Sistemas Operativos
Tema 4. Memoria virtual

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

© 2015 – José Miguel Santos Espino

Contenidos

- Características de la memoria virtual
- Memoria virtual paginada
- Algoritmos de reemplazo de páginas
- Asignación de marcos
- Conjunto de trabajo (working set)
- ~~Ficheros mapeados en memoria~~

ULPGC

El objetivo

- ¿Cómo podemos sacarle el jugo a la memoria?
- Un proceso no utiliza todo su código y sus datos a lo largo de su vida.
    - Word: editar macros, guardar como PDF, comparar documentos, combinar correspondencia…
- Si pudiéramos cargar en memoria principal sólo lo que se necesita en cada momento, podríamos tener más espacio disponible para otros procesos.

El objetivo

- Incluso puede ocurrir que un programa ocupe más que la memoria principal (ej. si utiliza estructuras de datos muy grandes).
- En este caso, es inevitable recurrir a alguna técnica para permitir la carga parcial del código o los datos.

Carga parcial de programas

- Históricamente se han inventado técnicas para conseguir que no todo el código/datos esté en memoria principal:
  - Ficheros para guardar datos
  - Recubrimientos (overlays) → técnica primitiva
  - Intercambio (swapping) → enviar al disco procesos completos que lleven bloqueados mucho tiempo
  - Bibliotecas dinámicas (DLL) → cargar código compartido bajo demanda
  - **MEMORIA VIRTUAL**

Memoria virtual

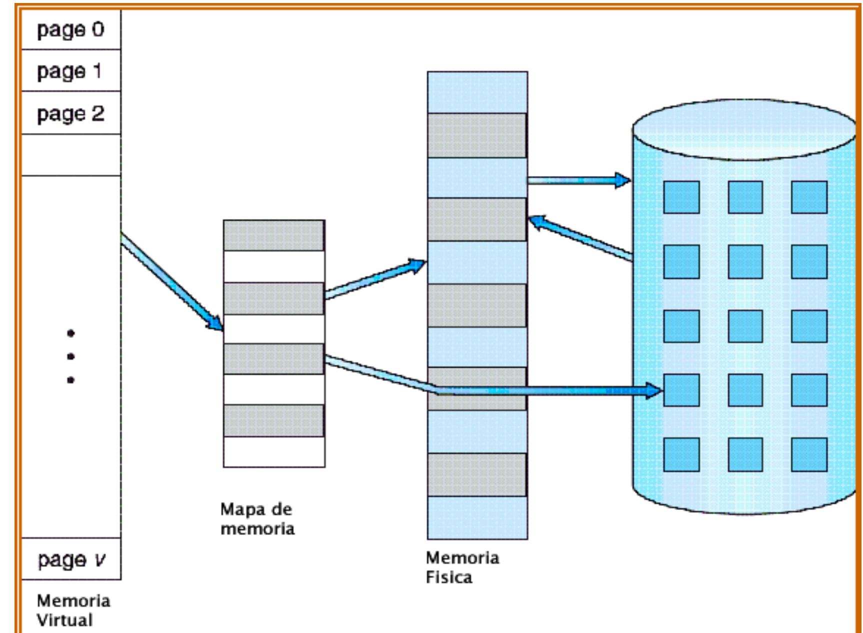
- Algunas áreas de la memoria lógica del proceso se encuentran en memoria principal y otras en el almacenamiento secundario.
- Es el sistema operativo el que se encarga de gestionar qué áreas están en memoria secundaria. *(el programador de aplicaciones no interviene en esta gestión)*
- El sistema operativo intenta que en memoria principal se encuentren las zonas de código y de datos que se vayan a utilizar más en cada momento. *(hay que evitar que la lentitud de la memoria secundaria penalice demasiado el rendimiento)*

Localidad

- La mayoría de los programas exhibe una fuerte **localidad** → en cada momento, los accesos a código y a datos se concentran sobre áreas pequeñas del espacio de memoria.
- Esta es la base para que funcione la memoria virtual → en cada momento, si conseguimos traer a memoria principal sólo esas pequeñas áreas, los accesos al disco serán mínimos.
- ¿cómo *adivinar* cuáles son esas áreas? Esa es la parte difícil.
- Vamos primero con la arquitectura del sistema…

