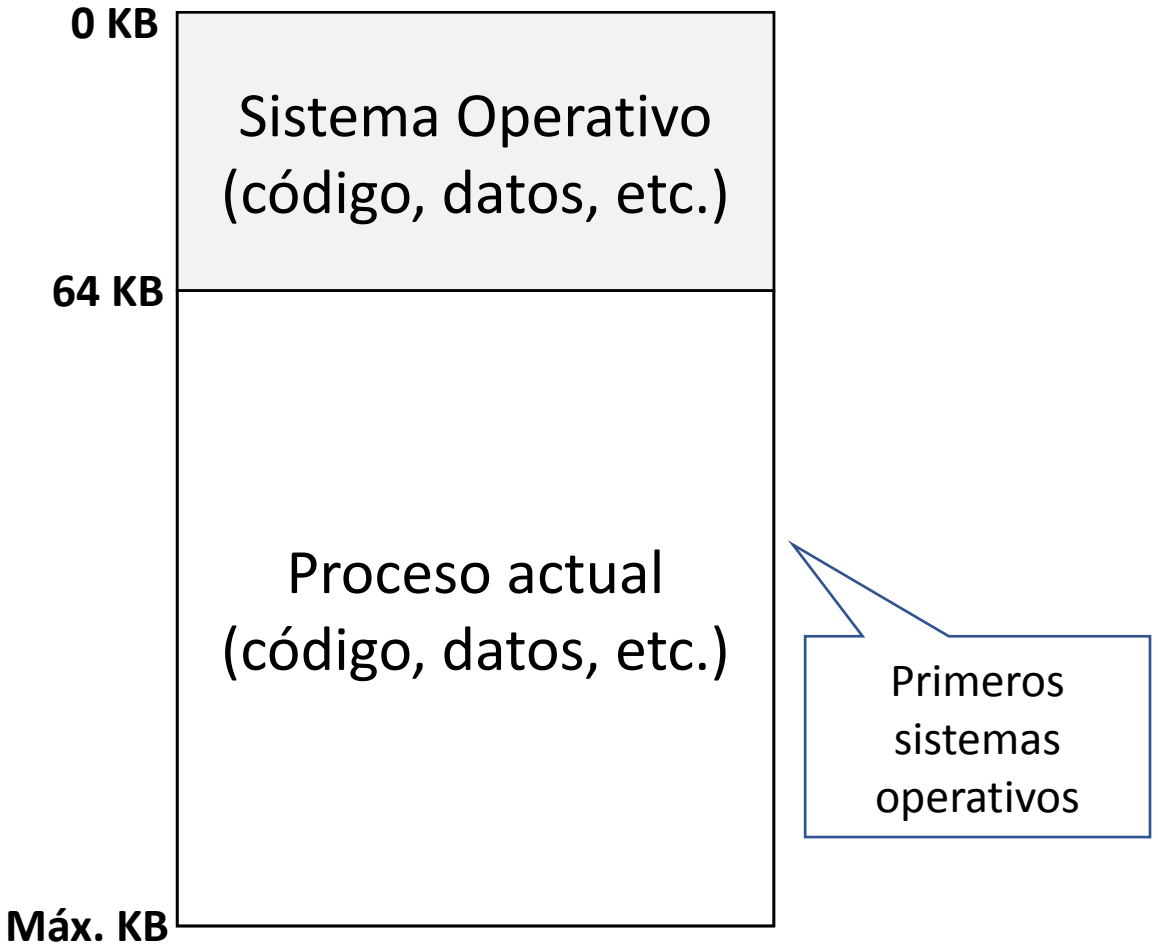
Administración de memoria

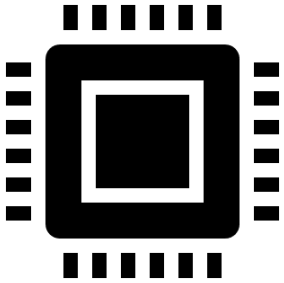
Adaptación (ver referencias al final)

Monotarea y multiprogramación



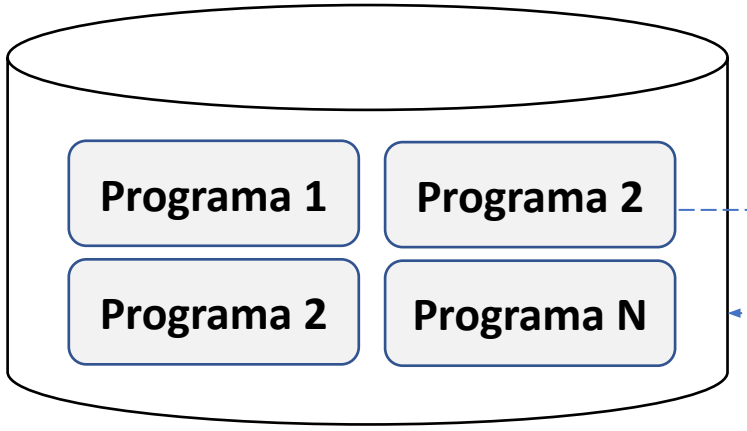
- Maximizar el uso del sistema
  - Varios procesos concurrentemente
- CPU compartida (en el tiempo) por varios procesos.
- Memoria principal (RAM) compartida (en el espacio) por varios procesos.
- ¿Qué implica compartir la memoria?
  - Gestión de memoria

1. Fetch
2. Decode
3. Execute



PC: 0x100

Apunta a la siguiente instrucción (en memoria) a ejecutar



Cargar en memoria para ejecutar

Expulsar de memoria

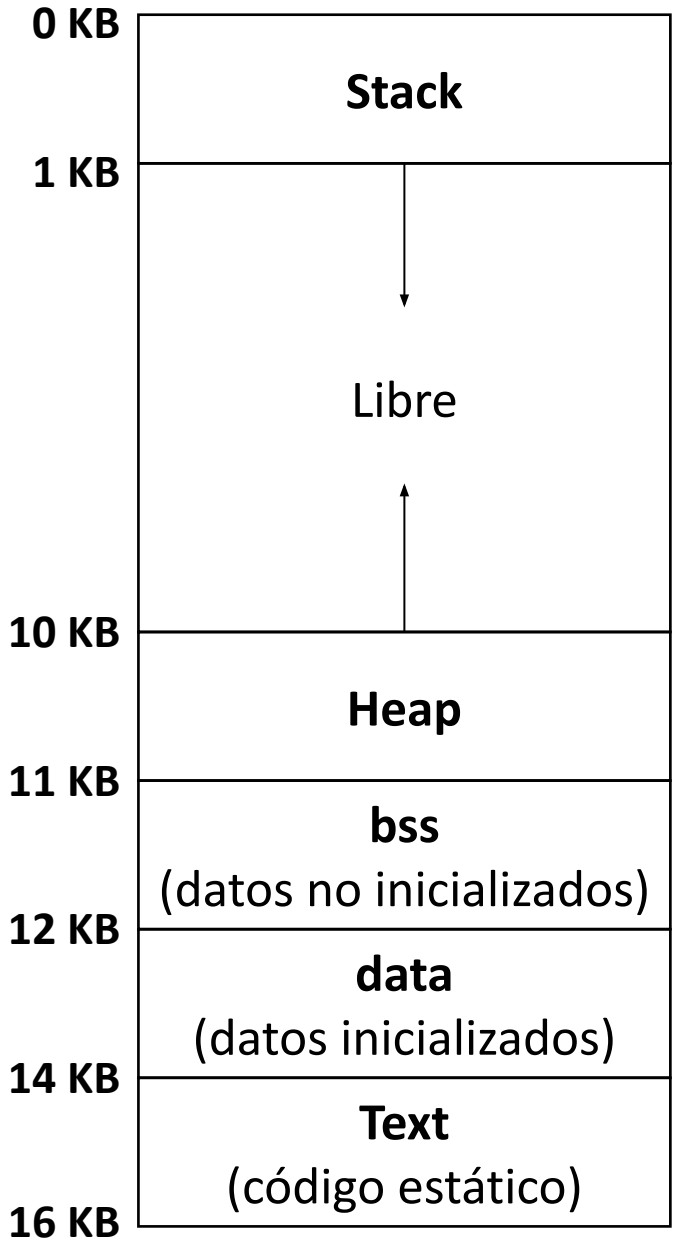
0 KB
Sistema Operativo
(código, datos, etc.)
64 KB
100 KB
MOV EAX, [0x70]
Proceso actual
(código, datos, etc.)
Máx. KB

¿Qué implicaciones tiene la expulsión?

Programas residen en disco (entidades estáticas)
Se ejecutan cuando se pueden cargar en memoria RAM

Espacio de direccionamiento

- Abstracción que crea el S.O
- Arreglo de bytes direccionables
- La abstracción da la idea de que el mapa de memoria inicia en la dirección 0 KB y termina en la dirección 16 KB
  - Físicamente esto no sucede así
- El proceso “cree” que ocupa toda la memoria disponible (16 KB) del sistema



loop.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    while(1) {
       printf("Infinito\n");
    }
    return 0;
}
```

hello.c

```
#include <stdio.h>

int main (int arcg, char* argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

 # objdump -d -M i386 loop > loop.s

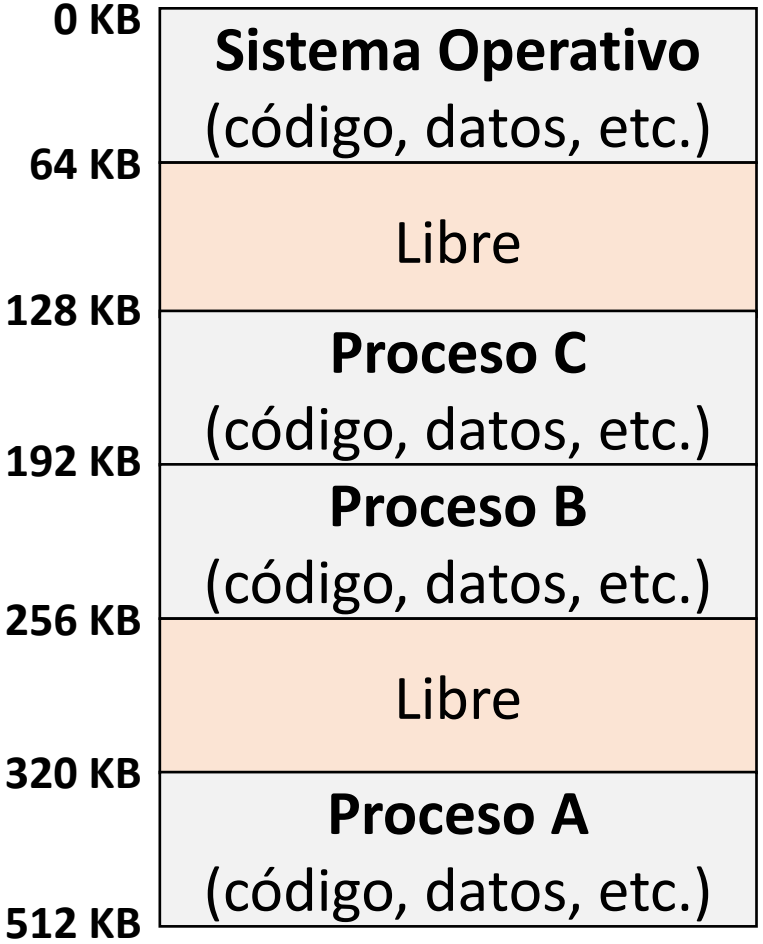
```
000000000040052d <main>:
  40052d: 55       push   %ebp
  40052e: 48       dec    %eax
  40052f: 89 e5    mov    %esp,%ebp
  400531: 48       dec    %eax
```

 # objdump -d -M i386 hello > hello.s

```
000000000040052d <main>:
  40052d: 55       push   %ebp
  40052e: 48       dec    %eax
  40052f: 89 e5    mov    %esp,%ebp
  400531: 48       dec    %eax
```

¿Misma dirección de memoria?
¿Qué sucede si los ejecuto a la vez? ¿Se solapan?

Multiprogramación



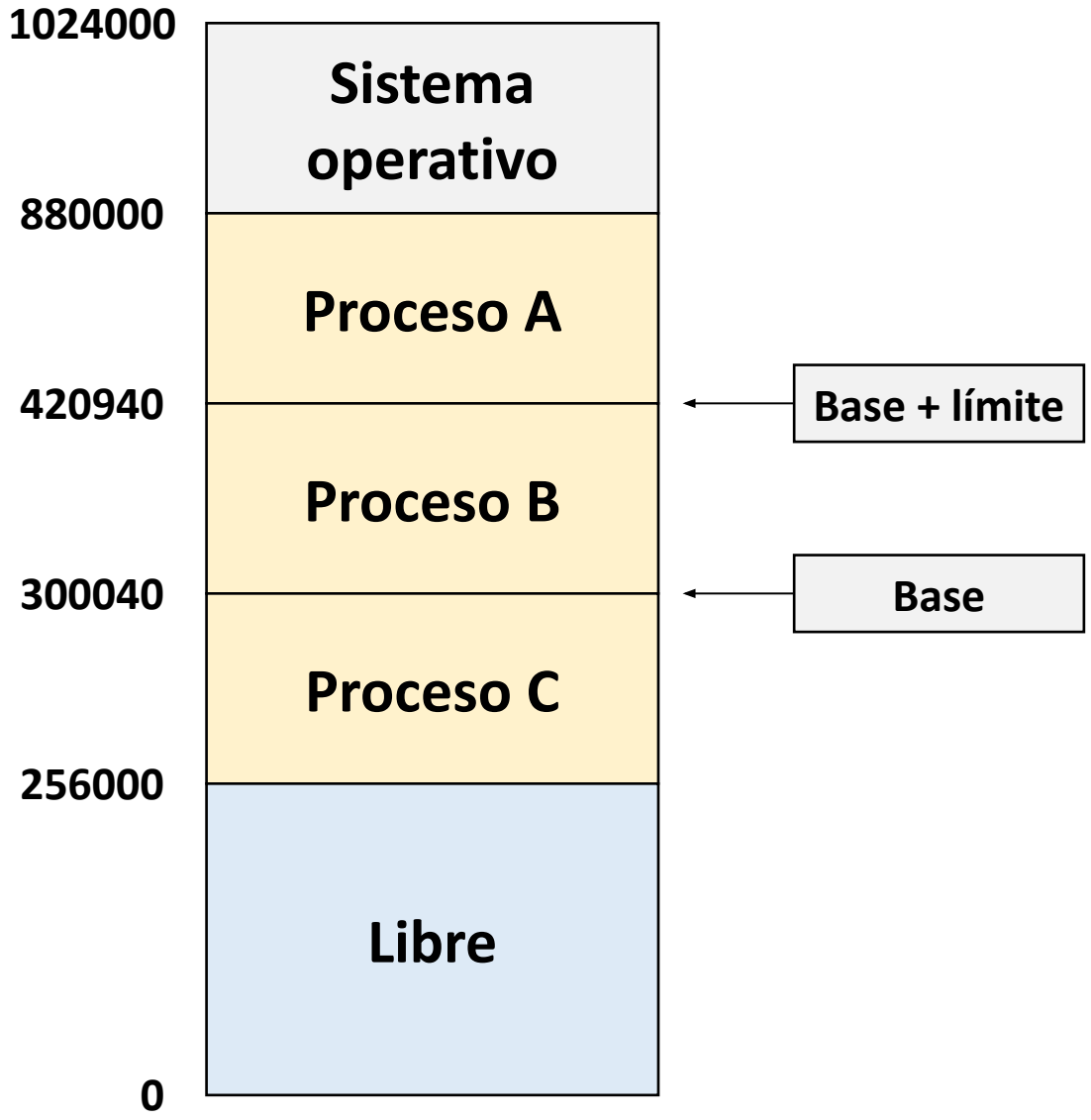
- Los procesos “creen” que tienen una máquina para ellos solos
  - CPU
  - RAM
- S.O debe permitir coexistencia **segura** de múltiples procesos en RAM
- **Físicamente** los procesos están en espacios de memoria diferentes

Objetivos de la gestión de memoria

- Ofrecer a cada proceso un espacio de direccionamiento lógico propio
  - Proceso se comporta como su propio espacio físico de direccionamiento
  - Separar los espacios de memoria de cada proceso
- Proporcionar protección entre los procesos
  - Operaciones de procesos NO pueden incidir en el espacio de direccionamiento de otros procesos
- Eficiencia
  - Tiempo: NO introducir retardos en la ejecución de los programas
  - Espacio: NO usar muchas estructuras de datos para la gestión de la memoria de cada proceso.
- **Para estos objetivos se requiere apoyo del hardware (MMU)**

Hardware

- El S.O no interviene en las operaciones de acceso de la CPU a memoria.
- Se necesita determinar el rango de direcciones válidas de cada proceso.
  - Operaciones de cada proceso deben hacer referencia al rango válido de su espacio de direccionamiento



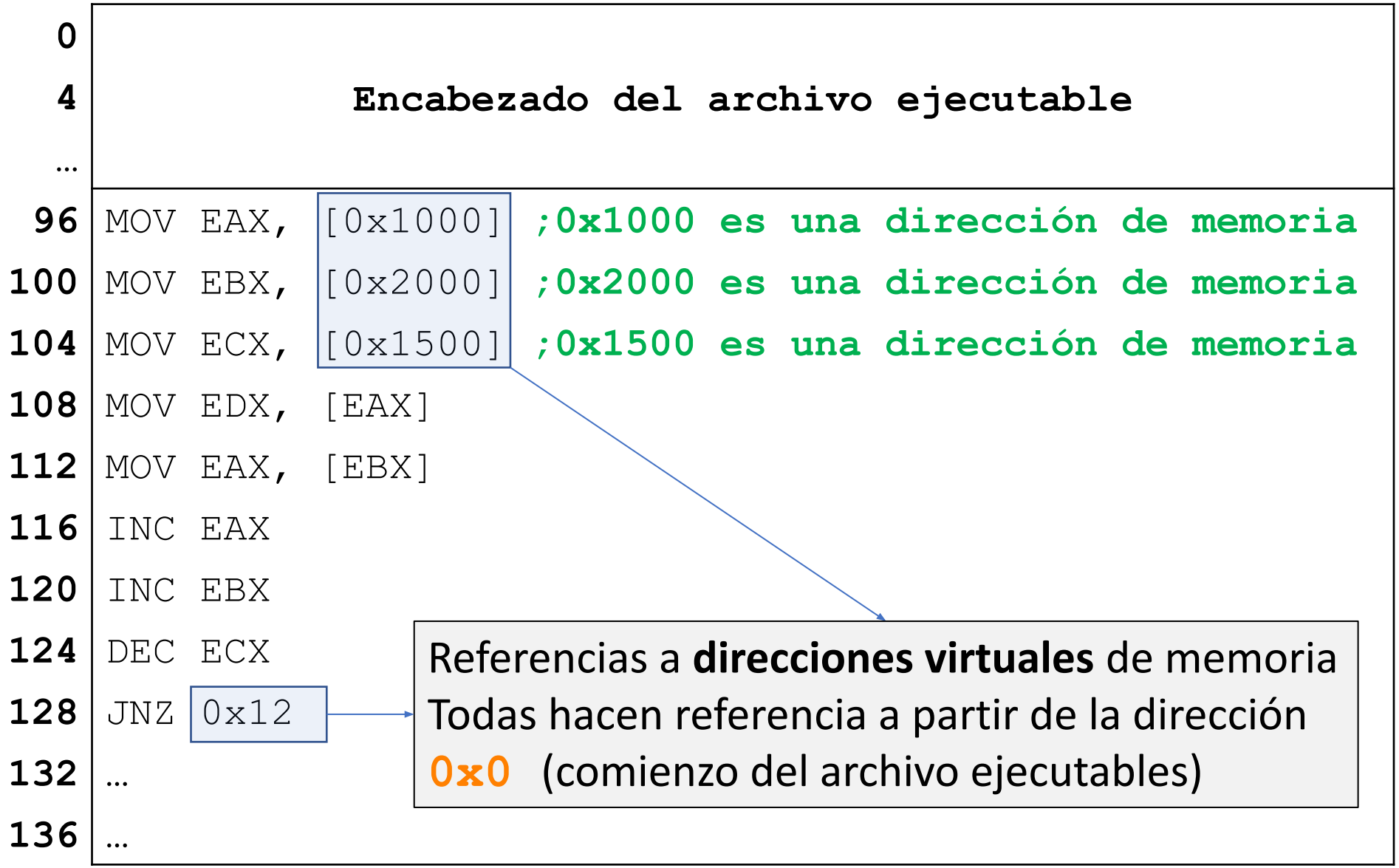
Hardware

- Hardware **MMU** (Memory Management Unit)
- Registros están implementados a nivel de hardware.
  - S.O es el único que tiene acceso a los registros.
- Registros controlan el espacio físico de direcciones.
- La protección la realiza la MMU comparando cada dirección generada en el espacio de usuario con los valores de los registros.

  - **Espacio de memoria virtual**
  - **Espacio de memoria física**

Traducción del direccionamiento

- En sistemas multiprogramados no se puede conocer ***a priori*** en donde será ubicado el proceso en memoria
  - Depende de la ocupación de memoria
- S.O.'s modernos permiten que los proceso de usuario residan en cualquier parte de la memoria física
  - La dirección `0x40052d` no necesariamente corresponde con la dirección física.
  - La dirección `0x40052d` es una **dirección virtual**.
- Direcciones en código fuente usualmente son simbólicas
  - `int i = 0;`

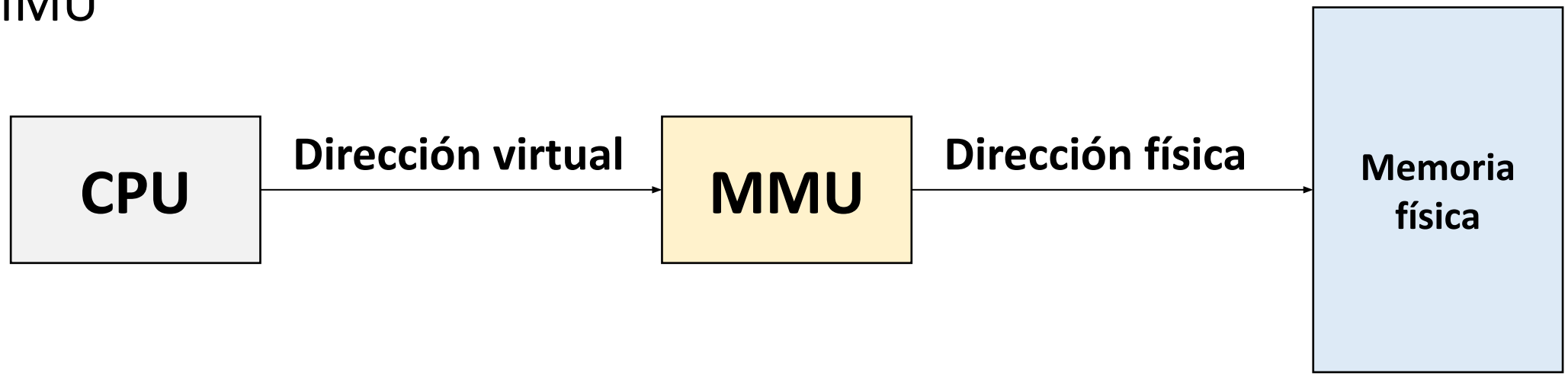

0
4
…
Encabezado del archivo ejecutable
96 MOV EAX, [0x1000] ;0x1000 es una dirección de memoria
100 MOV EBX, [0x2000] ;0x2000 es una dirección de memoria
104 MOV ECX, [0x1500] ;0x1500 es una dirección de memoria
108 MOV EDX, [EAX]
112 MOV EAX, [EBX]
116 INC EAX
120 INC EBX
124 DEC ECX
128 JNZ 0x12
132 …
136 …
Referencias a **direcciones virtuales** de memoria
Todas hacen referencia a partir de la dirección
0x0 (comienzo del archivo ejecutables)

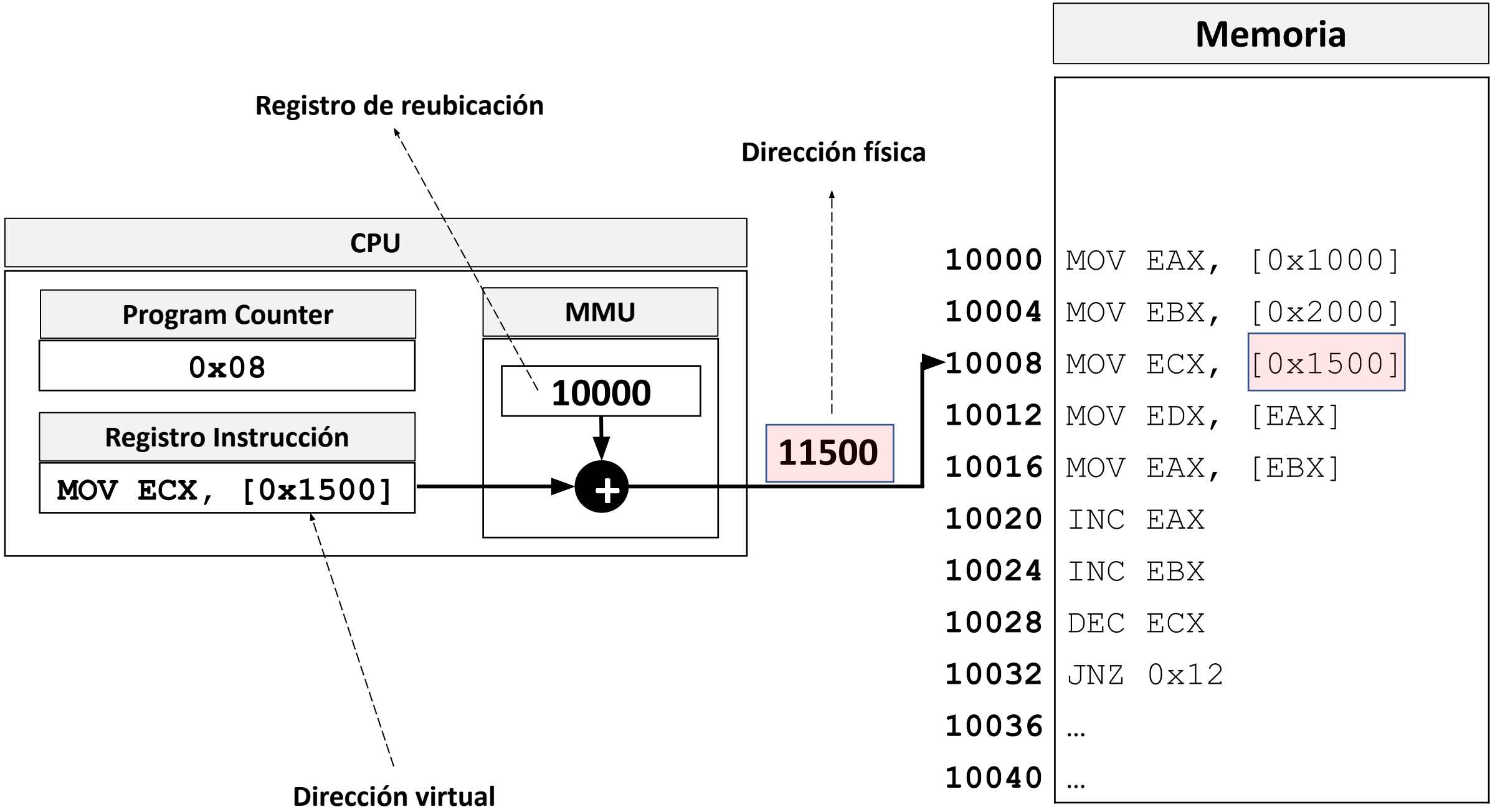
Traducción del direccionamiento

- Tiempo de compilación
  - Si se sabe a *a priori* donde va a estar el proceso, se genera código absoluto
  - Si la ubicación del proceso cambia, hay que recompilar
  - No hay traducción/mapeo: virtual corresponde con el físico.
- Tiempo de carga
  - Se debe generar código reubicable.
  - Si ubicación del proceso cambia se carga de nuevo el código para reflejar el cambio
- Tiempo de ejecución
  - La mayoría de S.O.'s modernos usan esta opción. Se necesita de la MMU
  - El mapeo/traducción no se hace hasta la ejecución del proceso.

Espacio de direccionamiento físico y virtual

- Direcciones lógicas/virtuales
  - Direcciones de memoria generadas por la CPU
- Direcciones físicas
  - Direcciones de memoria que ve la MMU
- La traducción virtual física se hace en tiempo de ejecución por la MMU




Memoria
Registro de reubicación
Dirección física
CPU
Program Counter
0x08
MMU
10000
Registro Instrucción
MOV ECX, [0x1500]
11500
10000 MOV EAX, [0x1000]
10004 MOV EBX, [0x2000]
10008 MOV ECX, [0x1500]
10012 MOV EDX, [EAX]
10016 MOV EAX, [EBX]
10020 INC EAX
10024 INC EBX
10028 DEC ECX
10032 JNZ 0x12
10036 …
10040 …
Dirección virtual

Demostración con gdb

- gdb <ejecutable>
- set disassembly-flavor intel
- layout next
- break main
- break *main+22
- break *0x40054d
- clear *0x40054d
- run
- info registers
- p/x $eax
- x/1dw 0x400535
- x/1dw $rbp
- info break
- ni
- si

Referencias

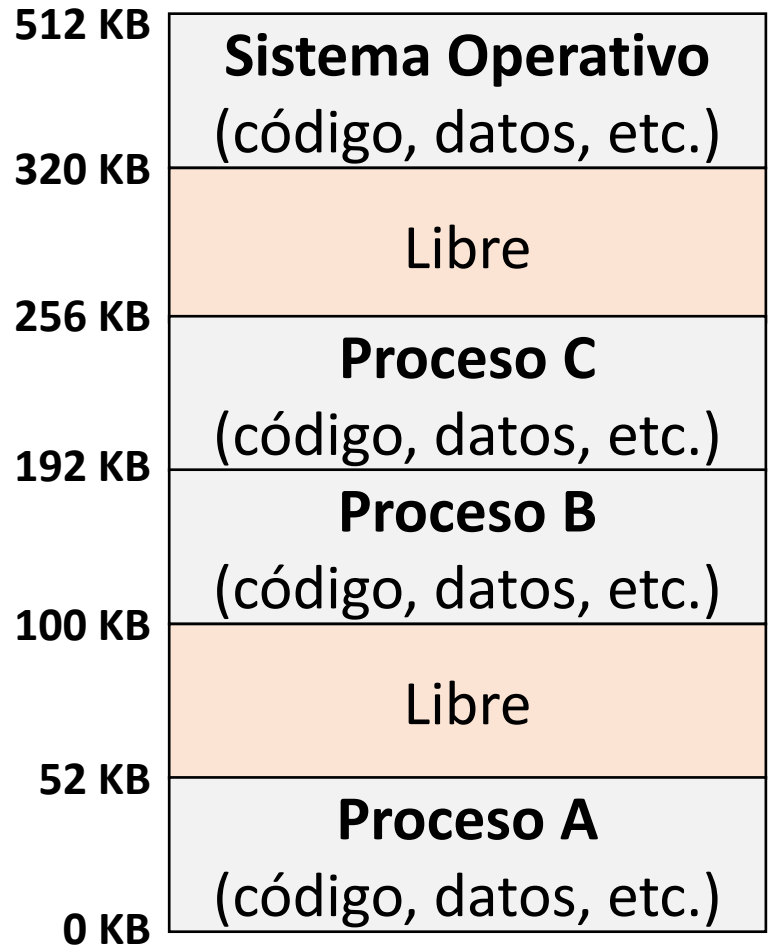
- Carretero Pérez, J., García Carballeíra, F., De Miguel Anasagasti, P., & Pérez Costoya, F. (2001). Gestión de memoria. In *Sistemas operativos. Una Visión Aplicada* (pp. 164–171). McGraw Hill.
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Main Memory. In *Operating Systems Concepts* (10th ed., pp. 349–356). John Wiley & Sons, Inc.

Asignación contigua

Adaptación (ver referencias al final)

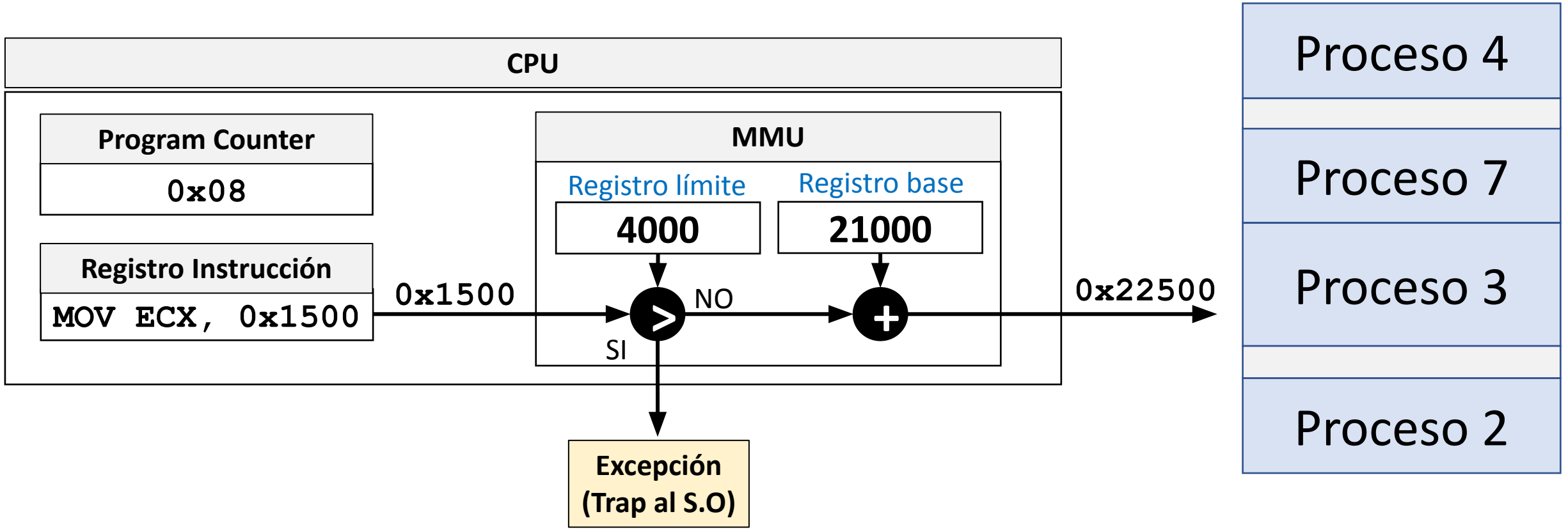
Asignación contigua

- Se necesita asignar memoria a los procesos y S.O de la mejor manera posible.
- Usualmente la memoria tiene dos grandes particiones
  - Partición para el S.O (usualmente direcciones altas): Linux, Windows
  - Partición para para los procesos
- Se requiere que los procesos residan en memoria

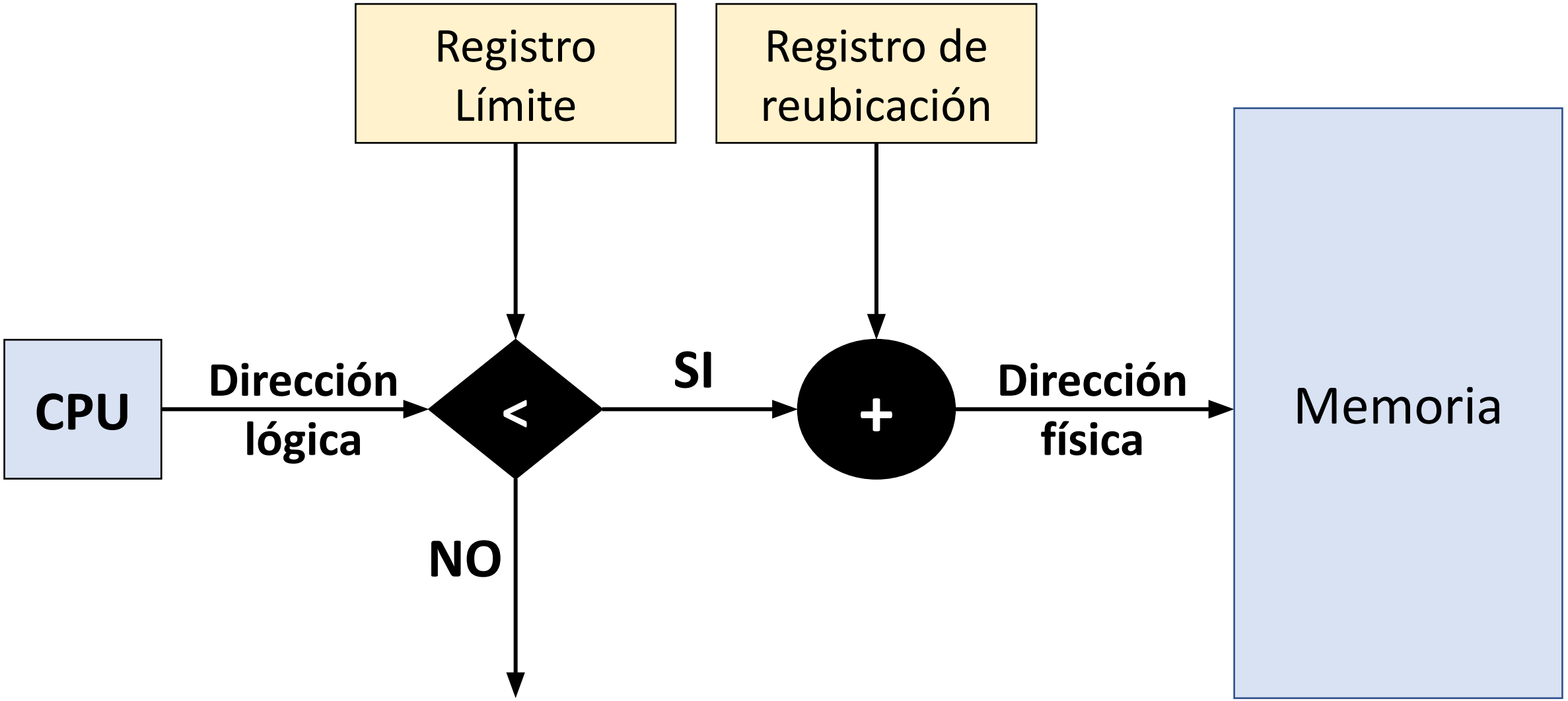


Protección

- **Registro límite:** comprueba que cada dirección virtual NO es mayor que el valor de este registro.
- **Registro base:** Si dirección virtual no supera el límite, se suma a la dirección el valor del registro base para obtener la dirección física.



- **Registro base** y **registro límite** se pueden acceder solo en modo privilegiado.

