ARCOS Group
Computer Science and Engineering Department
Universidad Carlos III de Madrid

Lesson 5 (b)
Memory Management

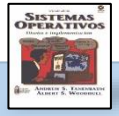
Operating System Design
Degree in Computer Science and Engineering, Double Degree CS&E + BA

Recommended readings

Base


1. Carretero 2007:
   1. Chapter 4

Recommended



1. Tanenbaum 2006(en):
   1. Chapter 4
2. Stallings 2005:
   1. Part three
3. Silberschatz 2006:
   1. Chapter 4

Remember…

1. To study the associated theory.
   - Better study the bibliography readings because the slides are not enough.
   - To add questions with answers, and proper justification.
1. To review what in class is introduced.
   - To do the practical Linux task step-by-step.
2. To practice the knowledge and capacities.
   - To do the practical tasks as soon as possible.
   - To do as much exercises as possible.

Overview

1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

1) Management policies and management guidelines

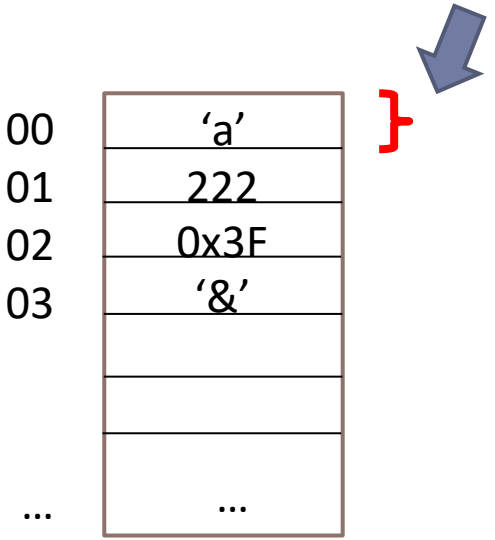
Overview

1) Introduction
   a) **Memory allocator**
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

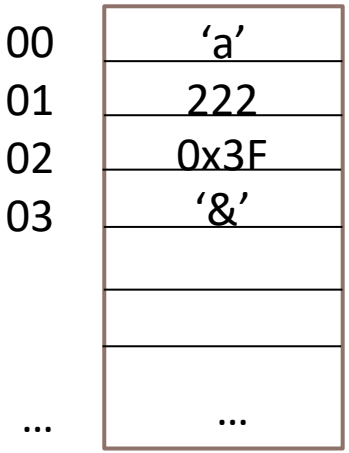
1) Management policies and management guidelines

Basic usage of memory
address, value, and size



- Value
  - Element stored in memory.
- Address
  - Place in memory.
- Size
  - Number of bytes needed to stored the value.

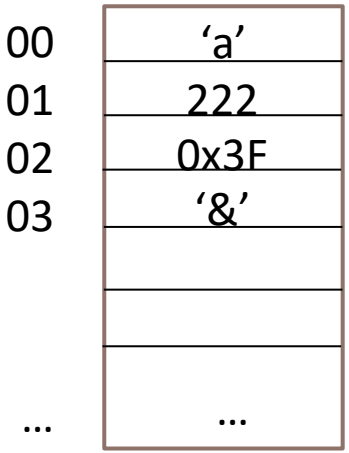
Basic usage of memory
functional interface



- Value = **read** (Address)
- **write** (Address, Value)

(Tip) Before to access into a address, it must point to a memory area previously allocated.

Basic usage of memory
functional interface



- **Value = read (Address)**
- **write (Address, Value)**

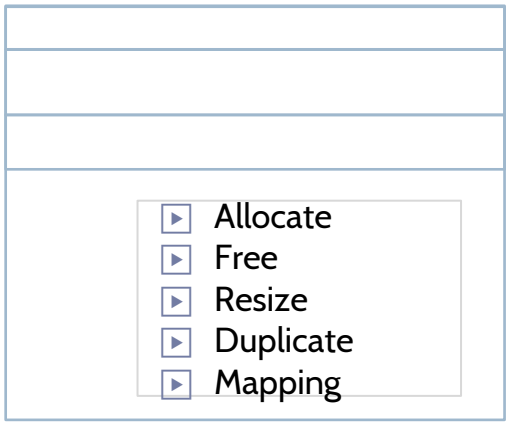
(Tip) Before to access into a address, it must point to a memory area previously allocated.
=> Memory allocator

Memory management: memory allocator

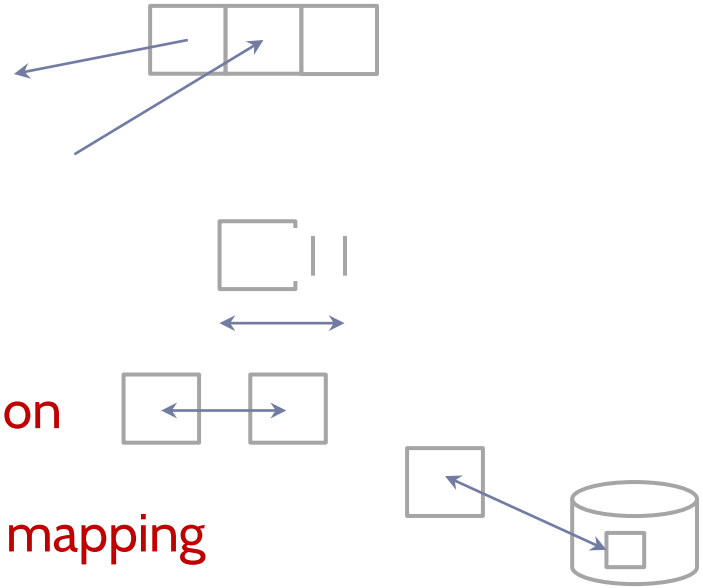
memory allocator = Block

Memory management: memory allocator

memory allocator = Block + Interface

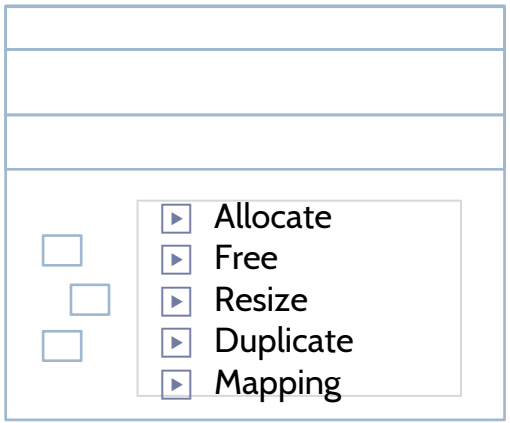


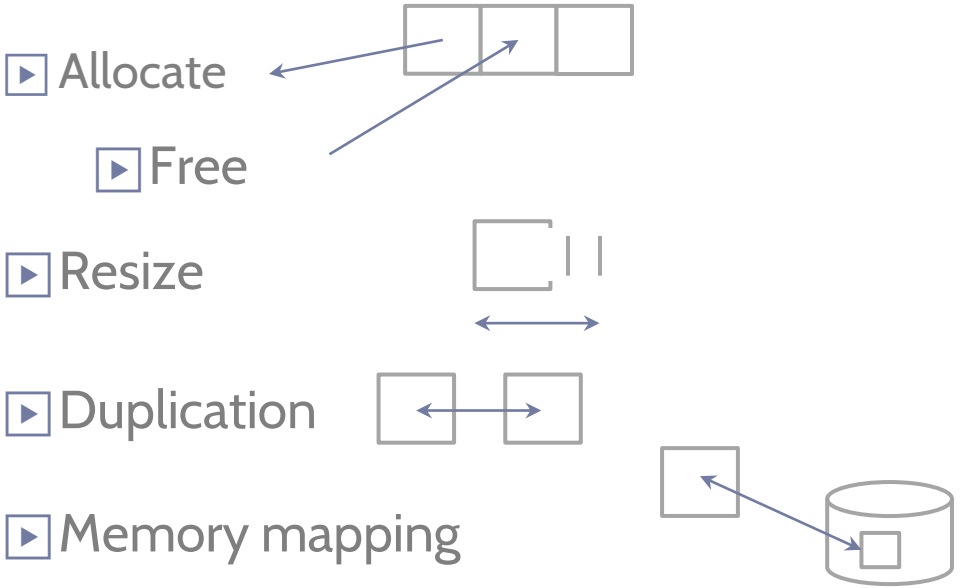
- Allocate
- Free
- Resize
- Duplication
- Memory mapping



Memory management: memory allocator

memory allocator = Block + Interface + Metadata





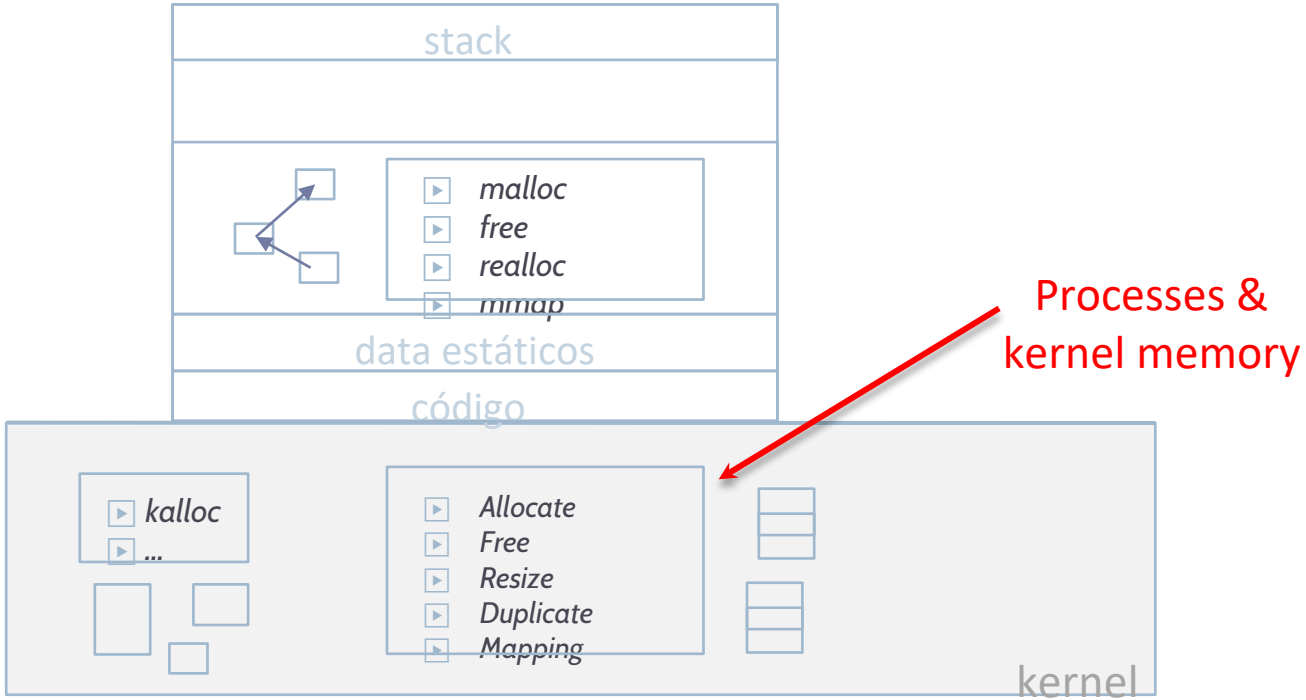
Overview

1) Introduction
   a) Memory allocator
   b) **Memory allocator hierarchy**

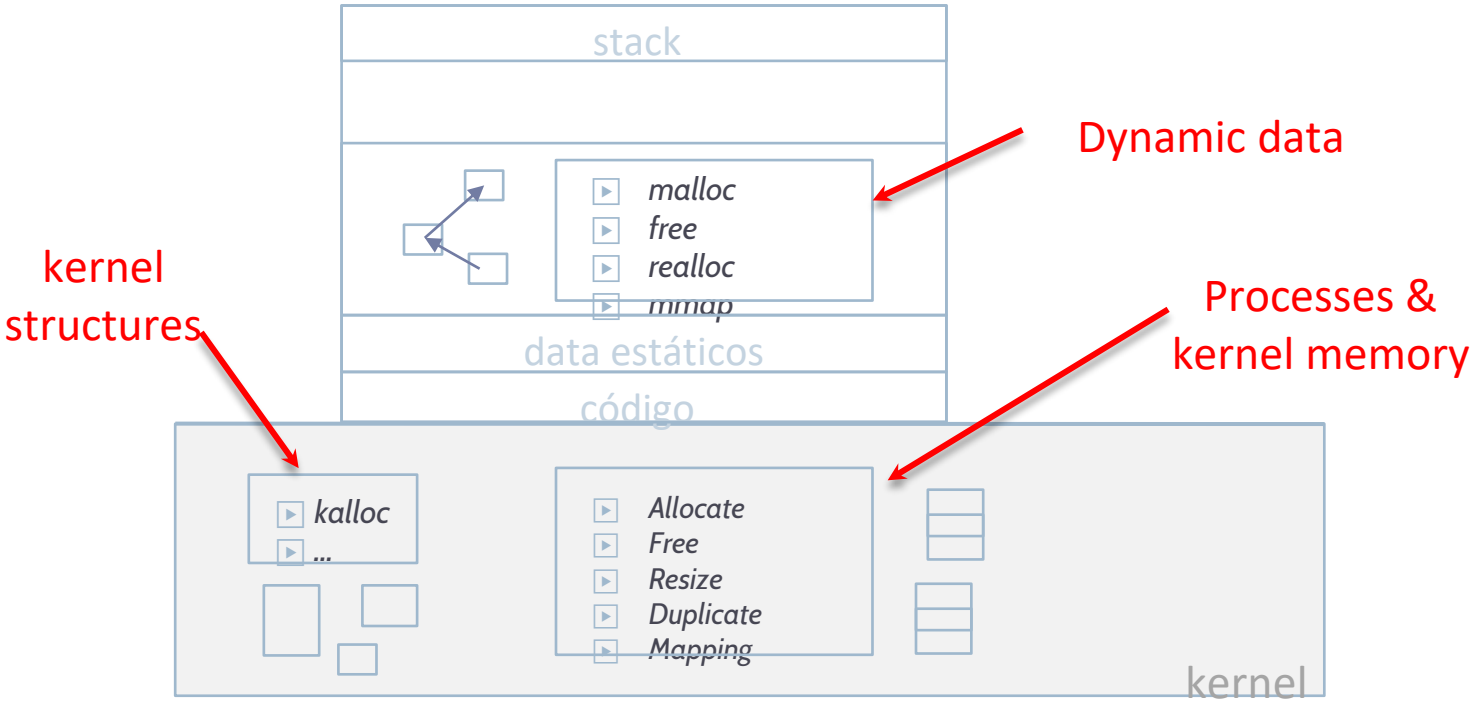
1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

