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

Lesson 5 (a)
Memory Management

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

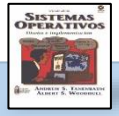
CC BY NC SA

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…

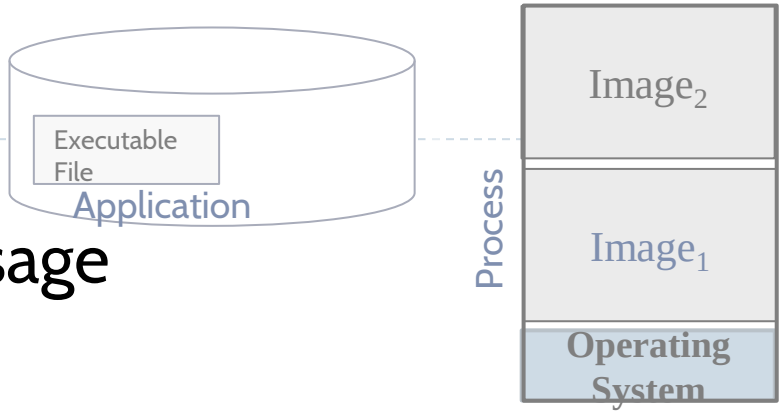
1. To study the associated theory.
   - Better study the bibliography readings because the slides are not enough.
   - To add questions with answers, and proper justification.
1. To review what in class is introduced.
   - To do the practical Linux task step-by-step.
2. To practice the knowledge and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

Overview

1. Introduction to memory usage
    1. Abstract model
    2. Definitions and environments
    3. Regions of process memory
    4. How to prepare an executable

1. Introduction to Virtual Memory

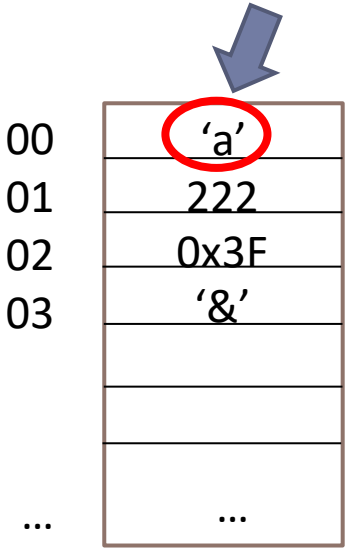
Overview



1. Introduction to memory usage
    1. **Abstract model**
    2. Definitions and environments
    3. Regions of process memory
    4. How to prepare an executable

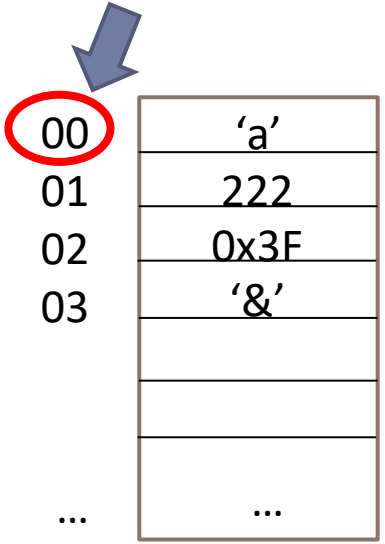
1. Introduction to Virtual Memory

Basic usage of memory
address, value, and size



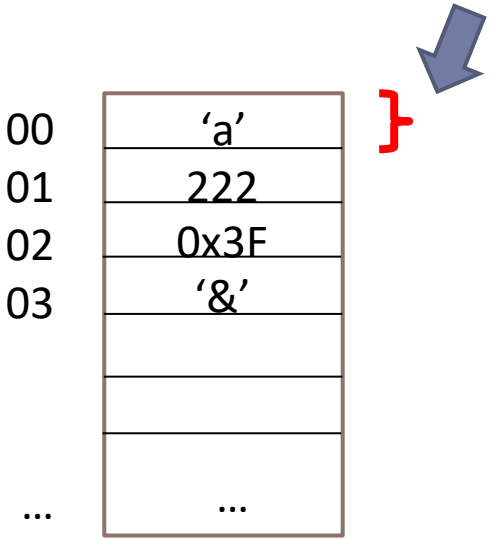
- Value
  - Element stored in memory.

Basic usage of memory
address, value, and size



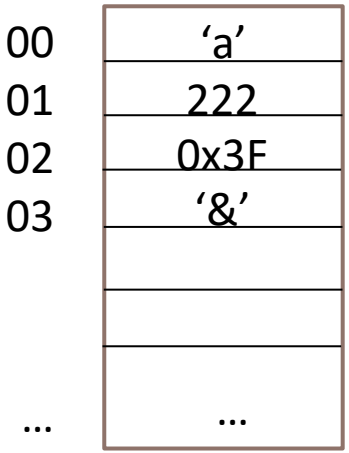
- Value
  - Element stored in memory.
- Address
  - Place in memory.

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.

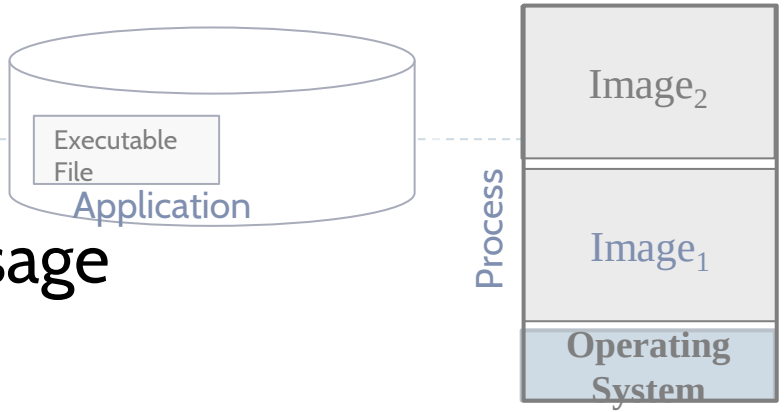
Basic usage of memory
functional interface



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

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

Overview



1. Introduction to memory usage
    1. Abstract model
    2. **Definitions and environments**
    3. Regions of process memory
    4. How to prepare an executable

1. Introduction to Virtual Memory

Introduction

- Definitions

  Application

  Process image

  Process

- Environments

  monoprogramming

  multiprogramming

Introduction

- Definitions

  Application

  Process image

  Process

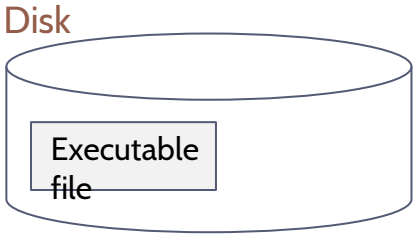
- Environments

  monoprogramming

  multiprogramming

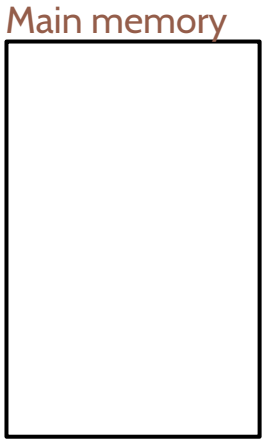
Application and Process

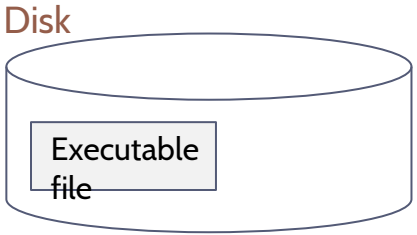
- Application: set of data and instruction sequence that is able to perform a specific task or work.



Application and Process

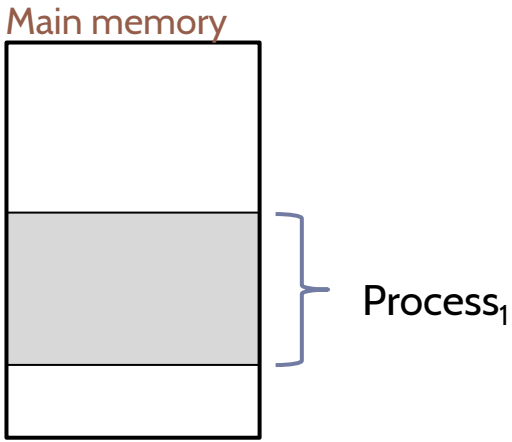
- Application: set of data and instruction sequence that is able to perform a specific task or work.
  - In order to be executed it has to be in memory.

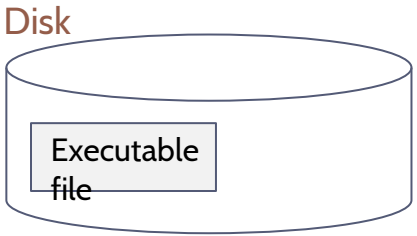




Application and Process

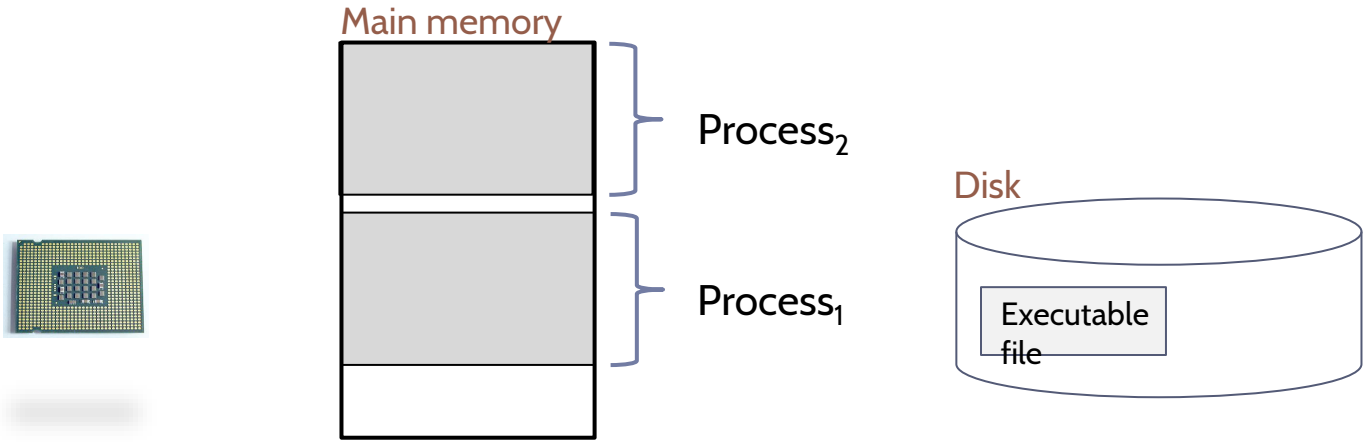
- Process: executing application.





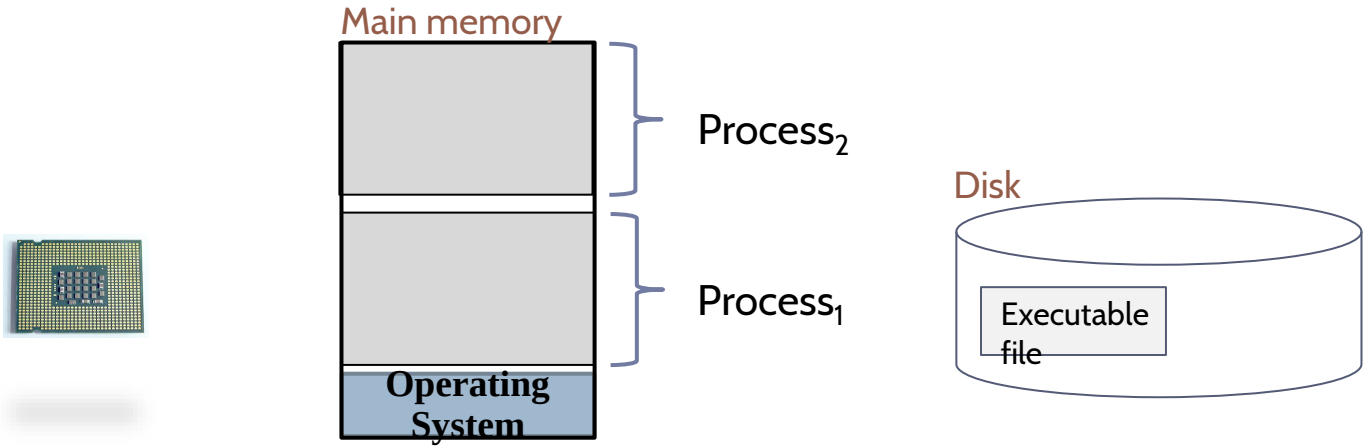
Application and Process

- Process: executing application.
  - It is possible that the same application executes several times (we will found several process in memory)



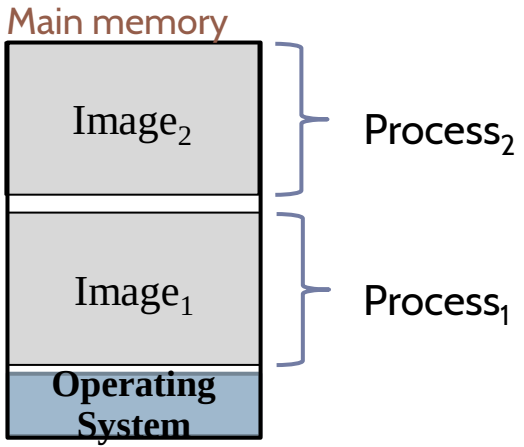
Application and Process

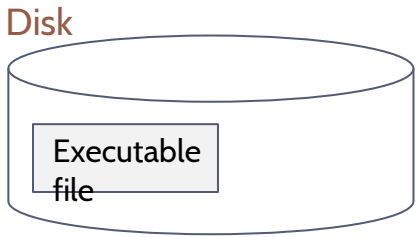
- Process: executing application.
  - The Operating System is loaded in memory and it performs the memory management in order to shared the memory among all process (in a similar way like it does with the CPU).



Process image

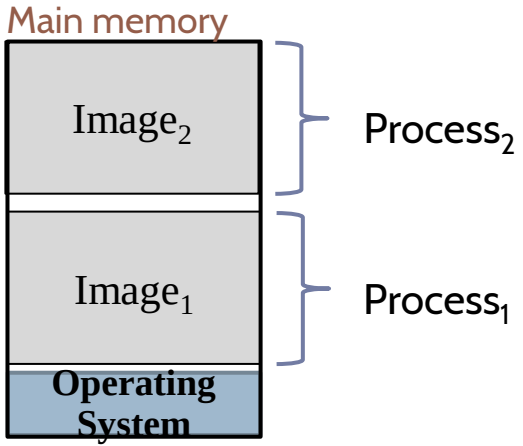
- Memory image: sequence of memory addresses (and its contents) allocated for a process.

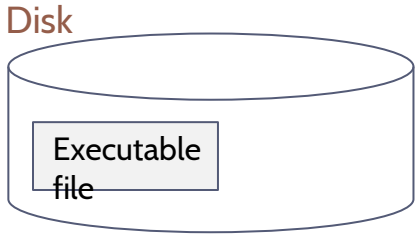




Process image

- Memory image: sequence of memory addresses (and its contents) allocated for a process.
  - Part of the control information is not in the PCB because efficiency reasons and sharing reasons.





Introduction

- Definitions

Application

Process image

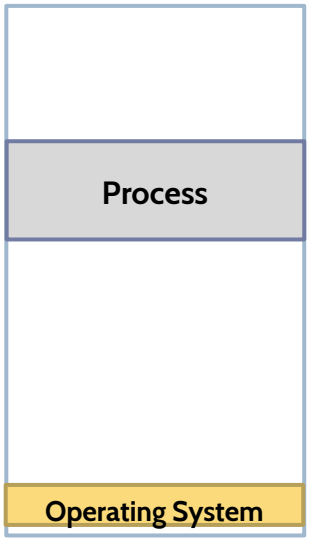
Process

- Environments

monoprogramming

multiprogramming

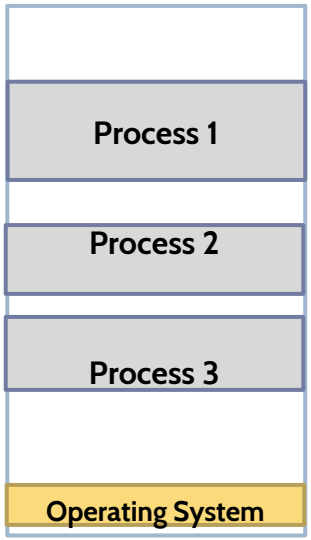
Monoprogramming Systems



- Only one process is executed at a time.
- Memory is shared between the operating system and the process.

- Ej.: MS-DOS, DR-DOS, etc.

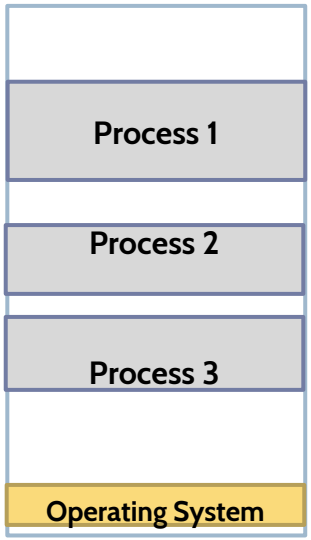
Multiprogramming systems



- More than one process is kept in main memory.
- It improves the CPU occupancy:
  - If process blocks -> execute another
- The memory management work is basically a constrained optimization task.

- E.g.: Unix, Windows NT, etc.

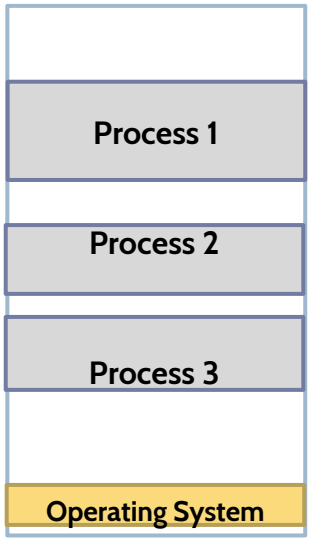
Multiprogramming systems



- More than one process is kept in main memory.
- It improves the CPU occupancy:
  - If process blocks -> execute another
- The memory management work is basically a constrained optimization task.
- E.g.: Unix, Windows NT, etc.

Multiprogramming systems

example of computing the CPU usage

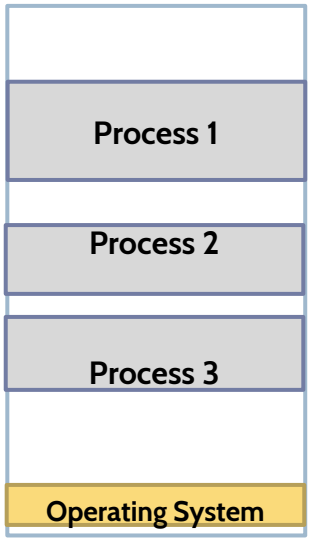


- n = 5 independent process
- p = 0,8 of time blocked (20% on CPU)
- **usage = 5*20% → 100%**

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

Multiprogramming systems

example of computing the CPU usage



- **p** = % of time that a process is blocked
- $\mathbf{p^n}$ = probability of n independent processes being all blocked
- **1 - $\mathbf{p^n}$** = probability of CPU **not** being idle
  - n = 5 independent process
  - p = 0,8 of time blocked (20% on CPU)
  - ~~usage = 5*20% → 100%~~
  - usage = $1-0{,}8^5$ → 67%
    $1-0{,}8^{10}$ → 89%

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

Introduction
summary

- Definitions

Application

Process image

Process

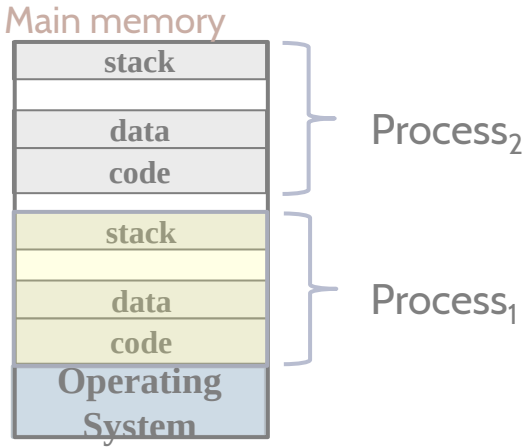
- Environments

monoprogramming

multiprogramming

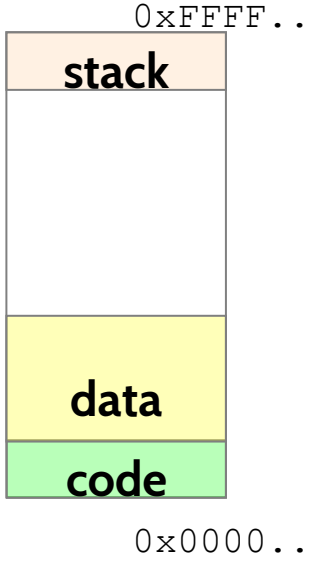
Overview

1. Introduction to memory usage
    1. Abstract model
    2. Definitions and environments
    3. **Regions of process memory**
    4. How to prepare an executable
1. Introduction to Virtual Memory



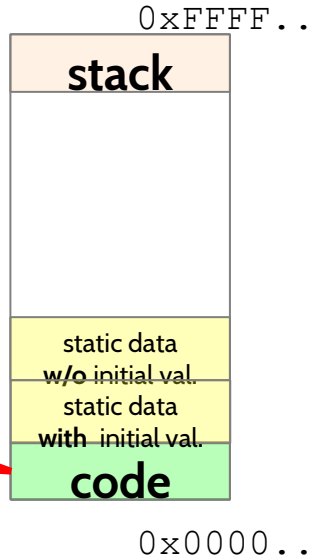
Logical organization (applications)
process memory model

- One process consists of a list of memory regions.
- One region is one contiguous memory area of the process address space where all elements have the same properties.
- Main properties:
  - Permissions: read, write, and execution.
  - Shared among threads: private or shared
  - Size (fixed/variable)
  - Initial value (with or without support)
  - Static or dynamic creation
  - Growth direction



Main process regions
code (*text*)

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```

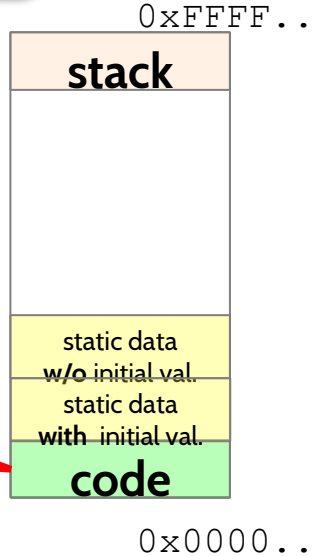


Main process regions

code (*text*)

code
- Static (fixed)
- It is known at compilation time.
- Instruction sequence to be executed.

