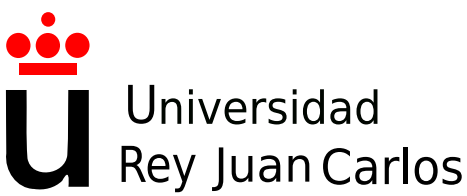
Tema 6: Gestión de memoria

Enrique Soriano

Laboratorio de Sistemas,
Grupo de Sistemas y Comunicaciones,
URJC

26 de abril de 2011



Asignación dinámica de memoria

- Gestión implícita (e.g. Java): un *recolector de basura* (Garbage Collector) se encarga de liberar memoria no referenciada.
- Gestión explícita (e.g. C):
  `malloc`: reserva memoria dinámica.
  `free`: libera memoria dinámica.
- La memoria dinámica está en el montón o *heap*, después del BSS. El montón crece en demanda.
- Una vez comprometida una reserva, no se puede mover.
- Estrategia, política, mecanismo.

Fragmentación externa

- 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.
- La compactación solucionaría el problema... ¿se puede si hablamos de memoria dinámica?

Fragmentación interna

- Solución parcial a la fragmentación externa: reservar en base a bloques fijos $\longrightarrow$ un hueco libre siempre puede ser útil.
- No perdemos 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.

Estrategias

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

Políticas de asignación dinámica de memoria

En realidad, tienen distintas aplicaciones: gestión de particiones de memoria, caches, malloc, etc.

- **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.
- **Worst Fit:** el que se ajuste peor. Peor resultado que los anteriores.
- **Quick Fit:** se mantiene una serie de trozos de varios tamaños populares para reservas rápidas. Se comporta bien.

Ejemplo real: glibc 2.3

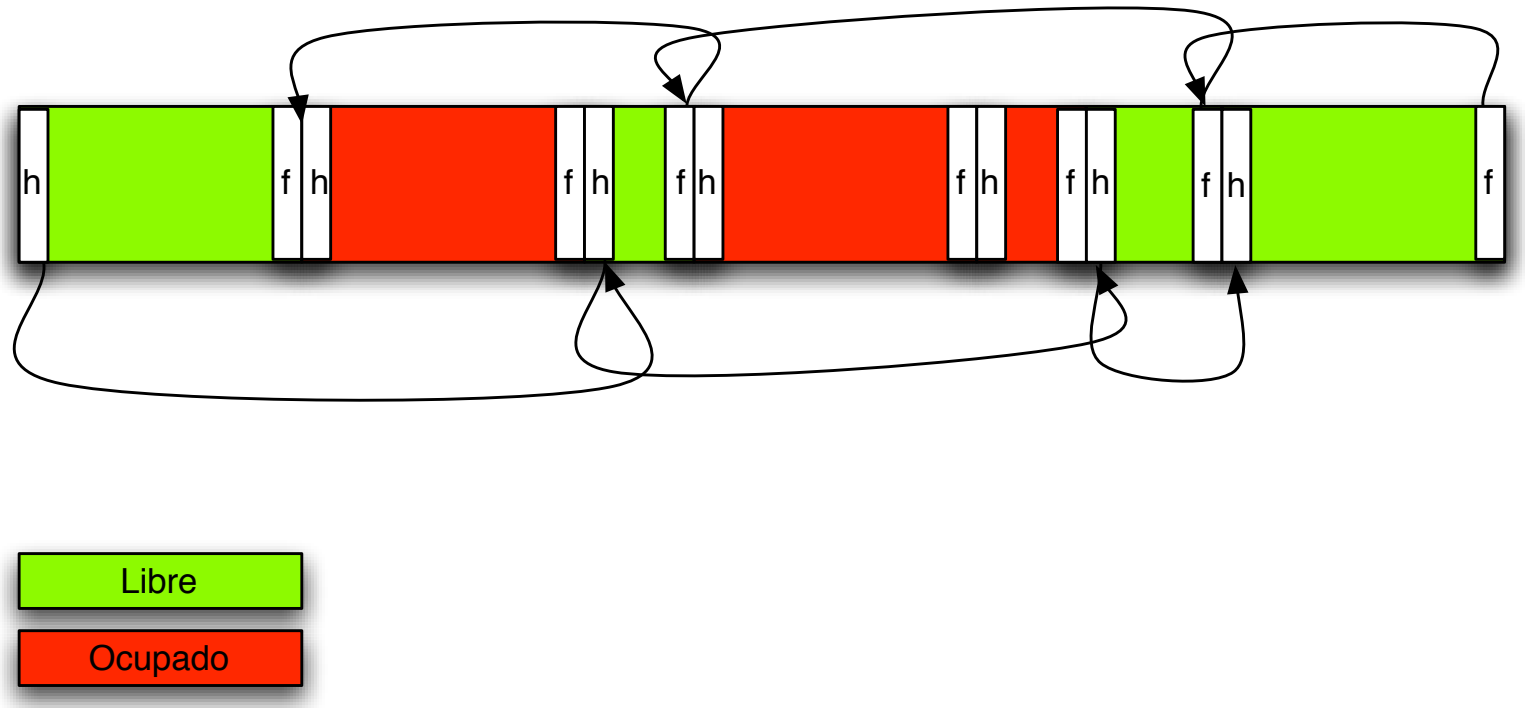
- Para peticiones grandes $>=$ 512 bytes, un asignador Best Fit puro.
- Para peticiones pequeñas $<=$ 64 bytes, un asignador rápido con trozos de ese tamaño.
- Para peticiones intermedias usa una política mezcla de las anteriores.
- Para peticiones muy grandes $>=$ 128 Kb, realiza un `mmap` anónimo para solicitar al kernel memoria que no irá en el *heap*.

