

Gestión de memoria

Sistemas Operativos

Enrique Soriano, Gorka Guardiola

GSYC

5 de diciembre de 2022



Este trabajo se entrega bajo la licencia “Atribución-CompartirIgual 4.0 Internacional” [1] (cc-by-sa).

Usted es libre de:

- ▶ **Compartir:** *Copiar y redistribuir el material en cualquier medio o formato.*
- ▶ **Adaptar:** *Remezclar, transformar y construir a partir del material para cualquier propósito, incluso comercialmente.*
- ▶ **Se pueden dispensar estas restricciones si se obtiene el permiso de los autores.**
- ▶ *Las imágenes de terceros mantienen sus derechos originales.*

©2022 Gorka Guardiola y Enrique Soriano.

[1] Algunos derechos reservados. “Atribución-CompartirIgual 4.0 Internacional” (cc-by-sa). Para obtener la licencia completa, véase <https://creativecommons.org/licenses/by-sa/4.0/deed.es>. También puede solicitarse a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Asignación dinámica de memoria

- ▶ Problema: tengo un área de memoria para repartir, me van reservando y liberando trozos de diferentes tamaños de forma dinámica.
- ▶ Una vez comprometida una reserva, no se puede mover.
- ▶ Gestión implícita: un *recolector de basura* (Garbage Collector) se encarga de liberar memoria no referenciada.
- ▶ Gestión explícita: lo haces manualmente. P. ej. conocemos:
 malloc: reserva memoria dinámica.
 free: libera memoria dinámica.

Asignación dinámica de memoria

- ▶ Esto se necesita implementar tanto en área de usuario como en el kernel.
- ▶ P. ej. el kernel de Linux tiene una función llamada `kmalloc`, que se apoya en un asignador que se llama *slab allocator*.
- ▶ En área de usuario: `malloc` es una función de la libc que reparte la memoria del *heap*. El heap puede crecer en demanda (con la llamada al sistema `brk`).
- ▶ Problemas de fragmentación: ya los conocemos, los hemos visto en sistemas de ficheros.

Problema: fragmentación externa

Fragmentación externa: vamos 4 personas al cine y quedan 12 butacas libres, pero no hay 4 contiguas.

- ▶ Después de reservar/liberar, quedan huecos inservibles.
- ▶ La suma de los fragmentos sí sería útil.
- ▶ Ley 50 %: dados N bloques, $0.5 * N$ se pierden por la fragmentación externa (en media).
- ▶ La compactación solucionaría el problema... ¿se puede si hablamos de memoria dinámica?

Problema: fragmentación interna

- ▶ Posible solución a la fragmentación externa: reservar en base a bloques fijos → un hueco libre siempre puede ser útil.
- ▶ Además, ahora no malgastamos recursos para apuntar huecos inservibles.
- ▶ Problema: dentro de la memoria reservada sobra espacio, y la suma del espacio sobrante en todas las reservas sería útil...

Fragmentación interna: vamos 2 personas a cenar, y nos dan una mesa de 4 comensales.

- ▶ Para minimizar fragmentación externa: tamaño mínimo, agrupación de reservas relacionadas (tiempo y tamaño), etc.
- ▶ Para minimizar fragmentación interna: buscar el mejor ajuste, etc.
- ▶ Para minimizar tiempo en las reservas: caches, segregación por tamaño, etc.

Políticas de asignación dinámica de memoria

- ▶ **First Fit:** el primero en el que cabe empezando por el principio. Se suele comportar bien. Es rápido y simple.
- ▶ **Next Fit:** el primero en el que cabe empezando por donde te quedaste. Empíricamente se comporta peor que First-Fit (más fragmentación).
- ▶ **Best Fit:** el que se ajuste mejor. Ayuda a tener los fragmentos pequeños. Más lento. Tiende a dejar huecos muy pequeños o muy grandes.
- ▶ **Worst Fit:** el que se ajuste peor. Peor resultado que los anteriores.
- ▶ **Quick Fit:** se mantienen listas de de trozos de tamaños populares para reservas rápidas. Segregación: tener distintas listas para trozos de distintos tamaños. Acelera la búsqueda, se comporta bien. Contra: más complejo, más estructuras.
- ▶ Hay otras: buddy system, bump allocator, etc.

