# OPERATING SYSTEMS

## UNIT 2

## PROCESS MANAGEMENT

# What we will learn ?

- **What does the operating system do to run…**
  - A program?
  - Several programs simultaneously?

} Processes, threads, Multiprogramming Timesharing ...

- **How does it do it?**
  - Algorithms
  - Data Structures

} Lifecycle PCB, Process Table ...

# CONTENTS

1. **Process Management basics**
2. Elements of a Process
3. Process lifecycle
4. Threads (or lightweight processes)
5. Process and thread planning
6. System services for process and thread management

# THE CONCEPT OF PROCESS

## 1 - PROCESS MANAGEMENT BASICS

- A process is **any running program**.
- A process needs certain **RESOURCES** to perform its task satisfactorily:
  - CPU time.
  - Memory.
  - Files.
  - I / O Devices.
- Resources are allocated to a process:
  - When creating.
  - During execution

# THE CONCEPT OF PROCESS

## 1 - PROCESS MANAGEMENT BASICS

- **A process is alive**
- An instant in the process's life is given by
  - Its **code**
  - **Data** that has at that moment
  - The value of the execution **stack**
  - The value of all processor **registers**
  - The **program counter**
    - A pointer to the next instruction to be executed

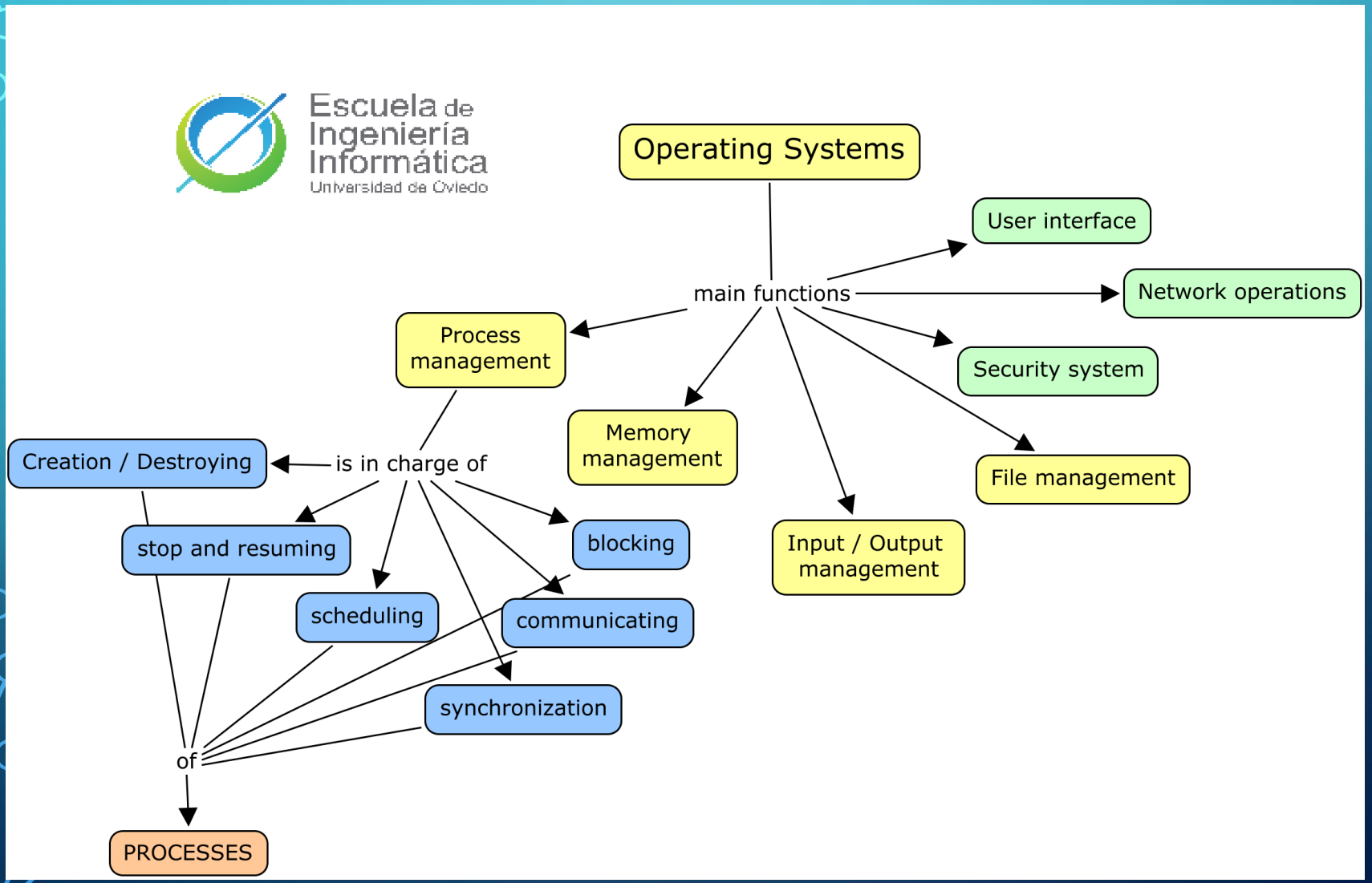
# PROCESS MANAGER FUNCTIONS

## 1 - PROCESS MANAGEMENT BASICS

- The obligations of the OS as process manager are:
  - **Creating and deleting** processes.
  - **Process scheduling** (ensuring the execution of multiple processes to maximize processor utilization).
  - Establishing mechanisms for process **synchronization and communication**.
  - Handling **deadlocks**.

# PROCESS MANAGER FUNCTIONS

## 1- PROCESS MANAGEMENT BASICS

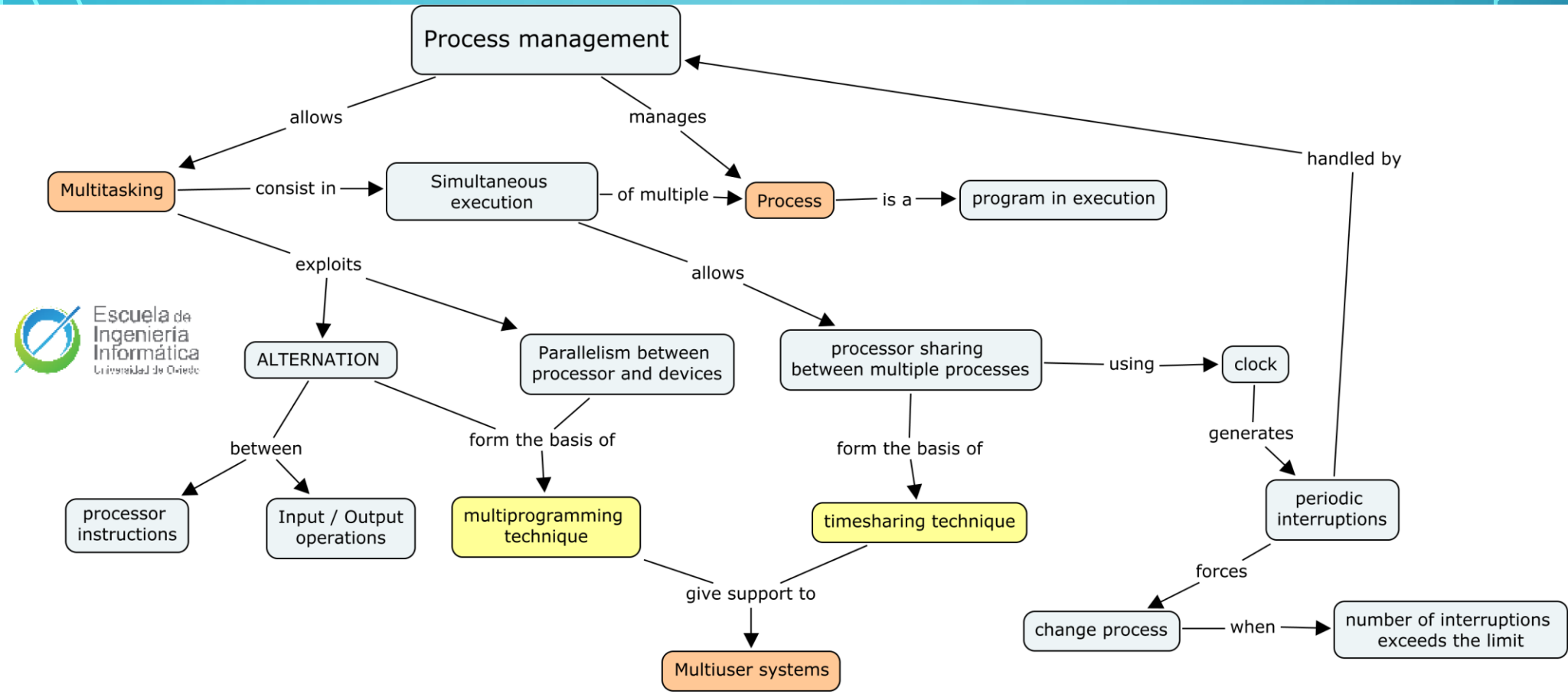


- **How can many processes be executed simultaneously on a machine with a single processor ?**

1. *Parallelism between processor and I / O*
2. *Programs alternation between processor and I / O*
3. *Processor time sharing among processes*

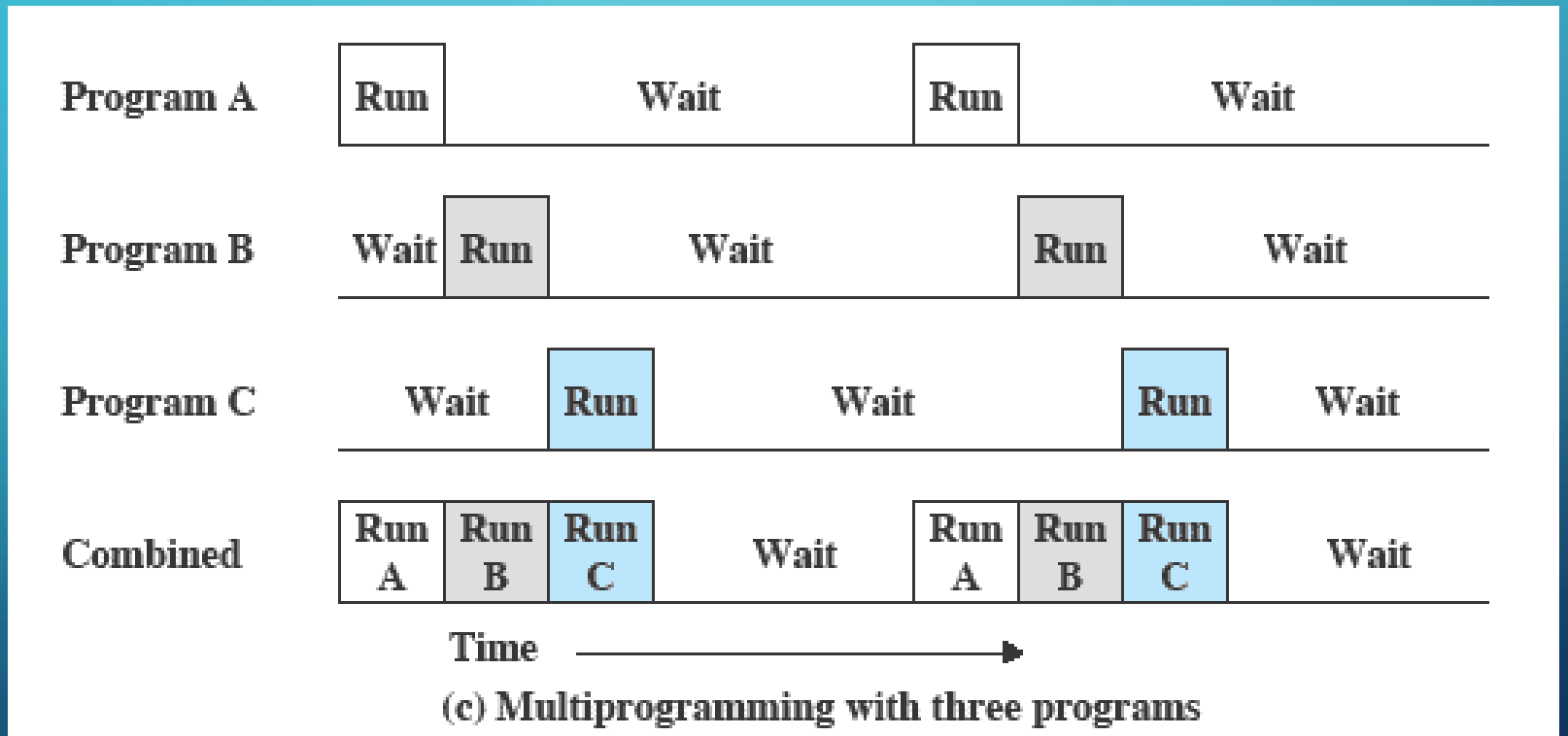
# MULTITASKING

## 1- PROCESS MANAGEMENT BASICS



# MULTITASKING

## 1- PROCESS MANAGEMENT BASICS




(c) Multiprogramming with three programs

# CONTENTS

1. Process Management basics
2. **Elements of a Process**
3. Process lifecycle
4. Threads (or lightweight Processes)
5. Process and thread planning
6. System services for process and thread management

