# *Operating Systems*

## Introduction

***Spring 2015***
***Francesco Fontanella***

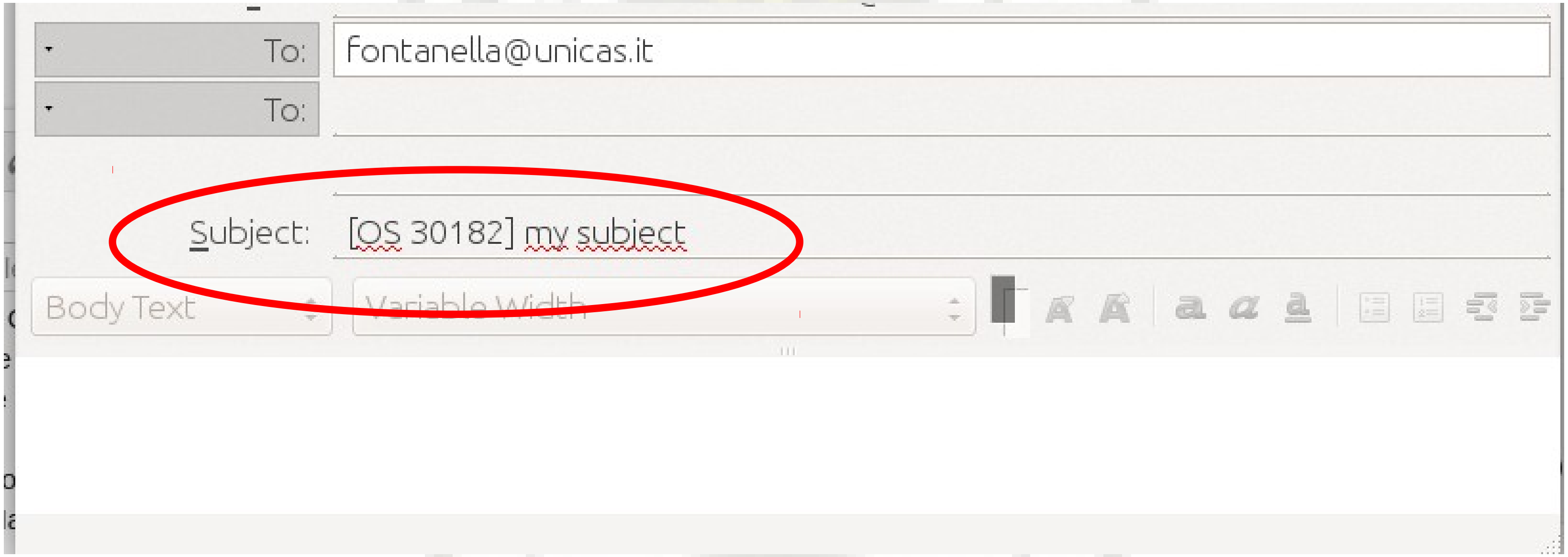
# Instructor

**Francesco Fontanella**

- **E-mail**: `fontanella@unicas.it`
- **Phone**: (+39) 0776 2993382
- **Office hours**:
  - Thursday 11:00-13:00
  - on appointment (via e-mail)
- **Address**: room 20

# E-mails

- When you need to send me an e-mail:



# Course site

- You can find all course stuff on the Piazza site of the course:
  **https://piazza.com/unicas.it/spring2016/os30182/home**
- Piazza also contains a forum, for student collaboration
- You can also post  question to the instructor

# Course organization

- **Class lessons**
  - **Monday**: 11.00 – 13.00 (room 1N.4)
  - **Thursday:** 9.00 – 11.00 (room 1N.4)

- **Lab:**
  - **Tuesday** 15.00 - 18.00 (room 1.4)

# Exam

- Programming practice exam:
  – 50% of grading;

- Written exam:
  – 40% of grading

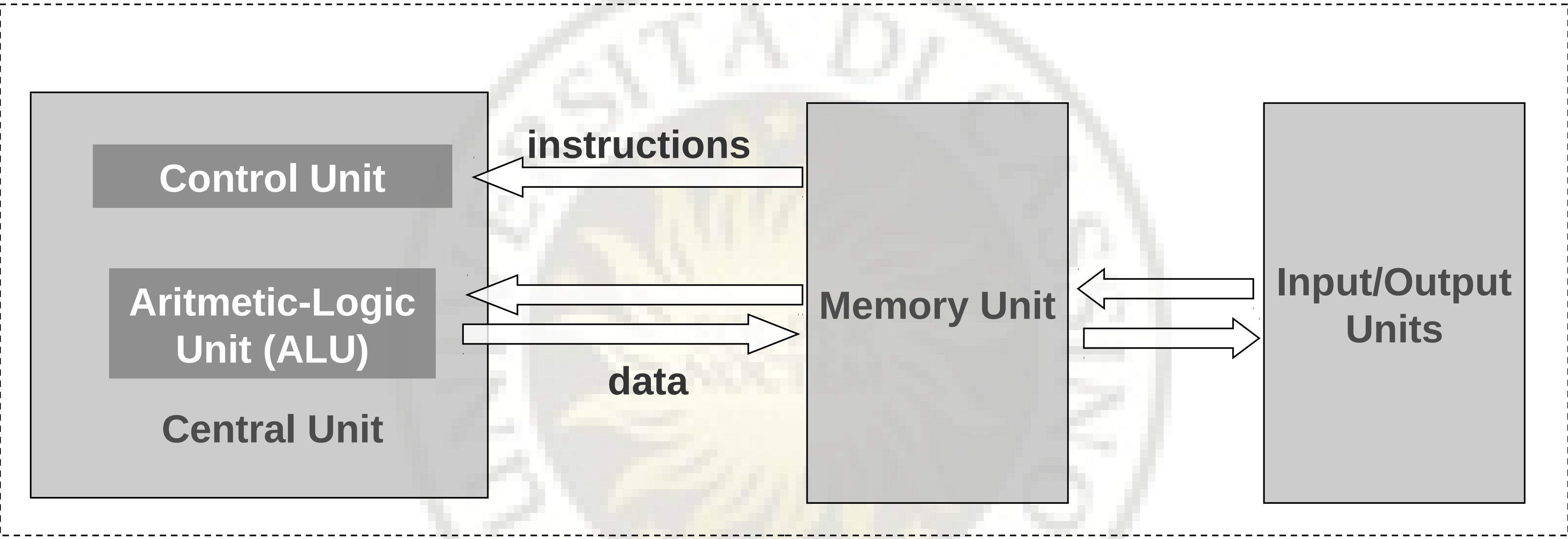
# Homework

- Every week
- Programming assignments
- Submission via dropbox
- 10% of grading

# Course materials

- Textbooks:
  - “Operating Systems, Internals and priniciples”, W. STALLINGS, Perason
  - “Operating Systems concept and examples” (8th ed.), A. SILBERSCHATZ, P.B. GALVIN, G. GAGNE, Pearson.
  - “Modern operating system”, (4th ed.), A.S. TANENBAUM, H. BOS, Pearson
  - “Understanding the Linux kernel”, (3rd ed.) di D.P. Bovet e M. Cesati.
- Lesson slides and some instructor notes

# Von Neumann's Model

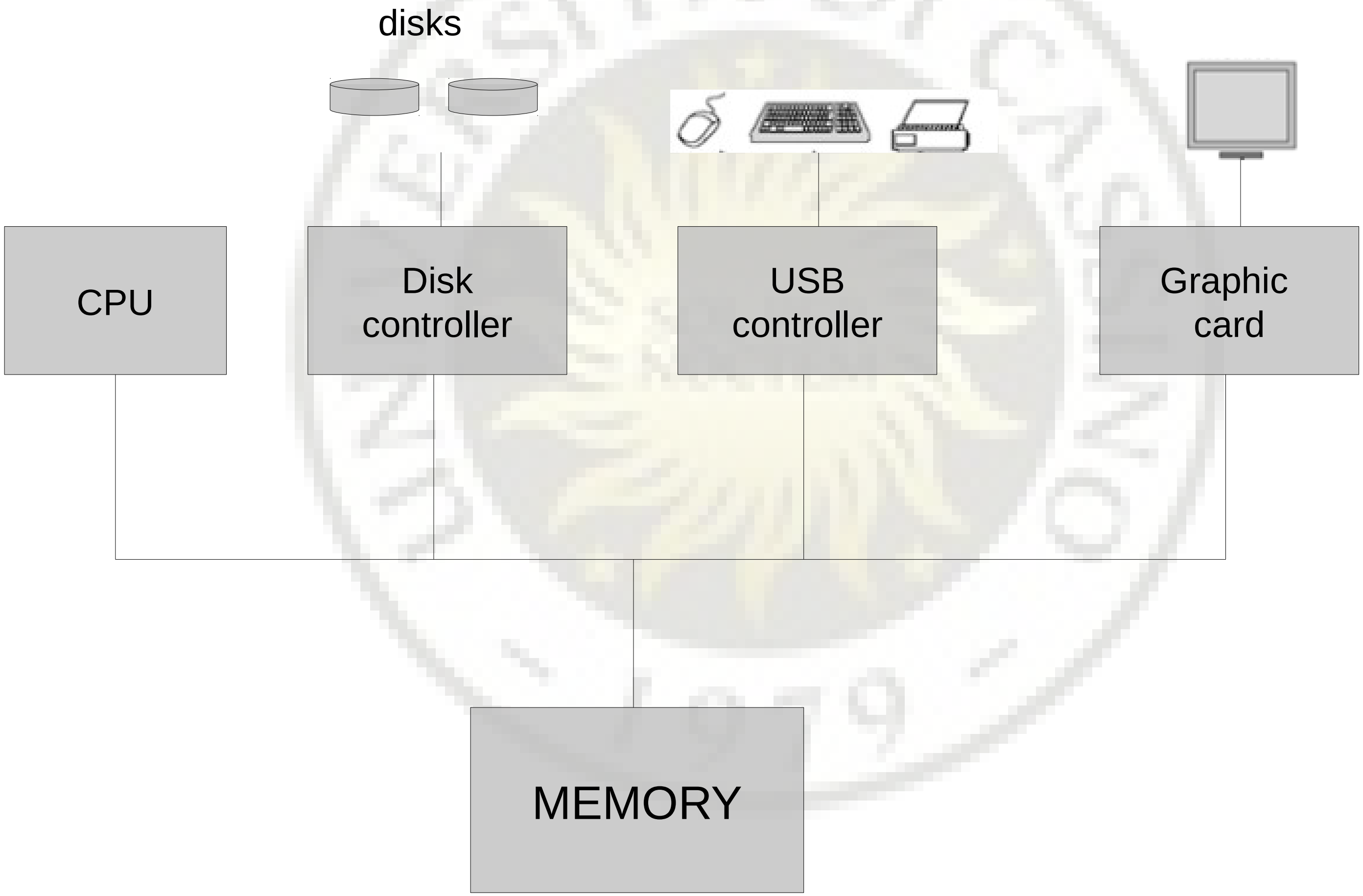


# Modern Computer Systems

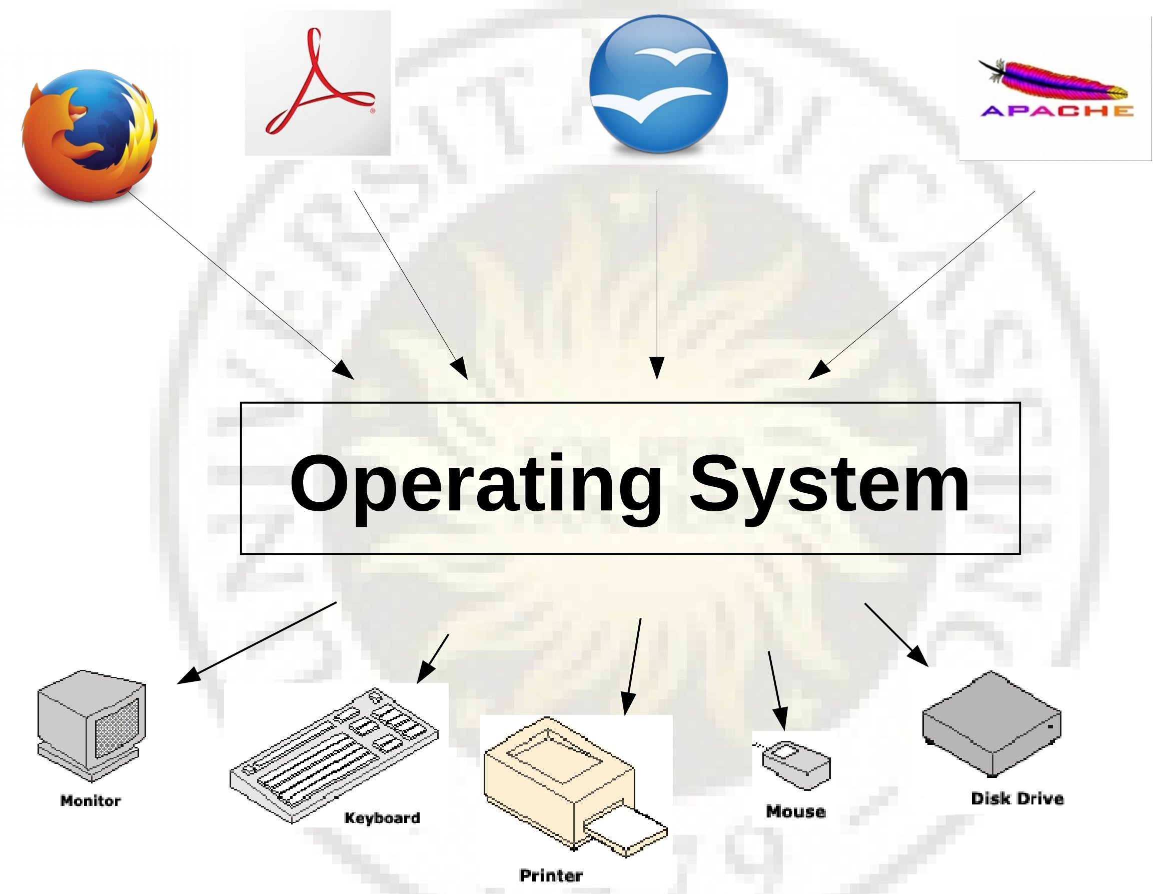
- Von Neumann Model is functionally correct, but very simple.

- Nowadays it exists:
  - Different mass storage, even very different from each other;
  - Many types of peripherical devices

- Peripherical devices offer low level service
- Writing software for managing these devices requires a deep knowledge on how they works
- Hardware architectures may vary a lot


disks
CPU
Disk controller
USB controller
Graphic card
MEMORY

# The Operating System



# Operating system: two definitions

- **Extended machine**
  - hardware abstraction layer,
  - turns hardware into something that application programmers can easily use
  - Top-down perpesctive
- **Resource manager**
  - OS manages the available computer's resources, e.g. CPU time, memory space, etc.
  - Bottom-up perspective

# Operating System ZOO

- Many types of operating systems:
  - **Mainframe/ server**
  - **Smartphone**
  - **Embedded systems**
  - **Wireless sensor networks**
  - **Real-time**
  - **Smart card**

- **Mainframe / Server:**
  - High parallelism
  - Huge I/O workloads  I/O (network, disks, etc,)
  - Example: financial transactions, e-commerce sites, booking and billing systems, etc.