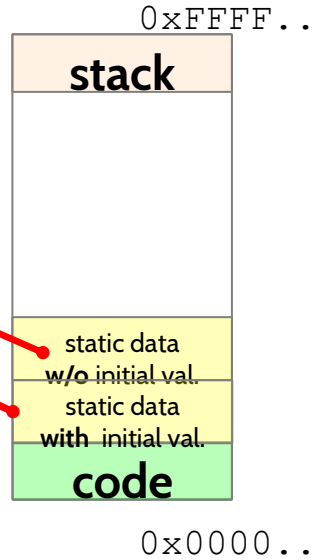
```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```



Main process regions

data (*data*)

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```

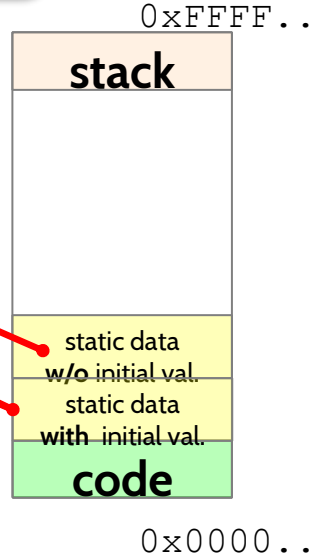


Main process regions

data (*data*)

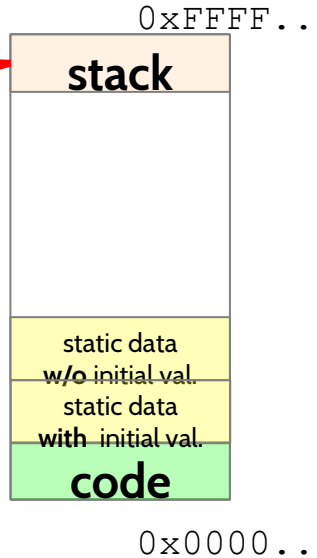
Global variables
- Static (fixed)
- They are allocated when application starts.
- They exit during the execution.
- Fixed address on memory and executable.

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```



Main process regions
stack (*stack*)

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```



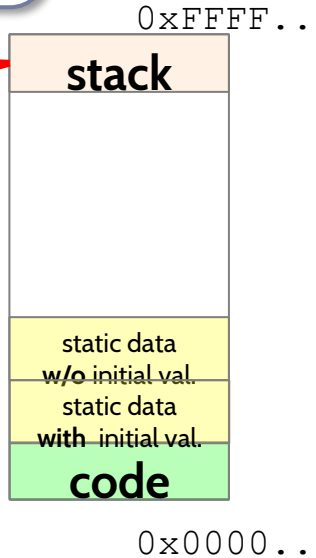
Main process regions

stack (*stack*)

Local variables and parameters

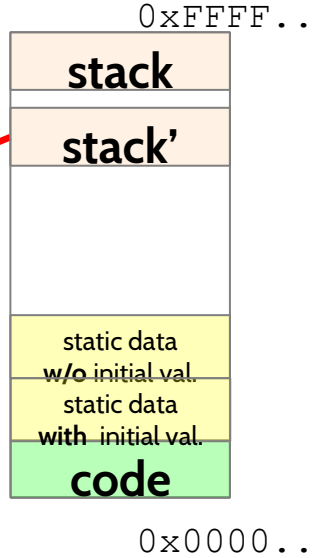
- Dynamics
- They are created on function invocation
- They are destroyed on function call return
- Recursivity: several instances of a variable

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```



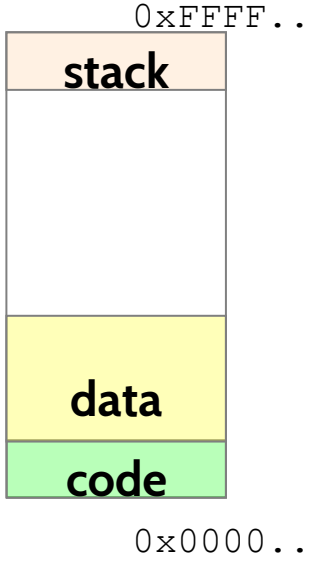
Main process regions
stack (*stack*)

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                ....... pthread_create(f…)
                free (p)
                ....
                exit (0)
}
```



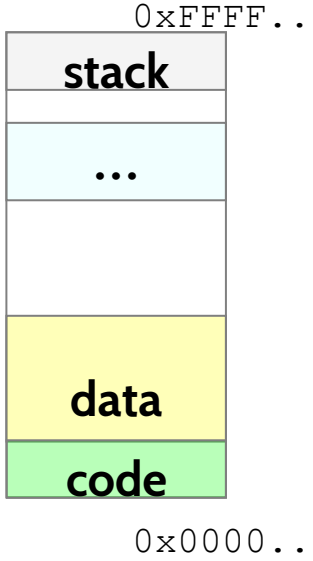
Logical organization (applications)
process memory model

- Code or text
  - Shared, RX, Fixed Size, executable support
- Data
  - With initial value
    - Shared, RW, Fixed size, executable support
  - Without initial value
    - Shared, RW, Fixed size, w/o support (0 filled)
- Stack
  - Private, RW, Variable size, w/o support (0 filled)
  - It grows toward lower addresses.
  - Initial stack: application arguments.



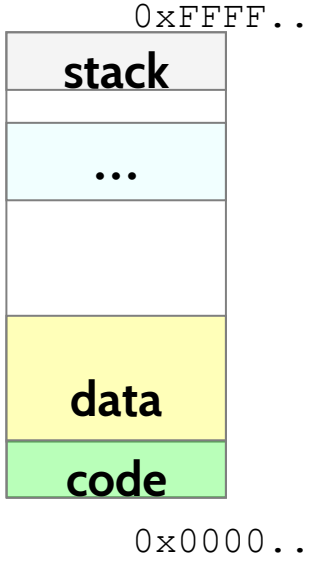
Logical organization (applications)
process memory model

- *Heap*
  - Support for dynamic memory (C malloc)
  - Shared, RW, Variable size, w/o support (0 filled?)
  - It grows toward higher addresses.
- Mapped files
  - Region associated to a mapped file.
  - Private/Shared, variable size, support on file.
  - Protection depends on projection.



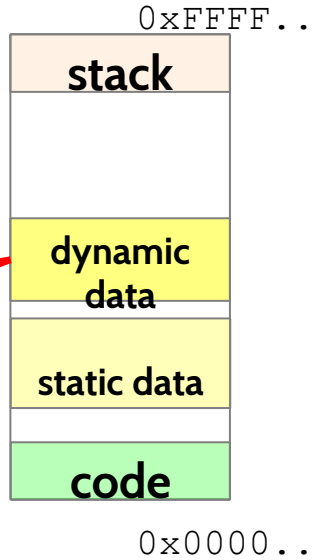
Logical organization (applications)
process memory model

- *Heap*
  - Support for dynamic memory (C malloc)
  - Shared, RW, Variable size, w/o support (0 filled?)
  - It grows toward higher addresses.
- Mapped files
  - Region associated to a mapped file.
  - Private/Shared, variable size, support on file.
  - Protection depends on projection.



Main process regions
dynamic data (*heap*)

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```



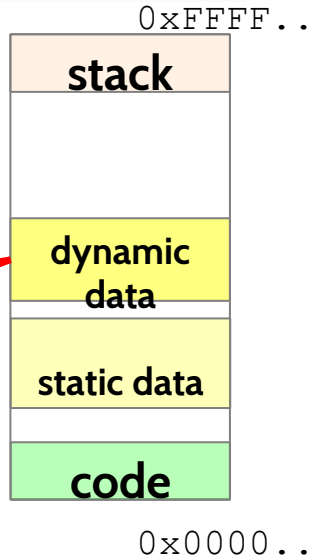
Main process regions

dynamic data (*heap*)

Dynamic variable

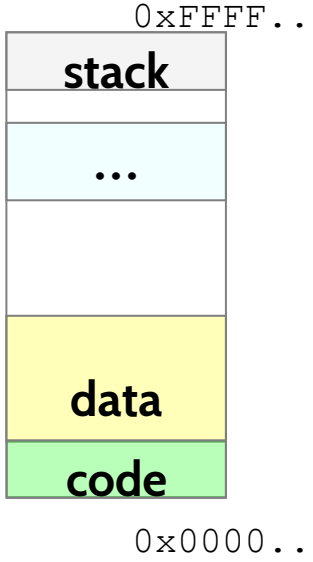
- Variables with no memory space assigned on compilation time.
- The memory space is allocated (and free) on runtime.

```
int a;
int b = 5;

void f(int c) {
                int d;
                static e = 2;

                b = d + 5;
                .......
                return;
}

main (int argc, char **argv) {
                char *p;
                p = (char *) malloc (1024)
                f(b)
                .......
                free (p)
                ....
                exit (0)
}
```

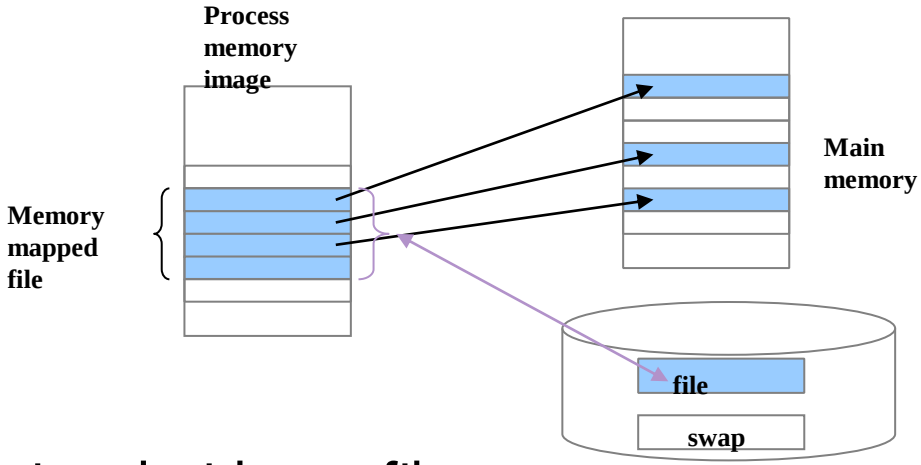


Logical organization (applications)
process memory model

- *Heap*
  - Soporte de memoria dinámica (C malloc)
  - Shared, RW, Variable size, w/o support (0 filled?)
  - It grows toward higher addresses.
- Mapped files
  - Region associated to a mapped file.
  - Private/Shared, variable size, support on file.
  - Protection depends on projection.



Memory mapped file (1/3)



- One process region is associated with one file.
- Some file blocks (pages) are in main memory.
- The process gets the file contents like it are consecutively in memory (access in memory rather than traditional read/write).

Memory mapped file (2/3)

*void *mmap (void *addr, size_t len, int prot, int flags, int fildes, off_t off);*

- Set a map between the process address space and the file descriptor or shared memory object.
  - It returns the memory address where the maps is going to be.
  - `addr` where mapping. If `NULL` then the O.S. will choose one.
  - `len` is the number of bytes to be mapped.
  - `prot` type of access (reading, writing, or executing).
  - `flags` options for the mapping operation (shared, private, etc.).
  - `fildes` file descriptor of a file or memory object to be mapped.
  - `off` offset within the file where the mapping is going to start.

*void munmap (void *addr, size_t len);*

- Unmaps part of the process address space that was mapped on the `addr` address.

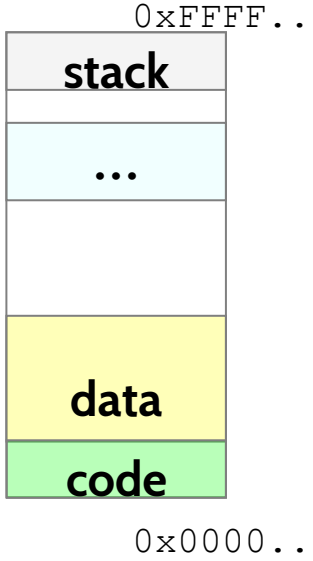
Memory mapped file (3/3)

- How many times a character appears in a memory mapped file.

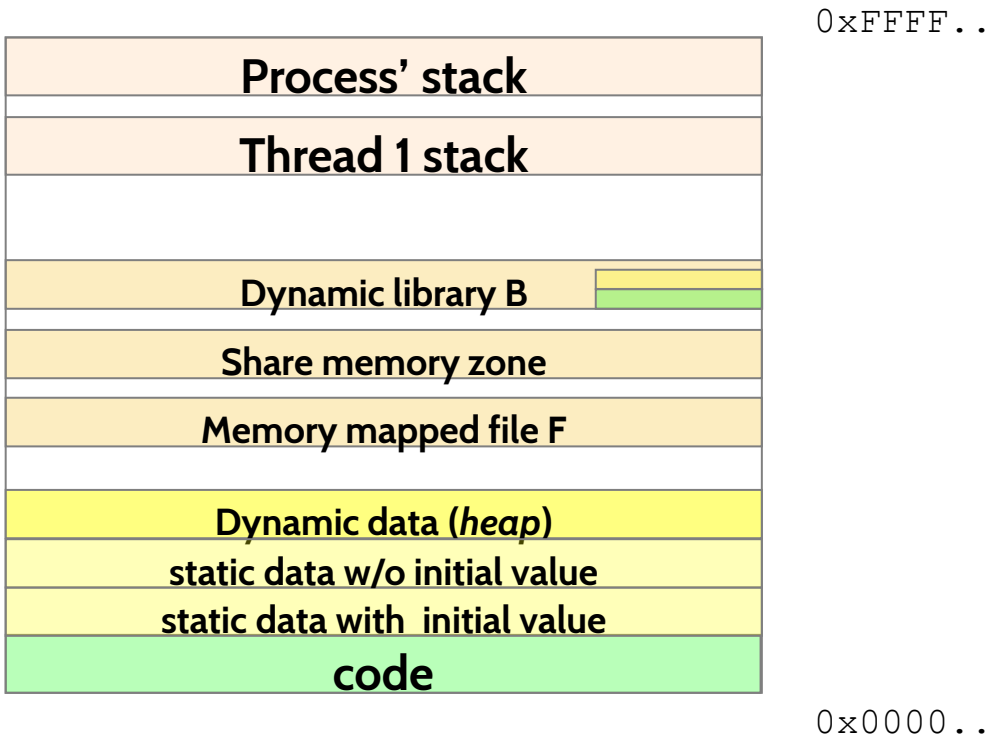
```
/* 1) To open the file descriptor */
fd=open(argv[2], O_RDONLY));
fstat(fd, &fs); /* Get the file size */

/* 2) To map the file */
org=mmap((caddr_t)0, fs.st_size, PROT_READ, MAP_SHARED, fd, 0));
close(fd); /* close the file descriptor */

/* 3) Access loop */
p=org;
for (i=0; i<fs.st_size; i++)
     if (*p++==caracter) contador++;

/* 4) To unmap the file */
munmap(org, fs.st_size);
printf("%d\n", contador);
```

Logical organization (applications)
process memory model

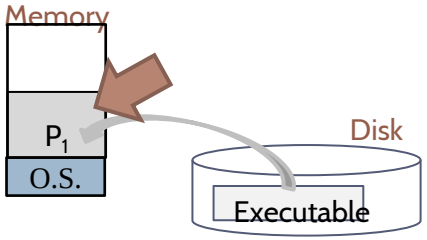
- *Heap*
  - Soporte de memoria dinámica (C malloc)
  - Private, RW, Variable size, w/o support (0 filled?)
  - It grows toward higher addresses.
- Mapped files
  - Region associated to a mapped file.
  - Shared, variable size, support on file.
  - Protection depends on projection.
- Dynamic libraries
  - Particular case of mapped files.
  - Code & data library is mapped to be executed.



Example of a process memory map



To inspect a process



- Details of the sections of a process:

```
acaldero@phoenix:~/infodso/$ cat /proc/1/maps
b7688000-b7692000 r-xp 00000000 08:02 1491       /lib/libnss_files-2.12.1.so
b7692000-b7693000 r--p 00009000 08:02 1491       /lib/libnss_files-2.12.1.so
b7693000-b7694000 rw-p 0000a000 08:02 1491       /lib/libnss_files-2.12.1.so
b7694000-b769d000 r-xp 00000000 08:02 3380       /lib/libnss_nis-2.12.1.so
b769d000-b769e000 r--p 00008000 08:02 3380       /lib/libnss_nis-2.12.1.so
b769e000-b769f000 rw-p 00009000 08:02 3380       /lib/libnss_nis-2.12.1.so
b769f000-b76b2000 r-xp 00000000 08:02 1414       /lib/libnsl-2.12.1.so
b76b2000-b76b3000 r--p 00012000 08:02 1414       /lib/libnsl-2.12.1.so
b76b3000-b76b4000 rw-p 00013000 08:02 1414       /lib/libnsl-2.12.1.so
b76b4000-b76b6000 rw-p 00000000 00:00 0
..
b78b7000-b78b8000 r-xp 00000000 00:00 0          [vdso]
b78b8000-b78d4000 r-xp 00000000 08:02 811        /lib/ld-2.12.1.so
b78d4000-b78d5000 r--p 0001b000 08:02 811        /lib/ld-2.12.1.so
b78d5000-b78d6000 rw-p 0001c000 08:02 811        /lib/ld-2.12.1.so
b78d6000-b78ef000 r-xp 00000000 08:02 1699       /sbin/init
b78ef000-b78f0000 r--p 00019000 08:02 1699       /sbin/init
b78f0000-b78f1000 rw-p 0001a000 08:02 1699       /sbin/init
b81e5000-b8247000 rw-p 00000000 00:00 0          [heap]
bf851000-bf872000 rw-p 00000000 00:00 0          [stack]
```

address **perm.** **offset** **dev** **i-node** **name**

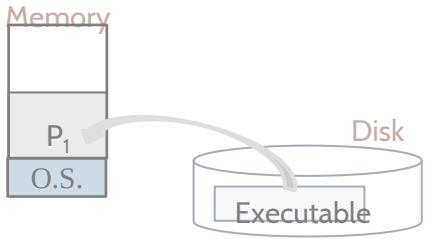
Exercice
fill the attribute for typical regions

Region	Support	Protection	Shared/Priv.	Size
code	File	RX	Shared	Fixed
…	…	…	…	…

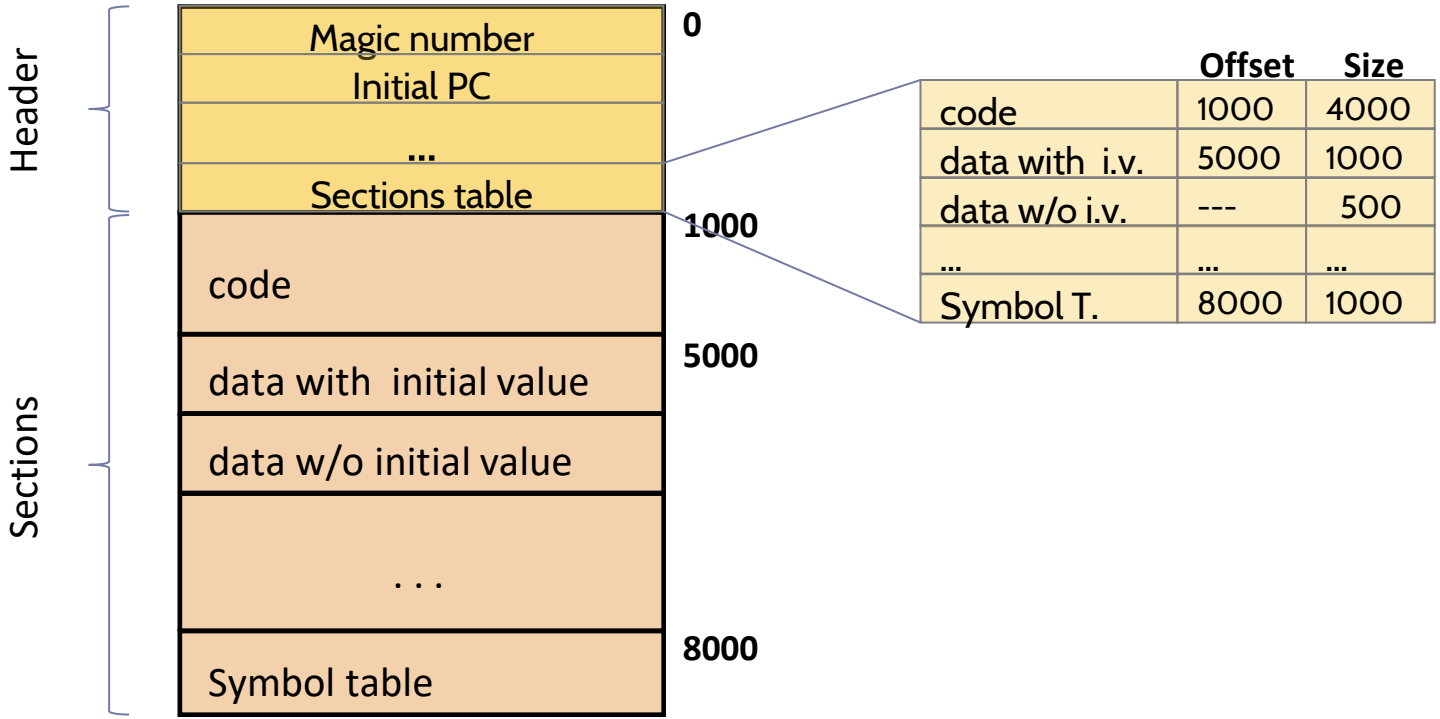
Overview

1. Introduction to memory usage
    1. Abstract model
    2. Definitions and environments
    3. Regions of process memory
    4. **How to prepare an executable**

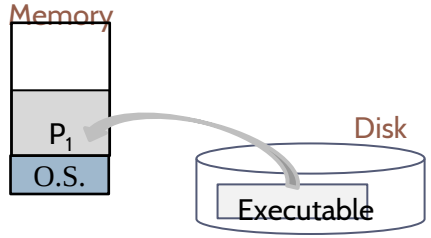
1. Introduction to Virtual Memory

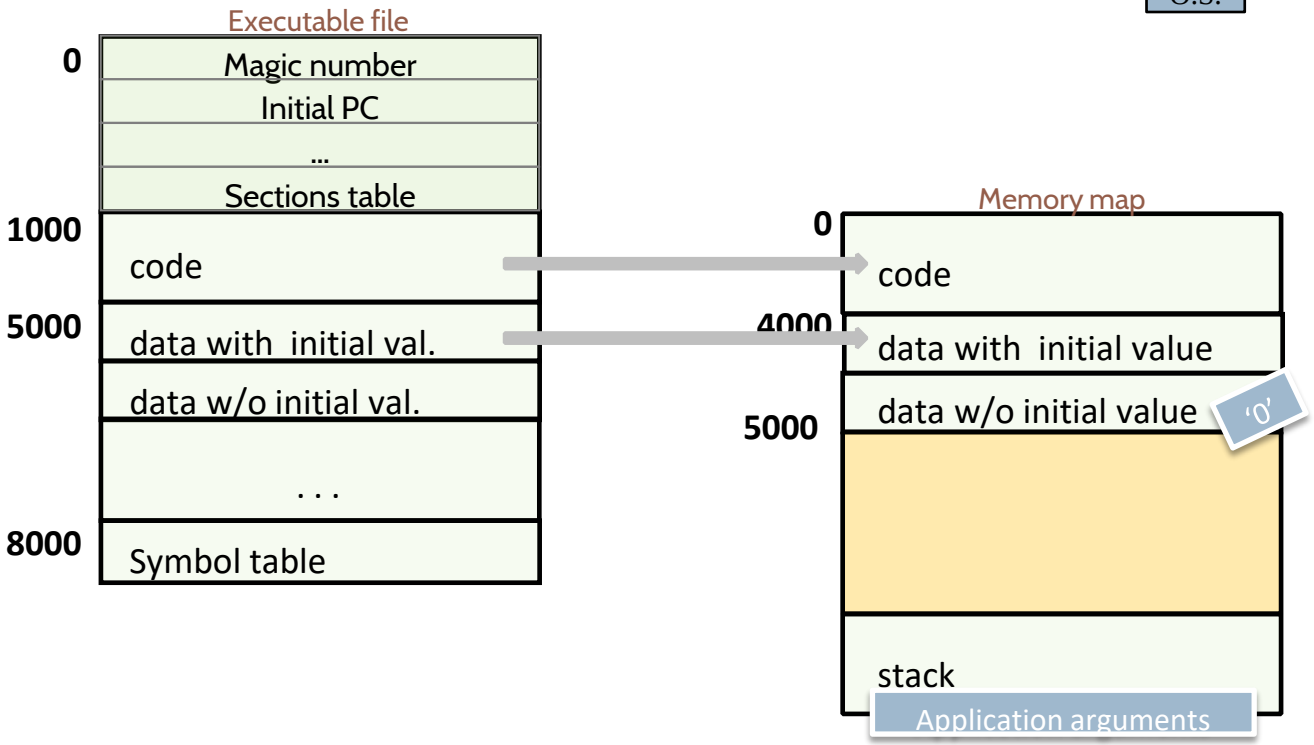


Example of executable file format

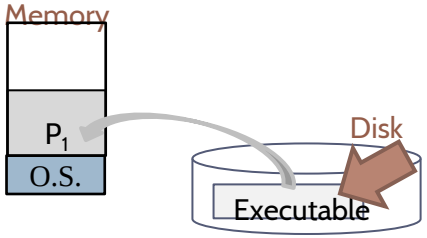


Build memory map from exe.