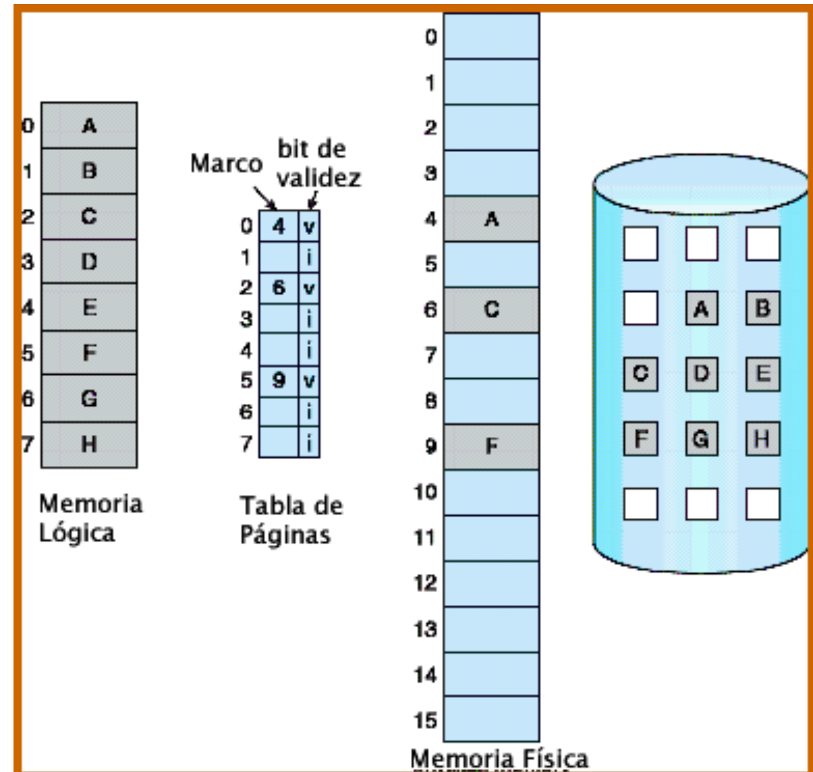
Memoria virtual paginada

- Espacio lógico paginado
- Algunas páginas en m.p. y otras en disco
- En la tabla de páginas se marca qué páginas están en memoria principal → **bit de validez**



Bit de validez + fallo de página

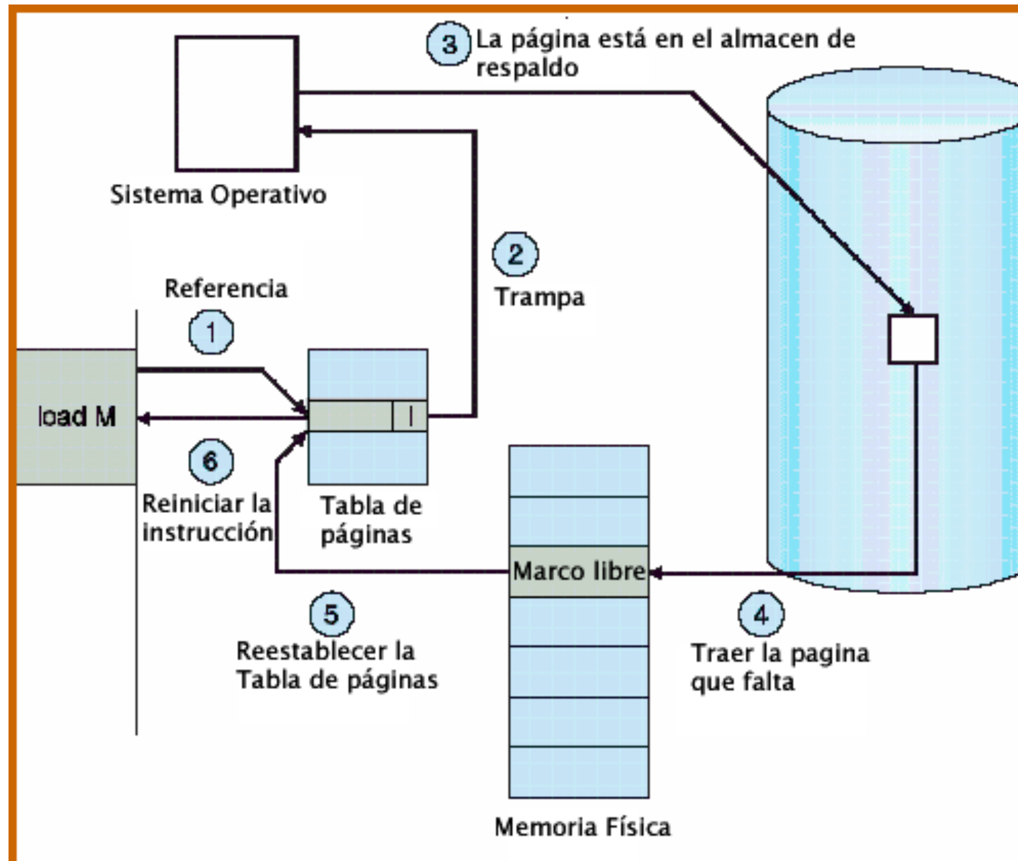
- Cada entrada de la tabla de páginas tiene anotado si es *válida* = está en memoria principal
- Si la página es *inválida* y se intenta acceder a ella, se genera una excepción → **fallo de página**



Fallo de página (*page fault*)

- Si la CPU intenta acceder a una página marcada como no válida (bit de validez activado), la MMU genera una excepción llamada **fallo de página** (*page fault*).
- El fallo de página activa una rutina del SO, que hace lo siguiente:
  1. Busca un marco físico libre para la página solicitada.
  2. Copia del disco la página solicitada, en el marco elegido.
  3. Actualiza la tabla de páginas (marca la entrada como válida y anota en qué marco físico copió la página).
  4. Entrega el control al proceso → se reintenta la instrucción que provocó el fallo de página.