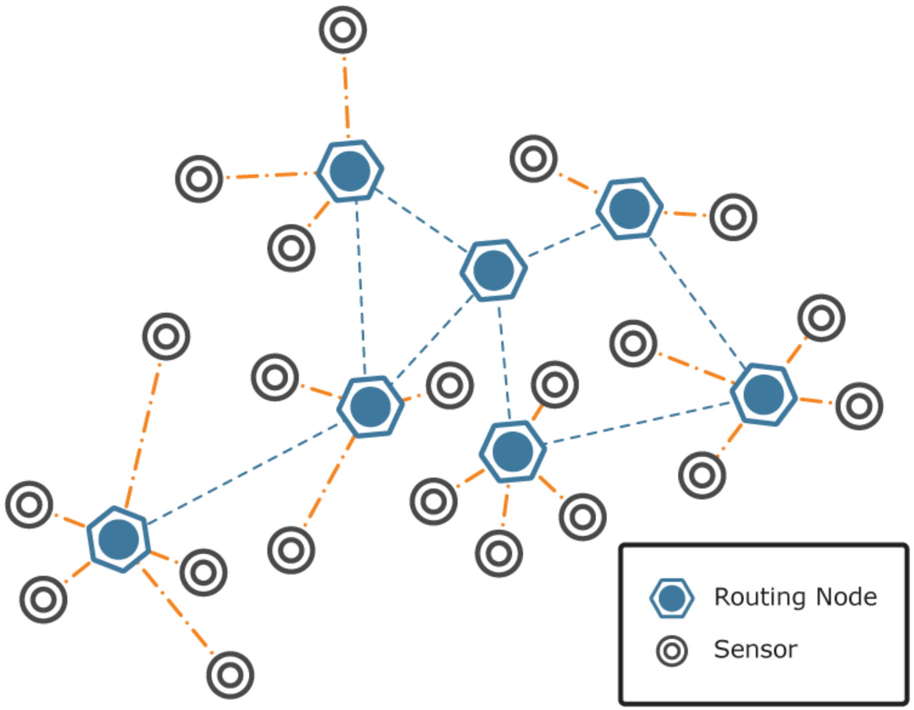
- **Smartphone**
  - Little memory (both RAM and storage)
  - Energy efficiency problems

# Embedded systems

- Developed for managing single devices  (TV, motor controller, etc.)
- On  firmware
- Installed applications are a-priori known
- No protection
- Many of them are **real-time**
- Examples: QNX, VxWorks

# Wireless Sensors

- Wireless sensor (WS) networks can be used in many scenarios: environmental monitoring, battle fields, etc.



- A WS is a very little computer: CPU, RAM, ROM, I/O (sensors, wireless communications)

- The OS must :
  - Be as much  as possible simple
  - Consume as low as possible energy
- Example: TinyOS

# Real Time

- In these OS time is a **key issue**
- Actions must be accomplished within precise time limits. Ex: industrial production (car welding)
- Also in this case applications are a-priori known: the protection problem is much simpler.

- **Hard real-time** systems: the action absolutely must occur within a time range. Missing the limit is harmful.

- **Soft real-time** systems: deadline can be sometime missed (it should be avoided because it represents a performance decay).

# Smart Cards

- Modern smart cards are CPU equipped.
- Strong limits for memory (very little) and I/O (slow)
- Small processing power
- Very simple (some have a single function)
- Most are proprietary
- Recently, JavaCard: OS is a JVM (Java virtual machine), applications are applets (easily portable)

# OS evolution

- OS evolution is strongly tied to hardware evolution:
  - –Hardware technology advances push OS evolution
  - –SO designers drive hardware evolution. Examples: interrupts, memory protection, virtual memory

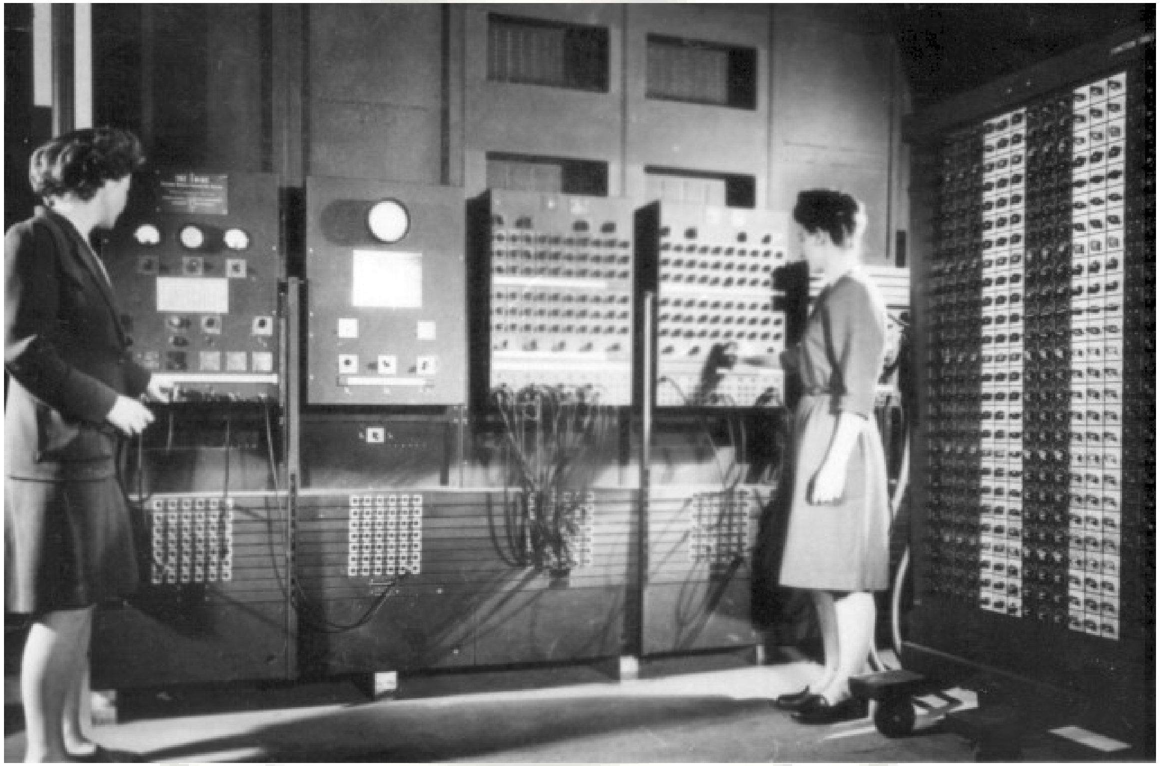
# Earliest Computers

- NO operating systems!
- Programmers interacted directly with the hardware
- Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
- One user at a time (serial access)

# Problems

- **Scheduling:**
  - hardcopy sign-up sheet for time slots
  - wasted (very) expensive CPU time
- **Setup time**:
  - Setting up a program run (named ***job***) needed a lot of time
  - Even more wasted time

# ENIAC

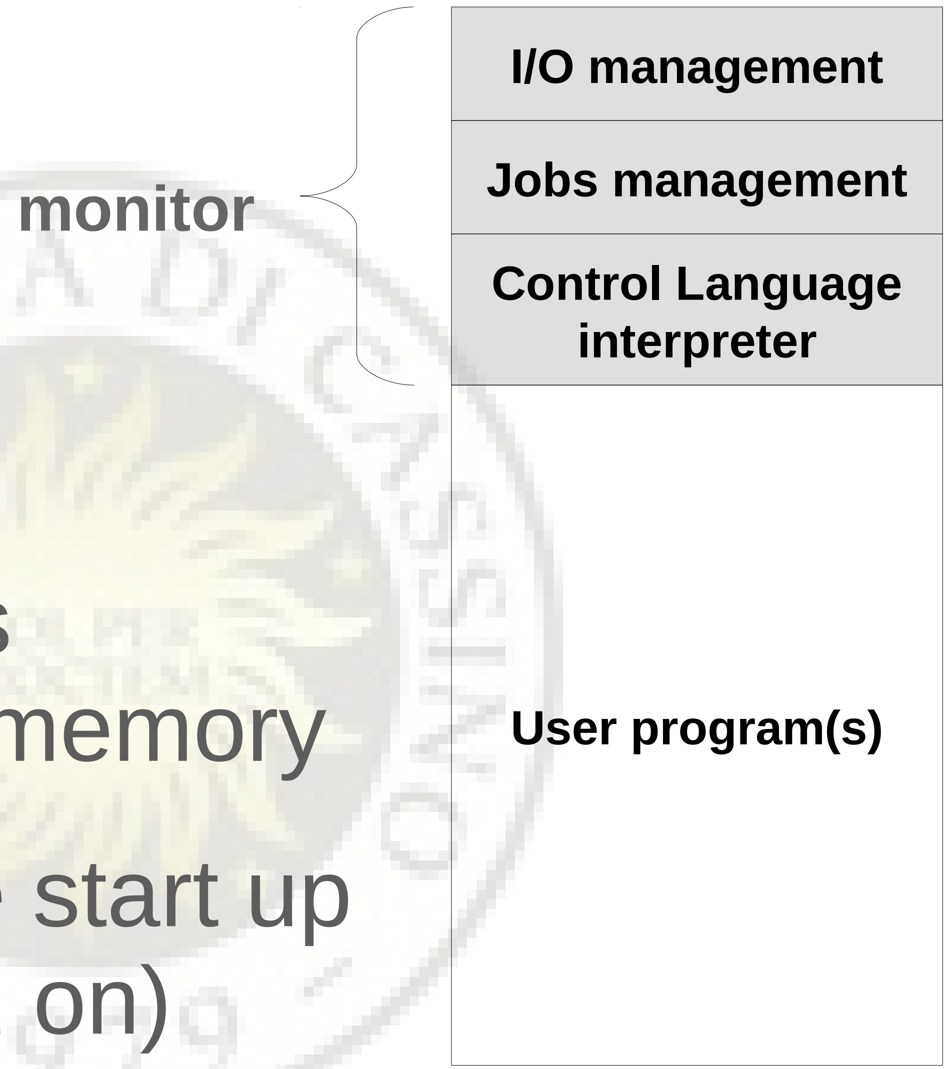


Programmers at ENIAC main control panel

http://en.wikipedia.org/wiki/ENIAC

# Batch systems

- Monitor (the first OS):
  - No CPU direct access
  - jobs  are batched together on an input device
  - Monitor copies job from I/O devices to central memory and gives control to the job
  - At the end the job gives the control back to the monitor




- Monitor is always resident in main memory
- It is loaded at the start up (computer turned on)

# Multiprogramming

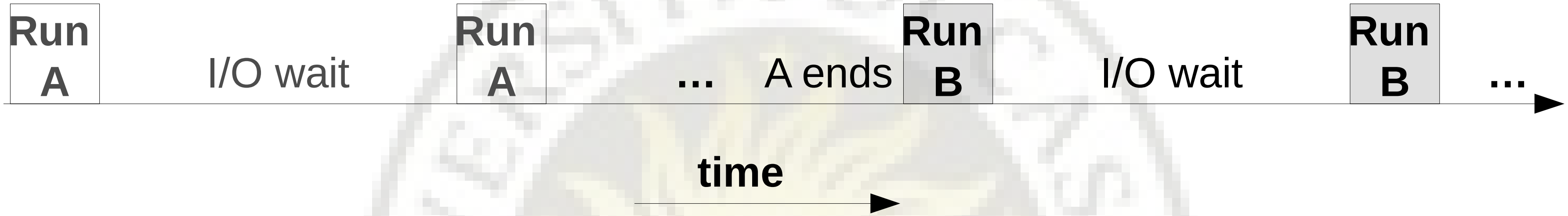
- I/O devices are very slow
- CPU must wait I/O instruction completion
- CPU may be **often idle**.
- Example: database processing

| | |
|---|---|
| **Read data from I/O** | 10 µs |
| **100 CPU instructions** | 1 µs |
| **Write data to I/O** | 10 µs |

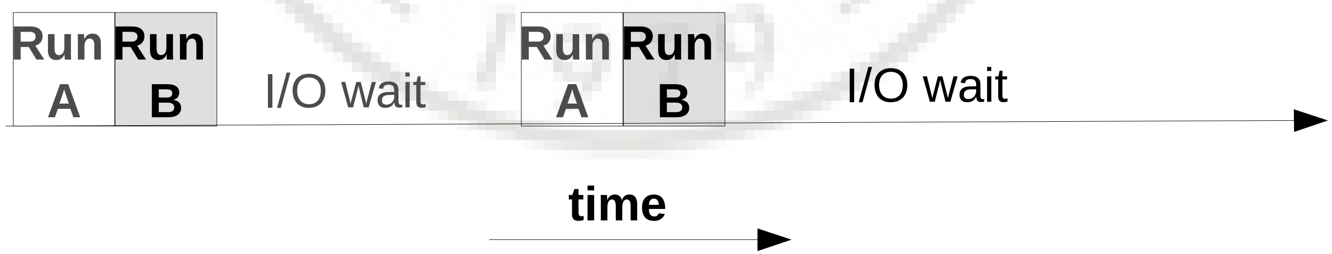
**CPU utilization**

1 (CPU) / 21 (I/O) ~ 5%

# Uniprogramming




# Multiprogramming



- In main memory:
  - all **running programs**
  - the monitor
- Multiprogramming is also known as multitasking
- a program in execution is named **process**

- With multiprogramming new problems arise:
  - Memory management: allocation/deallocation, protection
  - CPU scheduling: choice among more jobs ready to run
  - I/O management: allocation/deallocation, concurrent access

# Time-sharing

- Human beings are much more slower than CPUs
- Time-sharing systems handle multiple **interactive** processes/users (through terminals);
- CPU time is shared among many users:
  - –system clock periodically interrupts the running process

# Operating Systems nowadays

# Hardware protection

- Multiprogramming requires protection. You must avoid that:
  - Concurrent processes interfere each other. Example:
    - process A writes into the memory of the process B
  - User processes interfere with the OS

- You need dedicated hardware

# kernel/user mode

**kernel mode:**

– Processes can execute all instructions, including those which allows the OS to manage the whole system (privileged instructions)

**User mode**

– Processes cannot run privileged instructions

- The CPU has a "Mode bit" in the program status register (PSW register) to distinguish between kernel/user mode

- Examples of privileged instruction:
  - Interrupt disabling
  - Accessing to the I/O port/memory
  - Modifying  the mode bit

# Program Status Word

| | | | |
|---|---|---|---|
| **Condition Code** | **Mode** | **Interrupt Mask** | **Interrupt Code** |

**Condition code**: stores information about the last operation performed by the ALU (Ex: >,<,= zero, overflow, etc.)

**Mode**: running mode: *user mode* (1) or *kernel mode* (0)

**Interrupt Mask**: stores the enabled/disabled interrupts

**Interruput code:** stores the code of the last condition/event which caused the last interrupt

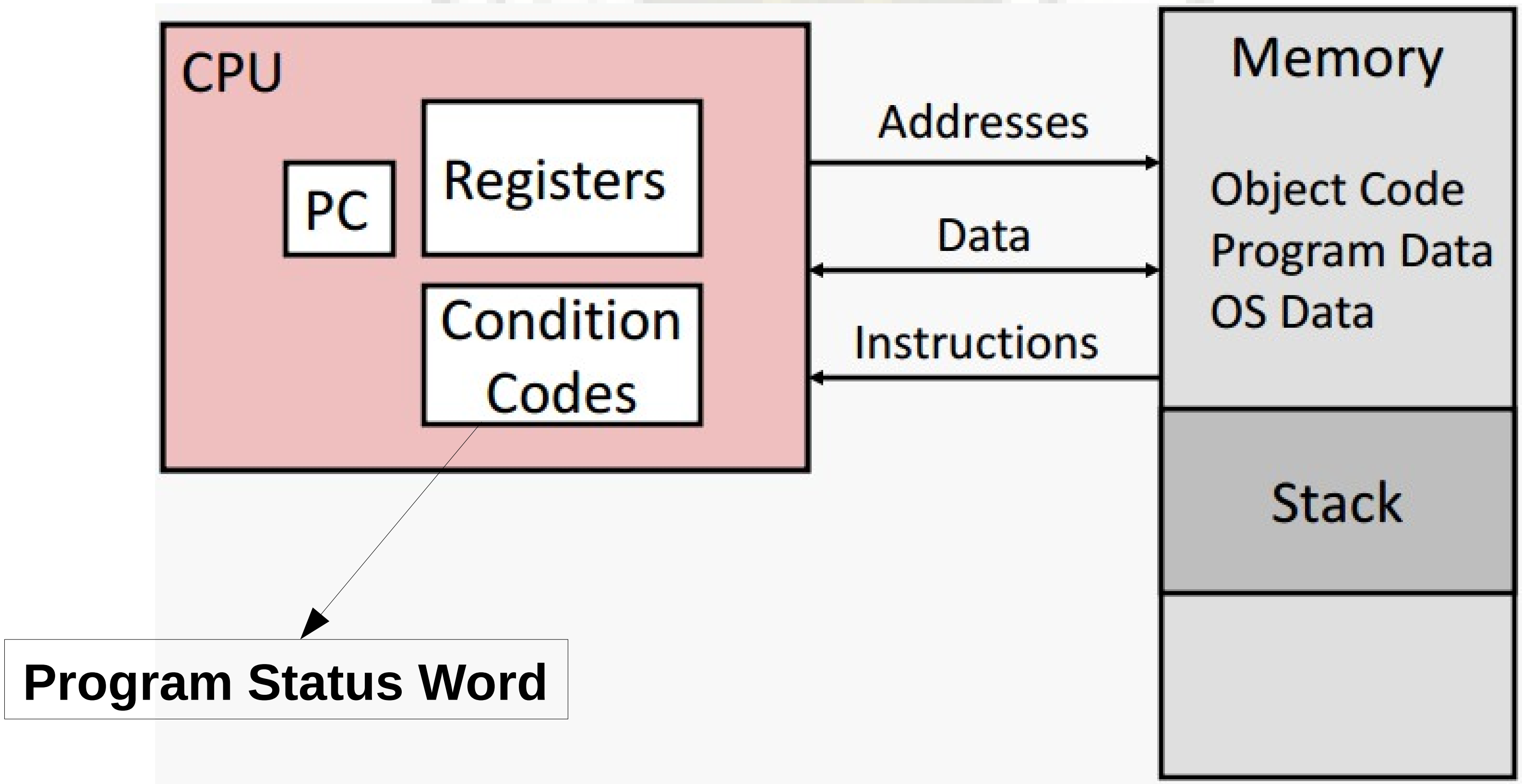
# kernel/user mode

- At  boot time CPU is in kernel mode
- OS is loaded (bootstrap) and then executed
- Before giving the CPU control to user processes, the OS switches the CPU in user mode
- **Interrupts automatically switch the CPU mode kernel**

