

# Administración de Memoria

## Sistemas Operativos

Carlos A. Di Pietro

Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

17 de mayo de 2016

- ▶ Gestor de Memoria (Memory Manager):  
Componente del sistema operativo encargado de administrar la memoria.

- ▶ Gestor de Memoria (Memory Manager):

Componente del sistema operativo encargado de administrar la memoria.

¿Más específico?

- ▶ Gestor de Memoria (Memory Manager):

Componente del sistema operativo encargado de administrar la memoria.

¿Más específico?

Se encarga de (entre otras cosas):

1. Asegurar la disponibilidad de memoria
2. Asignar y liberar memoria.
3. Organizar la memoria disponible.
4. Asegurar la protección de la memoria.
5. Permitir acceso a memoria compartida.

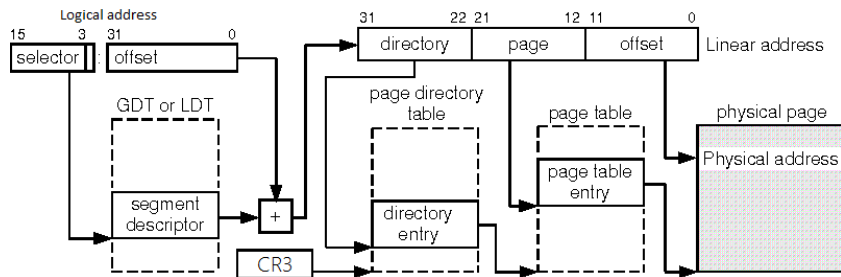
## ▶ MMU

- ▶ Unidad de Gestión de Memoria (Memory Management Unit)
- ▶ Permite traducir direcciones virtuales a físicas.

# Repaso de Orga II

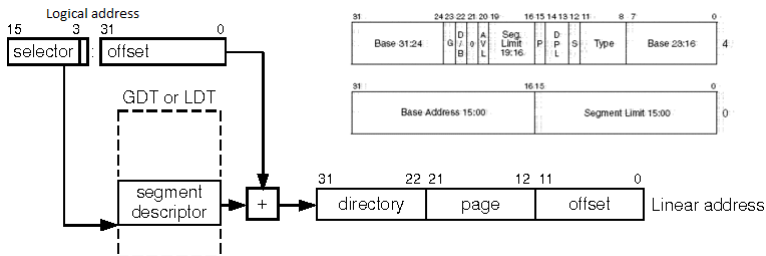
- ▶ MMU
  - ▶ Unidad de Gestión de Memoria (Memory Management Unit)
  - ▶ Permite traducir direcciones virtuales a físicas.
- ▶ ¿Memoria virtual?

- ▶ MMU
  - ▶ Unidad de Gestión de Memoria (Memory Management Unit)
  - ▶ Permite traducir direcciones virtuales a físicas.
- ▶ ¿Memoria virtual?



# Segmentación

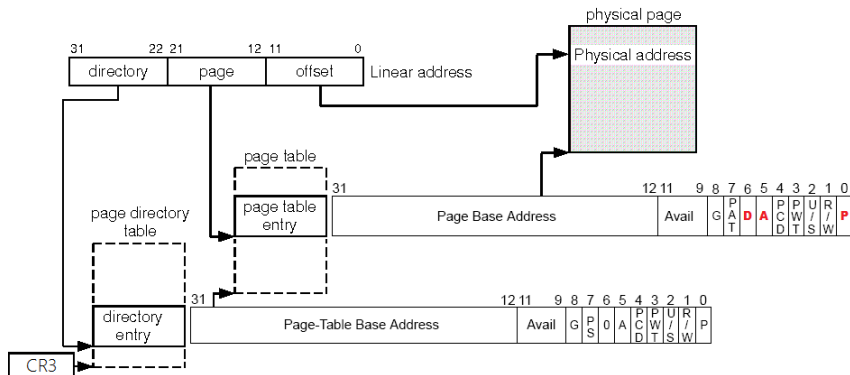
- ▶ Primeros bits de la dirección indexan el **Descriptor de Segmento** en la tabla de descriptores.
- ▶ El resto es el offset dentro del segmento.
- ▶ La **GDT** (y la **LDT** de cada proceso) contiene los descriptores de los segmentos.
- ▶ Los descriptores almacenan información sobre los segmentos (inicio en memoria física, **tamaño**, atributos).





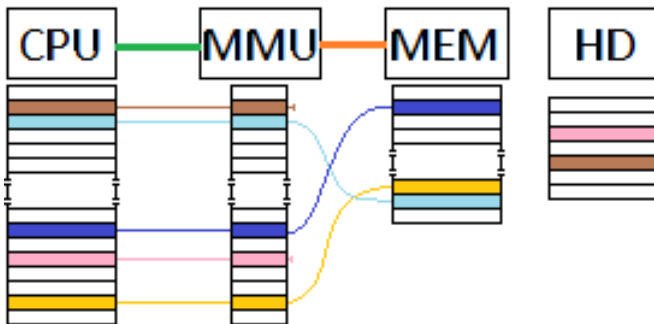
# Paginación

- ▶ Los conjuntos de bits indexan en las sucesivas tablas de páginas. Los últimos son el offset en la página.
- ▶ Cada entrada contiene la dirección base de la página más algunos atributos.



