# MODULE 1: INTRODUCTION TO OPERATING SYSTEM

# SYLLABUS

Introduction, System Components, Open-Source Operating Systems, Operating System Services, System Calls, Process Management- Process Structure, Process states, Types of Schedulers, Scheduling Criteria, Scheduling algorithms. Deadlock and Starvation- Principles of Deadlock, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection and Recovery. Linux Environment, Fundamental Commands. System Shell and User Shells.

# WHAT IS AN OPERATING SYSTEM?

A program that acts as an intermediary between a user of a computer and the computer hardware

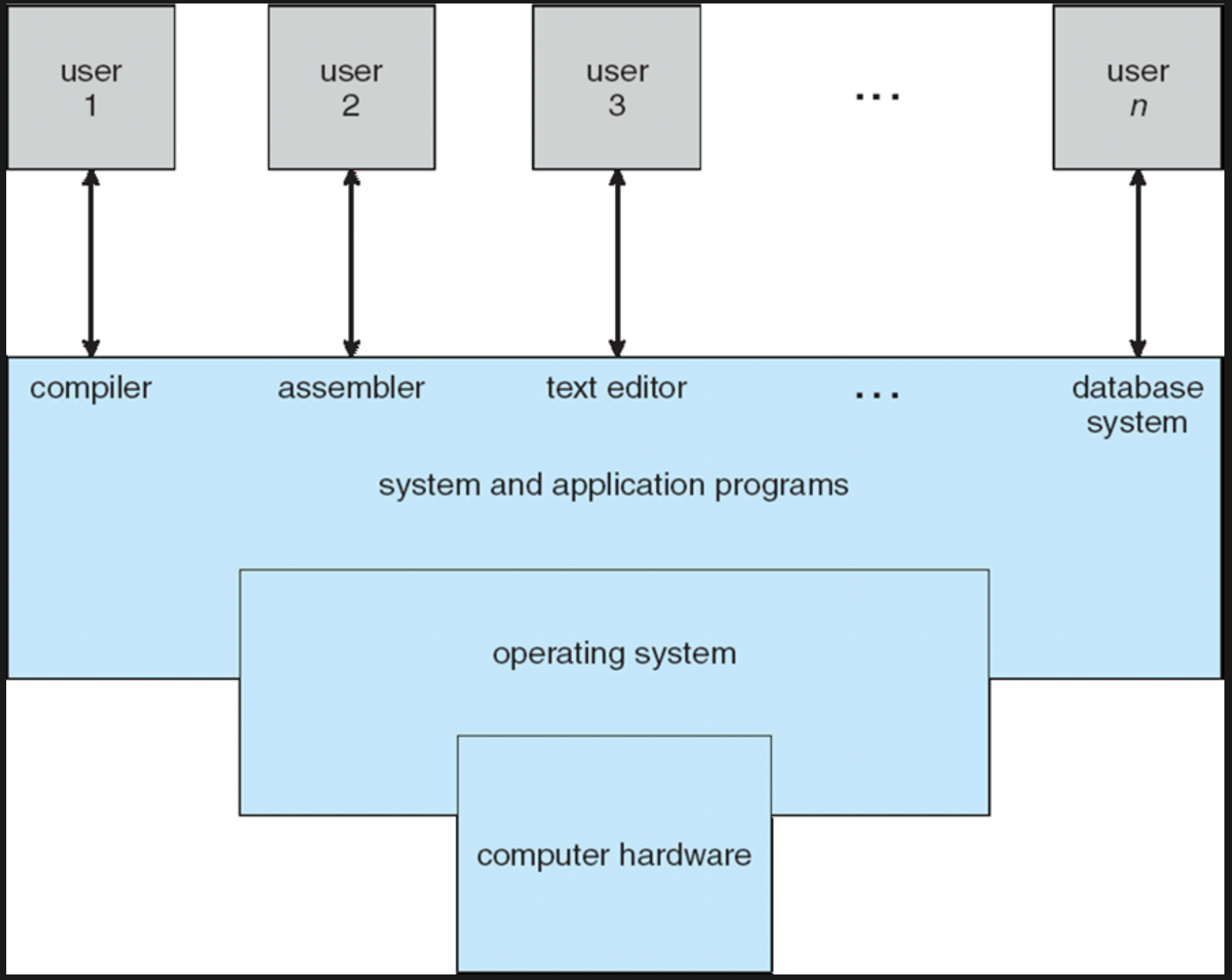
Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer system can be divided into four components:

- **Hardware** – provides basic computing resources: CPU, memory, I/O devices
- **Operating system** - Controls and coordinates use of hardware among various applications and users
- **Application programs** – define the ways in which the system resources are used to solve the computing problems of the users: Word

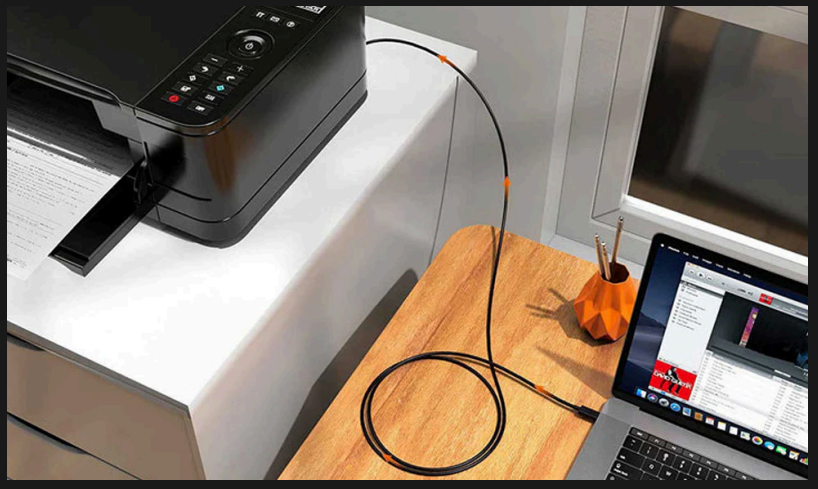
processors, compilers, web browsers, database systems, video games
- **Users**: People, machines, other computers


user 1
user 2
user 3
...
user n
compiler
assembler
text editor
...
database system
system and application programs
operating system
computer hardware

# WHAT OPERATING SYSTEMS DO

- Users want convenience, ease of use and good performance
    - Don’t care about resource utilization
- But shared computer such as mainframe or minicomputer must keep all users happy

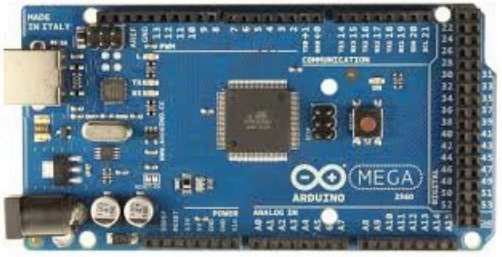
- Users of dedicated systems such as **workstations** have dedicated resources but frequently use **shared resources** from servers.
- **Handheld computers** are resource-poor, optimized for **usability** and **battery life**.
- Some computers have **little or no user interface**, such as **embedded computers** in devices and automobiles.

