

Administración de memoria: Asignación de memoria

Gunnar Wolf



Índice

- 1 Memoria contigua
- 2 Segmentación
- 3 Paginación



Compartiendo la memoria desde...

- Como en tantos otros temas, comencemos viendo cómo compartían la memoria los *primeros* sistemas multiprocesados
- Las primeras implementaciones siempre son las más sencillas
 - ... y las más ingenuas
- ¿Cómo eran estos primeros sistemas?
 - Poca memoria
 - Sin MMU
 - No interactivos



Particiones fijas

- Primer acercamiento: *Partir* la memoria en varios bloques
 - Originalmente del mismo tamaño (¡más sencillo!)
 - Por ejemplo: En 512KB de memoria física caben el sistema operativo mas otros 7 programas de 64KB (16 bits) cada uno
- El sistema operativo típicamente usa la *región más baja*, a partir de 0x0000
 - La *memoria mapeada* a los diversos dispositivos queda dentro del segmento del SO



Particiones fijas

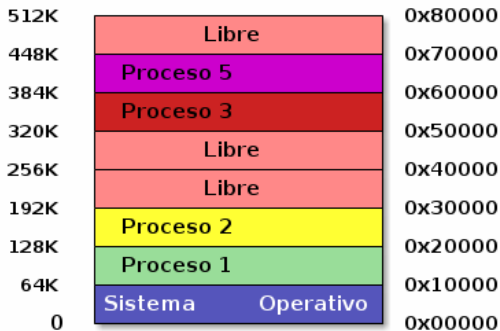


Figura: Particiones fijas, con 3 particiones libres



Particiones fijas: Ventajas y desventajas

- ¿Ventajas? Principalmente, simplicidad
 - Resolución de direcciones en tiempo de carga
 - *Registro base* (no requiere siquiera de un *registro límite*)
 - Puede limitarse simplemente con un espacio de direccionamiento acorde en el compilador
- ¿Desventajas? Rigidez
 - *Grado de multiprocesamiento* limitado
 - Si hay menos de 7 procesos, se desperdician recursos
 - Si hay más de 7, tienen que esperar a que se les abra espacio
 - Desperdicio de espacio (*fragmentación interna*)
 - Al asignarse la memoria en bloques fijos, un proceso pequeño podría desperdiciar mucho espacio



Particiones flexibles

- Cada proceso declara sus requisitos de memoria *al iniciar su ejecución*
 - Debe indicar su uso *máximo previsto* de memoria
 - Hay mecanismos para *ajustar el tamaño* de un proceso preexistente — ¡Pero pueden fallar! (p.ej. si falta memoria para satisfacer una solicitud)
- El OS tiene acceso directo a toda la memoria *como un continuo*
- Cada región en memoria está limitada (ahora sí) por un registro base y un registro límite



Particiones flexibles

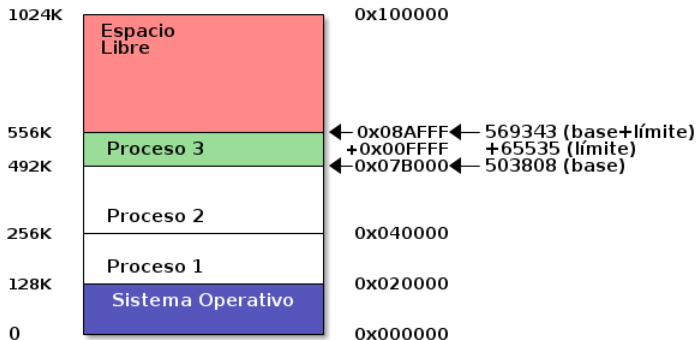


Figura: Espacio de direcciones válidas para el proceso 3 definido por un registro base y un registro límite



Particiones flexibles: Ventajas y desventajas

- ¿Ventajas? Sigue siendo: Simplicidad
 - Cuando inicia la ejecución del sistema, este esquema parece ideal
 - Sobrecarga mínima, con un MMU muy básico
 - Cada proceso puede direccionar el total de memoria disponible
- ¿Desventajas? Vienen con el tiempo. . .
 - Conforme van iniciando y terminando los procesos, se van creando *agujeros* en la asignación de memoria
 - Según análisis estadístico (Silberschatz, p.289), por cada N bloques asignados se pierden del orden de $0,5N$ por fragmentación



Fragmentación en la memoria



Figura: Termina el proceso 1 (de 128K); inician 4 (de 64K), 5 (de 156K) y 6 (de 128K); termina 3 (64K). Se va fragmentando la memoria libre.



Fragmentación interna y externa

Fragmentación interna Espacio desperdiciado *dentro* de la memoria asignada a un proceso

- Porque tuvo que solicitar *toda la memoria que emplearía* desde un principio (y la desperdicia la mayor parte del tiempo)
- Por tener que *alinearse* a cierta frontera de memoria (p.ej. con particiones pre-establecidas)

Fragmentación externa Espacio de memoria desperdiciado *entre los distintos fragmentos*

- En el esquema anterior, hay 320K libres, pero no puede lanzarse ningún proceso $> 192K$, porque no es un bloque *contiguo*



¿Cómo ubicar un nuevo proceso?

