Operating System Process and Resource Management

Basic System Elements

Course Objectives

- ✓ Understand the components of an operating system
- ✓ Discover the problematics and mechanisms of operating systems
- ✓ Take advantage of the benefits of the OS

Course Plan

1. Definitions
2. Process Operations
3. Mechanisms
4. Communication

1. Definitions

1. Definitions

Introduction

- The course does not present an operating system in detail, but aims to provide a general description of how an operating system manages a computer

- The course also proposes the presentation of concurrent algorithms, useful for writing programs with data shared by several users

- The course will take the position of a system programmer, with emphasis on the presentation of algorithms and their uses
  - We will always make the link between machines and programming because architecture and systems are closely related

1. Definitions

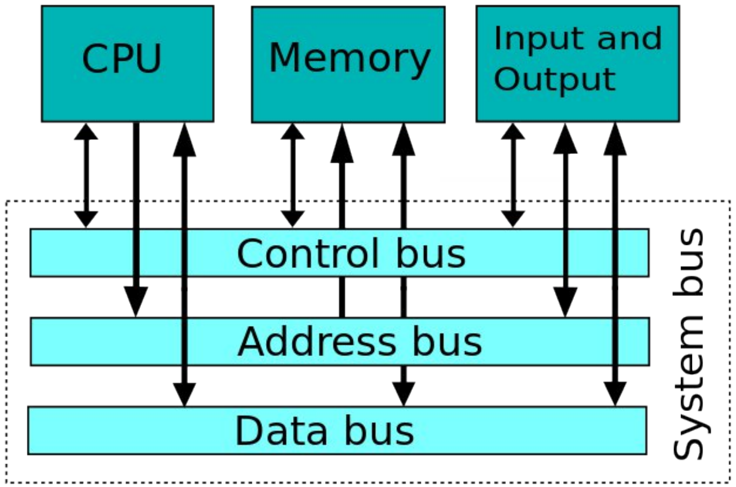
Introduction

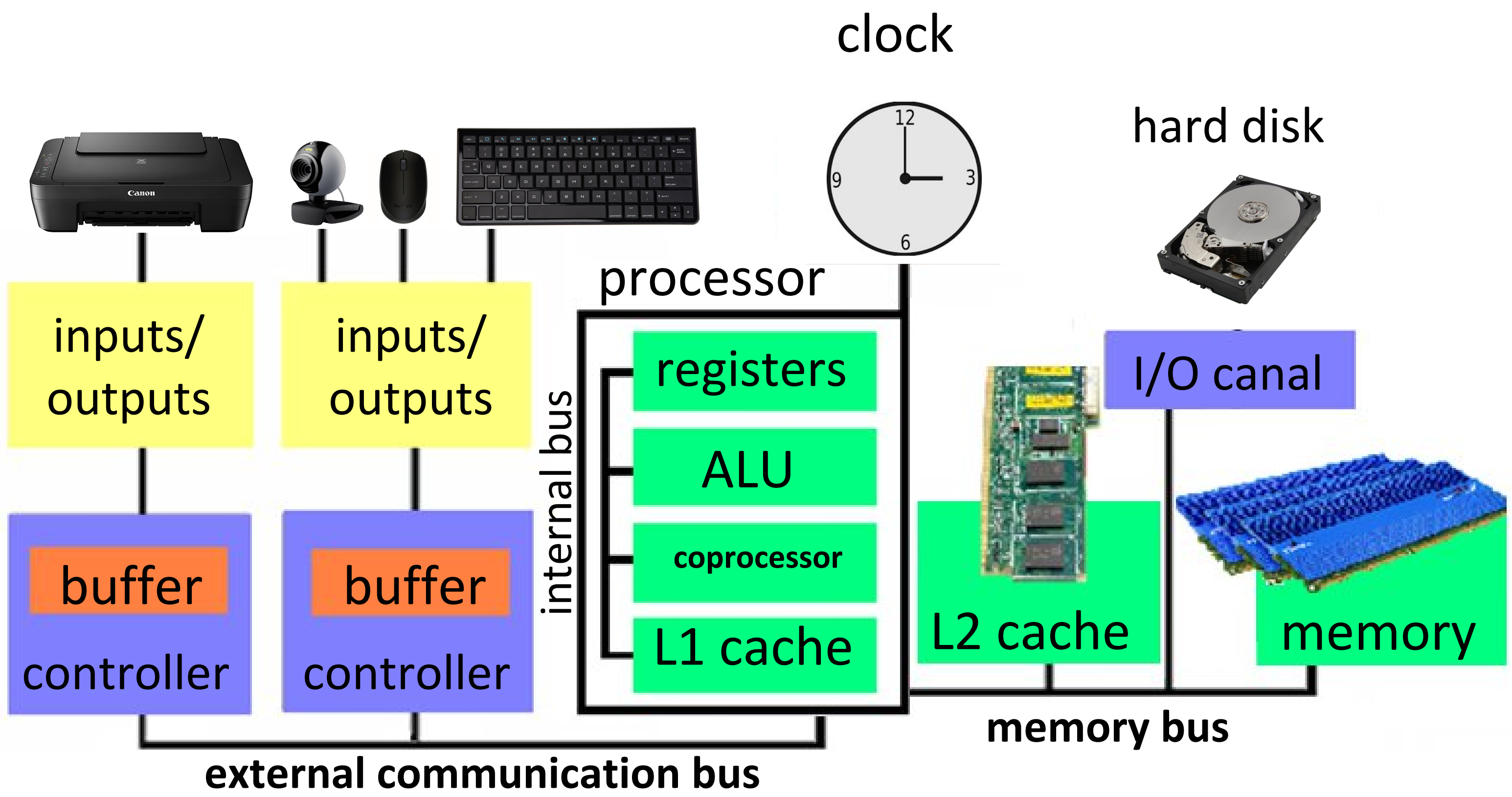
- The operating system is a program that abstracts from the hardware:
  - It organizes and carries out the device management, the CPU, the memory, *etc*. while hiding them to the user
  - It determines the conditions of use of the machine such as time sharing (real time systems)
  - It manages data protection (passwords/privileges)
- The user is only concerned by the essential: the result of his/her actions
- A simple computer consists of:
  - A processor (microprocessor)
  - A volatile memory unit for storing data
  - Peripherals

1. Definitions

Computer architecture

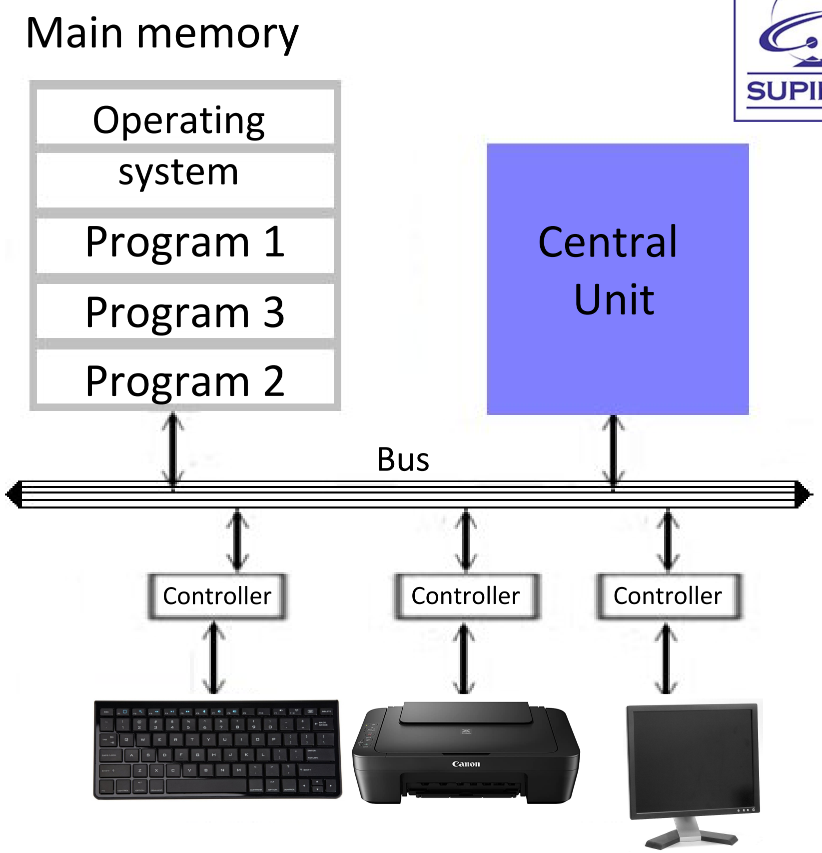
- These components communicate with each other through a communication bus





1. Definitions

Computer architecture



- The central unit allows calculations and processing
- The main memory contains:
  - The OS
  - The user programs
  - The commercial programs loaded
- The peripherals that can be reached via the controllers, are managed by the system according to interrupts
- The data are available in the buffers

1. Definitions

Programs

- There are 2 operating modes for the processors:
  - A **user** mode
  - A **kernel** mode (supervisor)

- These modes are used by the operating system to allow control and access to the computer's resources

- A program is a finite set of exclusively static instructions
  - A program (written in a high-level language) is composed of data structures and processing structures
  - The operating system consists of a set of programs without which the hardware would be unusable
  - A program is a task to be executed by the system

1. Definitions

Programs

- An executable program after link editing is stored on the disk
- Compiling gives a production line transforming a program written in high-level language into a low-level program (in machine language, an executable)
- When loaded, this program is placed in main memory to be executed
- At each execution step of a program, the content of the ordinal counter registers evolves
  - At the same time the content of the main memory is modified with **WRITE** or **READ** functions

1. Definitions

Programs

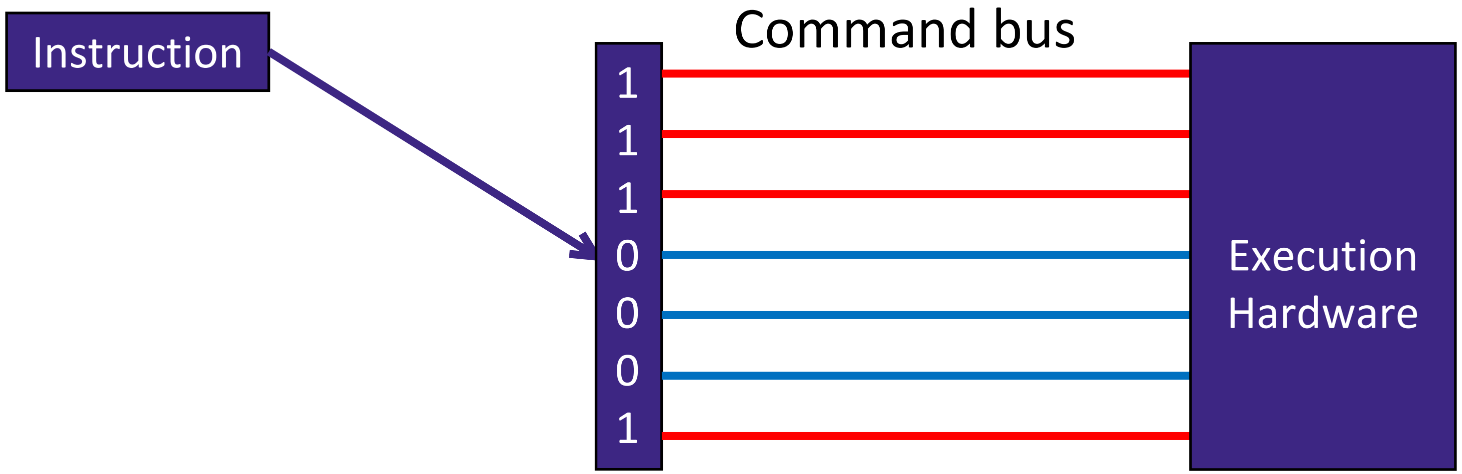
- Programs are executed in **user mode**
- On the other hand, for all programs that use operations with resources internal to the machine (for example, file management operations to modify data) the **user mode** calls the operating system and switches to **kernel mode**
  - These mode changes are **system call** operations
- All access to the machine's resources and data is controlled by the operating system
  - The **kernel mode** represents a fundamental protection for the system
  - The **system call** uses a system function (a routine) composed of instructions