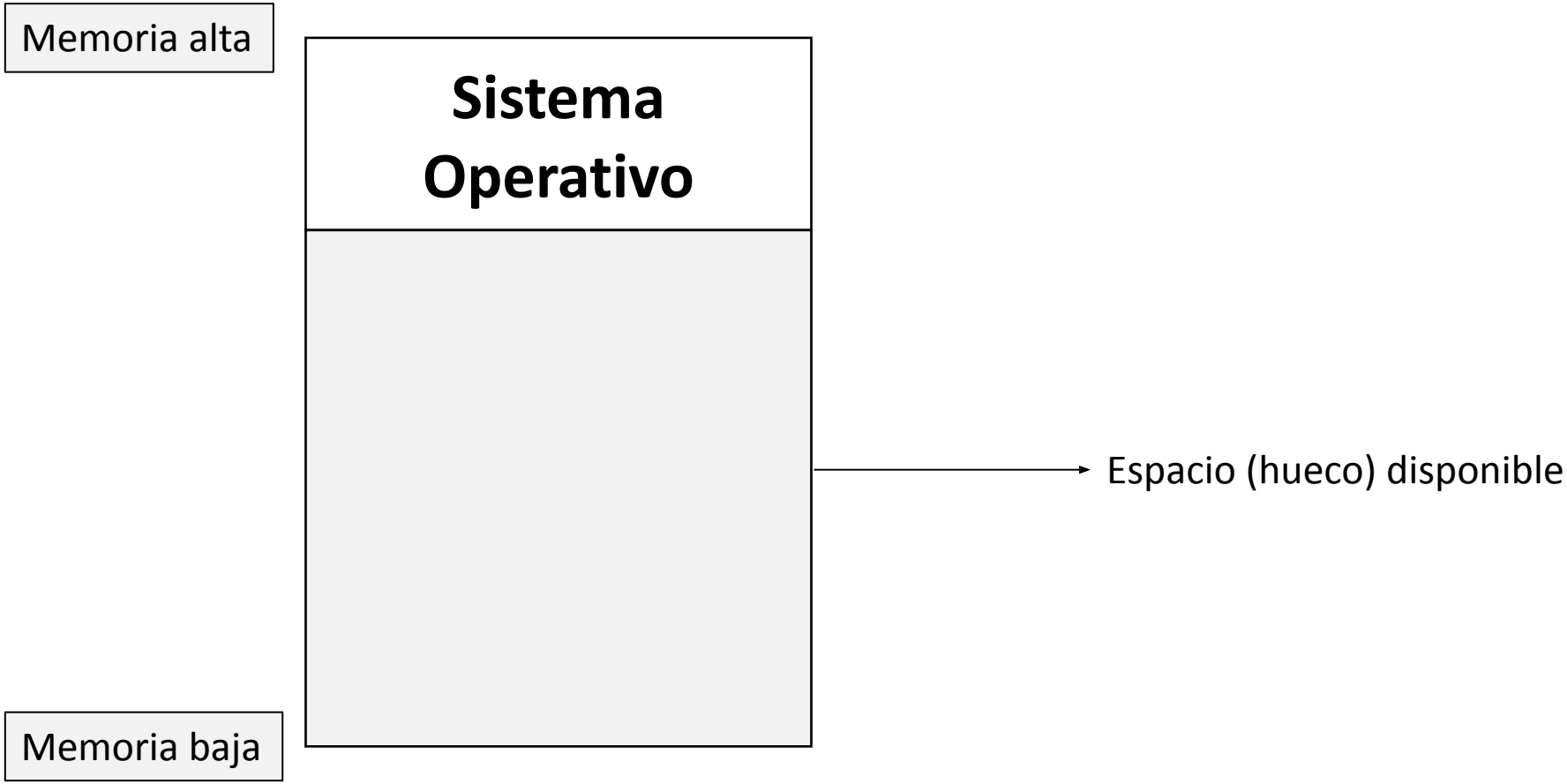


- S.O almacena en el PCB los valores de ambos registros para cada proceso
- En cada cambio de contexto se deben volver a cargar los registros

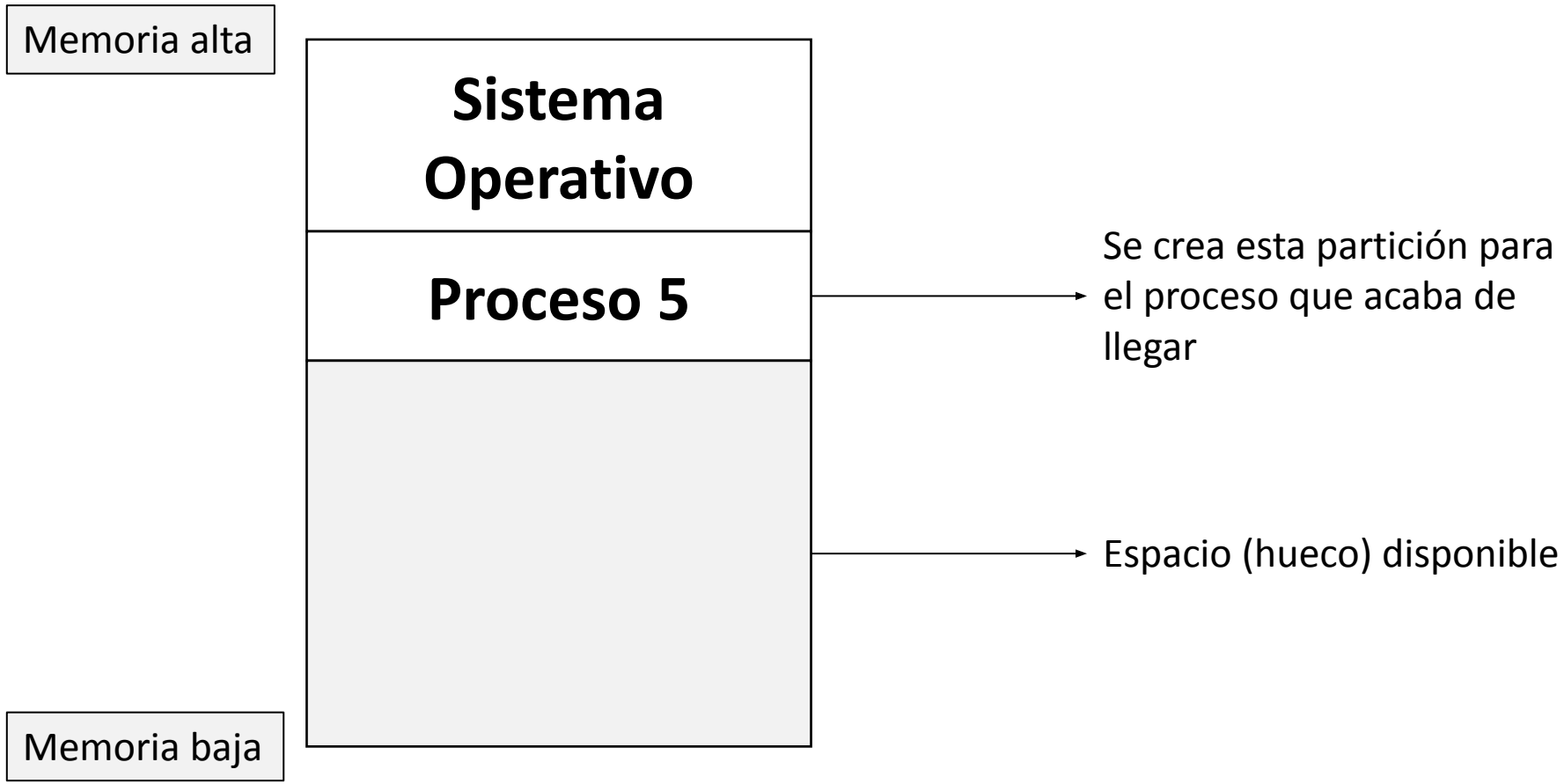
Asignación contigua

- Cada proceso se asigna a una partición de memoria
- Cada partición (de tamaño variable) es asignada a un solo proceso
  - Particiones variables crean problema de fragmentación (luego…)
- S.O debe llevar registro de:
  - Espacios/particiones de memoria están ocupados (asignados a los procesos)
  - Espacios/particiones (huecos) de memoria disponible