Hay tres estrategias principales para dar espacio en la memoria a un nuevo proceso:

Primer ajuste Asigna al nuevo proceso al *primer bloque* de tamaño suficiente

Mejor ajuste Asigna al nuevo proceso al *bloque más chico* en que quepa

Peor ajuste Asigna al nuevo proceso al *bloque más grande* que haya disponible

¿Qué ventajas / desventajas? puede tener cada uno?



Primer ajuste

- El mecanismo más fácil de implementar
- Ejecución más rápida
- Pero no considera facilitar las cosas para el futuro...



Mejor ajuste

- Requiere revisión completa de los bloques disponibles
 - ... O mantenerlos en una lista ordenada
 - Empleando un ordenamiento en *montículo* (*heap*), puede ser tan ágil/simple como el *primer ajuste*
- Busca que el desperdicio sea el menor posible
 - Pero va generando muchos bloques muy pequeños



Peor ajuste

- Requiere revisión completa de los bloques disponibles
 - ... O mantenerlos en una lista ordenada
 - Empleando un ordenamiento en *montículo* (*heap*), puede ser tan ágil/simple como el *primer ajuste*
- Busca que los bloques que van quedando tras la creación de nuevos procesos *tiendan a ser* del mismo tamaño
 - Balanceando el tamaño de los bloques remanentes



Compactación

- Independientemente del esquema que elijamos, bajo particiones flexibles se irá fragmentando cada vez más la memoria
 - Si no se hace nada al respecto, no podrán lanzarse procesos nuevos
- La *compactación* consiste en:
 - Suspender temporalmente a un proceso
 - *Moverlo* a otra dirección de memoria
 - Ajustar su *registro base*
 - Continuar con el siguiente, hasta crear un sólo bloque disponible (de los muchos existentes)
- Tiene un costo alto, porque requiere:
 - Muchas transferencias de memoria
 - Suspensión sensible de los procesos implicados



Compactación

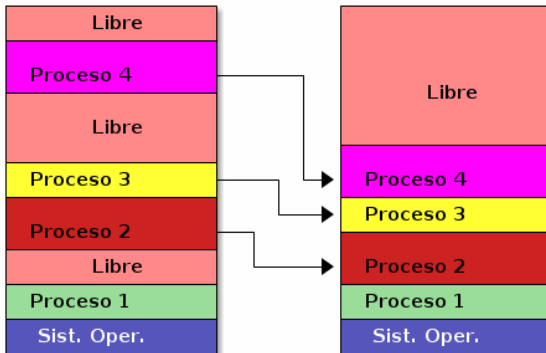


Figura: Compactación de la memoria de procesos en ejecución



¿Cuándo compactar?

No hay una sólo respuesta

- Basado en umbrales, verificando periódicamente el estado del sistema
- Basado en eventos, cada vez que no pueda satisfacerse una solicitud por haber demasiada fragmentación

Señales que indican necesidad de compactar

- Relación entre el número de bloques libres y ocupados
- Relación entre la memoria total disponible y el tamaño del bloque más grande
- ...



Intercambio (*swap*)

- El SO puede *comprometer* más memoria de la que tiene disponible
- Cuando inicia un sistema que *no cabe en memoria*, puede elegir suspender a un proceso y *grabarlo a almacenamiento secundario*
 - Por ejemplo, un proceso que esté bloqueado esperando un bloqueo externo
- ¿Qué pasa con las operaciones E/S que tiene pendientes el proceso?
 - Puede exigirse que sólo se pueda hacer E/S empleando **buffers** en el espacio del SO



Costos del intercambio

- Esta modalidad de swap fue popular en equipos de escritorio de fines de los 1980 y principios de 1990
 - Entre 1 y 8MB RAM
- Hoy en día resultarían inaceptablemente lentos
 - Si un proceso ocupa 100MB
 - Y la tasa de transferencia sostenida al disco duro es de 50MB/s
 - SATA ofrece máximos entre 150 y 600 MB/s (dependiendo de la generación)
 - Pero típicamente hay varios procesos compitiendo por el acceso
 - Suspender el proceso a disco toma un mínimo de 2s *de acceso exclusivo*
 - Traerlo de vuelta a memoria, otros 2s



Índice

- 1 Memoria contigua
- 2 Segmentación
- 3 Paginación



¿Cómo es la visión del programador?

- El trabajo del compilador es traducir lo que ve/entiende el programador a algo que pueda entender la computadora
- Para el programador, la memoria no es un *espacio contiguo*, sino que hay separaciones muy claras
 - El programador no tiene por qué ver relación entre las secciones de texto y datos
 - No tiene por qué preocuparse de la *cercanía* entre el *espacio de libres* y la *pila*
 - No tiene por qué importarle la estructura representada en la pila
 - Las bibliotecas externas enlazadas son meras *cajas negras*



Traduciendo la visión del programador

- ... ¿No podría traducirse esta separación a algo generado por el compilador?
 - Y que, de paso, pueda aprovechar el sistema...
- El espacio de un proceso se traduce en *varios segmentos* en memoria
 - En vez de sólo un registro *base* y un registro *desplazamiento*, requerimos de uno por segmento
 - Una *tabla de segmentos* por proceso.
 - Típicamente, un juego de *registros especiales* en el CPU
 - La resolución de direcciones es análoga a la descrita anteriormente, con apoyo del MMU
 - El direccionamiento se hace *explícitamente* indicando segmento y desplazamiento