Inspect an executable

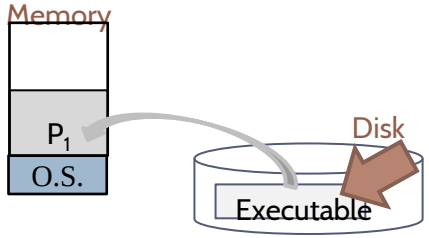


- Dependences of one executable (dynamic libraries):

```
acaldero@phoenix:~/infodso/$ ldd main.exe
        linux-gate.so.1 =>  (0xb7797000)
        libdinamica.so.1 => not found
        libc.so.6 => /lib/libc.so.6 (0xb761c000)
        /lib/ld-linux.so.2 (0xb7798000)
```

```
acaldero@phoenix:~/infodso/$ nm main.exe
08049f20 d _DYNAMIC
08049ff4 d _GLOBAL_OFFSET_TABLE_
0804856c R _IO_stdin_used
         w _Jv_RegisterClasses
08049f10 d __CTOR_END__
08049f0c d __CTOR_LIST__
...
```

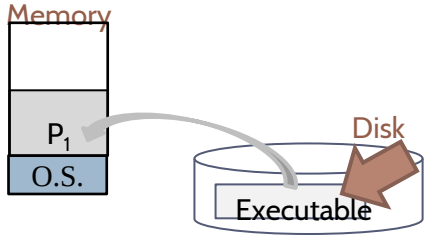
To inspect an executable



- Details of the executable sections:

```
acaldero@phoenix:~/infodso/$ objdump -x main.exe
...

Program Header:
...
 DYNAMIC off    0x00000f20 vaddr 0x08049f20 paddr 0x08049f20 align 2**2
         filesz 0x000000d0 memsz 0x000000d0 flags rw-
...
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x00000000 memsz 0x00000000 flags rw-
...

Dynamic Section:
  NEEDED               libdinamica.so
  NEEDED               libc.so.6
  INIT                 0x08048368
...
```

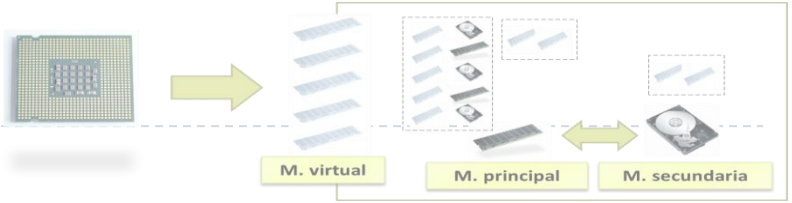
To inspect an executable



(contd.)

