The Linux Kernel

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

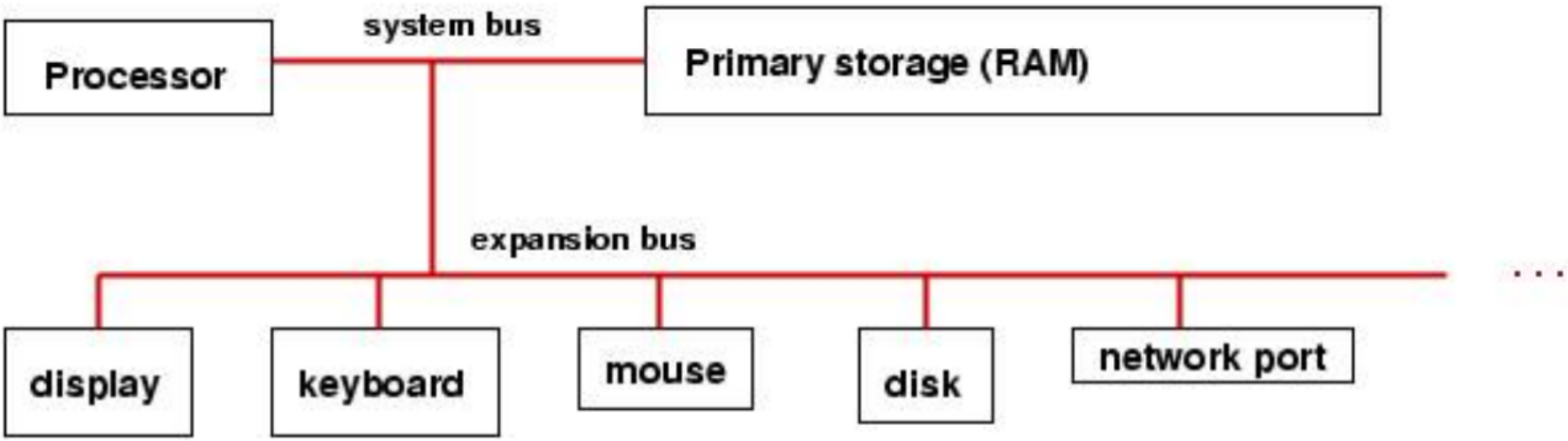
1. Basic of Program Execution
2. Linux Operating System
3. The Linux kernel
4. Interfacing with hardware and software
5. Process Management
6. Memory Management
7. Dealing with time
8. Other Subsystems

Basic of Program Execution

A General-purpose Computer (I)

- **Processor/Central Processing Unit** (CPU).
  - The **brain** that does arithmetic, responds to incoming information, and generates outgoing information.
- Primary storage, **memory** or **Random-Access Memory** (RAM)
  - The **scratchpad** that remembers information that can be used by the processor.
  - It is connected to the processor by a system bus (wiring).
- System and expansion buses
  - The transfer mechanisms (wiring plus connectors) that connect the processor to primary storage and input/output devices.
- Input/output devices
  - For input: a keyboard, a mouse;
  - For output, a display (monitor), a printer;
  - For both input and output: an internal disk drive, memory cards, and etc., as well as network interfaces.

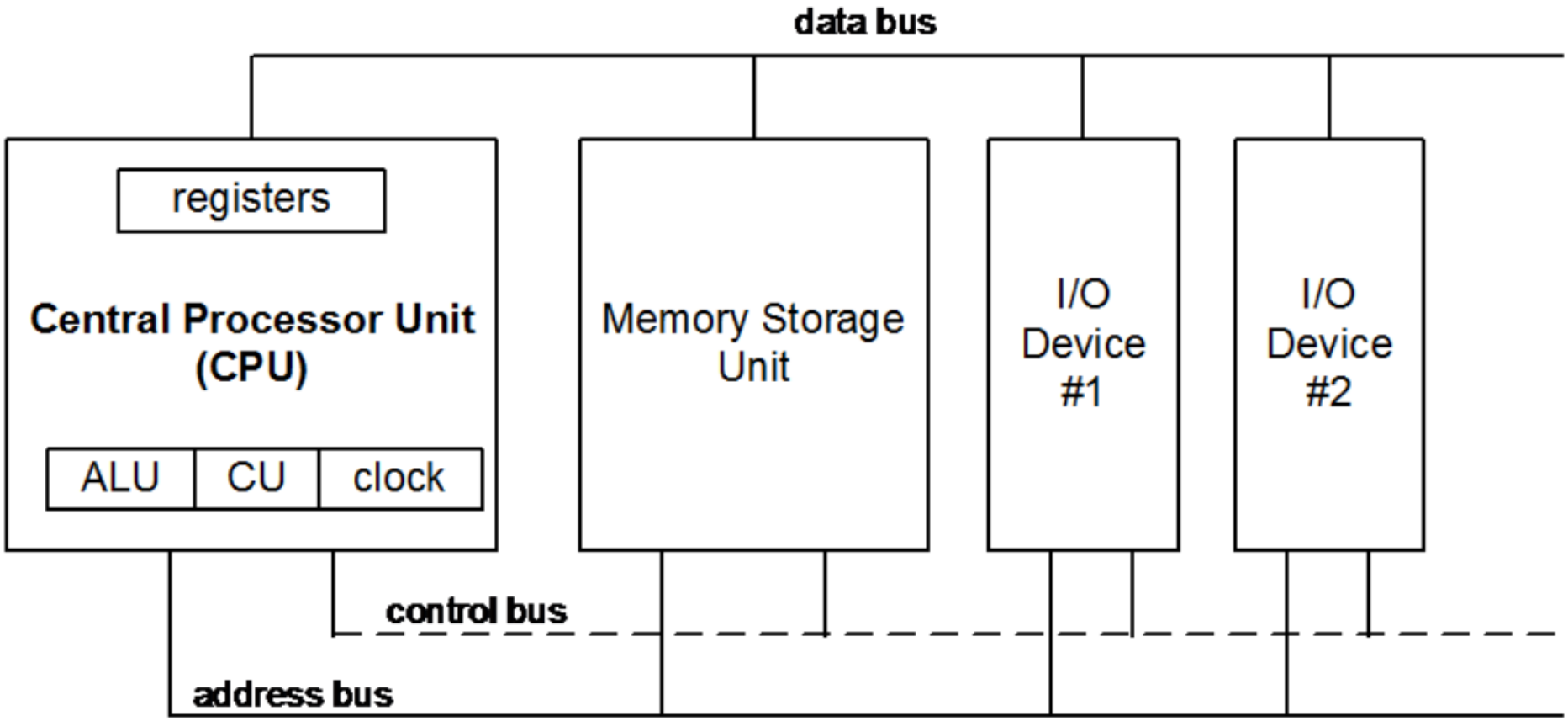
A General-purpose Computer (II)



Basic CPU block (I)

- **Arithmetic Logic Unit** (ALU)
  - Performs **arithmetic computations** (addition, subtraction), and logical operations (AND, OR, NOT).
- **Control Unit** (CU)
  - Interprets **machine language instructions and transmits signals that control the various components as they perform operations necessary to execute instructions**.
- **Clock**
  - **Synchronizes operations** of the CPU with other system components.
- **Registers**
  - The register set of the CPU **provides temporary storage of numeric data within the CPU**.
  - Registers **act like scratchpads** for the CPU keeping track of instructions, data, memory addresses, and status indicators.

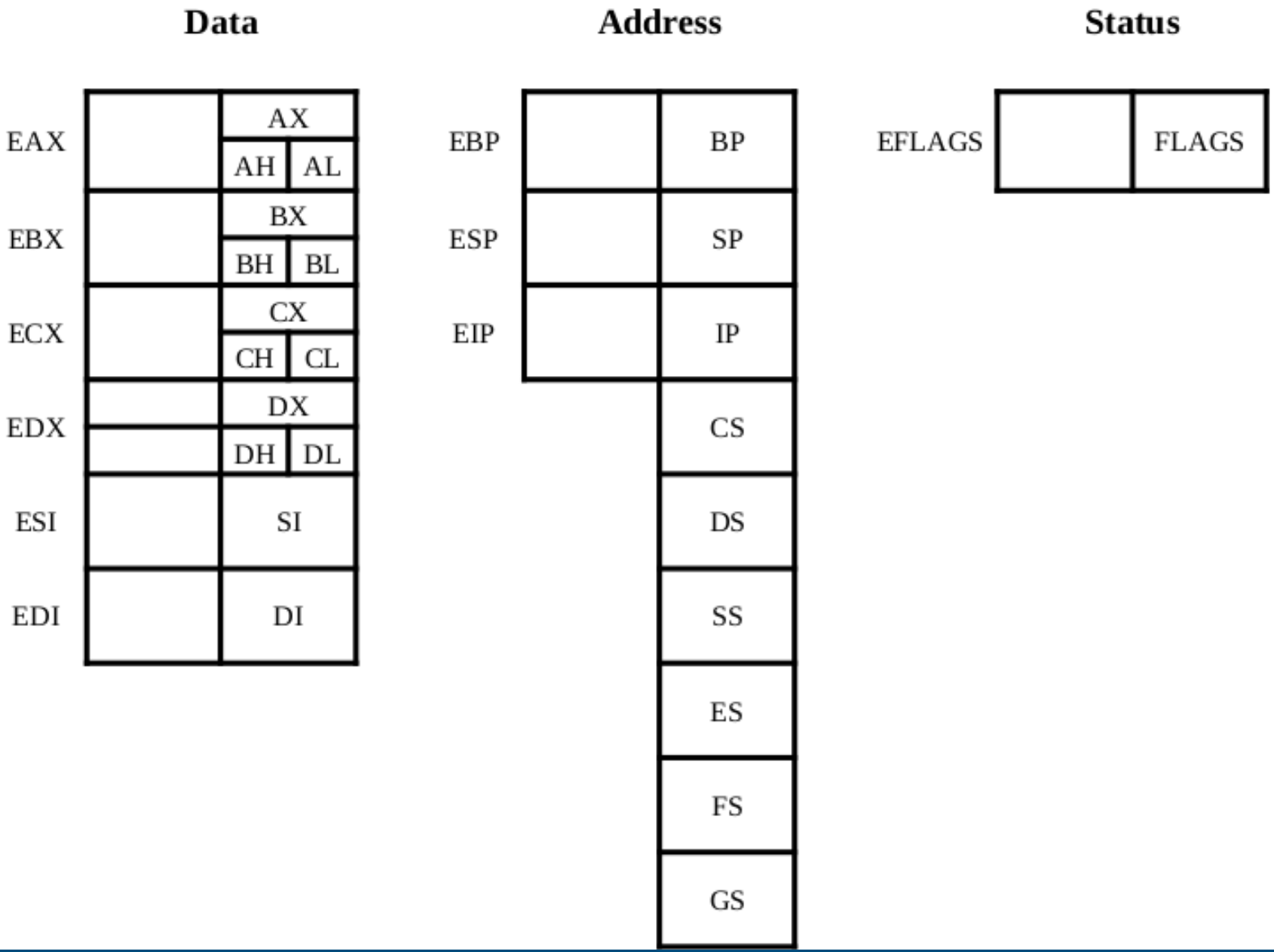
Basic CPU block (II)



