

Tema 3: Gestión de memoria

Profesorado de Sistemas Operativos

Universidad de Granada

Sistemas Operativos

Grado de Ingeniería Informática

Grado en Ingeniería Informática y Matemáticas

Grado en Ingeniería Informática y Administración y Dirección de Empresas

2025-2026

Index

- 1 Objetivos
- 2 Jerarquía de memoria
- 3 Espacio físico y lógico
- 4 Gestión de la memoria. Paginación
- 5 Memoria virtual
- 6 Algoritmos de reemplazo y reparto de espacio
- 7 Gestión de memoria en Linux
- 8 Bibliografía

- Distinguir entre dirección relativa o lógica y dirección física o real y entre espacio de direcciones lógico y físico.
- Conocer las distintas formas en las que el sistema operativo puede organizar y gestionar la memoria física.
- Saber en qué consiste y para qué se utiliza el intercambio (swapping).
- Conocer el mecanismo de paginación en un sistema con memoria virtual.
- Comprender qué es la propiedad de localidad y su relación con el comportamiento de un programa en ejecución.
- Conocer características de la gestión de la memoria de un proceso en sistemas Linux.

Jerarquía de memoria

Jerarquía de memoria

- Existen **distintos tipos de dispositivos** dentro de la categoría del almacenamiento.
- En general se cumple que el almacenamiento de menor coste es el que ofrece un acceso menos rápido y viceversa.
- Los elementos frecuentemente accedidos se intentará almacenarlos en la memoria más rápida disponible, que tendrá un menor tamaño, y el resto en memoria más lenta, aunque de mayor capacidad y menor coste.
- Existen sin embargo restricciones tales como que es necesario que la parte de un proceso en ejecución resida en memoria principal.

Jerarquía de memoria cont.

Jerarquía de memoria cont.

En orden decreciente de velocidad:

- **Registros del procesador.**
- **Caché del procesador.** Existen distintos niveles y es habitual que esté dentro del encapsulado del mismo. Sin embargo, puede ser exclusiva de un núcleo o compartida entre varios núcleos del mismo procesador.
- **Memoria principal:** su acceso requiere el acceso al bus que la conecta con el procesador.
- **Memoria auxiliar/secundaria:** su acceso requiere el acceso al bus de entrada/salida.

Jerarquía de memoria cont. Caché

Caché

- Los programas cumplen el principio de localidad: si un dato o instrucción es accedido, es altamente probable que en un tiempo cercano vuelvan a ser accedidos o lo sean datos/instrucciones cercanas.
- Acierto de cache: el ítem buscado se encuentra en la misma y se evita el acceso a un espacio de almacenamiento más lento. En caso contrario se produce un fallo de caché.
- Al cumplirse el principio de localidad, los aciertos de caché superan por mucho los fallos y hacen que el uso de cachés proporcione un incremento del rendimiento.
- Además de la caché entre los registros CPU y la memoria principal se utiliza, por parte del sistema operativo, parte de la memoria principal como caché de datos almacenados en memoria secundaria.

Espacio de direcciones lógico y espacio físico

Direcciones lógicas y físicas

- Espacio de direcciones lógico: direcciones lógicas generadas por un programa.
- Espacio de direcciones físico: conjunto de direcciones físicas correspondientes a las direcciones lógicas en un momento del tiempo.
- La existencia de ambos espacios requiere que se realice una traducción. Este proceso de traducción es altamente dependiente del esquema de organización utilizado por el sistema operativo.
- MMU (Memory Management Unit): dispositivo hardware que realiza dicha traducción. Forma parte del procesador y es gestionado por el sistema operativo.
- Para reducir el coste asociado a la traducción de todas las direcciones de memoria lógicas usadas por un programa, la MMU dispone de una caché: el TLB (Translation Lookaside Buffer).

Espacios de direcciones físicos y lógicos

CASO 1: Dirección lógico = Dirección física

CASO 2: Proceso se carga en otra zona de memoria

Fichero ejecutable

0
4
...
96
100
104
108
112
116
120
124
128
132
136

Cabecera

Memoria

0
4
8
12
16
20
24
28
32
36

LOAD R1, #1000
LOAD R2, #2000
LOAD R3, (1500)
LOAD R4, (R1)
STORE R4, (R2)
INC R1
INC R2
DEC R3
JNZ 12
....

Memoria

100
104
108
112
116
120
124
128
132
136

LOAD R1, #1000
LOAD R2, #2000
LOAD R3, (1500)
LOAD R4, (R1)
STORE R4, (R2)
INC R1
INC R2
DEC R3
JNZ 12
....

```
for (i=0; i<tam; i++)
{
    v2[i] = v1[i];
}
```

Los vectores están almacenados a partir de las direcciones 1000 y 2000

Tamaño de los vectores en la dirección 1500

```
0 LOAD R1, #1000 ; R1 = 1000 -> apunta a v1[0]
4 LOAD R2, #2000 ; R2 = 2000 -> apunta a v2[0]
8 LOAD R3, (1500) ; R3 = [1500] -> R3 = tam (tamaño del vector)
```

```
12 LOOP: LOAD R4, (R1) ; R4 = [R1] -> R4 = v1[1]
16 STORE R4, (R2); [R2] = R4 -> v2[1] = R4

20 INC R1 ; R1 = R1 + 1 -> avanza al siguiente elemento de v1
24 INC R2 ; R2 = R2 + 1 -> avanza al siguiente elemento de v2

28 DEC R3 ; R3 = R3 - 1 -> decrementa contador de elementos

32 JNZ 12 ; si R3 ≠ 0, salta a LOOP (dirección 12)
```

