

Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

Part 1: Introduction

Unit 2

System Call Concept

f SO

DISCA



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

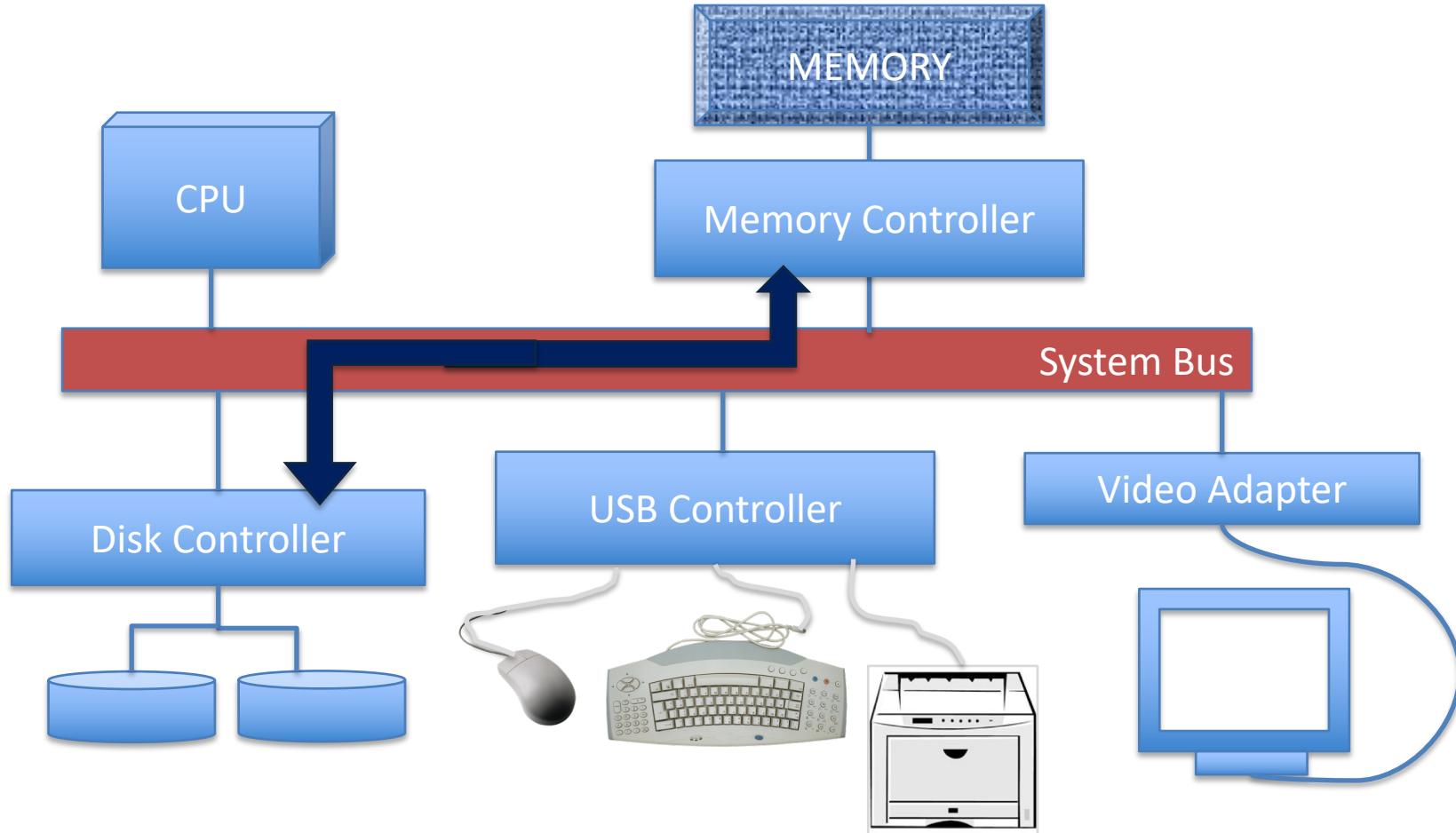
- Goals
 - To offer an **overview** of the computer operation from the point of view of the **operating system**.
 - To introduce the **concept of system call** as the mechanism for accessing operating system services.
 - To describe the **services** the operating system provides to users and processes.
- Bibliography
 - A. Silberschatz, P. B. Galvin. “Operating Systems Concepts” 8th ed. Chapters. 1 and 2

- Computer hardware architecture
- Interrupts
- Execution modes
- System calls
- System utilities

Terms:

DMA	Direct Memory Access
SO	Operating System
USB	Universal Serial Bus
INT	Interrupt
POSIX	Portable Operating System Interface

