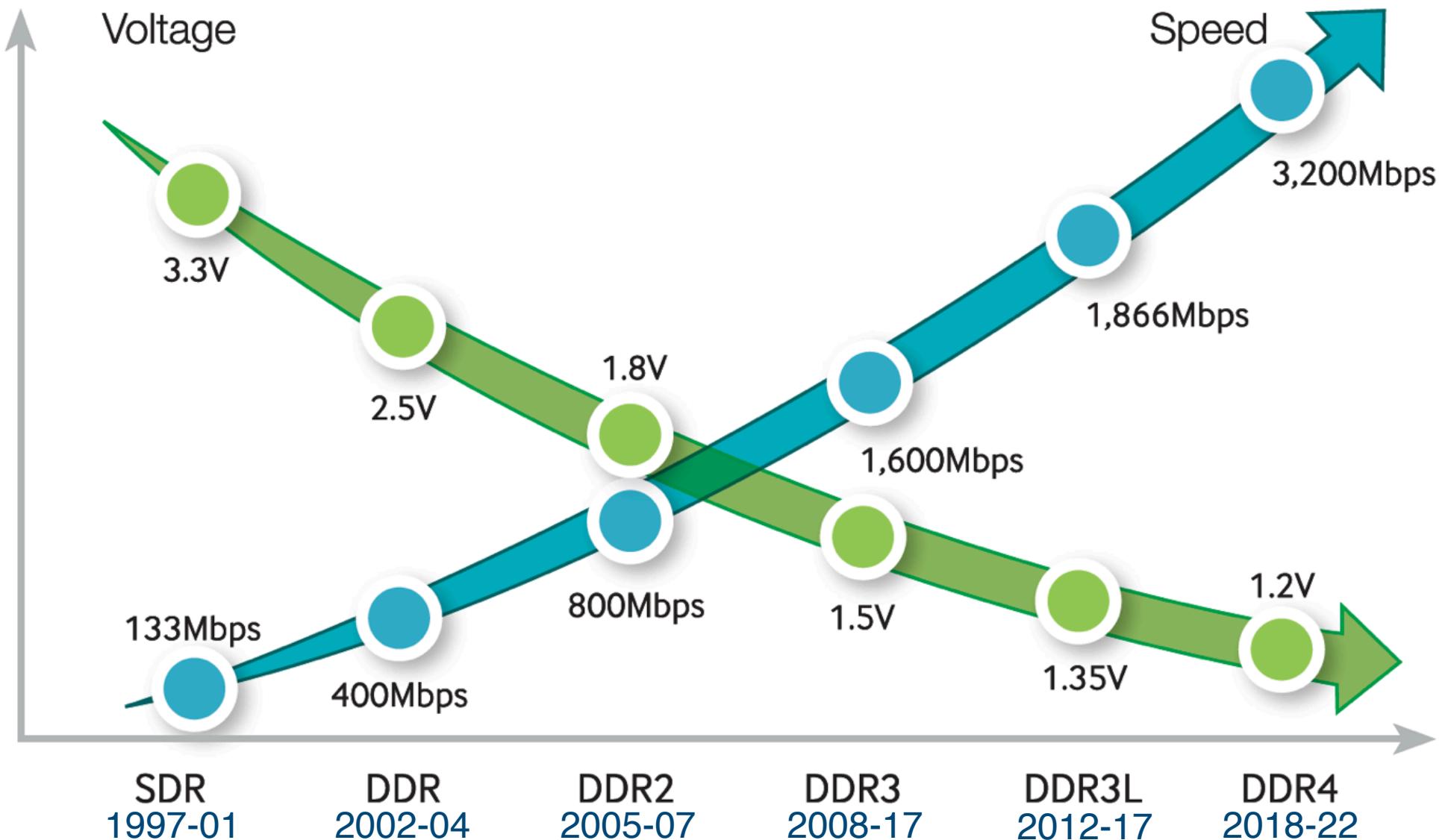
Fundamentos

- El Sistema Operativo puede asignar a un programa una cantidad de memoria muy superior a la disponible en DRAM. Para ello, utiliza el disco como una extensión donde guarda las áreas menos necesarias para la ejecución del programa.
- Memoria y disco intercambian así datos de forma continua, en un proceso denominado *swapping* o intercambio.
- Para agilizar el tiempo de acceso, la memoria DRAM realiza un proceso de selección similar en dirección a la CPU, alojando en caché la información más utilizada.
- Se completa así una jerarquía con 3 niveles de memoria: estática (SRAM, de transistores), dinámica (DRAM, de condensadores) y disco (de estado sólido o magnético).

La memoria DRAM y las tecnologías que le arropan más allá como *swapping*

Tipo de memoria	Uso en el PC	Latencia en microsegundos	Ancho de banda (MB/sg.)	Precio medio aproximado (€/GB)
DRAM (celdas de memoria dinámica)	Módulos de memoria principal	0.012 - 0.030	25000	5
Flash (no volátil)	BIOS y pendrives (almacenamiento a pequeña escala)	25 - 50	15	0.25
SSD (disco de estado sólido)	Almacenamiento masivo sin partes mecánicas/móviles	25 - 50	500	0.1
HD (disco duro magnético)	Almacenamiento en cilindros accesibles mediante cabezales	3-6 milisegundos (5.5 msg. @ 5400 RPM) (3 msg. @ 10000 RPM)	250	0.05

DRAM evoluciona según la saga DDR, a mayor ancho de banda y menor consumo



Carga y enlace dinámico de librerías en memoria (DLLs)

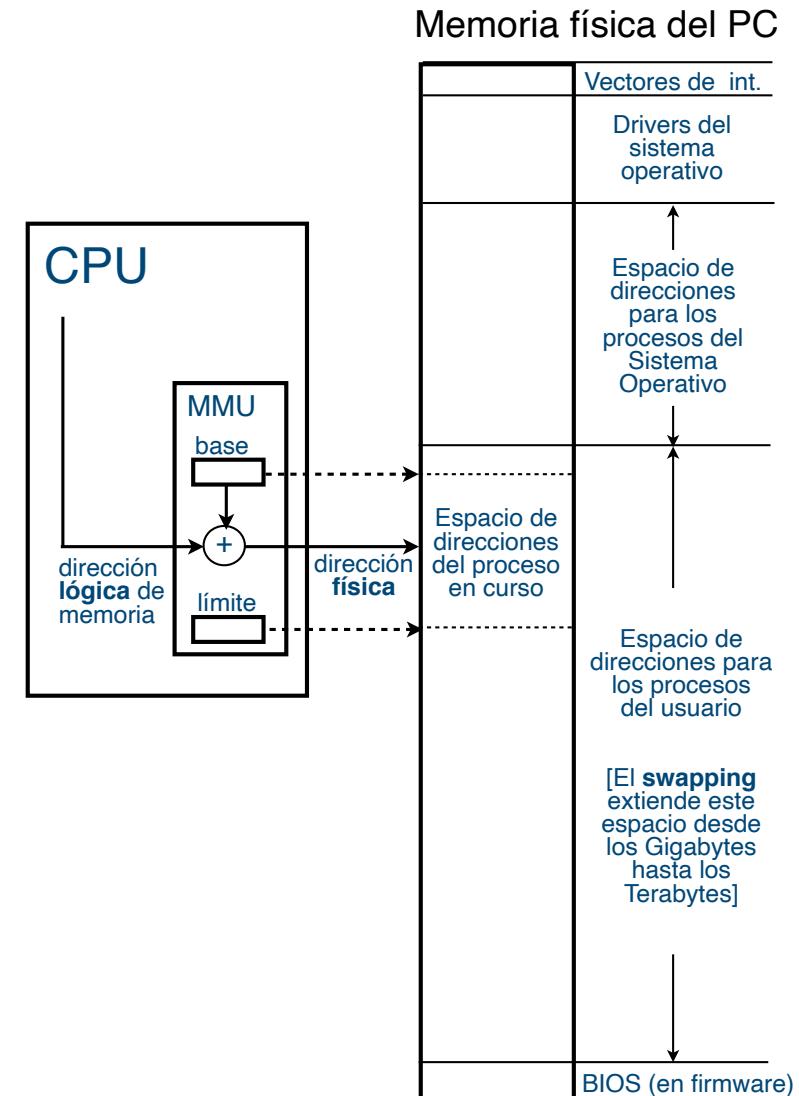
- Permiten al S.O. demorar la carga de las librerías usadas por el programa hasta el mismo momento en que se llaman.
- Optimiza el uso de la memoria: Sólo se cargan las rutinas de la librería que son finalmente utilizadas por el programa.
- Resulta especialmente útil cuando la librería es enorme y pretendemos usar una pequeña parte de ella y/o usarla de forma infrecuente.
- Un pequeño código, el *stub*, hace de sustituto para localizar la rutina residente en memoria y reemplazarse por su dirección en tiempo de ejecución.
- Muy popular en los S.O. Windows de la última década.

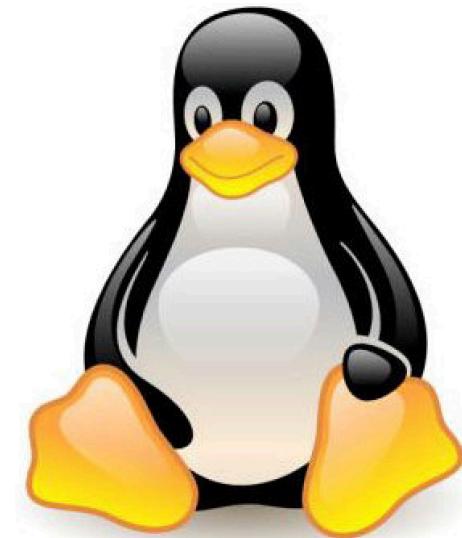
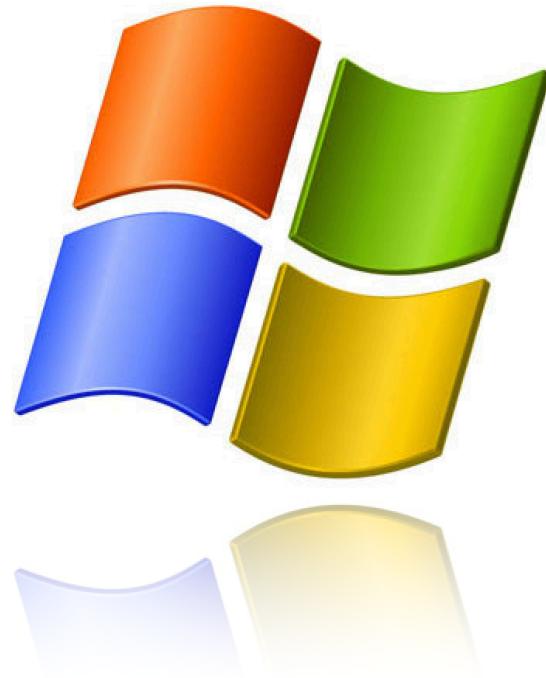
Ubicación de un proceso en memoria

● Cada proceso define su espacio de direcciones lógico o virtual mediante dos registros que albergan:

- La primera dirección (o base).
- Su tamaño a partir de ésta (o límite).

● El usuario ve siempre este espacio lógico, que el S.O. traduce en espacio físico (el real) durante la ejecución del proceso a través de la **MMU (Memory Management Unit)** de la CPU para gestionar estos registros por hardware dedicado.