# CPU State

- CPUs have internal registers:
  - **General-purpose registers (GPRs)**: can be modified by programs and OS (program-accessible registers), and may contain: data, addresses, stack pointers, etc.
  - **Control registers:** PSW, Program Counter, etc.
- The values contained in these registers identify the (so called):

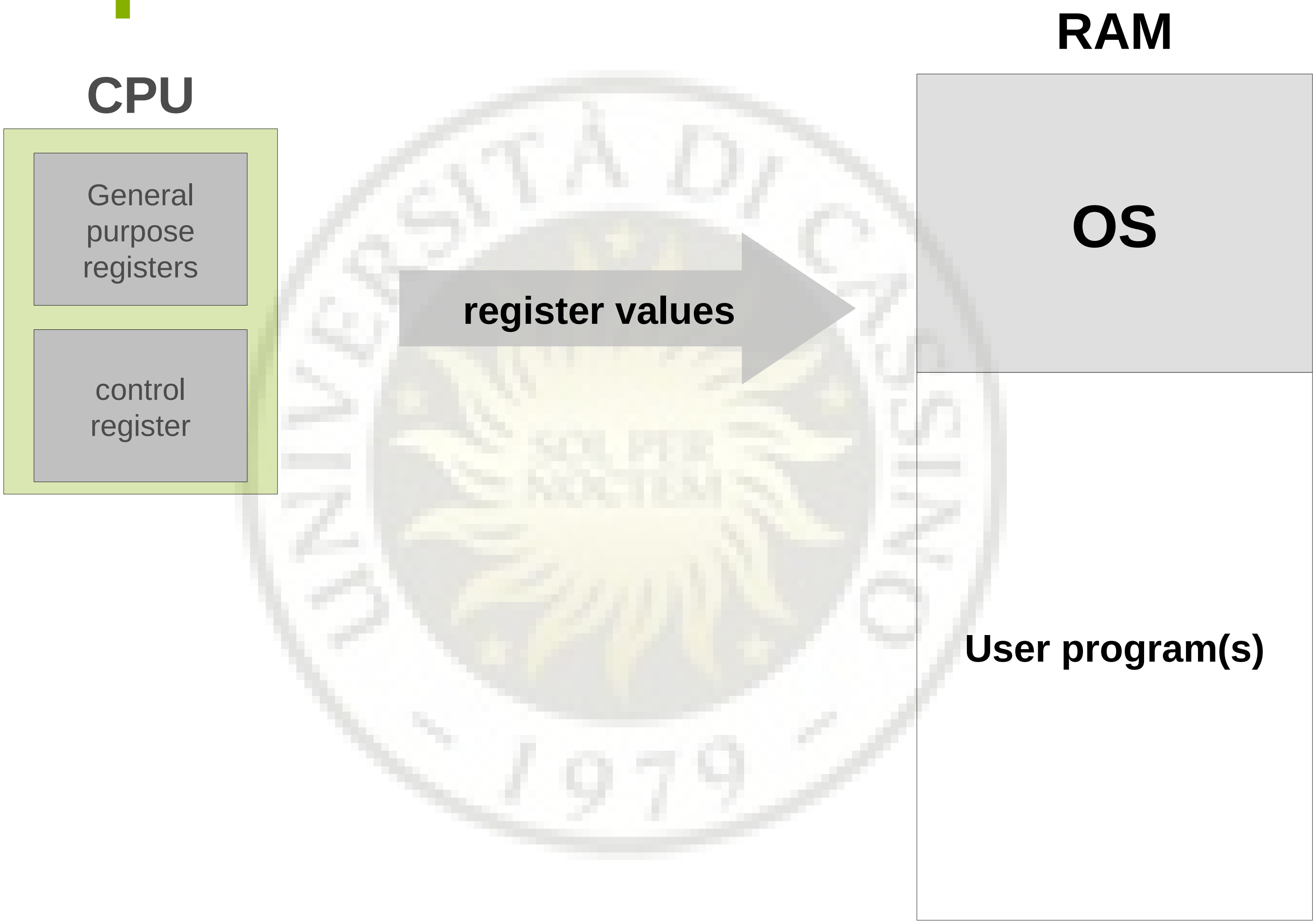
**CPU state**


CPU
PC
Registers
Condition Codes
Addresses
Data
Instructions
Memory
Object Code
Program Data
OS Data
Stack
Program Status Word

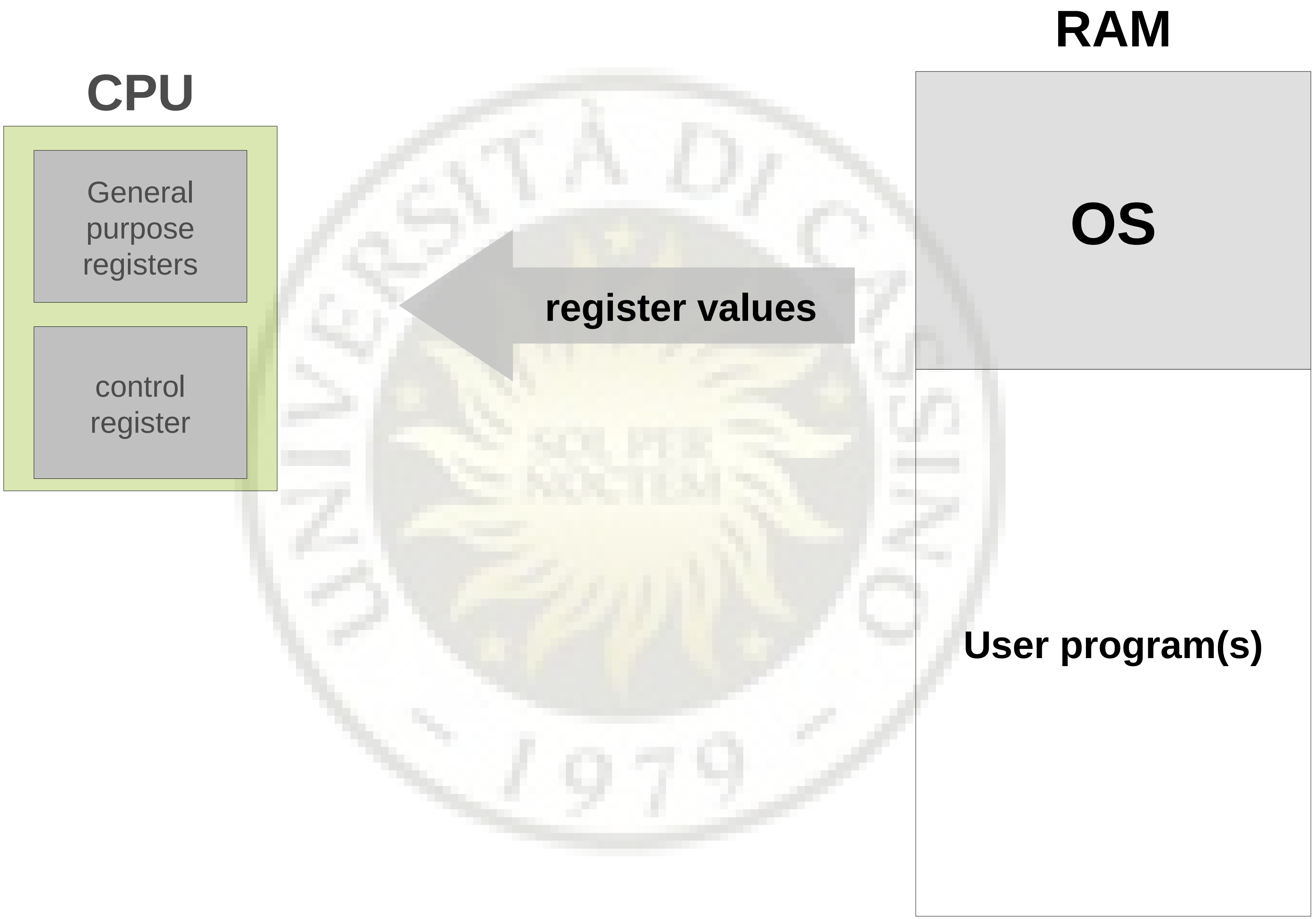
- You can imagine the set of values of the CPU registers like a snapshot: they exactly represent **what** the CPU was doing **at the moment** they was stored

- OS can stop/restart any running program by storing/restoring these register values

# Stop

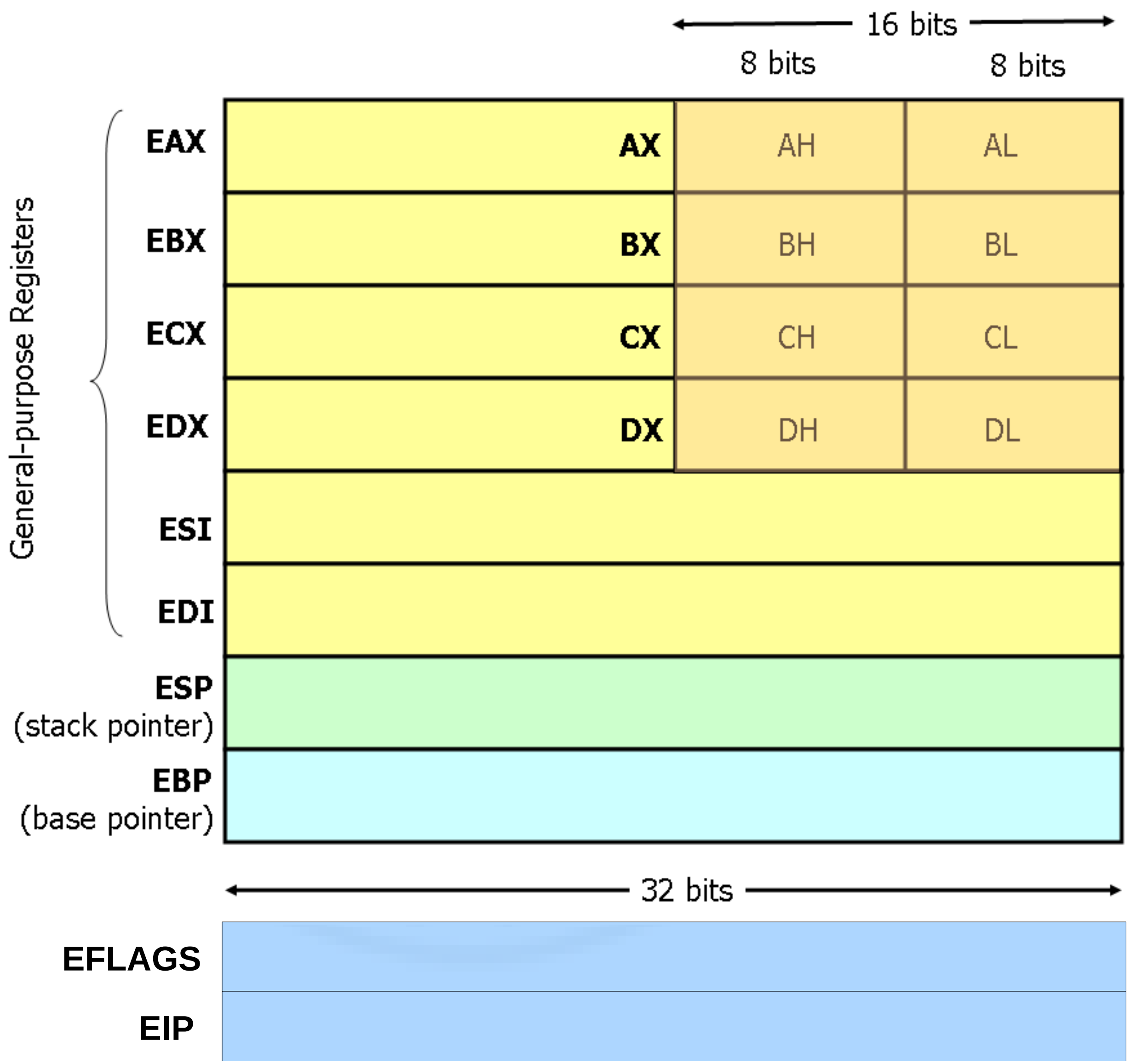

RAM
CPU
General purpose registers
control register
register values
OS
User program(s)