Gestión de un fallo de página



Gestión de un fallo de página

- Mientras el SO va leyendo del disco la página que originó el fallo (paso 4 de la figura anterior), se puede entregar la CPU a otros procesos que estén disponibles.
- Ojo, también hay que invalidar la entrada de TLB.
- Ojo al reinicio de la instrucción:
  - una instrucción de CPU puede generar *varios* fallos de página (ej. en x86: `ADD [EAX],[EBX]` genera tres accesos a datos)
  - La instrucción pudo haber hecho modificaciones antes de generar el fallo de página

ULPGC

¿Qué páginas cargamos inicialmente?

- ¿Qué páginas del proceso cargamos inicialmente?
- Solución minimalista: NINGUNA
  → **paginación bajo demanda (*demand paging*)**
- Las páginas se van cargando a medida que se necesitan (se van resolviendo fallos de página)
- Otras opciones más complejas:
  - Precargar las primeras páginas de código
  - Registrar el comportamiento de las aplicaciones y con esa información, precargar las páginas más probables

¿Y si no hay marcos libres?

- ¿Qué ocurre si hay un fallo de página y todos los marcos físicos están ocupados?
- Hay que elegir una página *víctima* de las que están en memoria física y sustituirla con la página solicitada.
- ¿Cuál elegimos?
  - ¿la menos usada? ¿la más vieja?
  - ¿elegimos una página del proceso que causó el fallo, o elegimos de entre toda la memoria?
  - ¿qué pasa si la víctima es una página que se ha modificado?

ALGORITMOS DE REEMPLAZO

Algoritmos de gestión de memoria virtual

- **Políticas de asignación de marcos**
  - Cuántos marcos se asignan a cada proceso
- **Políticas de reemplazo de páginas**
  - Qué página(s) sustituimos cuando hay un fallo de página y la memoria física está llena

Reemplazo de páginas

- Algoritmo general:
  - 1. Ocurre un fallo de página.
  - 2. Se observa que no hay marcos libres.
  - **3. Se elige una página víctima.**
  - 4. Si la víctima ha sido modificada, guardarla en disco.
  - 5. Leer la nueva página del disco.
  - 6. Marcar la página víctima como inválida en la t.pág.
  - 7. Actualizar la entrada de la nueva página (bit de validez y marco físico al que apunta).

Bit de modificación

- ¿Cómo sabemos si la página víctima se ha modificado?
- Añadimos un *bit de modificación* en las entradas de la tabla de páginas.
- Inicialmente el bit de modificación está a 0.
- La MMU pone a 1 ese bit cada vez que se hace una escritura sobre la página correspondiente.

Algoritmos de reemplazo: objetivos

- Hay que tratar de minimizar la frecuencia de fallos de página.
- Por tanto, hay que elegir páginas víctimas que no se vayan a utilizar mucho en el futuro.
- Algunas ideas:
  - La menos recientemente usada (LRU)
  - La menos frecuentemente usada (LFU)
  - La más vieja (FIFO)

Algunos algoritmos de reemplazo

- *(en negrita, los que han tenido uso significativo)*
- Algoritmos *canónicos*:
  - **FIFO**
  - OPT / MIN – algoritmo óptimo
  - LRU – least recently used
  - LFU – least frequently used
- Aproximaciones realistas:
  - **NRU** – not recently used
  - **segunda oportunidad o reloj**
  - **aging**
  - NFU – not frequently used
  - **WSClock**