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  08048134  08048134  00000134  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 12 .text         0000016c  080483e0  080483e0  000003e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
...
 23 .bss          00000008  0804a014  0804a014  00001014  2**2
                  ALLOC
...
SYMBOL TABLE:
08048134 l    d  .interp              00000000           .interp
08048148 l    d  .note.ABI-tag        00000000           .note.ABI-tag
08048168 l    d  .note.gnu.build-id   00000000           .note.gnu.build-id
...
0804851a g     F .text  00000000              .hidden __i686.get_pc_thunk.bx
08048494 g     F .text  00000014              main
08048368 g     F .init  00000000              _init
```

Overview

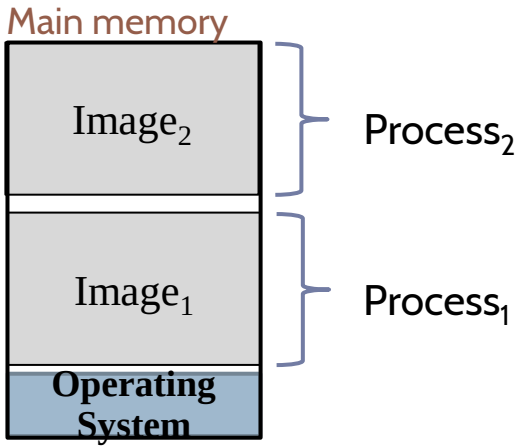


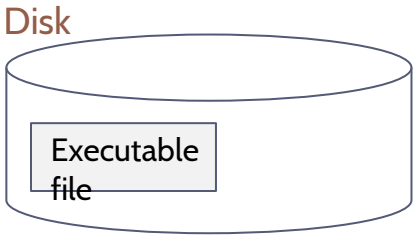
1. Introduction to memory usage
    1. Abstract model
    2. Definitions and environments
    3. Regions of process memory
    4. How to prepare an executable

1. **Introduction to Virtual Memory**

Process image

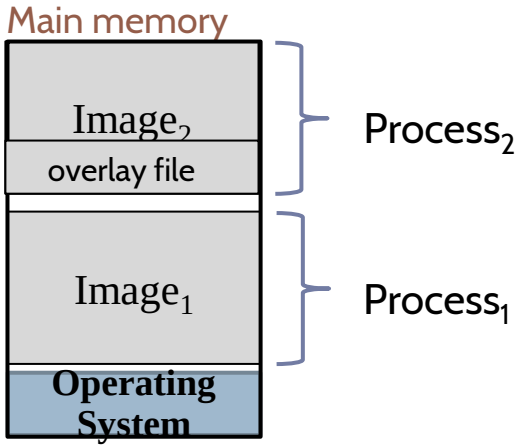
- The Operating System takes care of memory image parts:
  - Load/unload memory parts (only needed parts are in memory)
  - For multiprogramming + **out-of-core**

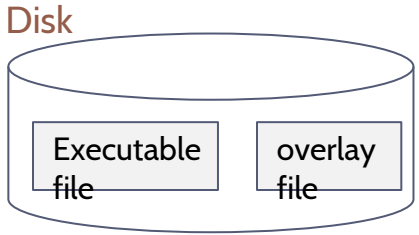




Process image

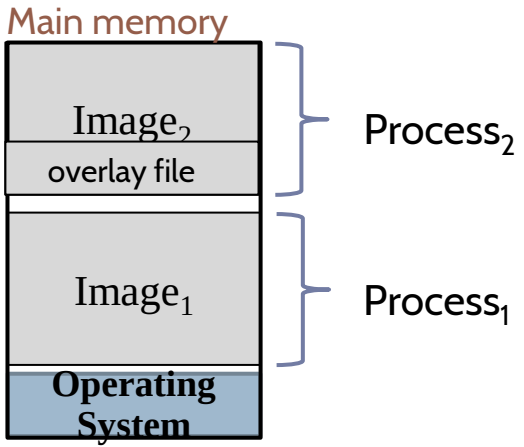
- Initially used the overlay mechanism:
  - Load/unload memory parts (only needed parts are in memory)
  - Programmer must manage the overlays of its processes

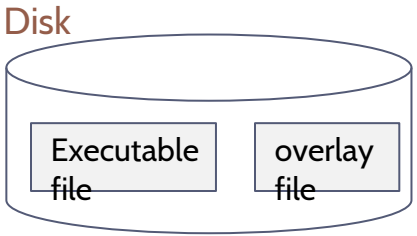




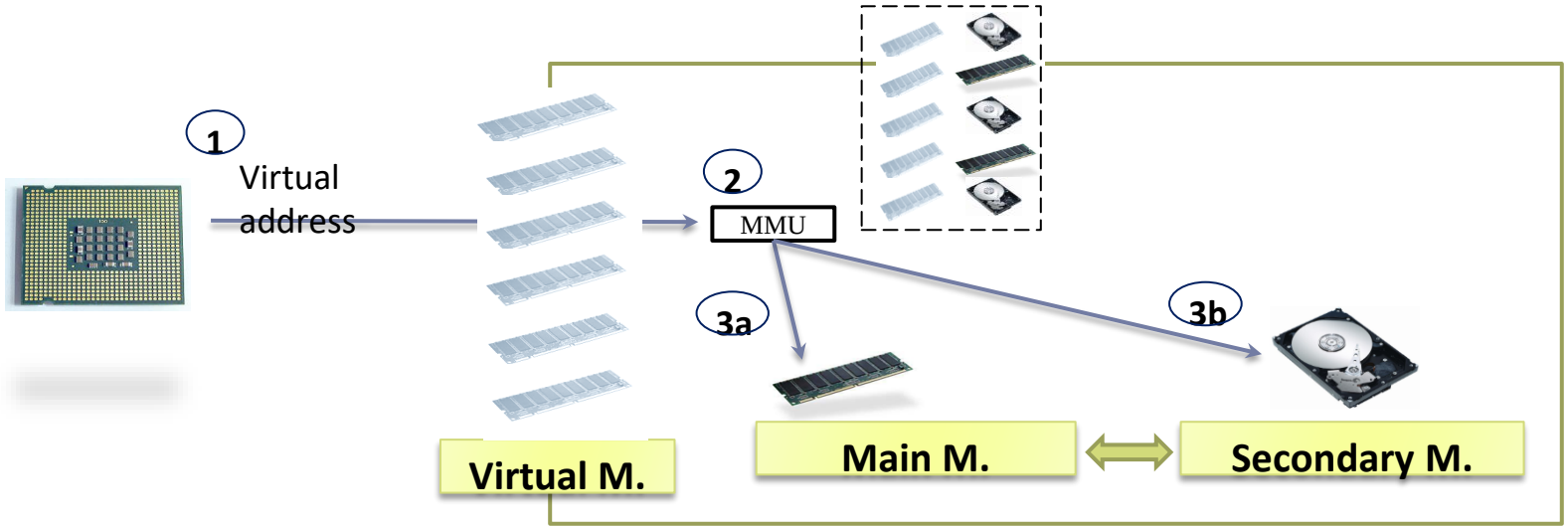
Process image

- CPU+O.S. offers an alternative mechanism to overlay:
  - Virtual Memory: only needed fragments are kept in memory
  - Transparent to programmer

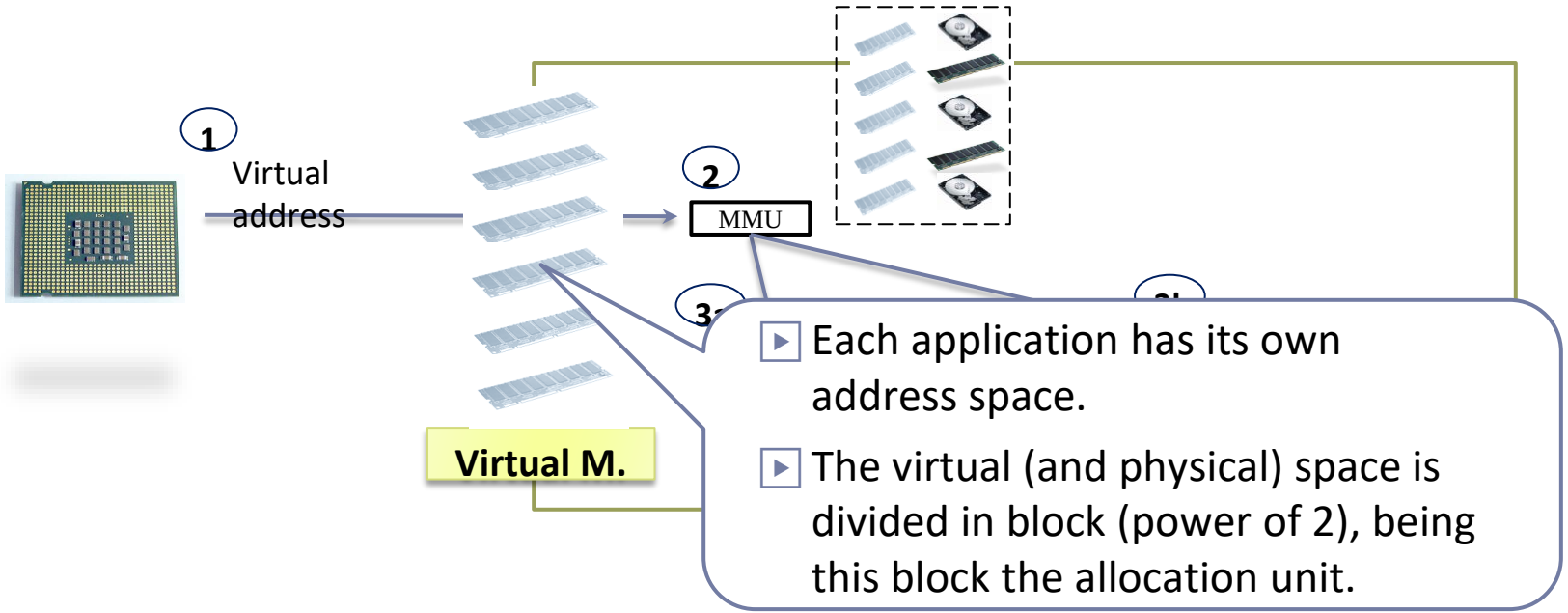




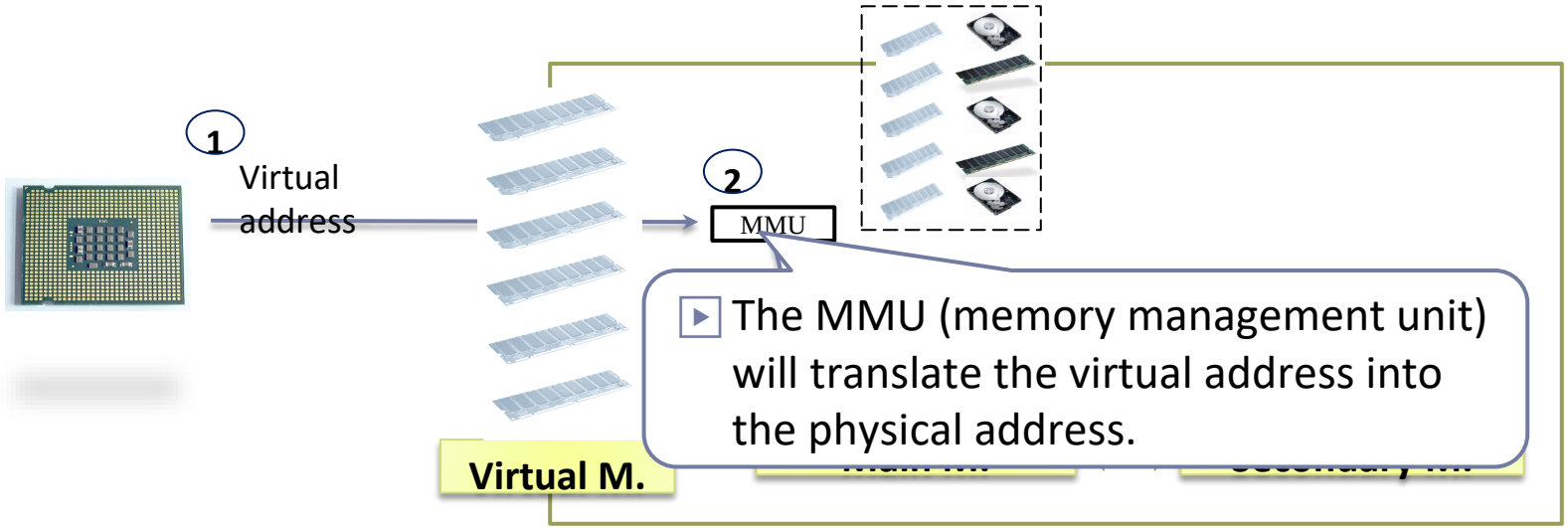
Virtual memory based systems



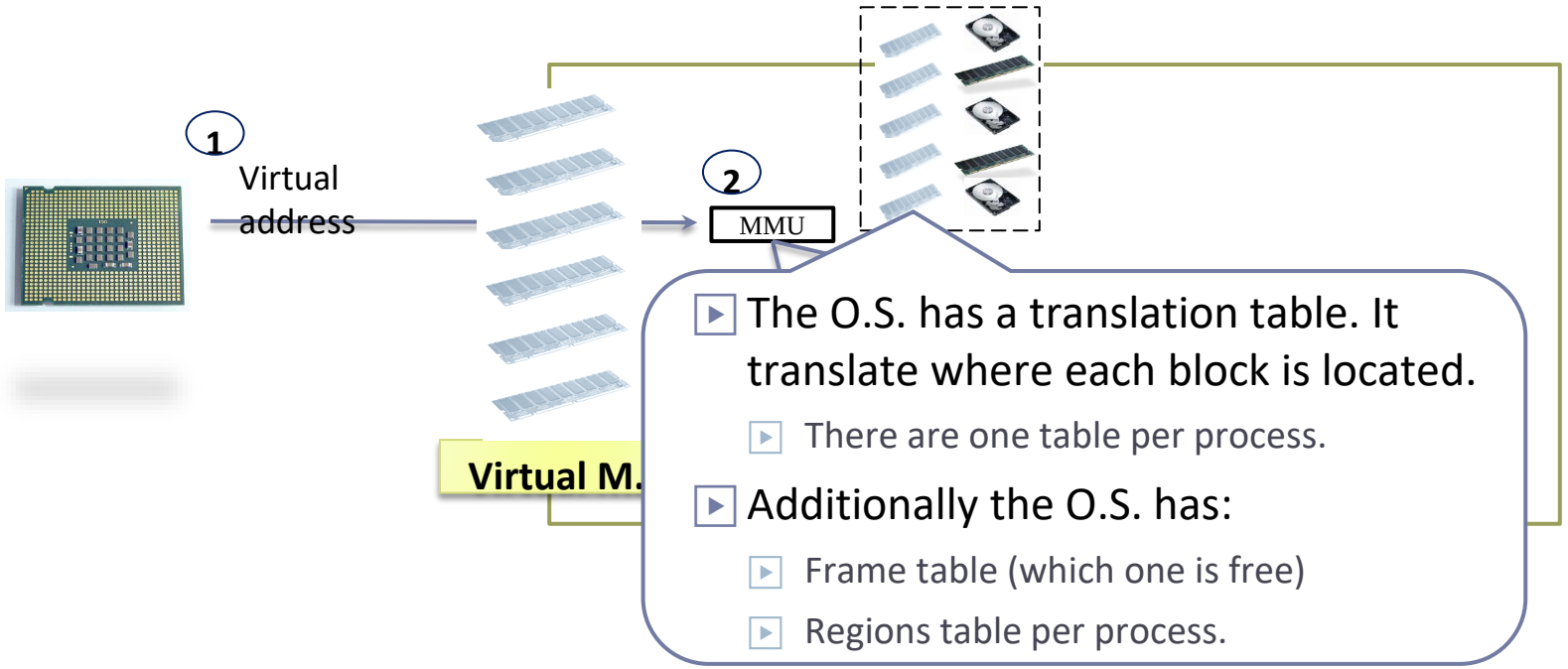
Virtual memory based systems



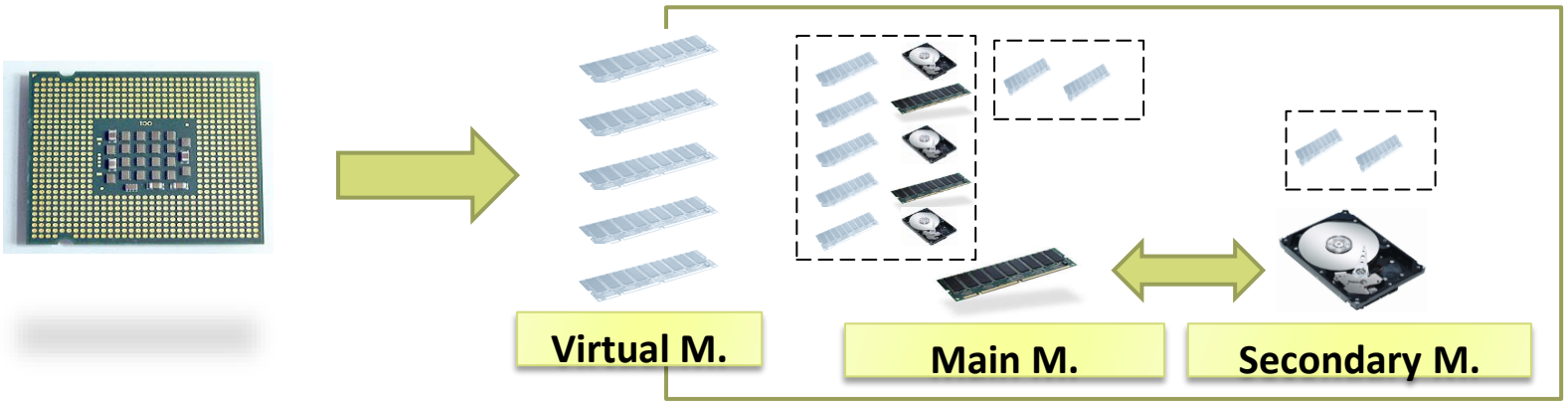
Virtual memory based systems