# What information does the OS need to handle about a process to run it on Multitasking?

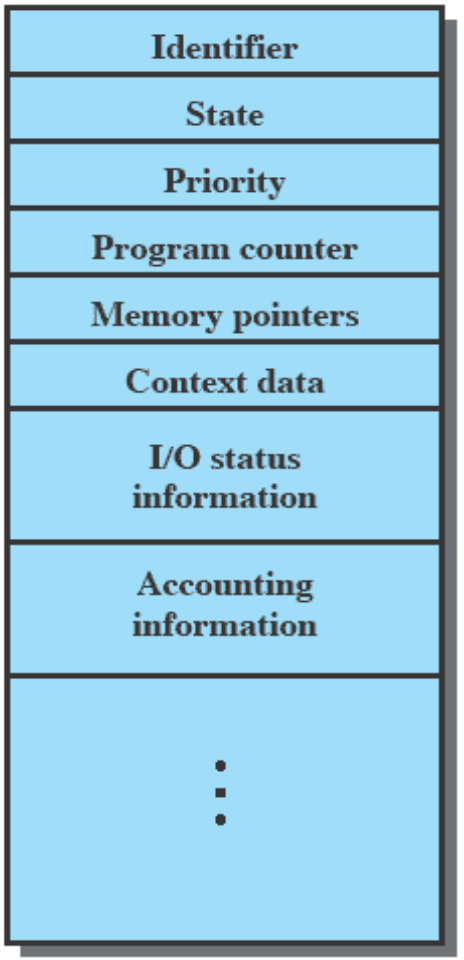
# 2.- ELEMENTS OF A PROCESS

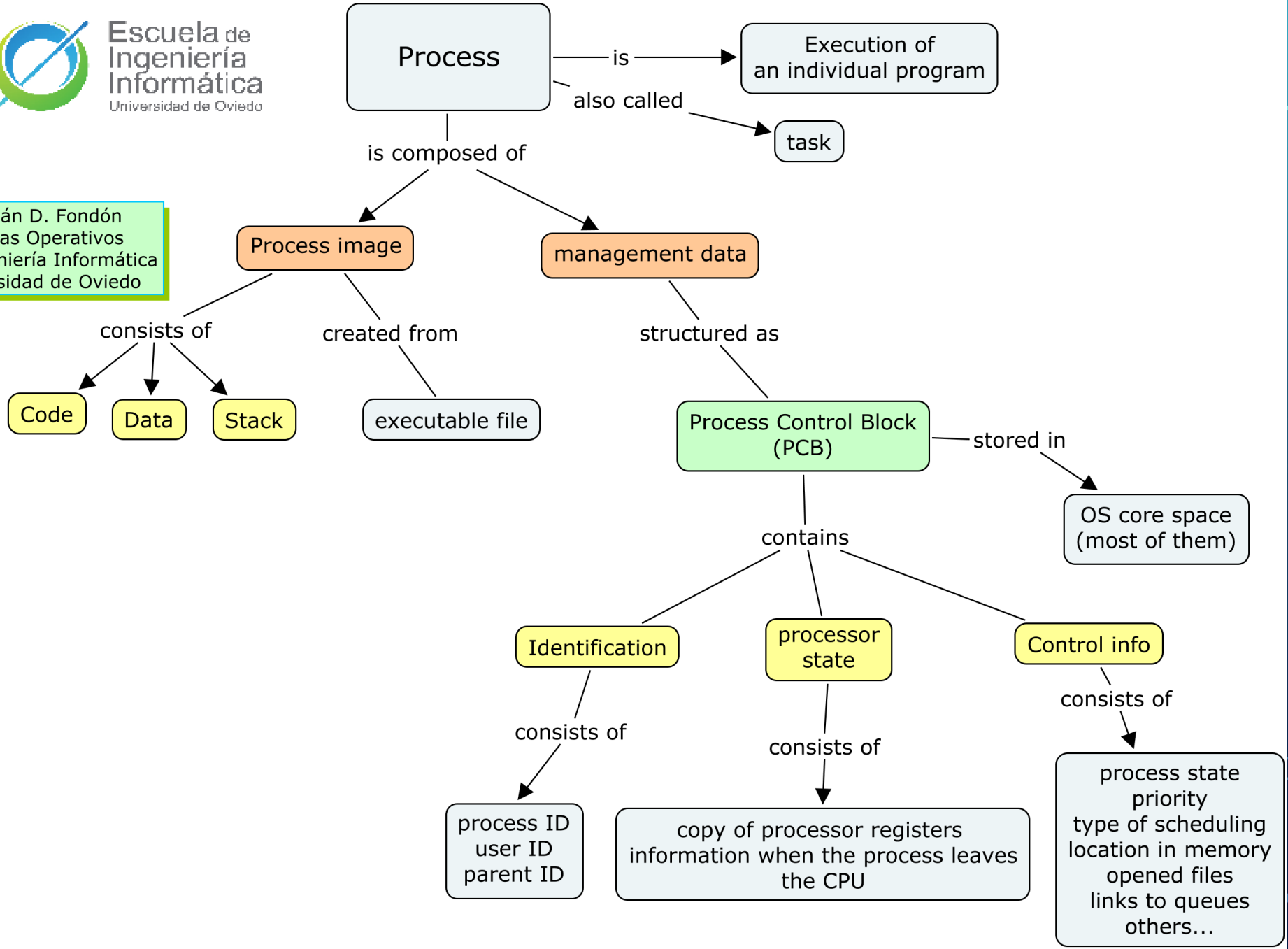
- A process is:
  - **Text section** (program code)
  - **Data section** containing *global variables* and *dynamic memory*
  - **Stack**, which contains *temporary data* (subroutine parameters, return addresses and local variables)
  - Current activity, represented in a **PCB** data structure:
    - Value of the **program counter**
    - Contents of **processor registers**
    - Priority, status, open files, memory space location, ...
- The OS usually uses a **Table of PCBs** to locate all processes in the system.

# Process Control Block

## 2.- Elements of a Process

- Data structure that contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes



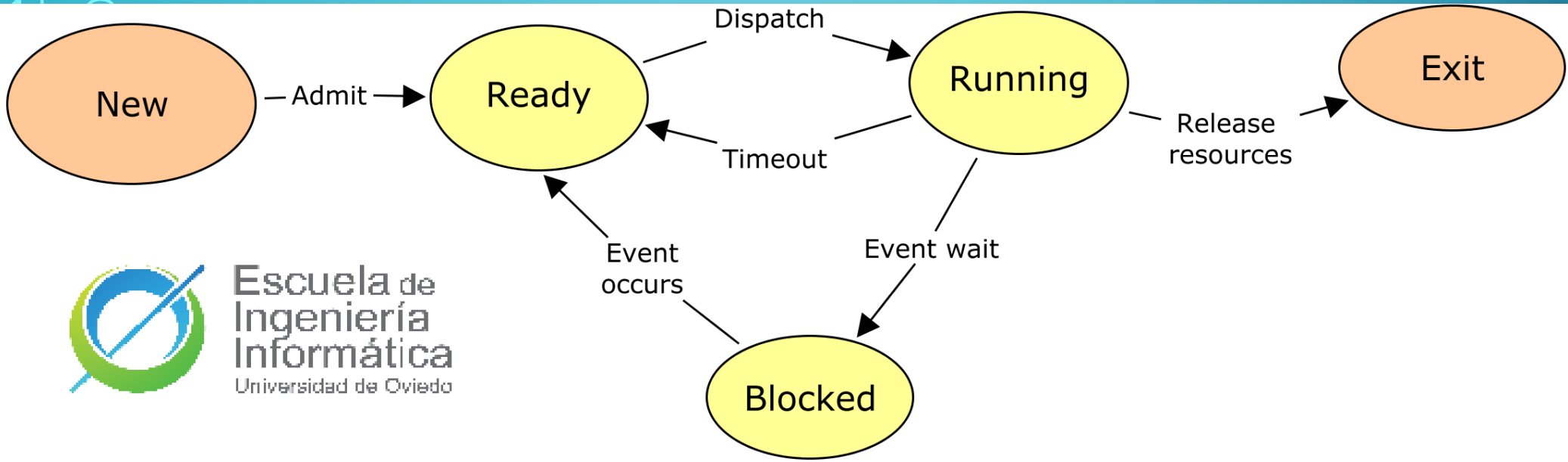


# CONTENTS

1. Process Management basics
2. Elements of a Process
3. **Process lifecycle**
4. Threads
5. Process and thread planning
6. System services for process and thread management

# FIVE STATE MODEL

## 3.- PROCESS LIFECYCLE

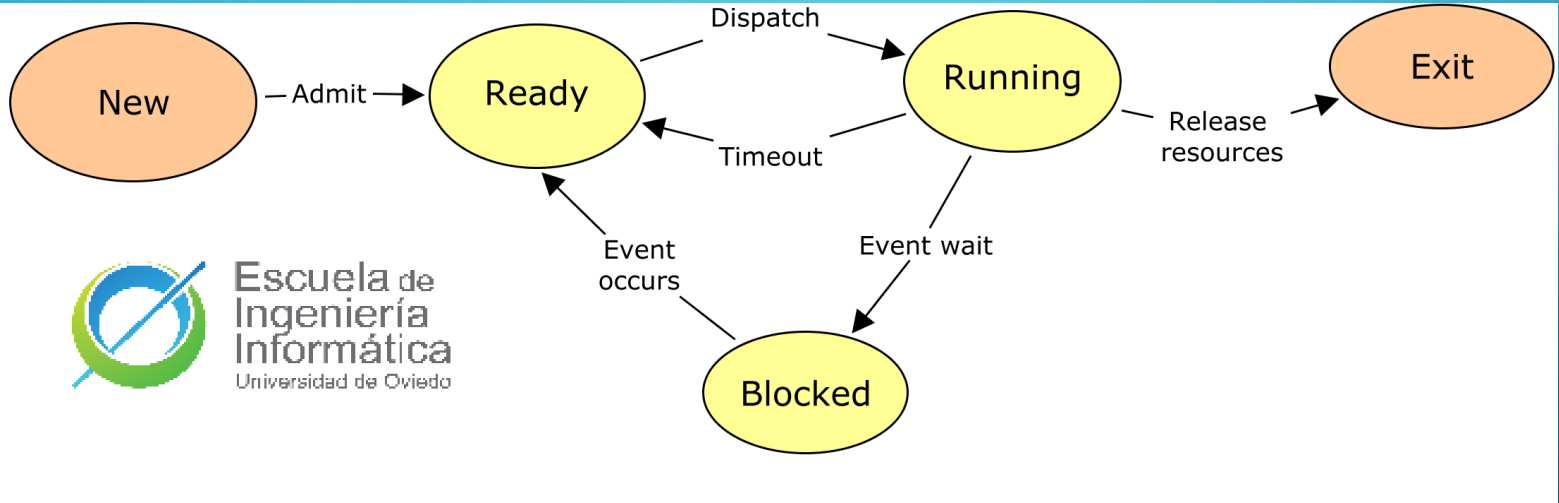



Escuela de Ingeniería Informática
Universidad de Oviedo

# FIVE STATE MODEL

## 3- PROCESS LIFECYCLE

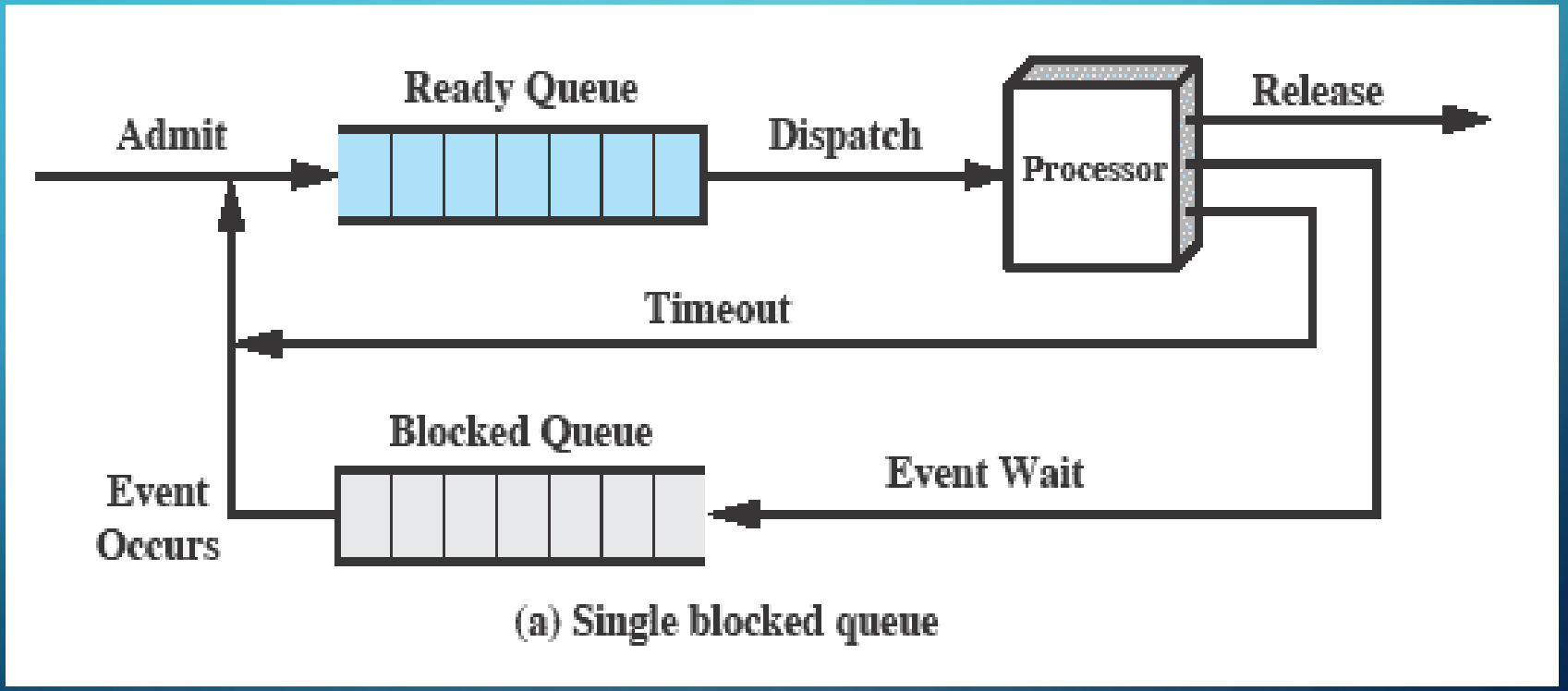
1. **What situations does imply a state change ?**




Escuela de Ingeniería Informática
Universidad de Oviedo

# FIVE STATE MODEL

## 3- PROCESS LIFECYCLE




(a) Single blocked queue

# FIVE STATE MODEL

## 3- PROCESS LIFECYCLE

- As a process is executed, its **state changes**. Each process can be in one of this states:
  - **New:** The process is being created.
  - **Running**: The process is in the CPU executing instructions.
  - **Blocked**: The process is waiting for an event to occur (e.g. I/O completion or signal reception).
  - **Ready**: waiting to be assigned to a processor.
  - **Finished:** execution completed, thus no further instructions to execute and the OS will withdraw the resources it consumes.

