

# Administración de memoria

Diego Fernandez Slezak<sup>1</sup>

<sup>1</sup>Departamento de Computación, FCEyN,  
Universidad de Buenos Aires, Buenos Aires, Argentina

Sistemas Operativos, primer cuatrimestre de 2024

## (2) Hasta ahora...

- Hablamos sobre cómo compartir procesador: los procesos, la multiprogramación, el scheduling.
- Y sobre cómo compartir información: comunicación entre procesos, concurrencia, locking, etc.
- ¿Qué pasa con la memoria?


### (3) La memoria también

- La memoria también se comparte.
- No sólo como una forma de comunicar procesos.
- También para implementar la multiprogramación.
- Vamos a ver distintas formas de organizar la memoria, de complejidad creciente.
- De hecho, vamos a estudiar un subsistema de casi todos los SO: el *manejador de memoria*.
- Sus responsabilidades son:
  - Manejar el espacio libre/ocupado.
  - Asignar y liberar memoria.
  - Controlar el swapping.

## (4) Muy importante

- ¿Cómo identifican a un bruto informático?
- Dice “alocar”.
- No digan “alocar”, digan “reservar”. No sean brutos.

## (5) Lo más sencillo

- Si tenemos un único proceso en memoria, no hay necesidad de compartir nada.
- Sin embargo, si pensamos en multiprogramación, cuando un proceso está bloqueado y ponemos a ejecutar otro, ¿qué hacemos con la memoria?
- El mecanismo más sencillo es el *swapping*, que consiste en pasar a disco el espacio de memoria de los procesos que no se están ejecutando. 
- El problema es que es lento: intercambiar dos procesos requiere grabar uno y leer el otro. Pero tal vez los dos programas entren en memoria.
- OK, entonces dejémoslos en memoria mientras podamos.
- El problema es que tal vez en algún momento sí necesitemos mandarlos a disco, y luego, cuando les corresponda correr de nuevo, el espacio de memoria que les toque no sea el mismo.

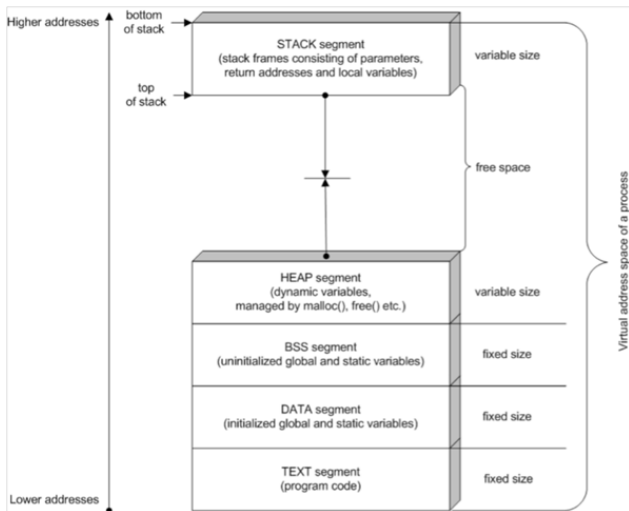
## (6) Algunos problemas

- Podríamos tener fragmentos fijos, pero sería muy ineficiente.
- Podríamos recorrer todo el programa modificándolo, pero es muy costoso.
- Una solución es tener un registro que haga de base, y hacer que todas las direcciones del programa sean relativas.
- Además, ¿cómo aseguramos que un proceso no lea los datos del otro?
- Por otra parte, ¿cómo sabemos qué pedazos de memoria tenemos libres y en dónde conviene ubicar un programa?
- Estos problemas se conocen como:
  - Reubicación (cambio de contexto, swapping).
  - Protección (memoria privada de los procesos).
  - Manejo del espacio libre (evitando la fragmentación).
- Empecemos por la fragmentación.