1. Definitions

Programs

- An instruction is a binary command (the translation of high-level code)

- The execution of an instruction corresponds to a software command on the hardware in the form of a finite set of electrical impulses:

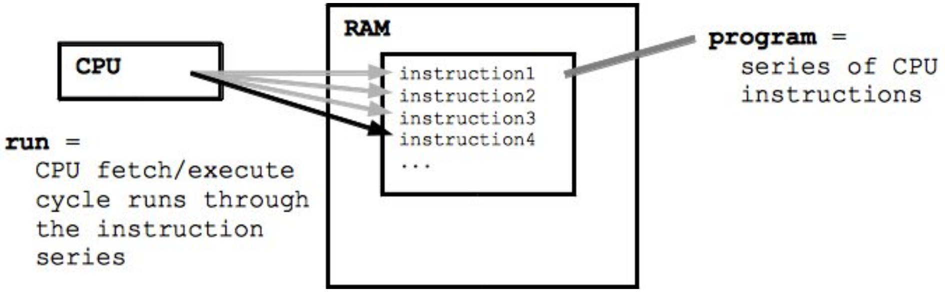


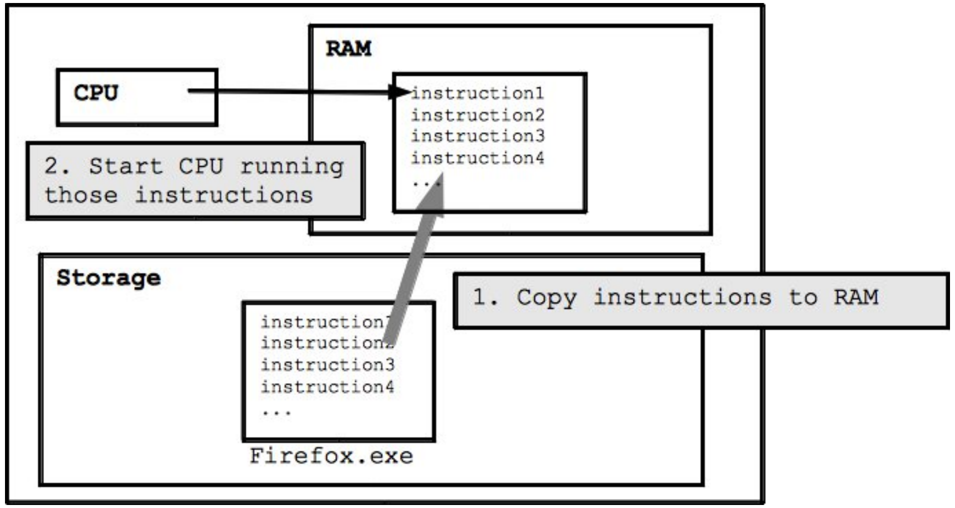
- The instruction depends on the hardware; an instruction set is the finite set of instructions

1. Definitions

Programs

- The instructions are classified in 5 categories:
  - Arithmetic and logic (subtraction, addition, and, or, *etc.*)
  - Floating (operations on floating numbers)
  - Data transfer (memory-registers)
  - Control (procedure calls, jumps, *etc.*)
  - System (traps, operating system call, *etc.*)





1. Definitions

Programs

- The management of the sharing of the physical machine and hardware resources must address the issues of sharing the single processor, sharing peripherals and sharing main memory:
  - Which of all the programs loaded in main memory will be executed?
  - Among all the programs which one will be able to get access to the peripherals? And when?
  - How to allocate the central memory to the different programs (users, commercial, *etc.*)?
  - How to ensure the protection between several user programs?
  - How to protect the operating system from user programs?

- To ensure protection, one program never accesses any part of the memory allocated to another program

1. Definitions

Programs

- The role of the operating system is to facilitate the user's access to the physical machine:
  1. To perform an I/O operation, does the user need to know how the device is managed?
  2. To execute a program, does the user need to know how it is loaded into main memory and does he need to know how the allocation of memory words is managed?
  - These functions are complicated and tedious for the user

- The operating system provides an interface to encapsulate and use the hardware functions
  - This interface is composed of a finite set of (primitives) functions managing the hardware resources and providing access to the user

1. Definitions

Programs

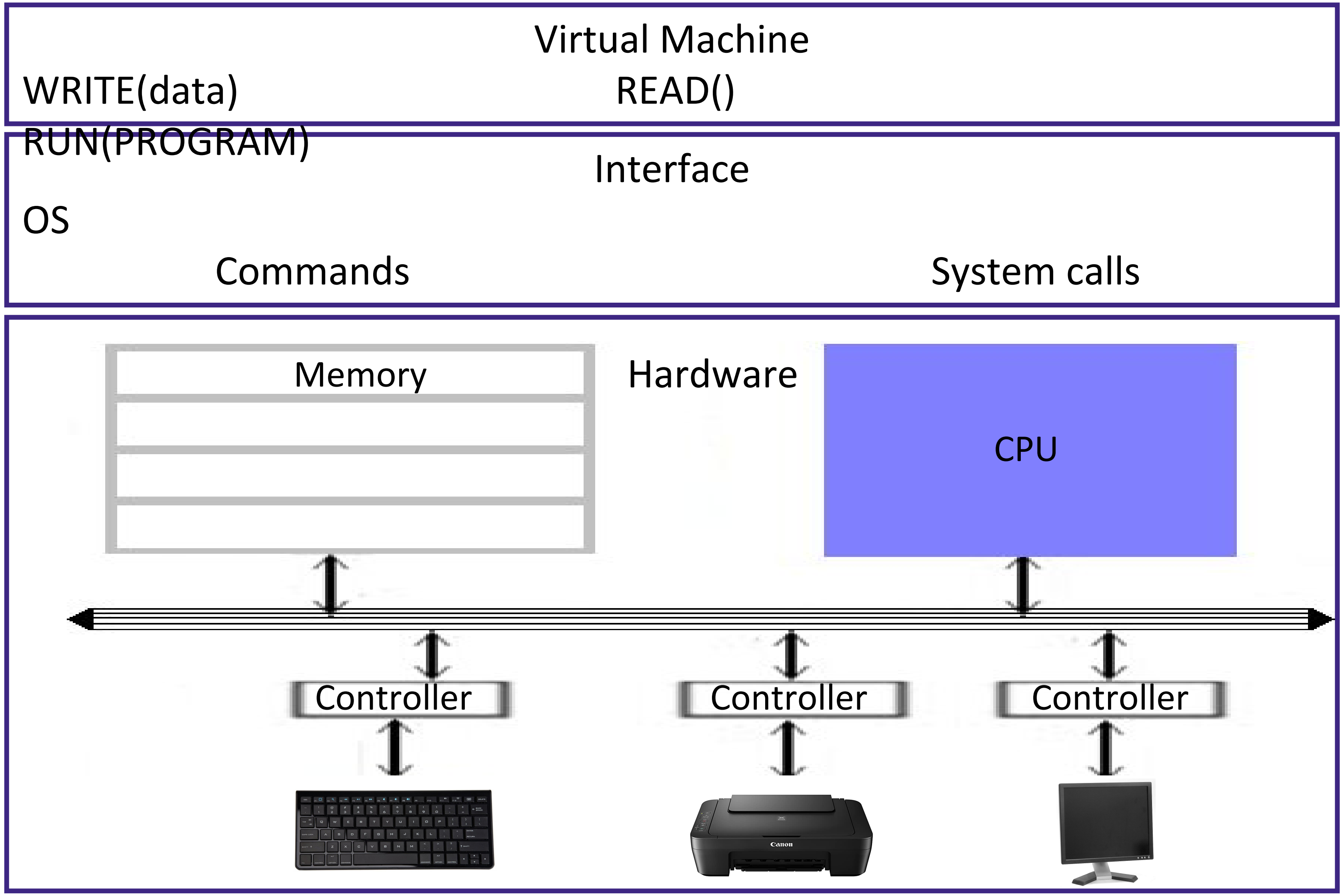
- For example, the **READ** or **WRITE** functions (like those seen in the algorithms) allow the user to perform an I/O action without worrying about the actual controls and connectors (hardware) of the requested device

- The finite set of operating system primitives creates a virtual machine on top of the physical machine that is easier to use and more user-friendly

- We distinguish 2 types of primitives:
  - System calls
  - Commands

1. Definitions

Programs



1. Definitions

Programs

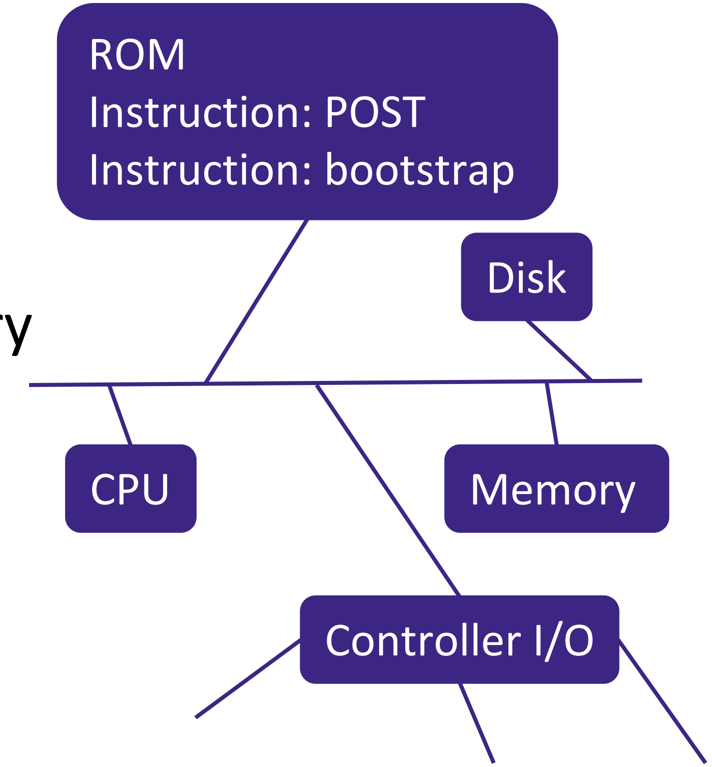
- System resources are used for communication between the processor and other components of a computer
- 3 common system resources:
  - Interrupt requests (IRQ)
  - Input/output (I/O) port addresses
  - **DMA** (Direct Memory Access) for direct access to memory (RAM) from peripherals
- The processor will be completely disconnected from the data and address buses; the peripherals are thus autonomous on the RAM by using the buffers

1. Definitions

Programs

- The operating system, loaded at boot time via the bootstrap (BIOS), interfaces with the resources defined by the BIOS

- At startup, the BIOS uses the POST (Power On Self Test) to test the components, then it does:
  - Installation of the interrupt vector table in main memory
  - Loading of the operating system files in main memory from the disk



1. Definitions

Programs

- The BIOS is a small program located in the ROM and EEPROM of the motherboard
- The BIOS is the first program loaded into memory to check the simple devices (keyboard, monitor, *etc*.)
- Now some operating systems can connect and check devices without going through the BIO
- Flashing (because now ROMs are replaced by flash memories) the BIOS indicates to use a setup to set the parameters for checking and controlling the devices

1. Definitions

Programs

- The operating system is a software layer placed between the hardware machine and the applications:
  - It allows users to develop without worrying about the details of hardware operation and management
  - It interfaces with applications through the primitives it offers (system calls or services, and commands)
  - The operating system can be broken down into several major functions that constitute its executive model

- **Concurrency management**:
  - Several programs coexist in main memory, they want and can communicate to exchange data
  - Access to shared data must be synchronized to maintain consistency

1. Definitions

Programs

- **Input/output management**:
  - The system makes the connection between the high-level calls of the programs (for example in C language **printf()** and **scanf()**) and the low-level operations of the peripheral (keyboard, screen, *etc.*)
  - Note that it is an input/output program (driver) that makes this correspondence