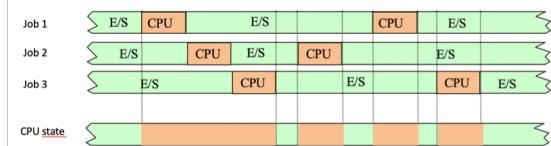
- Diagram of computer operation



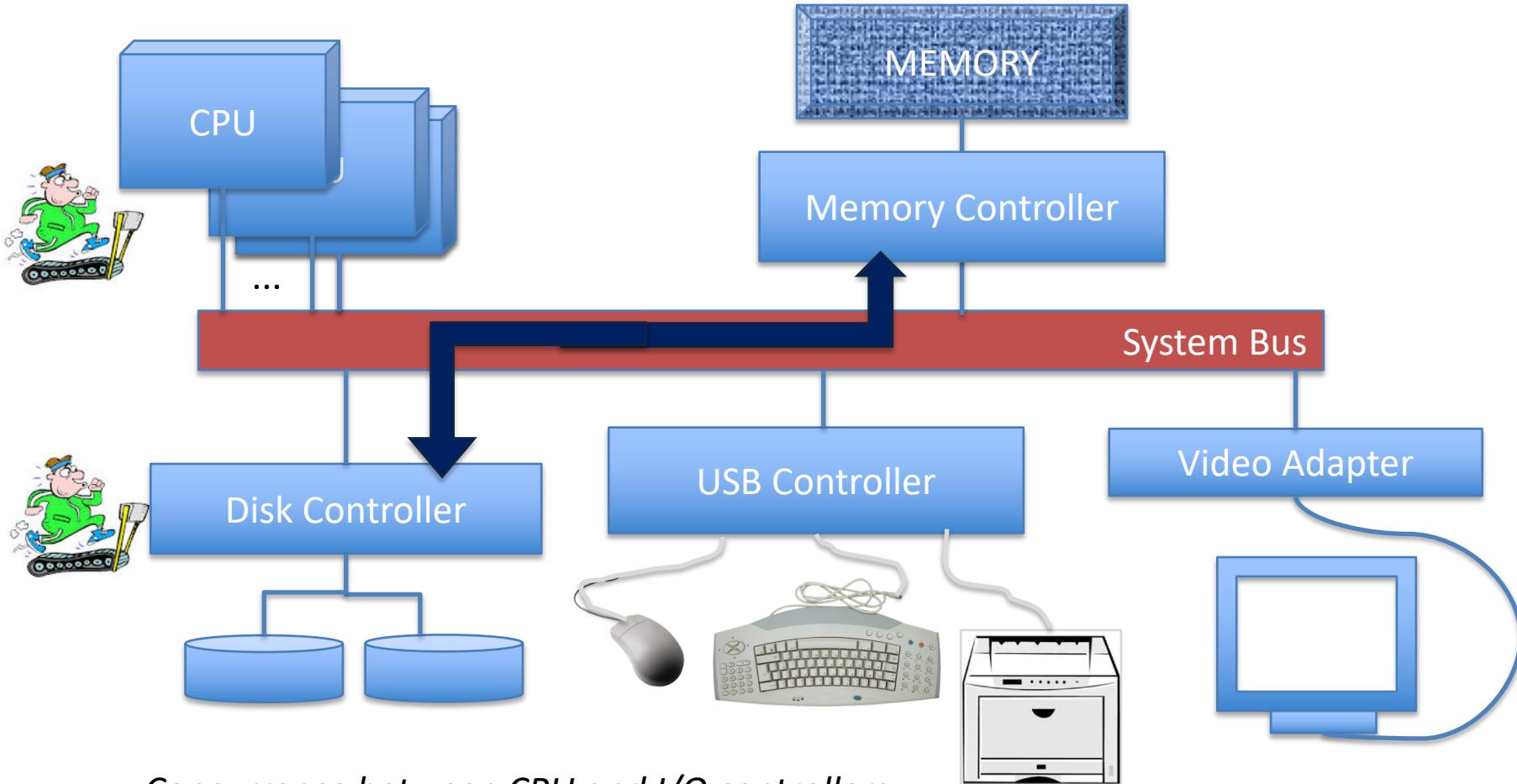
- Concurrency between I/O and CPU
 - I/O devices are much slower than processors
 - i.e. the time it takes to access information stored on disk
 - A modern processor (mono or multi-core) can execute **billions of machine instructions** in the time it takes to access disk.
 - It is important that while performing I/O, the processor can execute useful instructions instead of waiting

→ Concurrency between CPU and I/O

- Multiprogramming
 - Alternative use of the CPU by running programs
 - When a process is blocked waiting for a pending I/O operation, the CPU executes instructions from another ready process
 - A "context switch" is performed when an I/O operation is demanded
 - CPU utilization increases
 - The system performance increases: more jobs end with less time



- Diagram of computer operation



Concurrence between CPU and I/O controllers

CPU and I/O devices work simultaneously along time

- Device driver vs device controller

Operating System

Device Driver

- Operating system component
- Software capable to:
 - Program the controller
 - Provide a friendly interface to use the controller (API)