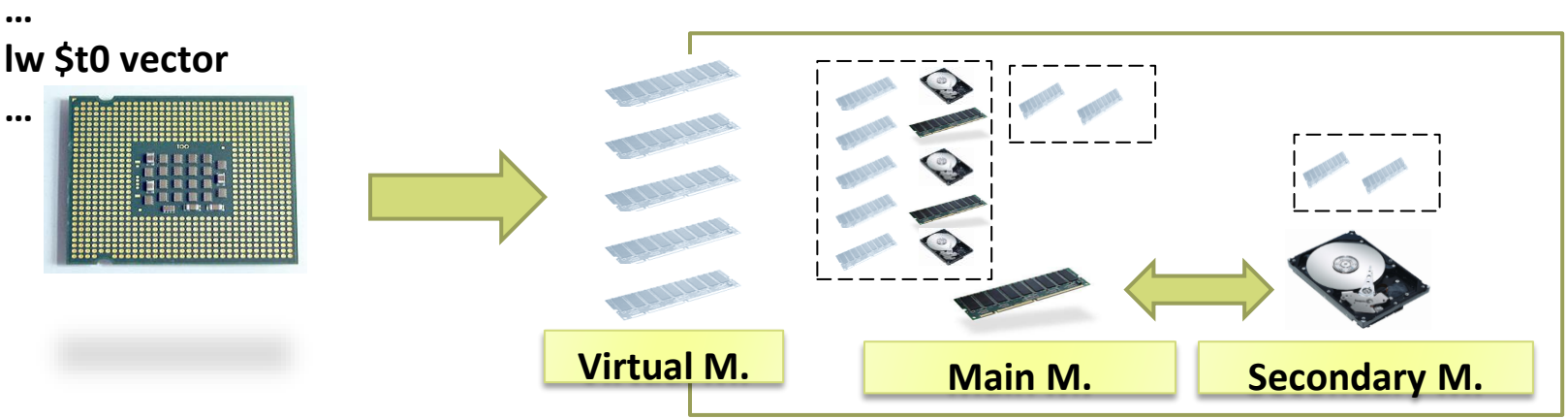
Virtual memory based systems



Virtual memory introduction

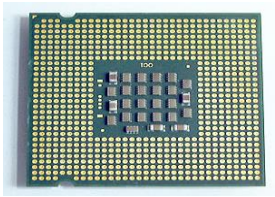


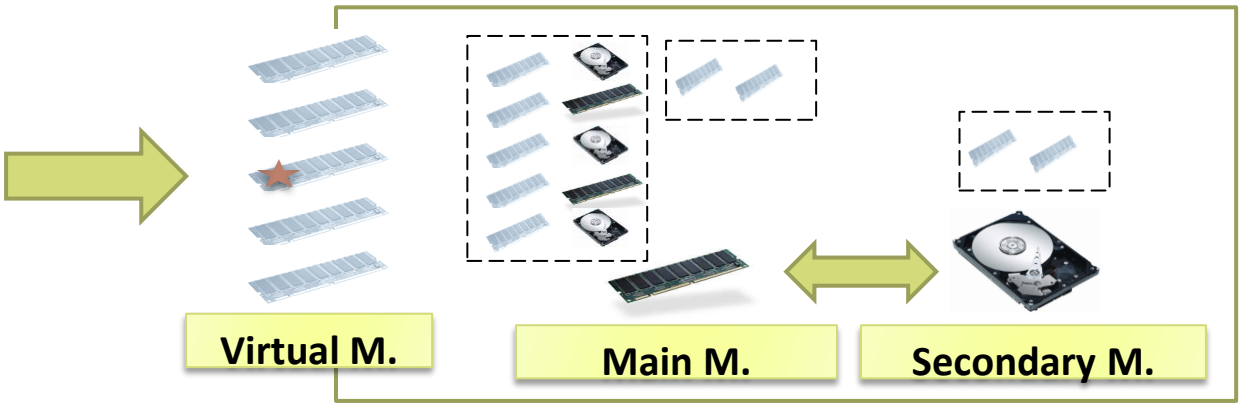
Virtual memory introduction



Virtual memory introduction

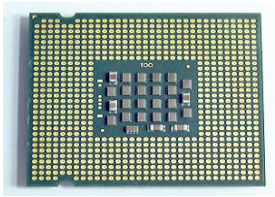
...
lw $t0 vector
...

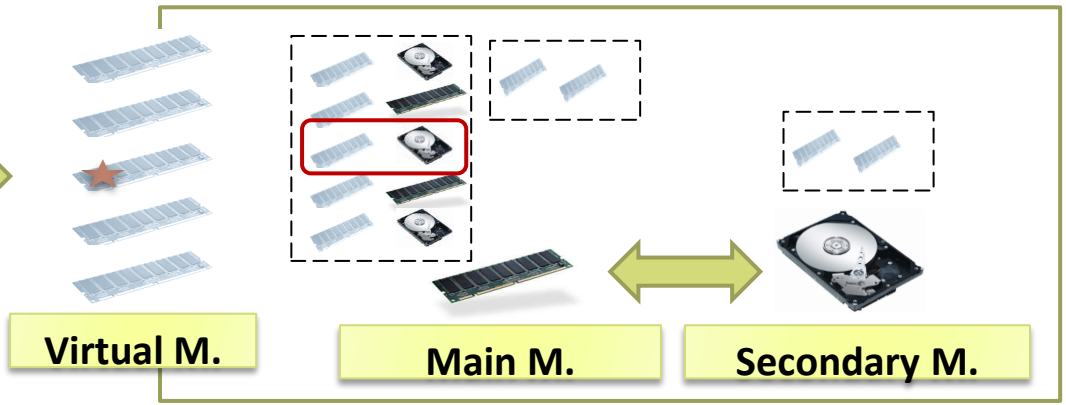




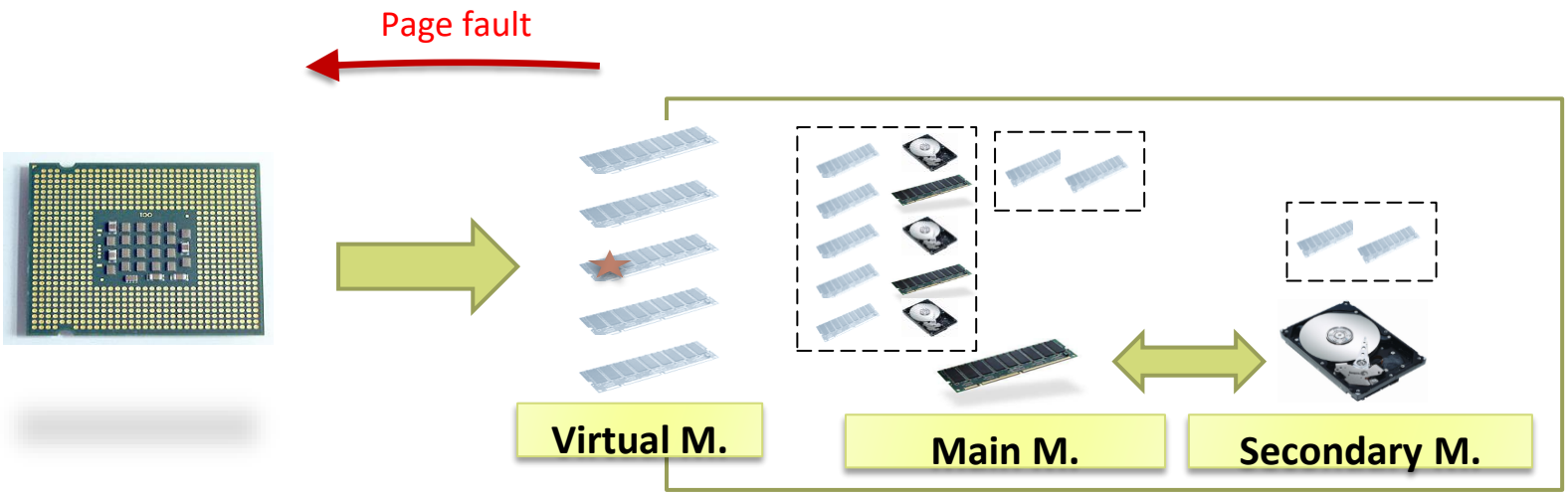
Virtual memory introduction

...
lw $t0 vector
...



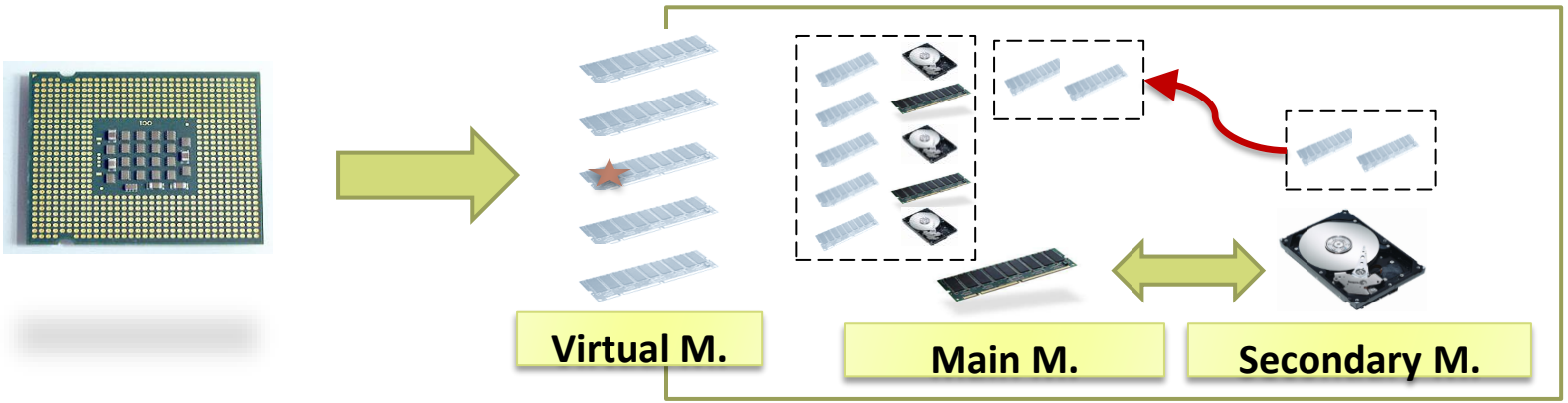


Virtual memory introduction



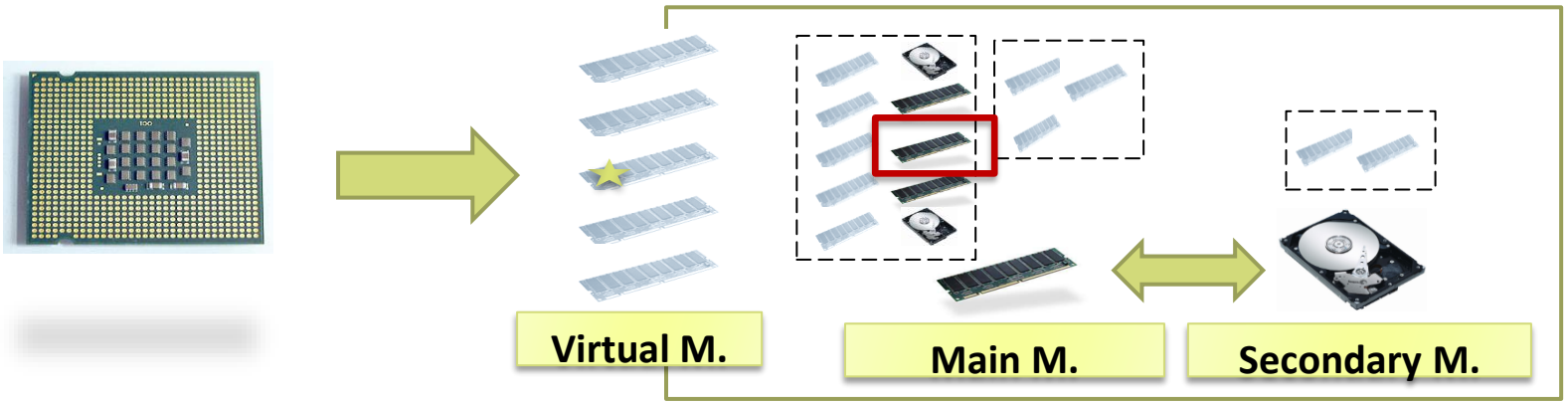
- The page fault is an exception that executes the associated handler in the O.S.
- The handler requests the associated disk blocks, and blocks the process.

Virtual memory introduction



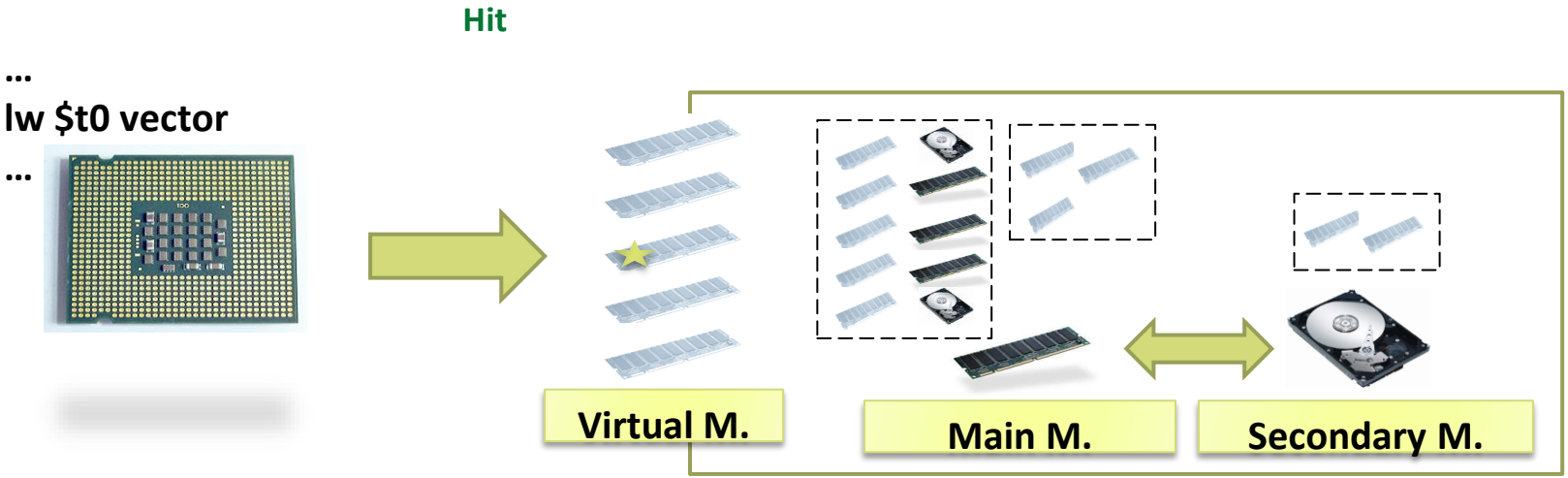
- The disk hardware interruption handler transfers the requested blocks into main memory, and fire a disk software interruption.

Virtual memory introduction



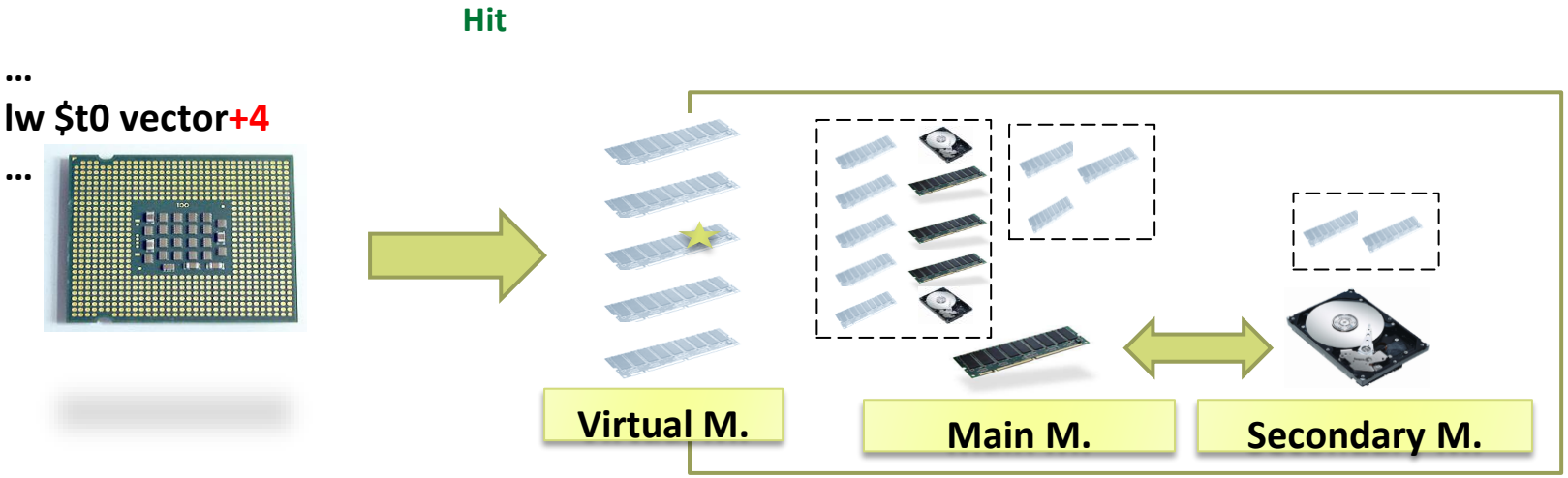
- The disk software interruption updates the ‘block’ table, and turns the process’ status into ready for execution.

Virtual memory introduction



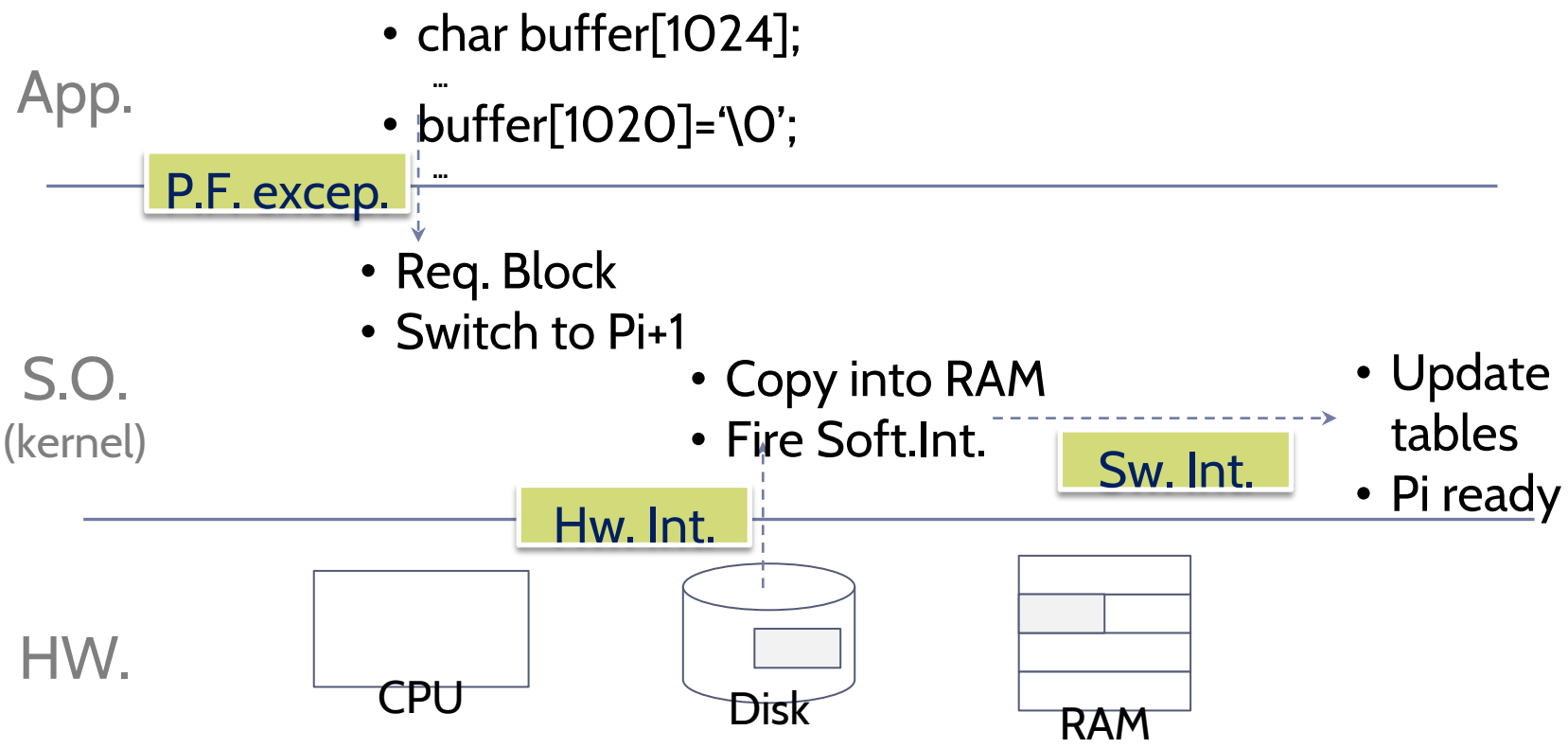
- It is resumed the execution of the instruction that leads to the page fault.

Virtual memory introduction



- The next instruction from the same block will not lead to page fault.

Summary of page fault exception



Virtual memory introduction

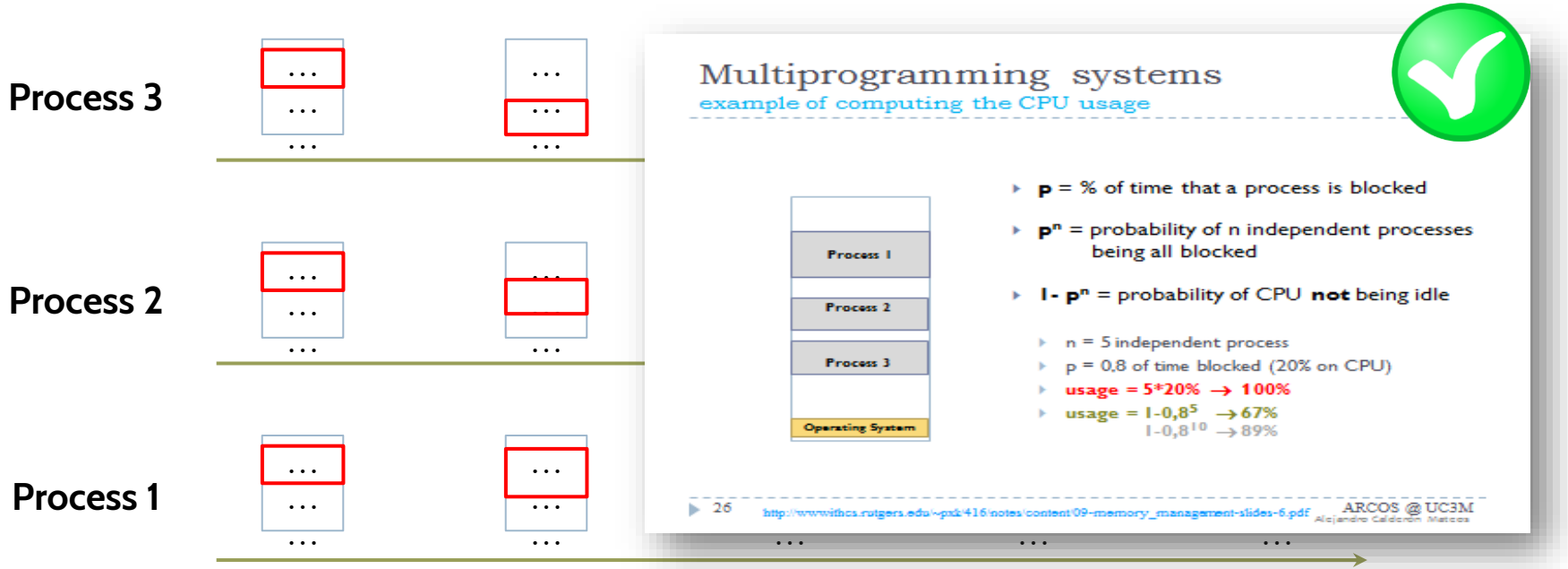
Workspace and multiprogramming

Virtual memory introduction

Workspace and multiprogramming

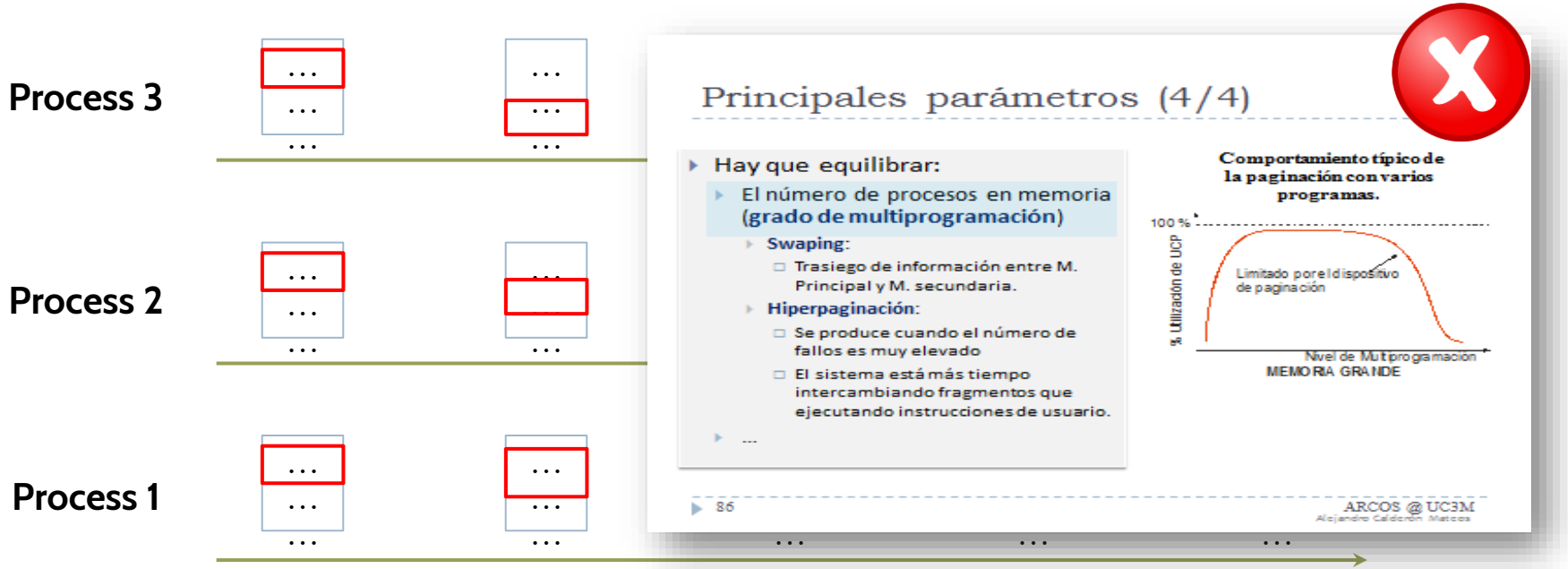
Virtual memory introduction

Workspace and multiprogramming

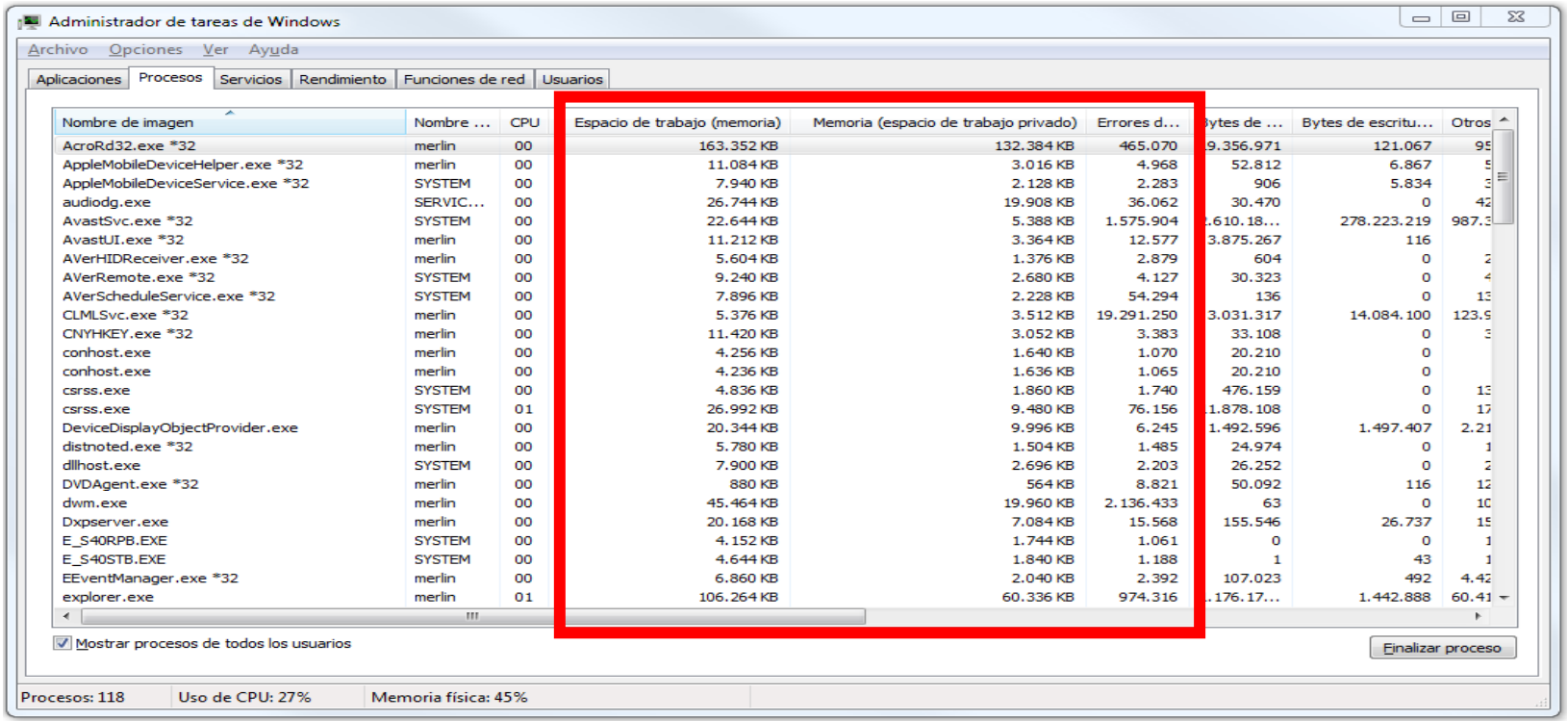


Virtual memory introduction

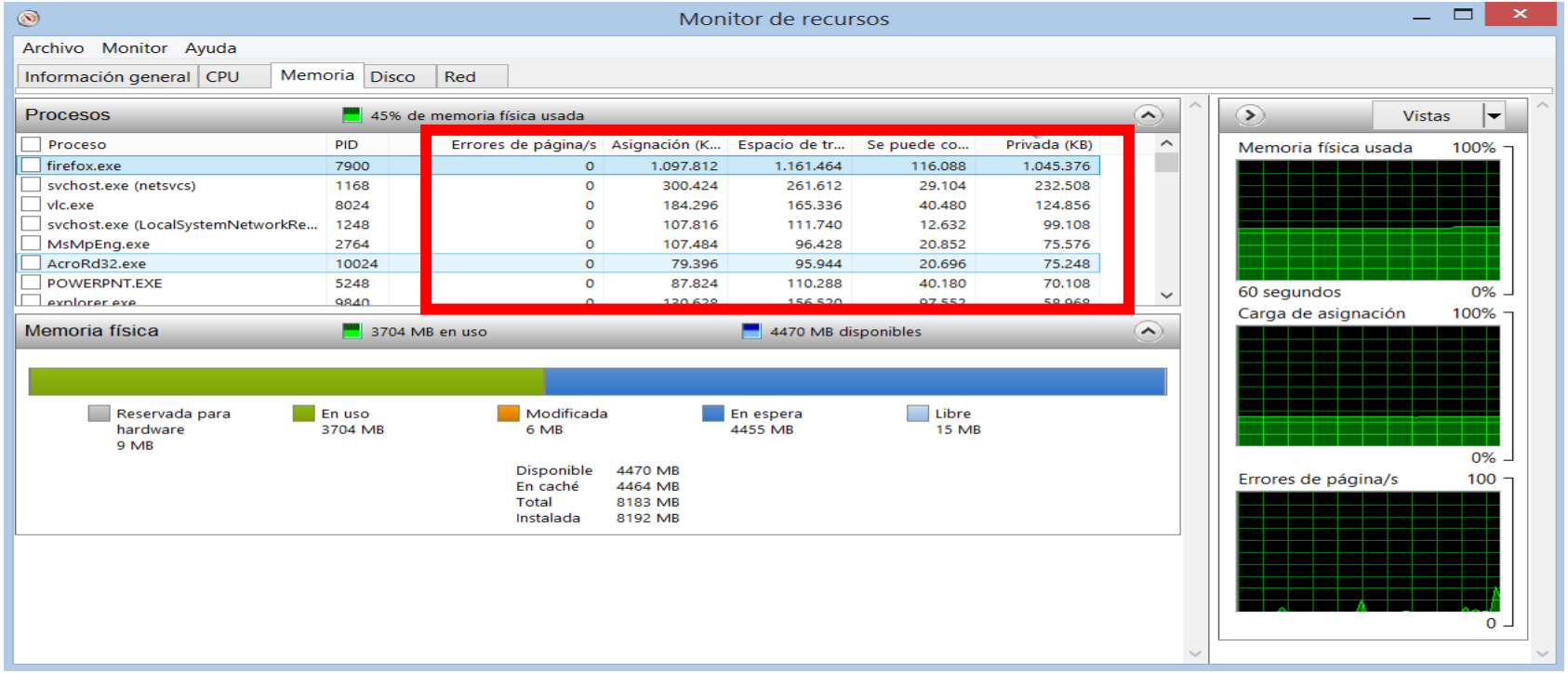
Workspace and multiprogramming



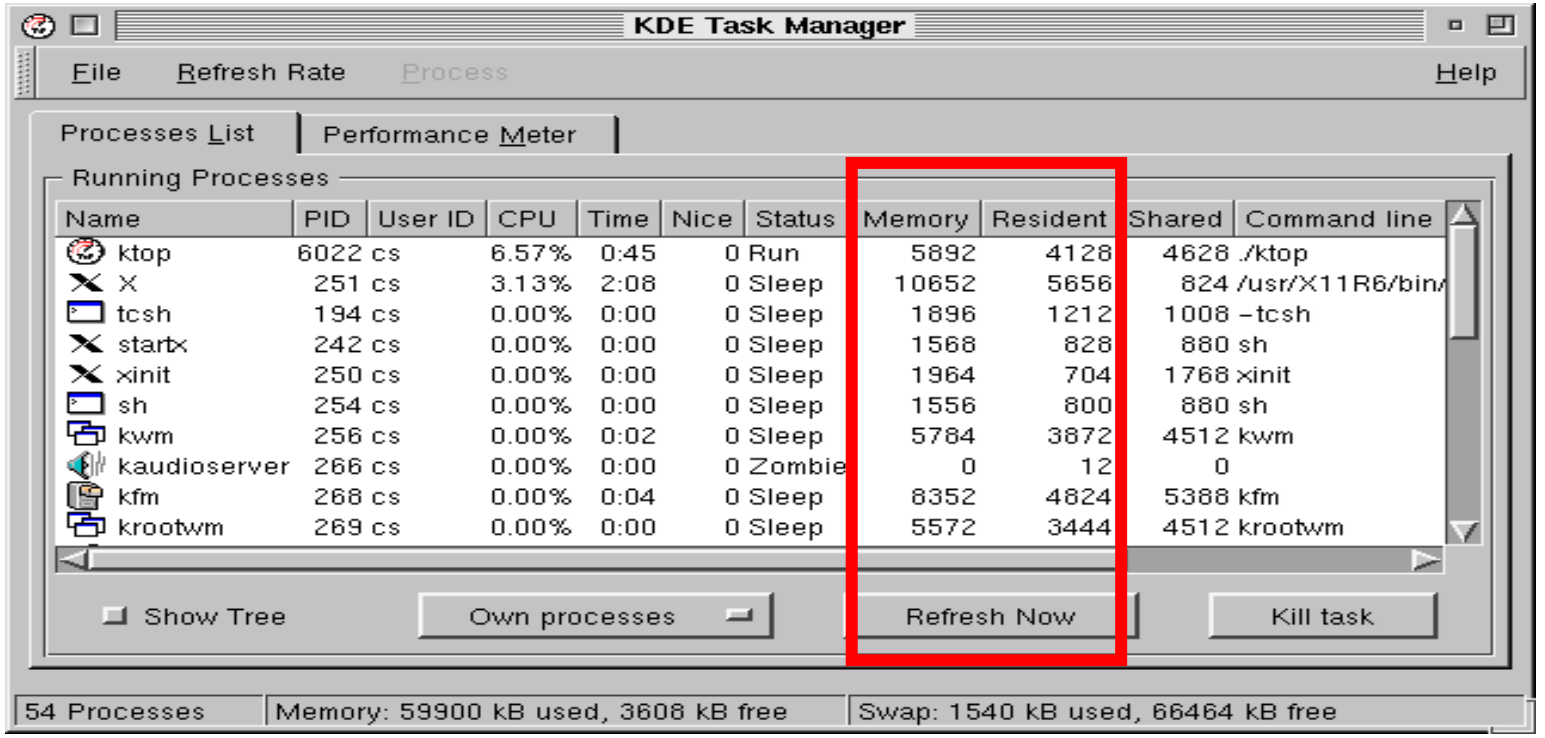
Virtual memory: Windows 7



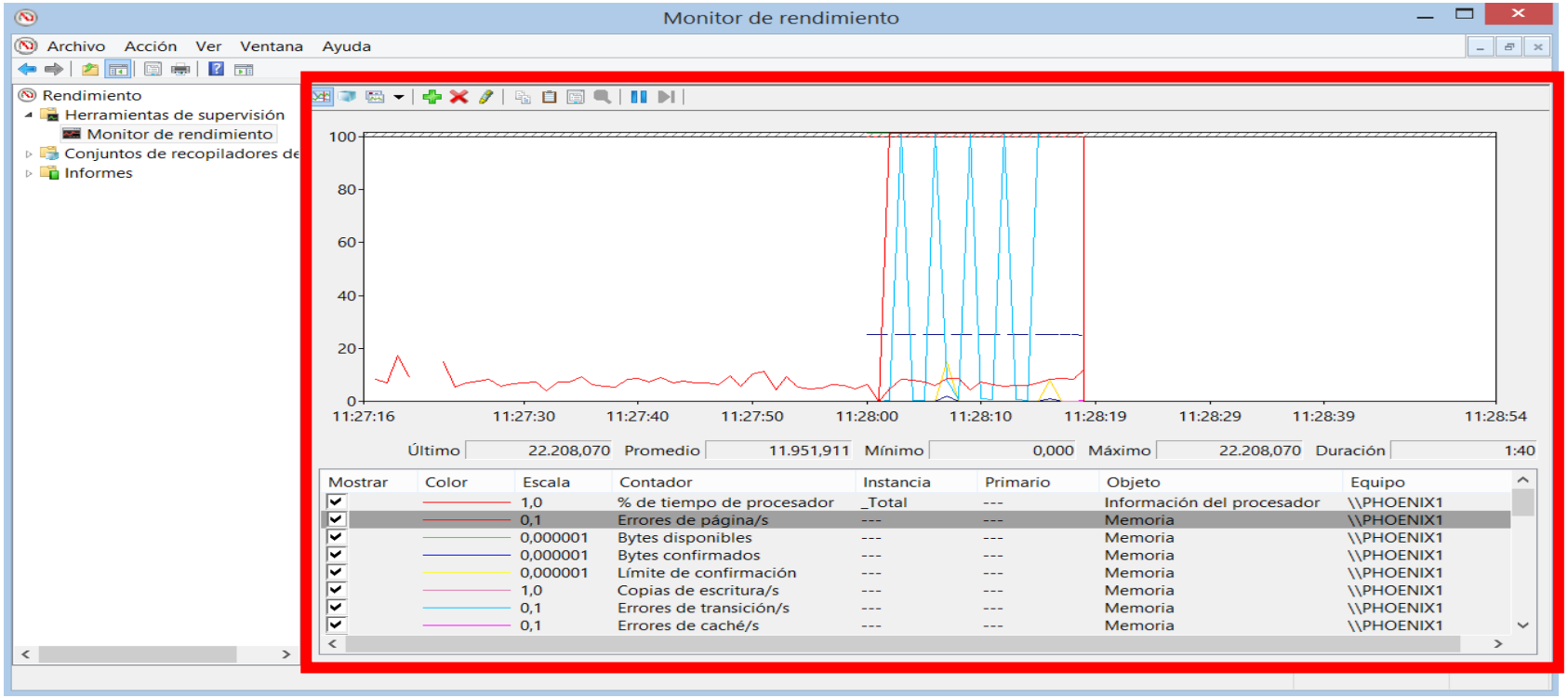
Virtual memory: Windows 8.x



Virtual memory: Linux



Windows: perfmon



http://technet.microsoft.com/en-us/library/cc958290.aspx

Linux: ps, top, ...