# Direcciones lógicas vs direcciones físicas

- ▶ Memoria física: Una celda dentro de la memoria del sistema. El tamaño está determinado por el hardware.
  - ▶ Un **marco de página** es una porción de memoria física.
- ▶ Memoria virtual: Una representación de la información almacenada. El tamaño depende de la unidad de direccionamiento y la cantidad de bits de direccionamiento.
  - ▶ Una **página** es una porción de memoria virtual.



# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

- ▶ Cantidad de bytes de memoria física = MEM\_SIZE.
- ▶ Cantidad de bytes de memoria virtual =  $2^{DIR\_BITS} * DIR\_UNIT$ .
- ▶ Cantidad de celdas de memoria física = MEM\_SIZE / DIR\_UNIT.
- ▶ Cantidad de celdas de memoria virtual =  $2^{DIR\_BITS}$ .

# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

- ▶ Cantidad de bytes de memoria física = MEM\_SIZE.
- ▶ Cantidad de bytes de memoria virtual =  $2^{DIR\_BITS} * DIR\_UNIT$ .
- ▶ Cantidad de celdas de memoria física = MEM\_SIZE / DIR\_UNIT.
- ▶ Cantidad de celdas de memoria virtual =  $2^{DIR\_BITS}$ .

Ejercicios:

# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

- ▶ Cantidad de bytes de memoria física = MEM\_SIZE.
- ▶ Cantidad de bytes de memoria virtual =  $2^{DIR\_BITS} * DIR\_UNIT$ .
- ▶ Cantidad de celdas de memoria física = MEM\_SIZE / DIR\_UNIT.
- ▶ Cantidad de celdas de memoria virtual =  $2^{DIR\_BITS}$ .

Ejercicios:

- ▶ Tengo 16 bits para direccionar. ¿Cuánta memoria virtual puedo tener direccionando a byte? ¿Cuánta física?

# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

- ▶ Cantidad de bytes de memoria física = MEM\_SIZE.
- ▶ Cantidad de bytes de memoria virtual =  $2^{DIR\_BITS} * DIR\_UNIT$ .
- ▶ Cantidad de celdas de memoria física = MEM\_SIZE / DIR\_UNIT.
- ▶ Cantidad de celdas de memoria virtual =  $2^{DIR\_BITS}$ .

Ejercicios:

- ▶ Tengo 16 bits para direccionar. ¿Cuánta memoria virtual puedo tener direccionando a byte? ¿Cuánta física?
- ▶ Tengo 8 bits para direccionar 256 ( $2^8$ ) bytes de memoria física. ¿Cómo puedo duplicar la memoria virtual?



# 1. Asegurar la disponibilidad de memoria

- ▶ Otra definición de memoria virtual: Hacerle creer al procesador que dispone de más memoria de la que realmente tiene.

Entonces... ¿Cuánta memoria tiene mi sistema?

- ▶ Cantidad de bytes de memoria física = MEM\_SIZE.
- ▶ Cantidad de bytes de memoria virtual =  $2^{DIR\_BITS} * DIR\_UNIT$ .
- ▶ Cantidad de celdas de memoria física = MEM\_SIZE / DIR\_UNIT.
- ▶ Cantidad de celdas de memoria virtual =  $2^{DIR\_BITS}$ .

Ejercicios:

- ▶ Tengo 16 bits para direccionar. ¿Cuánta memoria virtual puedo tener direccionando a byte? ¿Cuánta física?
- ▶ Tengo 8 bits para direccionar 256 ( $2^8$ ) bytes de memoria física. ¿Cómo puedo duplicar la memoria virtual?
- ▶ Tengo 65536 ( $2^{16}$ ) bytes de memoria física dividida en celdas de 16 bits. ¿Cuántos bits necesito para direccionar?

## 2. Asignar y liberar memoria

- ▶ Asignar = reservar una porción de memoria para un proceso.
  - ▶ La porción de memoria pasa a estar ocupada por el proceso que la solicitó.
  - ▶ Tenemos que saber quién es el dueño de esa porción de memoria.
- ▶ Liberar = una porción de la memoria vuelve a estar disponible para cualquier proceso.

## 2. Asignar y liberar memoria

- ▶ Asignar = reservar una porción de memoria para un proceso.
  - ▶ La porción de memoria pasa a estar ocupada por el proceso que la solicitó.
  - ▶ Tenemos que saber quién es el dueño de esa porción de memoria.
- ▶ Liberar = una porción de la memoria vuelve a estar disponible para cualquier proceso.

Cuando hablamos de *memoria*, ¿nos referimos a memoria virtual o física?

### 3. Organizar la memoria disponible

Pensemos en la memoria como una colección de celdas.  
El MM debe elegir qué porción de memoria (qué celdas) asignar.

### 3. Organizar la memoria disponible

Pensemos en la memoria como una colección de celdas.

El MM debe elegir qué porción de memoria (qué celdas) asignar.

- ▶ ¿Cómo agrupamos la memoria?
  - ▶ De a celdas.
  - ▶ En Bloques de tamaño fijo.
  - ▶ En Bloques de tamaño variable.