# FIVE STATE MODEL

## 3- PROCESS LIFECYCLE

- State transitions in this model are:
  - **None to New**: creates a new process to run a program
  - **New to Ready**: the system is ready to accept a new process as it has enough resources for it.
  - **Ready to Running**: the system chooses one of the processes in the ready state to carry out execution.
  - **Running to Finished**: the running process is terminated by the OS if it indicates that it has ended, abandoned or canceled.
  - **Running to Ready**: the process has exhausted its running time, or yields voluntarily or it is stopped to execute other process (possibly with higher priority).

# FIVE STATE MODEL

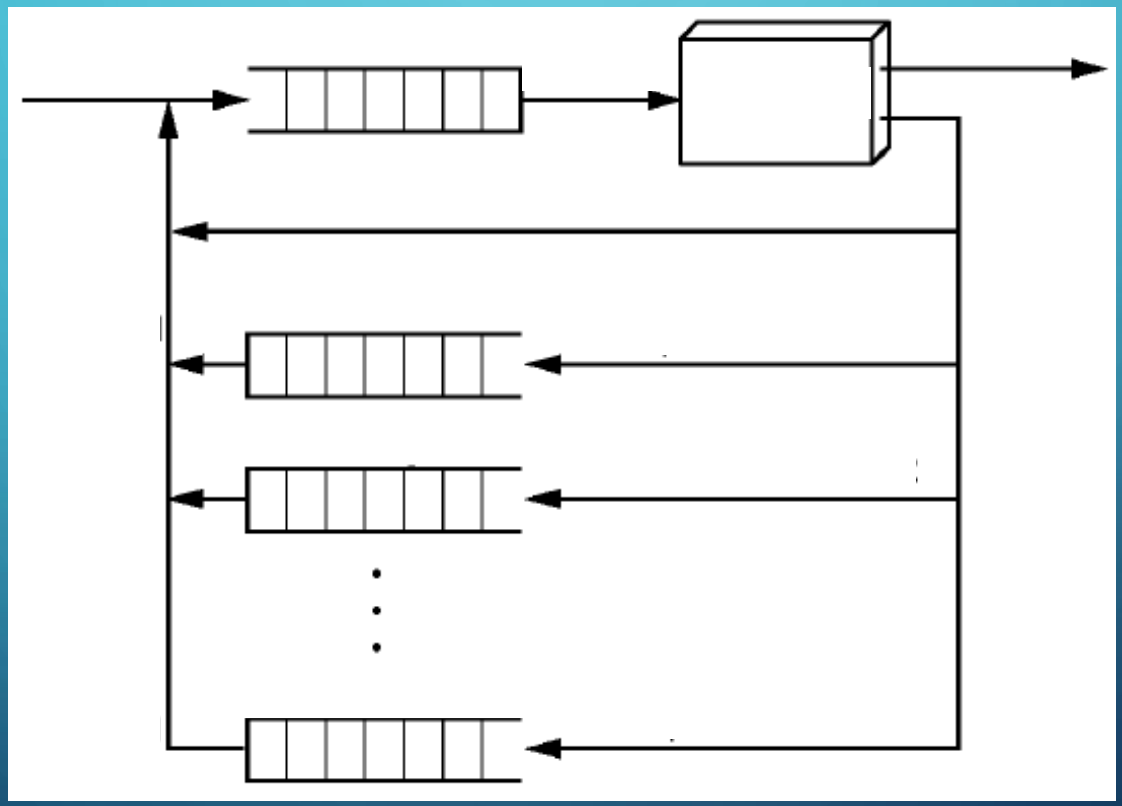
## 3- PROCESS LIFECYCLE

- State transitions 2nd part:
  - **Running to Blocked:** the process asks for a (slow) operation and waits for it
  - **Blocked to Ready:** the (slow) operation the process is waiting for eventually finishes.
  - **Ready to Finished** (not shown in the figure): a parent may terminate a child process at any time, or if the parent is completed all children can be completed.
  - **Blocked to Finished:** the same criteria as above.

# IMPLEMENTATION

## 3- PROCESS LIFECYCLE

The states have queues associated for the processes that are in that state



# CONTEXT SWITCHING

## 3- PROCESS LIFECYCLE

**Context Switching**

*Operations performed by the OS to make a process release the processor, and make another one obtain the processor.*

*Implies:*

- Saving processor state in the PCB for the process that leaves the CPU.
- Restoring the processor state from the process's PCB that is assigned to the CPU.

# OS INTERRUPTION HANDLING

## 3- PROCESS LIFECYCLE

**Process  management issues**

- What happens when an interruption occurs? How do you jump to the OS code?
- What does a process change imply?
- What kinds of interruption necessarily cause a process change and which don´t?
- What does the creation of a process imply?

# OS interruption handling

## 3- Process lifecycle

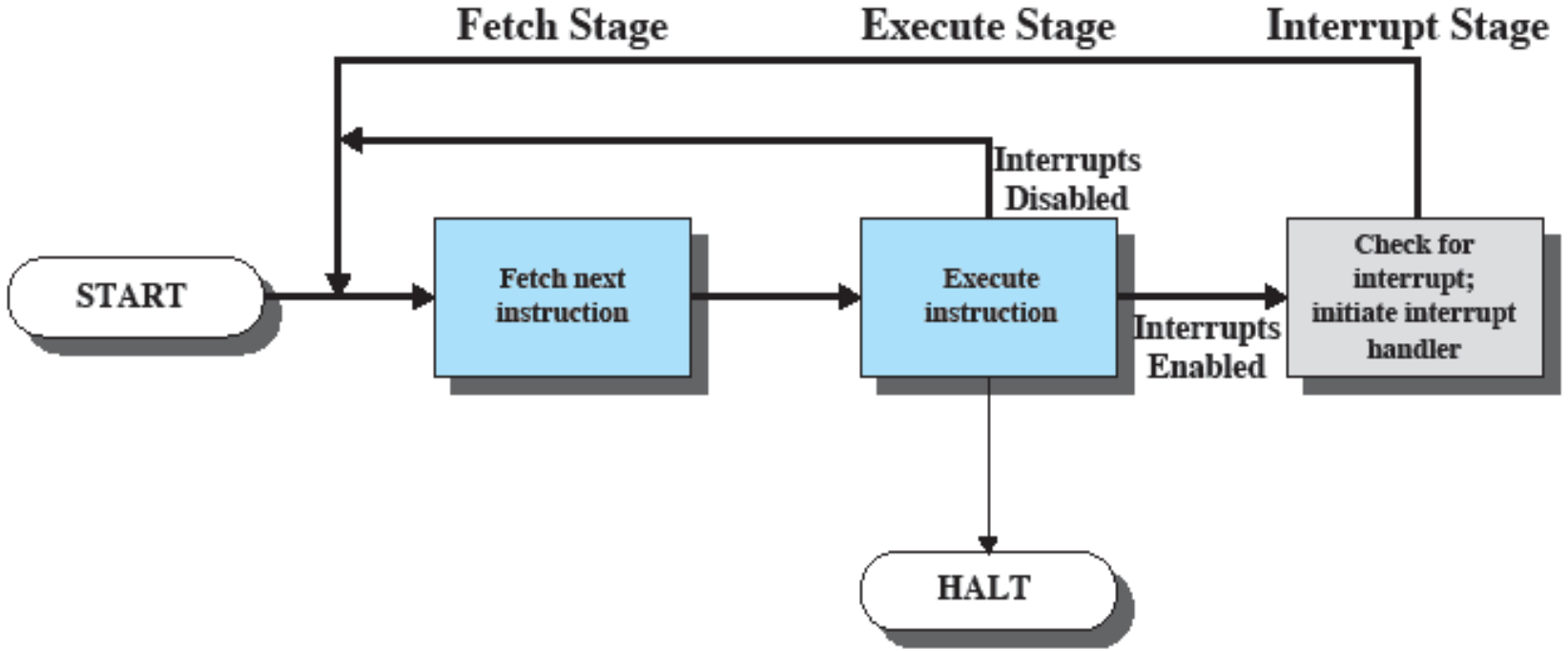



Figure 1.7 Instruction Cycle with Interrupts

# What will we learn ?

- **Process an interruption in detail**
  - A **very simple** interruption
  - It doesn't involve any process switching
  - The OS just does some very small task and goes back to the program
  - The OS is in the same address space as the program (!?)

# INTERRUPTION HANDLING

## 1 - IRQ number 5 arrives

- **Tasks for Figure 1:**
  - Draw the lines (pointers) from PC, BSP, SP to the memory
  - Figure out at which **memory address** the handling routine for IRQ 5 (i.e., the code to be executed) starts

– Note: IRQ means “Interruption ReQuest”
– Note: First bit in PSW is mode (0=user, 1=kernel or privileged)

# INTERRUPTION HANDLING

**2 - Hardware, in only one operation, a) stores PC and PSW at the bottom of the "control stack", b) activates kernel mode, c) points PC to first instruction in the handling routine code**

- **Tasks for Figure 2:**
  - Update numbers in the picture, both processor and memory
  - Then draw pointers as needed

# INTERRUPTION HANDLING

**3 – The rest of the processor registers are stored in control stack, and stack pointers (SP and BSP) point to that stack.**

- **Tasks for Figure 3:**
  - Update numbers in the picture, both processor and memory
  - Then draw pointers as needed

# INTERRUPTION HANDLING

**4 – Instructions of interruption handler are executed, until <u>just before</u> the return (IRET is located at memory address 13999).**

- **Tasks for Figure 4:**
  - Update numbers and draw pointers as usual

# INTERRUPTION HANDLING

**5 – Last instructions of the Interrupt Handler restores general registers and stack registers to the processor**

- **Tasks for Figure 5:**
  - Update numbers and draw pointers as usual

# INTERRUPTION HANDLING

**6 – As the last step of "IRET", hardware restores PC and PSW.**

- **Tasks for Figure 6:**
  - Update numbers and draw pointers as usual

– After that… is the processor in user mode or in kernel mode?

# OS INTERRUPTION HANDLING

## 3- PROCESS LIFECYCLE

**Process  management issues**

What happens when an interruption occurs? How do you jump to the OS code?

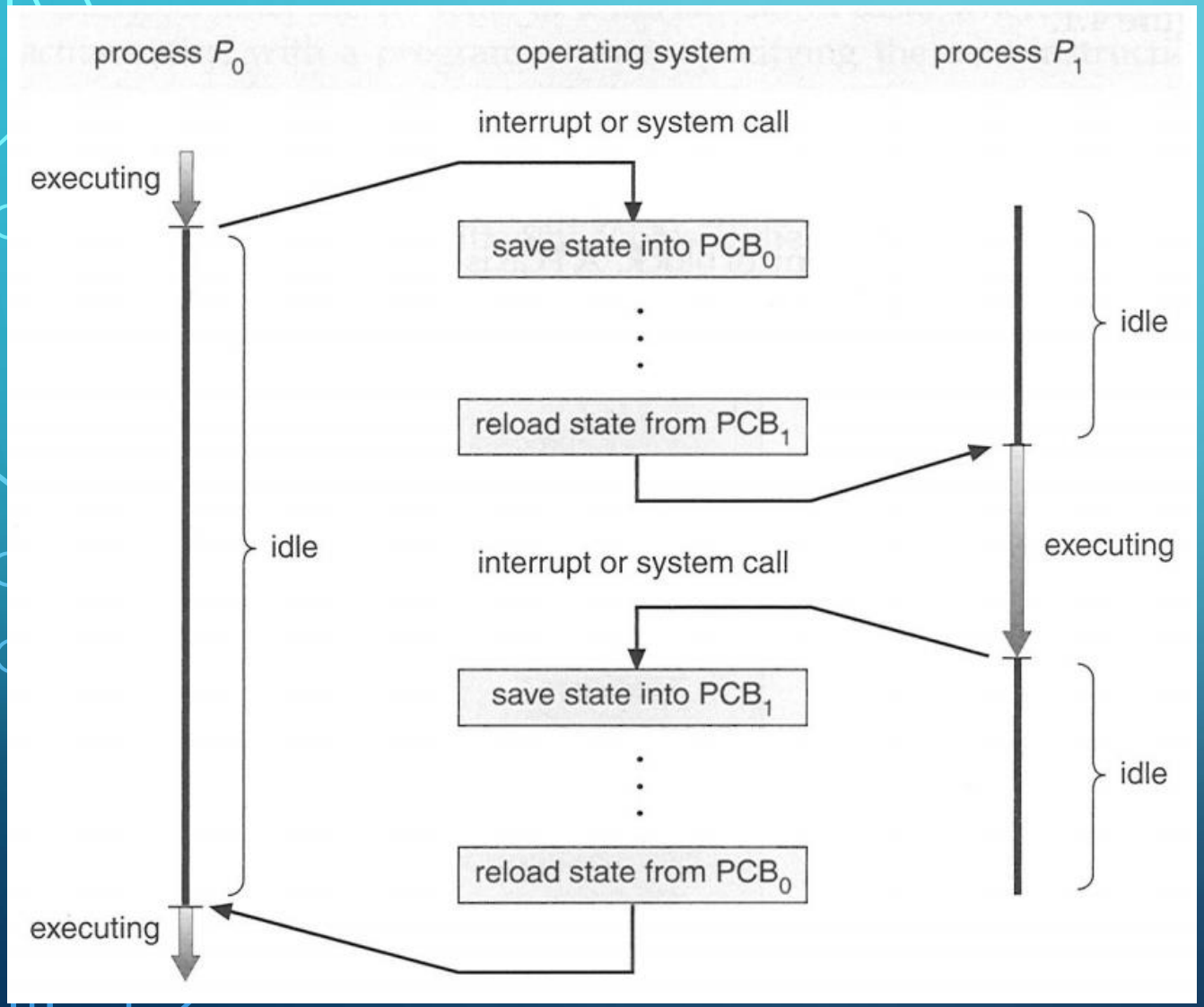
What does imply a process change?