```
arcos:~$ ps -o min_flt,maj_flt 1
  MINFL  MAJFL
  18333     25
```

Minor fault: it bookings a page

Major fault: it needs to access to Disk

http://wwwithcyberciti.biz/faq/linux-command-to-see-major-minor-pagefaults/

Linux: ps, top, ...

```
arcos:~$ vmstat 1 5
procs   -----------memory----------  ---swap-- -----io---- -system-- ----cpu-----
 r  b   swpd   free    buff   cache    si   so    bi    bo   in   cs us sy  id wa
 1  0    140 3092132 1575132 2298820    0    0    12    19   20   32  1  2  97  0
 0  0    140 3092124 1575132 2298820    0    0     0     0  128  250  0  0 100  0
 0  0    140 3092124 1575132 2298820    0    0     0    16  143  281  0  0 100  1
 0  0    140 3092124 1575132 2298820    0    0     0     0  137  247  0  0 100  0
 0  0    140 3092124 1575132 2298820    0    0     0     0  138  270  0  0 100  0
```

Procs
- **r**: The number of processes waiting for run time.
- **b**: The number of processes in uninterruptible sleep.

Memory
- **swpd**: the amount of virtual memory used.
- **free**: the amount of idle memory.
- **buff**: the amount of memory used as buffers.
- **cache**: the amount of memory used as cache.
- **inact**: the amount of inactive memory. (-a option)
- **active**: the amount of active memory. (-a option)

Swap
- **si**: Amount of memory swapped in from disk (/s).
- **so**: Amount of memory swapped to disk (/s).

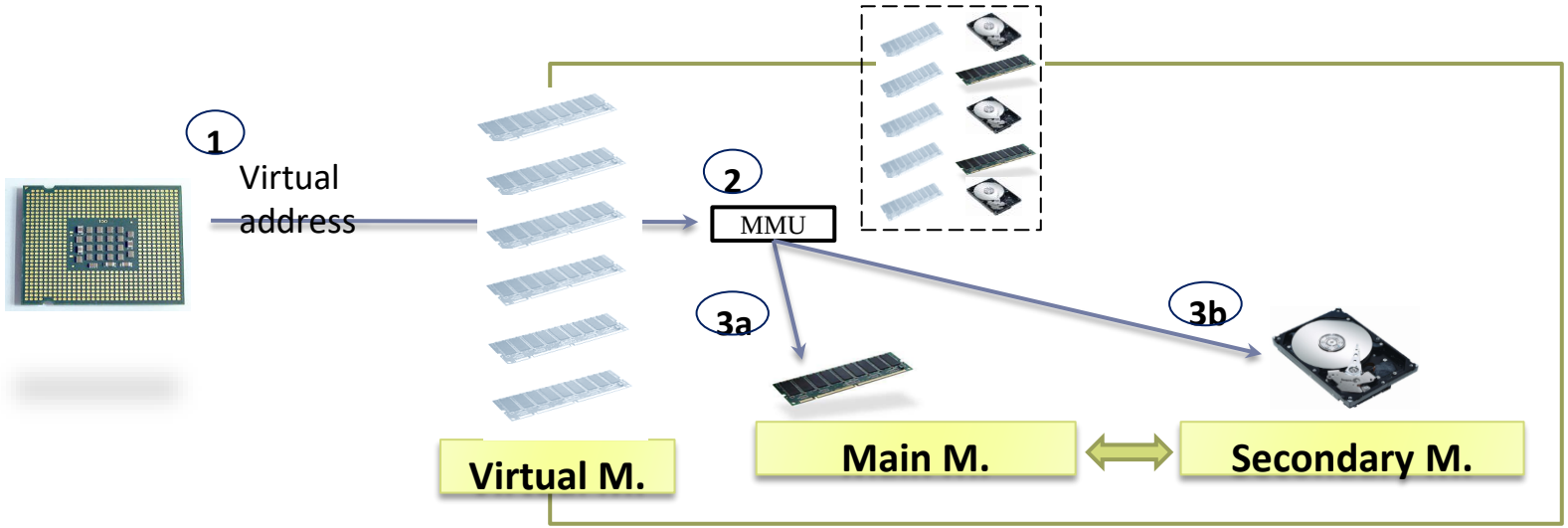
IO
- **bi**: Blocks received from a block device (blocks/s).
- **bo**: Blocks sent to a block device (blocks/s).

System
- **in**: The number of interrupts per second, including the clock.
- **cs**: The number of context switches per second.

CPU
These are percentages of total CPU time.
- **us**: Time spent running non-kernel code. (user time, including nice time)
- **sy**: Time spent running kernel code. (system time)
- **id**: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
- **wa**: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
- **st**: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.

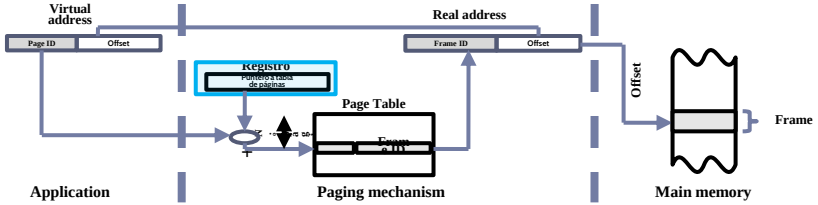
http://wwwithcyberciti.biz/faq/linux-command-to-see-major-minor-pagefaults/

Virtual memory based systems

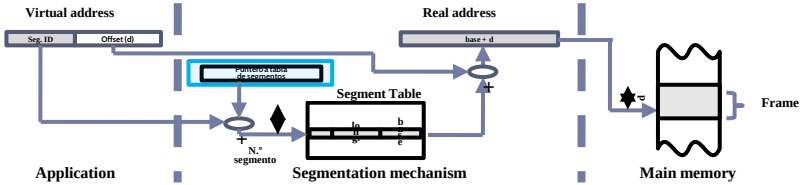


Virtual Memory

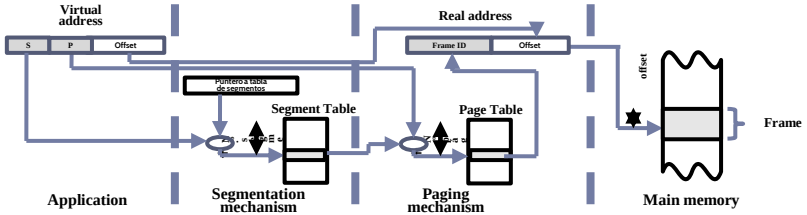
- Paging



- Segmentation

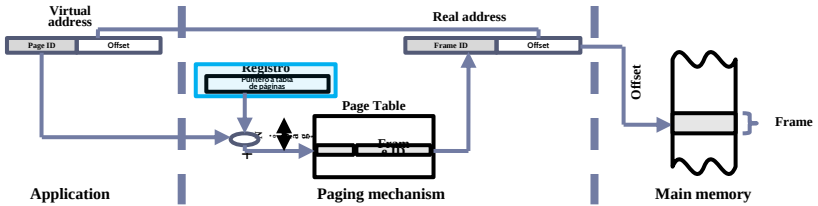


- Segmentation w. paging