- **Memory management**:
  - The system manages the allocation of the main memory between the various programs
  - This management is done according to the principle of virtual memory (paging) since the physical memory is often too small to contain all the programs

1. Definitions

Programs

- **File management (external objects)**:
  - The main memory is volatile; thus, the management and access to data are based on the notion of files and file management systems (FMS)
  - Therefore, all the data that must be kept after the machine is shut down will be stored in a mass memory (like an internal hard disk)

- **Processor management**:
  - The system manages the allocation of the processor to the different programs
  - The allocation is done with a scheduling algorithm that plans the executions
  - Depending on the type of operating system, the scheduling algorithm meets different objectives

1. Definitions

Programs

- **Protection management**:
  - The system provides mechanisms to ensure that information and resources (CPU, memory, files) can only be used by programs that have been granted the necessary rights

<table>
<tr><td>High-level applications</td><td colspan="4">Firefox, Notepad, Emacs, programs, user, etc.</td></tr>
<tr><td>Low-level applications</td><td colspan="4">Compiler, links editors, etc.</td></tr>
<tr><td rowspan="3">Operating system</td><td colspan="2">System calls</td><td colspan="2">Commands</td></tr>
<tr><td>concurrency</td><td colspan="2">protection</td><td>external files</td></tr>
<tr><td>processor</td><td colspan="2">memory</td><td>inputs/outputs</td></tr>
<tr><td colspan="5">Interrupt routines</td></tr>
<tr><td colspan="5">Physical computer</td></tr>
</table>

1. Definitions

Programs

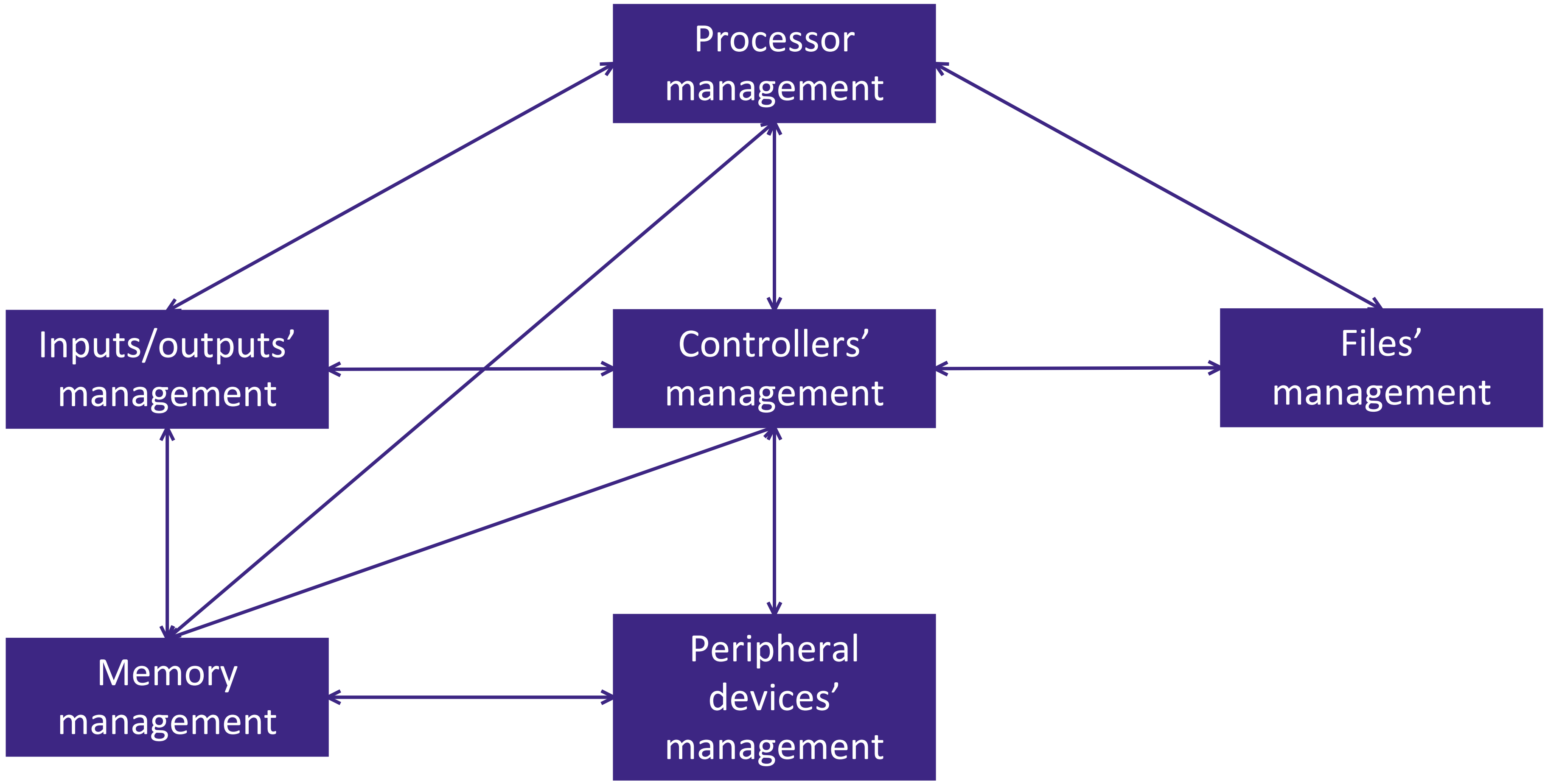
- The operating system is organized as a modular program, this design allows it to be relatively **portable** (hardware independent)

- Programs (drivers) manage the peripherals at the physical level and propose procedures for the different types of peripheral devices

- The part of the system that uses these drivers does not depend on the hardware and can be used on another machine (it is enough to rewrite the low-level modules)

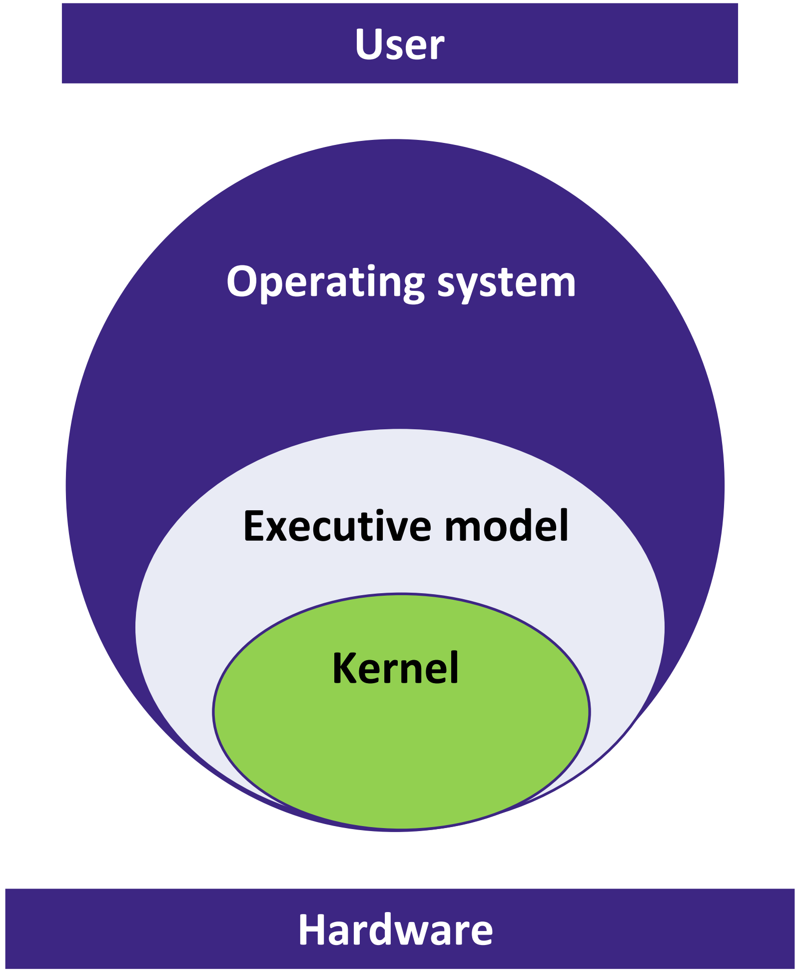
1. Definitions

Programs



1. Definitions

Programs



- A resource is an entity that a program needs to run
  - Hardware resource (CPU, peripheral devices, *etc*.)
  - Software resource (variable, array, file, *etc*.)

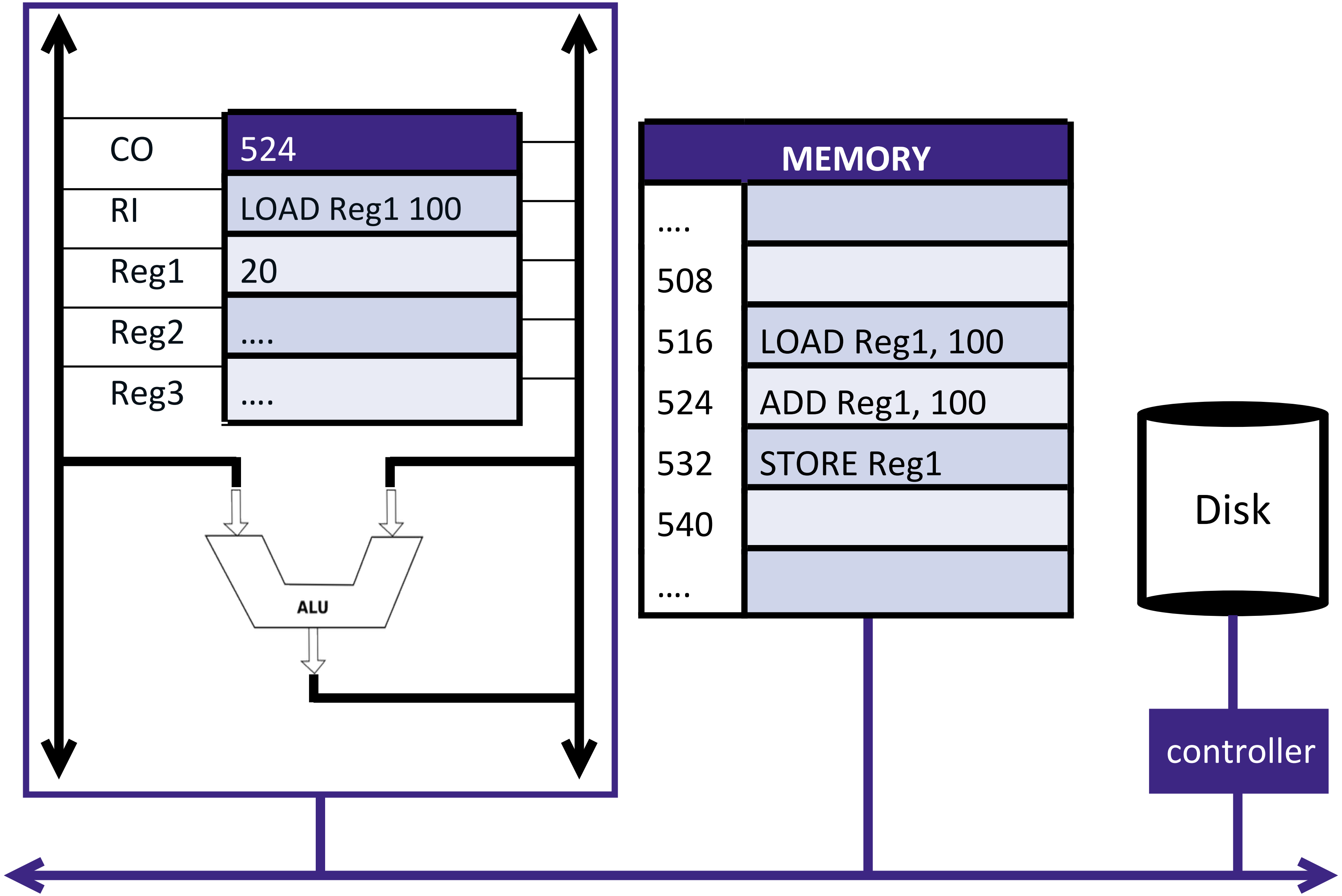
- A resource is characterized:
  - By a state (free/busy resource)
  - By its number of access points

- Executive services: layer allowing access to power, windows, plug and play, I/O, memory, *etc*.