I/O hardware devices

Device Controller

- Hardware-component
- DMA capable

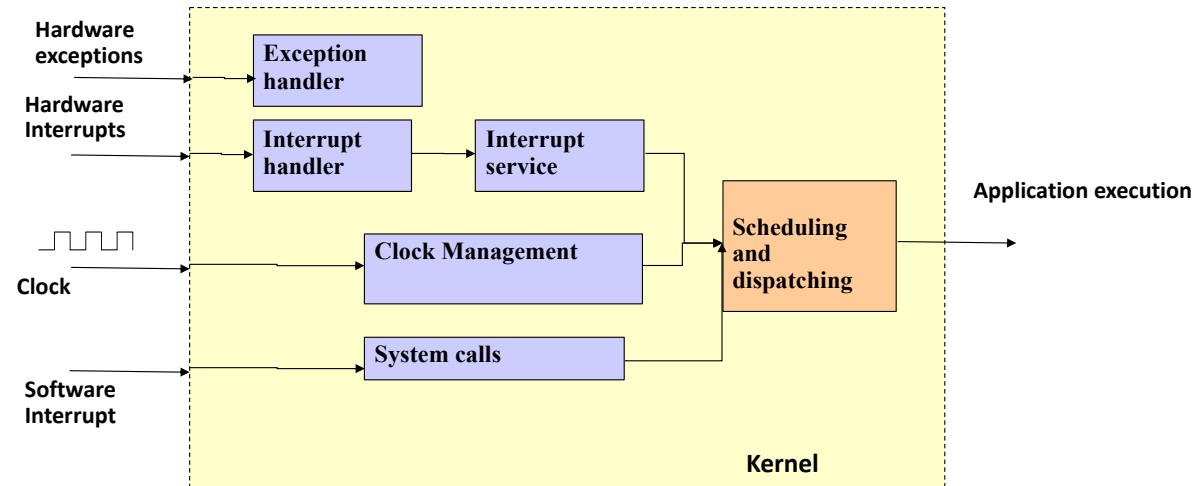
Control Register

Status Register

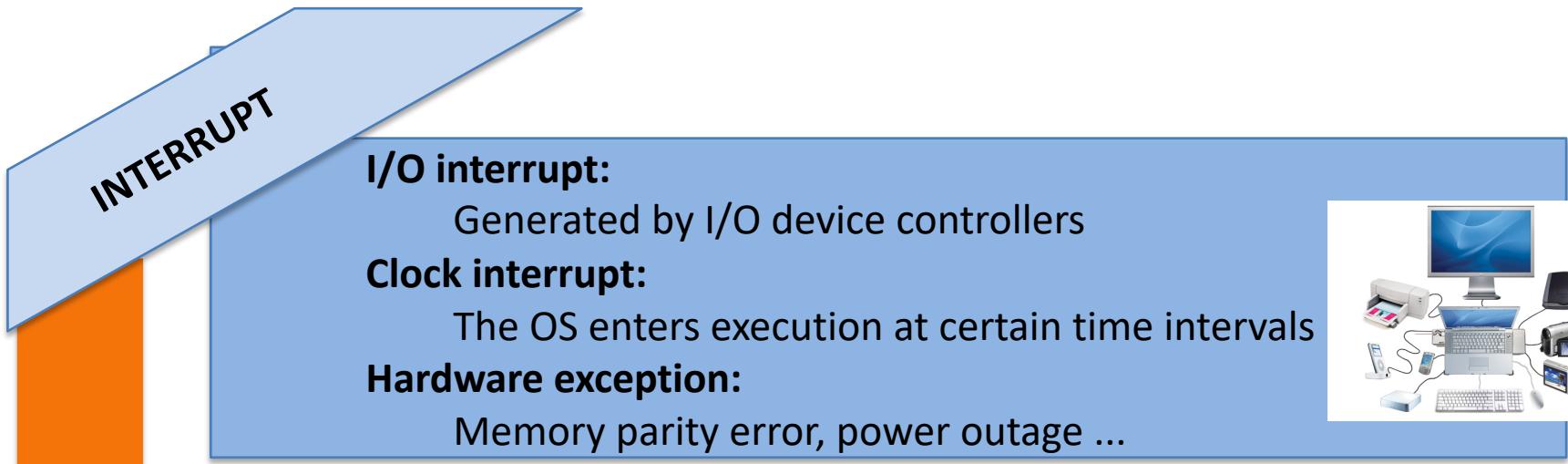
Data Buffer

- Computer hardware architecture
- **Interrupts**
- Execution modes
- System calls
- System utilities

- An OS is an **event-driven program**
- Events are **hardware interrupts, software interrupts** and **exceptions**
- The operating system **acts as a server** program waiting for work that is commissioned by interrupts
- OS processes and I/O devices perform OS service requests



- What generates an interrupt request?
- When an interrupt request occurs?



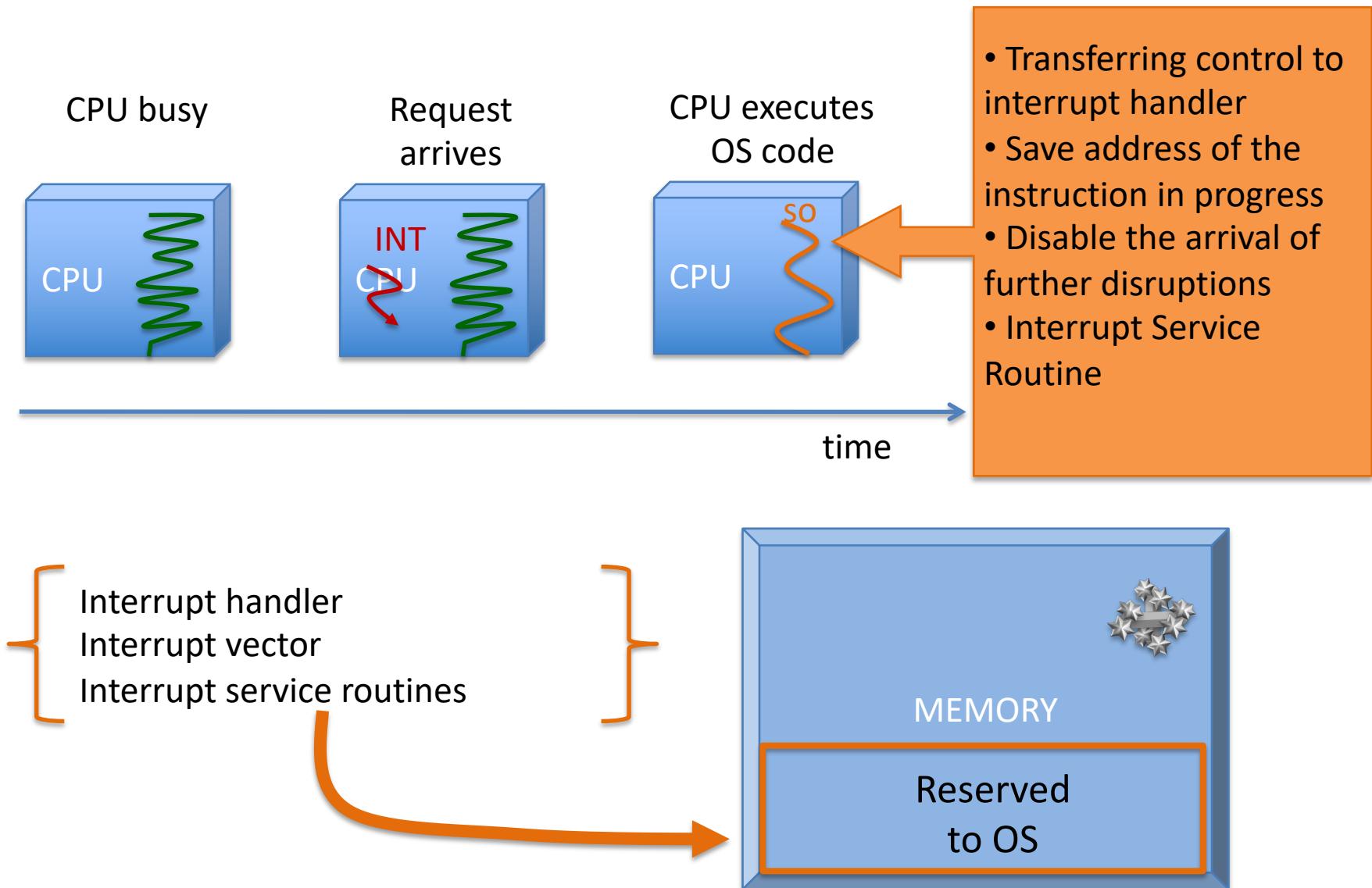
Traps (sw interrupts):

Used by programs to request OS services

Software exceptions:

Are generated when happen some events like a division by zero, an arithmetic overflow, an addressing to a memory location prohibited, etc

- Interrupt mechanism



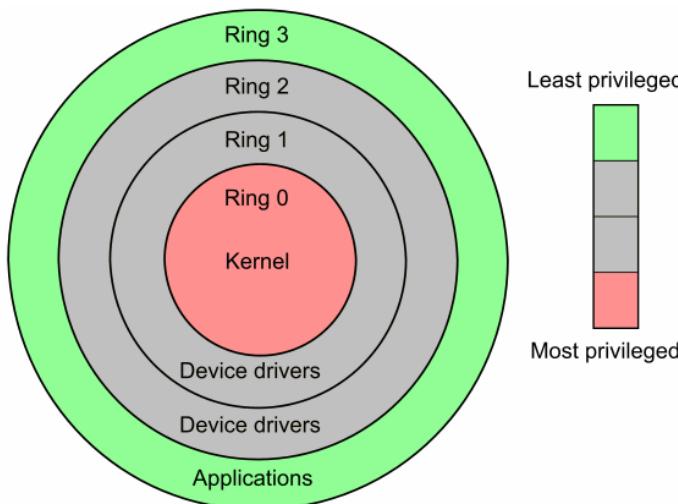
- Computer hardware architecture
- Interrupts
- **Execution modes**
- System calls
- System utilities

