

Gestión de memoria

Yolanda Becerra Fontal
Juan José Costa Prats

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

SO2/SOA

Índice

- Memoria dinámica
- Memoria virtual
- Memoria compartida

SO2/SOA

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)

SO2/SOA

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 límites por culpa de variables estáticas
 - Mejor aprovechamiento de la memoria
 - Operaciones
 - Reservar (*malloc*)
 - Liberar (*free*)

SO2/SOA

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?

SO2/SOA

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

Memoria dinámica SO

SO2/SOA

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

Memoria dinámica SO

SO2/SOA

Cutre-system

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




Memoria dinámica SO

SO2/SOA

Cutre-system

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

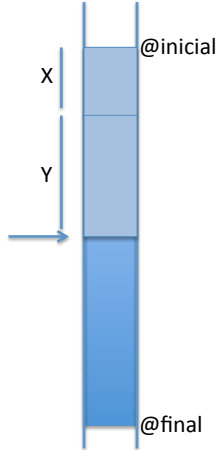


Memoria dinámica SO

SO2/SOA

Cutre-system

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



Memoria dinámica SO

SO2/SOA

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

Memoria dinámica SO

SO2/SOA

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)


Memoria dinámica SO

SO2/SOA

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

Memoria dinámica SO




SO2/SOA

Buddy system

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

Memoria dinámica SO



SO2/SOA

Buddy system

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

```

graph TD
    A[32 Kb] --> B[16 Kb]
    A --> C[16 Kb]
          
```

@inicial

16 Kb

16 Kb

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

```

graph TD
    A[32 Kb] --> B[8 Kb]
    A --> C[8 Kb]
    A --> D[16 Kb]
          
```

@inicial

8 Kb

8 Kb

16 Kb

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

p: @inicial

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

p: @inicial

q: @inicial + 4Kb

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

p: @inicial

q:

16 Kb

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

p: @inicial

@final

Memoria dinámica SO

SO2/SOA

Buddy system

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

Diagram illustrating the buddy system structure. A root node branches into two 8 Kb nodes. These two 8 Kb nodes branch into two 8 Kb nodes and one 16 Kb node.

Diagram illustrating the memory layout. A vertical stack of memory blocks: 8 Kb, 8 Kb, and 16 Kb, with @inicial at the top and @final at the bottom.

Memoria dinámica SO

SO2/SOA

Buddy system

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

Diagram illustrating the buddy system structure. A root node branches into two 16 Kb nodes.


Diagram illustrating the memory layout. A vertical stack of memory blocks: 16 Kb and 16 Kb, with @inicial at the top and @final at the bottom.

