



Sistemas Operativos

Gestión de memoria
Espacio de direcciones. Memoria virtual.



Agenda

- 1 Objetivos del sistema de gestión de memoria**
- 2 Modelo de memoria de un proceso**
- 3 Asignación contigua**
 - Intercambio (swapping)
- 4 Memoria virtual con Paginación Bajo Demanda**
 - Fundamentos de la memoria virtual
 - Políticas de reemplazo



Agenda

- 1 Objetivos del sistema de gestión de memoria**
- 2 Modelo de memoria de un proceso**
- 3 Asignación contigua**
 - Intercambio (swapping)
- 4 Memoria virtual con Paginación Bajo Demanda**
 - Fundamentos de la memoria virtual
 - Políticas de reemplazo



Objetivos del sistema de gestión de memoria

El SO multiplexa recursos entre procesos

- Cada proceso cree que tiene una máquina para él solo
- Gestión de procesos: Reparto de procesador
- Gestión de memoria: Reparto de memoria

Objetivos:

- Ofrecer a cada proceso un espacio lógico propio
- Dar soporte a las regiones del proceso
- Proporcionar protección entre procesos
- Permitir que procesos compartan memoria
- Maximizar el grado de multiprogramación
- Proporcionar a los procesos mapas de memoria muy grandes

Espacios lógicos independientes



- No se conoce la posición de memoria donde un programa se ejecutará
- Código ejecutable genera referencias entre 0 y N
- Ejemplo: Archivo ejecutable de un programa que copia un vector

Ejemplo

```
for (i=0; i<tam; i++)
Vdest[i]=Vorg[i];
```

- Cabecera de 100Bytes
- Instrucción ASM 4Bytes
- Vector origen a partir de dirección 1000
- Vector destino a partir de dirección 2000
- Tamaño del vector en dirección 1500

Fichero ejecutable

0	
4	
...	
96	Cabecera
100	LDR R1,#1000
104	LDR R2,#2000
108	LDR R3,#1500
112	LDR R4,[R3]
116	LDR R5,[R1],#1
120	STR R5,[R2],#1
124	SUBS R4,R4,#1
128	BNE 16
132



Reubicación

- Traducir direcciones lógicas a físicas
 - Dir. lógicas: direcciones de memoria generadas por el programa
 - Dir. físicas: direcciones de mem. principal asignadas al proceso
- Necesaria en SO con multiprogramación:
 - $\text{Traducción}(PID, \text{dir_lógica}) \rightarrow \text{dir_física}$
- Reubicación crea espacio lógico independiente para proceso
 - SO debe poder acceder a espacios lógicos de los procesos
- Dos alternativas de reubicación:
 - Software
 - Hardware



Reubicación software

- Traducción de direcciones durante carga del programa
- Programa en memoria distinto del ejecutable
- Desventajas:
 - No asegura protección
 - No permite mover programa en tiempo de ejecución

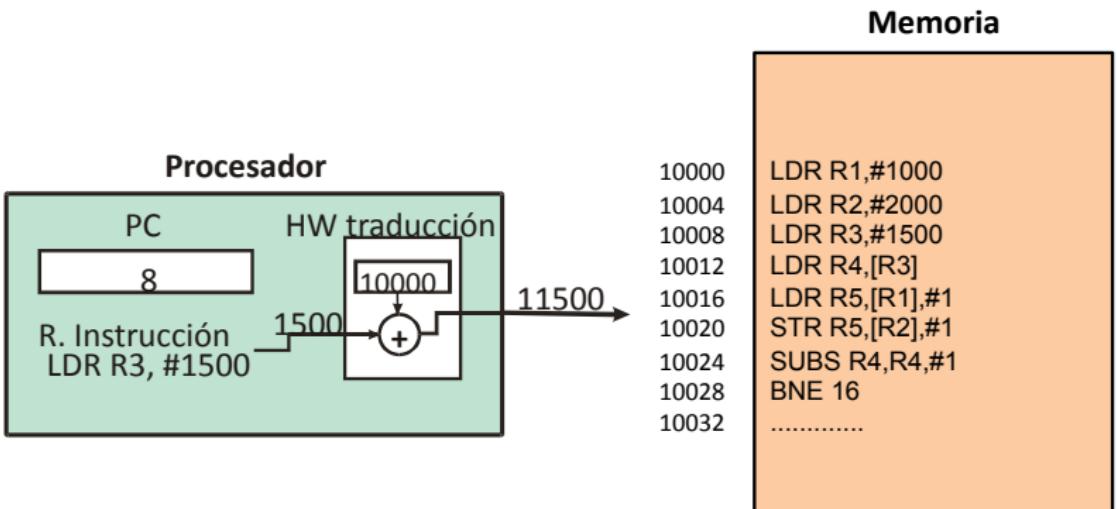
Memoria

10000	LDR R1,#11000
10004	LDR R2,#12000
10008	LDR R3,#11500
10012	LDR R4,[R3]
10016	LDR R5,[R1],#1
10020	STR R5,[R2],#1
10024	SUBS R4,R4,#1
10028	BNE 10016
10032



Reubicación hardware

- Hardware MMU encargado de hacer la traducción
- El SO se encarga de:
 - Almacenar por cada proceso una función de traducción
 - Especifica al hardware qué función aplicar para cada proceso
- El programa se carga en memoria sin modificar





Soporte de regiones

- Mapa de proceso no homogéneo
 - Conjunto de regiones con distintas características
 - Ejemplo: Región de código no modifiable
- Mapa de proceso dinámico
 - Regiones cambian de tamaño (p.ej. pila)
 - Se crean y destruyen regiones
 - Existen zonas sin asignar (huecos)
- Gestor de memoria debe dar soporte a estas características:
 - 1 Detectar accesos no permitidos a una región
 - 2 Detectar accesos a huecos
 - 3 Evitar reservar espacio para huecos
- SO debe guardar una tabla de regiones para cada proceso



Protección

- Monoprogramación: Proteger al SO
- Multiprogramación: Además procesos entre sí

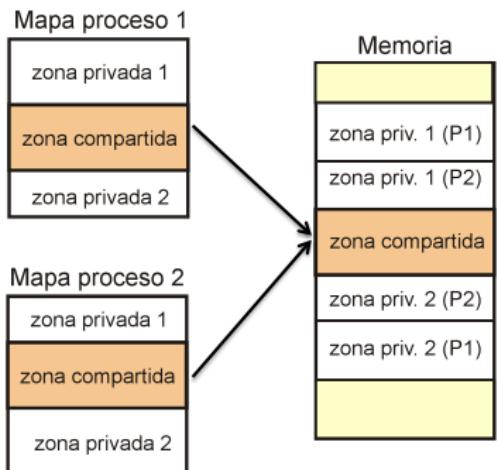
Requisitos mínimos:

- 1 La traducción debe crear espacios de direcciones disjuntos
- 2 Es necesario validar todas las direcciones que genera el programa
 - La detección se realiza por HW → generación de excepción
 - El tratamiento lo hace el SO



Compartición de memoria

- Direcciones lógicas de 2 o más procesos se corresponden con misma dirección física
- Bajo control del SO
- Beneficios:
 - Mecanismo de comunicación entre procesos muy rápido
 - Procesos ejecutando mismo programa comparten su código
- Requiere asignación no contigua





Utilización de la memoria

- Reparto de memoria maximizando grado de multiprogramación
- Se *desperdicia* memoria debido a:
 - Restos inutilizables (fragmentación)
 - Tablas requeridas por gestor de memoria
- Compromiso → Paginación
- Menor fragmentación implica tablas más grandes
- Uso de memoria virtual para aumentar grado de multiprogramación del SO

Aprovechamiento de memoria óptimo pero irrealizable

Memoria

0	Dirección 50 del proceso 4
1	Dirección 10 del proceso 6
2	Dirección 95 del proceso 7
3	Dirección 56 del proceso 8
4	Dirección 0 del proceso 12
5	Dirección 5 del proceso 20
6	Dirección 0 del proceso 1
.....	
.....	
N-1	Dirección 88 del proceso 9
N	Dirección 51 del proceso 4

Mapas de memoria muy grandes para procesos



- Procesos necesitan cada vez mapas más grandes
 - Aplicaciones más avanzadas o novedosas
- Resuelto gracias al uso de memoria virtual
- Antes se usaban *overlays*:
 - Programa dividido en fases que se ejecutan sucesivamente
 - En cada momento sólo hay una fase residente en memoria
 - Cada fase realiza su labor y carga la siguiente
 - No es transparente ya que toda la labor la realizaba el programador



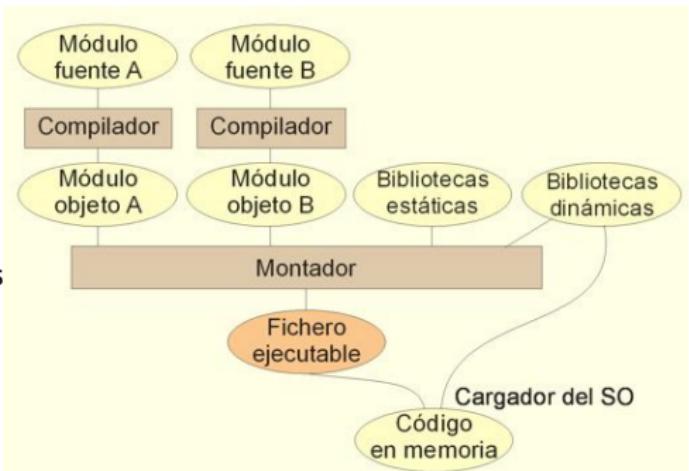
Agenda

- 1 Objetivos del sistema de gestión de memoria**
- 2 Modelo de memoria de un proceso**
- 3 Asignación contigua**
 - Intercambio (swapping)
- 4 Memoria virtual con Paginación Bajo Demanda**
 - Fundamentos de la memoria virtual
 - Políticas de reemplazo

Modelo de memoria de un proceso



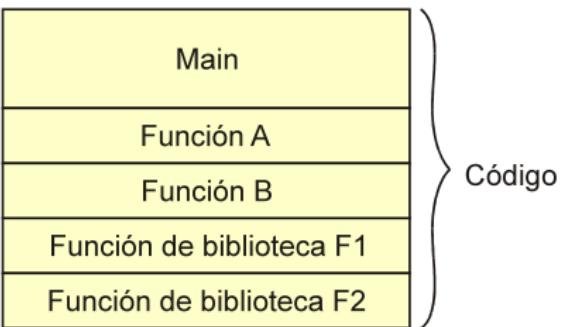
- Aplicación: conjunto de módulos en lenguaje de alto nivel
- Procesado en dos fases: Compilación y Montaje/Enlazado
- Compilación:
 - Resuelve referencias dentro cada módulo fuente
 - Genera módulo objeto
- Montaje (o enlazado):
 - Resuelve referencias entre módulos objeto
 - Resuelve referencias a símbolos de bibliotecas
 - Genera ejecutable incluyendo bibliotecas





Bibliotecas estáticas

- Colección de módulos objeto relacionados que exportan un conjunto de símbolos globales (funciones, variables, ...)
- Estáticas: el *linker* enlaza los módulos objeto del programa y de las bibliotecas creando un ejecutable autocontenido



Desventajas de las bibliotecas estáticas

- Ejecutables grandes
- Código de biblioteca repetido en muchos ejecutables y memoria
- Actualización de biblioteca implica volver a generar el ejecutable



Bibliotecas dinámicas

- Enlazado y carga en tiempo de ejecución
 - El ejecutable contiene:
 - 1 Nombre de la biblioteca
 - 2 Rutina de carga y montaje en tiempo de ejecución
 - La rutina de carga se invoca en la 1^a referencia a un símbolo de la bib.

Ventajas

- Menor tamaño ejecutables
 - Código de rutinas de biblioteca sólo en archivo de biblioteca
- Procesos pueden compartir código de biblioteca dinámica
- Actualización automática de bibliotecas: Uso de versiones

Desventajas

- Ejecutable no es autocontenido
- Mayor tiempo de ejecución debido a carga y montaje
 - Tolerable, compensado por las ventajas

Compartición de bibliotecas dinámicas



- Biblioteca dinámica contiene referencias internas
 - Problema de zona compartida con autoreferencias
- Código independiente de posición (PIC)
 - Solventa problema de compartición
 - Simplifica carga de múltiples bibliotecas en mapa de memoria

Memoria

```
100    LDR R4,[PC,#32]    ←  
104    BL [PC,#8]  
108    ADD R1,R1,#1  
112    BAL [PC,#-16]    ←  
116    → FUNCION  
120    .....  
124    .....  
128    .....  
132    .....  
136    → DATO
```

Memoria

```
120    LDR R4,[PC,#32]    ←  
124    BL [PC,#8]  
128    ADD R1,R1,#1  
132    BAL [PC,#-16]    ←  
136    → FUNCION  
140    .....  
144    .....  
148    .....  
152    .....  
156    → DATO
```



Formato del ejecutable

- En UNIX *Executable and Linkable Format* (ELF)