CPU Registers (I)

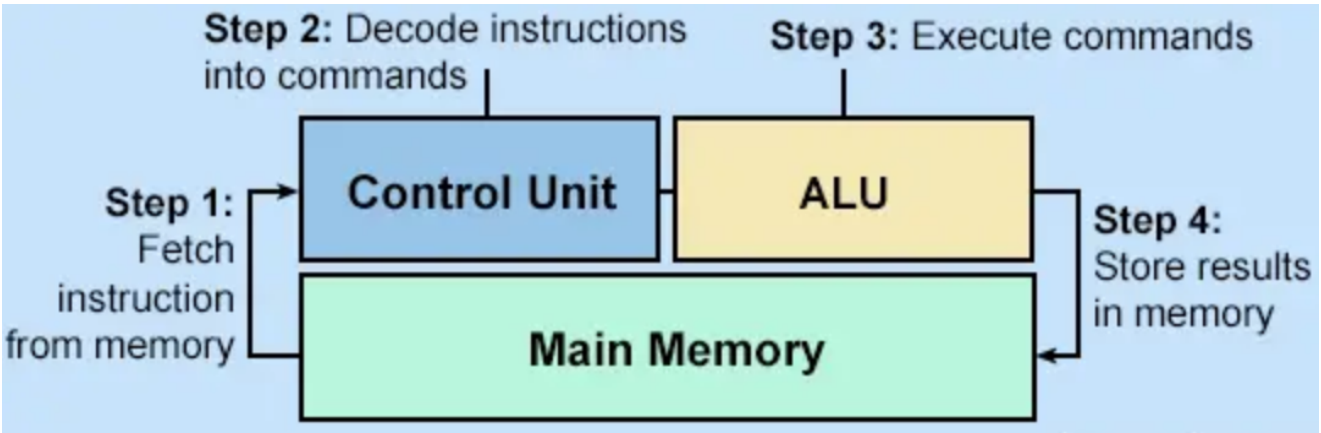
- The **x86 architecture contains fourteen registers**.
- **Data Registers**: Holds data for operations.
  - Accumulator Register (AX), Base Register (BX), Count Register (CX), and Data Register (DX).
- **Address Register**: Holds Address of instruction.
  - Segment Registers
    - Code Segment (CS), Data Segment (DS), Stack Segment (SS), and Extra Segment (ES, FS, GS)
  - Pointer Registers
    - Stack Pointer (SP), Base Pointer (BP), and Instruction Pointer (IP)
  - Index Registers
    - Source Index (SI) and Destination Index (DI)
- **Status Registers**: Store the current status of the processor.
  - Status Flags
  - Control Flags

CPU Registers (II)



Instruction cycle

- The **instruction cycle is the time required by the CPU to execute one single instruction**.
- The **instruction cycle is the basic operation** of the CPU which consists on four steps:
  - **Fetch** the next instruction from memory
  - **Decode** the instruction just fetched
  - **Execute** this instruction as decoded
  - **Store** the result

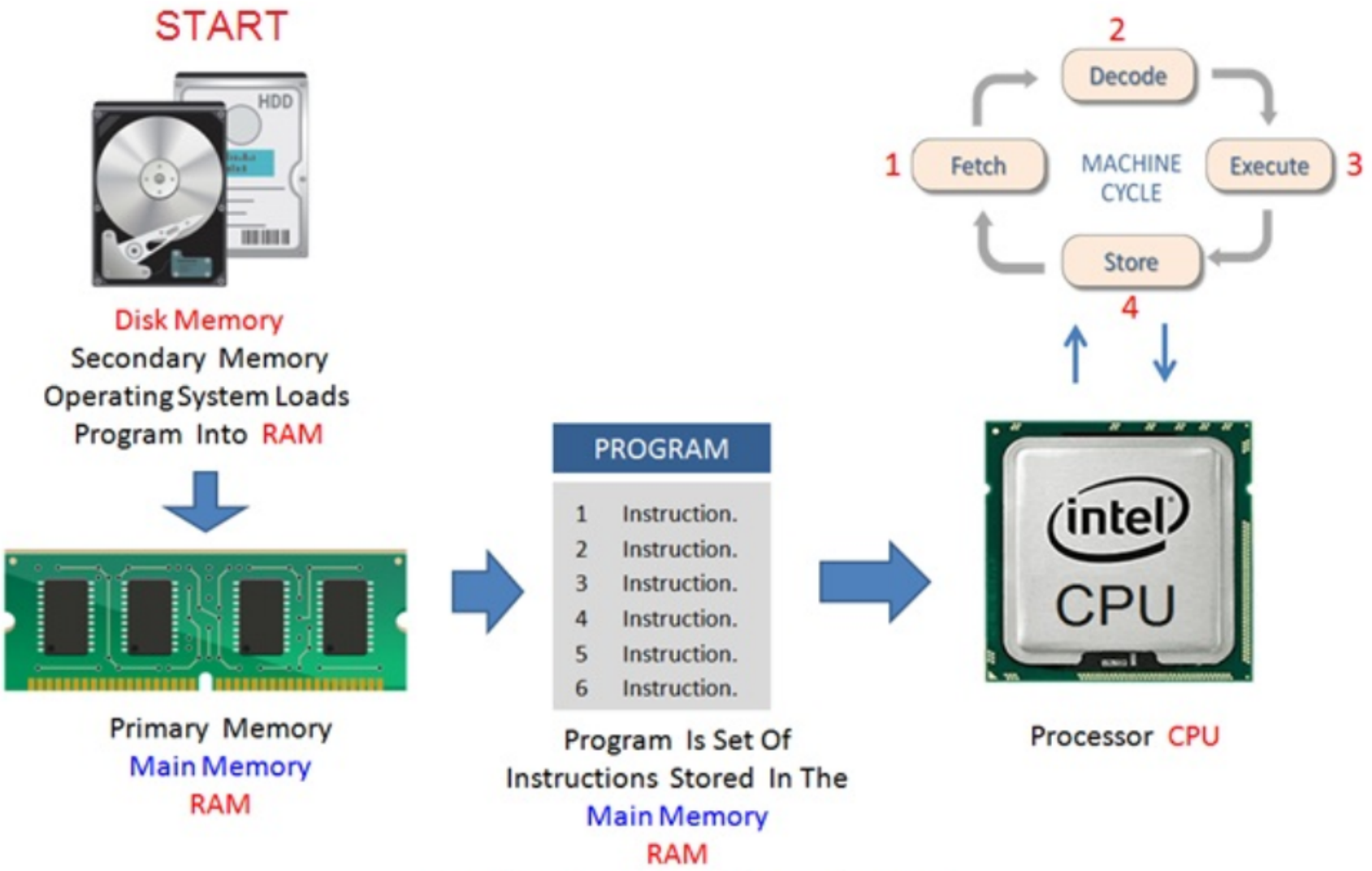


- At the end of each instruction cycle **CPU advances IP register**.

How Computer Executes a Program (I)

- A computer **program is a file**, in which its content is a **set of CPU instructions**.
- The program **instructions are loaded into the main memory** (RAM).
  - The RAM **is organized into number of cells (bytes)**, and **each byte has an address**.
- The CPU **initiates the program execution by fetching the instructions one by one from memory to registers**.
- The CPU executes these instructions **by repetitively performing an instruction cycle**.
- **At the end of each instruction cycle it increments the IP register**

How Computer Executes a Program (II)



Linux Operating System

Linux Was Born

- Linus Torvalds, a young man studying computer science at the university of Helsinki, thought it would be a good idea to have some sort of freely available academic version of UNIX, and promptly started to code.
- He started to ask questions, looking for answers and solutions that would help him get UNIX on his PC.
- Below is one of his first posts in `comp.os.minix`, dating from 1991:

```
From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: Gcc-1.40 and a posix-question
Message-ID: <1991Jul3.100050.9886@klaava.Helsinki.FI>
Date: 3 Jul 91 10:00:50 GMT
Hello netlanders,
Due to a project I'm working on (in minix), I'm interested in the posix
     standard definition. Could somebody please point me to a (preferably)
     machine-readable format of the latest posix rules?
Ftp-sites would be nice.
```

Linux Development Model

- From the start, it was Linus' **goal to have a free system** that was completely compliant with the original UNIX.
- Linux people involved were called **nerds** or **freaks**.
  - Thanks to these people, Linux is now not only ideal to run on new PC's, but is also the system of choice for many devices.
- Two years after Linus' post, there were 12000 Linux users.
- Linux is an **open-source project** (GPLv2 License), for which the development process follows a collaborative approach, with thousands of developers worldwide **constituting the largest collaborative development project in history of computing**.
  - Contributors: companies, academia and independent developers.
- Development cycle takes 3 to 4 months which consists of a 1 or 2 week merge window.
  - Features are only allowed in the merge window
- After the merge window a **release candidate** is done on a weekly basis (rc1, rc2, etc.)