Mecanismos para asignación dinámica de memoria

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

Ejemplo: doble lista de libres, con cabeceras y pies

Footer / Header: estructuras con el estado (libre, ocupado), tamaño y puntero al siguiente/anterior libre.



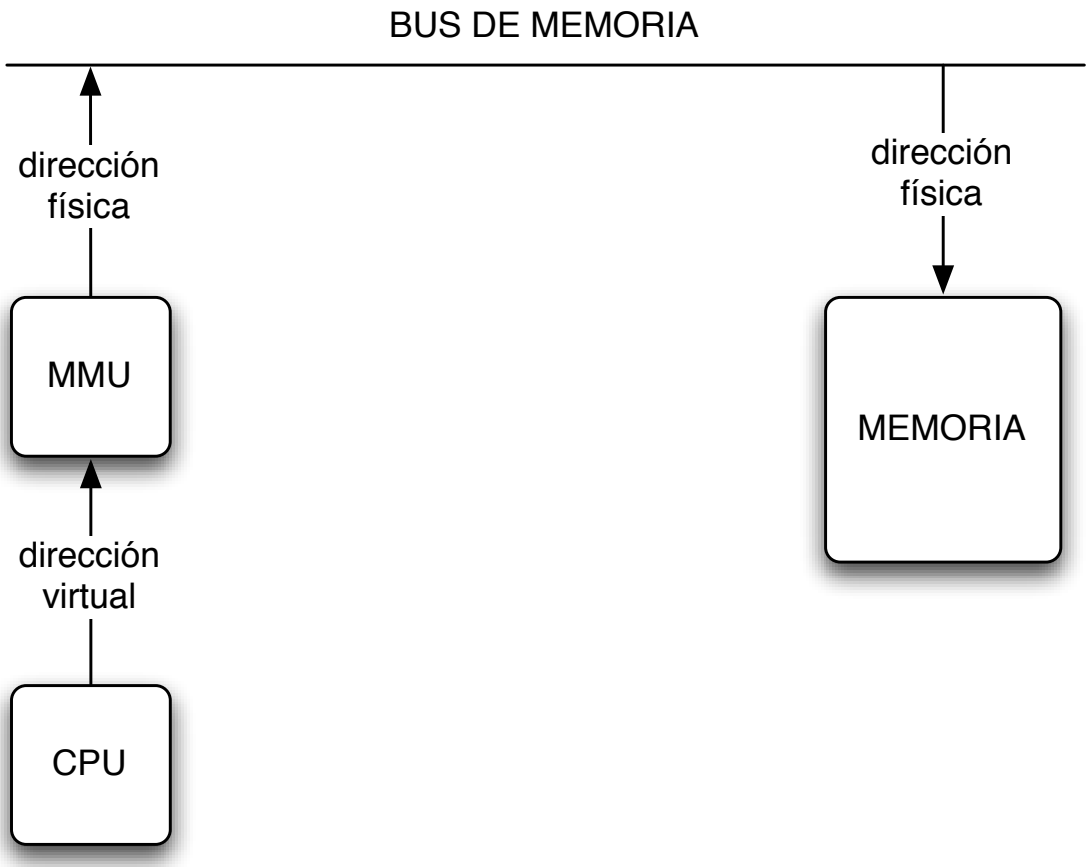
Vinculación (Binding)

