

# Gestión de memoria

Yolanda Becerra Fontal  
Juan José Costa Prats

Facultat d'Informàtica de Barcelona  
Universitat Politècnica de Catalunya  
BarcelonaTech  
2014-2015QT

# Índice

- Memoria dinámica
- Memoria virtual
- Memoria compartida

# Índice: Memoria dinámica

- Introducción
- Memoria dinámica para el sistema operativo
  - Primera aproximación: “Cutre-system”
  - Buddy System
  - Slab allocator
- Memoria dinámica para el usuario
  - sbrk
  - malloc/free
    - Doug Lea allocator (dlmalloc)

# Introducción

- Memoria dinámica
  - Que es?
    - Mecanismo para gestionar el espacio dentro de una zona de memoria
  - Para que sirve?
    - Permite reservar/liberar memoria bajo demanda
    - Facilita el trabajo al programador a la hora de implementar estructuras de datos dinámicas como listas, arboles,... en el que no se conoce a priori su tamaño final
    - Evita limites por culpa de variables estáticas
    - Mejor aprovechamiento de la memoria
  - Operaciones
    - Reservar (*malloc*)
    - Liberar (*free*)

# Introducción

- Validar nuevas zonas del espacio lógico de direcciones
  - MMU
  - Estructura de datos del SO que describe el espacio
- Asignar memoria física
  - ¿Cuándo?
    - En el momento de hacer la reserva
    - O cuando se accede por primera vez
  - ¿Cómo?
    - ¿Consecutiva?
  - ¿Cuánto?
    - ¿Unidad de asignación?

# Memoria dinámica para el sistema operativo

- SO no usa paginación para su espacio lógico
  - Mecanismos para reducir la fragmentación y acelerar la reserva de memoria
  - Soporte a dispositivos que interactúan directamente con la memoria física
- Reserva un segmento de memoria física para sus datos

# Primera aproximación

- Cutre-system
  - Definir zona de memoria estática
  - Solo aceptamos reservas
    - Puntero a la última dirección válida no usada
    - Reserva sólo incrementa este puntero

# Cutre-system

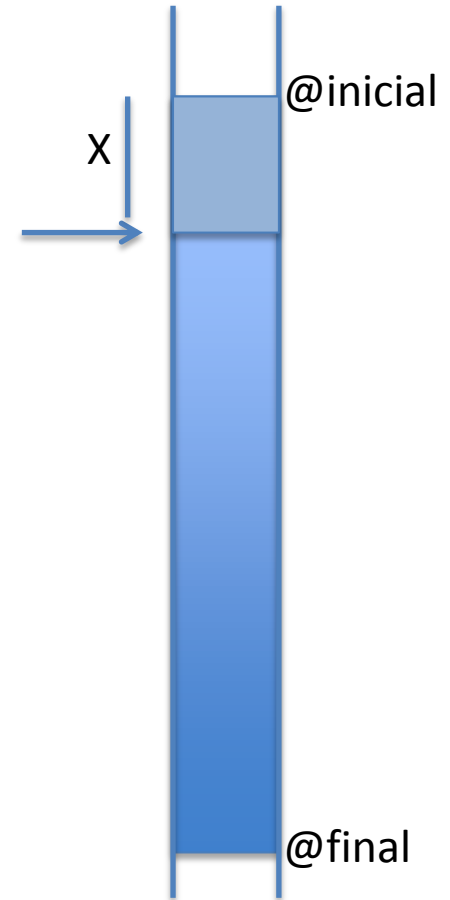
- Zona de memoria para gestión dinámica
  - Direcciones entre @inicial i @final
  - Puntero a la 1ª dirección libre





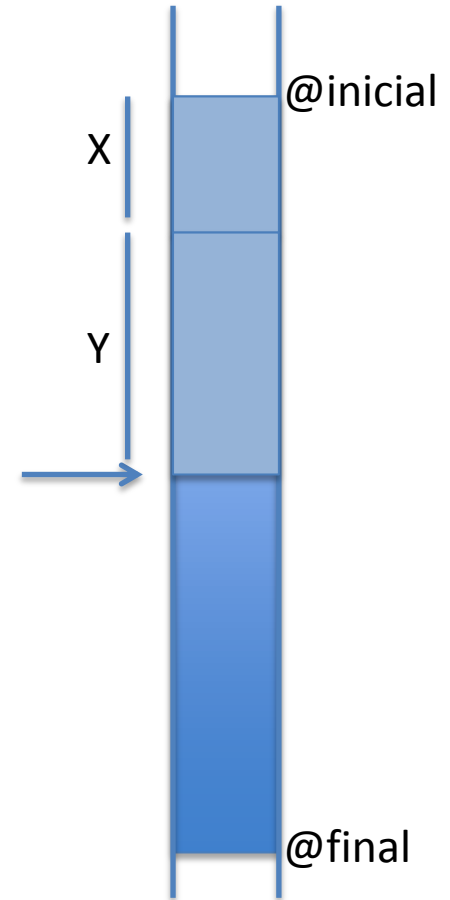
# Cutre-system

- Reserva de memoria
  - Petición de X bytes
    - `p = malloc(X)`
  - Actualización puntero
  - Devolvemos valor del puntero
    - `p = @inicial`



# Cutre-system

- Reserva de memoria 2
  - Petición de Y bytes
    - $p = \text{malloc}(Y)$
  - Actualización puntero
  - Devolvemos valor del puntero
    - $p = @inicial + X$



# Cutre-system

- Ventajas:
  - Fácil de implementar
  - Eficiente
    - Hacer una reserva solo implica incrementar un puntero
- Inconvenientes:
  - No es posible liberar memoria y, por lo tanto, reutilizar una petición de memoria usada previamente

# Buddy system

- Power-of-2 allocator
  - Estructura para mantener los bloques libres de la memoria física
  - Solo reserva tamaños que son potencias de 2
  - Operaciones para
    - dividir un bloque en 2 (splitting)
    - o para juntar 2 bloques consecutivos (coalescing)