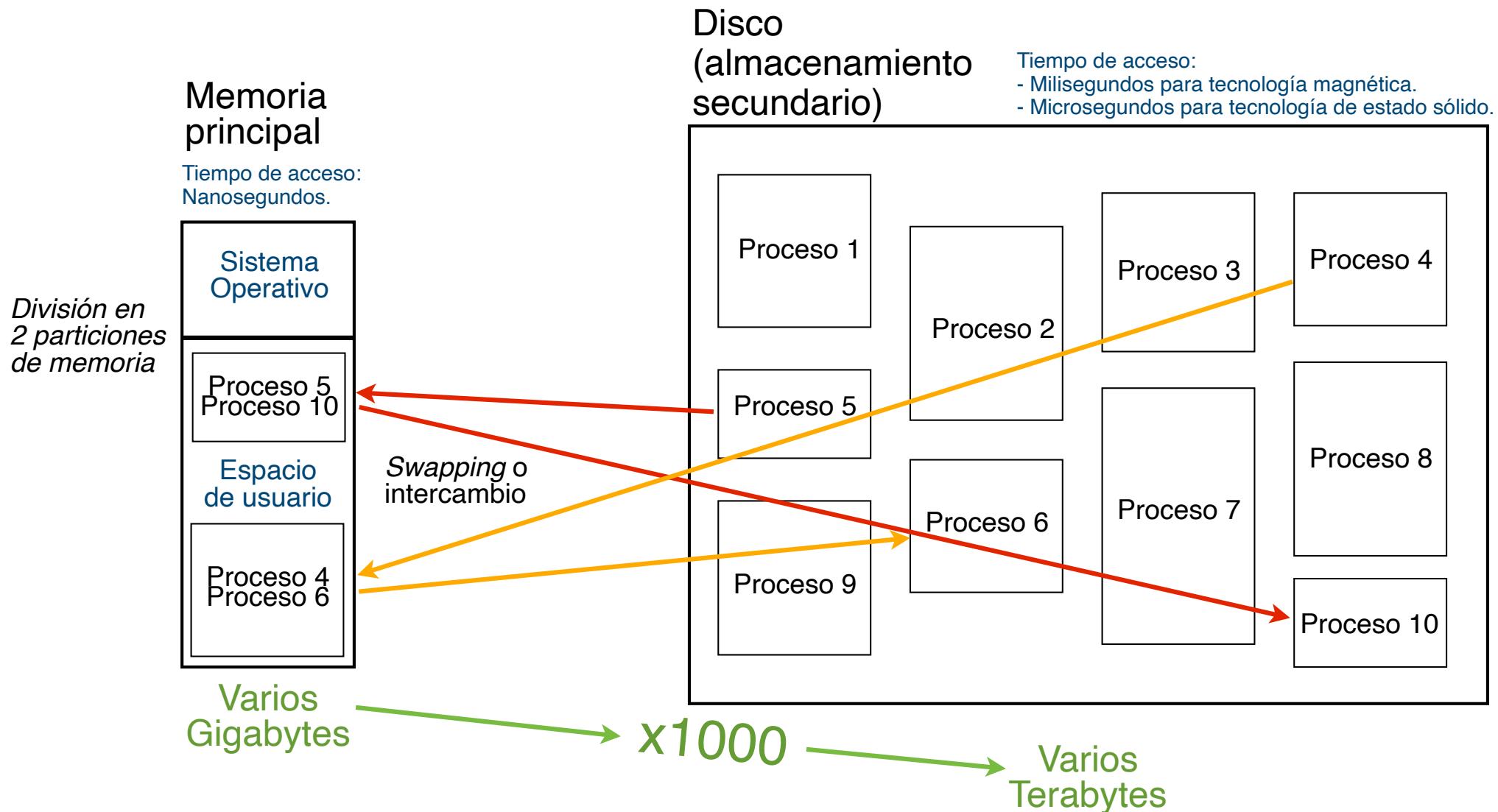


II. Swapping

Swapping. Conceptos

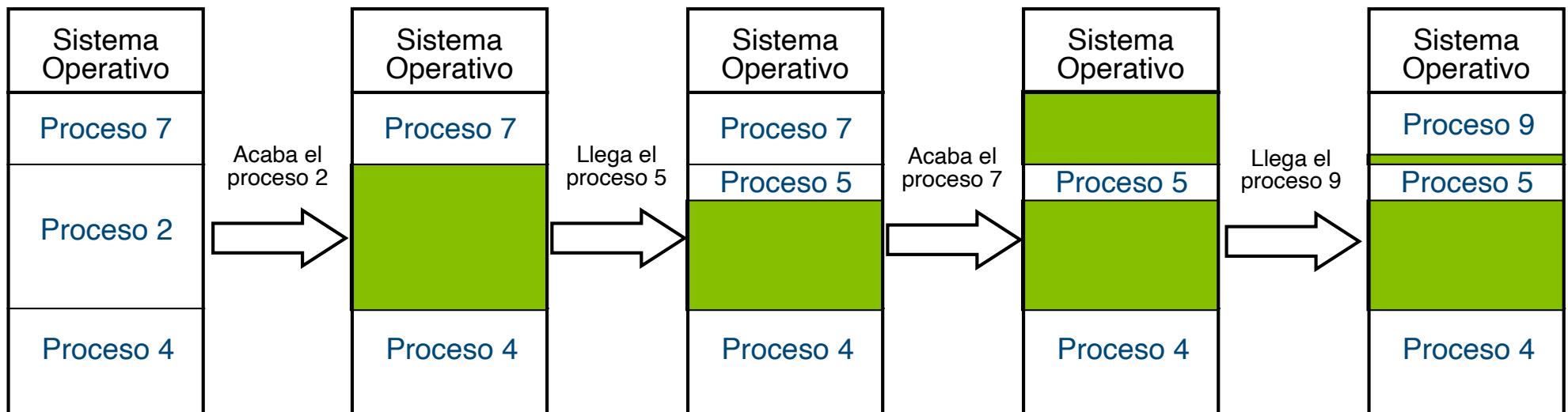
- Sólo la memoria más usada de los procesos más activos permanecerá en DRAM. El resto queda en disco, y cuando se necesite, se reemplaza por otra que se decida relegar.
- La memoria se va moviendo así entre DRAM y disco bajo demanda o prioridad, y el usuario percibe la ilusión de que dispone de un espacio de memoria muy superior al real.
- El intercambio entre DRAM y disco debe producirse para tamaños grandes que aprovechen el gran ancho de banda.
- Todos los S.O. modernos habilitan *swapping*, puesto que la DRAM es mucho más cara que el disco.
- Lo esencial es acertar con las áreas de memoria a priorizar en DRAM y relegar al disco a modo de repositorio.

Visión global del proceso de swapping



Asignación del espacio en memoria: Alojamiento contiguo

- La memoria se gestiona como un espacio consecutivo de direcciones físicas en el que se van intercalando procesos y huecos libres.
- Cuando llega un proceso se busca un hueco para él, y cuando sale un proceso se libera su hueco.
- El S.O. debe gestionar el espacio ocupado y los huecos.

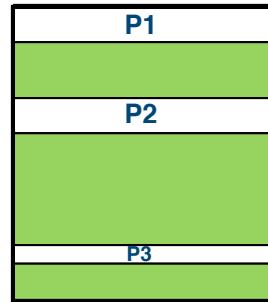


Estrategias para asignar los huecos libres a los procesos que van solicitando memoria

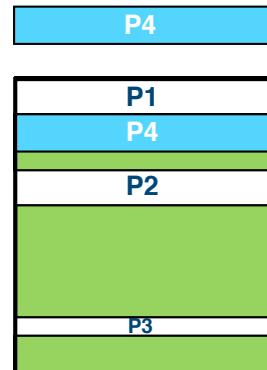
- La primera idea que viene a la mente:
 - **First-fit:** Asignar el primer hueco disponible lo suficientemente grande.
- También podemos recorrer toda la secuencia de huecos para:
 - **Best-fit:** Buscar el hueco que mejor se ajuste, dejando residuos mínimos.
 - **Worst-fit:** Buscar el hueco más grande disponible, dejando residuos máximos.
- Ambas requieren recorrer toda la lista, salvo que tengamos los huecos ordenados por tamaño.
- Comparativa:
 - First-fit es el más sencillo y rápido, y bastante eficiente.
 - Best-fit no es tan simple, pero sí el más eficiente.
 - Worst-fit es el más lento y el que peor gestiona la memoria.

Ejemplo: Hay 3 procesos y llegan otros 3

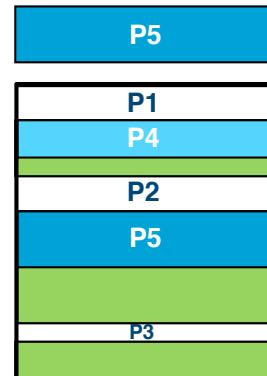
First-fit:



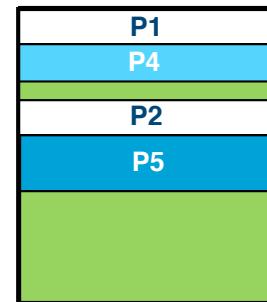
Llega P4:



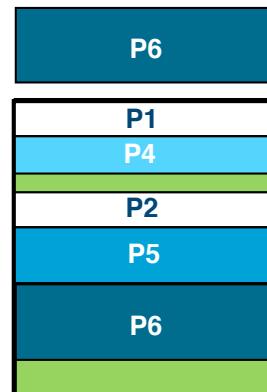
Llega P5:



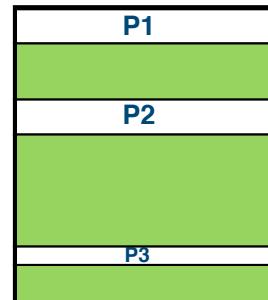
Sale P3:



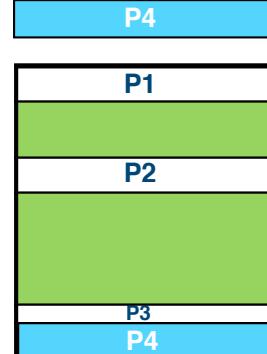
Llega P6:



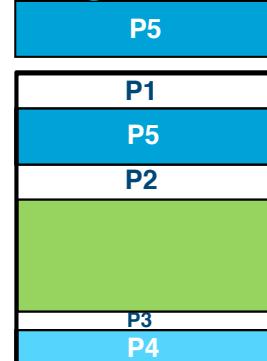
Best-fit:



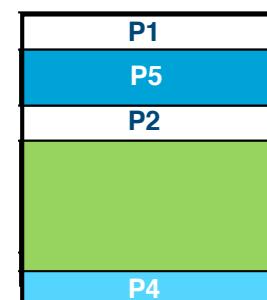
Llega P4:



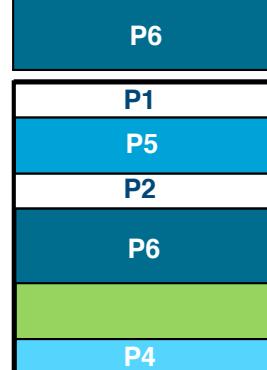
Llega P5:



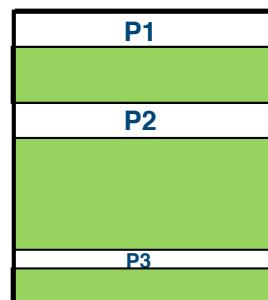
Sale P3:



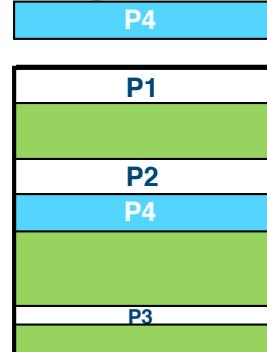
Llega P6:



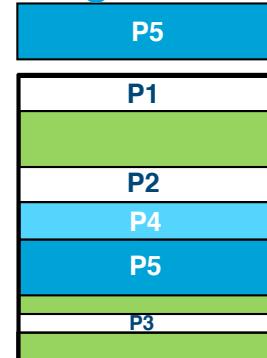
Worst-fit:



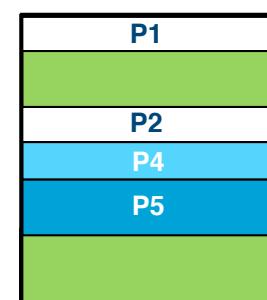
Llega P4:



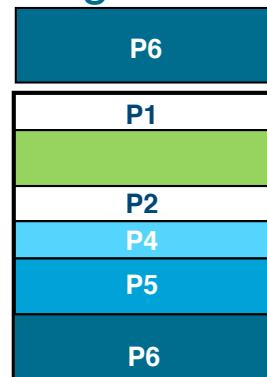
Llega P5:



Sale P3:



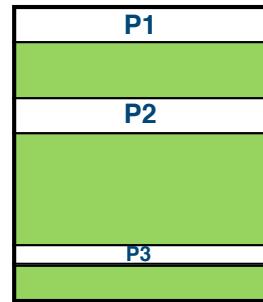
Llega P6:



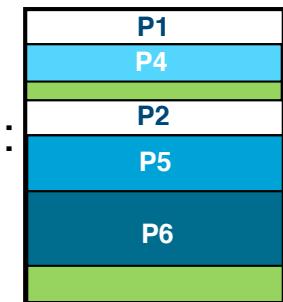
Fragmentación externa

- Se produce por los huecos libres que van quedando en memoria, cada vez más pequeños, como consecuencia de la continua llegada y salida de los procesos. Fijémonos en el ejemplo anterior de first-fit:

Estado inicial:



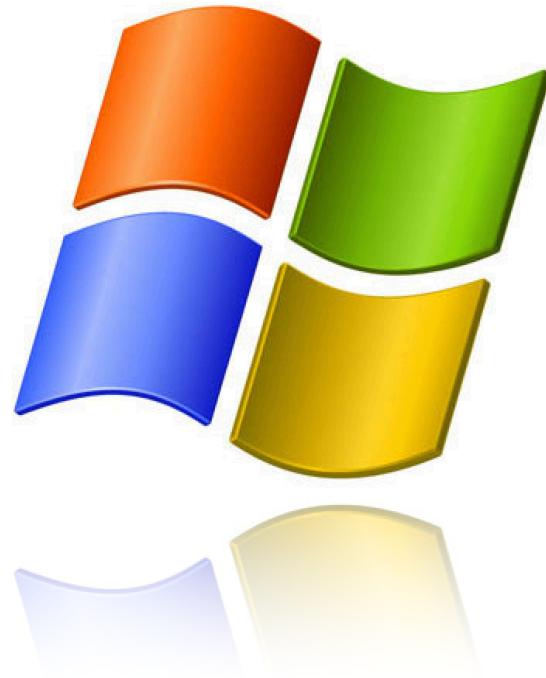
Estado final:



- Si ahora llegase un proceso del tamaño de P5, no podría alojarse de forma contigua a pesar de que hay memoria suficiente (su tamaño azul es igual a los dos huecos verdes).
- Este fenómeno también se produce en el espacio de disco, y el S.O. lo mitiga lanzando un proceso defragmentador de forma periódica que agrupa los huecos en un único espacio.

Fragmentación interna

- Para su direccionamiento eficiente, el espacio de memoria se gestiona en múltiplos de un tamaño fijo denominados **páginas** (por ejemplo, de 16 KB). Si un proceso necesita 40 KB, el S.O. le asigna 3 págs. (48 KB), y se pierden 8 KB por fragmentación interna.
- Reduciendo el tamaño de página a 8 KB solucionaremos el problema, pero entonces aparecerán otros dos:
 - El número de páginas se duplica, por lo que sus direcciones son más grandes.
 - La memoria DDR4 tiene un gran ancho de banda, pero a costa de incurrir en una latencia muy alta. Es decir, se tarda mucho en acceder a la primera palabra de la página, pero a partir de ahí, las demás salen muy rápido. Esto se aprovecha mejor con páginas grandes.



III. Paginación

La idea de la paginación

- Según evolucionan los procesos en memoria, van siendo acotados por otros y a la vez creciendo, requiriendo nuevas áreas de memoria no contiguas para esta expansión.
- El proceso es simétrico con los ficheros en el espacio de disco. En ambos casos, el S.O. habilita un esquema de direccionamiento versátil basado en unidades atómicas de almacenamiento que son asignadas y liberadas.
- Estas unidades son las **páginas** de memoria y los sectores de disco, y para su gestión, el S.O. emplea tablas de páginas y tablas de asignación de ficheros (FAT), respectivamente.
- Este capítulo lo dedicaremos a las tablas de páginas. El siguiente veremos las FAT y sus alternativas en UNIX.

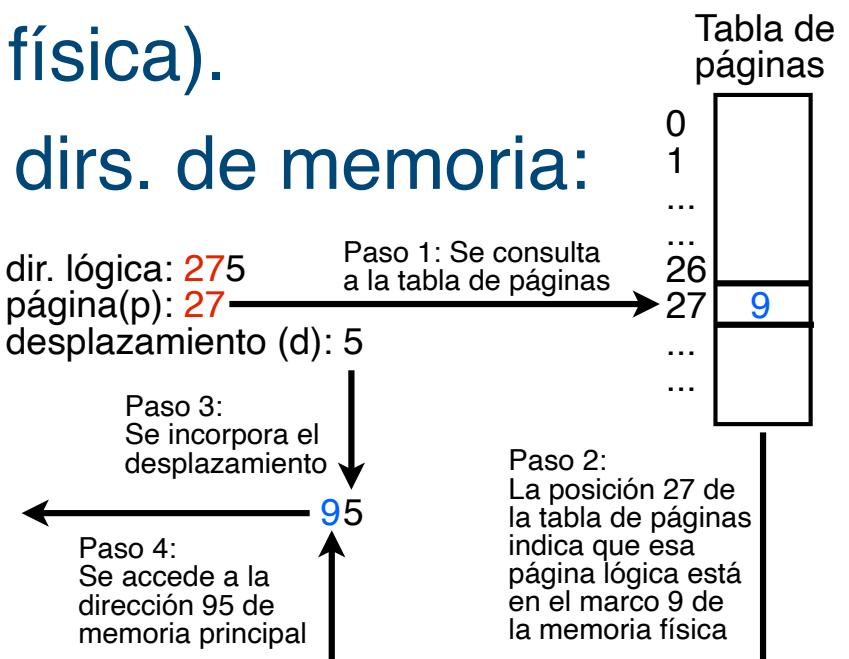
Páginas y marcos

- El conjunto de páginas ocupado por un proceso es su espacio lógico de direcciones. A diferencia del espacio físico de los ejemplos anteriores, el espacio lógico:
 - Es siempre consecutivo para un proceso.
 - Suele ser muy superior al espacio físico disponible (utilizaremos para ello la memoria virtual más adelante).
- Cada página de un proceso debe encontrar su hueco en memoria física, al que llamaremos **marco** (del inglés, *frame*). Esto requiere habilitar un mecanismo para la traducción de páginas lógicas a físicas, y dentro de ellas, cada dirección de memoria tendrá un desplazamiento (*offset*) diferente.
- El S.O. gestiona también los marcos del espacio libre.