## (7) Fragmentación de memoria


- La fragmentación es un problema porque podría generar situaciones donde tenemos suficiente memoria para atender una solicitud, pero no es continua.
- Es fácil ver que si no se hace nada para evitarla el problema se puede volver arbitrariamente grave.
- Una alternativa sería compactar, pero esto además de requerir de reubicación, es muy costoso en tiempo. Es directamente imposible en los SO RT.
- Por lo tanto, es mejor evitarla.
- Un primer enfoque, que es muy usado, consiste en organizar la memoria de la siguiente manera:

## (8) Organización de la memoria

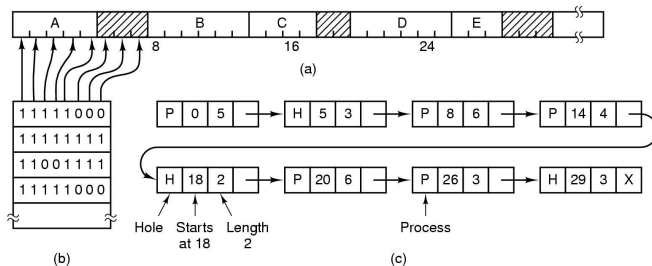




## (9) Cómo organizar la memoria

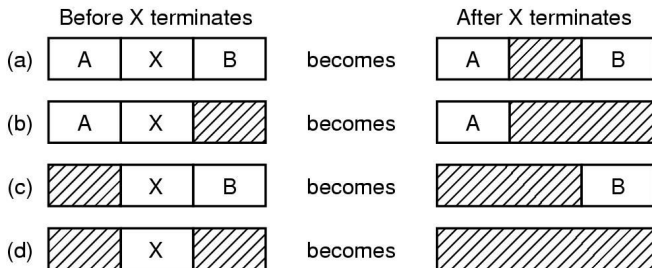
- Una forma de organizar la memoria es a través de un bitmap.
  - La divido en bloques de igual tamaño. Digamos, 4 KB.
  - Cada posición del bitmap representa un bloque, que tiene 0 si está libre, 1 si está ocupado.
  - Si bien asignar y liberar es sencillo, encontrar bloques consecutivos requiere una barrida lineal.
  - Además, hay una tensión constante entre granularidad vs. tamaño del bitmap.
  - No es muy usado.
- Otra, es mediante una lista enlazada. 
  - Cada nodo de la lista representa a un proceso o a un bloque libre, en cuyo caso figuran el tamaño del bloque y sus límites.
  - Liberar sigue siendo  $O(1)$ : una manipulación de punteros más ver si los nodos previos y anteriores eran también espacio libre.
  - Asignar es similar una vez que decidí dónde.
  - Ahora, ¿dónde?

## (10) Bitmaps y listas enlazadas



Pedazo de memoria con 5 procesos y 3 espacios libres

## (11) Listas enlazadas



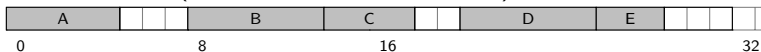
Cuatro combinaciones de vecinos para el proceso X (*coalescing*: (b), (c) y (d)).

## (12) Organización de memoria (cont.)

- Dónde asignar en una lista de bloques:
  - *First fit*: en el primer bloque donde entra, asigno.
    - Es rápido.
    - Tiende a fragmentar la memoria, partiendo bloques grandes.
  - *Best fit*: me fijo dónde entra más justo.
    - Es más lento.
    - Y sorprendentemente, tampoco es mejor: llena la memoria de pequeños bloquecitos inservibles.
  - Variación: *Quick fit*: mantengo además una lista de los bloques libres de los tamaños más frecuentemente solicitados.
  - *Buddy system*: usa *splitting* de bloques.
- Todo estos esquemas fallan porque son muy ingenuos.
- Algunos producen *fragmentación externa* (bloques libres pero pequeños y dispersos), otros *fragmentación interna* (espacio desperdiciado dentro de los propios bloques).