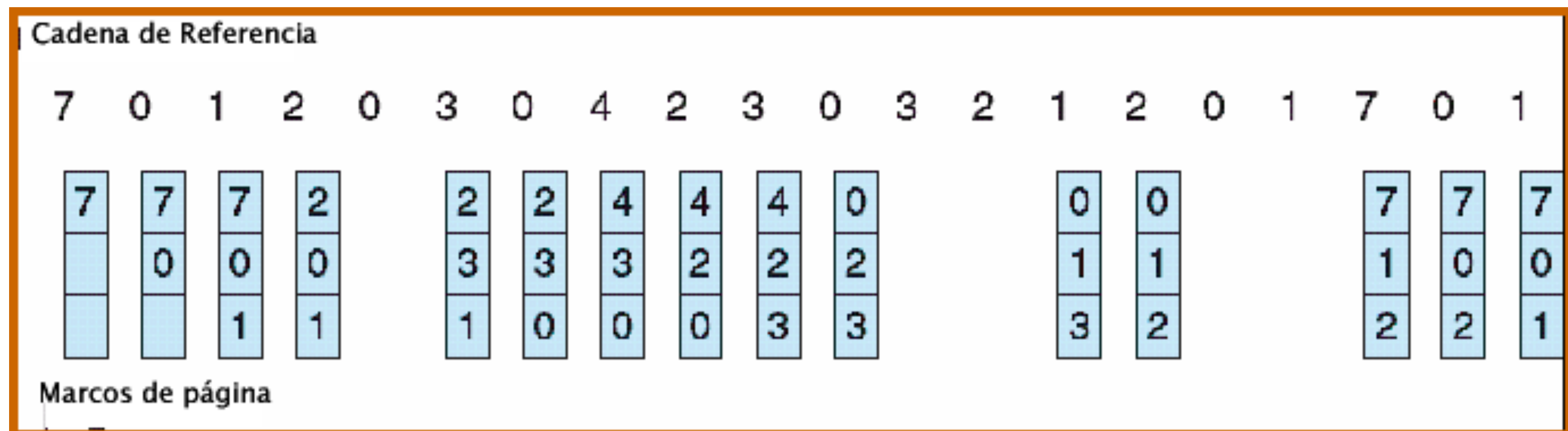
ULPGC

Algoritmo FIFO

- El primero que se utilizó en sistemas paginados.
- Se sustituye la página residente que lleve más tiempo en memoria.
- Muy fácil de implementar.
- No requiere hardware adicional.
- **Problema:** al no tener en cuenta el uso real de las páginas, FIFO puede prescindir de páginas a las que se accede con mucha frecuencia (ej. páginas del núcleo).

Ejemplo: FIFO

Simulación con una **cadena de referencias** a páginas lógicas

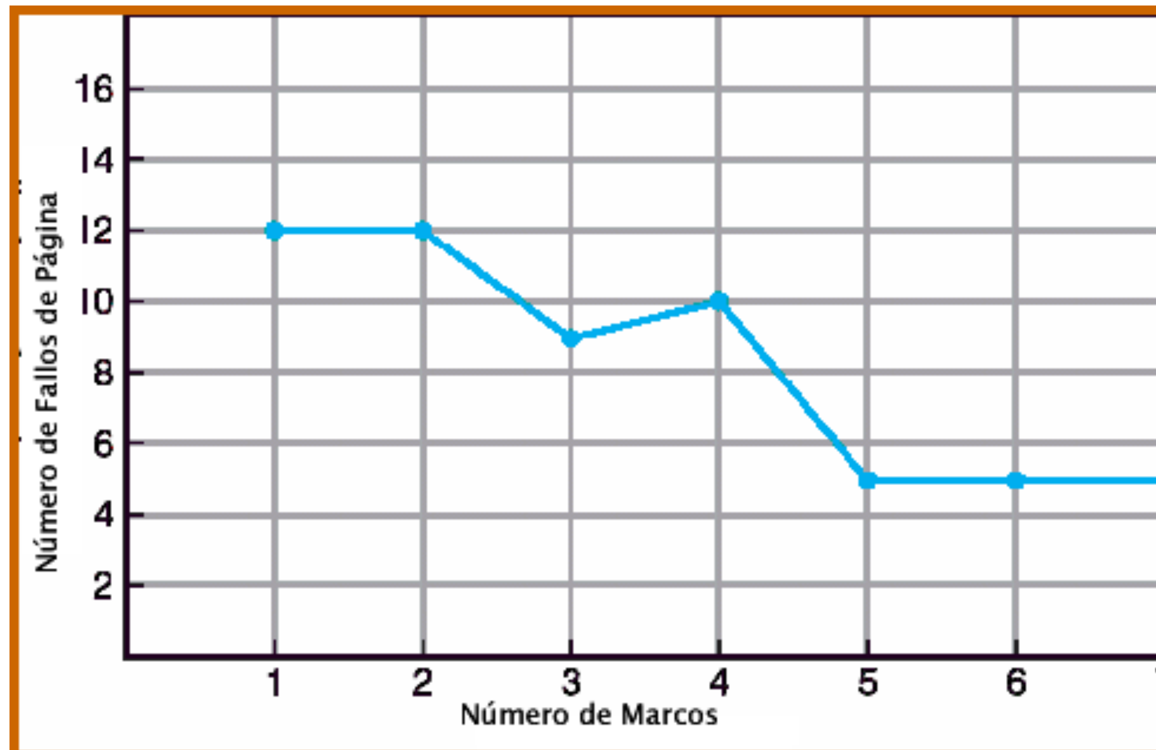


FIFO: anomalía de Bélády

- FIFO padece un fenómeno paradójico…
- …si aumentamos el número de marcos físicos, ¡puede aumentar la cantidad de fallos de página!
- Descrito por László Bélády (IBM) en 1969