Traducción de direcciones lógicas a físicas

- El mapeo entre las páginas de los procesos y los marcos de la memoria física se establece con una tabla de páginas.
- La traducción se completa agregando el desplazamiento relativo de la dirección a traducir, esto es, su distancia desde el comienzo de la página (coincidente con la del marco en la dirección de memoria física).
- Ejemplo para 100 páginas de 10 dirs. de memoria:

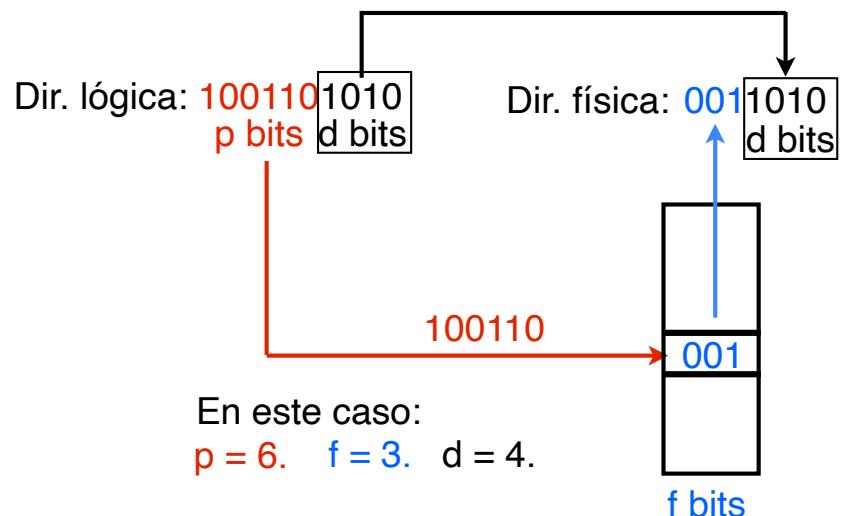
- La dirección de memoria 275 será la dirección 5 dentro de la página 27.
- Si la página 27 está en el marco 9 de memoria física, la dirección lógica 275 estará en la dirección física 95.



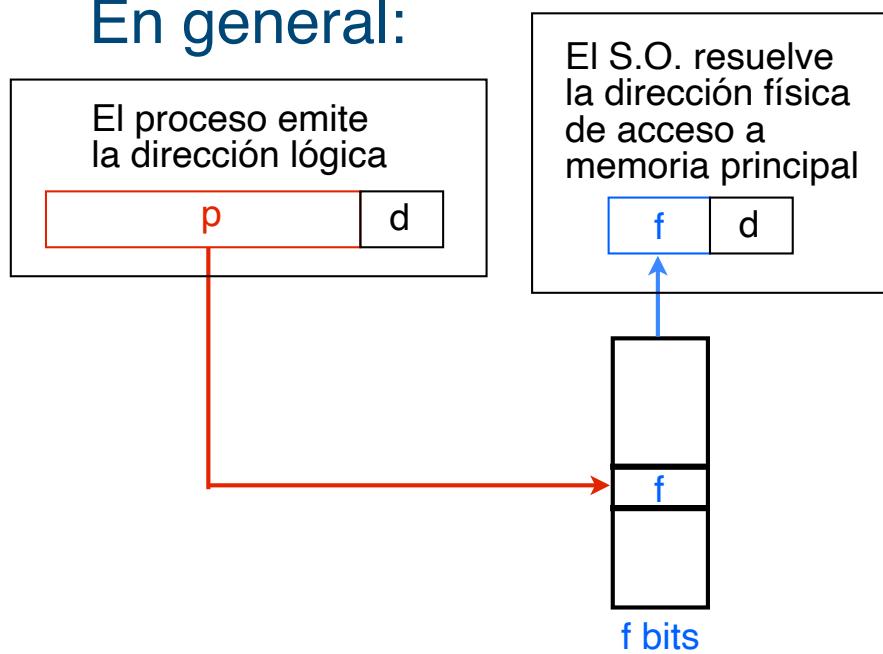
Ahora en binario

- Para un espacio lógico de 2^p páginas de 2^d dirs. y una memoria física de 2^f páginas (también de 2^d dirs. cada una, ya que el tamaño de página debe coincidir en ambas).

Caso concreto:



En general:



- Es habitual que los campos “p” y “f” sean más extensos que el “d”, ya que los espacios de direcciones constan de varios Terabytes descompuestos en Giga-páginas de Kilo-palabras de memoria de 1 byte.

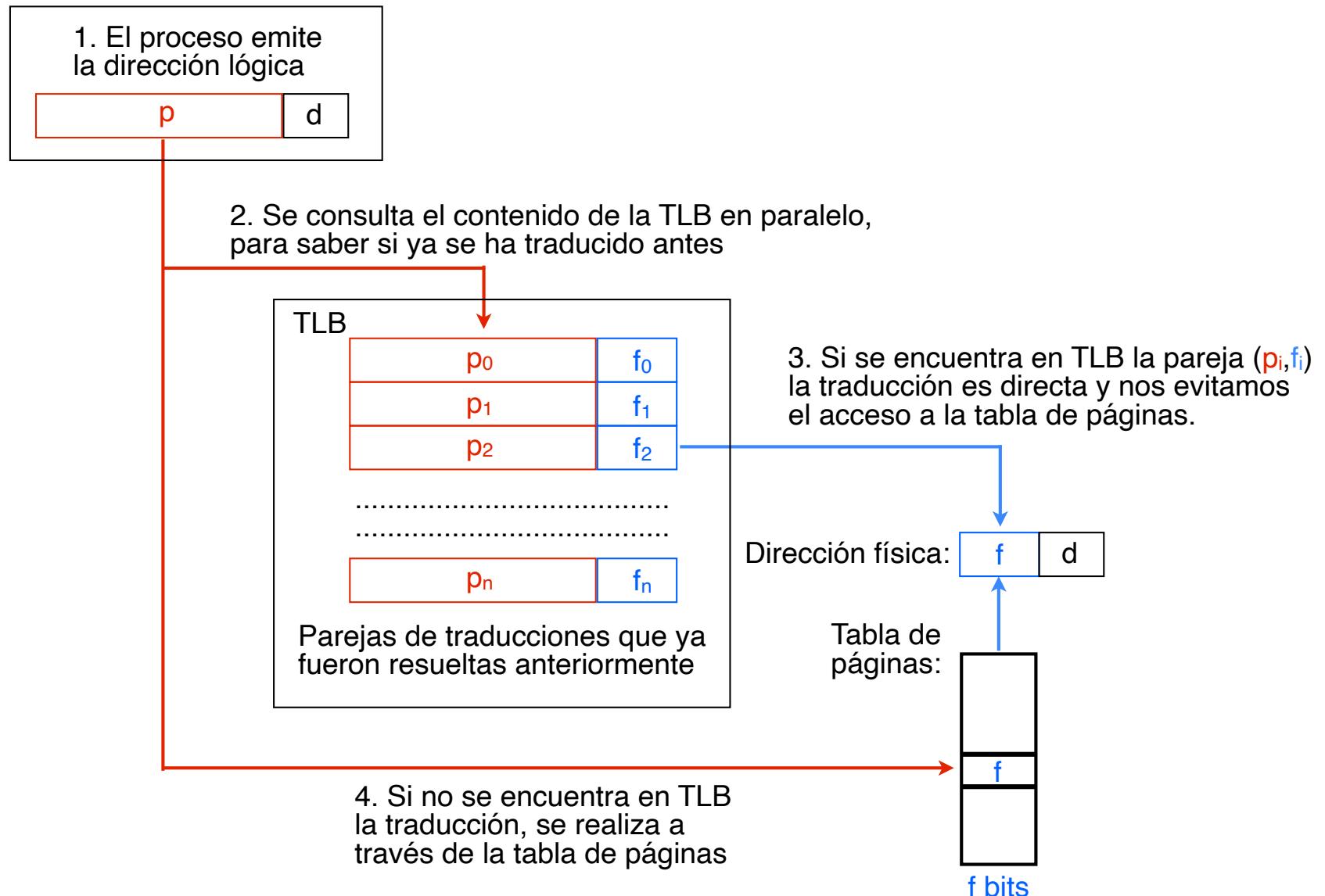
Un caso típico en los PCs actuales

- Espacios de direcciones lógicas de 2^{48} palabras de memoria direccionables a nivel de byte, lo que determina:
 - 48 bits de longitud para los punteros.
 - 256 TB. como el máximo tamaño en memoria para un proceso.
- Páginas de 16 KB. (2^{14} palabras de memoria direccionable a nivel de byte).
- Memoria física DDR4 de 32 GB. (2^{35} palabras de un byte).
- Con todo esto, tendríamos las siguientes longitudes para los campos del esquema de traducciones:
 - d = 14 (longitud del desplazamiento de página).
 - p = 34 (ya que $p+d = 48$, la longitud del puntero).
 - f = 21 (ya que $f+d = 35$, el exponente binario del número de palabras que tiene la memoria principal).

Implementación de la tabla de páginas

- El S.O. mantiene la tabla de páginas en memoria principal, que como otras áreas queda registrada por dos cotas:
 - Una dirección base de inicio (*PTBR - Page Table Base Register*).
 - Una longitud (*PRLR - Page Table Length Register*).
- El esquema necesita acceder dos veces a memoria por cada acceso:
 - Uno para realizar la traducción a través de la tabla.
 - Otro para acceder al dato a partir de la dirección física resultante.
- Para acelerar la traducción podemos colocar una caché (*TLB - Translation Look-Aside Buffer*) que recuerde las parejas de traducciones lógica a física ya realizadas y usadas de forma frecuente, evitando hacerlas de nuevo.

Detalle de la TLB (Translation Look-Aside Buffer)



Análisis de la TLB

- Al ser la memoria DRAM cada vez más lenta, la TLB ha venido cobrando mayor protagonismo, integrándose dentro del procesador y aportando decenas de entradas para recordar un mayor número de traducciones ya hechas.
- Con esto se busca maximizar la probabilidad de traducir automáticamente, siendo notable la recompensa al ahorrarnos el acceso a la tabla en memoria principal.
- Sin embargo, no resulta barata ni fácil de implementar:
 - Tiene una memoria asociativa (se consulta por contenido en lugar de por dirección) similar a la que usa la memoria caché para saber si alberga el contenido de una dirección de memoria antes de acceder a memoria principal.

Compartición y protección de páginas

- Muchas páginas contienen información que es compartida por los procesos, debiendo éstos registrar para cada una de sus páginas si corresponde o no a su espacio lógico de dirs.
- En ese caso, la página compartirá su ubicación lógica y la traducción a dirección física en todos los procesos.
- Adicionalmente, cada página puede incluir un bit de validación para indicar si es lícito que el proceso acceda a sus contenidos. De esta manera, se le puede denegar el acceso para proteger la información o porque los datos hayan sido borrados por el proceso y aún no hayan sido reescritos por otros nuevos.

Compartición de páginas:

Espacio de direcciones del proceso P_1 :

0	Datos X
1	Datos Y
2	Datos Z
3	Datos P_1

Espacio de direcciones del proceso P_2 :

0	Datos X
1	Datos Y
2	Datos Z
3	Datos P_2

Espacio de direcciones del proceso P_N :

0	Datos X
1	Datos Y
2	Datos Z
3	Datos P_N

Tabla de páginas para P_1 :

0	4
1	5
2	2
3	3

Tabla de páginas para P_2 :

0	4
1	5
2	2
3	7

Tabla de páginas para P_N :

0	4
1	5
2	2
3	8

Espacio de direcciones físicas (los marcos):

0	
1	
2	Datos Z
3	Datos P_1
4	Datos X
5	Datos Y
6	
7	Datos P_2
8	Datos P_N
9	
...	

Memoria principal (DRAM)

Protección/validación de páginas:

Espacio de direcciones lógicas de un proceso (sus páginas):

0	Datos X
1	Datos Y
2	Datos Z
3	Datos P_1
4	Datos P
5	Datos Q
6	Datos R
7	Datos X

Tabla de páginas para ese proceso:

0	7	v
1	4	v
2	3	v
3	5	v
4	9	v
5	6	v
6	8	i
7	0	i

Número de página

Número de marco

Bit de validación

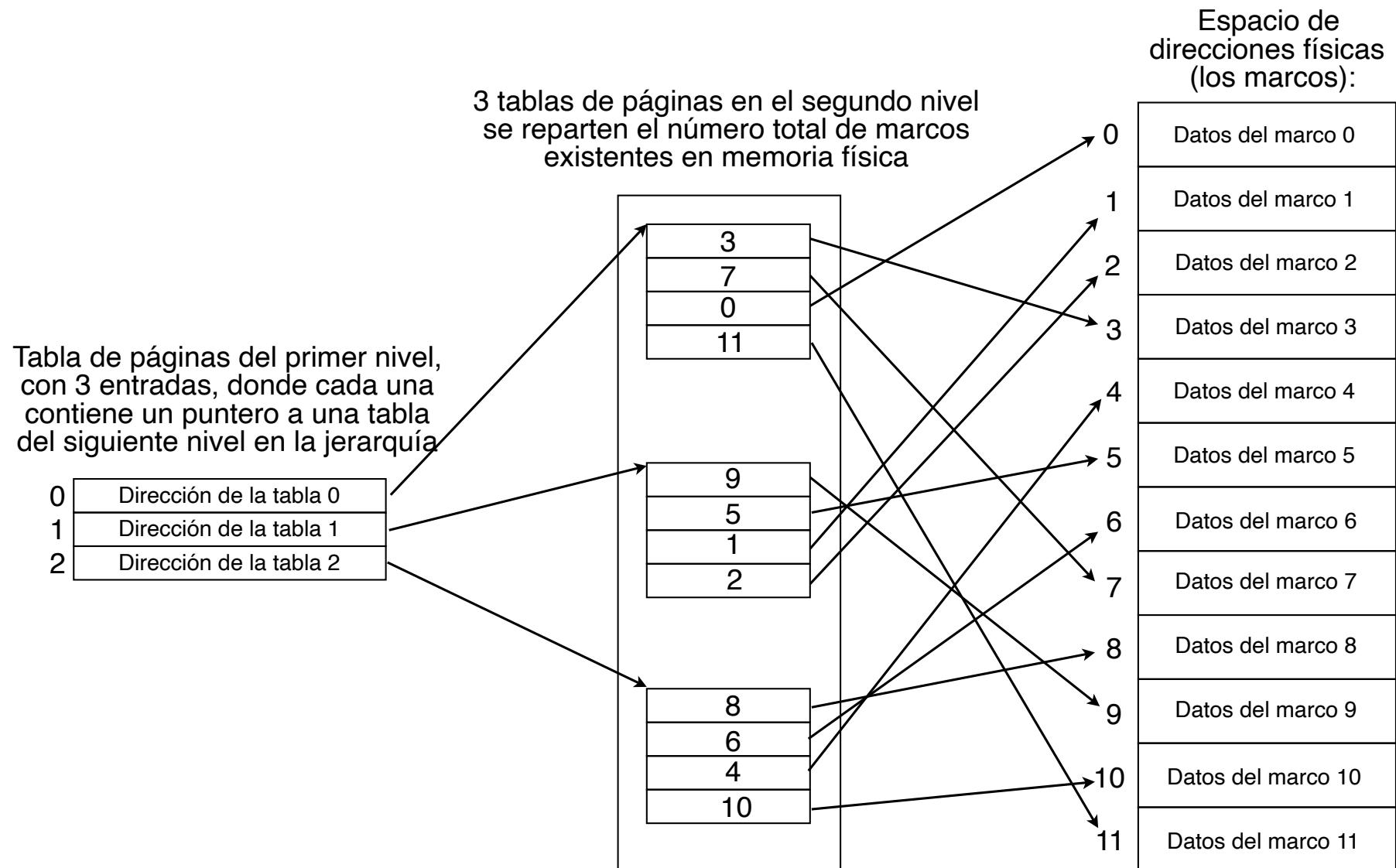
Espacio de direcciones físicas (los marcos):

0	
1	
2	
3	Datos Z
4	Datos Y
5	Datos P
6	Datos R
7	Datos X
8	
9	Datos Q
...	