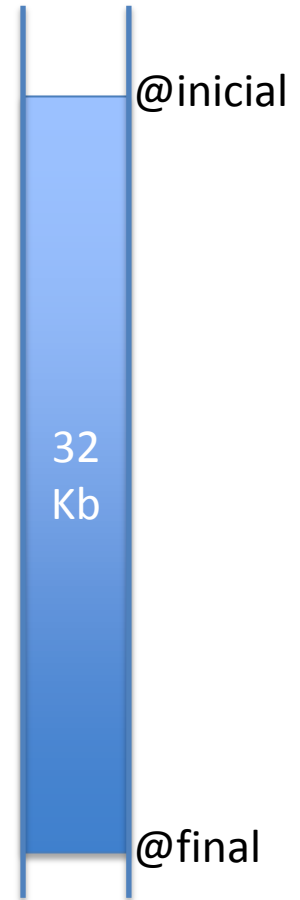
# Buddy system

- Zona de memoria para gestión dinámica
  - Direcciones entre @inicial i @final
  - Direcciones físicas consecutivas
- Estructura para mantener lista de bloques libres del mismo tamaño
  - Inicialmente 1 único bloque con toda la memoria



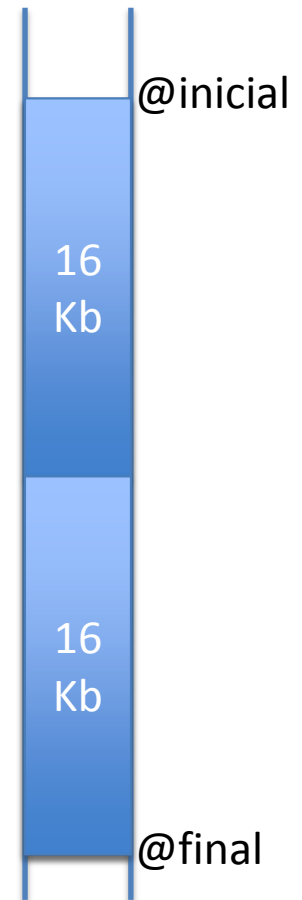
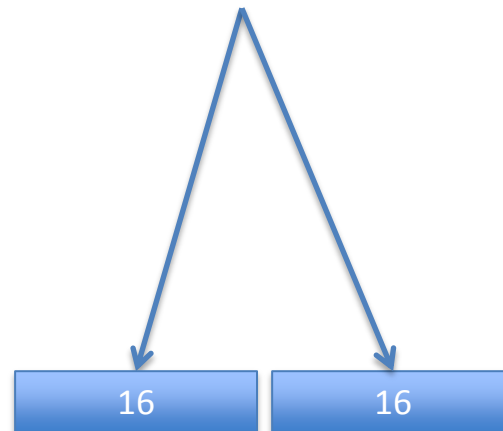
# Buddy system

- Reserva de memoria
  - Petición de 4K bytes
    - `p = malloc(4096)`



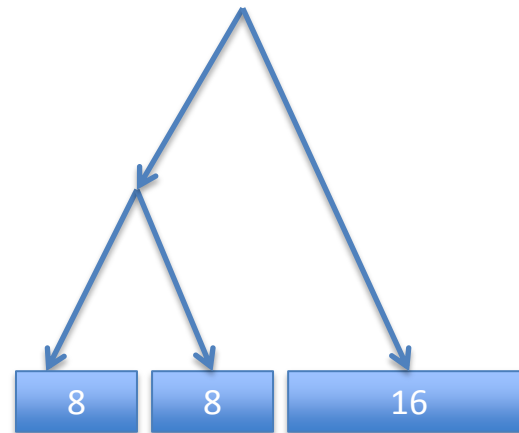
# Buddy system

- Reserva de memoria
  - Petición de 4K bytes
    - `p = malloc(4096)`
  - Split a 16Kb



# Buddy system

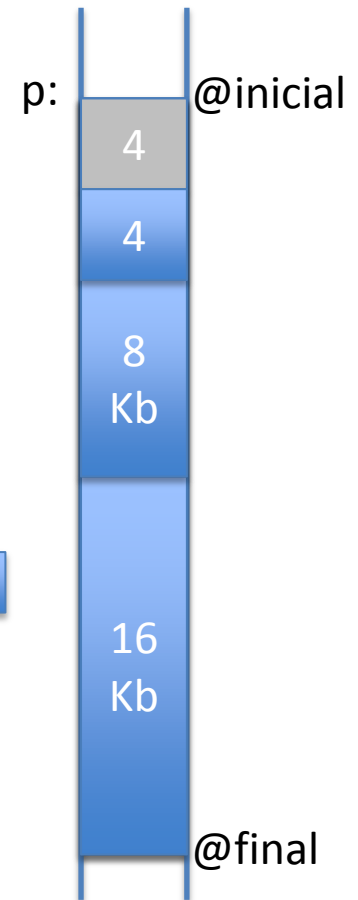
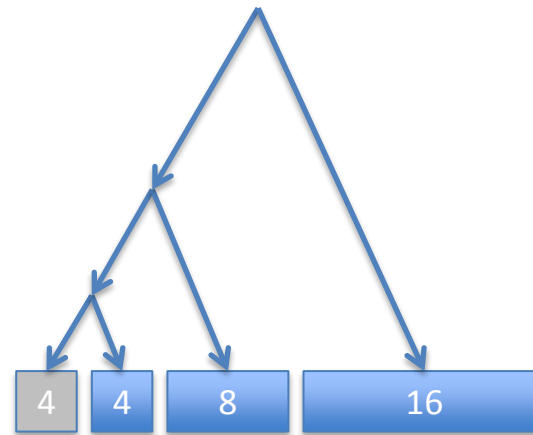
- Reserva de memoria
  - Petición de 4K bytes
    - `p = malloc(4096)`
  - Split a 16Kb
  - Split a 8Kb