Estructura simplificada

1 Cabecera

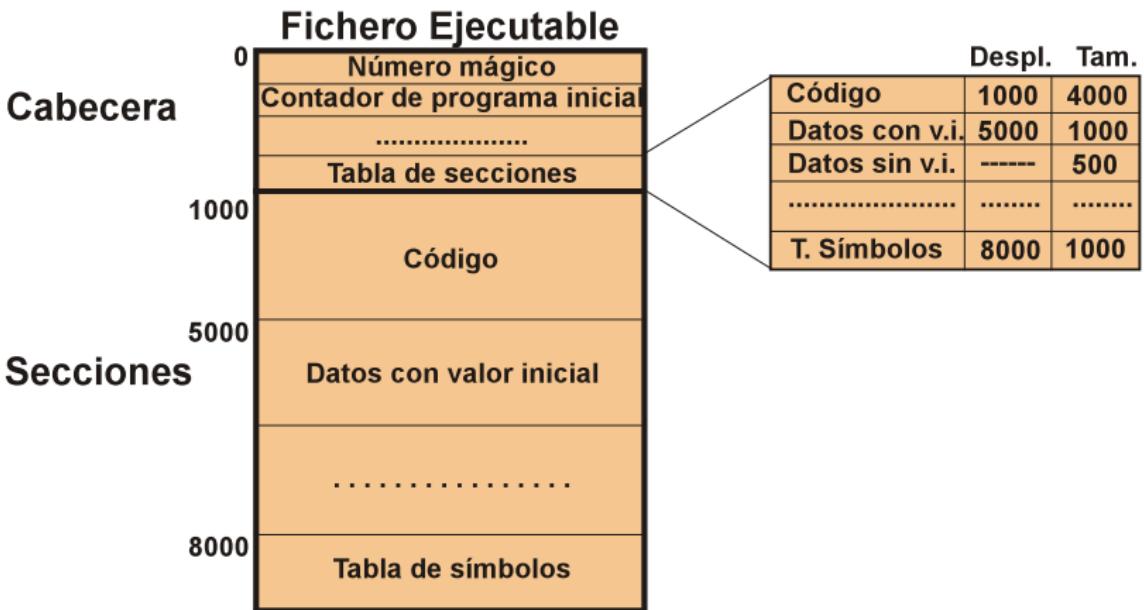
- Número mágico que identifica a ejecutable (0x7f+'ELF')
- Punto de entrada del programa
- Tabla de secciones

2 Secciones

- Código (text)
- Datos inicializados
- Datos no inicializados (sólo se especifica el tamaño)
- Tabla de símbolos de depuración
- Tabla de bibliotecas dinámicas



Formato del ejecutable





Variables globales vs. locales

Variables globales (con sección)

- Estáticas
- Se crean al iniciarse programa
- Existen durante ejecución del mismo
- Dirección fija en memoria y en ejecutable

Variables locales y parámetros (sin sección)

- Dinámicas
- Se crean al invocar función
- Se destruyen al retornar
- La dirección se calcula en tiempo de ejecución
- Recursividad: varias instancias de una variable

Variables globales vs. locales



Ejemplo

```
int x=8;          /* Variable global con valor inicial */
int y;            /* Variable global sin valor inicial */
int j=5;          /* Inicialización y reserva de espacio
                    para var. global */

int f(int t) {    /* Parámetro */
    int z;          /* Variable local */
    ...
}

int main() {
    ...
    f(4);
    ...
}
```

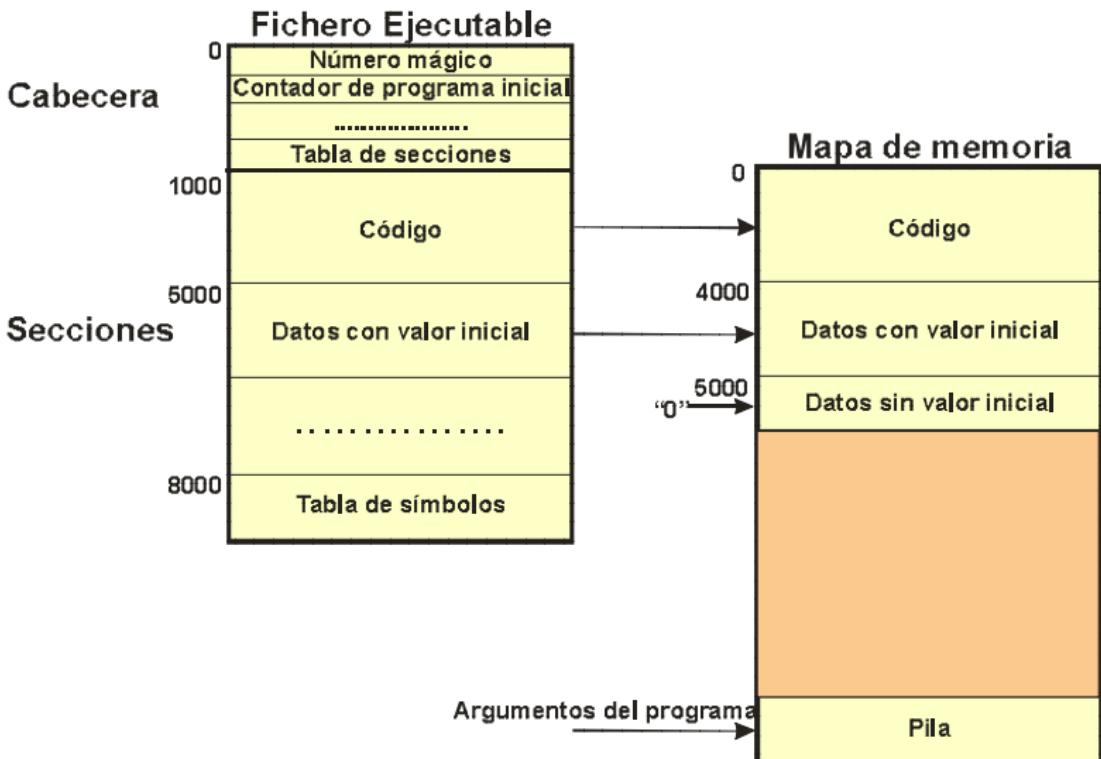
Mapa de memoria de un proceso



- EL MMP es un **conjunto de regiones** de memoria donde se almacena todo lo necesario para que un programa pueda ejecutarse
- Cada región es una zona contigua tratada como unidad
- Cada región posee datos + metainformación (que mantiene el SO)
 - Dirección de comienzo y tamaño inicial
 - Soporte: dónde se almacena su contenido inicial si lo tuviese (ej. fichero ejecutable)
 - Protección: RWX
 - Uso compartido o privado
 - Tamaño fijo o variable (modo de crecimiento ↑↓)



Crear mapa de memoria desde ejecutable



Otras regiones del mapa de memoria



- Durante la ejecución del proceso se crean nuevas regiones
 - Mapa de memoria tiene un carácter dinámico

Otras regiones

- **Región de Heap**
 - Soporte de memoria dinámica (`malloc()` en C)
 - Privada, RW, T. Variable, Sin Soporte (rellenar 0's)
 - Crece hacia direcciones más altas
- **Regiones de bibliotecas dinámicas**
 - Se crean regiones asociadas al código y datos de la biblioteca
- **Pilas de threads**
 - Cada pila de *thread* corresponde con una región
 - Mismas características que pila del proceso

Otras regiones del mapa de memoria



Otras regiones (cont.)