# Restart

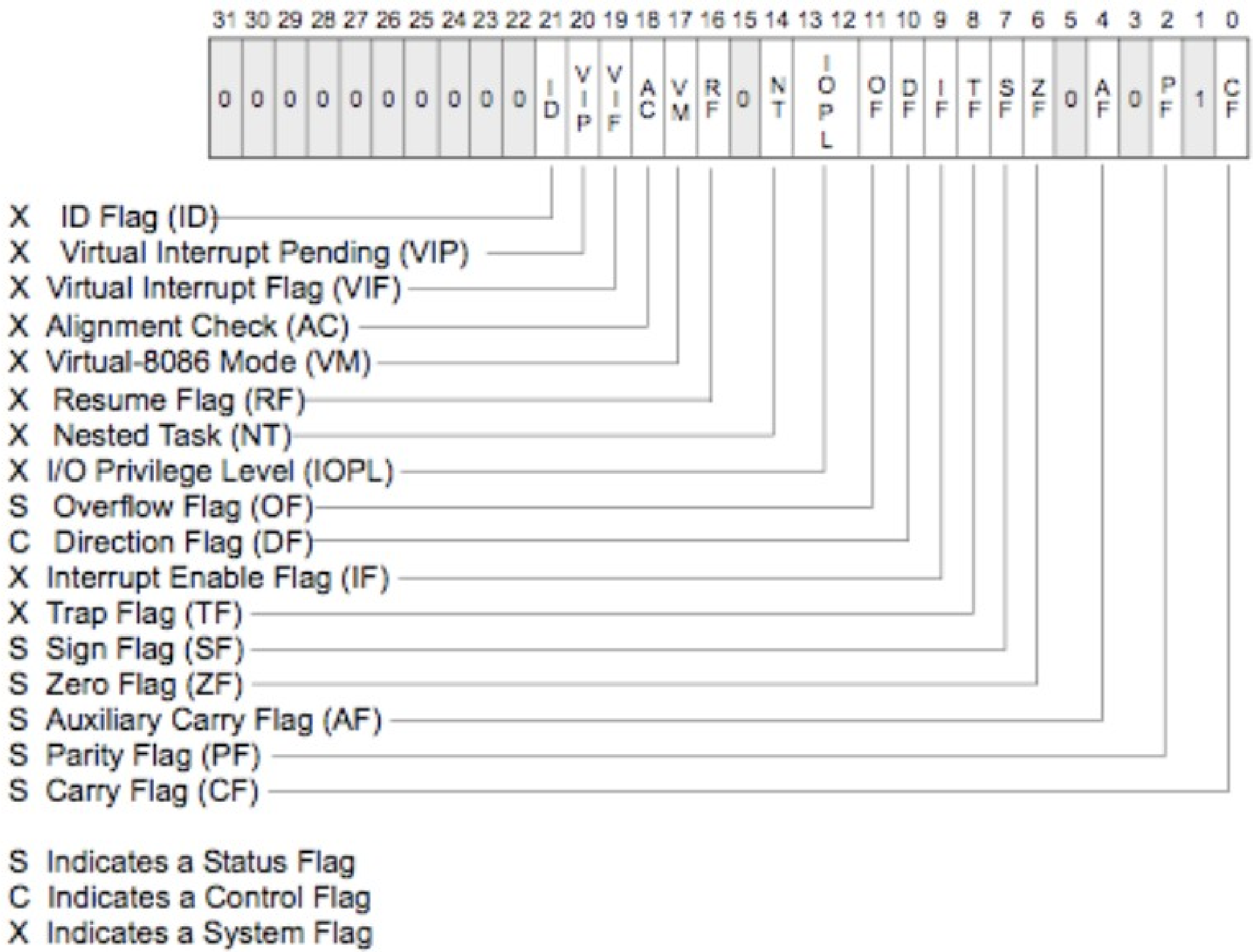


RAM
CPU
General purpose registers
control register
register values
OS
User program(s)

# X86 Registers



# EFLAGS register



# EIP register

- EIP register contains the address of the next instruction to be executed (it is the **program counter** register of the INTEL architecture)

- Its value can be modified, in two ways:
  - Automatically incremented (by the hardware) during the execution of the current instruction
  - by control instructions:
    - JMP, Jxx, CALL, RET, nRET, IRET,

# X86 Instructions (assembly)

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>

push <reg32>
push <mem>

pop <reg32>
pop <mem>
```

```
add <reg>,<reg>
add <reg>,<mem>

sub <reg>,<reg>
sub <reg>,<mem>

inc <reg>
inc <mem>
```

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

# Interrupts and traps

- They allow OS to stop the normal fetch-execute cycle of the CPU
- The OS  gets the control over the CPU to stop the running program
- Always in **kernel mode**
- Either hardware (interrupts) or software (traps)
- Cause the execution of OS code (handlers)

# Interrupt vs trap

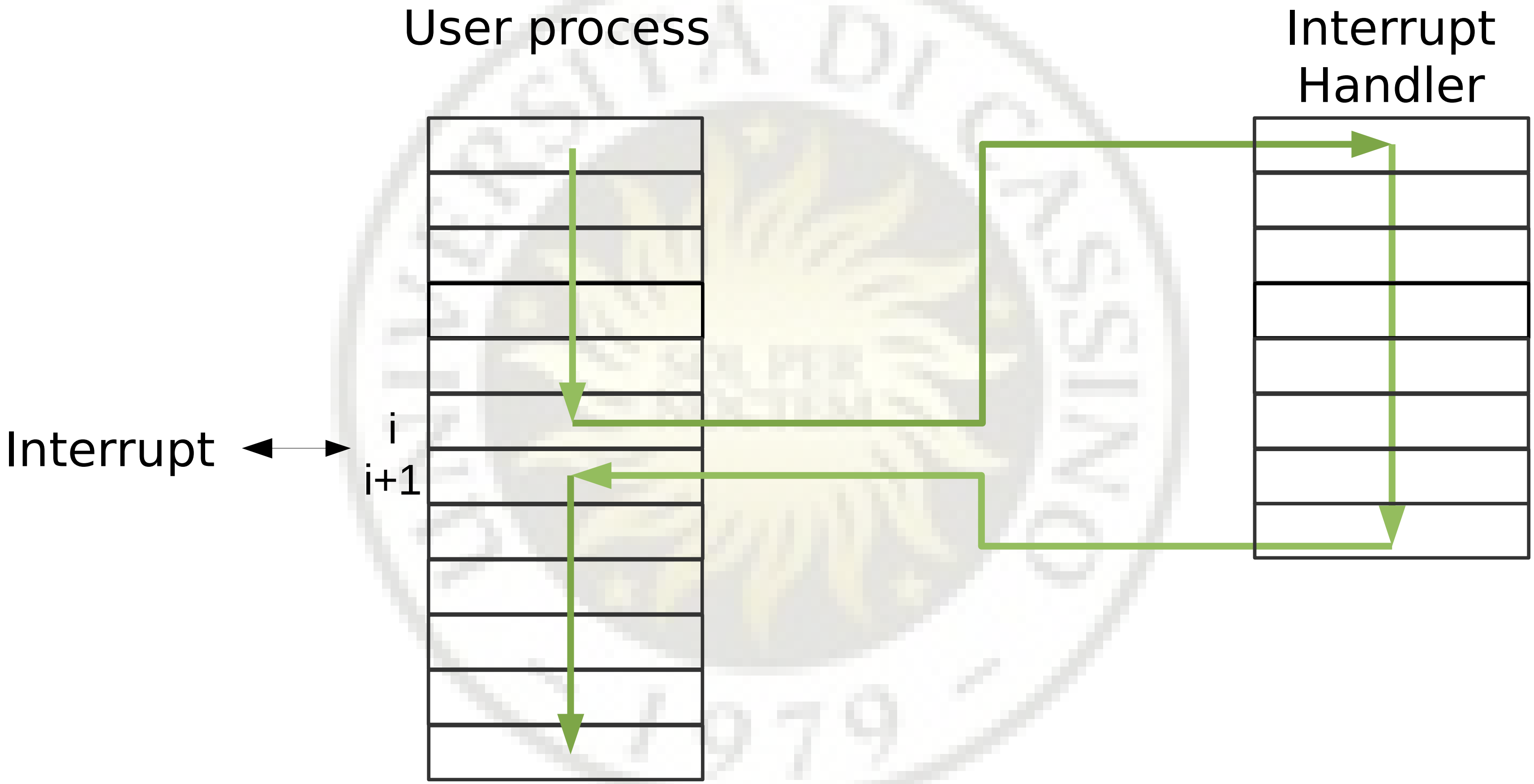
**Interrupt**

- **asincronous hardware** event, generated by
  - I/O devices (disks, keyboards, mouse, etc)
  - system clocks (time quantum expired)

**Trap**

- **sincronous software** event, generated by program in execution :
  - Programming errors: Division by zero, memory addressing errors
  - Requests of service to the OS (system calls)

# Interrupt



# “Event Driven” OS

- OS intervenes when  certain events occur:
  - **interrupts** by  peripheral devices (disks, mouse, keyboard, clock, etc)
  - **traps** by the executing program (errors or syscalls) System calls or program expections by user programs

# OS "Interrupt Driven"

- After every instruction the CPU check if any interrupt occured

```
while (fetch next instruction) {
  run instruction;
  if (interrupt) {
    save EIP and EFLAGS                  // user mode
    jump to the interrupt handler        // kernel mode
    restore EIP                          // user mode
  }
}
```

# Questions

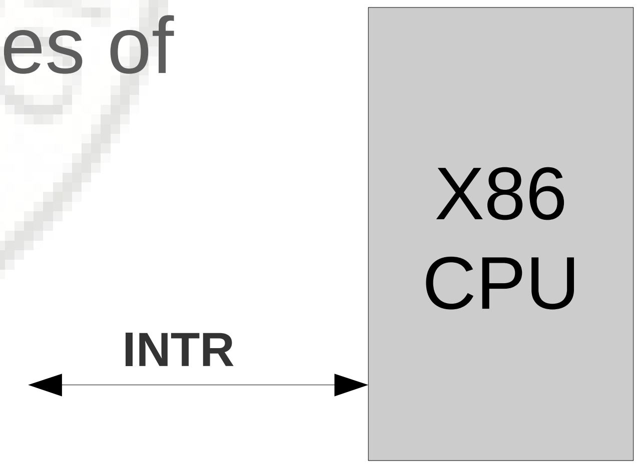
1) How does CPU check if an interrupt has occurred?

2) How does CPU know which instruction to execute next?

3) What does the interrupt handler do?

# Answer 1

- (Modern) CPUs have a special line connected to all the I/O devices
- After every instruction,  the CPU checks the line
- It the line is up, the CPU (its hardware):
  – interrupts its normal execution cycle
  – Automatically saves the values of EIP and EFLAGS registers

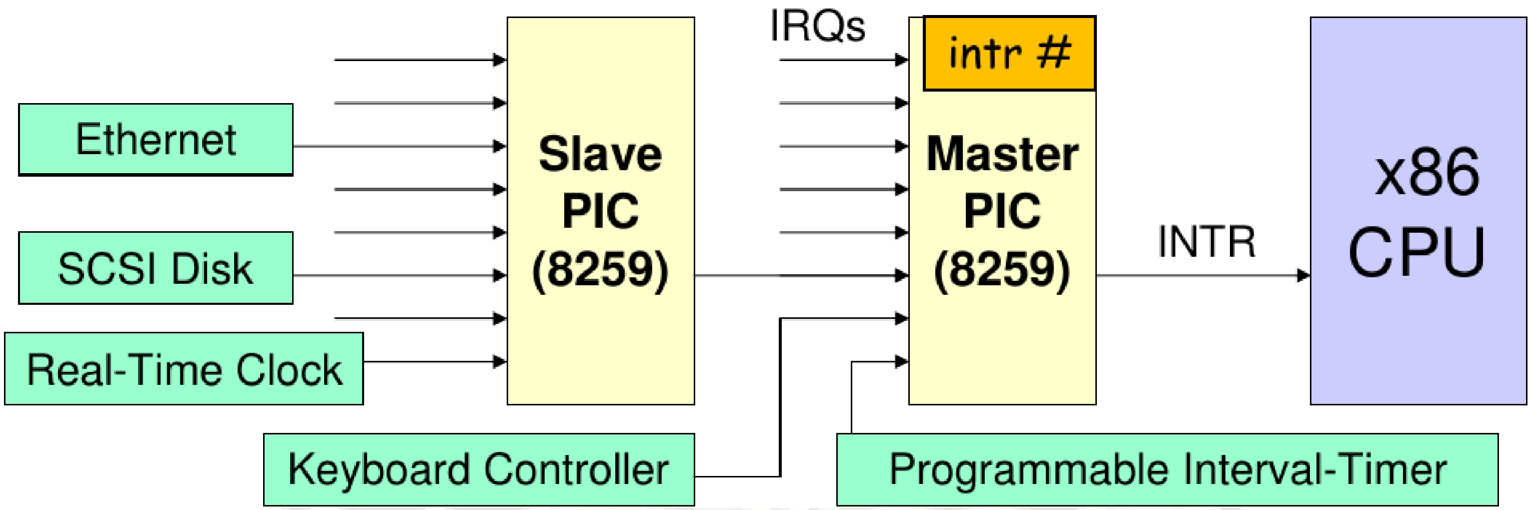


# Answer 2

- Each device is assigned an interrupt number
- At boot time the OS loads in memory the **Interrupt Description Table (IDT)**, also called **Interruput vector**
- IDT entries point to an **interrupt handler:**
  - a special routine able to manage the device that generated the interrupt
- In the x86 CPU the OS can use the instruction `lidt` to load in the **IDT register** the address and the size of the IDT

# Programmable Interrupt Circuit (PIC)

- I/O devices trigger interrupt requests to the PIC
- The PIC:
  - associates  at each device an interrupt request (IRQ) number
  - activates the INTR of the CPU


IRQs
intr #
Ethernet
SCSI Disk
Real-Time Clock
Slave
PIC
(8259)
Master
PIC
(8259)
INTR
x86
CPU
Keyboard Controller
Programmable Interval-Timer

# Interrupt Descriptor Table (IDT)

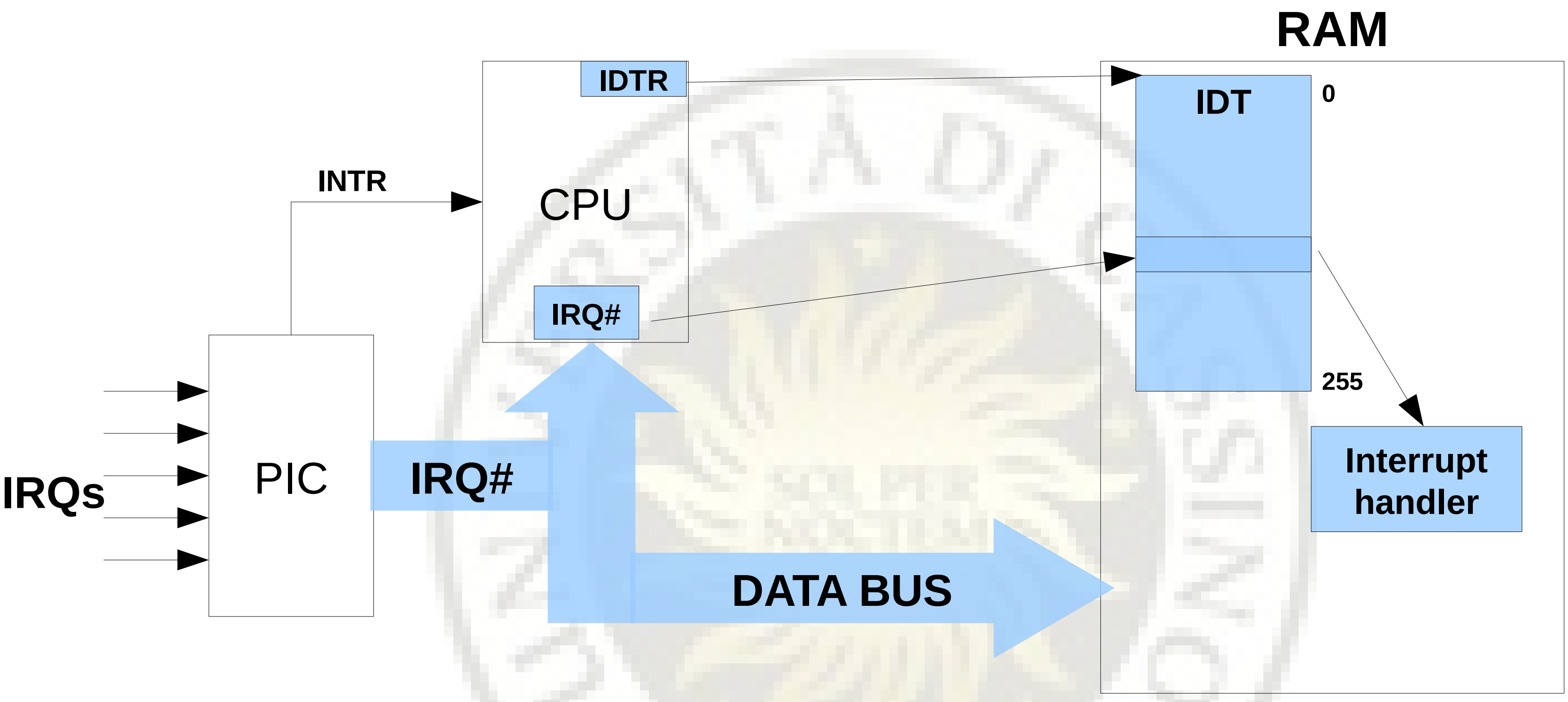
- In the x86 architecture implements the interrupt vector
- It may contain up to 256 entry (8 bytes each). The first 32 are reserved to the CPU
- It can be anywhere in main memory. The address of the first entry is in the IDTR register
- For each device, the IDT makes a connection between the IRQ number (IRQ#) of the device and the instructions to execute for managing its interrupt requests (the handler)

**`Handler's address = IDT[IRQ#]`**

# Interrupt mechanism

- If the INTR line is up, the CPU (automatically):
  - Stores on the stack the current values of the EIP and EFLAGS registers
  - Switch in kernel mode
  - Loads from the data bus IRQ# (from the PIC)
  - Loads in the EIP the address stored at:

**`IDTR+8*IRQ#`**




In practice the CPU automatically jumps to (execute) the handler of the device which generated the interrupt

# Interrupt Handler

- What does the interrupt handler do?
- Usually, the handler:
  - Uses an assembly routine to save the register values (the **context**)
  - Calls a routine (written in C) to manage the interrupt. Example: read/write of the device registers
  - Restores the context of the interrupted process and give the control back to it or (sometimes) call the scheduler

# Linux: the save_ALL macro

- Linux interrupt handlers start by calling this macro
- The instruction **`push %reg`** saves on the stack the value of the register **`%reg`**

```
cld
push %es
push %ds
pushl %eax
pushl %ebp
pushl %edi
pushl %esi
pushl %edx
pushl %ecx
pushl %ebx
movl $ __USER_DS,%edx
movl %edx,%ds
movl %edx,%es
```

# Keyboard interrupt handler (C code)

```
void irq_handler(int irq, ...)
{

   static unsigned char scancode;
   unsigned char status;

   /* Read keyboard status */
   status = inb(0x64);
   scancode = inb(0x60);
          .
          .
          .