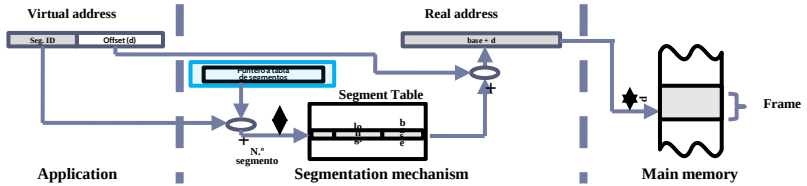


Virtual Memory

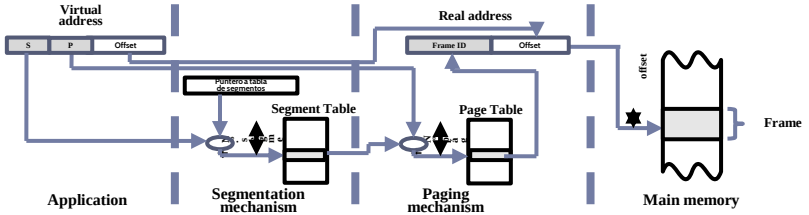
- Paging



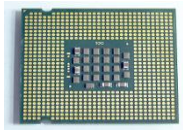
- Segmentation



- Segmentation w. paging



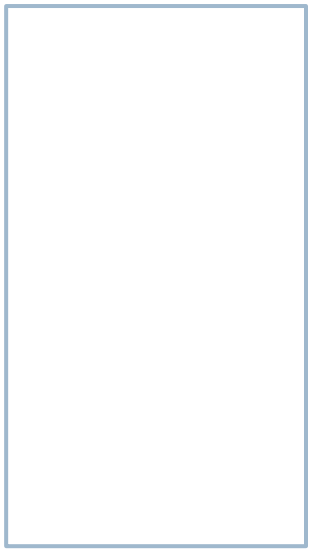
Example



32 bits

011010001001010

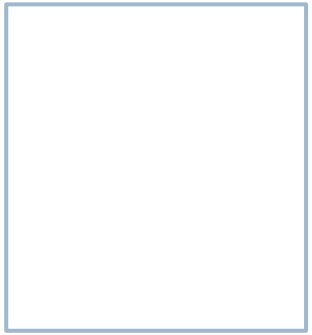
Virtual address

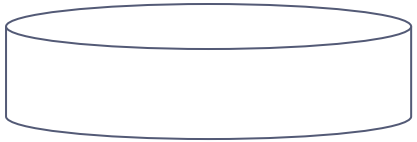


22 bits

1111100100000

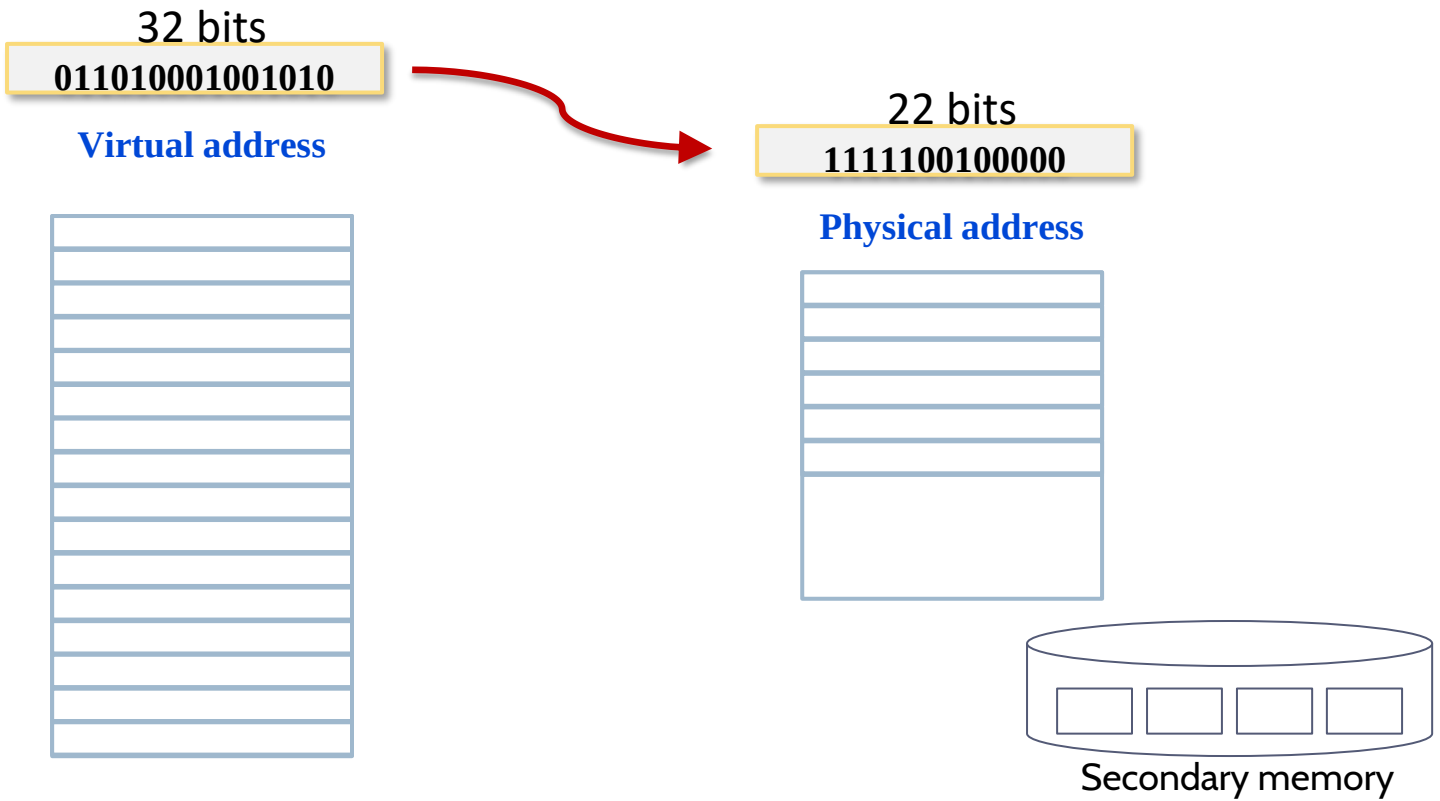
Physical address





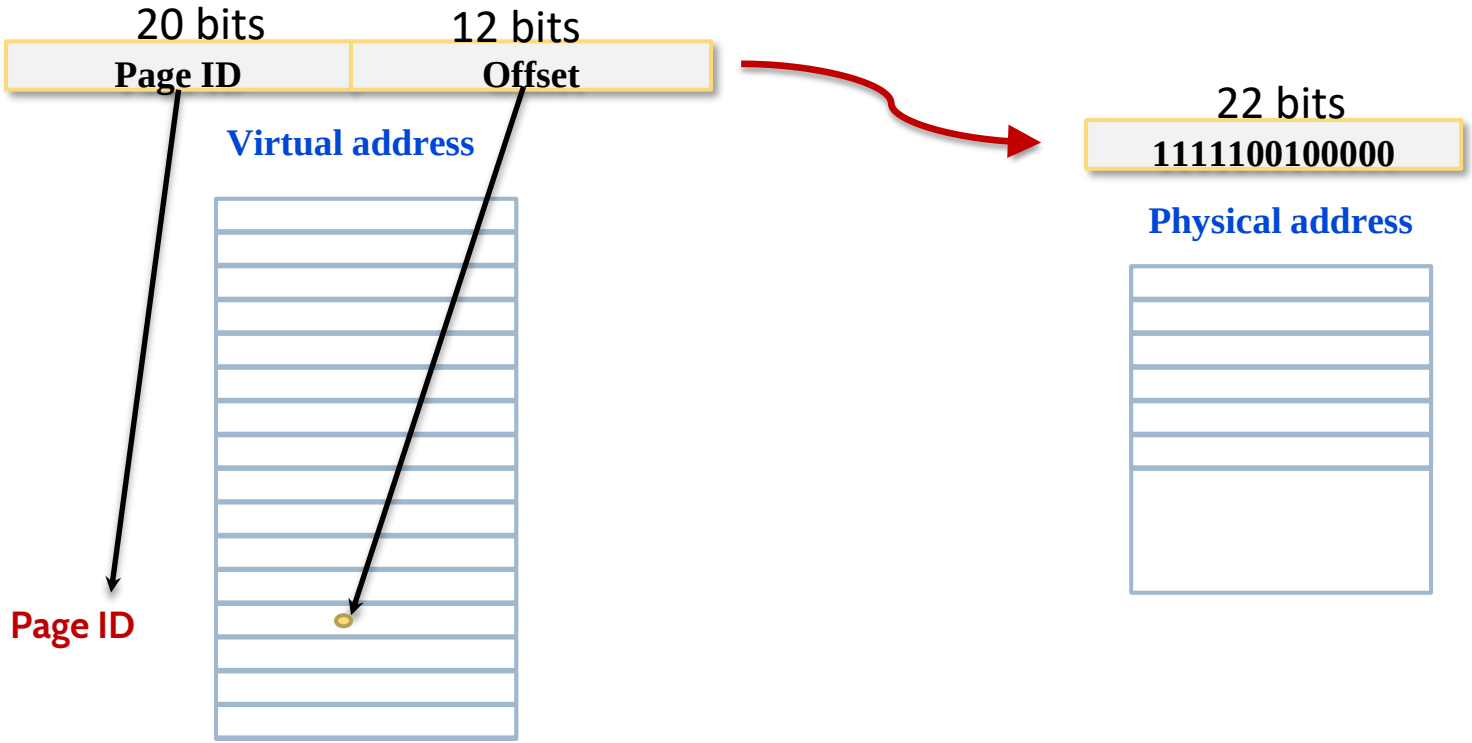
Secondary memory

Example



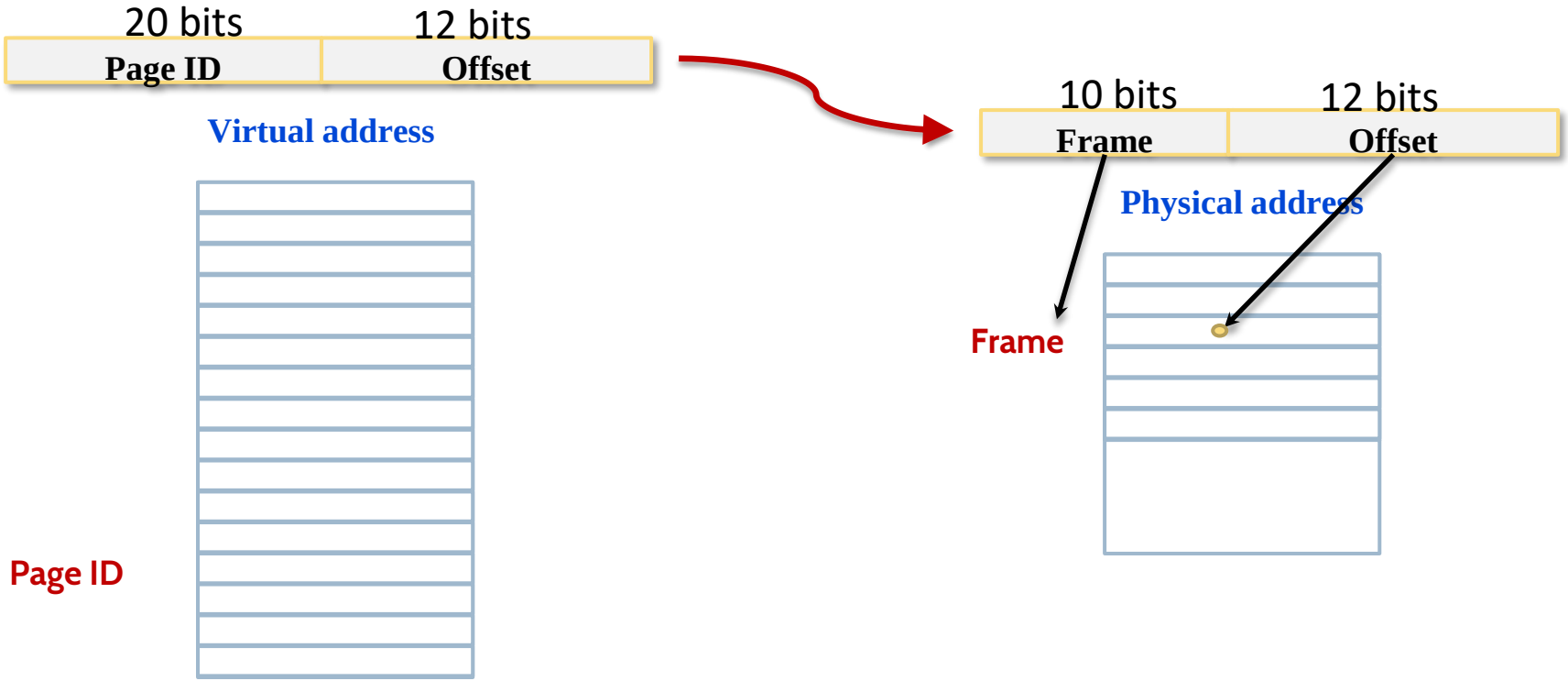
Divided into equal sized blocks -> pages

Example



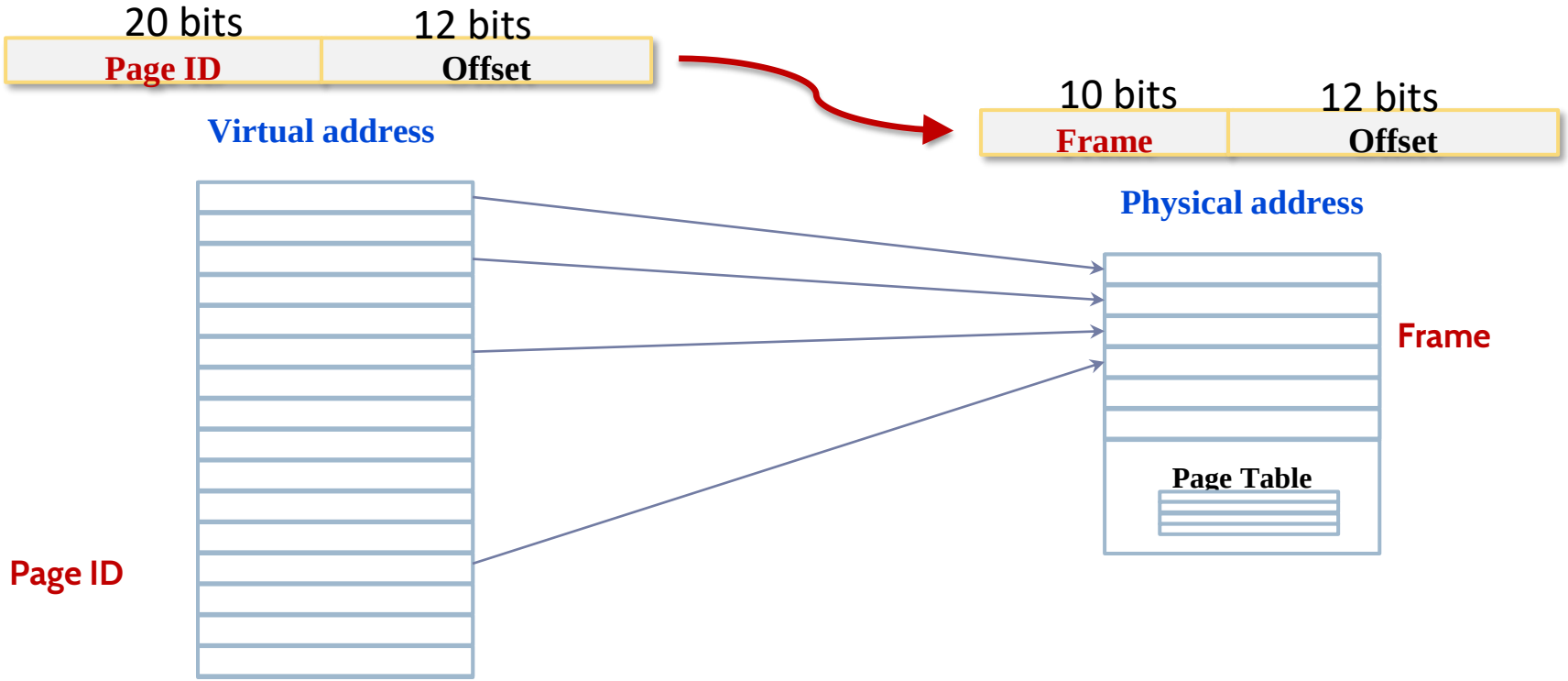
Divided into equal sized blocks -> pages

Example



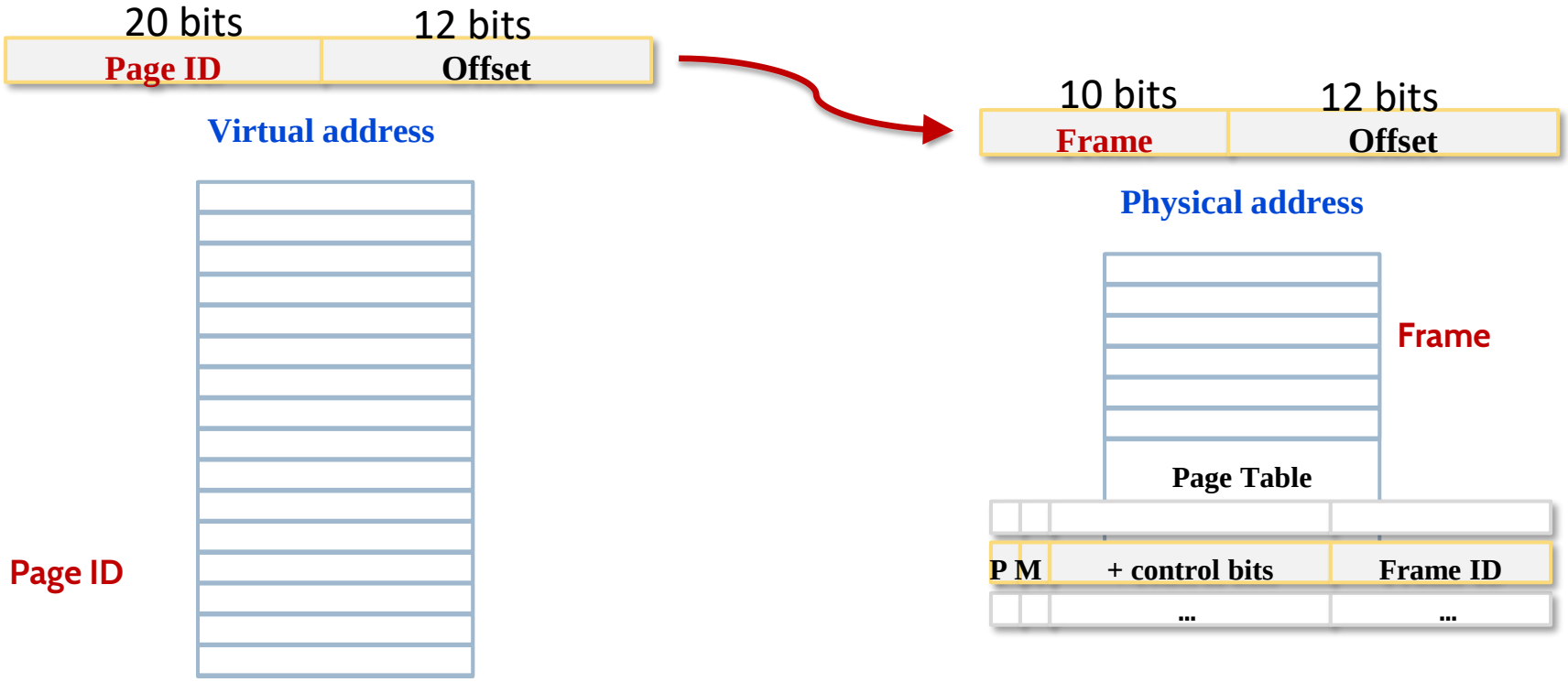
Divided into equal sized blocks -> pages

Example



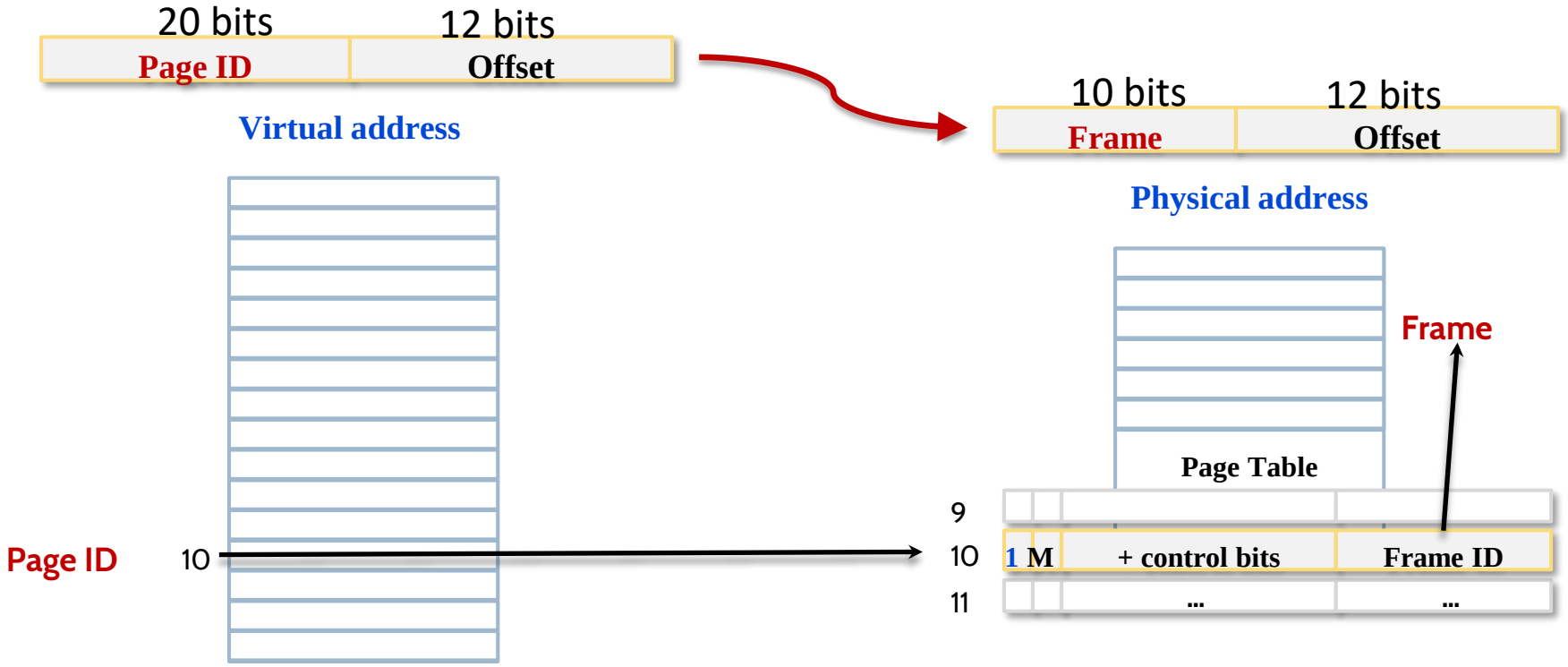
Translation from Page ID to Frame -> Page Table

Example



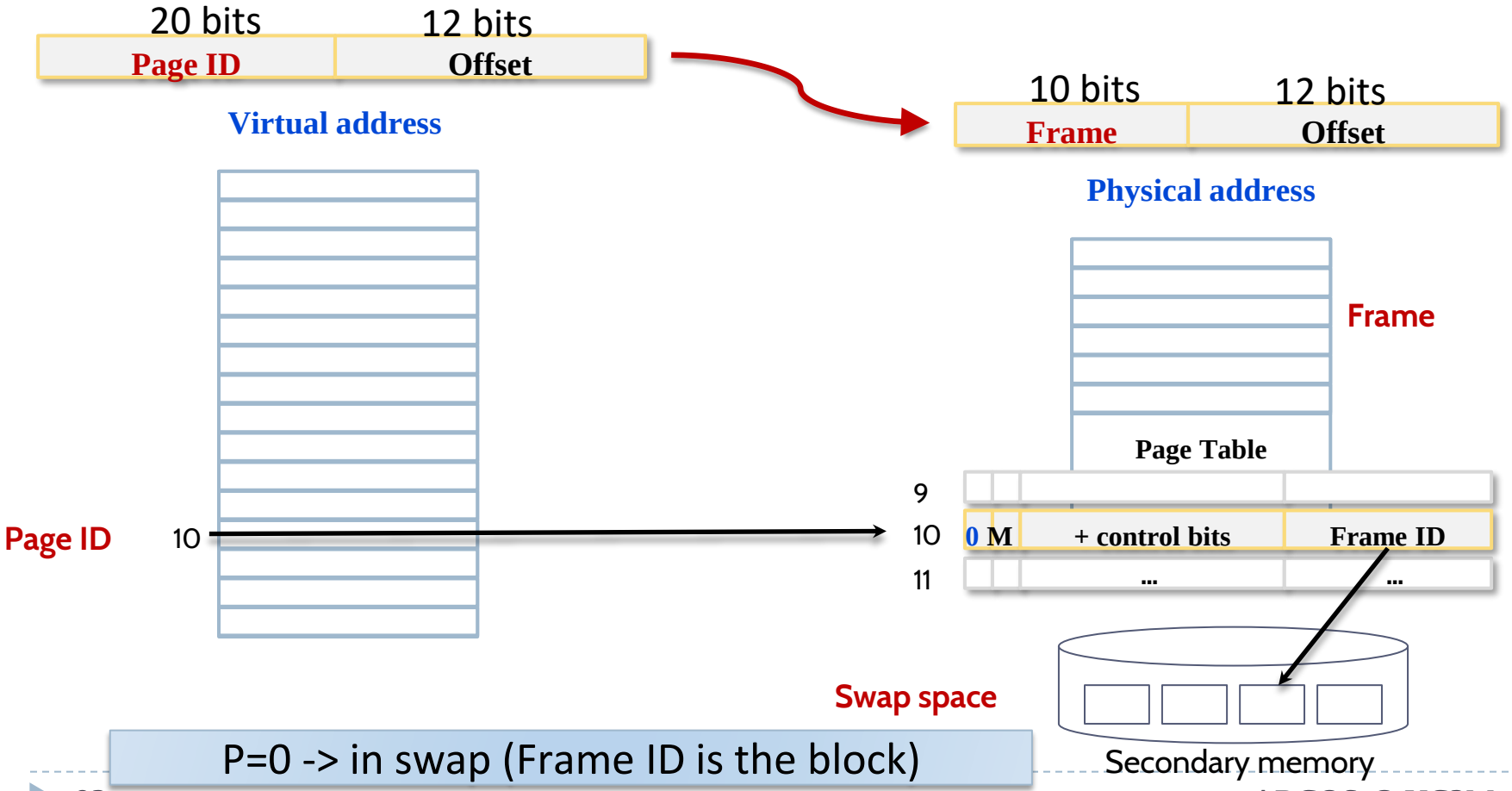
Translation from Page ID to Frame -> Page Table

Example



P=1 -> present in memory (Frame ID is where is it)

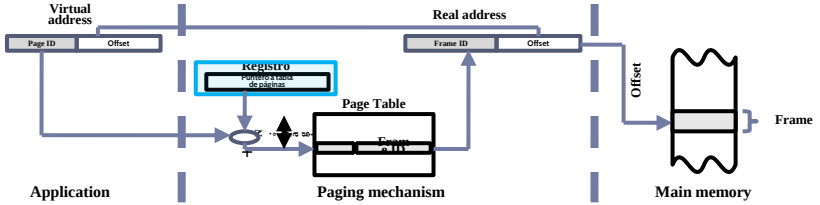
Example



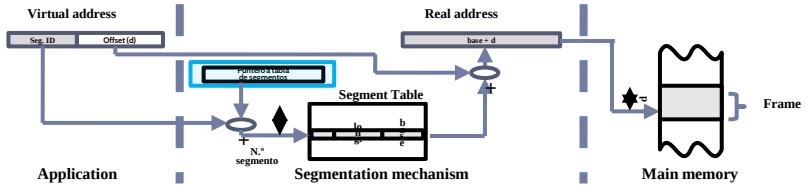
P=0 -> in swap (Frame ID is the block)

Virtual Memory

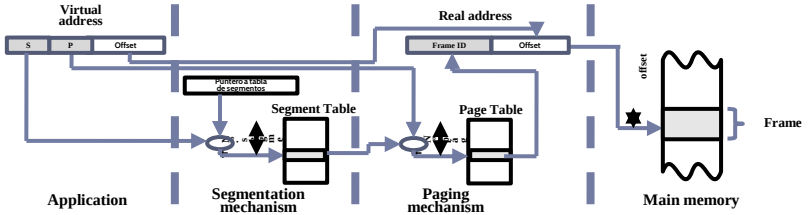
- Paging



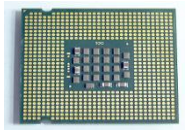
- Segmentation



- Segmentation w. paging



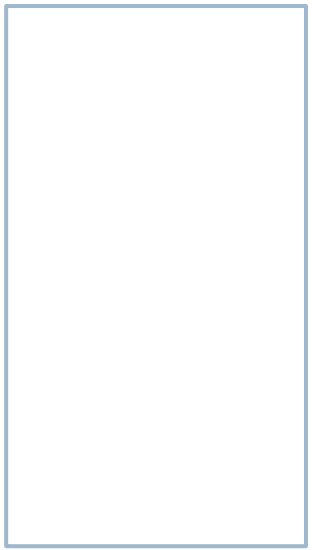
Example



32 bits

011010001001010

Virtual address

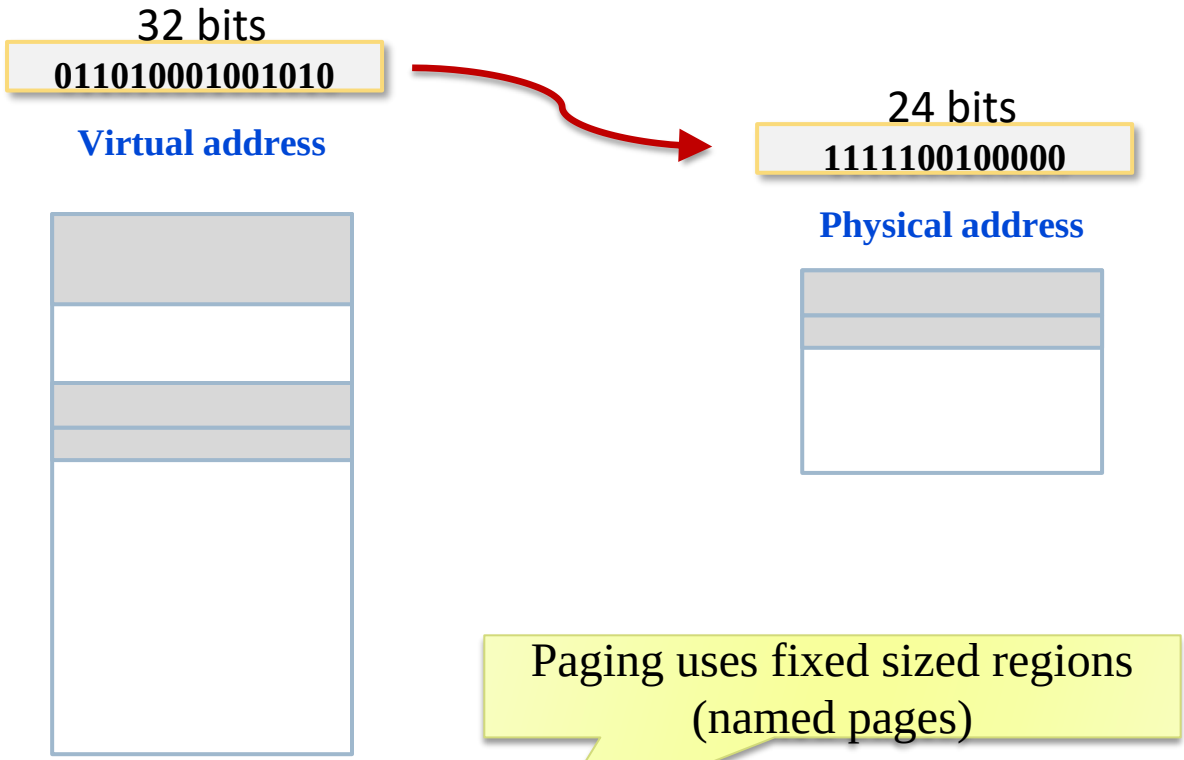


24 bits

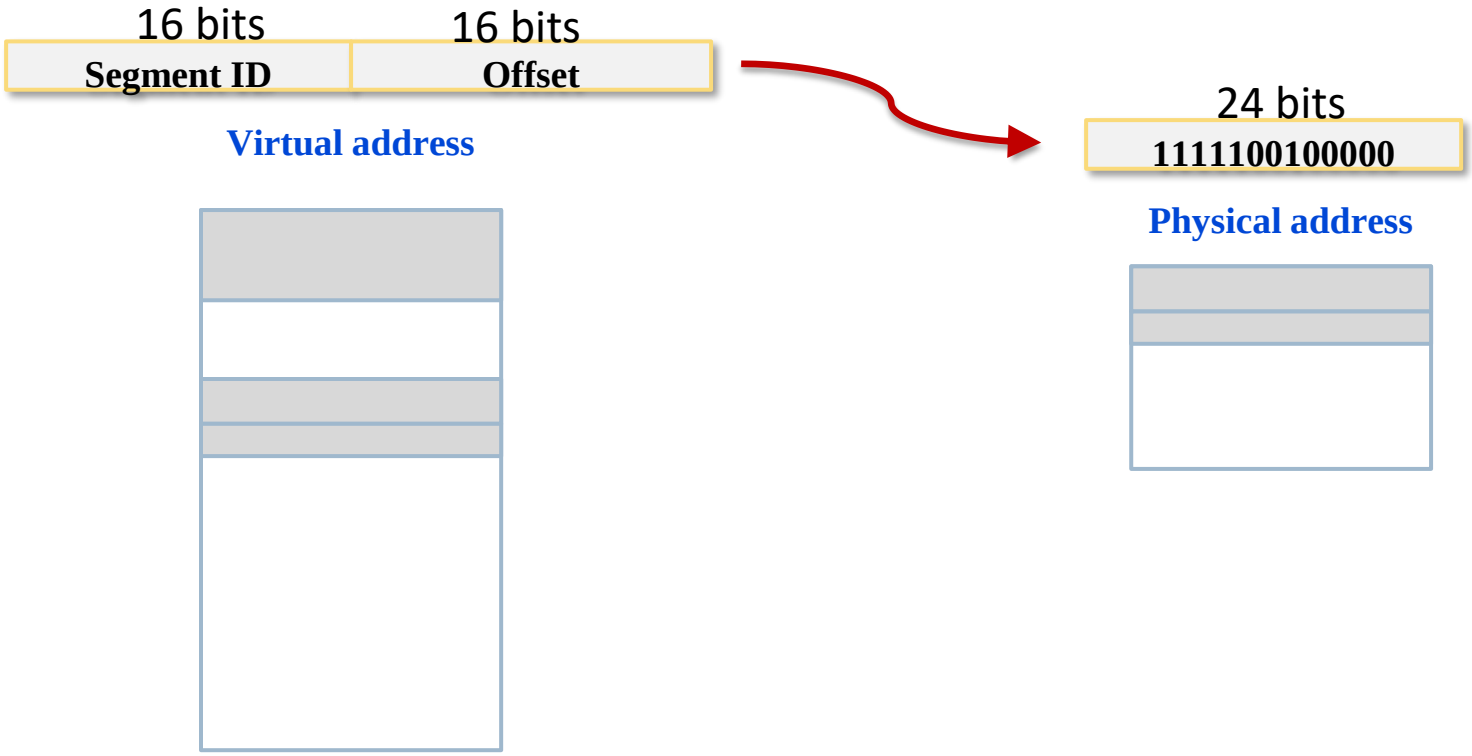
1111100100000

Physical address

Example

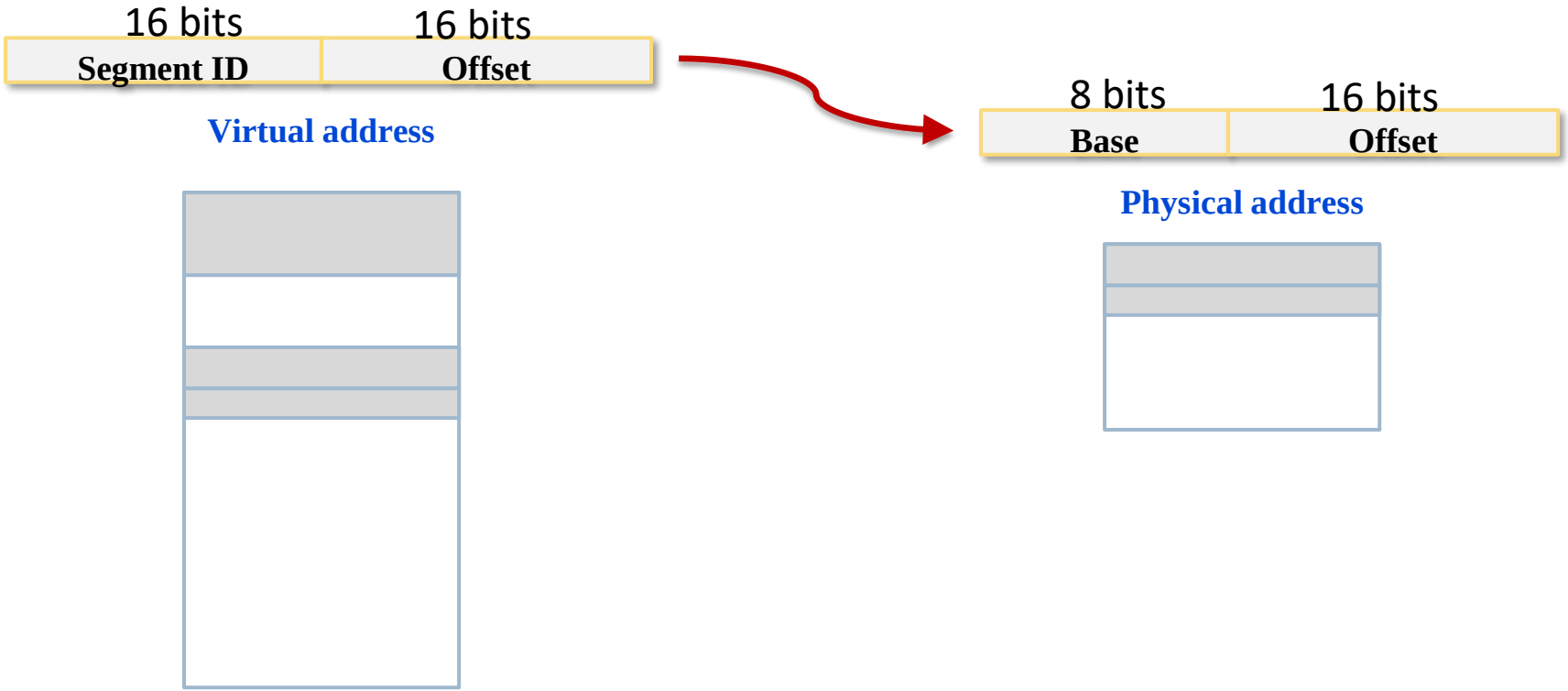

32 bits
011010001001010
Virtual address
24 bits
1111100100000
Physical address
Paging uses fixed sized regions
(named pages)
Variable sized regions -> segments

Example



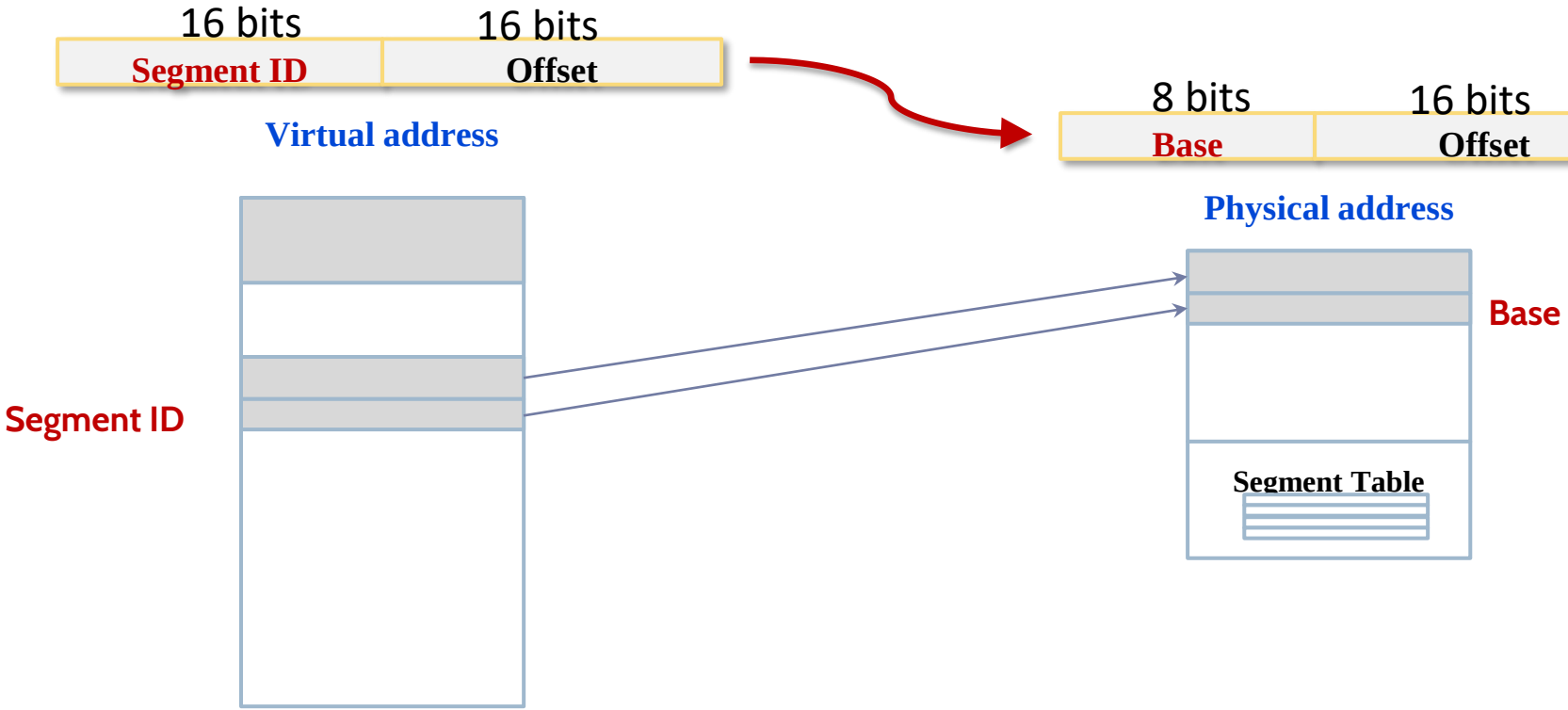
Variable sized regions -> segments

Example



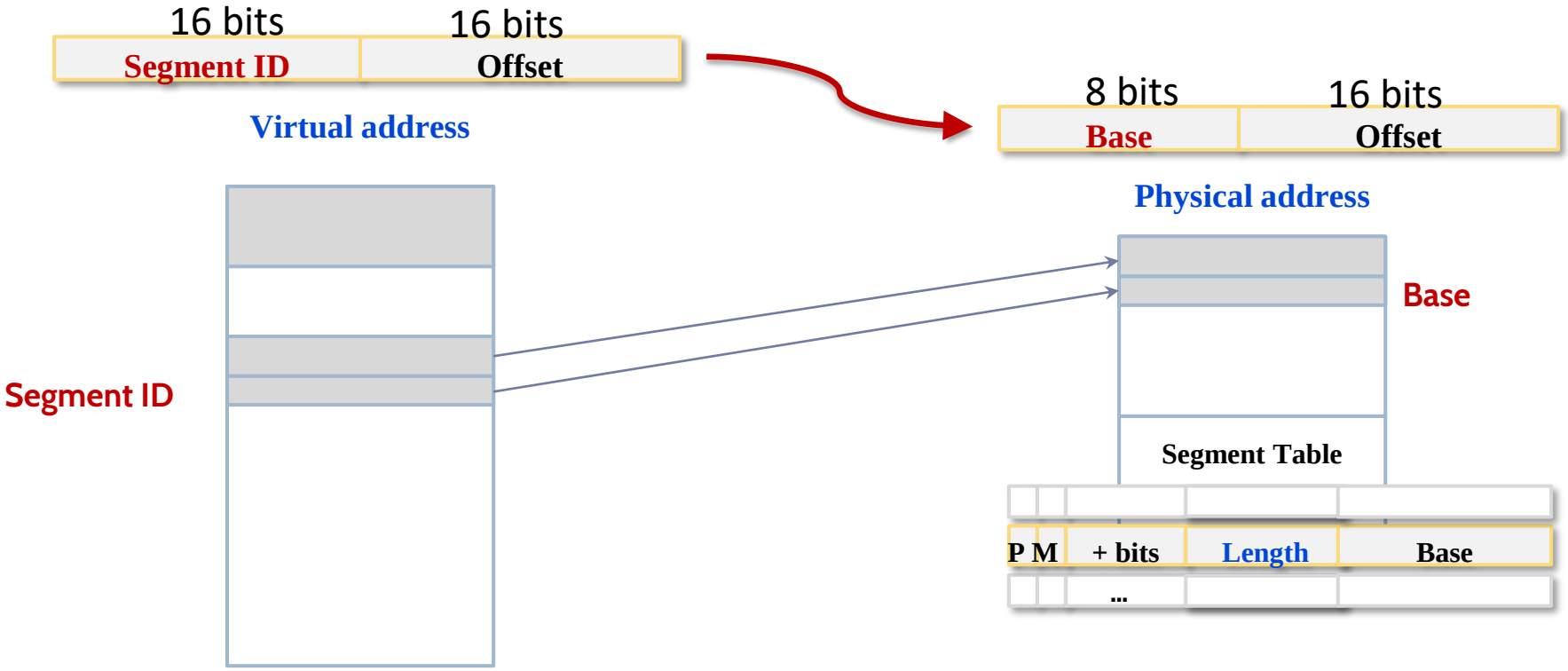
Variable sized regions -> segments

Example



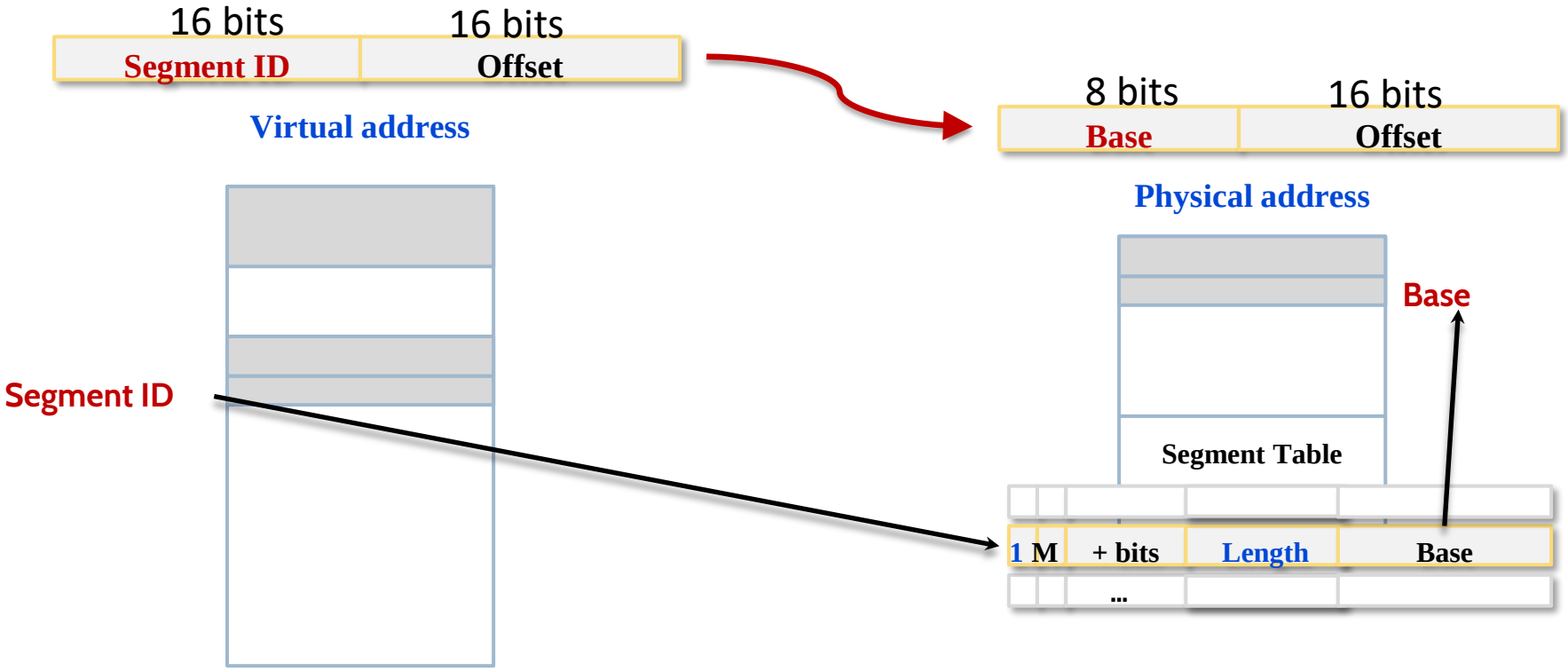
Translation from V.M. to P.M. -> segment table

Example



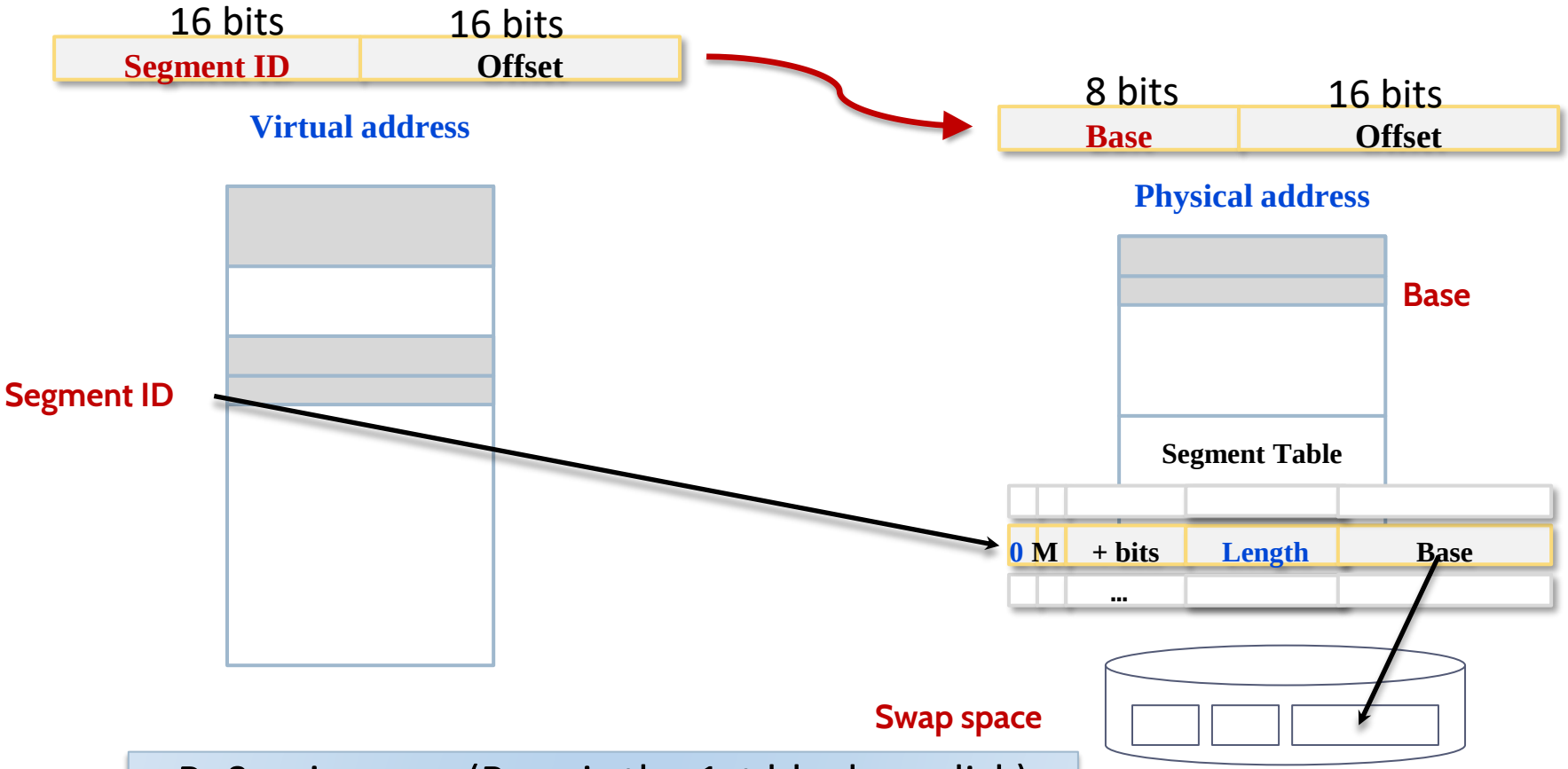
Translation from V.M. to P.M. -> segment table

Example



P=1 -> present in memory (Base is where is it)

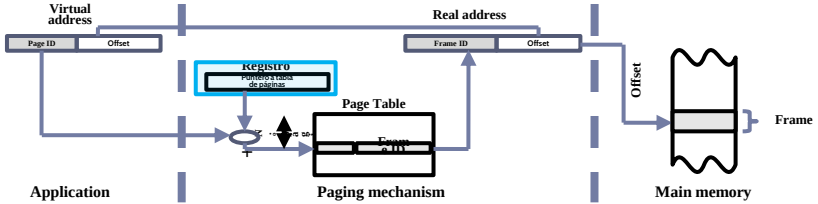
Example



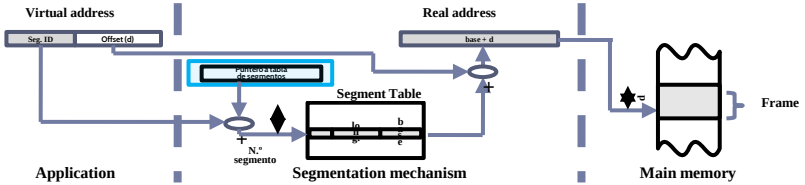
P=0 -> in swap (Base is the 1st block on disk)

Virtual Memory