■ Memoria compartida

- Región asociada a la zona de memoria compartida
- Compartida, T. Variable, Sin Soporte (rellenar 0's)
- Protección especificada en proyección

■ Fichero proyectado en memoria (`mmap()`)

- Región asociada al archivo proyectado
- T. Variable, Soporte en fichero
- Protección y carácter compartido o privado especificado en proyección



Características de las regiones

Mapa de memoria

Código
Datos con valor inicial
Datos sin valor inicial
Heap
Fichero proyectado F
Zona de memoria compartida
Código biblioteca dinámica B
Datos biblioteca dinámica B
Pila de thread 1
Pila del proceso

Región	Soporte	Protección	Comp/Priv	Tamaño
Código	Fichero	RX	Compartida	Fijo
Dat. con v.i.	Fichero	RW	Privada	Fijo
Dat. sin v.i.	Sin soporte	RW	Privada	Fijo
Pilas	Sin soporte	RW	Privada	Variable
Heap	Sin soporte	RW	Privada	Variable
F. Proyect.	Fichero	por usuario	Comp./Priv.	Variable
M. Comp.	Sin soporte	por usuario	Compartida	Variable



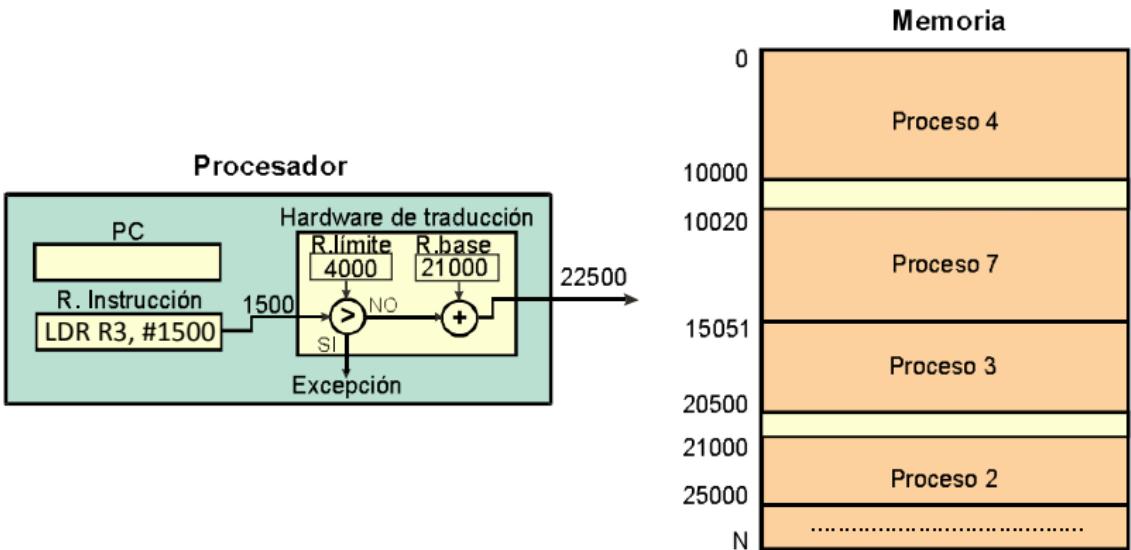
Agenda

- 1 Objetivos del sistema de gestión de memoria**
- 2 Modelo de memoria de un proceso**
- 3 Asignación contigua**
 - Intercambio (swapping)
- 4 Memoria virtual con Paginación Bajo Demanda**
 - Fundamentos de la memoria virtual
 - Políticas de reemplazo



Asignación contigua (I)

- Mapa de memoria de cada proceso en zona contigua de memoria principal
- Hardware requerido: Regs. valla (R. base y R. límite)
 - Sólo accesibles en modo privilegiado.





Asignación contigua (II)

SO mantiene información sobre:

- 1** El valor de regs. valla de cada proceso en su BCP
 - En cambio de contexto SO carga en regs. valor adecuado
- 2** Estado de ocupación de la memoria
 - Estructuras de datos que identifiquen huecos y zonas asignadas
 - Regiones de cada proceso

Este esquema presenta fragmentación externa

- Se generan pequeños fragmentos libres entre zonas asignadas
- Posible solución: compactación → proceso costoso



Operaciones sobre regiones con a. contigua

- Al crear un proceso se le asigna una zona de memoria de tamaño fijo
 - Suficiente para albergar regiones iniciales
 - Con huecos para permitir cambios de tamaño y añadir nuevas regiones (p.ej. bibliotecas dinámicas o pilas de threads)
 - Difícil asignación adecuada
 - Si es grande se desperdicia espacio, si es pequeña se puede agotar
 - Algoritmos: primer ajuste, mejor ajuste, peor ajuste y siguiente ajuste
- Operaciones:
 - Crear/liberar/cambiar tamaño usan la tabla de regiones para gestionar la zona asignada al proceso
 - Duplicar región requiere crear región nueva y copiar contenido
 - Limitaciones del hardware impiden compartir memoria



Intercambio

- *¿Qué hacer si no caben todos los programas en mem. principal?* → swapping
- Usado en primeras versiones de UNIX
- Zona de Intercambio o *Swap*: partición (fichero) de disco que almacena imágenes de procesos



Intercambio: Swap out

Swap out

- Cuando no caben en memoria todos los procesos activos, se expulsa un proceso de memoria copiando su imagen a *swap*
- Diversos criterios de selección del proceso a expulsar
 - P.ej. Dependiendo de la prioridad del proceso
 - Preferiblemente un proceso bloqueado
 - No expulsar un proceso si tiene activa una operación de DMA
- No es necesario copiar todo el mapa:
 - El código está en el fichero
 - Los huecos no contienen información del proceso



Intercambio: Swap in

Swap in

- Cuando haya espacio en memoria principal, se lee el proceso a memoria copiando la imagen desde *swap*
- También cuando un proceso lleva un cierto tiempo expulsado
 - En este caso antes de *swap in*, hay *swap out* de otro

Intercambio (II)



- Asignación de espacio en el dispositivo de swap
 - Con preasignación: se asigna espacio al crear el proceso
 - Sin preasignación: se asigna espacio al expulsarlo
- Los procesos activos han de residir completamente en MP
 - Grado de multiprogramación depende del tamaño de proc. y MP
- Solución general → Uso de esquemas de memoria virtual
 - Aunque se sigue usando integrado con esa técnica



Agenda

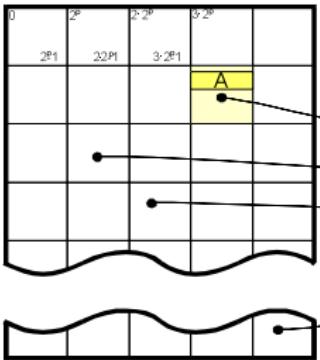
- 1 Objetivos del sistema de gestión de memoria**
- 2 Modelo de memoria de un proceso**
- 3 Asignación contigua**
 - Intercambio (swapping)
- 4 Memoria virtual con Paginación Bajo Demanda**
 - Fundamentos de la memoria virtual
 - Políticas de reemplazo