What kinds of interruption necessarily cause a process change and which don´t?

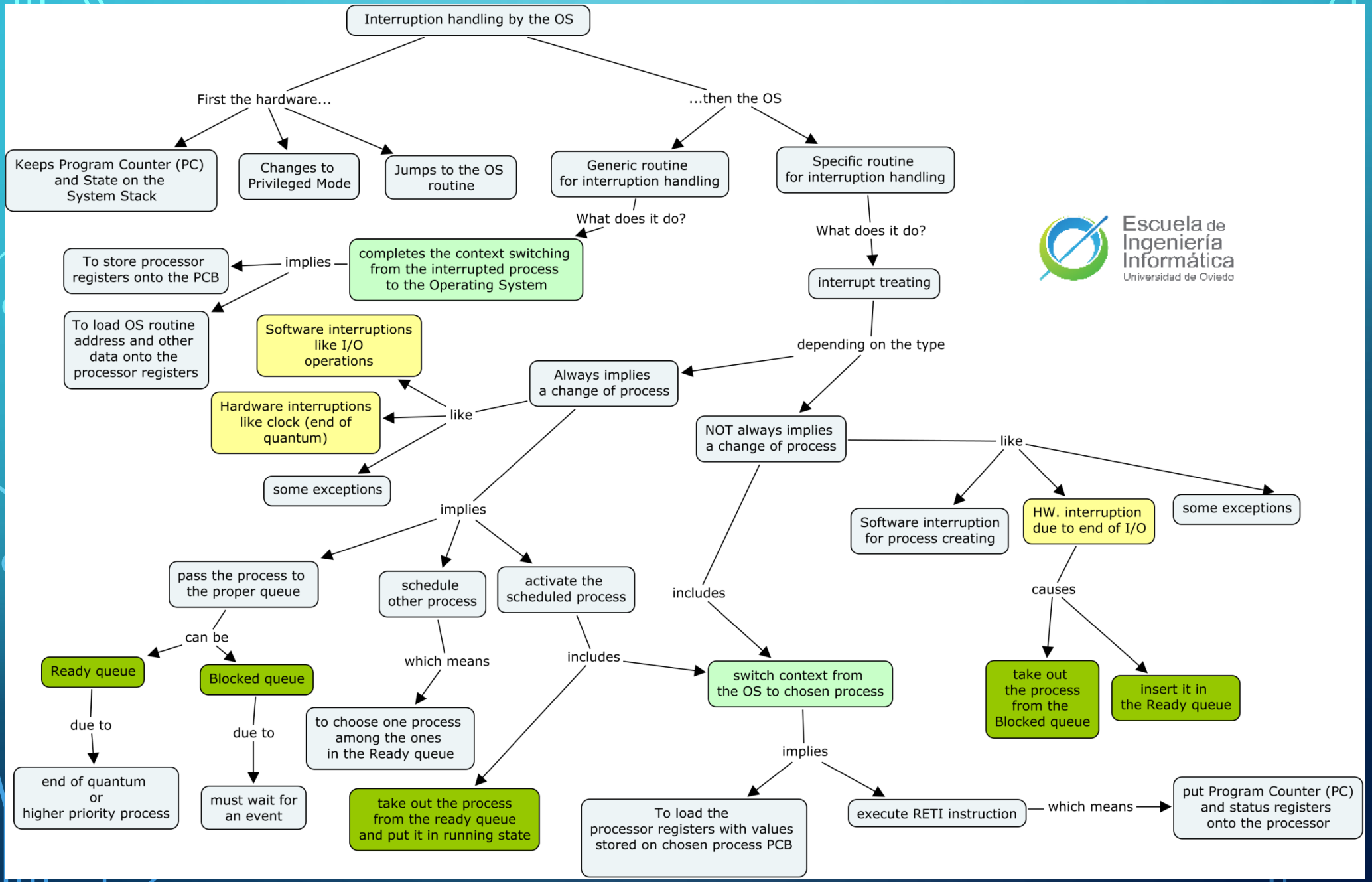
What does imply the creation of a process?

# Context Switching

### 3- Process lifecycle



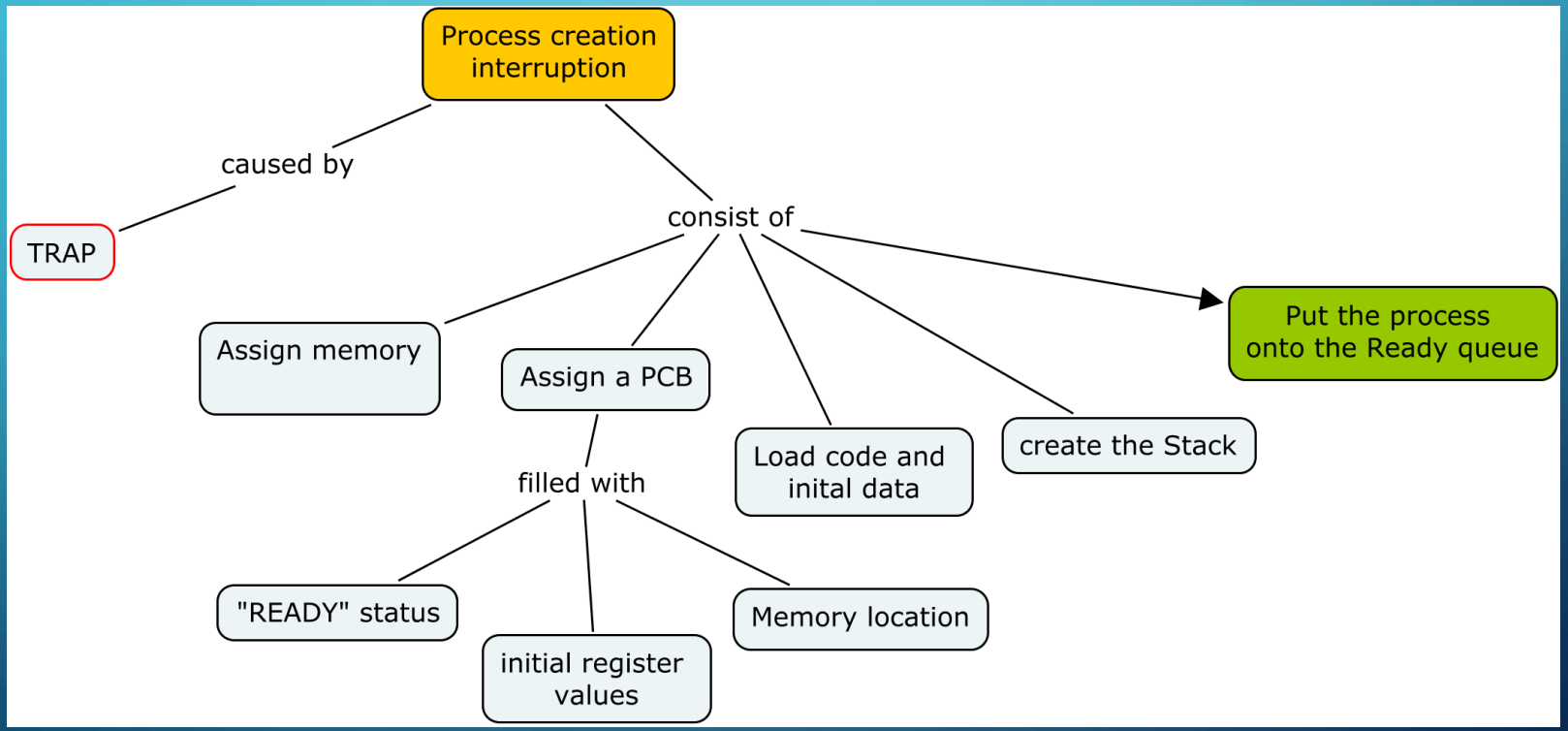

What does imply a process change?


Interruption handling by the OS
First the hardware...
...then the OS
Keeps Program Counter (PC) and State on the System Stack
Changes to Privileged Mode
Jumps to the OS routine
Generic routine for interruption handling
Specific routine for interruption handling
What does it do?
What does it do?
completes the context switching from the interrupted process to the Operating System
implies
To store processor registers onto the PCB
To load OS routine address and other data onto the processor registers
interrupt treating
depending on the type
Always implies a change of process
NOT always implies a change of process
like
Software interruptions like I/O operations
Hardware interruptions like clock (end of quantum)
some exceptions
like
Software interruption for process creating
HW. interruption due to end of I/O
some exceptions
implies
pass the process to the proper queue
schedule other process
activate the scheduled process
includes
causes
can be
Ready queue
Blocked queue
which means
includes
switch context from the OS to chosen process
take out the process from the Blocked queue
insert it in the Ready queue
due to
due to
to choose one process among the ones in the Ready queue
implies
end of quantum or higher priority process
must wait for an event
take out the process from the ready queue and put it in running state
To load the processor registers with values stored on chosen process PCB
execute RETI instruction
which means
put Program Counter (PC) and status registers onto the processor
Escuela de Ingeniería Informática
Universidad de Oviedo

# Process creation

## 3- Process lifecycle

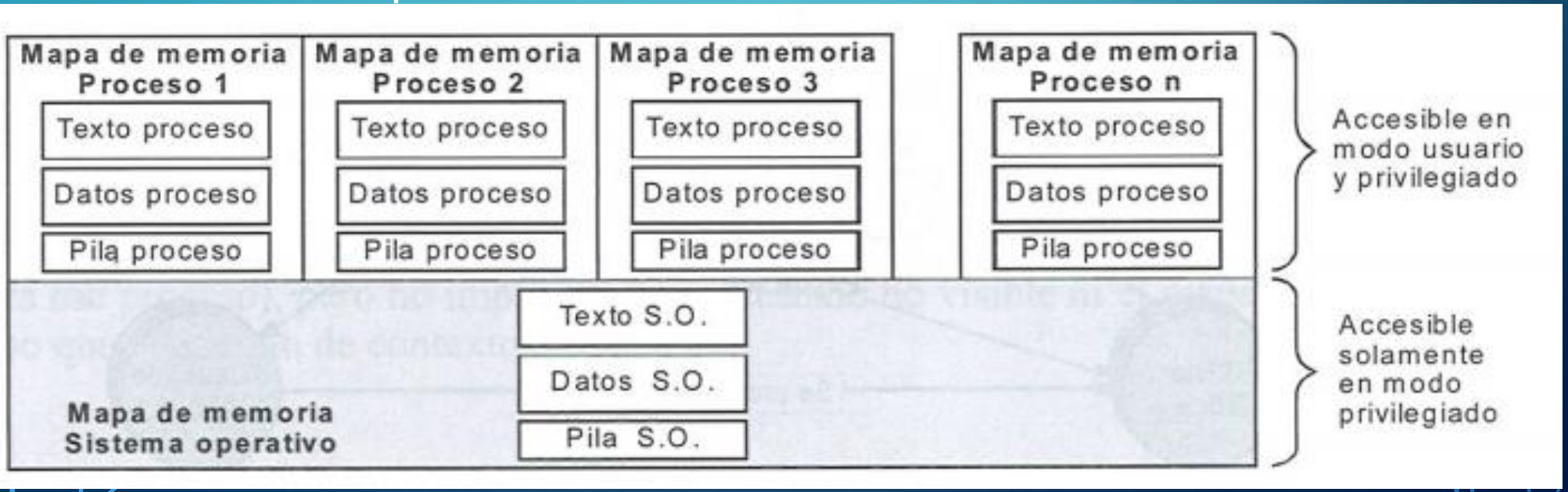
What does imply the creation of a process?