Conceptualización de la segmentación

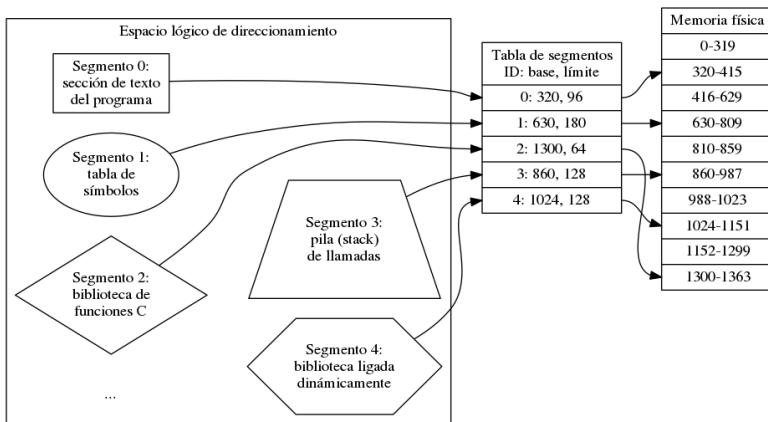


Figura: Ejemplo de segmentación (Silberschatz, p.305)



Facilidades que nos brinda la segmentación

- Ayuda a incrementar la *modularidad* de nuestro programa, incluso facilita el proceso de carga
 - Aún tiene que efectuarse la resolución de direcciones, pero puede delegarse parcialmente al MMU (en tiempo de ejecución)
- Permite especificar *permisos diferenciados* por tipo de memoria
 - Restringir escritura en segmentos de texto
 - Restringir ejecución en segmentos *no* de texto
 - Mejora la seguridad del sistema resultante
 - Ojo: No es magia, siguen existiendo muchos *vectores de ataque* que explotan el acomodo en memoria
- Hace más simple / conveniente el manejo del intercambio (swap)



Intercambio (swap) parcial

- Integrando la operación del SO y el MMU, pueden intercambiarse a disco *algunos* de los segmentos de un proceso
- Es muy probable que no todos los segmentos de un proceso sean usados *aproximadamente* al mismo tiempo, p.ej.:
 - Bibliotecas de importación/exportación de archivos, no son empleadas durante una ráfaga de cálculo
 - Bibliotecas de comunicación por red, no son empleadas mientras se prepara un archivo para su almacenamiento



Decidiendo qué y cuándo intercambiar

- El proceso puede indicar al CPU que no requiere por el momento determinado segmento
- El SO puede elegir, según ciertas métricas, cuál segmento mandar a disco
 - El más estorbo (grande)
 - Apelando a la localidad de referencia: El *Menos Recientemente Utilizado (LRU, Least Recently Used)*
- El MMU debe indicar al SO cuando el proceso solicita acceso a un segmento intercambiado
 - El SO suspende al proceso y carga al segmento de vuelta a memoria
 - ... Aunque con poca memoria disponible, eso puede llevar a que intercambie un segmento de otro proceso (o del mismo)
 - Volveremos a este punto al hablar de la *memoria virtual*



Rendimiento del intercambio

- Cada uno de los segmentos de un proceso es (obviamente) más chico que el proceso completo
 - La sobrecarga por transferir un segmento de/a disco es mucho menor
- El proceso puede seguir ejecutándose incluso si está parcialmente intercambiado
- El SO puede aprovechar los permisos para reducir muchas veces a la mitad el tiempo necesario para el intercambio



Reduciendo la transferencia basada en permisos

- Cuando un segmento no tiene permiso de escritura, sabemos que el proceso no puede cambiarlo
- Una vez que fue intercambiado, si el SO mantiene el espacio en disco reservado", no requiere volver a ser escrito
 - Tenemos garantía de que se mantendrá sin modificaciones
- Bajo ciertos supuestos, podemos incluso ahorrar la copia inicial
 - Cuando el segmento proviene de una biblioteca *reposicionable* y la imagen en disco es idéntica a la imagen en memoria, el archivo es un volcado directo del segmento



Ejemplificando

Si un proceso tiene la siguiente tabla de procesos, veamos la respuesta del MMU a diversas solicitudes

Segmento	Inicio	Tamaño	Permisos	Presente
0	15208	160	RWX	sí
1	1400	100	R	sí
2	964	96	RX	sí
3	-	184	W	no
4	10000	320	RWX	sí

R = Lectura; W = Escritura; X = Ejecución

El segmento 3 está en espacio de intercambio (Inicio nulo, presente=no).



Respuesta al atrapar una excepción

- Una excepción puede *lanzarse* ante diversas circunstancias
- Vemos a continuación algunos ejemplos
- El OS debe reaccionar de diferente forma ante cada una de ellas
 - El acceso a un *segmento faltante* debe llevar a suspender el proceso y traer el segmento de vuelta a memoria
 - Una violación de seguridad, o un acceso fuera de rango, normalmente llevarán a que el proceso sea terminado con una *falla de segmentación (segmentation fault)*
- *Ojo:* Puede presentarse más de un evento a la vez. ¿Cómo debemos reaccionar ante ello?