Ejemplo real de implementación de malloc:

- ▶ Para peticiones grandes ≥ 512 bytes, un asignador Best Fit puro.
- ▶ Para peticiones pequeñas ≤ 64 bytes, un asignador Quick Fit con trozos de ese tamaño.
- ▶ Para peticiones intermedias usa una política mezcla de las anteriores.
- ▶ Para peticiones muy grandes ≥ 128 Kb, no irá en el *heap*: se crea una región nueva de memoria¹.

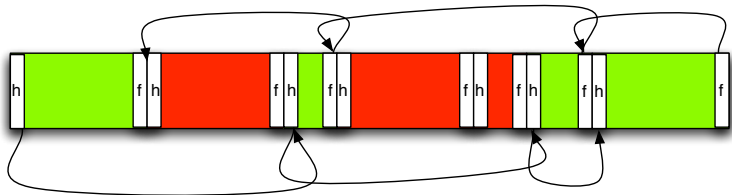
¹mmap anónimo.

Mecanismos para asignación dinámica de memoria

- ▶ Implementación: lista enlazada de trozos libres, de trozos ocupados, de ambos, circular, doblemente enlazada, varias listas...
- ▶ Coalescing: fundir dos trozos libres contiguos en uno, ¿cuándo lo hago?
- ▶ Headers y Footers: para moverse rápido entre nodos adyacentes. Acelera el fundido.
- ▶ Envenenamiento: valores que indican zonas liberadas para detectar bugs.

Ejemplo: doble lista de libres, con cabeceras y pies

¿Qué hay en el header (cabecera) y footer (pie)? el estado (libre, ocupado), el tamaño del trozo y (dependiendo del tipo) el puntero al siguiente/anterior libre.

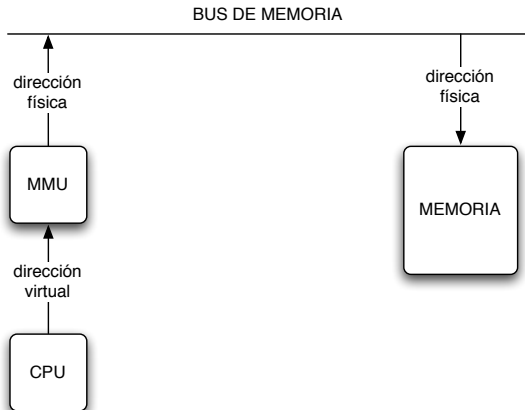


Libre

Ocupado

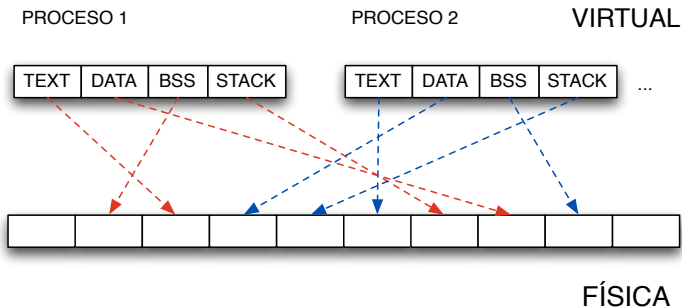
Memoria virtual

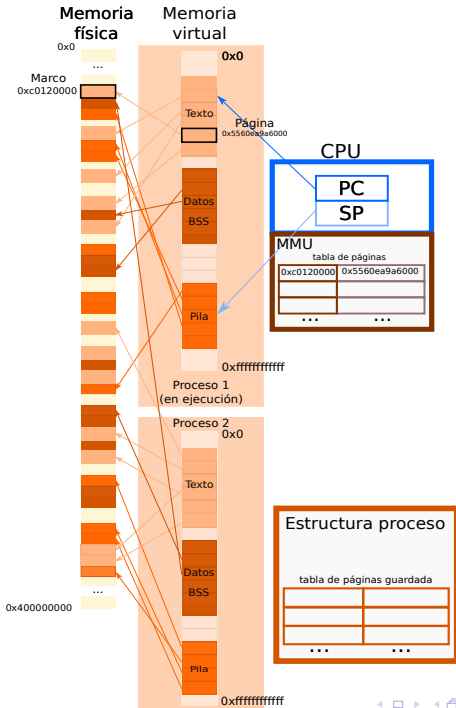
¡La CPU nunca ve direcciones físicas! La MMU se encarga de todo:



Memoria virtual

El kernel mantiene la correspondencia entre direcciones físicas y lógicas para cada proceso. Cuando se cambia el contexto a un proceso, se instala su mapa (tabla de páginas) en el hardware (simplificado, ver la siguiente transparencia).





Memoria virtual

- ▶ **Protección:** un proceso no puede acceder a la memoria de otro proceso ni a la del kernel (errores, ataques).
- ▶ **Simplicidad:** para el proceso, toda la memoria es contigua.
- ▶ **Abstracción:** un proceso cree que está en su propia máquina y que toda la memoria es suya.
- ▶ **Depuración:** la organización de la memoria es similar para todos los procesos y la misma para distintas ejecuciones de un programa.
- ▶ **Vinculación:** permite que sea en tiempo de ejecución (resolución de símbolos y relocalizaciones).
- ▶ **Reutilización/compartición:** una zona de memoria física se puede mapear en distintas zonas de la memoria virtual de distintos procesos, evitando copias.
- ▶ **Intercambio (swapping):** se puede usar almacenamiento secundario para almacenar la memoria de un proceso si no tenemos suficiente memoria física. **Es un error pensar en que esta es la única utilidad de la memoria virtual.**

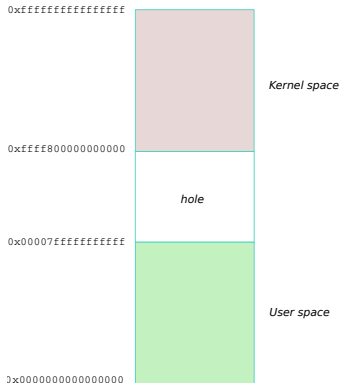
Memoria virtual

- ▶ Cuando se necesita tratar con direcciones *físicas*, el kernel puede instalar un *mapa identidad* (p. ej. para la inicialización del sistema).
- ▶ El kernel necesita comunicarse con el hardware a través de áreas mapeadas en memoria (p. ej. PCI):

```
$> cat /proc/iomem
...
000a0000-000bffff : PCI Bus 0000:00
  000a0000-000bffff : Reserved
000c0000-000c3fff : PCI Bus 0000:00
...
```

Memoria virtual

- ▶ El espacio de direcciones virtuales se suele dividir para direcciones del kernel y direcciones de usuario. En Linux x86_64²:



²Direcciones de 48 bits.

Intercambio

- ▶ Consiste en mover la memoria (parte o todo) de algunos procesos de RAM a un dispositivo de almacenamiento (disco): swap.
- ▶ Cuando toca ejecutar, se trae de vuelta y se lleva la de otro proceso.
- ▶ Grano: toda la memoria del proceso, segmentos, **páginas** ...
- ▶ Es muy caro:
I/O intercambio a disco + I/O intercambio desde el disco
 - ▶ Serial ATA (SATA-150) 1,200 Mbit/s
 - ▶ DDR3-SDRAM 136.4 Gbit/s
- ▶ Actualmente, no es tan útil.

MLock

- ▶ `mlock`: llamada que permite que las páginas correspondientes al rango dado nunca vayan a swap. Únicamente se permite hacer esto a procesos privilegiados.
- ▶ `mlockall`: lo hace con toda la memoria del proceso.