FIFO: anomalía de Bélády

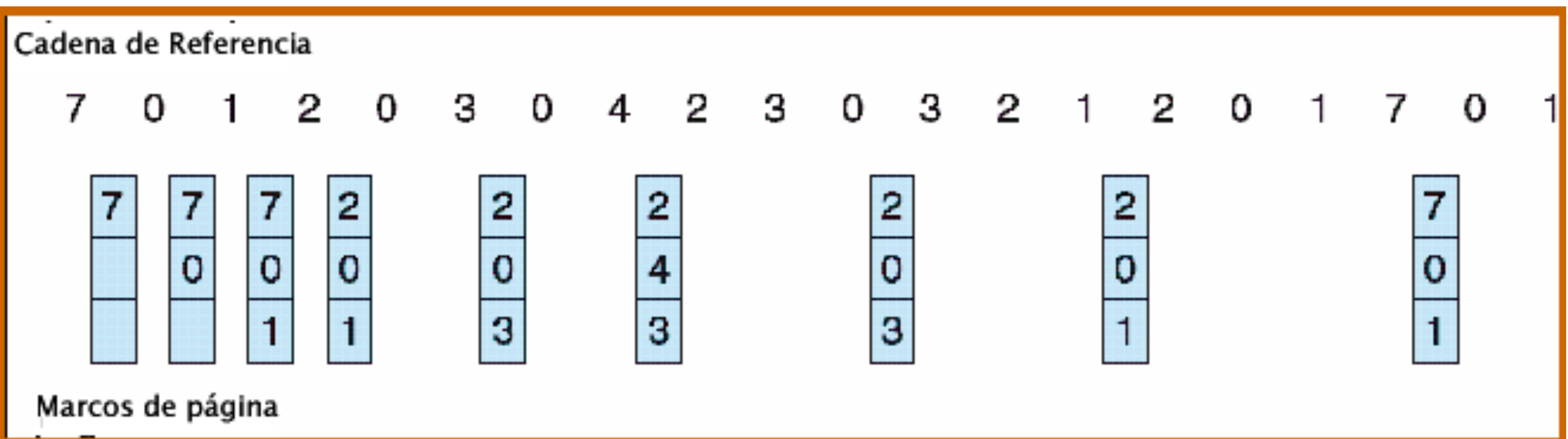
Cadena de referencias: 3 2 1 0 3 2 4 3 2 1 0 4



Algoritmo óptimo (OPT o MIN)

- Bélády demostró (1966) que si se escoge como víctima *la página que más tarde en volver a ser accedida*, el número de fallos de página es mínimo.
- «problemilla» → no implementable en el caso general (require saber cómo se va a comportar el proceso en el futuro)

OPT: ejemplo



¿Cómo podemos aproximarnos al reemplazo óptimo?

- El reto: adivinar cuál va a ser la página que más tarde vamos a acceder.
- Dos estrategias clásicas para adivinar:
  - La usada menos recientemente → **LRU**
  - La usada menos frecuentemente → **LFU**