# Independent Kernel

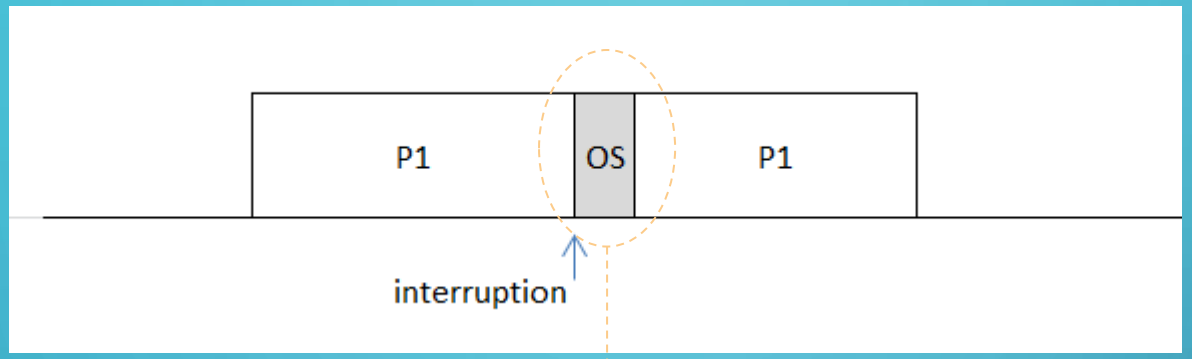
### 3- Process lifecycle

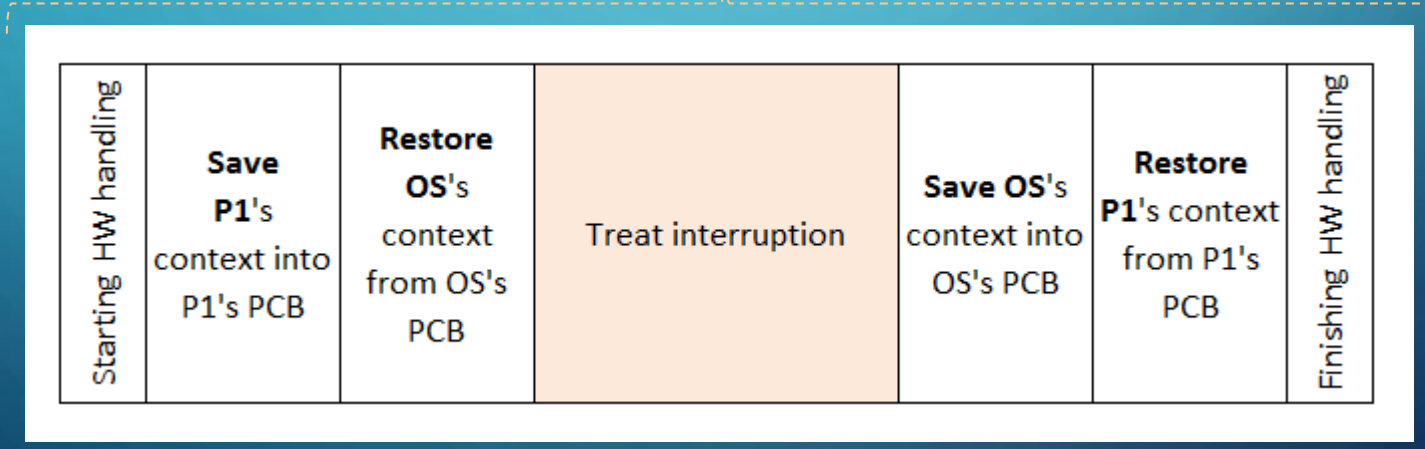
- Independent Kernel
  - Performs two context switching on every interruption
  - Implies to change the processor mode and the memory map
    - Memory map can be expensive to change
  - Can be improved?



# Independent kernel execution

## 3- Process lifecycle

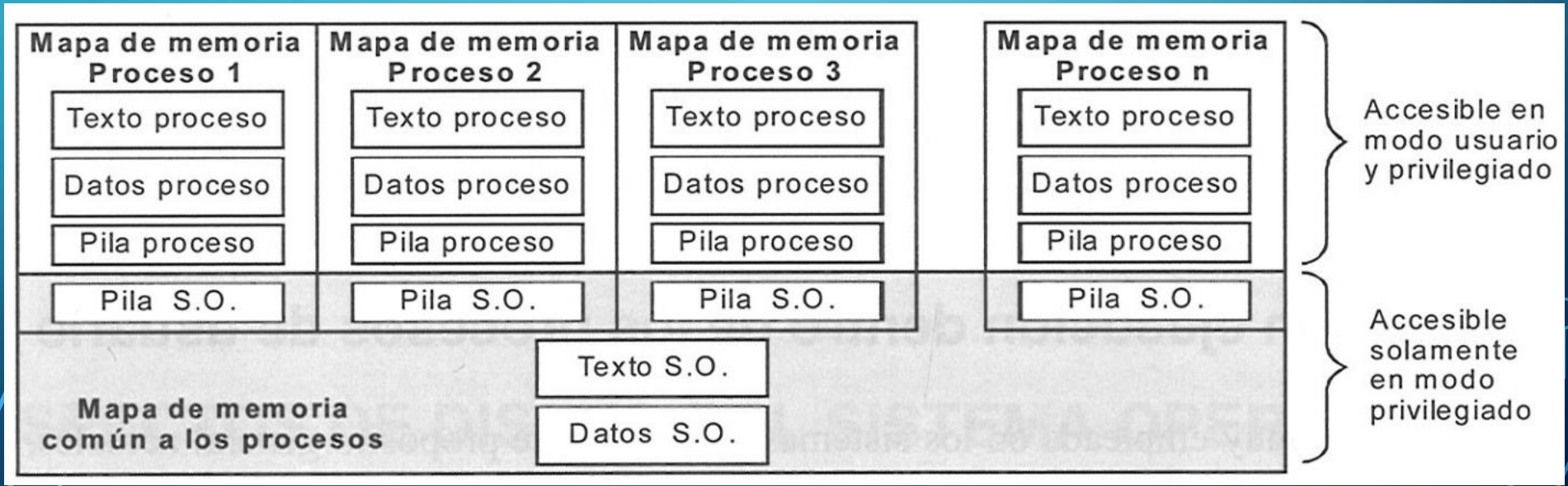


# In-process execution

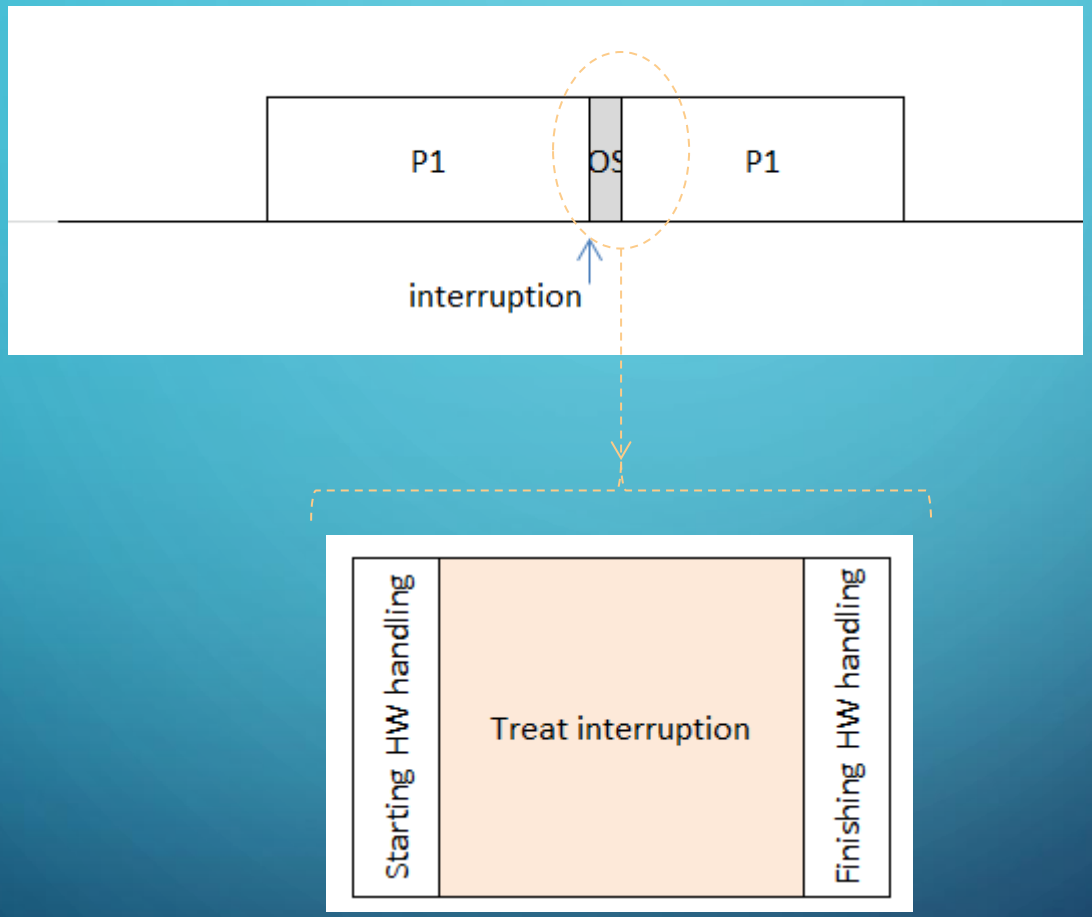
## 3- Process lifecycle

- In-process execution can avoid context switching
  - Just changes the processor mode



# In-process execution

## 3- Process lifecycle



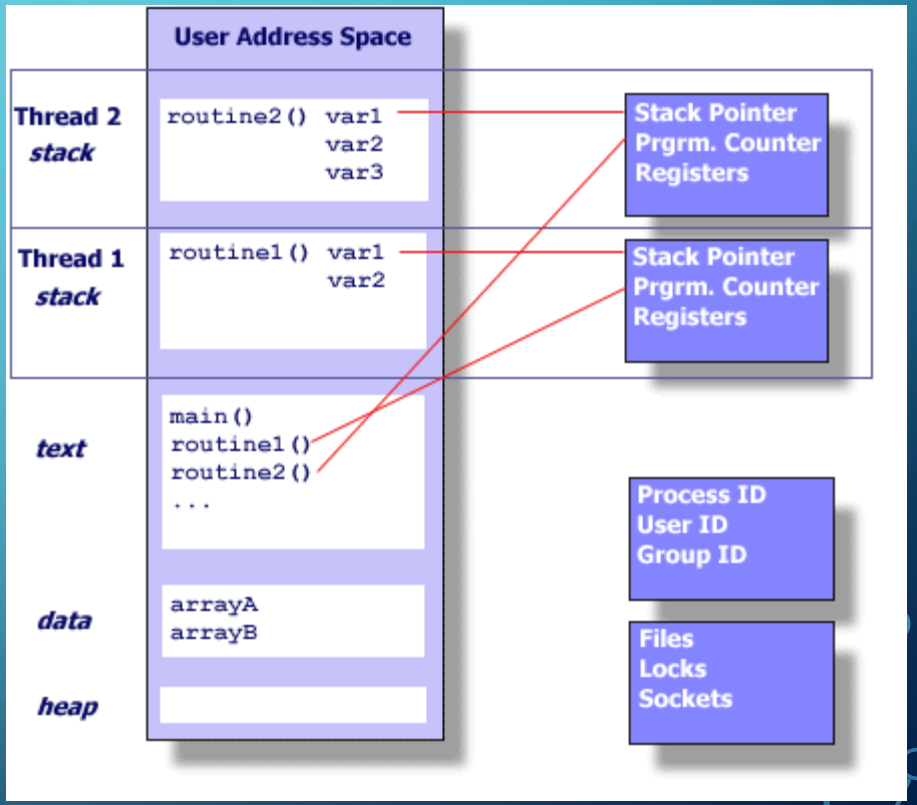
# CONTENTS

1. Process Management basics
2. Elements of a Process
3. Process lifecycle
4. **Threads**
5. Process and thread planning
6. System services for process and thread management

# 4- THREADS

***A flow of execution within a process***

- *Multiple threads in the same task*
- *Each thread has:*
  - *Program Counter,*
  - *Values of processor registers,*
  - *Stack*
- *Threads share the*
  - *Code (text), data and heap*
  - *Open files, signals, etc..*



# 4- THREADS

**Applications**

- Parallel Programming (task separation, modularity)
- Information Servers

*Applications… for example?*

*Advantages… Why?*

**Advantages**

- Ease communication between threads.
- Increase the speed of execution.
- Switching between threads of the same process has little cost.
- The cost of creation and destruction of threads is much lower than with process
- It therefore improves performance.

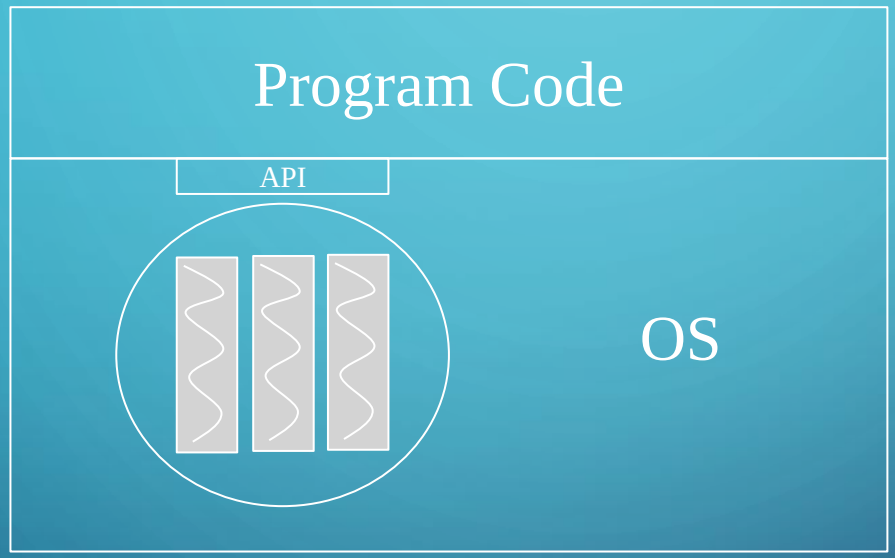
**Implementation**

- At kernel level (KLT)
- At user level (ULT)
- Combination of both

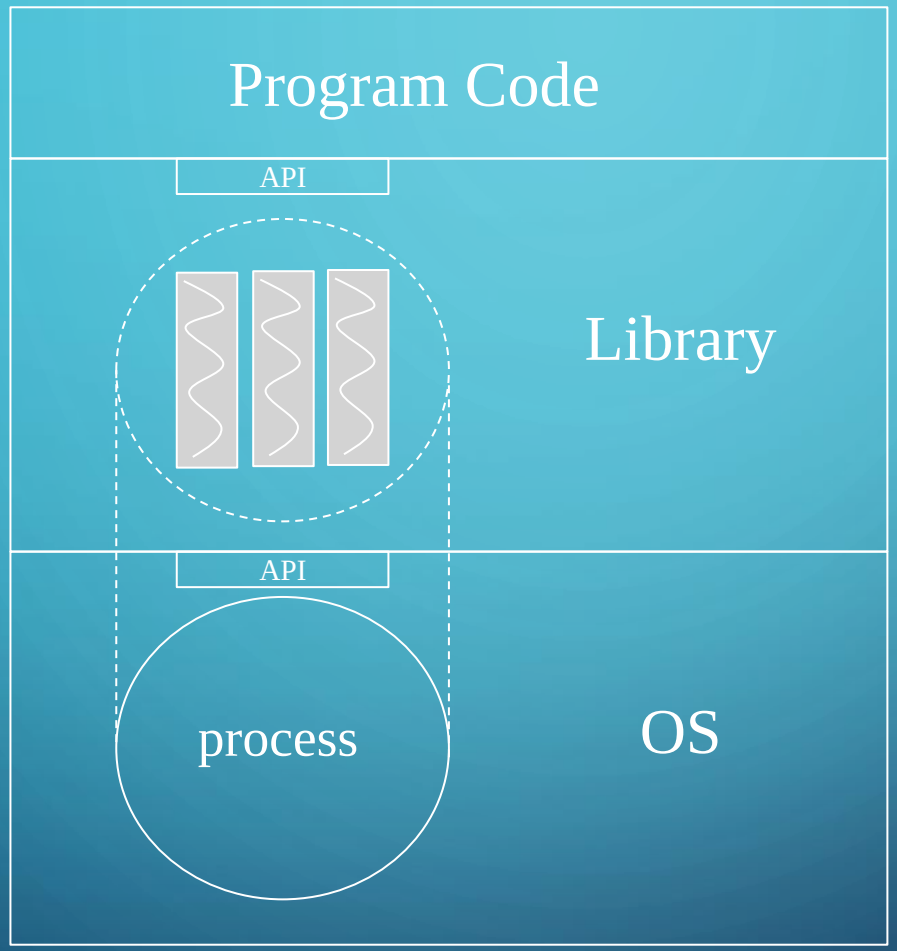
**Disadvantages**

- Sharing memory space (concurrent access to shared resources)
- Greater programming difficulty