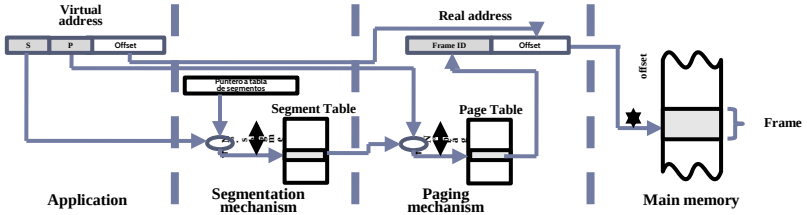
- Paging



- Segmentation

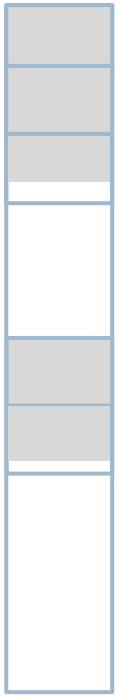


- Segmentation w. paging

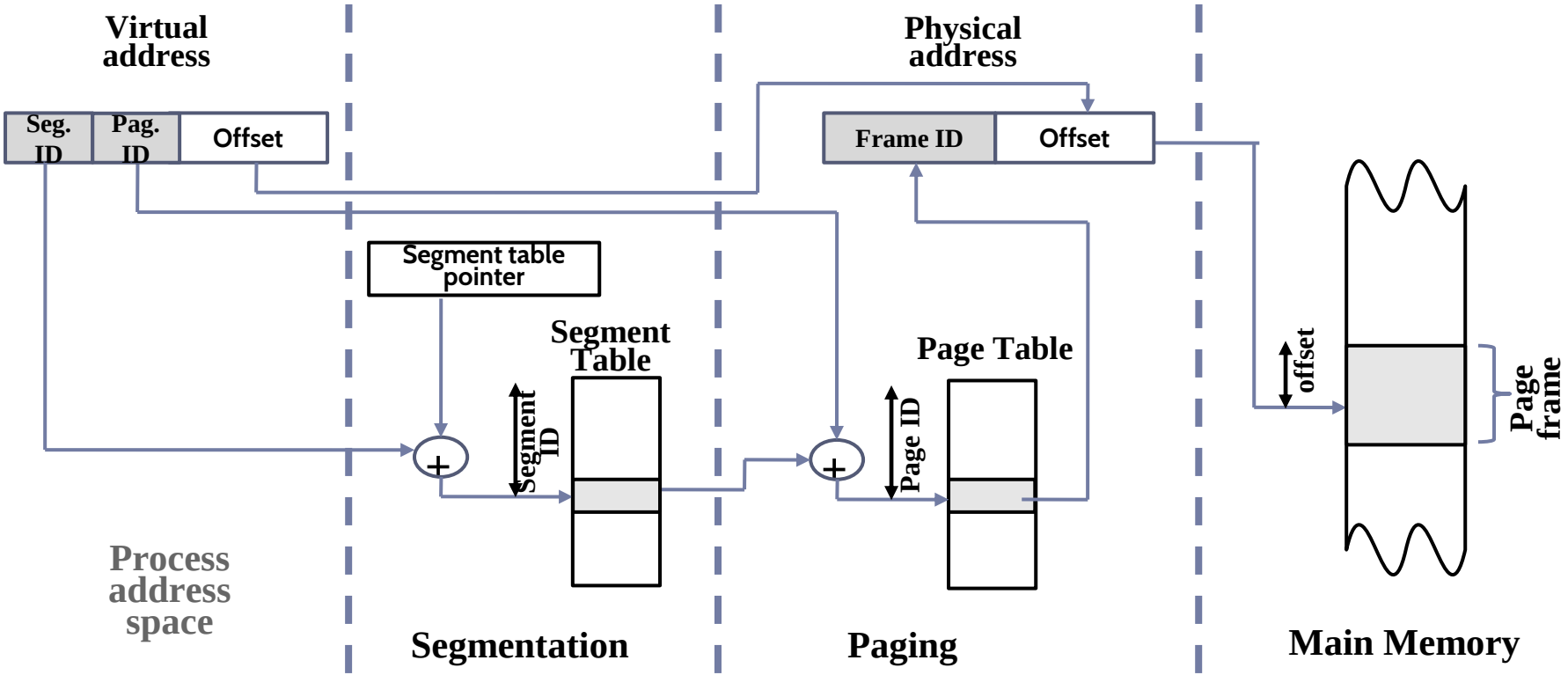


Virtual Memory:
segmentation with paging

- An entry of the segment table "points to" a page table associated with the segment.
  - The variable sized segments are build up from fixed sized pages.



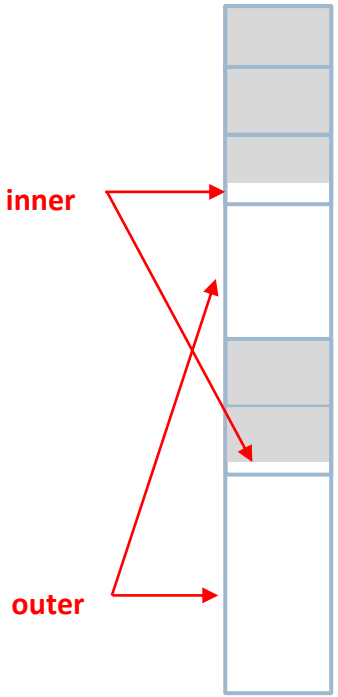
Address translation
segmentation with paging



Virtual Memory: segmentation with paging

- An entry of the segment table "points to" a page table associated with the segment.
  - The variable sized segments are build up from fixed sized pages.

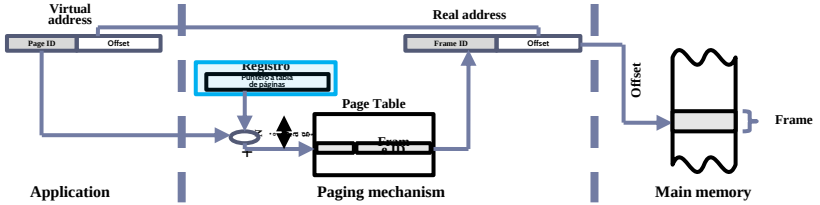
- Best from both worlds:
  - Segmentation:
    - It facilitates memory regions management
    - It avoids the inner fragmentation (it has outer one)
  - Paging:
    - It optimizes the secondary memory access
    - It avoids the outer fragmentation (it has inner one)



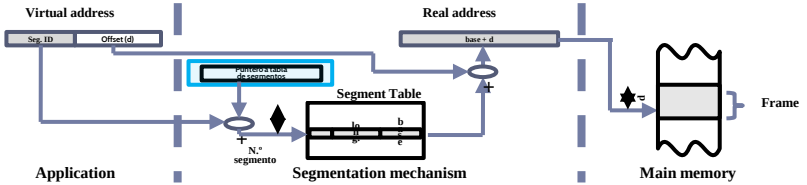
Virtual Memory

summary

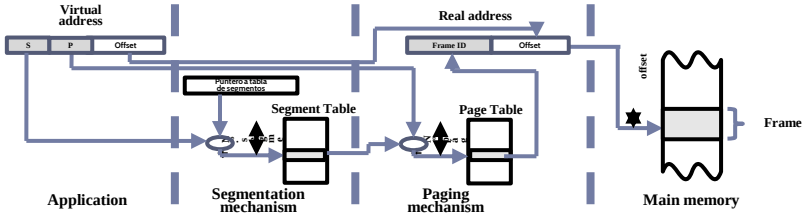
- Paging



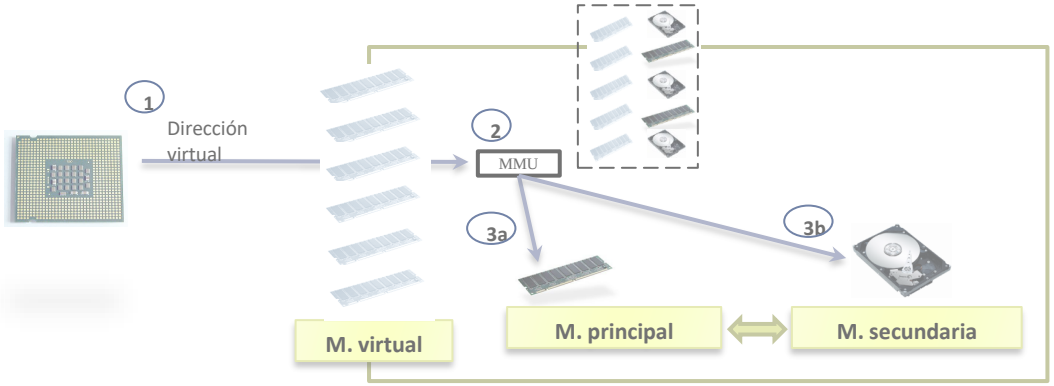
- Segmentation



- Segmentation w. paging



Memory management
advanced aspects



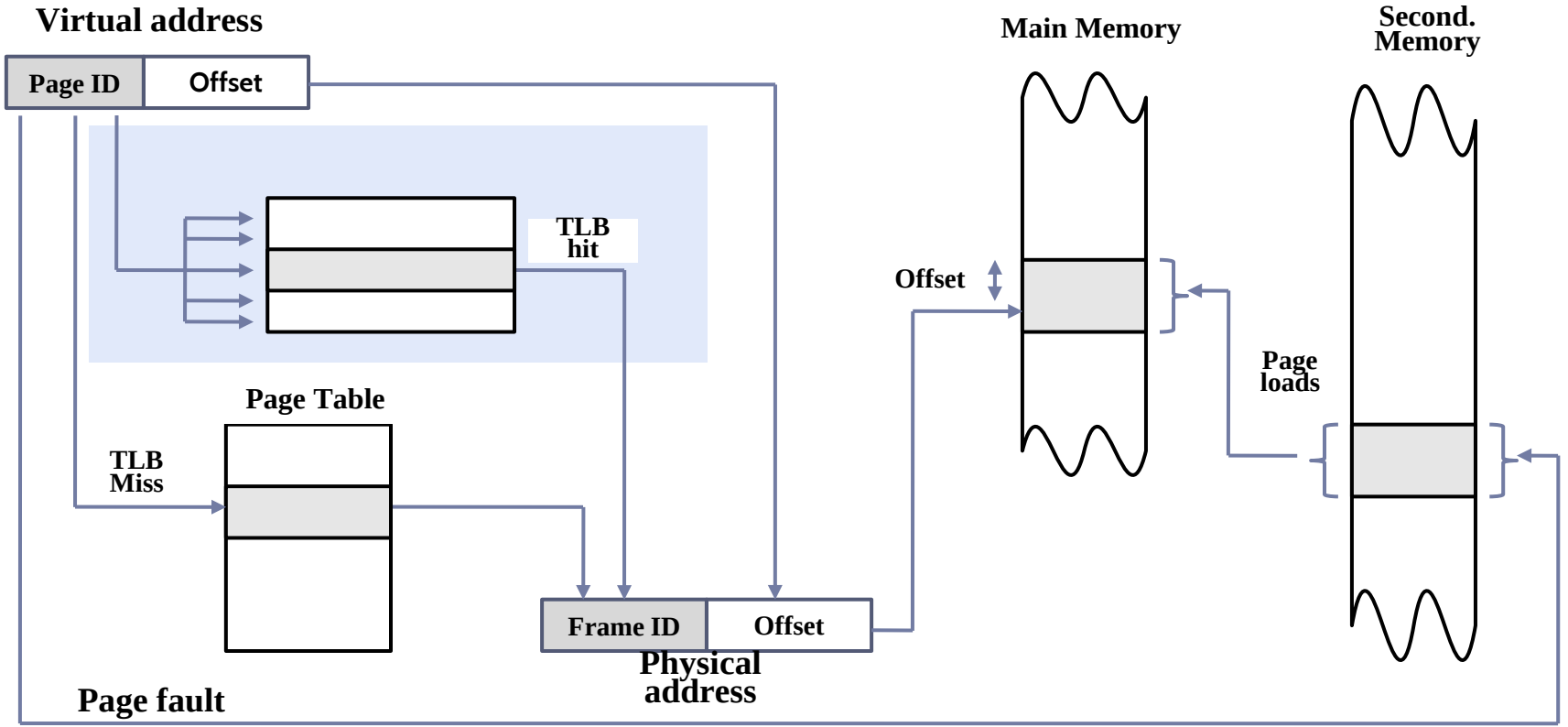
- TLB
- Multi-level tables

Translatation cache

- Virtual memory based on page tables:
  - Problem: two access to memory (slow)
    - To the segment/paging table + to the data/instruction itself
  - Solution: TLB
    - Translation cache

- TLB (Translation Lookaside Buffer)
  - Associative Cache Memory that stores the page table entries most used recently.
  - It is used to speedup the search stage.

Address translation (with TLB)

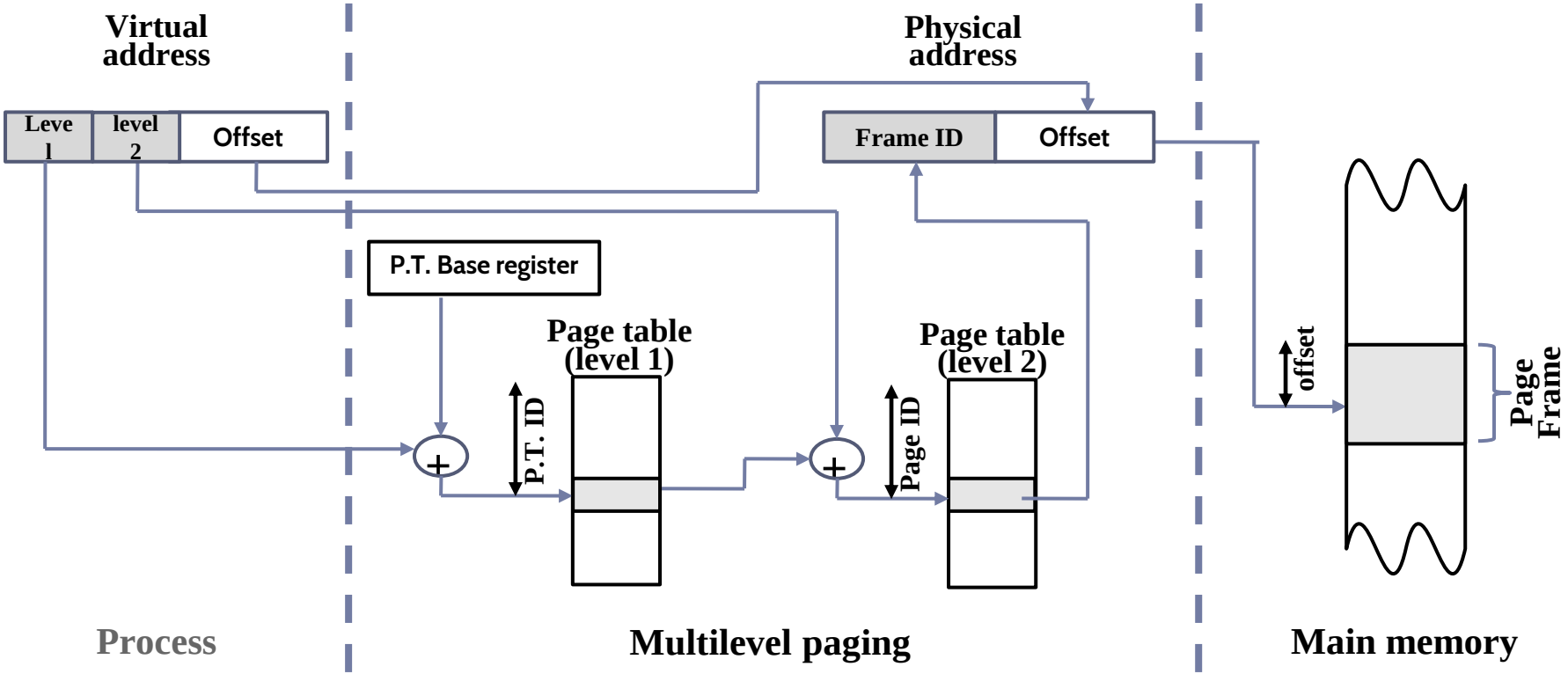


Multilevel paging

- Virtual memory based on page tables:
  - Problem: amount of memory needed for all tables
    - E.g.: 4KB pages, 32 bits v. address, and 4 bytes per entry: $2^{20} * 4 = 4MB$ per process
  - Solution: multilevel tables

- Multi-level table
  - Two level translation scheme:
    - The first level page table is always in main memory
    - Only the second level page tables needed are in main memory
  - More compact tables: $2^{10} * 4 = 4KB$ per table

Multi-level table



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

Lesson 5 (a)
Memory Management

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

CC BY NC SA