Ejemplificando

Dirección virtual	Tipo de acceso	Dirección física
0-100	R	15308
2-84	X	1048
2-84	W	Atrapada: Violación de seguridad
2-132	R	Atrapada: Desplazamiento fuera de rango
3-16	W	Atrapada: Segmento faltante
3-132	R	Atrapada: Segmento faltante; violación de seguridad
4-128	X	10130
5-16	X	Atrapada: Segmento invalido



Limitaciones de la segmentación

- Número y semántica de segmentos disponible en la arquitectura
 - En Intel 8086, 6 registros de segmento, mas un *desplazamiento*, dan el espacio de direccionamiento de 20 bits (1MB); (extendido a partir de 80386):
 - CS *Code Segment* (sección de texto)
 - DS *Data Segment* (sección de datos)
 - SS *Stack Segment* (pila de llamadas)
 - ES, FS, GS *Extra Segment* (cualquier otro acceso)



¿Qué tanto se aprovecha *en verdad*?

- El código que mejor aprovecha una arquitectura de segmentación es *muy difícil* de transportar a otra arquitectura
 - El compilador puede generar acorde a la arquitectura objetivo, pero... ¿Qué tanto beneficia al *programador*?
 - Recuerden que para eso se presentó (supuestamente)...
- Técnicas para direccionar más memoria de la que podemos *comprender*
 - El Pentium Pro (1995) permite, a través de segmentación, ver 36 bits (64GB) de memoria en una arquitectura de 32 bits
 - En PowerPC, hay 16 segmentos de 24 bits, que permiten direccionar hasta 52 bits



La segmentación hoy

- La segmentación va cayendo en desuso en casi todas las arquitecturas modernas
 - Incluso en x86: Al entrar en modo de 64 bits, se inhabilitan *casi* todos los registros de segmento (quedan FS y GS)
- Es muy susceptible a fragmentación (interna y externa)
 - Es natural, al tener cada proceso no uno, sino que varios bloques en memoria
 - Fragmentación interna → 0 en secciones de datos y texto, pero alta en libres, pila
- Prácticamente todos los sistemas modernos emplean un esquema de *memoria plana* mediante *paginación*
 - ... Que es, precisamente, el siguiente tema



Índice

- 1 Memoria contigua
- 2 Segmentación
- 3 Paginación



Evitando la fragmentación externa

- La paginación nace para evitar *definitivamente* la fragmentación externa
 - Y, por tanto, la necesidad de compactación
- Requiere hardware más especializado y dar seguimiento a mucha más información
- Simplifica fuertemente las cosas desde el punto de vista del proceso
 - A costo de complicarlas para el SO



Adios a los registros base+desplazamiento

- Al proceso se le muestra únicamente una *representación lógica* de la memoria
- Para cada proceso, el espacio de direccionamiento comienza en 0, y llega hasta el máximo posible de la arquitectura
- La memoria está dividida en una serie de *páginas*, todas ellas del mismo tamaño
 - Históricamente desde 512 bytes (2^9)
 - Típicamente 4 u 8K (2^{12} o 2^{13})
 - Máximos hoy, 16MB (2^{24})
- La memoria asignada ya no requiere ser contigua
 - A pesar de usar un *modelo plano* (no segmentado)



Esquemmatización del proceso de paginación

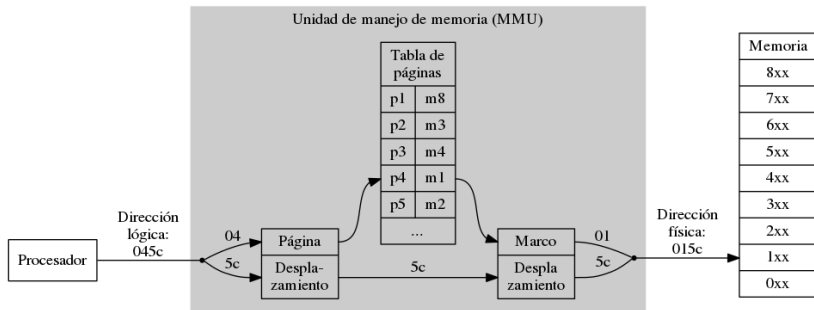


Figura: Esquema del proceso de paginación, ilustrando el rol del MMU



Ubicación física y lógica → Marcos y páginas

- Cada página corresponde a un *marco* (*frame*) en la memoria física
 - Relacionados a través de *tablas de páginas*
 - Los marcos *siempre* miden lo mismo que las páginas
 - El tamaño siempre será una potencia de 2
- La ubicación real de un *marco* de memoria es su *ubicación física*
- La dirección que conoce el proceso es su *ubicación lógica*
- La porción *más significativa* de una dirección de memoria indica la página; la *menos significativa*, el desplazamiento



Ejemplo de página y desplazamiento

Dirección especificada — 0001 1101 0111 0010

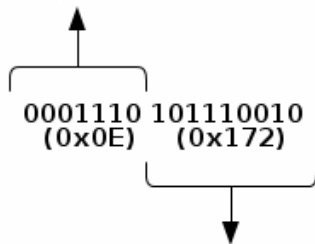


Figura: Página y desplazamiento, en un esquema de direccionamiento de 16 bits y páginas de 512 bytes



La tabla de páginas (*page table*)

- Guarda la relación entre cada *página* y el *marco* correspondiente
- El MMU no puede ya operar basado en unos cuantos registros
 - En un esquema limitado como el recién presentado, tenemos hasta 128 páginas (2^7)
 - Cada entrada requiere 14 bits (7 para la página, 7 para el marco)
 - → 1792 bytes (en un sistema bastante primitivo)
- El manejo de una tabla de páginas resulta en una suerte de resolución en tiempo de ejecución
 - Pero realizada por el MMU, transparente al programa (e incluso al procesador)
 - Con una *dirección base* distinta para cada una de las páginas