Tres alternativas:

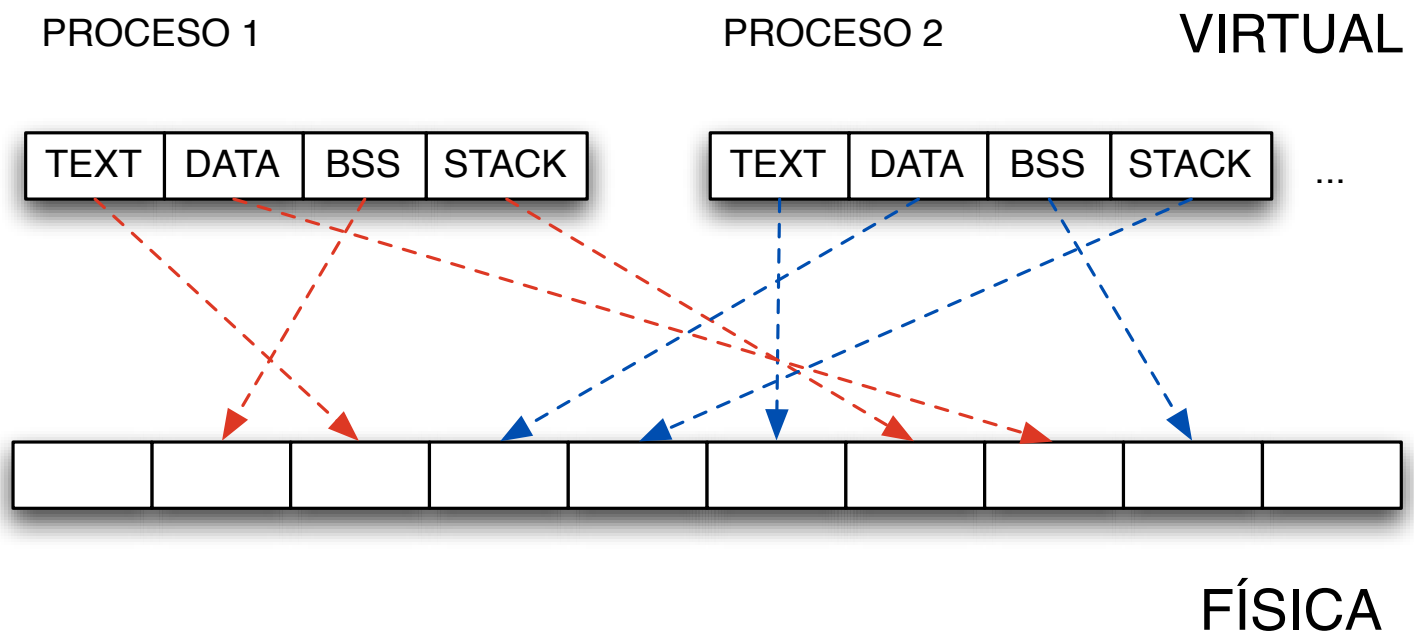
- **En tiempo de compilación: código absoluto.**
  Dirección lógica $\equiv$ Dirección física
- **En tiempo de carga: código relocalizable.**
  Dirección lógica $\equiv$ Dirección física
  Se comprometen las direcciones en el momento de carga.
- **En tiempo de ejecución.** $\leftarrow$
  Dirección lógica (dirección virtual) $\neq$ Dirección física
  Se necesita apoyo del hardware para mapear: MMU.