Fundamentos de la memoria virtual (I)

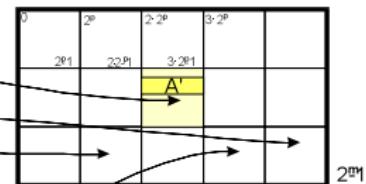


- Los procesos (y el SO) generan direcciones virtuales al ejecutar
 - Las dir. virtuales pertenecen al *mapa de memoria virtual*
- El mapa de memoria virtual de un proceso se divide en *páginas*
 - Página: unidad mínima de asignación de MV (Ej. 4KB)
- Parte del mapa de memoria virtual de un proceso está en MP (memoria principal) y parte en disco (*swap* o memoria secundaria)
- El SO se encarga de que estén en memoria principal las páginas *necesarias* del mapa de memoria virtual de cada proceso

MAPA VIRTUAL
(RESIDENTE EN DISCO)



MEMORIA PRINCIPAL



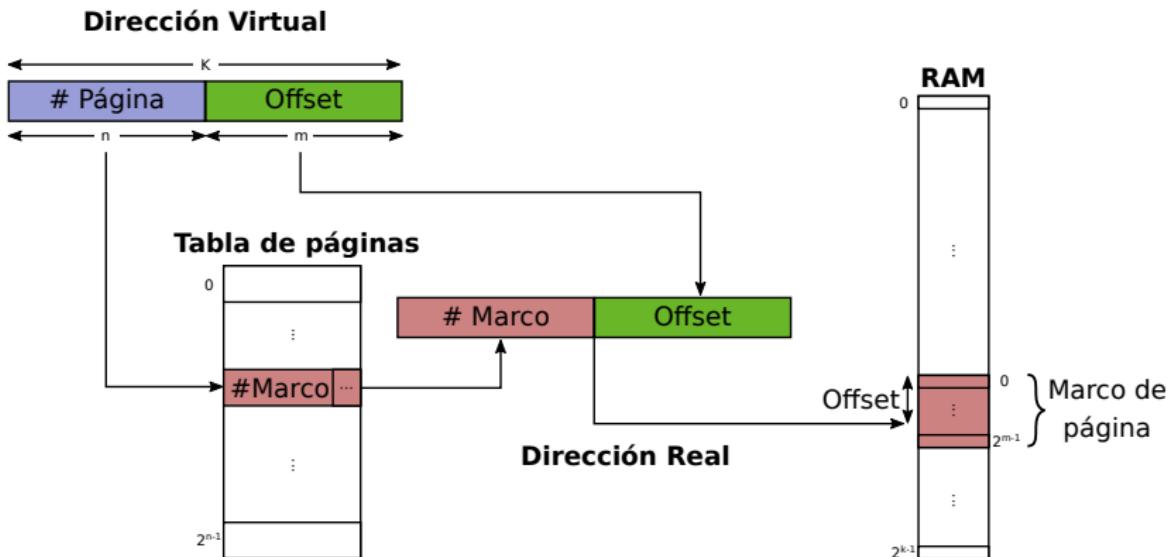
Proyección de página
virtual a memoria física

Fundamentos de la memoria virtual (III)



- Transferencia de páginas entre ambos niveles (*swap* y MP)
 - De M. secundaria a principal: bajo demanda
 - De M. principal a secundaria: por expulsión
- La memoria virtual es eficaz porque los procesos exhiben localidad de referencias
 - Localidad espacial (p.ej., página perteneciente a la región de código.)
 - Localidad temporal (p.ej., múltiples referencias a una misma variable)
 - En la práctica, los procesos sólo usan parte de su mapa de memoria en un intervalo de tiempo
 - Para maximizar el rendimiento, el SO intenta que la parte de la memoria de un proceso que esté siendo utilizada (*conjunto de trabajo*) resida en MP (*conjunto residente*)
- Beneficios:
 - Aumenta el grado de multiprogramación
 - Permite la ejecución de programas que no quepan en MP

Traducción con tabla de páginas





Contenido de entrada de TP

- Número de marco asociado
- Bit de página válida/inválida
 - Indica si la página pertenece a una región del proceso y por tanto es válido referenciarla
 - Si es 0 → *Segmentation Fault*
- Bit de Presencia
 - Si la página está cargada en RAM o no
 - Si página no presente → Excepción de fallo de página
- Información de protección: RWX
 - Si operación no permitida → Excepción
- Información de nivel de privilegio: Usuario/Kernel
- Bit de página accedida (*Ref*)
 - MMU lo activa cuando se accede a esta página desde que la página se trajo a MP (lo puede borrar el algoritmo de reemplazo)
- Bit de página modificada (*Mod*)
 - MMU lo activa cuando se escribe en esta página
- Bit de desactivación de cache



Tamaño de página

Condicionado por diversos factores contrapuestos:

- Potencia de 2 y múltiplo del tamaño del bloque de E/S
- Mejor pequeño por:
 - Menor fragmentación interna
 - Se ajusta mejor al conjunto de trabajo
- Mejor grande por:
 - Tablas más pequeñas
 - Mejor rendimiento en las transferencias MP disp. E/S (disco)
- Compromiso: entre 2KB y 16KB
 - En SSOO tipo UNIX actuales consultar con: `getconf PAGESIZE`



Fragmentación interna en paginación

- Tendremos *fragmentación interna* si Mem. asignada >Mem. requerida
 - Puede desperdiciarse parte de un marco asignado

T. Páginas Pr. 1

Página 0	Marco 2
Página 1	Marco N

Página M	Marco 3

T. Páginas Pr. 2

Página 0	Marco 4
Página 1	Marco 0

Página P	Marco 1

Memoria

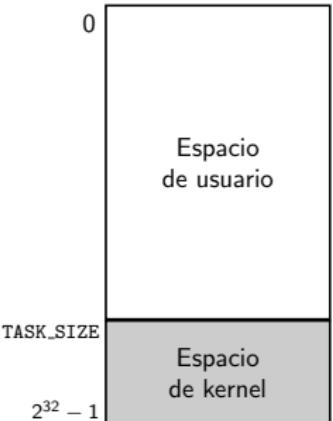
Pág. 1 Pr. 2	Marco 0
Pág. P Pr. 2	Marco 1
Pág. 0 Pr. 1	Marco 2
Pág. M Pr. 1	Marco 3
Pág. 0 Pr. 2	Marco 4