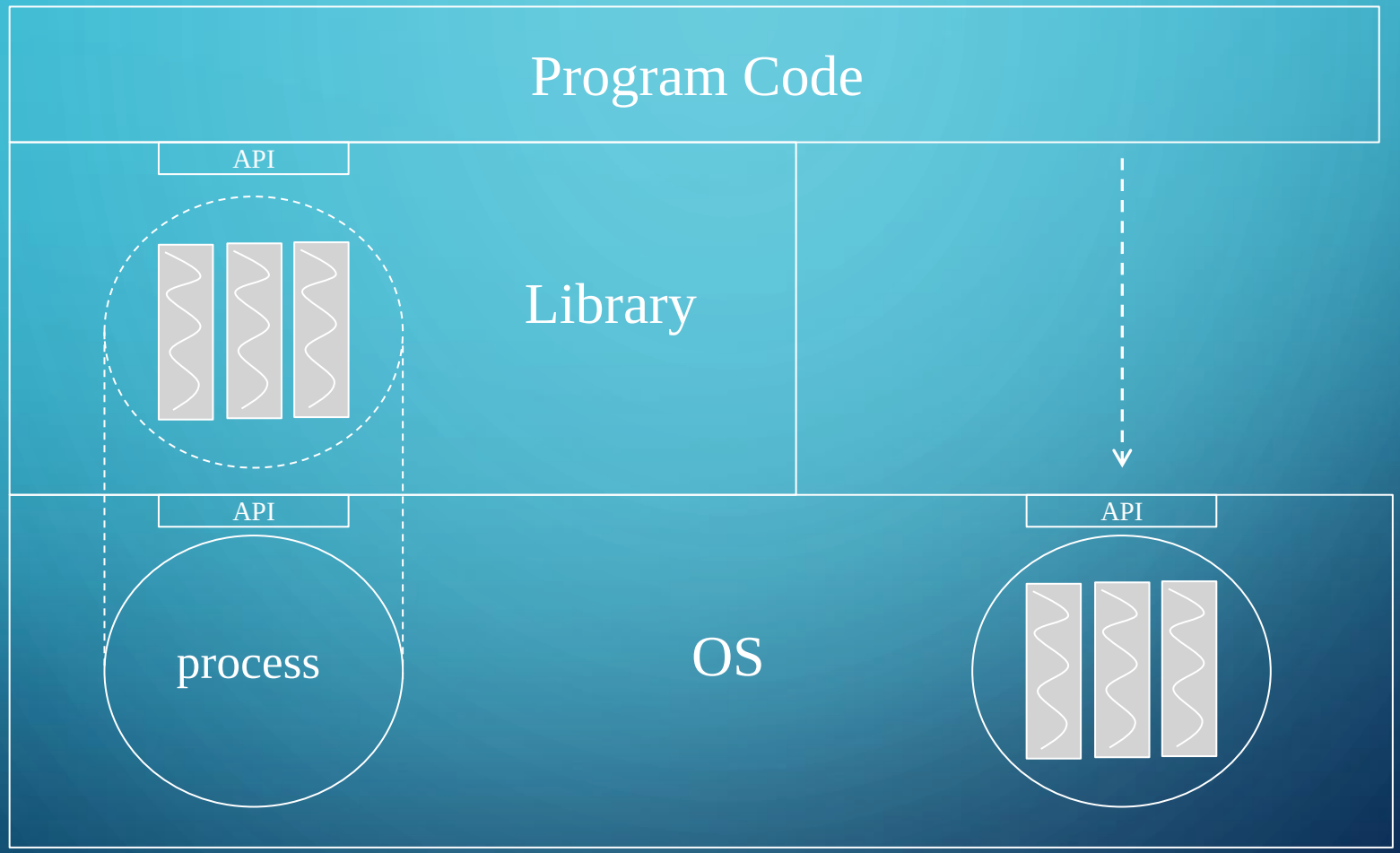
# Kernel Level Implementation



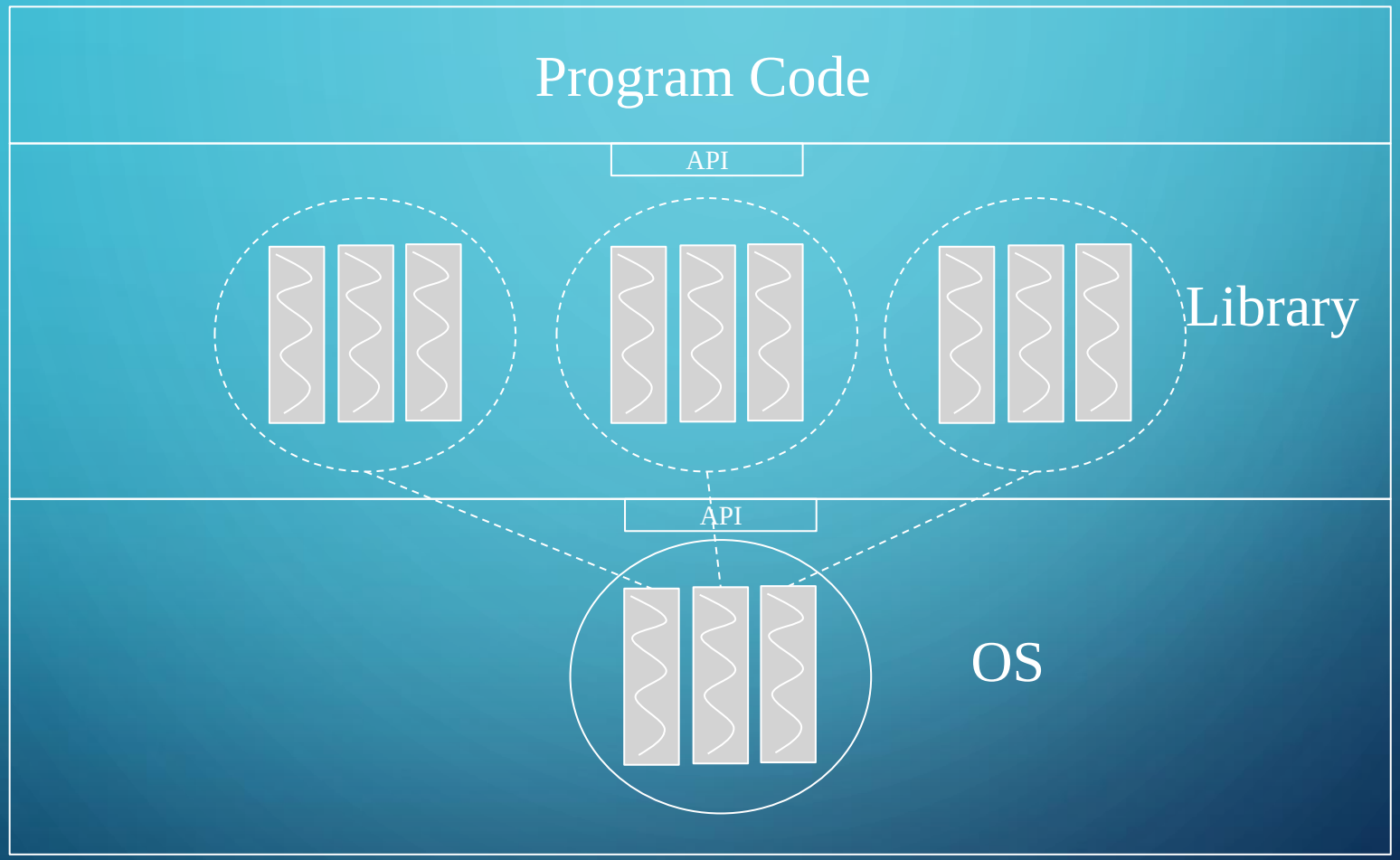
# User Level Implementation



# ULT and KLT combination



# ULT and KLT combination



# ULT and KLT comparison

| | Advantages | Disadvantages |
|---|---|---|
| ULT | • More policies available<br>• Management operations are more efficient (creation, switching, blocking) | • Program code tends to be more complex<br>• There is no real parallelism |
| KLT | • Program code is simpler<br>• Real parallelism is possible | • Management operations are less efficient |

**1 / 10 / 100 relationship**

- *ULT Thread creation: 1 time unit*
- *KLT Thread creation: 10 time units*
- *Process creation: 100 time units*

# CONTENTS

1. Process Management basics
2. Elements of a Process
3. Process lifecycle
4. Threads
5. **Process and thread scheduling**
6. System services for process and thread management

## Scheduling is…

***… to share processor time between processes that can be executed.***

- Scheduling Levels (classically)
  - *Short term:* assigns processor to ready processes (5 state model)
  - *Medium term:* includes process suspension in secondary memory (7 states model)
  - *Long term:* decides which processes are to be admitted to the ready queue
- **Scheduler:** module that decides which process to move from ready to running. (Short-term scheduler)
- **Dispatcher:** module that puts into execution the scheduled process.

# SEVEN STATE MODEL

## 5- PROCESS AND THREAD SCHEDULING

***Scheduling in the…***
***Short-term?***
***Medium-term?***
***Long-term?***

# SEVEN STATE MODEL

## 5- PROCESS AND THREAD SCHEDULING

- **Short-term scheduler:** decides which of the ready to run processes is going to use the CPU.
- **Medium-term scheduler:** decides which of the processes in memory (ready to run or blocked) is/are going to be swapped out. Its goal is to reduce the multiprogramming degree of the system (the remaining *in memory* processes have available more CPU time and more memory).
- **Long-term scheduler:** decides which of the processes created or swapped out is/are admitted into the memory of the system. It increments the multiprogramming degree of the system.

# Scheduling objectives

## Metrics about scheduling algorithms performance

### For processes (or threads)

- Execution (or return) time (Te)
- Waiting time (Tw)
- Response time (Ta)

### For the System

- Processor use (C %)
- Completed works rate (or Throughput) (P)

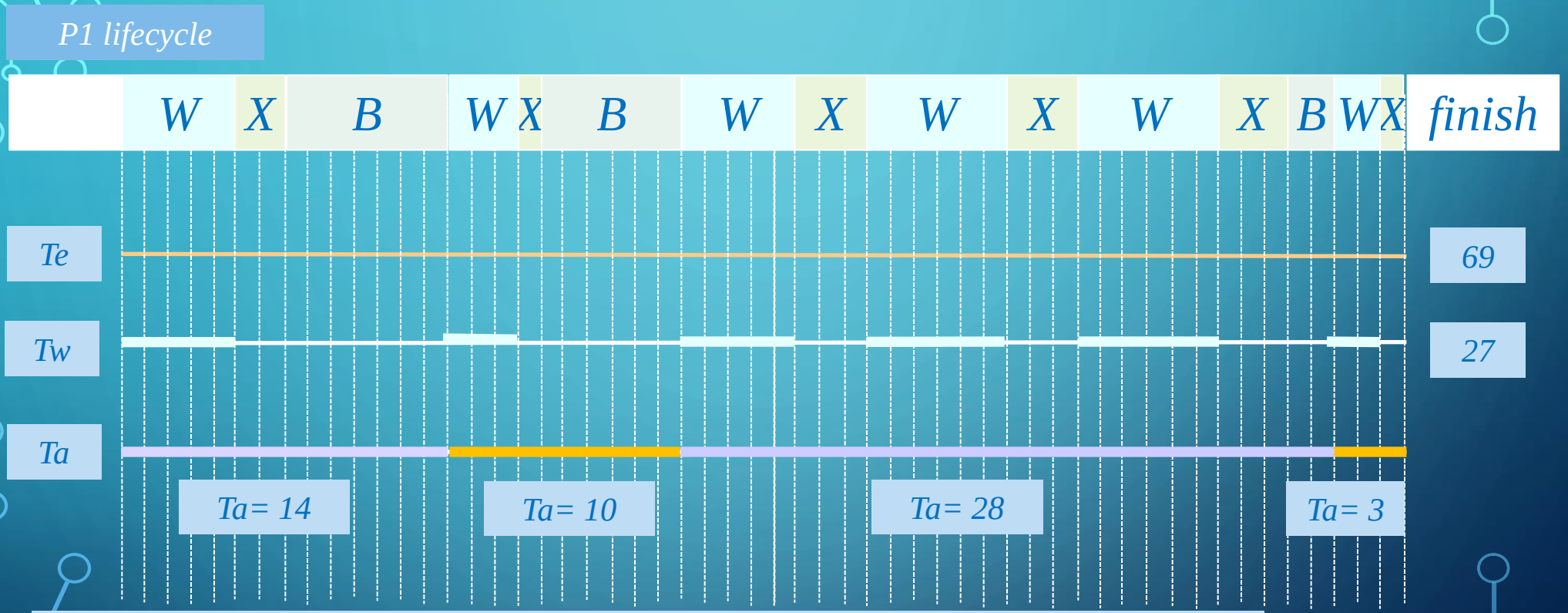
### General objective

- Minimize process metrics
- Maximize system metrics

### Another objectives…

- Impartiality
- Equitable processor sharing
- Processor efficiency (usage maximization)
- Execution predictability (real time)
- Resource usage balance
- Minimize response time variance
- Reduce context switching time