Figura: Resultados según la zona en la que se carga el proceso

Mapa de memoria de un proceso y regiones

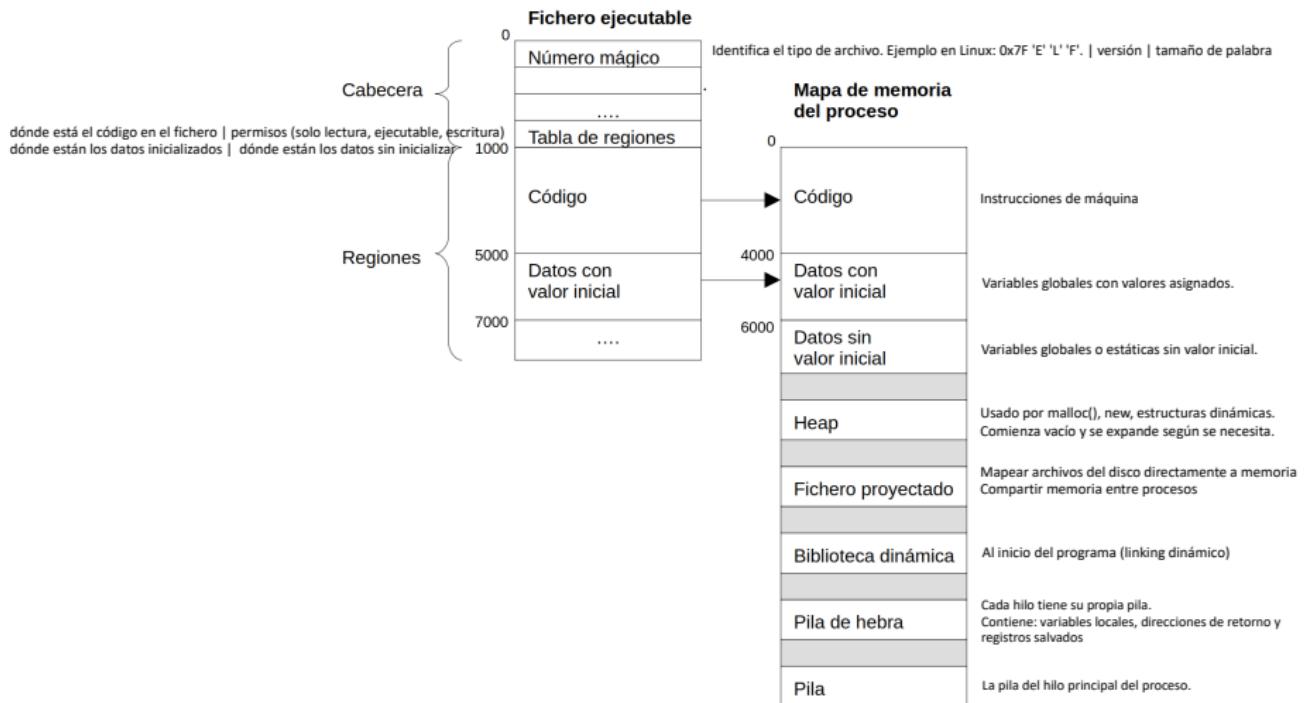


Figura: Mapa de memoria de un proceso

Aspectos generales de la gestión de la memoria

Aspectos relacionados con la gestión de la memoria

- Organización: cómo se divide y estructura la memoria.
- Gestión: estrategias a aplicar al esquema organizativo elegido para maximizar el rendimiento.
 - ▶ Estrategias de asignación (Ejemplos: contigua o no contigua)
 - ▶ Estrategias de sustitución o reemplazo.
 - ▶ Estrategias de búsqueda o recuperación.
- Protección:
 - ▶ Del sistema operativo de los procesos de usuario.
 - ▶ Los procesos de usuarios entre ellos.

Asignación contigua y no continua

Asignación contigua

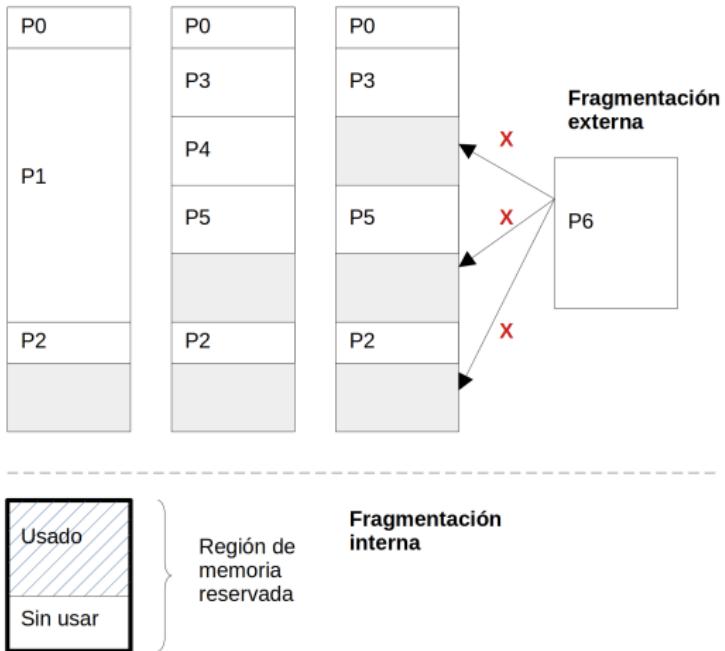
El asignar todo el almacenamiento para un proceso en un único bloque de posiciones contiguas de memoria, aunque conceptualmente sencillo, genera una alta fragmentación externa.

Asignación no contigua

Se divide el proceso en bloques que se almacenan en zonas no necesariamente contiguas.

Fragmentación

Memoria principal

**Figura:** Fragmentación externa e interna

Asignación no continua

Segmentación

- Consiste en la aplicación de la asignación contigua por regiones.
- También produce fragmentación externa.
- En la arquitectura x86 existen registros dedicados para algunos segmentos: Code Segment (CS), Data Segment (DS), Stack Segment (SS) y Extra Segment (ES).
- En la arquitectura x86-64 ya no se utiliza y los registros CS, DS, SS y ES se fijan a 0.

CS – Code Segment
Base del código.

DS – Data Segment
Base de los datos y variables globales.