Pág. 1 Pr. 1	Marco N

Páginas y mapa de memoria



- El SO gestiona dos tipos de página:
 - Página de usuario: almacena código o datos de un proceso
 - Página de sistema: almacena código o datos (variables y estructuras de datos) del SO
- Por simplicidad, el SO divide el mapa de memoria (virtual) del proceso en *espacio de kernel* (páginas de sistema) y *espacio de usuario* (páginas de usuario)
- Cuando el proceso ejecuta instrucciones en modo usuario solo puede generar direcciones (referenciar páginas) del espacio de usuario
 - La MMU impide que el proceso acceda a direcciones del esp. de kernel
- Cuando el proceso invoca una llamada al sistema (paso a modo kernel) el SO accede a ambos espacios





Páginas y mapa de memoria

Características de esta organización

- Cada proceso tiene su propia tabla de páginas (TP) pero . . .
 - Parte del espacio de kernel similar para todos los procesos
 - Hilos de un mismo proceso comparten TP
 - Tabla de páginas común para proceso de usuario en ejecución y kernel
 - El proceso de usuario no puede acceder a páginas del kernel
- Simplifica implementación de llamadas al sistema y gestión de excepciones
 - No exige cambiar tabla de páginas al entrar y salir de modo kernel
 - Fácil traducción de direcciones de parámetros tipo puntero en *syscalls*



Valoración de la paginación

¿Proporciona las funciones deseables en un gestor de memoria?

- Espacios independientes para procesos:
 - Mediante TP
- Protección:
 - Mediante TP
- Compartir memoria:
 - Entradas de las TPs de dos o más procesos corresponden con mismo marco (Varias dir. virtuales → 1 única dir. física)
- Soporte de regiones:
 - Bits de protección
 - Bit de validez: no se reserva espacio para huecos
- Maximizar utilización de la memoria y soporte de mapas grandes
 - permite mapas de memoria virtual >> cantidad de memoria física
- *Problema:* Mucho mayor gasto en tablas del SO que con asignación contigua
 - Es el precio de mucha mayor funcionalidad



Problemática de las TPs

Eficiencia:

- Cada acceso lógico requiere dos accesos a memoria principal (uno detrás del otro):
 - A la tabla de páginas + al propio dato o instrucción
- Solución: Cache de traducciones (TLB)

Gasto de almacenamiento:

- Tablas muy grandes
 - Ejemplo: páginas 4KB, dir. virtuales de 32 bits y 4 bytes por entrada
 - Tamaño TP: $2^{20} * 4 = 4\text{MB/proceso}$
- Solución:
 - Tablas multinivel
 - Tablas invertidas

Translation Look-aside Buffer (TLB)



- Memoria asociativa con info. sobre últimas páginas accedidas
 - cache de entradas de TP correspondientes a estos accesos

2 alternativas de diseño:

- 1** Entradas en TLB no incluyen información sobre proceso
 - Exige invalidar TLB en cambios de contexto
- 2** Entradas en TLB incluyen información sobre proceso
 - Registro de CPU debe mantener un ident. de proceso actual



Tratamiento del fallo de página

Tratamiento de excepción

- El HW almacena la dirección de fallo en un registro
- Si dirección inválida → Aborta proceso o le manda señal
- Consulta T. marcos, si no hay ningún marco libre
 - Selección de víctima: pág P marco M
 - Marca P como no presente
 - Si P modificada (bit Mod activo)
 - Inicia escritura P en mem. secundaria
- Hay marco libre (se ha liberado o lo había previamente):
 - Inicia lectura de página en marco M
 - Marca entrada de página presente referenciando a M
 - Pone M como ocupado en T. marcos (si no lo estaba)
- Fallo de página puede implicar 2 accesos a disco



Política de reemplazo:

- ¿Qué página reemplazar si fallo y no hay marco libre?
- Reemplazo local
 - Sólo puede usarse para reemplazo un marco asignado al proceso que causa fallo
- Reemplazo global
 - Puede usarse para reemplazo cualquier marco

Política de asignación de espacio a los procesos:

- ¿Cómo se reparten los marcos entre los procesos?
 - Asignación fija o dinámica



Algoritmos de reemplazo

- Objetivo: Minimizar la tasa de fallos de página
- Cada algoritmo descrito tiene versión local y global:
 - local: criterio se aplica a las páginas residentes del proceso
 - global: criterio se aplica a todas las páginas residentes
- Algoritmos presentados
 - Óptimo
 - FIFO
 - Reloj (o segunda oportunidad)
 - LRU
- Uso de técnicas de *buffering* de páginas



Algoritmo óptimo

- Criterio: se conoce la página residente que tardará más ser accedida
→ Irrealizable
- Interés para estudios analíticos comparativos
- Ejemplo:
 - Memoria física de 4 marcos de página
 - Referencias *a b g a d e a b a d e g d e*

Ref	a	b	g	a	d	e	a	b	a	d	e	g	d	e
0	a	a	a	a	a	a	a	a	a	a	a	g	g	g
1	b	b	b	b	b	b	b	b	b	b	b	b	b	b
2	g	g	g	e	e	e	e	e	e	e	e	e	e	e
3			d	d	d	d	d	d	d	d	d	d	d	d

6 fallos



Algoritmo FIFO

- Criterio: página que lleva más tiempo residente
- Implementación sencilla:
 - páginas residentes en orden FIFO → se expulsa la primera
 - no requiere el bit de página accedida (Ref)
- No es una buena estrategia:
 - Página que lleva mucho tiempo residente puede seguir accediéndose frecuentemente
 - Su criterio no se basa en el uso de la página
- Ejemplo:
 - Memoria física de 4 marcos de página
 - Referencias $a\ b\ g\ a\ d\ e\ a\ b\ a\ d\ e\ g\ d\ e$

Ref	a	b	g	a	d	e	a	b	a	d	e	g	d	e
0	a_1	a	a	a_4	a_5	e_6	e	e	e	e	e_9	e_{10}	d_{13}	d
1		b_2	b	b	b_5	b_6	a_7	a	a_9	a	a	a	a_{12}	e_{14}
2			g_3	g	g	g	g_7	b_8	b	b	b	b	b	b
3				d_5	d	d	d	d	d_{10}	d_{11}	g_{12}	g	g	