## (13) Políticas de asignación: análisis

- Fragmentación ¿Best fit es mejor que First fit?
- Consideremos el siguiente ejemplo.
- Estado inicial (el mismo que vimos antes)



- Secuencia de solicitudes: (F, 1), (G, 2), (H, 2), (I, 2)

- First fit



- Best fit: ¡no hay lugar para I!



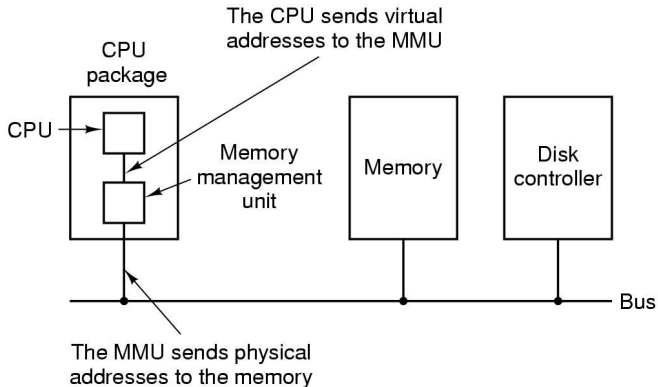
## (14) Organización de la memoria (cont.)

- En la práctica se usan esquemas que conocen un poco más sobre la distribución de los pedidos y realizan manejos más sofisticados.
- Curiosos, leer: “Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel”, de MK McKusick y J Karels.
- Cómo asignar memoria de manera rápida y eficiente sigue siendo un problema interesante.
- Las soluciones actualmente usadas en la práctica no son nada triviales.
- Supongamos que esa parte del problema está solucionada, sólo para poder concentrarnos en las otras.

## (15) Memoria virtual

- Mencionamos antes el problema de la reubicación.
- Va de la mano con otro problema: si tengo  $N$  bytes de memoria y un programa de tamaño  $M > N$  pero que no necesita más de  $K < N$  bytes a la vez, debería poder correrlo. ¿Cómo hago?
- Básicamente la solución consiste en combinar swapping con *virtualización del espacio de direcciones*, y a eso se lo llama *memoria virtual*. ⚠
- Requiere un poco de ayuda del HW. Una unidad llamada *Memory Management Unit (MMU)*. ⚠


## (16) MMU



### Ubicación y funcionamiento de la MMU



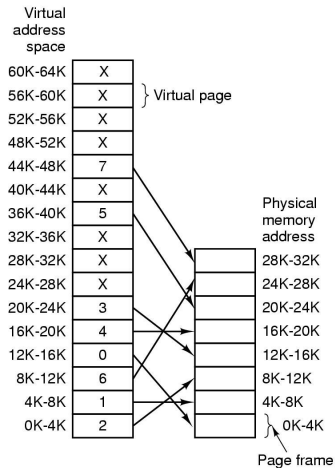
## (17) Memoria virtual (cont.)

- Sin memoria virtual:
  - Espacio de direcciones: tamaño de la memoria física.
  - Obtener una celda:
    - Pongo la dirección en el bus de memoria.
    - Obtengo el contenido.
- Con memoria virtual (a nivel conceptual): 
  - Espacio de direcciones: tamaños de la mem física + swap.
  - Los programas usan *direcciones virtuales*.
  - Obtener una celda:
    - Pongo la dirección en el bus de memoria.
    - La MMU traduce la dirección virtual a una dirección física.
    - La tabla de traducción tiene un bit que indica si el pedazo de memoria correspondiente está cargado o no.
    - Si no lo está, hay que cargarlo.
    - La dirección física se pone en el bus que llega a la placa de memoria.
    - Obtengo el contenido.
- Hay varios detalles que resolver...