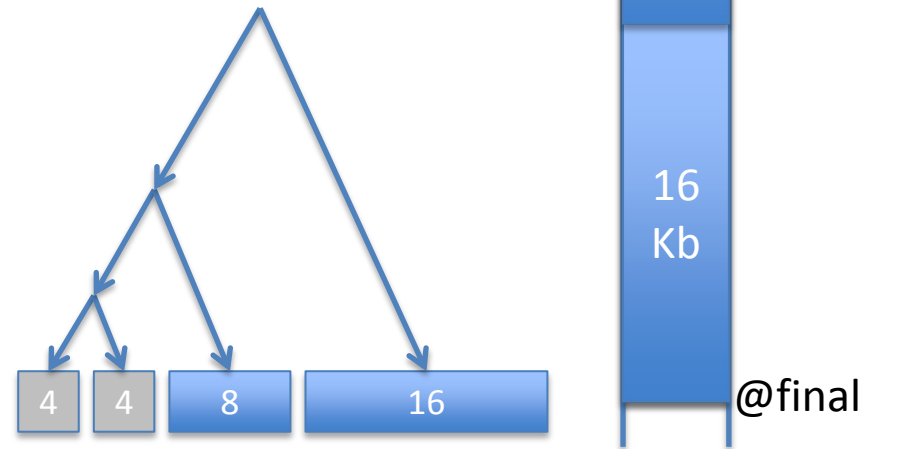
# Buddy system

- Reserva de memoria
  - Petición de 4K bytes
    - `p = malloc(4096)`
  - Split a 16Kb
  - Split a 8Kb
  - Split a 4Kb
  - Devolvemos dirección del 1r bloque
    - `p = @inicial`



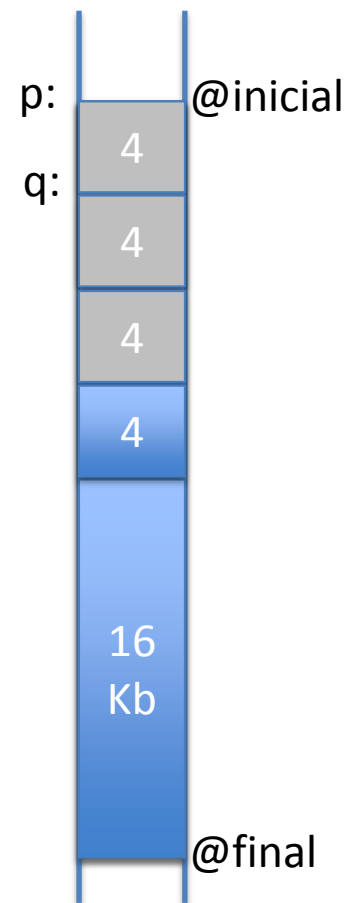
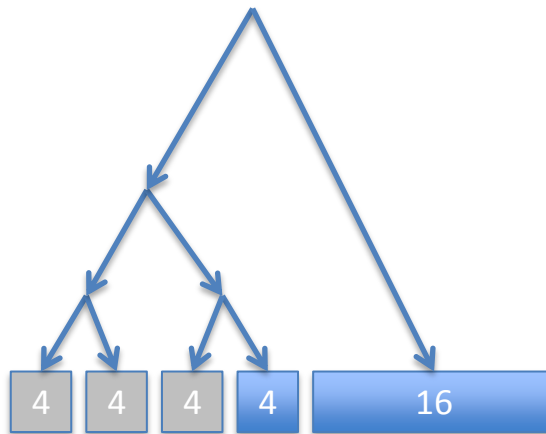
# Buddy system

- Reserva de memoria 2
  - Petición de 4K bytes
    - $q = \text{malloc}(4096)$
  - Devolvemos dirección del  $2n$  bloque
    - $q = @inicial + 4Kb$



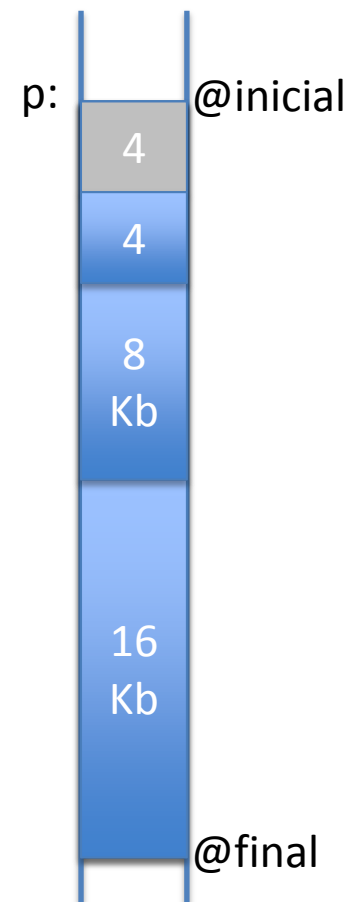
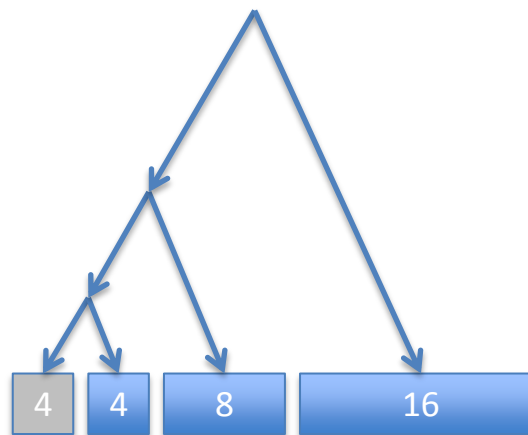
# Buddy system

- Puede satisfacer
  - malloc (8kb)
  - malloc (16kb)
- Pero:
  - malloc (4kb)
  - malloc (20kb)
    - NO ok!
    - Fragmentación externa!



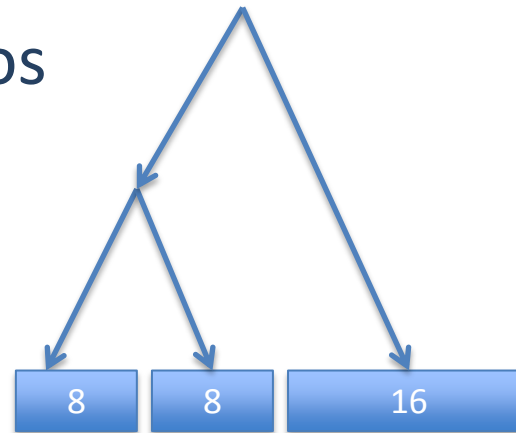
# Buddy system

- Liberar memoria
  - free (q)
  - free (p)