SS – Stack Segment
Base de la pila.

ES – Extra Segment
Segmento adicional para ciertas operaciones (ej. cadenas).

Asignación no continua

Paginación

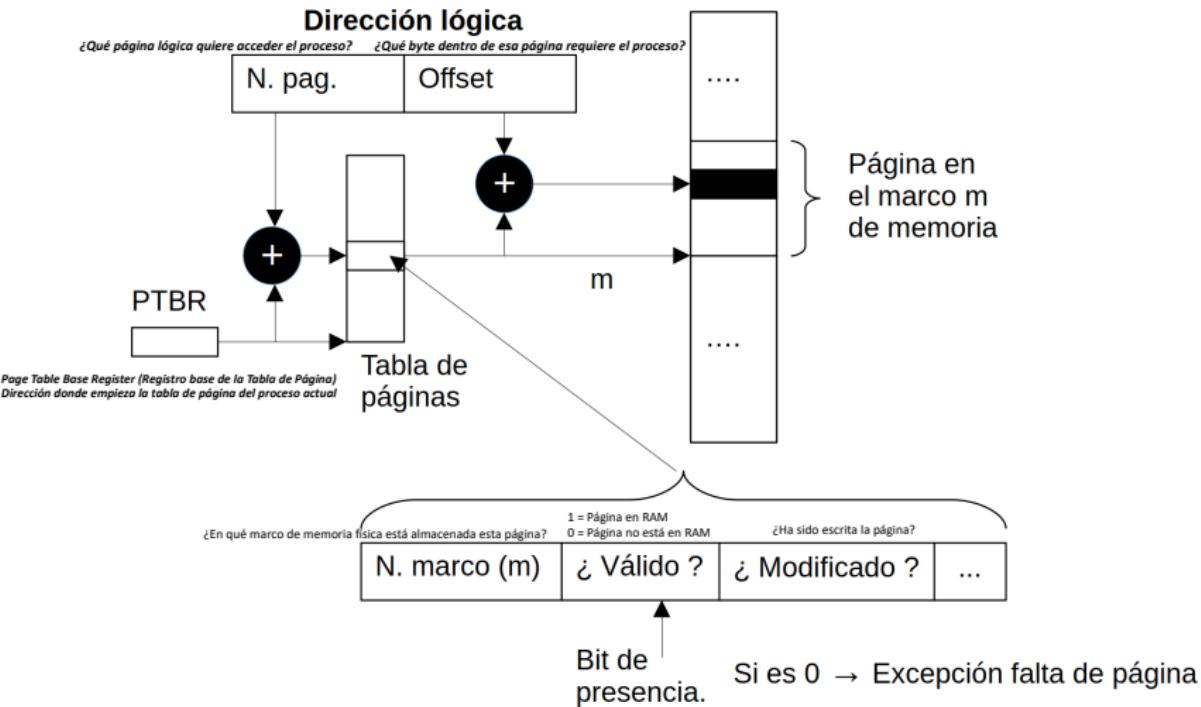
- Para minimizar la fragmentación externa se divide la memoria principal en bloques: marcos de página.
- El mapa del proceso se divide en páginas, siendo los marcos y las páginas de igual tamaño.
- 4KB un tamaño muy utilizado.
- Para disponer de un determinado contenido en memoria principal se almacena la página que lo contiene en un marco de página.
- Si las páginas tuvieran el tamaño de una posición de memoria la fragmentación sería nula, pero los requisitos del proceso de traducción serían inasumibles.

Paginación

Tablas de páginas

- La MMU realiza la traducción de página a marco de página en la que se encuentra.
- Dicha traducción se hace mediante una tabla de páginas. El crecimiento del espacio del espacio de almacenamiento ha obligado a implementar esquemas multinivel para evitar tablas de página demasiado grandes. Con el esquema multinivel solo se almacena en memoria principal la parte de la tabla de páginas que es requerida en cada momento.
- Tamaño de página: si se disminuye se reduce la fragmentación interna pero aumenta el coste de la gestión de la paginación. Si se aumenta el tamaño de la página, aumenta la fragmentación interna, pero se reduce dicho coste.

Traducción de direcciones lógicas mediante paginación



Ejemplo de paginación

Mapa de memoria

Pag 0
Pag 1
Pag 2
Pag 3
Pag 4
Pag 5
Pag 6
Pag 7
Pag 8
Pag 9
Pag 10
Pag 11

Tabla de páginas

(Página)	Marco	V/I	
MMU	0	7	V
	1	I	...
	2	0	V
	3	1	V
	4		I
	5		I
	6	5	V
	7		I
	8		I
	9	4	V
	10		I
	11	9	V

Memoria

(Marco)	Contenido
0	Pag 2
1	Pag 3
2	
3	
4	Pag 9
5	Pag 6
6	
7	Pag 0
8	
9	Pag 11
10
11

Página 3 → Marco 1 (Válida)

Página 1 → I → no está en memoria → Page Fault

El SO deberá traer Pág 1 desde disco a un marco vacío (por ejemplo marco 2), y actualizar la tabla: Página 1 → Marco 2, V

Figura: Ejemplo de paginación

Paginación multinivel

- Para evitar utilizar, y mantener en memoria, tablas de páginas demasiado grandes, es habitual la implementación de un esquema multinivel.
- Se realiza una paginación de la propia tabla de páginas.
- Las direcciones lógicas indican las tablas de páginas de cada nivel que hay que recorrer para obtener el marco de página.

Dirección lógica 14 bits



Figura: Ejemplo de dirección lógica con paginación con 2 niveles

Ejemplo de paginación con 2 niveles

El primer nivel decide qué tabla de nivel 2 debe usar.

El segundo nivel contiene la traducción real (pág → marco)
¿Qué marco corresponde a cada página?