Estructura de la tabla de páginas

- El crecimiento de la memoria ha aumentado el tamaño de las tablas de páginas, obligando al S.O. a afinar su gestión.
- Una primera idea consiste en establecer una jerarquía:
 - La tabla de páginas se va paginando sucesivamente, de forma que el último nivel indexe a los marcos de página, el penúltimo nivel indexe a las páginas del último nivel, y así hasta llegar a un nivel en el que todos sus índices quepan en una sola página.
- Otra idea es utilizar una tabla de páginas **invertida**:
 - En lugar de que la tabla de páginas tenga una entrada por página que indique su nº de marco en memoria física, albergará una entrada por cada marco, que indicará el nº de página alojado en dicho marco. Así la longitud de la tabla queda reducida al nº de marcos (que es muy inferior al nº de páginas - ver diapositiva anterior).

Tabla de páginas jerarquizada en dos niveles

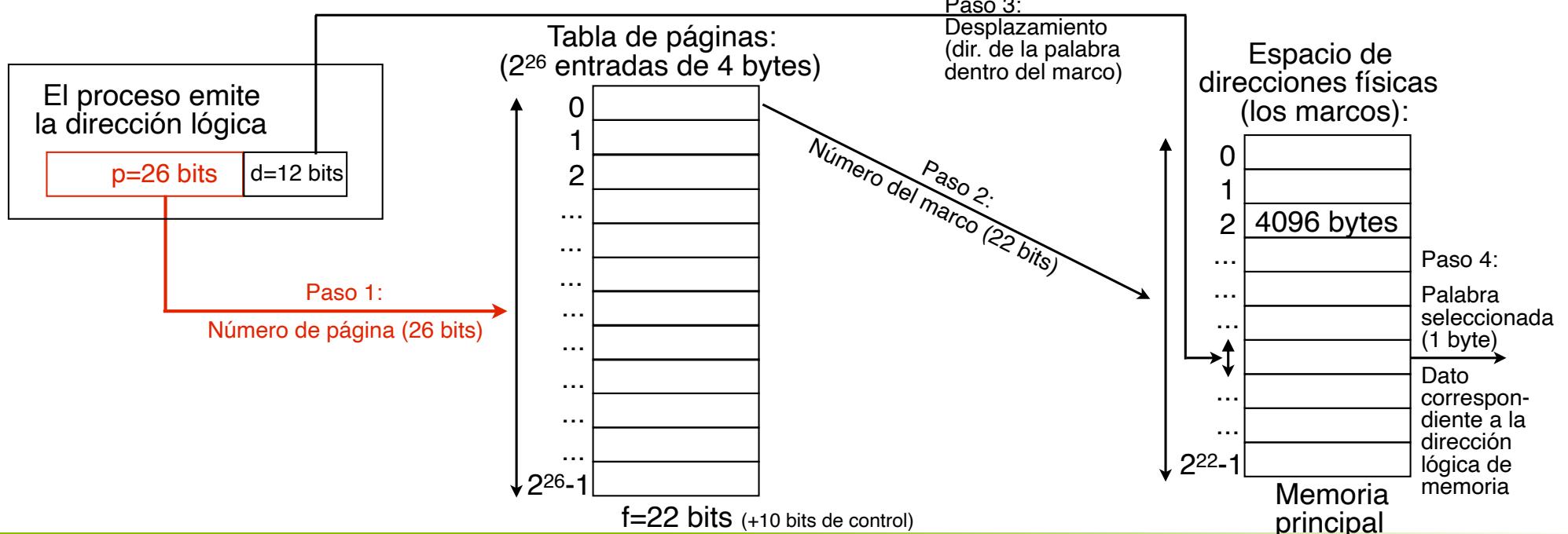


Ejemplo de paginación a varios niveles (1/4)

- Sea un PC con 16 Gbytes de DRAM que tiene instalado el S.O. Linux más habitual, donde:
 - Cada proceso tiene un espacio de direcciones lógico de 256 Gbytes (GB).
 - El tamaño de la página de memoria es de 4 Kbytes (KB).
 - La memoria es direccionable a nivel de byte.
- Esto nos deja:
 - **d=12 bits** de longitud para el *desplazamiento* dentro de la pág (ya que $2^{12} = 4$ KB).
 - **p+d=38 bits** de longitud para los punteros de memoria (ya que $2^{38} = 256$ Gbytes).
 - Por lo tanto, **p=26 bits** delimitan el número de entradas de la tabla de páginas para cada proceso, albergando cada entrada un marco de **f** bits de longitud.
- La memoria física se direcciona con 34 bits (ya que $2^{34} = 16$ GB), de los que **d=12 bits** corresponden al desplazamiento (cuyo valor debe coincidir con el ya calculado para las páginas), y los 22 bits restantes son para el marco (esto es, **f=22 bits**).

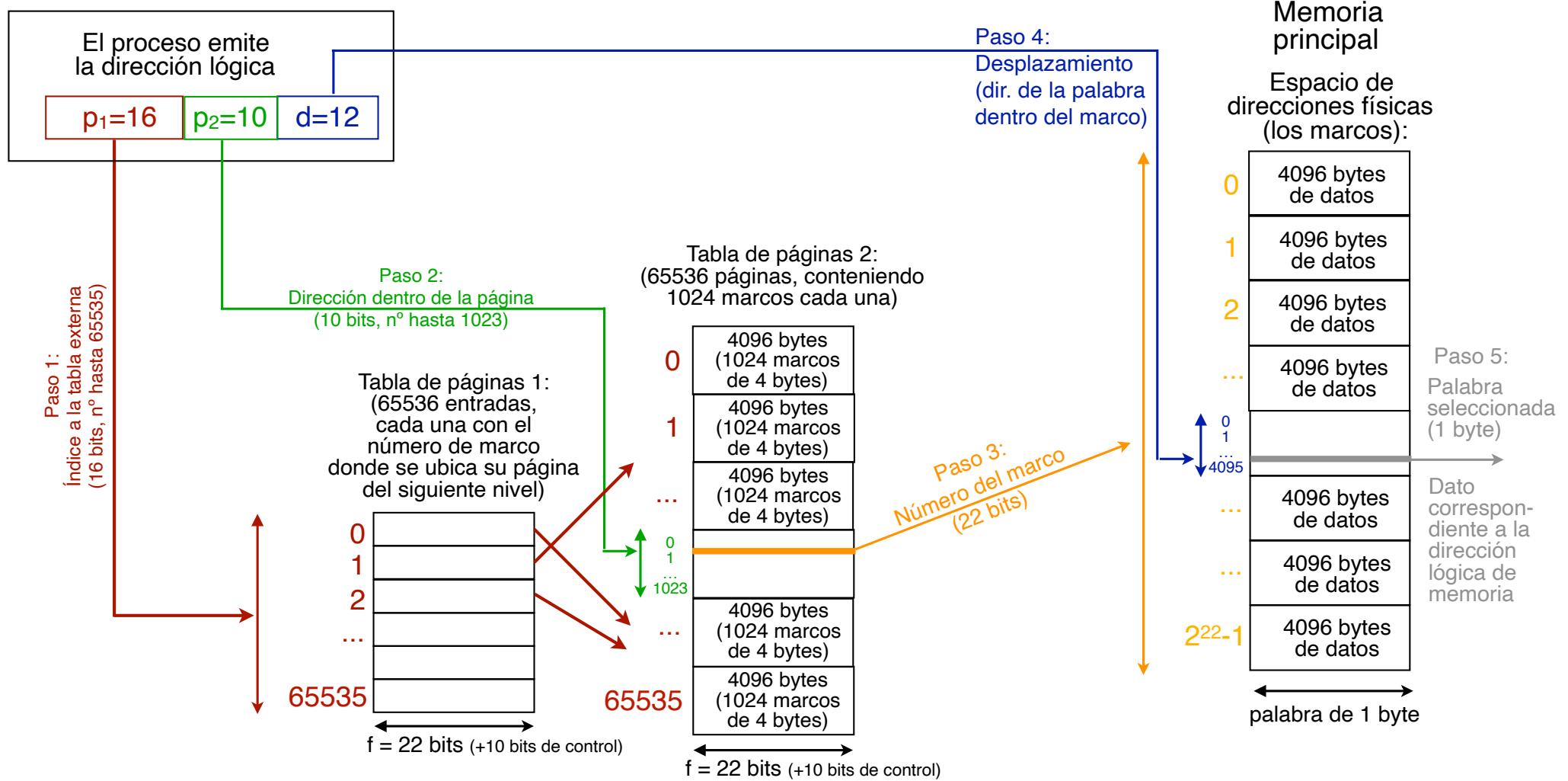
Ejemplo de paginación a varios niveles (2/4)

- La tabla de páginas de cada proceso tendría 2^{26} marcos de 22 bits cada uno (añadimos 10 bits de control en cada entrada de la tabla para completar a 32 bits, esto es, 4 bytes).
 - Por otro lado, en cada página de 4 KB caben $4\text{Kbytes/pág.} / 4 \text{ bytes/marco} = 1024$ marcos/página, así que hacen falta $2^{26} \text{ marcos} / 1024 \text{ marcos/pág.} = 65536$ páginas en el 2º nivel, y 65536 índices a éstas en el 1º nivel.



Ejemplo de paginación a varios niveles (3/4)

- Con una tabla de páginas de dos niveles, tendríamos:



Ejemplo de paginación a varios niveles (4/4)

Con una tabla de páginas de tres niveles, tendríamos:

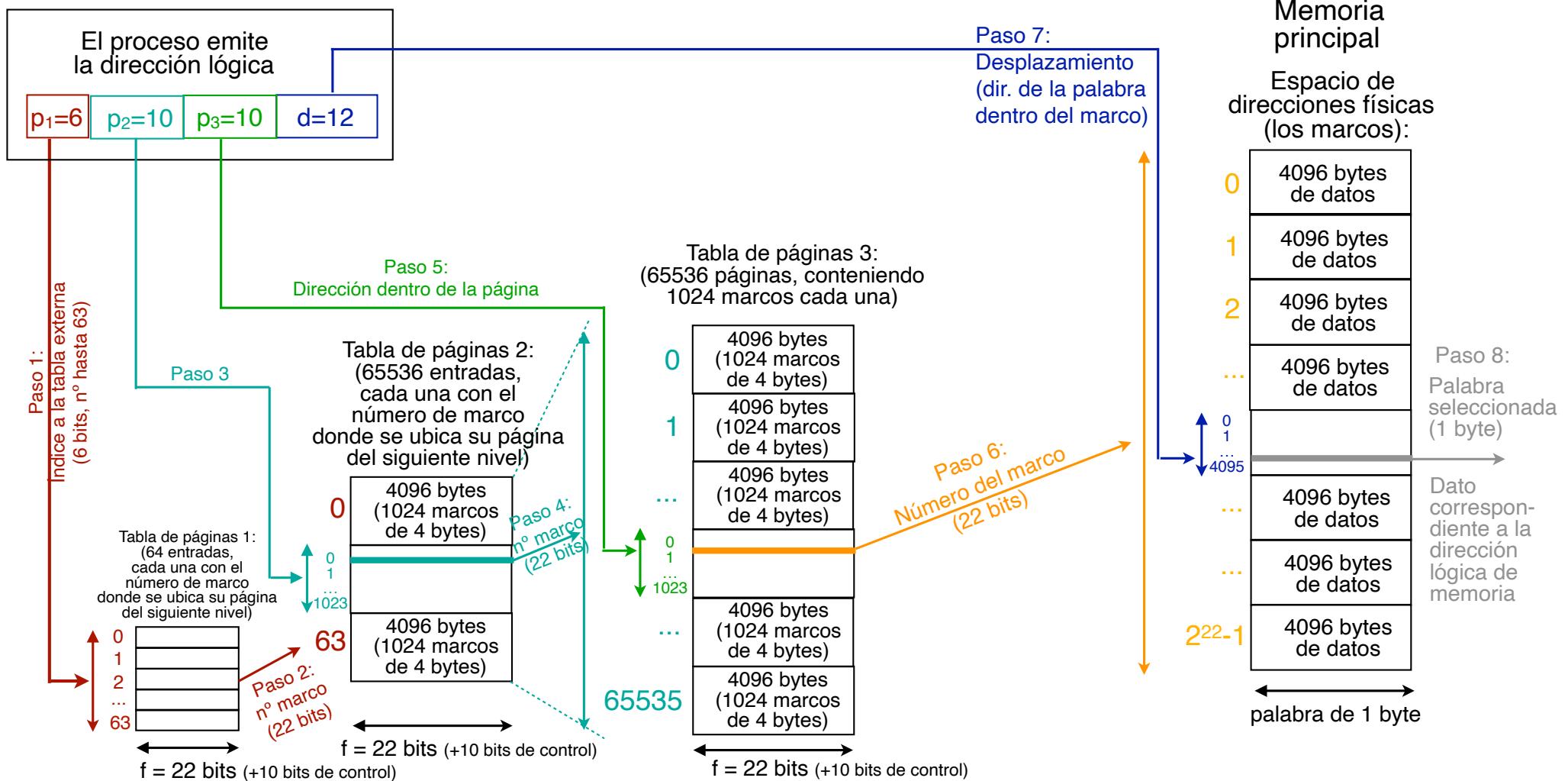


Tabla de páginas invertida

- En lugar de contener una entrada por cada página lógica donde se resuelva su página física o marco, la tabla habilita una entrada por cada marco, indicando la página lógica que contiene.
- Ventaja: La tabla reduce su tamaño al ser proporcional al espacio físico, que es muy inferior al lógico.
- Desventaja: Aumenta el tiempo de traducción de la dirección lógica, al tener que buscar en todos los marcos para saber en cual está. Se puede optimizar la búsqueda utilizando una tabla hash.
- En la memoria, el tiempo es máspreciado que el espacio, por lo que la tabla de páginas invertida resulta una idea poco útil en los nuevos S.O.

Ejemplo de tabla de páginas invertida

- Sea un espacio de direcciones lógico de 1024 páginas de 8 bytes cada una, sobre una memoria física de 64 bytes direccionable a nivel de byte.
 - Estos datos nos indican que las palabras de memoria son de 1 byte y que el espacio de direcciones lógico tiene 13 bits, con $p=10$ y $d=3$ (ya que tenemos 2^{10} páginas de 2^3 palabras cada una).
 - De forma similar, el espacio de direcciones físico se compone de 8 marcos de página, igualmente compuesto de 8 palabras cada uno, por lo que $f=3$ y $d=3$ (memoria física con 2^3 marcos de 2^3 palabras).
- Entra un proceso A de 128 bytes que referencia en orden ascendente sus últimas 64 direcciones: [A64, A65, ... A127]. Se pide representar el contenido de la tabla de páginas invertida y el esquema de traducciones (ver pág. siguiente).