- Processors have **two or more execution modes**
 - Execution modes are included to **support the operating system**
 - Processes running simultaneously share machine resources → **they need protection**
- **Protecting access to the hardware** to prevent user programs to:
 - Access system **memory** freely
 - Monopolize the **CPU** and system registers
 - Direct access to **I/O devices**
- Processors have a **mode bit** that indicates the actual operation mode: ***kernel (0) or user (1)***

ARM Cortex Processors

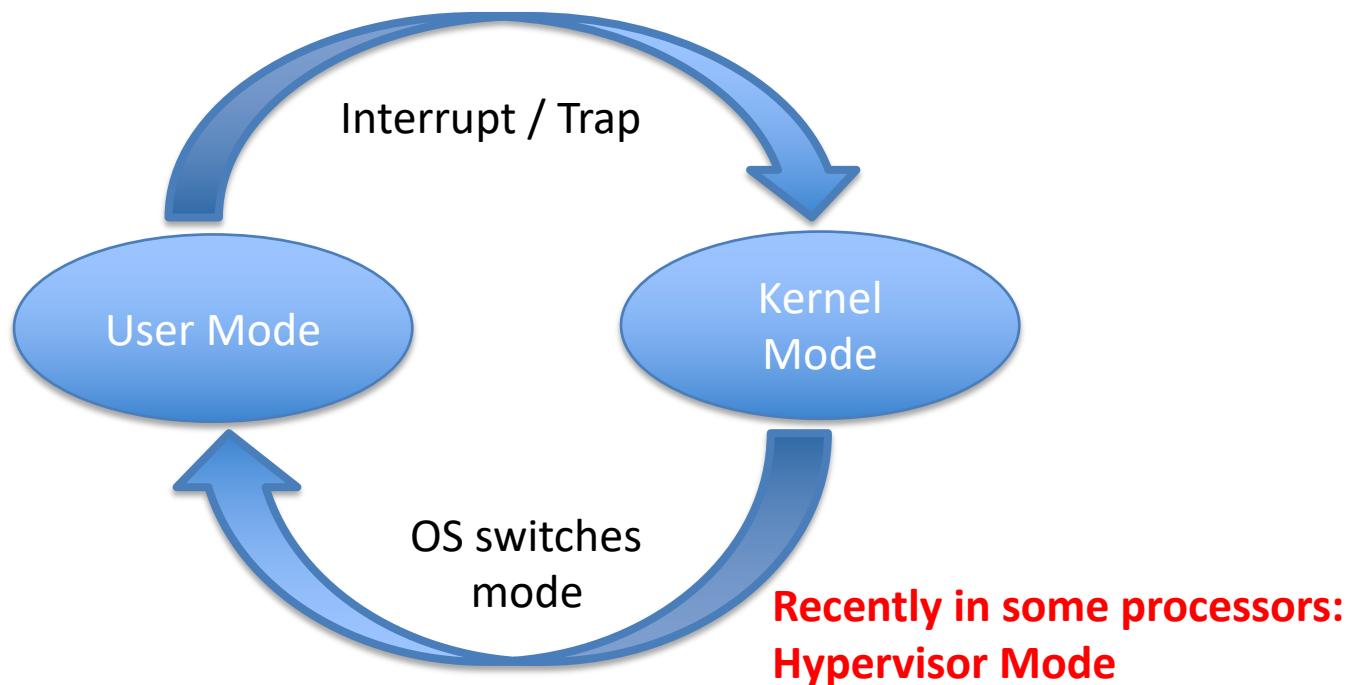
Mode	Description	
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a normal priority interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

Intel x86 Processors



Recently in some processors:
Hypervisor Mode

- Two modes of execution
 - **User Mode or Regular:** has restricted instruction set
 - **Kernel mode or Privileged:** Run any type of hardware operations, memory access and I/O device
- Dual operation mode



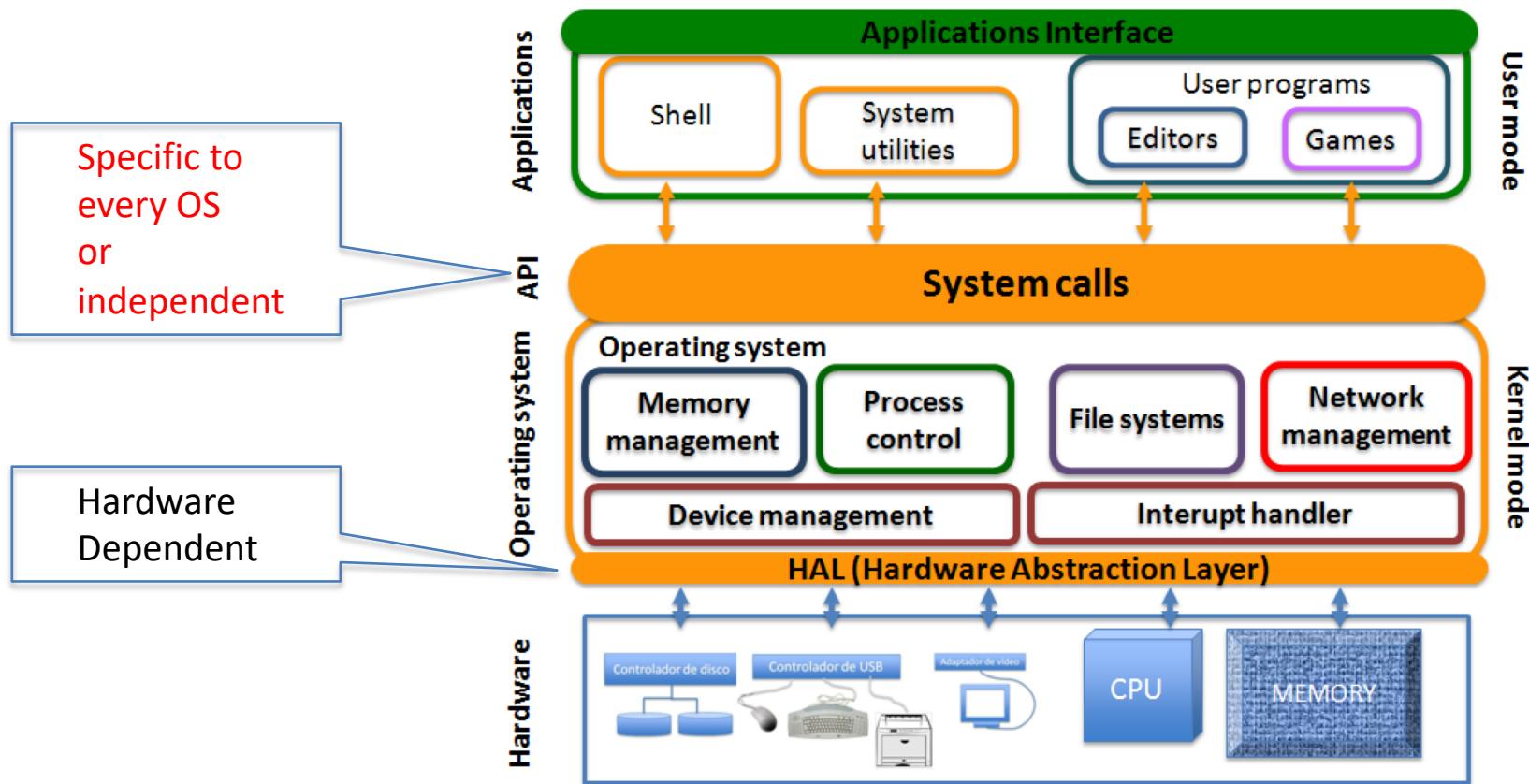
- Privileged instructions
 - **Instructions available in kernel mode** and which are not in user mode are called privileged instructions.
 - Privileged instructions are mainly associated with three types of protection
 - I/O protection
 - Memory protection
 - Process protection