1) Management policies and management guidelines

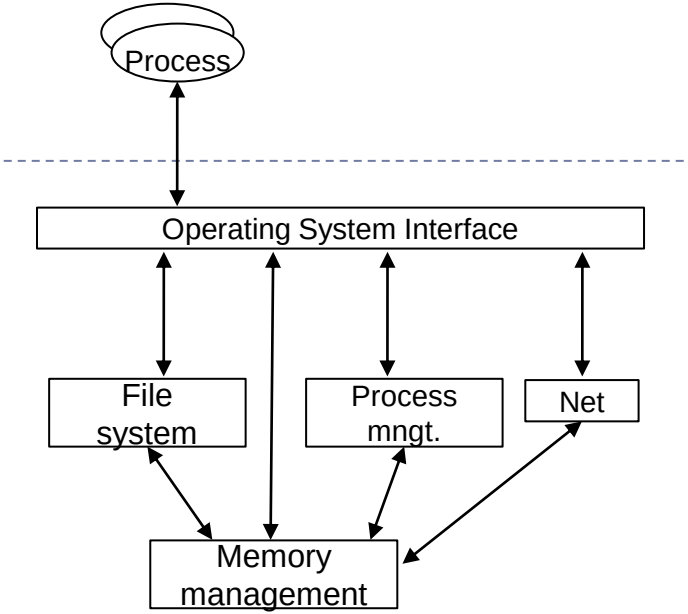
Memory managers at several levels: Level 1



Memory managers at several levels: Level 2



Memory management Scope: Level 1



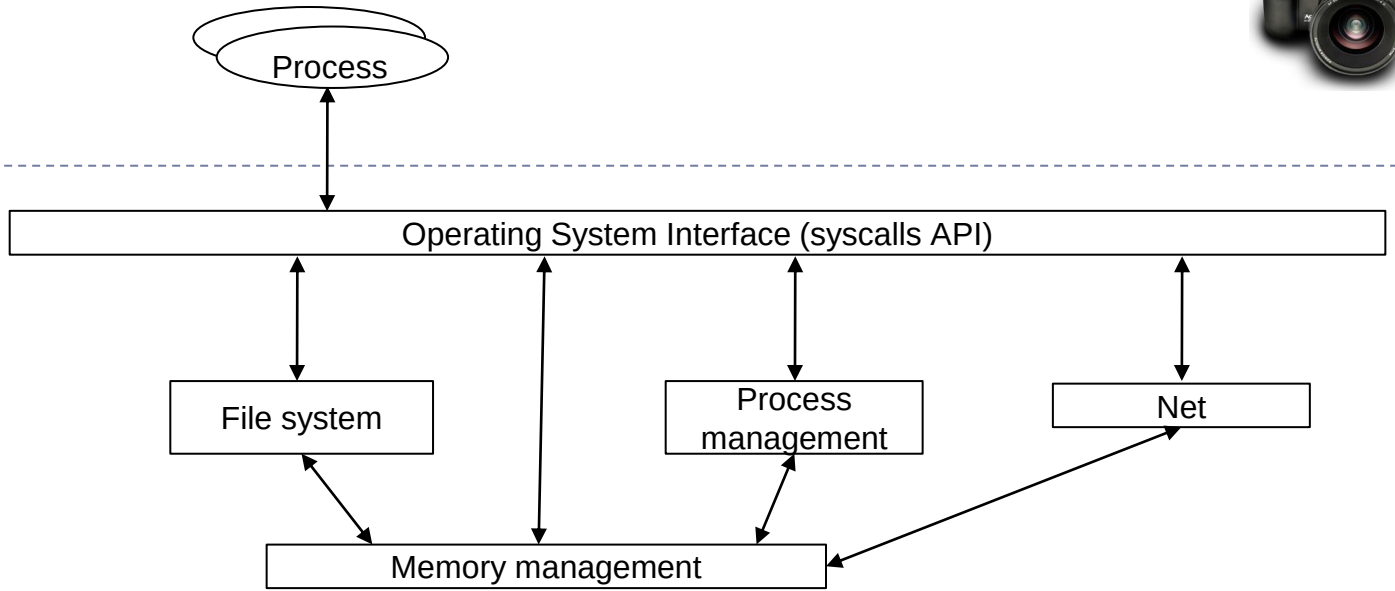
- In charge of memory management for processes and kernel.
- The rest of the operating system is its best client:
  - Process management
  - File system (File management)
- But it reflects the needs of the processes

Memory management architecture

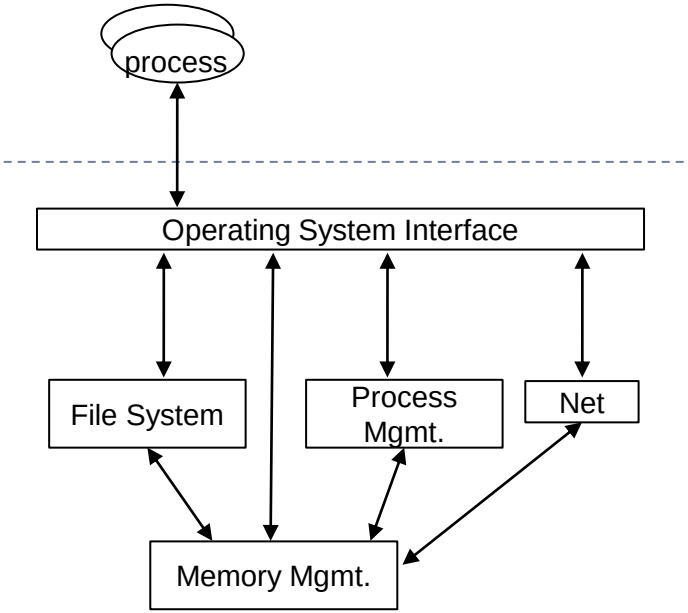
old architecture

Memory management architecture

new architecture



General goals of the memory mgmt.



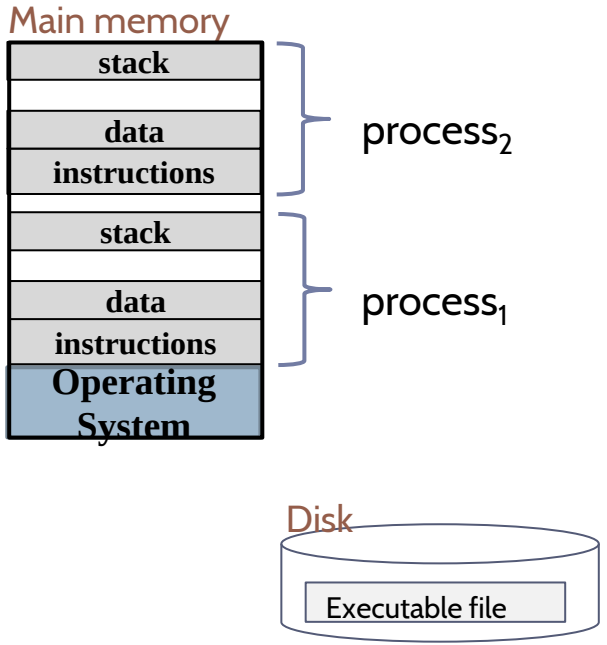
1. Locating references of memory
   - Translate memory references into physical addresses
2. Protection of memory spaces
   - Forbid references between different processes
3. Sharing memory spaces
   - Allow multiple processes to access a common memory space area
4. Logic organization (of the process)
   - Process are divided into memory segments
5. Physical organization (of memory)
   - Fill memory with multiple process and segments

http://doursat.free.fr/docs/CS446_F05/CS446_F05_3_Memory1.pdf

General goals of the memory mgmt.

1.- Locating references of memory

- The programmer does not have to know where the program will be placed in memory while it is executed.
  - A program can be executed several times, and each time it might be placed into a different memory position
- As long as the program is executed it can also be sent to disk and returned to memory in a different position.
- Therefore, logical memory references (relatives) must be translated into physical memory addresses (absolutes).

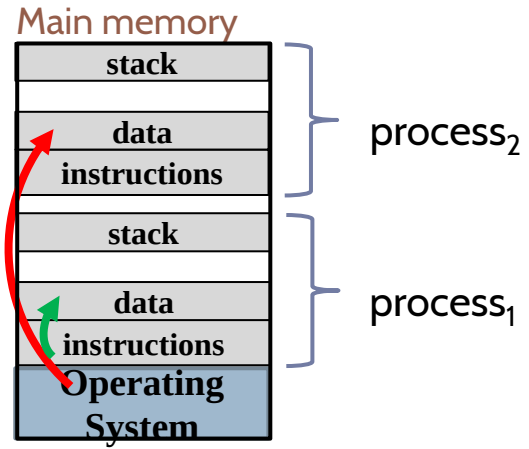


http://doursat.free.fr/docs/CS446_F05/CS446_F05_3_Memory1.pdf

General goals of the memory mgmt.

2.- Protection of memory spaces

- Processes must not use memory locations of other processes
  - Exception: debugger, ...
- Memory locations would have to be checked at runtime
  - It is not possible to check accesses to physical memory at compilation time.
- Memory locations must be checked by hardware
  - The operating system cannot anticipate the (calculated) memory references that a process is going to perform.

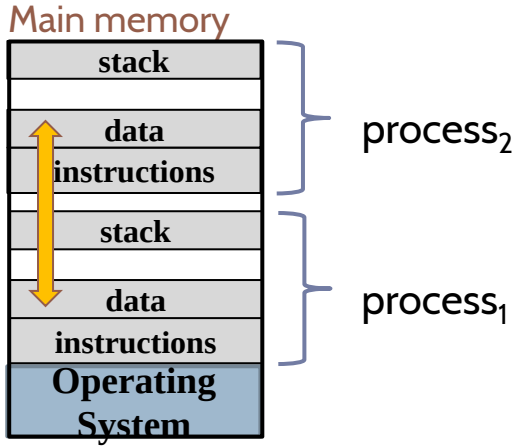


http://doursat.free.fr/docs/CS446_F05/CS446_F05_3_Memory1.pdf

General goals of the memory mgmt.

3.- Sharing memory spaces

- Opposite to the previous point (apparently). It must be possible for several processes to access the same portion of memory:
  - Processes running the same code could share the same copy of code in memory
  - Processes cooperating on the same task may need access to the same data structures.
- Must be explicitly requested and granted
  - Debugger, etc.

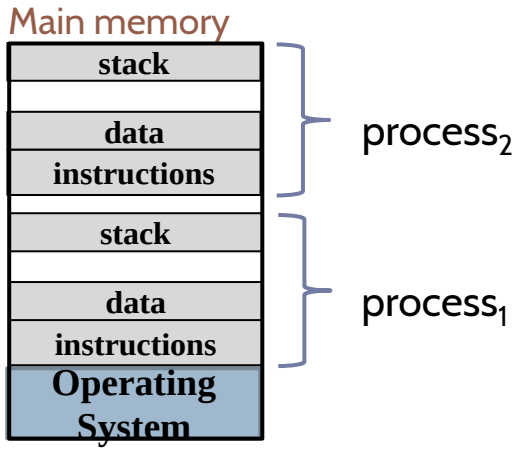


http://doursat.free.fr/docs/CS446_F05/CS446_F05_3_Memory1.pdf

General goals of the memory mgmt.

4.- Logic organization (of the process)

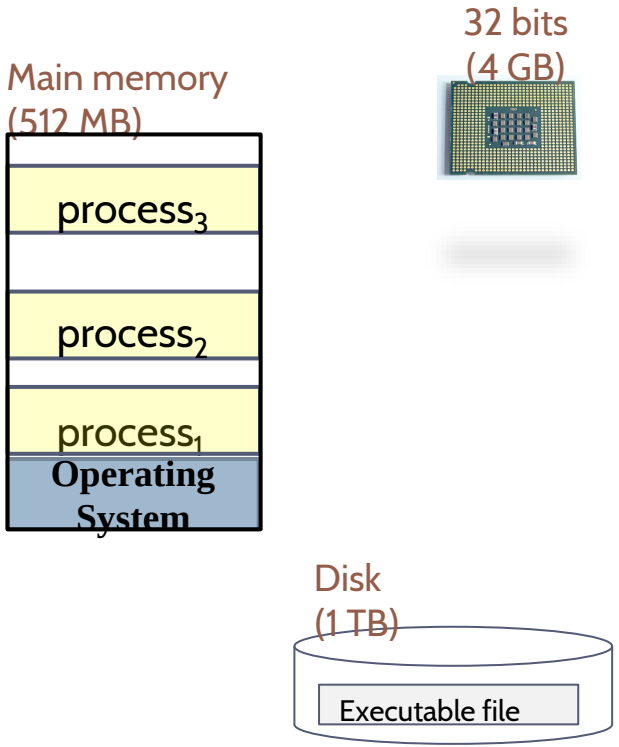
- The data of a process are not homogeneous
  - E.g.: code, local variables, etc.
  - Each information type has different needs
    - Read, write, execution, etc.
    - Static or dynamic creation
- The information of a process (its image) is divided into different regions
  - Each region is adapted to a specific type of dat (code, dynamic variables, etc.)
    - Unallocated zones (gaps) must be managed
- To manage the memory of a process is to manage each of its regions.



General goals of the memory mgmt.