Maintainer Hierarchy

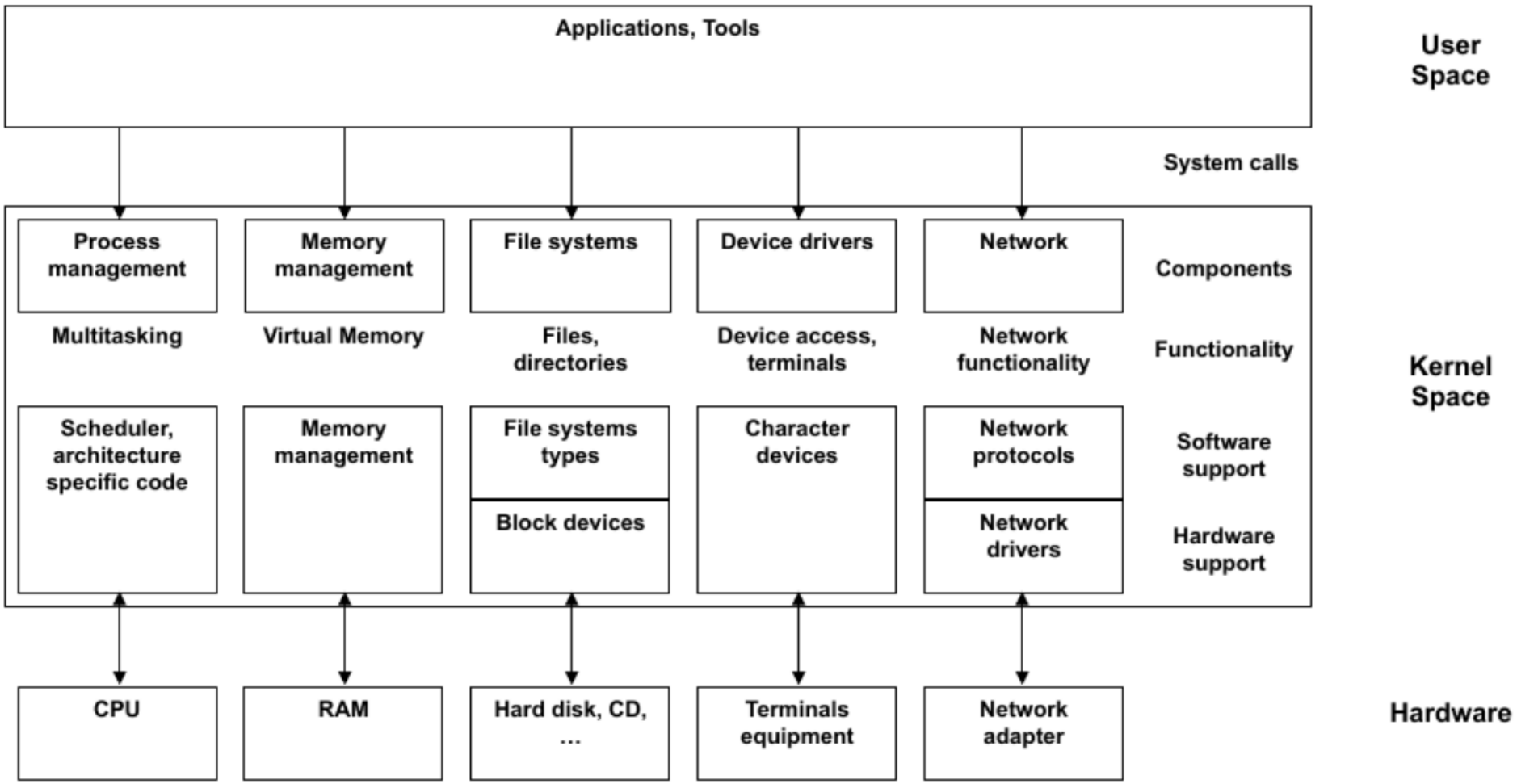
- Linus Torvalds **is the maintainer of the Linux kernel** and merges pull requests from subsystem maintainers
- Each subsystem **has one or more maintainers that accept patches or pull requests from developers or device driver maintainers**.
  - List of maintainers:
    `https://www.kernel.org/doc/linux/MAINTAINERS`
- Each subsystem may maintain a `-next tree` where **developers can submit patches** for the next merge window.

The Linux kernel

Overview

- Linux is a **monolithic kernel**.
  - Executes in a **single address space** entirely in kernel mode.
- However, **borrows much of the good from microkernels**.
  - Boasts a modular design: Linux **supports the dynamic loading of kernel modules**.
    - It can dynamically load and unload kernel code on demand.
- The Linux kernel **is preemptive**.
  - It can preempt a task even as it executes in the kernel;
- Linux takes an interesting approach to thread support: It does not differentiate **between threads and normal processes**.
- Support for kernel threads;
- Execution modes
  - **Kernel mode and User mode**
- Memory protection
  - **Kernel-space and User-space**

Structure (I)


Applications, Tools
User Space
System calls
Process management
Memory management
File systems
Device drivers
Network
Components
Multitasking
Virtual Memory
Files, directories
Device access, terminals
Network functionality
Functionality
Kernel Space
Scheduler, architecture specific code
Memory management
File systems types
Block devices
Character devices
Network protocols
Network drivers
Software support
Hardware support
CPU
RAM
Hard disk, CD, ...
Terminals equipment
Network adapter
Hardware

Structure (II)

- Components **can be enabled or disabled at compile time**.
- It follows a **layered architecture**.
  - User space, Kernel space, and hardware
- Organize the kernel in **logical and independent subsystems** (components).
- Strict interfaces between layers.
  - System calls **for interfacing User and Kernel space**.
  - Device drivers and others (typically, architecture-dependent code) **for interfacing Kernel space and hardware**.

Interfacing with hardware and software

Managing hardware

- How to deal **with different processing/response speeds**?
  - For instance, CPUs are much faster than other hardware components;
  - It **is not ideal for the kernel to issue a request and wait for a response** from the significantly slower hardware;
  - The kernel must **be free to go and handle other work, dealing with the hardware only after that hardware has actually completed its work**.
- How can the CPU work with hardware without impacting the machine's overall performance?
  - **Polling**:
    - **Periodically**, the kernel can check the status of the hardware in the system and respond accordingly.
  - **Interrupt**:
    - A mechanism for the hardware **to signal** to the kernel when attention is needed.

Interrupts

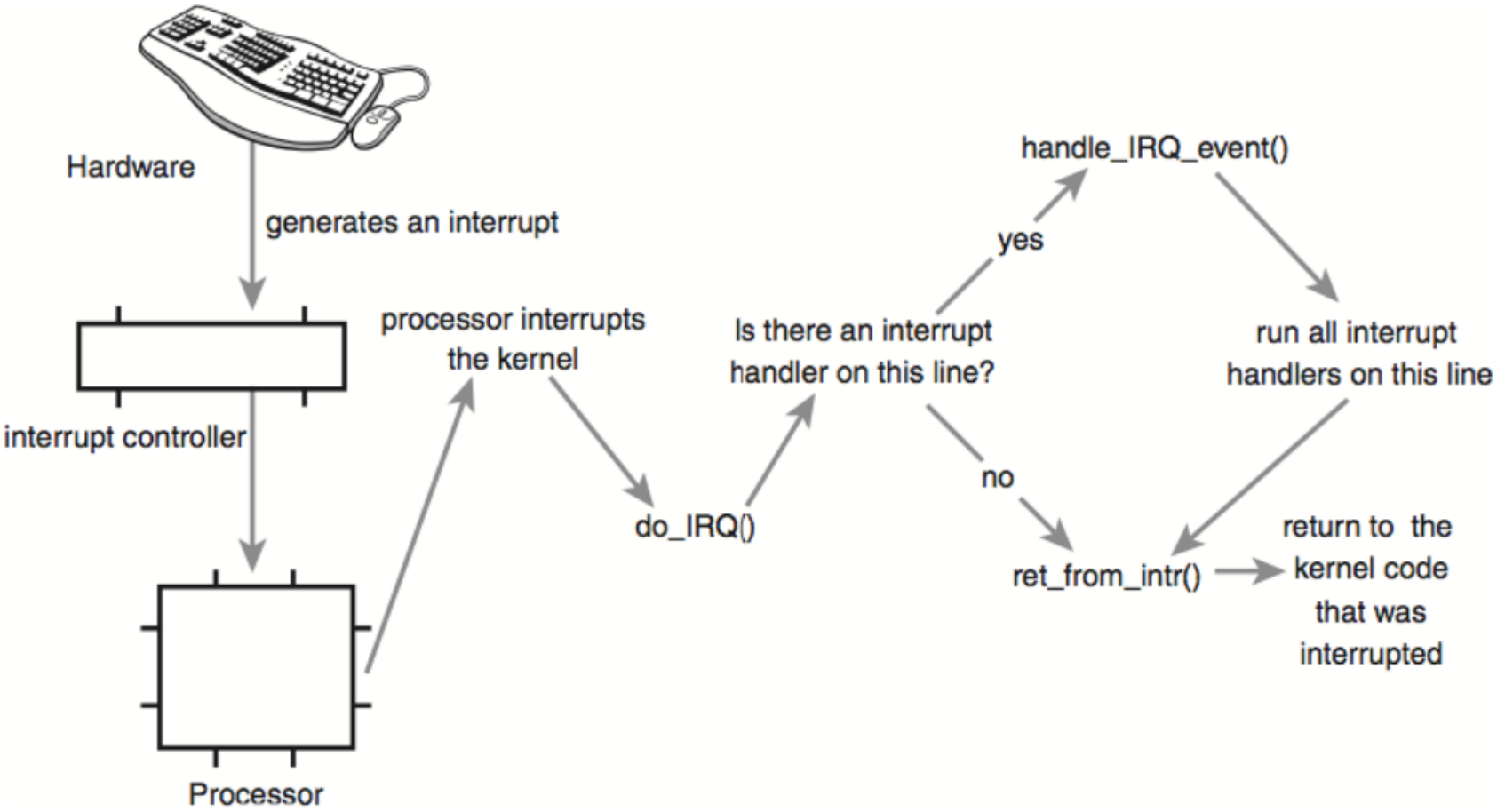
- An interrupt is usually **defined as an event that alters the sequence of instructions executed by a CPU**.
  - This literally interrupts the current process, which in turn interrupts the kernel.
- Interrupts are often divided into synchronous and asynchronous interrupts:
  - Synchronous interrupts **are produced by the CPU** control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction;
  - Asynchronous interrupts **are generated by other hardware devices** at arbitrary times with respect to the CPU clock signals.

Notice

Intel microprocessor manuals designate synchronous and asynchronous interrupts as **exceptions** and **interrupts**, respectively.

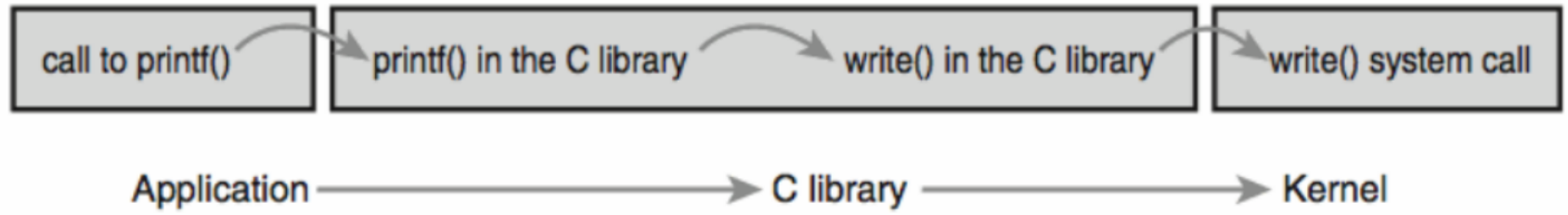
Interrupt handler

- The kernel uses the **interrupt's number** to execute a specific interrupt handler.