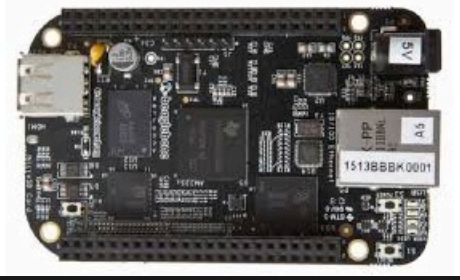


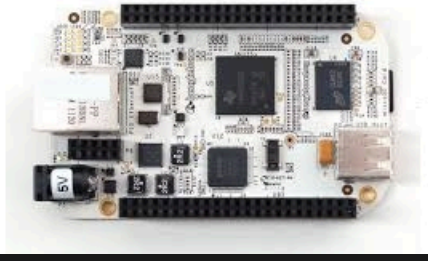




# What is Embedded Computer ???



# DEFINITION OF OS

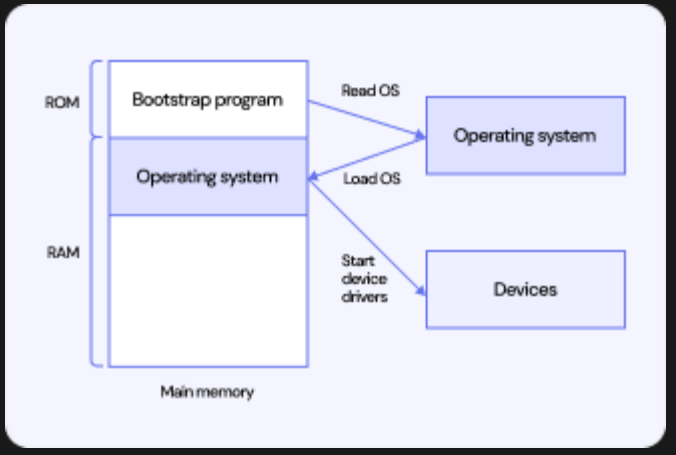
- OS is a **resource allocator**
- OS is a **control program**

# “THE ONE PROGRAM RUNNING AT ALL TIMES ON THE COMPUTER” IS THE KERNEL.

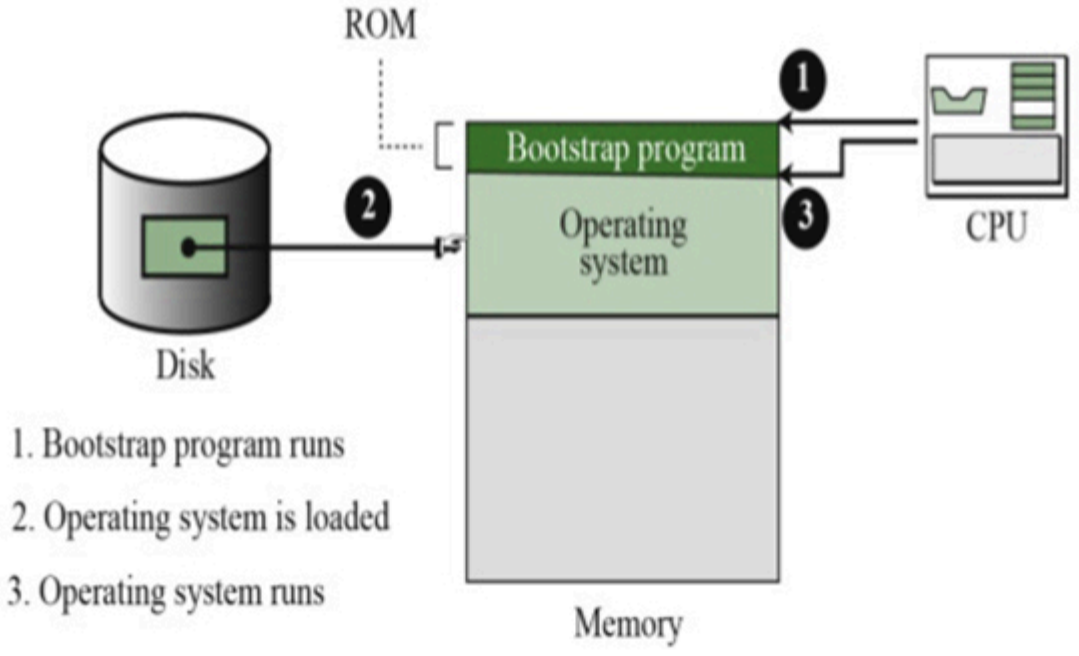
Everything else is either a system program (ships with the operating system) , or an application program.

# COMPUTER STARTUP

- **Bootstrap program** is loaded at power-up or reboot.
- Typically stored in **ROM** or **EPROM**, generally known as **firmware**.
- **Initializes** all aspects of the system.
- **Loads the operating system kernel** and starts execution.

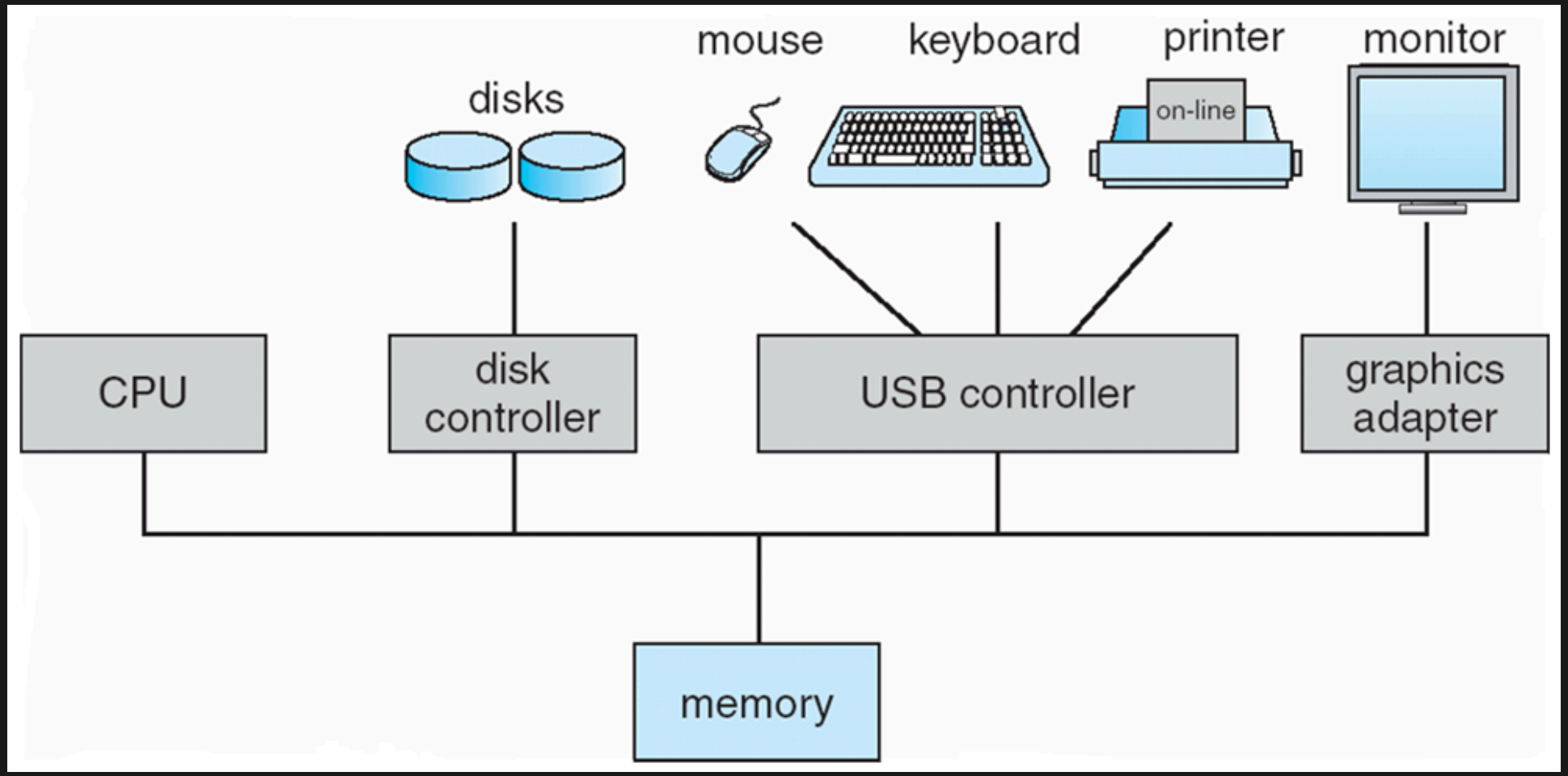

ROM
RAM
Bootstrap program
Operating system
Main memory
Read OS
Load OS
Start device drivers
Operating system
Devices