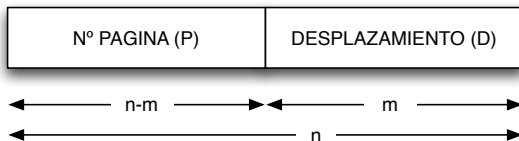
```
int mlock(const void *addr, size_t len);  
int munlock(const void *addr, size_t len);  
int mlockall(int flags);  
int munlockall(void);
```

Paginación

- ▶ Objetivo: poder tener la memoria de un proceso distribuida en zonas no contiguas de la memoria física.
- ▶ La memoria física se divide en **marcos**.
- ▶ La memoria lógica se divide en **páginas**.
- ▶ El hardware determina el tamaño de página.
- ▶ El medio de almacenamiento para el intercambio también se divide en porciones del tamaño de un marco.

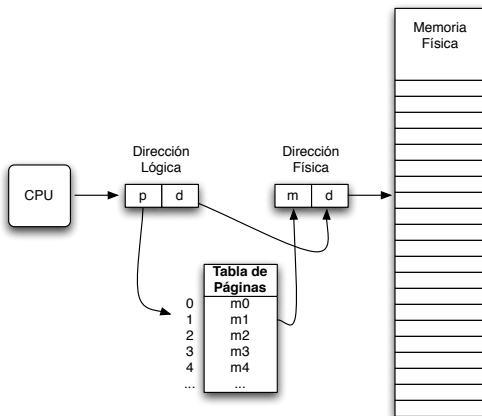
Paginación

- ▶ Tabla de páginas: dirección base para cada página. Cada proceso tiene su tabla de páginas.
- ▶ El sistema lleva la cuenta de los marcos de página.
- ▶ Si una dirección no tiene traducción: fallo de página.
- ▶ Dirección virtual:



$2^m = \text{tamaño de página}$

Paginación



Paginación

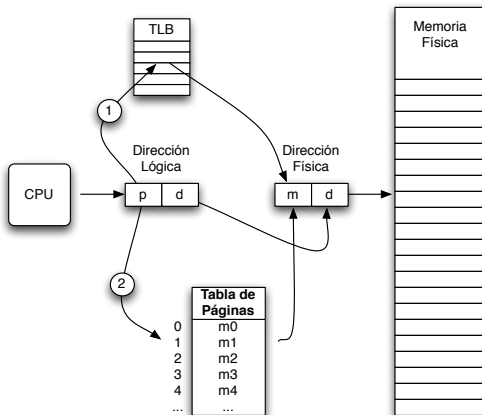
- ▶ De nuevo tenemos problemas de fragmentación interna.
- ▶ Si el tamaño de página es pequeño...
 - ▶ hay menos fragmentación.
 - ▶ necesitamos tablas de página con muchas entradas → búsqueda lenta.
- ▶ Si el tamaño de página es grande...
 - ▶ ganamos tiempo de I/O a la hora de traer páginas de almacenamiento.
- ▶ Hay que llegar a un compromiso: actualmente son de 4Kb - 8Kb.
- ▶ Algunas arquitecturas tienen superpáginas: hasta 2Gb.

- ▶ Cada proceso tiene su tabla de páginas, que forma parte de su contexto.
- ▶ Las tabla de páginas tienen que estar en memoria... y es grande.
- ▶ Se usan registros especiales de la CPU para apuntar a la tabla (los detalles dependen de la arquitectura).
- ▶ **Problema:** dos accesos a memoria principal para cada acceso real:
 1. acceso a la tabla de páginas
 2. acceso a la memoria

Paginación: TLB

- ▶ Solución hardware: TLB (*Translation Look-aside Buffer*).
- ▶ Es una pequeña memoria cache de la tabla de páginas (\approx cientos de entradas).
- ▶ El acceso a TLB es muy rápido:
 1. TLB, SRAM fully-associative: 1 ciclo
 2. Cache L1, SRAM set-associative: 3 ciclos
 3. Cache L2, SRAM: 14 ciclos
 4. Memoria principal, DRAM: 240 ciclos
- ▶ Traducción
 1. Si está en la TLB, se traduce directamente.
 2. Si no está, se busca en la tabla de páginas.
- ▶ Cuanto mayor tamaño de página, mayor tasa de acierto en TLB.