## (18) En los detalles está el Diablo


- ¿Qué son los “pedazos de memoria” de los que hablé antes?
- En realidad el espacio de memoria virtual está dividido en bloques de tamaño fijo llamados *páginas* y el de memoria física en bloques del mismo tamaño llamados *marcos* o *page frames*. ⚠
- La MMU traduce páginas a frames. Es decir, interpreta las direcciones como página + offset. Los  $n$  bits más significativos son la página, el resto, el offset.
- Lo que se swappea son siempre páginas.
- Cuando una página no está en memoria la MMU emite una *page fault* que es atrapada por el SO. ⚠
- Es el SO el encargado de sacar alguna página de memoria y subir la correspondiente del disco.
- Cuál sacar es un tema no menor que afecta muchísimo al rendimiento. Lo vamos a ver en detalle más adelante. ⚠

## (19) Páginas

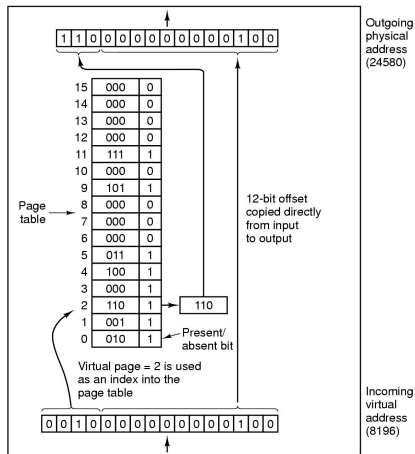


Relación entre direcciones de memoria virtuales y físicas

## (20) Manejando la MMU

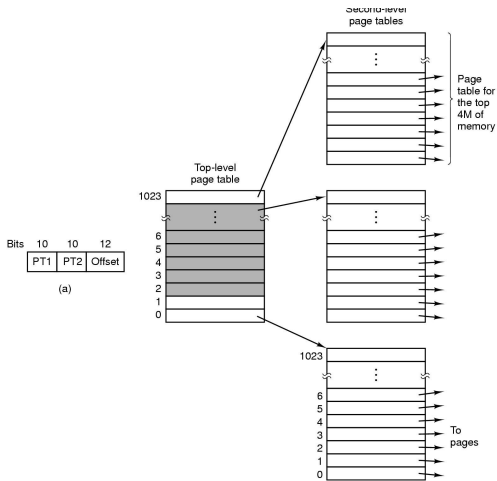
- La MMU es básicamente su *tabla de páginas*, que es lo que se usa para el mapeo.
- Queremos que la búsqueda sea rápida y que la tabla no ocupe mucho espacio.
- Si tenemos 2 GB de RAM y 2 GB de swap, con bloques de 4 KB necesitamos 1 millón de entradas en la tabla de páginas. Mucha memoria, mucho tiempo.
- Una solución es tener una tabla de páginas multinivel. 
- Los primeros bits nos llevan hacia la tabla que tenemos que consultar, y ahí usamos el resto de los bits igual que antes.
- La ventaja es que no hace falta tener toda la tabla en memoria. Se pueden swappear sus partes.

## (21) Tablas de páginas



Operación interna MMU con 16 pag de 4KB


## (22) Tabla de páginas multinivel



## (23) Las entradas de la tabla de páginas


- ¿Qué hay en cada entrada de la tabla de páginas?
  - El page frame (el objetivo de la tabla).
  - El bit de ausencia/presencia.
  - Bits de protección. Pueden ser muy complicados o tan sencillos como una marca de página de sólo lectura.
  - El bit de *dirty* que indica si la página fue modificada desde que se cargó del disco. Sólo las páginas *dirty* deben ser grabadas a disco al desalojarlas de memoria.
  - El bit de *referenciada*, que indica si una página fue accedida desde que se cargó de disco. Se usa para decidir qué página bajar a disco.
  - Puede tener otra información en arquitecturas particulares.