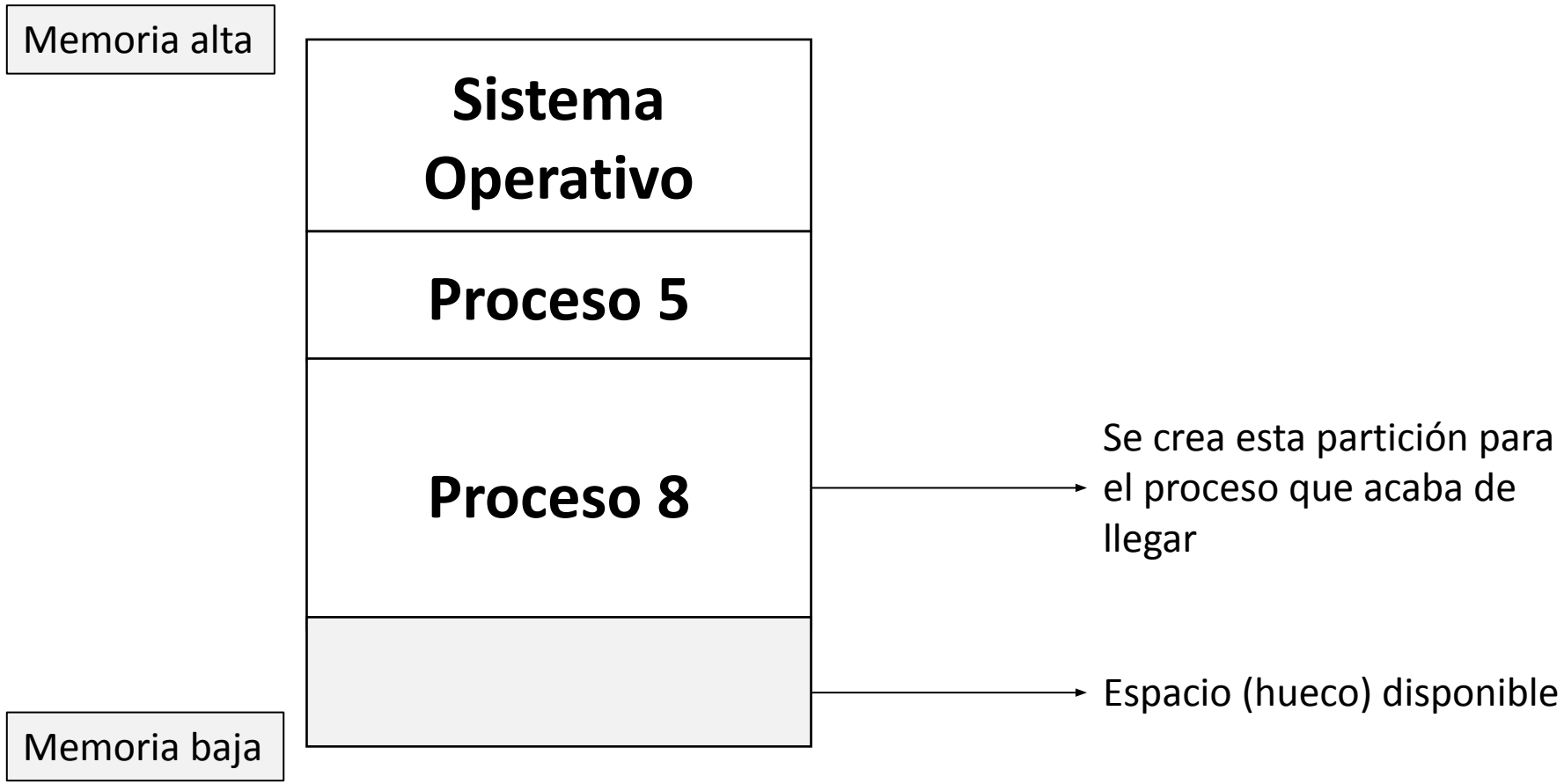
Asignación contigua



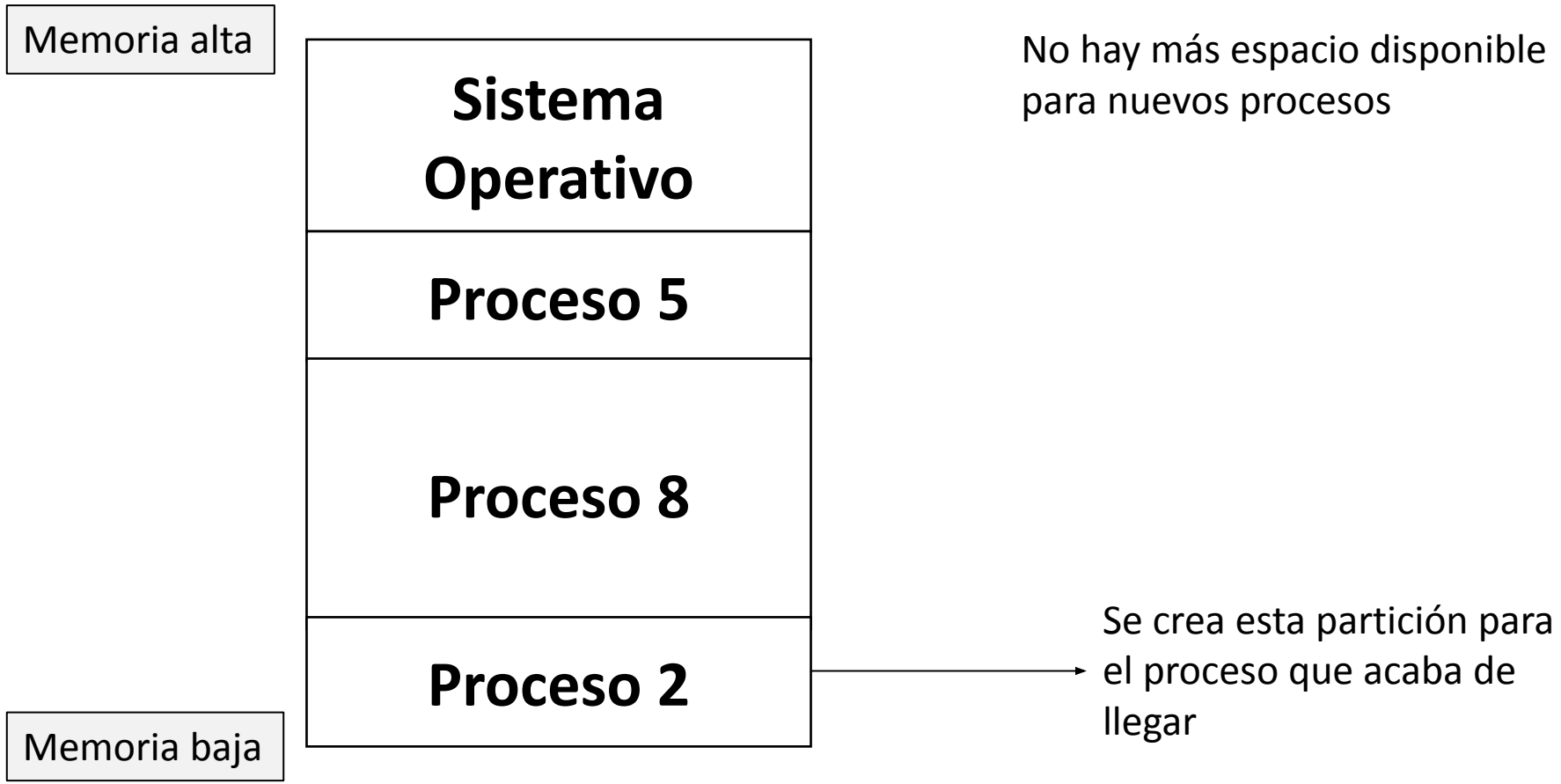
Asignación contigua



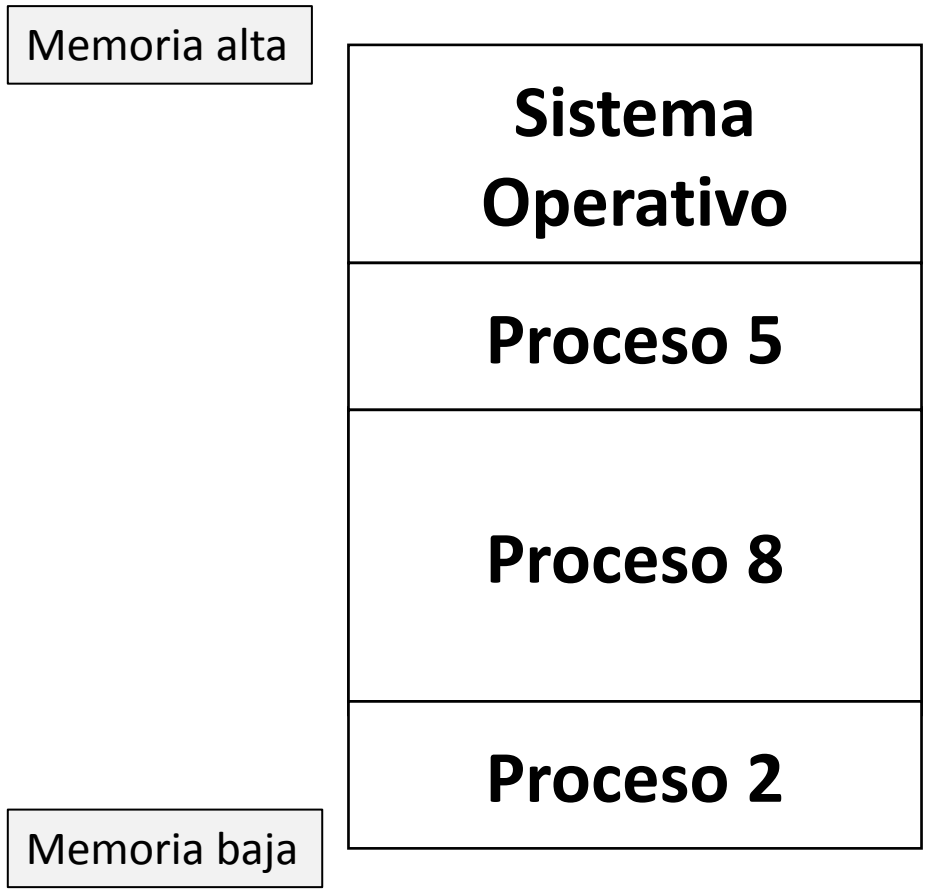
Asignación contigua



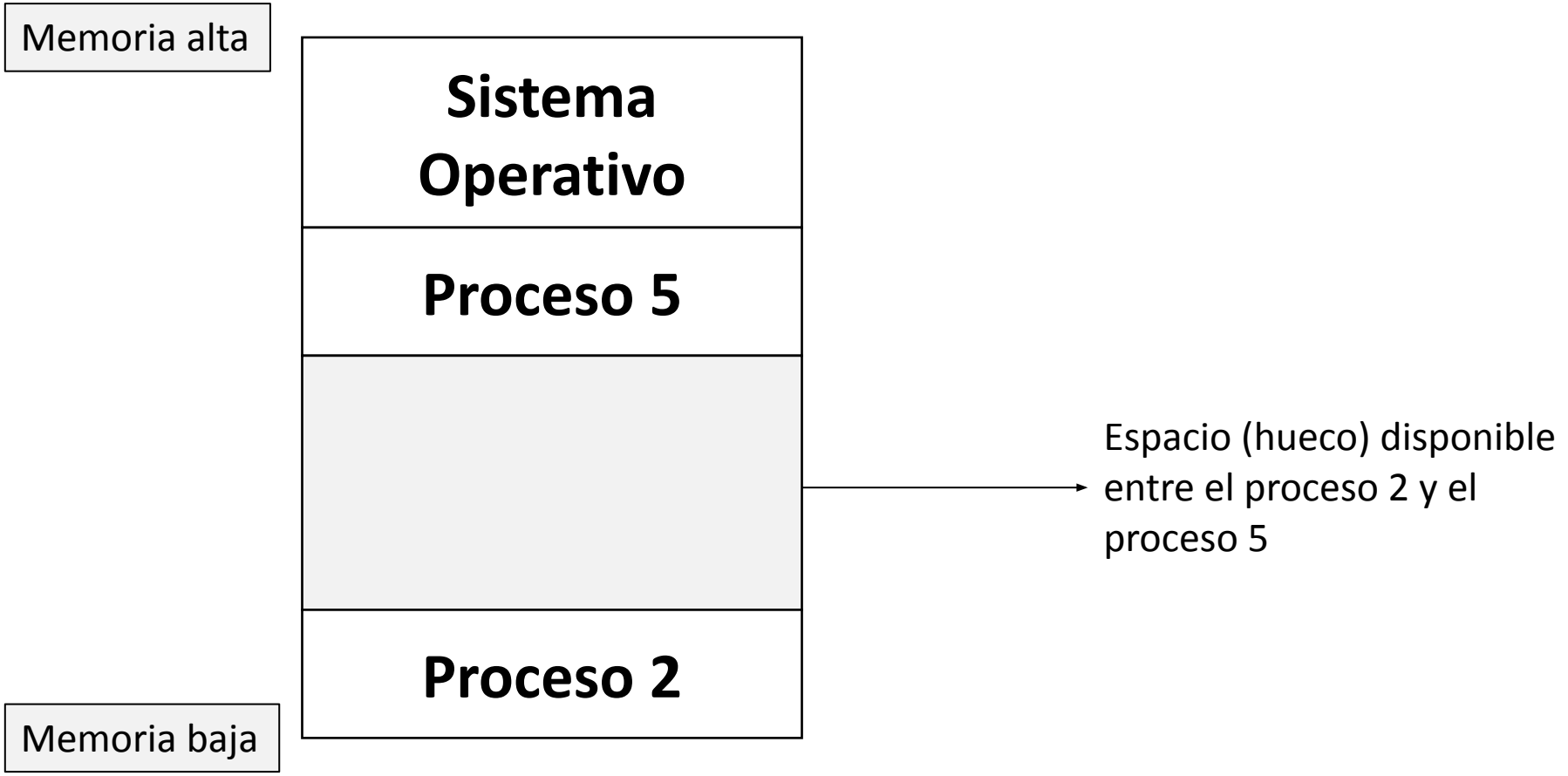
Asignación contigua



Asignación contigua



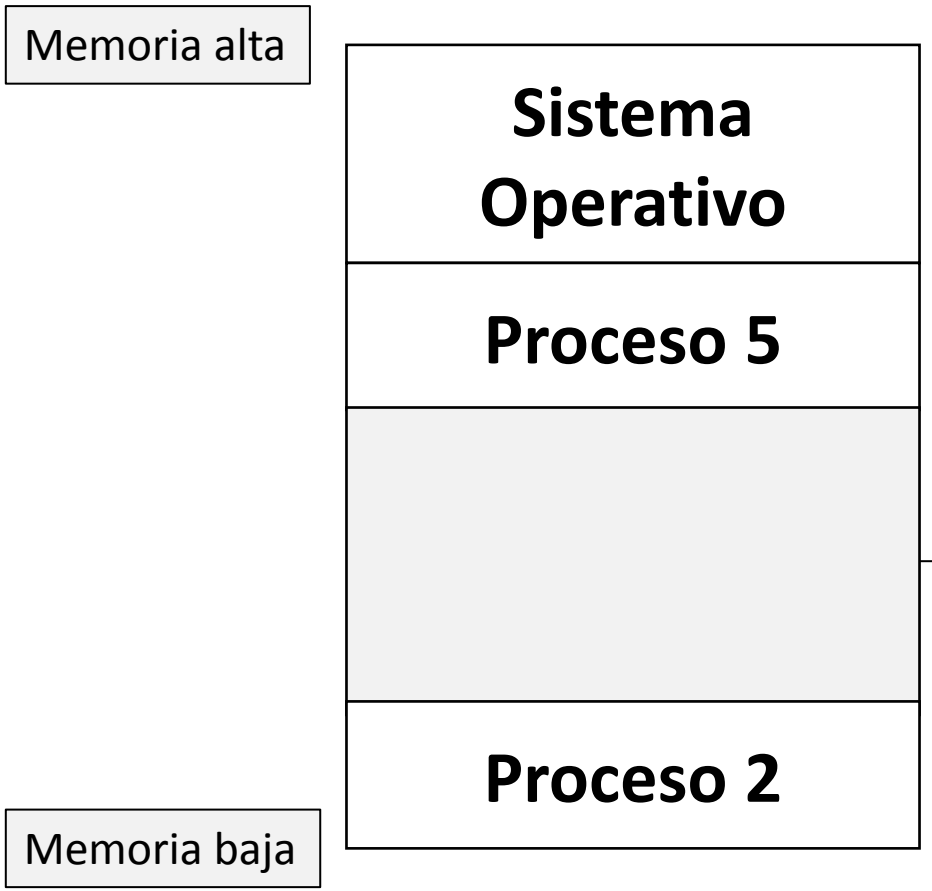
Asignación contigua



Asignación contigua

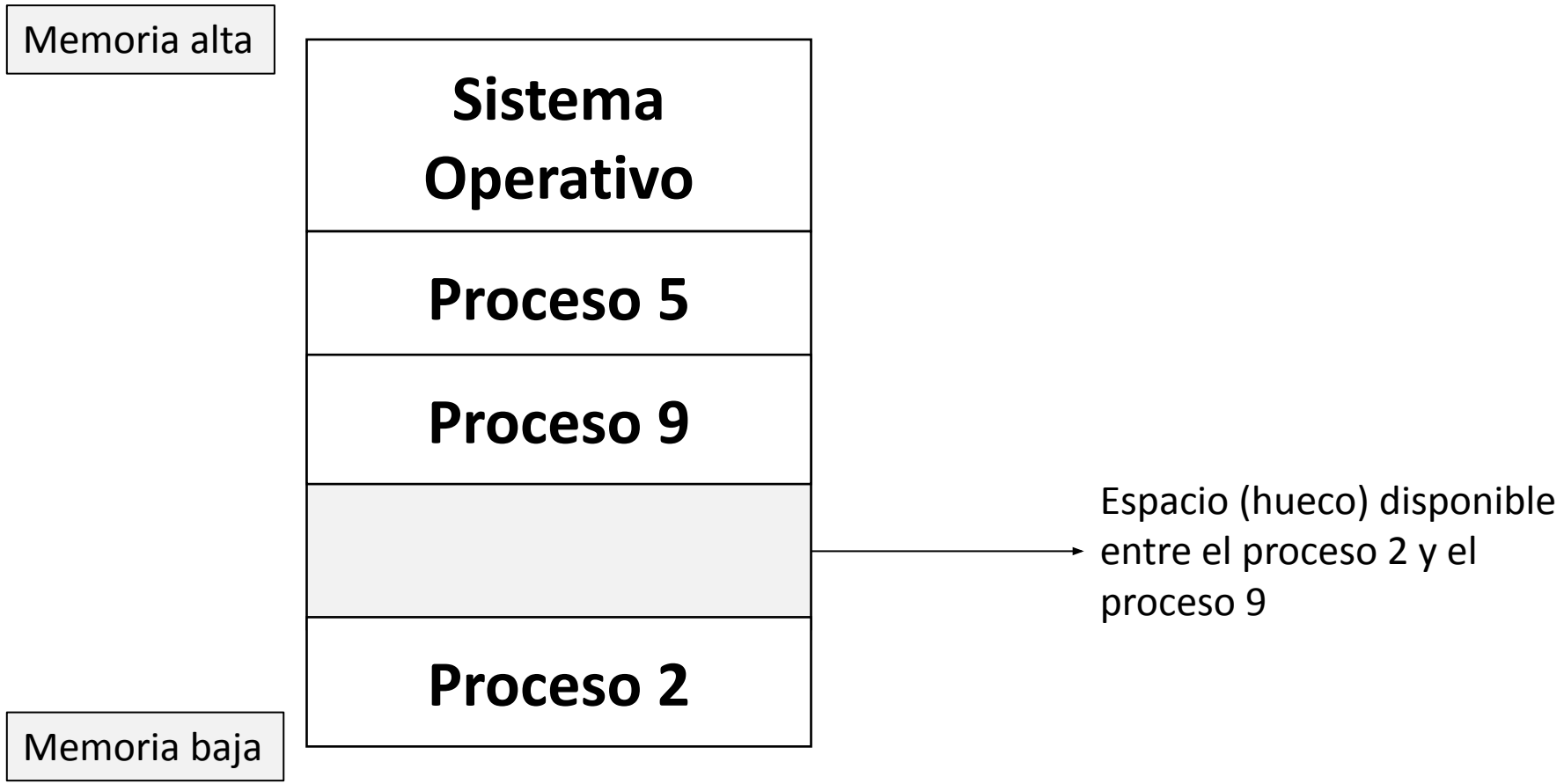
Proceso 9

Llega este nuevo proceso

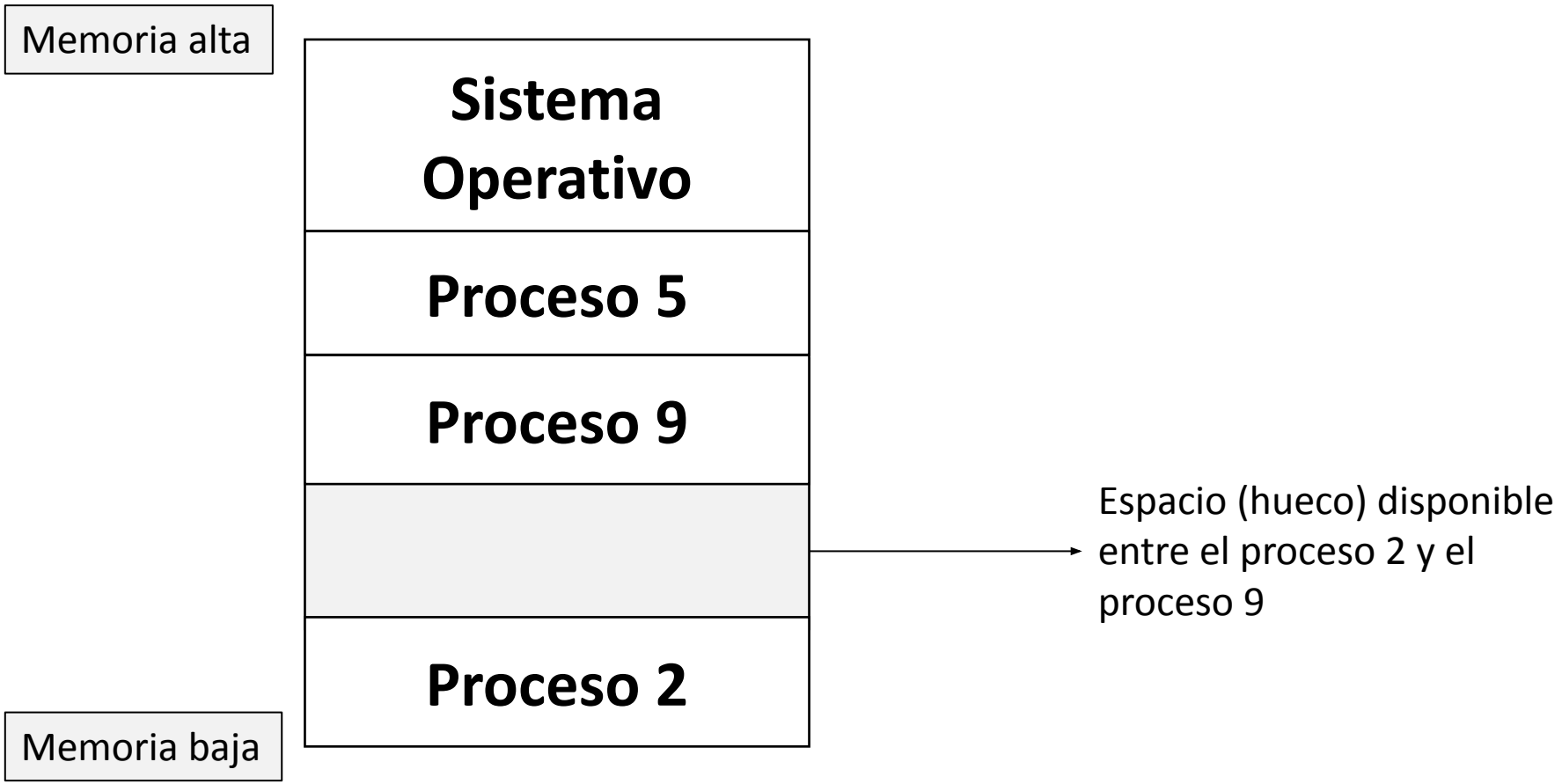


Espacio (hueco) disponible entre el proceso 2 y el proceso 5

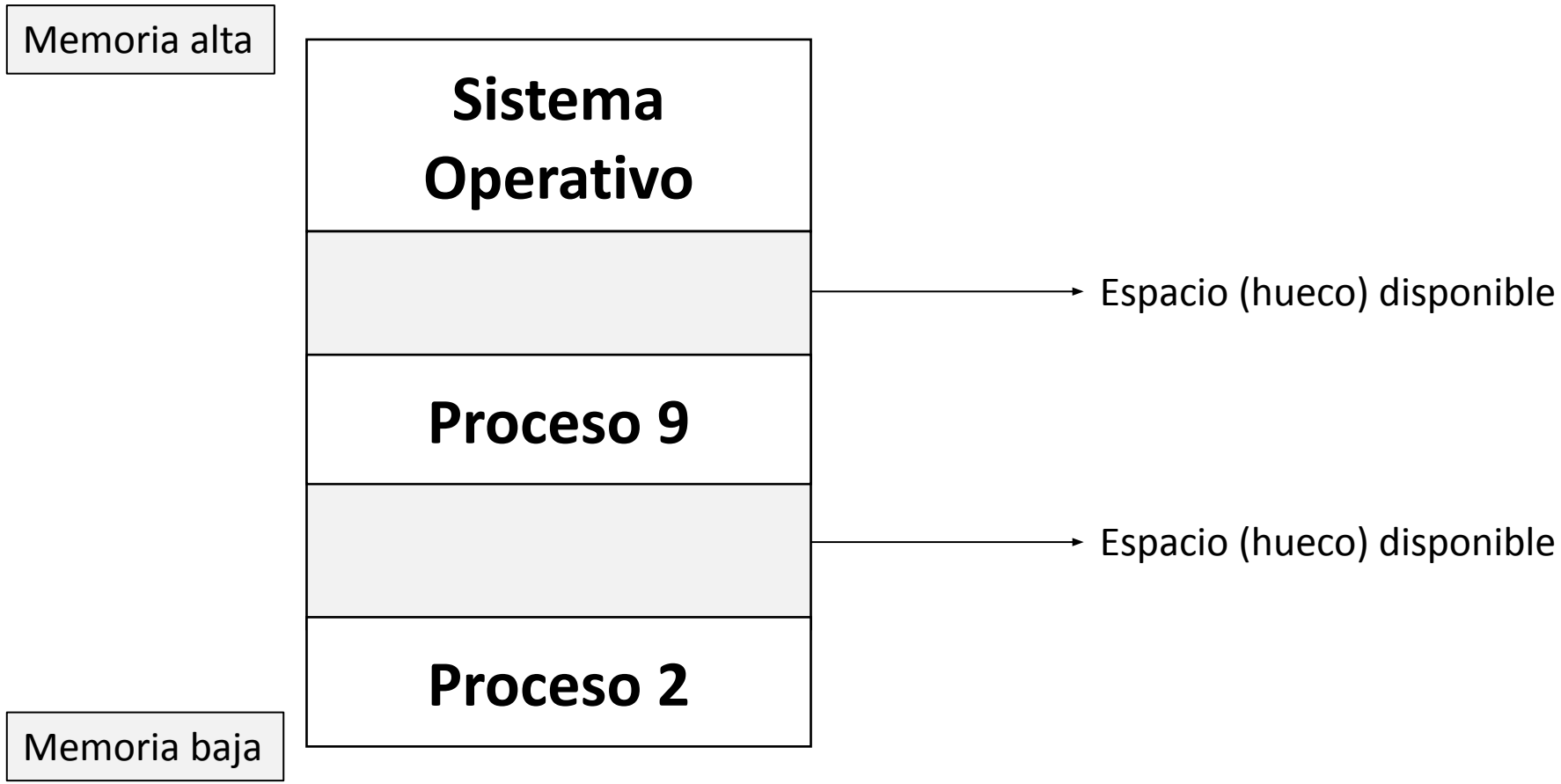
Asignación contigua



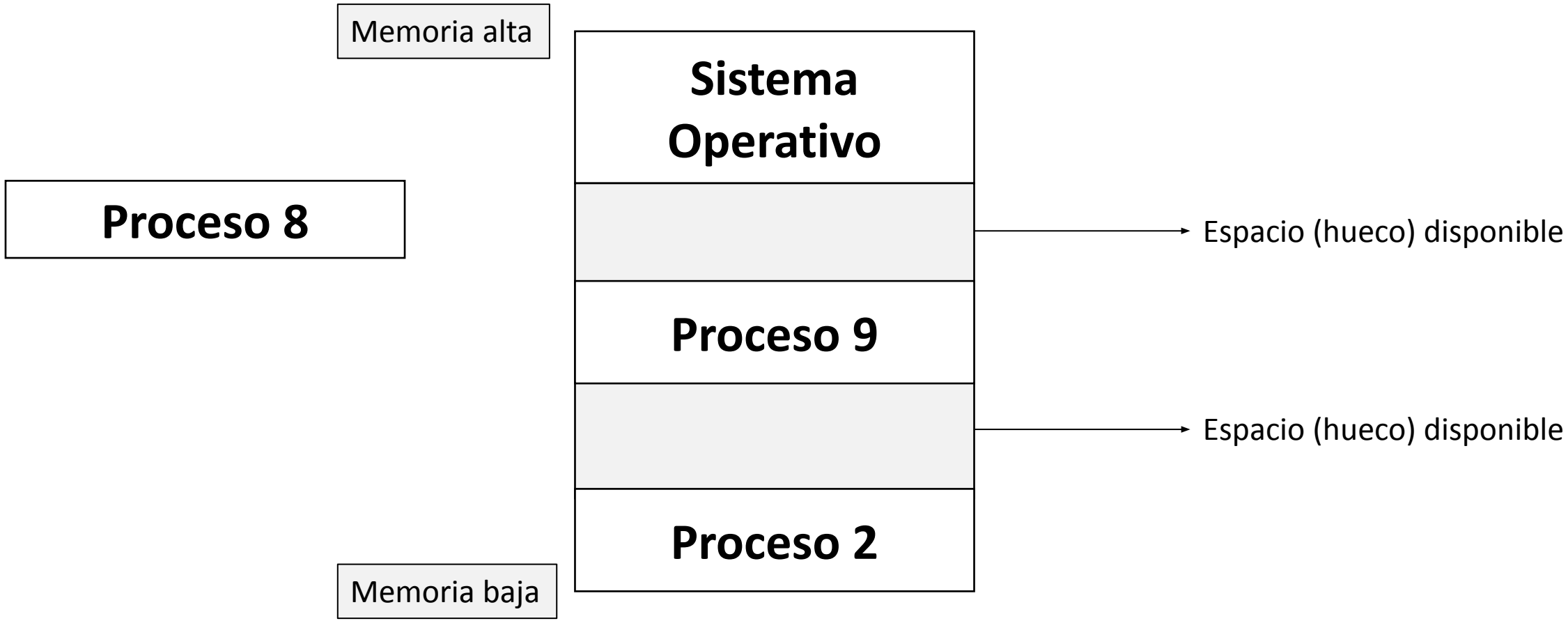
Asignación contigua



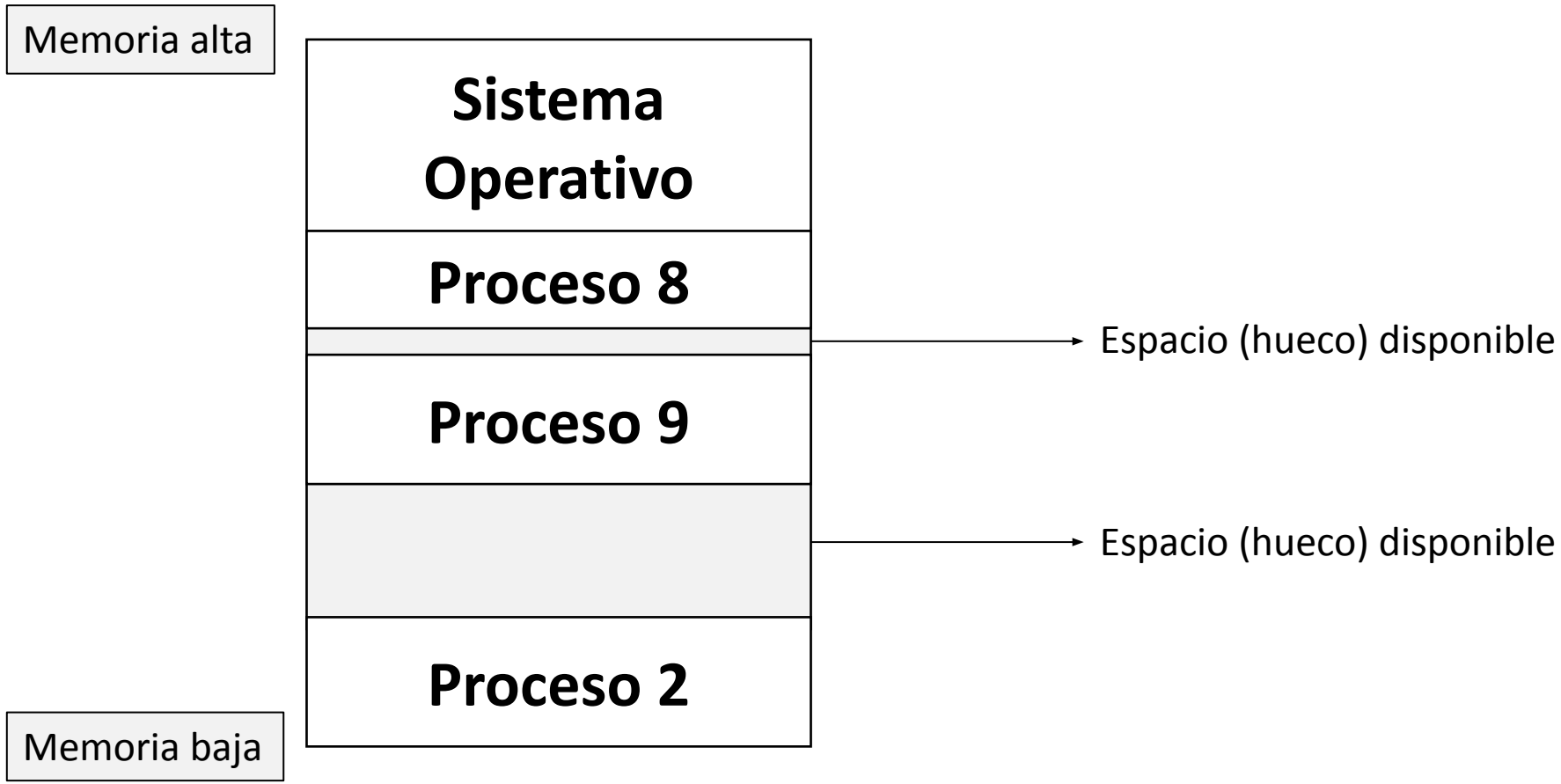
Asignación contigua



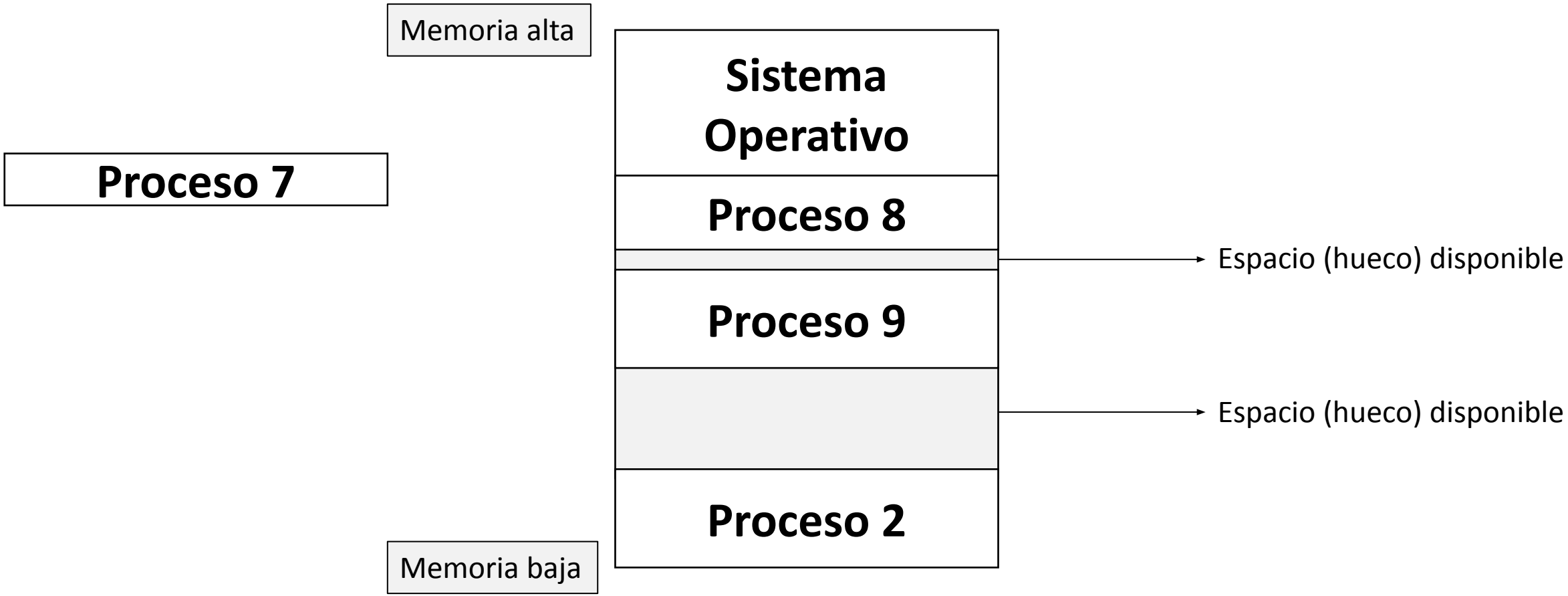
Asignación contigua



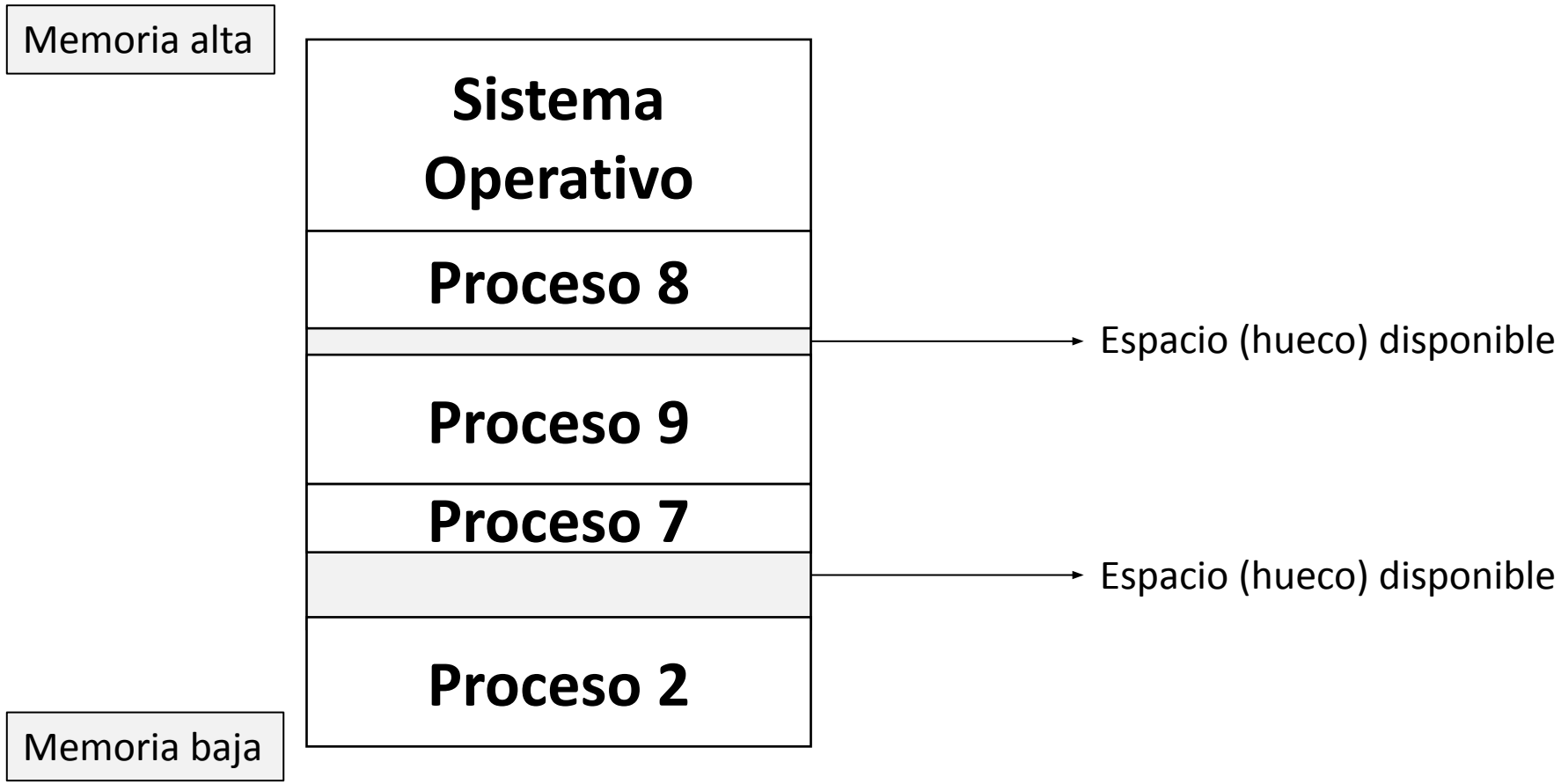
Asignación contigua



Asignación contigua



Asignación contigua



Asignación contigua

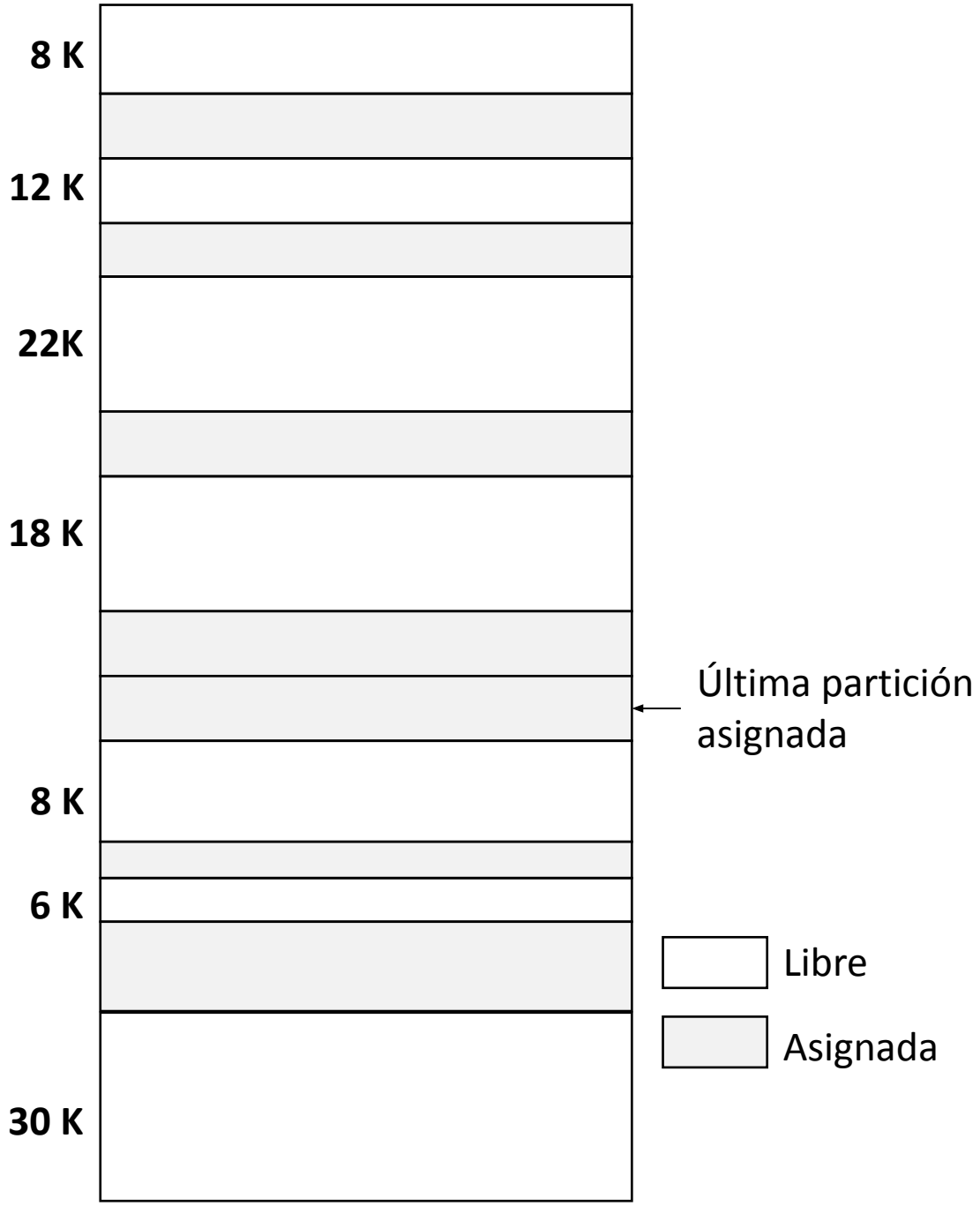
- ¿Qué sucede cuando no hay suficiente memoria para almacenar un nuevo proceso?
  - No aceptar el proceso
  - Ponerlo en cola de espera a que se libere el recurso
- Procesos que terminan devuelven el recurso de memoria ocupada
- Se debe llevar un registro de qué espacios (huecos) están disponibles
  - Espacios disponibles contiguos se unen
- En general se debe satisfacer la demanda de n bytes a partir del espacio disponible.

Estrategias de asignación contigua

- Estrategias de asignación
  - First-fit (primer ajuste)
  - Best-fit (mejor ajuste)
  - Worst-fit (peor ajuste)
- Estrategias usadas para seleccionar un espacio libre para asignar a un proceso.

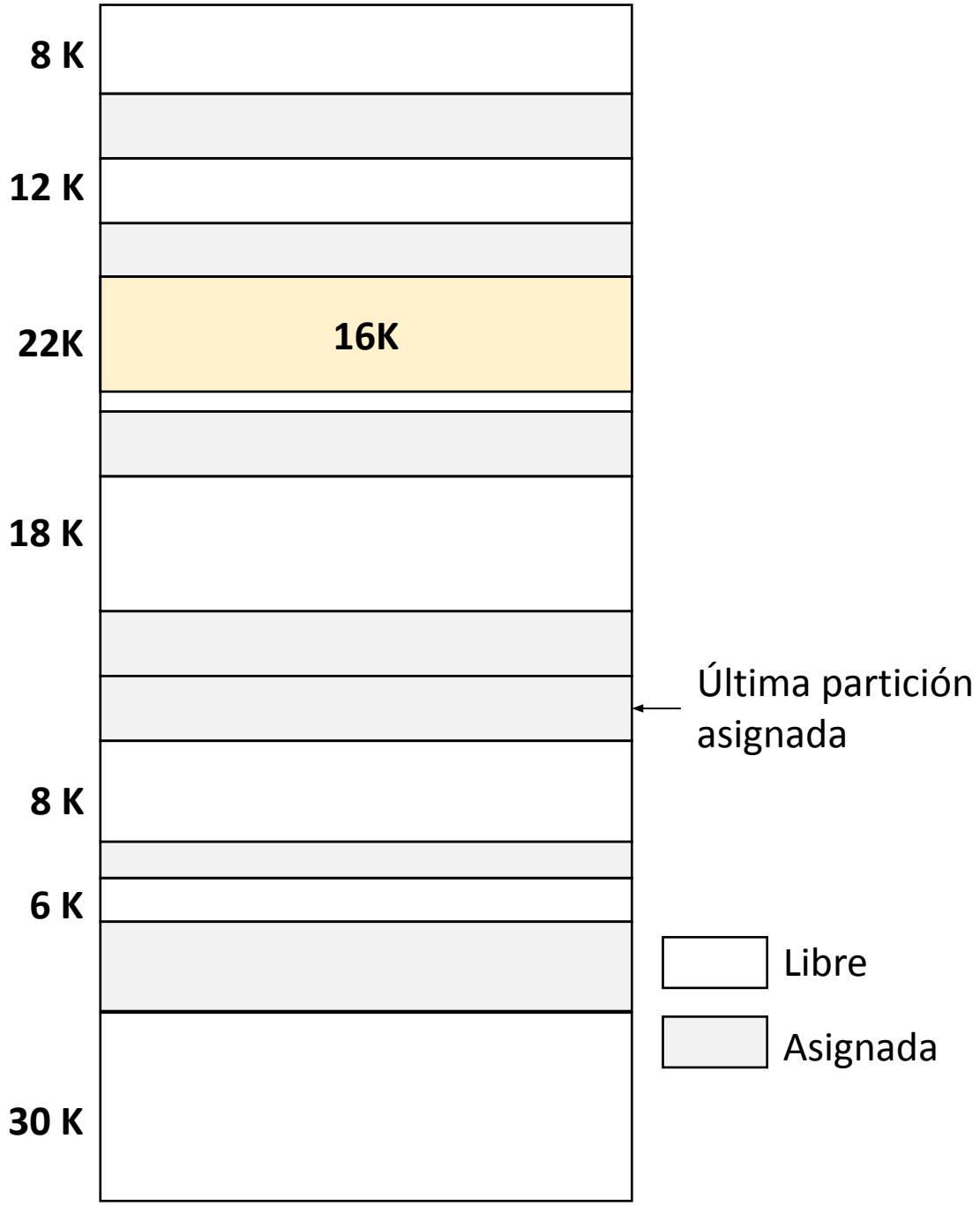
Estrategias

- Se necesita espacio para un proceso que mide 16K



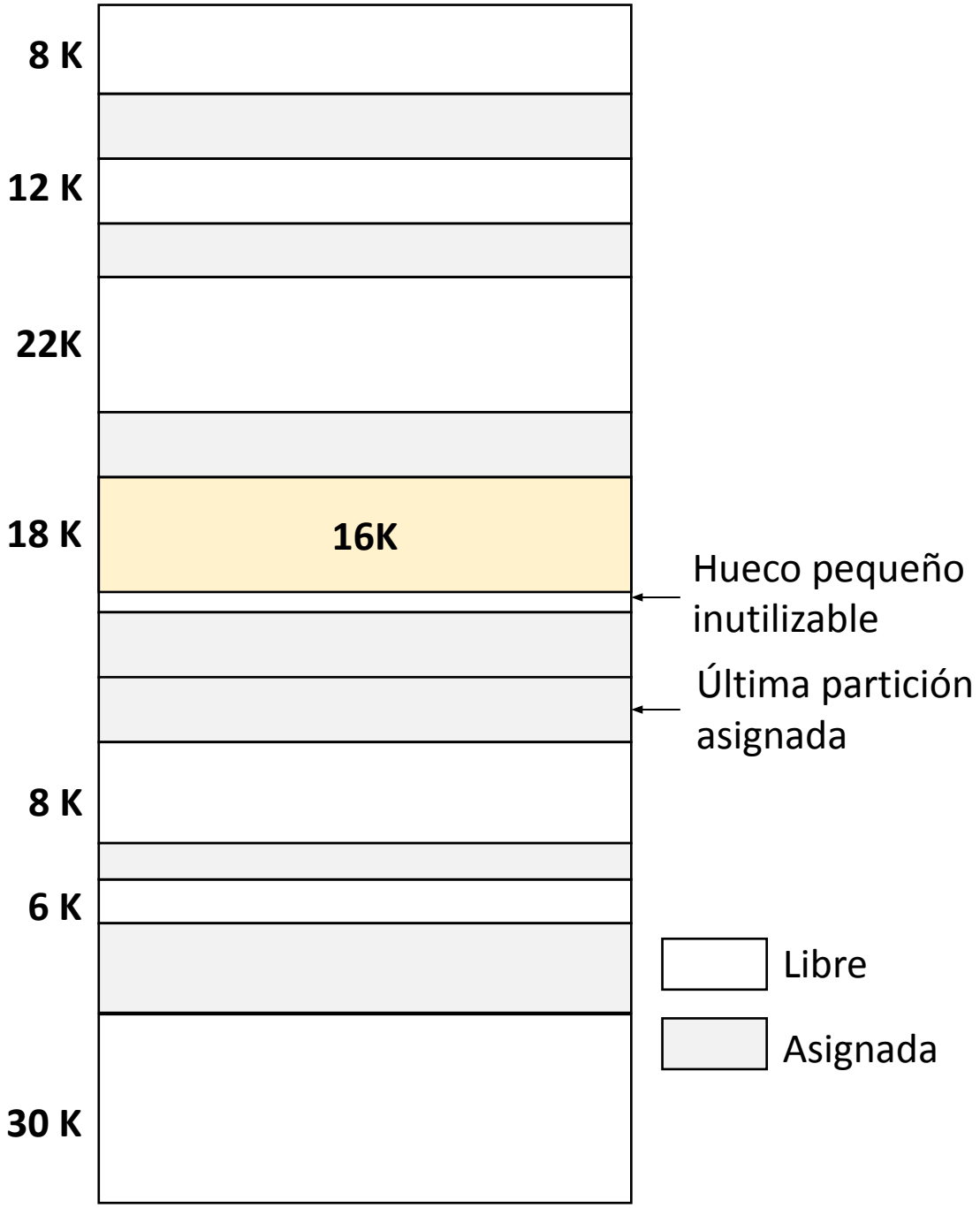
Estrategias

- **Primer ajuste**
  - Primer hueco en donde quepa el proceso.
  - Es la mejor política de ajuste



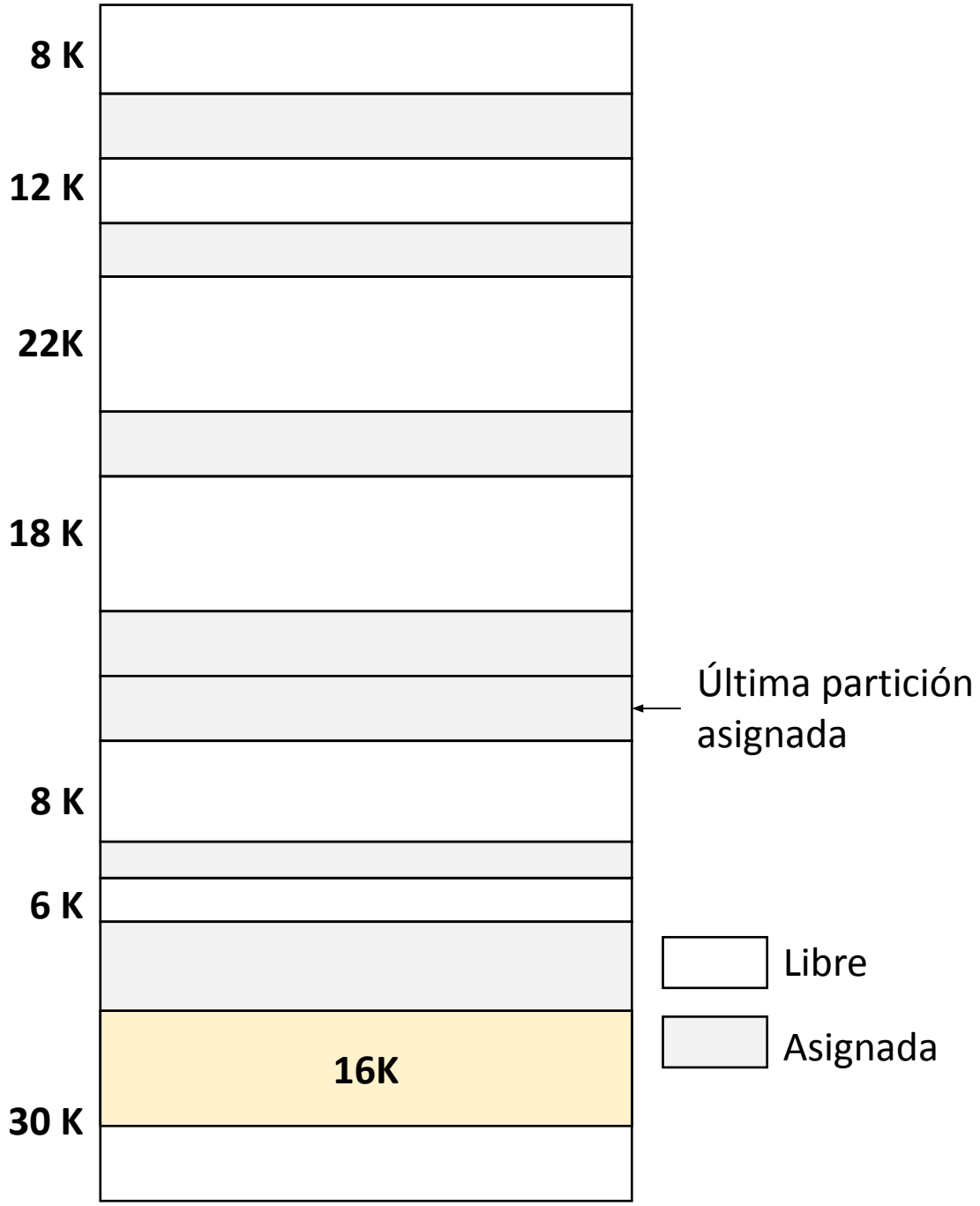
Estrategias

- **Mejor ajuste**
  - Buscar el hueco más pequeño en donde quepa el proceso
  - Quedan huecos inutilizables
  - Implica mantener una estructura (lista) ordenada por tamaño de los huecos
  - Produce fragmentos muy pequeños



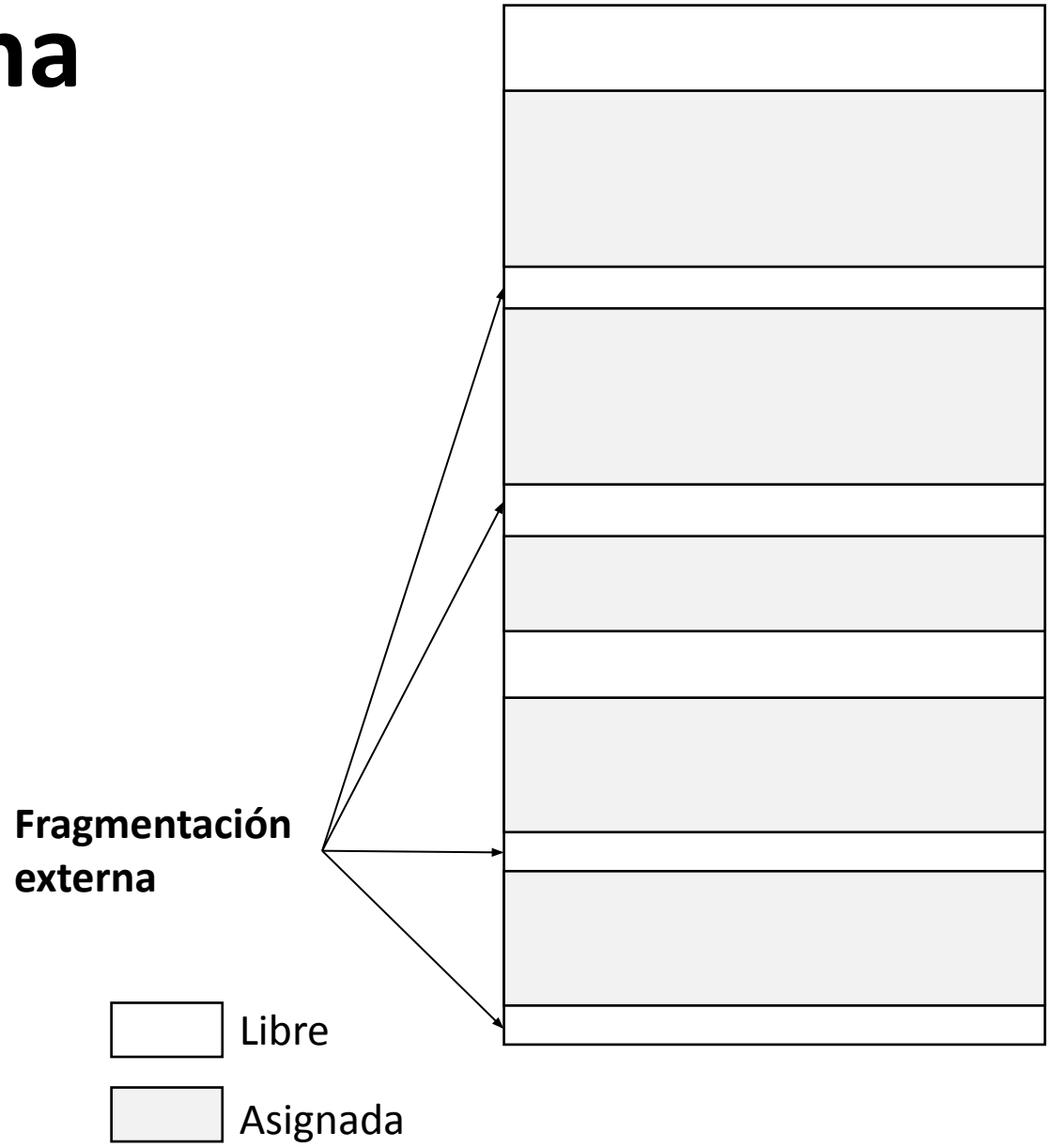
Estrategias

- **Peor ajuste**
  - Buscar el hueco más grande.
  - Hay que mantener una lista de huecos ordenada por tamaño.
  - Se buscar dejar huecos no tan pequeños que sí sean utilizables
  - Produce fragmentos muy grandes



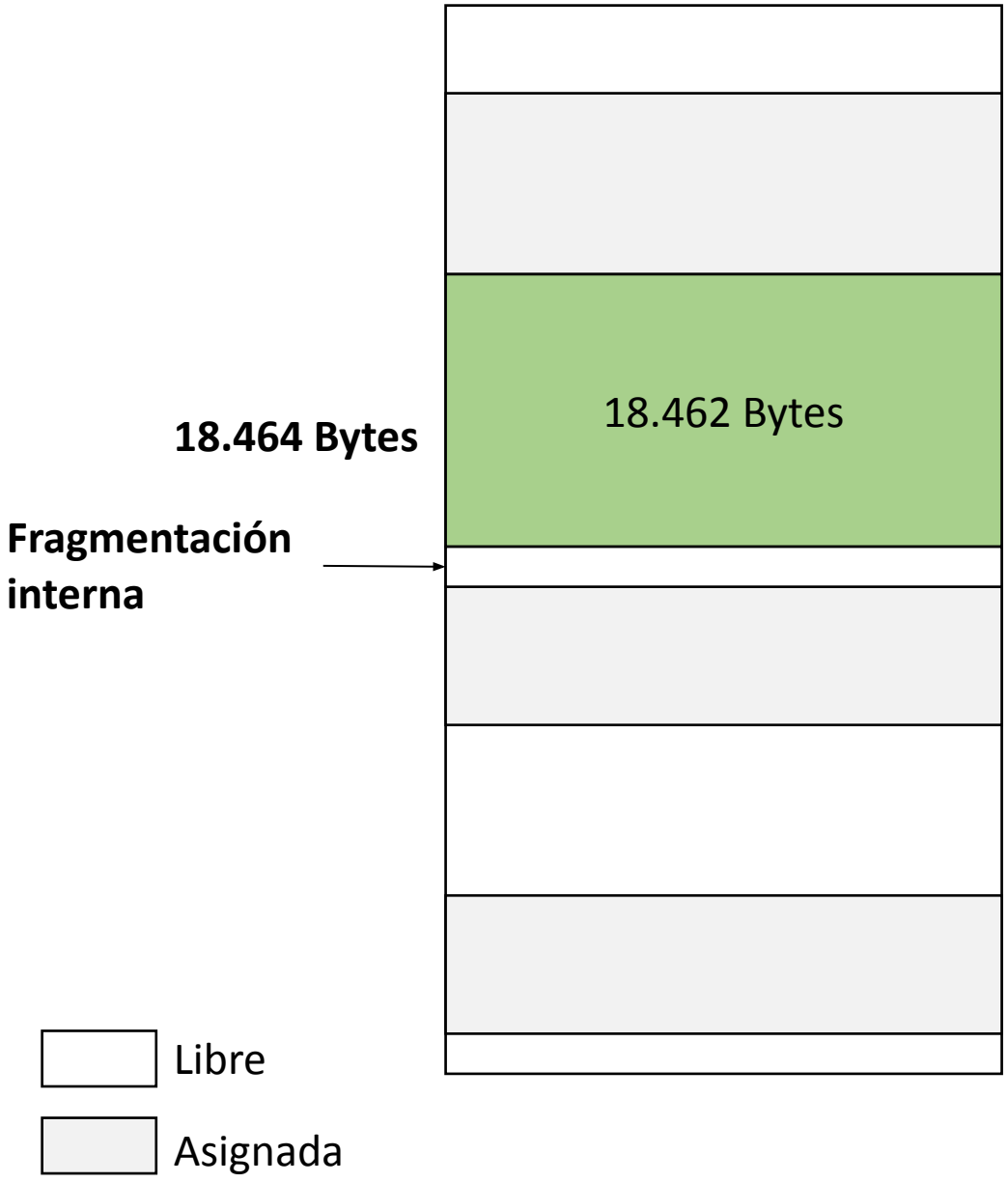
Fragmentación externa

- Existe espacio disponible para satisfacer la demanda pero el espacio disponible es no contiguo.
- El espacio disponible está distribuido en huecos pequeños.
- Huecos entre las particiones asignadas,



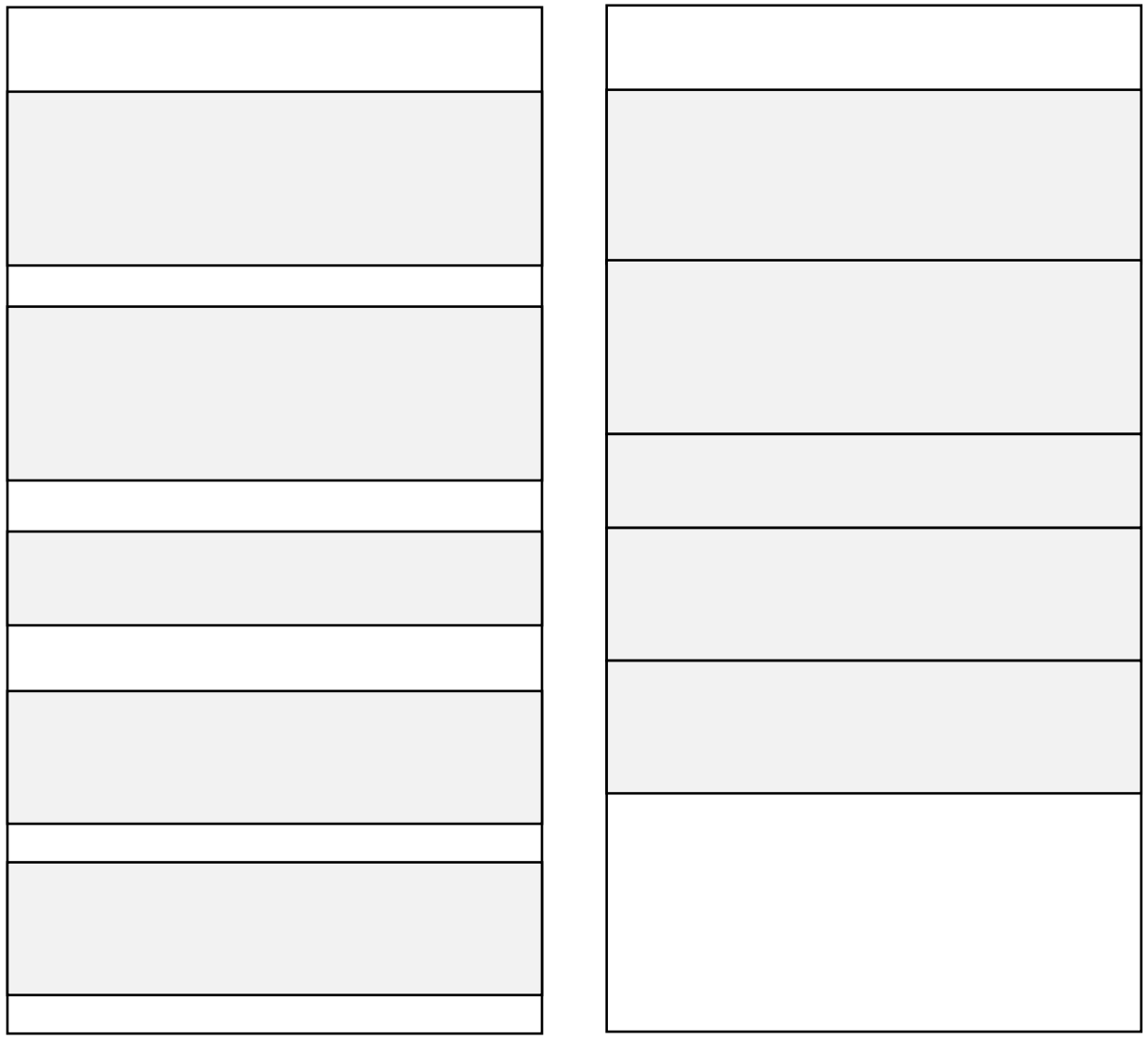
Fragmentación interna

- Llega proceso solicitando 18.462 Bytes
- Se asigna todo el espacio disponible de los 18.464 Bytes
- Sobran 2 Bytes que es mejor tenerlos asignados a un proceso
  - Podría (o no) pedirlos (**`malloc`**, **`new`**)
- La memoria asignada es un poco más grande de la solicitada.



Compactación

- Técnica para disminuir la fragmentación externa.
- Funciona siempre y cuando la **reubicación** de direcciones sea dinámica
  - Las direcciones se calculan en tiempo de ejecución
- Implica mover los procesos a nuevos espacios de memoria
  - ¿Cada cuanto?
  - ¿Costo?



Referencias

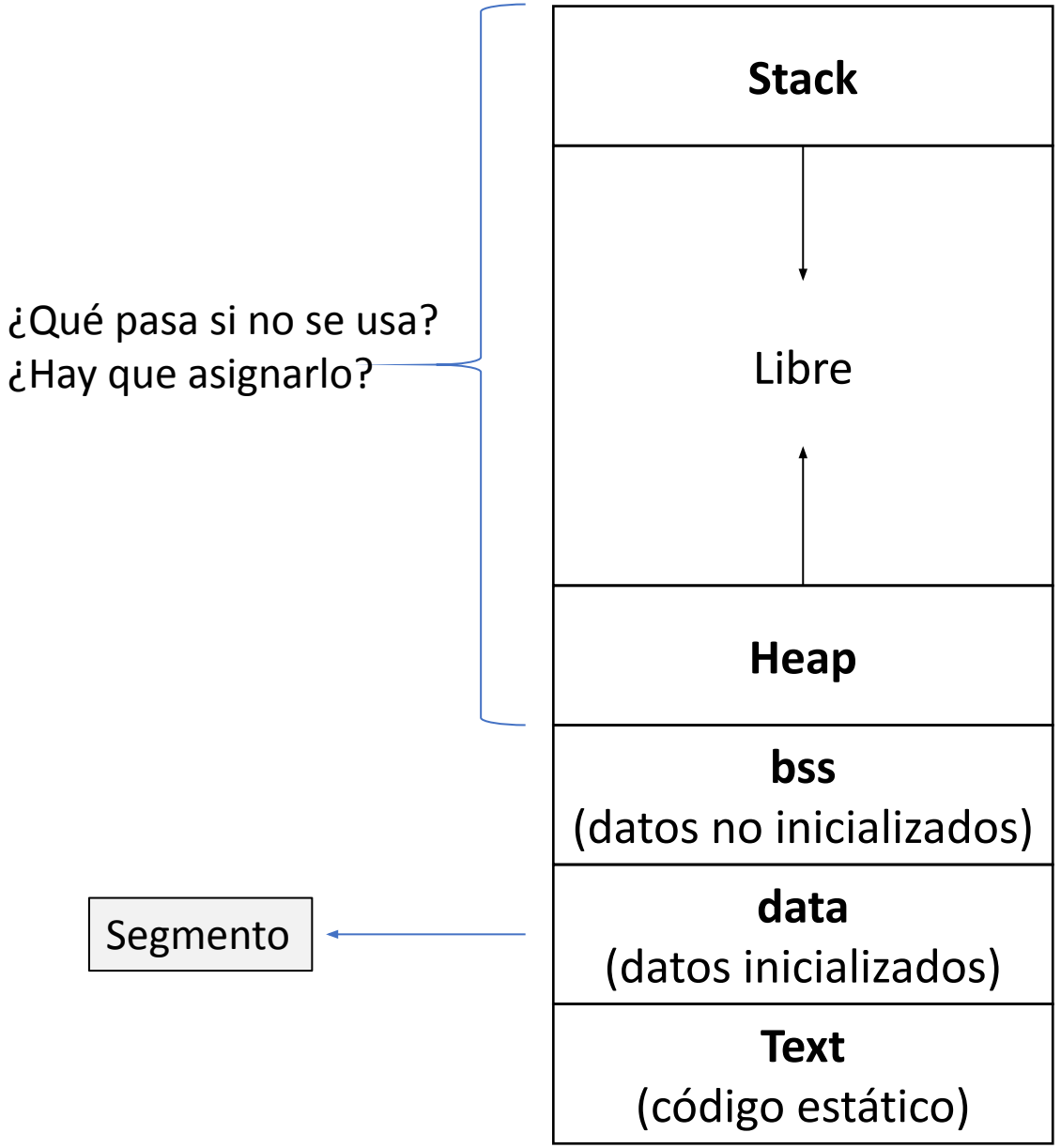
- Carretero Pérez, J., García Carballeíra, F., De Miguel Anasagasti, P., & Pérez Costoya, F. (2018). Contiguos Memory Allocation. In *Operating Systems Concepts* (10th ed., pp. 356–360). John Wiley & Sons, Inc.

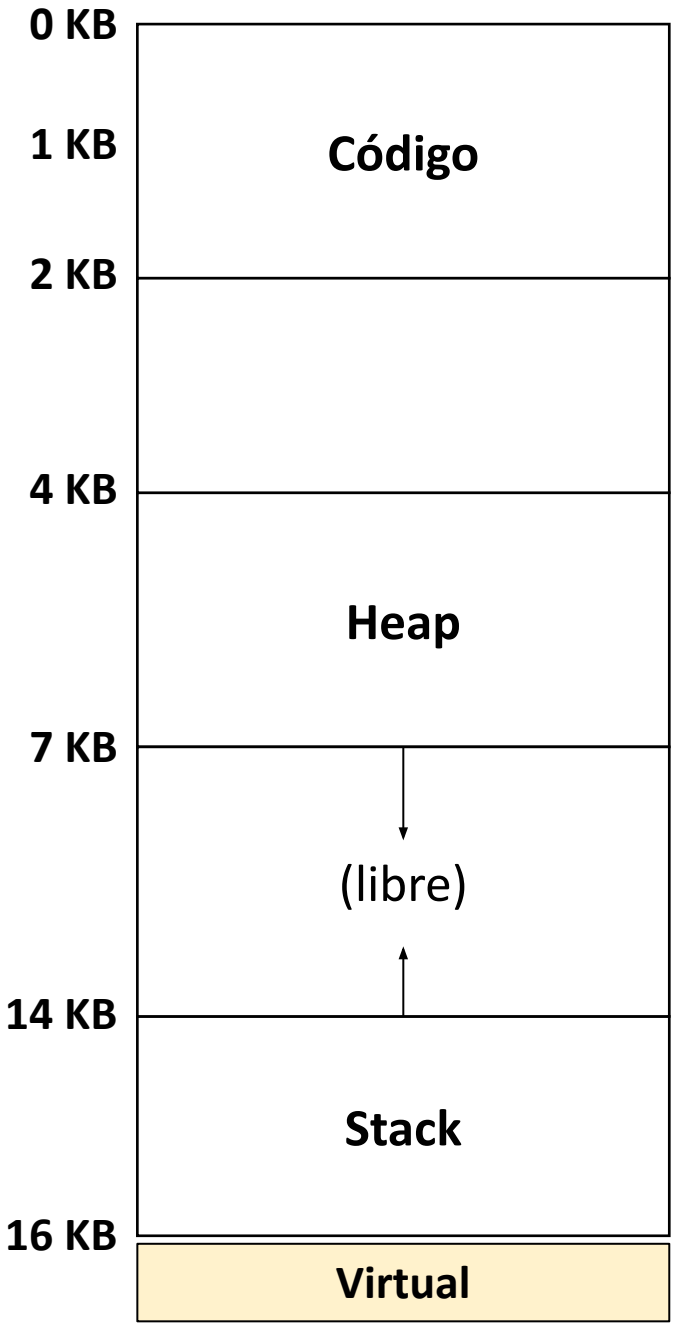
Segmentación

Adaptación (ver referencias al final)

Segmentación

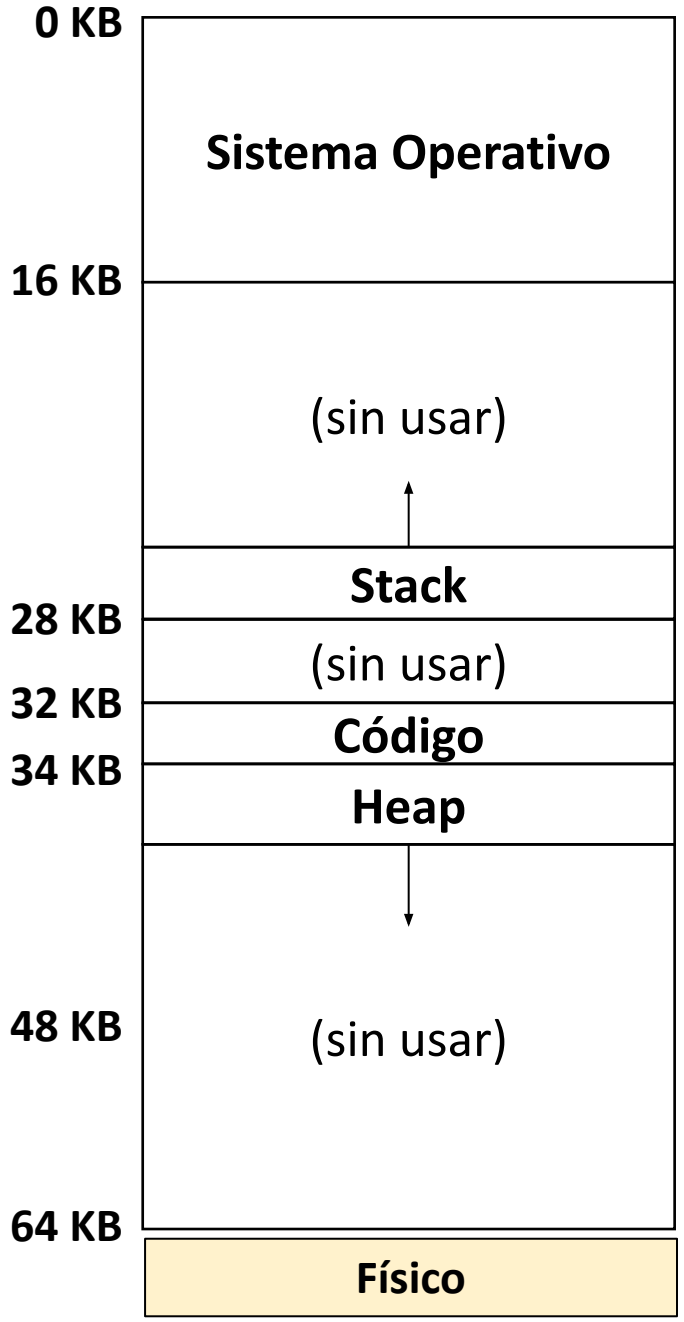
- Tener **más de un registro base y un registro límite** por segmento lógico del espacio de direccionamiento de un proceso.
- Ubicar cada segmento en diferentes partes de la memoria física
  - Evitar asignar espacio no usado. P. Ej.: Heap y Stack
- No se hace asignación contigua de todo la imagen del proceso.