Implementación y acceso a una tabla de páginas invertida

Estructura de las direcciones lógicas:

$p=10$

$d=3$

página

desplazamiento

Estructura de las direcciones físicas:

$f=3$

$d=3$

marco

desplazamiento

Direcciones emitidas por el proceso A
(sólo incluimos la primera y última de cada página):

A64	00000001000	000
A71	00000001000	111
A72	00000001001	000
A79	00000001001	111
A80	00000001010	000
A87	00000001010	111
A88	00000001011	000
A95	00000001011	111
A96	00000001100	000
A103	00000001100	111
A104	00000001101	000
A111	00000001101	111
A112	00000001110	000
A119	00000001110	111
A120	00000001111	000
A127	00000001111	111

Traducción para la dirección lógica A77:

00000001001 101

Marco:

000
001
010
011
100
101
110
111

Tabla de páginas invertida con 8 entradas:

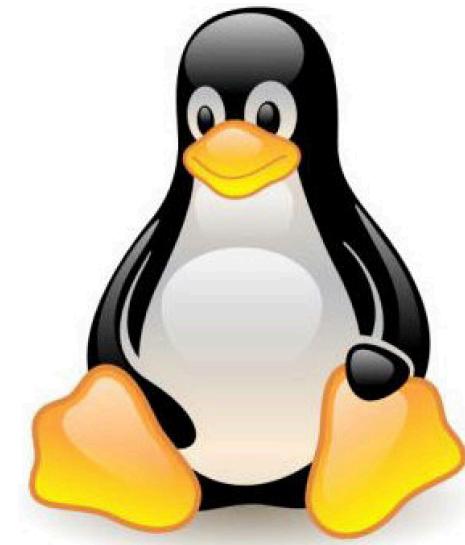
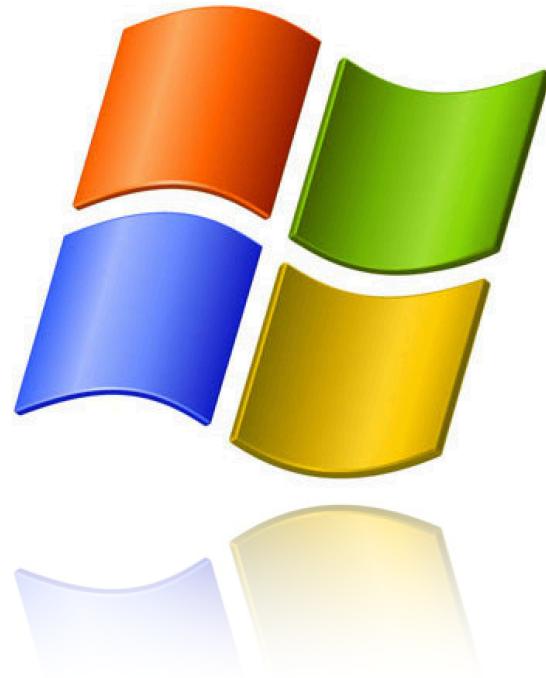
00000001000
00000001001
00000001010
00000001011
00000001100
00000001101
00000001110
00000001111

Está en la palabra 101 del marco 001, es decir, en la sexta palabra del segundo marco, que es la dirección 13 de memoria principal

Dirección física:

0	Marco 0 con 8 bytes de datos
7	Marco 1 con 8 bytes de datos
8	Marco 2 con 8 bytes de datos
15	Marco 3 con 8 bytes de datos
16	Marco 4 con 8 bytes de datos
23	Marco 5 con 8 bytes de datos
24	Marco 6 con 8 bytes de datos
31	Marco 7 con 8 bytes de datos
32	Marco 8 con 8 bytes de datos
39	Marco 9 con 8 bytes de datos
40	Marco 10 con 8 bytes de datos
47	Marco 11 con 8 bytes de datos
48	Marco 12 con 8 bytes de datos
55	Marco 13 con 8 bytes de datos
56	Marco 14 con 8 bytes de datos
63	Marco 15 con 8 bytes de datos

Memoria principal con 8 marcos de 8 palabras de 1 byte cada una



IV. Segmentación

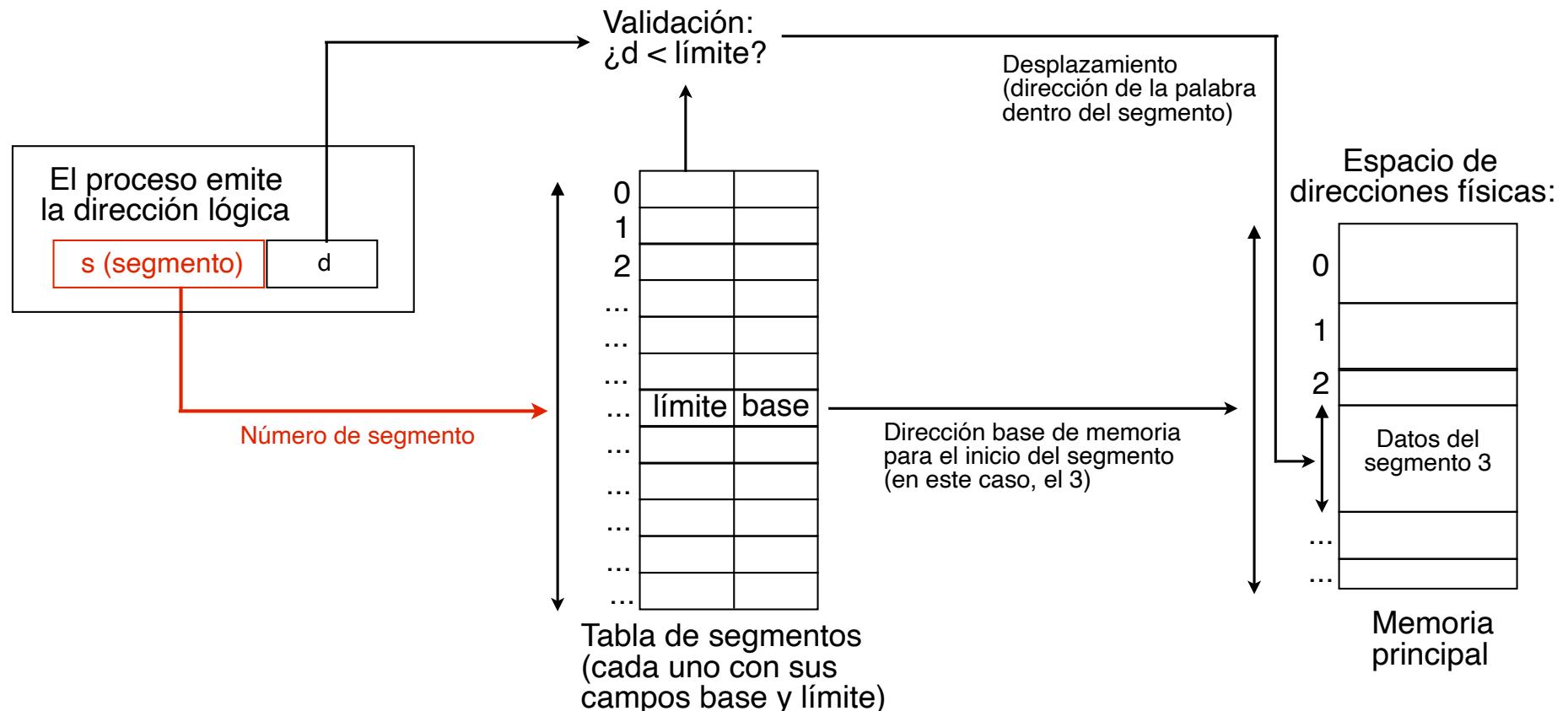
La idea de la segmentación

- Consiste en estructurar la memoria en segmentos que reflejen la relación que guardan sus datos para el usuario. Por ejemplo, un programa escrito por el usuario puede descomponerse en cuatro segmentos básicos:
 - El código máquina una vez compilado.
 - Los datos, con todas las variables y *arrays* del programa.
 - La pila, que gestiona las llamadas a funciones durante la ejecución.
 - El *heap*, o memoria dinámica que se solicita durante la ejecución.
- Los segmentos de código y datos tienen un tamaño fijo durante la ejecución, mientras que la pila y el *heap* crecen en direcciones opuestas, y si se topan sus límites, el programa se habrá quedado sin memoria.

Implementación

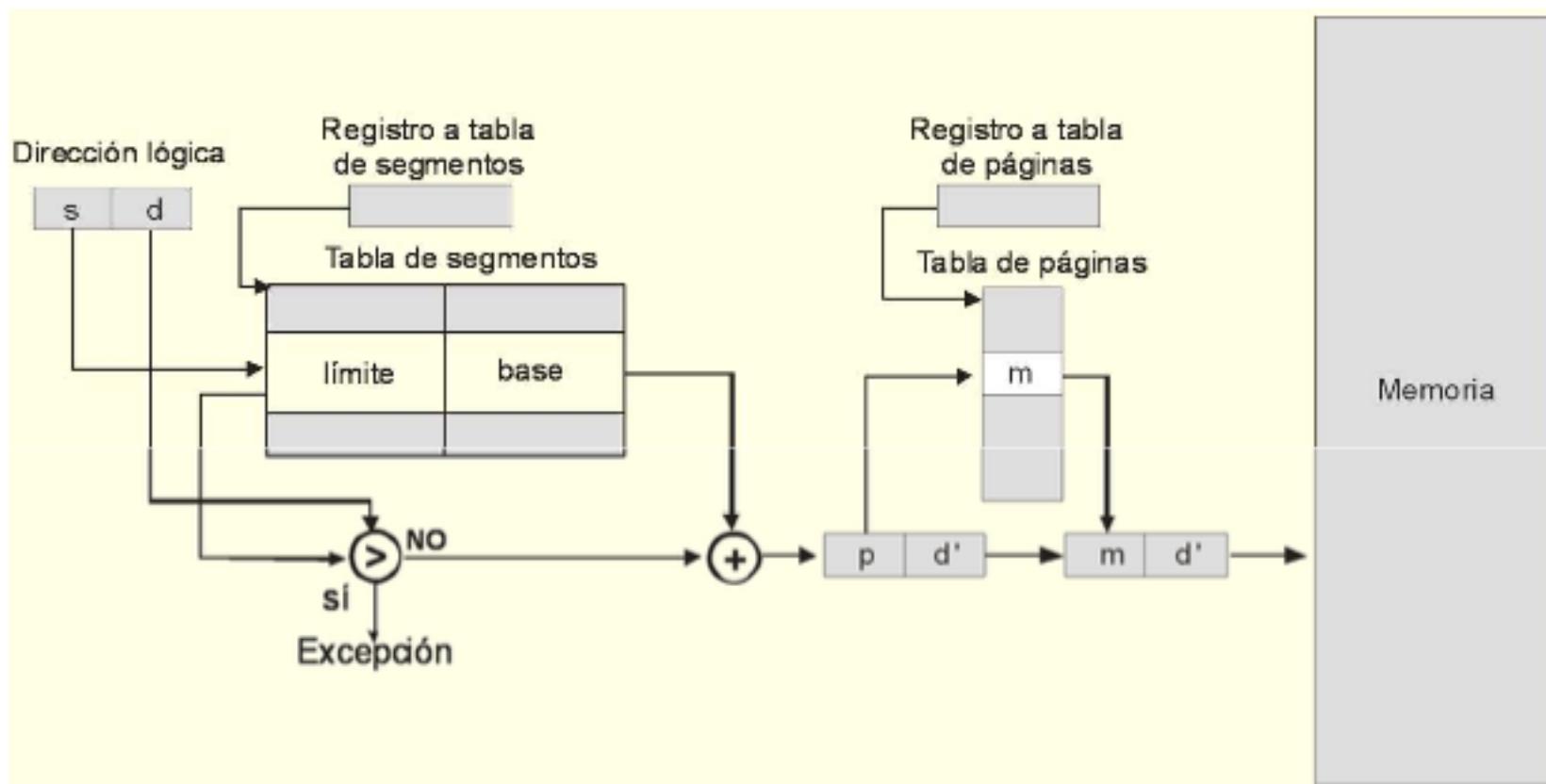
- La segmentación comparte algunos mecanismos con la paginación. La principal diferencia es que sus “páginas” son de longitud variable, lo que complica un poco su gestión, pero la traducción es similar:
 - Tendremos igualmente una dirección base del segmento y su límite.
 - Un registro base apuntará a la tabla de segmentos y un registro longitud indicará cuántos segmentos tiene el programa.
- Los mecanismos de validación también son afines a la paginación, con un bit para cada segmento que refleje su validez.

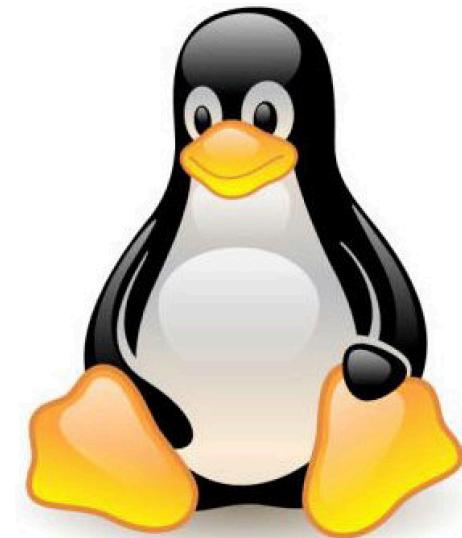
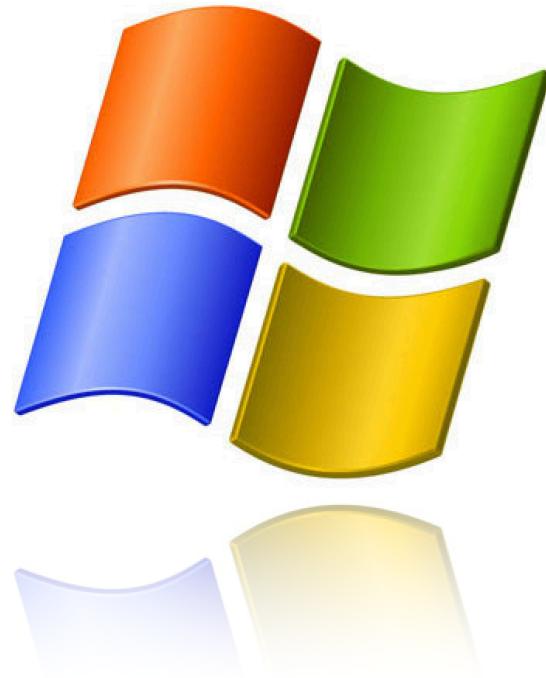
Traducción de dirección lógica a física



Segmentación paginada

- Combina los dos conceptos ya vistos: Divide la memoria en segmentos y gestiona éstos mediante páginas. Resulta el esquema más habitual en los S.O. actuales.