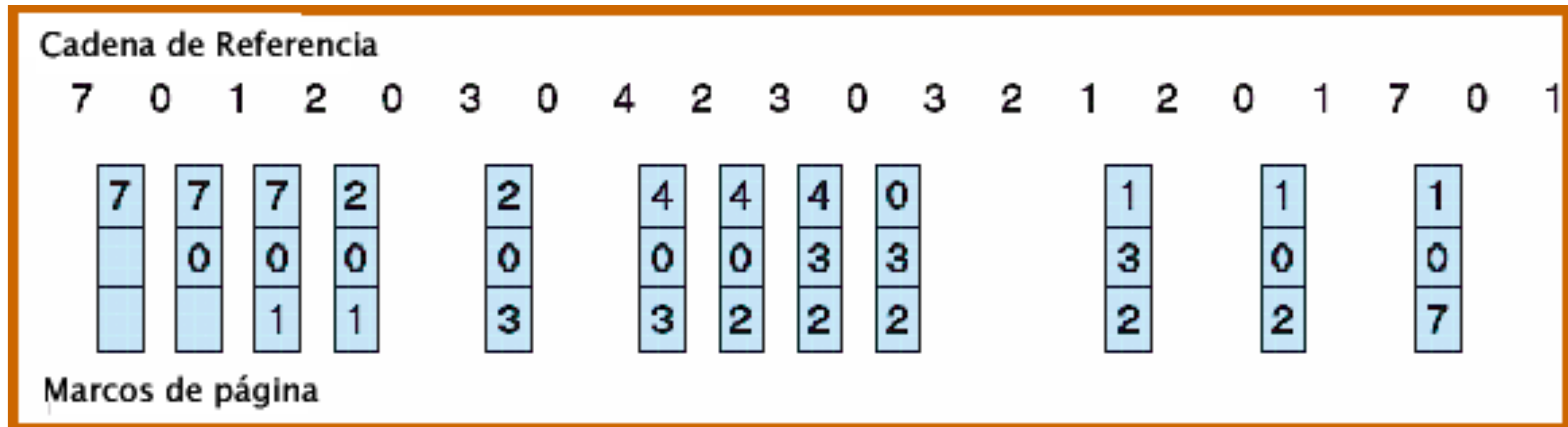
ULPGC

Algoritmo LRU

- LRU = least recently used / usada menos recientemente
- Tomamos como víctima la página que lleva más tiempo sin usarse
- Intuitivamente, parece una buena aproximación a OPT
- No padece la anomalía de Bélády
- Se puede implementar de dos maneras:
  - Poniendo marcas de tiempo a cada página
  - Manteniendo las páginas ordenadas según su tiempo de acceso → cada vez que se accede a una página, pasa al primer puesto de la lista
- Requiere *hardware* adicional y es costoso

ULPGC

LRU: ejemplo



Aproximación a la LRU: bit de referencia

- Cada entrada de la tabla de páginas tiene un *bit de referencia*.
- Inicialmente todas las páginas con el bit a 0.
- La MMU pone el bit a 1 cuando se hace referencia a la página.
- Las páginas con el bit de referencia a 0 son páginas «recientemente accedidas»… no es información exacta, pero sirve para tener un conjunto de *páginas candidatas a ser víctimas.*
- Este mecanismo es MUCHO más sencillo de implementar y más rápido que la lista ordenada que requiere la LRU.

Ejemplo con bit de referencia: algoritmo NRU

- NRU = Not Recently Used / no recientemente usado
- Cada cierto tiempo T, se interrumpe el sistema y se ponen a cero todos los bits de referencia
- Cuando ocurre un fallo de página, el SO distingue estas cuatro clases de páginas:
  - 0 – no referenciada, no modificada
  - 1 – no referenciada, modificada
  - 2 – referenciada, no modificada
  - 3 – referenciada, modificada
- Se escoge una página de la clase más baja posible (al azar o FIFO, según variantes del algoritmo)

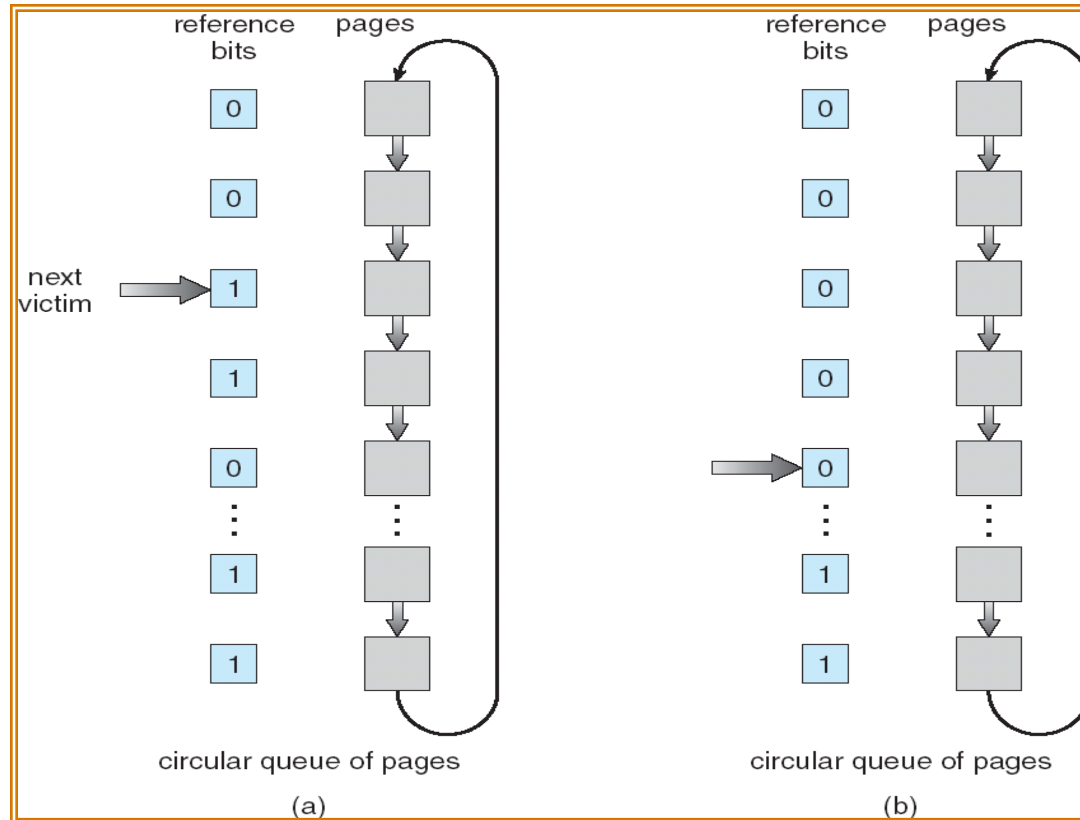
Recapitulando: bits en la tabla de páginas