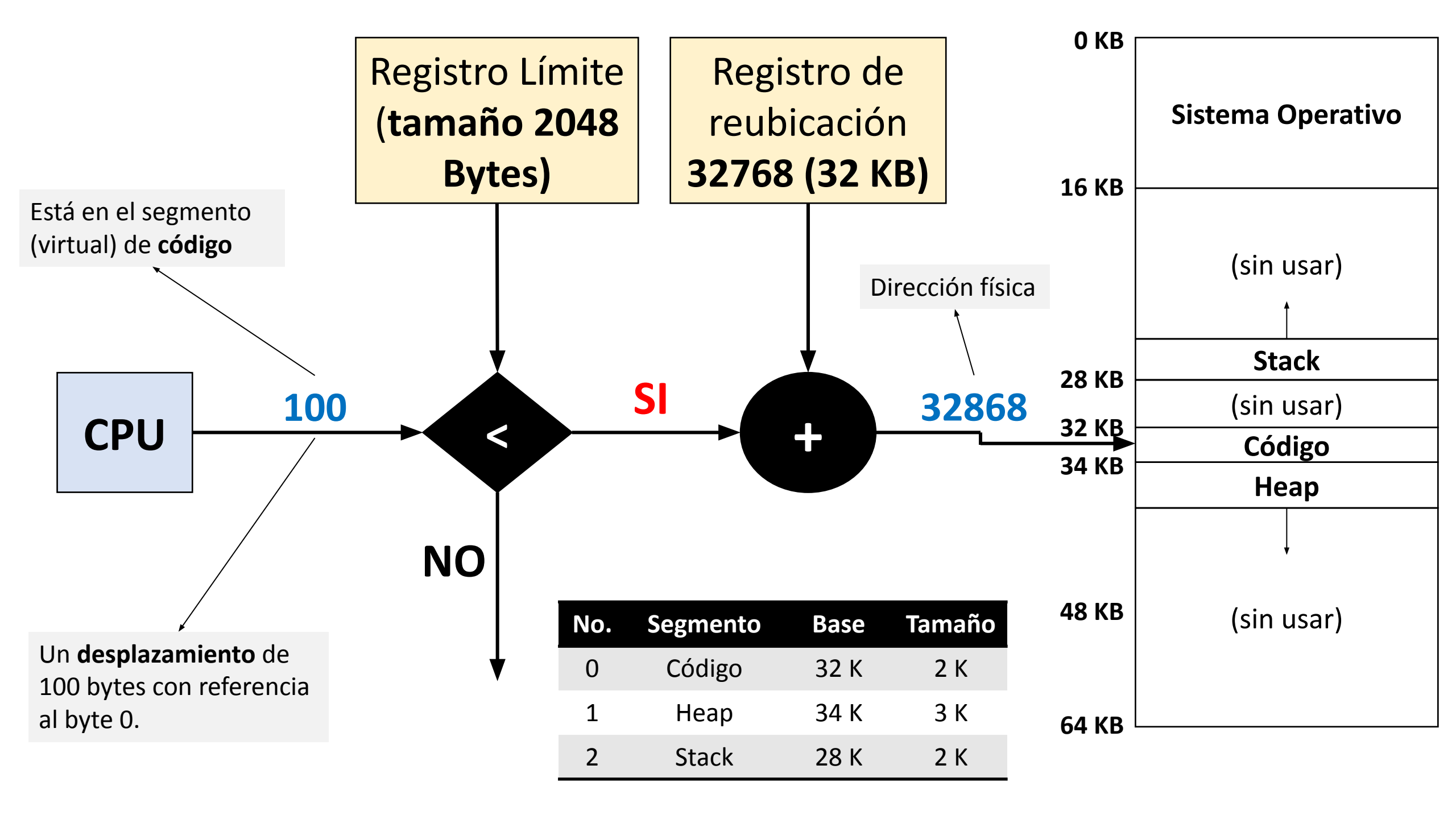




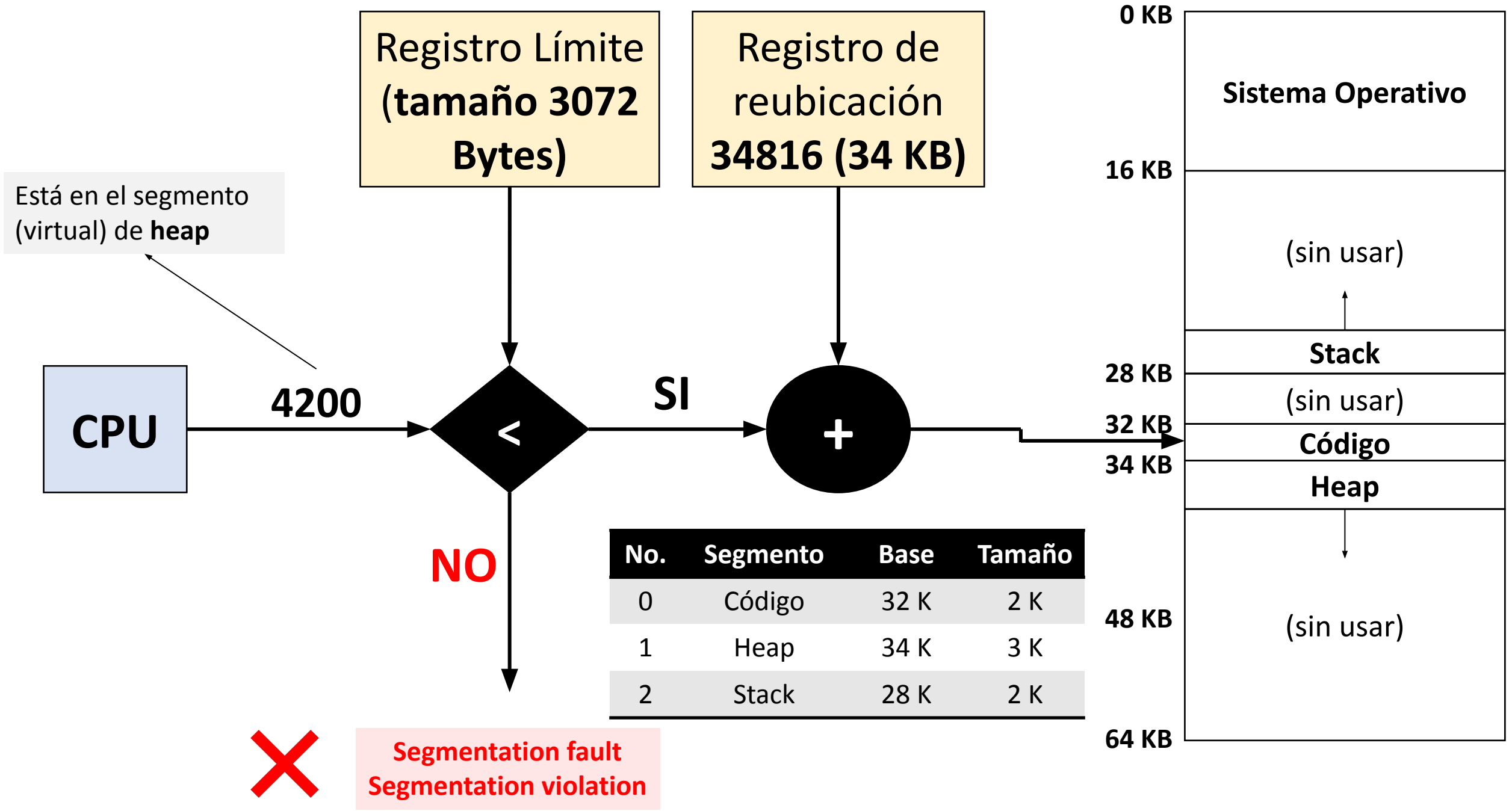
No.	Segmento	Base	Tamaño
0	Código	32 K	2 K
1	Heap	34 K	3 K
2	Stack	28 K	2 K

- Espacio lógico de direccionamiento (izquierda) asignado en memoria física (derecha)
- Cada segmento independientemente en memoria física





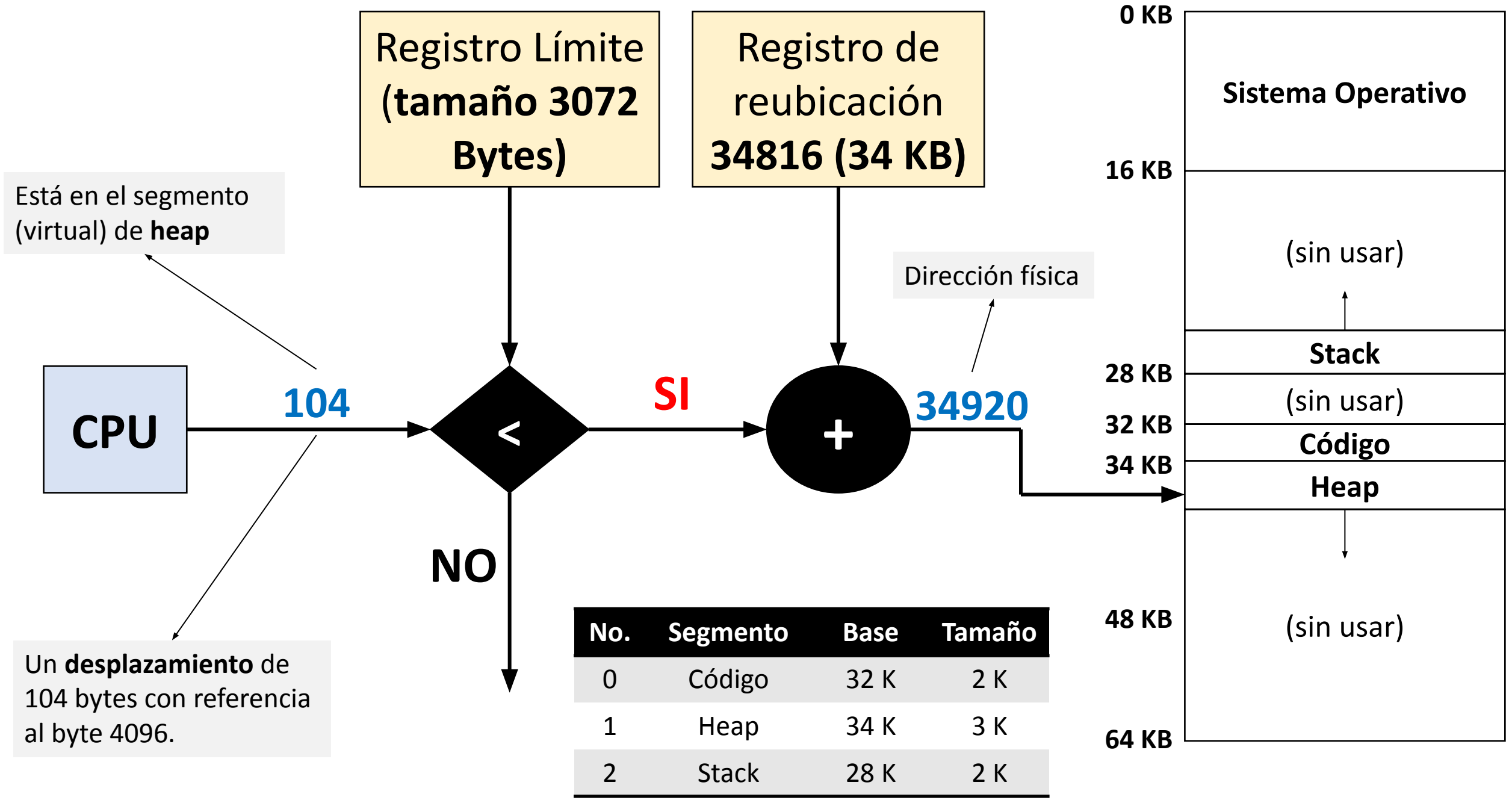
No.	Segmento	Base	Tamaño
0	Código	32 K	2 K
1	Heap	34 K	3 K
2	Stack	28 K	2 K



No.	Segmento	Base	Tamaño
0	Código	32 K	2 K
1	Heap	34 K	3 K
2	Stack	28 K	2 K

Traducción del direccionamiento

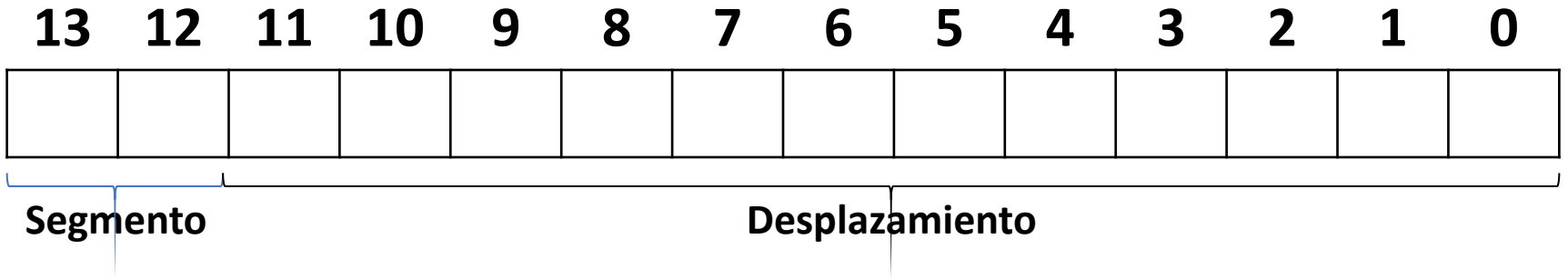
- En este último caso hay extraer primero el desplazamiento.
  - ¿A qué bytes se está haciendo referencia en este segmento?
- Cálculo del desplazamiento
  - 4200 (dirección virtual de referencia) – 4096 (inicio del segmento) $= \mathbf{104}$



No.	Segmento	Base	Tamaño
0	Código	32 K	2 K
1	Heap	34 K	3 K
2	Stack	28 K	2 K

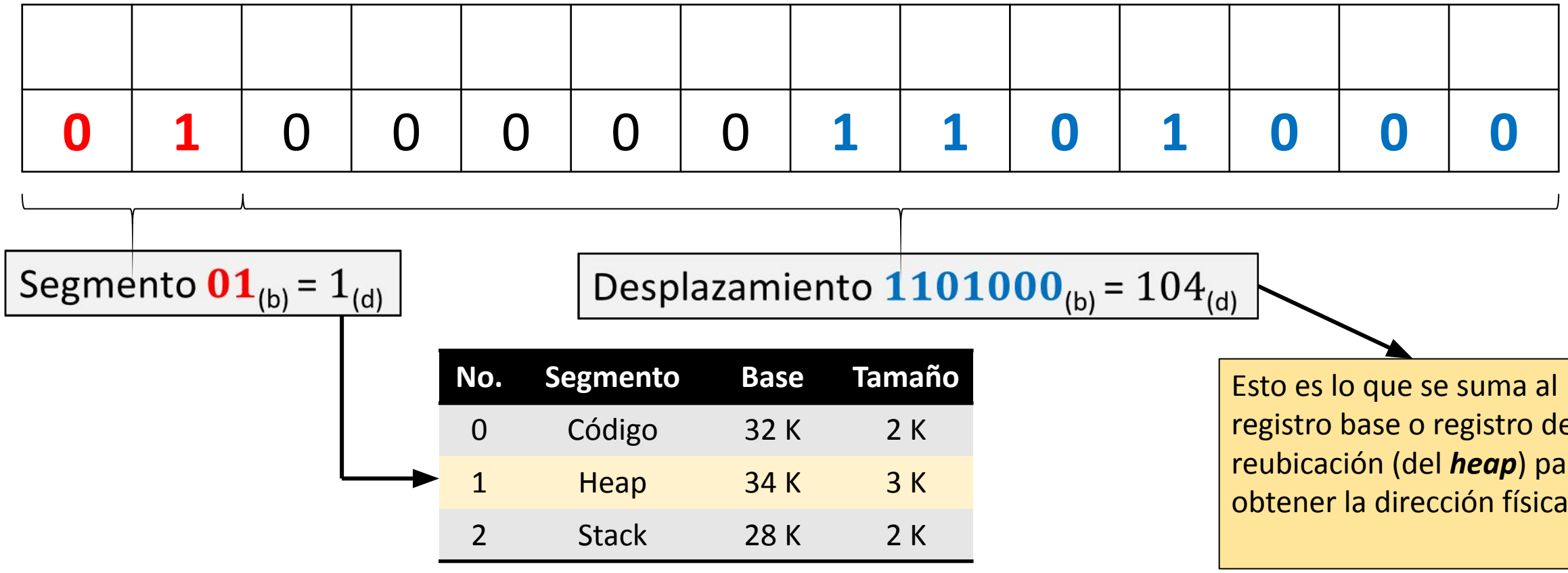
Traducción del direccionamiento

- Espacio virtual de direccionamiento (16 KB): 16384 Bytes
- Tamaño en bits de las direcciones: $2^{14} = 16384$
  - Se necesitan 14 bits para direccionar un espacio de 16KB.
- Se tienen tres segmentos: código, *heap* y *stack*
- De los 14 bits de la dirección se usarán 2 bits (los más significativos) para indicar el segmento.
  - $2^2 = 4$ Se necesitan al menos dos bits para direccionar tres segmentos.



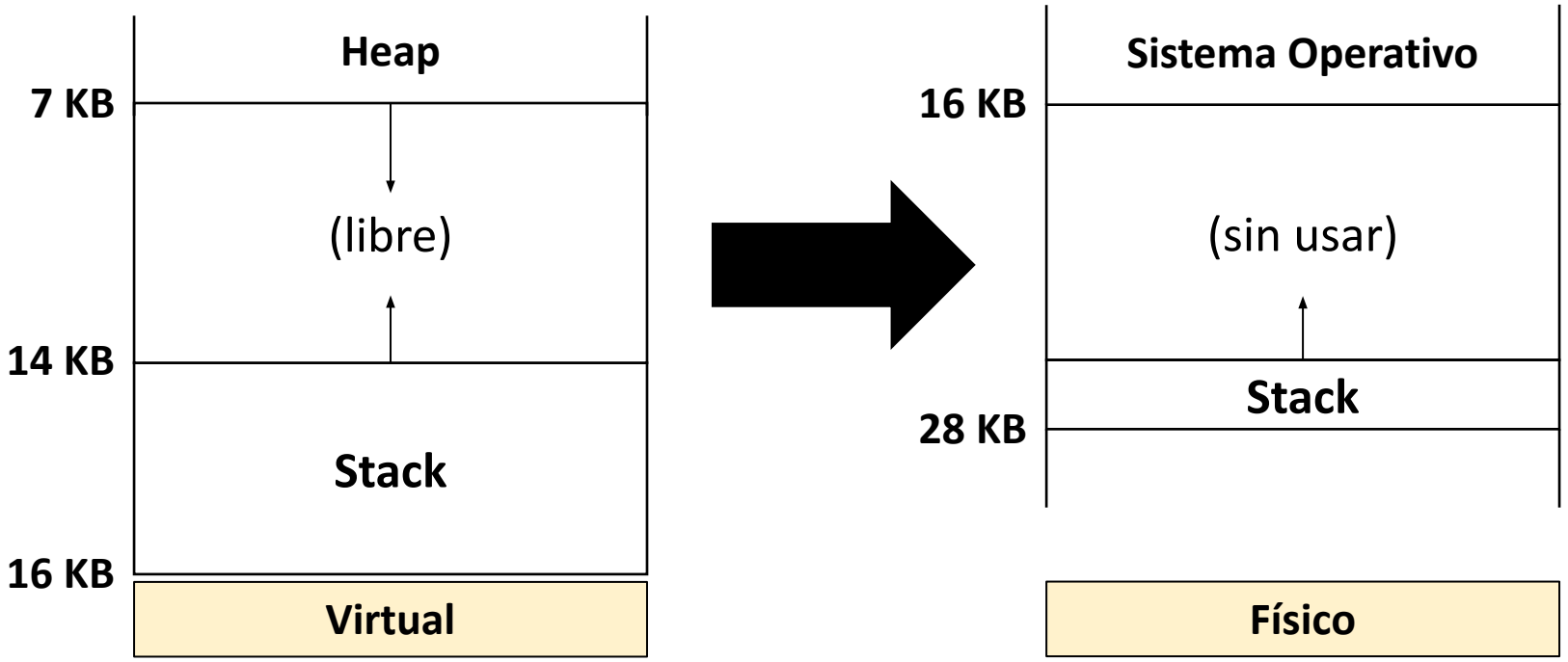
Traducción del direccionamiento

- Dirección virtual: 4200



No.	Segmento	Base	Tamaño
0	Código	32 K	2 K
1	Heap	34 K	3 K
2	Stack	28 K	2 K

¿Qué pasa con el segmento *stack*?



- El segmento *stack* crece en **hacia atrás**
- Crece desde direcciones altas hacia direcciones bajas
- Esto es así en la práctica

¿Qué pasa con el segmento *stack*?

- Se necesita soporte adicional del hardware para indicar esta situación

Segmento	Base	Tamaño (máx. 4K)	Crece positivamente
Código$_{00}$	32 K	2K	1
Heap$_{01}$	34 K	3K	1
Stack$_{11}$	28 K	2K	0

¿Qué pasa con el *stack*?

- Dirección virtual (15 KB) = 15360

1	**1**	1	1	0	0	0	0	0	0	0	0	0	0

Segmento $\mathbf{11}_{(b)} = 3_{(d)}$

Desplazamiento $\mathbf{11000000000}_{(b)} = 3072_{(d)} = 3$ KB

Segmento	Base	Tamaño (máx. 4K)	Crece positivamente
Código_{00}	32 K	2 K	1
Heap_{01}	34 K	3 K	1
Stack_{11}	28 K	2 K	0

¿Qué pasa con el *stack*?

- Para esta traducción es necesario restar al desplazamiento el valor del tamaño máximo de segmento.
  - $\mathbf{3\ KB - 4\ KB = -1\ KB}$ (desplazamiento negativo)
- El resultado anterior se le suma al registro base (registro de reubicación) del *stack*
  - $\mathbf{28\ KB + (-1\ KB) = 27\ KB}$
- El registro límite valida que el valor absoluto del desplazamiento sea menor o igual al tamaño del segmento
  - $\mathbf{|-1KB| \leq 2KB}$ (en ese caso **2KB** es el tamaño del stack en memoria física)

Segmentos compartidos

- Para compartir segmentos se necesita soporte adicional del hardware
  - P. Ej.: shared objects, DLL's

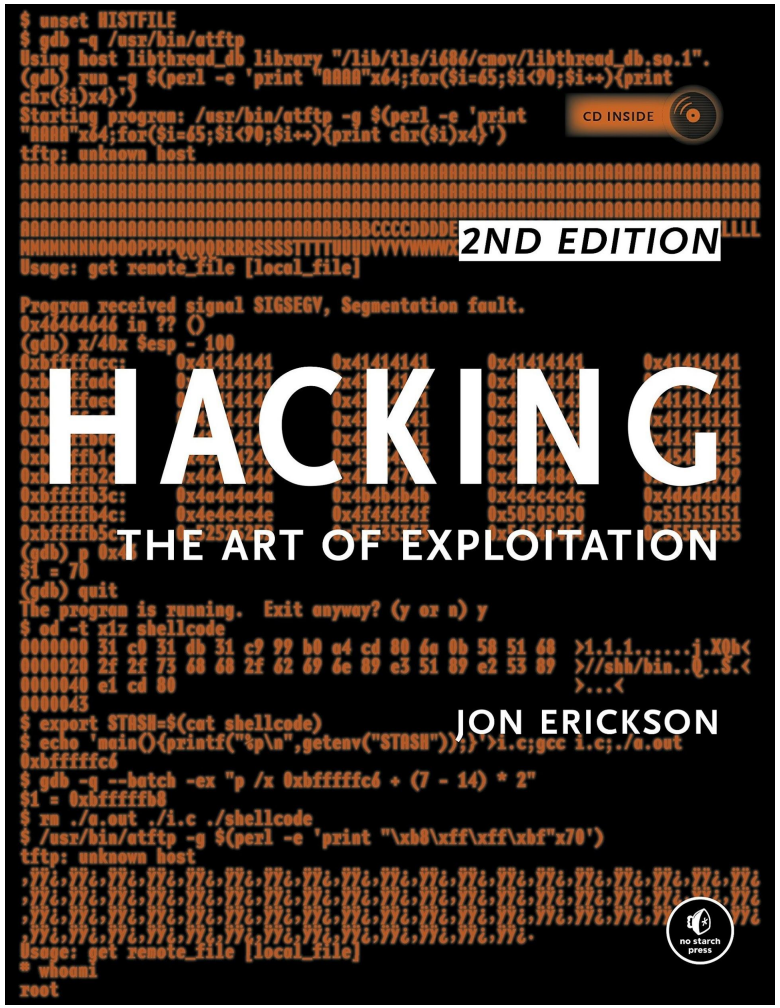
Segmento	Base	Tamaño (máx. 4K)	Crece positivamente	Protección
$Código_{00}$	32 K	2K	1	Read-Execute
$Heap_{01}$	34 K	3K	1	Read-Write
$Stack_{11}$	28 K	2K	0	Read-Write

Soporte desde el S.O

- ¿Qué hacer en el cambio de contexto?
  - Guardar registros por proceso
  - ¿Qué hacer frente a demanda dinámica de memoria? P. Ej.: `malloc()`
  - Encontrar espacio libre para asignar
  - Actualizar registros en (tabla de hardware)
  - Rechazar solicitudes cuando no hay suficiente memoria disponible
- Administración del espacio libre
  - Encontrar espacio libre para los segmentos de un nuevo proceso
  - Segmentos de diferente tamaño
  - Problemas de fragmentación externa

Lecturas recomendadas

- Smashing The Stack For Fun And Profit
  - Aleph One, Underground.org
  - Buffer Overflow, Stack Overflow
- File Infection Techniques
  - Bill Harrison
  - CS 4440/7440 Malware Analysis & Defense
- Hacking The Art Of Exploitation
  - Jon Erickson



Referencias

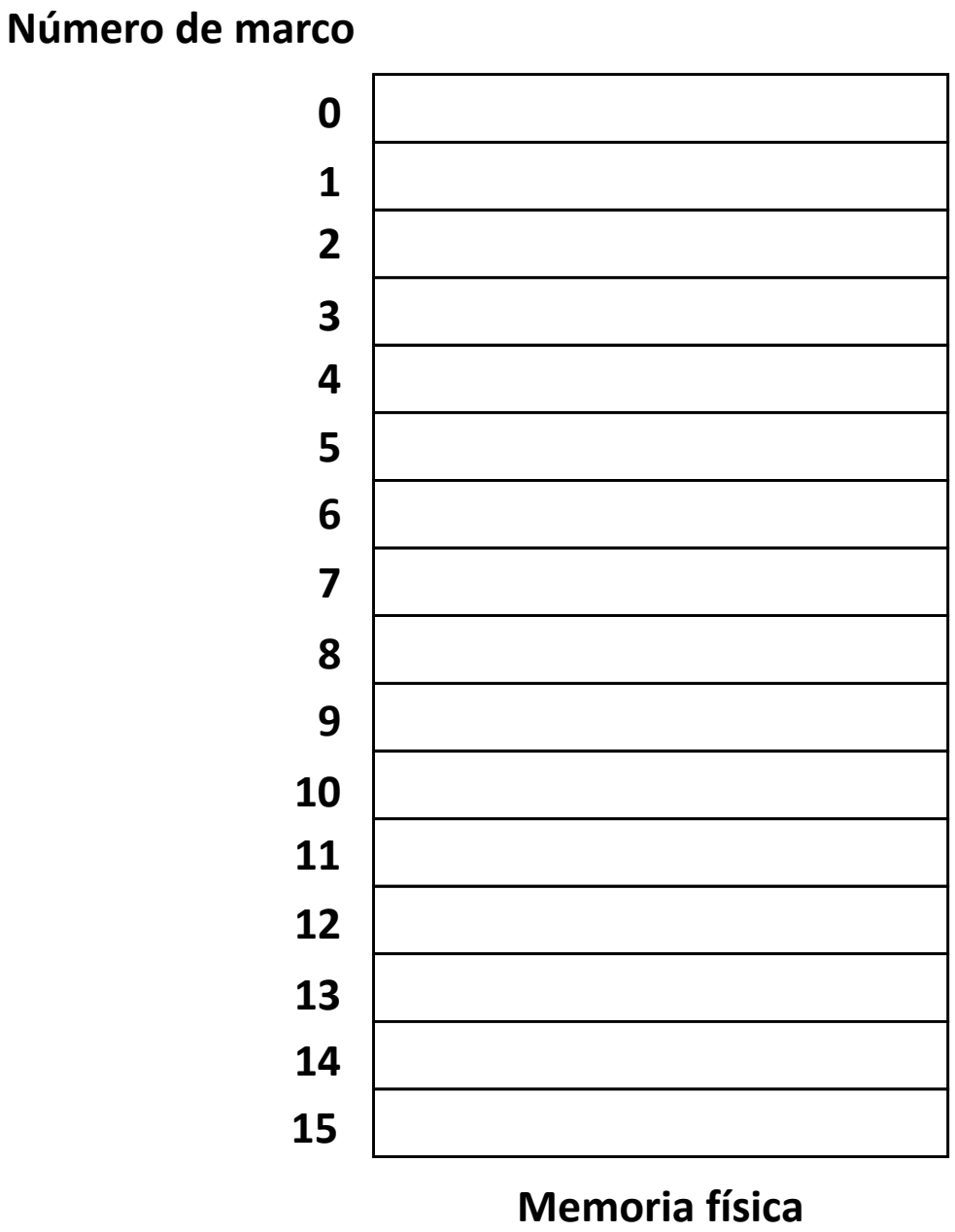
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. (2018). Segmentation. In *Operating Systems. Three Easy Pieces* (pp. 1–12). Arpaci-Dusseau Books.

Paginación de memoria

Adaptación (ver referencias al final)

Paginación

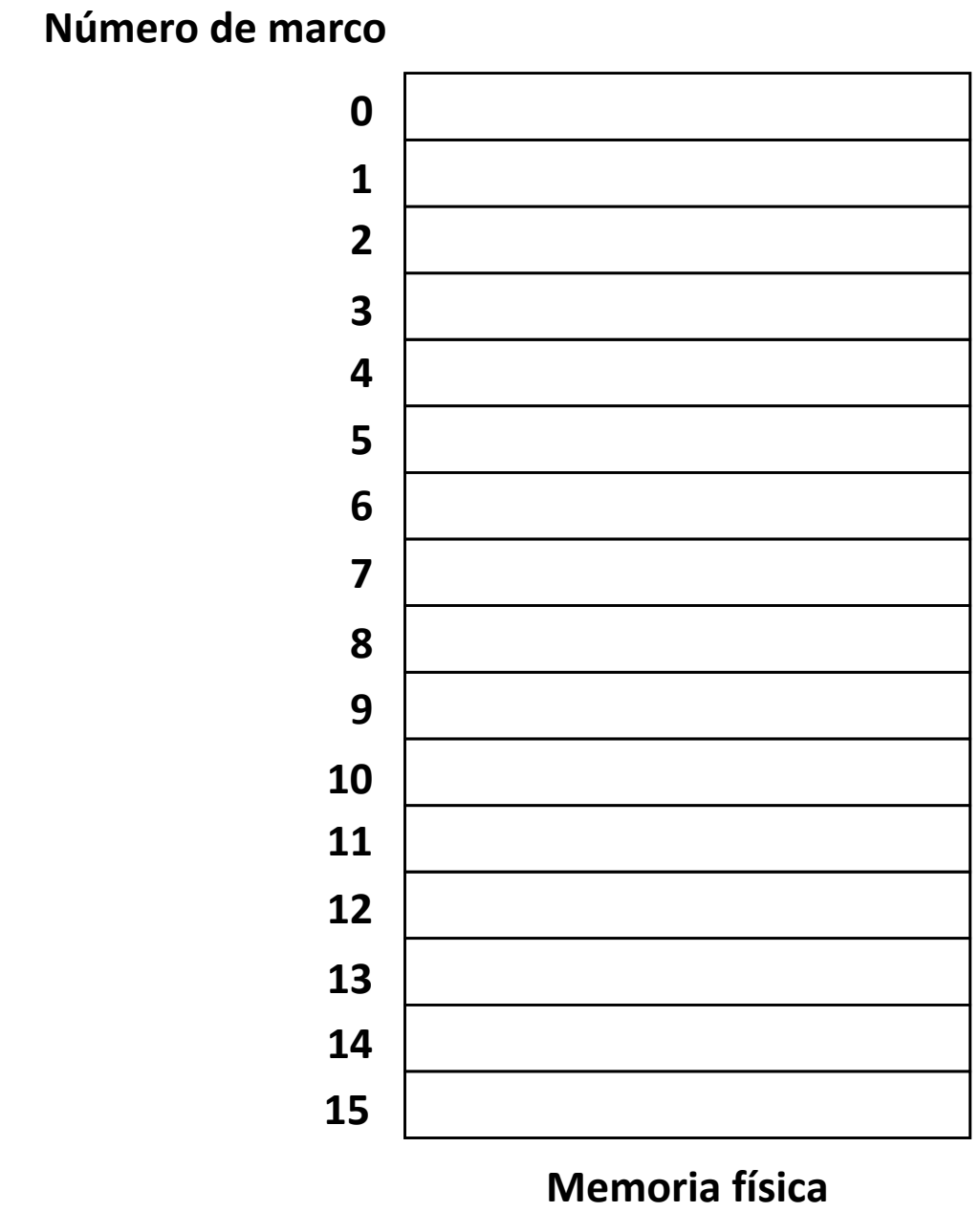
- Evita la fragmentación externa
- Técnica usada en la mayoría de sistemas operativos modernos
- Dividir la **memoria física** en bloques de tamaño fijo llamados **marcos**.
- Tamaño del bloque es una potencia de 2: 512 Bytes a 8192 bytes



Paginación

- La **memoria lógica** también se va a dividir en bloques **del mismo tamaño** de los marcos
- Estos bloques se llaman **páginas**.

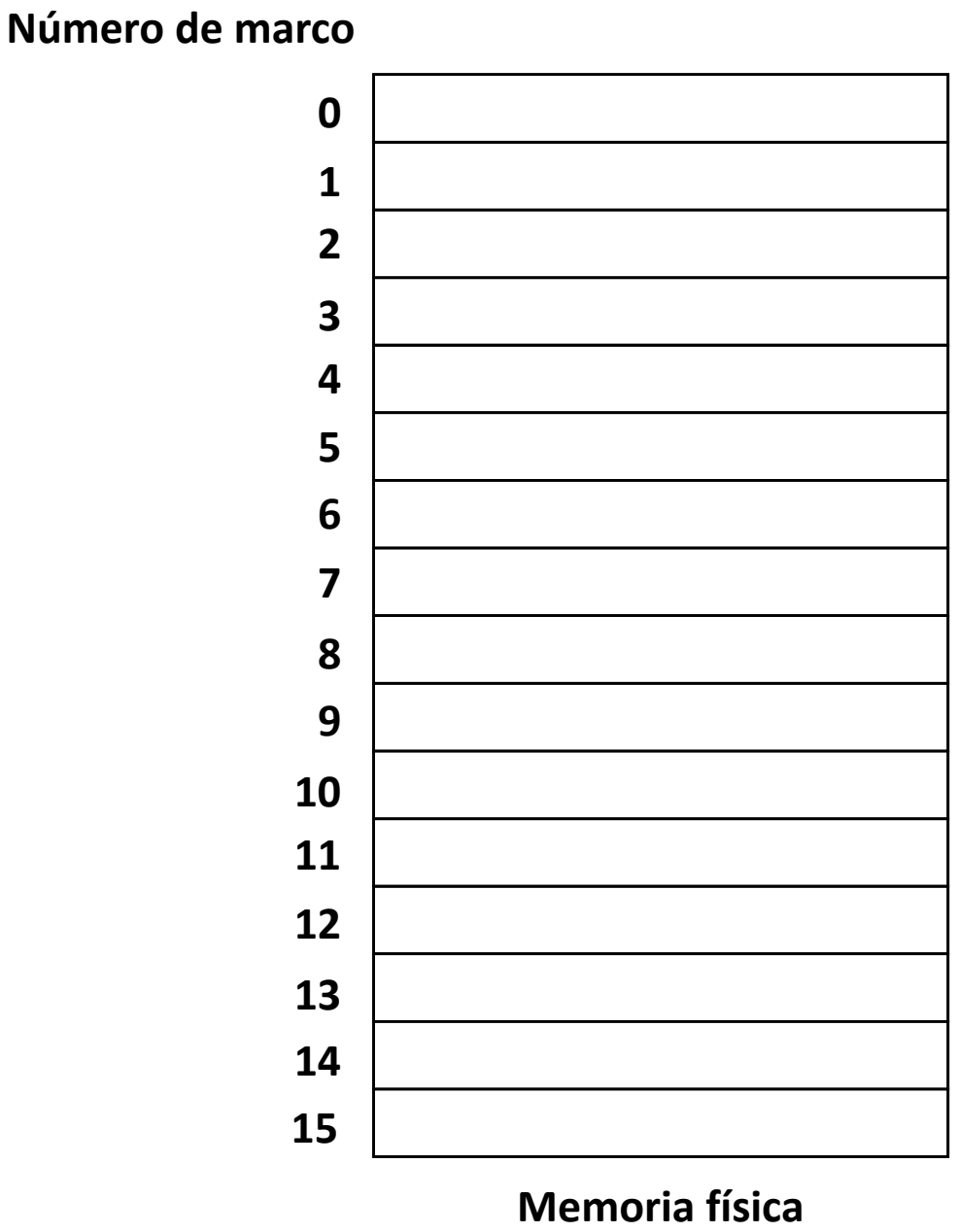
Proceso A
A1
A2
A3
A4
A5



Paginación

- Las páginas se cargan en los marcos.
- El espacio de direcciones lógicas de un proceso puede no estar contiguo en memoria física.

Proceso A
A1
A2
A3
A4
A5

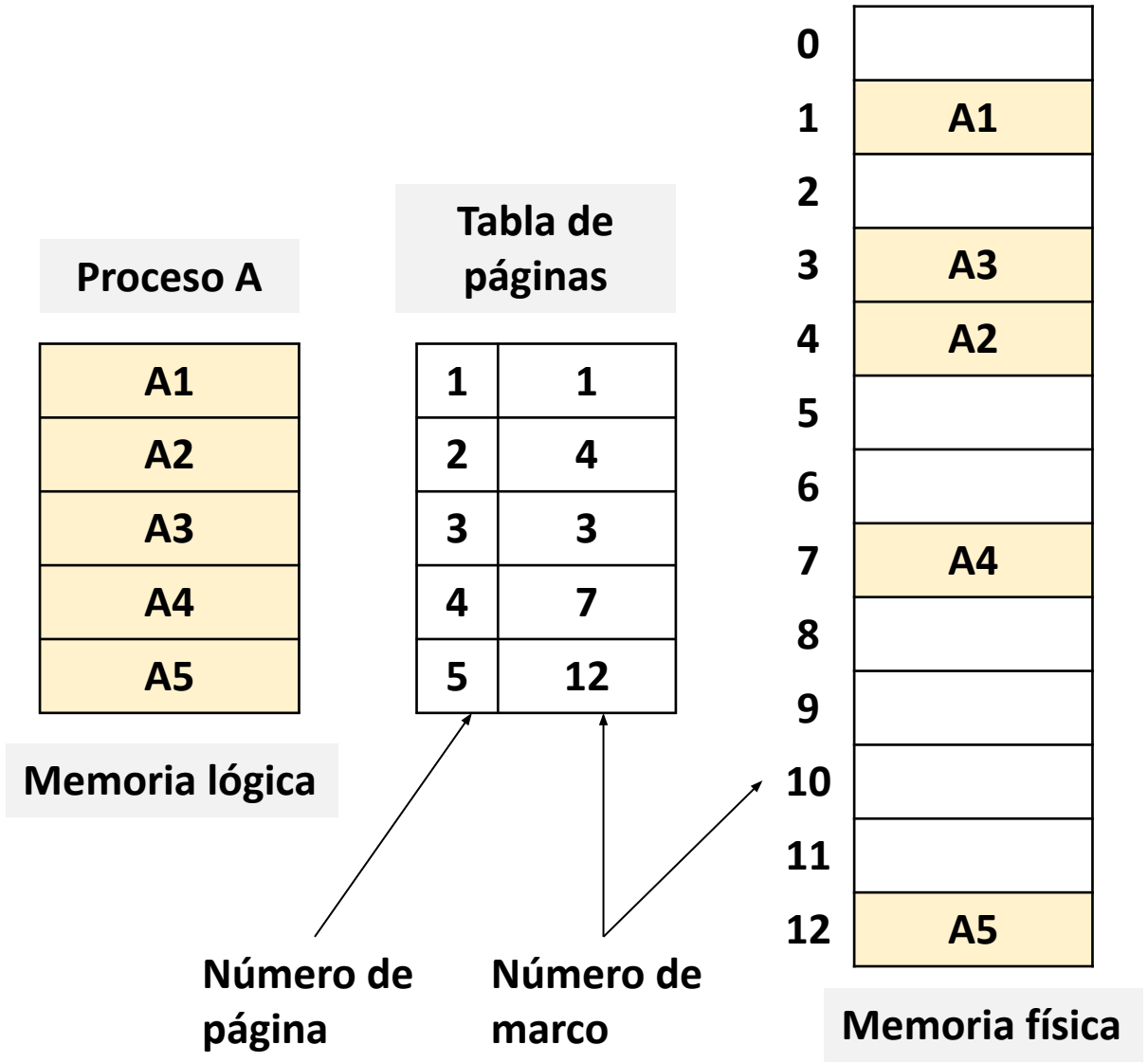


Paginación

- Se logra separar el espacio lógico del espacio físico.
- Podría existir más espacio lógico que físico.
  - Proceso podría pensar que tiene más de 2^{64} bytes direccionables.
  - Memoria física podría ser de menos de 2^{64} bytes.

Traducción de las direcciones

- Las direcciones virtuales generadas por la CPU necesitan traducción.
- Para la traducción se usa una estructura llamada **tabla de páginas**.
- La **tabla de páginas** es **por cada proceso**.



Traducción de las direcciones

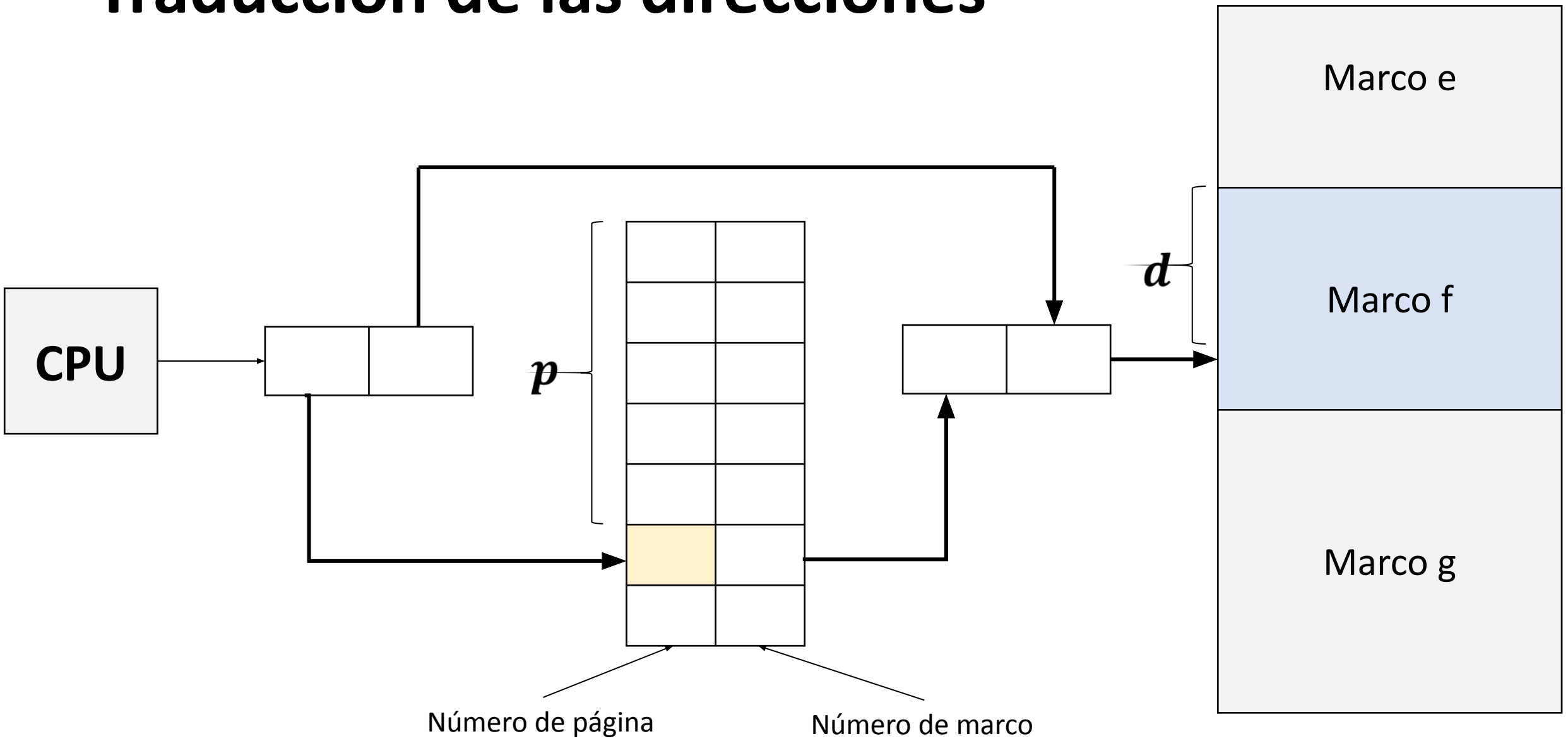
- Las direcciones **lógicas** se dividen en
  - Número de página (p)
  - Desplazamiento (d)
- p se usa como índice en la tabla de páginas de cada proceso.
- d se usa para referenciar la información al interior de cada marco.