- The system surrounding the kernel (processor controller) allows the isolation of fundamental processor and hardware I/O operations, from application processes

1. Definitions

Programs



1. Definitions

Programs

- The program to be executed is placed in main memory from address **516** (see previous diagram)

- The processor has begun the execution of the program: the first instruction has been loaded into the instruction register (**IR**), and the ordinal counter (**PC**) contains the address of the next instruction to be executed (**524**)

- When the current instruction will have been executed, the processor will load in the IR the instruction pointed by the PC (**ADD Reg1, 100**) and the PC will take the value **532**

- The execution of the previous instruction will change the contents of the PSW register (status register) since it is an arithmetic instruction: flags, nullity, *etc*.

1. Definitions

Programs

- The execution of concurrent programs for an operating system is:
  - The manipulation (create, destroy, start, stop, *etc.*) of concurrent structures (tasks or processes)
  - The implementation of communication and synchronization systems between these concurrent structures
  - The modeling of the real execution of tasks or processes on a parallel architecture or not

1. Definitions

Processes

- A process is a dynamic entity associated with the sequence of actions performed by a program (set of instructions and data accessible in memory, in a protected **address space**)

- The process is the snapshot state of the processor and the memory during the execution of a program

- The process is a set of tasks sharing global variables

- A process is a running program with an associated **processor environment** (PC, PSW, RSP, general registers) and **memory environment** called process context

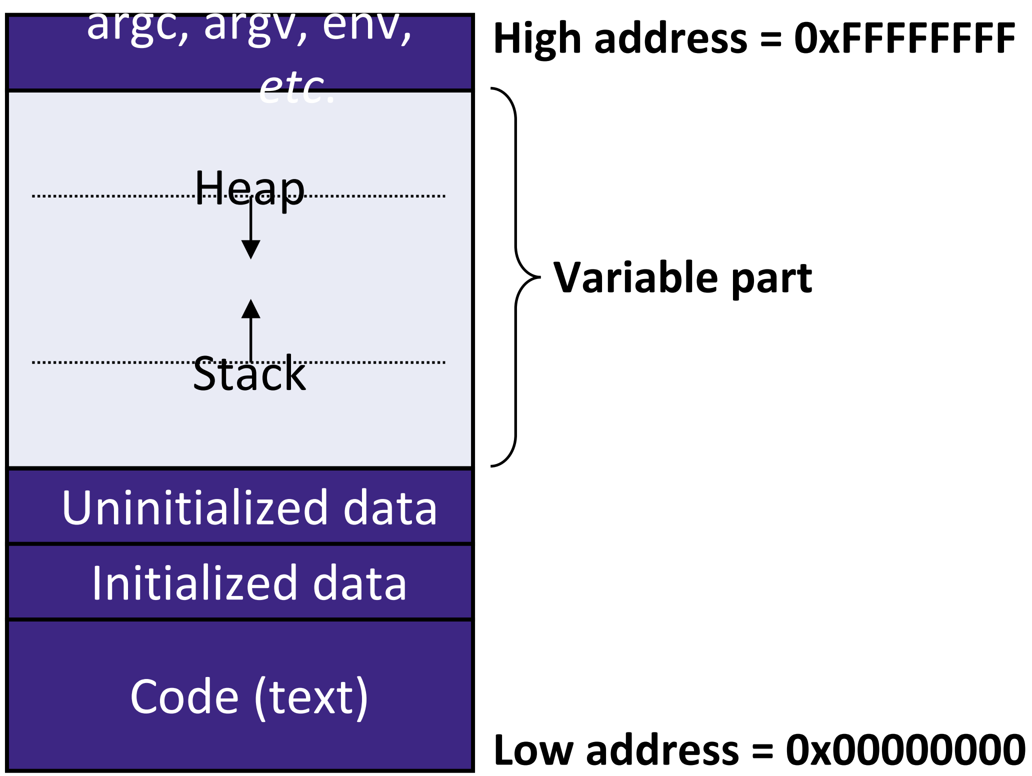
- A process is an abstraction of data defined by a state and a behavior

1. Definitions

Processes

- The process has an address space in memory to execute, and in which it can read and write

- This area is divided into several parts:
  - The code (text)
  - The data
  - The stack which allows to store the state of the processor during a function call
  - The heap which allows to store the variables
  - The arguments passed as parameters

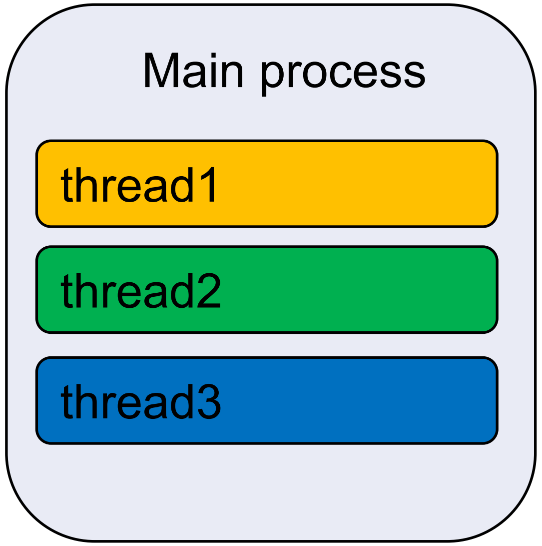


1. Definitions

Processes

- Initial processes (also called heavy-weight processes) have many disadvantages:
  - A heavy management by copying to create new processes
  - A waste of memory
  - A waste of time because of the sequential execution of the code

- We must dissociate several paths or threads of parallel execution in the same heavy-weight process



- We have at least one light-weight process (thread or execution thread or processing unit or instruction thread) in the same initial process

1. Definitions

Processes

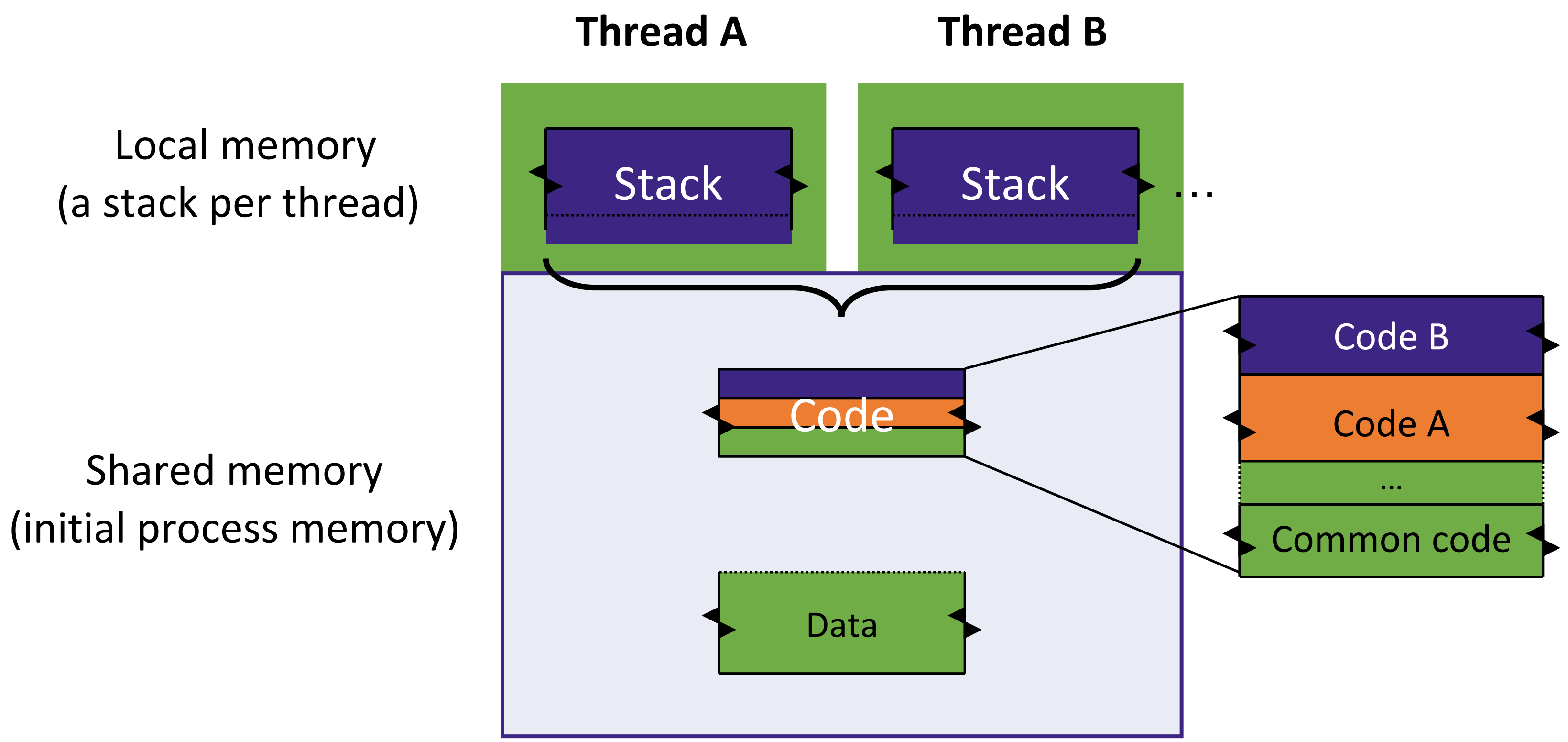
- A light-weight process (thread) is by appointment lighter to manage, and its management can be customized

- Context switching is easier between light-weight processes

- Unique states for each thread:
  - Thread identifier (PID)
  - Register status
  - Stack
  - Signal masks (describe which signals the thread responds to)
  - Priority
  - Thread private data