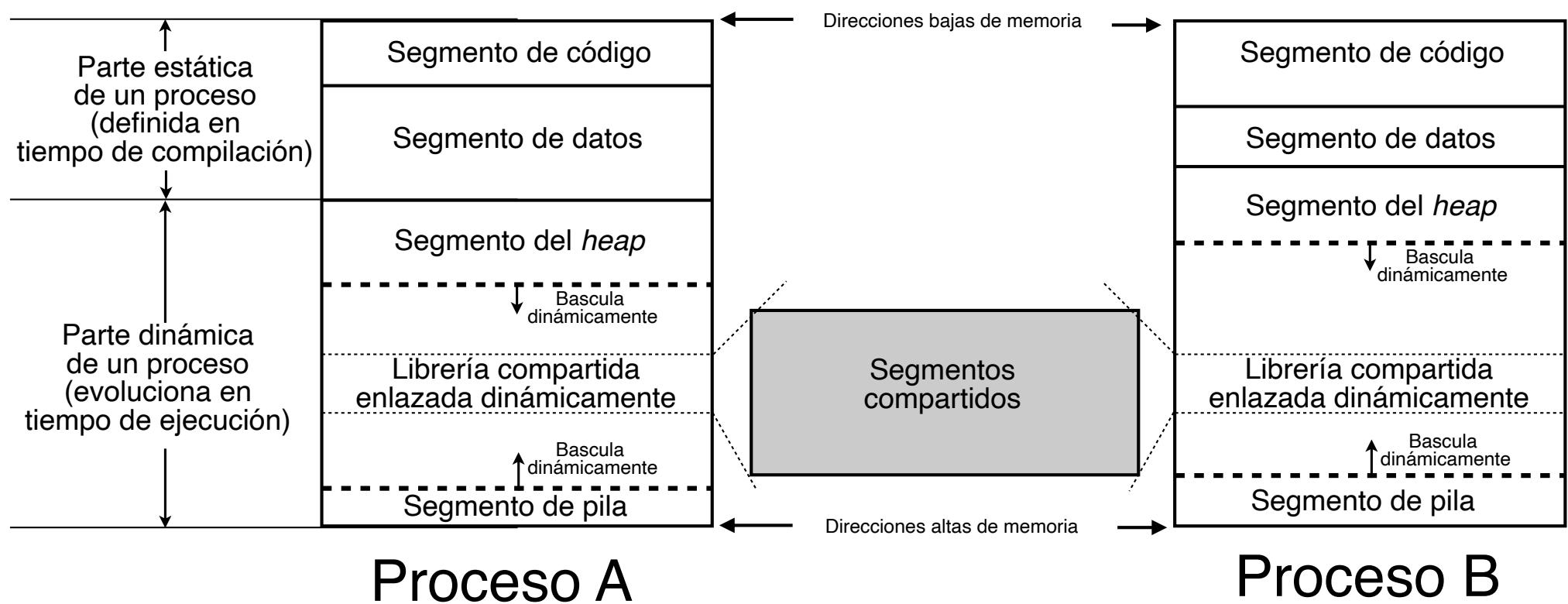
V. Memoria virtual

La idea de memoria virtual

- Contempla la separación de la memoria lógica de un proceso de la memoria física disponible, de tal forma que la primera pueda ser muy superior, utilizando para ello el disco como memoria secundaria.
- El ejemplo que pusimos para la tabla invertida ya asumía este escenario: un proceso con espacio lógico de 128 bytes para una memoria física de tan sólo 64 bytes.
- La memoria virtual también facilita a los procesos compartir los segmentos, como el código de una librería.
- Veremos dos formas de implementación bajo demanda:
 - Paginada.
 - Segmentada.

Ejemplo de memoria virtual segmentada

- Junto a los 4 segmentos típicos (código, datos, pila y *heap*) podemos incorporar una librería compartida de enlace dinámico, situada en el área que crece durante la ejecución.



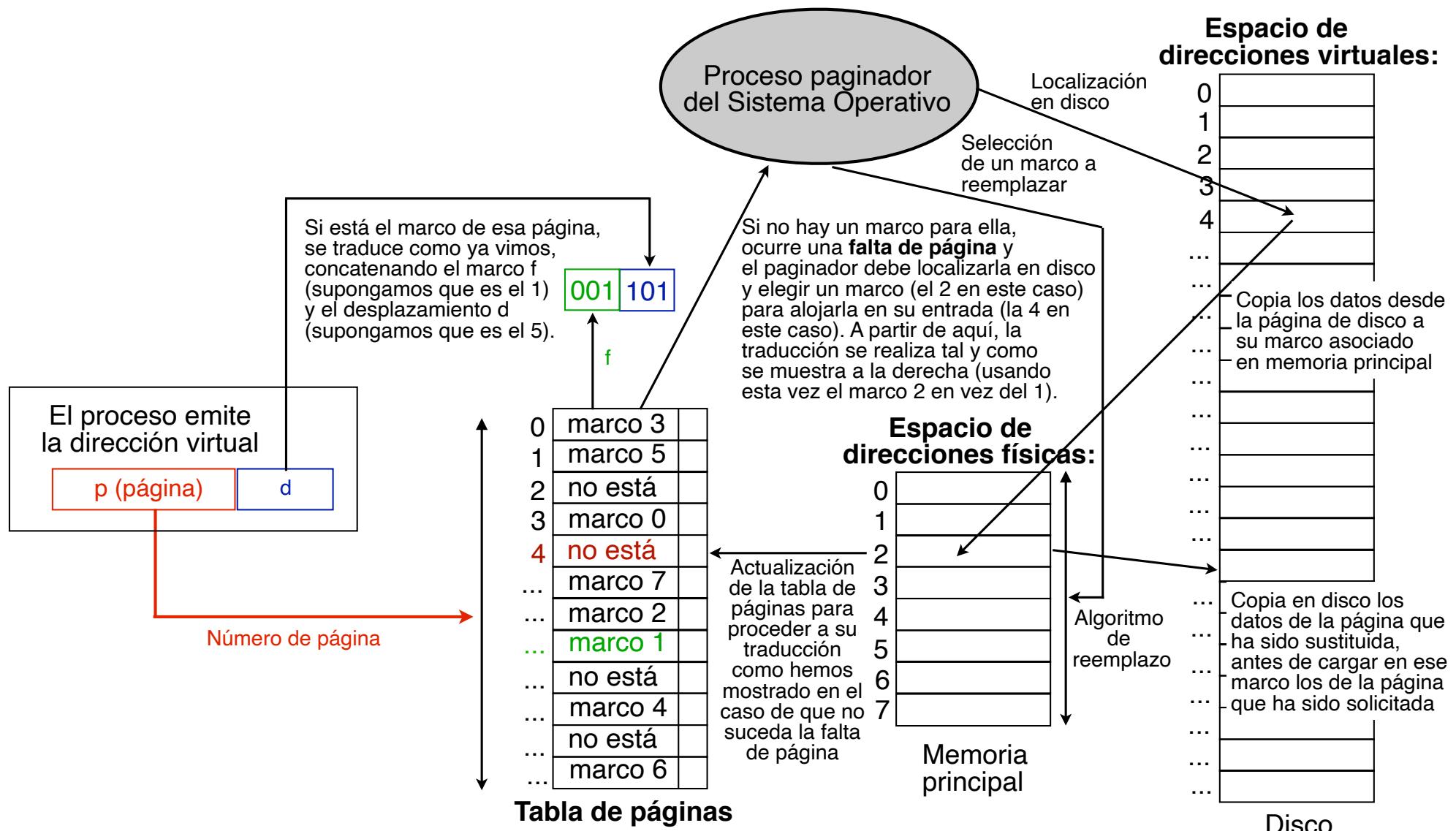
Paginación bajo demanda

- El espacio de memoria virtual se organiza en páginas, que residen en disco y se traerán a memoria al ser solicitadas.
- Una alternativa es traer las páginas de forma anticipada:
 - Intentan adelantarse a las solicitudes del proceso para reducir el tiempo de acceso a memoria, precargando la página desde disco a costa de aumentar el tráfico con éste.
- El proceso del S.O. que realiza estas funciones se denomina *swapper* en general, y *pager* (paginador) en este caso.
 - El paginador trata de beneficiarse de las tecnologías de memoria y disco para aprovechar su elevado ancho de banda y minimizar su gran latencia. Para ello, resulta más rentable apostar por páginas grandes.
 - Los tamaños de página más usuales son 4 KB., 8 KB. y 16 KB.

Tabla de páginas de la memoria virtual

- Hereda la estructura ya vista, con una entrada para cada página virtual, que actuará como espacio lógico en este caso, y se asociará con una página del espacio físico.
 - Si la página lógica está en memoria física, su entrada en la tabla de páginas indicará el marco que contiene la información de esa página.
 - Si no lo está, la entrada tendrá un valor nulo indicando que debe traerse de disco cuando sea referenciada, en cuyo caso decimos que ha ocurrido una **falta de página** (*page fault*).
 - El S.O. tendrá entonces que descartar un marco de memoria física y asociarlo a esa entrada de la tabla. La selección de este descarte tiene lugar mediante un **algoritmo de reemplazo** (que luego veremos).
 - La entrada de la tabla se completa con el bit de validación ya visto.

Traducción de dirección virtual a física y gestión de las faltas de página



Rendimiento de la paginación bajo demanda

- La paginación ha sido ampliamente estudiada y está muy optimizada, logrando un bajo porcentaje de faltas de página gracias al aprovechamiento de las propiedades de localidad espacial y temporal que exhiben la mayoría de programas:
 - Localidad espacial: La próxima página solicitada estará cerca de la que acaba de solicitarse. Ejemplo: Flujo secuencial del programa.
 - Localidad temporal: Una página que acaba de solicitarse volverá a solicitarse en breve. Ejemplo: Presencia de bucles en los programas.
- Una falta de página sigue estando muy penalizada porque el tiempo de acceso cuando entra en juego el disco pasa de decenas de nanosegundos a:
 - Milisegundos para un disco magnético.
 - Microsegundos para un disco SSD.

Rendimiento (2/3)

- El elevado tiempo de acceso es debido a la alta latencia en el acceso a la primera palabra. Una vez más, las demás palabras de la página se transfieren muy rápido gracias al ancho de banda, y apenas ralentizan el proceso.
- Al principio del tema indicamos latencias típicas, que vamos a tomar ahora como referencia para nuestro análisis:
 - Memoria DDR4: Entre 12 y 30 ns. Tomaremos 15 ns. de promedio.
 - Disco SSD: Entre 25.000 y 50.000 ns. Tomaremos 30.000 ns.
 - Disco magnético: Entre 3 y 6 msg. Tomaremos 3.000.000 ns.
- El tiempo de acceso efectivo (EAT) se calcula pesando los aciertos a memoria por 15 ns. y los fallos de página por 30.000 ns. ó 3.000.000 ns., sucediendo éstos con probab. p.

Rendimiento (3/3)

- Para el disco SSD, tenemos:

- $EAT = 15*(1-p) + 30000*p = 29985*p + 15$

- Para el disco magnético, tenemos:

- $EAT = 15*(1-p) + 300000*p = 2999985*p + 15$

- Aún suponiendo un caso de que sólo se produce una falta de página por cada mil accesos a memoria (suele ser algo superior), el tiempo efectivo de acceso a memoria es:

 - Para el disco SSD:

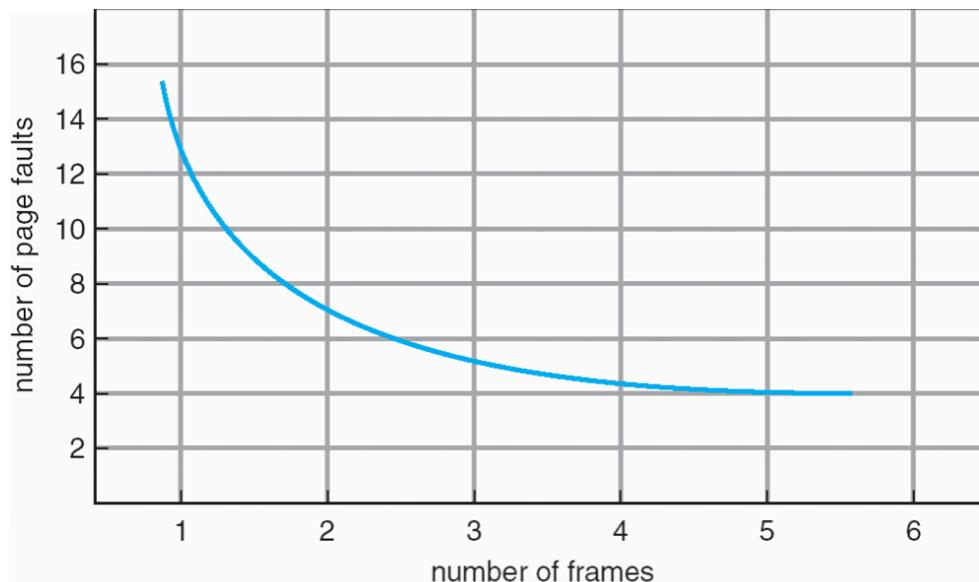
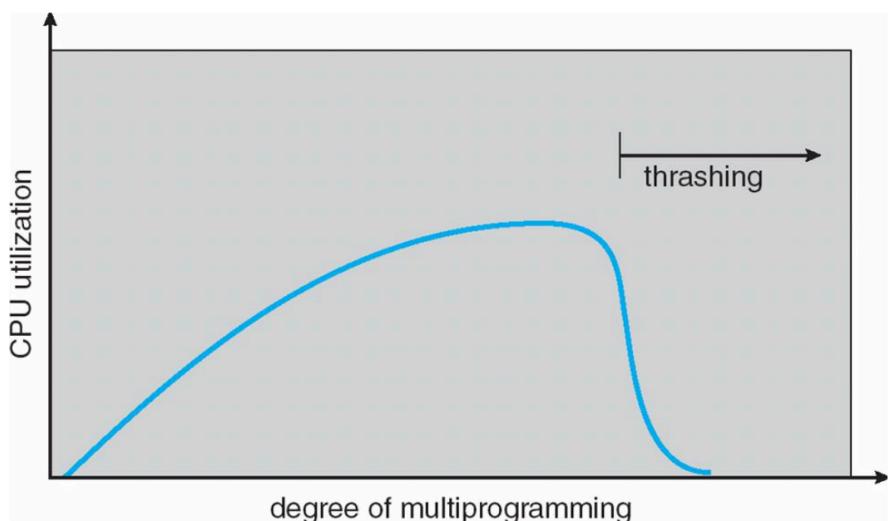
 - $EAT = 44.98 \text{ ns}$. [el factor de ralentización es de 3 veces - o un 300% - respecto al caso en el que pudiésemos disponer de memoria física para todo el espacio de direcciones de memoria].

 - Para el disco magnético:

 - $EAT = 3014.98 \text{ ns}$ [el factor de ralentización aquí es de 200 veces!].

Thrashing, la congestión de una memoria virtual que no tiene suficiente espacio físico

- El número de faltas de página crece exponencialmente a medida que baja el ratio entre la memoria virtual y la física. Este ratio puede alcanzar un punto crítico en el que el S.O. se colapsa porque su proceso paginador consume la mayor parte del tiempo, conduciendo al fenómeno de **thrashing**: se dedica más tiempo a la gestión del S.O. que al progreso de los procesos de usuario.



Reemplazo de páginas

- Dado que la memoria virtual contempla un espacio de direcciones muy superior a la memoria física disponible, las páginas de memoria virtual compiten por los marcos de página como lo hacen los coches en un parking abarrotado: Para aparcar tu coche antes debe irse otro, y el S.O. debe **reemplazar** el que tenga el menor interés en regresar para **minimizar el número de faltas de página**, factor clave para el rendimiento del sistema.
- Al escribir en una página de memoria, sus contenidos sólo se actualizan en la copia del disco cuando sea reemplazada. Así se alivia el trasiego memoria-disco, utilizando un bit *dirty* en la tabla de páginas para indicar esta operación pendiente.