Recently in some processors:
Hypervisor instructions

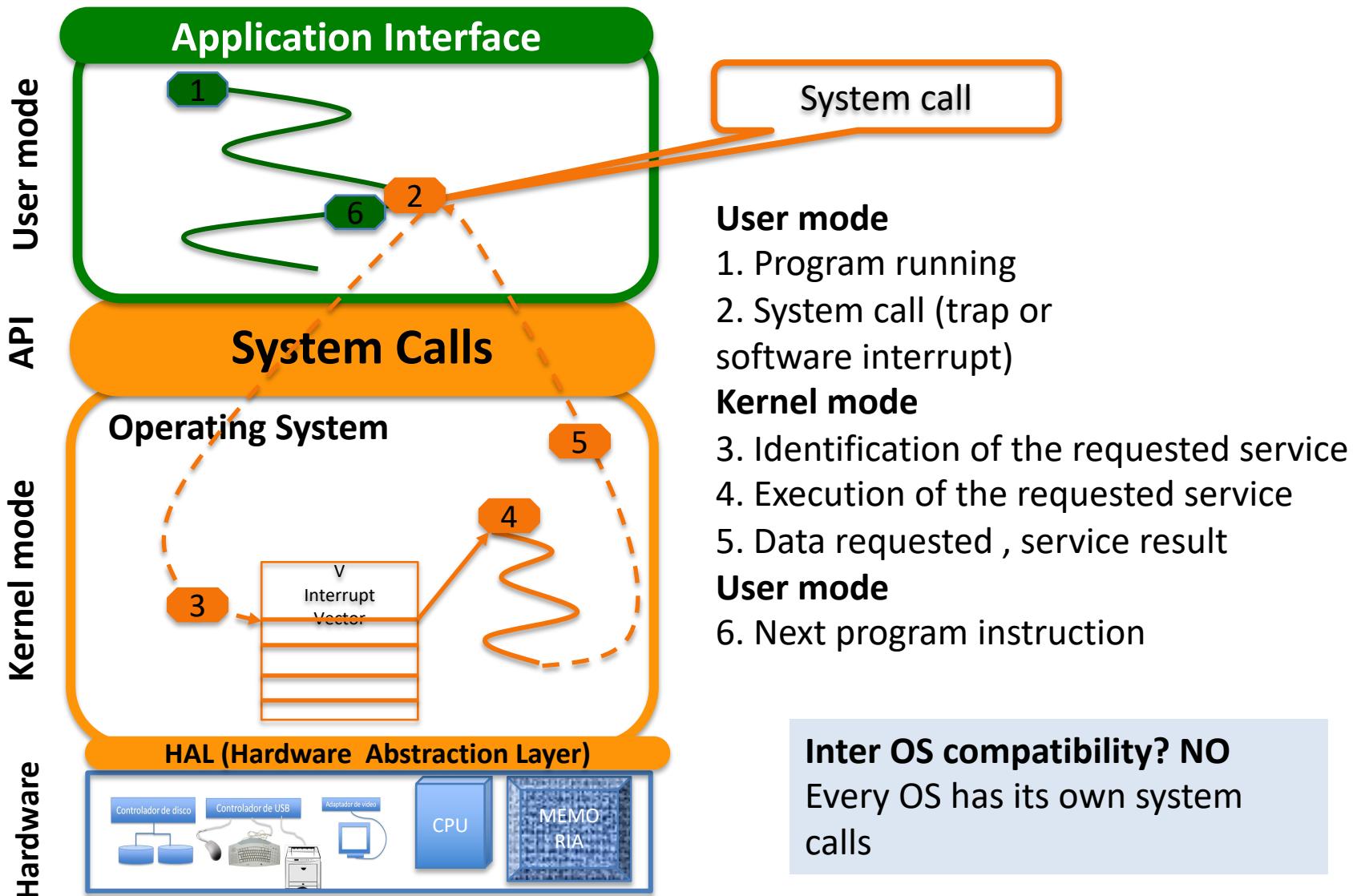
- Computer hardware architecture
- Interruptions
- Execution modes
- **System calls**
- System utilities

System calls

- OS mechanism to **provide services on-demand**.
- **Interface** that defines the operation set provided by the OS to allow accessing the machine resources (**API**).
- It is implemented as **C library functions**.