1. Definitions

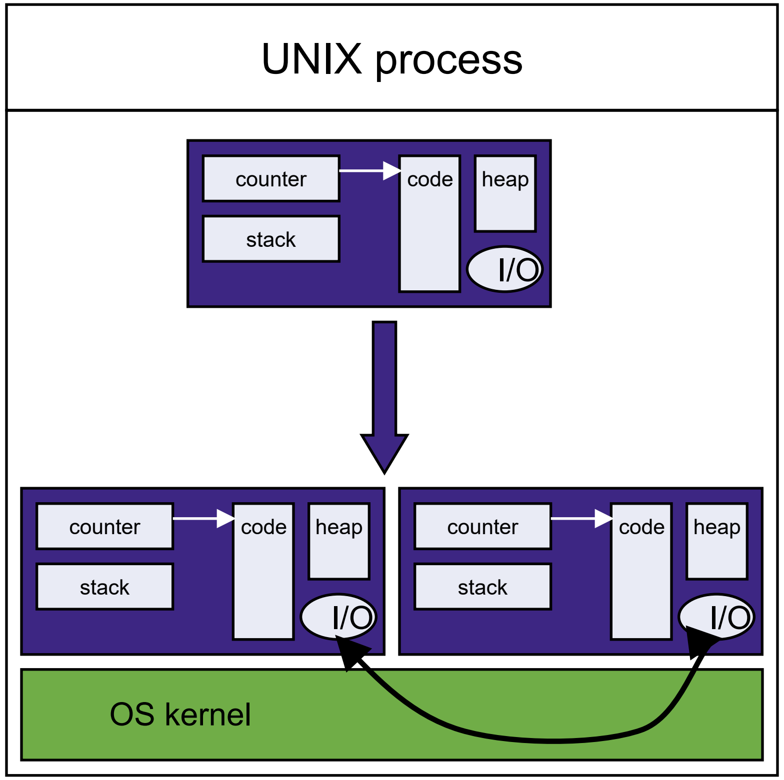
Processes



- Both processes have their own memory and run as parallel tasks

1. Definitions

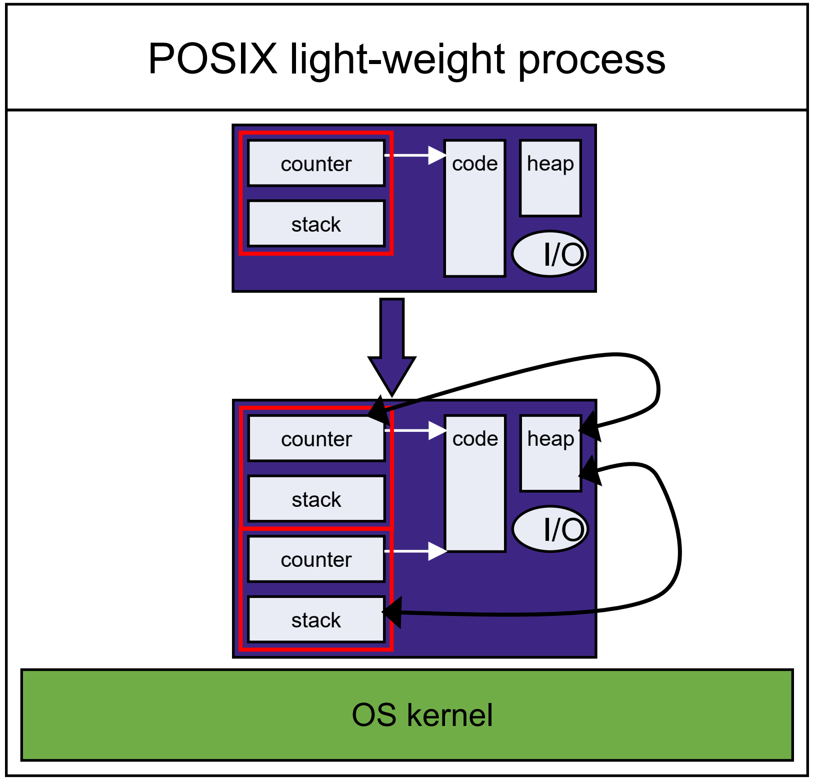
Processes



- Creation by full process copy (like in 1.5ms); different execution contexts imply **kernel** communications: slower and requires specific programming

1. Definitions

Processes

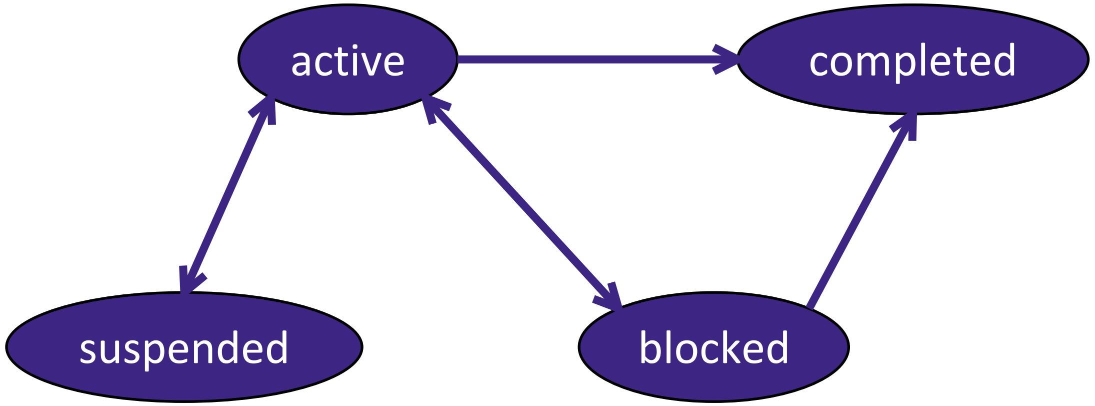


- Creation by copying some elements (like in 0.05ms); the same execution context leads to communications through the common memory: faster and less programming effort

1. Definitions

Processes

- Several processes want to run at the same time and each process interacts with the microprocessor (use of the program counter, registers, *etc*.)
- The hardware following the status word can act on the process (block it, suspend it, reroute it, *etc*.)
- The operating system sets up a management of the processes so that they run on the processor:



1. Definitions

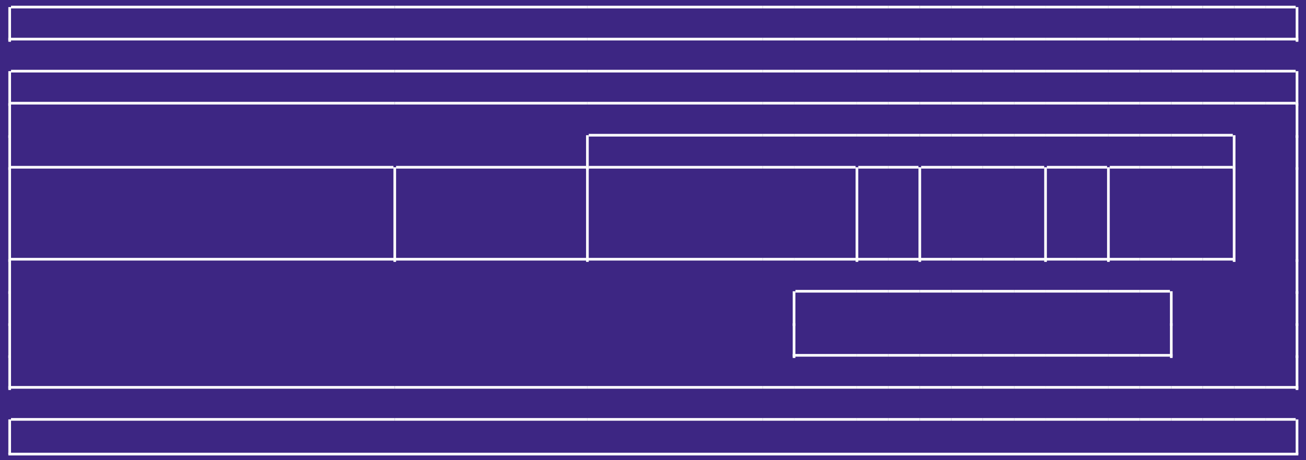
Processes

- **Active state**: the instructions of the process are executed by the processor
- **Suspended state**: sub-state of the active state, it allows to make a temporization, after this delay the process becomes active again
- **Blocked state**: the process is stopped while waiting for a resource; the unscheduled wait can be long or fast
- **Completed state**: the process has finished all its instructions
- The element in charge of sharing the processor between tasks and processes is the **scheduler**

1. Definitions

Exercise

- Using the diagram and the following items, rebuild an OS:
  - User mode
  - Kernel mode
  - Executive services
  - Memory manager
  - I/O manager
  - Process manager
  - Device driver
  - Hardware



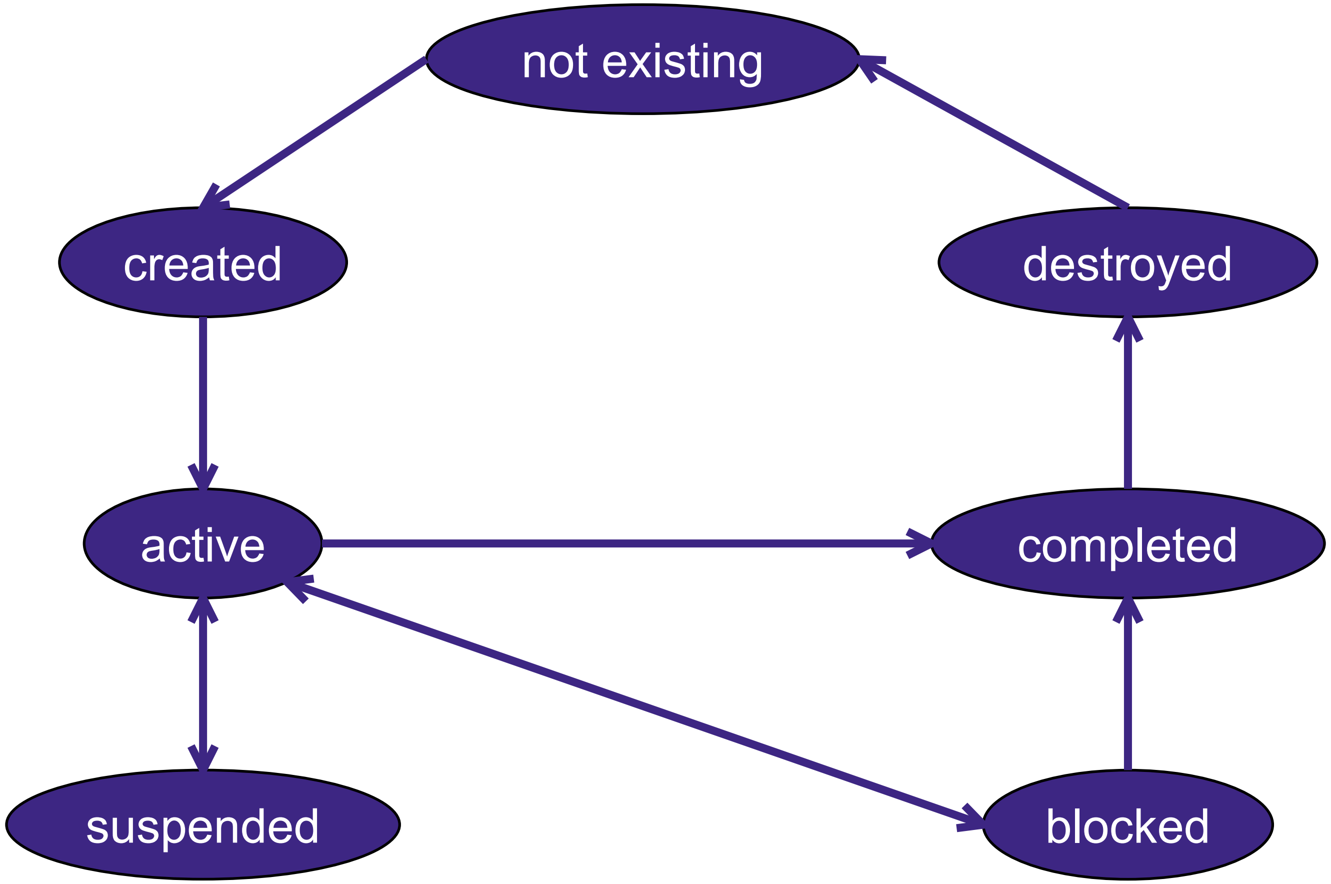
1. Definitions

Questions

2. Process Operations

2. Process Operations

Creation



2. Process Operations

Creation

- The operating system creates a process description structure associated with the new executable process: this is the **PCB** (Process Control Block)
  - The PCB allows to save and restore the **memory** and **processor context** during context switching operations

- The PCB contains the following information:
  - A unique process identifier (PID integer)
  - The current state of the process (active, blocked, suspended, terminated)
  - The processor context of the process
  - The memory context of the process
  - Various accounting information for system performance statistics
  - Information related to the process scheduling