Mapa de memoria

Pag 0
Pag 1
Pag 2
Pag 3
Pag 4
Pag 5
Pag 6
Pag 7
Pag 8
Pag 9
Pag 10
Pag 11

(Tabla nivel 2)

Grupo | Páginas que contiene
 Tabla de nivel 2 nº 0 | páginas 0–3
 Tabla de nivel 2 nº 1 | páginas 4–7
 Tabla de nivel 2 nº 2 | páginas 8–11

Tabla de páginas nivel 1

Marco	V/I	
2	V	...
1	I	...
6	V	...

(Página)

→ 0

1

2

3

(Página)

8

9

10

11

Tabla de páginas nivel 2

Marco	V/I	
7	V	...
1	I	...
0	V	...
1	V	...

Tabla de páginas nivel 2

Marco	V/I	
	I	...
4	V	...
	I	...
9	V	...

(Marco)	Contenido
0	Pag 2
1	Pag 3
- 2	TP_Pag 0
3	
4	Pag 9
5	
6	TP_Pag 2
→ 7	Pag 0
8	
9	Pag 11
10
11

¿Cómo se traduce una dirección lógica en 2 niveles?

Supón que el proceso accede a página 3:

* La MMU toma el índice de nivel 1

→ página 3 está en el grupo 0 (páginas 0–3)

* Nivel 1 entrada 0 → marco 2

→ ahí está la tabla nivel 2 correspondiente

* En esa tabla: Entrada 3 → marco 1 (V)

Se suma el offset → marco 1 + desplazamiento

Figura: Paginando la tabla de páginas

Cálculo de dirección física con paginación con 2 niveles

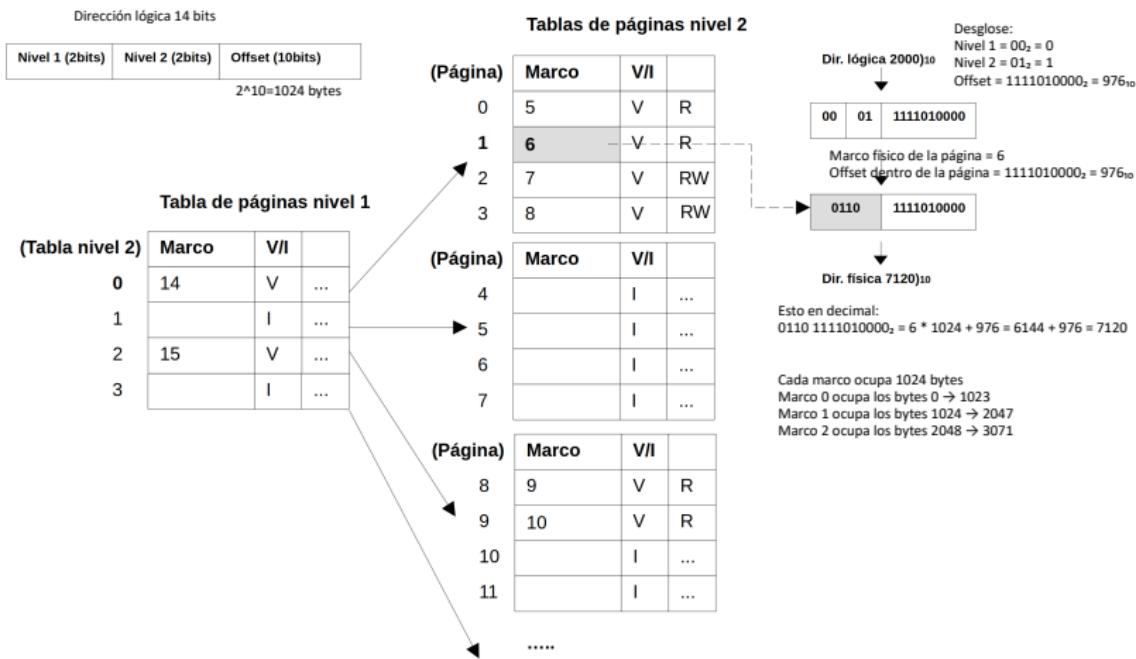


Figura: Cálculo de dirección física con paginación con 2 niveles

Regiones de memoria y paginación

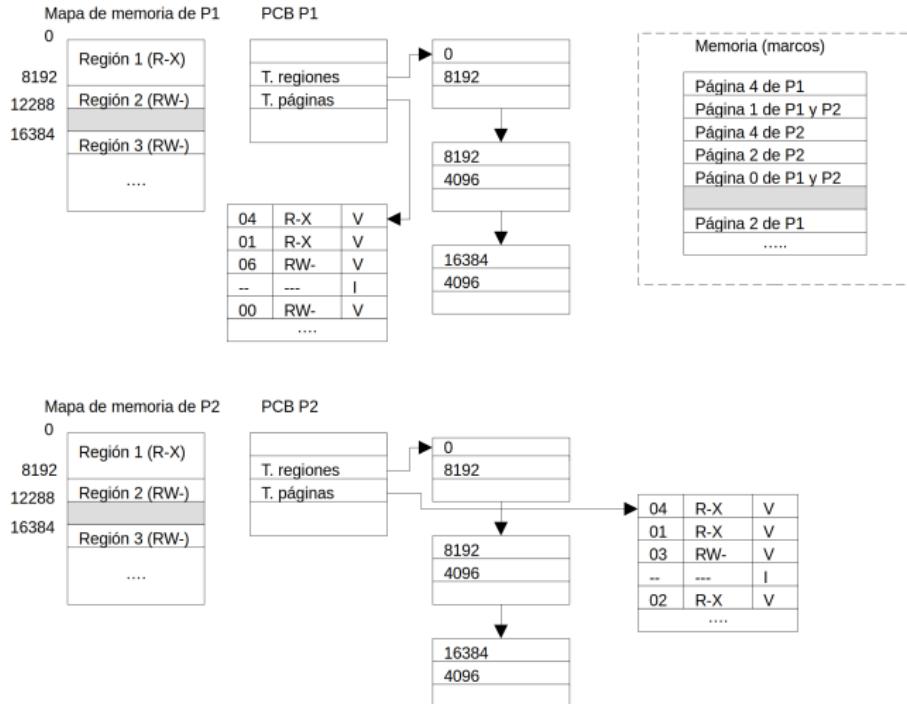


Figura: Regiones de memoria y paginación

Memoria virtual

La memoria virtual es una técnica que permite:

- Ejecutar procesos cuyo mapa de memoria es mayor que la memoria principal disponible.
- Aumentar el nivel de multiprogramación ejecutando más procesos de los que caben en memoria principal.