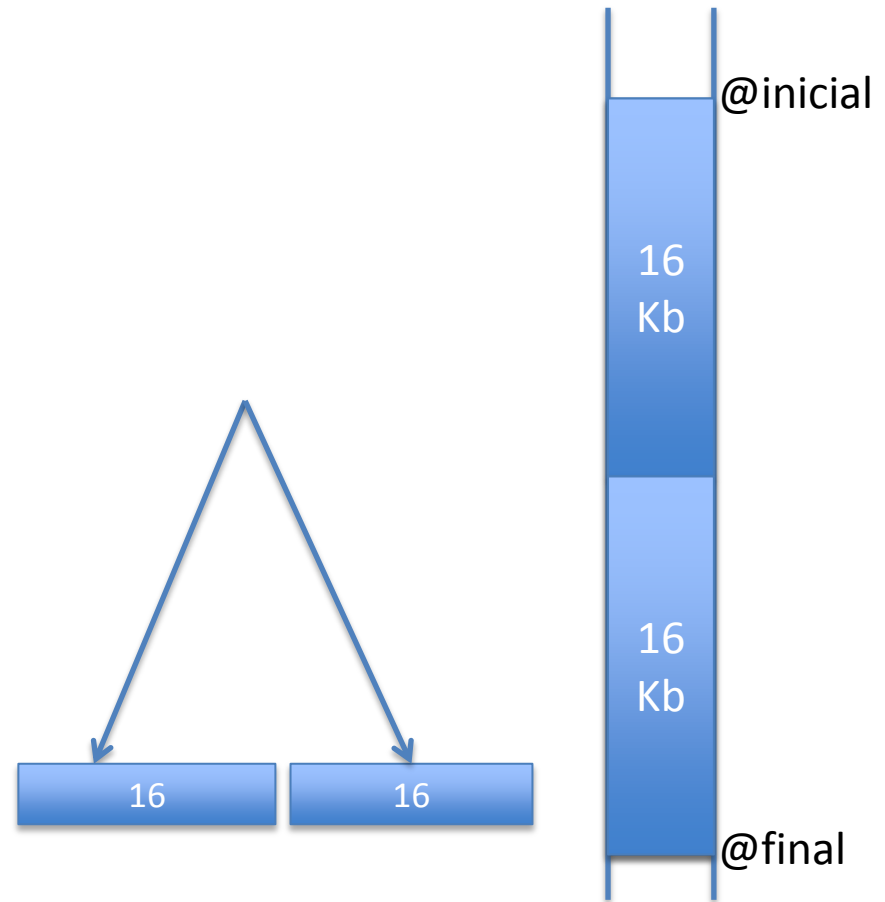
# Buddy system

- Liberar memoria
  - free (q)
  - free (p)
  - 2 bloques consecutivos libres → Coalesce



# Buddy system

- Liberar memoria
  - free (q)
  - free (p)
  - 2 bloques consecutivos libres → Coalesce
  - 2 bloques consecutivos libres → Coalesce



# Buddy system

- Liberar memoria
  - free (q)
  - free (p)
  - 2 bloques consecutivos libres → Coalesce
  - 2 bloques consecutivos libres → Coalesce
  - 2 bloques consecutivos libres → Coalesce



# Buddy system

- Que estructuras se necesitarían para implementar este sistema?
- Como se detectan los bloques consecutivos?



# Buddy system

- Ventajas
  - Relativamente fácil de implementar
  - Rápido
- Inconvenientes
  - Tamaños sólo pueden ser potencias de 2
    - Fragmentación interna
  - Aunque haya memoria libre, puede no satisfacer la petición