5.- Physical organization (of memory)

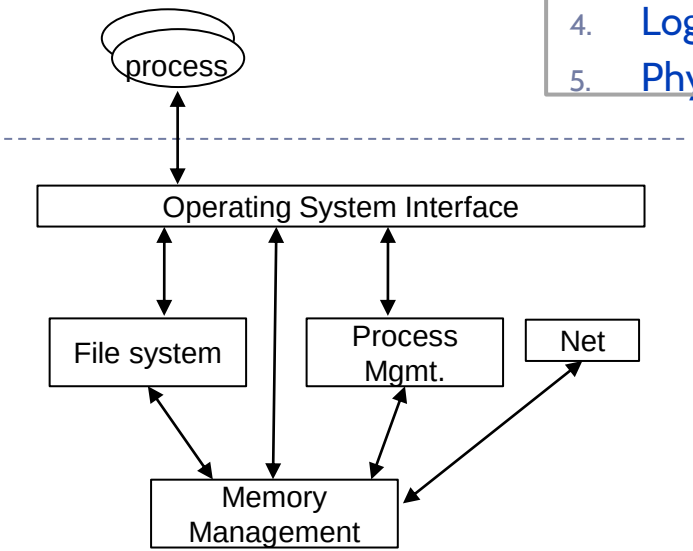
- Be able to run a process when your memory image is larger than the main memory:
  - The parts of the process that are not used at the moment are saved in disk.
- To be able to execute a set of processes whose total memory size is greater than the main memory size.
- Avoid losing memory by fragmentation:
  - There is free physical memory but it is fragmented in non-contiguous spaces that the memory management system cannot take advantage of.



Scope, architecture, and goals

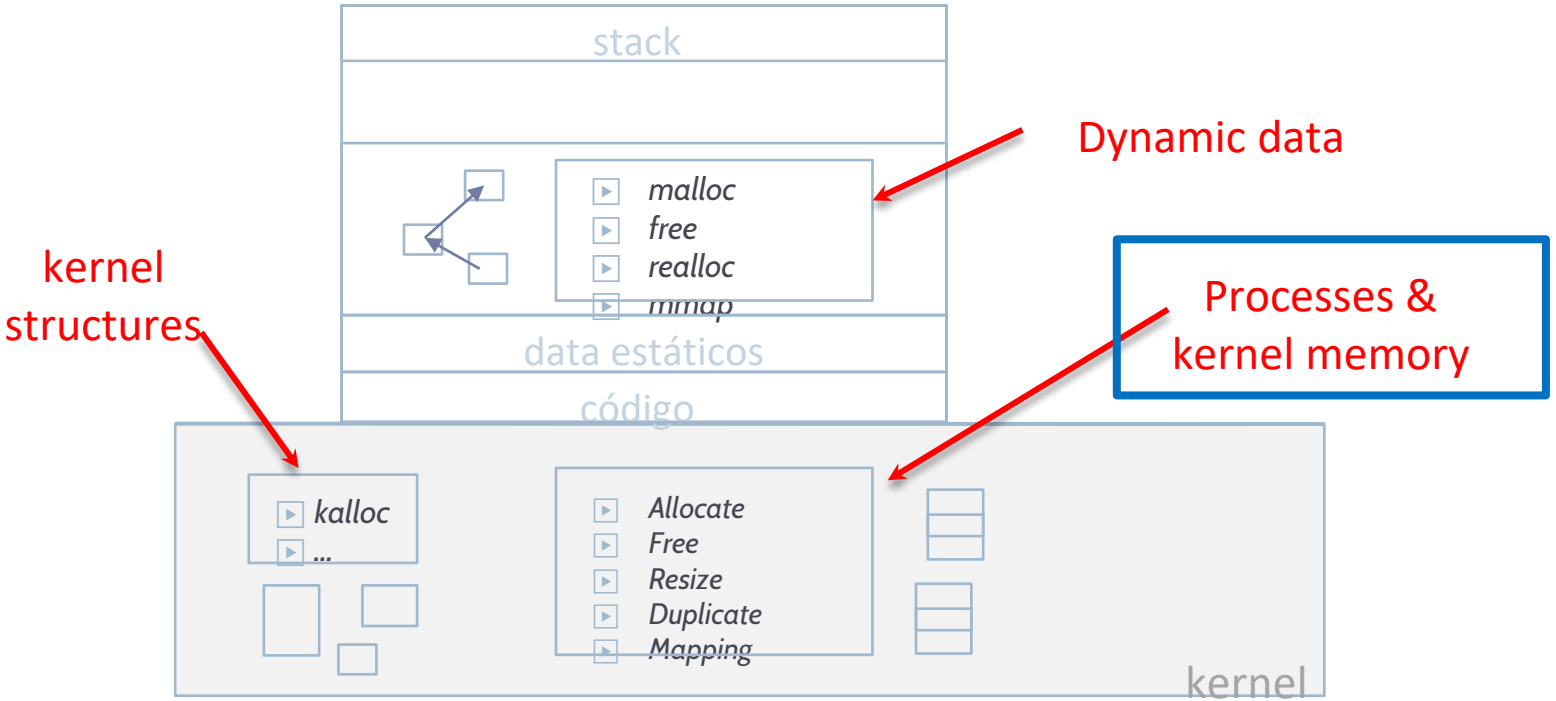
summary

1. Locating references of memory
2. Protection of memory spaces
3. Sharing memory spaces
4. Logic organization (of the process)
5. Physical organization (of memory)



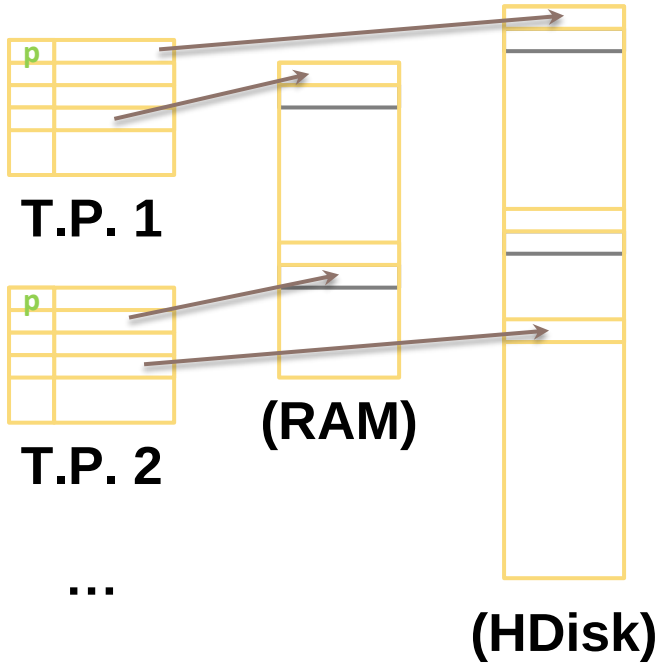
Sistemas operativos: una visión aplicada
http://doursat.free.fr/docs/CS446_F05/CS446_F05_3_Memory1.pdf

Memory managers at several levels: Level 1

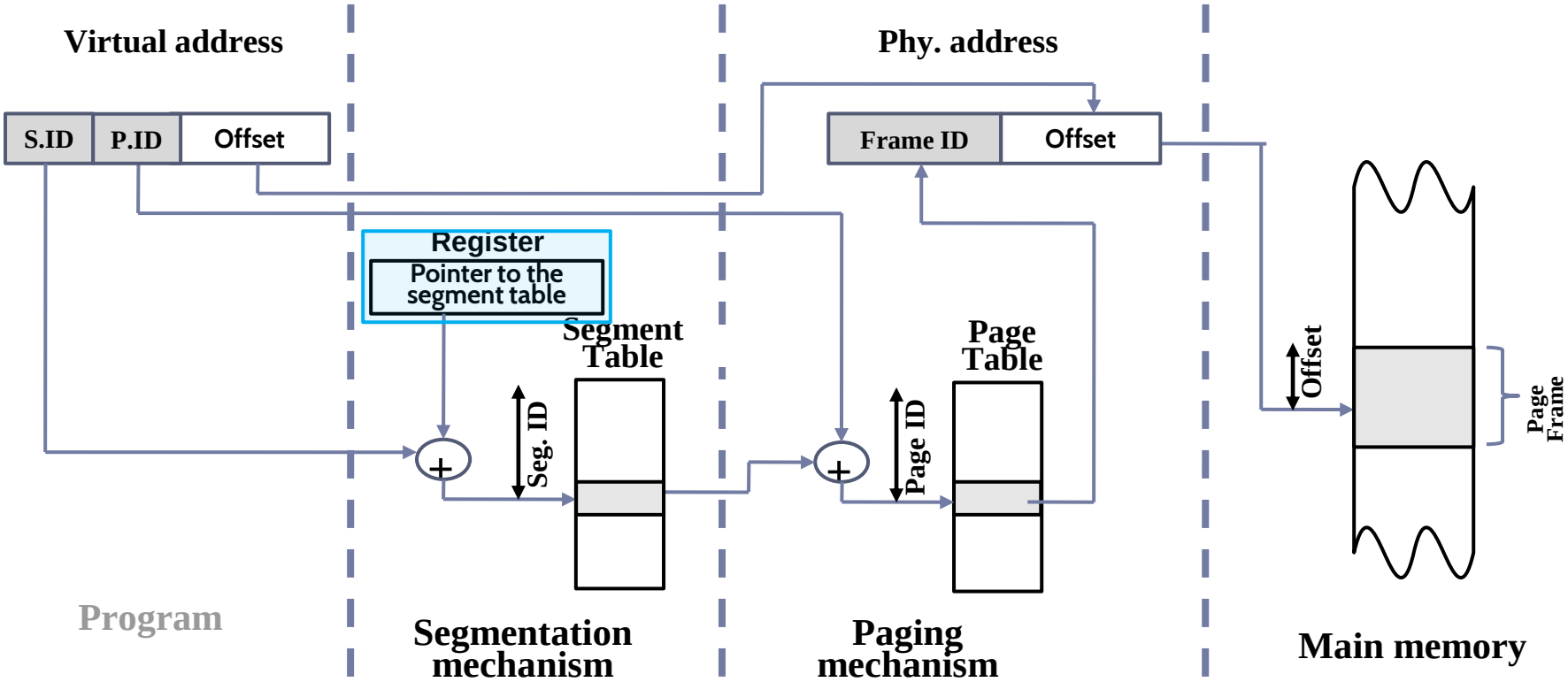


Working with different memory spaces

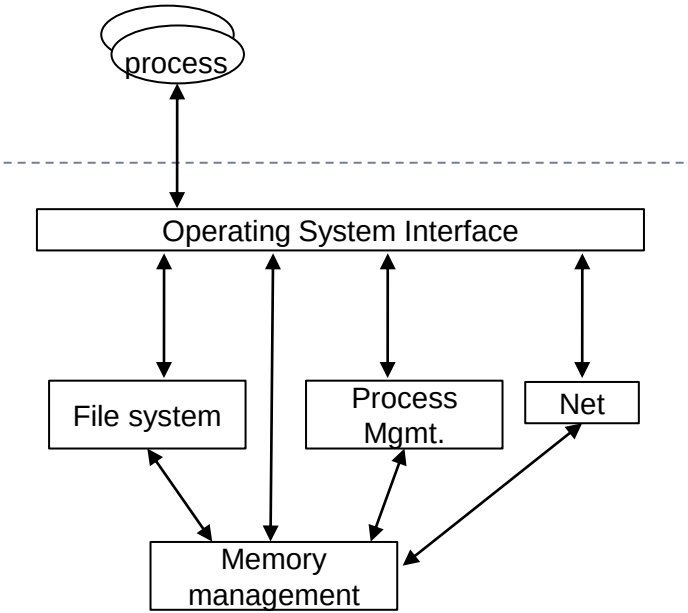
- Segment table per process
- One register points to the table of the current process



Working with different memory spaces
paging segmentation



General goals of the memory mgmt.
using virtual memory



1. Locating references of memory
   - MMU is responsible for translating a virtual space into the real one
2. Protection of memory spaces
   - Each process access to its own memory space
3. Sharing memory spaces
   - Page table entries that point to the same page frames
4. Logic organization (of the process)
   - Use of segmentation
5. Physical organization (of memory)
   - Paged segmentation offer greater flexibility in the memory organization

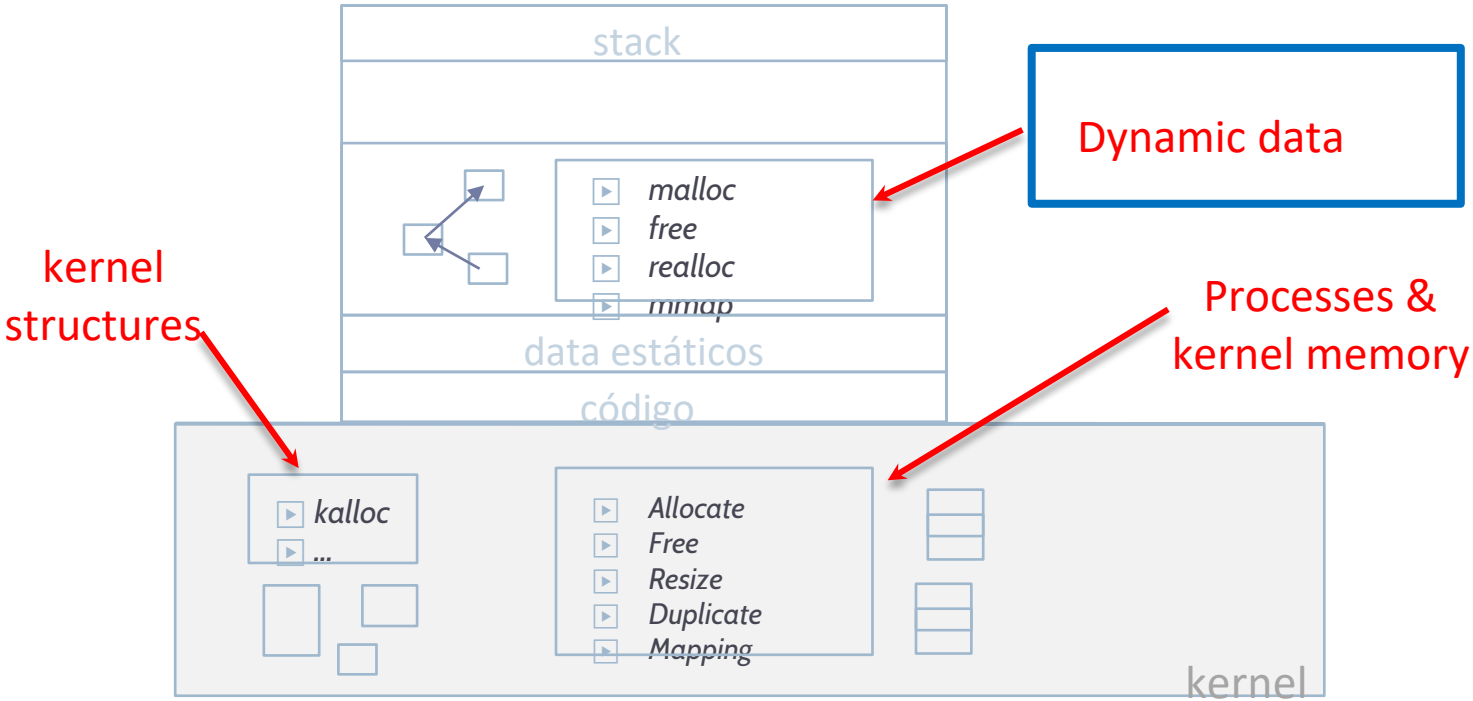
Overview

1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) **Dynamic memory allocator in user space**
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

1) Management policies and management guidelines

Memory managers at several levels: Level 1



Dynamic memory management

- Why dynamic memory is really so ‘fragile’?