Estos objetivos se consiguen utilizando parte del almacenamiento secundario para almacenar contenido de la memoria principal asegurando que en memoria principal estén los elementos que son necesarios en cada instante y los que van a ser requeridos a corto plazo.

Memoria virtual y paginación

Memoria virtual, paginación y faltas de página

En determinados momentos, páginas que se requiere no estarán en memoria principal (falta de página). El tratamiento a alto nivel implica:

- Bloquear el proceso (otro proceso recibe el uso del procesador).
- Encontrar la ubicación en memoria secundaria de la página solicitada utilizando una tabla de ubicación en disco.
- Encontrar un marco libre. Si no lo hubiera, desplazar una página de memoria principal a memoria secundaria.
- Cargar la página desde disco al marco asignado.
- Actualizar tabla de páginas, incluyendo el bit de presencia, permisos, etc.
- Desbloquear el proceso. Este podrá reiniciar la instrucción que originó la falta de página cuando vuelva a usar el procesador.

Algoritmos de reemplazo de páginas

Algoritmos de reemplazo

- Algoritmo de reemplazo óptimo: sustituye la página que va a tardar más en referenciarse (o que no se va a referenciar más). Es un algoritmo no implementable ya que requeriría conocer el futuro. No es en general posible saber cuál va a ser el patrón de acceso a las páginas.
- FIFO: sustituye la página más antigua. Presenta problemas con páginas que son utilizadas durante largos intervalos de tiempo y que serán retiradas de memoria solo por su antigüedad.
- LRU (Least Recently Used): sustituye la página que fue objeto de la referencia más antigua.
- Algoritmo del reloj: se explica con detalle a continuación.

Algoritmo del reloj

- Es una versión mejorada del algoritmo FIFO proporcionando una segunda oportunidad a las páginas más usadas.
- El propósito es evitar que páginas muy utilizadas sean eliminadas simplemente por el hecho de llevar mucho tiempo en memoria principal.
- Cada página residente tiene asociado un bit de referencia. Cuando una página es usada dicho bit se pone a 1.
- Si por antigüedad a una página le correspondería ser sustituida, si dicho bit de referencia está a 1, se pone a 0 y se pasa a la siguiente página candidata.
- Recibe su nombre de la implementación del mismo mediante una lista circular donde se utiliza un puntero para señalar la posición de dicha lista candidata a ser ocupada por la siguiente página que requiera ser cargada.

Ejemplo de aplicación del algoritmo del reajuste

1º vuelta: 2, 3,1
2º vuelta: 2 reemplazado

P. requerida		2		3		2		1		5		2
	→	2*		2*		2*	→	2*		5*		5*

P. requerida 4 5 3 2 5 2

	\rightarrow	5^*	\rightarrow	5^*		3^*	$\xrightarrow{\text{Mantiene}}$	3^*	\rightarrow	3^*	\rightarrow	3^*
		2^*		2^*	\rightarrow	2		2^*		2		2^*
		4^*		4^*		4		4		5^*		5^*

¿Quitar p. X X X

1º vuelta: 5, 2, 4

2^a vuelta: 5 reemplazados

1º vuelta: 3, 2

4 reemplazado

- X Falta de página con todos los marcos de página en uso
 - * Bit de referencia con valor 1. Su valor es 0 en otro caso

Errores comunes detectados al aplicar este algoritmo

- Si no se añade una nueva página, o se sustituye una residente por otra, el puntero no se mueve. Si no hay por tanto una falta de página no se modifica la posición de dicho puntero.
- Si se accede a una página residente y el bit de referencia de la misma tiene el valor 0, se cambia a 1 independientemente de la posición del puntero. Además, esta situación no produce cambios en la posición del puntero.
- ¿Has localizado en el ejemplo las situaciones en las que una falta de página provoca que el puntero recorra toda la estructura poniendo los bits de referencia a 0 para acabar sustituyendo la página apuntada inicialmente?

Algoritmos de reparto de espacio entre procesos

- Los procesos en general requieren disponer en memoria de un conjunto de páginas y este conjunto se mantiene más o menos constante durante un cierto intervalo de tiempo, después del cual se produce un cambio relevante en ese conjunto para volver a mantenerse más o menos constante. Ese conjunto se llama conjunto de trabajo (WS).
- La afirmación anterior es consecuencia del principio de localidad.
- Para conseguir un buen rendimiento es por tanto necesario intentar mantener en memoria principal las páginas del conjunto de trabajo de todos los procesos entre los que se reparte el tiempo de CPU.
- Se intenta evitar la hiperpaginación. Esta se produce cuando, a partir de cierto nivel de multiprogramación, el rendimiento del sistema cae en picado debido a que el coste de las faltas de página domina sobre el resto.

Ejemplo de aplicación de algoritmo basado en WS

- Proceso de 5 páginas. Se muestran las cargadas en memoria principal en cada instante.
- $X = 4$ Algoritmo WS mantiene en memoria solo las páginas que se han usado en los últimos 4 instantes.
- En un instante t , el conjunto de trabajo se considera formado por las páginas referenciadas en el intervalo $(t-X,t]$. Estas son las que se mantienen cargadas en memoria.

P. requerida	E	D	A	C	C	D	B	C	E	C	E	A	D
	-	-	A	A	A	A	-	-	-	-	-	A	A
	-	-	-	-	-	-	B	B	B	B	-	-	-
	-	-	-	C									
	-	D	D	D	D	D	D	D	D	-	-	-	D
	E	E	E	E	-	-	-	-	E	E	E	E	E

¿Falta de p.?