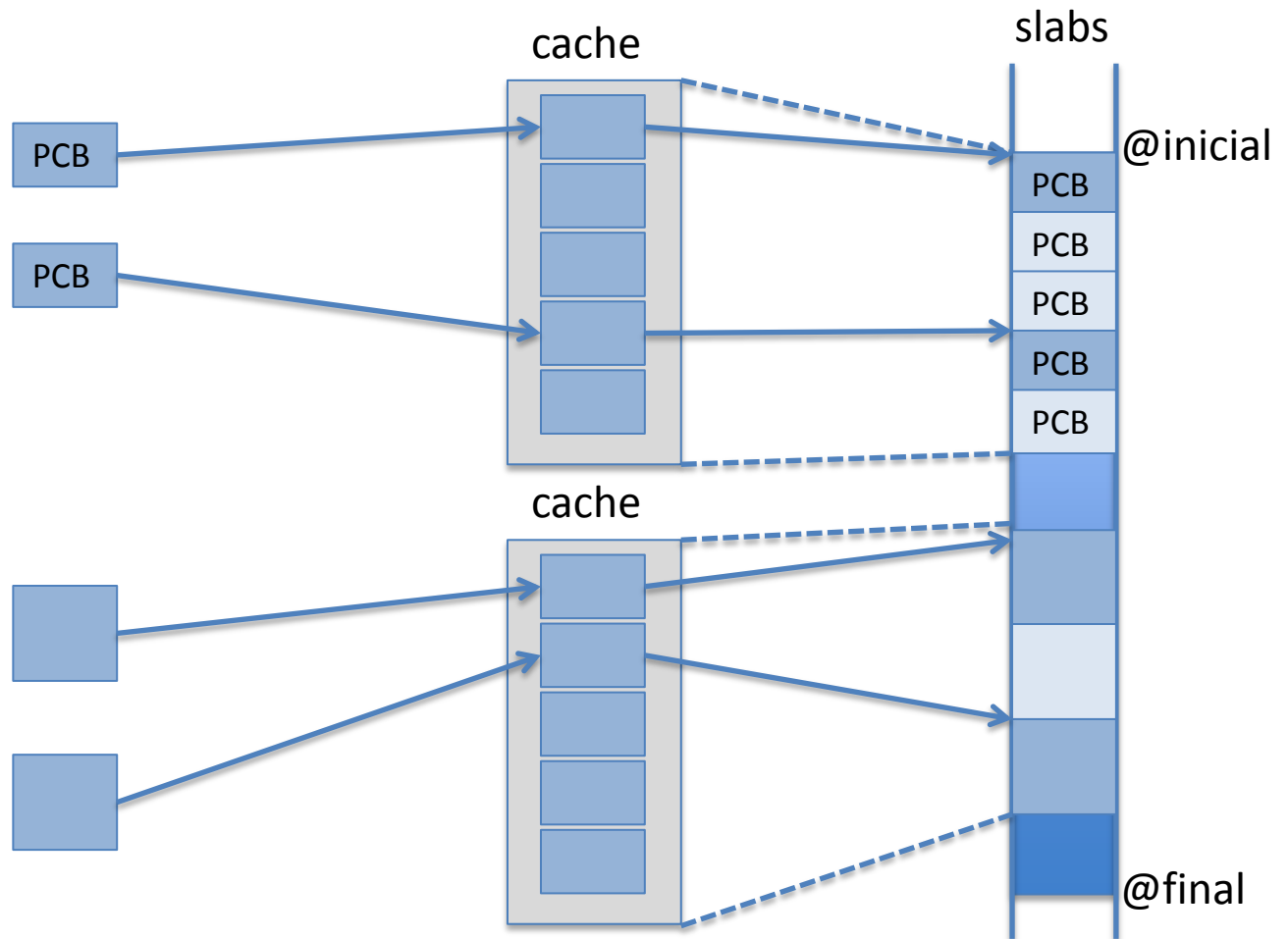
# Slab allocator

- Intenta resolver problemas buddy system
- Idea:
  - Estructuras que se usan y destruyen continuamente
  - Reaprovechar estructuras creadas previamente
    - Por ej: PCBs, semáforos, ...
- Usar caches para guardar objetos de kernel

# Slab allocator

- *Slab*
  - Región de memoria de 1 o más páginas consecutivas
- *Cache*
  - Agrupación de 1 o más *slabs*
  - Cada *cache* contiene objetos del mismo tipo (mismo tamaño) y información de si está en uso o no
    - 1 cache para PCBs, 1 para semáforos, 1 para ficheros, ...

# Slab allocator



# Slab allocator

- Ventajas
  - No hay perdida de espacio
  - Añadir espacio para la cache es simplemente añadir un nueva zona de *slab*
  - Muy rapido
- Inconvenientes
  - Preallocatar todos los objetos en el slab, marcandolos como llibres

# Usuario

- El sistema será el encargado de satisfacer las peticiones del usuario
- Implica que el espacio de direcciones varíe
  - Zona especial dedicada a mem. dinámica: Heap

# sbrk

- `void * sbrk (int incr)`
  - Incrementa la zona de memoria dinámica (Heap) en *incr* bytes, reservando esa cantidad en sistema
  - Si el incremento es negativo, libera esa cantidad
    - El espacio de direcciones se modifica
  - Devuelve la dirección de memoria a usar
  - El usuario debe ser totalmente consciente del uso

# sbrk

- Ventajas
  - Rápido
- Inconveniente
  - La reserva/liberación es lineal, sólo se incrementa o decrementa el espacio dedicado para memoria dinámica
    - Gestión interna de ese espacio → usuario



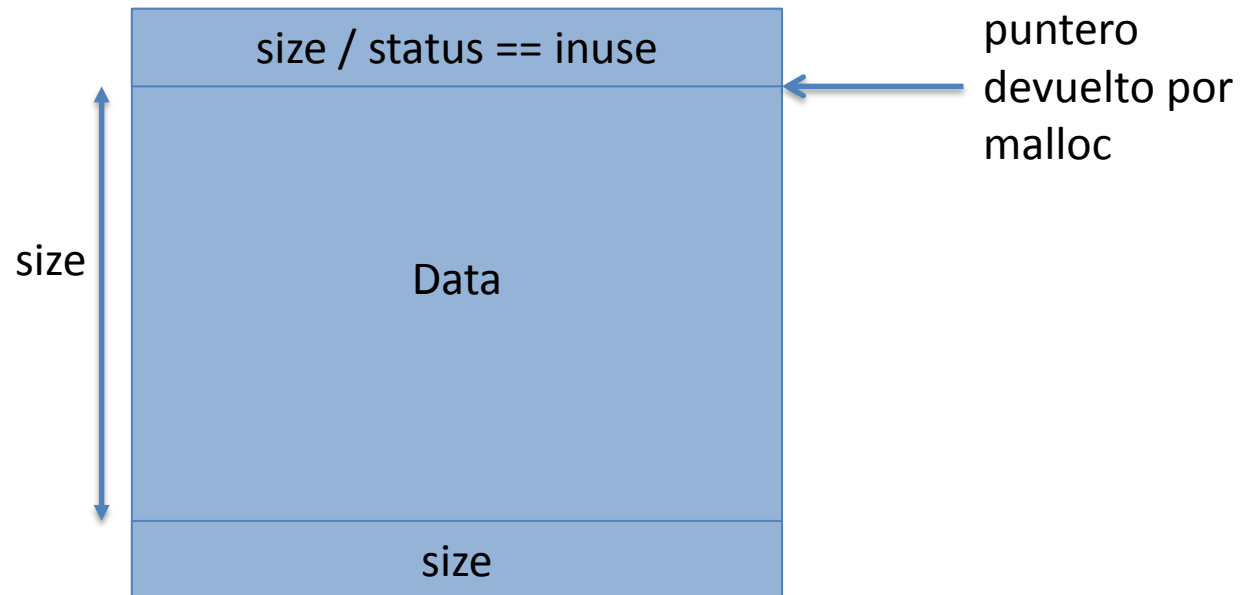
# Doug Lea Malloc

- Gestión del espacio de memoria dinámica
- Memoria en el *Heap* asignada mediante *chunks* alineados a 8-bytes con:
  - Cabecera
  - Zona de memoria usable por el usuario

Fuente: "A memory allocator" Doug Lea. December 1996. (<http://gee.cs.oswego.edu/dl/html/malloc.html>)