Algoritmos de reemplazo

- Tratan de seleccionar la página víctima a reemplazar por la que necesita un marco libre para alojarse en memoria.
- Los dos esquemas básicos son:
 - **FIFO (First In First Out)**: Se descarta la primera que entró, esto es, la que lleva más tiempo en memoria física. Sencillo de implementar, pero poco eficaz. Ocasionalmente, puede aumentar el número de fallos de página con más memoria disponible (anomalía de Belady).
 - **LRU (Least Recently Used)**: Es el algoritmo más útil y efectivo. Descarta la página que menos se haya referenciado recientemente. Más adelante describiremos hasta 4 variantes de implementación.
- La efectividad de un algoritmo se compara con el óptimo: Pretender que sabemos el futuro y que siempre elegimos como víctima la página que más tarda en volver a usarse.

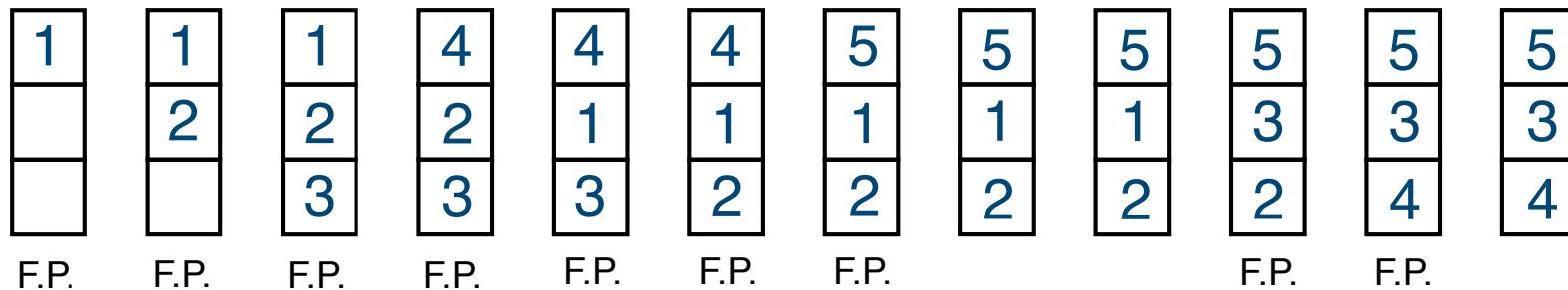
Ejemplos de algoritmo FIFO

(ilustrando además la anomalía de Belady)

- Secuencia de páginas solicitadas a memoria:

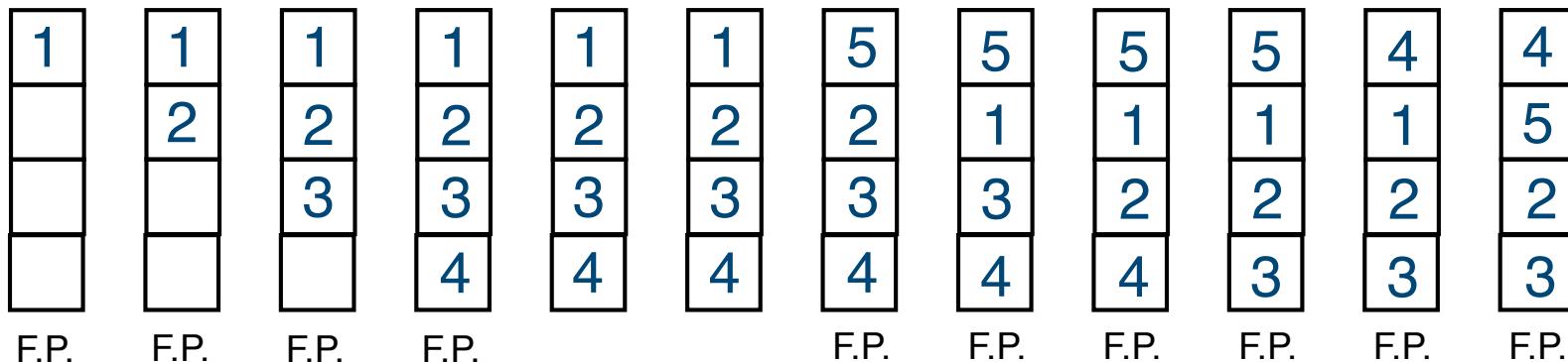
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Ubicación de páginas en marcos con 3 marcos disponibles:



Se producen 9 faltas de página (F.P.).

- Ubicación de páginas en marcos con 4 marcos disponibles:



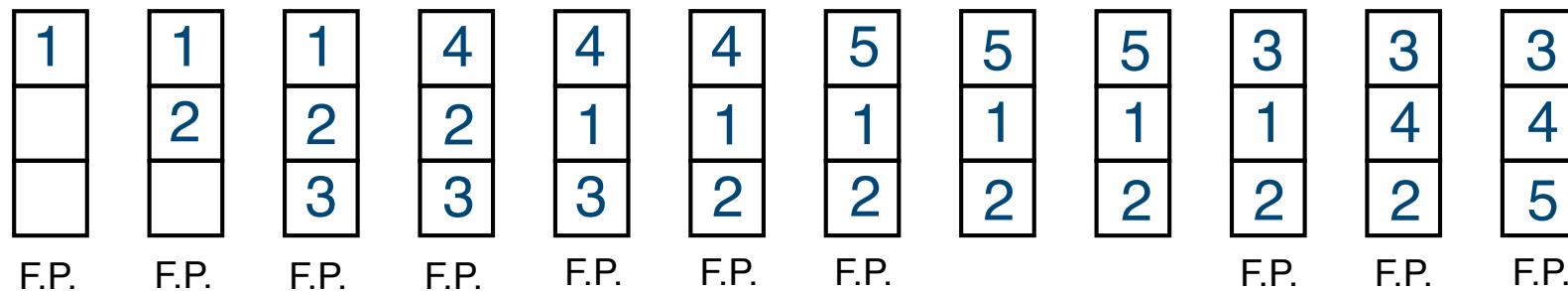
Se producen 10 faltas de página (F.P.) con más memoria física disponible.

Ejemplos de LRU: Mirar hacia atrás para descartar la página usada hace más tiempo

Secuencia de páginas solicitadas a memoria:

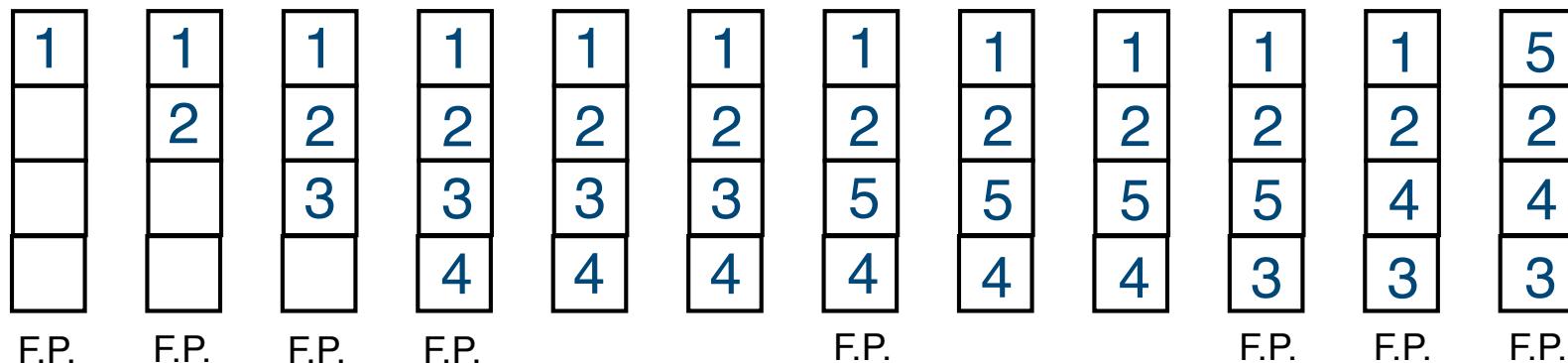
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Ubicación de páginas en marcos con 3 marcos disponibles:



Se producen
10 faltas de
página (F.P.).

Ubicación de páginas en marcos con 4 marcos disponibles:



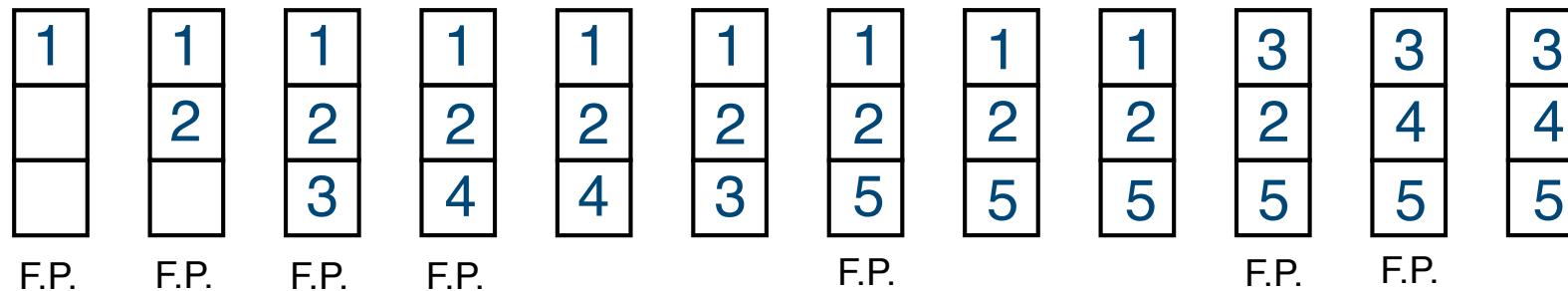
Al incorporar
una página más
de memoria física,
el número de faltas
de página se
reduce de 10 a 8.

Caso óptimo: Mirar hacia el futuro para descartar la página que se usará más tarde

- Secuencia de páginas solicitadas a memoria:

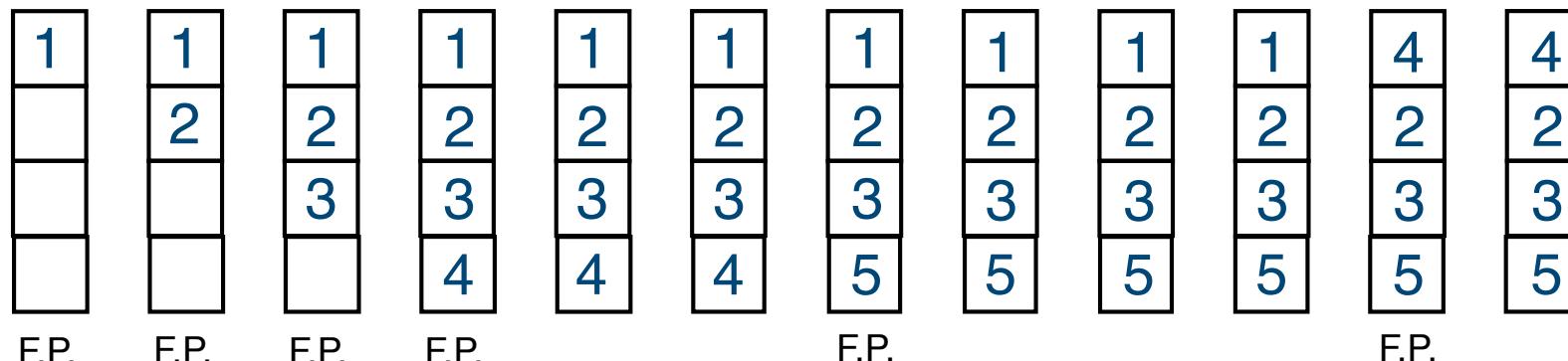
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Ubicación de páginas en marcos con 3 marcos disponibles:



7 faltas de página
es el mínimo.

- Ubicación de páginas en marcos con 4 marcos disponibles:



6 faltas de página
es la cota mínima
en este caso.
LRU queda 2 por
encima y FIFO
queda 4.

Resumen comparativo

Algoritmo	Implementación	Número de faltas de página para la secuencia anterior con 3 marcos de memoria	Número de faltas de página para la secuencia anterior con 4 marcos de memoria	Rendimiento
FIFO	Sencilla	9	10	Mejorable
LRU	Difícil	10	8	Bueno
Óptimo	Utópica	7	6	Óptimo

Consideraciones de rendimiento

- Incluso en el caso óptimo, las faltas de página necesarias para llenar la memoria física son inevitables. Aunque en los ejemplos previos éstas parezcan tener gran importancia, es justo lo contrario, ya que el **llenado** de la memoria apenas supone tiempo comparado con la ejecución del programa.
- Las faltas de página son muy sensibles a la **localidad** de referencia de un programa: Si el código no tiene localidad espacial ni temporal, ningún algoritmo logra resultados satisfactorios. Si la tiene, LRU la aprovecha bastante bien.
- Los niveles de **memoria caché** actuales capturan más del 97% de los accesos a memoria principal, aliviando la presión sobre esta memoria y su algoritmo de reemplazo.

Implementación de los algoritmos de reemplazo

- FIFO admite dos implementaciones a cuál más sencilla:
 - En cada marco de página se anota la hora de llegada de su última página virtual, eligiéndose como víctima el marco más “viejo”.
 - Se mantiene una cola por software, donde las páginas entran por un extremo y salen por el opuesto.
- LRU trata de aproximar el algoritmo óptimo mirando a las referencias pasadas como el óptimo lo haría a las futuras. Aunque la idea es sencilla, su implementación NO lo es. A continuación describiremos 4 posibles variantes, que usan:
 - Un registro para cada página que guarde la hora en que se usó por última vez.
 - Una pila para incorporar a su cima las págs. más recientemente referenciadas.
 - Un bit de referencia para registrar el uso de cada una de las páginas.
 - Una cola circular (algoritmo del reloj o de la segunda oportunidad).

Variantes de implementación LRU (1/2)

- Usar un registro para cada marco de página:

- Cada vez que se referencia a la página que alberga ese marco, el reloj del sistema se copia en ese registro.
- La página víctima será la que se encuentre en el marco más “viejo”.
- Es como la primera implementación FIFO, sólo que aquí debemos actualizar el registro de cada página no sólo a su llegada, sino cada vez que se accede a ella.

- Usar una pila con tantas posiciones como marcos, y los números de página doblemente enlazados, con un puntero a la página anterior y otro a la posterior dentro de la pila:

- Cada vez que se referencia a la página, se actualizan seis punteros para colocar su número en la cima de la pila.
- La página víctima será la que se encuentre en la base de la pila.