### 3. Organizar la memoria disponible

Pensemos en la memoria como una colección de celdas.

El MM debe elegir qué porción de memoria (qué celdas) asignar.

- ▶ ¿Cómo agrupamos la memoria?
  - ▶ De a celdas.
  - ▶ En Bloques de tamaño fijo.
  - ▶ En Bloques de tamaño variable.
- ▶ ¿Cómo organizamos la memoria libre?
  - ▶ Con un mapa de bits.
  - ▶ Con una lista enlazada.

### 3. Organizar la memoria disponible

Pensemos en la memoria como una colección de celdas.

El MM debe elegir qué porción de memoria (qué celdas) asignar.

- ▶ ¿Cómo agrupamos la memoria?
  - ▶ De a celdas.
  - ▶ En Bloques de tamaño fijo.
  - ▶ En Bloques de tamaño variable.
- ▶ ¿Cómo organizamos la memoria libre?
  - ▶ Con un mapa de bits.
  - ▶ Con una lista enlazada.
- ▶ Mecanismos más sofisticados:
  - ▶ Segmentación.
  - ▶ Paginación.
  - ▶ Segmentación + paginación.

# Más sobre paginación

- ▶ Paginación nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.



# Más sobre paginación

- ▶ Paginación nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.

Pero... ¿Qué pasa si no hay lugar en la memoria? (por lo general,  $\text{MEM\_VIRTUAL} > \text{MEM\_FISICA}$ ).

# Más sobre paginación

- ▶ Paginación nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.

Pero... ¿Qué pasa si no hay lugar en la memoria? (por lo general,  $\text{MEM\_VIRTUAL} > \text{MEM\_FISICA}$ ).

- ▶ Algoritmos de remoción:
  - ▶ FIFO
  - ▶ LRU
  - ▶ Segunda oportunidad
  - ▶ Not Recently Used

# Más sobre paginación

- ▶ Paginación nos permite mapear mucha más memoria de la que realmente tiene el sistema, Si una página no está cargada en ningún marco de página, el MM se encarga de ir a buscarla al disco y cargarla en memoria.

Pero... ¿Qué pasa si no hay lugar en la memoria? (por lo general,  $\text{MEM\_VIRTUAL} > \text{MEM\_FISICA}$ ).

- ▶ Algoritmos de remoción:
  - ▶ FIFO  
La clásica de siempre.
  - ▶ LRU  
Desalojo la página que hace más tiempo que no se usa.
  - ▶ Segunda oportunidad  
Si fue referenciada, le doy otra chance.
  - ▶ Not Recently Used  
Primero desalojo las ni referenciadas ni modificadas. Después las referenciadas y por último las modificadas.

# Ejercicio: Algoritmos de remoción

- ▶ Tengo un sistema con 6 páginas y sólo 4 marcos de página. La memoria comienza vacía.  
Llegan los siguientes pedidos de memoria (número de página) en ese orden:  
1, 2, 1, 3, 4, 3, 5, 6, 2
- ▶ Indique qué página se desaloja tras cada pedido utilizando los algoritmos FIFO, LRU y Second Chance y calcule el *hit-rate* en cada caso.

# Solución

	<b>FIFO</b>		<b>LRU</b>		<b>Second Chance</b>	
	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>
1	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div></div><div></div><div></div></div>	<div><div>1</div><div></div><div></div><div></div></div>
2	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>
1	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>2</div><div>1</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>	<div><div>1</div><div>2</div><div></div><div></div></div>
3	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>2</div><div>1</div><div>3</div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>	<div><div>1</div><div>2</div><div>3</div><div></div></div>
4	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>2</div><div>1</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>
3	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>2</div><div>1</div><div>4</div><div>3</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>
5	<div><div>5</div><div>2</div><div>3</div><div>4</div></div>	<div><div>2</div><div>3</div><div>4</div><div>5</div></div>	<div><div>1</div><div>5</div><div>3</div><div>4</div></div>	<div><div>1</div><div>4</div><div>3</div><div>5</div></div>	<div><div>1</div><div>2</div><div>3</div><div>4</div></div>	<div><div>2</div><div>3</div><div>4</div><div>1</div></div>
	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div>1</div><div>5</div><div>3</div><div>4</div></div>	<div><div>3</div><div>4</div><div>1</div><div>5</div></div>
6	<div><div>5</div><div>6</div><div>3</div><div>4</div></div>	<div><div>3</div><div>4</div><div>5</div><div>6</div></div>	<div><div>6</div><div>5</div><div>3</div><div>4</div></div>	<div><div>4</div><div>3</div><div>5</div><div>6</div></div>	<div><div>1</div><div>5</div><div>3</div><div>4</div></div>	<div><div>4</div><div>1</div><div>5</div><div>3</div></div>
	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div></div>	<div><div>1</div><div>5</div><div>3</div><div>6</div></div>	<div><div>1</div><div>5</div><div>3</div><div>6</div></div>
2	<div><div>5</div><div>6</div><div>2</div><div>4</div></div>	<div><div>4</div><div>5</div><div>6</div><div>2</div></div>	<div><div>6</div><div>5</div><div>3</div><div>2</div></div>	<div><div>3</div><div>5</div><div>6</div><div>2</div></div>	<div><div>2</div><div>5</div><div>3</div><div>6</div></div>	<div><div>5</div><div>3</div><div>6</div><div>2</div></div>

- ▶ Hit-Rate (FIFO) =  $2 / 9$
- ▶ Hit-Rate (LRU) =  $2 / 9$
- ▶ Hit-Rate (SC) =  $2 / 9$

## 4. Asegurar la protección de la memoria

Un proceso no debería poder usar memoria que no reservó.