(1-4, 1]

{1}

(3-4, 3]

{1,2,3}

(6-4, 6]

{3,4,5,6}

- * Falta de página

Algoritmo FFP (frecuencia de falta de página)

Se utiliza la duración del intervalo de tiempo transcurrido entre faltas de página consecutivas para ajustar el conjunto de páginas residentes en memoria principal de un proceso.

- Si el tiempo transcurrido es mayor que Y , todas las páginas no referenciadas en dicho intervalo son retiradas de memoria principal.
- En otro caso, la página requerida es añadida al conjunto de páginas residentes en memoria principal.

Algoritmo FFP (cont.)

Formalmente

Sea:

- t_c instante de tiempo de la falta actual de página.
- t_{c-1} instante de tiempo de la falta de página anterior.
- Z es el conjunto de páginas referenciadas en un intervalo de tiempo.
- R Conjunto de páginas residentes en memoria principal.

$$R(t_c, Y) = \begin{cases} Z(t_{c-1}, t_c) & \text{si } t_c - t_{c-1} > Y \\ R(t_{c-1}, Y) + Z(t_c) & \text{en otro caso} \end{cases} \quad (1)$$

Ejemplo de aplicación de algoritmo FFP

- Proceso de 5 páginas. Se muestran las cargadas en memoria principal en cada instante.
 - E no está → falta (*). Tiempo desde la falta anterior = 1 ≤ Y. R = { A }
 - C ya está en memoria → SIN falta. R no cambia.
- $Y = 2$
 - @ B no está → falta de página. Tiempo desde la falta anterior: falta anterior en acceso 4, ahora estamos en 7 → 7-4 = 3 > Y
→ se marca con X en la tabla. Desde el acceso 4 hasta el 6 se usaron: C, C, D → conjunto { C, D }. Expulsamos las páginas que NO están en { C, D } → se van A y E.
Añadimos la página pedida B. R = { B, C, D }.

P. requerida	A	E	D	C	C	D	B	C	E	C	E	A	D
	A	A	A	A	A	A	-	-	-	-	-	A	A
	-	-	-	-	-	-	B	B	B	B	B	-	-
	-	-	-	C	C	C	C	C	C	C	C	C	C
	-	-	D	D	D	D	D	D	D	D	D	-	D
	-	E	E	E	E	E	-	-	E	E	E	E	E

¿Falta de p.?

*	*	*	*			X		*			X	*
---	---	---	---	--	--	---	--	---	--	--	---	---

->

@

- * El tiempo respecto a la falta de página anterior **no** es mayor que Y. Se añade la página requerida.
- X El tiempo respecto a la falta de página anterior es mayor que Y. Las páginas no referenciadas desde la falta de página anterior son retiradas.

Gestión de memoria en Linux

- El kernel utiliza paginación multinivel para la gestión de la memoria.
- Utiliza paginación por demanda y variable. Después de reservar memoria, al intentar escribir en ella se producirá una falta de página y se añade bajo demanda. La cantidad de páginas en memoria de cada proceso no es fija.
- La política de sustitución es global. Si es necesario eliminar páginas de memoria principal, las víctimas se seleccionan globalmente usando una estimación de su utilidad en el futuro.

Detalles sobre la paginación

- Actualmente se puede llegar hasta el nivel 5 de paginación motivado por el sucesivo incremento de la capacidad de almacenamiento en memoria principal.
- Existe hardware relativamente reciente que solo soporta 4 niveles (máximo original de la arquitectura x86-64). Esto limitaba a 64TB el tamaño máximo de la memoria principal y a 256TB el tamaño del espacio virtual.
- El paso a 5 niveles permite utilizar hasta 4PB de memoria principal y disponer de 128PB de espacio virtual.
- El tamaño por defecto de las páginas es 4KB pero el kernel soporta otros tamaños permitidos por el hardware (4KB, 64KB y 1GB en x86-64)