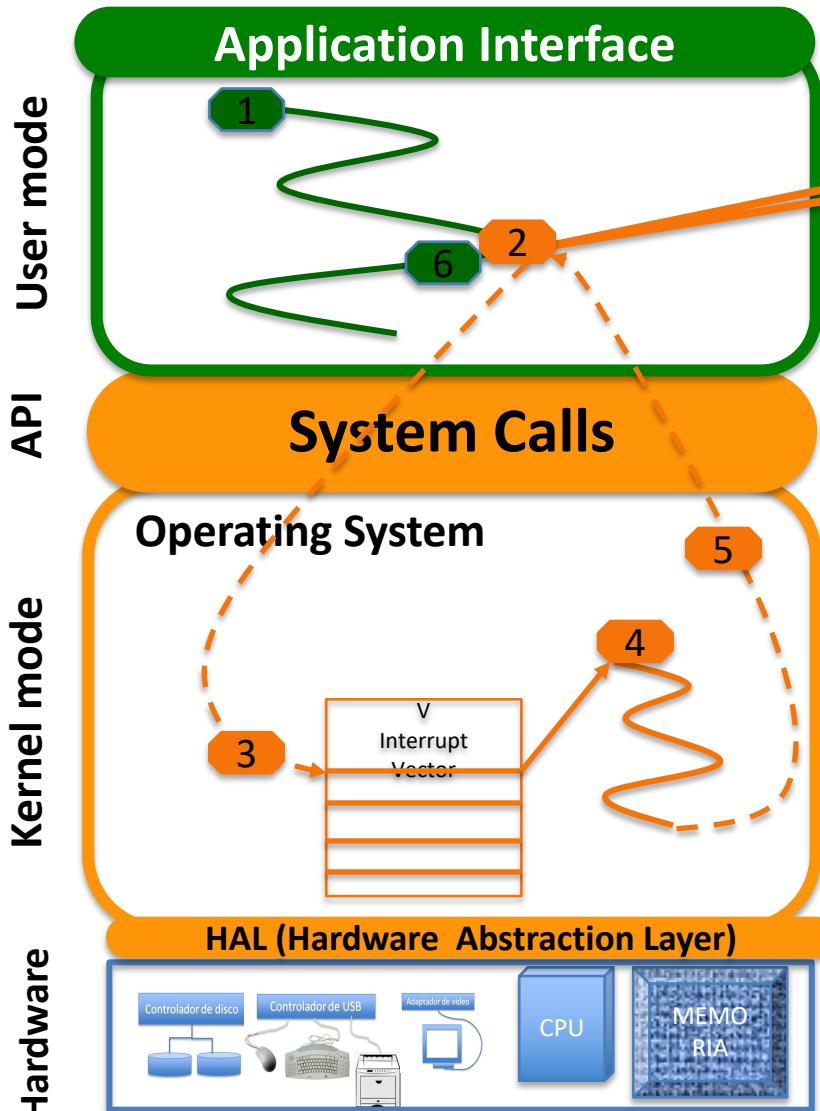


- Service request to the OS



System calls

- Service request to the OS



System call

User mode

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

`man read`

on the command line. A description of this API appears below:

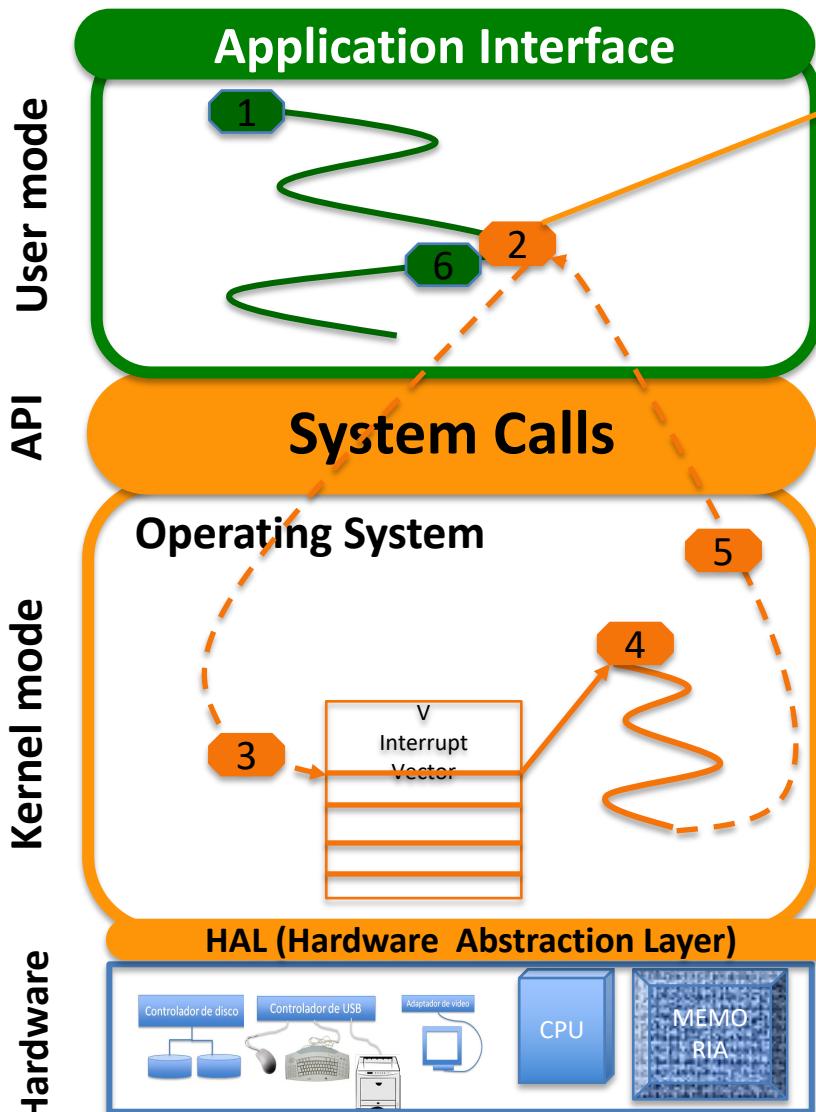
```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
    return value   function name   parameters
```

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

- Service request to the OS



libc library:

```

read: push <readId>
      push <fd>
      push <buff address>
      push <count>
sw int
      pop retcode
      return retcode
  