## (24) Memoria asociativa

- Como las tablas de páginas están en memoria (por su tamaño), el acceso a ellas, que es demasiado frecuente, puede ser muy lento.
- Estos accesos llegaron a llevar la performance al 20 % de lo que se obtenía sin paginado en el mismo HW (Motorola 68030).
- Solución: agregar un poquito de HW, un caché.
- Es pequeño, tiene poca capacidad, pero: 
  - Está implementado con registros muy rápidos.
  - Mapea directamente páginas a frames sin consultar las tablas de páginas (que pueden tener varios niveles).
  - Puede buscar en paralelo en toda su tabla.
  - Cuando una entrada no está, se busca en la tabla como siempre, pero además se ubica en el caché, apuntando a que futuros accesos sean más rápidos.
- Se lo suele llamar *memoria asociativa* o *Translation Lookaside Buffer (TLB)*.



## (25) Reemplazo de páginas

- Como dijimos, saber qué páginas dejar en memoria afecta dramáticamente al rendimiento.
- Hay varios algoritmos posibles. 
  - A nivel teórico, si registro la evolución del sistema después puedo calcular el óptimo: eliminar siempre la página que se va a volver a usar dentro de más tiempo. Obviamente no se puede hacer en la práctica, pero sirve para comparar el rendimiento de los otros algoritmos contra el óptimo. Esta técnica es muy usada para comparar algoritmos.
  - FIFO.
  - Segunda oportunidad. Hago como FIFO, pero si la página que voy a bajar tiene Referenced prendido la considero como recién subida y paso a la siguiente. Es bastante fácil de implementar y mucho mejor que FIFO.
  - *Not Recently Used*. La idea es establecer una prioridad para desalojar una página: las que no fueron ni referenciadas ni modificadas son las más convenientes. Le siguen las que fueron referenciadas pero no modificadas, que son “baratas” de bajar, y por último las modificadas.


## (26) Reemplazo de páginas (cont.)

- Más algoritmos:
  - *Least Recently Used*. Es el que suele funcionar mejor en la práctica, aunque es caro de implementar. La idea es que la que se usó menos recientemente tiene menor probabilidad de volver a ser usada en lo inmediato. Implementación: tenemos un contador global de instrucciones ejecutadas. Cuando una página se accede se la marca con el valor del contador. A la hora de desalojar, se busca la de menor marca.
- De todos estos algoritmos hay variaciones que apuntan a lograr un balance entre el costo de implementación y la calidad de los resultados.
- Otro punto a considerar: ¿desalojo las páginas del mismo proceso o de otro cualquiera?

## (27) Reemplazo de páginas (cont.)

- Estas estrategias a veces se combinan con carga de páginas por adelantado, en lugar de que cada página deba ocasionar un page fault.
- Es bueno considerar que los programas muchas veces exhiben *localidad de referencia*: es decir, acceden a direcciones que están más bien juntitas.
- Además, podemos encontrar páginas especiales:
  - Páginas de sólo lectura.
  - Páginas no swappable (seguridad).

## (28) Page faults


- Veamos en detalle qué pasa en un page fault. 
- Se emite el page fault, que es una interrupción. Lo atrapa el kernel.
- Se guardan el IP y eventualmente otros registros en la pila.
- El kernel determina que la interrupción es de tipo page fault, y llama a la rutina específica.
- Hay que averiguar qué dirección virtual se estaba buscando. Usualmente queda en algún registro.
- Se chequea que sea una dirección válida y que el proceso que la pide tenga permisos para accederla. Si no es así, se mata al proceso (en Unix se envía una señal de *segmentation violation*, lo que lo hace terminar).
- Se selecciona un page frame libre si lo hubiese y si no se libera mediante el algoritmo de reemplazo de páginas.