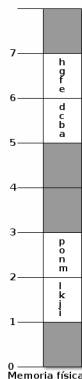
Ejemplo (minúsculo) de tabla de páginas

15	p
14	o
13	n
12	m
3	
11	i
10	k
9	i
8	
2	
7	h
6	g
5	e
4	
1	
3	d
2	c
1	b
0	a

Memoria lógica

3	2
2	1
1	6
0	5

Tabla de páginas



- Direccinamiento: 5 bits
- Página: 3 bits
- Desplazamiento: 2 bits
- Para referirse a la letra *f*

Figura: Ejemplo de paginación
(Silberschatz, p.292)



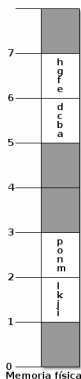
Ejemplo (minúsculo) de tabla de páginas

15	p
14	o
13	n
12	m
3	i
11	k
10	i
9	i
8	i
2	h
7	g
6	f
5	e
4	e
1	d
3	c
2	b
1	a
0	a

Memoria lógica

3	2
2	1
1	6
0	5

Tabla de páginas



- Direccionamiento: 5 bits
- Página: 3 bits
- Desplazamiento: 2 bits
- Para referirse a la letra *f*:
 - 1 Dirección 00101 (5)

Figura: Ejemplo de paginación
(Silberschatz, p.292)



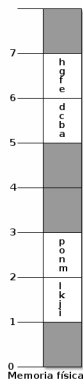
Ejemplo (minúsculo) de tabla de páginas

15	p
14	o
13	n
12	m
3	i
11	k
10	i
9	i
8	
2	h
7	g
6	e
5	e
4	
1	d
3	c
2	b
1	a
0	

Memoria lógica

3	2
2	1
1	6
0	5

Tabla de páginas



- Direccionamiento: 5 bits
- Página: 3 bits
- Desplazamiento: 2 bits
- Para referirse a la letra *f*:
 - 1 Dirección 00101 (5)
 - 2 Página 001 (1), posición 01 (1)

Figura: Ejemplo de paginación
(Silberschatz, p.292)



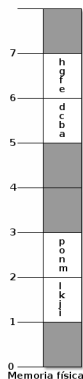
Ejemplo (minúsculo) de tabla de páginas

15	p
14	o
13	n
12	m
3	i
11	k
10	i
9	i
8	i
2	h
7	g
6	f
5	e
4	e
1	d
3	c
2	b
1	a
0	a

Memoria lógica

3	2
2	1
1	6
0	5

Tabla de páginas



- Direccionamiento: 5 bits
- Página: 3 bits
- Desplazamiento: 2 bits
- Para referirse a la letra *f*:
 - 1 Dirección 00101 (5)
 - 2 Página 001 (1), posición 01 (1)
 - 3 MMU: Página 001 → marco 6 (110)



Figura: Ejemplo de paginación
(Silberschatz, p.292)

Ejemplo (minúsculo) de tabla de páginas

15	p
14	o
13	n
12	m
3	l
11	k
10	i
9	i
8	h
7	g
6	e
5	e
4	d
1	c
3	b
2	a
1	a
0	a

Memoria lógica

3	2
2	1
1	6
0	5

Tabla de páginas



- Direccionamiento: 5 bits
- Página: 3 bits
- Desplazamiento: 2 bits
- Para referirse a la letra *f*:
 - 1 Dirección 00101 (5)
 - 2 Página 001 (1), posición 01 (1)
 - 3 MMU: Página 001 → marco 6 (110)
 - 4 Dirección física: 11001 (26)



Figura: Ejemplo de paginación
(Silberschatz, p.292)

Permisos y validez

- Al emplear memoria paginada, seguimos manteniendo los modos de *permisos diferenciados* de la segmentación
 - Lectura, escritura, ejecución (RWX)
- En vez de aplicarse a segmentos, se aplican página por página
- Muchas veces, *la cantidad de bits* necesaria para alinear las tablas con el acceso a memoria
- Un bit adicional: Página válida / inválida
 - El espacio total de direccionamiento disponible al proceso es muy grande (32/64 bits)
 - Indica cuáles páginas tiene asignadas el proceso y cuáles están libres



Efecto ante la fragmentación

- La fragmentación externa desaparece
 - O más bien, nos deja de preocupar
- ¿Y la fragmentación interna?



Efecto ante la fragmentación

- La fragmentación externa desaparece
 - O más bien, nos deja de preocupar
- ¿Y la fragmentación interna?
- Al dividirse la memoria en bloques de 2^n bytes, cada proceso desperdiciará en promedio $\frac{2^n}{2}$ bytes
 - Peor caso: $2^n - 1$ bytes
- Podemos tener cientos o miles de procesos en el sistema



Efecto ante la fragmentación

- La fragmentación externa desaparece
 - O más bien, nos deja de preocupar
- ¿Y la fragmentación interna?
- Al dividirse la memoria en bloques de 2^n bytes, cada proceso desperdiciará en promedio $\frac{2^n}{2}$ bytes
 - Peor caso: $2^n - 1$ bytes
- Podemos tener cientos o miles de procesos en el sistema
- Respuesta lógica: Hagamos a n tan pequeño como sea posible
- Muchas páginas, muy pequeñas. . .