Número de página	Desplazamiento

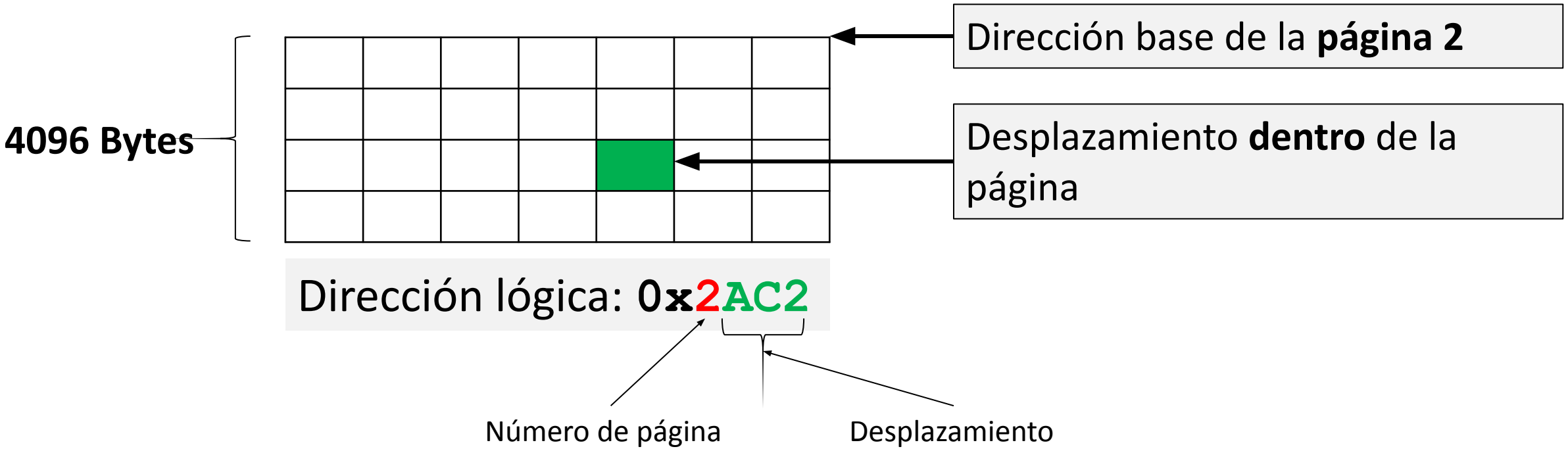
1. Obtener el número de página y usarlo como índice en la tabla de páginas.
2. Obtener el número de marco (f).
3. Reemplazar p por f. Esta es la dirección física.

- d se mantiene igual en la dirección.

Traducción de las direcciones



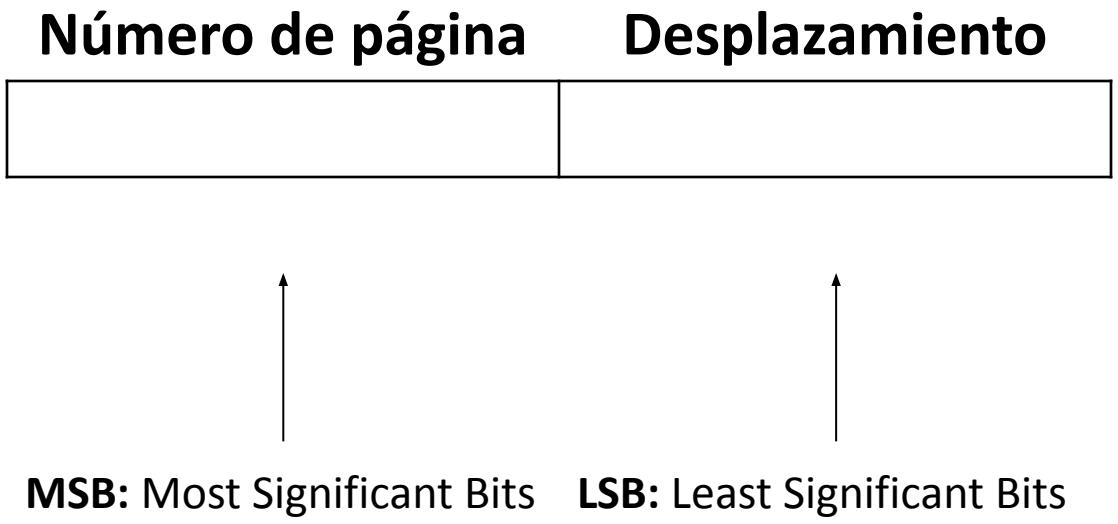
¿Qué es el desplazamiento?



- El desplazamiento indica cuánto hay que desplazarse desde la dirección base de la página hasta el byte que se está referenciando.
- La memoria se puede ver como un arreglo de bytes donde cada byte tiene una dirección de memoria.

¿Cuáles bits son de p y cuáles de d?

- Suponga espacio **lógico** de direccionamiento de 2^m
- Suponga tamaño de páginas de 2^n
- Dirección lógica



Proceso A

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

4 Bytes

Memoria lógica

Tabla de páginas

0	5
1	6
2	1
3	2

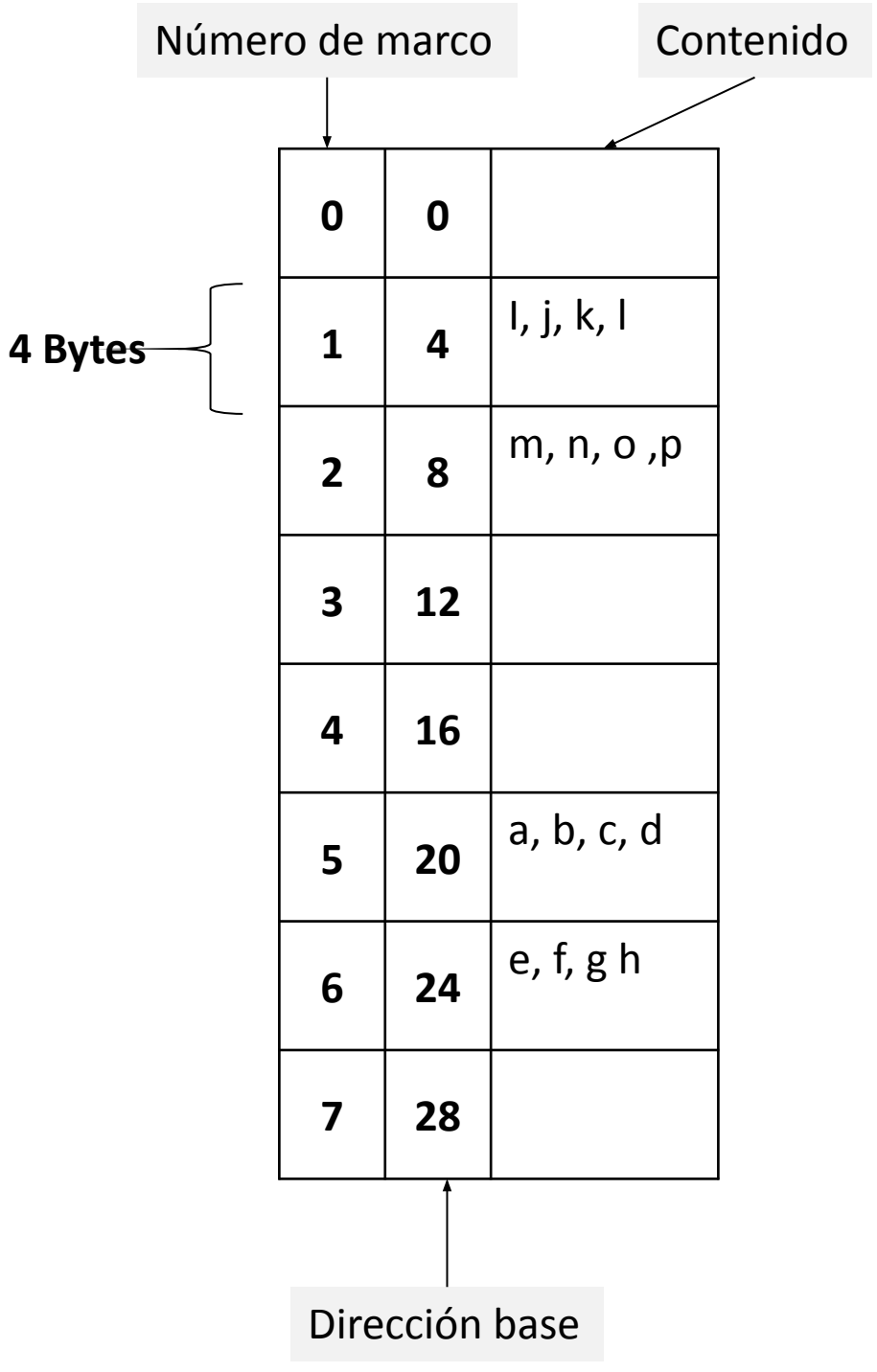
$n = 2 \rightarrow 2^2 = 4$ Bytes

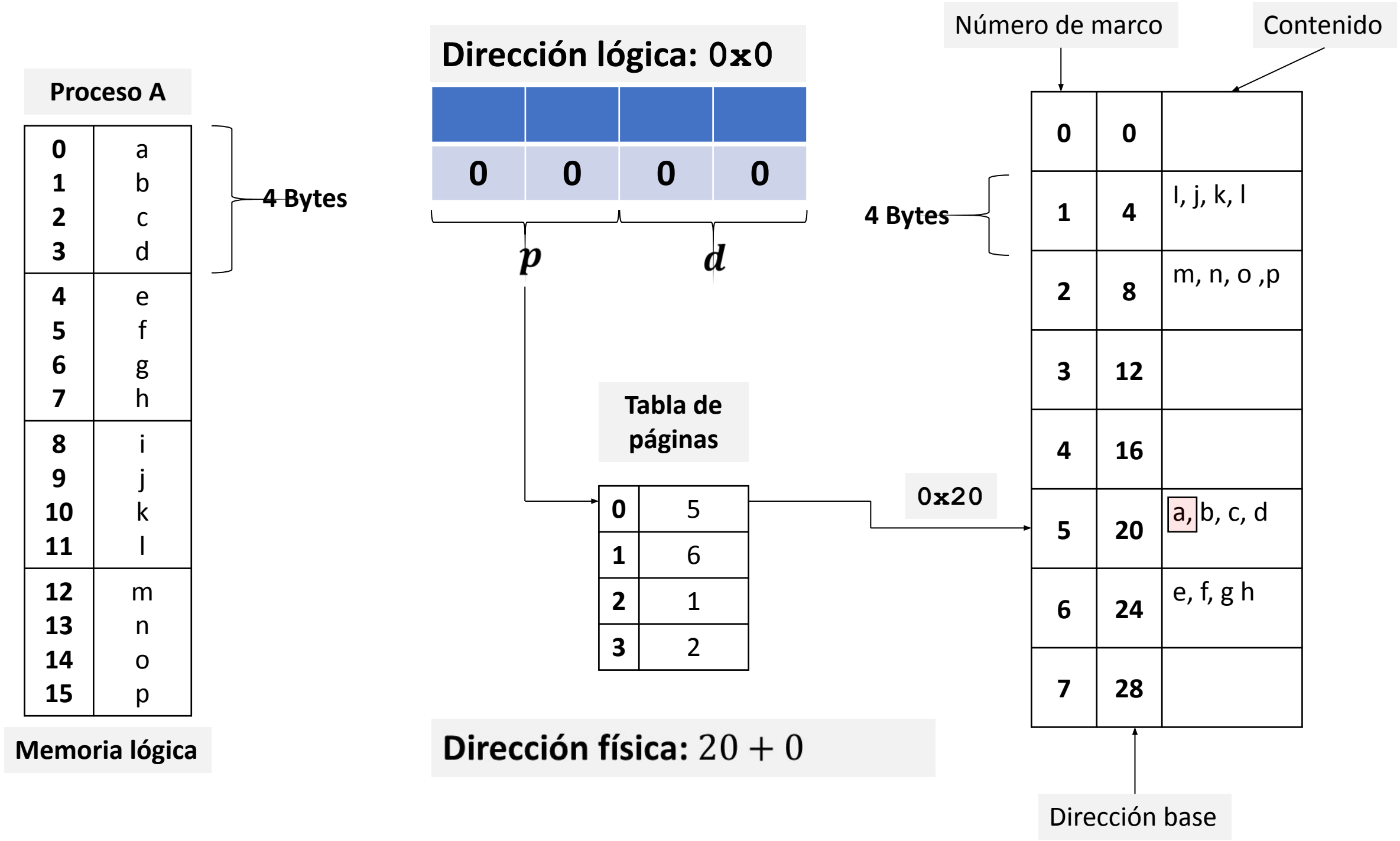
$m = 4 \rightarrow 2^4 = 16$ Bytes

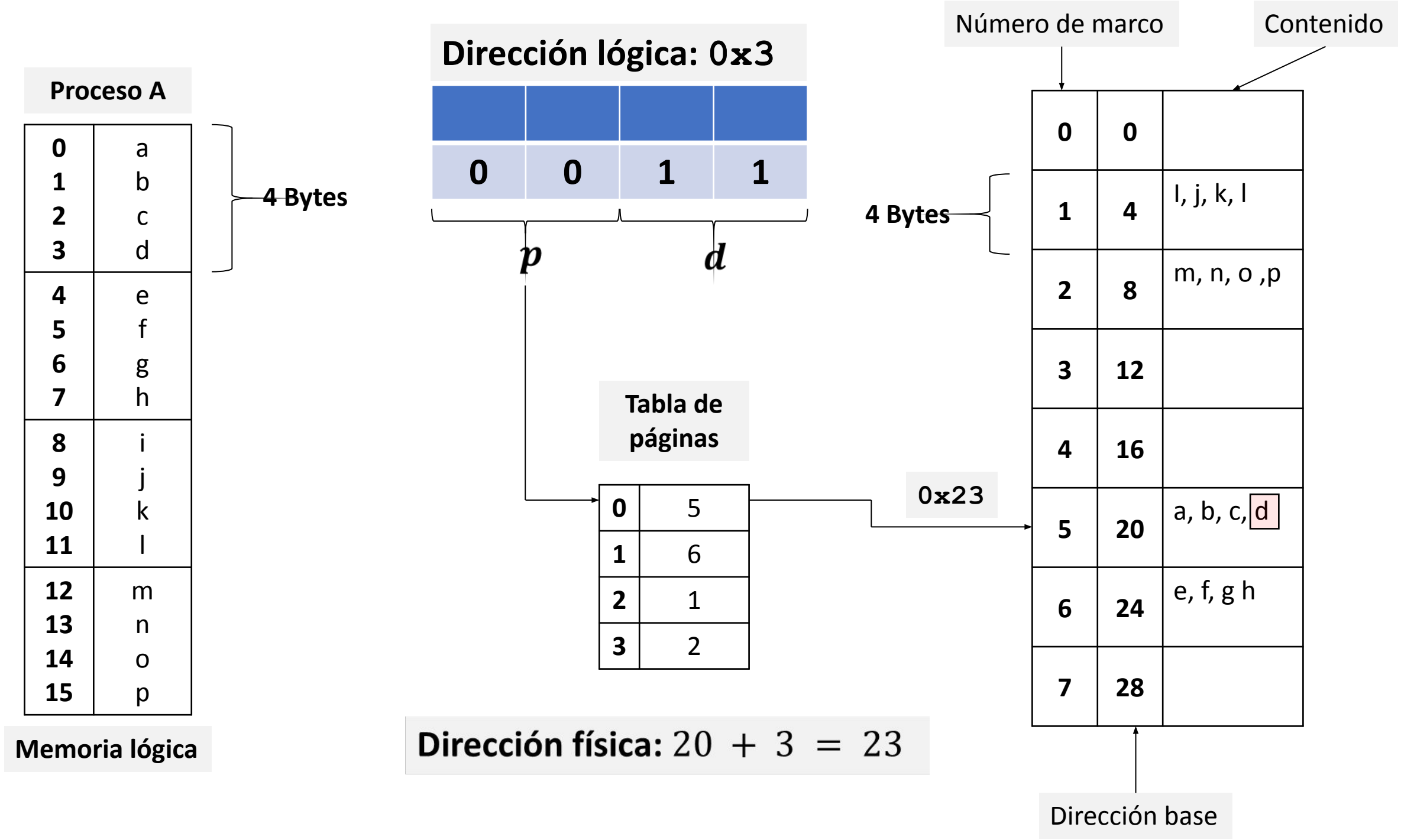
- **Dirección lógica de 4 bits, ya que**

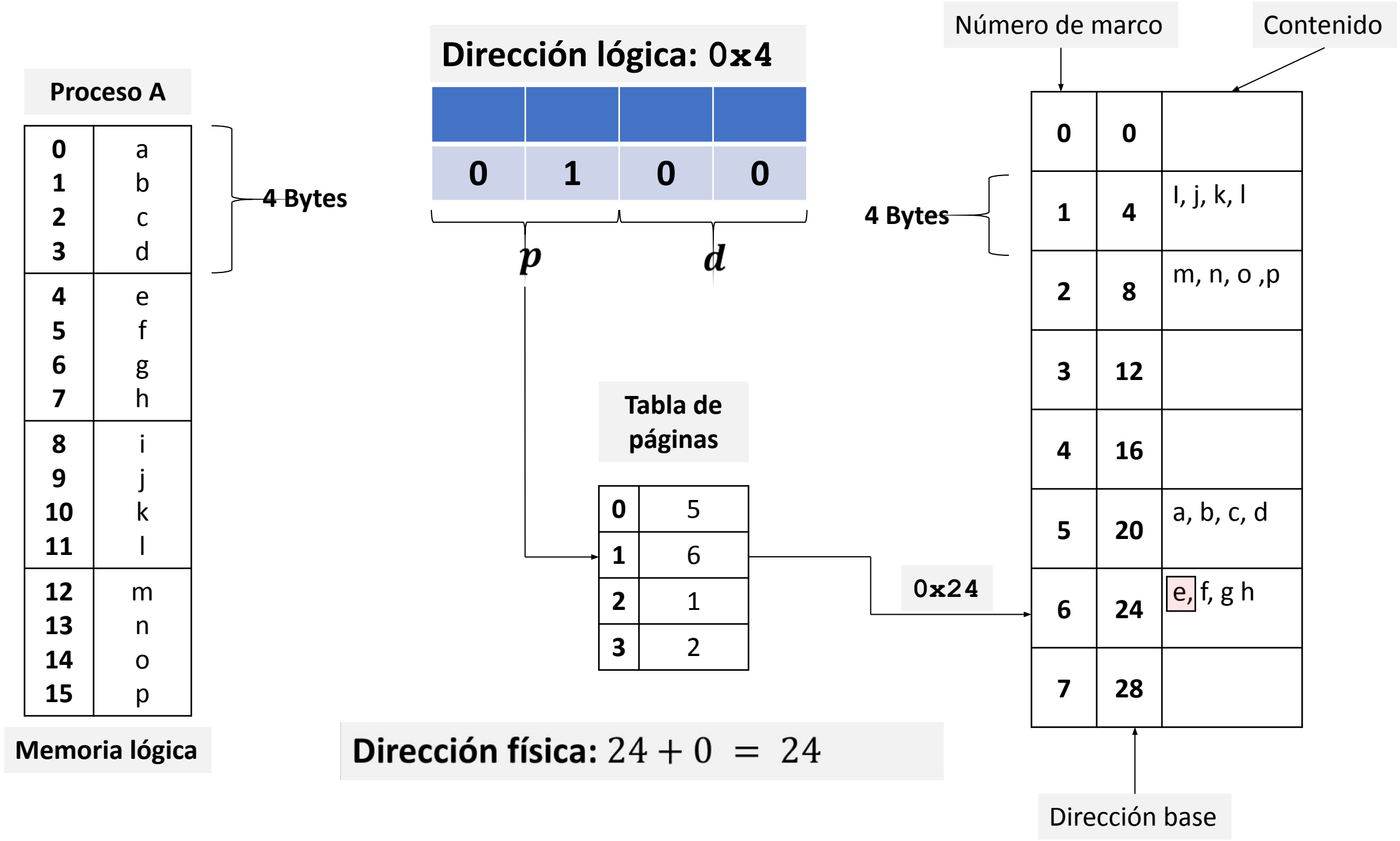
$p: m - n = 2$ Bits

$d: n = 2$ Bits




Proceso A
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
4 Bytes
Memoria lógica
Dirección lógica: 0x0
0 0 0 0
p
d
Tabla de páginas
0 5
1 6
2 1
3 2
Número de marco
Contenido
4 Bytes
0 0
1 4 l, j, k, l
2 8 m, n, o ,p
3 12
4 16
5 20 a, b, c, d
6 24 e, f, g h
7 28
0x20
Dirección física: 20 + 0
Dirección base


Proceso A
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
4 Bytes
Memoria lógica
Dirección lógica: 0x3
0 0 1 1
p
d
Tabla de páginas
0 5
1 6
2 1
3 2
Número de marco
Contenido
4 Bytes
0 0
1 4 l, j, k, l
2 8 m, n, o ,p
3 12
4 16
5 20 a, b, c, d
6 24 e, f, g h
7 28
0x23
Dirección física: 20 + 3 = 23
Dirección base


Proceso A
0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
4 Bytes
Memoria lógica
Dirección lógica: 0x4
0 1 0 0
p
d
Tabla de páginas
0 5
1 6
2 1
3 2
Número de marco
Contenido
4 Bytes
0 0
1 4 l, j, k, l
2 8 m, n, o ,p
3 12
4 16
5 20 a, b, c, d
6 24 e, f, g h
7 28
0x24
Dirección base
Dirección física: 24 + 0 = 24

Proceso A

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

4 Bytes

Memoria lógica

Dirección lógica: 0xD

1	**1**	**0**	**1**

p d

Tabla de páginas

0	5
1	6
2	1
3	2

Dirección física: $8 + 1 = 9$

Número de marco

Contenido

4 Bytes

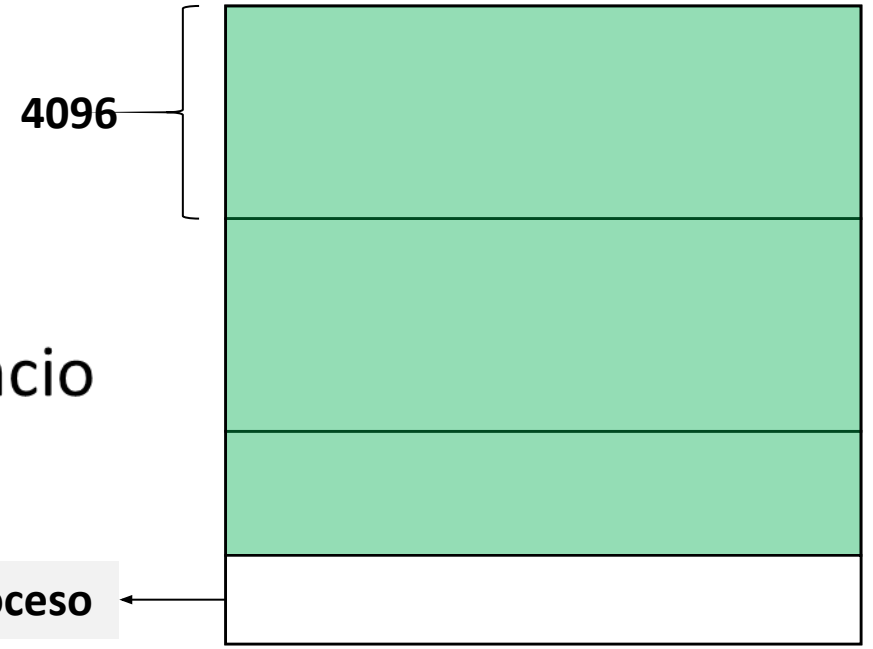
0x9

0	**0**	
1	**4**	l, j, k, l
2	**8**	m, n, o ,p
3	**12**	
4	**16**	
5	**20**	a, b, c, d
6	**24**	e, f, g h
7	**28**	

Dirección base

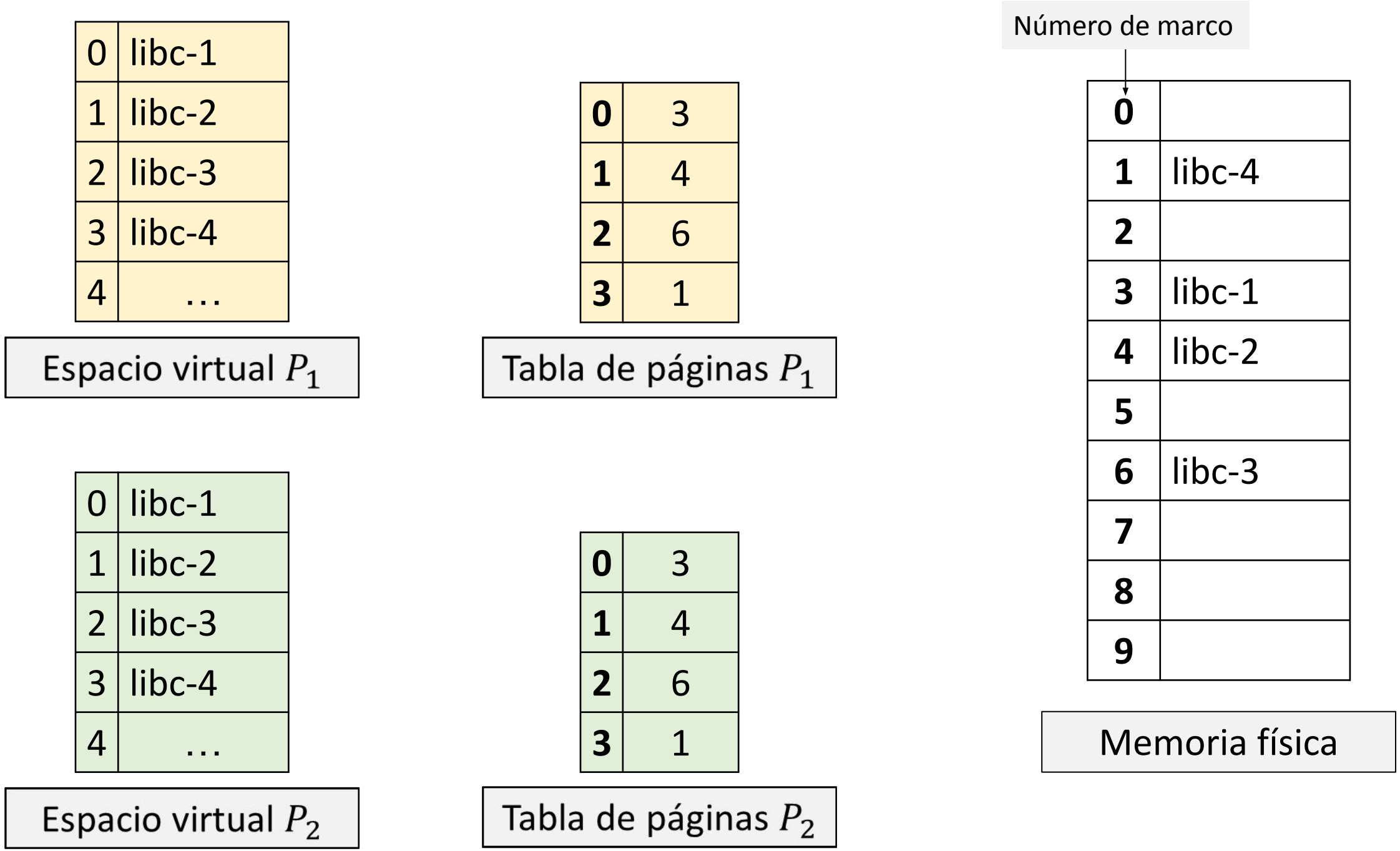
Fragmentación

- No ocurre fragmentación externa pero **SI** podría ocurrir fragmentación interna
- Tamaño de página: **4096 Bytes**
- Proceso mide **10240 Bytes**, necesita
  - $\frac{10240}{4096} = 2.5$ páginas
- Se le asignan tres páginas así le sobre espacio
  - Espacio que se le podría asignar si lo pide



Páginas compartidas

- Se posibilita compartir código entre los procesos
  - Shared Objects (.so)
  - Dynamic Link Libraries (.dll)
- Caso librería estándar de C (P. ej.: `libc-2.17.so`) en Linux
  - Una sola copia en memoria física


0 libc-1
1 libc-2
2 libc-3
3 libc-4
4 …
Espacio virtual P_1
0 3
1 4
2 6
3 1
Tabla de páginas P_1
Número de marco
0
1 libc-4
2
3 libc-1
4 libc-2
5
6 libc-3
7
8
9
Memoria física
0 libc-1
1 libc-2
2 libc-3
3 libc-4
4 …
Espacio virtual P_2
0 3
1 4
2 6
3 1
Tabla de páginas P_2

```
0000000000400000      4K r-x-- va
0000000000600000      4K r---- va
0000000000601000      4K rw--- va
0000000001df8000    132K rw---   [ anon ]
00007f851ee9e000   1808K r-x-- libc-2.17.so
00007f851f062000   2044K ----- libc-2.17.so
00007f851f261000     16K r---- libc-2.17.so
00007f851f265000      8K rw--- libc-2.17.so
00007f851f267000     20K rw---   [ anon ]
00007f851f26c000    136K r-x-- ld-2.17.so
00007f851f481000     12K rw---   [ anon ]
00007f851f48b000      8K rw---   [ anon ]
00007f851f48d000      4K r---- ld-2.17.so
00007f851f48e000      4K rw--- ld-2.17.so
00007f851f48f000      4K rw---   [ anon ]
00007fff38715000    132K rw---   [ stack ]
00007fff3878c000      8K r-x--   [ anon ]
ffffffffff600000      4K r-x--   [ anon ]
 total             4352K
```

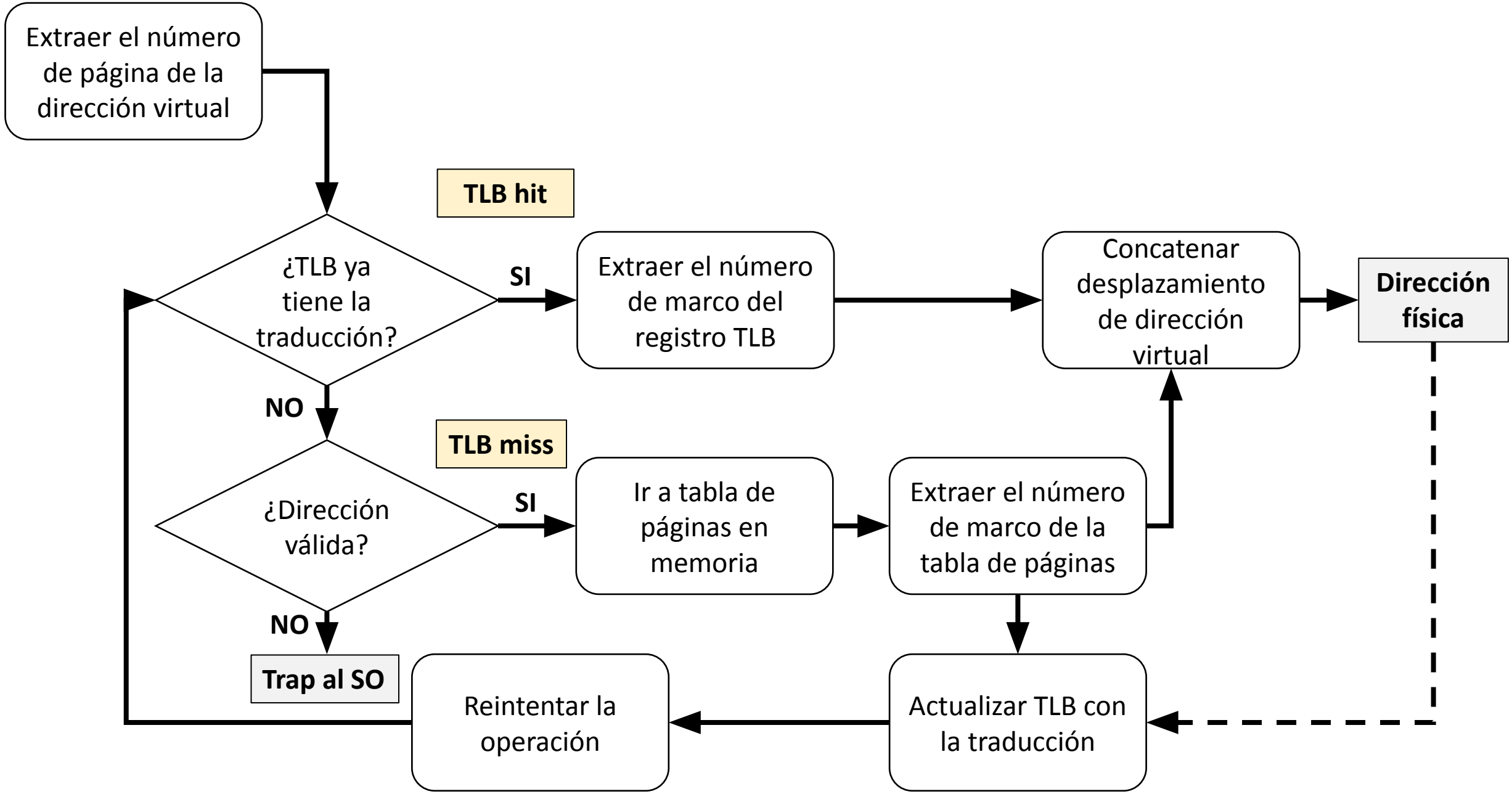

Heap

Librería estándar de C

Stack (gracias Faryd)

Translation-lookaside buffer (TLB)

- La tabla de páginas está implementada en memoria principal
- Cada referencia a memoria necesita una traducción a física
  - Esto implica un acceso adicional a memoria para consultar la tabla de páginas.
- TLB es un mecanismo de la MMU para caché de la traducción virtual a física.
  - Evita ir a consultar a memoria la tabla de páginas.
- Es único para cada proceso
  - En cambios de contexto se debe limpiar TLB o si hay memoria compartida se pueden tener las mismas entradas con un identificador único.


Extraer el número de página de la dirección virtual
TLB hit
¿TLB ya tiene la traducción?
SI
Extraer el número de marco del registro TLB
Concatenar desplazamiento de dirección virtual
Dirección física
NO
TLB miss
¿Dirección válida?
SI
Ir a tabla de páginas en memoria
Extraer el número de marco de la tabla de páginas
NO
Trap al SO
Reintentar la operación
Actualizar TLB con la traducción

Dirección	00	04	08	12	16
0	Página 00				
16	Página 01				
32	Página 02				
48	Página 03				
64	Página 04				
80	Página 05				
96	Página 06		**A[0]**	**A[1]**	**A[2]**
112	Página 07	**A[3]**	**A[4]**	**A[5]**	**A[6]**
128	Página 08	**A[7]**	**A[8]**	**A[9]**	
144	Página 09				
160	Página 10				
176	Página 11				
192	Página 12				
208	Página 13				
224	Página 14				
240	Página 15				

```
int A[10] = {0,1,2,3,4,5,6,7,8,9}
```

- Espacio virtual: $2^8 = 256$ Bytes
- Tamaño de paginas $2^4 = 16$ Bytes
- Dirección de 8 bits: $(8 - 4) + 4 = 8$ Bits

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

- `A[0]` está en la dirección virtual **100**
- Hardware extrae número de página desde dirección virtual 100
- Se verifica si **TLB** ya tiene la traducción para la dirección 100
  - Como no la tiene, **TLB miss**
  - Buscar tabla de páginas
- `A[1]` y `A[2]` están en la misma página, **TLB hit**
- `A[3]` produce de nuevo **TLB miss**
- `A[4]`,`A[5]`,`A[6]` Están en la misma página, **TBL hit**

- Tasa de aciertos (**TLB hit**): $7/10 = 70\%$
- Con páginas más grandes se puede mejorar la tasa de aciertos

Tabla de páginas

Página virtual	Marco de página	Otros bits

- Puede verse como un arreglo asociativo
- Otros bits
  - **Válido:** indica si la traducción es válida o no. Espacio no usado entre *heap* y *stack* se marca como no válido.
  - **Permisos:** lectura, escritura, ejecución
  - **Presente:** indica si la página está en memoria o está en disco
  - **Modificada:** indica si la página ha sido modificada desde que se llevó a memoria
  - **Referencia:** indica si la página ha sido referenciada para determinar qué páginas son populares (las no populares podrían ser llevadas a disco)

Referencias

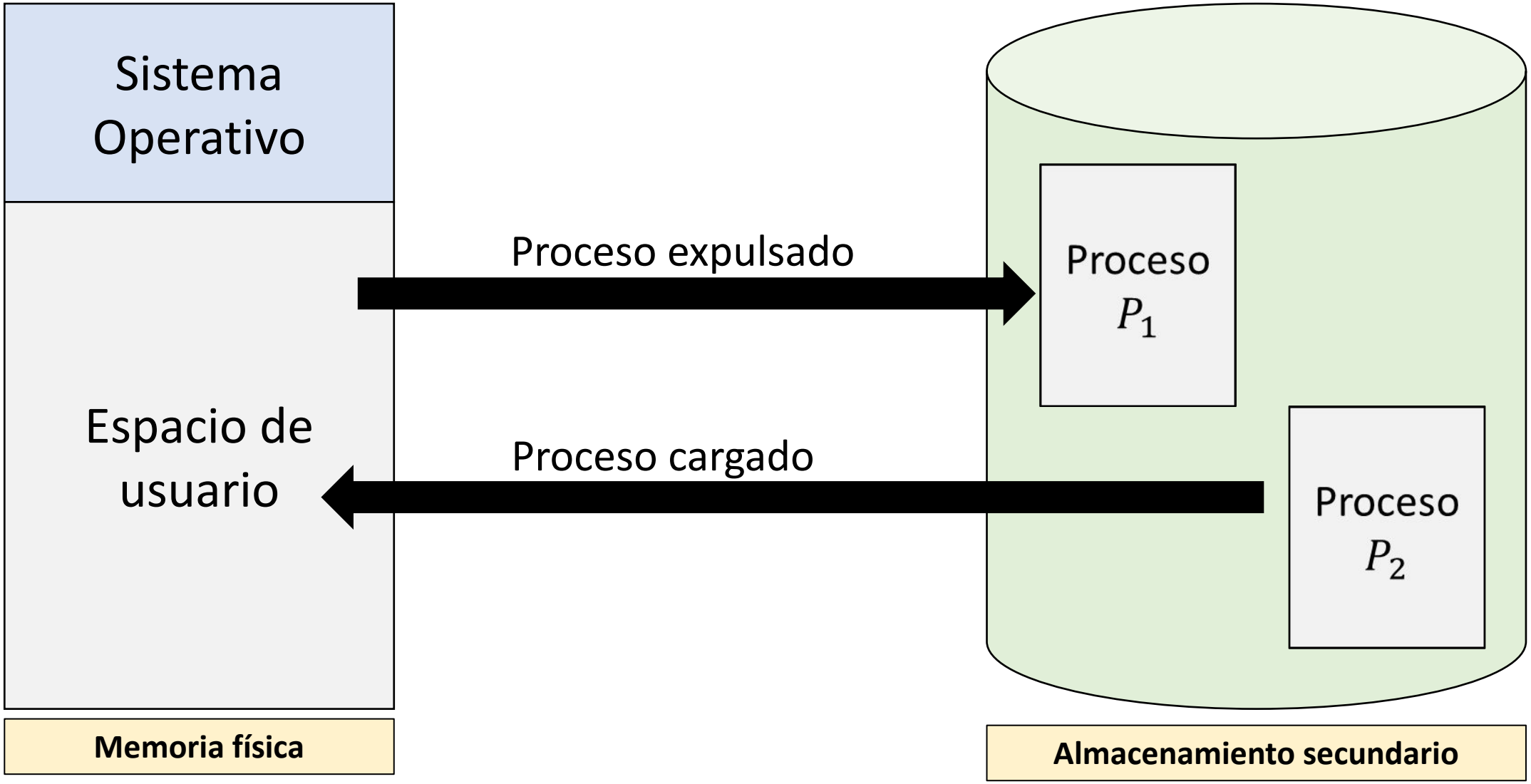
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. (2018). Paging: Faster Translations (TLBs). In *Operating Systems. Three Easy Pieces* (pp. 1–16). Arpaci-Dusseau Books.
- Silberschatz, A., Baer Galvin, P., & Gagne, G. (2018). Paging. In *Operating Systems Concepts* (10th ed., pp. 360–363). John Wiley & Sons, Inc.

Intercambio

Adaptación (ver referencias al final)

Intercambio

- Los procesos pueden ser expulsados temporalmente de memoria principal
  - A un almacenamiento secundario (p. ej.: disco)
- Permite a los procesos “ver” más memoria física de la que realmente hay en el sistema
  - Incrementa el grado de multi programación en el sistema.
  - Potencialmente más procesos pueden cargarse para su ejecución.
- Permite a los programadores “despreocuparse” de las limitantes de memoria física
  - No importa si cabe o no, siempre (en el mejor caso) habrá espacio.