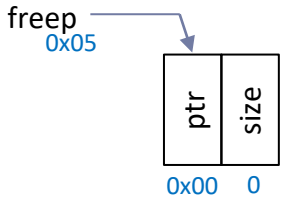
```
acaldero@phoenix:~/infodso/$ ./ptr
Violación de segmento

acaldero@phoenix:~/infodso/$ gdb ptr
GNU gdb (GDB) 7.2-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Para las instructions de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>...
Leyendo sÃmbolos desde /home/acaldero/work/infodso/memoria/ptr...hecho.
(gdb) run
Starting program: /home/acaldero/work/infodso/memoria/ptr

Program received signal SIGSEGV, Segmentation fault.
0xb7f79221 in ?? () from /lib/libc.so.6
```

Example: libc storage allocator

Header

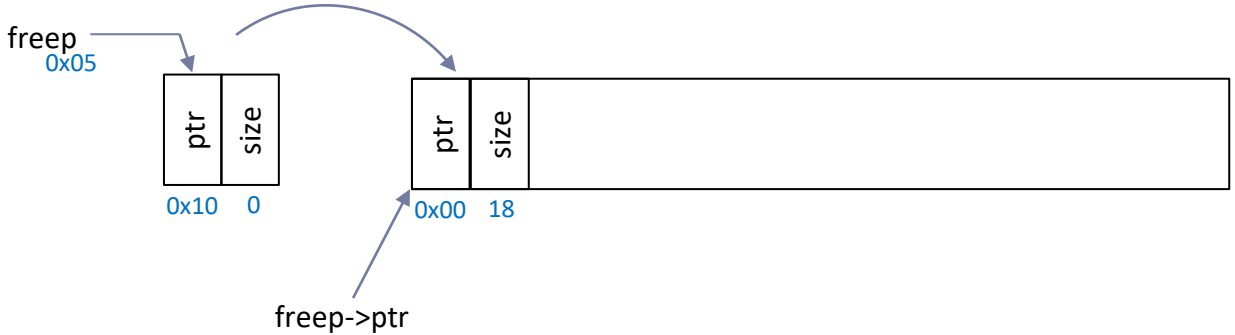


- static Header base:
  - First element in the list
  - With zero size (in headers)

```
typedef long Align;    /* for alignment to long boundary */
union header {         /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;           /* force alignment of blocks */
};
typedef union header Header;
```

Example: libc storage allocator

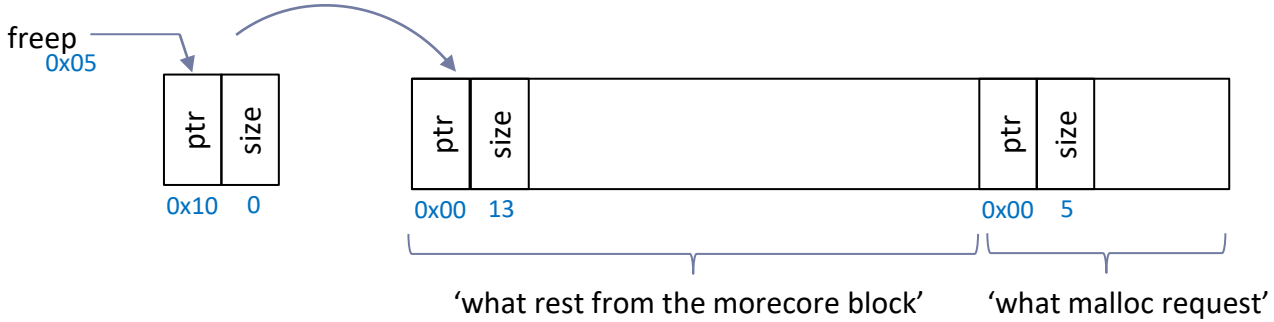
morecore



- morecore (int n_cab)
  - SI (n_cab < min_ncab)
    n_cab = min_ncab; // 144 bytes = 18 headers
  - freep->ptr = sbrk(n_cab*2*sizeof(int))
  - freep->ptr->ptr=null;
  - freep->ptr->size=n_cab;

Example: libc storage allocator

malloc

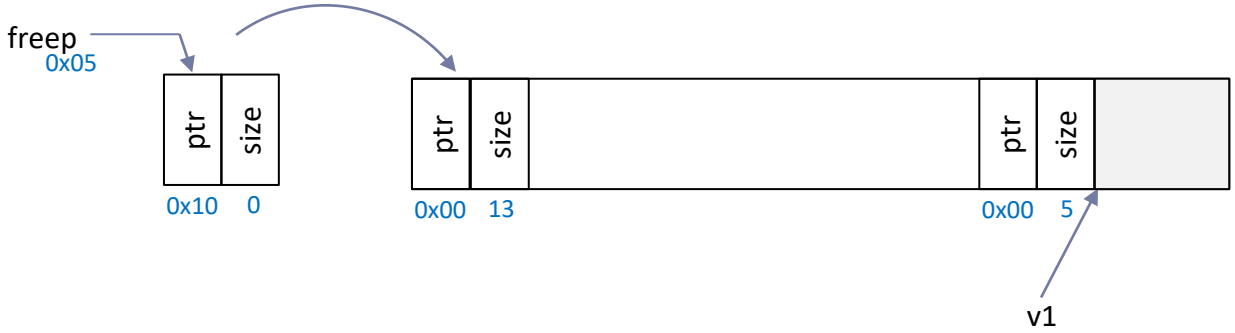


- int *v1;
- char *v2 ;

- v1 = malloc(8*sizeof(int)) ;

Example: libc storage allocator

malloc

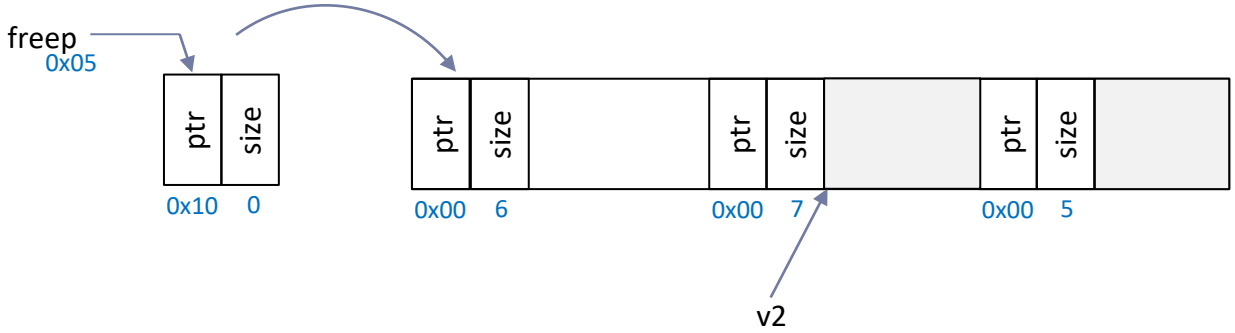


- int *v1;
- char *v2 ;

- v1 = malloc(8*sizeof(int)) ;

Example: libc storage allocator

malloc

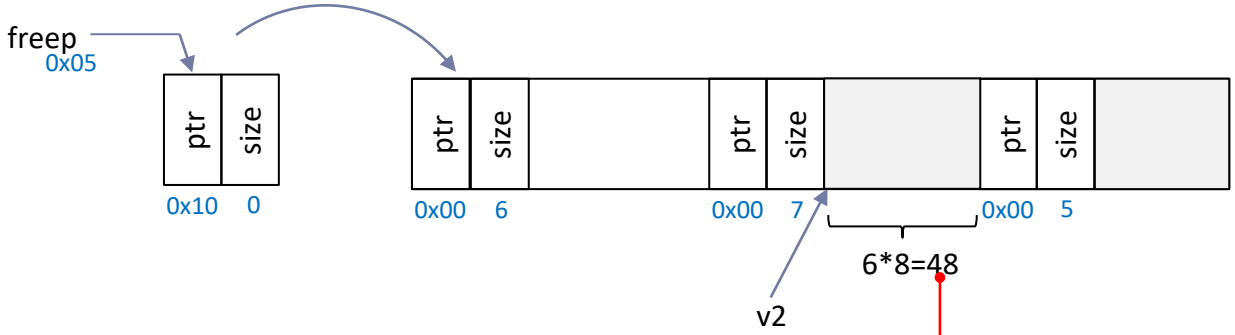


- int *v1;
- char *v2 ;

- v1 = malloc(8*sizeof(int)) ;
- v2 = malloc(41) ;

Example: libc storage allocator

internal fragmentation problem



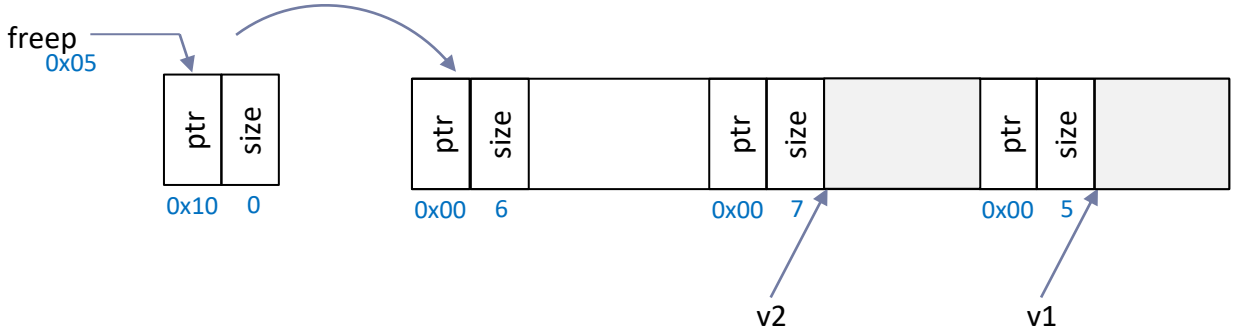
- int *v1;
- char *v2 ;

- v1 = malloc(8*sizeof(int)) ;
- v2 = malloc(41) ;

- Allocation unit is 8 bytes (1 header of 2 integer/word)
- It is rounded up to multiple of a allocation unit

Example: libc storage allocator

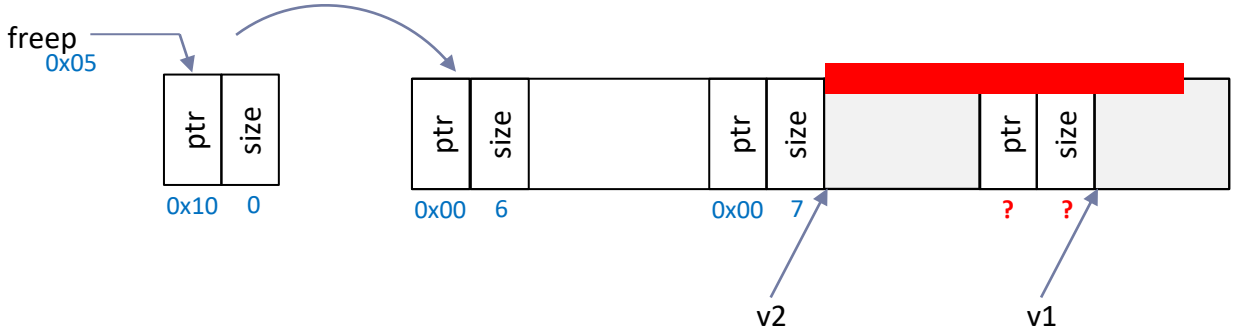
overwrite problem



- // it is allocated only 41 characters for v2
- for (int i=0; i<64; i++)
  - v2[i] = 'x' ;
- free(v1) ;

Example: libc storage allocator

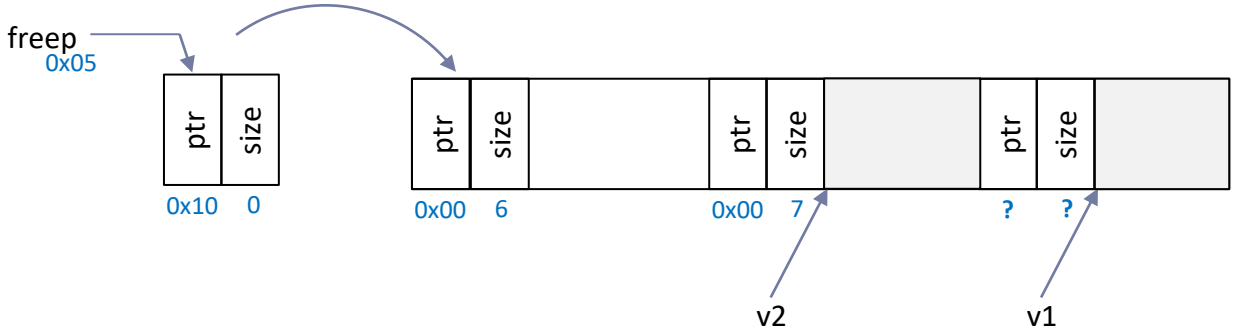
overwrite problem



- // it is allocated only 41 characters for v2
- for (int i=0; i<64; i++)
  - v2[i] = 'x' ;
- free(v1) ;