Memoria virtual

¡La CPU nunca ve direcciones físicas!



Memoria virtual



Intercambio (Swap)

- El espacio de memoria (parte o todo) de algunos procesos se mueve a un dispositivo de almacenamiento (disco).
- 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
- Cuando teníamos poca memoria, era muy útil. ¿Ahora?

Memoria virtual

Permite:

- Aumentar la memoria usando almacenamiento secundario (antes era necesario, ahora hay sistemas que no lo implementan).
- Crear la ilusión de un espacio *virtual* de direcciones grande y contiguo. $\longrightarrow$ Crear la ilusión de que el proceso está ejecutando en su propia máquina *virtual* $\longrightarrow$ simplicidad.
- Reusar/compartir memoria.
- Proteger la memoria de los distintos procesos (errores y ataques).

Particiones (muy antiguo)

- El espacio de memoria se divide en particiones.
- Partición: memoria de un único proceso.
- Hasta que no hay memoria disponible para un proceso, el programa no se pone a ejecutar.
- Antiguo: en sistemas de procesamiento por lotes.
  P. ej. IBM OS/360.
- Protección: MMU con dos registros
  - Registro límite (LR): dirección lógica máxima del proceso.
  - Registro de relocalización (RR): offset para la dirección lógica.

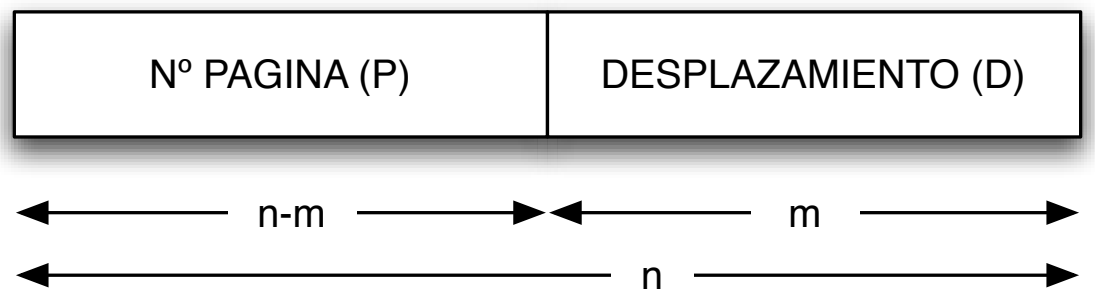
```
if logical > lr
   trap
else
   physical = logical + rr
```

Paginación

- Objetivo: 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 medio de almacenamiento para *swapping* se divide en porciones del tamaño de un marco.