# Bootstrap process

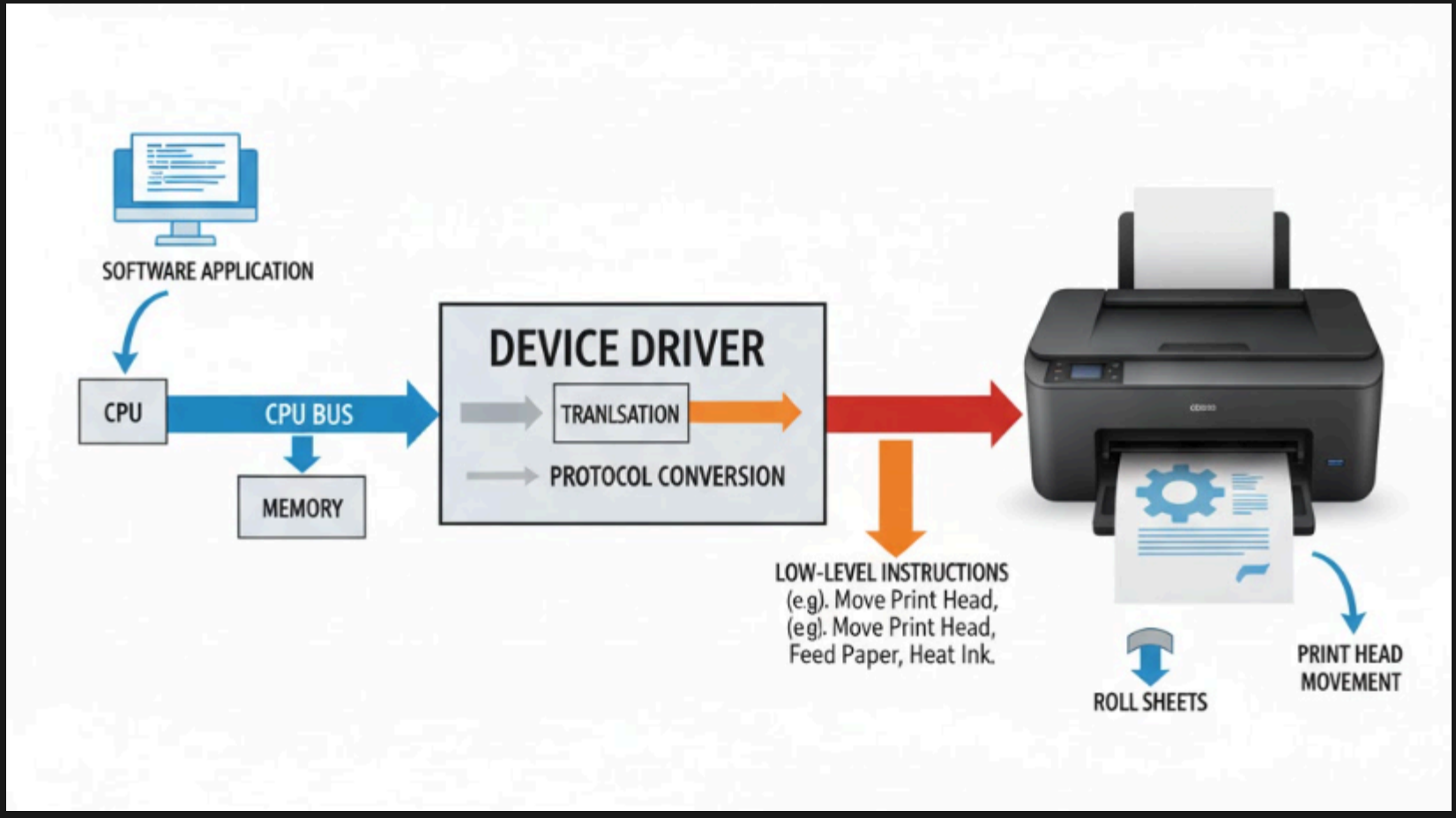


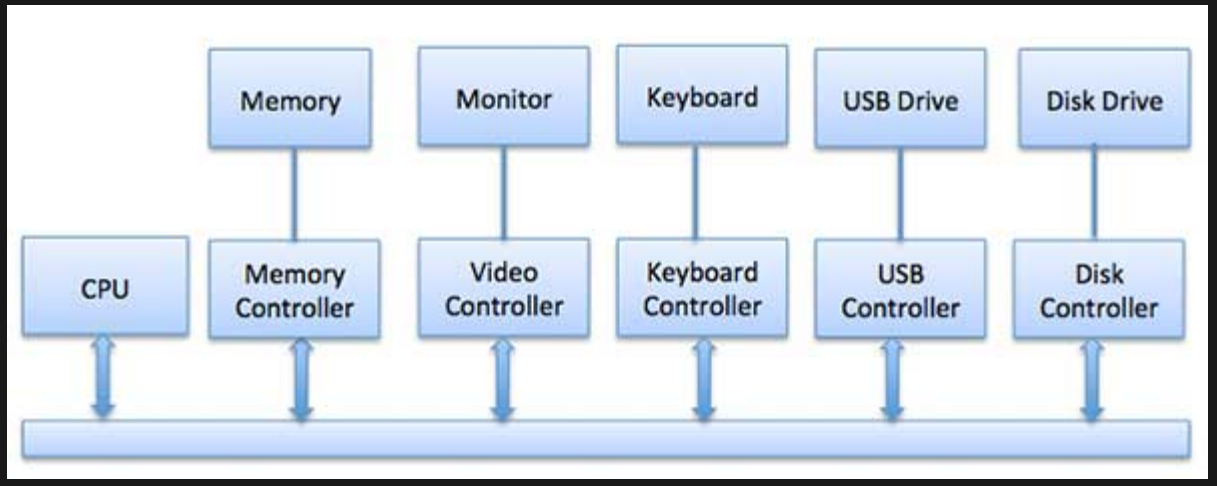
# COMPUTER SYSTEM ORGANIZATION

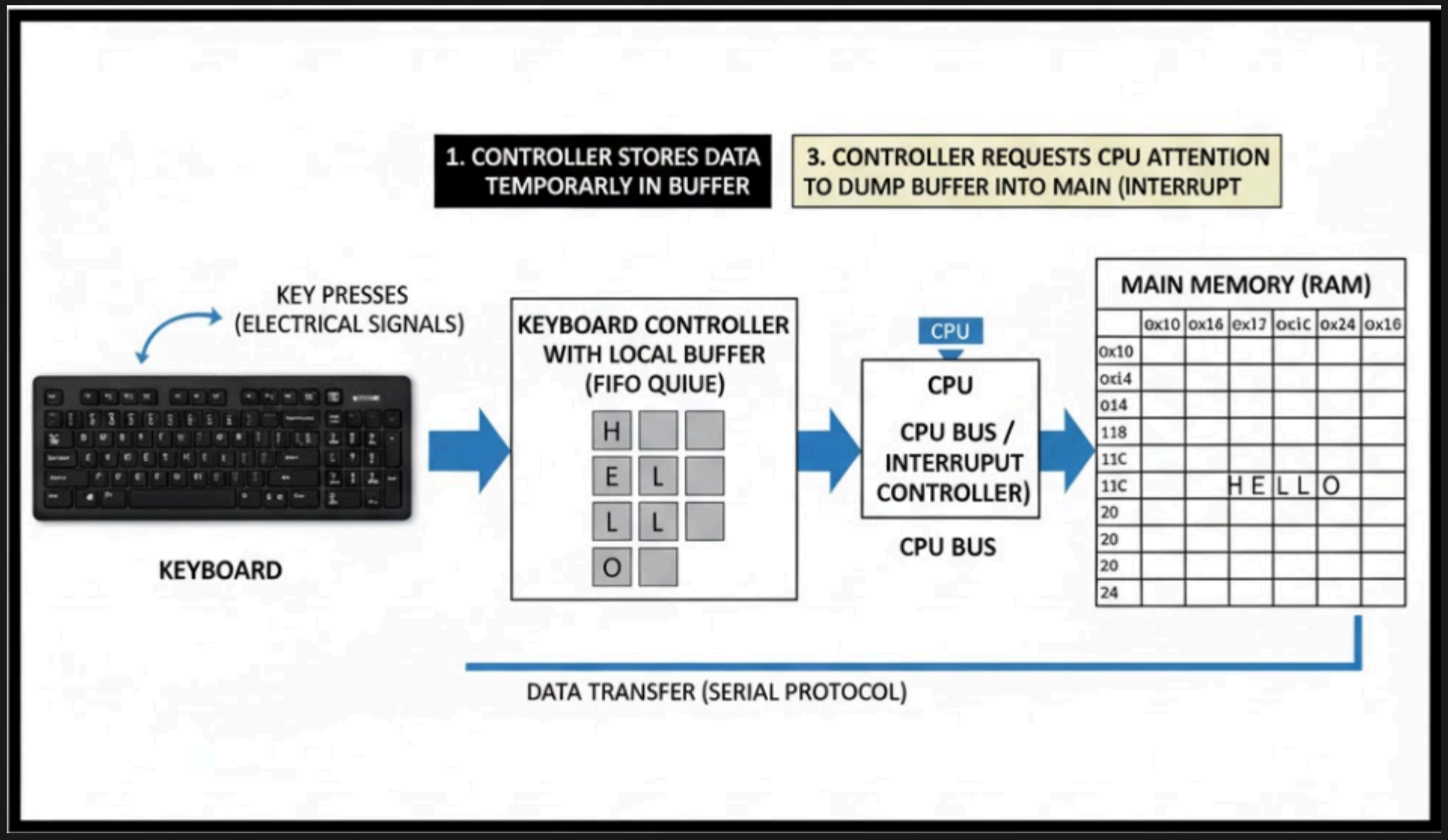
- **Computer-system operation:**
  - One or more **CPUs** and **device controllers** connect through a **common bus**, providing access to **shared memory**.
  - **Concurrent execution** of CPUs and devices occurs, **competing for memory cycles**.