Variantes de implementación LRU (2/2)

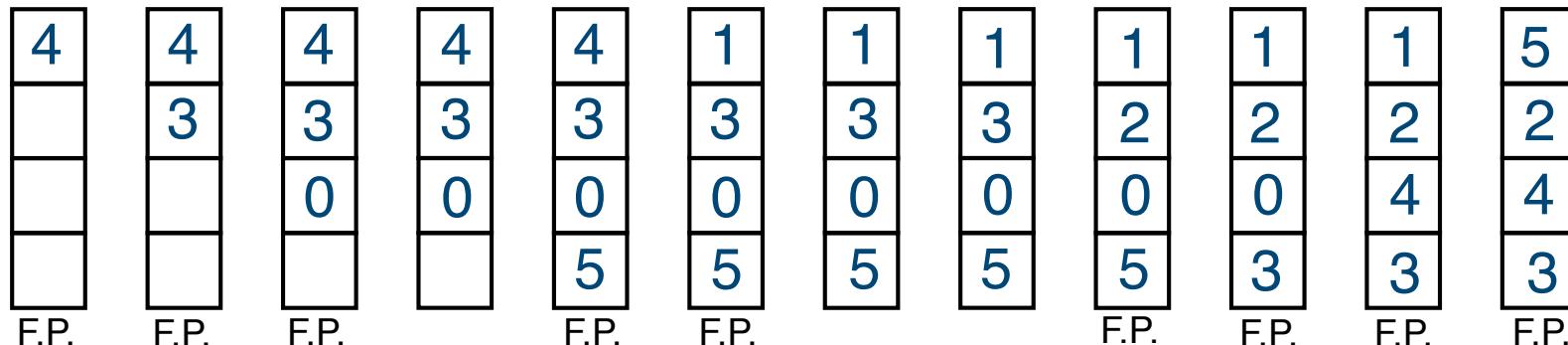
- Usar un bit de referencia para cada marco de página:
 - Cuando se carga una página en ese marco se coloca el bit a 0.
 - Cada vez que se haga referencia a esa página se coloca el bit a 1.
 - La página víctima será la que tenga el bit a 0, y si existen varias, se desempata por FIFO, retirándose la que más tiempo lleva cargada.
- Algoritmo del reloj o de la segunda oportunidad:
 - Se utiliza el bit de referencia anterior, y cuando hay que elegir una víctima, se recorren las páginas de forma circular hasta encontrar la primera con el bit a 0.
 - A todas las páginas que se encuentren con el bit a 1 hasta llegar a ese 0 se les resetea su bit y se les mantiene en memoria para concederles una segunda oportunidad.

Ejemplo para ilustrar el movimiento de la pila. 6 páginas compiten por 4 marcos de memoria

- Secuencia de 12 páginas solicitadas a memoria:

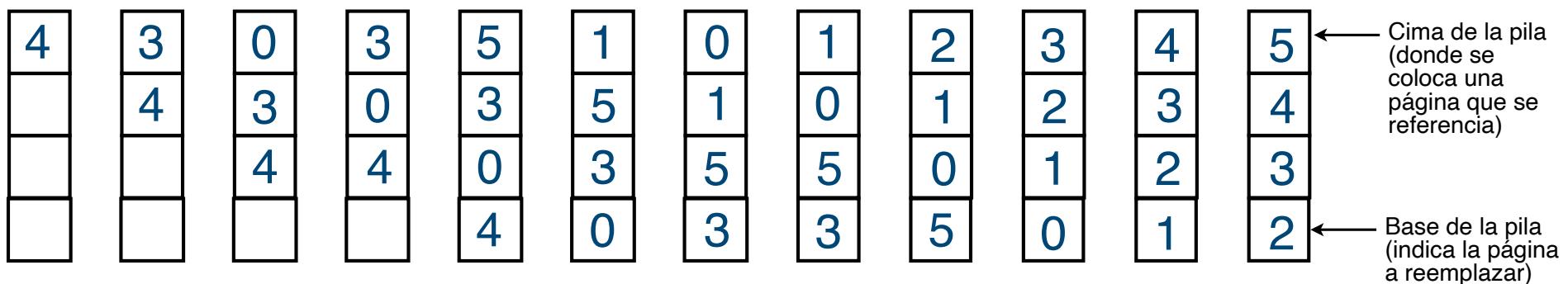
4, 3, 0, 3, 5, 1, 0, 1, 2, 3, 4, 5

- Ubicación de las 6 páginas en los 4 marcos disponibles:



Se producen 9 faltas de página (F.P.) para el total de 12 referencias a memoria.

- Evolución de la pila de 4 posiciones:

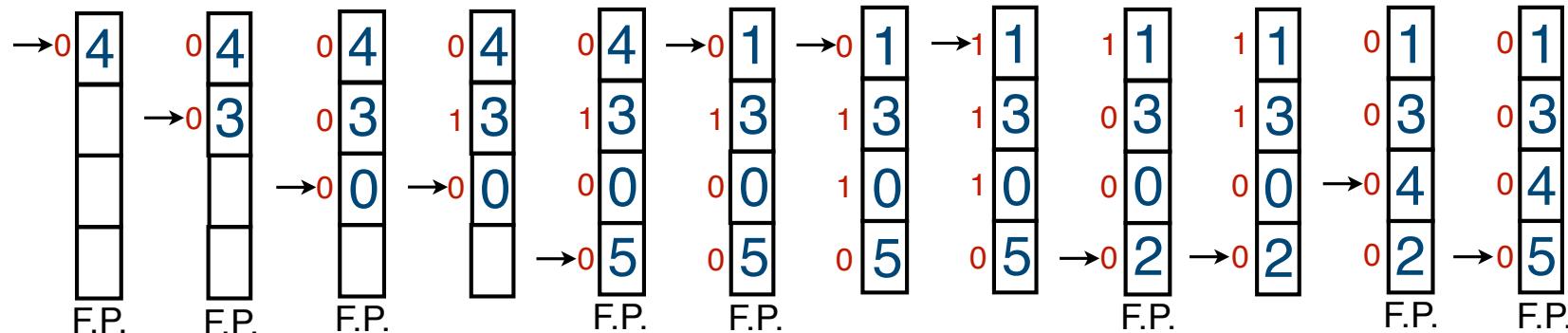


Ejemplo para ilustrar el algoritmo del reloj o de la segunda oportunidad

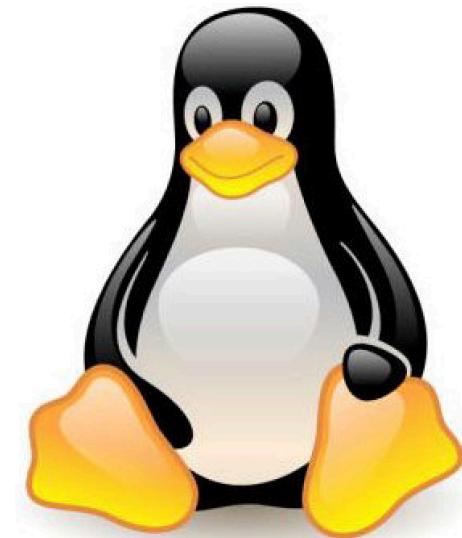
- Secuencia de 12 páginas solicitadas a memoria:

4, 3, 0, 3, 5, 1, 0, 1, 2, 3, 4, 5

- Ubicación de las 6 páginas en los 4 marcos disponibles a cada nueva solicitud de memoria. Junto los marcos, situamos en rojo el bit de referencia y el cursor que recorre la cola de forma circular para elegir la página a reemplazar:



Se producen 8 faltas de página para el total de 12 solicitudes de páginas de memoria.



VI. Optimizaciones

1. Alojamiento global frente a local

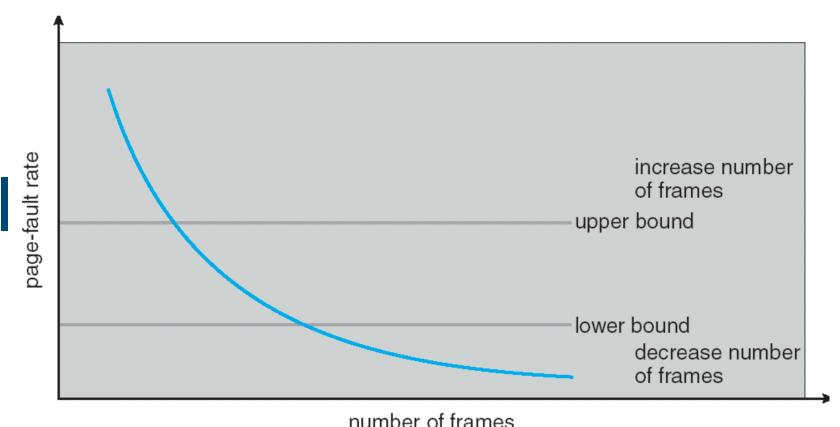
- Con el creciente nº de procesos activos que soporta un S.O. y una memoria cada vez más extensa, cobra sentido establecer un modelo **solidario** para su gestión que permita a un proceso que haga poco uso de sus páginas colocar sus marcos a disposición de otros que les saquen un mayor provecho.
- Desafortunadamente, las aplicaciones científicas sólo son un 30% del primer tipo (*compute-bound*), siendo el 70% restante del tipo *memory-bound*, esto es, la memoria constituye el principal factor limitador de su rendimiento.
- Con pocos procesos donantes y muchos receptores, globalizar la gestión apenas supone un alivio, y el S.O. debe vigilar que ningún proceso acapare un gran número de páginas ni quede **ahogado** en la memoria. Eso pretende el modelo *working set*.

2. El modelo del conjunto de trabajo (*working set model*)

- Pretende analizar los patrones de acceso a memoria de cada proceso para garantizar en cada ventana temporal de uso de la memoria un número **mínimo de páginas** físicas que permita a los procesos ejecutarse con cierto desahogo.
- Este modelo también previene el **riesgo** colectivo de colapso o ***thrashing***, suspendiendo alguno de los procesos cuando no pueda alcanzarse ese nº mínimo de páginas para el grado de multiprogramación vigente en ese momento.
- Para ello, el modelo *working set* monitoriza el uso que cada proceso hace de las páginas físicas que le son otorgadas, tratando de **anticipar** sus necesidades futuras y tomando las medidas oportunas para reducir la presión sobre la memoria.

3. Frecuencia óptima de faltas de página

- Es otra política global con la que el S.O. maximiza el rendimiento conjunto del sistema actuando de forma local sobre cada proceso.
- Si un proceso incurre en pocas faltas de páginas se le requisa uno de sus marcos para asignarlo a otro proceso que esté generando muchas faltas de página.
- La idea es **minimizar el promedio** de faltas manteniendo a todos los procesos dentro de un intervalo ideal de faltas de página en el que pueda ejecutarse sin excesos ni carencias.



4. Prebúsqueda de páginas en disco (prefetch)

• Trata de anticipar las páginas que va a necesitar un proceso, para traerlas a memoria física antes de que se soliciten, evitando así las faltas de página. Esta predicción suele apoyarse en las propiedades de localidad espacial y temporal, una tendencia cuya fiabilidad ya justificamos. Adicionalmente, aprovecharemos aquí algunas situaciones claras:

- Cuando se inicia un proceso, precargar sus N primeras páginas.
- Cuando se solicita una página, precargar la siguiente confiando en que el programa seguirá ejecutándose de forma secuencial (algo muy usual).
- Cuando hay pocas transferencias a disco, la precarga puede aprovechar sus tiempos muertos para traer sin coste alguna página estratégica.

• En el lado negativo, la precarga tiene un doble riesgo:

- Que la página precargada no sea finalmente referenciada.
- Que la página precargada reemplace a otra que sea más útil que ella.

5. Anclaje de páginas en memoria física (*pinned memory*)

- Existen diversas situaciones en las que una página alberga información sensible que desaconseja su reemplazo:
 - Se utiliza para operaciones del S.O. que no pueden interrumpirse, como un *buffer* donde se transfiere el contenido de un fichero.
 - Contiene datos críticos del S.O. que necesita acceder con presteza.
 - Aloja datos imprescindibles para un proceso de tiempo real.
 - Pertenece a un proceso de una prioridad muy elevada.
- Para casos así, el S.O. usa páginas *pinned*, que se fijan en memoria de forma permanente sin que puedan reemplazarse.
- Los S.O. y lenguajes de programación más modernos también ofrecen este mecanismo al programador, delegando en él cierta responsabilidad y mejoras de rendimiento.

6. El tamaño ideal de página

- La elección del tamaño de página afecta de forma decisiva a la gestión de memoria de un S.O., puesto que determina:
 - El espacio que se desperdicia por fragmentación interna y externa. A mayor tamaño de página, mayor es la interna y menor la externa.
 - El tamaño de la tabla de páginas. A mayor tamaño de página, serán necesarias menos entradas para cubrir el espacio de direcciones.
 - La eficiencia de las transferencias con el disco. A mayor tamaño de página, mayor peso del ancho de banda de las transferencias en la ecuación de rendimiento y menor peso de la latencia del disco.
 - El aprovechamiento de la localidad espacial en las referencias a memoria, que es mayor con páginas grandes.
- El riesgo de una página grande es similar al de la precarga: que se traigan datos que luego no sean referenciados.

7. Geometría de los accesos a memoria (1/2)

- Si el programador conoce el tamaño de página y la forma en que su lenguaje de programación articula el acceso a los datos en memoria, puede ejercer una influencia decisiva en la reducción del número de faltas de página.
- Un ejemplo es el lenguaje C, donde los datos de `A[i][j]` y `A[i][j+1]` son contiguos en memoria. Si en una página caben 32 datos de A y declaramos `float A[32][32];`
 - Recorriendo A en el orden contiguo en memoria, accederemos a todos los datos de cada página en el orden consecutivo de sus dirs.
 - Recorriendo A en el orden inverso, sólo accederemos al primer dato de cada página y saltaremos todos los siguientes, puesto que el siguiente acceso tiene lugar 32 direcciones de memoria más adelante.

7. Geometría de los accesos a memoria (2/2)

Programa C que explota la localidad espacial de referencia a memoria y el tamaño de página

```
float A[32][32];
for (i=0; i<32; i++)
    for (j=0; j<32; j++)
        A[i][j] = 0;
```

A[0][0], A[0][1], … , A[0][31] están todos en la misma página, así que el recorrido del lazo j queda dentro de la página y sólo se produce una falta de página por cada iteración del lazo i (si inicialmente no hay páginas de A en memoria). En total se producen 32 faltas de página.

Programa C que desaprovecha la localidad espacial y sólo accede a un dato de cada página cargada en memoria

```
float A[32][32];
for (j=0; j<32; j++)
    for (i=0; i<32; i++)
        A[i][j] = 0;
```

A[0][0], A[1][0], … , A[31][0] están todos en páginas diferentes, por lo que el recorrido del lazo j genera 32 faltas de página. Así, cada iteración del lazo i genera 32 faltas de página, y sus 32 iteraciones producen un total de $32 \times 32 = 1024$ faltas de página (igualmente suponemos que no tenemos ninguna página de A en memoria cuando empieza el programa).

Resumen

- El acceso a memoria es el principal cuello de botella de las aplicaciones, y lo seguirá siendo en el futuro porque cada vez es más lenta y cara respecto al procesador.
- Todas las tecnologías de memoria son más lentas cuanto mayor tamaño ofrecen, por lo que la clave está en articular una jerarquía caché-DRAM-disco que proporcione velocidad en los primeros niveles y capacidad en los últimos.
- Para ello, el S.O. debe gestionar la memoria de forma transparente al usuario para concentrar sus accesos en los niveles más rápidos, y articular mecanismos como la paginación, segmentación y memoria virtual que proporcionen una gran capacidad a un tiempo medio de acceso razonable.

Bibliografía

- Se recomienda la lectura del libro que colocamos en la portada de la asignatura en el Campus Virtual UMA:
 - Operating Systems Concepts, 8th Edition, de los autores A. Silberschatz, G. Gagne, P.B. Galvin, publicado por Wiley en 2008.
- Sus contenidos están disponibles íntegramente para los estudiantes de la UMA, a través del enlace Web:
 - <https://www.oreilly.com/library/view/operating-system-concepts/9780470128725/?ar>
- Este tercer capítulo del temario de S.O. está cubierto por la tercera parte del libro, que consta de dos capítulos:
 - Chapter 8: Main Memory (para nuestras secciones I, II, III y IV)
 - Chapter 9: Virtual Memory (para nuestras secciones V y VI)