Paginación: TLB



Paginación: TLB

- ▶ Cuando se cambia de contexto, se tiene que limpiar la TLB (*TLB flush*).
- ▶ Si se accede a una página que no estaba, se inserta.
- ▶ Si la TLB está llena, se debe desalojar una entrada.
- ▶ Se pueden bloquear entradas.
- ▶ Tasa de aciertos (*Hit Ratio*): porcentaje de veces que la página está en la TLB.
- ▶ Tiempo de acceso efectivo:

$$T_{ef} = P_{acierto} * T_{acierto} + P_{fallo} * T_{fallo}$$

e.g.

$$0,98 * (20ns + 100ns) + 0,02 * (20ns + 100ns + 100ns) = 122ns$$

penalización del 22 % en el acceso vs. 100 % sin TLB.

Page Table Entry (PTE)

Las entradas pueden tener ciertos bits:

- ▶ Bit de presente (o bit de válido): bit en cada entrada de la tabla de páginas que indica si la página tiene traducción o no. Si no la tiene, puede ser no válida válida o porque esté en *swap*.
- ▶ Bit de modo: permiso para escribirla o no.
- ▶ Otros bits: si puede ir a caché, permiso de ejecución, si se ha accedido, ...

Paginación: tabla de páginas

- ▶ Problema: espacios de direcciones demasiado grandes (p. ej. 2^{48}).
- ▶ La tabla de páginas excesivamente grande como para tener todas sus entradas contiguas.

Paginación multinivel

- ▶ Una solución: dividir la tabla de páginas en N niveles.
- ▶ Ejemplo: en Intel i7 la tabla es de 4 niveles.
- ▶ Dirección lógica para 2 niveles: **(p1, p2, d)**.
 - ▶ p1: índice en la tabla exterior.
 - ▶ p2: índice en la tabla interior.
 - ▶ d: desplazamiento en la página.

Paginación multinivel

- ▶ Problema: un acceso puede provocar hasta $N+1$ accesos reales a memoria.
- ▶ En la práctica, depende de la tasa de aciertos de la TLB.
- ▶ Ejemplo: con una tasa de aciertos del 98 %, tiempo de acceso a la TLB de 20 ns, y tiempo de acceso a memoria de 100 ns:

$$T_{\text{acceso-efectivo}} = 0,98 * (20ns + 100ns) + 0,02 * (20ns + 100ns + 100ns + 100ns) = 124ns$$

Paginación: compartiendo páginas

Los procesos pueden compartir marcos de página:

- ▶ Ejemplo: el código (*text*) de un binario se puede compartir si es reentrante → si no se modifica durante la ejecución, no hace malgastar más marcos para guardar exactamente lo mismo.
- ▶ Shared memory: los procesos que comparten su memoria (*data*, *bss*, *heap*) comparten sus marcos de páginas.
- ▶ **Copy-on-write**: se puede compartir mientras que no se modifique. Cuando se modifica, se hace una copia y cada uno tiene la suya. Es lo que se hace cuando llamamos a `fork`: padre e hijo comparten las páginas hasta que uno intenta modificar algo.

Paginación en demanda

Consiste en

- ▶ Consumir únicamente los marcos de página que se necesitan, no todos los necesarios para las páginas del fichero → se ahorra memoria si hay páginas que nunca se llegan a usar.
- ▶ Aproximación perezosa: las páginas se van llevando a memoria física según se van necesitando, a medida que avanza la ejecución.

Paginación en demanda

- ▶ ¿Cuándo se reserva el marco? El proceso intenta acceder a la dirección de memoria, Si la página no está presente, habrá un fallo de página.
- ▶ Contra: tenemos muchos más fallos de página que manejar → más lento que reservar todo al principio.
- ▶ ¿Se debe cambiar un binario mientras que hay procesos ejecutándolo?