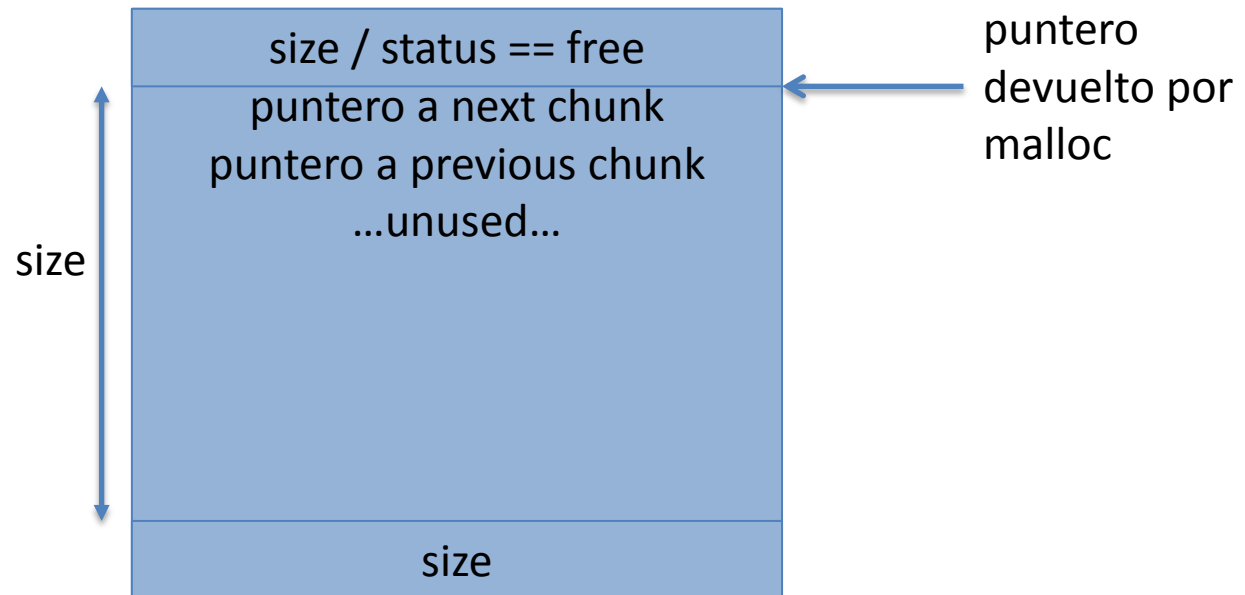
# Doug Lea Malloc

- Chunk de memoria usado
  - Boundary tags





# Doug Lea Malloc

- Chunk de memoria libre
  - Boundary tags



# Doug Lea Malloc

- Estructura para mantener zonas de memoria libres agrupadas por tamaño
  - *Bins* de *chunks* libres de tamaño fijo
  - bins[128]
  - index    2   3   4   ...   31 ...   64    65    66 ...   127
  - size       16 24 32       256    512    576    640    2<sup>31</sup>
  - chunks              

# Doug Lea Malloc

- Lista doblemente encadenada de *chunks*
  - Para eliminar rápidamente
- *Chunks* de tamaño  $\leq 512$  bytes
  - Se guardan directamente en la posición asociada a su tamaño
- *Chunks* mayores
  - Se guardan en una posición próxima a su tamaño
- Peticiones grandes  $\rightarrow$  mmap
  - Por defecto  $\rightarrow$  peticiones  $\geq 1\text{Mb}$
- Búsqueda de libres:
  - smaller-first, best-fit

# Doug Lea Malloc

- Al reservar puede hacer splitting
- Al liberar puede hacer coalescing
  - Si hay bloques libres consecutivos
    - Para ello miramos el chunk anterior y el posterior

# Doug Lea Malloc

- Ventajas
  - Totalmente genérico: cualquier objeto
- Inconvenientes
  - Perdida de espacio por la codificación del chunk
    - Mínimo de 16 bytes! (arquitecturas de 32 bits)

# Indice: Memoria Virtual

- ¿Qué es?
- ¿Qué necesitamos para implementarlo?
- Estructuras de datos: linux



# ¿Qué es?

- Extiende la idea de la carga bajo demanda
- Objetivo
  - Reducir la cantidad de memoria física asignada a un proceso en ejecución
    - Un proceso realmente sólo necesita memoria física para la instrucción actual y los datos que esa instrucción referencia
  - Aumentar el grado de multiprogramación
    - Cantidad de procesos en ejecución simultáneamente
- Técnica que permite espacios de direcciones lógicos mayores que la memoria física instalada en la máquina

# ¿Qué es?

- Primera aproximación: intercambio de procesos (*swapping*)
  - Idea: proceso activo en memoria (el que tiene la CPU asignada)
    - Si no suficiente memoria libre → expulsar a otro proceso (*swap out*)
      - Procesos no residentes: *swapped out*
    - Almacén secundario o de soporte (*backing storage*):
      - Mayor capacidad que la que ofrece la memoria física
      - Típicamente una zona de disco: espacio de intercambio (*swap area*)
    - Reanudar la ejecución de un proceso *swapped out* → cargarlo de nuevo en memoria (*swap in*)
      - Ralentiza la ejecución
  - Evolución de la idea
    - Expulsar sólo partes de procesos
    - Se aprovecha la granularidad que ofrece la paginación

# Algoritmo

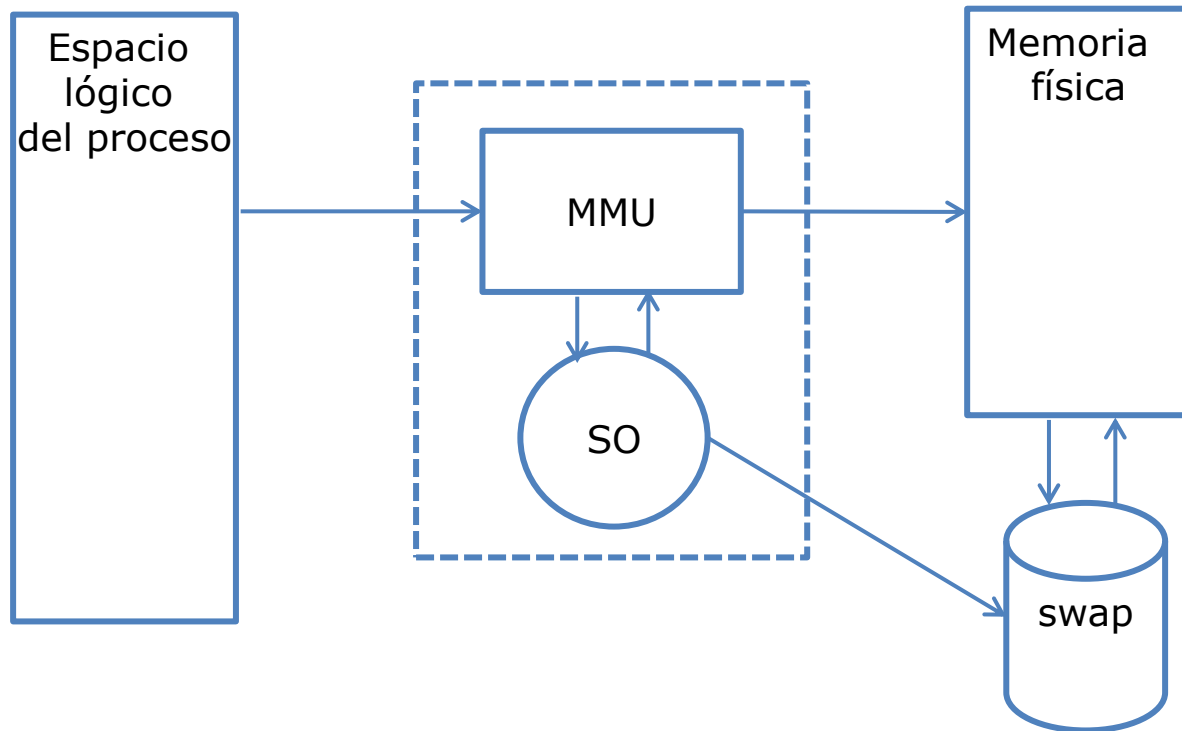
- Detectar memoria no residente
- Asignación de memoria física
- Algoritmo de reemplazo
  - Seleccionar memoria víctima
- Gestión del Backing storage
  - ¿Qué almacén de soporte?
  - Localizar memoria en backing storage