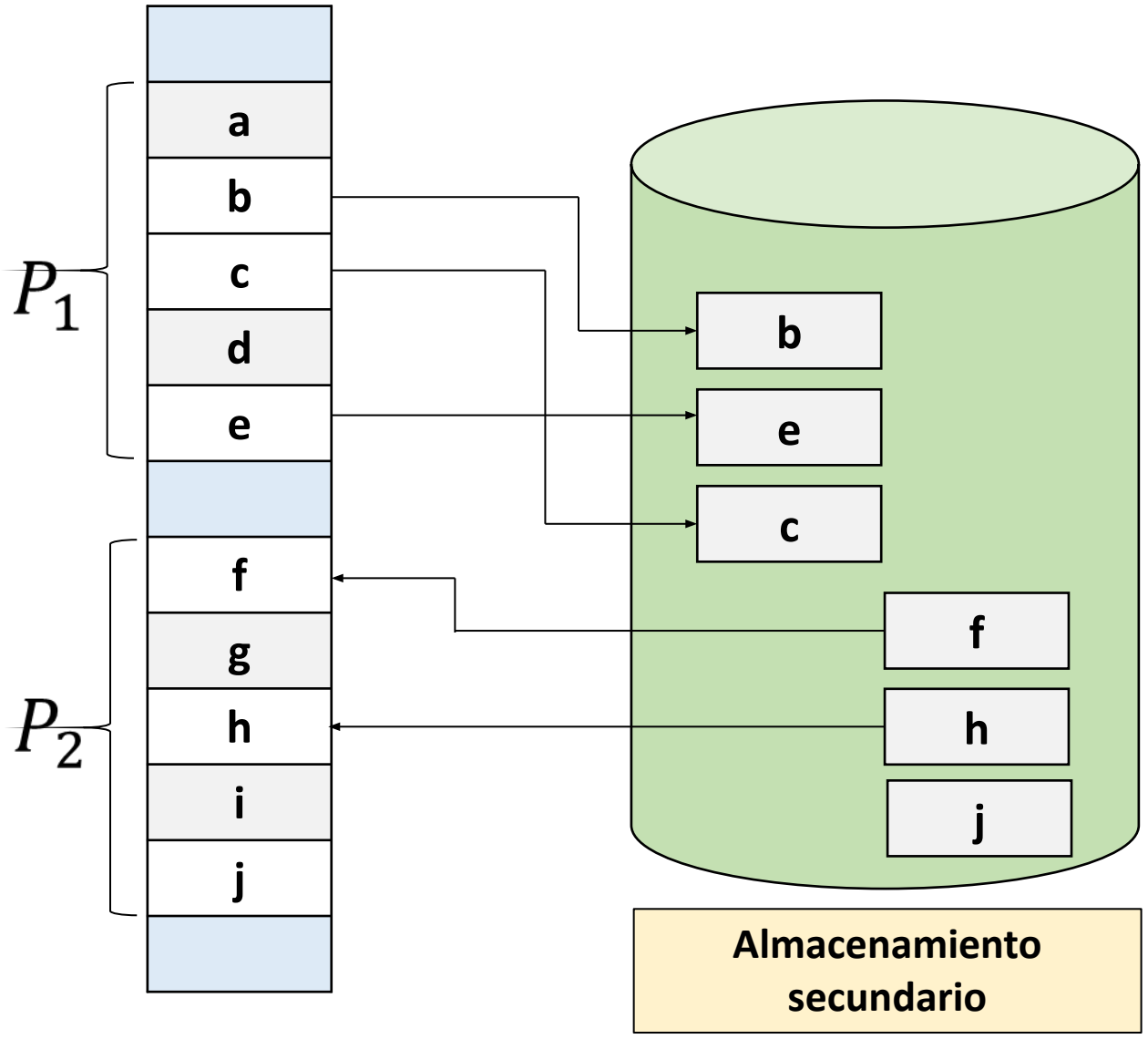

Sistema Operativo
Espacio de usuario
Memoria física
Proceso expulsado
Proceso cargado
Proceso P_1
Proceso P_2
Almacenamiento secundario

Intercambio sin paginación

- Implica mover TODO el proceso de memoria a disco y viceversa
  - Toda la imagen y el estado del proceso en memoria debe escribirse a disco
  - Requiere mucho tiempo mover todo el proceso
- S.O debe mantener metadatos del proceso que es expulsado de memoria
- La expulsión no implica que el proceso haya finalizado
- Permite un reúso (sobrevender) de la memoria
  - El sistema puede admitir más procesos que las limitantes de memoria física imponen
  - Procesos inactivos pueden ser buenos candidatos para expulsar de memoria
  - Procesos inactivos que se activen regresan a memoria

Intercambio con paginación

- Linux y Windows
- Se hace el intercambio de páginas
- Algunas se pueden quedar en almacenamiento secundario.



Intercambio con paginación

- Es más eficiente intercambiar páginas que TODO el proceso
- Cuando se presenta mucha paginación en un sistema
  - Indica que hay más procesos activos que memoria disponible
  - Solución 1: Comprar más memoria
  - Solución 2: Terminar procesos (liberar memoria)
- Páginas que no se demandan con mucha frecuencia son candidatas a llevar a almacenamiento secundario.
  - Para ello se puede usar el bit **Referencia** en la tabla de páginas y/o **TLB**.

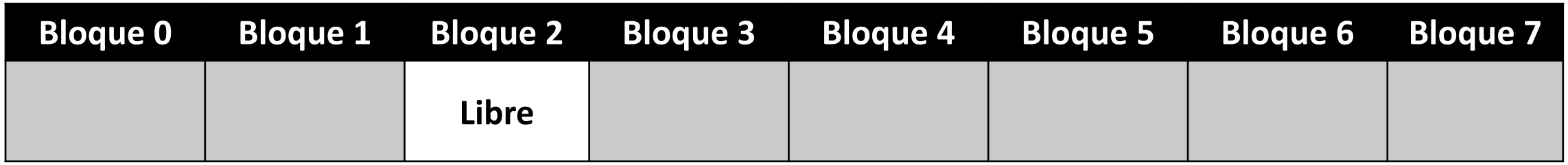
Espacio de intercambio

- Se debe reservar en disco un área de intercambio
  - *Swap space* (Linux, una partición del disco)
  - Archivo de paginación (Windows)
- S.O debe recordar la dirección en disco de una página de memoria
- El tamaño del área de intercambio debe ser lo suficientemente grande
  - Determina el número de páginas de memoria que pueden estar en uso en un sistema
- Los procesos pueden ser admitidos y todas sus páginas llevadas al área de intercambio
  - Las páginas se van cargando en memoria en la medida que se demandan

Espacio de intercambio

Memoria física

Existe un proceso admitido pero no cargado en memoria física



Área de intercambio

Fallos de página

- Se producen cuando se hace referencia a una dirección de memoria (virtual) de una página que no está en memoria física.
  - La página está en el área de intercambio.
  - Se usa el bit de **Presente** en la tabla de páginas o en el TLB.
- Sistemas operativo debe
  - Ubicar marcos de página disponible en memoria física.
  - Ubicar la página en disco (operación de E/S, proceso bloqueado).
  - Traerla de disco y asignarla en un marco disponible.
  - Actualizar la entrada en la tabla de páginas indicando que la página está disponible en memoria.

Memoria llena

- Debe existir una **política de reemplazo** para permitir llevar a memoria páginas que están en el área de intercambio.
- Se debe abrir espacio en memoria para permitir la ejecución de procesos
  - Llevar páginas poco usadas al área de intercambio.
- Expulsar incorrectamente páginas de memoria tienen un alto costo en desempeño.
  - Procesos con una tasa alta de fallos de página se ejecutan muy lento.

Liberar memoria

- Se usan dos umbrales para mantener memoria principal libre
  - Umbral bajo
  - Umbral alto
- Si S.O detecta que hay disponible menos memoria que **umbral bajo**
  - Se ejecuta hilo en *background* que lleva a área de intercambio páginas.
  - Se libera memoria hasta que se alcanza **umbral alto** y luego `sleep()`.
  - *Swap daemon* o *page daemon*.
- Las páginas que se llevan a memoria se llevan con alguna **política de reemplazo**.

¿Qué tanto espacio para *swap* en Linux?

RAM física	Espacio para swap recomendado	Espacio para swap recomendado para hibernación
≤ 2GB	dos veces la RAM	tres veces la RAM
2 GB – 8 GB	Igual a la RAM	Dos veces la RAM
8 GB – 64 GB	4 GB al 50% de la RAM	1.5 veces la RAM
> 64 GB	Mínimo 4GB	No se recomienda hibernación

Fuente: https://docs.fedoraproject.org/en-US/fedora/f28/install-guide/

IMPORTANTE: cuando el espacio de memoria virtual se esté agotando la solución **NO** es incrementarlo. La solución es añadir **mas memoria RAM.**

Referencias

- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. (2018). Beyond Physical Memory: Mechanisms. In *Operating Systems. Three Easy Pieces* (pp. 1–11). Arpaci-Dusseau Books.

Políticas de reemplazo de páginas

Adaptación (ver referencias al final)

Políticas de reemplazo de páginas

- Se requieren criterios de expulsión de páginas de memoria al área de intercambio (swap).
  - Cuando el espacio libre en RAM empieza a agotarse
  - Se incrementa en número de procesos que requieren tiempo de CPU
- Estos criterios se denominan políticas de reemplazo
- Responden a la pregunta
  - ¿Cómo decidir cuál o cuáles páginas deben ser expulsadas de memoria?

Políticas de reemplazo de páginas

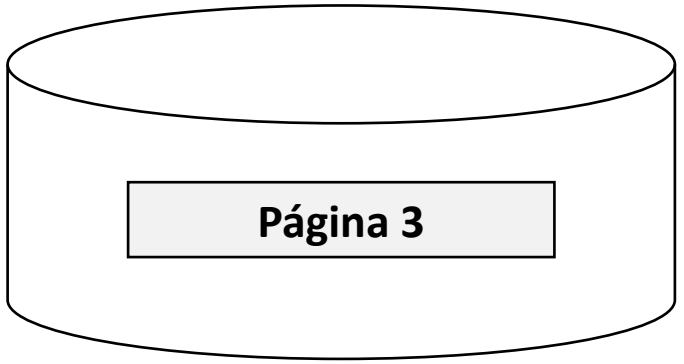
- Objetivos excluyentes entre sí
  - **Minimizar** el número de veces que se tiene que traer un página del disco (***cache misses***). Cuando se hace referencia a una página que **NO** está en memoria.
  - **Maximizar** el número de veces que la página se encuentra en memoria (***cache hits***). Cuando se hace referencia a una página que **SI** está en memoria.

Tiempo medio de acceso a memoria

- $$AMAT = T_M + (P_{MISS} \cdot T_D)$$
- Donde,
  - T_M representa el costo de acceso a memoria (en unidades de tiempo)
  - T_D representa el costo de acceso a disco (en unidades de tiempo)
  - P_{MISS} representa la probabilidad/tasa de un ***cache miss***: no encontrar la página en memoria.

CPU

Referencia a memoria	Hit/Miss
Página 0	Hit
Página 1	Hit
Página 2	Hit
Página 3	**Miss**
Página 4	Hit
Página 5	Hit
Página 6	Hit
Página 7	Hit
Página 8	Hit
Página 9	Hit



Página 0
Página 4
Página 1
Página 2
Página 8
Página 6
Página 7
Página 5
Página 9

Memoria física

Tasa de hits: $\frac{9}{10} = 90\%$

Tasa de fallos: $\frac{1}{10} = 10\%$

$T_M = 100 \text{ ns}$

$T_D = 10 \text{ ms}$

$AMAT = T_M + (P_{MISS} \cdot T_D)$

$AMAT = \frac{100}{1 \times 10^6} \text{ms} + (0.1 \cdot 10\text{ms})$

$\boldsymbol{AMAT = 1.0001 \text{ ms}}$

Ejemplo

- El costo de acceso a disco es muy alto y domina la métrica **AMAT** de cualquier proceso.
  - Si tasa de se aproxima a 100% entonces AMAT = 100 ns
- Hay que evitar una alta tasa de fallos de página (*misses*)
  - El acceso a disco es costoso en términos de tiempo
  - **Mantener al mínimo la tasa de fallos de página**
  - Se requieren políticas adecuadas de reemplazo de páginas
  - ¿Por qué la página tres está en disco?

Política de reemplazo óptimo

- Reemplazar la página más lejana en el futuro
- Política que minimiza la tasa de fallos de página
- Ha demostrado ser la mejor política de reemplazo de páginas pero de difícil implementación práctica
  - Básicamente hay que adivinar el futuro
- Para el ejemplo a continuación
  - Tasa de hits: $\frac{6}{11} = 54\%$

Referencia	Hit/Miss	Expulsión	Estado de memoria
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	**Miss**	**2**	**0, 1, 3**
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	**Miss**	**3**	**0, 1, 2**
1	Hit		0, 1, 2

- La memoria inicia sin páginas: las tres primeras referencias son fallos (obligados)

Aquí el S.O debe tomar una decisión: La página **2** es la más lejana en el futuro, por lo que se expulsa

Aquí la página **0** también podría haberse expulsado: No hay referencia en el futuro

Política FIFO

- Páginas se acomodan en una cola cuando ingresan al sistema
- De fácil implementación
  - Una estructura tipo cola
- Se expulsa la página que esté de primera en la cola
  - La primera página en llegar
- No tiene en cuenta la importancia de las páginas
  - La expulsión tiene lugar de acuerdo con la disciplina FIFO

Referencia	Hit/Miss	Expulsión	Estado de memoria
0	Miss		Cola □ 0
1	Miss		Cola □ 0, 1
2	Miss		Cola □ 0, 1, 2
0	Hit		Cola □ 0, 1, 2
1	Hit		Cola □ 0, 1, 2
3	**Miss**	**0**	**Cola □ 1, 2, 3**
0	**Miss**	**1**	**Cola □ 2, 3, 0**
3	Hit		Cola □ 2, 3, 0
1	**Miss**	**2**	**Cola □ 3, 0, 1**
2	**Miss**	**3**	**Cola □ 0, 1, 2**
1	Hit		Cola □ 0, 1, 2

Tasa de éxitos: $\frac{4}{11} = 36.4\%$

Política aleatoria

- Seleccionar de manera aleatoria una página para la expulsión
- De fácil implementación
- Algunas veces será tan bueno como la política de reemplazo óptimo
  - Depende de la suerte

Referencia	Hit/Miss	Expulsión	Estado de memoria
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	**Miss**	**0**	**1, 2, 3**
0	**Miss**	**1**	**2, 3, 0**
3	Hit		2, 3, 0
1	**Miss**	**3**	**2, 0, 1**
2	Hit		2, 0, 1
1	Hit		2, 0, 1

Tasa de éxitos: $\frac{5}{11} = 45\%$

Política LRU: menos usada recientemente

- LRU: Least Recently Used
- Política que tiene en cuenta el historial del uso de las páginas
- Si una página se referencia muchas veces, probablemente es una página importante
- Una página recientemente usada es bastante probable que sea referenciada próximamente
  - Localidad espacial y temporal
- LFU (Least Frequently Used)
  - Página usada con menos frecuencia

Referencia	Hit/Miss	Expulsión	Estado de memoria
0	Miss		LRU 0
1	Miss		LRU 0, 1
2	Miss		LRU 0, 1, 2
0	Hit		LRU 1, 2, 0
1	Hit		LRU 2, 0, 1
3	**Miss**	**2**	**LRU 0, 1, 3**
0	Hit		LRU 1, 3, 0
3	Hit		LRU 1, 0, 3
1	Hit		LRU 0, 3, 1
2	**Miss**	**0**	**LRU 3, 1, 2**
1	Hit		LRU 3, 2, 1

La menos usada va quedando de primera y es la que se expulsa

La más recientemente usada se coloca al final

Otras políticas basadas en el historial de uso

- Políticas opuestas a LFU y LRU
  - Ignoran el principio de localidad espacial y temporal
  - En la mayoría de los casos no se desempeñan bien
- MFU (Most Frequently Used)
  - Página más utilizada con frecuencia
- MRU (Most Recently Used)
  - Página más recientemente utilizada

Localidad espacial y temporal

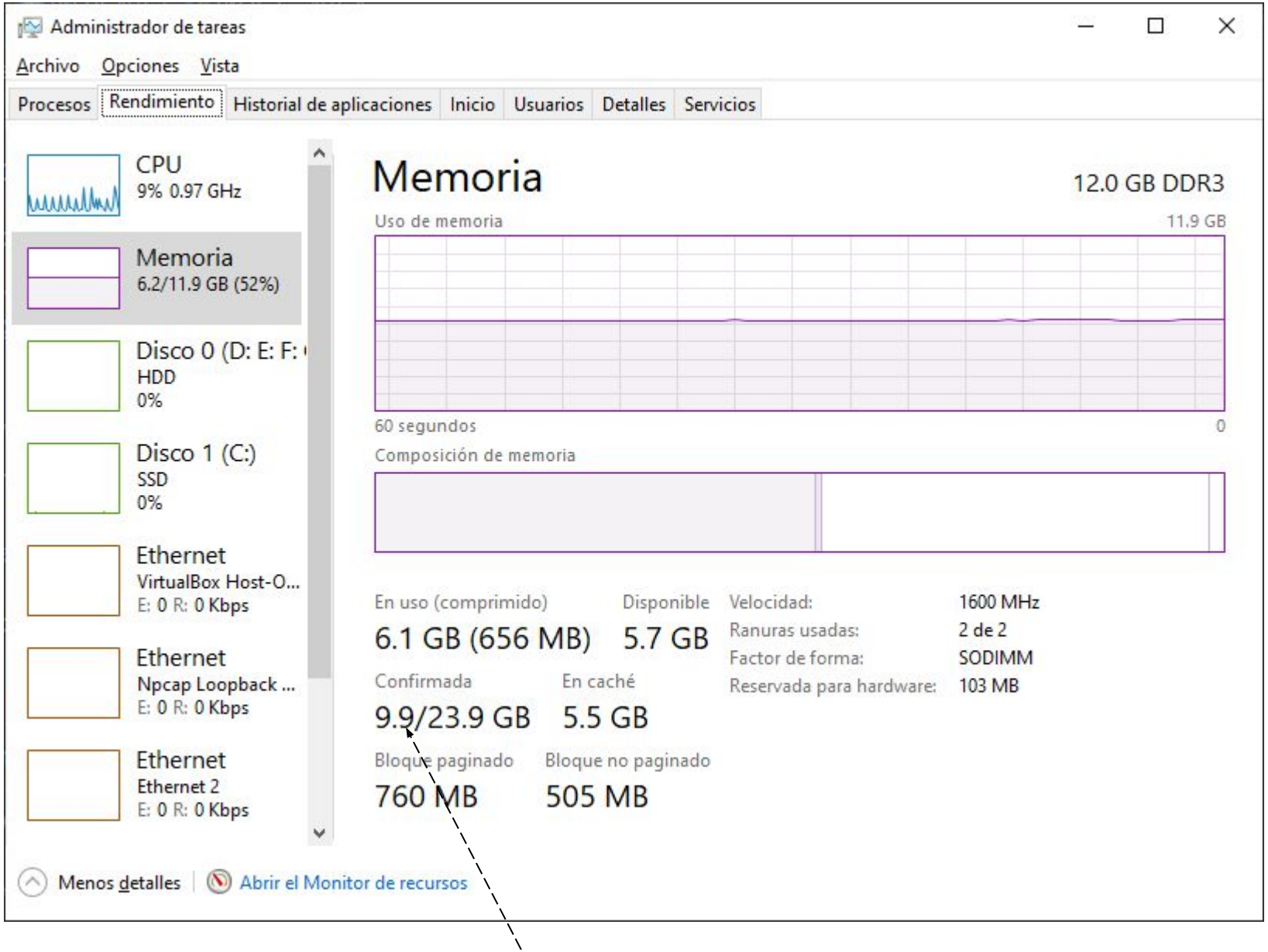
- Los programas tienden a acceder código de acceso secuencial con bastante frecuencia
  - Ciclos
  - Arreglos
- Localidad espacial
  - Si una página $\boldsymbol{P}$ es referenciada, es probable que las páginas que están cerca a $\boldsymbol{P}$ ($\boldsymbol{P+1}$, $\boldsymbol{P-1}$) también sean referenciadas.
- Localidad temporal
  - Una página referenciadas en el pasado cercano, es probable que sea referenciada de nuevo en el futuro cercano
- El principio de localidad es una heurística, no es una regla.

Implementación de LRU

- Se necesita llevar un registro en cada referencia a memoria
  - Resulta costoso si se implementa una solución por software
  - Se requiere mover las páginas conforme se van referenciando
- Para mejorar el desempeño se necesita apoyo del hardware
  - Bit de referencia
- Bit de referencia
  - Cuando una página se lee o se escribe el bit de referencia se establece a 1
  - Solo el S.O puede establecer el bit de referencia a 0.
- Política de reemplazo revisa bit de referencia
  - Si bit es igual 1, página no es candidata a reemplazo

Otras políticas de administración de memoria

- Política de selección de página
  - Cuándo llevar una página del disco a la memoria RAM
- Usualmente se usa una política **bajo demanda**
  - Página que se referencia página que se lleva a memoria
- Criterio se combina con principio de localidad espacial para anticipar la siguiente página
  - Página $\boldsymbol{P}$ referenciada, llevar a memoria de una vez página $\boldsymbol{P} + \mathbf{1}$
- Políticas de escritura de página en disco
  - Seleccionar varias página (grupo) para hacer una sola escritura



Límite: S.O abre espacio en la memoria **TOTAL** por si se necesita.

Confirmada: memoria física en uso para la que se ha reservado espacio en el archivo de paginación. Suma total de memoria virtual + memoria física.

Bloque paginado: páginas que se encuentran en memoria virtual (disco). P. Ej.: de procesos inactivos.

Bloque no paginado: páginas de memoria que deben quedarse en RAM.

Referencias

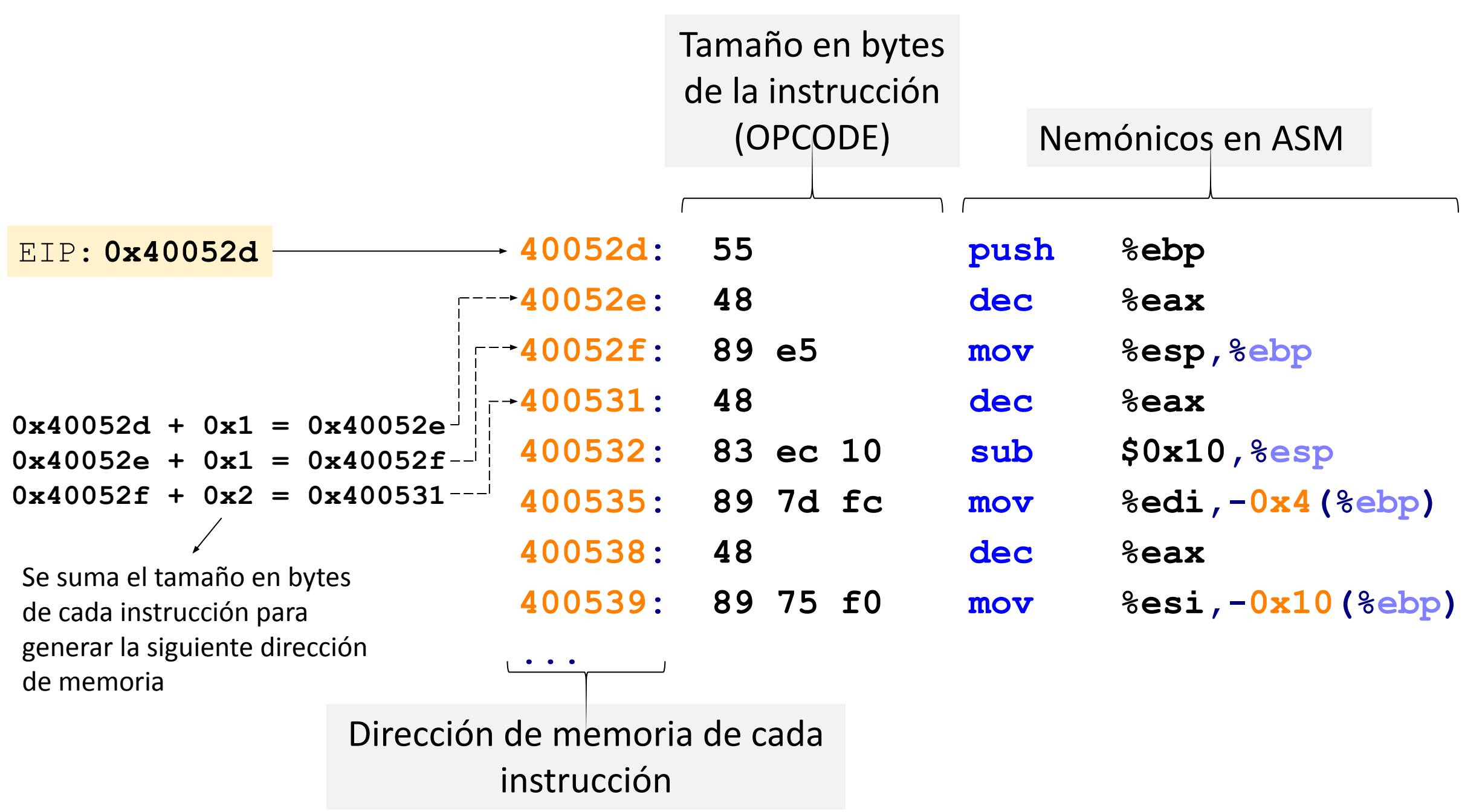
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. (2018). Beyond Physical Memory: Policies. In *Operating Systems. Three Easy Pieces*. Arpaci-Dusseau Books.

Segmentación de memoria

Un ejemplo práctico

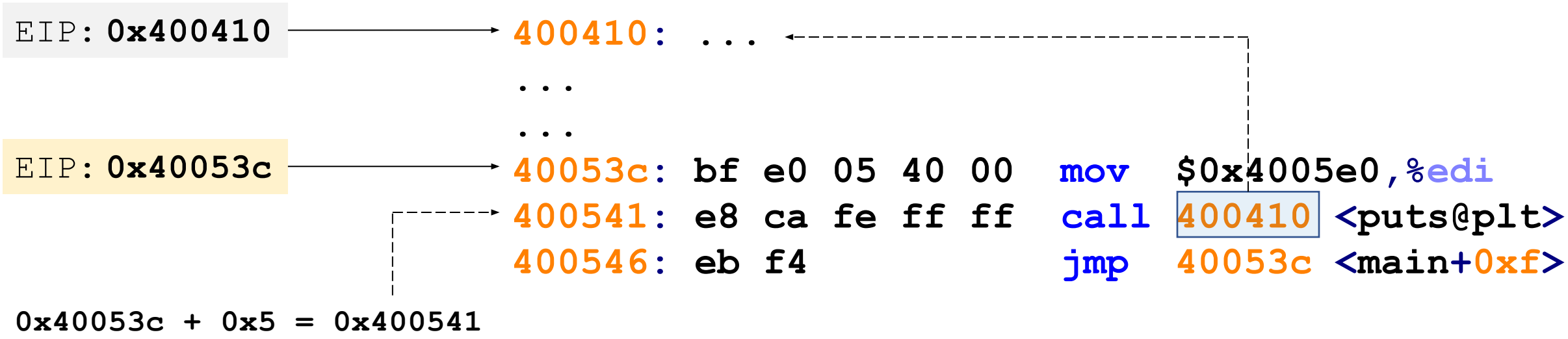
Segmentación de un programa en memoria

- Segmento de código
  - Código del programa
- Ciclo de ejecución del procesador
  1. Leer la instrucción a la que apunta el registro EIP (PC: *Program Counter*)
  2. Sumar a EIP la longitud en bytes de la instrucción leída
     - Esta suma hace que el EIP apunte a la siguiente instrucción en memoria.
  3. Ejecutar la instrucción leída en el paso No. 1
  4. Repetir de nuevo desde el paso No. 1.


Tamaño en bytes de la instrucción (OPCODE)
Nemónicos en ASM
EIP: 0x40052d
40052d: 55 push %ebp
40052e: 48 dec %eax
40052f: 89 e5 mov %esp,%ebp
400531: 48 dec %eax
400532: 83 ec 10 sub $0x10,%esp
400535: 89 7d fc mov %edi,-0x4(%ebp)
400538: 48 dec %eax
400539: 89 75 f0 mov %esi,-0x10(%ebp)
...
0x40052d + 0x1 = 0x40052e
0x40052e + 0x1 = 0x40052f
0x40052f + 0x2 = 0x400531
Se suma el tamaño en bytes de cada instrucción para generar la siguiente dirección de memoria
Dirección de memoria de cada instrucción

Segmentación de un programa en memoria

- Excepción al acceso secuencial
  - Saltos condicionales: **JNZ**, **JNE**, etc.
  - Saltos incondicionales: **JMP**
  - Llamadas a **funciones**: **CALL**



Segmentación de un programa en memoria

- Segmento *stack*
  - Variables locales de una función
  - Parámetros pasados a la función
  - Valor del **EIP** antes del llamado a una función: se *empuja* al stack

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

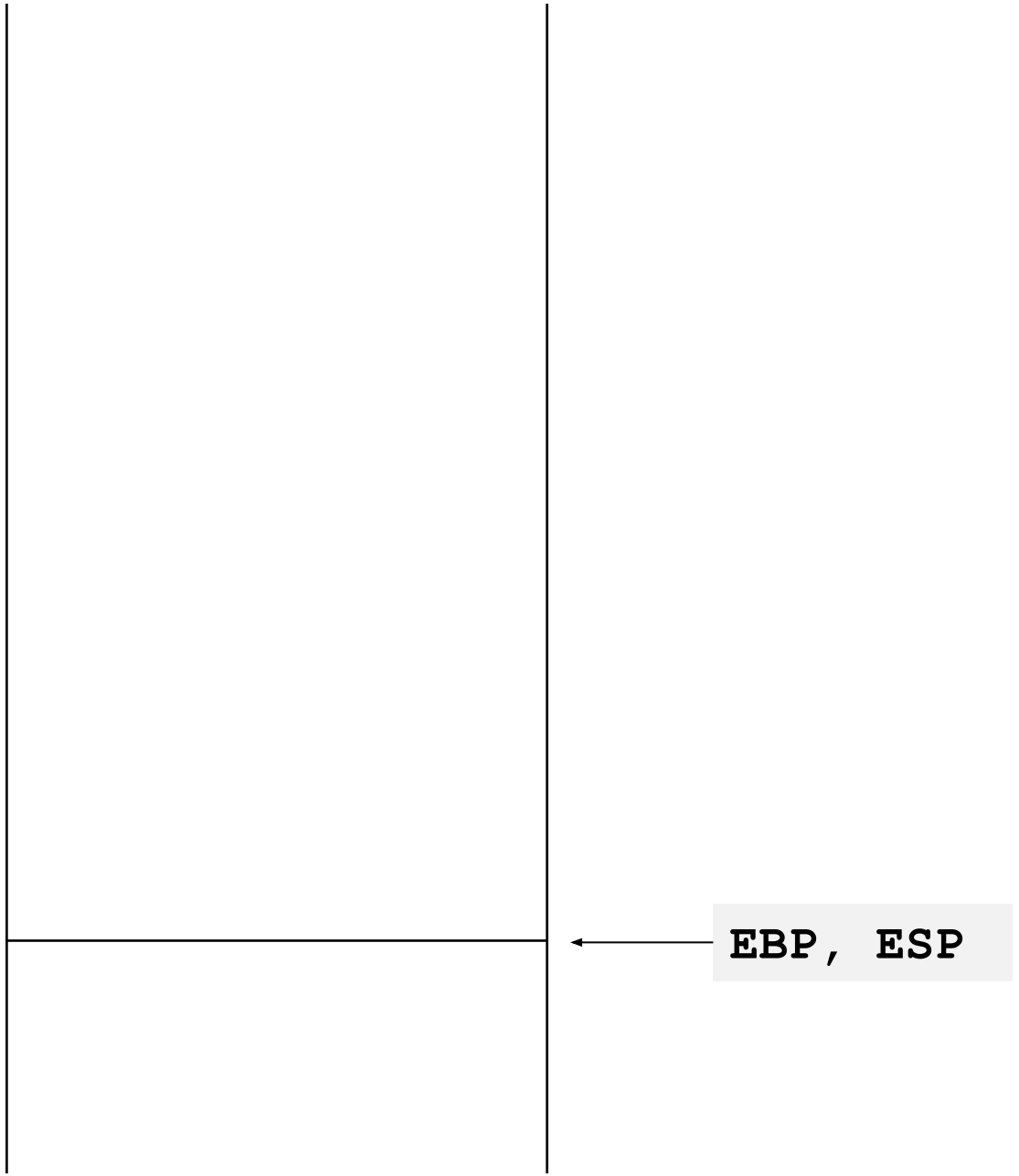
Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

Segmentación de un programa en memoria

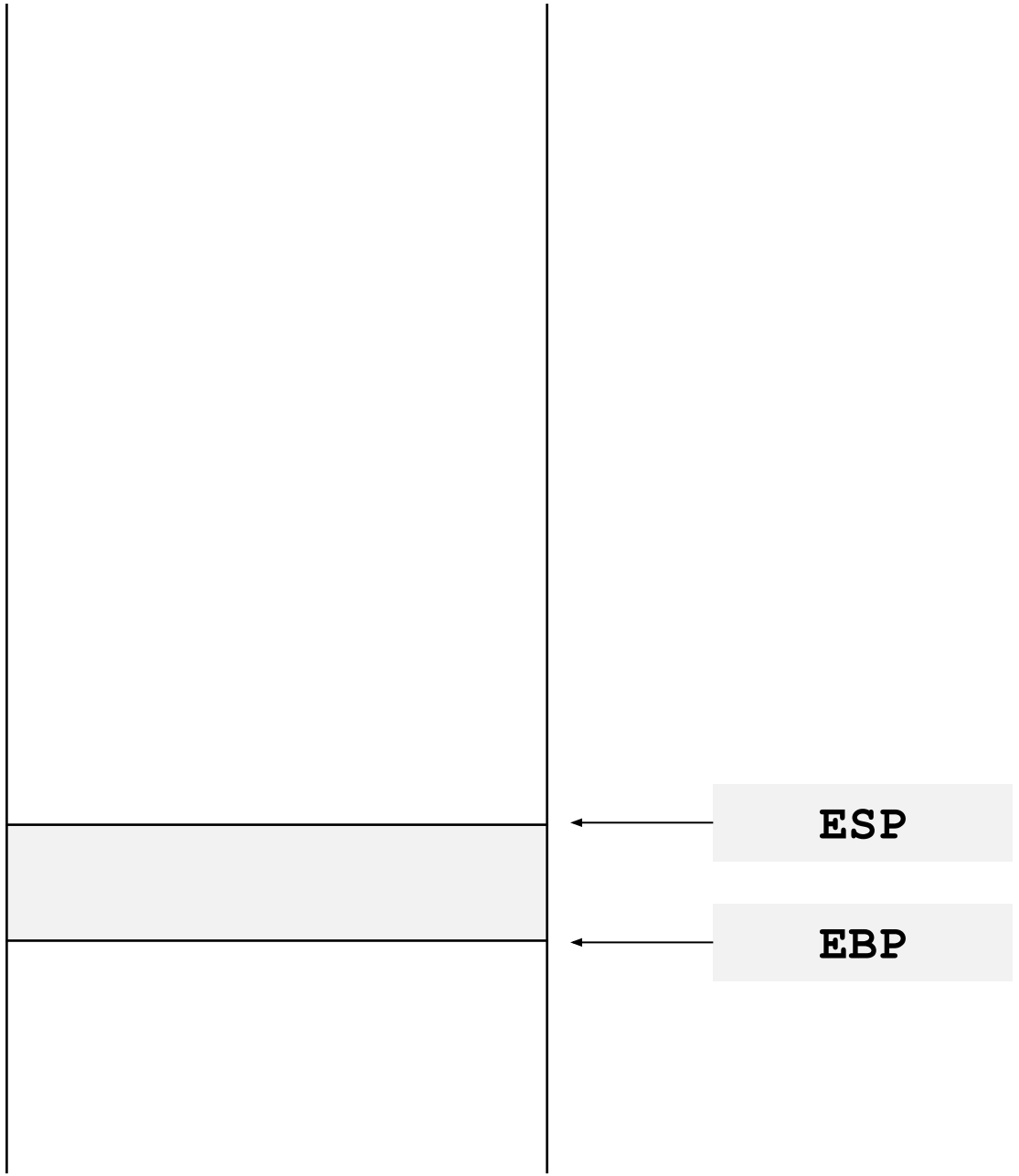
- Registro **ESP** (*stack pointer*) apunta a la dirección de memoria del último elemento en el segmento *stack.*
- Registro **EBP** se usa para indicar el inicio de un *stack frame*.
- La distancia entre el **EBP** y el **ESP** se denomina *stack frame*.
  - Cada función tiene su *stack frame*
- ¿Qué hay en el *stack frame*?
  - Variables locales de una función
  - Parámetros pasados a la función
  - Dirección de retorno (a donde retorna el flujo: valor del **EIP** *empujado*)
  - Apuntador al **EBP** antes del llamado (dónde estaba el **EBP** antes del llamado) .