Efecto ante la fragmentación

- La fragmentación externa desaparece
 - O más bien, nos deja de preocupar
- ¿Y la fragmentación interna?
- Al dividirse la memoria en bloques de 2^n bytes, cada proceso desperdiciará en promedio $\frac{2^n}{2}$ bytes
 - Peor caso: $2^n - 1$ bytes
- Podemos tener cientos o miles de procesos en el sistema
- Respuesta lógica: Hagamos a n tan pequeño como sea posible
- Muchas páginas, muy pequeñas. . . ¿O no?



Impacto del tamaño de las páginas

¿En qué se expresaría la *sobrecarga administrativa* de tener páginas demasiado pequeñas?

- Mayor carga en un cambio de contexto
 - Crecimiento del PCB
- Mayor número transferencias DMA requerido (p.ej. para leer/escribir del disco)
- Disminución de velocidad de *todas* las transferencias a memoria



Impacto en las transferencias DMA

- Es mucho más eficiente hacer una transferencia larga que varias cortas
 - Recordemos: Una transferencia DMA se *inicia* indicando:
 - Dirección física base de memoria
 - Cantidad de datos a transferir
 - *Puerto* del dispositivo
 - *Dirección* de la transferencia
- Tenemos que hacer (como máximo) transferencias del tamaño de la página en memoria
 - Es poco probable que las páginas lógicamente consecutivas estén físicamente contiguas
- Fragmentar la memoria física en páginas muy pequeñas reduce la velocidad de transferencia de disco



Impacto en el PCB

Volvamos a hacer cuentas:
¿Cuánto pesa cada esquema de asignación de memoria?

Partición fija Un *registro base* para indicar el *inicio* del espacio del proceso

Partición flexible Un registro base y un *registro límite* para indicar su tamaño total

Segmentación Varios registros *de segmento* (p.ej. 6 en x86)

Paginación Modelo presentado (128 páginas de 512 bytes, espacio de direccionamiento de 16 bits): 1792 bytes



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .

- 1,048,576 páginas (2^{20})



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .

- 1,048,576 páginas (2^{20})
- Cada entrada de 40 bits (20 para la página, 20 para el marco)



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .

- 1,048,576 páginas (2^{20})
- Cada entrada de 40 bits (20 para la página, 20 para el marco)
- $\frac{40}{8} \times 1048576 \rightarrow$ ¡¿5MB para la tabla de páginas?!



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .

- 1,048,576 páginas (2^{20})
- Cada entrada de 40 bits (20 para la página, 20 para el marco)
- $\frac{40}{8} \times 1048576 \rightarrow$ ¡¿5MB para la tabla de páginas?!
- $\frac{5242880}{4096} = 1280 \rightarrow$ ¡¿La tabla de páginas mide 1280 páginas?!



¿Cuánto ocuparía en una computadora real actual?

Computadora sencilla actual: Páginas de 4096 bytes (2^{12}), espacio de direccionamiento de 32 bits. . .

- 1,048,576 páginas (2^{20})
- Cada entrada de 40 bits (20 para la página, 20 para el marco)
- $\frac{40}{8} \times 1048576 \rightarrow$ ¡¿5MB para la tabla de páginas?!
- $\frac{5242880}{4096} = 1280 \rightarrow$ ¡¿La tabla de páginas mide 1280 páginas?!
- Con memoria DDR3-14900 y transferencia pico de 15GB/s, le agrega 0.3ms al cambio de contexto
 - Quedamos en que un cambio de contexto *típico* debe ser <1ms. . .



¿Cuánto ocuparía en una computadora real actual?

... ¿Quién se aventura a sacar números para un sistema de 64 bits?



¿Cuánto ocuparía en una computadora real actual?

... ¿Quién se aventura a sacar números para un sistema de 64 bits?
Un tip: Basta multiplicar los números de la lámina anterior por
4,294,967,296



Almacenamiento de la tabla de páginas en memoria

- Almacenar toda esta información en registros es sencillamente imposible
 - Costo económico
 - Tiempo perdido en el cambio de contexto
- Una estrategia: Almacenar *la propia tabla de páginas* en memoria
 - Emplear *sólo dos* registros especiales:
 - PTBR Registro Base de Tabla de Páginas (*Page Table Base Register*)
 - PTLR Registro Límite de Tabla de Páginas (*Page Table Limit Register*)
 - ¿Por qué el PTLR? La mayor parte de los procesos, además, nunca requerirá direccionarlo completo; para no tener un mapa completo mayormente vacío, PTLR indica la extensión máxima empleada



Desventaja de tener la tabla de páginas en memoria:

- Velocidad: *Cada acceso a memoria* se ve penalizado con (al menos) un acceso adicional
 - Para resolver dónde está la página que buscamos
- *Duplica* el tiempo efectivo de acceso a memoria → Inaceptable.