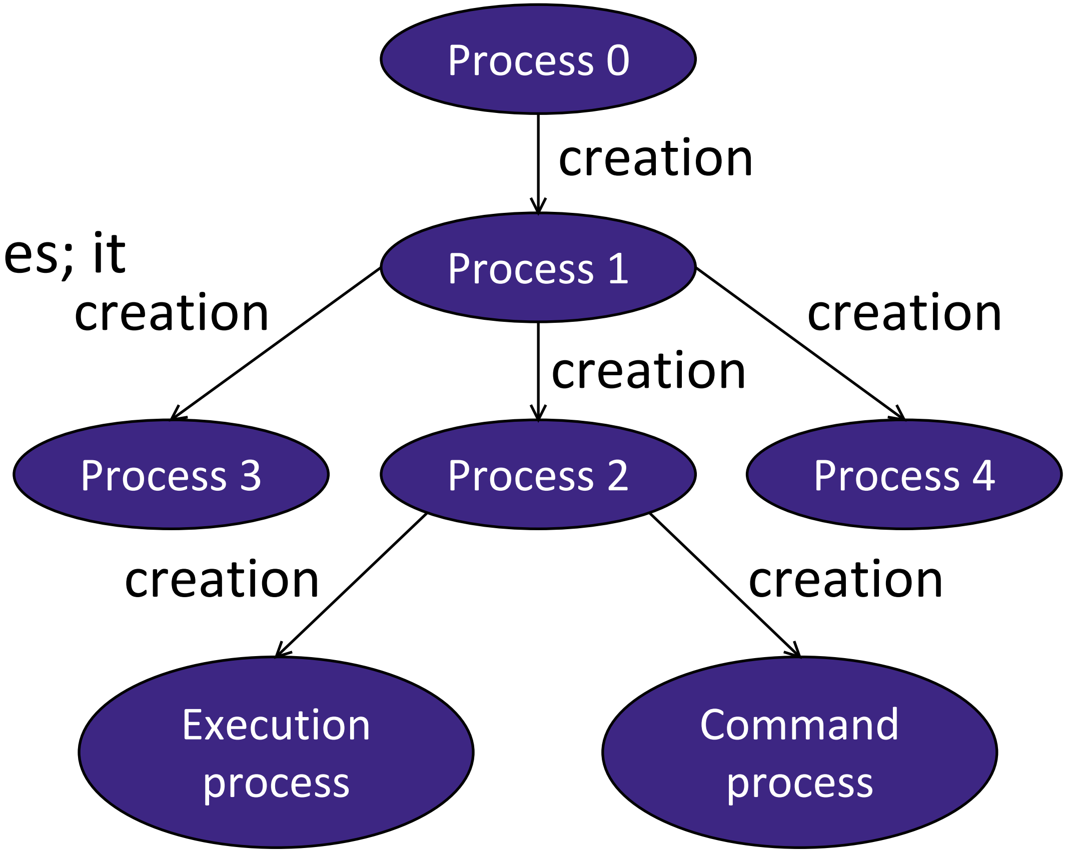
2. Process Operations

UNIX processes

- A process can create another one; the first one is called parent and the second one child

- The son process can also create other processes; it becomes the parent of its processes

- Thus, we build a graph of the processes where a **root** appears



2. Process Operations

UNIX processes

- The UNIX system is entirely built from the notion of processes.
- When the system starts, a first process is created, **process 0**, it creates in its turn another process, the **process 1** or **init**
- The **init** reads the **/etc/inittab** file and creates each of the 2 types of processes described in it:
  - **daemon** processes (suffixed with a “d”) which are system processes responsible for a function (**inetd** monitors the network, **lpd** manages printers, **crond** manages schedule)
  - **getty** processes that monitor terminals

2. Process Operations

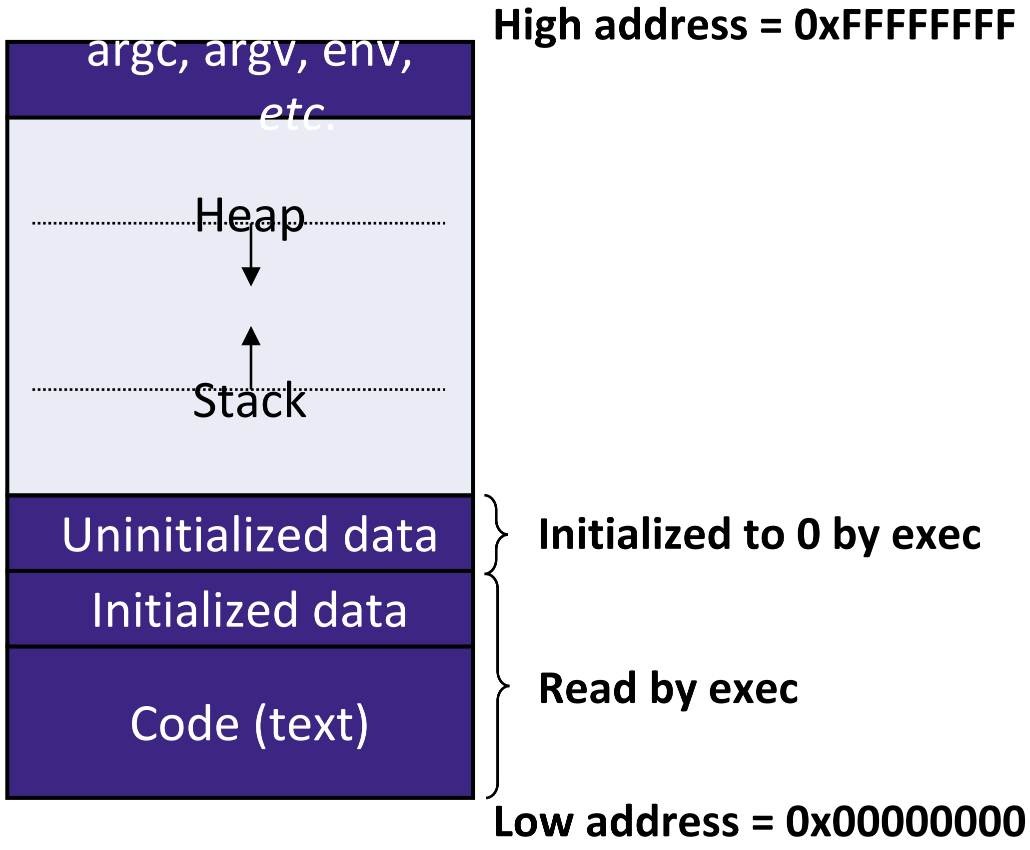
UNIX processes

- When a user logs on to a terminal, a **login** process is created which reads the user's name and password
- This **login** process checks the validity of this information using the **/etc/passwd** file for passwords
- If the information is valid, the **login** process creates a **shell** process which is the UNIX **command interpreter**
- This shell then executes the user's commands and programs, creating a **new process** each time

2. Process Operations

UNIX processes

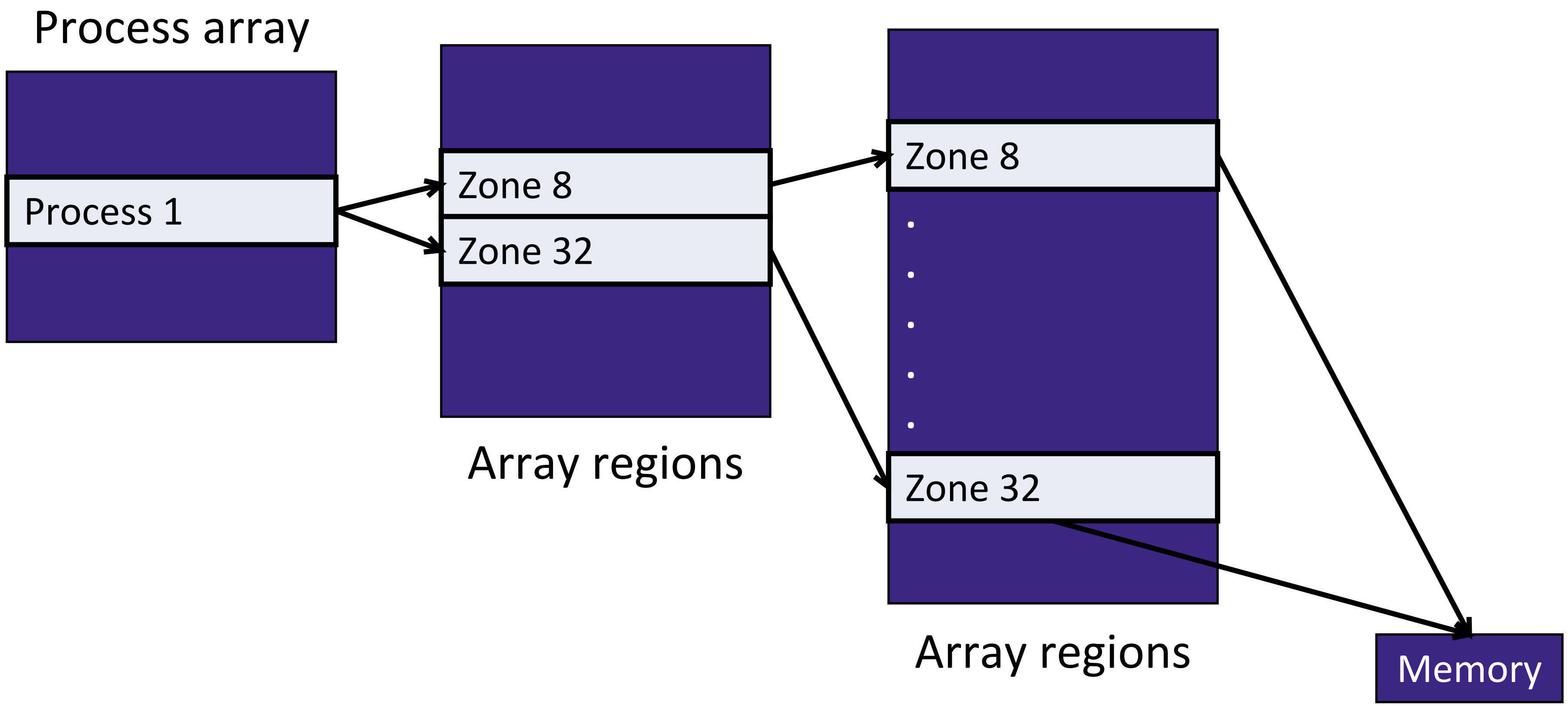
- Processes are created in the UNIX system:
  - New processes are created by the operating system primitive **fork()** (this function is used in C language when simulating process creation); this system call creates a child process clone of the calling parent process
  - The **exec()** primitive (this function is used in C language when simulating process creation) completes the process creation by providing the code that is specific to this new process



2. Process Operations

UNIX processes

- UNIX PCB:



2. Process Operations

Destruction

- PCB is divided in 2 parts:
  - Each process has an entry in a general system table, the process table: this entry contains information about the process that is always useful to the system regardless of the process state:
    - PID
    - Process state
    - Scheduling information
    - Memory information (address of the memory regions allocated to the process)
  - Each process also has another structure, the u-area: this zone contains other information about the process, but it is information that can be temporarily swapped to the disk

2. Process Operations

Destruction

- The child process can be terminated in different ways:
  - It ends normally after the last instruction of the code associated to it
  - It can execute a self-destruct instruction, **exit()** primitive for example
  - A process can be destroyed by another process, **kill()** primitive for example
  - A parent can expect the end of its children, (ex: **join()**, **wait()**, **waitpid()**)

- A termination may not occur or may occur because of an error

- The hierarchical relationship between a process and its descendants is used mainly to control the destruction of processes, the destruction of a process generally leads to:
  - The release of the resources that had been allocated to it
  - The PCB deletion, it disappears from the array and from the system queues

2. Process Operations

Questions

3. Mechanisms

3. Mechanisms

Context switching

- System calls are the interface to the operating system and are the entry points for the execution of a system function
  - They can be called directly from a program

- The commands allow to call the system functions from the command prompt of the command interpreter (shell, DOS prompt, cmd, *etc.*)

- Switching from **user mode** to **supervisor mode** is a context switch that is accompanied by a user context backup operation

- When the execution of the system function is completed, the program switches back from supervisor mode to user mode

3. Mechanisms

Context switching

- There is again a context switching operation with restoration of the user context saved during the system call