Example: libc storage allocator

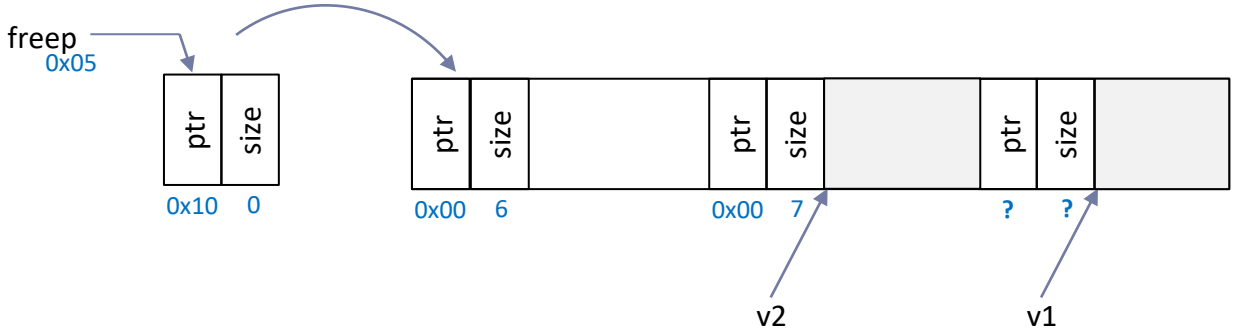
overwrite problem



- // it is allocated only 41 characters for v2
- for (int i=0; i<64; i++)
  - v2[i] = 'x' ;
- free(v1) ; <- unable to recover the valid header... SIGSEV

Example: libc storage allocator

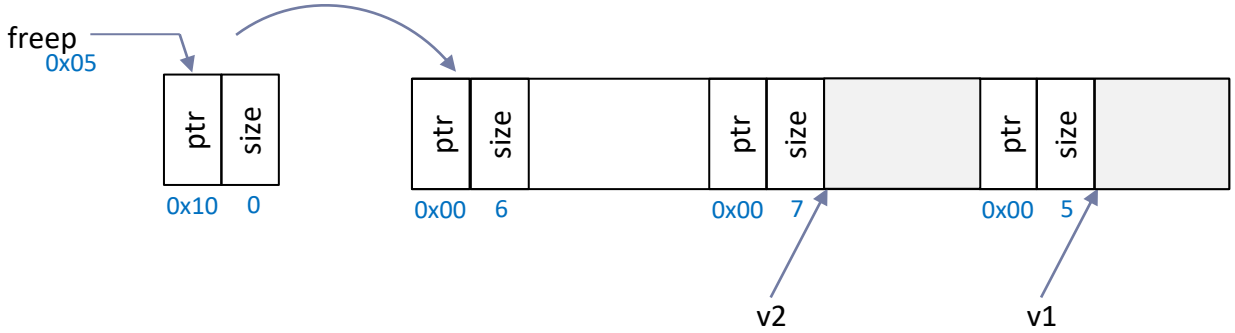
other typical problems



- Free a non-dynamic memory area:
  - int i; free(&i);
- Free two times the same memory area
- Access to a memory area that still was not requested
  - char *pchar; printf("%s",pchar);

Example: libc storage allocator

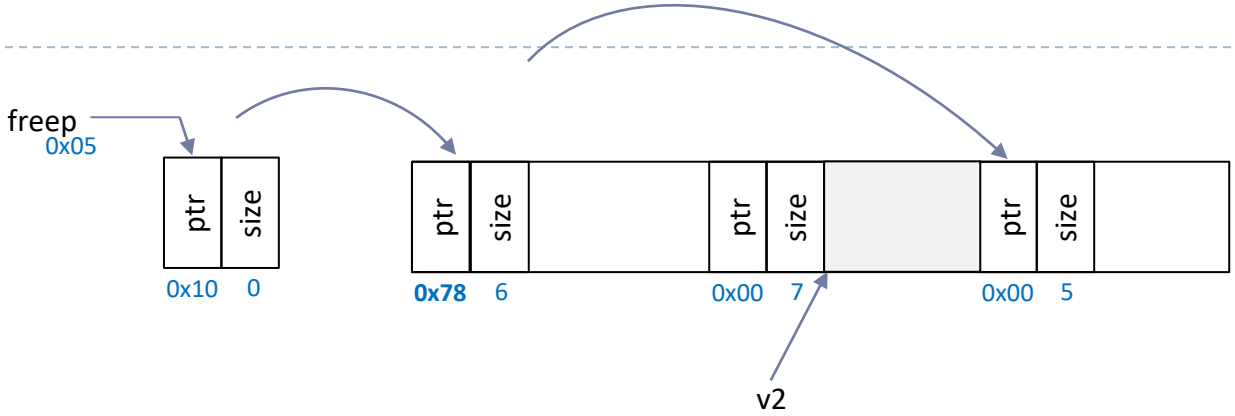
free



- free(v1) ;

Example: libc storage allocator

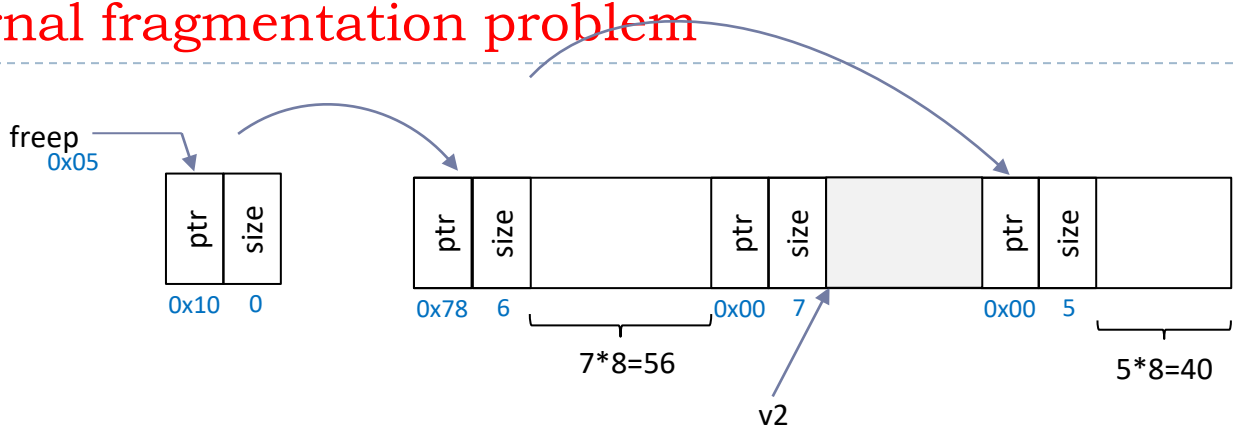
free



- free(v1) ;

Example: libc storage allocator

external fragmentation problem



- v1 = malloc(20*sizeof(int)) ; // 20*4 = 80 bytes

- Over time, several malloc+free calls left many empty holes between used blocks.
  - Slow search in linked list
  - There are free space to satisfy one request, but not in a single piece block

Libc storage allocator

```
acaldero@phoenix:~/infodso/$ ./ptr
Violación de segmento

acaldero@phoenix:~/infodso/$ gdb ptr
GNU gdb (GDB) 7.2-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and
redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying"
and "show warranty" for details.
```

1) Main problems
   a) Internal and external fragmentation
   b) Overwrite metadata
   c) Free non-allocated memory
   d) Free two times the same allocated memory
   e) ...
2) Any advantages?
   a) Simple?
   b) Fast?

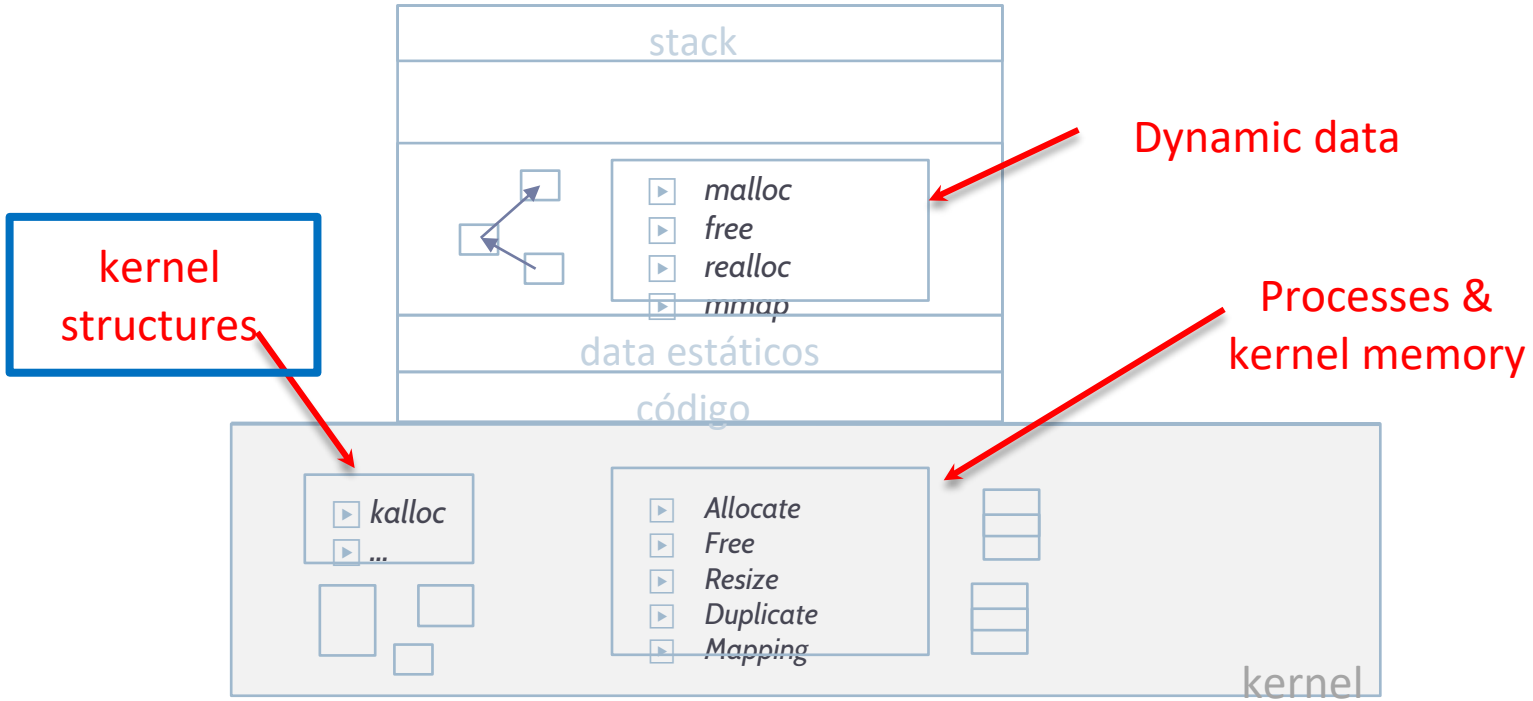
Overview

1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) **Dynamic memory allocator in kernel**
3) Virtual memory allocator

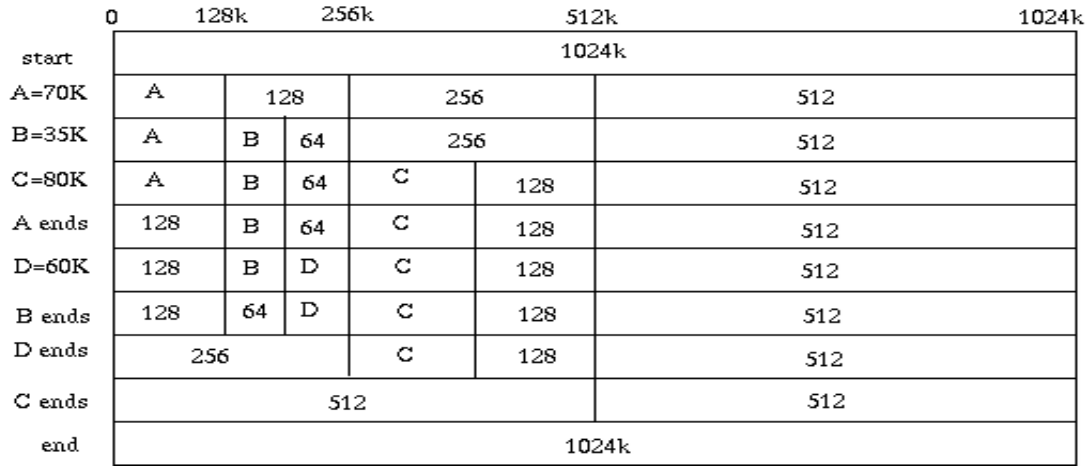
1) Management policies and management guidelines

Memory managers at several levels: Level 1



Memory management in the kernel

- With less external fragmentation and less overload in the compaction: *the buddy memory allocator*



https://secure.wikimedia.org/wikipedia/en/wiki/Buddy_memory_allocation

The memory management kernel

- Many kernels use the *slab allocation*
  - E.g.: Solaris, FreeBSD, Linux, etc.
- Based on *Mach's zone allocator*
- It pre-assign portions of memory for common (and more frequently used) data types
  - Easier to find a free portion, and natural memory compaction after freeing
  - In this conditions is more efficient and avoids many memory fragmentation
- It is possible to see how the kernel is using it by:
  - cat /proc/slabinfo

- http://www.ibm.com/developerworks/linux/library/l-linux-slab-allocator/
- http://www.arl.wustl.edu/~fredk/Courses/cse522/fall03/Lectures/kmem.ppt