mouse
keyboard
printer
monitor
disks
on-line
CPU
disk controller
USB controller
graphics adapter
memory

- **I/O devices** and the **CPU** can execute concurrently.
- Each **device controller** is responsible for a particular **device type**.
- Each device controller has a **local buffer**.
- **CPU** moves data **from/to main memory** to/from **local buffers**.
- **I/O** occurs **from the device to the local buffer** of the controller.
- The **device controller informs the CPU** that it has finished its operation by causing an **interrupt**.


SOFTWARE APPLICATION
CPU
CPU BUS
MEMORY
DEVICE DRIVER
TRANLSATION
PROTOCOL CONVERSION
LOW-LEVEL INSTRUCTIONS
(e.g). Move Print Head,
(e.g). Move Print Head,
Feed Paper, Heat Ink.
ROLL SHEETS
PRINT HEAD
MOVEMENT

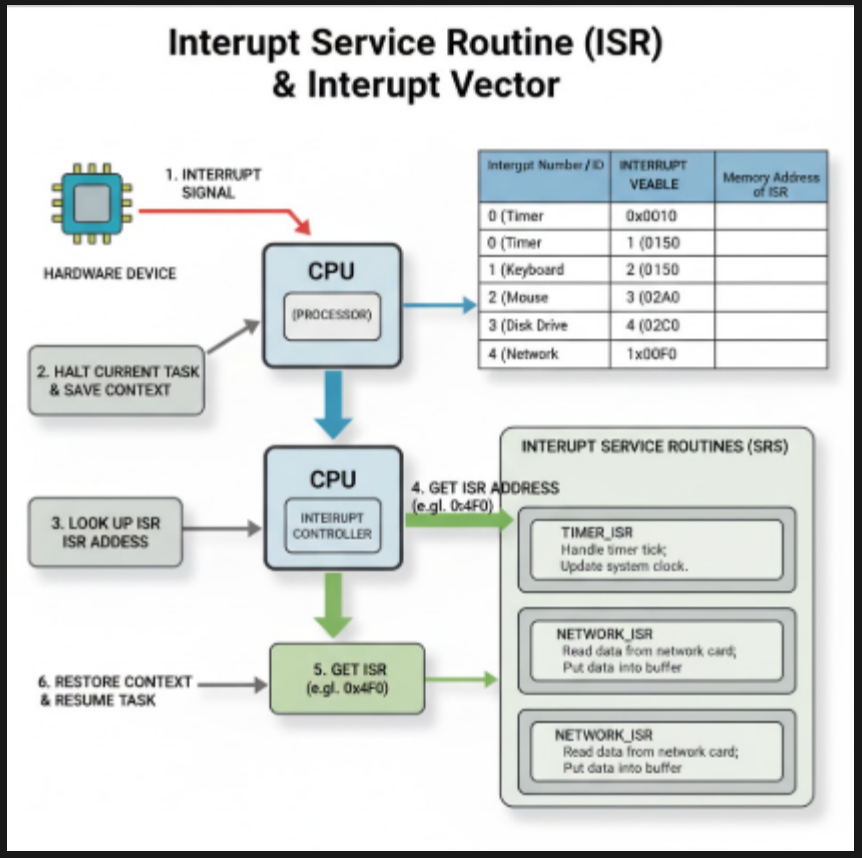

Memory
Monitor
Keyboard
USB Drive
Disk Drive
CPU
Memory Controller
Video Controller
Keyboard Controller
USB Controller
Disk Controller


1. CONTROLLER STORES DATA TEMPORARLY IN BUFFER
3. CONTROLLER REQUESTS CPU ATTENTION TO DUMP BUFFER INTO MAIN (INTERRUPT
KEY PRESSES (ELECTRICAL SIGNALS)
KEYBOARD
KEYBOARD CONTROLLER WITH LOCAL BUFFER (FIFO QUIUE)
H
E L
L L
O
CPU
CPU
CPU BUS / INTERRUPUT CONTROLLER)
CPU BUS
MAIN MEMORY (RAM)
0x10 0x16 ex17 ociC 0x24 0x16
0x10
oci4
014
118
11C
11C H E L L O
20
20
20
24
DATA TRANSFER (SERIAL PROTOCOL)