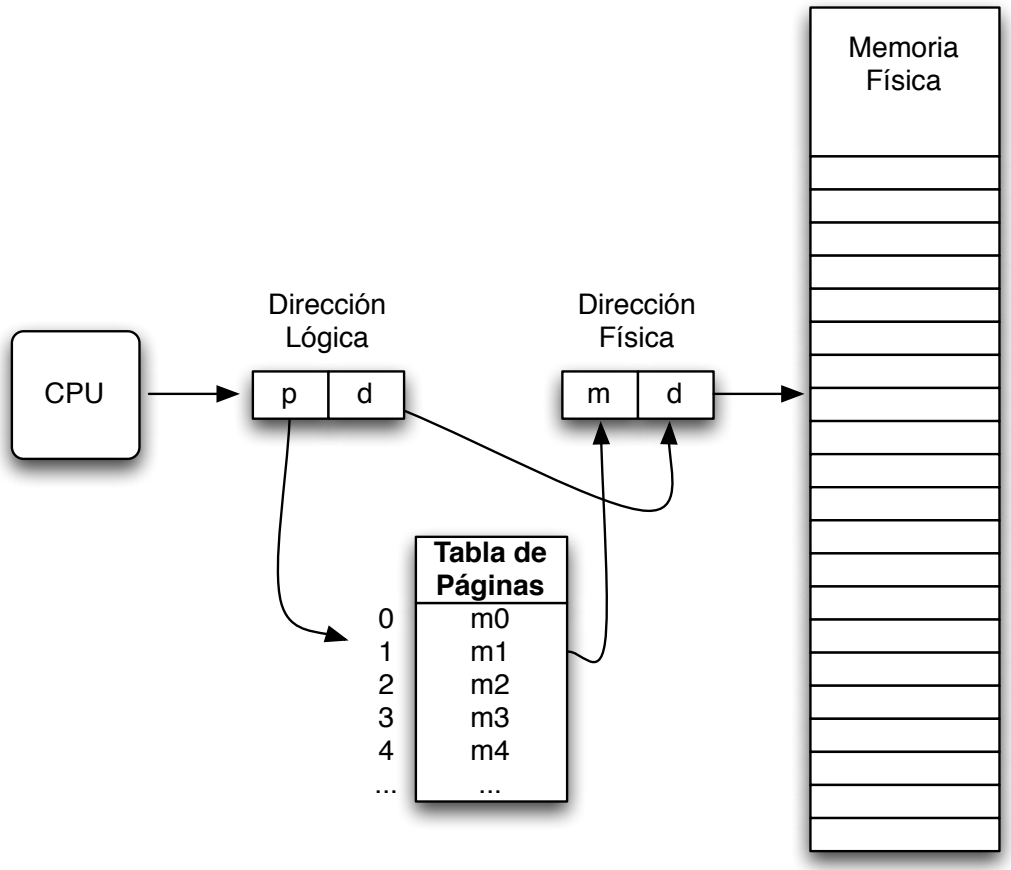
Paginación

- 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.
- El hardware determina el tamaño de página.
- Dirección lógica:



2^m = tamaño de página

Paginación


Memoria
Física
CPU
Dirección
Lógica
p
d
Dirección
Física
m
d
Tabla de
Páginas
0
1
2
3
4
...
m0
m1
m2
m3
m4
...

Paginación

- Fragmentación interna: promedio de media página perdida por proceso.
- Si el tamaño de página es pequeño...
  - hay menos fragmentación.
  - necesitamos tablas de página con muchas entradas $\longrightarrow$ 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.
- Superpáginas: hasta 256Mb.