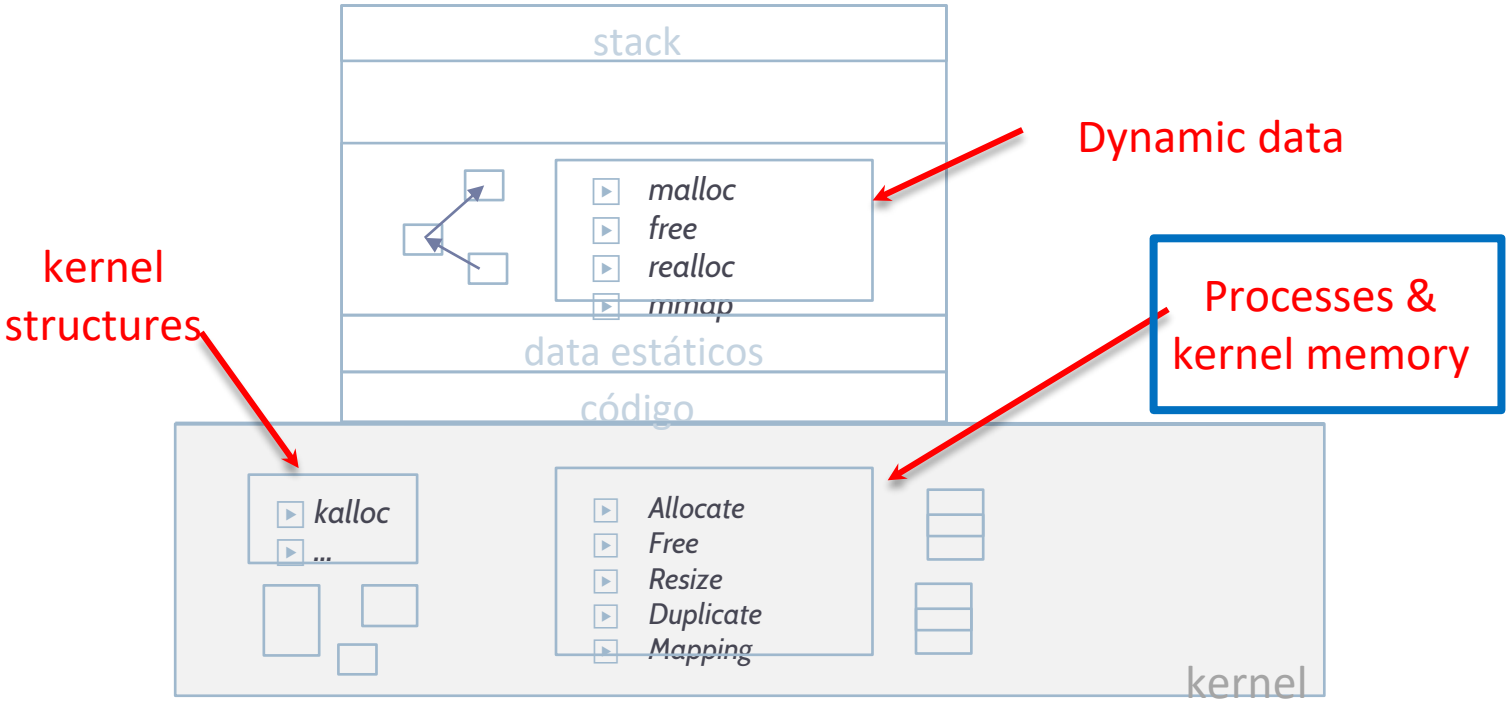
Overview

1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) **Virtual memory allocator**

1) Management policies and management guidelines

Memory managers at several levels: Level 1



Operations on regions

create region

- Main memory is not assigned to new region (loaded by demand)
  - Region pages are marked as invalid
- Depending on the kind of support:
  - Soporte on file
    - Pages are marked as Load From File (LFF)
    - There is stored the address of the corresponding disk block
  - Without support
    - For safety, pages are marked as Fill with Zeros (FC)
    - Page fault does not imply to read from device
- Once a region is created, when a modified page is expelled
  - If region is private then it is written in swap
  - If region is shared then it is written in support file area
- Stack is “special”: must contain the program arguments
  - Arguments are typically copied in swap block(s)

Operations on regions
free region

- Update region table to remove region
- Mark associated pages as invalid
- If region is private then the associated swap space is freed up

- Release/free an area may be due to:
  - Explicit request (E.g.: memory unmap)
  - Process termination (E.g.: EXIT on POSIX)
  - EXEC on POSIX release the current process map before building a new map linked to the executable be 'exec'

Operations on regions
resize region (change its size)

- If it decrease then similar to release but only one part affected
- If it increases in size:
  - Check for overlapping (to avoid it)
  - If pre-allocation then allocated swap space for the new pages
- Special cases:
  - Expansion of heap and mapped files
    - Requested by program through O.S. system calls
  - Expansion of stack is more complex: it is “automatic”
    - Program decrease SP value and accesses expanded zoe
      - Page fault
    - Page fault treatment:
      - If address is really invalid
        - If address < SP –› abort process and send signal
        - If not –› expasion of stack

Operations on regions

copy-on-write (lazy copy)

- Required for FORK in UNIX
  - Costly and non-efficient operation: all content must be copied
- Optimization: copy-on-write (COW)
  - Duplicate region pages are shared but:
    - are marked read-only and COW bit set
    - first write –› protection exception –› private copying
  - There can be several processes with the same duplicated region
    - There is one usage counter per page
    - Each time a private copy is created it decrements the counter
    - If it reaches 1 then COW bit is reset (there are no duplicates)
  - FORK now does not duplicate memory content, only the Page Table

Operations on regions
files mapped in memory

- Program requests map a file (or part) in its image
  - Program can specify protection level and if it is private or shared
- O.S. fill corresponding page/segment entries with:
  - Non-resident, LFF
  - Private/Shared and protection, as specified by mmap system call
  - TP entries refers to a user file

- It is used as:
  - Alternative form of file access for read/write data
  - Loading dynamic libraries
  - Globally: generalization of virtual memory (more explicit access to it)

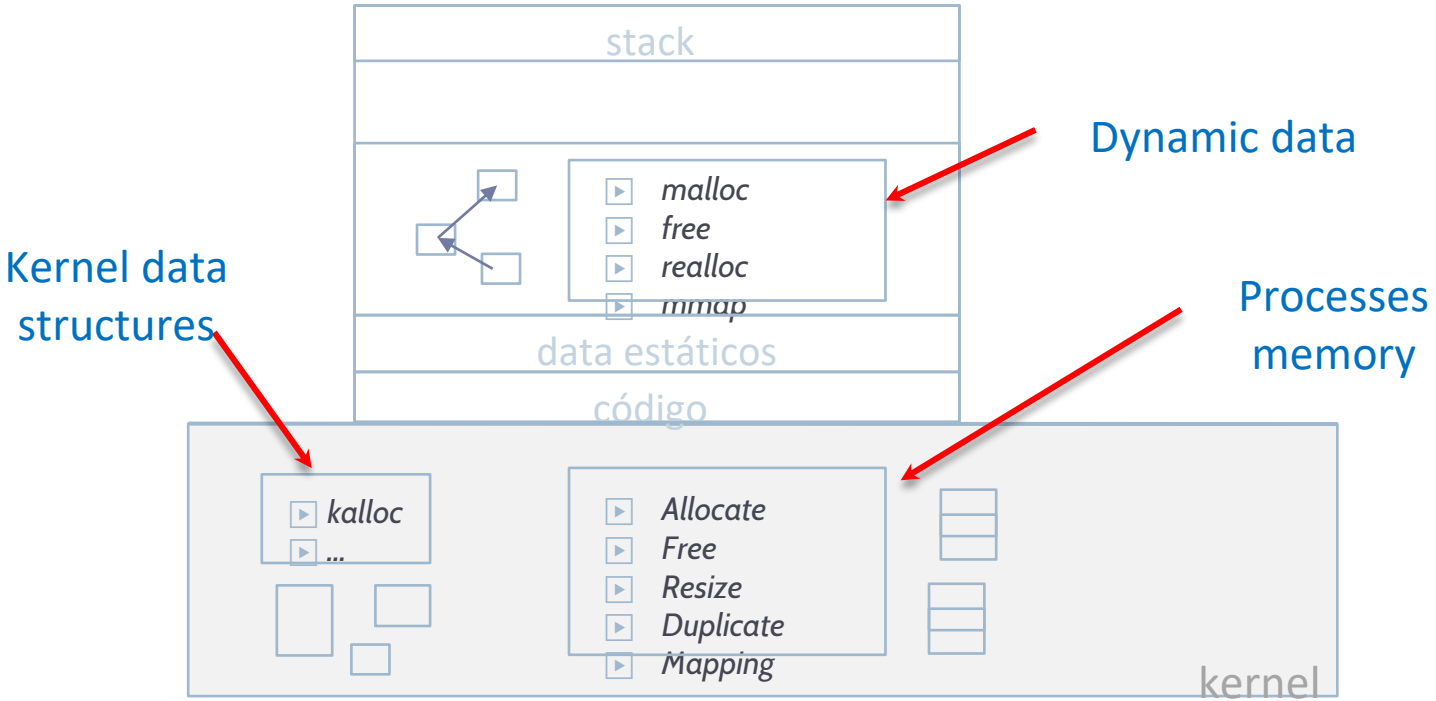
Operations on regions
deduplication of a region

New

- copy-on-write (COW) seeks to share pages between father and son processes to avoid duplicate pages with the same content.
  - When it is modified is when a copy is made and that copy is modified with the new content.
- Deduplication (KSM) seeks to share pages between unrelated processes to avoid duplicate pages with same content.
  - When two pages with same content are detected then the page/segment table is updated to share them.
  - When the page is going to be modified then a copy is made and that copy is modified. Then the page/segment table is updated with the new page.

http://en.wikipedia.org/wiki/Kernel_same-page_merging

Memory managers at several levels
resumen



Overview

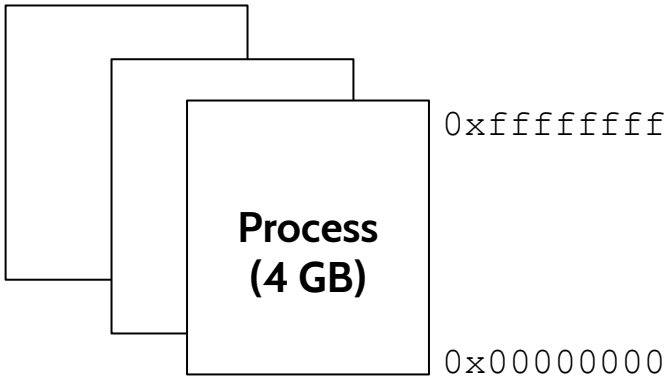
1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

1) **Management policies and management guidelines**
   a) **Kernel/Processes**
   b) Parameters
   c) Extended aspects

Memory Space: process + kernel

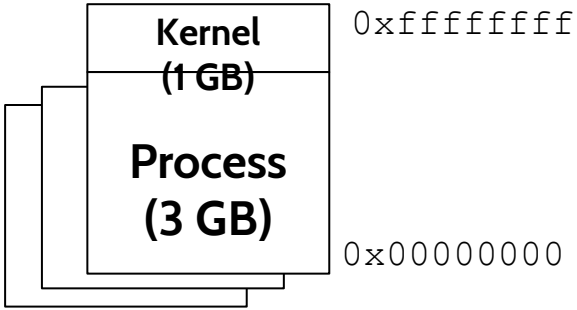
- Each process 'see' a lineal and flat address space
  - Each process could access to all available memory space

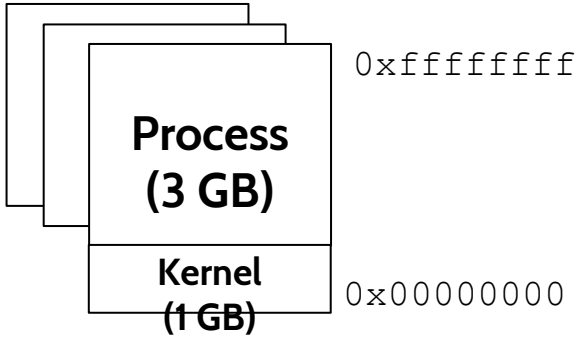


http://www.cs.rutgers.edu/~pxk/416/notes/content/09-memory_management-slides-6.pdf

Memory Space: process + kernel

- Space used by kernel is mapped (and shared) by all processes
  - It does not change in context switching
- The kernel space is protected (read, no write, and execution)
  - Faster system calls because avoid to change mode (u→k and k → u)



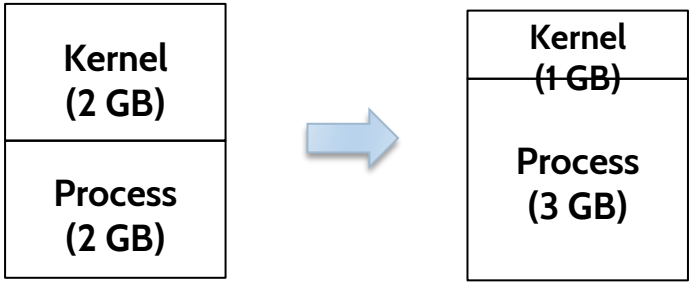


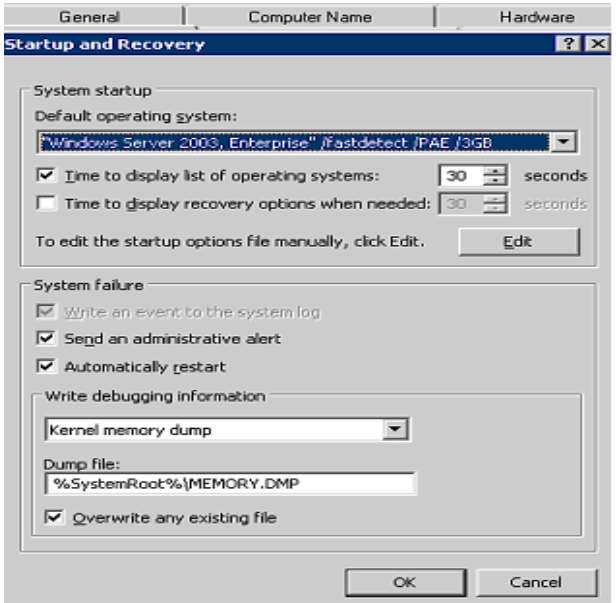
http://www.cs.rutgers.edu/~pxk/416/notes/content/09-memory_management-slides-6.pdf