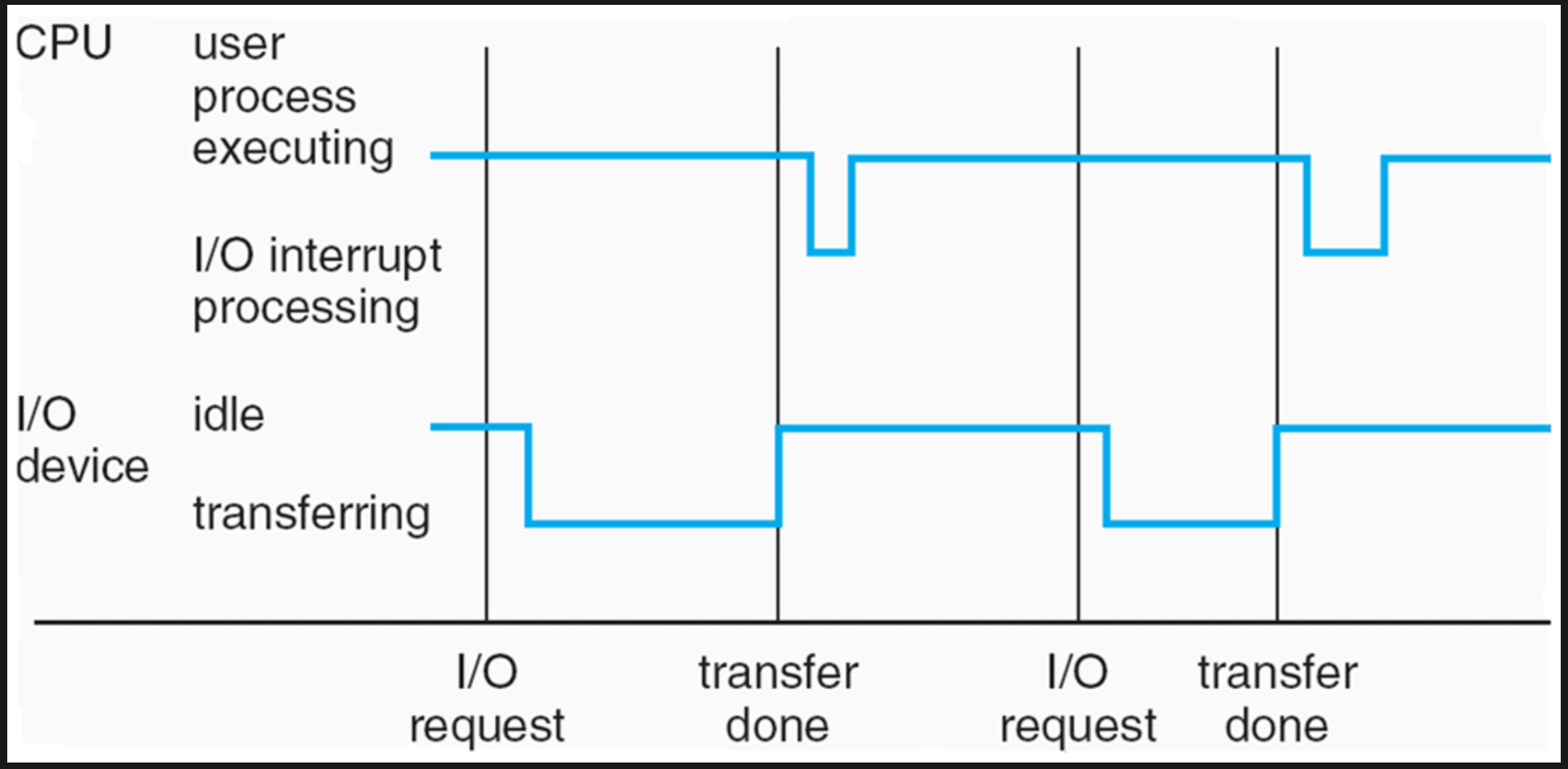
# INTERRUPTS

- **Interrupt** transfers control to the **interrupt service routine (ISR)**, generally through the **interrupt vector**, which contains the addresses of all service routines.
- **Interrupt architecture** must save the **address of the interrupted instruction**.
- A **trap** or **exception** is a **software-generated interrupt** caused either by an **error** or a **user request**.
- An **operating system** is **interrupt-driven**.

# Interupt Service Routine (ISR) & Interupt Vector

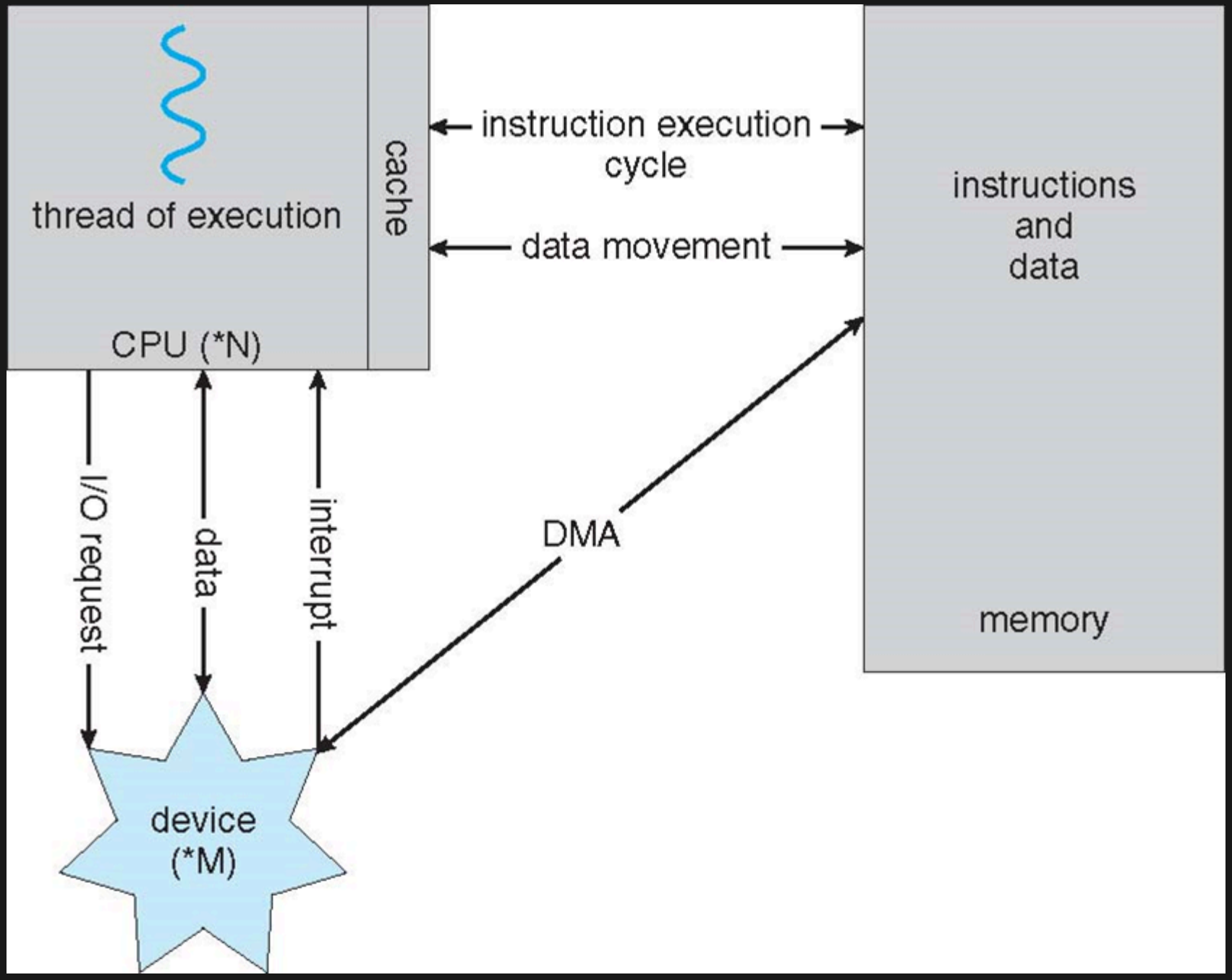



| Intergpt Number / ID | INTERRUPT VEABLE | Memory Address of ISR |
|---|---|---|
| 0 (Timer | 0x0010 | |
| 0 (Timer | 1 (0150 | |
| 1 (Keyboard | 2 (0150 | |
| 2 (Mouse | 3 (02A0 | |
| 3 (Disk Drive | 4 (02C0 | |
| 4 (Network | 1x00F0 | |


CPU
user process executing
I/O interrupt processing
I/O device
idle
transferring
I/O request
transfer done
I/O request
transfer done

# DIRECT MEMORY ACCESS (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte


thread of execution
cache
CPU (*N)
instruction execution cycle
data movement
instructions and data
memory
I/O request
data
interrupt
DMA
device (*M)

# COMPUTER-SYSTEM ARCHITECTURE

- Most systems use a **single general-purpose processor**.
- Most systems also have **special-purpose processors**.
- **Multiprocessor systems** are growing in use and importance.
- Also known as **parallel systems** or **tightly-coupled systems**.