## 4. Asegurar la protección de la memoria

Un proceso no debería poder usar memoria que no reservó.

- ▶ Paginación provee una solución muy elegante para esto:  
Cada proceso tiene su propia tabla de páginas.



## 4. Asegurar la protección de la memoria

Un proceso no debería poder usar memoria que no reservó.

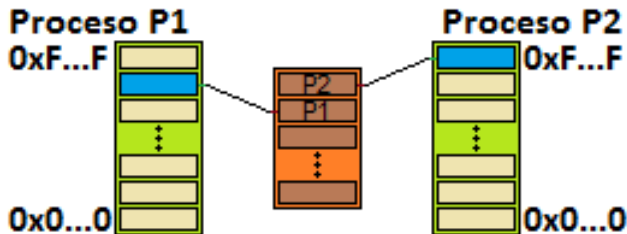
- ▶ Paginación provee una solución muy elegante para esto:  
Cada proceso tiene su propia tabla de páginas.
- ▶ De forma similar, puede utilizarse segmentación para lo mismo:  
Cada proceso tiene su propia tabla de segmentos.

## 4. Asegurar la protección de la memoria

Un proceso no debería poder usar memoria que no reservó.

- ▶ Paginación provee una solución muy elegante para esto:  
Cada proceso tiene su propia tabla de páginas.
- ▶ De forma similar, puede utilizarse segmentación para lo mismo:

Cada proceso tiene su propia tabla de segmentos.



## 5. Permitir acceso a memoria compartida

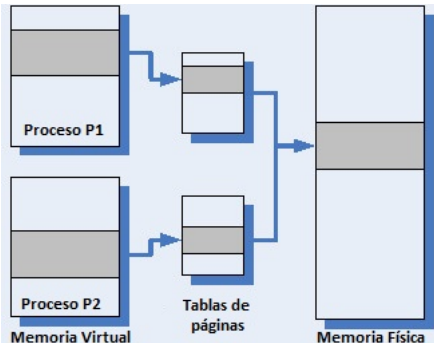
Queremos que dos procesos lean y escriban sobre una misma variable.

## 5. Permitir acceso a memoria compartida

Queremos que dos procesos lean y escriban sobre una misma variable.

- Una vez más, paginación es la respuesta.

Podemos mapear dos páginas al mismo marco de página.



# ¿Qué es malloc?

# ¿Qué es malloc?

Opciones:

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.



# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

Veamos qué dicen las man-pages...

- ▶ man 2 syscalls (<http://goo.gl/a0w0Jk>)
- ▶ man malloc (<http://goo.gl/0AoxAs>)

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

Veamos qué dicen las man-pages...

- ▶ man 2 syscalls (<http://goo.gl/a0w0Jk>)
- ▶ man malloc (<http://goo.gl/0AoxAs>)

Entonces..

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

Veamos qué dicen las man-pages...

- ▶ man 2 syscalls (<http://goo.gl/a0w0Jk>)
- ▶ man malloc (<http://goo.gl/0AoxAs>)

Entonces..

- ▶ malloc no es syscall sino un “memory manager” provisto por la lib-C. Ergo, malloc vive y se ejecuta en el espacio de usuario.

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

Veamos qué dicen las man-pages...

- ▶ man 2 syscalls (<http://goo.gl/a0w0Jk>)
- ▶ man malloc (<http://goo.gl/0AoxAs>)

Entonces..

- ▶ malloc no es syscall sino un “memory manager” provisto por la lib-C. Ergo, malloc vive y se ejecuta en el espacio de usuario.
- ▶ Pertenece a una familia de funciones integrada por malloc, *calloc*, *realloc* y *free*.

# ¿Qué es malloc?

Opciones:

- ▶ Lo contrario de buennoc (cuak!).
- ▶ Una syscall.
- ▶ Una función provista por una lib.

Veamos qué dicen las man-pages...

- ▶ man 2 syscalls (<http://goo.gl/a0w0Jk>)
- ▶ man malloc (<http://goo.gl/0AoxAs>)

Entonces..

- ▶ malloc no es syscall sino un “memory manager” provisto por la lib-C. Ergo, malloc vive y se ejecuta en el espacio de usuario.
- ▶ Pertenece a una familia de funciones integrada por malloc, *calloc*, *realloc* y *free*.
- ▶ El standard de C sólo define el comportamiento de estas funciones. Cómo se implementan depende de cada implementación y del SO subyacente.

# ¿Cómo funciona malloc?

Como dijimos antes, hay muchas implementaciones con diversos comportamientos.

# ¿Cómo funciona malloc?

Como dijimos antes, hay muchas implementaciones con diversos comportamientos.