System Calls (I)

- Applications typically call functions in libraries.
  - An intermediate layer to standardise and simplify the management of kernel routines across different architectures and systems.
- Libraries, in turn, rely on a **system call interface** to instruct the kernel to carry out tasks on the application's behalf .



- System calls are APIs for the interface between the user space and the kernel space.

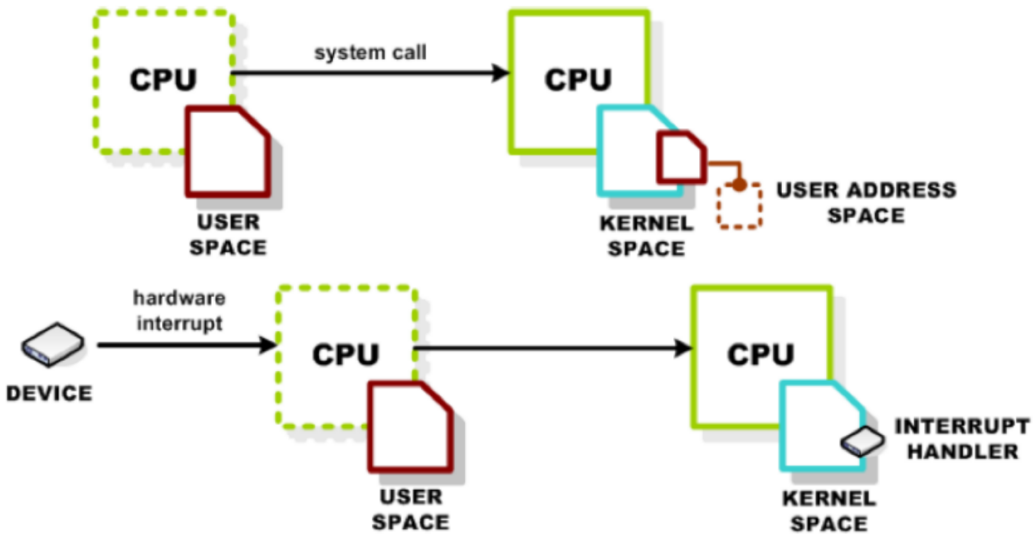
System Calls (II)

- Steps for invoking a system call:
  1. Put the system call number in the EAX register.
  2. Store the arguments to the system call in the registers EBX, ECX, etc.
  3. Call the relevant interrupt (80h).
  4. The result is usually returned in the EAX register.

```
mov edx,4   ; message length
mov ecx,msg   ; message to write
mov ebx,1   ; file descriptor (stdout)
mov eax,4   ; system call number (sys_write)
int 0x80   ; call kernel
```

Kernel space *vs* User space

- Linux is always in one of three states at a given moment:
  - **Process context** - in user-space.
    - When it is executing user code in a process (even when idle).
  - **Process context** - in kernel-space.
    - When it is executing on behalf of a specific process.
    - Kernel code is running in the context of the process that invoked the system call
  - **Interrupt context** - in kernel-space.
    - When it is executing an interrupt handler.



Kernel-space contexts

- In **process context**:
  - The kernel is capable **of sleeping** (for example, if the system call blocks on a call or explicitly calls `schedule()` - a kernel function);
  - The kernel is fully preemptible.
- In **interrupt context**:
  - It is not associated to any process;
  - It is lighter than a process (it has less context and require less time to set up or tear down);
  - This special context is occasionally called **atomic context because code executing in this context is unable to block**.

Process Management

Overview

- It is multiuser because it allows **multiple users to use a system** and not affect each other's "stuff" (files, preferences, etc.).
- It is multitasking – the ability to perform **several operations** in parallel (concurrent).
  - The **switching intervals are so short**, users **do not notice the intervening brief periods of inactivity** and **gain the impression that the system is actually doing several things at once**.
- This kind of process management gives rise to several issues:
  - Processes **must not interfere with each other** unless this is expressly desired.
  - CPU **time must be shared** as fairly as possible between the various applications.

Processes

The Linux kernel data structures used to represent individual processes have connections with nearly every subsystem of the kernel.

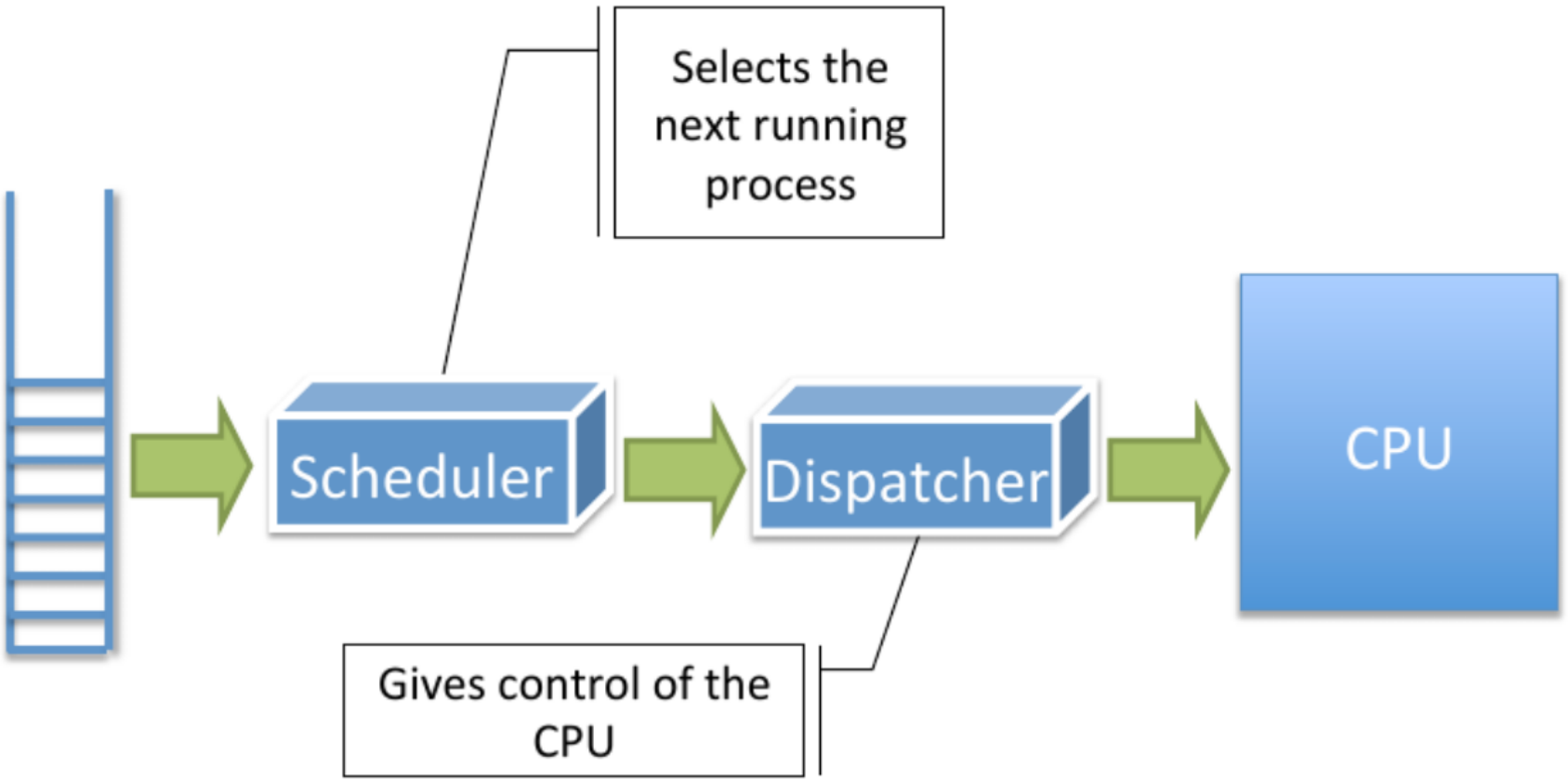
Process

- A process **is an active program and related resources**.
  - From the kernel's point of view, the purpose of a **process is to act as an entity to which system resources (CPU time, memory, etc.) are allocated**.
- It may have one or more threads of execution
  - Each thread includes a unique program counter, process stack, and set of processor registers
- Provides two virtualisations, **giving the illusion that it alone monopolises the system**.
  - **Virtualised processor**.
  - **Virtualised memory**.

Process scheduling

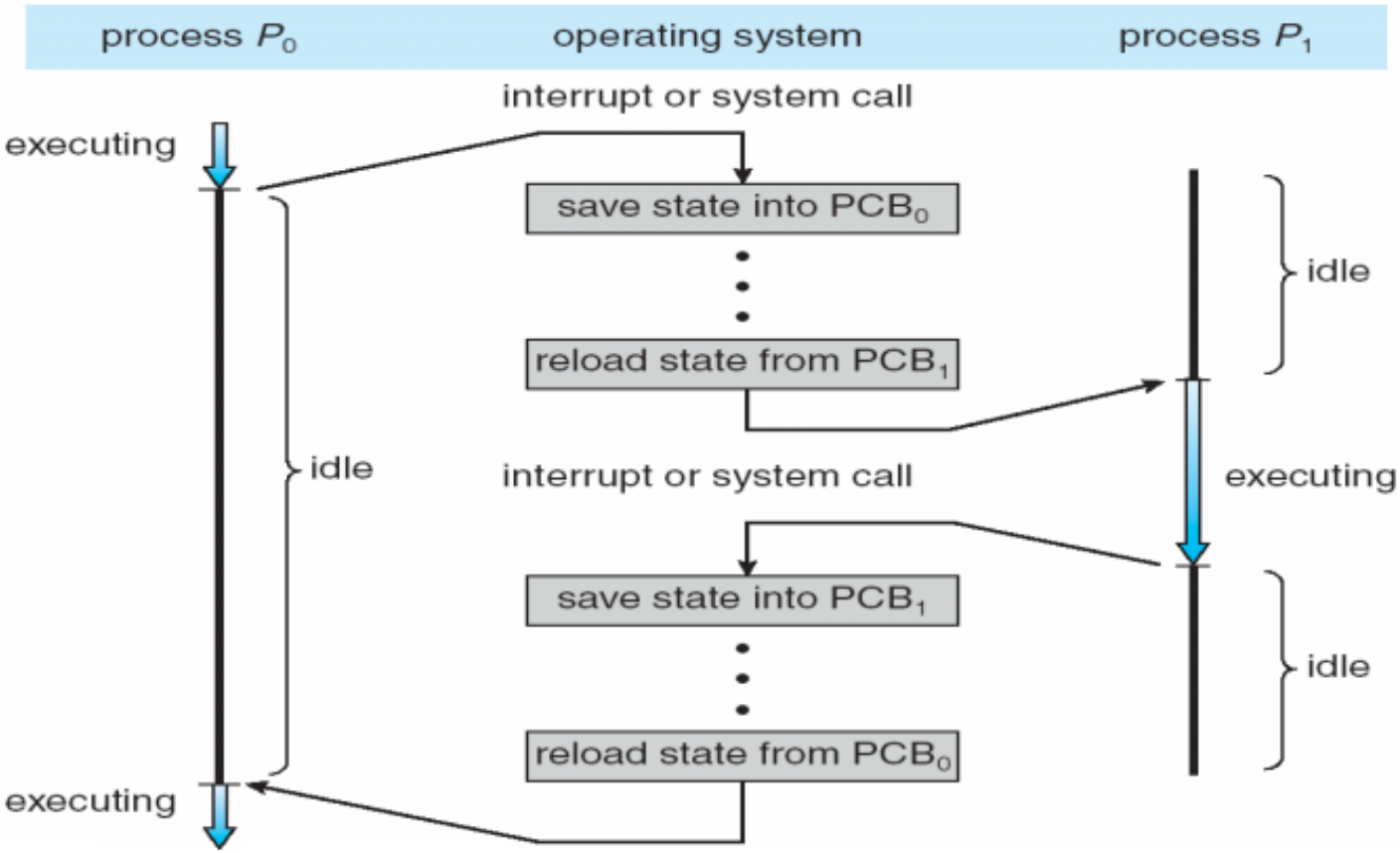
- The process scheduler **decides which process runs, when, and for how long**;
- By deciding which process runs next, the scheduler **is responsible for best utilizing the system**.
  - It divides the finite resource of CPU time between the runnable processes on a system
- It needs **a scheduling policy**, which defines the **scheduling rules**.
- The Dispatcher is the module that gives control of the CPU to the process selected by the scheduler
  - Switching context;
  - Switching to user space.
  - ...