Paginación multinivel

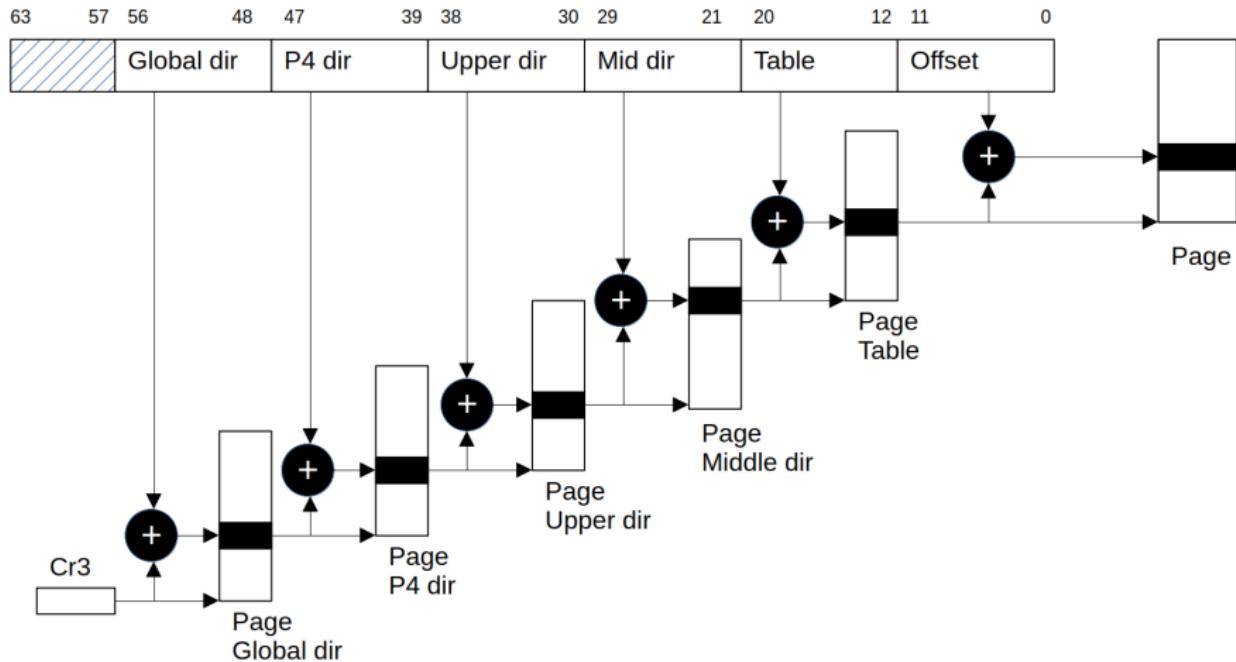


Figura: Paginación multinivel. 5 niveles

Gestión de memoria.

struct page

La estructura básica para representar los marcos de página es la *struct page*. Alguno de sus campos más relevantes:

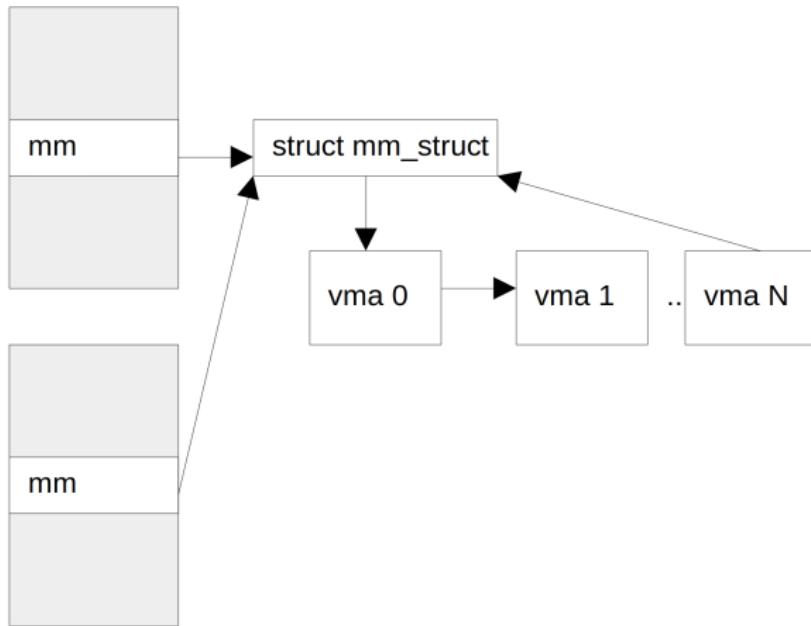
- unsigned long flags; // PG_dirty, PG_locked
- atomic_t _count; // cantidad de referencias a la página
- struct address_space *mapping;

Para páginas en la caché de páginas, referencia a la información necesaria para acceder al fichero asociado a esa página. En otro caso, si es una página anónima, referencia a una estructura *anon_vma* desde la que se puede llegar a la tabla de páginas de la página almacenada. Si es una página compartida, referencia a una lista de estructuras *anon_vma* para y así poder obtener todas las tablas de páginas que la incluyen.

Gestión de memoria. VMAs

Las regiones de memoria se representan mediante VMAs (Virtual Memory Area)

task struct



Gestión de páginas

- El kernel mantiene 5 listas de páginas de memoria que gestiona con una política pseudo-LRU.
- Se distingue entre:
 - ▶ Páginas asociadas a ficheros: se utilizan como cache de información almacenada en disco.
 - ▶ Páginas anónimas: no están asociadas a ficheros y son resultado de reservas de memoria realizadas dinámicamente como parte del ciclo de vida de la mayor parte de las VMAs (pila, heap, etc.)
- Estos dos tipos de páginas se mantienen almacenadas por separado siendo las asociadas a datos de ficheros las que tienen más probabilidad de ser desalojadas de memoria si es necesario liberar memoria principal. De entre estas se elegirán primero las *limpias* (no modificadas desde su carga) frente a las *sucias* (modificadas después de ser cargadas).

Gestión de páginas cont. (*inactive list vs. active list*)

- Para implementar la política LRU sin incurrir en costes computacionales demasiado altos, se recurre al uso de dos tipos de listas para cada tipo de página.
 - ▶ Active list: en ella se intenta mantener las páginas que están siendo más usadas. Esta lista contiene las páginas que en principio no se consideran candidatas a ser desalojadas de memoria.
 - ▶ Inactive list: en ella se intenta mantener las páginas raramente usadas y que por lo tanto son candidatas a ser eliminadas de memoria principal si es necesario.
- Las páginas con mayor probabilidad de ser eliminadas de memoria principal son las de la cola de la *inactive list* de las páginas asociadas a ficheros.
- Se intenta mantener una relación correcta entre la longitud de ambos tipos de listas. Por ejemplo, si la *active list* crece demasiado, el contenido de parte de la cola de la misma será trasvasado a la *inactive list*.

Gestión de páginas cont. (gestión de las listas)

- La gestión de la *active list* e *inactive list* de cada tipo de página es distinta según este tipo y depende también de la versión del kernel.
 - ▶ Las páginas anónimas recién creadas pueden insertarse en la cabecera de la *inactive list* y esperar a que promocionen a la *active list* en función de su uso o ser insertadas directamente en la cabecera de la *active list*. Las páginas de la caché de páginas se insertan en su *inactive list* correspondiente.
 - ▶ La gestión de la *active list* puede implicar que se intente comprobar si siguen siendo realmente accedidas con frecuencia o dejar que se vayan dirigiendo de la cabecera a la cola y acaben en la *inactive list* simplemente por envejecimiento (tiempo en la lista). En este último caso, si siguen siendo accedidas con frecuencia volverán a la *active list*.