En líneas generales, malloc administra un pool de “cachitos” de memoria dentro del *Heap* de un proceso. Esos cachitos comúnmente se conocen como *buckets* y malloc se encarga de tener agrupados dentro de su pool a los *buckets* de igual tamaño (generalmente por medio de una lista enlazada) para poder entregarlos rápidamente al usuario cuando este los solicita.



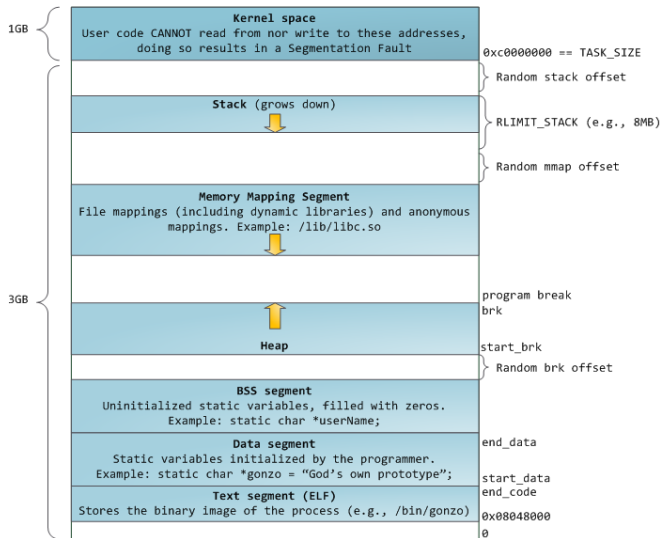
# ¿Cómo funciona malloc?

Como dijimos antes, hay muchas implementaciones con diversos comportamientos.

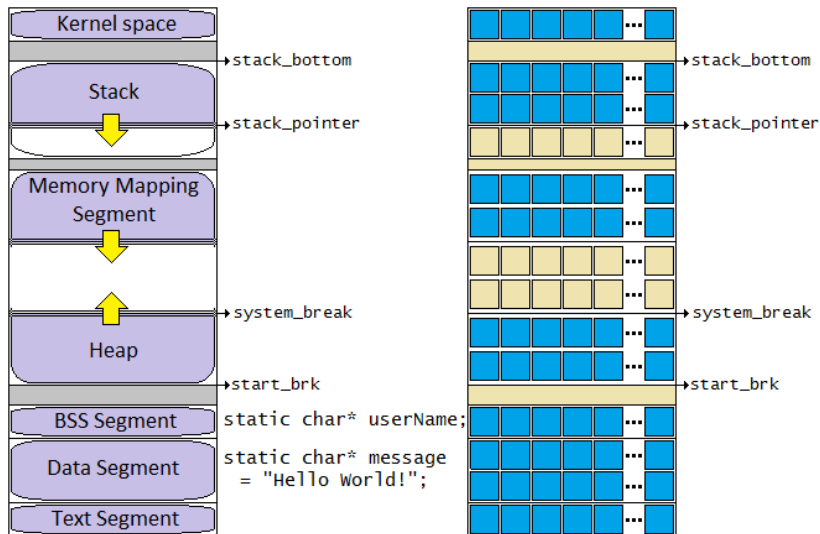
En líneas generales, malloc administra un pool de “cachitos” de memoria dentro del *Heap* de un proceso. Esos cachitos comúnmente se conocen como *buckets* y malloc se encarga de tener agrupados dentro de su pool a los *buckets* de igual tamaño (generalmente por medio de una lista enlazada) para poder entregarlos rápidamente al usuario cuando este los solicita.

Naturalmente, el espacio inicialmente reservado para el *Heap* puede ser insuficiente para un proceso, en cuyo caso, malloc debe encargarse de pedir más memoria al kernel para poder seguir armando buckets y cumplir con la petición de memoria por parte del proceso.

# ¿Recuerdan esto de la teórica?



# Ahora tratemos de pensarlo así...



En Linux, la implementación más común de malloc lo que hace es pedir memoria al kernel por medio de:

- ▶ `brk()` y `sbrk()`: cambian el límite del heap del proceso
- ▶ `mmap()`: mapea una porción grande de memoria (una página) en el espacio de direcciones del proceso.

En Linux, la implementación más común de malloc lo que hace es pedir memoria al kernel por medio de:

- ▶ `brk()` y `sbrk()`: cambian el límite del heap del proceso
- ▶ `mmap()`: mapea una porción grande de memoria (una página) en el espacio de direcciones del proceso.

¿`brk()`, `sbrk()` y `mmap()` son syscalls?

# ¿Cómo pide más memoria malloc? (1)

Frente a un nuevo pedido de memoria malloc hace lo siguiente:

1. malloc busca entre las porciones libres de memoria del *Heap* uno o más buckets contiguos que puedan satisfacer el tamaño pedido de memoria.
2. Si eso falla, entonces o bien ocupamos toda la memoria del *Heap*, o bien (lo más probable) la distribución de memoria fue tal que el *Heap* quedó fragmentado y no hay un espacio de memoria contiguo que podamos ofrecer. En cualquier caso, malloc intentará extender el *Heap* del proceso a través `brk()`/`sbrk()` (o de `mmap()` en algunos casos).