Scheduler and dispatcher



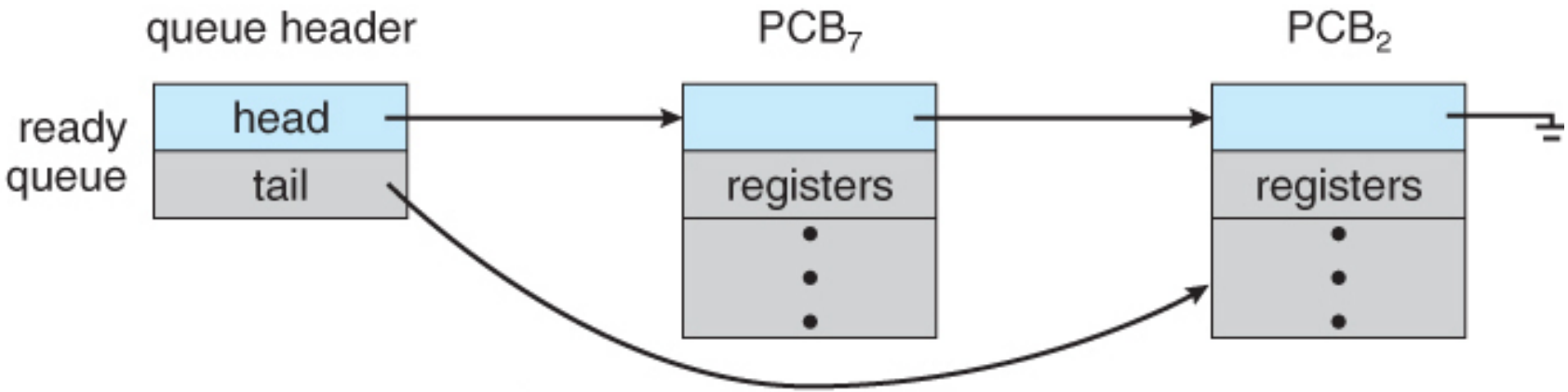
Context switching (I)

- The **act of switching from one process to another**.
- The system has to:
  - Save the **context of the current process**.
  - Restore the **context of the new process**.



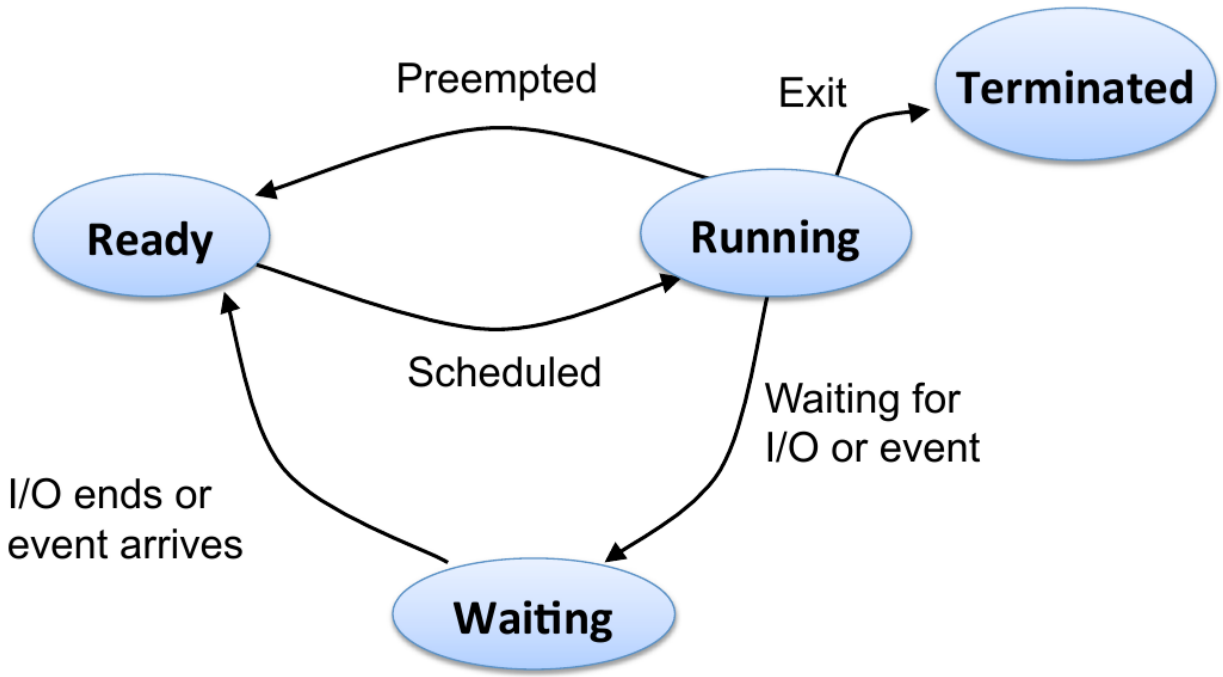
Context switching (II)

- What is the **context of a process**?
  - Program Counter
  - Stack Pointer
  - Registers
  - Code + Data + Stack (also called Address Space)
  - Other state information maintained by the OS for the process (open files, scheduling info, I/O devices being used, etc.)
- All this information is usually stored in a structure called **Process Control Block** (PCB)



Process State

- Multitasking implies process states.
  - Names for these states are not standardised, but they have similar functionality:
    - **Ready**: ready to run, but waiting to be scheduled;
    - **Running**: executing at the moment;
    - **Waiting**: waiting for I/O or an event;
    - **Terminated**: no longer ready to execute.



Scheduling events

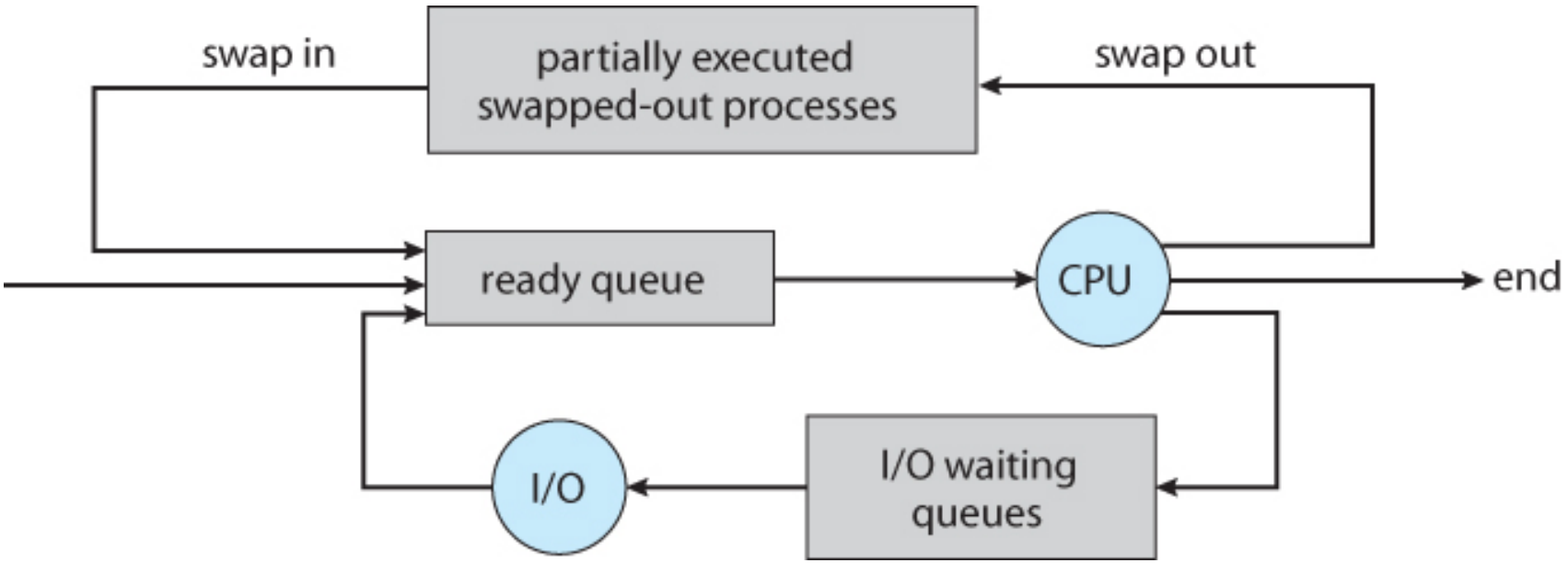
- A process switches **from the running state to waiting state** (e.g. I/O request);
- A process switches **from the running state to the ready state** (e.g. time slice expires);
- A process switches **from waiting state to ready state** (e.g. completion of an I/O operation);
- A process **terminates**

Notice

All executing tasks go from ready to running state. Note that, there is only one task in the running state, per CPU.