Memoria dinámica SO

SO2/SOA

Buddy system

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



Memoria dinámica SO

SO2/SOA

Buddy system

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

Memoria dinámica SO

SO2/SOA

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

Memoria dinámica SO

SO2/SOA

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

Memoria dinámica SO

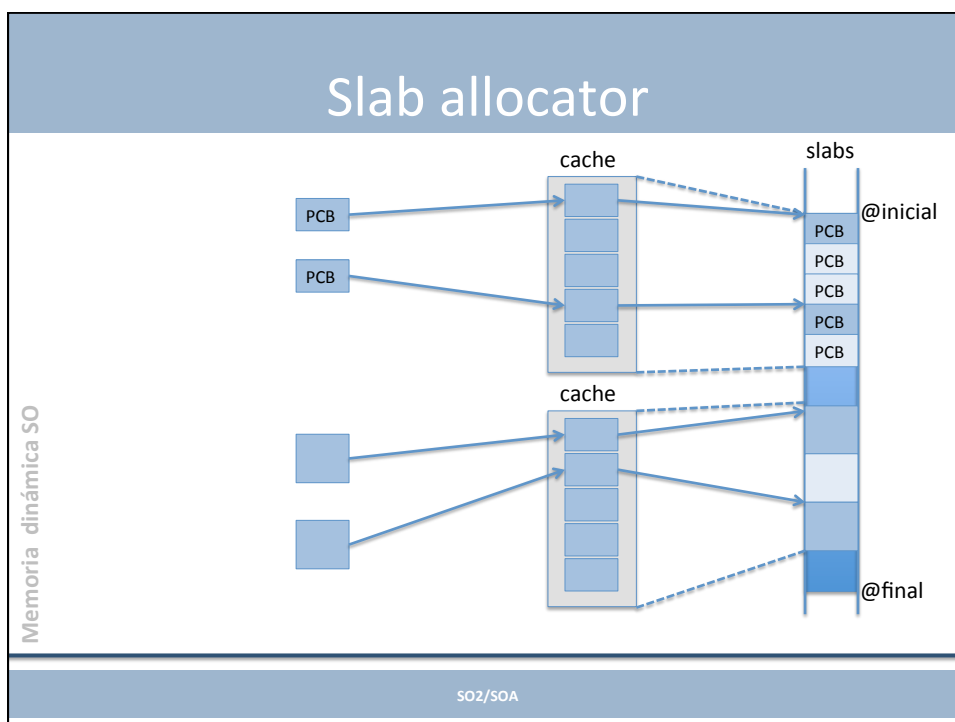
SO2/SOA

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, ...

Memoria dinámica SO

SO2/SOA



Slab allocator

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

Memoria dinámica SO

SO2/SOA

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

Memoria dinámica usuario

SO2/SOA

sbrk	
Memoria dinámica usuario	<ul style="list-style-type: none"> • void * sbrk (int incr) <ul style="list-style-type: none"> – Incrementa la zona de memoria dinámica (Heap) en <i>incr</i> bytes, reservando esa cantidad en sistema – Si el incremento es negativo, libera esa cantidad <ul style="list-style-type: none"> • El espacio de direcciones se modifica – Devuelve la dirección de memoria a usar – El usuario debe ser totalmente consciente del uso
SO2/SOA	

sbrk	
Memoria dinámica usuario	<ul style="list-style-type: none"> • Ventajas <ul style="list-style-type: none"> – Rápido • Inconveniente <ul style="list-style-type: none"> – La reserva/liberación es lineal, sólo se incrementa o decrementa el espacio dedicado para memoria dinámica <ul style="list-style-type: none"> • Gestión interna de ese espacio → usuario
SO2/SOA	

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

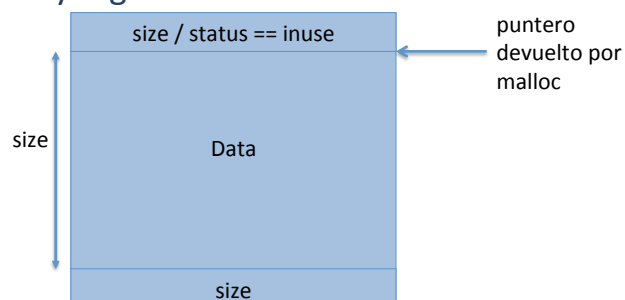
Memoria dinámica usuario

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

SO2/SOA

Doug Lea Malloc

- Chunk de memoria usado
 - Boundary tags



Memoria dinámica usuario

SO2/SOA

Doug Lea Malloc

- Chunk de memoria libre
 - Boundary tags

Memoria dinámica usuario

size

size / status == free
 puntero a next chunk
 puntero a previous chunk
 ...unused...

size

size

← puntero devuelto por malloc

SO2/SOA

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 2^31
 - chunks

Memoria dinámica usuario

SO2/SOA

Doug Lea Malloc

- Lista doblemente encadenada de *chunks*
 - Para eliminar rápidamente
- *Chunks* de tamaño ≤ 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 ≥ 1 Mb
- Búsqueda de libres:
 - smaller-first, best-fit

Memoria dinámica usuario

SO2/SOA

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

Memoria dinámica usuario

SO2/SOA

Doug Lea Malloc

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

Memoria dinámica usuario

SO2/SOA

Indice: Memoria Virtual

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

SO2/SOA

¿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

SO2/SOA

¿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

SO2/SOA

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

SO2/SOA

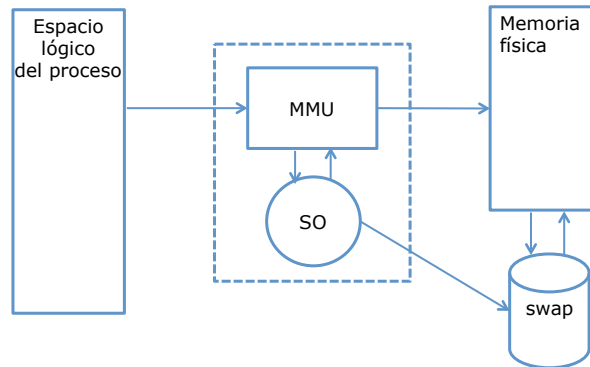
¿Qué necesitamos?