## ¿Cómo pide más memoria malloc? (2)

- 3 La llamadas `brk()`/`sbrk()` al Kernel ajustarán, dentro del *mm\_struct* del proceso, el tope del *Heap* para que éste sea más extenso. Esto no aumenta la memoria directamente, ya que no se está mapeando memoria física al proceso, sino que lo que se hace es ampliar el espacio de direcciones reservado para el *Heap*.
- 4 Finalmente, cuando alguna de esas posiciones de memorias no mapeadas es accedida (generalmente por una lectura/escritura de la implementación de `malloc`) se produce una excepción de *Page Fault*. Dicha excepción será atrapada por el Kernel y producirá una invocación al administrador de páginas para obtener una nueva página de memoria física para el frame que generó la excepción.

# ¿Cómo pide más memoria malloc? (3)

Algunas observaciones importantes respecto a lo anterior:

- ▶ `brk()` NO aloja memoria. Simplemente amplía el espacio de direccionamiento del *Heap*. La asignación de memoria se dará cuando al intentar acceder a ese rango de direcciones se produzca un *Page Fault* y dispare el mecanismo del Kernel para manejar esa excepción y obtener una página libre en memoria.
- ▶ Si usan `strace` en un programa que use mucho `malloc` van a ver que no siempre se realiza llamados a `brk()`. En promedio, se dará cada 100-200 llamados a `malloc`.
- ▶ Si el pedido de memoria es muy grande, éste en lugar de devolver *buckets* del *Heap*, `malloc` optará por invocar a `mmap()` y crear un mapping anónimo de una página entera de memoria dentro del segmento de *Memory Mapping*.
- ▶ Algunas implementaciones de `malloc` usan una estrategia de expansión exponencial entre pedidos. Es decir, cada vez que agotan el *Heap*, van pidiendo más y más memoria al Kernel.



# ¿Cómo devuelve memoria malloc? (1)

Para devolver la memoria pedido usando malloc se puede hacer uso de la función `free`. Esta función simplemente le avisa a malloc que el o los buckets de memoria solicitados previamente ahora se encuentran nuevamente libres como para poder ser reutilizados en el futuro.

# ¿Cómo devuelve memoria malloc? (1)

Para devolver la memoria pedido usando malloc se puede hacer uso de la función `free`. Esta función simplemente le avisa a malloc que el o los buckets de memoria solicitados previamente ahora se encuentran nuevamente libres como para poder ser reutilizados en el futuro.

**Pero entonces... ¿Cómo le devolvemos la memoria al Kernel?**

# ¿Cómo devuelve memoria malloc? (1)

Para devolver la memoria pedido usando malloc se puede hacer uso de la función `free`. Esta función simplemente le avisa a malloc que el o los buckets de memoria solicitados previamente ahora se encuentran nuevamente libres como para poder ser reutilizados en el futuro.

## **Pero entonces... ¿Cómo le devolvemos la memoria al Kernel?**

En la mayoría de las implementaciones `free` no devuelve nada al Kernel, sino que devuelve la memoria liberada al pool de buckets en el *Heap*. La principal implicancia de esto es que el *footprinting* de memoria de un proceso típicamente crece y no se achica independientemente de cuánta memoria la aplicación libere usando `free`.

En otras palabras, el *Heap* no se achica nunca y en consecuencia nunca se desalojan páginas de memoria para devolverlas al Kernel.

# ¿Cómo devuelve memoria malloc? (2)

## ¿Qué hace el Kernel si nadie le devuelve la memoria?

El Kernel de Linux funciona más o menos como una línea aérea. Básicamente, no tiene problema en sobrevenderte un vuelo y después dejarte en tierra. Su política entonces es siempre decirte que sí frente a un pedido de memoria.

Si eventualmente, suficientes programas quisieran realmente usar toda la memoria que solicitaron, entonces tendrá lugar la siguiente serie de eventos:

1. el Kernel comienza a hacer swapping de páginas.
2. el Kernel entra en *trashing*.
3. entra en acción el *OOM killer* (Out of Memory Killer. A.K.A: Hasta la vista baby!) y mata procesos aleatoriamente con SIGQUIT.

# ¿Cómo devuelve memoria malloc? (3)

## ¿Qué puedo hacer yo como programador?

En Linux existe la función `malloc_trim()` de la lib-C para forzar a que el proceso devuelva la memoria al Kernel. Sin embargo, no es portable.

En la práctica, este no es un problema usual. Sin embargo, si tienen un proceso que corre por mucho tiempo (por ejemplo un daemon) y que ocasionalmente realiza muchos llamado a `malloc`, entonces sí puede aparecer este tipo de problema. En ese caso pueden:

1. Reemplazar la implementación standard de `malloc` por una que admita liberación real de memoria.
2. Cambiar su aplicación para que haga `fork()` antes de que haga muchos pedidos de memoria, dejar que los pedidos los haga el hijo, que termine su procesamiento y devuelva el resultado por alguna forma de IPC. Luego matarlo o hacerlo terminar para que la memoria se libere y vuelva al Kernel.