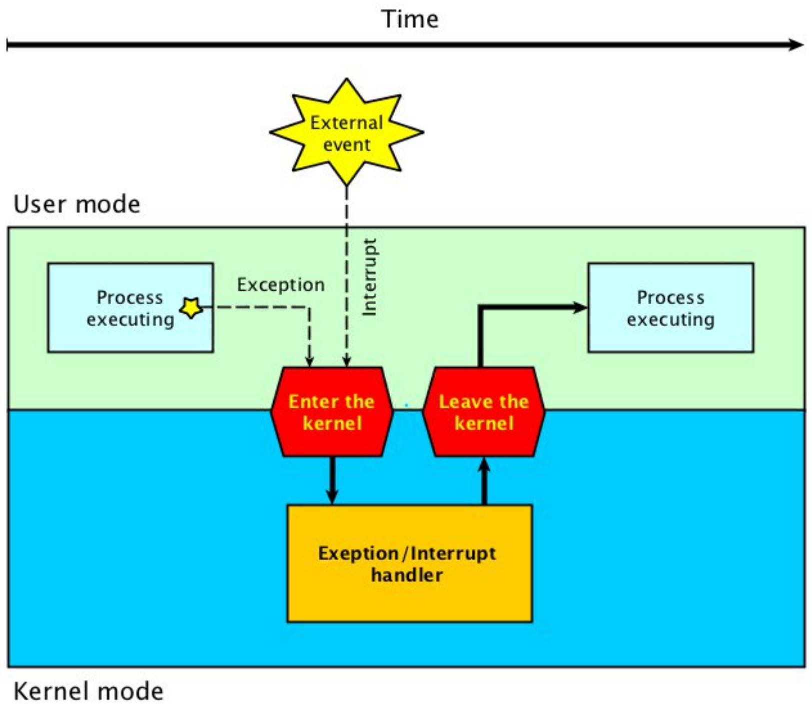
- 3 main causes for switching from user mode to supervisor mode:
  - The fact that the user program calls a system function (explicit request to switch to supervisor mode)
  - The execution by the user program of an illicit operation (division by 0, prohibited machine instruction, *etc*.): this is the trap, the execution of the user program is then stopped
  - When an interrupt is considered by the hardware and the operating system, the user program is stopped, and the execution of the interrupt routine associated with the occurred interrupt is executed in supervisor mode

3. Mechanisms

Interrupt

- An interrupt is a context switching caused by a signal managed by the hardware, this signal is itself an event that can be:
  - Internal to the process and resulting from its execution (*ex*: I/O device)
  - External and independent from this execution (*ex*: user)

- The signal modifies an indicator that is regularly consulted by the operating system to determine the cause of the interruption



3. Mechanisms

Interrupt

- Interrupt subroutine:
  - Transfers the next data item to the data register (DR)
  - Interrupt return (RTI)

- Main program:
  - Verification of the interrupt mechanism
  - Transfer of the first data (DR)
  - Unmask the interrupt
  - Do not make more action on the inputs/outputs

- Processing of inputs/outputs:
  - The controller indicates that an interrupt is ready
  - The processor can process the interrupt

Nobody:
Hardware interrupts:

3. Mechanisms

Interrupt

- The interrupt is delivered by the exchange unit to the processor when it is ready to store or load new data
- The processor, with the interrupt accounting, will carry out the associated routine which reads the data present in DR or puts in DR the next data to be written
- The processor does not need to read the status register (SR) of the exchange unit to know if it is ready to read or write
- The process remains the only one to be able to access the main memory and must manage the input/output to take care of the main memory (DR, main memory transfers)

3. Mechanisms

Interrupt

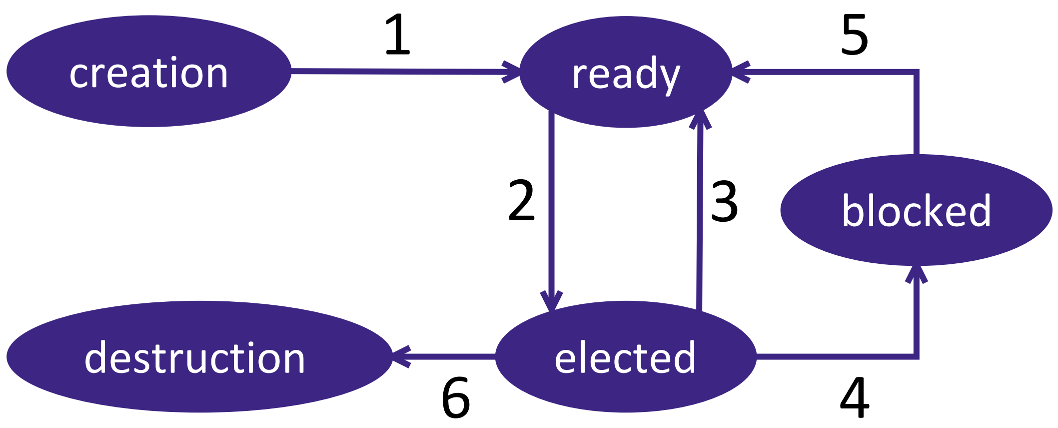
- At a given moment, several interruption indicators can be positioned at the same time; hence the use of **priorities** on the interrupt levels

- However, some processes may need to be executed completely without interruption

- **Masking** meets this need; this action disarms the priorities on the interrupts:
  - By **masking** interrupts, we no longer allow a running process to be stopped
  - By **unmasking**, we re-arm the interrupt mechanism

3. Mechanisms

Request management



1. The process is created and becomes ready

2. The process is selected by the OS

3. The process has reached its execution time (quantum)

4. The process is blocked while waiting for a resource

5. The process is unblocked by receiving the resource it was waiting for

6. End

3. Mechanisms

Request management

- During its execution, a process is characterized by:
  1. **Elected state**: execution state of the process (when the process gets the processor and runs)
  2. **Blocked state**: waiting (sleeping) state for a resource other than the processor
     - During its execution, the process can request access to a resource that is not immediately available (I/O realization, protected variable access, *etc.*)
     - The process cannot continue its execution until it has obtained the resource (*ex*: the process must wait for the end of the I/O that delivers the data on which it performs the next calculations in the code)
     - The process then leaves the processor and goes into the **blocked** state

3. Mechanisms

Request management

- During its execution, a process is characterized by:
  3. **Ready state**: waiting state for the processor
     - When the process has obtained the resource that it was waiting for, it can potentially resume its execution
     - However, we are in the context of multi-programmed systems (there are several programs in central memory and thus several processes)
     - When the process has entered the **blocked** state, the processor has been allocated to another process
     - The processor is not necessarily free: the process passes then into the **ready** state

3. Mechanisms

Request management

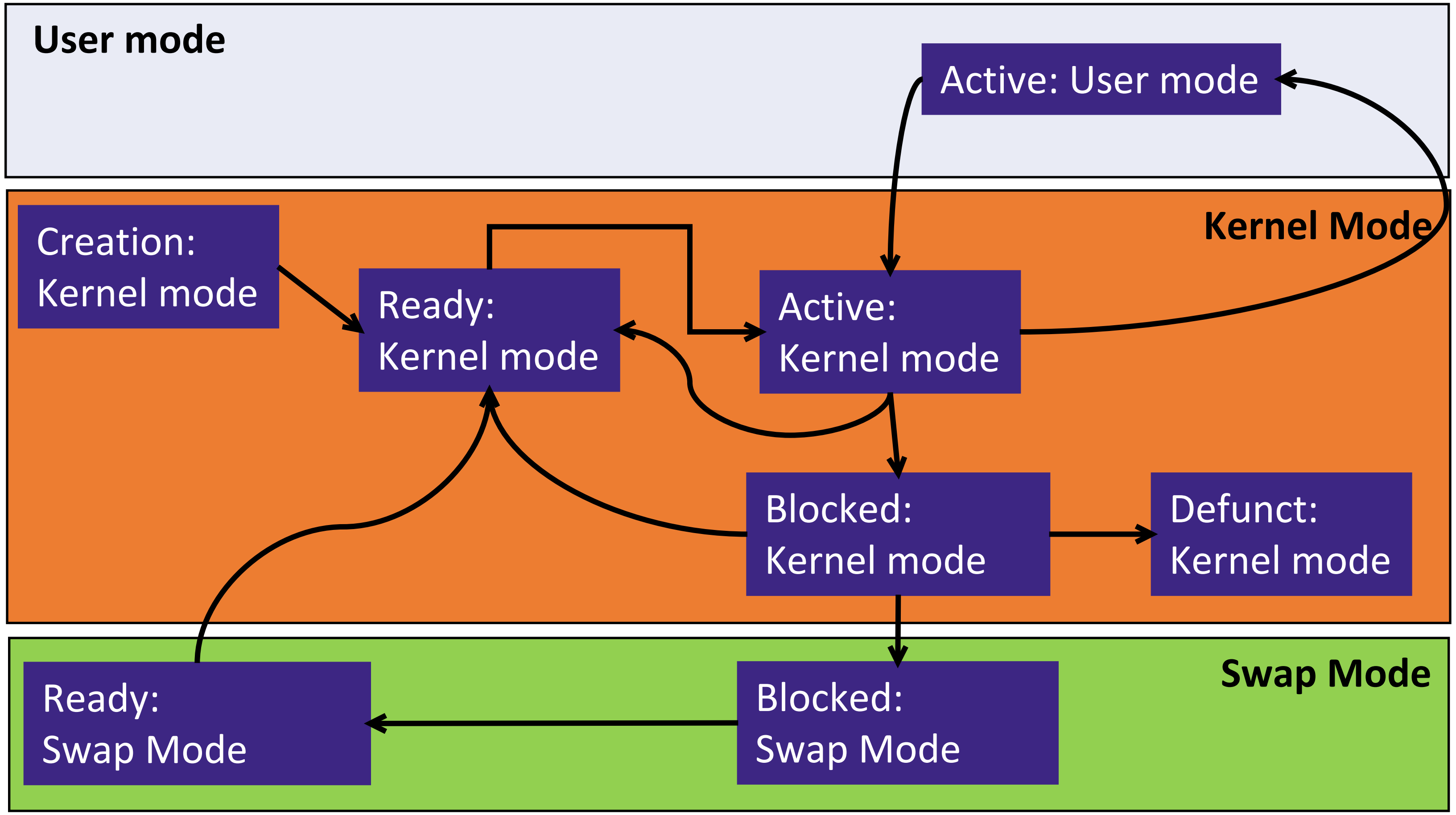
- State change:
  - The change from the **ready** state to the **elected** state is the **election operation**
  - The change from the **elected** state to the **blocked** state is the **blocking operation**
  - The change from the **blocked** state to the **ready** state is the **unblocking operation**

- A process is always created in the **ready** state

- A process is always terminated from the **elected** state (unless there is an anomaly)

3. Mechanisms

Request management



3. Mechanisms

Request management

- The **kernel mode** or **supervisor mode** is privileged because it gives access to a greater number of machine instructions than the **user mode** (it allows the execution of instructions for masking and unmasking interrupts forbidden in **user mode**)

- **User mode**: normal execution mode

- **Kernel mode** (in memory): mode in which a process is **ready** or **blocked** (sleeping)

- **Swap mode**: mode in which a **blocked** (sleeping) process is unloaded from the main memory into the swap zone on the hard disk