```

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

`man read`

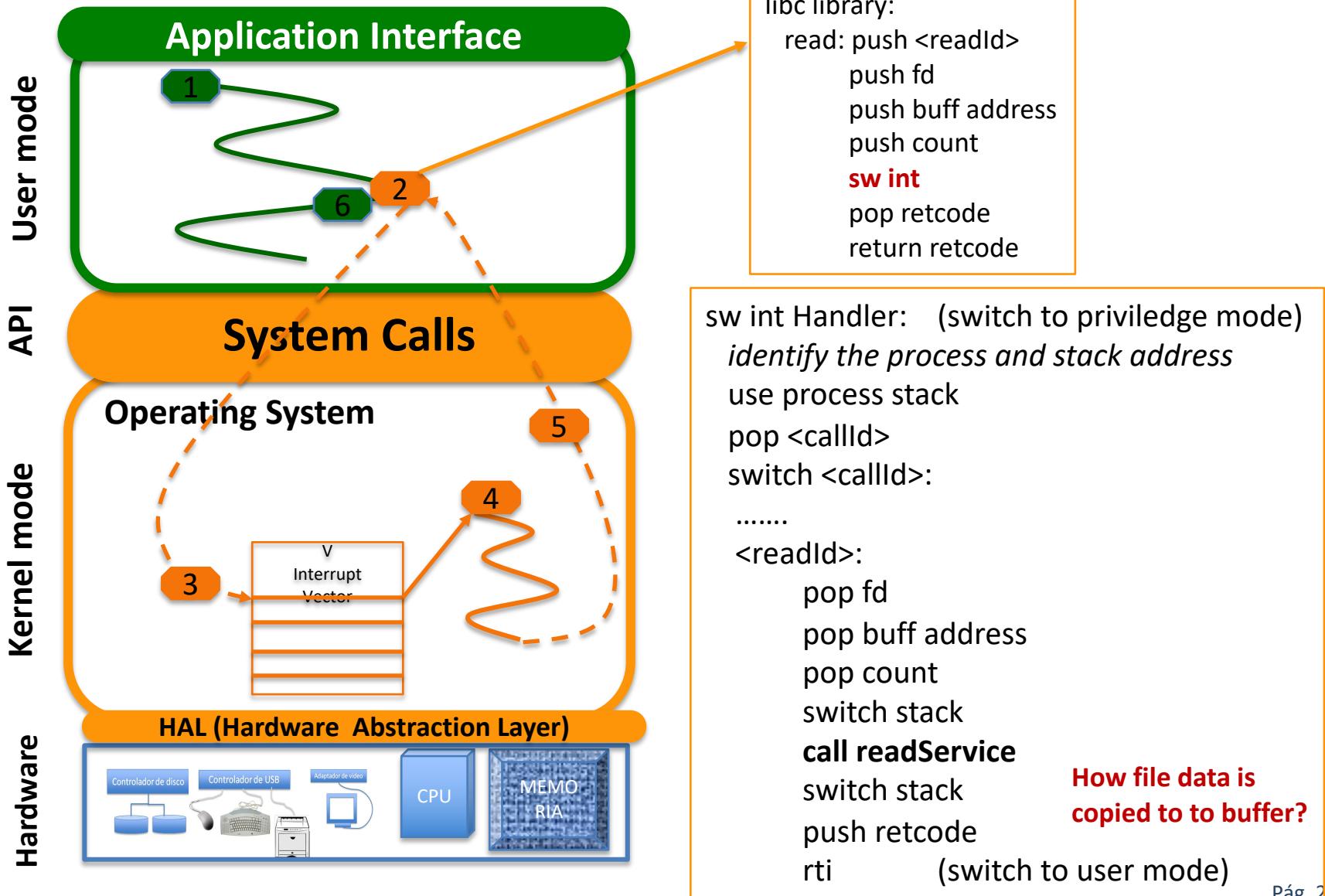
on the command line. A description of this API appears below:

#include <unistd.h>	ssize_t	read(int fd, void *buf, size_t count)
return value	function name	parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- I • `int fd`—the file descriptor to be read
 - E • `void *buf`—a buffer into which the data will be read
 - C • `size_t count`—the maximum number of bytes to be read into the buffer
- C On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns -1.

- Service request to the OS



System calls

- POSIX system calls sample set
 - POSIX (Portable Operating System Interface)

	Processes
fork	Creates a child process
exit	Ends process execution
wait	Waits for a process end
exec	Executes an specific program from another program
getpid	Get process attributes
setsid	Changes process attributes

	Protection
chmod	Sets file permission bits (rwx, suid, sgid, etc)
chown	Sets file group and owner
umask	Set the protection mask of a process

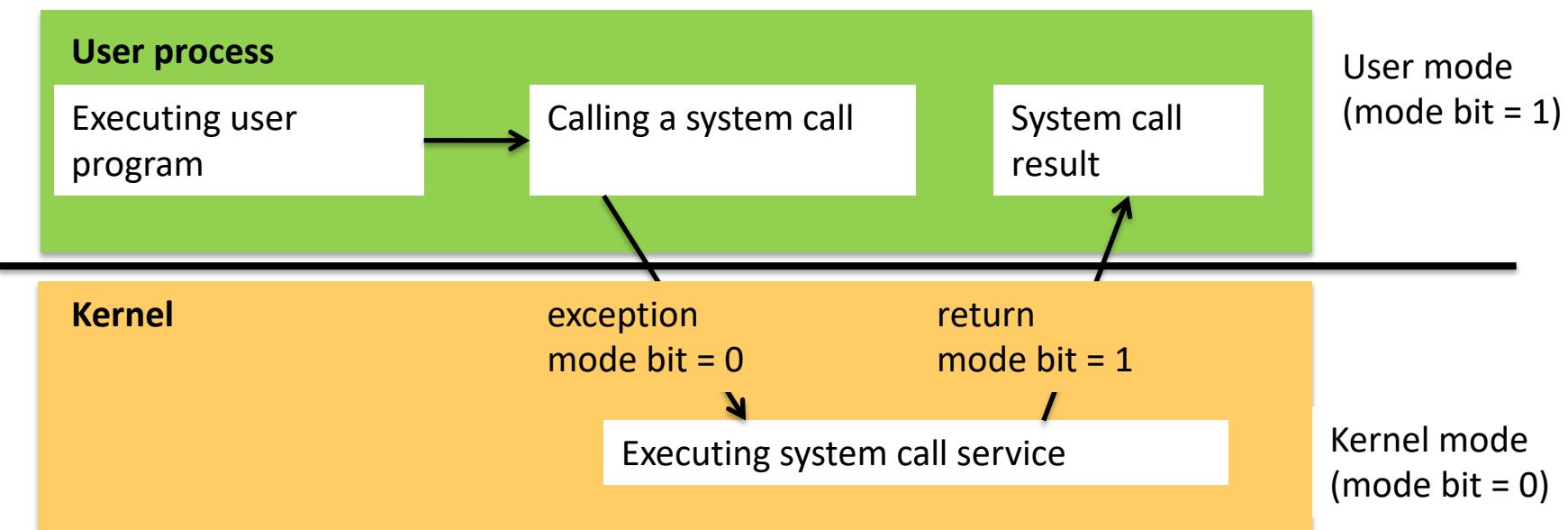
	Directories
mkdir	Create directory
rmdir	Remove an empty directory
opendir	Open directory
readdir	Get the next directory entry
closedir	Close directory
link	Get information about file i-node
unlink	Remove a directory entry

	Signals
kill	Send signals
alarm	Generate an alarm (clock signal)
sigemptyset	Init a mask with no signals set
sigfillset	Init a mask with signals set
sigaddset	Append a signal to a signal set
sigdelset	Delete a signals in a signal set
sigismember	Check if a signals belongs to a signal set
sigprocmask	Check/Modify/Set a signal mask
sigaction	Capture/Manage a signal
sigsuspend	Wait signals capture