- Para gestionar la memoria virtual, se han tenido que inventar estos tres bits para la tabla de páginas:
  - **Bit de validez:** nos dice si la entrada en la tabla apunta a una página residente en memoria.
  - **Bit de modificación:** nos dice si la página se ha modificado.
  - **Bit de referencia:** nos dice si la página se ha accedido recientemente.

Algoritmo de la 2ª oportunidad / del reloj

- FIFO + bit de referencia (R)
- Cuando hay un fallo de página, recorremos los marcos en orden de llegada. Para cada página:
  - Si R=0, la escogemos como víctima
  - Si R=1, le damos una *segunda oportunidad*: ponemos R=0 y seguimos buscando
  - Si llegamos al final de la lista, seguimos buscando desde el principio
- Si R=1 en todas las páginas, degenera en un FIFO.

Algoritmo de la 2ª oportunidad / del reloj



Varios bits de referencia (aging)

- Vamos tomando muestras del bit de referencia a intervalos regulares.
- Para cada página se guarda un *histórico* de sus últimos K bits, que forma un número binario, ej.:
  - 11000100
  - 01110111
- Tomamos como víctima la página con el histórico más pequeño (que será una de las menos recientemente accedidas).

Algoritmo LFU

- LFU = least frequently used / menos frecuentemente usada
- Mantenemos un contador del número de referencias
- Escogemos como víctima la que tenga el valor más bajo
- Implementación costosa
- No funciona muy bien, comparado con LRU
- Ningún sistema real lo implementa

Aproximación a LFU: NFU

- NFU = Not Frequently Used
- Mantenemos un contador software por cada página.
- Periódicamente muestreamos los bit de referencia.
- Si para una página R=1, incrementamos su contador.
- Cuando hay un fallo de página, tomamos como víctima la que tiene el contador más bajo.
- Implementable sólo con el bit de referencia.
- No se utiliza en la práctica, no es muy eficiente.

ASIGNACIÓN DE MARCOS

Asignación de marcos a los procesos

- Es conveniente definir un sistema de reparto de los marcos a los procesos en ejecución
- Todo proceso debería tener una reserva mínima de marcos (depende del repertorio de instrucciones)
- ¿Cómo asignar los marcos a los procesos?
  - Todos los procesos por igual
  - Reparto proporcional (por tamaño, por prioridad)
  - Reemplazo ¿global o local?

TEORÍA DEL CONJUNTO DE TRABAJO (WORKING SET)

ULPGC

Hiperpaginación (*thrashing*)

- Si el sistema tiene un grado de multiprogramación excesivo, puede ocurrir que se generen muchos fallos de página, porque ninguno de los procesos tiene «suficiente memoria para vivir tranquilo»:
- Un proceso tiene un fallo de página y se elige como víctima la página de otro proceso que a su vez la necesita. Esto puede causar una reacción en cadena que hace que todo el sistema esté paginando continuamente.
- Los sistemas primitivos se tropezaron con este problema y no se sabía muy bien cómo darle solución… ¿cuándo sabes que un proceso «no tiene suficiente memoria»?

Frecuencia de fallos de página (PFF)

- Técnica indirecta para resolver la hiperpaginación.
- Podemos establecer límites superiores e inferiores para la frecuencia de fallos de página deseada.
- Si la PFF de un proceso es muy baja, le quitamos páginas; si es muy alta, le damos más páginas.
- Si la PFF global aumenta y no hay marcos libres, seleccionamos un proceso y lo suspendemos.
  - Los marcos de página liberados se repartirán entre los procesos que tengan fallos de página muy frecuentes.

Teoría del conjunto de trabajo (Denning)

- Formulada por Peter Denning (1968-1970).
- Cada proceso trabaja en cada momento con unas zonas de código y datos bien delimitadas: **localidad.**
- La localidad va cambiando a medida que el proceso se ejecuta (ej. al llamar a un método).
- **Si un proceso tiene toda su localidad en memoria principal, no genera fallos de página.**
- **No hace falta darle al proceso más páginas de las que contienen su localidad actual.**