STACK



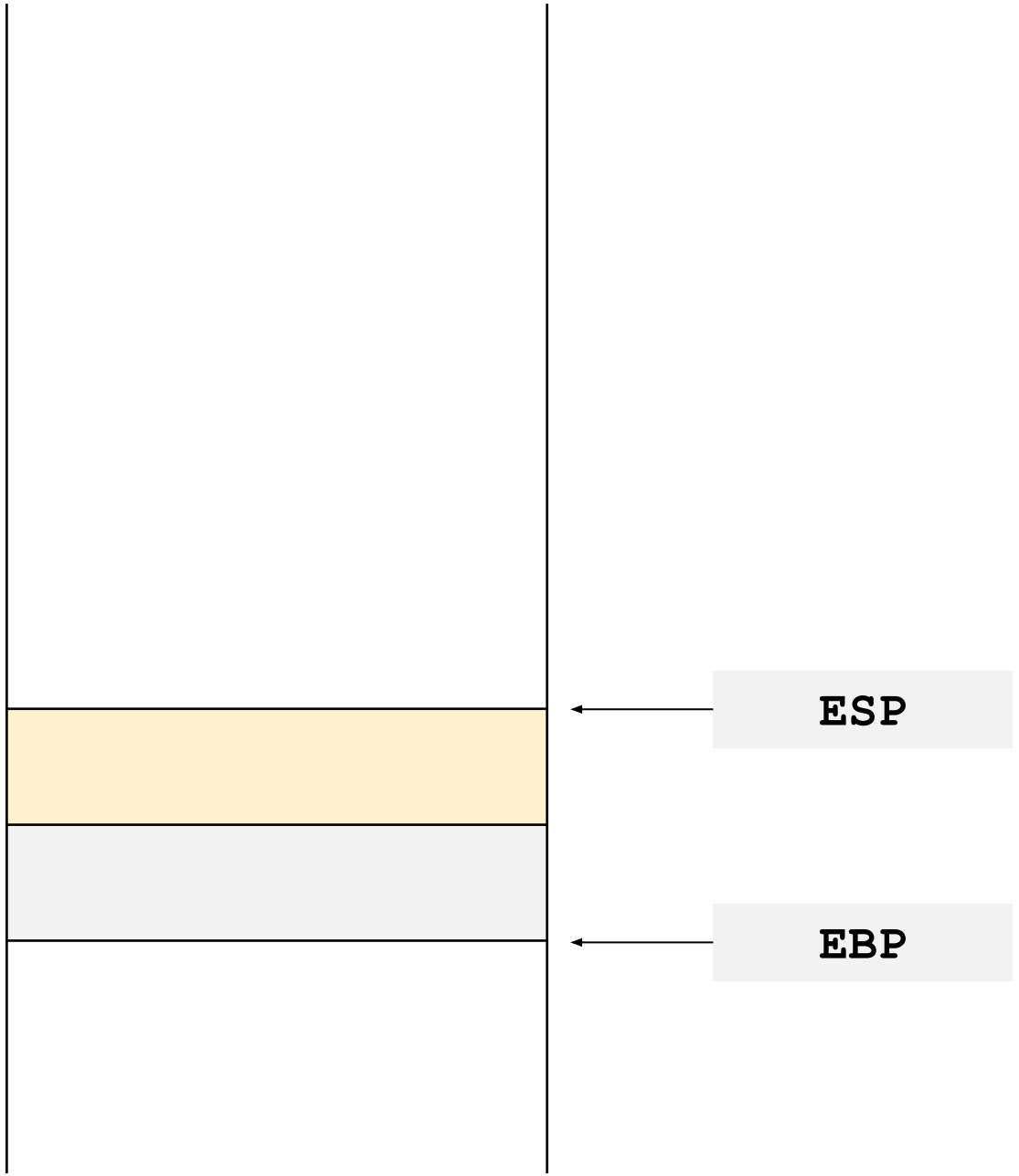
STACK

PUSH



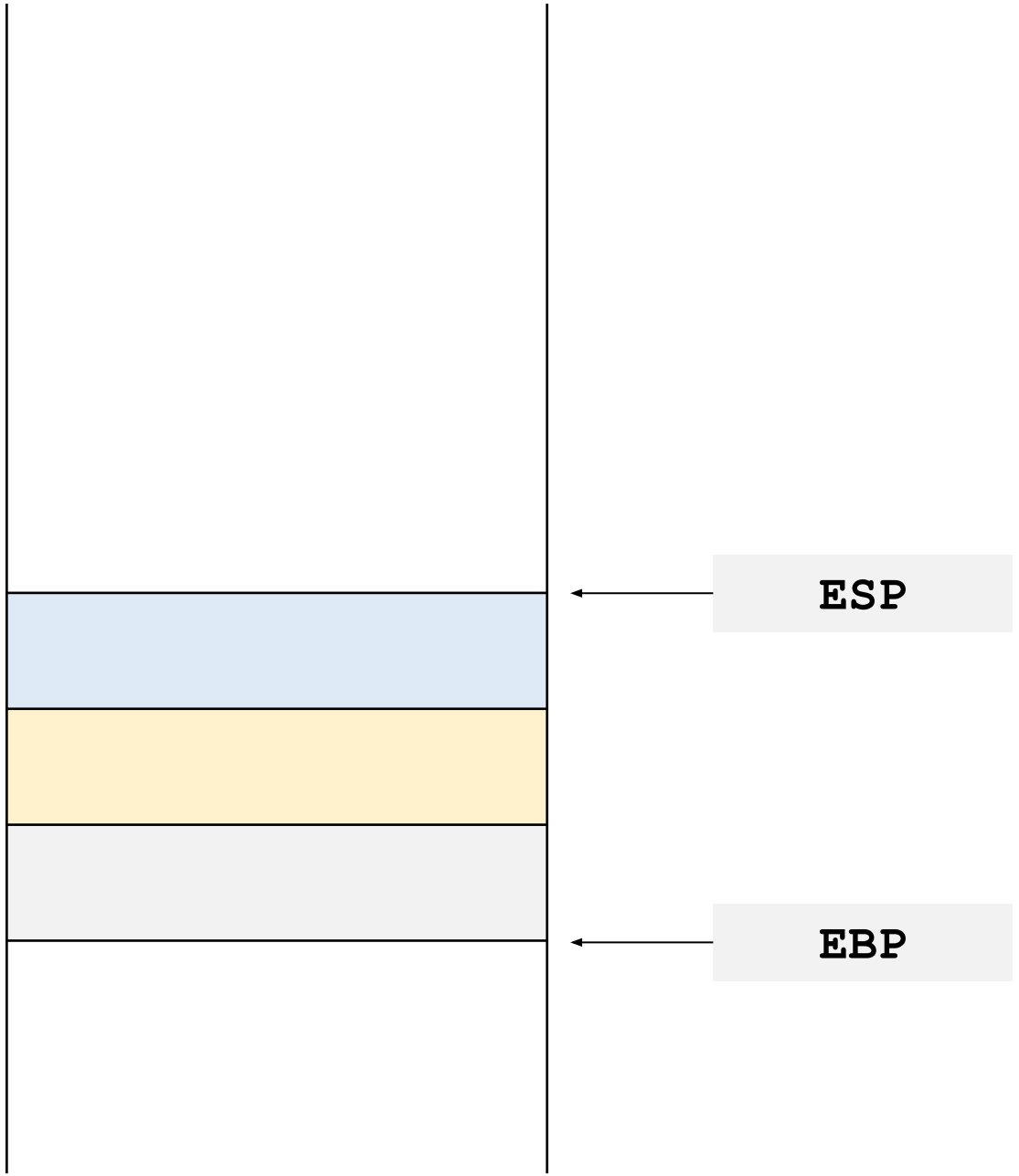
STACK

PUSH



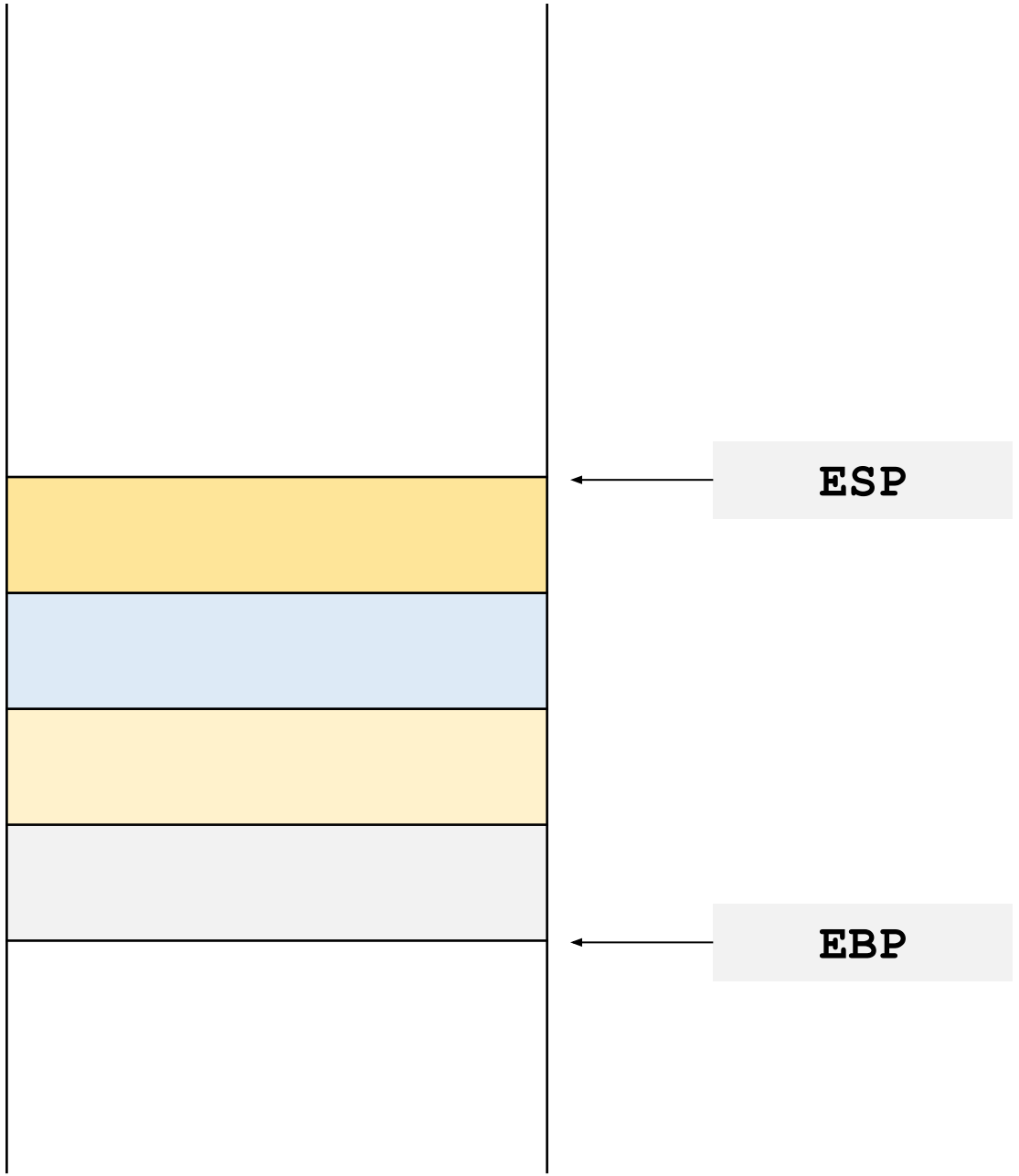
STACK

PUSH

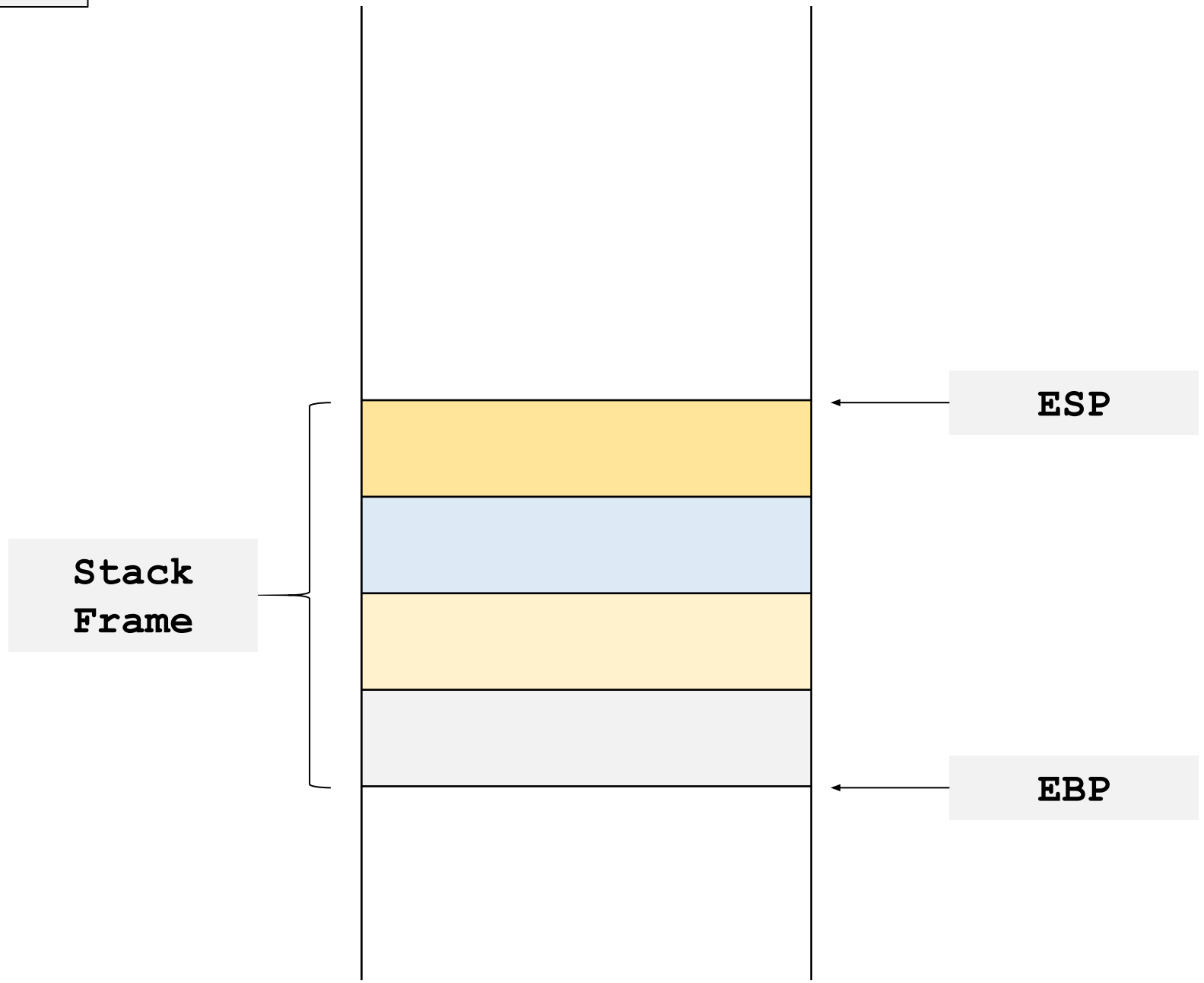


STACK

PUSH

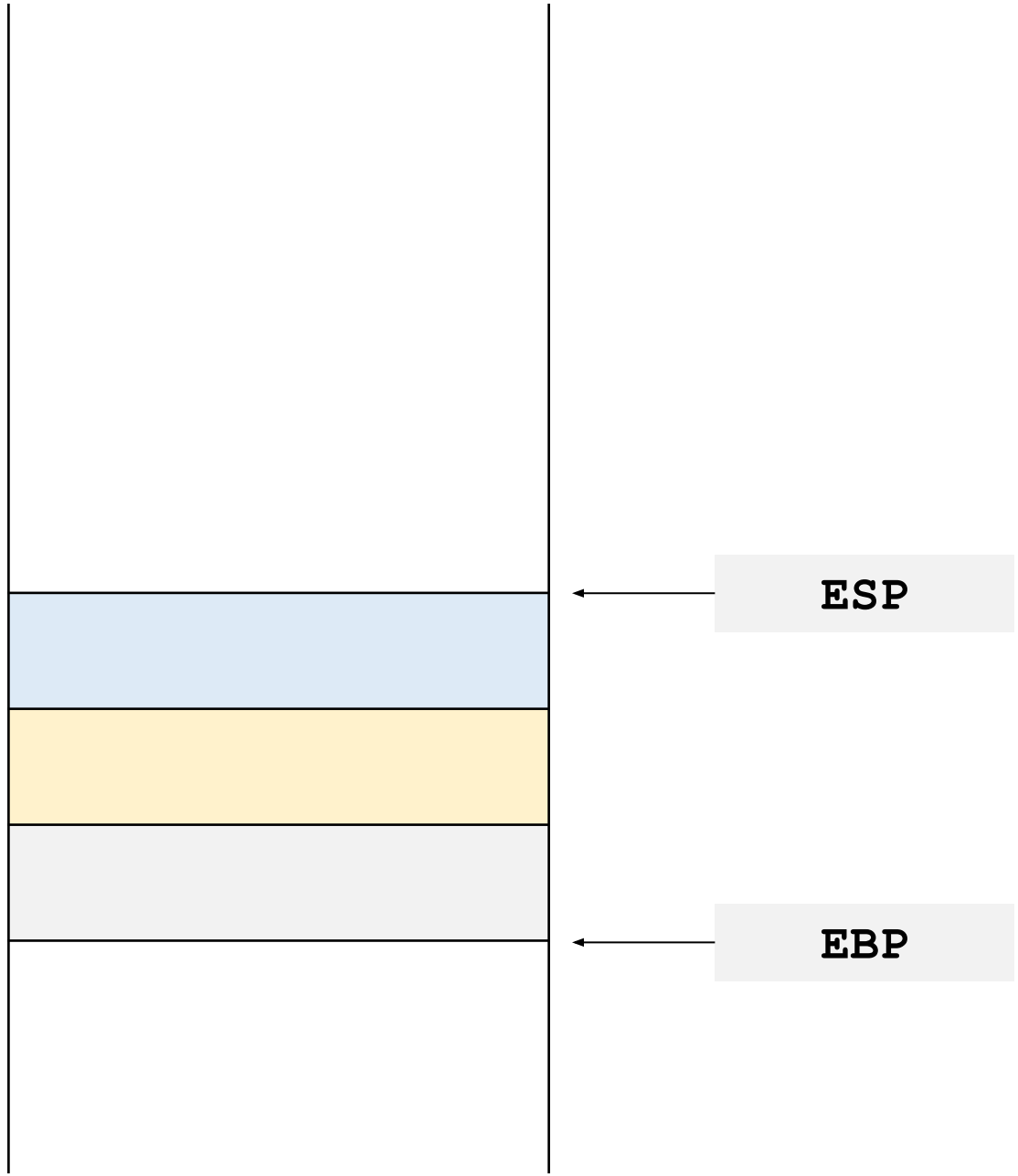


STACK



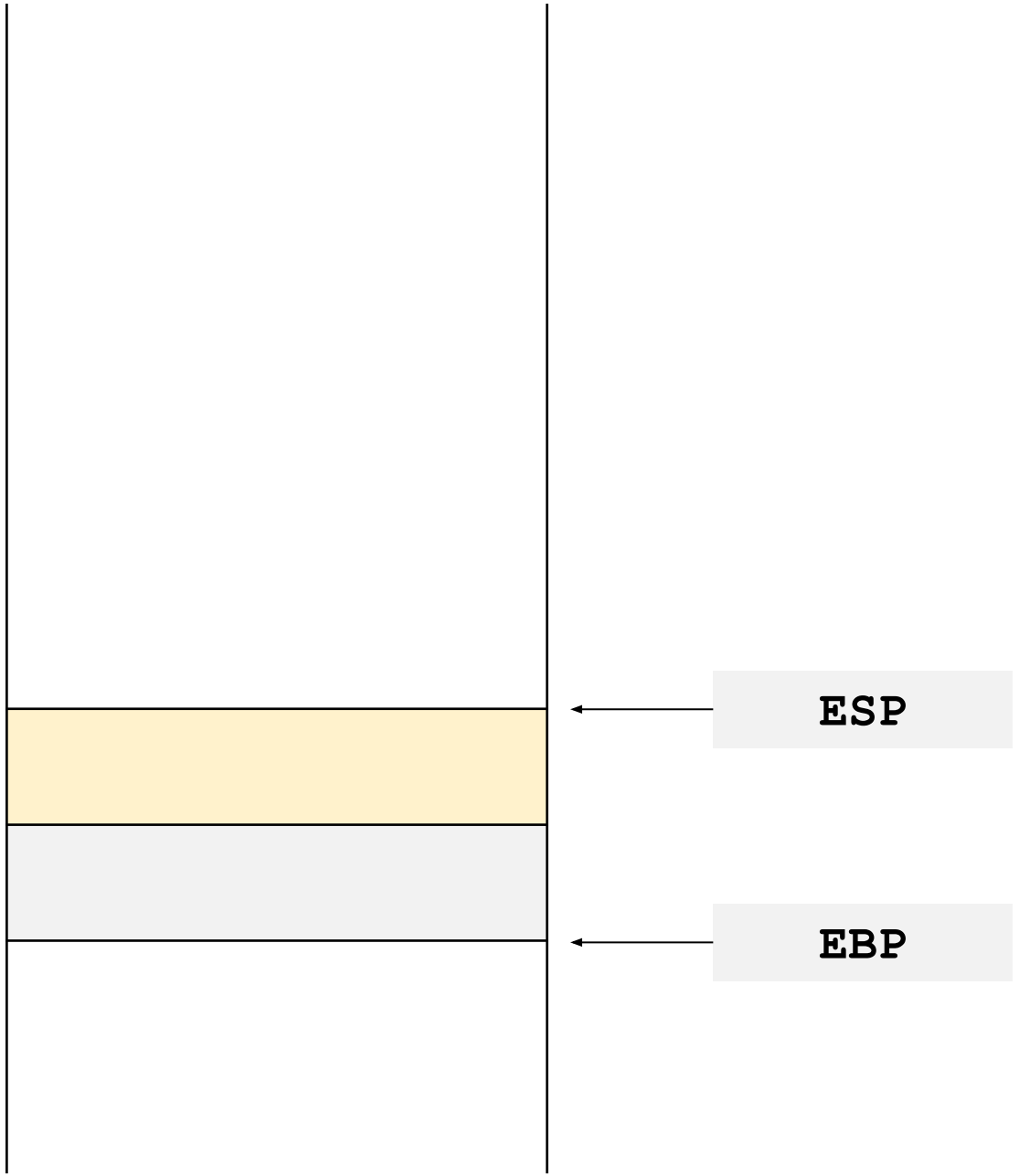
STACK

POP



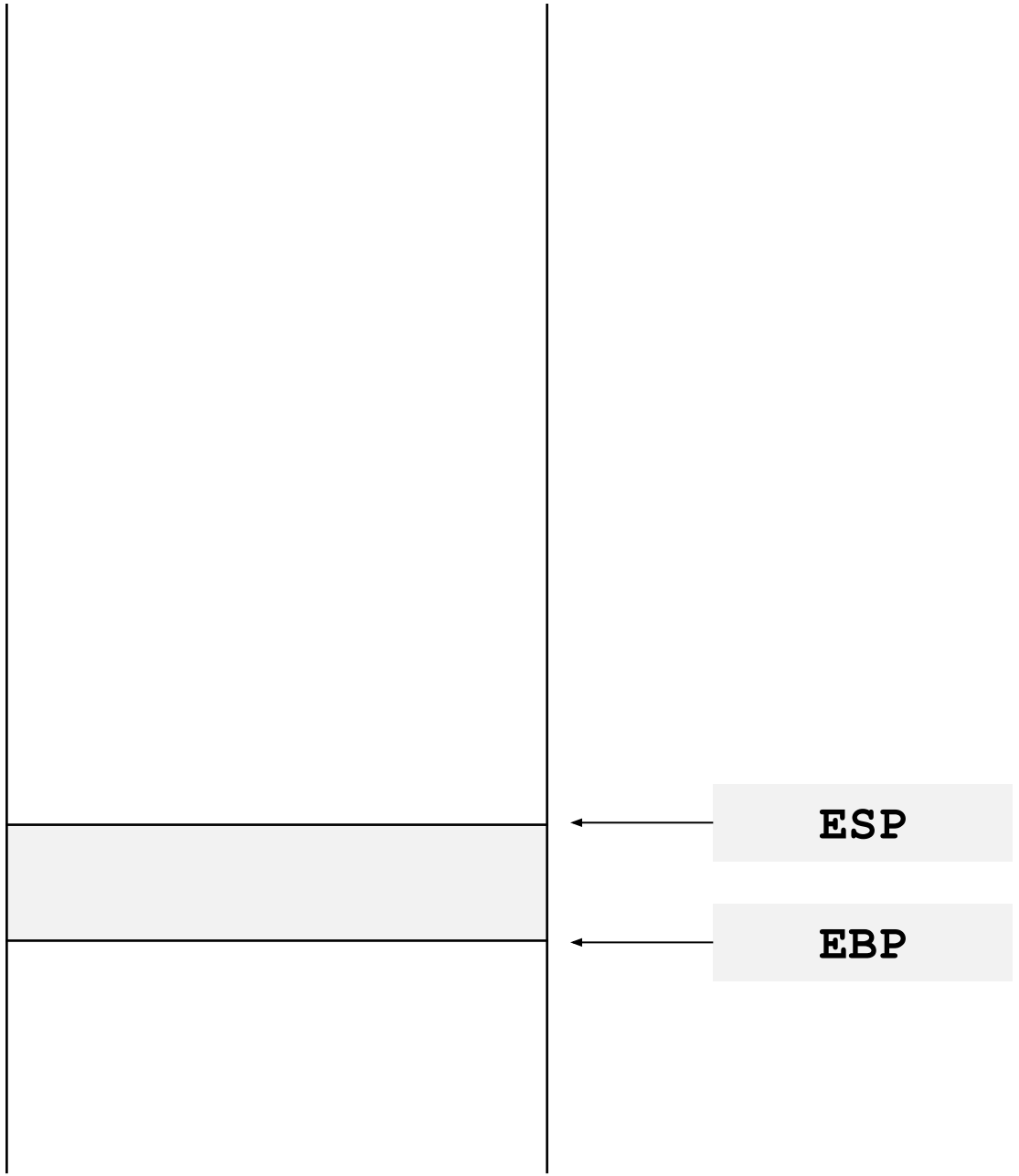
STACK

POP



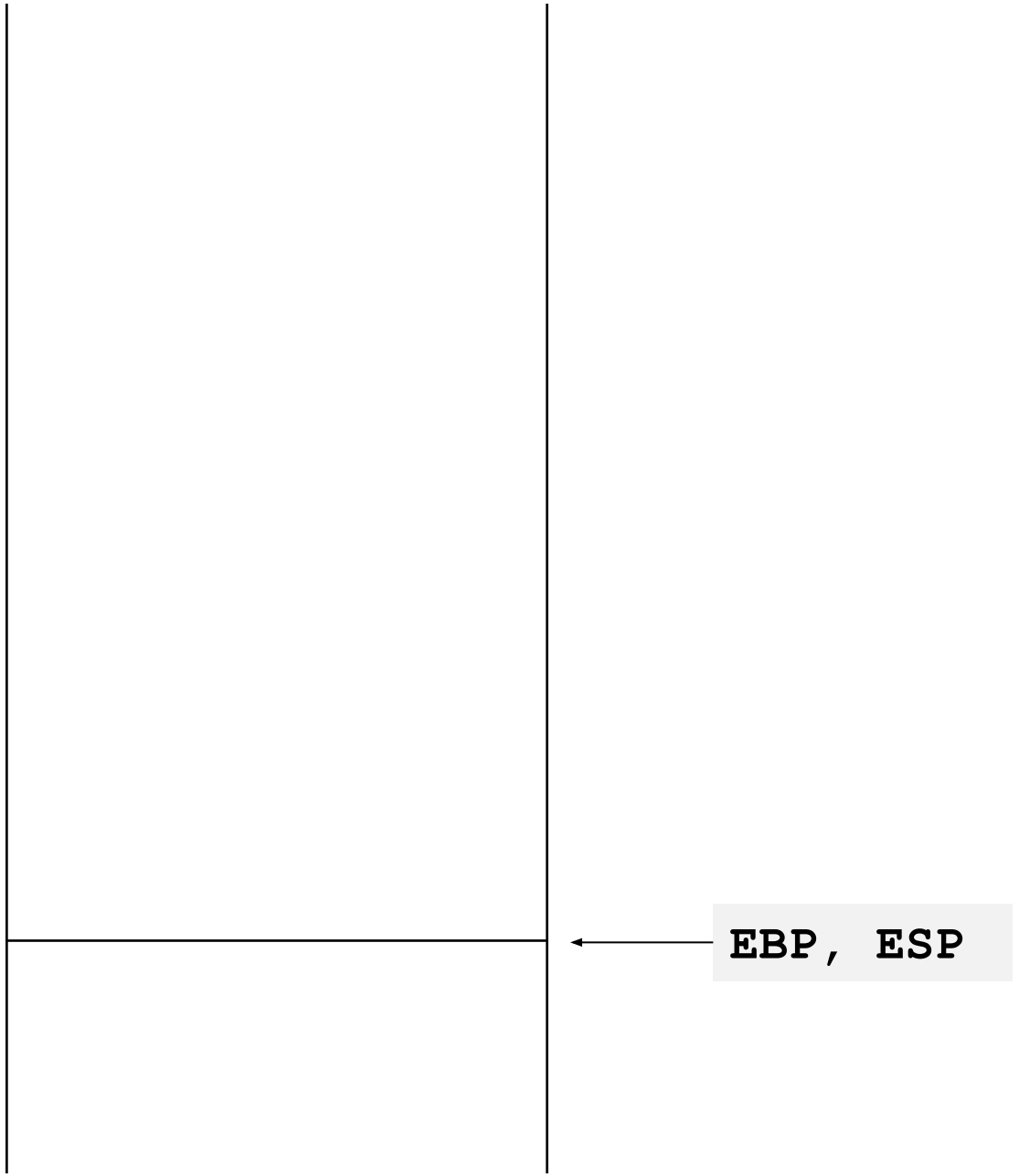
STACK

POP



STACK

POP



```
# gcc -m32 -g -o segmentos segmentos.c
# gdb ejemplo
(gdb) set disassembly-flavor intel
(gdb) disassemble main
```

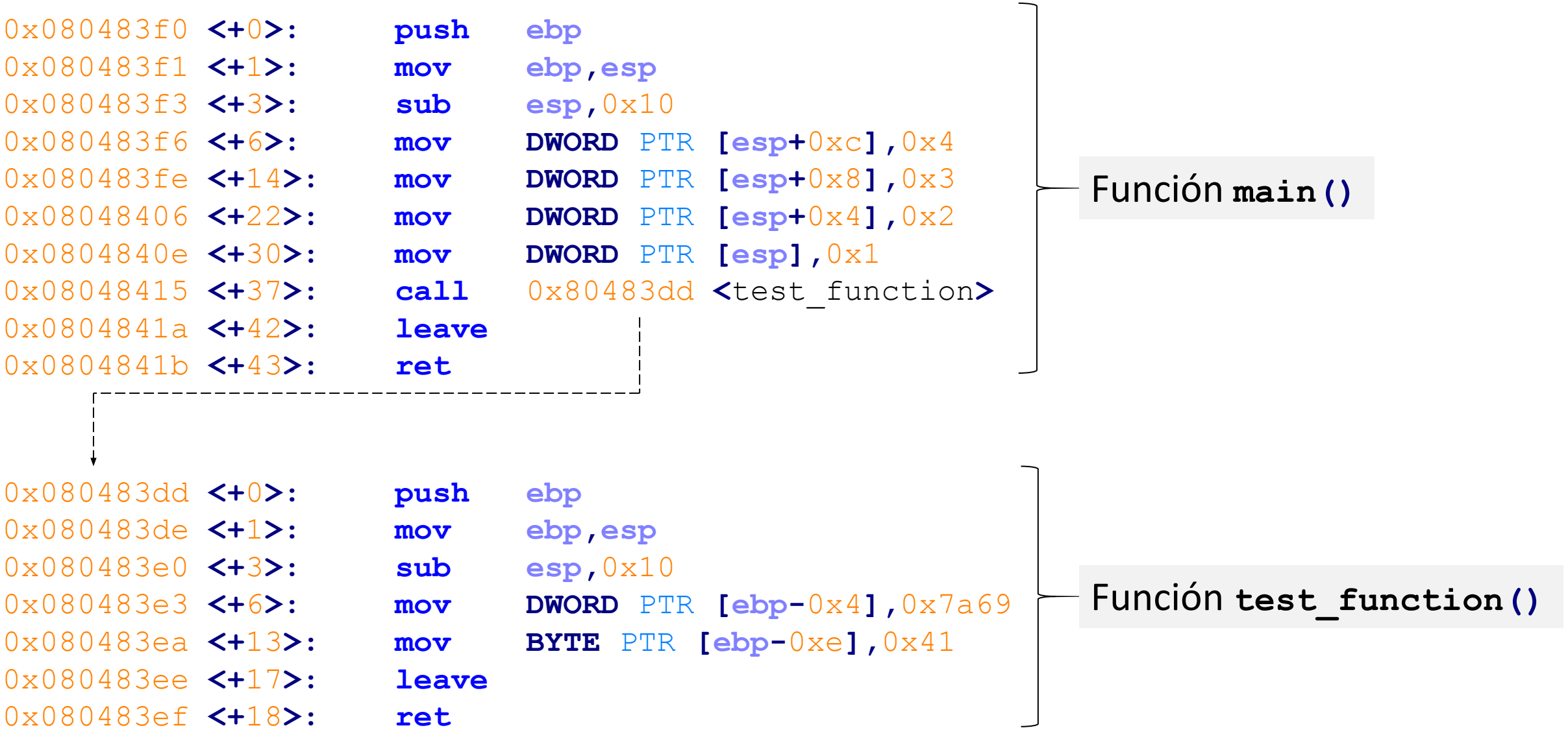
```c
#include <stdlib.h>

void test_function(int a, int b, int c, int d) {
    int flag;
    char buffer[10];
    flag = 31337;
    buffer[0] = 'A';
}

int main() {
    test_function(1, 2, 3, 4);
}
```

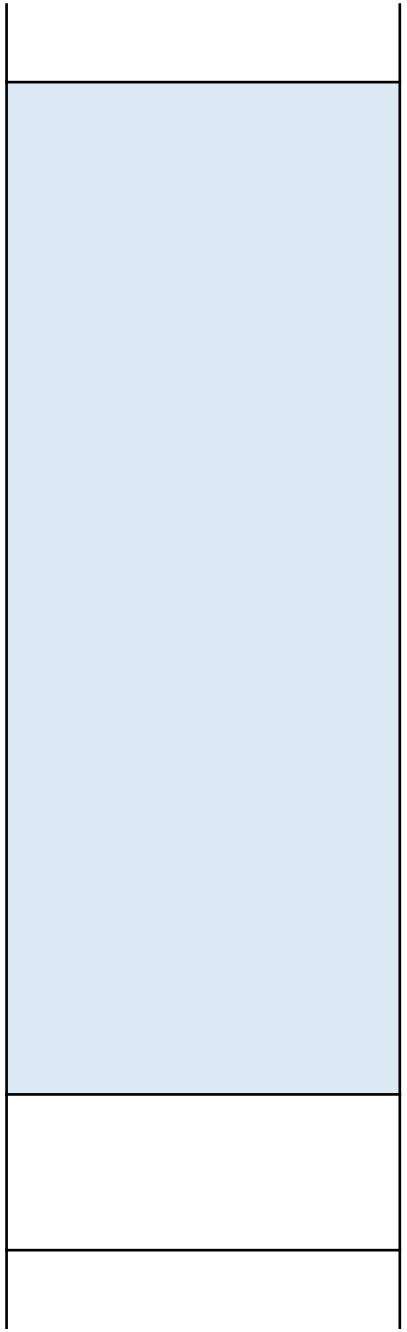
Instalar soporte para compilar a 32 bits

```
# yum install libgcc.i686 glibc-devel.i686
```


0x080483f0 <+0>: push ebp
0x080483f1 <+1>: mov ebp,esp
0x080483f3 <+3>: sub esp,0x10
0x080483f6 <+6>: mov DWORD PTR [esp+0xc],0x4
0x080483fe <+14>: mov DWORD PTR [esp+0x8],0x3
0x08048406 <+22>: mov DWORD PTR [esp+0x4],0x2
0x0804840e <+30>: mov DWORD PTR [esp],0x1
0x08048415 <+37>: call 0x80483dd <test_function>
0x0804841a <+42>: leave
0x0804841b <+43>: ret
Función main()
0x080483dd <+0>: push ebp
0x080483de <+1>: mov ebp,esp
0x080483e0 <+3>: sub esp,0x10
0x080483e3 <+6>: mov DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>: mov BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>: leave
0x080483ef <+18>: ret
Función test_function()

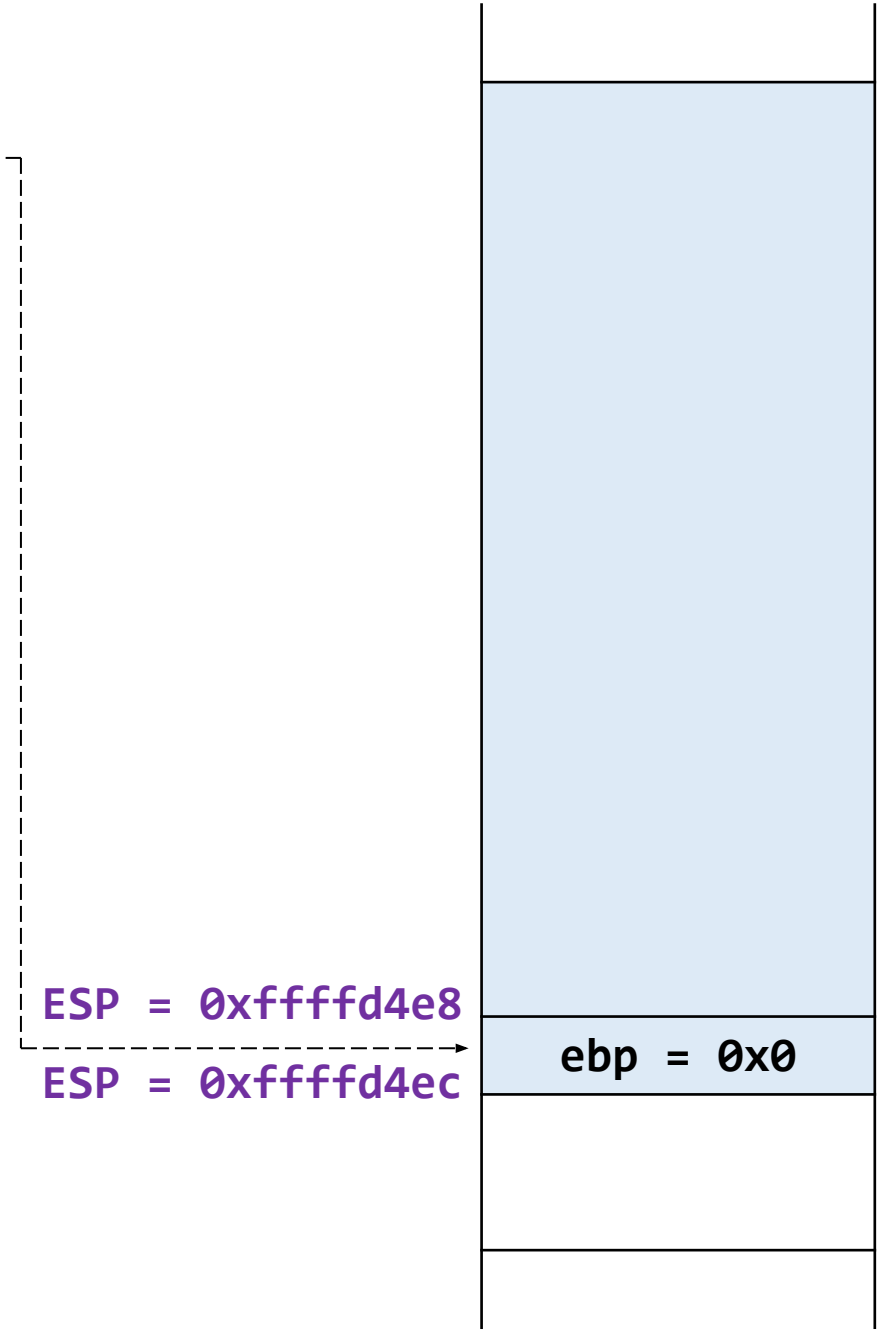
```
(gdb) break main
(gdb) run
(gdb) info proc mappings
(gdb) print $esp
```

```
Start Addr   End Addr       Size     Offset objfile
 0x8048000  0x8049000     0x1000        0x0 /root/segmentos
 0x8049000  0x804a000     0x1000        0x0 /root/segmentos
 0x804a000  0x804b000     0x1000     0x1000 /root/segmentos
0xf7e03000 0xf7e04000     0x1000        0x0
0xf7e04000 0xf7fc8000   0x1c4000        0x0 /usr/lib/libc-2.17.so
0xf7fc8000 0xf7fc9000     0x1000   0x1c4000 /usr/lib/libc-2.17.so
0xf7fc9000 0xf7fcb000     0x2000   0x1c4000 /usr/lib/libc-2.17.so
0xf7fcb000 0xf7fcc000     0x1000   0x1c6000 /usr/lib/libc-2.17.so
0xf7fcc000 0xf7fcf000     0x3000        0x0
0xf7fd8000 0xf7fd9000     0x1000        0x0
0xf7fd9000 0xf7fda000     0x1000        0x0 [vdso]
0xf7fda000 0xf7ffc000    0x22000        0x0 /usr/lib/ld-2.17.so
0xf7ffc000 0xf7ffd000     0x1000    0x21000 /usr/lib/ld-2.17.so
0xf7ffd000 0xf7ffe000     0x1000    0x22000 /usr/lib/ld-2.17.so
0xfffdd000 0xffffe000    0x21000        0x0 [stack]
```



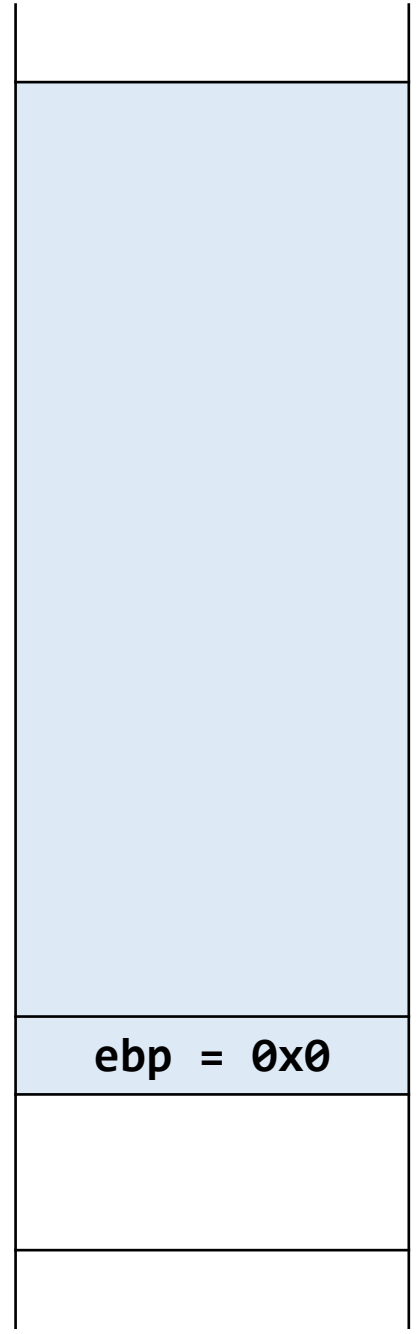
```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
```

- Inicialmente **EBP = 0x0**
- Inicialmente **ESP = 0xffffd4ec**
- Cuando se hace **push ebp** se empujan **4** bytes al *stack.*
- El **ESP** debe actualizarse restando **4** a la dirección **0xffffd4ec**
  - **ESP – 4 = 0xffffd4e8**
- Ahora **ESP** apunta a la siguiente dirección disponible en el *stack*



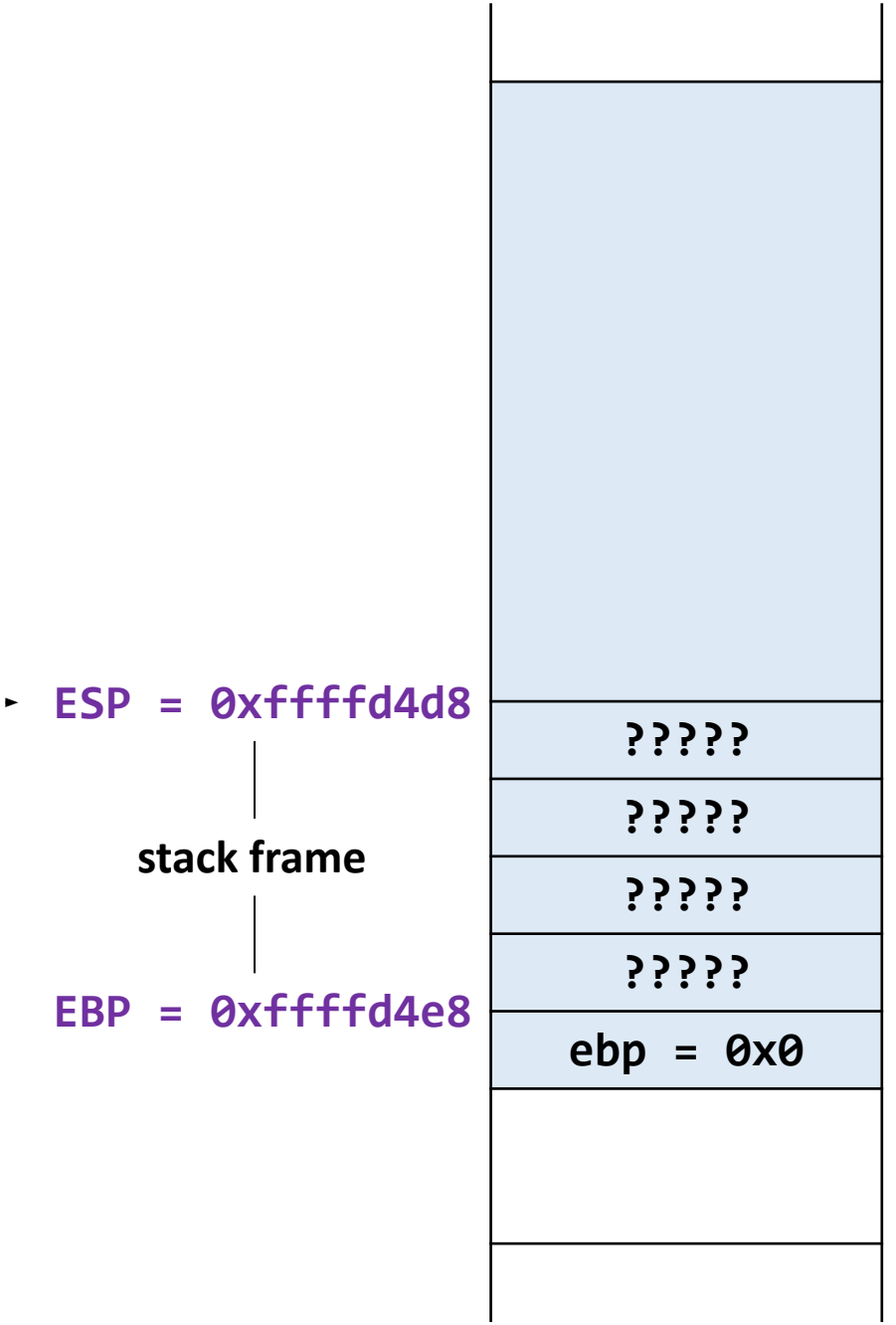
```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
```

EBP = ESP = 0xffffd4e8



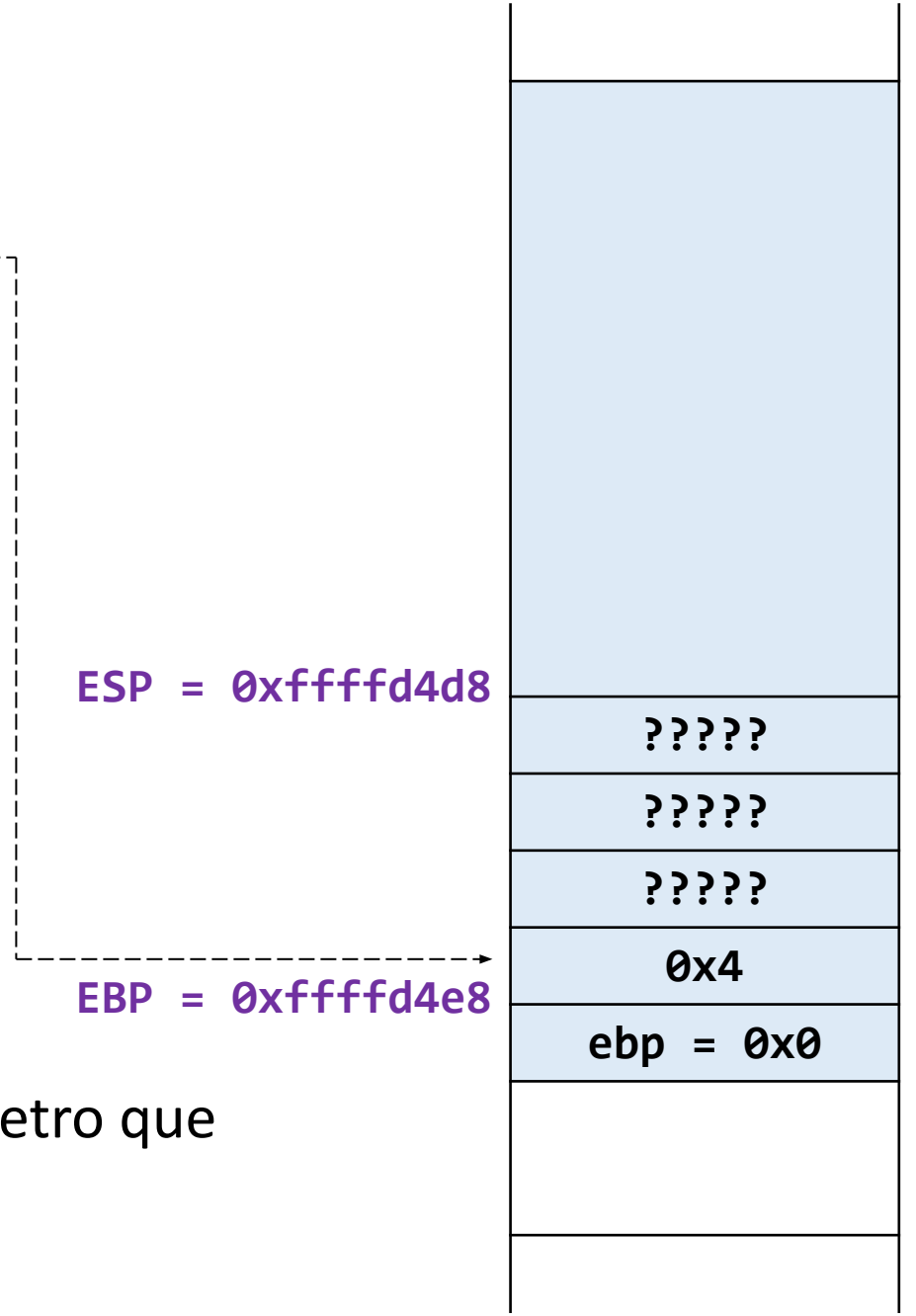
- Se iguala **EBP** a **ESP** para preparar un *stack frame* para la función **`main()`**.
- Estamos analizando las primeras líneas de código de la función **`main()`**.