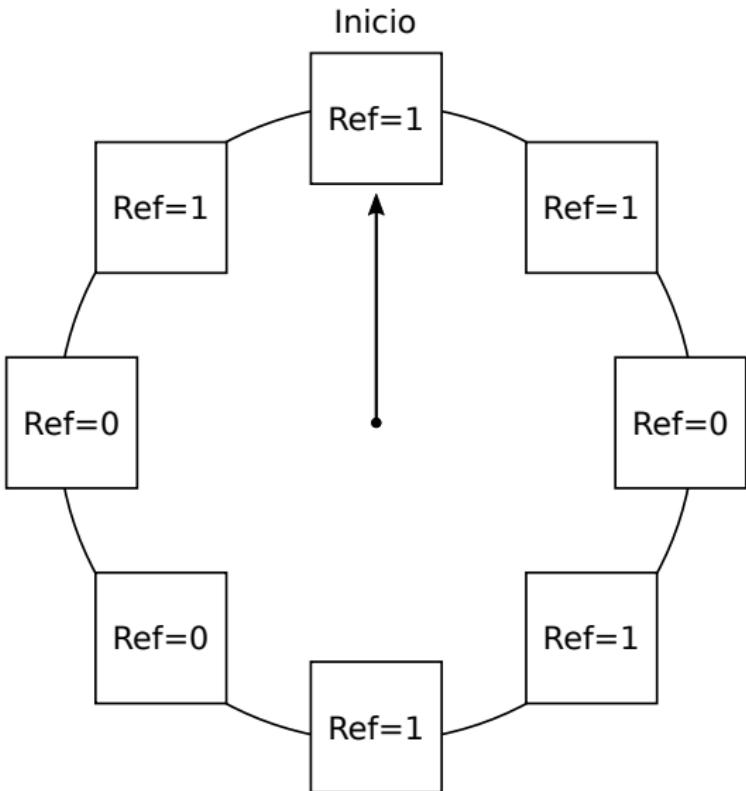
10 fallos

Algoritmo del reloj (o 2^a oportunidad)

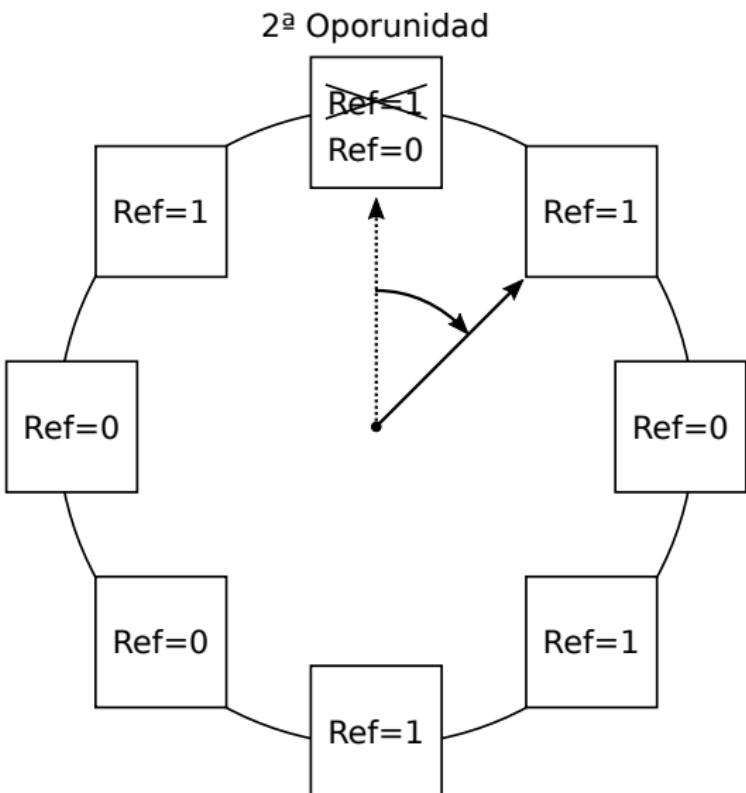


- FIFO + uso de bit de referencia Ref (de página accedida)
- Criterio:
 - Si página elegida por FIFO no tiene activo Ref
 - Es la página expulsada
 - Si lo tiene activo (2^a oportunidad)
 - Se desactiva Ref
 - Se pone página al final de FIFO
 - Se aplica criterio a la siguiente página
- Se puede implementar orden FIFO como lista circular con una referencia a la primera página de la lista:
 - Se visualiza como un reloj donde la referencia a la primera página es la aguja del reloj

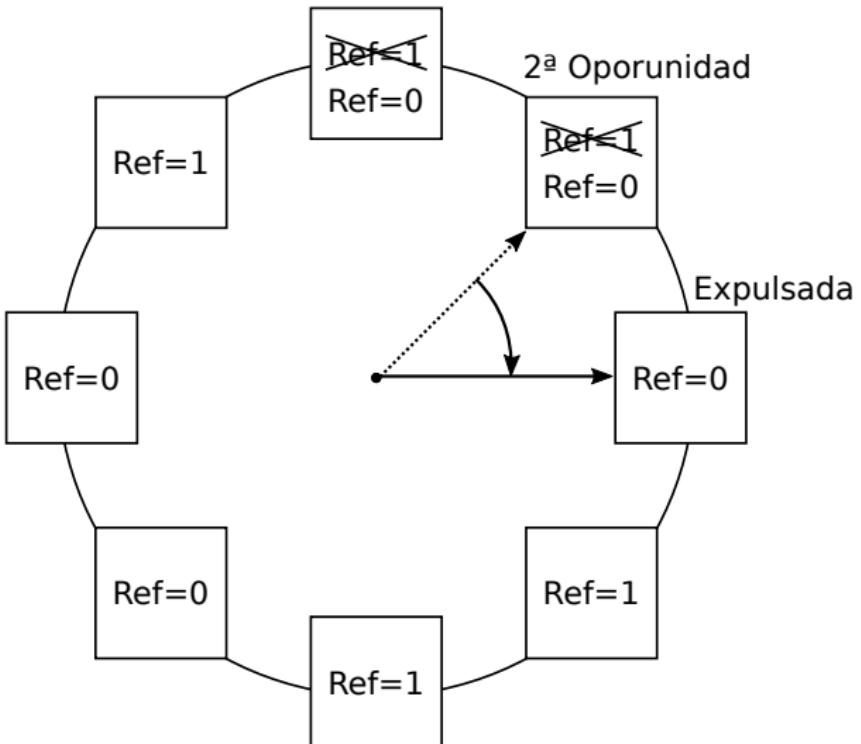
Algoritmo del reloj (o 2^a oportunidad)



Algoritmo del reloj (o 2^a oportunidad)



Algoritmo del reloj (o 2^a oportunidad)



Algoritmo del reloj (o 2^a oportunidad)



■ Ejemplo:

- Memoria física de 4 marcos de página
- Referencias *a b g a d e a b a d e g d e*

Ref	a	b	g	a	d	e	a	b	a	d	e	g	d	e
0	$\downarrow a_0$	$\downarrow a_0$	$\downarrow a_0$	$\downarrow a_1$	$\downarrow a_1$	a_0	a_0	a_1	a_1	a_1	a_1	a_0	a_0	a_0
1	b_0	b_0	b_0	b_0	b_0	$\downarrow b_0$	e_0	e_0	e_0	e_0	e_0	e_1	e_0	e_0
2		g_0	g_0	g_0	g_0	g_0	$\downarrow g_0$	$\downarrow g_0$	b_0	b_0	b_0	g_0	g_0	g_0
3				d_0	d_0	d_0	d_0	$\downarrow d_0$	$\downarrow d_0$	$\downarrow d_0$	$\downarrow d_1$	$\downarrow d_1$	$\downarrow d_0$	$\downarrow d_1$