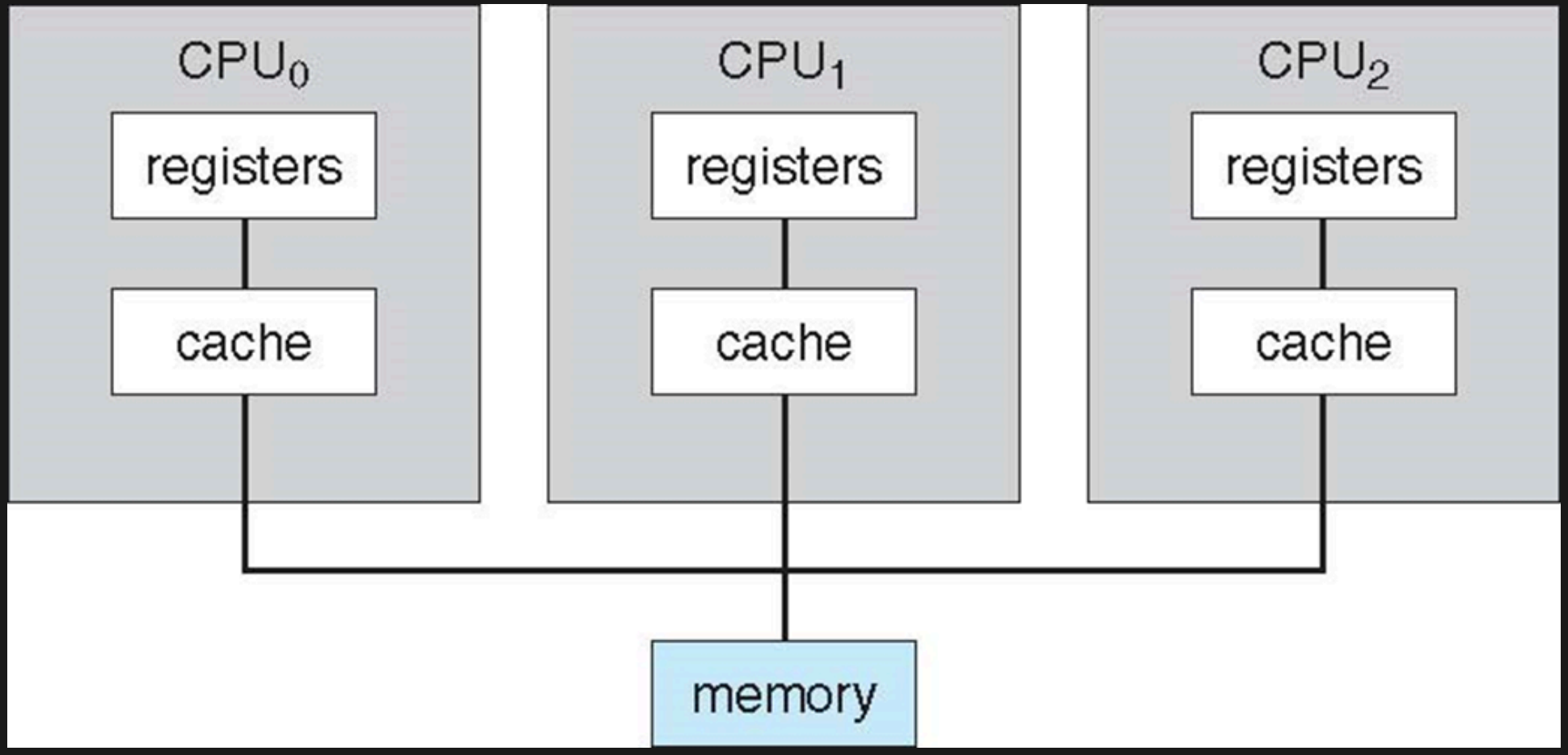
**Advantages include:**

- **Increased throughput**
- **Economy of scale**
- **Increased reliability** – graceful degradation or fault tolerance

**Two types:**

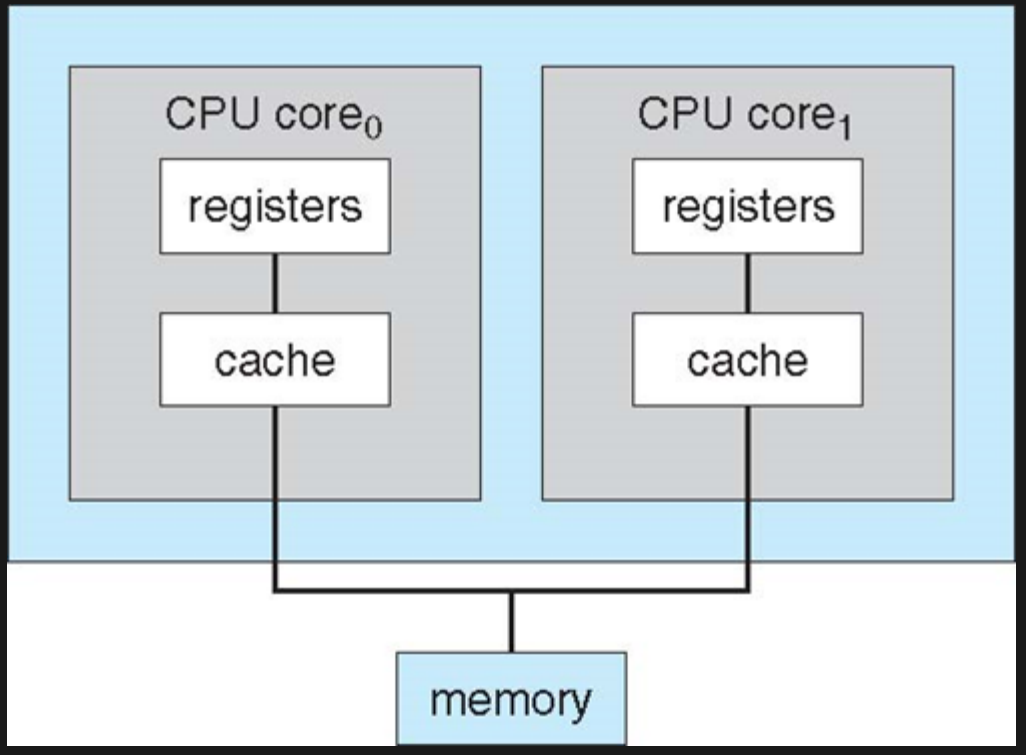
- **Asymmetric Multiprocessing (AMP):** Each processor is assigned a specific task.
- **Symmetric Multiprocessing (SMP):** Each processor performs all tasks.

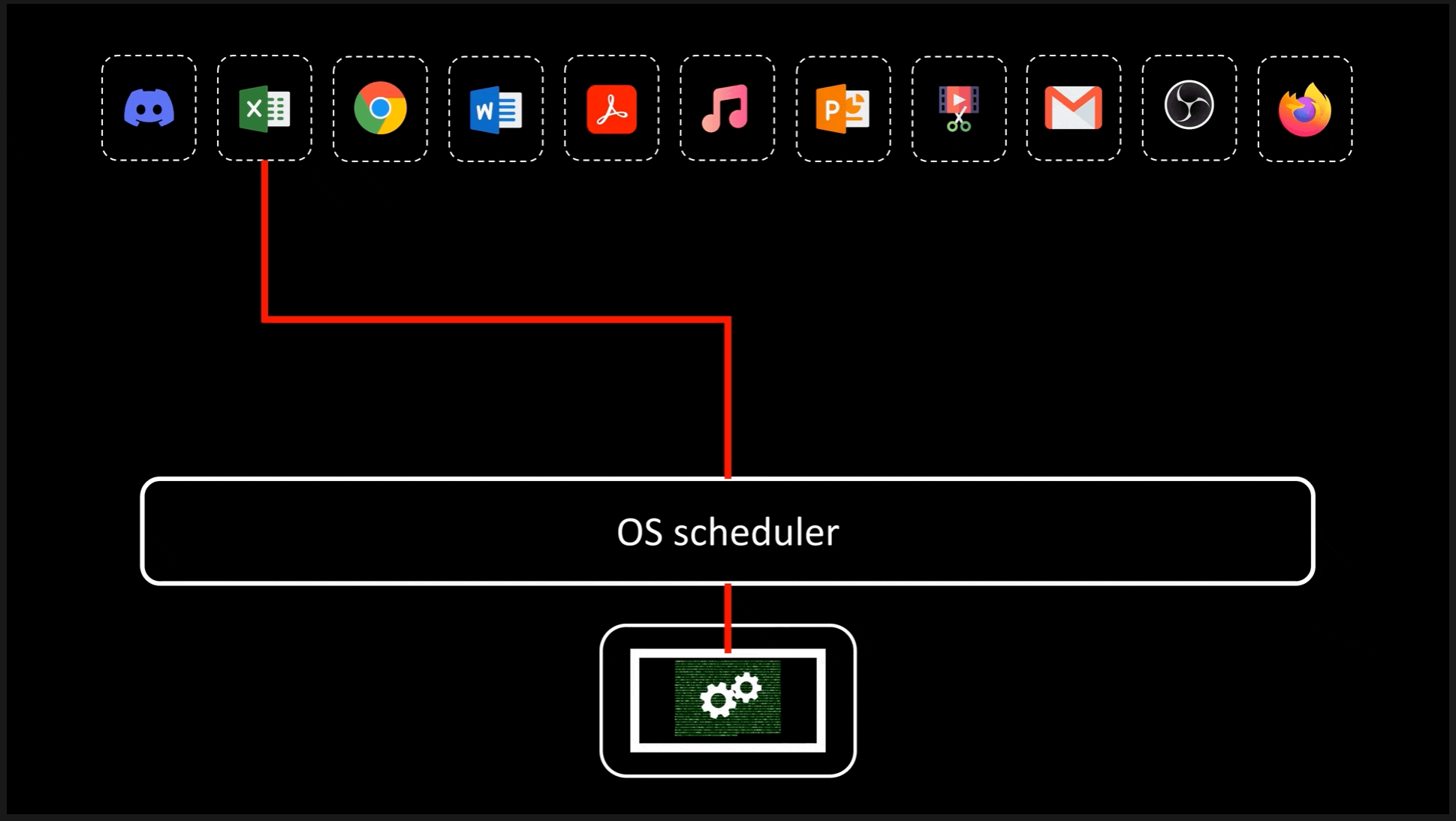
# SYMMETRIC MULTIPROCESSING ARCHITECTURE