Scheduling queues

- **Ready queue**: All processes that are **ready for execution**
- **Wait queues**: When a process **is blocked** in an I/O operation or waiting for an event, it is put in a device/event queue



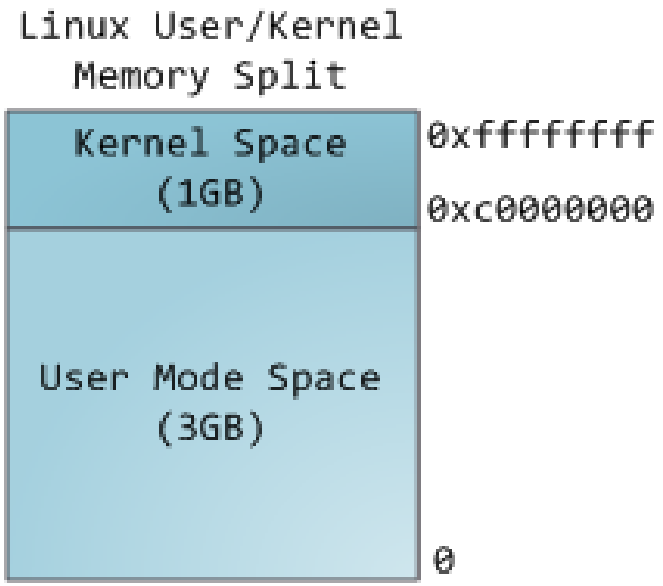
Memory Management

Memory Management

- The memory management subsystem is one of the most important parts of the operating system.
- Since the early days of computing, there has been a need for more memory than exists physically in a system.
- Strategies have been developed to overcome this limitation and the most successful of these is **virtual memory**.
- Virtual memory **makes the system appear to have more memory than it actually has** by sharing it between competing processes as they need it.

Process' Virtual Memory

- Each process in a multi-tasking OS runs in its **own memory sandbox**.
  - This sandbox is the **virtual address space**, which in 32-bit mode is always a 4GB block of memory addresses.
  - In Linux, **kernel space is constantly present and maps the same physical memory in all processes**.
  - **Kernel code and data are always addressable**, ready to handle interrupts or system calls at any time.

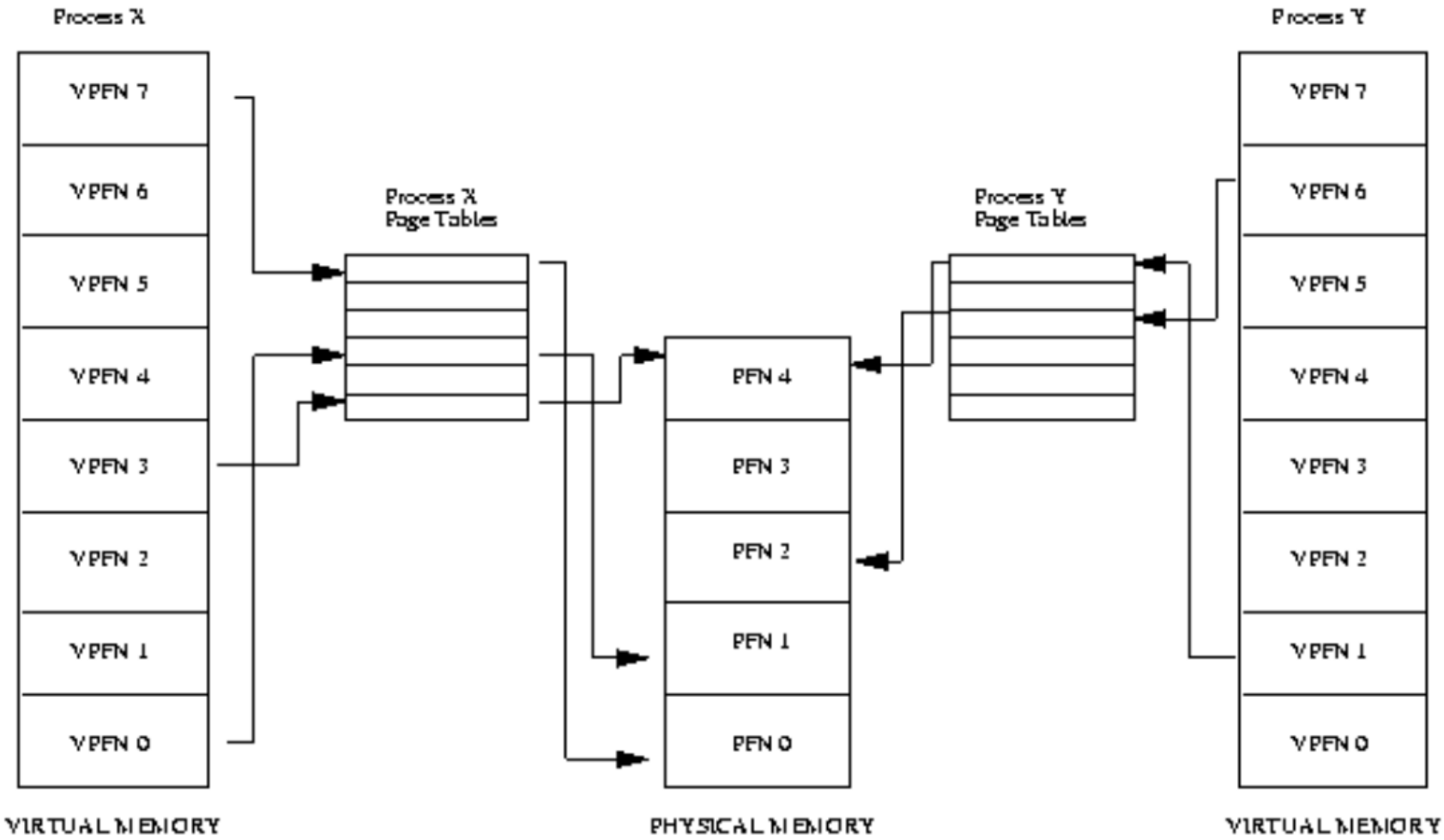


- These **virtual addresses are mapped to physical memory by page tables**, which **are maintained by the operating system kernel** and **consulted by the processor**.
  - Each process has its own set of page tables.

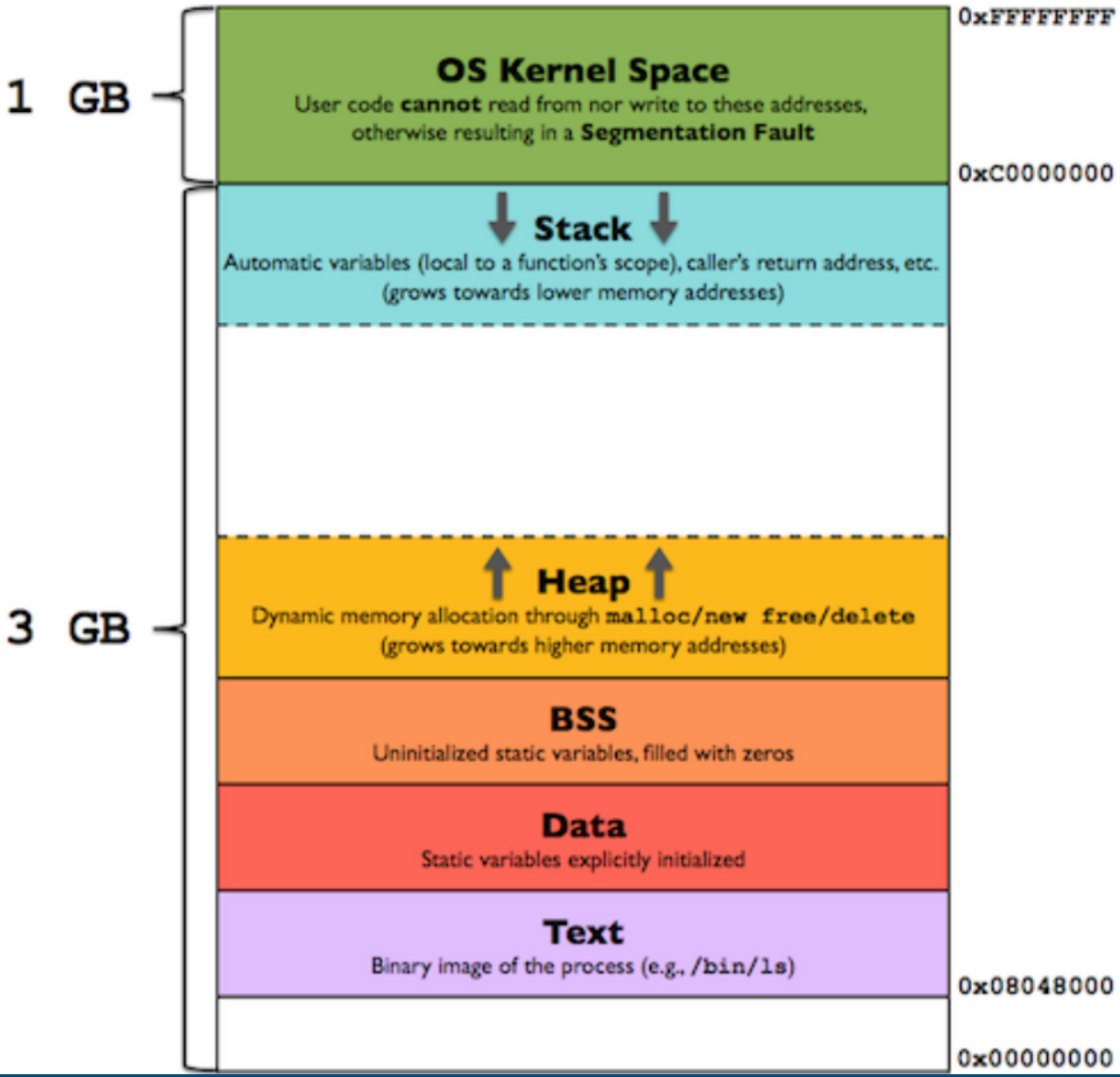
Mapping Virtual to Physical Memory (I)

- **Virtual and physical memory** are divided into chunks called **pages**.
- These pages are all the same size.
- Each of these pages is given a unique number;
  - The physical Page Frame Number (PFN).
  - The Virtual Page Frame Number (VPFN).
- Page Table
  - It is **a data structure used by the virtual memory system to store the mapping between logical addresses and physical addresses**.
  - Each process **owns its Page Table**.