## (29) Page faults (cont.)


- Si la página tenía el bit *dirty* prendido, hay que bajarla a disco. Es decir, el “proceso” del kernel que maneja E/S debe ser suspendido, generándose un cambio de contexto y permitiendo que otros ejecuten. La página se marca como *busy* para evitar que se use.
- Cuando el SO es notificado de que se terminó de bajar la página a disco comienza otra operación de E/S, esta vez para cargar la página que hay que subir. De nuevo se deja ejecutar a otros procesos.
- Cuando llega la interrupción que indica que la E/S para subir la página terminó, hay que actualizar la tabla de páginas para indicar que está cargada.
- La instrucción que causó el page fault se recomienza, tomando el IP que había quedado en el stack y los valores anteriores de los registros.

- Se devuelve el control al proceso de usuario. En algún momento el scheduler lo va a hacer correr de nuevo, y cuando reintente la instrucción la página ya va a estar en memoria.

## (31) Thrashing

- Cuando no alcanza la memoria y hay mucha competencia entre los procesos por usarla...
- ...y el SO se la pasa cambiando páginas de memoria a disco ida y vuelta...
- ...se dice que está haciendo *thrashing*. 
- Es una situación altamente indeseable, porque la mayor parte del tiempo se pasa haciendo “mantenimiento” en lugar de trabajo productivo.


## (32) Protección y reubicación

- Todavía nos queda el problema de la protección y el de la reubicación.
- Para la protección hay una solución fácil: cada proceso tiene su propia tabla de páginas. No hay forma de acceder a una página de otro.
- Una manera de solucionarlo es hacer que cada proceso tenga su propio espacio de memoria.
- Cada uno estos espacios se llama *segmentos*. 
- Se usa un registro especial para saber a qué segmento hacen referencia las direcciones.
- Esto no sólo facilita la protección. Además permite que cada segmento crezca sin tener que cambiar el programa.
- Además se facilita el trabajo con bibliotecas compartidas (cada una en su segmento).




- “La librería no me linkea”
- Biblioteca compartida.
- =
- Shared library.
- $\neq$
- Librería compartida.
- De nuevo, no sean brutos.

## (34) Segmentación

- El programador (assembler) sabe que existen los segmentos. Tiene que decidir a cuál apuntar. Las páginas son invisibles.
- Los segmentos, a diferencia de las páginas, no son de tamaño fijo. 
- Sin embargo, tenemos los mismos problemas de fragmentación y swapping que mencionamos antes.
- Por eso, la alternativa más común es combinar segmentación con paginado.


## (35) Segmentación

- Veamos cómo se hace en Intel. (Pentium) 
  - Cada proceso tiene su *Local Descriptor Table (LDT)*.
  - Los programas suelen tener un segmento para código, otro para datos y otro para el stack.
  - Hay una *Global Descriptor Table (GDT)* compartida. Figuran allí los segmentos del sistema.
  - Hay dos registros de 16 bits, DS y CS, que sirven para indicar qué segmento se va a usar. De esos 16 bits 13 son un índice, 1 indica si se trata de un segmento global o local y quedan 2 bits para protección.
  - Cuando se carga algo en uno de ellos, se trae la correspondiente entrada de la LDT o la GDT.
  - Cada entrada tiene la base de las direcciones de memoria y el tamaño del segmento.
  - La dirección que se obtiene así se interpreta como física si no se usa paginado o como virtual si está habilitado el paginado.
  - ¿Y cómo funciona la protección?
  - Si tocamos DS o CS se produce un trap, que lo recibe el SO y nos corta los dedos.

## (36) Segmentación vs. Paginación

- Las páginas son invisibles para el programador (assembler), los segmentos no.
- Los segmentos proveen varios espacios de direccionamiento, que pueden estar solapados.
- Los segmentos facilitan la protección, aunque también se puede implementar mediante paginado.
- La segmentación brinda espacios de memoria separados al mismo proceso.

## (37) Copy-on-write

- Vimos que es muy común el uso de `fork()` para crear nuevos procesos.
- ¿Copiamos sus páginas de memoria?
- La estrategia más común consiste en hacer *copy-on-write*: 
  - Al crear un nuevo proceso se utilizan las mismas páginas.
  - Hasta que alguno de los dos escribe en una de ellas.
  - Ahí se duplican y cada uno queda con su copia independiente.