# A DUAL-CORE DESIGN

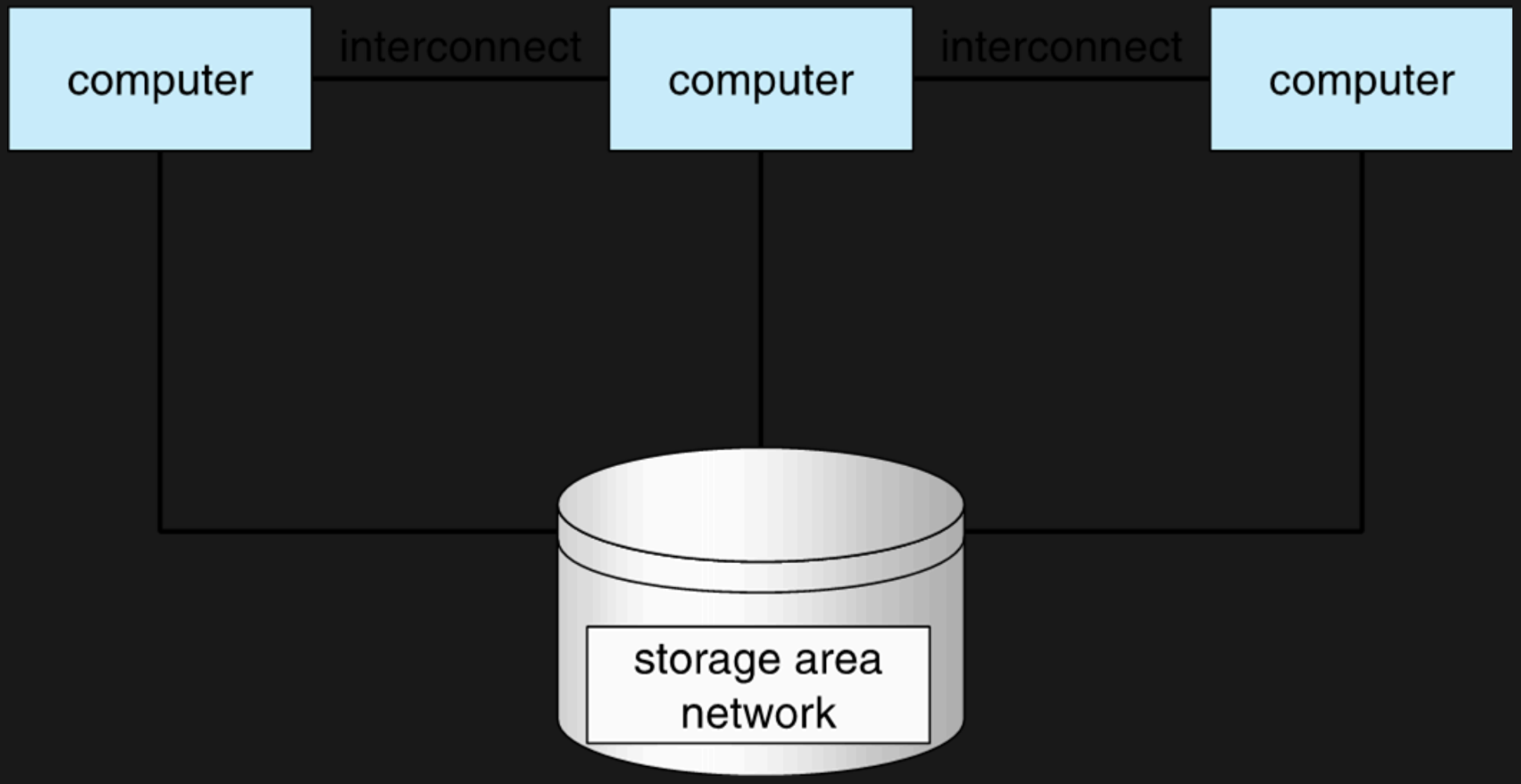
- Multi-chip and multicore
- Systems containing all chips
- Chassis containing multiple separate systems




OS scheduler

# CLUSTERED SYSTEMS

- Multiple systems working together
- Usually share storage via a **Storage-Area Network (SAN)**


computer
interconnect
computer
interconnect
computer
storage area
network

- Provide **high availability** and survive failures

- **Asymmetric clustering:** one machine in hot-standby mode (monitors all)
- **Symmetric clustering:** multiple active nodes monitoring each other

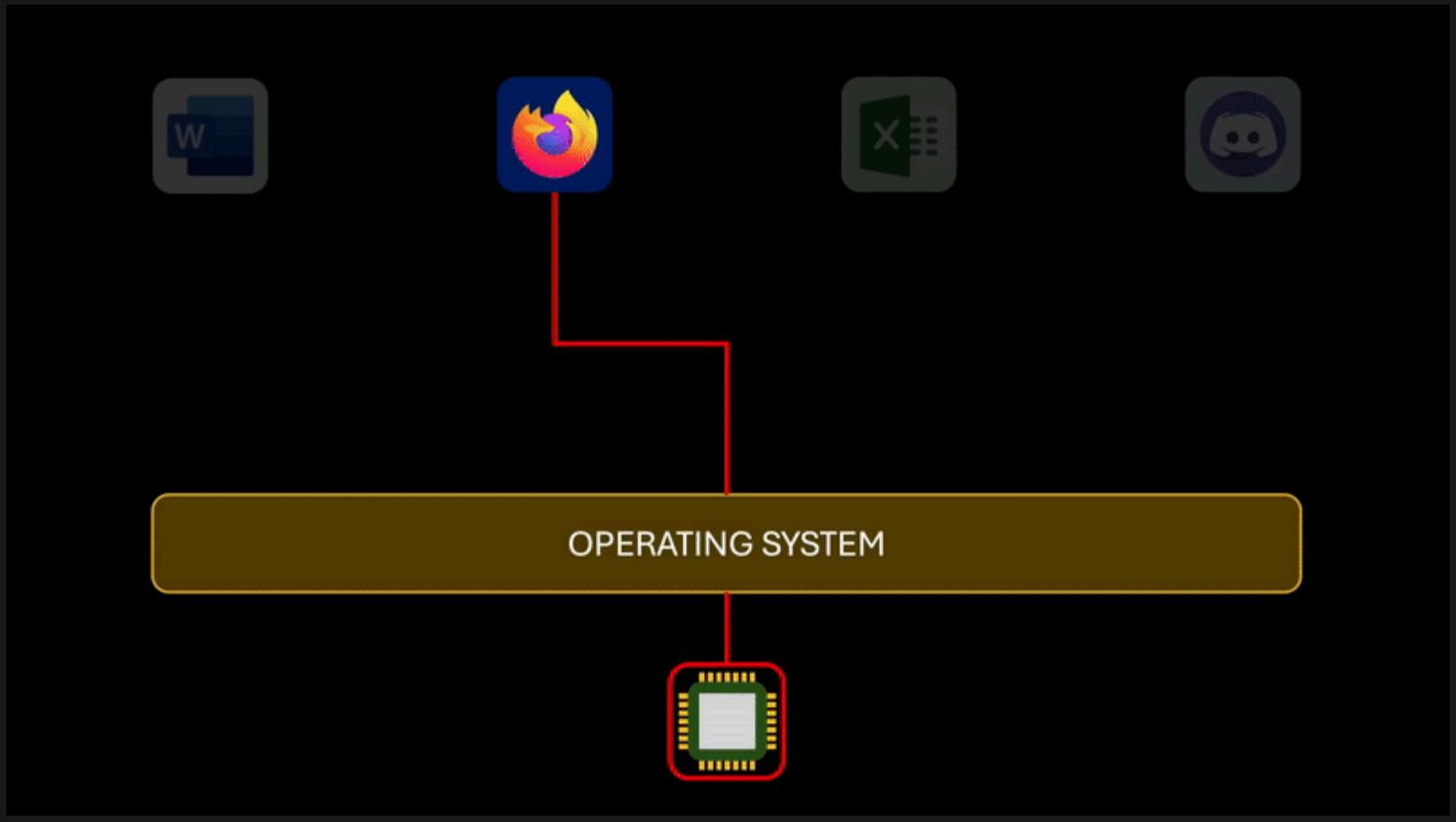
- Some clusters are for **High-Performance Computing (HPC)**
- Applications must support **parallelization**