}
```

# Interrupt management: overview

- When a device interrupt occurs:
  - The interrupt request is sent to CPU (via the INTR line)
  - The CPU
    - Stops the running process
    - Jump to the address containing the routine for managing that interrupt (**interrupt handler**)
  - L'interrupt handler
    - manage the interrupt
    - Give the control back to the stopped process (or to another process)
    - The interrupted process resume its computation, as if nothing ever happened

Hardware

Software

# Interrupt management: details

- The change of the value EIP register imply a jump to the code of the handler
- At this point:
  - the CPU resume its normal fetch-execute cycle
  - The (OS) handler  takes the control of the CPU

# Multiple interrupts

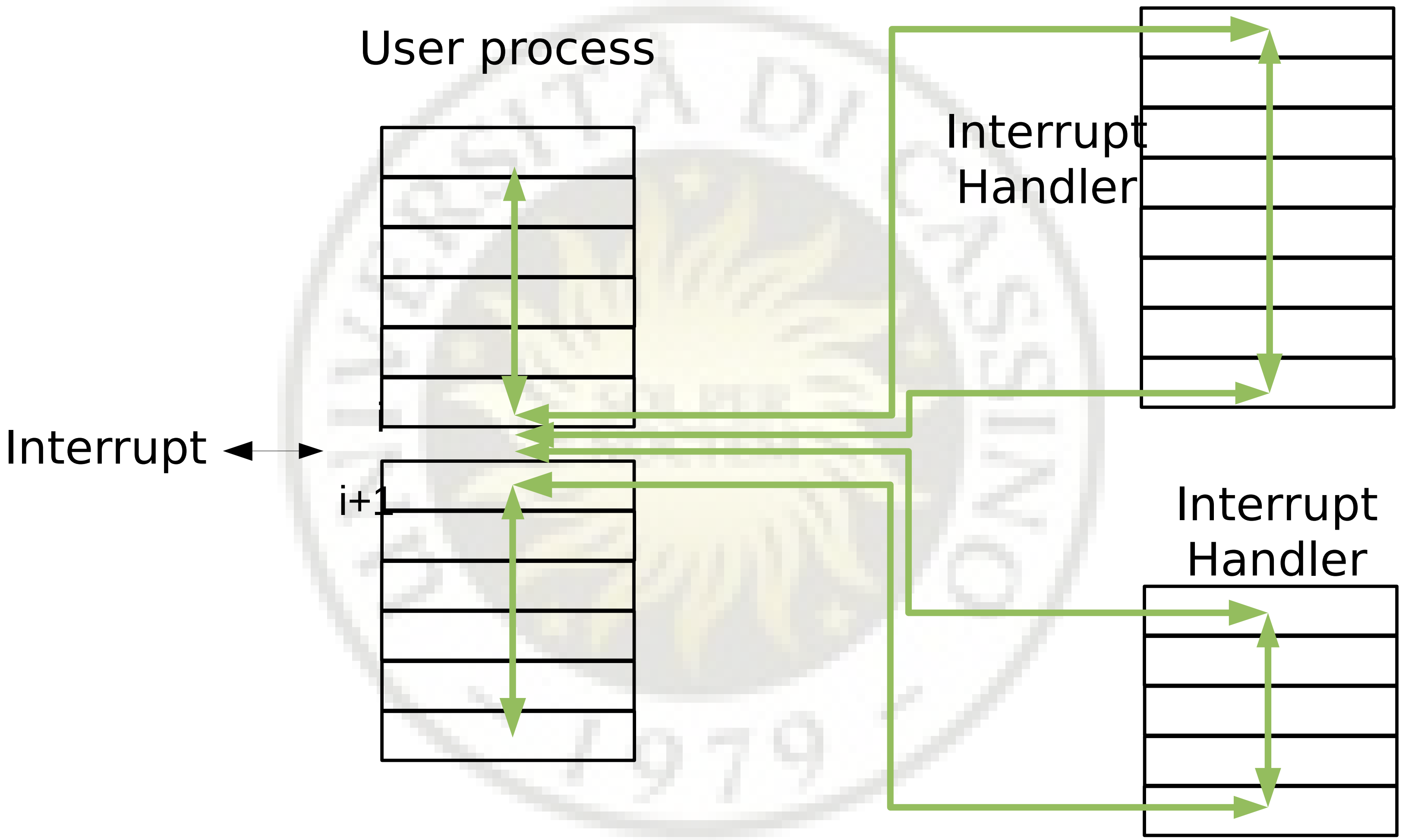
- During the management of an interrupt a new interrupt from a different device may occur;
- Two possible solutions:
  – Interrupt disambling
  – Nested interrupts

# Interrupt disabling

- When an interrupt is served new interrupt are (temporarily) ignored (the IF flag of the EEFLAGS is set down);
- The ignored interrupt is pending;
- interrupts are reenabled after that the interrupt has been served;

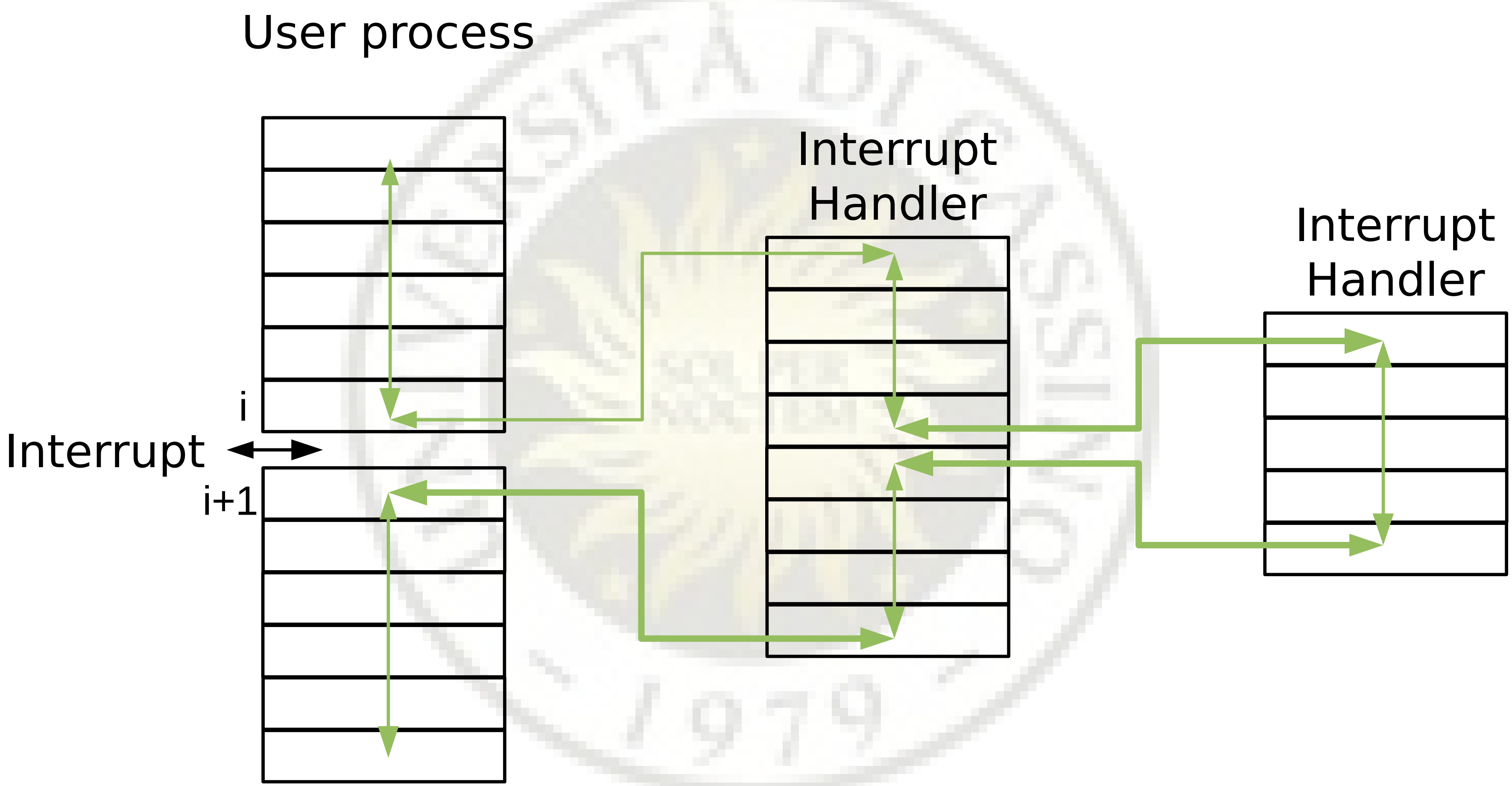
# Interrupt disabling

- The CPU then check if a new interrupt occurred; if so the corresponding handler is called
- Simple approach: interrupts are managed  sequentially
- Does not take into account "time-critical" conditions


User process
Interrupt
Handler
Interrupt
i
i+1
Interrupt
Handler

# Nested Interrupt

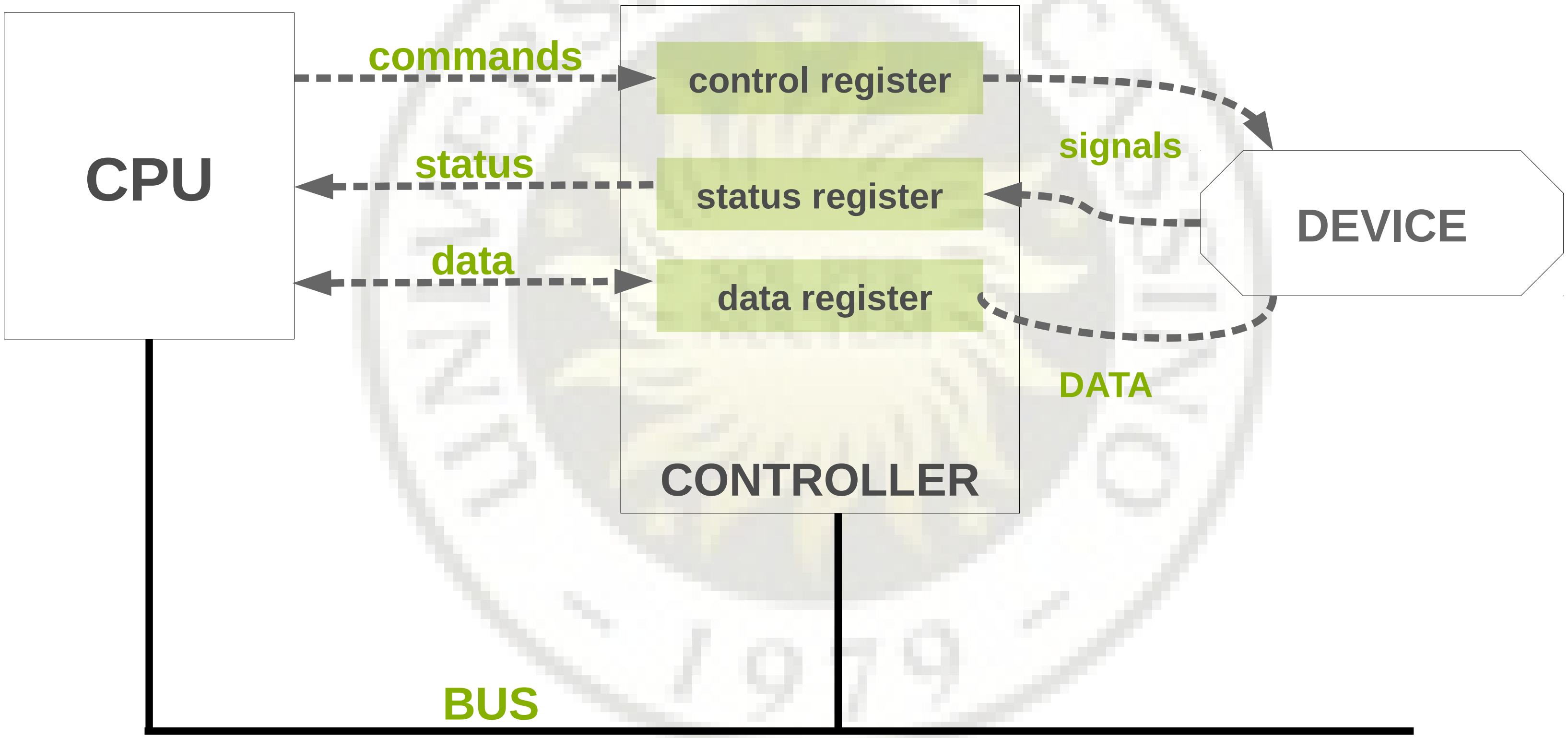
- Priorities
- Lower priority interrupts can be stopped by higher priority interrupts
- It  needs a suitable mechanism for restore the previous interrupt
- Faster device (network cards) usually have higher priority


User process
Interrupt
Handler
Interrupt
Handler
i
Interrupt
i+1

# I/O devices

- Every I/O device is managed by the OS through its **controller**

- An I/O controller is an electronic device which accept commands from the OS and performs the corresponding  action


CPU
commands
status
data
control register
status register
data register
CONTROLLER
signals
DEVICE
DATA
BUS

- Access policies to devices depends on their controllers

**Example**

- –disk controllers accept one request at time
- –Queuing disk requests is an OS task

- Three ways to manage the interaction between OS and I/O devices:
  - **Programmed I/O**
  - **Interrupt-Driven I/O**
  - **Direct Memory Access (DMA)**

# Programmed I/O: input

1) OS loads the input request parameters into the control register of the controller.

2) The controller starts to execute the request

3) **The OS starts a cycle to check the device status register (busy wait cycle)**

4) Once the data are available, the controller:
   1) stores them into its own memory buffer
   2) uses the status register to inform the OS that the operation has been completed

5) Finally, the OS copies the data from the controller buffer to the main memory.

# Interrupt-Driven I/O: Input

1) OS loads the input request parameters into the control register of the controller.

2) The controller starts to execute the request