# ¿Qué necesitamos?

- Soporte hw para la traducción y detección de memoria no residente
  - Mismo mecanismo que para carga bajo demanda
  - Excepción de fallo de página
    - ¿Página válida?
    - ¿De dónde se recupera su contenido?

# ¿Qué necesitamos?

- Memoria virtual basada en paginación
  - **Espacio lógico de un proceso está distribuido entre memoria física (páginas residentes) y área de swap (páginas no residentes)**



# Asignación de memoria física

- ¿Qué memoria está disponible?
  - Estructura de datos para saber los frames libres
    - Ej: Lista de frames disponibles
  - Algoritmo de selección
    - Ej: Primero de la lista
- Actualizar espacio de direcciones con el frame seleccionado
- Working set
  - Cantidad de memoria física mínima para el proceso

# Algoritmo de reemplazo

- Algoritmo que decide cuándo es necesario hacer swap out de páginas
  - ¿Cuándo?
  - ¿Cuántas?
  - ¿Cuáles?
    - LRU, FIFO, Optimo
- Objetivo minimizar fallos de página e intentar que siempre haya marcos disponibles para resolver un fallo de página

# Algoritmos de reemplazo

- Optimo
  - Se expulsa la que no se va a utilizar en el futuro inmediato
    - Predicción
    - No se puede implementar
- FIFO
  - Se expulsa la que hace más tiempo que está en uso
  - Implementación sencilla
  - No tiene en cuenta la frecuencia de uso



# Algoritmos de reemplazo

- LRU (Least Recently Used)
  - Pasado reciente aproxima futuro inmediato
  - Contar accesos a páginas y se selecciona la que tiene un contador menor
  - Costoso de implementar
    - Deberían registrarse **TODOS** los accesos
  - Se usan aproximaciones
    - Segunda oportunidad
      - usada/no usada desde la última limpieza

# Gestión del backing storage

- ¿Qué dispositivo?
  - Zona de disco: área de swap
    - Acceso directo: no utiliza sistema de ficheros
- Operaciones de gestión
  - Guardar frame
    - Seleccionar bloque libre
  - Recuperar frame
    - SO debe almacenar la posición de cada frame en el backing storage