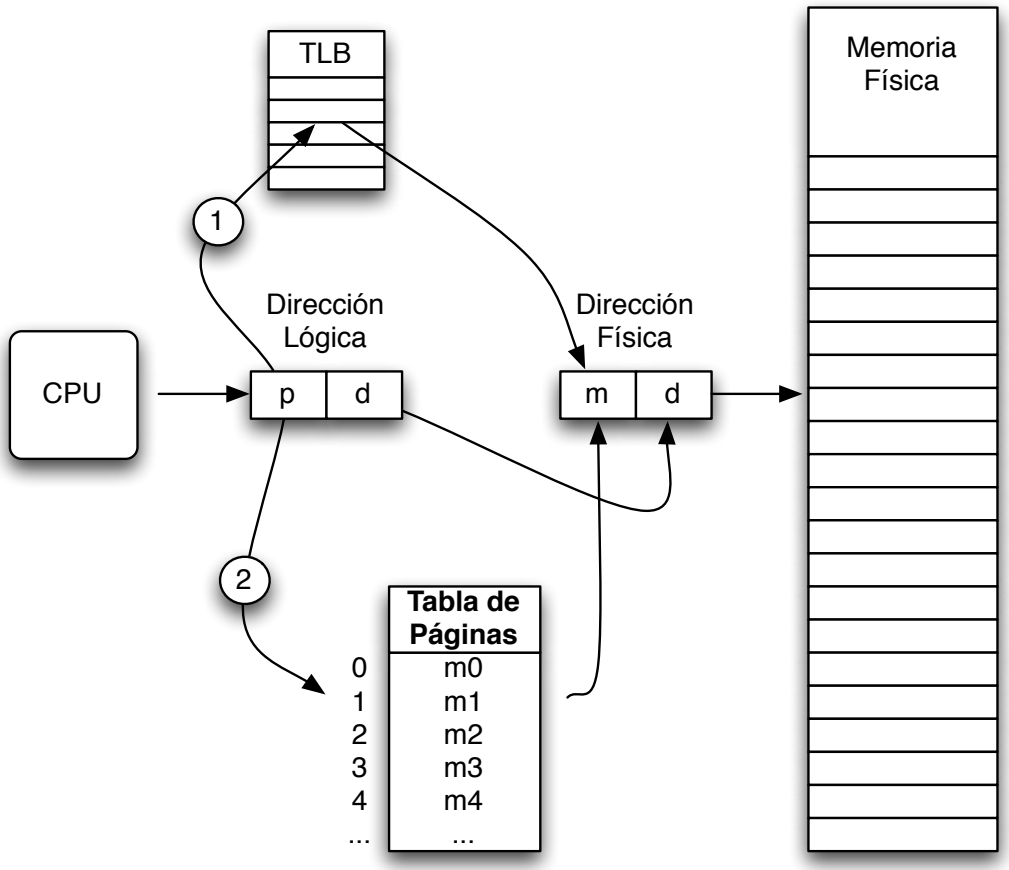
Paginación

- Cada proceso tiene su tabla de páginas, que forma parte de su contexto (registros, PC, SP, etc.).
- Hace mucho se usaban registros para almacenar la tabla de páginas.
- Ahora tiene que estar en memoria (¡son grandes!).
  Se usan registros para apuntarla (los detalles dependen de la arquitectura).
- Problema: dos accesos a memoria principal para cada acceso real:
  tabla de páginas $\longrightarrow$ acceso a memoria.

Paginación: TLB

- TLB (*Translation Look-aside Buffer*): pequeña memoria cache de la tabla de páginas, 8-2048 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 limpia 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:

$$T_{acceso-efectivo} = P_{acierto} * T_{acierto} + P_{fallo} * T_{fallo}$$

e.g.

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

penalización del 22 % en el acceso.

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, si se ha accedido...