3. Mechanisms

Request management

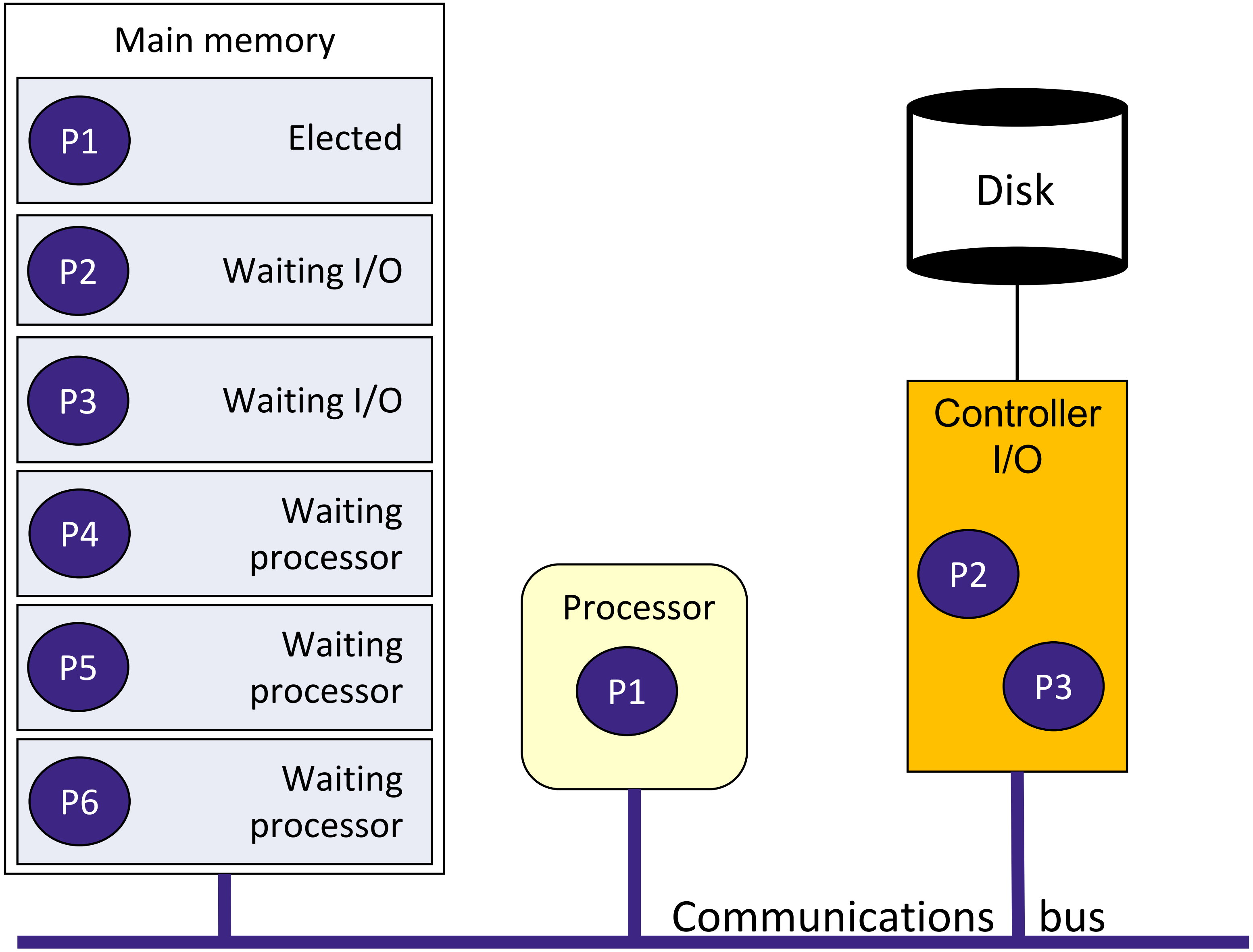
- The UNIX system unloads from main memory processes that have been sleeping for too long (they are in the **swapped sleeping** state)

- These processes return to main memory when they become ready again (transition from the **swapped ready** state to the **ready** state)

- A process that terminates goes into a so-called **zombie** state: it remains there as long as its PCB is not completely dismantled by the system

3. Mechanisms

Preemption



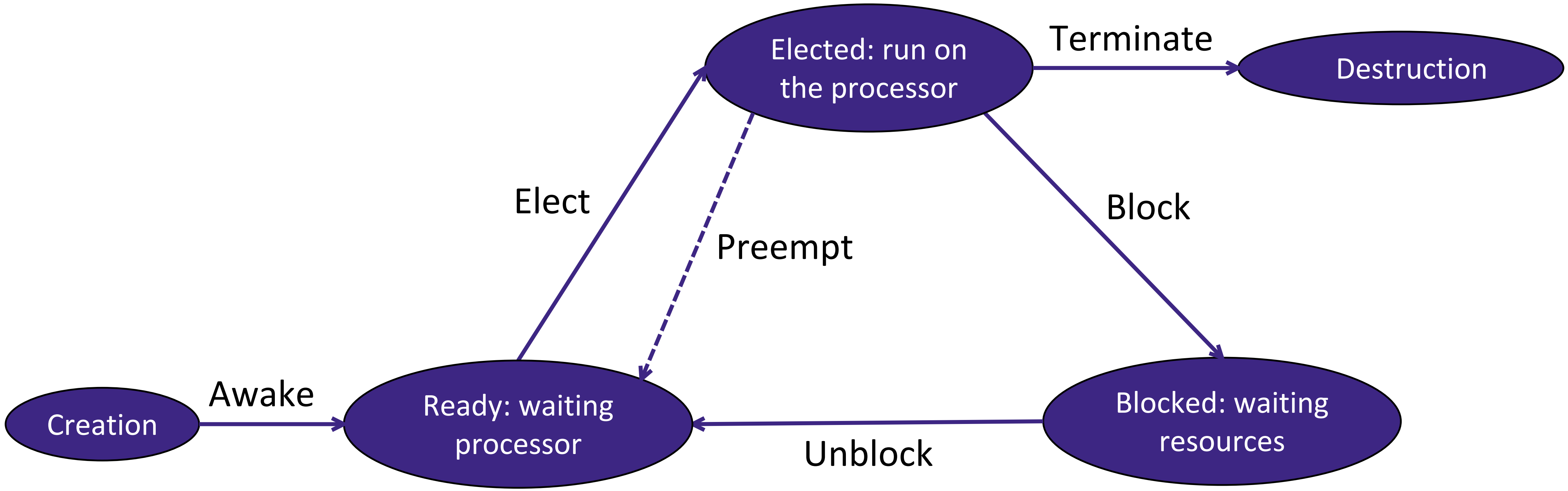
3. Mechanisms

Preemption

- The process P1 is **elected** and runs on the processor
- Processes P2 and P3 are in the **blocked** state, they are waiting for an I/O end with the disk
- Processes P4, P5 and P6 are in the **ready** state: they could run but they cannot because the processor is occupied by process P1
- When the process P1 will leave the processor, the 3 processes P4, P5 and P6 will have all 3 the right to obtain the processor; but the processor can only be allocated to one process at a time: it will be necessary to choose between the processes P4, P5 and P6
  - This is the role of **scheduling** which will choose one of the 3 processes

3. Mechanisms

Preemption



3. Mechanisms

Preemption

- The graph proposes the existing transitions between the **ready** state (waiting for processor state) and the **elected** state (busy processor state)

- The transition from the **ready** state to the **elected** state constitutes the election operation: it is the allocation of the processor to one of the ready processes

- The transition from the **elected** state to the **ready** state corresponds to a **requisition** of the processor (the processor is withdrawn from the elected process while it has all the resources necessary to continue its execution)

- This **requisition** is the **preemption**

3. Mechanisms

Preemption

- If the **requisition** operation is allowed or not, the scheduling will be called **preemptive** or **non-preemptive** scheduling:
  - If the scheduling is **non-preemptive**, the transition from the **elected** state to the **ready** state is **forbidden**: a process exits the processor if it has completed its execution or if it crashes
  - If the scheduling is **preemptive**, the transition from the **elected** state to the **ready** state is **allowed**: a process leaves the processor if it has finished its execution, if it crashes or if the processor is requisitioned

3. Mechanisms

Exercise

- Define and disassociate the following concepts:
  - System call
  - Context switching
  - Program
  - Process
  - Thread

- What are the main functions of an operating system?

3. Mechanisms

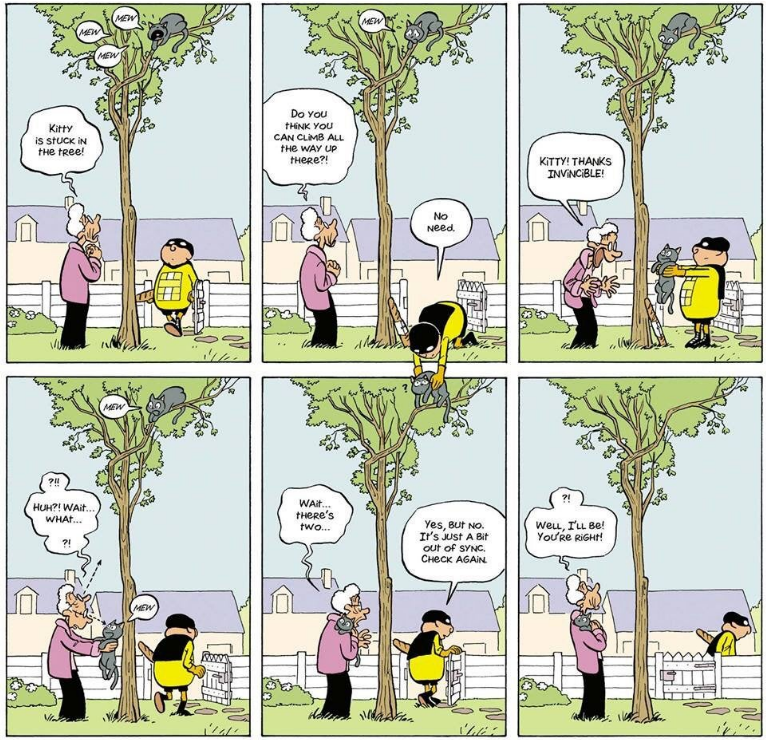
Questions

4. Communication

4. Communication

Overview

- The communication between processes is done:
  - By signals (asynchronous)
  - By pipes or files
  - By message queues
    - For different actors
    - Synchronous (with waiting)
    - Asynchronous (by appointment, without waiting)
  - By shared memory
  - By sockets (in networks or locally)
  - By protected objects
    - For the same actors
    - For the same resources



4. Communication

Signals

- A signal is an asynchronous atomic information sent to a process or group of processes by the operating system or by another process

- When a process receives a signal, the operating system informs it as an interrupt (trap); this process executes in return, a specific routine (program) to handle the signal (then the process resumes where it was interrupted)
  - Signals control the execution of a set of processes (*ex*: Linux shell)

- A process sends a signal to another process: **kill(pid_t pid, int num)**, with the signal number **num** is sent to the process identified by **pid**

```
#include <signal.h>
typdef void handler_t (int)
handler_t *Signal(int signum, handler_t *handler)
```

SIGINT (<control-C>)
SIGTSTP (<control-Z>)
SIGKILL (kill or exit)
SIGALRM (alarm)
SIGSEGV (memory protection violation)

4. Communication

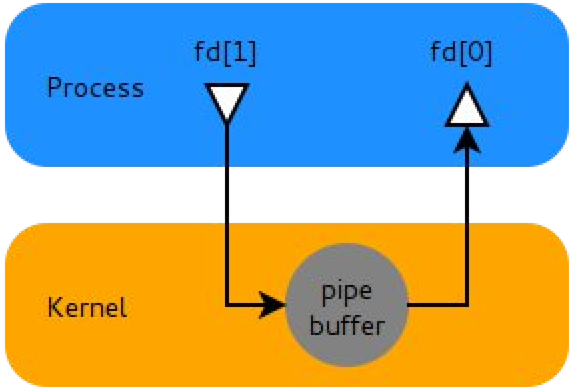
Pipes

- A pipe is an anonymous file that serves as a buffer for communication between 2 processes
- To be used, a pipe is opened by the 2 processes
- Each side of the pipe is a file descriptor opened either in reading or in writing, which makes it possible to use it very easily, by means of the traditional input/output functions
- Generally, pipes have 2 constraints:
  - Pipes only allow one-way communication
  - Processes that can communicate through a pipe must come from a common ancestor

4. Communication

Pipes

- Pipes give the possibility to synchronize 2 processes:
  - A process attempting to read a pipe in which there is nothing, is suspended until data is available
  - If the process that writes to the pipe does not do it faster than the one that reads, it is possible to synchronize the reader process with the writer process



4. Communication

Exercise

- Which of these instructions should only be allowed in kernel mode?
  - Disarm all interrupts
  - Read the clock showing the date
  - Write the clock showing the date
  - Change the address space map

- In a system, what is the role of the kernel?

4. Communication

Questions

Basic System Elements

Thank you for your attention