## (38) Administración de la memoria: API

- Sistema operativo
  - Memoria del kernel
  - Memoria de los procesos
- Lenguaje de programación
  - Explícita
    - Bibliotecas: e.g., `malloc()` y `free()` en `libc`.
    - Primitivas: e.g., `new` y `delete` en `C++`.
  - Implícita
    - No controlada por el programa: e.g., `Haskell`.
    - Parcialmente controlada/guiada: e.g., `Swift`.
  - Híbrida
    - Creación explícita y liberación implícita: e.g., `Java`.
    - Bibliotecas y primitivas: e.g., `C++`.
    - Bibliotecas y análisis a la compilación: e.g., `RTSJ`.
- Integrada: e.g., `Android`, `iOS`.

## (39) Administración de la memoria en POSIX (o casi)

- Heap

- No portable: incrementar/decrementar el *program break*

```
#include <unistd.h>
int brk(void *addr);
void *sbrk(intptr_t increment);
```

- Portable

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

## (40) Administración de la memoria en POSIX (o casi)

- Stack

- Típicamente usada por el compilador (e.g., gcc)

```
#include <alloca.h>
void *alloca(size_t size);
```

- File mapping

- Mapear un archivo directamente a memoria.
  - Usos: segmento de programa, bibliotecas dinámicas.
  - Anónimo o no, privado o compartido:

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```



## (41) Administración de la memoria: en la práctica

- Se usan esquemas que conocen un poco más sobre la distribución de los pedidos y realizan manejos más sofisticados.
- Integración/cooperación lenguaje/compilador/SO.
- Cómo asignar memoria de manera rápida y eficiente sigue siendo un problema interesante.
- Ejemplo: Doug Lea allocator: <http://goo.gl/JQqFCD>
  - Política de asignación:
    - Best-fit con Most-Recently-Used en caso de empate.
    - Para pedidos de menos de 256 bytes no usa best-fit.
  - Concurrencia: No *thread-safe* por default.
  - Seguridad:
    - Garantiza inmutabilidad de direcciones menores que la base del heap.
    - *Check word*: evita liberaciones no hechas por malloc.

## (42) Dónde estamos

- Vimos
  - Cómo compartir la memoria:
  - Swapping, paginación, segmentación.
  - Cómo manejar el espacio libre.
  - Cómo decidir qué páginas sacar de memoria.
  - MMU
  - Page faults.
  - Thrashing.
  - Copy-on-write.
- La próxima clase:
  - Analizamos en detalle los mecanismos de E/S.

## (43) Administración de la memoria: bibliografía adicional

- P.J. Denning. Thrashing: its causes and prevention. AFIPS 1968.  
<https://goo.gl/VCyZcK>
- J.M. Robson. Worst Case Fragmentation of First Fit and Best Fit Storage Allocation Strategies. Computer J. 20, 1977. <http://goo.gl/1hfGxG>
- P. Wilson, M. Johnstone, M. Neely, D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. <http://goo.gl/Uf6T17>
- S. Craciunas, C. Kirsch, H. Payer, A. Sokolova, H. Stadler, R. Staudinger: A Compacting R-T Memory Mgmt. System. <http://goo.gl/z9SCwt>
- M. K. McKusick, J. Karels. Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel. <http://goo.gl/RC8Vtp>
- J. Bonwick. The slab allocator: An object-caching kernel memory allocator. USENIX Summer, 87-98, 1994. <http://goo.gl/V4Vgur>
- Valgrind: an instrumentation framework for building dynamic analysis tools. <http://valgrind.org/>
- F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. <http://goo.gl/Bi8QHB>
- D Garbervetsky, C Nakhli, S Yovine, H Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. <http://goo.gl/t5g0RP>