TLB: El buffer de traducción adelantada

- ¿Respuesta? Emplear un caché
 - Pero, por su frecuencia y naturaleza de uso, un caché especializado
- El TLB (*Translation Lookaside Buffer*) es una tabla asociativa (*hash*)
 - Las consultas (*llaves*) son las páginas
 - Los *valores* son los marcos correspondientes
 - Las búsquedas se realizan en *tiempo constante*



Proceso de paginación con TLB

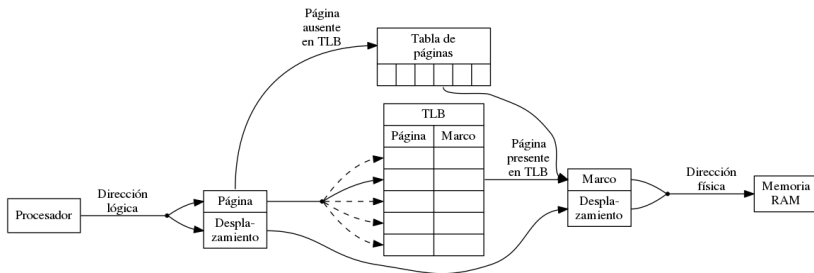


Figura: Esquema de paginación empleando un *buffer de traducción adelantada* (TLB)



Funcionamiento del TLB

- Típicamente, entre 64 y 1024 entradas
 - Busca aprovechar la *localidad de referencia*
- Si *conoce* ya a una página (*TLB hit*), el MMU traduce de inmediato para obtener el marco
 - Silberschatz: Penalización de hasta 10%
- Si *no* conoce la página (*TLB miss*), lanza una falla de página (*page fault*) y consulta en la memoria principal cuál es el marco correspondiente
 - Penalización >120%; la nueva página queda registrada en el TLB



Reemplazo de las entradas del TLB

- El TLB es *mucho más pequeño* que el espacio de páginas del proceso
- Al recibir una entrada nueva, puede ser necesario reemplazar la entrada de una página ya conocida
- Reemplazar la *menos recientemente utilizada (LRU)*
 - Ventaja: Localidad de referencia; probablemente no la necesitemos pronto
 - Desventaja: Contabilizar los accesos dentro del TLB (*muy frecuentes*) agrega latencia y costo
- Reemplazar una página al azar
 - Ventaja: Más simple y barato (pero... ¿Algo aleatorio es barato en el cómputo?)
 - Desventaja: Obvia. Puede reemplazarse una página en uso frecuente y causar demoras



La obesidad de las tablas de páginas

En un equipo moderno, incluso con TLB, el espacio de la tabla de páginas es demasiado grande

- Como vimos, una sobrecarga de 5MB por proceso en 32 bits. Y de nuevo... ¿en 64 bits?
- Tendríamos 2^{42} entradas con páginas de 4MB (2^{22})
- $2^{64}/2^{22} = 2^{42}$ entradas, cada una ocupando 42 bits para la página y 42 bits para el marco
- $2^{42} \times (42 + 42) \rightarrow$ ¡336TB *por proceso!*



Subdividiendo las tablas de páginas

- Aprovechamos que la mayor parte del espacio de direccionamiento de un proceso está siempre vacío
- Podemos dividir una dirección de memoria en dos (o más) niveles)
- Por ejemplo, una dirección de 32 bits:
 - Tabla externa 10 bits (1024 entradas; $1024 \times (10 + 10)$ bits
2560 bytes)
 - Tabla interna 10 bits cada una (2560 bytes cada una)
 - Desplazamiento 12 bits (dentro de cada página/marco)
- Sólo se crean las tablas internas que sean necesarias
- El TLB guarda la resolución para los *20 bits* de ambas tablas
 - O los 54 bits correspondientes en una arquitectura de 64 bits



Tablas de páginas con varios niveles

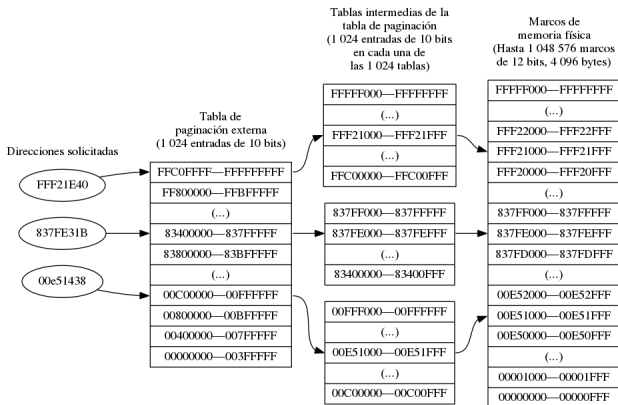


Figura: Paginación en dos niveles: 10+10 bits, con marcos de 12 bits.



Los costos de los niveles adicionales

- Este esquema hace *tratable* el direccionamiento de grandes cantidades de memoria
 - Pueden agregarse tres, cuatro, cinco niveles. . .
- Pero una *falla de página* puede ahora *triplicar* el tiempo de acceso a memoria en vez de duplicarlo
 - Porque hace falta consultar a la *tabla externa* y a la *tabla interna*
- En una arquitectura de 64 bits dividida en páginas en incrementos 10 bits. . . *Septuplicamos* el tiempo de acceso



Tablas de páginas con *funciones digestoras* (hash)

¿Qué es una *función digestora*?

$$H : U \rightarrow M$$

- Una función que *mapea* o *proyecta* al conjunto U en un conjunto M mucho menor
- La *distribución resultante* en M debe resultar homogénea
- Tan poco dependiente de la secuencialidad de la entrada como sea posible
- Permiten hacer *mapeos* a espacios muestrales mucho más pequeños
- Pero, en vez de apuntar a un sólo valor, deben apuntar a una *lista ligada* de valores
 - Al ser espacios más pequeños, *necesariamente* pueden presentar colisiones