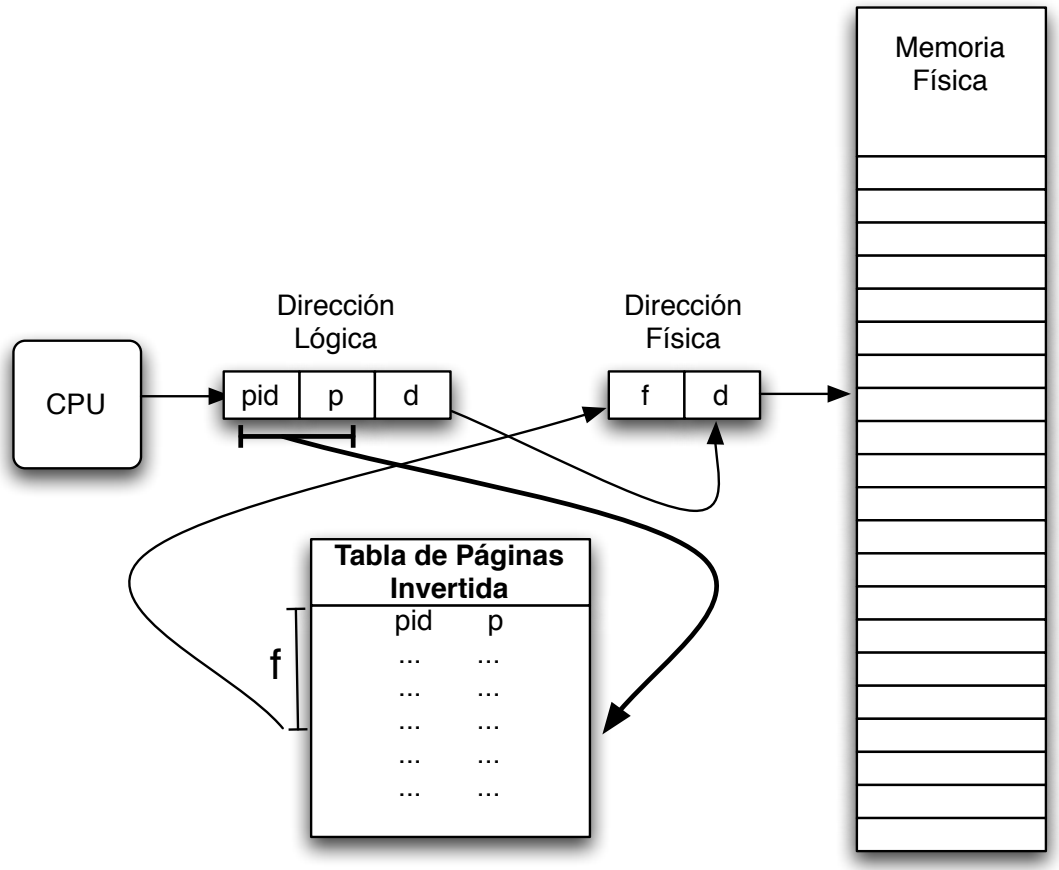
Paginación: tabla de páginas

- Problema: espacios de direcciones demasiado grandes: 2^{64}.
- Tabla de páginas excesivamente grande como para tener todas sus entradas contiguas.

Paginación: tabla de páginas invertida

- Una solución: mantener una única tabla de páginas invertida (IPT) para todos los procesos.
- Una entrada por cada marco de página: ordenada por dirección física.
- Dirección lógica: **(PID, p, d)**. Se busca en la tabla **(PID, p)**.
- Ejemplos de arquitecturas: UltraSPARC, PowerPC.