3) **The OS assigns the CPU to another process**

4) Once the data are available, the controller
   1) stores them into its memory buffer
   2) **generates an interrupt to inform the OS that the operation has been completed**

5) Finally, the OS copies the data from the controller buffer to the main memory

# Programmed I/O and Interrupt-Driven I/O

- Output operations are quite similar:
  1) data are copied into controller buffers
  2) Then request parameters are loaded into controller command registers

- **drawbacks:**
  – CPU time is wasted for data transferring
  – Data throughput depends on the (busy) CPU

# Direct Memory Access (DMA)

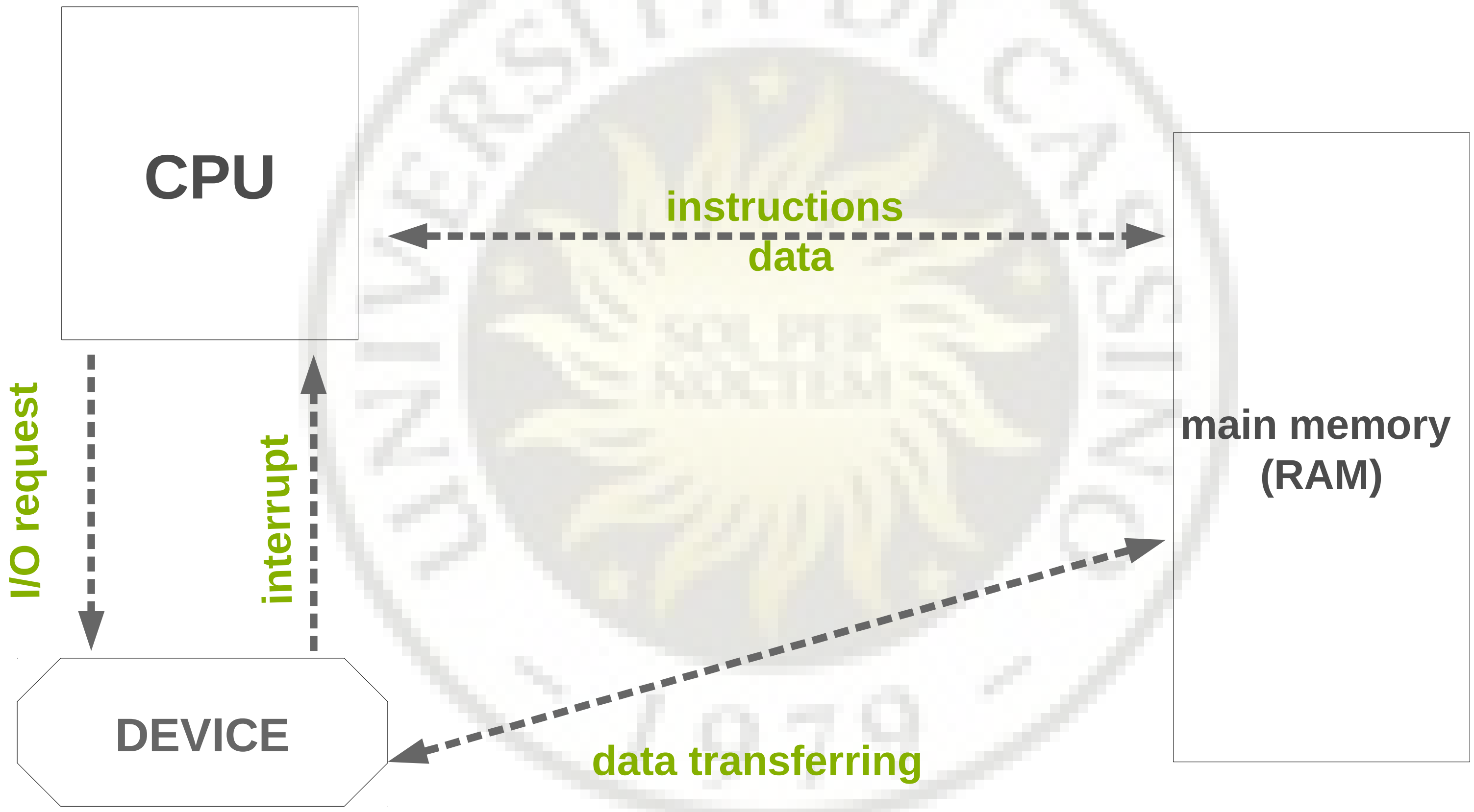
1) OS loads the input request parameters into the control register of the controller.

2) The controller starts to execute the request

3) **The OS assigns the CPU to another process**

4) Once the data are available, the controller
   1) **stores them directly from/to the main memory**
   2) generates an interrupt to inform the OS that the operation has been completed

# Direct Memory Access (DMA)



# System calls

**question**

- I/O instructions  can executed only in kernel mode, by the OS. How can user processes execute  I/O operations?

**answer**

- User processes must request I/O operations to the OS, through the **system calls** (or **syscall**).

- The set of available syscalls represents the **interface** between user processes (their programmers) and the OS (services)
- When a user process needs a service from the OS, it makes a **system call**
- In programming languages, syscalls are available through routines collected in libraries

- These libraries are usually provided with the compiler

- **EXAMPLE (**C language)
  - `printf`
  - `read`
  - `write`

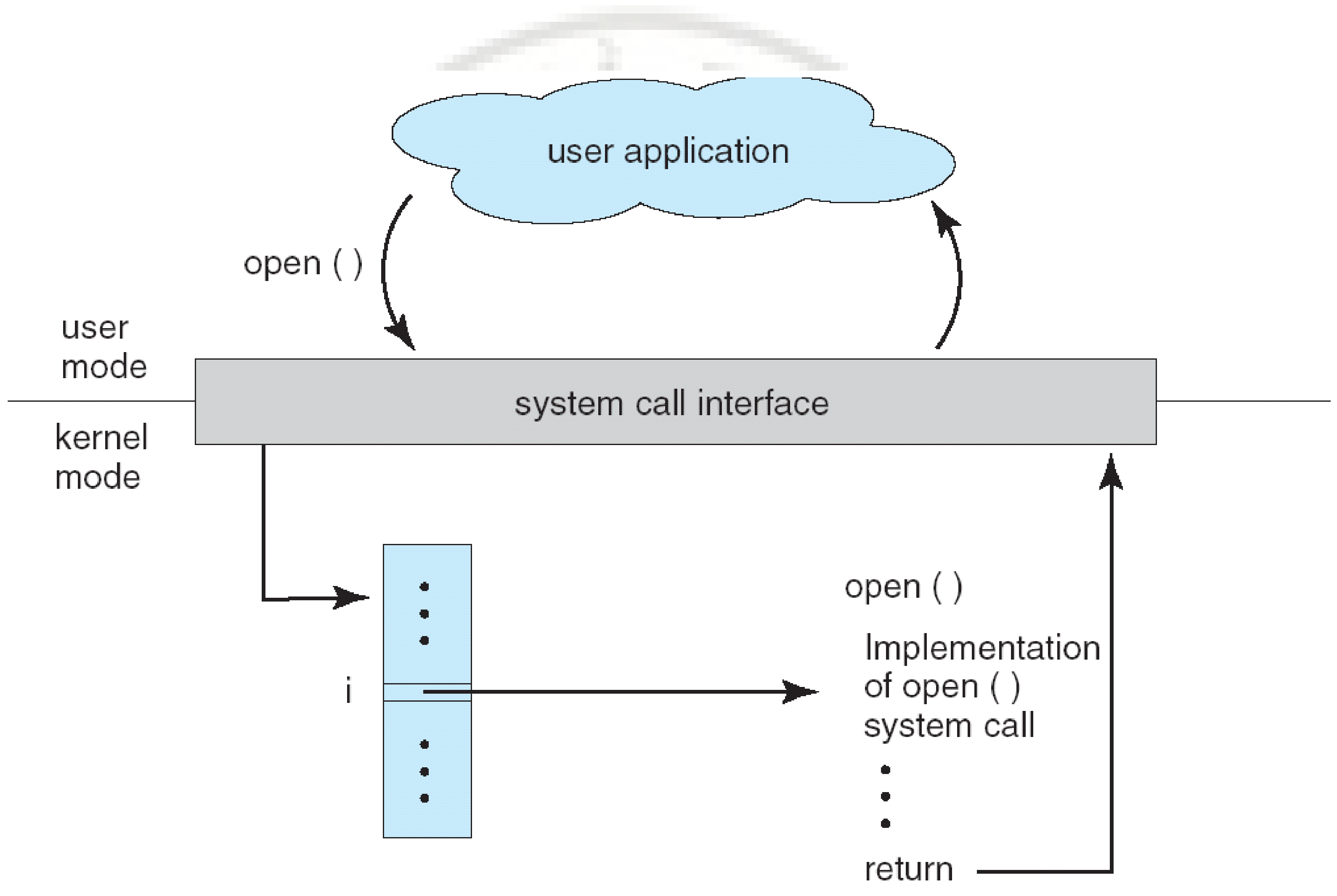
# Application Programming Interface (API)

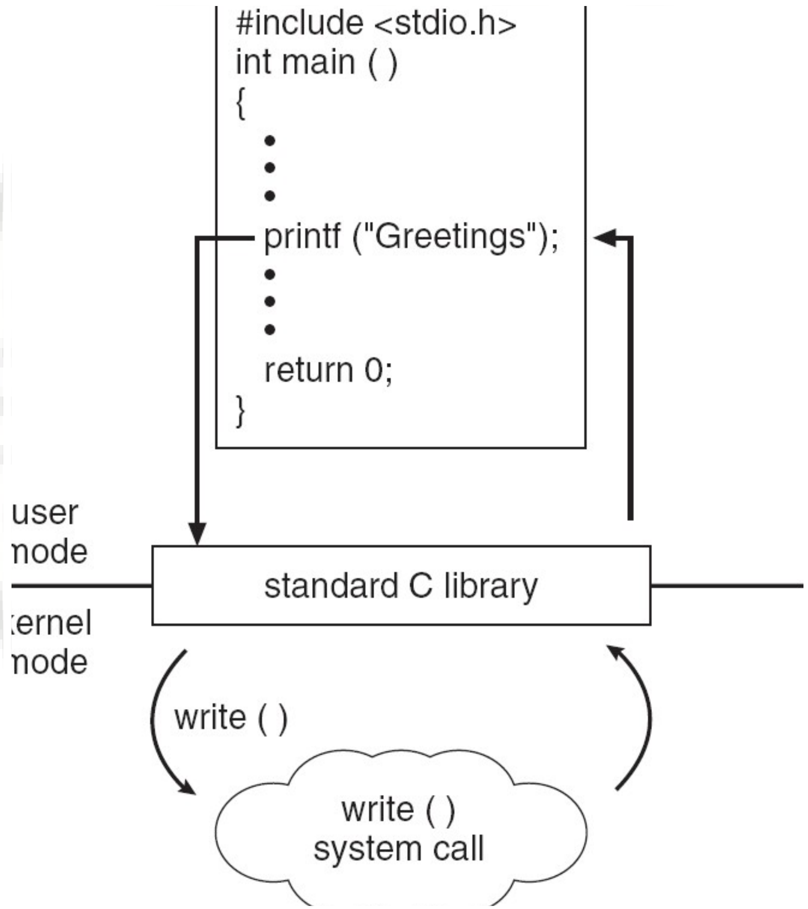
- An API details the set of available functions (services) provided by the OS
- APIs are **abstractions** of the services provided by the OS
- APIs make applications hardware independent
- API examples:
  - API Win32, API POSIX, API JAVA

# The C standard Library

- The **C standard library** has been defined by International Standard Organizazion (ISO)
- It provides and lot of functions
- The API of the libc is specified by the header files.
  - Example
    - `<math.h>`
    - `<stdio.h>`

# Syscalls: the mechanism




#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
user
mode
standard C library
kernel
mode
write ( )
write ( )
system call

# System calls: parameter passing

- There are three ways to pass parameters to syscalls:
  - CPU registers: it is the simplest one, but there should be more parameters than available registers
  - a memory block pointed by a CPU register
  - Stack

# Linux syscalls

1) Syscall number is stored in the **`eax`** register

2) Parameters are stored on the stack.

3) The instruction **`int $=x80`** is executed:

- The interrupt vector entry **`x80`** points to the syscall manager:

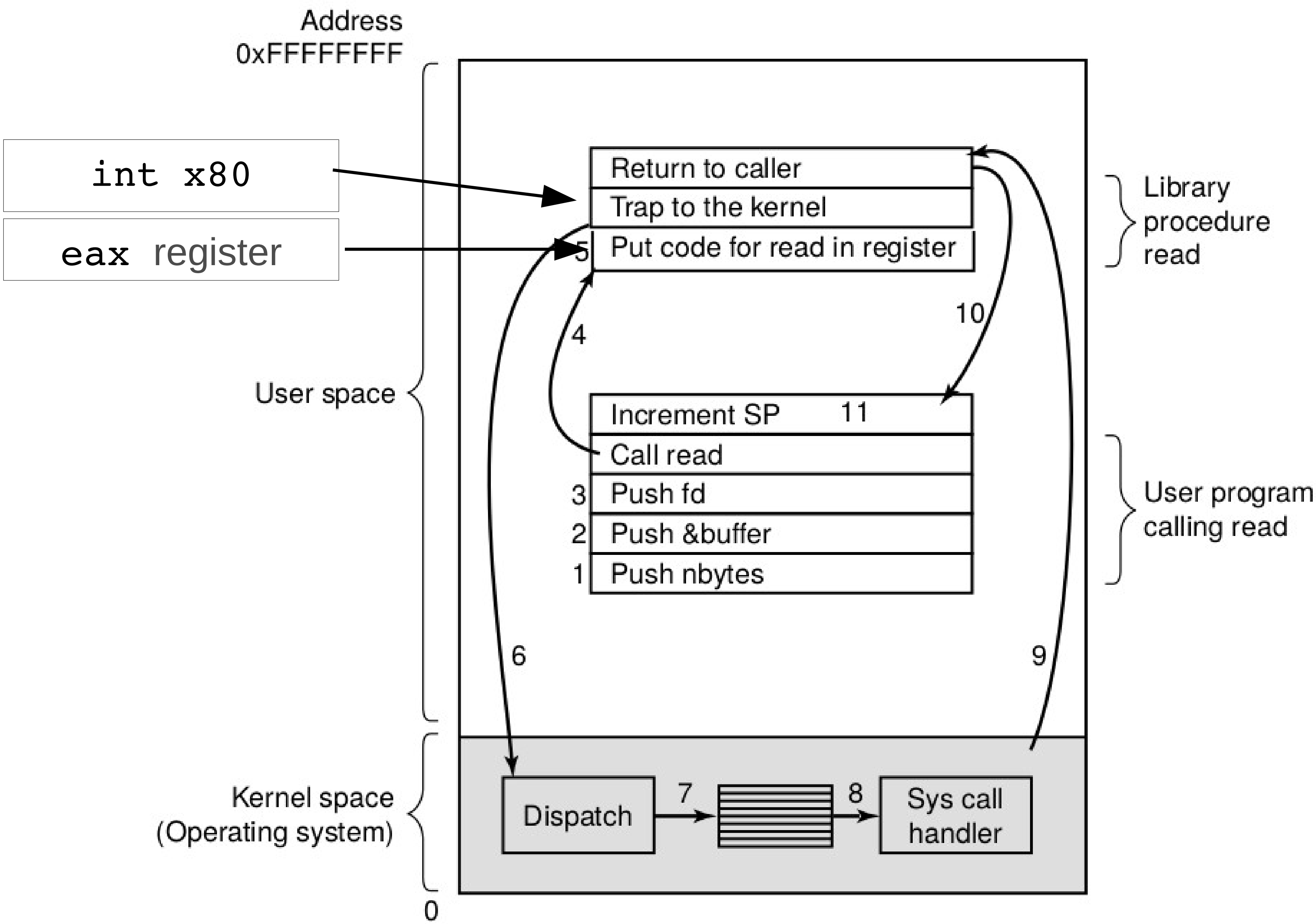
  **`syscall manager'address = IDTR+8*x80`**

4) The syscall manager reads the value contained in the **`eax`** register

# Sytem calls: example