Gestión de páginas cont. (gestión de las listas)

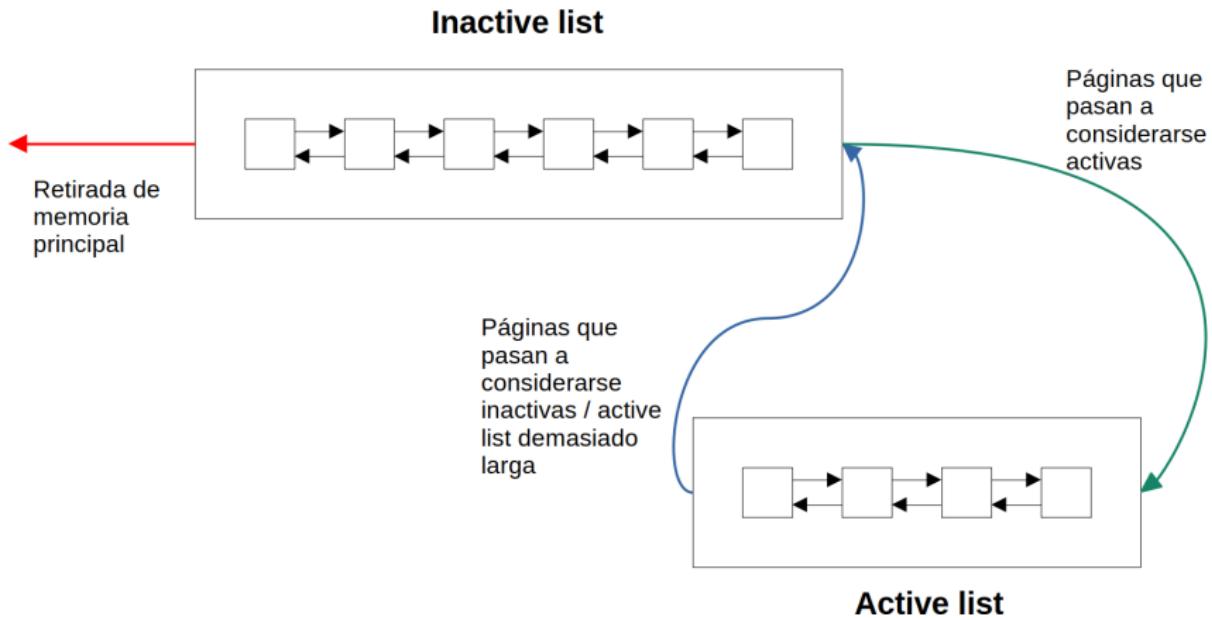


Figura: Gestión de páginas de memoria

Gestión de páginas cont. (páginas no eliminables)

- Existe también una lista de páginas no eliminables de memoria (*unevictable*) por distintos motivos (Ejemplo: ramfs o memoria compartida fijada con SHM_LOCK).
- Entre estas páginas están las que se han generado como resultado de la llamada a la función *mlock()* que permite a procesos reservar memoria que nunca será desplazada de memoria principal.

Gestión de páginas cont. (estimar la actividad)

- A nivel hardware suele existir soporte para un bit que indica que si una página ha sido referenciada. Este bit se activa por parte de la MMU al acceder a una página
- Este bit se va poniendo a 0, independientemente del valor que tenga, con cierta periodicidad por el sistema operativo.
- Si una página es referenciada y ya tiene ese bit con el valor 1, quiere decir que ha sido referenciada al menos una vez (o más) entre dos ejecuciones de la tarea que pone ese bit a 0. Esto es un indicativo que la página se usa con frecuencia.
- En un caso como el anterior, si la página está en la *inactive list*, pasaría a la *active list*.
- Este esquema utiliza recursos hardware generalmente disponibles y no requiere el uso de temporizadores que controlen de forma exacta el tiempo transcurrido desde el último uso de una página.

Gestión de páginas cont.

`cat /proc/meminfo`

...

Active:	3980440 kB	Active(anon)	+	Active(file)
Inactive:	1322744 kB	Inactive(anon)	+	Inactive(file)
Active(anon):	2406172 kB			
Inactive(anon):	0 kB			
Active(file):	1574268 kB			
Inactive(file):	1322744 kB			
Unevictable:	2246096 kB			
Mlocked:	32 kB			

...

Slab layer

Caché de estructuras de datos

- Esta caché existe para optimizar el proceso de solicitud y liberación de estructuras de datos de uso frecuente.
- La continua reserva y liberación en memoria de estructuras de uso frecuente supone un añadido de carga.
- Ejemplos de dichas estructuras son: *task struct* y los inodos.
- Existe por tanto una caché para cada tipo de estructura de datos.
- Estas caches almacenan bloques de estructuras, pudiendo estar cada bloque lleno, vacío o parcialmente lleno.
- Cuando el kernel requiere una de estas estructuras, se intenta obtener de un bloque parcial. Si no existe se recurre a un bloque vacío. Si no existiera ninguno se crearían nuevos.

Bibliografía

- Carretero Pérez Jesús, García Carballeira Félix, Pérez Costoya Fernando. Sistemas Operativos: Una visión Aplicada, Volúmen II. 3^a edición. 2021.
- Russinovich Mark E., Allievi Andrea, Ionescu Alex, Solomon David A. Windows Internals, Part 2 (Developer Reference). 7^a edición. Microsoft Press. 2021.
- Stallings William. Operating Systems: Internals and Design Principles, Global Edition. 9^a edición. Pearson-prentice Hall. 2017
- Silberschatz Abraham, Galvin Peter B., Gagne Greg. Silberschatz's Operating System Concepts, Global Edition. 10^a edición. John Wiley & Sons Inc. 2019

Tema 3: Gestión de memoria

Profesorado de Sistemas Operativos

Universidad de Granada

Sistemas Operativos

Grado de Ingeniería Informática

Grado en Ingeniería Informática y Matemáticas

Grado en Ingeniería Informática y Administración y Dirección de Empresas

2025-2026