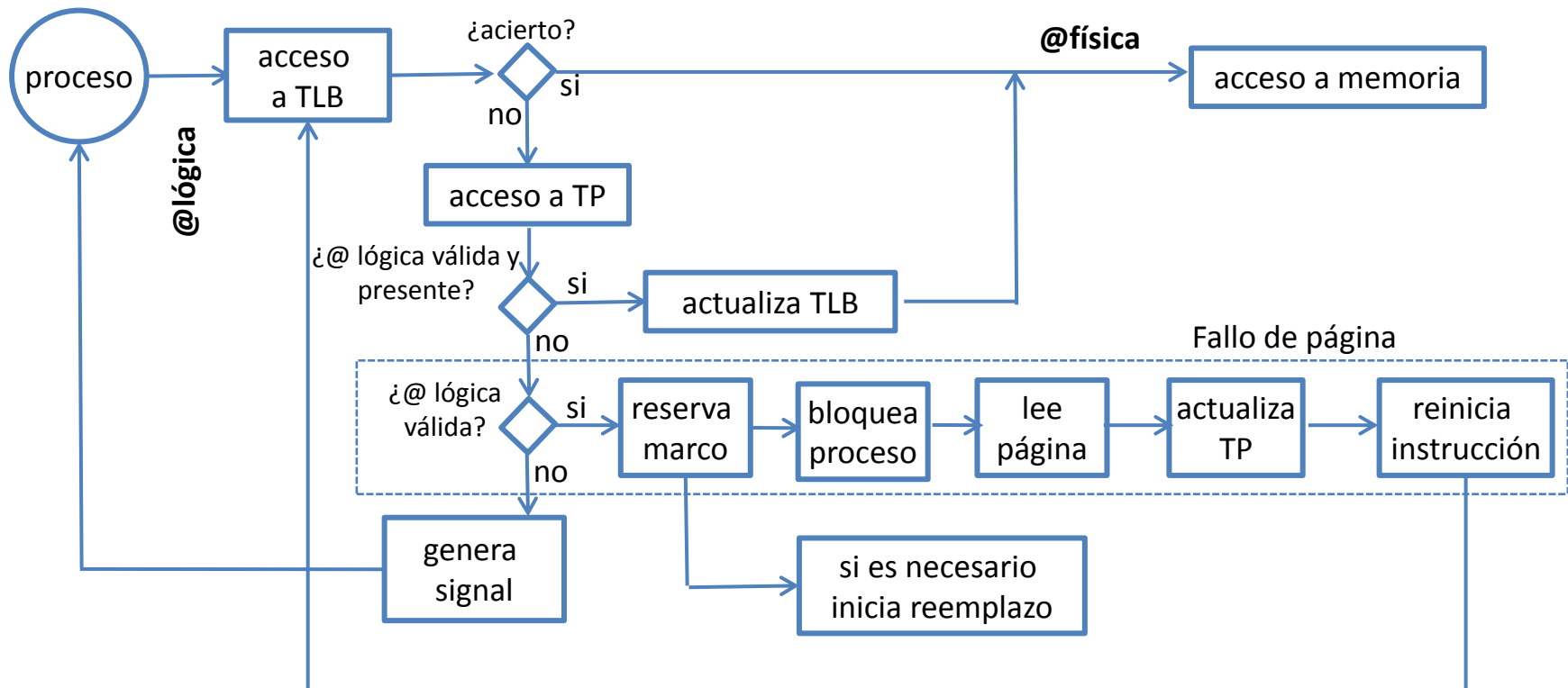
# Linux

- Estructuras de datos
  - Espacio de direcciones:
    - Tabla de páginas
    - mm\_struct: lista de regiones (vm\_area\_struct)
  - Frames libres
    - Organizados en listas
  - Area de swap
    - Partición de disco o fichero
    - vm\_area\_struct contiene la posición de la región en disco

# Linux

- Algoritmo de reemplazo: LRU second chance
  - Bit de referencia en la tabla de páginas
    - Cada vez que se accede a una página se marca como referenciada
    - Cada vez que se ejecuta el algoritmo de reemplazo
      - Páginas referenciadas: se limpia el bit y se invalida el acceso
      - Páginas no referenciadas: se seleccionan como víctimas
  - Rango de memoria libre
    - Se comprueba al servir un fallo de página y cada cierto tiempo
    - Se limpian n frames
  - Parámetros configurables por el administrador

# Algoritmo de acceso a memoria



# Índice: Memoria Compartida

- Introducción
- Nivel de usuario
- Implementación

# Introducción

- Variables compartidas entre procesos
  - Mecanismo para comunicación entre procesos
  - Interfaz de acceso sencillo y eficiente
  - Posibles complicaciones: condición de carrera
- Regiones compartidas por defecto
  - Entre procesos: **ninguna**
    - Es necesario llamadas a sistema para pedir regiones compartidas
  - Entre Threads de la misma tarea: **todas**
    - Incluso la pila, aunque hay que tener en cuenta la visibilidad de las variables
    - No hace falta ninguna llamada a sistema: todas las variables globales son visibles desde todos los threads del proceso
  - Entre Threads de tareas diferentes: **ninguna**
    - Es necesario llamadas a sistema para pedir regiones compartidas

# Nivel de usuario: POSIX

- Interfaz definido en la familia system V
  - Operaciones relacionadas con la memoria compartida
    - Crear región: `shmget`
      - “propietario” de la región
      - Asigna un identificador
    - Mapear en el espacio de direcciones: `shmat`
      - Necesario para poder acceder: asigna rango de direcciones
      - Cualquier proceso que conozca el identificador
    - Liberar del espacio de direcciones: `shmdt`
      - Procesos que tienen mapeada la región
    - Eliminar región: `shmctl`
      - “propietario” de la región



# Creación y mapeo

- `int shmget (key_t key, size_t size, int shmflag)`
  - Key: identificador de la región
  - Size: tamaño
  - Shmflag: `IPC_CREAT`, se puede combinar con `IPC_EXCL`
  - Crea una nueva región de memoria compartida, devuelve el identificador a utilizar en la operación de mapeo o -1 si hay error
- `void * shmat (int id, void *addr, int shmflag)`
  - Id: identificador devuelto por `shmget`
  - Addr: @ inicial en el espacio lógico. Si vale `NULL`, el SO elige una libre
  - Shmflag: permisos
  - Mapea la región compartida en la dirección especificada.
  - Las regiones compartidas se heredan en el `fork`
  - Las regiones compartidas se liberan automáticamente al mutar

# Desmapeo y eliminación

- `int shmdt (void *addr)`
  - `addr`: @ inicial de la región que se va a eliminar del espacio de direcciones
    - Libera la región del espacio de direcciones del proceso que la ejecuta. Devuelve 0 si todo va bien y -1 si hay error
- `int shmctl (int id, int cmd, struct shmid_ds *buf)`
  - `id`: shared memory id
  - `cmd`: operation to perform
    - `IPC_STAT`: fill up buf
    - `IPC_RMID`: mark shared region to be destroyed
    - (...)
  - `buf`: struct to store information about the region (permissions, size, time, pid of creator,...)

# Maapeo de ficheros

- Interfaz pensado para acceder a ficheros a través de memoria
  - Maapeo: mmap
  - Desmaapeo: munmap

# mmap

- `void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset)`
  - `addr`: hint para inicio de la región. Si NULL SO asigna una
  - `length`: tamaño de la región
  - `prot`: permisos de acceso de la región
  - `flags`: modificadores
    - `MAP_SHARED`: cambios se hacen efectivos en el fichero y son compartidos por todos los procesos que lo mapeen
    - `MAP_PRIVATE`: cambios no son persistentes, afectan sólo a la región en memoria
    - `MAP_ANONYMOUS`: no hay fichero de respaldo, memoria inicializada con 0
    - `MAP_FIXED`: `addr` debe ser obligatoriamente la @inicial de la región, si no es posible `mmap` devuelve error.
    - (...)
  - `fd`: fichero que contiene los datos
  - `offset`: desplazamiento dentro del fichero

# munmap

- `int munmap (void *addr, size_t length)`
  - `addr`: dirección de la región que se libera
  - `length`: tamaño de la región

# Implementación en ZeOS

- Simplificación
  - Limitar número de regiones compartidas que puede crear un proceso
  - Limitar tamaño regiones
- `id = shmget(key, size, IPC_CREAT|IPC_EXCL)`
- `addr = shmat(id, addr, NULL);`
  - Permisos siempre `rw`
  - Si `addr == NULL` → ZeOS asigna @ libre
- `shmdt(addr)`
- `shmctl(id, IPC_RMID, NULL)`
  - marca para borrar. Se eliminará en el último detach
- `fork`: hijo hereda regiones mapeadas
- `clone`: threads comparten regiones mapeadas
- `exit`: sólo se desmapea cuando muere el último flujo