- 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?

SO2/SOA

¿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)



SO2/SOA

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

SO2/SOA

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

SO2/SOA

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

SO2/SOA

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

SO2/SOA

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

SO2/SOA

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

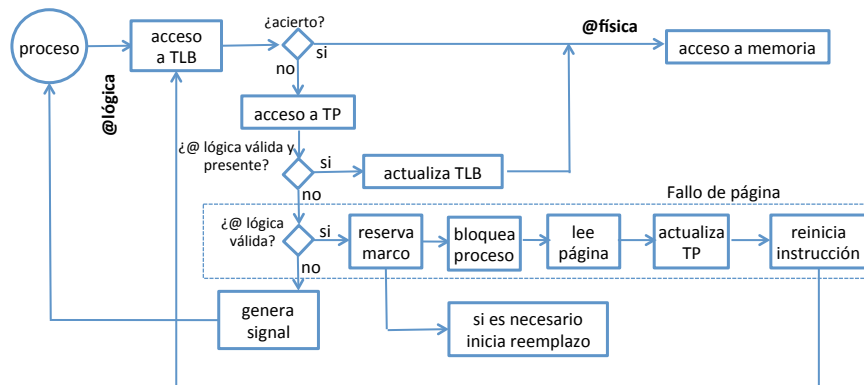
SO2/SOA

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

SO2/SOA

Algoritmo de acceso a memoria



SO2/SOA

Índice: Memoria Compartida

- Introducción
- Nivel de usuario
- Implementación

SO2/SOA

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

SO2/SOA

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

SO2/SOA

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

SO2/SOA

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,...)

SO2/SOA

Mapeo de ficheros

- Interfaz pensado para acceder a ficheros a través de memoria
 - Mapeo: mmap
 - Desmapeo: munmap

SO2/SOA

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

SO2/SOA

munmap

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

SO2/SOA

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

SO2/SOA

Implementación en ZeOS:shmget

- `busca_mem_física_libre(size, page_table_entry *list_pages)`
 - Recorrer vector `phys_mem`.
 - Dedicamos espacio al final
 - No hace falta que sean consecutivas
- `shared_mem_regions`
 - Estructura de datos en kernel que asocie key, con size, direcciones físicas, `num_referencias` y flag pendiente de borrar

SO2/SOA

Implementación en ZeOS:shmat

- `busca_mem_lógica_libre(addr, size)`
 - `address_map`
 - si espacio lógico no consecutivo: lista regiones por proceso: @inicial + size
 - simplificación: dirección inicial + dirección final heap
 - `shared_regions`
 - lista de regiones compartidas por proceso: key + id
- Busca región en `shared_mem_regions` y mapea direcciones físicas en la región lógica reservada

SO2/SOA

Implementación en ZeOS:shmdt

- Buscar región en lista de regiones de proceso
 - Accede a `shared_mem_regions` y decrementa `num_referencias`
 - Si `num_referencias == 0` y `pendiente de borrar == true` se elimina la región
 - Desmapea del espacio lógico: elimina región del campo `shared_regions` del proceso y actualiza tabla de páginas (pero no se libera memoria física, sólo se hará si hay que eliminar la región porque estaba marcada como pendiente de borrar)

SO2/SOA

Implementación en ZeOS:shmctl

- Accede a `shared_mem_regions` para obtener `num_referencias`: si 0 elimina región, si no la marca como pendiente de borrar
- Eliminar región: eliminar info de la lista de regiones del kernel, y marcar como libre la memoria física

SO2/SOA

Implementación en ZeOS:fork, clone, exit

- Fork
 - hereda info sobre regiones y regiones compartidas.
 - incrementar número de referencias de la región en la estructura del kernel
- Clone
 - Comparte info, no hay que hacer nada más
- Exit
 - Sólo ejecutará shmdt si es el último clone (si el número de referencias del directorio llega a 0)