# Scheduling objectives




*Te: Execution Time. From New to Finish.*
*Tw Waiting Time. Time expended in Ready queue.*
*Ta Response Time. From a request until the process responds.*

# SCHEDULING POLICIES

- **Non-expulsive**
  - FCFS / FIFO (first come, first served) (first in, first out)
  - SJF (the shortest, the first)
- **Expulsive / preemptive**
  - SRTF (shortest remaining time first)
  - Round Robin (rotation shift)
- **+ Priorities**
  - Static
  - Dynamic
    - With aging
    - Encouraging interactive processes
- **Current policies:** *expulsive policy mix*
  *Multilevel Queues with dynamic priorities + round robin*

*¿Advantages disadvantages?*

*Which one would you choose for your design? Why?*

# FCFS scheduling policy or non-expulsive FIFO

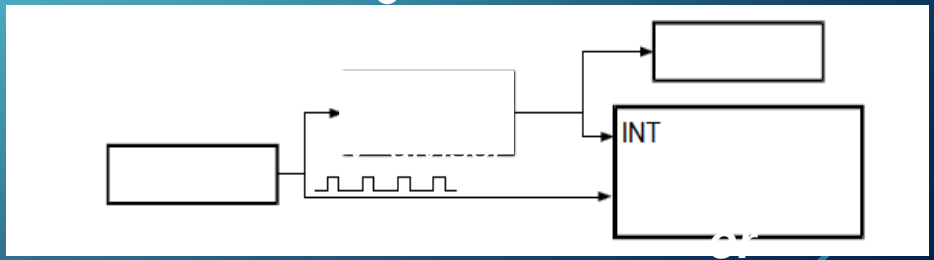
- *Ready* for execution processes are organized in a FIFO queue
  - When a process changes its state to *Ready* it goes to the end of the queue.
- A process leaves the CPU when:
  - The process *makes a system* call and as a result it changes to *Blocked*.
  - The process *ends*.
  - The process *voluntarily* leaves the CPU (yield).
- When the CPU is free the scheduler chooses the first process in the ready queue (FIFO).

# SJF scheduling policy (shortest first)

- We choose the process with **shorter total duration**.
  - Favors short jobs rather than longer ones.
- Requires knowing the runtime before running the job.
  - *Only useful in production environments where the same jobs are frequently run.*
- Non-expulsive policy (was used in batch processing systems).
- May cause long process starvation.
- Improves the average response time, but increases the variance.
- Deprecated Policy. Designed for batch processing system before multiprogramming.

# CYCLIC SCHEDULING POLICY OR ROUND ROBIN

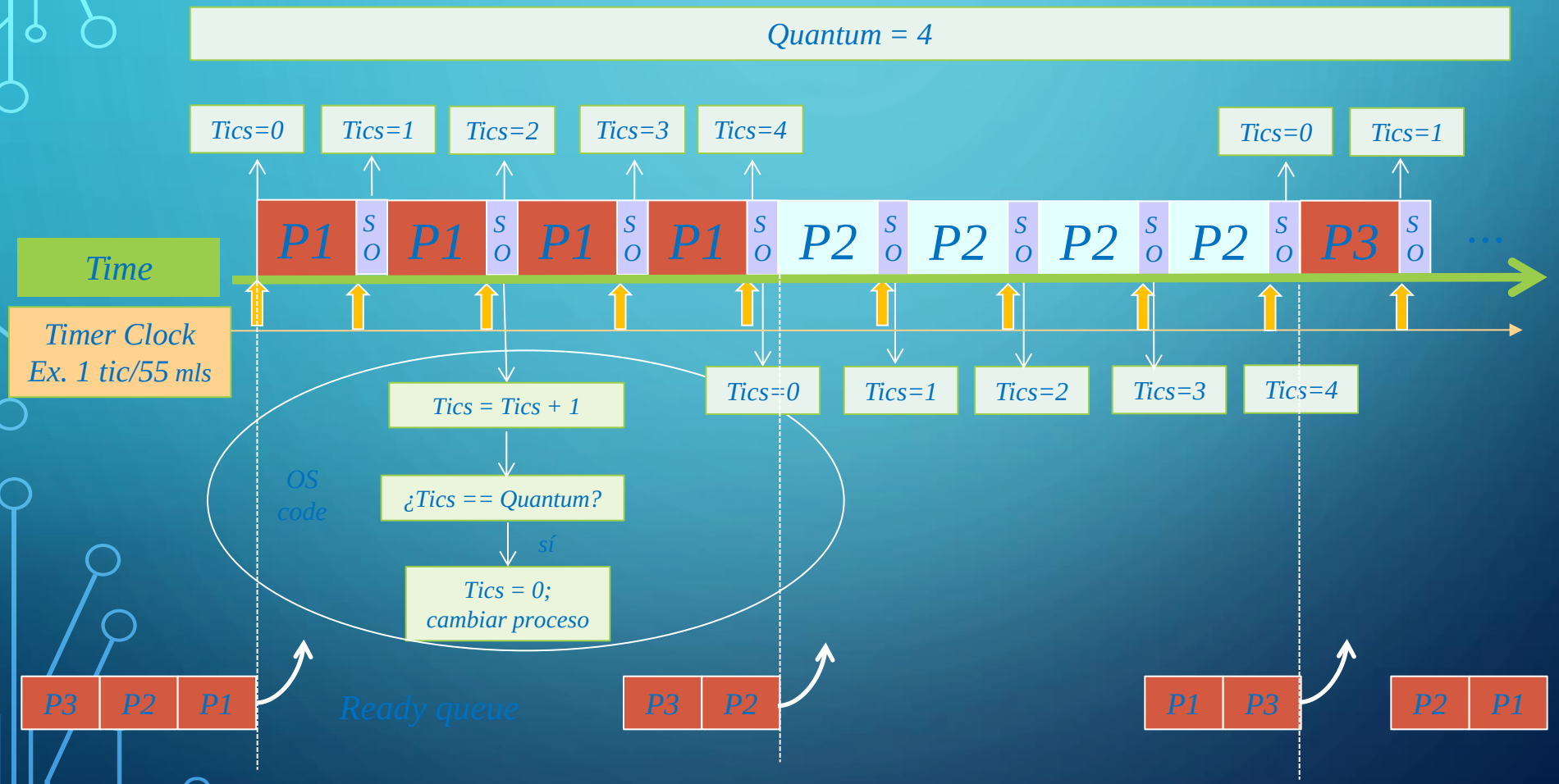
- **Get a fair allocation of processor time**
- The system assigns a *quantum*, or time slice, to the running process
- *Ready* for execution processes are organized into a FIFO queue
  - When a process changes its state to *Ready* it goes to the queue's tail
- A process leaves the CPU when:
  - The process *makes a system* call and as a result it changes to *Blocked*.
  - The process *ends*.
  - The process *exhausts its quantum*.




- When the CPU is free, the scheduler *chooses the first process* in the ready queue (FIFO).

# Cyclic scheduling policy or Round Robin

## *Preemption at the end of the quantum*



# Prioritized scheduling policy

- Each process is assigned a priority:
  - Automatically assigned by the system.
  - Selected by the user.
- The *ready queue* is sorted by priority. The scheduler chooses the highest priority process.
- Types of priority:
  - **Static**
  - **Dynamic**

# Prioritized scheduling policy

- **Static priority:**
  - **Static: fixed during the lifetime of the process.**
  - Easy to implement.
  - Introduces little overhead on the system.
  - It doesn't adapt to changes in the environment.
  - May cause *starvation*.

# Prioritized scheduling policy

- **Dynamic priority: varies during the process lifetime.**
  - More difficult to implement.
  - Enters more overhead on the system.
  - Avoids starvation.
- Example: priority aging.
  - An initial priority is assigned and increases as time passes without running. When running, it returns to the initial priority.
- Example:
  - It increases the priority when the process performs I/O operations and it decreases it when the quantum is exhausted.

# Scheduling policy for real-time systems

- The processes have to be executed in predetermined time.
- There are two types of processes:
  - Fixed term: run once at a specific time.
  - Periodic: repeatedly run after a period.
- The moment to run is associated with each process.
- Processes that have not reached their execution time are in a queue, the ones that have reached their execution time go to the ready queue.
- In critical systems each process is assigned a run time slot. The process must be executed *before* this time limit.

# *POSIX scheduling policy*

- POSIX Scheduling
- Applies to Processes and Threads
- Priority level between 0 and 31
- **It chooses the highest priority thread**
- Available policies:
  - FIFO
  - Round Robin
  - Other

***P**ortable **O**perating **S**ystem **I**nterface based on UNI**X***

# POSIX scheduling policy

- POSIX FIFO Scheduling
  - A FIFO queue per priority.
  - It expels the process if another one with higher priority arrives or if there is an I/O operation.
  - If other (process or thread) with higher priority changes to *Ready state* the current running process is expelled and it becomes the first of its associated priority queue.
  - A *Blocked* process changed to *Ready* goes to the tail of its priority queue.

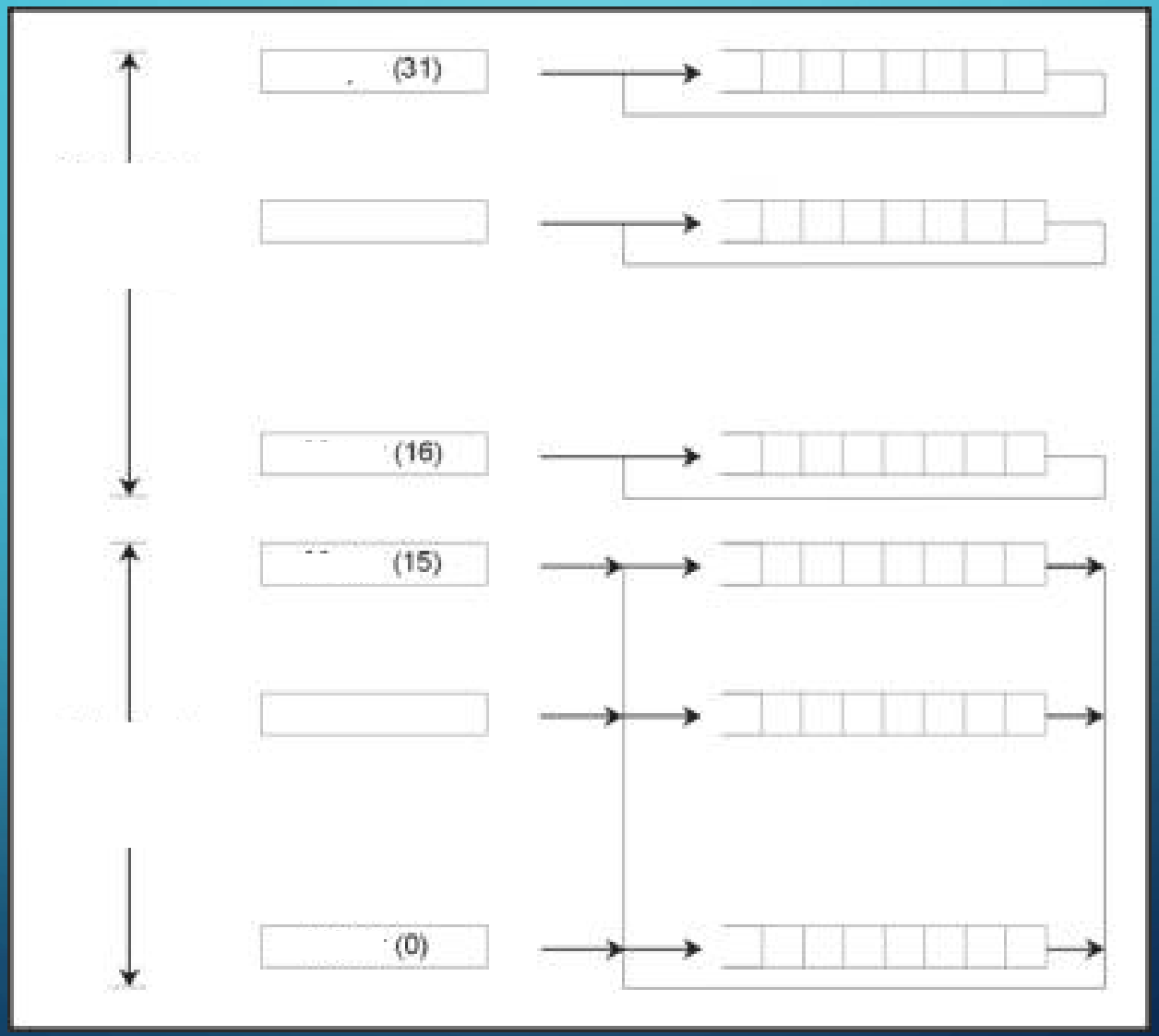
# POSIX scheduling policy

- Cyclic Scheduling in POSIX
  - A FIFO queue per priority, processes are assigned a time slice
  - When a process finishes its slice (quantum), it is inserted at the end of its queue
  - When a process is expelled by a higher priority one, it is introduced at the beginning of its queue *without initializing its time slice*.
- Another policy
  - POSIX standard allows to implement a different policy.
- All at once
  - **Each process chooses the policy**, so we have all policies at once.