Paginación en demanda

Cuando se ejecuta una instrucción que intenta acceder a una dirección de memoria sin mapeo → salta una *trap* por fallo de página → el control pasa al kernel y:

1. Se mira si la dirección pertenece al espacio de direcciones del proceso.
2. Si no pertenece, se manda un SIGSEGV al proceso (segmentation fault).
3. Si es correcta, se comprueban los permisos de la página y el tipo de acceso que se está haciendo (lectura, escritura, ejecución).
4. Si son correctos, se busca un marco para esa página.
5. Si es necesario, se copia el contenido del fichero binario al marco de página (p.ej. si es una página de TEXT o DATA).
6. Se modifica la tabla de páginas para poner su bit de presente.
7. Se ejecuta de nuevo la instrucción que ha generado el trap. Ya puede acceder a esa dirección de memoria virtual.

Overcommitment

- ▶ Cuando se necesita hacer crecer el heap (llamada al sistema `brk`) no se compromete memoria física hasta que se intenta usar (hasta que se genera un fallo de página). Con overcommitment, la llamada a `malloc` **no** va a fallar.
- ▶ Lo mismo pasa con las variables globales sin inicializar (BSS).
- ▶ Cuando se accede por primera vez a una dirección de memoria de esa página, se compromete su marco de página.
- ▶ ¿Cuándo falla tu programa si el sistema se queda sin memoria (OOM, out of memory)? Si no hay marcos en ese momento... ¡fallo!
- ▶ Algunos sistemas lo tienen y permiten configurar su comportamiento. En Linux, se controla con:

`/proc/sys/vm/overcommit_memory`

Los sistemas modernos usan una cache:

- ▶ Es una forma de aprovechar la memoria física que no se usa para los procesos.
- ▶ El kernel mantiene una cache de páginas para mantener los datos de los ficheros, directorios, dispositivos de bloques (buffers), etc. que se están usando y evitar operaciones de E/S.
- ▶ La mayoría de las llamadas `read` y `write` se satisfacen a través de esta cache.

- ▶ Muy eficiente: en algunos sistemas se ahorra hasta el 85 % de operaciones de E/S.
- ▶ En ocasiones es necesario saltarse la cache (p. ej. la opción `O_DIRECT` de `open` en Linux).
- ▶ El comando y llamada al sistema `sync` sincroniza las páginas *sucias* de la cache, las baja a disco. Se ejecuta periódicamente.

Mmap

- ▶ `mmap`: esta llamada al sistema permite crear una región nueva para el proceso.
- ▶ Se llama *mmap anónimo* cuando simplemente queremos una nueva región de memoria inicializada a cero.
- ▶ También permite proyectar un fichero en memoria, para acceder al fichero sin usar las llamadas al sistema `read`, `write`, etc.
- ▶ **No sustituye a esas llamadas**: no es apto para ficheros sintéticos (p. ej. `/proc`), ficheros que crecen, ficheros pequeños, sistemas de ficheros en red, etc.

```
void *mmap(void *addr, size_t length, int prot,  
           int flags, int fd, off_t offset);  
int munmap(void *addr, size_t length);
```

- ▶ En Linux podemos ver el estado de la memoria en
`/proc/meminfo`
- ▶ Campos:
 - ▶ MemTotal: memoria física usable (total menos el binario del kernel)
 - ▶ MemFree: memoria libre
 - ▶ Buffers: bloques de dispositivos en la page cache
 - ▶ Cached: la page cache
 - ▶ Dirty: memoria sucia que tiene que bajar a disco.
 - ▶ ...

Algoritmos de reemplazo

- ▶ Problema: cuando usamos respaldo (swap) y nos quedamos sin marcos de página, necesitamos expulsar marcos.
- ▶ Hay distintos algoritmos de reemplazo.
- ▶ La solución óptima (minimizar fallos) no es viable: consistiría en reemplazar la página que no se va a usar en el periodo de tiempo más largo. → ¡hay que saber la secuencia de accesos de antemano! → no nos sirve.