```
count = read(fd, buffer, n)
```

- **`count`**: #bytes actually read
- **`fd`**: file descriptor
- **`buffer`**: where to copy the data (memory address)
- **`n`**: #bytes to be read



int x80
eax register
Address
0xFFFFFFFF
Return to caller
Trap to the kernel
5 Put code for read in register
Library procedure read
10
4
Increment SP 11
Call read
3 Push fd
2 Push &buffer
1 Push nbytes
User space
User program calling read
6
9
Kernel space (Operating system)
Dispatch
7
8
Sys call handler
0

# System call handler

- It is pointed by the entry 128 (**`0x80`** exadecimal) of the  interrupt vector

- Then it carries out the following actions:
  - Saves the CPU registers  onto the stack (macro assembly SAVE_ALL)
  - Calls the OS function that implements the action requested:

```
call *sys_call_table[%eax]
```

  - CPU registers are restored
  - Switch back to user mode

# System call types

- Process management
- File managemet
- File system and directories

# Process management

- `pid = fork()`
  – Creates a (son) process identical to the father (the caller)
- `pid = waitpid(pid, &statloc, options)`
  – waits the termination of the son process
- `s = execve(name, argv, environment)`
  – executes a program
- `exit(status)`
  – Terminates the current process (the caller)

# Fork call: example

- A simple program for generating a son process:

```
int main()
{
   int pid;
   pid = fork();
   if (pid > 0)
      printf(“father process\n”);
   else if (pid == 0) {
            printf(“son process\n”);
         else printf(“Error!\n”);
}
```

# File management

- `fd = open(file, how, …)`
  – Open a file (read or write)
- `s = close(fd)`
  – Close a file
- `n = read(fd, buffer, nbytes)`
  – reads #bytes from file (fd file descriptor) and copies them to the buffer
- `n = write(fd, buffer, nbytes)`
  – Writes #bytes to file from the buffer
- `position = lseek(fd, offset, whence);`
  – Set the file pointer
- `s=stat(name, &buf)`
  – Status information about a file (name) copied into the buffer

# File management: example

- The following program reads 10 bytes starting from the 50th byte, from a file in the current folder

```
int main()
{
  int fd;
  char buffer[10];
  int read;
  fd = open("test.txt", "r");
  lseek(fd, 50, SEEK_SET);
  if (read(fd, buffer, 10) != 10)
    printf("ERROR reading 10 bytes!!!\n");
}
```

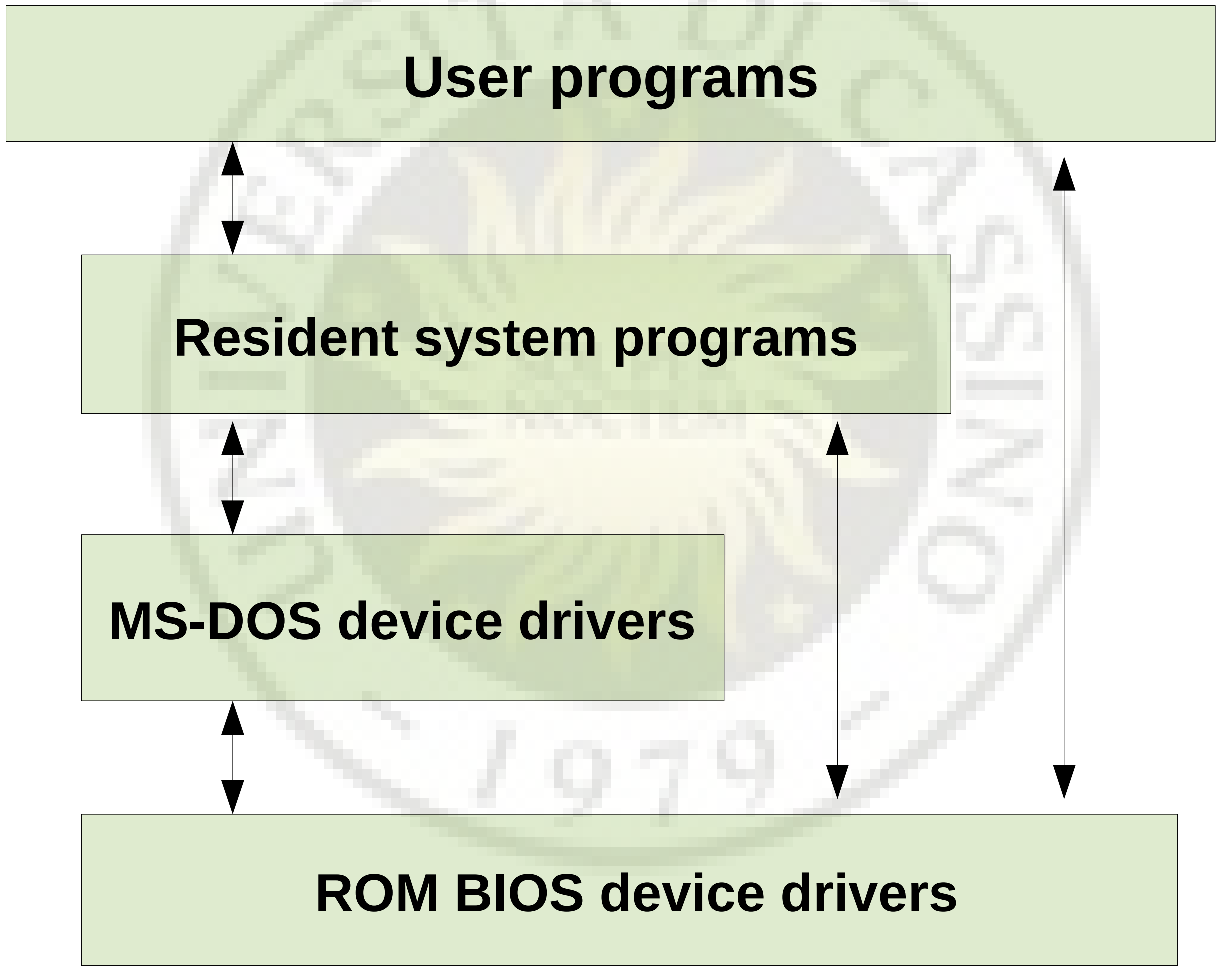
# OS structure

- **OS architecture** describes the OS components and how they are connected
- OS architectures can be very different from each other
- Typical OS components:
  - Process management (scheduler)
  - Memory management (main and secondary)
  - I/O device management
  - file system
  - Etc.

- SO design must consider:
  - –efficiency
  - –maintenance
  - –expandability
  - –Modularity
- Often trade-offs are needed. Example:
  - –Effciency vs modularity

- According to their structure, OS can be divided into two families:
  - –systems with a simple structure
  - –systems with a layer stucture

# Simple structure: MS-DOS



# MS-DOS

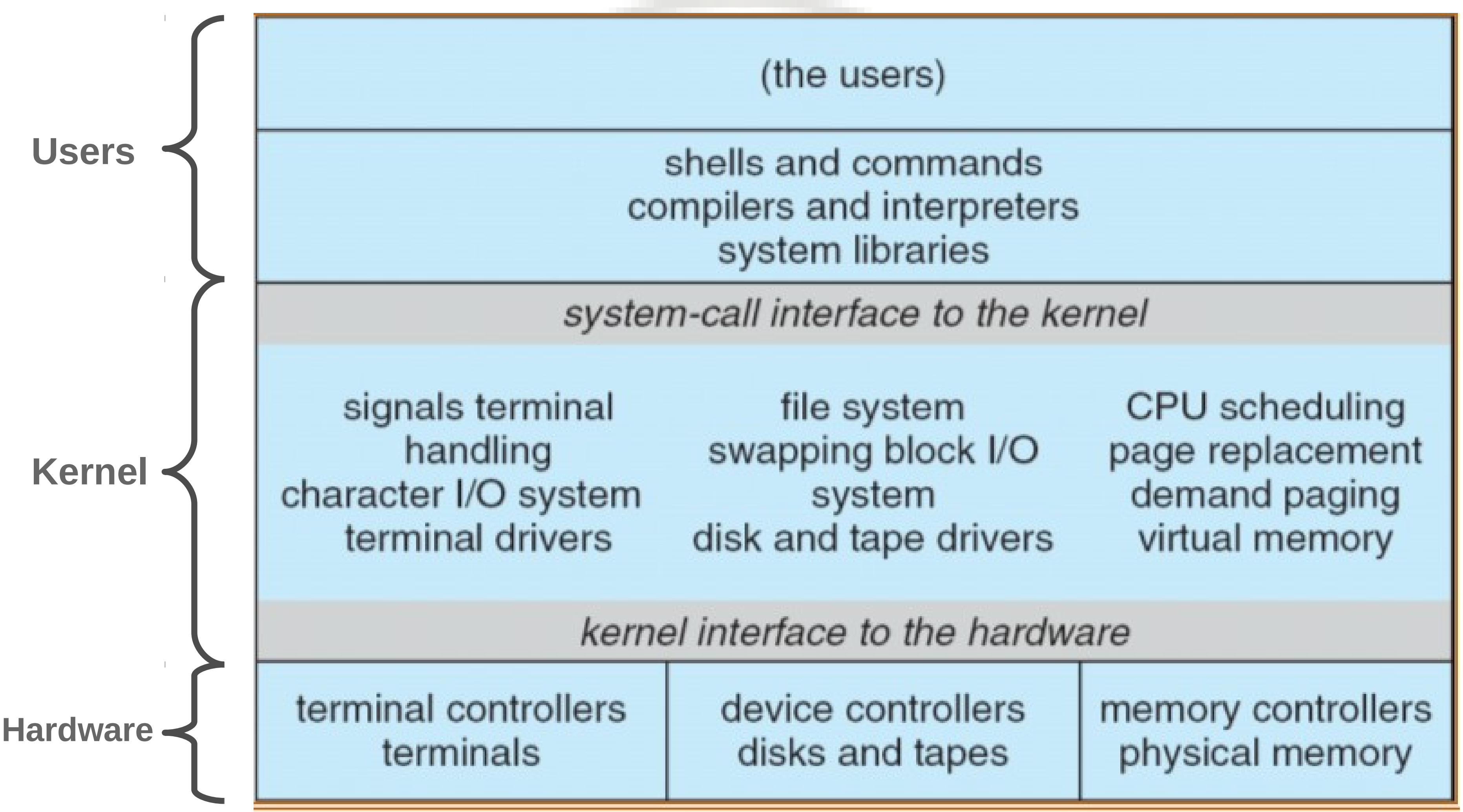
- Comments
  - Interfaces and layers are not well separated
  - Applications can directly access to the I/O devices
  - Security issues: wrong (malicious) programs can crash the system
- Motivations:
  - Designers was limited by the hardware
  - 8086, 8088, 80286 did not have kernel/user mode
  - designer first priority was: best functionality with least possible resources (CPU, RAM and  disk)

# UNIX

- Simple structure
- It is divided into two parts:
  - kernel
  - System programs

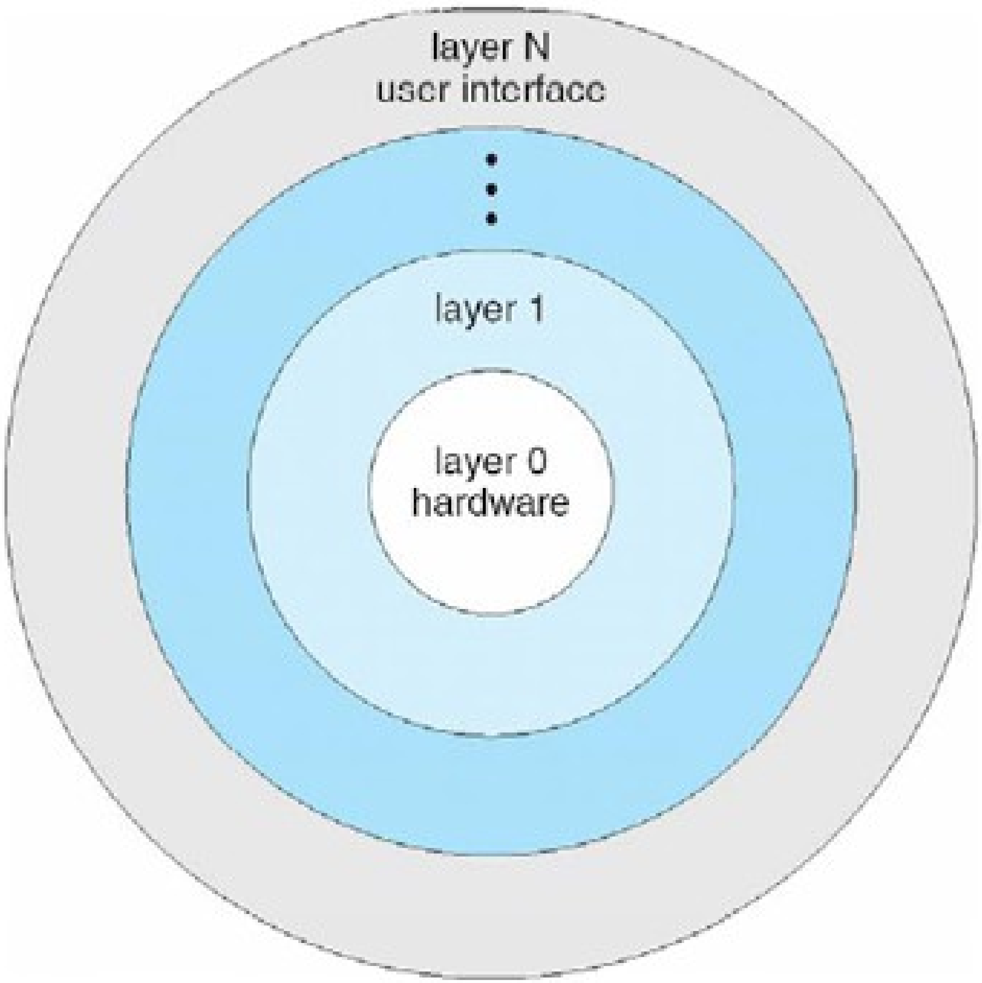
- Motivations
  - Also in this case hardware limitations
  - However with a more structured approach

# UNIX

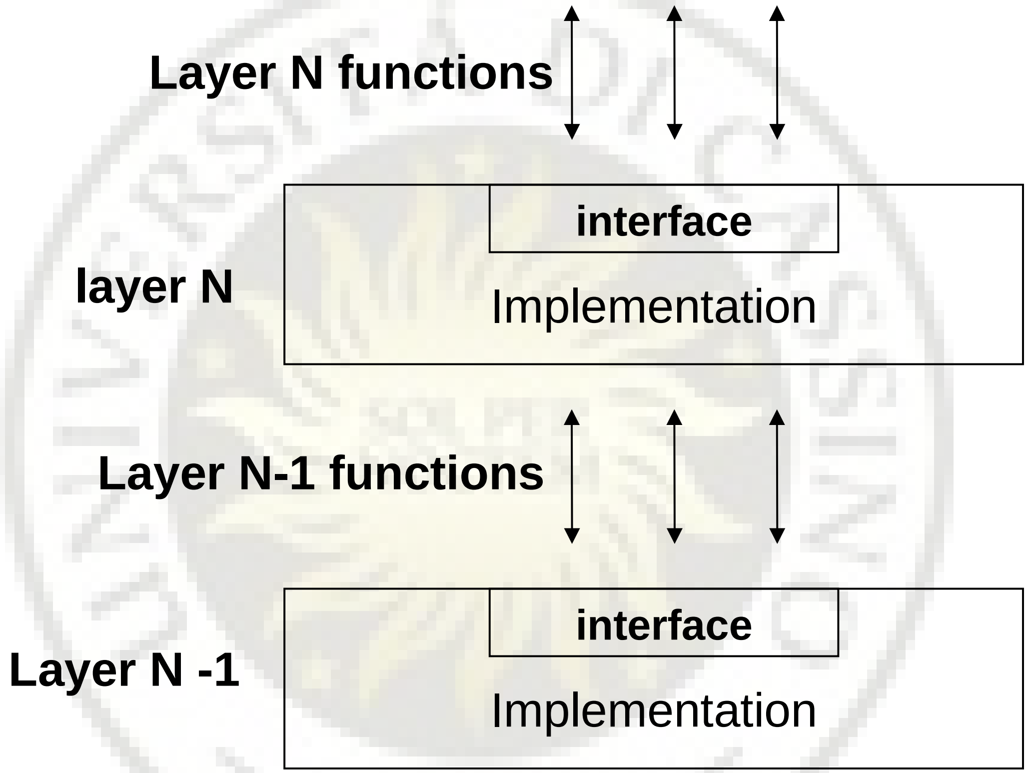

Users
Kernel
Hardware
(the users)
shells and commands
compilers and interpreters
system libraries
system-call interface to the kernel
signals terminal
handling
character I/O system
terminal drivers
file system
swapping block I/O
system
disk and tape drivers
CPU scheduling
page replacement
demand paging
virtual memory
kernel interface to the hardware
terminal controllers
terminals
device controllers
disks and tapes
memory controllers
physical memory

# Layered OS

- The OS is layer structured
- Each layer
  - Uses lower layers
  - offers services to the higher layers
- Motivations
  - the main advantage  is modularity
    - encapsulation and data hiding
    - abstract data types
  - Layer structure simplifies: implementation, debugging, system evolution


layer N
user interface
layer 1
layer 0
hardware

# Layer interaction



# examples

- THE OS (Dijkstra, 1968)
  5) user programs
  4) I/O management
  3) Console device/driver
  2) Memory management
  1) CPU Scheduling
  0) Hardware

- Venus OS (1970)
  6) user program
  5) Scheduler and drivers
  4) virtual Memory
  3) I/O channels
  2) CPU Scheduling
  1) instruction interpreter
  0) Hardware

- **drawbacks**
  - less efficient
    - Each Layer adds overhead
  - Layers must be studied carefully
    - Functions at layer N must be implemented using only the services offered by lower layers
    - This constraint, sometimesm can be hard to overcome

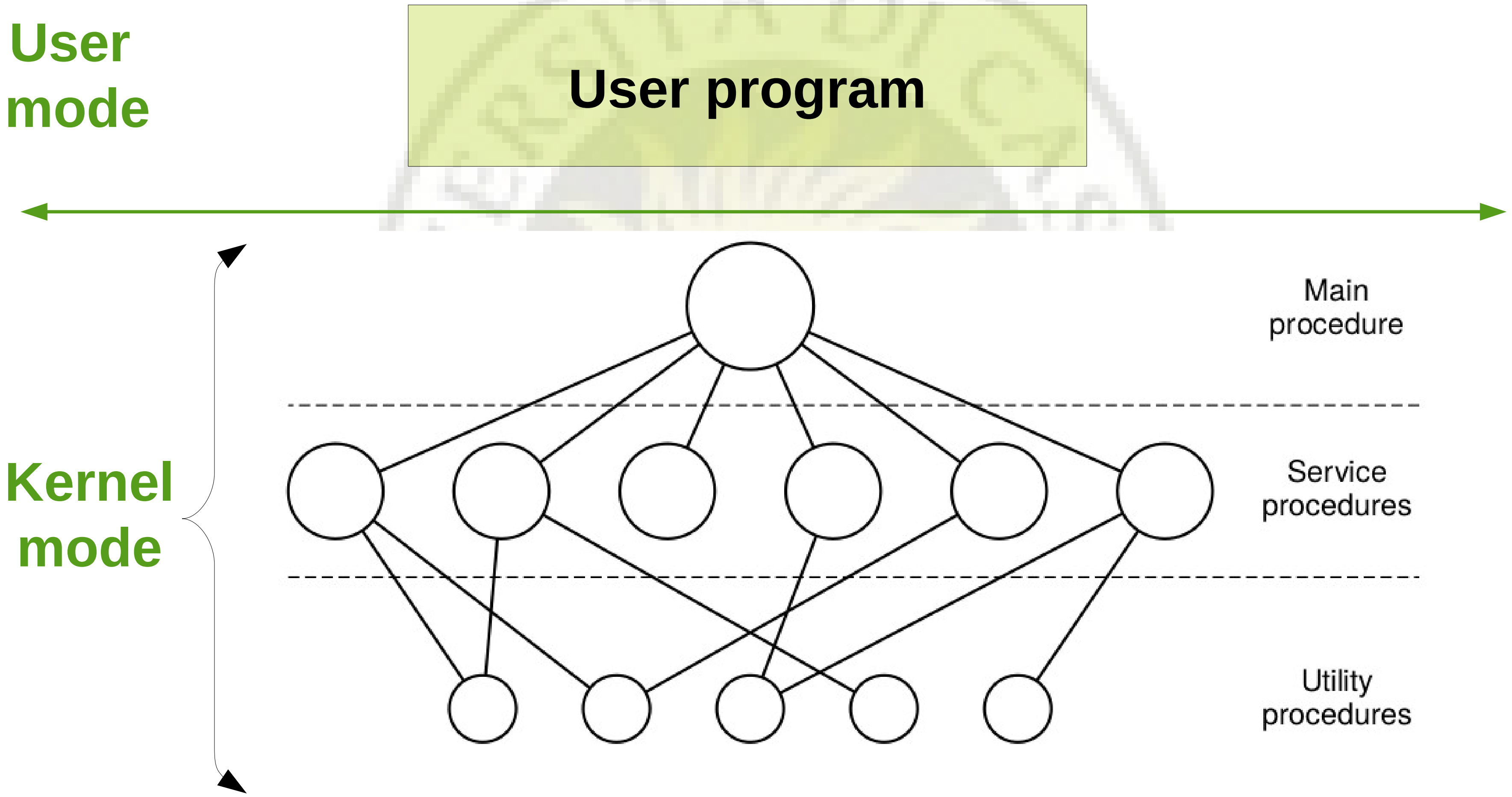
- **Result**
  - Modern SO have few (or none) layers

# Kernel organization

- three  categories
  - **Monolithic**
    - A single (and reach) aggregate of procedures, mutually coordinate
  - **Micro kernel**
    - Minimum kernel which provides process management (scheduler) and message passing
    - client/server paradigm
  - **Hybrid**
    - Similar to Micro Kernel, but some components run in kernel space

# Monolithic  kernels

- A set of procedures which makes a **single address space**
- Syscalls are implemented through modules running in kernel mode
- Monolithic kernel are  organized in modules, but these modules are executed in the same space


User mode
User program
Kernel mode
Main procedure
Service procedures
Utility procedures

- **Efficiency**
  - High, because routines are highly coordinated and integrated
- **Modularity**
  - Modern monolithic kernels allow runtime loading
  - Only actually needed modules are in main memory
  - Kernel is easily (and automatically) extensible
- **Examples**
  - LINUX, FreeBSD UNIX

# Linux modules

- Are portions of software that can be added/discarded (at runtime) to the kernel

- **Main advantage**
  – Kernel does not need to be ricompiled

**NOTE**

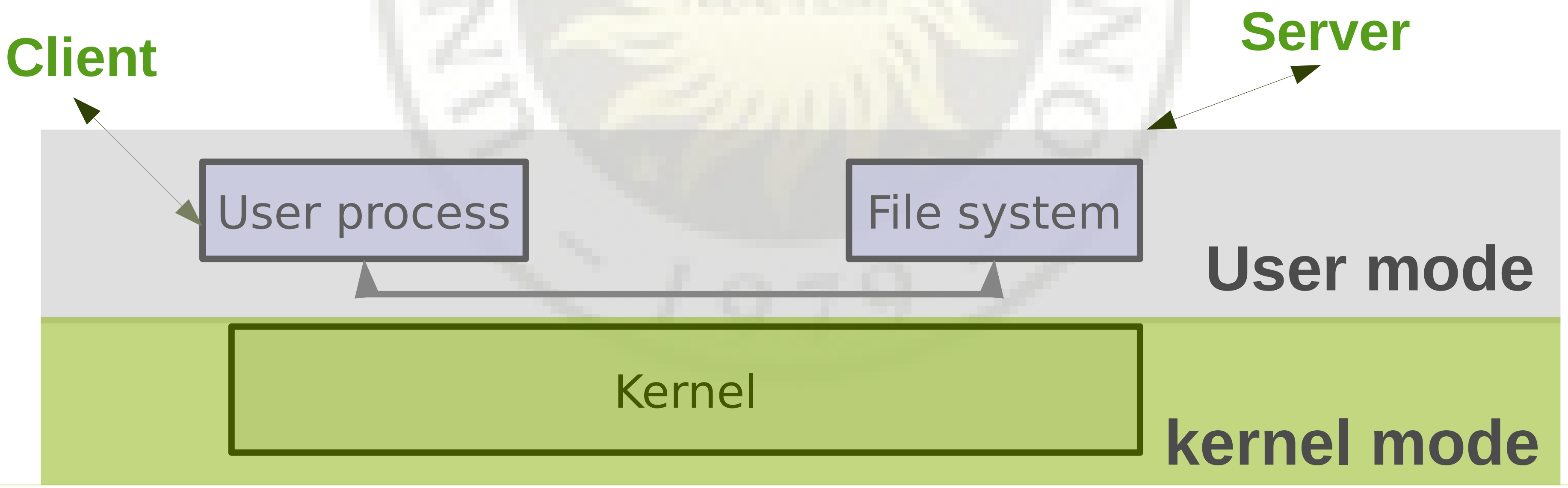
modules are not autonome unities: the kernel is still monolithic!

# Client/server systems

- **Problem**
  – Kernel complexity keeps growing
- **Idea!**
  – Remove from the kernel non essential parts (services) and implements them as user processes
- They implement **client-server** paradigm
- microkernel OS examples:
  – AIX, BeOS, L4, Mach, Minix, MorphOS, QNX, RadiOS, VST

# Microkernel

- Only manages CPU scheduling e memory
- ***message passing***
  - microkernel delivers messages among processes



# Microkernel system calls

- Only two system calls
  - – send
  - – receive
- Through them you can implement the standard API for an OS