Memory Space: process + kernel

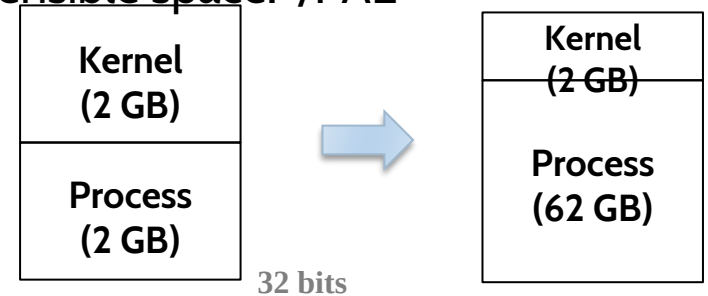
Windows

- Configurable división: /3GB





- Extensible space: /PAE



32 bits

36 bits

http://www.performancewiki.com/windows-tuning.html

Overview

1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

1) **Management policies and management guidelines**
   a) **Kernel/Processes**
   b) Parameters
   c) Extended aspects

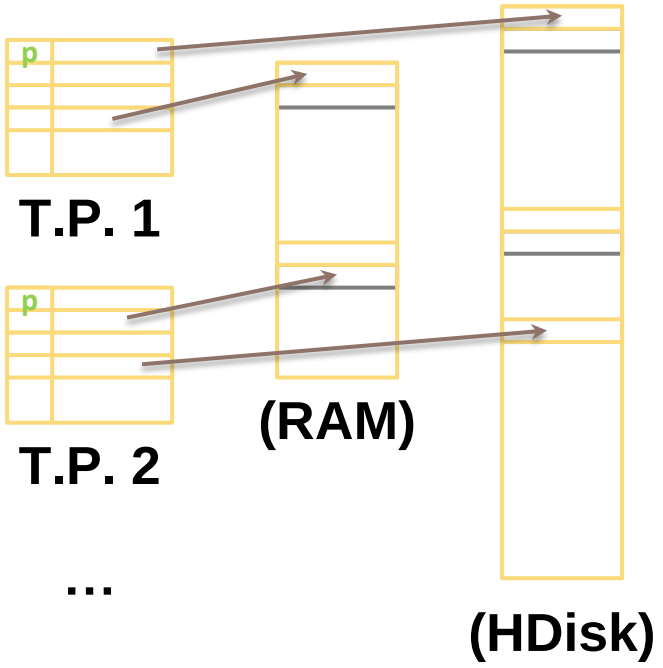
- **Page size**
- **Resident set**
- **Degree of multiprogramming**

Working with different memory spaces

Main parameters (1/4)

- Segment table per process
- One register points to the table of the current process

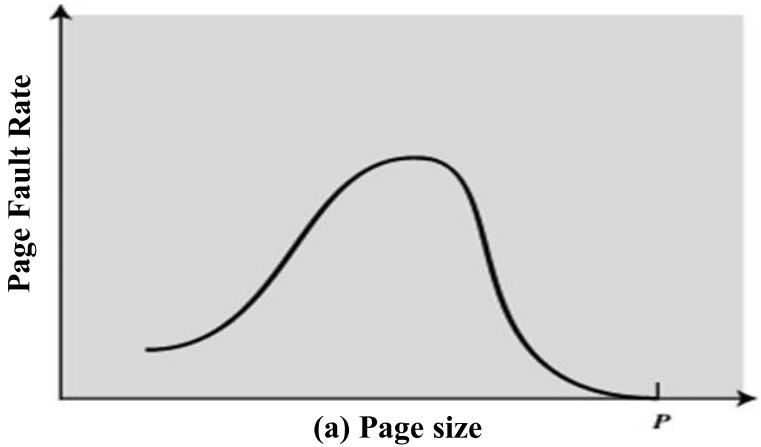
- **Degree of multiprogramming:**
  number of processes in memory at a given moment in time
- **Resident set:**
  number of pages of one process in main memory at a given moment in time
- **Page size:**
  Page size in bytes (usually at system level)



Main parameters (2/4)

- O.S. has to balance:
  - The number of processes in memory (**Degree of multiprogramming**)
  - The number of pages in main memory each process has (**Resident set**) with the minimal number it requires to work (**Working set**)
  - The Page size.
    - Size of Page Table, transfer with secondary memory, number of page Faults, etc.

Typical behavior of paging in a program.

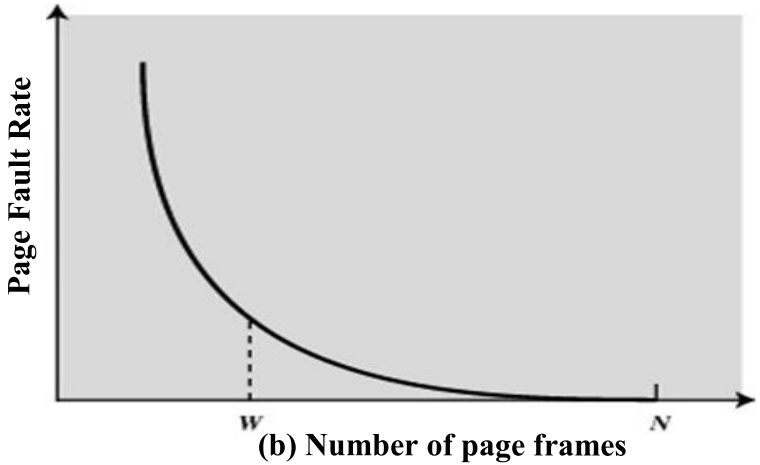


P **= Size of the whole process**
W **= Size of the working set**
N **= Total number of pages of the process**

Main parameters (3/4)

- O.S. has to balance:
  - The number of processes in memory (**Degree of multiprogramming**)
  - The number of pages in main memory each process has (**Resident set**) with the minimal number it requires to work (**Working set**)
  - The Page size.
    - Size of Page Table, transfer with secondary memory, number of page Faults, etc.

Typical behavior of paging in a program.

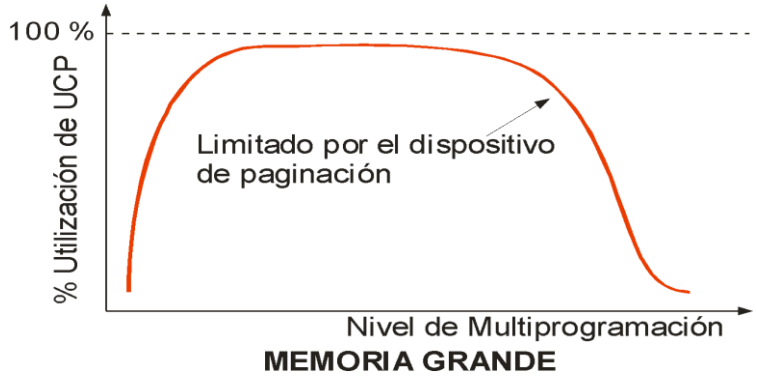


P **= Size of the whole process**
W **= Size of the working set**
N **= Total number of pages of the process**

Main parameters (4/4)

- O.S. has to balance:
  - The number of processes in memory (**Degree of multiprogramming**)
    - **Swapping**:
      - High transfer of information between Main Memory and Secondary Memory.
    - **HyperPaging**:
      - It occurs when the number of page faults is very high.
      - The system spends more time exchanging fragments than executing user instructions.
  - ...

Typical behavior of paging in a program.



Solutions for thrashing (1/2)

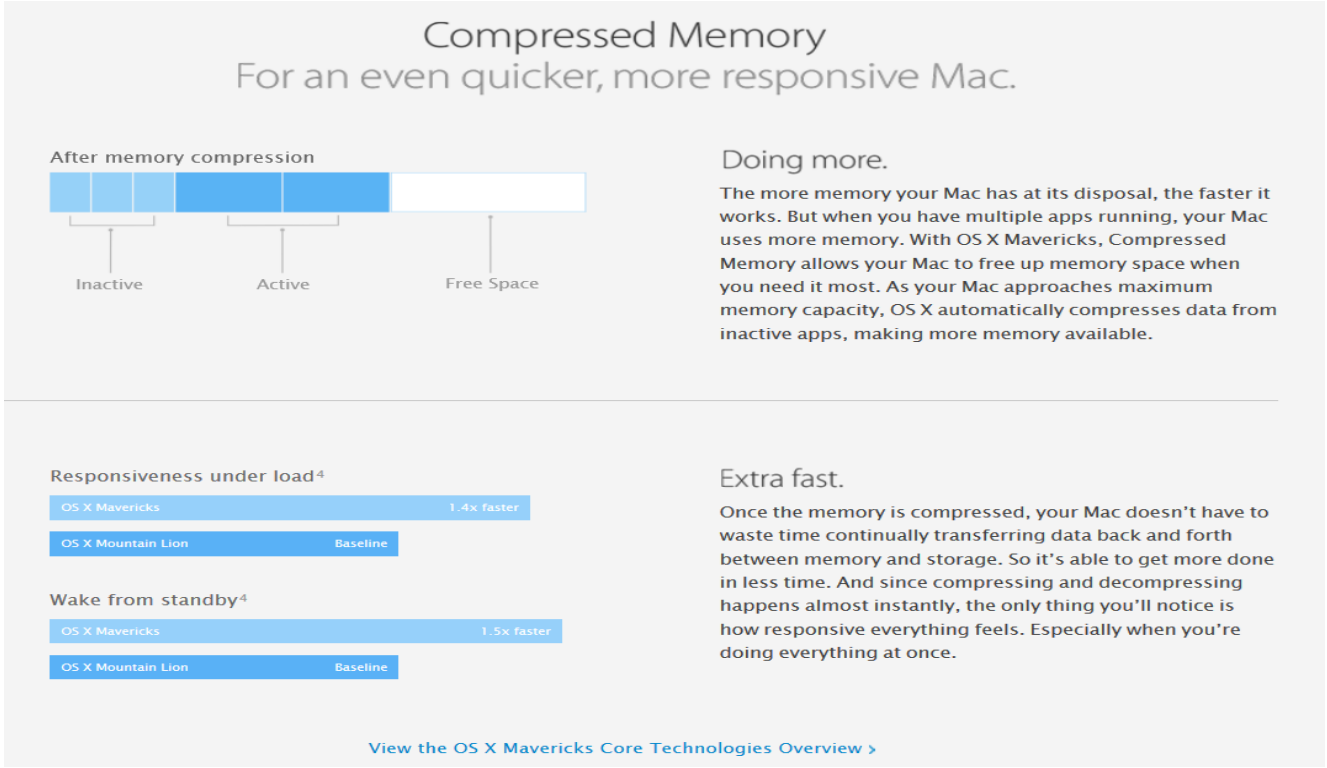
- Solutions with local replacement
  - Working set group strategy
    - Try to know the working set of each process
    - If working set decrease => free page frames
    - If working set increase => alloc more page frames
      - If there is not frames available: suspend processes
      - Processes are resumed when free frames are available again for the working set
  - Fault frequency-based strategy
    - If fault rate < lower limit => free page frames
    - If fault rate > upper limit => alloc page frames
      - If there is not free page frames => suspend some processes

Solutions for thrashing (2/2)

- Solutions with global replacement
  - There are no proper solutions.
  - BSD: buffering daemon activated by threshold.
    - If frequently activated -> suspend some process.
    - General idea: keep a reserve of free frames.
    - If number of free frames < threshold
      - "page daemon" repeatedly applies the replacement algorithm:
        - unmodified pages go to list of free frames
        - modified pages go to list of modified frames
    - If a page of the lists is referenced, it is used directly.

Ideas so reduce hyperpagination…

MacOS



http://www.zdnet.com/mavericks-memory-compression-review-7000022515/

Overview

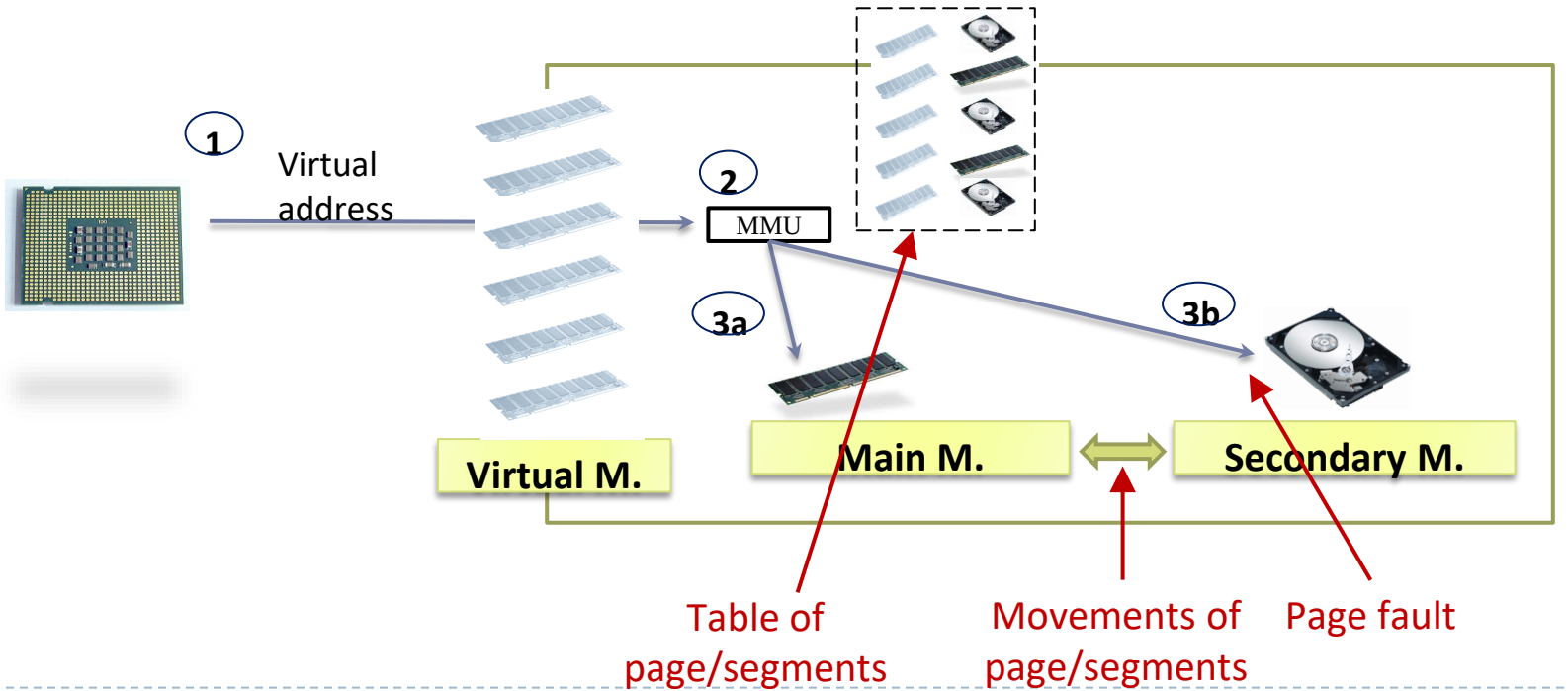
1) Introduction
   a) Memory allocator
   b) Memory allocator hierarchy