# La familia \*alloc

- malloc()** Reserva en memoria la cantidad solicitada de bytes y devuelve un puntero apuntando al primer byte a dicha porción de memoria.
- calloc()** Reserva un espacio en memoria para un array de elementos, inicializa ese espacio con ceros y devuelve un puntero a dicho espacio.
- realloc()** Cambia el tamaño de un espacio de memoria previamente reservado con con `malloc` o `calloc`.
- free()** Libera un espacio de memoria reservado previamente con `malloc` o `calloc`.
- kalloc()** Es de uso exclusivo del Kernel. Reserva un memoria física y contigua que no puede ser swapeada.
- valloc()** Similar a `kalloc`, pero devuelve porciones de memoria no necesariamente continuas.

# Juguemos a ser malloc

¿Qué porción de memoria me conviene asignar?

# Juguemos a ser malloc

¿Qué porción de memoria me conviene asignar?

¿Cómo agrupamos la memoria?

De a celdas.

En Bloques de tamaño fijo.

En Bloques de tamaño variable.

¿Cómo organizamos la memoria libre?

Con un mapa de bits.

Con una lista enlazada.



# Juguemos a ser malloc

¿Qué porción de memoria me conviene asignar?

¿Cómo agrupamos la memoria?

De a celdas.

En Bloques de tamaño fijo.

En Bloques de tamaño variable.

¿Cómo organizamos la memoria libre?

Con un mapa de bits.

Con una lista enlazada.

► Algoritmos de elección de bloque libre:

► First fit

► Best fit

► Worst fit

► Quick fit

# Juguemos a ser malloc

¿Qué porción de memoria me conviene asignar?

¿Cómo agrupamos la memoria?

De a celdas.

En Bloques de tamaño fijo.

En Bloques de tamaño variable.

¿Cómo organizamos la memoria libre?

Con un mapa de bits.

Con una lista enlazada.

► Algoritmos de elección de bloque libre:

► First fit

La primera sección de memoria contigua del tamaño necesario.

► Best fit

De todas las secciones de tamaño mayor o igual al tamaño necesario, tomo la más chica.

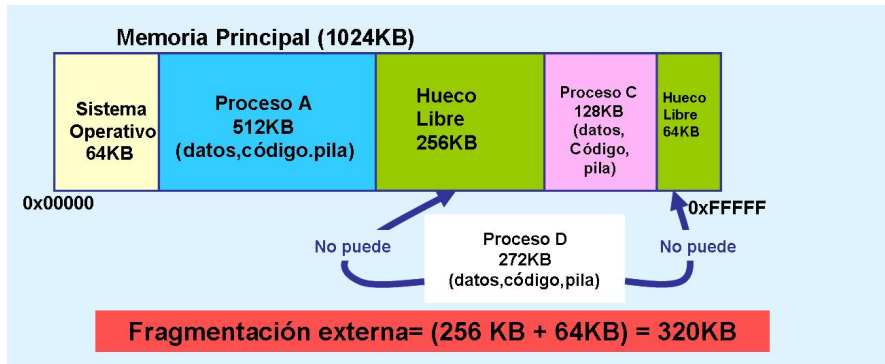
► Worst fit

Mejor tomo la más grande.

► Quick fit

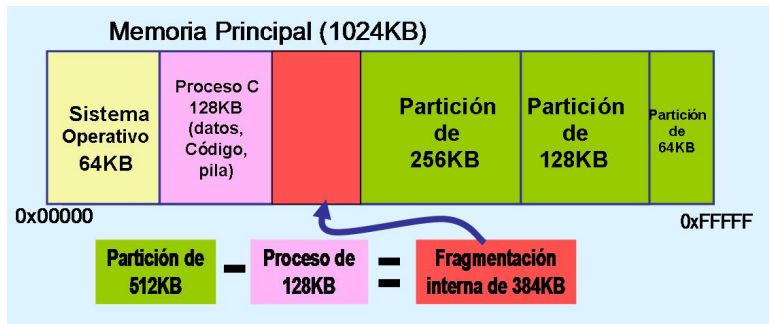
Además me guardo listas de bloques de determinados tamaños.

# Fragmentación Externa



Bloques *pequeños* de memoria no contiguos.

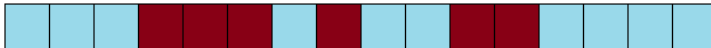
# Fragmentación Interna



Memoria desperdiciada dentro de una partición (un bloque o página).

# Ejercicio: Algoritmos de elección de bloque libre

- ▶ Tengo un sistema con 16 MB de memoria **sin particionar** que direcciona a byte. El estado actual de la memoria es el siguiente:

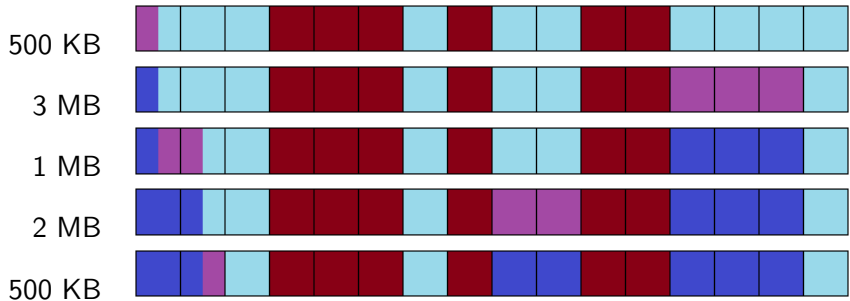


Llegan los siguientes pedidos de memoria en ese orden:

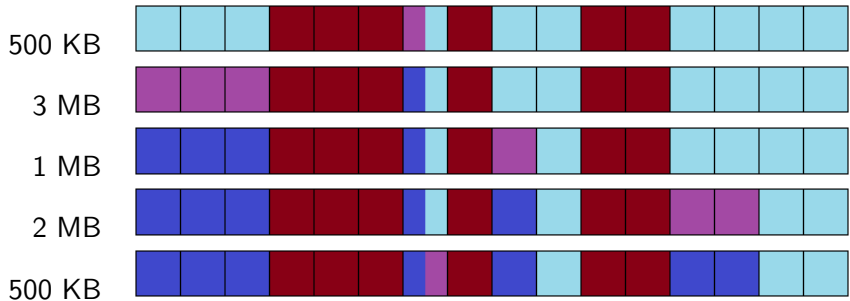
500 KB, 3 MB, 1 MB, 2MB, 500 KB.

- ▶ Indique qué bloques se asignan para cada pedido utilizando first-fit, best-fit y worst-fit.

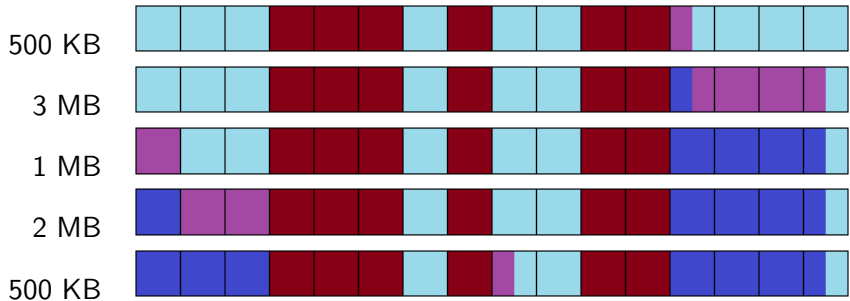
# Solución First-Fit



# Solución Best-Fit

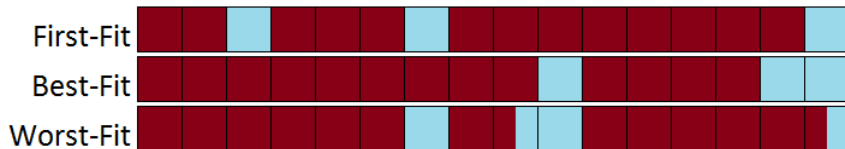


# Solución Worst-Fit





# Entonces, ¿Cuál es mejor?



- ▶ ¿Qué pasa cuando una página no está cargada en la memoria?

- ▶ ¿Qué pasa cuando una página no está cargada en la memoria?
1. Un proceso accede a una dirección (virtual) de memoria.
  2. La MMU traduce la dirección virtual a dirección física (accede a la entrada en la última tabla de páginas).
  3. Lee el atributo correspondiente a presencia en la memoria.
  4. Si es negativo, se produce la interrupción Page Fault.
  5. Se ejecuta la **RAI** correspondiente.
  6. Si la memoria está llena, se ejecuta el algoritmo de remoción.
  7. Si la página que se va a desalojar fue modificada, hay que bajarla al disco.
  8. Se carga en el lugar liberado la página solicitada.
  9. Se vuelve a ejecutar la instrucción del proceso que accede a la dirección solicitada.

# Thrashing

- ▶ Situación en la que el SO pasa más tiempo cargando páginas que ejecutando procesos.

Supongan que tenemos 2 procesos y un solo marco de página disponible.

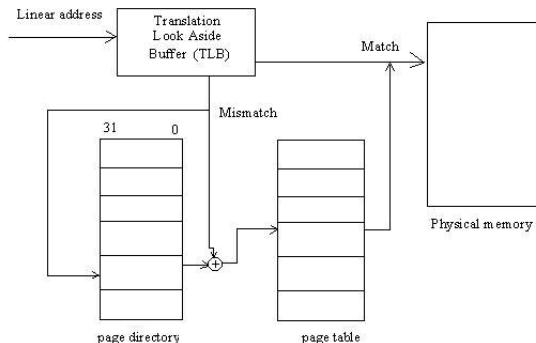
Cada proceso usa una sólo página, pero cada vez que ejecute, su página no va a estar cargada.

- ▶ Siempre va a estar cargada la página del otro.

# TLB (Translation Lookaside Buffer)

Buffer de Traducción Adelantada.

- ▶ Es una caché que guarda 'traducciones'.
- ▶ Paginación de 4 niveles: Cuatro accesos a memoria (1 por cada tabla) más uno para leer la página.



- ▶ ¿Entonces para qué me guardo todas las tablas?

- ▶ Dijimos que al crear un nuevo proceso se duplica toda su memoria.

Sabemos que, en general después de un `fork()` viene un `exec()`. `exec()` inutiliza todas las páginas de memoria, entonces ¿para qué nos gastamos en duplicar todo?

- ▶ Copy-on-Write: Sólo duplico (copy) cuando alguno de los procesos escribe (write).

# FIN

## ¿Preguntas?