```
int open(char* file, ...)
{
    msg = < OPEN, file, ... >;
    send(msg, file-server);
    fd = receive(file-server);

    return fd;
}
```

# Microkernel vantages

- *OS complexity is managed through the client/server paradigm*
- *OS is easily expandable and modifiable*
  - New services are added as user processes (no kernel modifications)
  - To update a given service: source code modification are limited to the service to be updated

- *easy porting on different architectures*
  - Only the (micro)kernel must be modifed,
  - other modules have only to be recompiled
- *High reliability and self-healing (repairing)*
  - If a service (its process) crashes, the OS  can still work
  - The service can be restarted

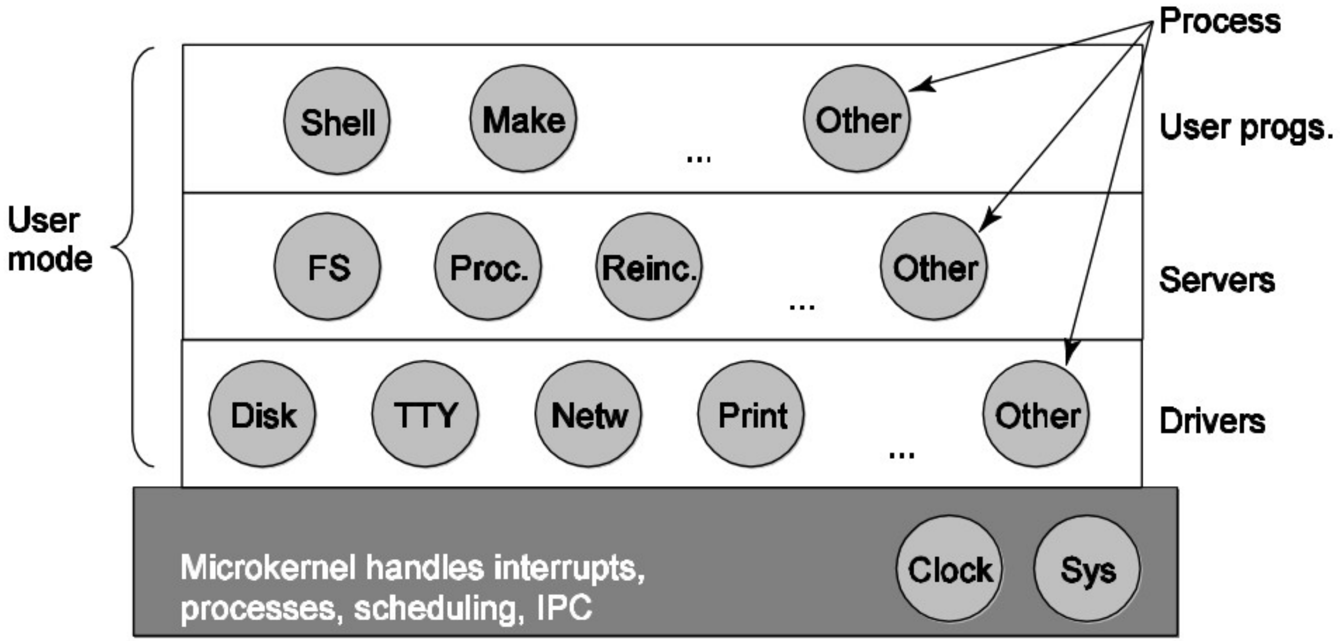
# Microkernel drawbacks

- Low efficiency due to communication overhead
- Instead of simple (and fast) procedure calls (like in monolithic kernel) you must use several (slow) kernel syscalls (send and receive) for process communication

# Minix

- kernel
  - Process manager (scheduler) and (hardware)

- Everything else in user space



# monolithic vs micro

- Monolithic
  - source code in a single address space: less complex to be managed
  - Easier to design

- Micro Kernel
  - It is used in when failures cannot be allowed
  - Ex. QNX OS it is used for arm robot of the Space shuttle

# Hybrid Kernels

- Are essentially micro kernels
- For efficiency reasons,  they retain some services in “kernel space “
- Use message passing for user process communication (like micro kernel)
- Examples
  – Microsoft Windows NT kernel
  – Es. XNU (MAC OS X kernel)

**NOTE**

hybrid kernels should not be confused with monolithic kernels which have runtime loadable modules

# Policies vs mechanisms

- **Policy**
  - What to do (criteria)
- **Mechanism**
  - How to do things (implementation)
- **Good practice**
  - Policies and mechanism must be separated: implementation choices should not influence policies (the choice of the criteria for resource management)
  - It is not an easy task

- microkernels
  - Kernel only implements mechanism
  - Policies are delegated to user space processes
- Example: MINIX
  - Memory manager is a process out of the kernel:
    - It manage memory blocks, but can't directly access to them
    - It can only access its own memory area  (like any other process)
  - it implements its memory management policy through kernel  syscalls (system tasks)

# Booting the system

- During the boot the kernel, or a part of it, is loaded in main memory (RAM)

- During the boot it needs to:
  - Initialize the kernel data stuctures
  - Create at least an user process
  - Give the control to the user process created

- The boot strongly depends on the hardware (we will refer to 80x86)

# The first instant

- Memory is empty!
- Just turned on, a hardware circuit enables the RESET pin of the CPU
- Afterwards, the CPU executes (in real mode) the instruction at the address:

  **`0xfffffff0`**

  which is memory mapped to an EEPROM memory (non volatile memory).
- This memory contains a set of routines called:

  Basic Input/Output System (BIOS)

# The BIOS

- It is a de facto standard
- It was the set of software routines for the I/O management developed for the operating system CP/M ( Intel 8080 and Zilog Z80)
- BIOS instructions are executed in **real mode**

# Real address mode

- It was the operating mode of the (INTEL) CPUs precedent the 286
- It has:
  - A 20 bit address space (1 MB)
  - Direct access to all the address space and all the peripherical devices
- It was defined to allow backward compatibility (before the 286 CPU!)
- Current processors stil have this operation mode (x86-64)

# The boot device

- After hardware initilization, the BIOS searches for the boot device
- Devices are searched by the BIOS according to a (modifiable) given order
- Once the boot device has been found, the BIOS:
  - copies the content of the first sector (boot sector) of this device in RAM memory at the address **`0x00007c00`**
  - Executes the code just loaded:

**`jmp 0x00007c00`**