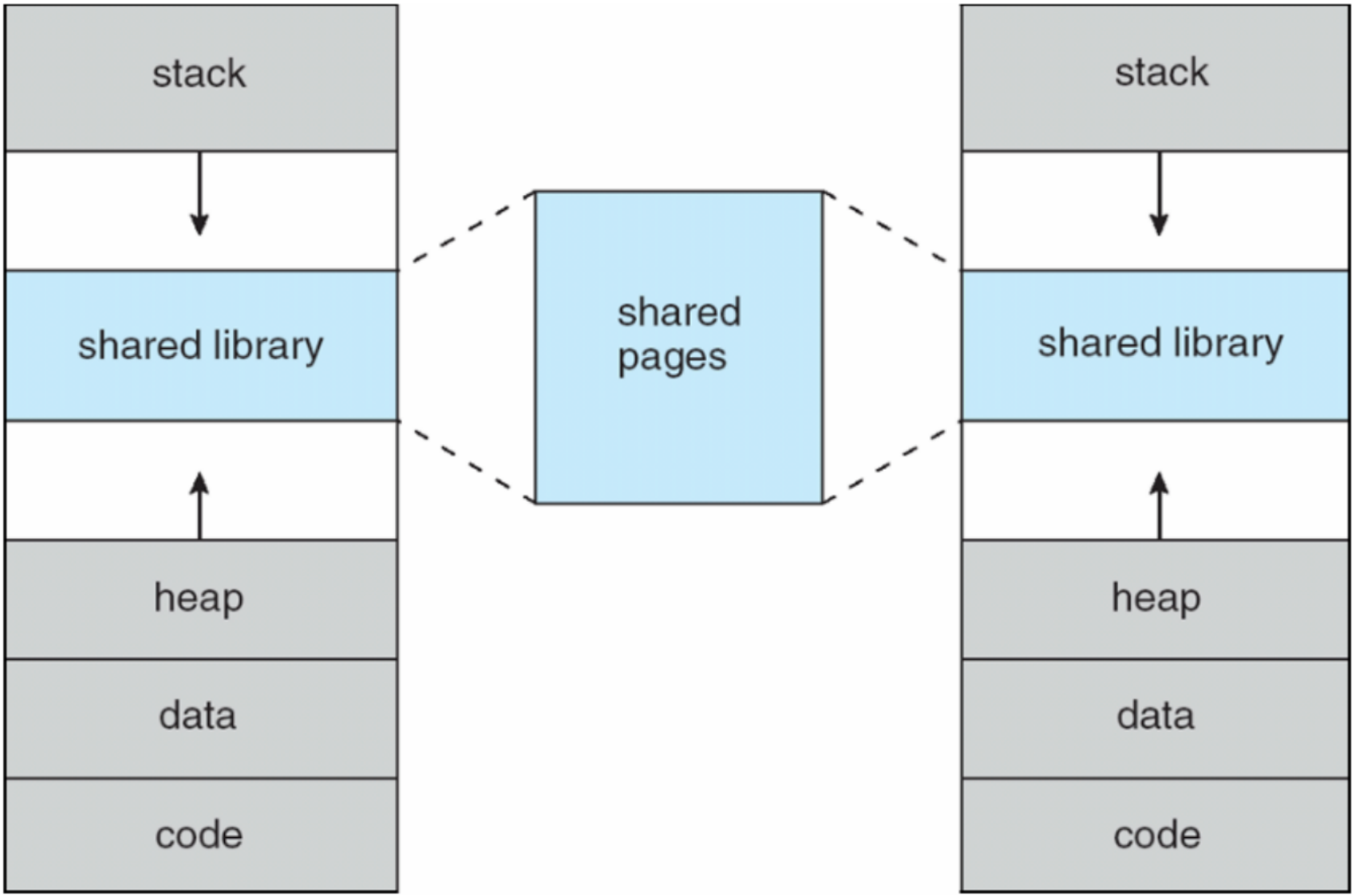
Mapping Virtual to Physical Memory (II)



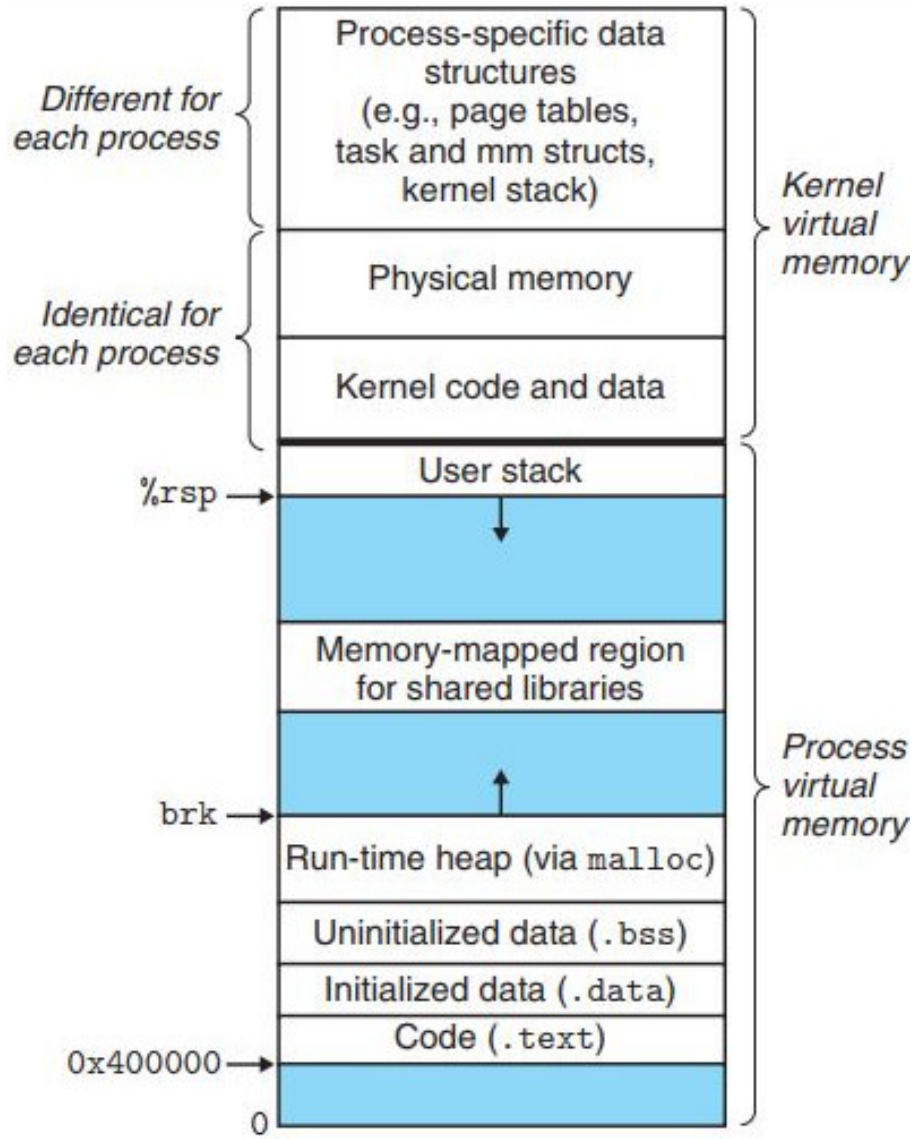
Process Memory Organization



Address Space Shared among Processes

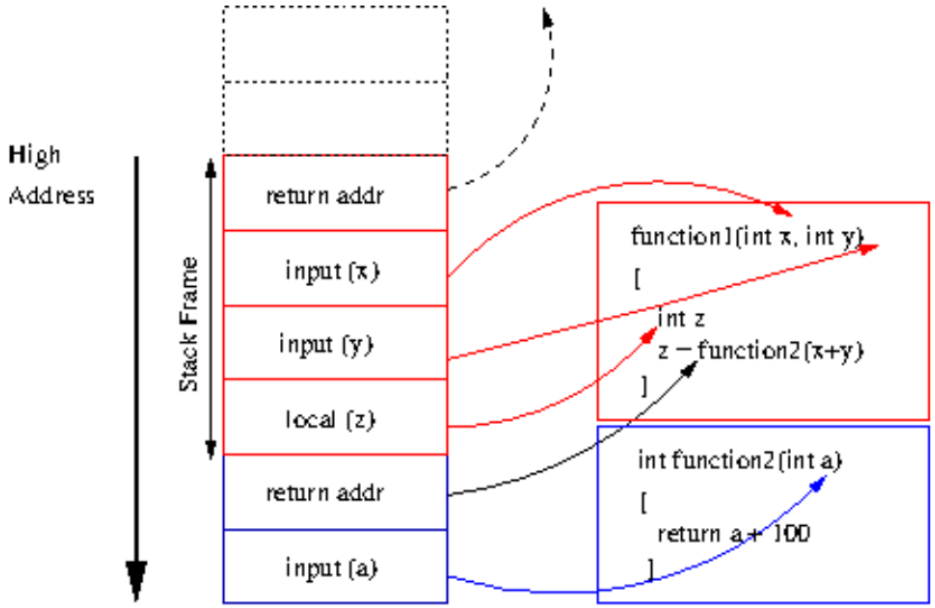


Address Space Shared among Processes



What is a Stack?

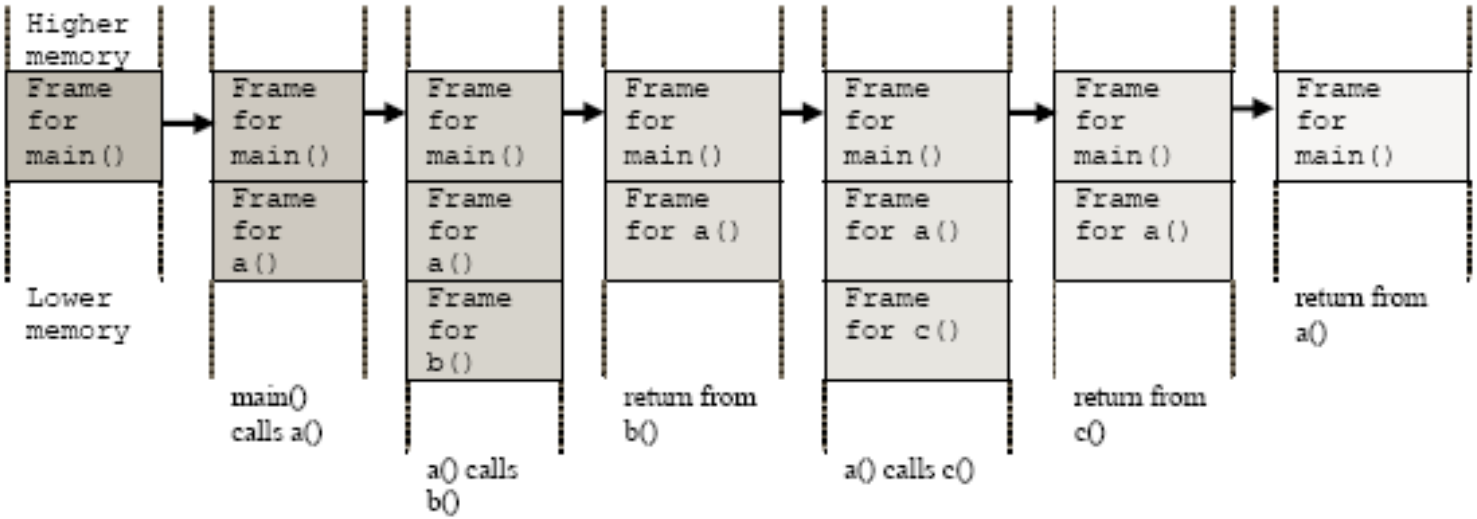
- Stack **is fundamental to function calls**.
  - It **keeps track of the functions running** in a program.
  - Functions are in turn the building blocks of programs.
- Each time a function is called it gets a new **stack frame**.
  - This is an area of memory which usually contains:
    - The **address to return** to when complete;
    - The **input arguments** to the function;
    - **Local variables**.