ULPGC

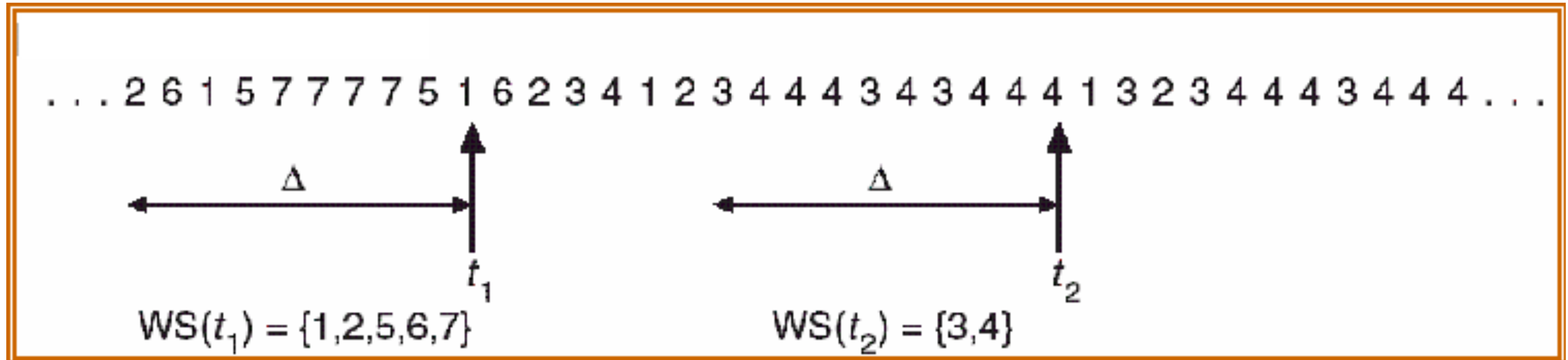
Conjunto de trabajo (*working set*)

- Aproximación a la localidad actual de un proceso.
- Es el conjunto de páginas con el que ha trabajado un proceso en un pasado reciente Δ:

$$WS(t, \Delta) = \text{páginas accedidas entre } t \text{ y } t\text{-}\Delta$$

- Si Δ tiene el tamaño adecuado (ni muy corto ni muy largo), es una fiel aproximación de la localidad actual.
- Otra forma de definirlo: **las páginas accedidas en los K últimos accesos a memoria.**

Conjunto de trabajo: ejemplo



Estimación del WS

- Usamos una ventana temporal Δ
- Cada página tiene una marca de tiempo de «último uso» = U
- Periódicamente revisamos los bits de referencia. A las páginas con R=1 se les pone U=reloj actual y se les vuelve a poner R=0.
- El WS serán las páginas del proceso que cumplan U >= reloj actual - Δ

ULPGC

Asignación de marcos basada en WS

- Cada proceso recibe un número de marcos igual al tamaño estimado de WS.
- Si la suma de los WS supera los marcos físicos disponibles, el SO envía procesos a disco (*swap out*) hasta que la suma está por debajo.
- Con eso se evita la hiperpaginación.
- Cuando se reanuda un proceso que estaba en *swap*, hay que restituirle todo su WS (si no se hace así, generará muchos fallos de página).

Algoritmo WSClock: reemplazo de páginas basado en el WS

- Parecido al algoritmo de 2ª oportunidad, pero con marcas de tiempo U.
- Si R=1, fijamos U=reloj actual, R=0 y seguimos buscando.
- Si R=0 y U>=reloj-Δ → seguimos buscando
- Si R=0 y U<reloj-Δ:
  - Modificada=1 → ordenamos guardarla en disco y seguimos buscando
  - Modificada=0 → es la víctima

OTRAS CONSIDERACIONES

Influencia de la estructura del código

Acceso por filas

```
double A[M][N];

for (i=0;i<M;i++)
  for (j=0;j<N;j++)
    A[i][j] = f(i,j);
```

Acceso por columnas

```
double A[M][N];

for (j=0;i<N;i++)
  for (i=0;j<M;j++)
    A[i][j] = f(i,j);
```

ULPGC

Influencia de la estructura del código

- Tamaño de las subrutinas/métodos
- Recursividad
- Uso de la pila (*stack*) vs uso de memoria dinámica

Limpieza anticipada de páginas (*precleaning*)

- El tiempo de copiar a disco una página *sucia* (modificada) puede generar un gran retardo en la gestión de un fallo de página.
- Solución → cuando el sistema está ocioso, ir *limpiando* (guardando en disco) las páginas modificadas (y poner a cero el bit de modificación).

Consideraciones actuales

- La localidad de los programas ha ido disminuyendo (OOP, recolector de basura, tablas hash…) → el concepto de «conjunto de trabajo» es más débil que hace 40 años.
- Hoy día la memoria virtual y la caché de disco se han unificado en un único sistema.
- Tamaño de página variable para adaptarse al WS.
- Los fallos de TLB y de página son más graves (hay más diferencias de velocidad entre niveles de la jerarquía de memorias).

Sistemas Operativos
Tema 4. Memoria

Fin del tema

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

© 2015 José Miguel Santos