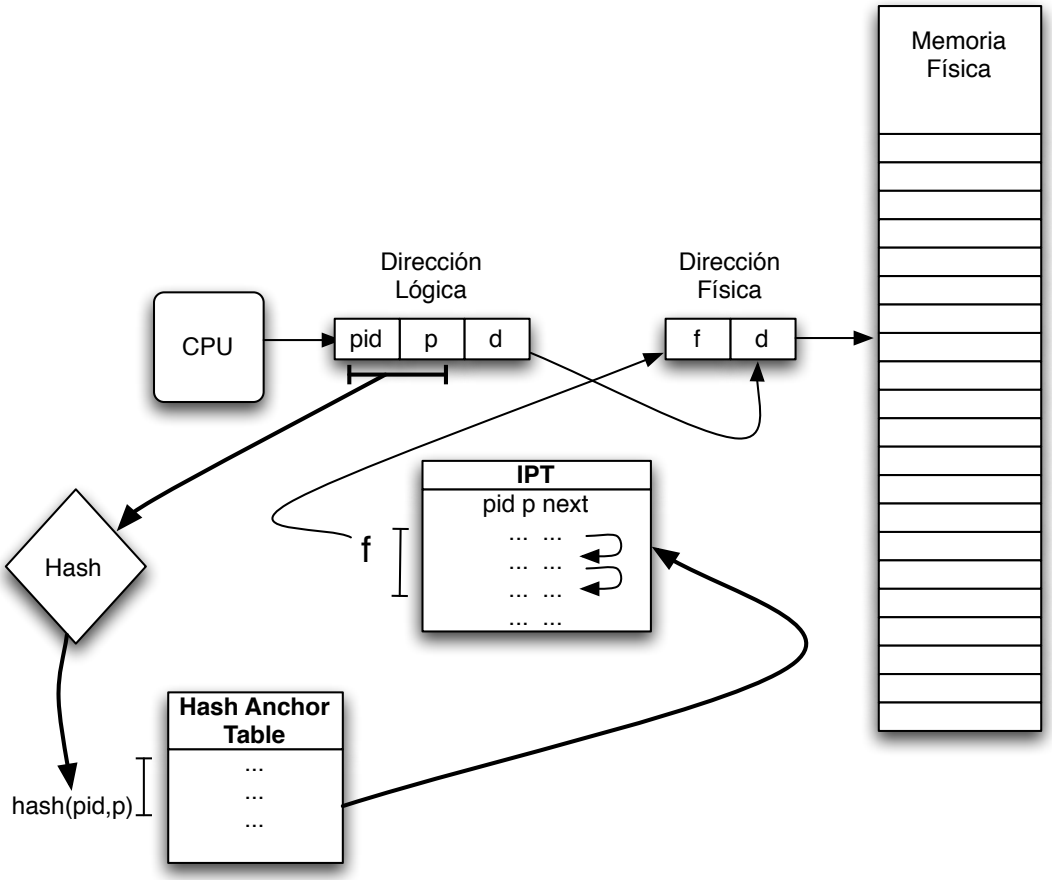
Paginación: tabla de páginas invertida



Paginación: tabla de páginas invertida

- Problema: la tabla no está ordenada por dirección lógica $\rightarrow$ búsqueda muy lenta.
- Solución: tabla hash con $Hash(PID, p)$.
- En caso de colisión, hay que recorrerse una lista de desbordamiento (hash chain).
- Efecto: más accesos a memoria para conseguir la dirección física al pasar por la tabla hash.

Paginación: tabla de páginas invertida con hash



Paginación multinivel

- Otra solución: dividir la tabla de páginas en N 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:

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

Paginación: compartiendo páginas

- El código (*text*) de un binario se puede compartir si es reentrante $\longrightarrow$ si no se modifica durante la ejecución.
- Los procesos que comparten memoria (*data*, *bss*, *heap*) también pueden compartir páginas.
- Copy-on-write.
- Las plataformas con tabla de páginas invertida tienen problemas para esto ¿Por qué?

Paginación en demanda

- Sólo se traen a memoria física las páginas que se necesitan, no todas las páginas del programa.
- Aproximación vaga: las páginas se van trayendo cuando se necesitan.
- Overcommit: tras una llamada a `malloc`, se hace crecer el segmento de BSS pero no se compromete memoria física hasta que se intenta usar (genera un fallo de página). Cuando se accede por primera vez a una página de la memoria reservada, se compromete el marco de página. Lo mismo pasa con las variables globales sin inicializar (BSS).
  ¿Cuándo falla tu programa si el sistema se queda sin memoria?

Paginación en demanda

Si salta un trap por fallo de página:

1. Se mira si la dirección es correcta o no (en una tabla del PCB).
2. Si es incorrecta, se acaba con el proceso directamente o se le interrumpe (nota).
3. Si es correcta, se busca un marco para ella.
4. Se inicia una trasferencia de I/O desde el disco al marco de página.
5. Se modifica la tabla de páginas para poner su bit de presente.
6. Se ejecuta la instrucción de nuevo y puede acceder a la dirección de memoria.

Paginación en demanda

Tiempo de acceso efectivo

- Proporcional a la tasa de fallo p:

$$T_{efectivo} = (1 - p) * T_{acierto} + p * T_{fallo}$$

- $T_{acierto} \approx 100$ nanosegundos.
- $T_{fallo} \approx 10$ milisegundos $= 10000000$ nanosegundos.
- Algoritmo de reemplazo: cuanto menos tasa de fallo, mejor.
- Buena noticia: localidad de referencia.

Algoritmos de reemplazo: FIFO

- Se reemplaza la página más vieja.
- Problema: 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: óptimo

- Se reemplaza la página que no se va a usar en el periodo de tiempo más largo.
- Es la que minimiza la tasa de fallos.
- No sufre la anomalía de Belady.
- ¡hay que saber la secuencia de accesos de antemano! $\longrightarrow$ no sirve.

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.
- Necesita apoyo del hardware para marcar el tiempo: no es admisible una interrupción por acceso para que lo haga el SO $\longrightarrow$ 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).

Algoritmos de reemplazo: reloj con dos manecillas

- La primera manecilla marca limpia el bit de referencia.
- La segunda elige víctima si el bit está a 0.
- Si en el intervalo entre el paso de ambas no se ha accedido, es una víctima.

Otras tácticas del paginador

- Se mantiene un buffer de marcos libres de reserva. Se desalojan páginas para mantener el buffer lleno.
- 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 (libre). Si después hay que traer la misma página, no hace falta I/O.

Asignación de marcos a los procesos

- **De forma equitativa**.
- **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:

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

- 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 desalojan una página de cualquier proceso.

Thrashing

- 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 (*working set window*) referencias más recientes.
- 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* con tamaño del conjunto de trabajo de cada proceso (*Working Set Size*). La demanda total de marcos es D:

$$D = \sum WSS_i$$