Stack (I)

```
int c(){
  return 0;
}
int b(){
  return 0;
}
int a(){
  b();
  c();
  return 0;

}
int main(){
  a();
  return 0;
}
```



Stack (II)

```
b:
pushl %ebp
movl %esp,%ebp
movl $0, %eax
movl %ebp,%esp
popl %ebp
ret
a:
pushl %ebp
movl %esp,%ebp
...
call b
...
movl %ebp,%esp
popl %ebp
ret
```

- The **Return address is the address of the next instruction right after `call`**
- `call <label>`
  - Push **Return address** on stack.
  - Changes the EIP register with the address represented by `label`
- `ret` instruction **pops the address from stack and set EIP register with popped value**.
  - It should be the **Return address** previously pushed by `call` instruction.

User stack *vs* Kernel stack

- What's the difference between kernel stack and user stack?
  - **In short, nothing** – apart from using a different location in memory, and usually different memory access protections.
  - When executing in user mode, **kernel memory will not be accessible even if mapped**.
  - Vice versa, without explicitly being requested by the kernel code **user memory is not usually directly accessible**.
- The underlying principles are separation of privileges and security.
- A currently executing process will enter in kernel-mode:
  - To execute **a system call**
  - An **interrupt** happens
  - An **exception** occurs

Dealing with time

Time management

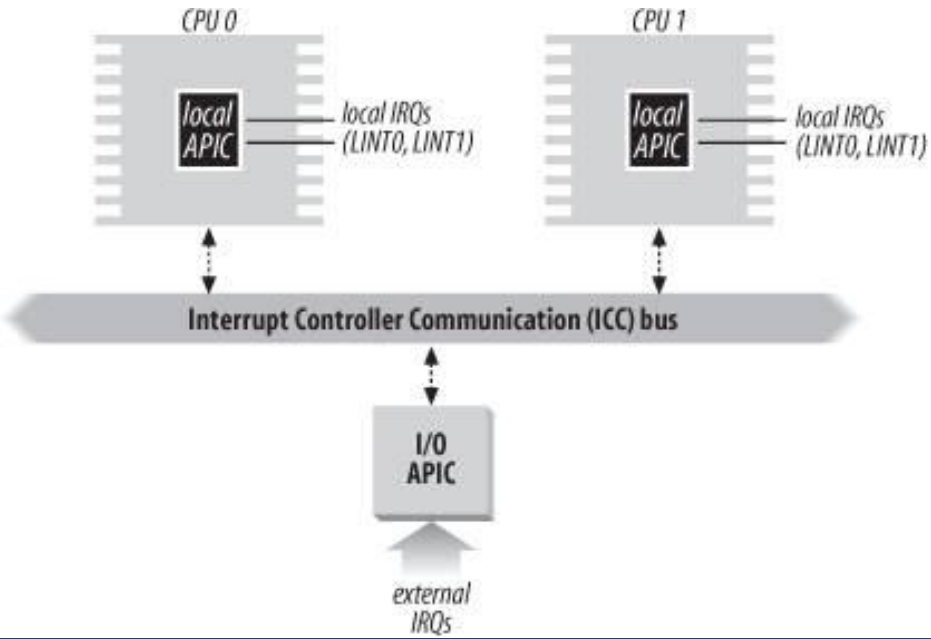
- There are a lot of kernel activities that are **time-driven**, such as balancing the processor's run-queues, process resource usage accounting, profiling, and so on;
- Time management **is based on some hardware devices that have oscillators of a known fixed frequency, counters, and comparators**;
  - Simpler devices **increment their counters** at fixed frequency, which is in some cases configurable.
  - Other devices **decrement their counters (also at fixed frequency) and are able to issue an interrupt when the counter reaches zero**.
  - More sophisticated devices **are provided with some comparison logic to compare the device counter against a specific value, and are also able to interrupt the processor when those values match**.

Time related devices (I)

- **Programmable Interval Timer** (PIT)
  - It is a counter that is able to generate an interrupt when it reaches the programmed count.
- **High Precision Event Timer** (HPET)
  - It consists of a counter and a set of comparators.
  - Each comparator has a match register and can generate an interrupt when the associated match register value matches with the main counter value;
  - The comparators can be put into one-shot mode or periodic mode.
- **Time Stamp Counter** (TSC)
  - It is a processor register that counts the number of cycles since reset;

Time related devices (II)

- Local Advanced Programmable Interrupt Controller (LAPIC), which is part of the processor chip.
  - It manages **all interrupts received by the processor either from external devices or internally generated by software**;
  - It is provided with **mechanisms for queueing, nesting, and masking interrupts**;
  - It is able **to send interrupts for external devices as well as to generate interrupts** for the respective processor.

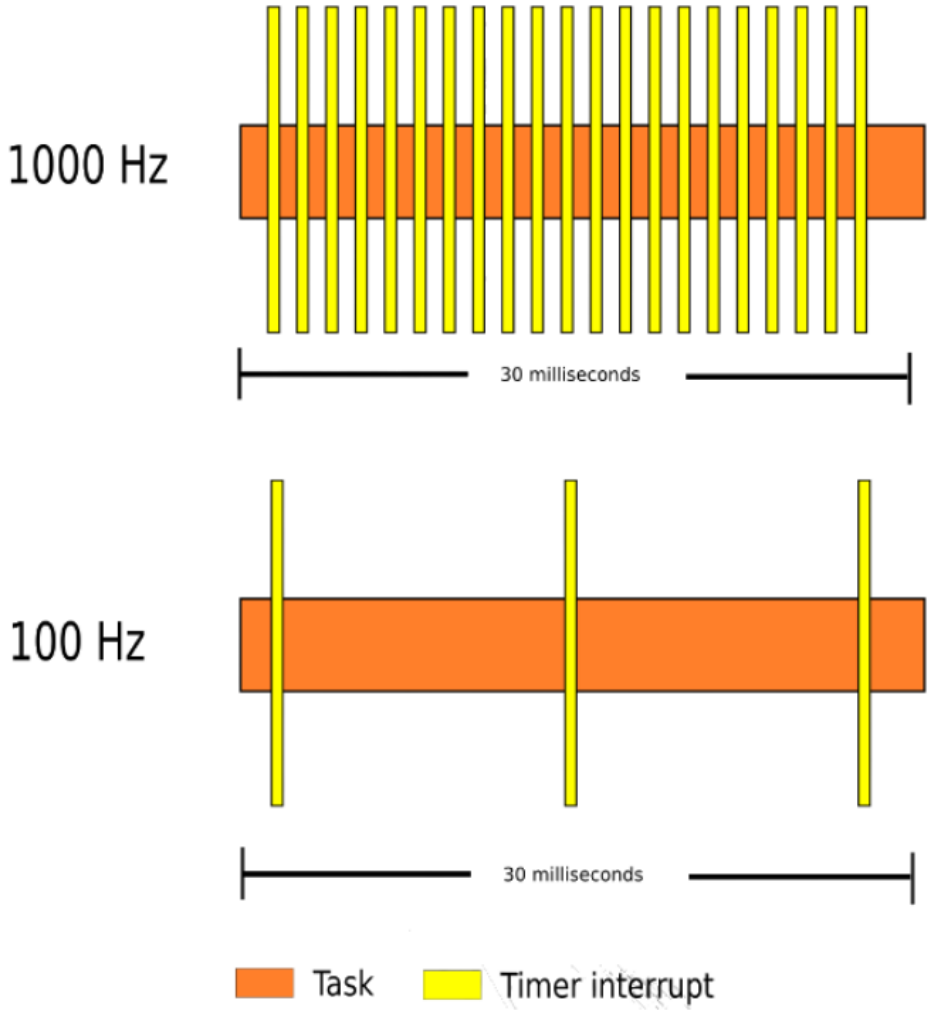


tick (I)

- Independently of the way that the time is managed, the Linux kernel **employs a periodic timer interrupt to get the control of the system**, called **tick**
  - It can be viewed as the kernel's heartbeat.
- The frequency of the this periodic timer interrupt (the tick rate) is defined by a kernel compilation option called `CONFIG_HZ`.
  - In the kernel code, this macro is only referred to by `HZ` , an acronym for hertz.

tick (II)

- Depending on the value of HZ , **the resolution of the tick** (the smallest time difference between consecutive ticks) **is more or less frequent**.



Other Subsystems

Modules

- Support for modules allows systems to have only a minimal base kernel image, with optional features and drivers supplied via loadable, separate objects.
- Modules also **enable the removal and reloading** of kernel code, facilitate debugging, and allow for the loading of new drivers on demand in response to the hot plugging of new devices.

Linux Kernel

Despite being monolithic in the sense that the whole kernel runs in a single address space, the Linux kernel is modular, supporting the dynamic insertion and removal of code from itself at runtime. Related subroutines, data, and entry and exit points are grouped together in a single binary image, a loadable kernel object, called a module.

Devices drivers

- They take on a special role in the Linux kernel.
- They **are distinct black boxes that make a particular piece of hardware respond to a well-defined internal programming interface**.
- They **hide completely the details of how the device works**.
- Typically device drivers represent physical hardware, but, not all device drivers represent physical devices.
  - Some device drivers are virtual, providing access to kernel functionality.
  - They are named pseudo devices;

Devices types

- Block devices are addressable in device-specified chunks called blocks and generally support seeking, the random access of data. Example block devices include hard drives, Blu-ray discs, and memory devices such as flash;
- Character devices are generally not addressable, providing access to data only as a stream, generally of characters (bytes). Example character devices include keyboards, mice, printers, and most pseudo-devices;
- Network devices provide access to a network (such as the Internet) via a physical adapter(such as laptop's 802.11 card) and a specific protocol (such as IP).