Algoritmos de reemplazo: FIFO

- ▶ Se reemplaza la página más vieja (la que lleva más tiempo).
- ▶ Problema: la antigüedad no tiene que ver con frecuencia de uso.
- ▶ Anomalía de Belady: siendo $N > M$, el número de fallos para N marcos puede ser mayor que para M marcos, (*ojo: no es exclusiva de FIFO*).
- ▶ Secuencia: 1,2,3,4,1,2,5,1,2,3,4,5
Con 4 marcos: 10 fallos.
Con 3 marcos: 9 fallos.

Algoritmos de reemplazo: LRU

- ▶ Idea: reemplazar la página que no ha sido usada desde hace más tiempo (*Least-Recently Used*).
- ▶ Posible implementación: apuntar en la tabla de páginas el valor de un contador global cuando se accede a la página. Se reemplaza la página con el contador más bajo.
- ▶ Problema: necesita apoyo del hardware para marcar el tiempo porque no es admisible una interrupción por acceso para que el sistema operativo se encargue de esto → ninguna máquina lo ofrece.

Algoritmos de reemplazo: NRU

- ▶ Se reemplaza una página que no ha sido usada recientemente.
- ▶ Se usan dos bits: bit de referencia y bit de sucio.
- ▶ Cuatro clases en orden de prioridad ascendente:
 1. no referenciada / limpia: ideal para reemplazar.
 2. no referenciada / sucia: hay que escribirla de vuelta.
 3. referenciada / limpia: puede ser referenciada de nuevo.
 4. referenciada / sucia: la peor opción.
- ▶ El bit de referencia se pone a 0 periódicamente.
- ▶ Problema: el bit de referencia indica que no se ha usado recientemente, pero no indica orden.

Algoritmos de reemplazo: segunda oportunidad

- ▶ FIFO dando una nueva oportunidad a las páginas con el bit de referencia a 1:
 1. se pone el bit de referencia a 0.
 2. se coloca al final de la cola (esto es, se pone el tiempo de llegada al tiempo actual).
- ▶ Se puede implementar con un array circular (reloj).
- ▶ Degenera en FIFO si todas las páginas están referenciadas (con una vuelta adicional).

Otras tácticas del paginador

- ▶ Siempre mantener un conjunto de marcos libres de reserva.
- ▶ Desalojar marcos de páginas de la Page Cache. P. ej. los ficheros binarios que no está ejecutando ningún proceso actualmente.
- ▶ En ratos ociosos se pueden escribir las páginas sucias a disco y marcarlas como limpias.
- ▶ Se pueden desalojar páginas dejando el contenido en el marco (aunque esté libre, se queda con el contenido). Si después hay que traer la misma página, no hace falta I/O.

Asignación de marcos a los procesos

Hay distintas políticas:

- ▶ **De forma equitativa:** a todos lo mismo
- ▶ **De forma proporcional** al tamaño del proceso en memoria.
Siendo s_i el tamaño en memoria del proceso i ,

$$S = \sum s_i$$

y M el número de marcos libres, al proceso i le corresponden:

$$a_i = \frac{s_i M}{S}$$

- ▶ Teniendo en cuenta la prioridad del proceso.
- ▶ ...

Asignación de marcos a los procesos

- ▶ **Asignación local:** se desaloja una página del proceso que causa el fallo de página.
- ▶ **Asignación global:** se desaloja una página de cualquier proceso.

Thrashing

- ▶ Cuando el sistema gasta más en paginación que en procesamiento útil.
- ▶ Causas:
 - ▶ aumento del grado de multiprogramación
 - ▶ asignación global
- ▶ Efecto: los procesos se roban marcos entre ellos.

Conjunto de trabajo

- ▶ Solución al *thrashing*: tener marcos suficientes para el **conjunto de trabajo de cada proceso**.
- ▶ **Conjunto de trabajo**: conjunto de páginas en las Δ referencias más recientes (*working set window*).
- ▶ Se mantiene el número de marcos asignado a un proceso igual al número de páginas de su conjunto de trabajo.
- ▶ Se estima si la creación de otro proceso provocará *thrashing* mirando el tamaño del conjunto de trabajo de todos los procesos.