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
```



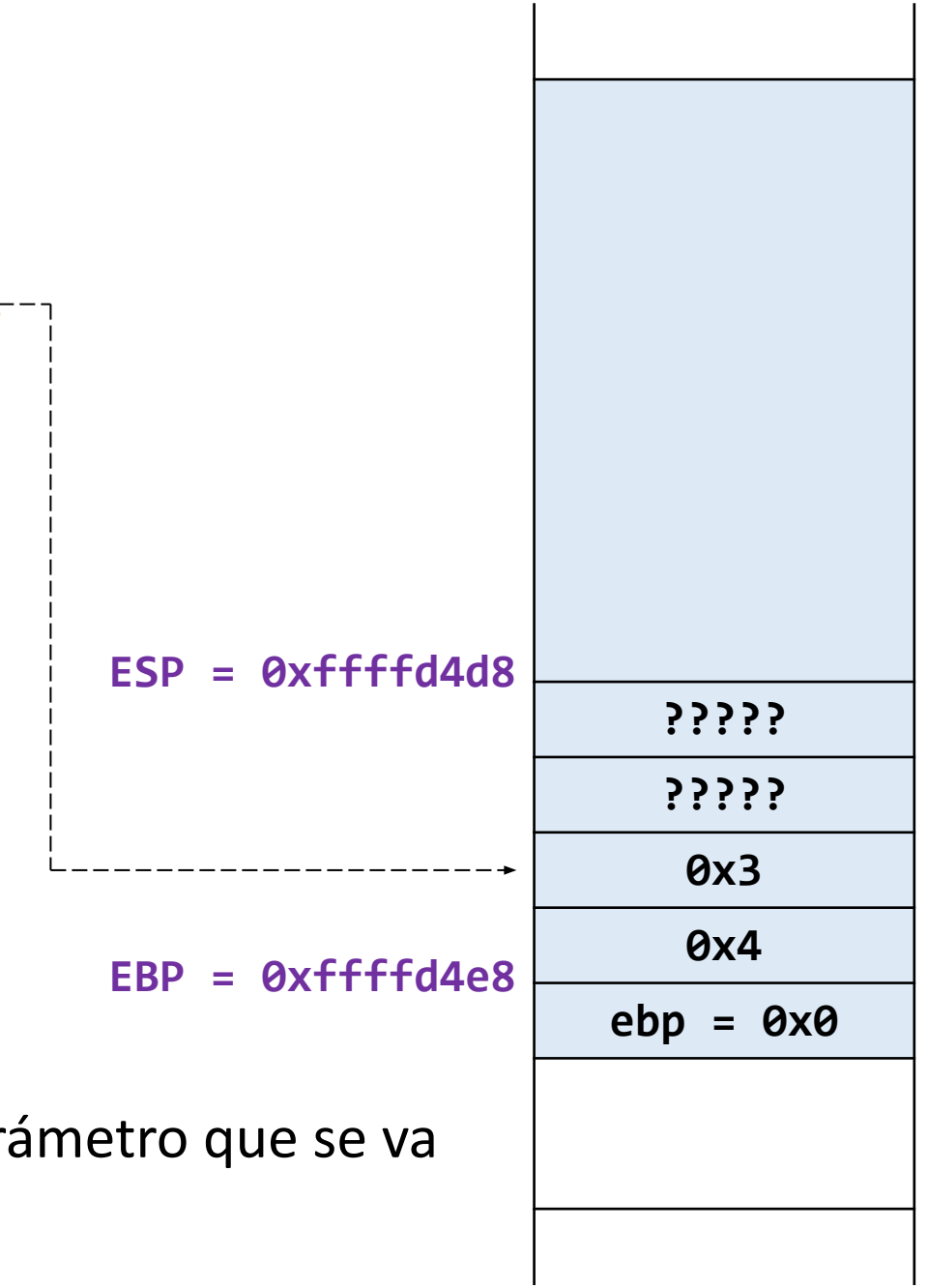
- Se restan 16 bytes (**0x10**) al registro **ESP**. El resultado de la resta se almacena en el registro **ESP**.
  - **ESP - 0x10 = 0xffffd4d8**
- En este punto es donde se crea el *stack frame* para la función **main()**.

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
0x080483f6 <+6>:     mov    DWORD PTR [esp+0xc],0x4
0x080483fe <+14>:    mov    DWORD PTR [esp+0x8],0x3
0x08048406 <+22>:    mov    DWORD PTR [esp+0x4],0x2
0x0804840e <+30>:    mov    DWORD PTR [esp],0x1
```



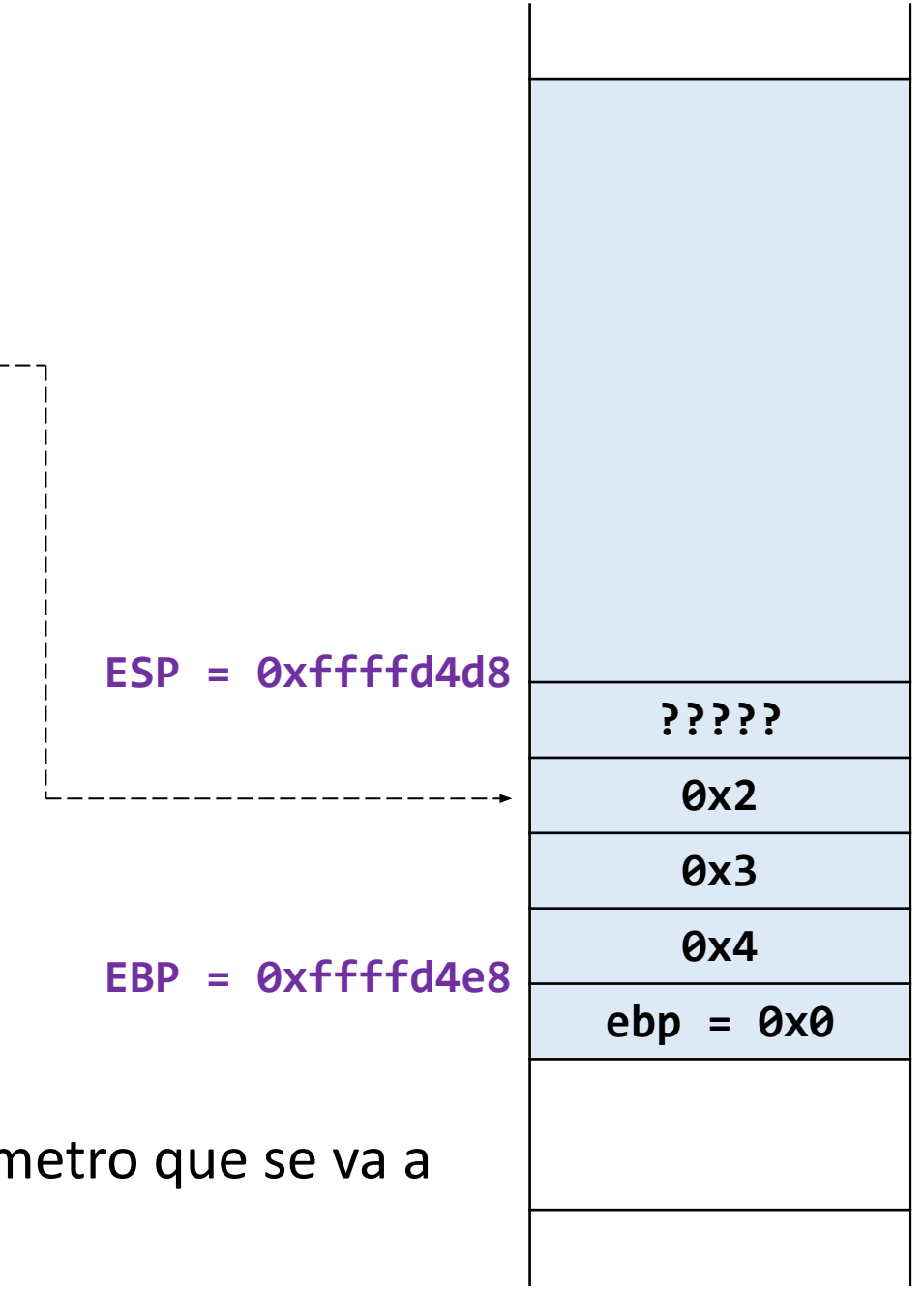
- Se almacena en **ESP + 0xc (ESP + 12)** el último parámetro que se va a pasar a la función **test_function()**.
  - **ESP + 0xc = 0xffffd4e4**

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
0x080483f6 <+6>:     mov    DWORD PTR [esp+0xc],0x4
0x080483fe <+14>:    mov    DWORD PTR [esp+0x8],0x3
0x08048406 <+22>:    mov    DWORD PTR [esp+0x4],0x2
0x0804840e <+30>:    mov    DWORD PTR [esp],0x1
```



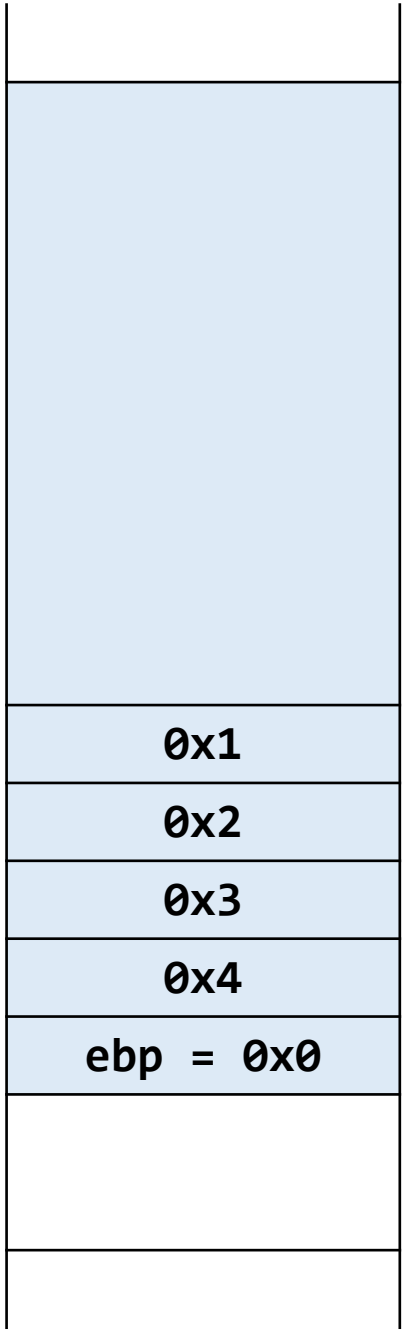
- De manera similar se empuja a la pila el penúltimo parámetro que se va a pasar a la función **`test_function()`**.

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
0x080483f6 <+6>:     mov    DWORD PTR [esp+0xc],0x4
0x080483fe <+14>:    mov    DWORD PTR [esp+0x8],0x3
0x08048406 <+22>:    mov    DWORD PTR [esp+0x4],0x2
0x0804840e <+30>:    mov    DWORD PTR [esp],0x1
```



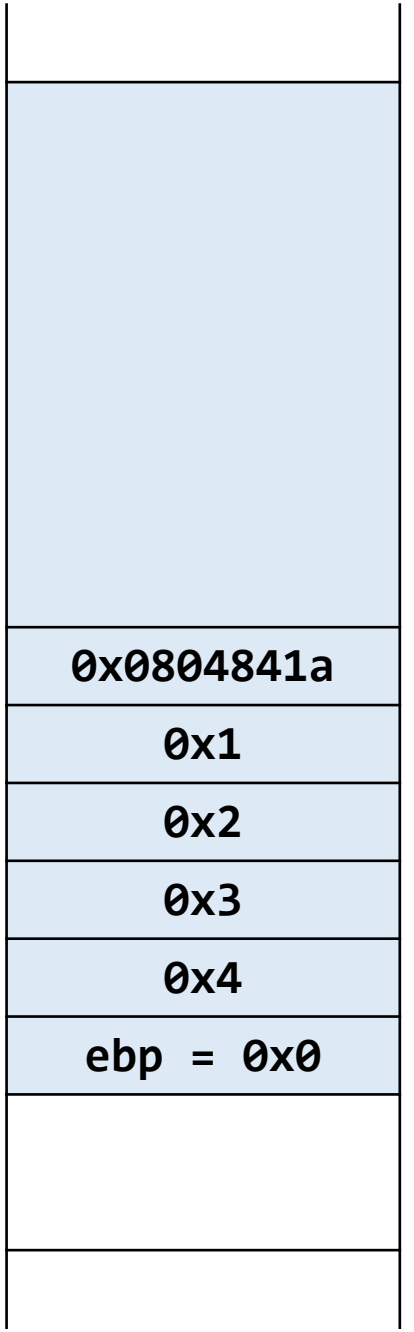
- De manera similar se empuja a la pila el segundo parámetro que se va a pasar a la función **`test_function()`**.

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
0x080483f6 <+6>:     mov    DWORD PTR [esp+0xc],0x4
0x080483fe <+14>:    mov    DWORD PTR [esp+0x8],0x3
0x08048406 <+22>:    mov    DWORD PTR [esp+0x4],0x2
0x0804840e <+30>:    mov    DWORD PTR [esp],0x1
```



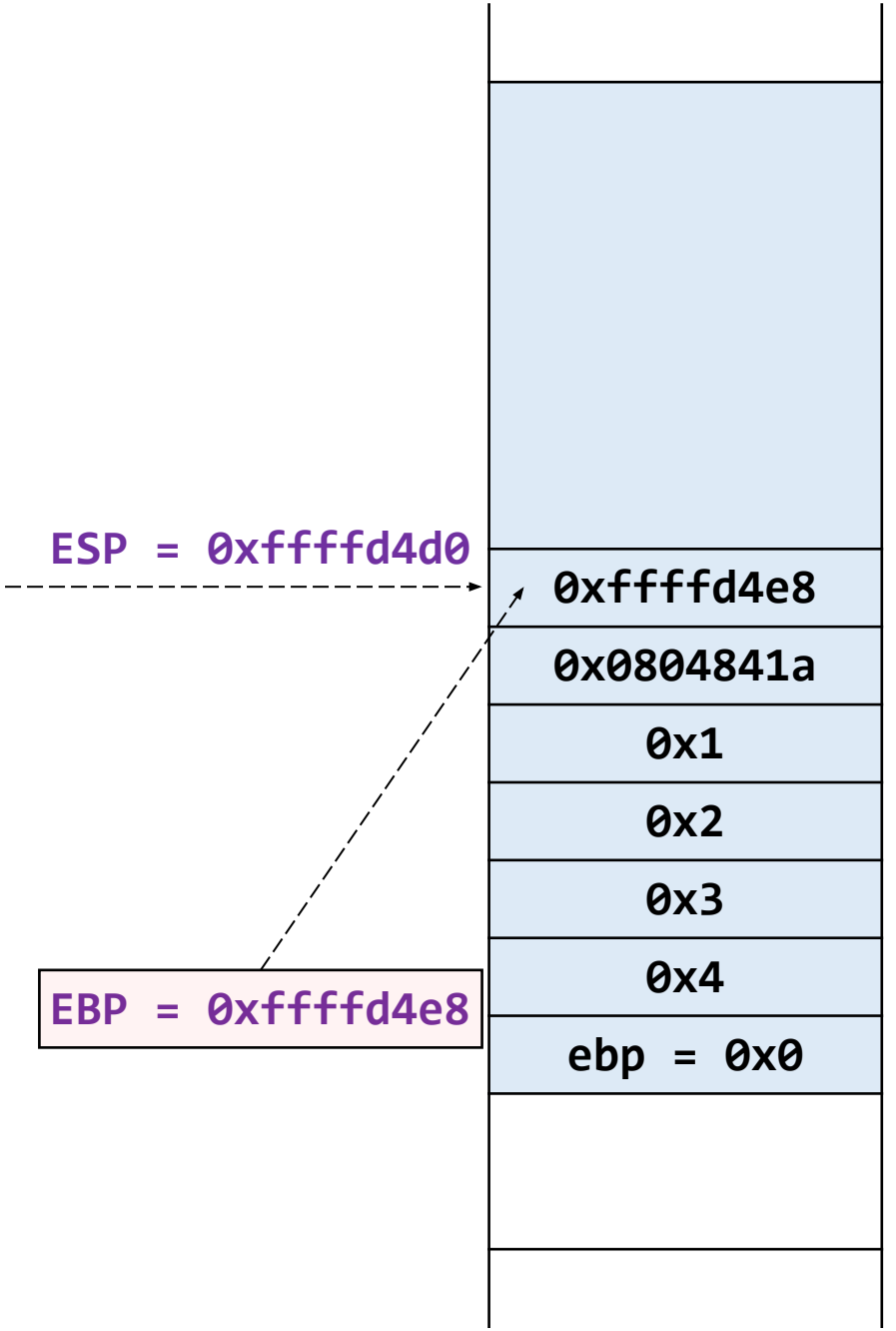
- De manera similar se empuja a la pila el primer parámetro que se va a pasar a la función **`test_function()`**.

```
0x080483f0 <+0>:     push   ebp
0x080483f1 <+1>:     mov    ebp,esp
0x080483f3 <+3>:     sub    esp,0x10
0x080483f6 <+6>:     mov    DWORD PTR [esp+0xc],0x4
0x080483fe <+14>:    mov    DWORD PTR [esp+0x8],0x3
0x08048406 <+22>:    mov    DWORD PTR [esp+0x4],0x2
0x0804840e <+30>:    mov    DWORD PTR [esp],0x1
0x08048415 <+37>:    call   0x80483dd <test function>
0x0804841a <+42>:    leave
0x0804841b <+43>:    ret
```



- Empuja a la pila la dirección posterior al **CALL**
  - **0x0804841a**
- Se actualiza el **ESP** restando al valor actual 4 bytes
  - **ESP = ESP - 4 = 0xffffd4d4**
- **EIP** apunta ahora a **0x80483dd**
  - La primera instrucción de la función **test_function()**.

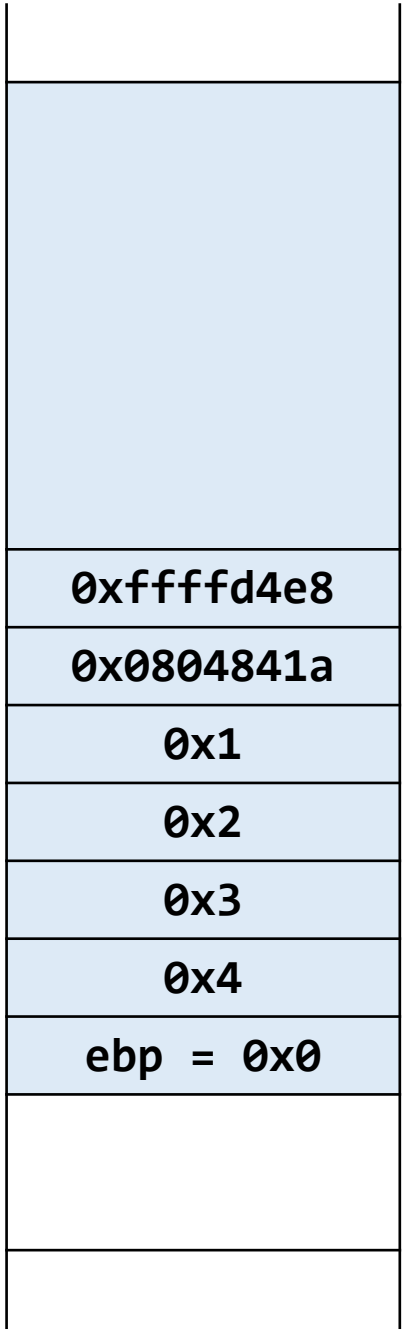
```
0x080483dd <+0>:    push   ebp
0x080483de <+1>:    mov    ebp,esp
0x080483e0 <+3>:    sub    esp,0x10
0x080483e3 <+6>:    mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:   mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:   leave
0x080483ef <+18>:   ret
```



- Ahora estamos en el código de la función **`test_function()`**
- Se empuja a la pila la dirección actual del **EBP**
  - Se está preparando otro *stack frame*: el de la función **`test_function()`**
- Se actualiza el **ESP** restando al valor actual 4 bytes
  - **`ESP = ESP - 4 = 0xffffd4d0`**

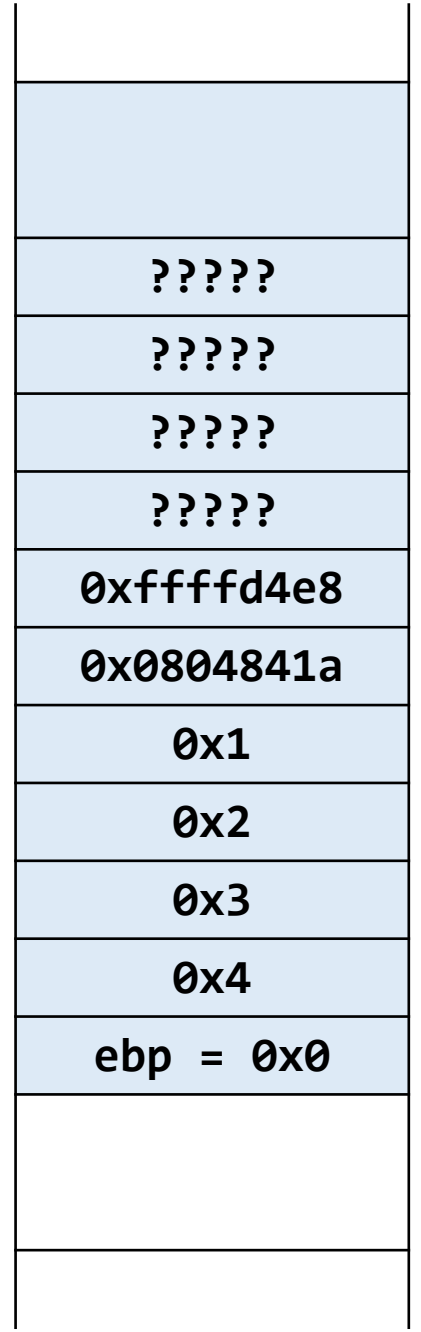
```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

EBP = ESP = 0xffffd4d0



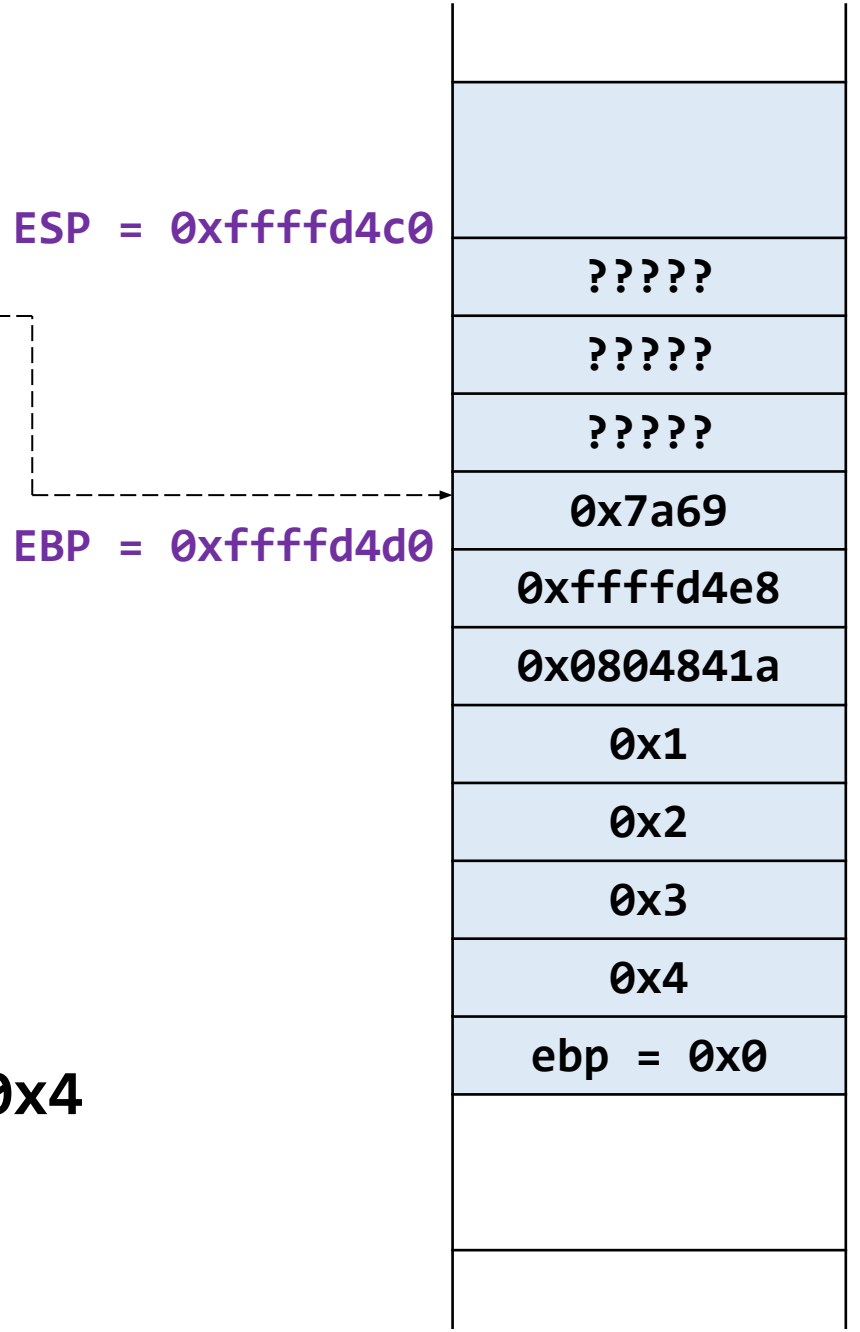
- Se asigna al **EBP** el valor del **ESP**
  - Prólogo de la función preparando un *stack frame*.

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10 ----------------------> ESP = 0xffffd4c0
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
                                                  EBP = 0xffffd4d0
```



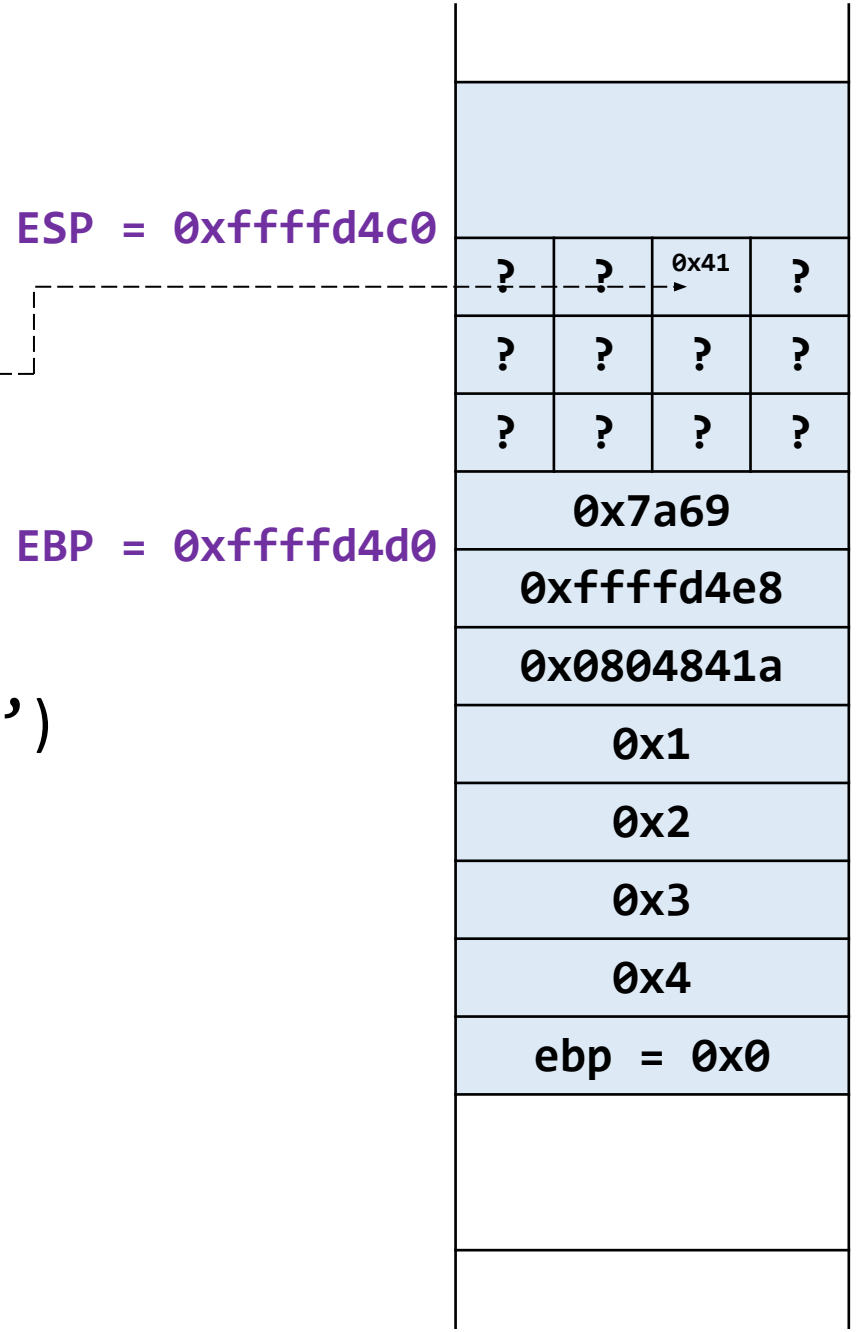
- Se crea un *stack frame* de 16 bytes (**0x10**)
  - Se restan 16 bytes al valor del **ESP**, el resultado se almacena en **ESP**
  - **ESP - 0x10 = 0xffffd4c0**

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```



- Almacena en **EBP-0x4** el valor de la variable **flag**.
  - 0x7a69 = 31337 (decimal)
  - La dirección de memoria de la variable **flag** es **EBP-0x4**

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

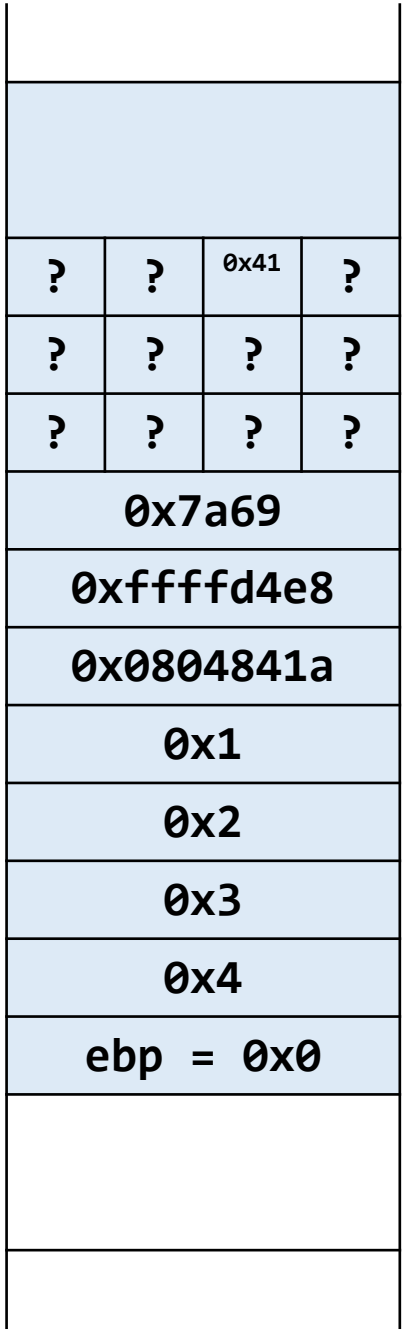


- Almacena en **EBP-0xe** (14) el valor 0x41 (ASCII 65 = ‘A’)
  - **EBP-0xe = 0xffffd4c2**
- El arreglo **buffer** empieza en esa dirección de memoria
  - **buffer[0] = 'A'**
- Sobran dos bytes al principio
  - Stack frame de 16 bytes
  - Arreglo de 10 bytes
  - Número entero **flag** de 4 bytes

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

ESP = 0xffffd4c0

EBP = 0xffffd4d0



Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	ZO	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	ZO	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	ZO	Valid	N.E.	Set RSP to RBP, then pop RBP.

Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4

Prólogo y epílogo de las funciones

- Prólogo

```
push ebp
mov ebp,esp
sub esp, <N bytes>
```

- Epílogo

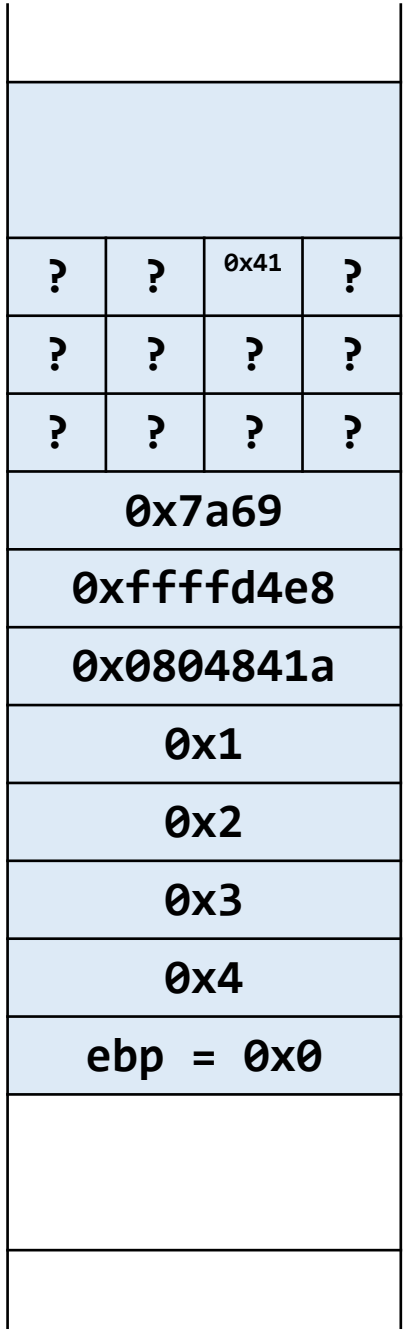
```
mov esp,ebp
pop ebp
ret
```

`leave`

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

```
mov esp,ebp
pop ebp
ret
```

ESP = EBP = 0xffffd4d0



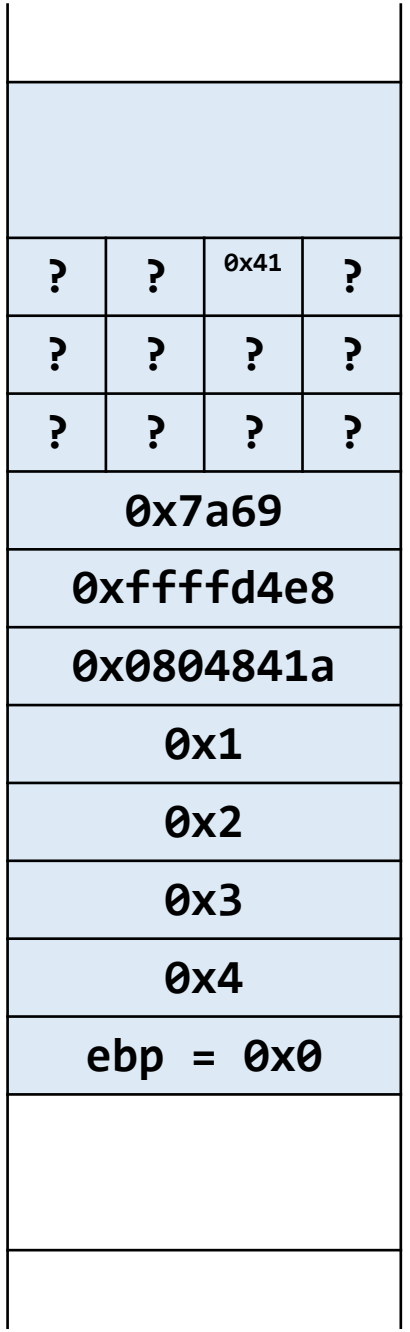
- Se destruye el *stack frame* que se había creado para la función **`test_function()`**.
- No se eliminan los datos de memoria.

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

```
mov esp,ebp
pop ebp
ret
```

ESP = 0xffffd4d4

EBP = 0xffffd4e8



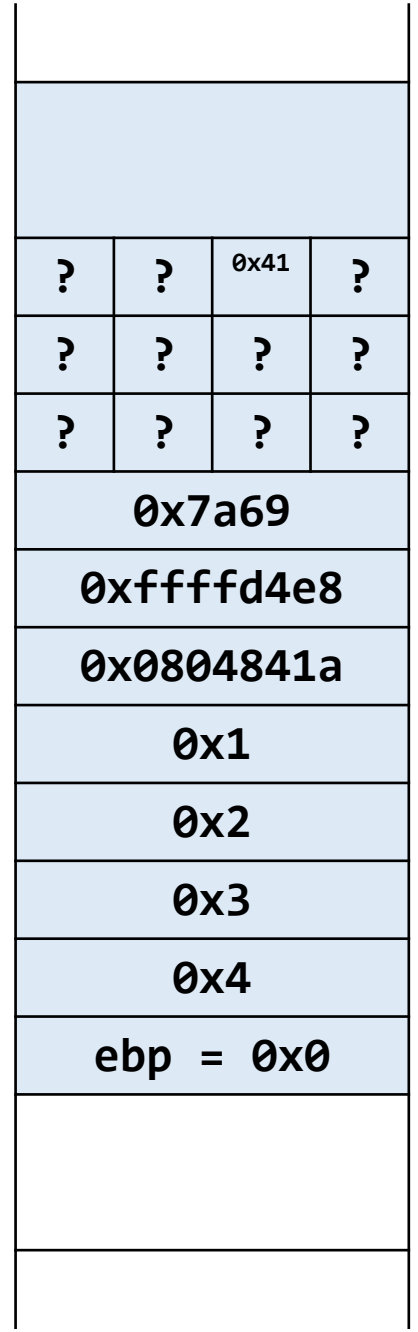
- La instrucción **pop ebp**
  - Saca de la pila lo apuntado por **ESP**
  - Lo que sale de la pila se asigna al registro **EBP**
  - Actualiza **ESP**: **ESP + 4 = 0xffffd4d0 + 4 = 0xffffd4d4**
  - Esta operación restaura el valor del **EBP** anterior al llamado de la función **test_function()**.

```
0x080483dd <+0>:     push   ebp
0x080483de <+1>:     mov    ebp,esp
0x080483e0 <+3>:     sub    esp,0x10
0x080483e3 <+6>:     mov    DWORD PTR [ebp-0x4],0x7a69
0x080483ea <+13>:    mov    BYTE PTR [ebp-0xe],0x41
0x080483ee <+17>:    leave
0x080483ef <+18>:    ret
```

```
mov esp,ebp
pop ebp
ret
```

ESP = 0xffffd4d8

EBP = 0xffffd4e8



- La instrucción **ret**
  - Saca de la pila la dirección apuntada por el **ESP**
  - Lo que sale de la pila se asigna al registro **EIP**
    - **EIP = 0x0804841a**
  - Actualiza ESP: **ESP + 4**
    - **ESP + 4 = 0xffffd4d8**

- En este punto la ejecución sigue en la dirección apuntada por EIP
- Instrucción posterior al CALL

```
...
0x08048415 <+37>:   call 0x80483dd <test_function>
0x0804841a <+42>:   leave
0x0804841b <+43>:   ret
```

- Se recuperó el *stack frame* de la función **`main()`** ya que la ejecución retorna a la función **`main()`** a la instrucción posterior al llamado a la instrucción **`test_function()`**.