Tablas de páginas con funciones digestoras (*hash*)

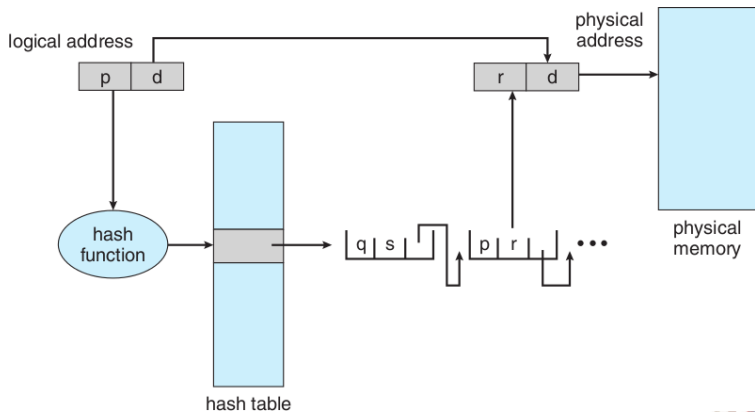


Figura: Tablas de páginas empleando funciones digestoras (Silberschatz, p. 302)



Características de las tablas basadas en *hash*

- Mayor complejidad → mayor latencia (¿mayor a qué?
Seguramente, no a 7 accesos a memoria)
 - Evaluación de una función matemática (hash)
 - Posiblemente, recorrer una lista
- El tamaño de la tabla puede variar según crece el uso de memoria del proceso
 - Aunque requiera recalcular (*rehash*) la tabla, es una operación poco frecuente



Memoria compartida (1)

- La memoria paginada hace simple definir áreas de memoria compartida
- Uso muy frecuente: Comunicación entre procesos (IPC)
 - Pueden compartirse estructuras complejas sin costo de copia
- Acceso a los datos dentro de estas estructuras por parte de los procesos: Requiere protección por mecanismos de *sincronización*
 - Ojo: Diferencias entre memoria compartida entre procesos y memoria compartida entre hilos



Memoria compartida (2)

Uso más frecuente aún: Compartir *código*

- No tiene sentido que el SO *repita* en memoria imágenes de bibliotecas *relocalizables* y *reentrantes*
- El mismo conjunto de marcos puede incluirse en las tablas de distintos procesos, aumentando la capacidad percibida de memoria
- Esto es lo que conocemos como *bibliotecas compartidas*
 - No sólo se comparten en disco — También en memoria
 - Explican (parte de) un uso efectivo de memoria muy superior al 100 %



Memoria compartida (3)

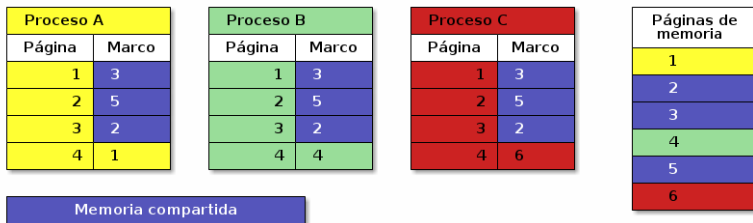


Figura: Uso de memoria compartida: Tres procesos comparten la memoria ocupada por el texto del programa (azul), difieren sólo en los datos.



La arquitectura x86-32: Paginación + Segmentación



Figura: Pasos de la traducción lógica a física en un Pentium

- La arquitectura x86 de 32 bits permite combinar paginación y segmentación para alcanzar un mayor espacio de direccionamiento
 - Comparable con la segmentación que permitía a la x86 llegar a 1MB RAM
 - La segmentación fue (casi) eliminada para x86-64



Segmentación en x86-32

- Hay dos tablas de segmentos, almacenadas en *registros programables* específicos:
 - LDT** *Tabla local de descriptores*, segmentos privados de cada uno de los procesos
 - GDT** *Tabla global de descriptores*, segmentos que pueden ser compartidos entre diversos procesos del sistema
- La dirección de segmento es de 16 bits
 - 13 bits indican el segmento
 - 1 bit indica si es en LDT o en GDT
 - 2 bits son para los permisos



Paginación en x86-32

- La familia Pentium (y equivalentes) pueden emplear páginas de 4KB o 4MB
 - La tabla de páginas está dividida en una externa de 10 bits y una interna de 10 bits (y desplazamiento, 12 bits)
 - La *tabla externa* incluye un bit que indica si apunta a una *página grande* o a una *tabla interna*



Paginación en x86-32

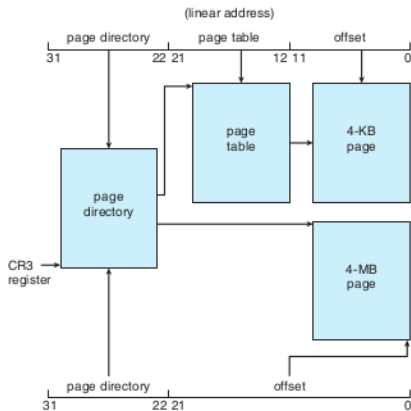


Figura: Paginación en x86-32 (silberschatz, p.309)