7 fallos

Algoritmo LRU



- Criterio: página residente menos recientemente usada
- Por proximidad de referencias:
 - Pasado reciente condiciona futuro próximo
- Sutileza:
 - En su versión global: menos recientemente usada en el tiempo lógico de cada proceso
- Difícil implementación estricta (hay aproximaciones):
 - Precisaría una MMU específica
- Posible implementación con HW específico:
 - La MMU mantiene un contador que se incrementa cada cierto tiempo
 - En entrada de TP hay una marca de tiempo (*timestamp*)
 - En cada acceso a memoria MMU copia contador del sistema a entrada referenciada
 - Reemplazo: Página con *timestamp* más bajo



Algoritmo LRU

■ Ejemplo:

- Memoria física de 4 marcos de página
- Referencias $a\ b\ g\ a\ d\ e\ a\ b\ a\ d\ e\ g\ d\ e$

Ref	a	b	g	a	d	e	a	b	a	d	e	g	d	e
0	a_1	a	a	a_4	a	a	a_7	a	a_9	a	a	a	a	a
1		b_2	b	b	b_5	e_6	e	e	e	e	e_{11}	e	e	e_{14}
2			g_3	g	g	g	g_8	b	b	b	b_{12}	g	g	g
3					d_5	d	d	d	d_{10}	d	d	d_{13}	d	

7 fallos



Buffering de páginas

- Caso peor en tratamiento de fallo de página:
 - 2 accesos a disco
- Alternativa: mantener una reserva de marcos libres
 - Fallo de página: siempre usa marco libre (no reemplazo)
- Si número de marcos libres < umbral
 - *Demonio de paginación* aplica repetidamente el algoritmo de reemplazo:
 - páginas no modificadas pasan a lista de marcos libres
 - páginas modificadas pasan a lista de marcos modificados
 - cuando se escriban a disco pasan a lista de libres
 - pueden escribirse en tandas (mejor rendimiento)
- Si se referencia una página mientras está en estas listas
 - fallo de página la recupera directamente de la lista (no E/S)
 - puede arreglar el comportamiento de algoritmos “malos”



Retención de páginas en memoria

- Páginas marcadas como no reemplazables
- Se aplica a páginas del propio SO
 - SO con páginas fijas en memoria es más sencillo
- También se aplica mientras se hace DMA sobre una página
- Algunos sistemas ofrecen a aplicaciones un servicio para fijar en memoria una o más páginas de su mapa
 - Adecuado para procesos de tiempo real
 - Puede afectar al rendimiento del sistema
 - En POSIX servicio `mlock()`

Estrategia de asignación fija



- Número de marcos asignados al proceso (conjunto residente) es constante
- Puede depender de características del proceso:
 - tamaño, prioridad,...
- No se adapta a las distintas fases del programa
- Comportamiento relativamente predecible
- Sólo tiene sentido usar reemplazo local
- Arquitectura impone nº mínimo: al menos una página de código, una de datos, y una de pila

Estrategia de asignación dinámica



- Número de marcos varía dependiendo de comportamiento del proceso (y posiblemente de los otros procesos)
- Asignación dinámica + reemplazo local
 - Proceso va aumentando o disminuyendo su conjunto residente dependiendo de su comportamiento
 - Comportamiento relativamente predecible
- Asignación dinámica + reemplazo global
 - Procesos se quitan las páginas entre ellos
 - Comportamiento difícilmente predecible



Hiperpaginación (Thrashing)

- Tasa excesiva de fallos de página de un proceso o en el sistema
- Con asignación fija: Hiperpaginación en P_i
 - Si conjunto residente de $P_i <$ conjunto de trabajo P_i
- Con asignación variable: Hiperpaginación en el sistema
 - Si nº marcos disponibles $< \Sigma$ conjuntos de trabajo de todos
 - Grado de utilización de CPU cae drásticamente
 - Procesos están casi siempre en colas de dispositivo de paginación
 - Solución (similar al *swapping*): Control de carga
 - Disminuir el grado de multiprogramación
 - Suspender 1 o más procesos liberando sus páginas residentes
 - Problema: ¿Cómo detectar esta situación?

Estrategia del conjunto de trabajo

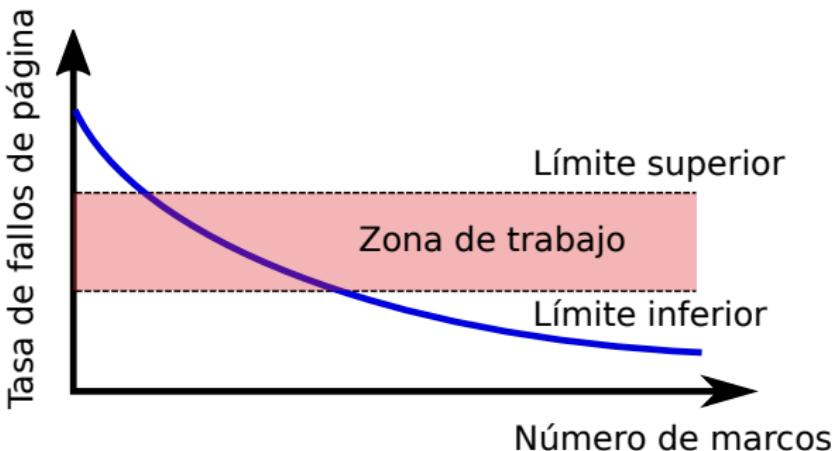


- Intentar conocer el conjunto de trabajo de cada proceso
 - Páginas usadas por el proceso en las últimas N referencias
- Si conjunto de trabajo decrece se liberan marcos
- Si conjunto de trabajo crece se asignan nuevos marcos
 - Si no hay disponibles: suspender proceso(s)
 - Se reactivan cuando hay marcos suficientes para c. de trabajo
- Asignación dinámica con reemplazo local
- Difícil implementación estricta
 - Precisaría una MMU específica
- Se pueden implementar aproximaciones:
 - Estrategia basada en frecuencia de fallos de página (PFF):
 - Controlar tasa de fallos de página de cada proceso



Estrategia basada en frecuencia de fallos

- Si tasa < límite inferior se liberan marcos aplicando un algoritmo de reemplazo
- Si tasa > límite superior se asignan nuevos marcos
 - Si no marcos libres se suspende algún proceso



Control de carga y reemplazo global



- Algoritmos de reemplazo global no controlan hiperpaginación
 - ¡Incluso el óptimo!
 - Necesitan cooperar con un algoritmo de control de carga
- Ejemplo: UNIX 4.3 BSD
 - Reemplazo global con algoritmo del reloj
 - Variante con dos “manecillas”
 - Uso de *buffering* de páginas
 - “demonio de paginación” controla nº de marcos libres
 - Si número de marcos libres < umbral
 - “demonio de paginación” aplica reemplazo
 - Si se repite con frecuencia la falta de marcos libres:
 - Proceso “swapper” suspende procesos