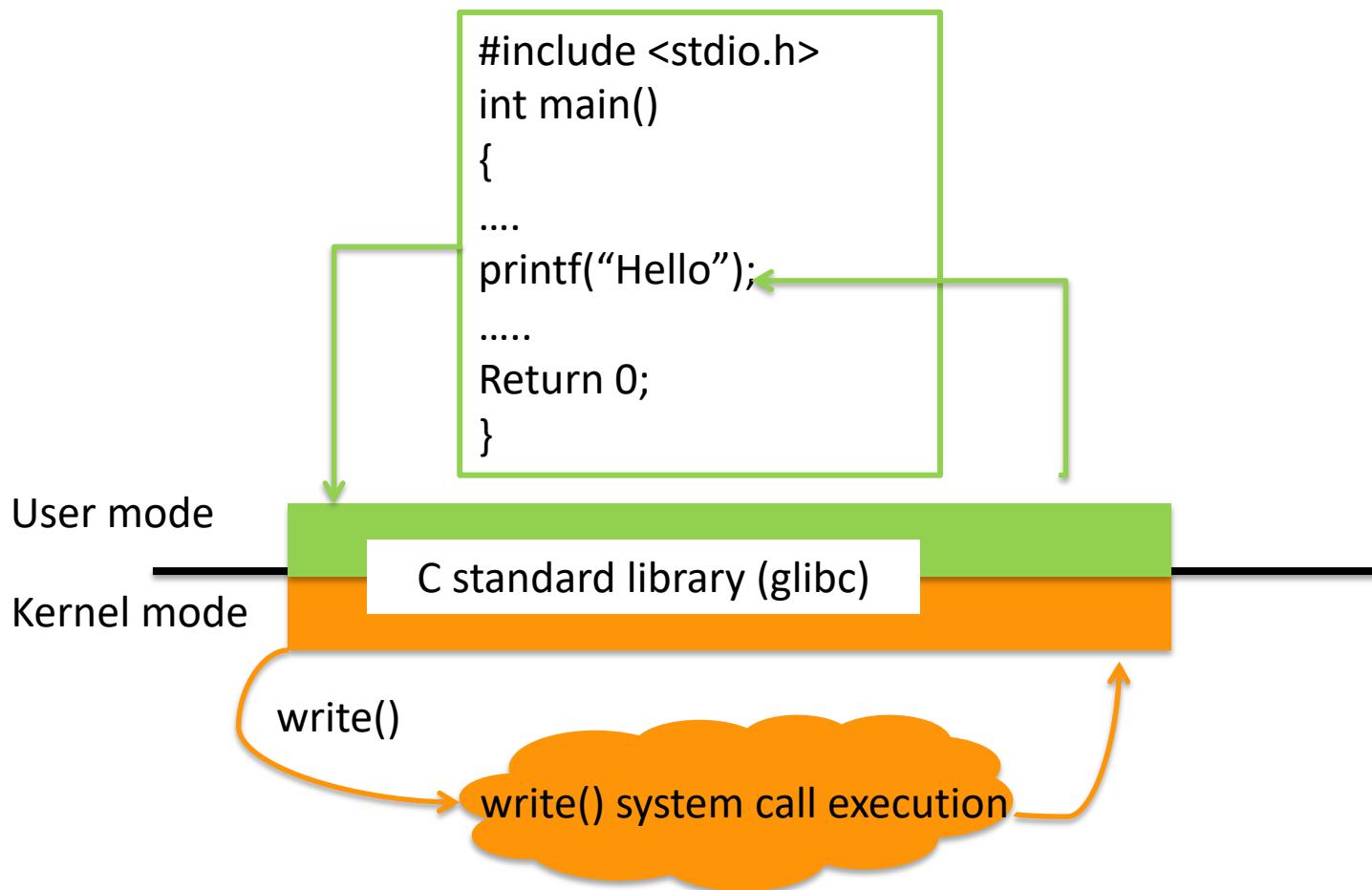
	Files
Open	Open/Create files
read	Read files
write	Write files
close	Close files
lseek	Locate read/write in a file
stat	Get information about file i-node
dup2	Duplicate a file descriptor
pipe	Create a pipe
mkfifo	Create a named pipe (fifo)

System calls

- The **mode bit** differentiates OS executing tasks from user executing tasks
- When the system starts up, the hardware is initialized in kernel mode (mode bit = 0), then the OS is loaded in main memory and the user applications can be launched (mode bit = 1)



- **C language standard libraries**
 - They provide a portable interface to many system calls



- Computer hardware architecture
- Interruptions
- Execution modes
- System calls
- **System utilities**

- Programs that run as user processes and provide a **more comfortable environment**.
- They are provided as part of the OS, but are not essential for machine operation.
- UNIX Examples
 - File management: mkdir, cp, mv, ls
 - Filters: grep, sort, head, tail
 - Editors, compilers, assemblers, linkers ...
 - Window systems : X11
 - Communication : mail, ftp, rlogin
 - Shells : sh, ksh, bash

System utilities

- Code sample “*my_copy*”

```
#define BUFSIZE 1024

void copy(char *from, char *to)
{
    int fromfd, tofd, nread;
    char buf[BUFSIZE];

    if ((fromfd = open(from, O_RDONLY)) == -1)
        { perror(from); exit(1) }
    if ((tofd = creat(to, 0666)) == -1)
        { perror(to); exit(1) }
    while ((nread = read(fromfd, buf, sizeof (buf))) > 0)
        if (write(tofd, buf, nread) != nread)
            { perror("write"); exit(1) }
    if (nread == -1) perror("read");
    if (close(fromfd) == -1 || close(tofd) == -1)perror("close");
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{
    if (argc!=3) {
        fprintf(stderr,"Use: %s forig"
                " fdest\n", argv[0]);
        exit(1); }
    copy(argv[1],argv[2]);
    return 0;
}
```

Summary

- We have analysed:
 - Driver vs controller
 - Types of interrupts
 - Processor execution modes
 - System call mechanism
 - Utilities

Questions

- Which of the following instructions should be privileged?
 - Set value of timer.
 - Read the clock.
 - Clear memory.
 - Issue a trap instruction.
 - Turn off interrupts.
 - Modify entries in device-status table.
 - Switch from user to kernel mode.
 - Access I/O device.
- Some CPUs provide for more than two modes of operation. What are two possible uses of these multiple modes?
- Timers could be used to compute the current time. Provide a short description of how this could be accomplished.
- How many hardware timers are needed to provide timers to processes.