1) Dynamic memory allocator in user space
2) Dynamic memory allocator in kernel
3) Virtual memory allocator

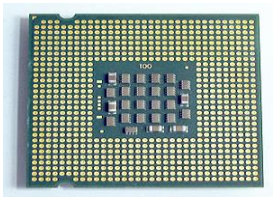
1) **Management policies and management guidelines**
   a) **Kernel/Processes**
   b) Parameters
   c) Extended aspects

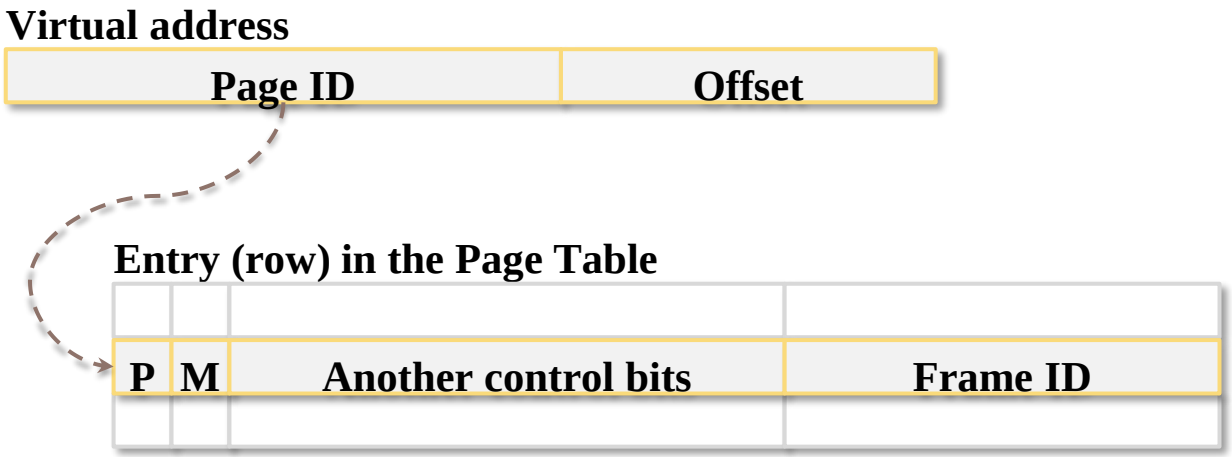
Page/segment table
Movement of page/segments

Virtual Memory Systems



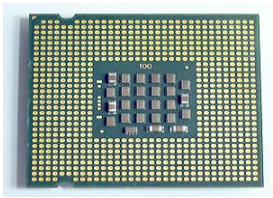
Page Table entries (rows)
Typical format

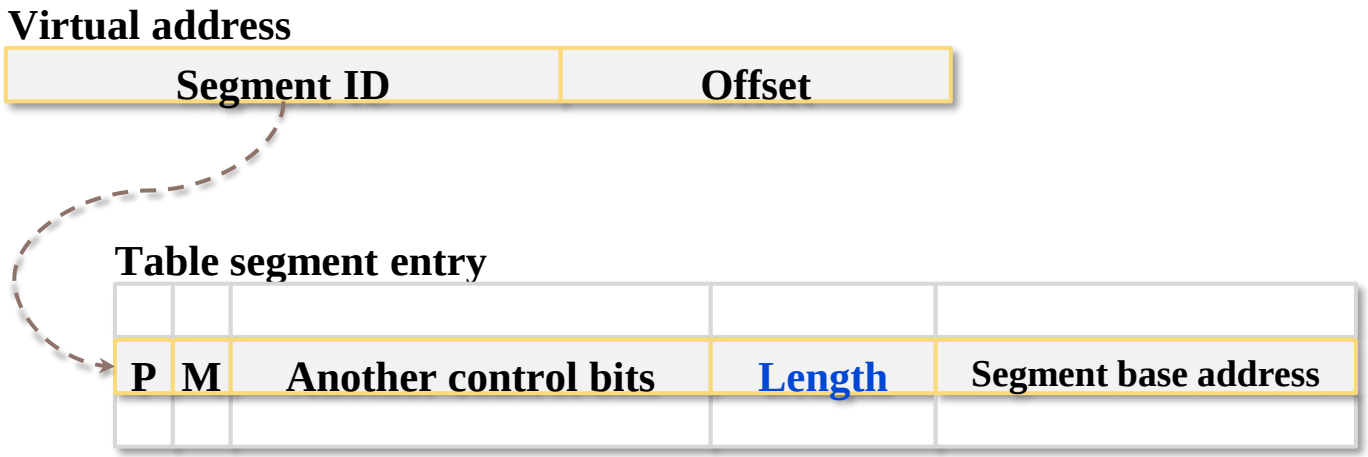




- Bit P: indicates if associated page is present in main memory
- Bit M: indicates if page content has been modify in main memory
- Other bits: protection (read, write, execute, etc.), mgmt. (cow, etc.)

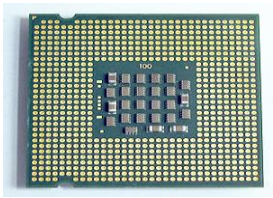
Segment Table entries (rows)
Typical format

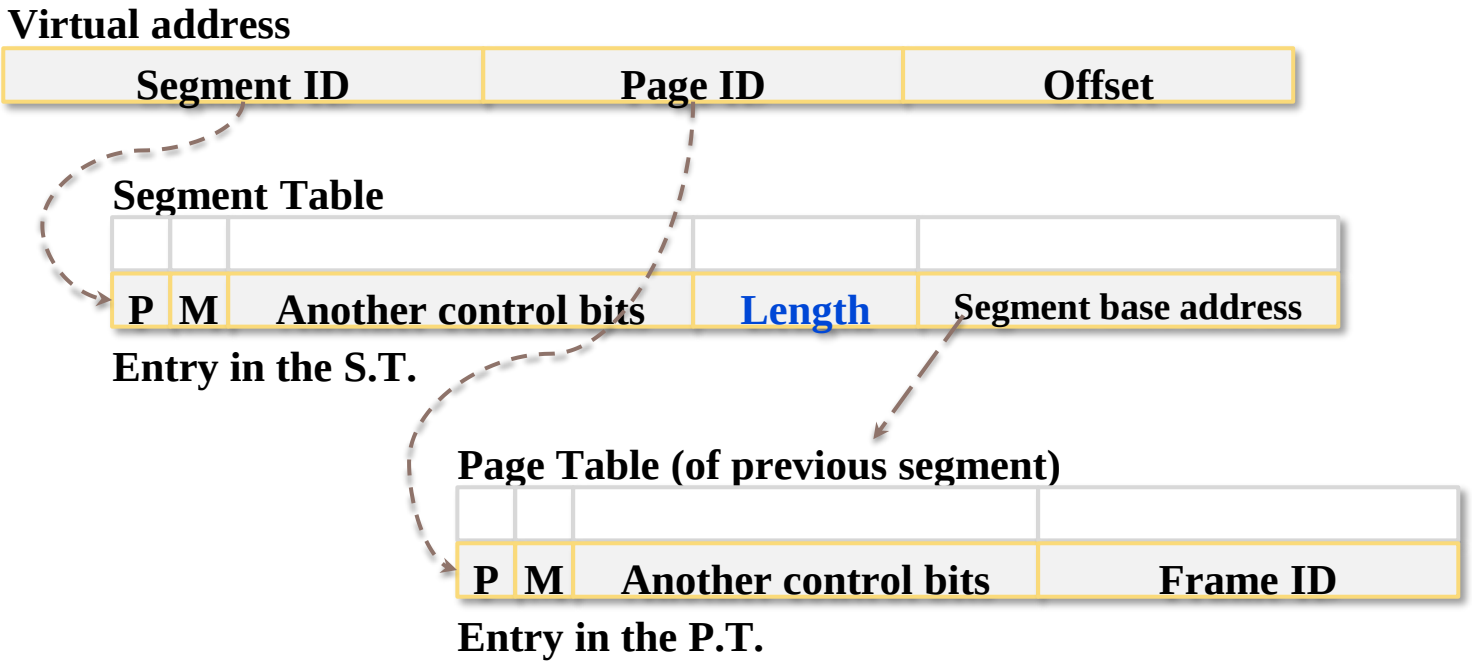




- Bit P: present in main memory
- Bit M: copy in main memory has been modified
- Another control bits: R,W,X,COW,…

Table entries (rows)
Typical format





Page Table management

- Initially:
  - O.S. creates when is going to execute one application.
- Used by:
  - MMU uses it in the translation process (Vir.A. -> Phy.A.).
- Updated by:
  - O.S. updates page tables in the page fault handler routine.

Movements of segments

- Initially:
  - **It is defined in the executable file of the application (defines process memory layout)**
  - **Code (text) is loaded, stack is initialized, etc.**
- From secondary mem. to main mem. (by demand):
  - Access to non-resident segment: segment fault
  - The O.S. read the segment from secondary mem. and takes it to main mem.
- From main mem. to secondary mem. (by expulsion):
  - There is no enough space in main mem. to load the segment in
  - An already resident segment is replaced by
  - O.S. save the released segment into secondary mem. (if M bit is set to 1)

Movements of pages

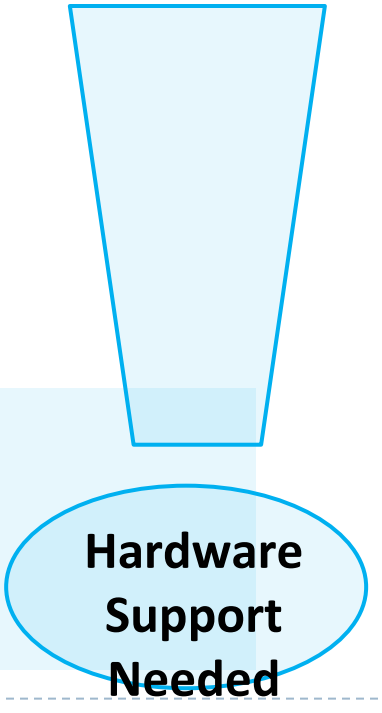
- Initially:
  - Non-resident page is marked as missing
  - The O.S. saves the swap block id. where the page is stored
- From secondary mem. to main mem. (by demand):
  - Access to non resident page: Page fault
  - The O.S. read the page from secondary mem. and takes it to main mem.
- From main mem. to secondary mem. (by expulsion):
  - There is no enough space in main mem. to load the page in
  - An already resident page is replaced
  - O.S. save the released page into secondary mem. (if M bit is set to 1)

(general) Page Fault Handling

- If invalid address -› aborts process or sends signal
- If there is no free frame (see frames table)
  - Victim selection (replacement alg.): page P frame M
    - Mark P as invalid
  - If P has been modified (*M* bit of P is active)
    - Starts write request of P in secondary memory
- If there is free frame (free available or it has been released previously):
  - Start reading page in frame M
  - Mark page entry as valid, referencing to M
  - Set M as occupied in the frames table (if it wasn't)

Movement of pages

- Initially:
  - Non-resident page is marked as missing
  - Page entry stores the address of the swap block containing it
- From secondary mem. to main mem. (by demand):
  - Access to non resident page: Page fault
  - O.S. reads page from Secondary M. to Main M.
- From main mem. to secondary mem. (by expulsion):
  - There is no space in Main M. to bring page back
  - A resident page is expelled (replaced)
  - If bit M=1 then O.S. writes expelled page to Secondary M.



Management Policies

- Replacement Policy:
  - Local replacement: within the process
  - Global replacement
- Replacement algorithms: valid for local and global
  - Optimum
  - FIFO
  - Clock (or second change)
  - LRU
- Policy for assigning frames to processes:
  - Fixed allocation (always with local replacement):
    - Resident set of process is constant
  - Dynamic assignment (local or global replacement):
    - Resident set of process is variable

No Replacement Algorithms

- Locking of frames:
  - When a frame is locked, the loaded page in that frame cannot be replaced.
- Examples of when a frame is locked:
  - The majority of the operating system kernel.
  - Control structures.
  - I/O buffers (E.g.: the one used for DMA).
- **Locking is achieved by associating a blocking bit to each frame.**

Replacement Algorithms

- Which page will be replaced.
- The page to be replaced must be the one that has the least chance of being referenced in the near future..
- Most policies try to predict future behavior based on past behavior (heuristics).
- Example of policies: **LRU, FIFO, etc.**

Basic replacement Algorithms

- **Optimal** policy:
  - Selects to replace the page that has to wait a longer amount of time until the following reference occurs.
  - Impossible to implement because it requires the operating system to have accurate knowledge of future events.

Basic replacement Algorithms

- Less recently used' policy (**LRU**):
  - Replaces the memory page that has not been referenced for a long time.
  - Due to the closeness principle, this would be the page with the least probability of being referenced in the near future.
  - **One solution would be to tag each page with the timestamp of its last reference.**

Basic replacement Algorithms

- First in, first out' policy (**FIFO**):
  - Replace the page in memory that was first loaded (the which one that has been longer time in memory)
  - These pages may be needed again and in a short period of time.
  - One of the easiest replacement policies to implement:
    - **Treats the frames assigned to a process as a circular buffer.**
    - **Pages are deleted from memory according to the round-robin technique.**

ARCOS Group
Computer Science and Engineering Department
Universidad Carlos III de Madrid

Lesson 5 (b)
Memory Management

Operating System Design
Degree in Computer Science and Engineering, Double Degree CS&E + BA