# WINDOWS NT scheduling policy

- Cyclic scheduling with priorities and expulsion.
- Thread scheduling
- 32 scheduling levels (from 0 to 31 maximum)
  - 16 levels with real-time priorities (16-31). Fixed priority, FIFO.
  - 15 levels with varying priorities (1-15). They vary according to the thread behavior. Round Robin.
    - Decremented if the thread runs out of time slice
    - Increased by performing an I/O operation
  - 0 level for system
  - Process and Threads have a base priority

# WINDOWS NT scheduling policy



# Multiprocessor scheduling

Running the OS on multiprocessors

- **Asymmetric multiprocessing**
  - A processor dedicated to running the OS
  - In can be slow but the OS is simpler
  - OS operations are not concurrent
- **Symmetric Multiprocessing**
  - The OS runs on all processors as needed
  - Concurrency issues inside the OS services
  - OS code is more complex
  - More efficient execution

# Multiprocessor scheduling

Scheduling criteria

- **Processor affinity**
  - Tendency to assign the same processor to a process for maximizing the cache information
    - (each processor has a cache)
  - Strict affinity
    - Possibility of indicating on the affinity process to a processor or group (for critical processes)
- **Load sharing**
  - Load Balancing processor execution.

# Multiprocessor scheduling

Two possibilities for process scheduling

- Unique queue (for Ready processes) for all processors
- One queue per processor

# *Multiprocessor scheduling*

**Single queue scheduling**

- If a processor is free:
  - Select a process from the queue
    - By priority
    - By affinity (is affine if that processor has executed it the last time)
- If there is NOT free processor:
  - A process changed to *Ready* (at birth or from *Blocked*) is assigned to a processor IF an affine processor executes a lower priority process (that is expelled)

**Advantage** → Load Balancing

**Drawback** → Single queue needs *protected access,* implies bottleneck

# Multiprocessor scheduling

## One queue per processor scheduling

- Each process is assigned to a processor
- Each processor executes process from its own queue

## Advantage

- Single queue bottleneck disappears

## Drawbacks

- Load Rebalancing is not automatic
  - Periodically
  - If a processor is freed
- Processes are changed from one queue to another.
- It takes into account affinities (strict and cache)

# Multiprocessor scheduling

**Thread scheduling policies**

- Not taken into account the relationships between same process' threads, OR
- It takes into account the relationships between threads
  - Shared time
    - Applications are multiplexed in time
    - Time is assigned to each application (set of threads) in a processor. All threads are running on the same processor.
  - Shared space (co-scheduling)
    - Applications are multiplexed in space and time
    - Application's threads are distributed among groups of processors
    - Applications are also multiplexed in the time

Both types can use shared or local queues

# Multiprocessor scheduling

**Current Systems**

- Symmetric Multiprocessing
  - The OS runs on all processors
- Scheduling
  - Windows (client) → One global queue
  - Windows (server) → Local queues
  - Linux → Local queues

# CONTENTS

1. Process Management basics
2. Elements of a Process
3. Process lifecycle
4. Threads (or lightweight Processes)
5. Process and thread scheduling
6. **System services for process and thread management**

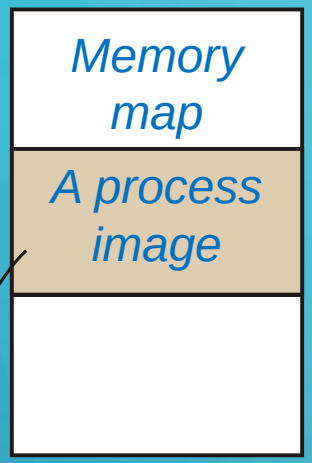
# POSIX: FORK() SYSTEM CALL

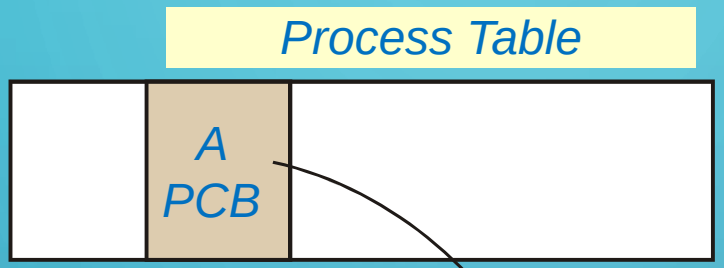
Services for Process Management

- Process Creation
  - **fork, execl, execv, execle, execve, execlp, execvp**
- Process Termination
  - **exit, wait, waitpid**
- Process identification
  - **getpid, getppid, getuid, geteuid, getgid, getegid,**
- The environment of a process
  - **getenv**

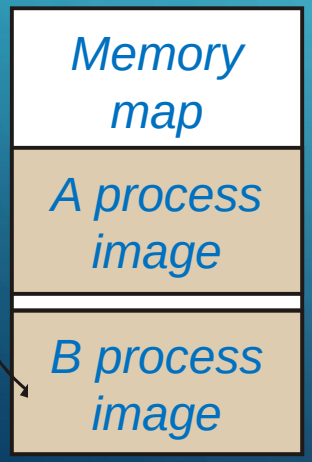
***P**ortable **O**perating **S**ystem **I**nterface based on UNI**X***

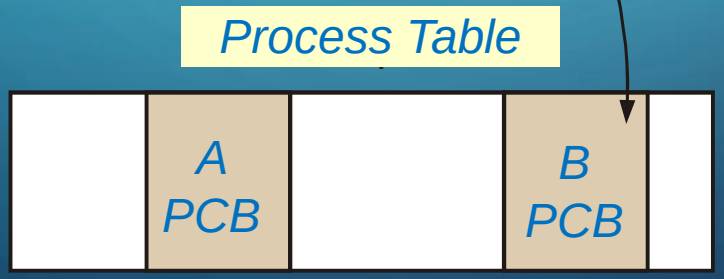
# POSIX: FORK() SYSTEM CALL




'A' process performs a fork() system call which creates a child process 'B'




New PID
New process image
Different return value (0 in child)

# POSIX: FORK() SYSTEM CALL

**Syntax:** *pid_t fork();*

**Operation:** Clones the process making the call

**Implications:**

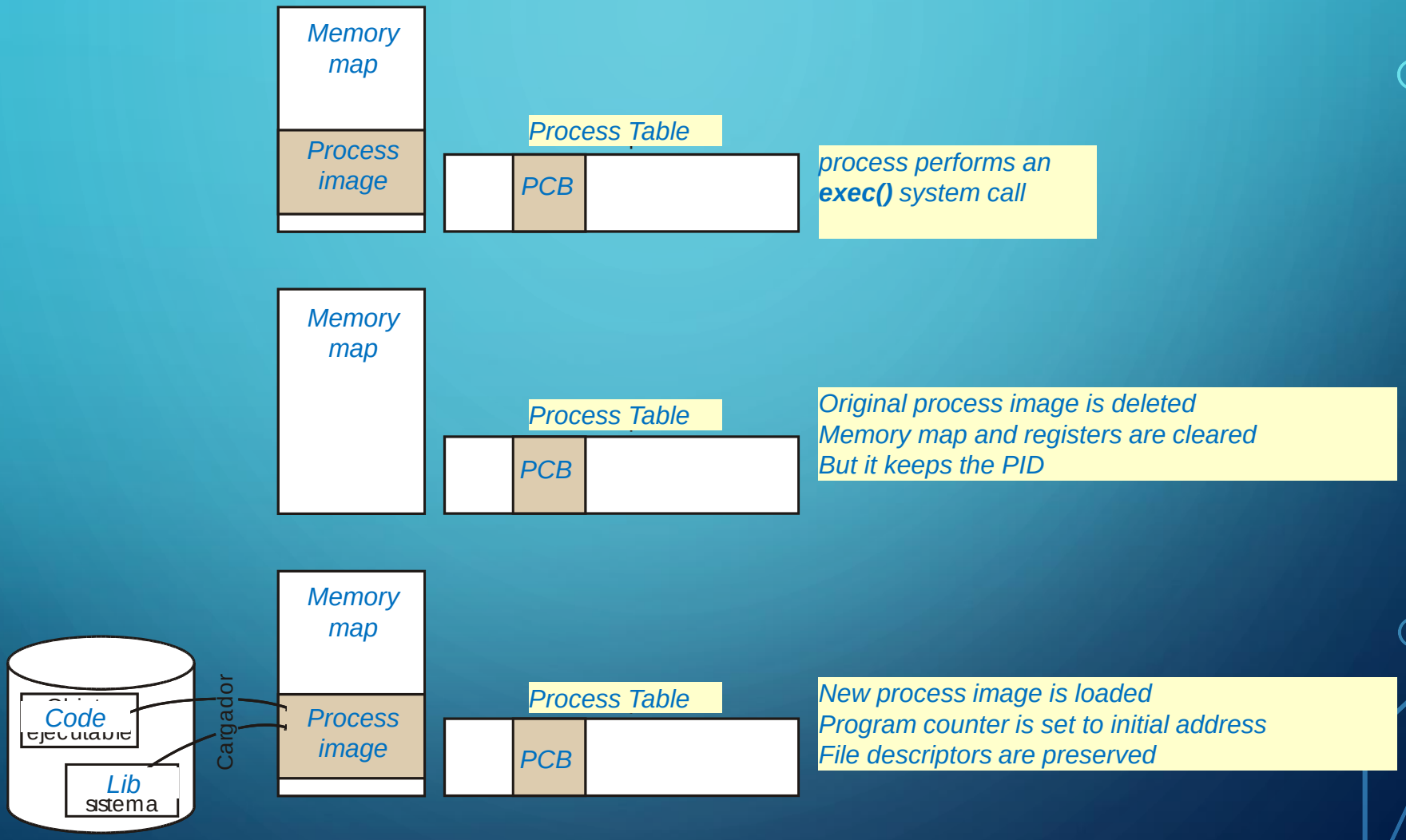
- The process making the call becomes the parent. The new process is the child.
- Process image and the PCB are duplicated
- Parent and child continue execution at the same point
- Some PCB's child values will be different
  - PID of the child
  - Addresses of the memory segments of the process image
  - Execution time counters and signals in the child are reset
- fork() return value is *0 in the child* and the *child's PID in the parent*

# POSIX: FORK() SYSTEM CALL

**Example**

```
main()
{
  if (fork() != 0)
     printf("I am the parent\n");
  else
     printf("I am the child\n");
}
```

# POSIX: FORK() SYSTEM CALL



# POSIX : EXEC() SYSTEM CALL

**Syntax:** exec()

**Operation:** Change the program that is running a process by another one indicated as parameter (an executable file).

**Implications:**

- Liberation of the process image memory
- Reassigning new address space to load the executable file
- Fill the PCB with initial values for registers and new memory segments

# POSIX : EXEC() SYSTEM CALL

**Example**

```
main()
{
   if (fork() == 0)
       execlp("ls","ls", "-l",NULL);
   printf("The parent continues alive\n");
}
```

**Command interpreter (Shell)**

```
Repeat
  read command line
  do fork()
  If this process is the child do exec(command line)
  If this process is the parent
       If the command line does not end with "&"
                wait() until child finishes
Until command line = "exit"
```

# EXERCISES

```
void main() {
   int a = 0;
   if (fork() != 0){
      a = a+1;
      printf(a);
   }
}
```

```
void main() {
   int a = 0;
   if (fork() != 0){
      a = a+1;
   }
   printf(a);
}
```

What is the result of its execution?

# EXERCISES

How many processes are created when you run this code?

```
for (i=0; i < n; i++){
   fork();
   execlp("ls", "ls", "-l", NULL)
}
```

```
for (i=0; i < n; i++){
   fork();
}
execlp("ls", "ls", "-l", NULL)
```

# EXERCISES

How many processes are created when you run this code?

```
for (i=0; i < n; i++){
   if (! fork()) execlp("ls", "ls", "-l", NULL)
}
```

```
for (i=0; i < n; i++){
   if (fork()) execlp("ls", "ls", "-l", NULL)
}
```

# EXERCISES

**Which of the following statements are correct?**

a) The content of a memory image of a process remains unchanged throughout the execution

b) The process execution stack is used to store local variables and parameters each time a method (or function) is invoked

c) The program code that has been loaded into memory is part of the memory image of the process

d) None of the above

# EXERCISES

*According to the basic model lifecycle (New-Ready-Running-Blocked-Finished)*

**Which of the following statements are correct?**

a) A running process does not always have a PCB

b) A running process must go through every state of its lifecycle before it finishes

c) The Ready processes queue stores the PCBs of the Ready state processes

d) None of the above

# EXERCISES

*According to the basic model lifecycle (New-Ready-Running-Blocked-Finished)*

**Which of the following statements are correct?**

a) A running process will change to *Ready* state when executing a system call that involves a slow I/O operation

b) When a process is born, the system must assign a PCB, which stores the process image, among other things

c) A *Blocked* process will change to *Running* when the event it was waiting for finally takes place

d) None of the above

# EXERCISES

**Which of the following statements are correct?**

a) When an interruption occurs, the hardware keeps track of the Program Counter (PC) and places on it the OS routine address for interruption handling

b) The interruption handling routine keeps the remaining registers before it manages the interruption

c) In case an interruption triggers a process change, the scheduler and dispatcher will be called

d) None of the above