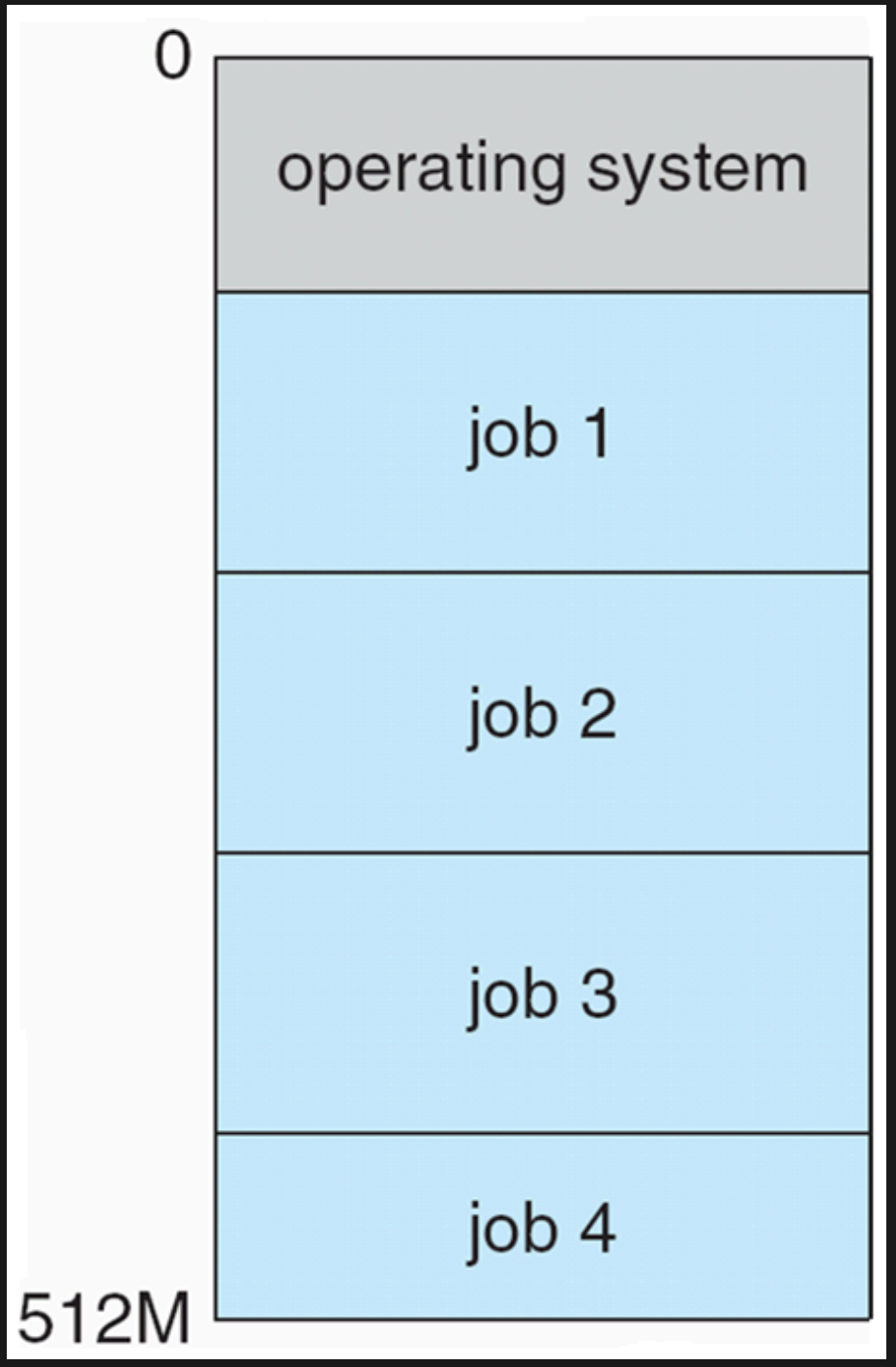
- Use **Distributed Lock Manager (DLM)** to prevent conflicting operations

# MULTIPROGRAMMING (BATCH SYSTEM)

- Needed for **efficiency**
- A **single user** cannot keep CPU and I/O devices busy at all times
- **Multiprogramming** organizes jobs so CPU always has one to execute

- A **subset of jobs** is kept in memory
- **Job scheduling** selects and runs one job at a time
- When a job waits (e.g., for I/O), **OS switches** to another job

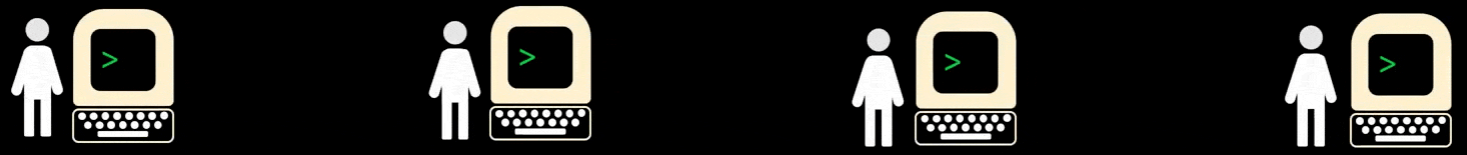

OPERATING SYSTEM

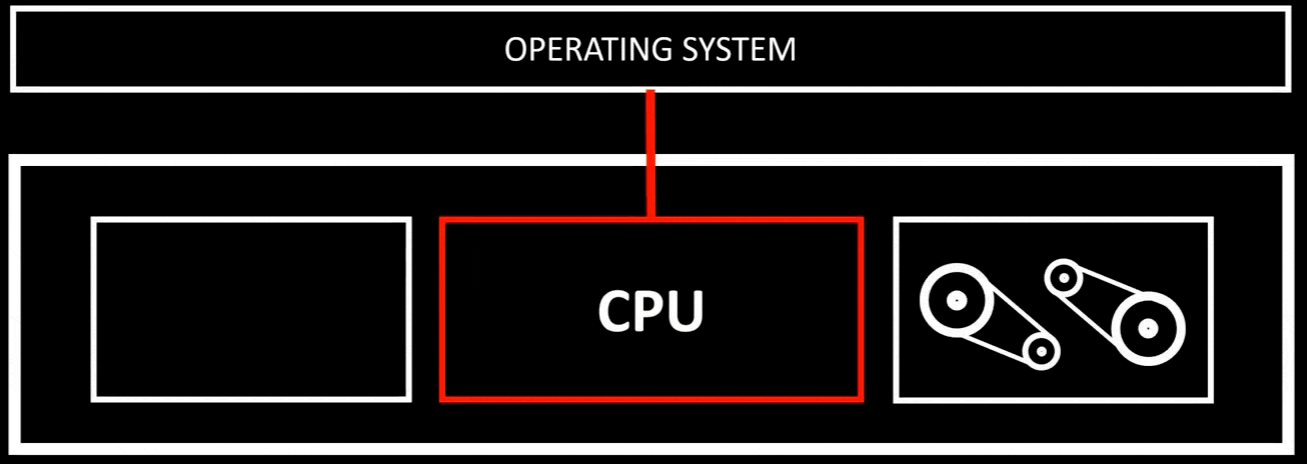

0
operating system
job 1
job 2
job 3
job 4
512M

# TIME SHARING (MULTITASKING)

- Logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive computing**
- **Response time should be < 1 second**
- Each user has at least one program executing in memory → **process**

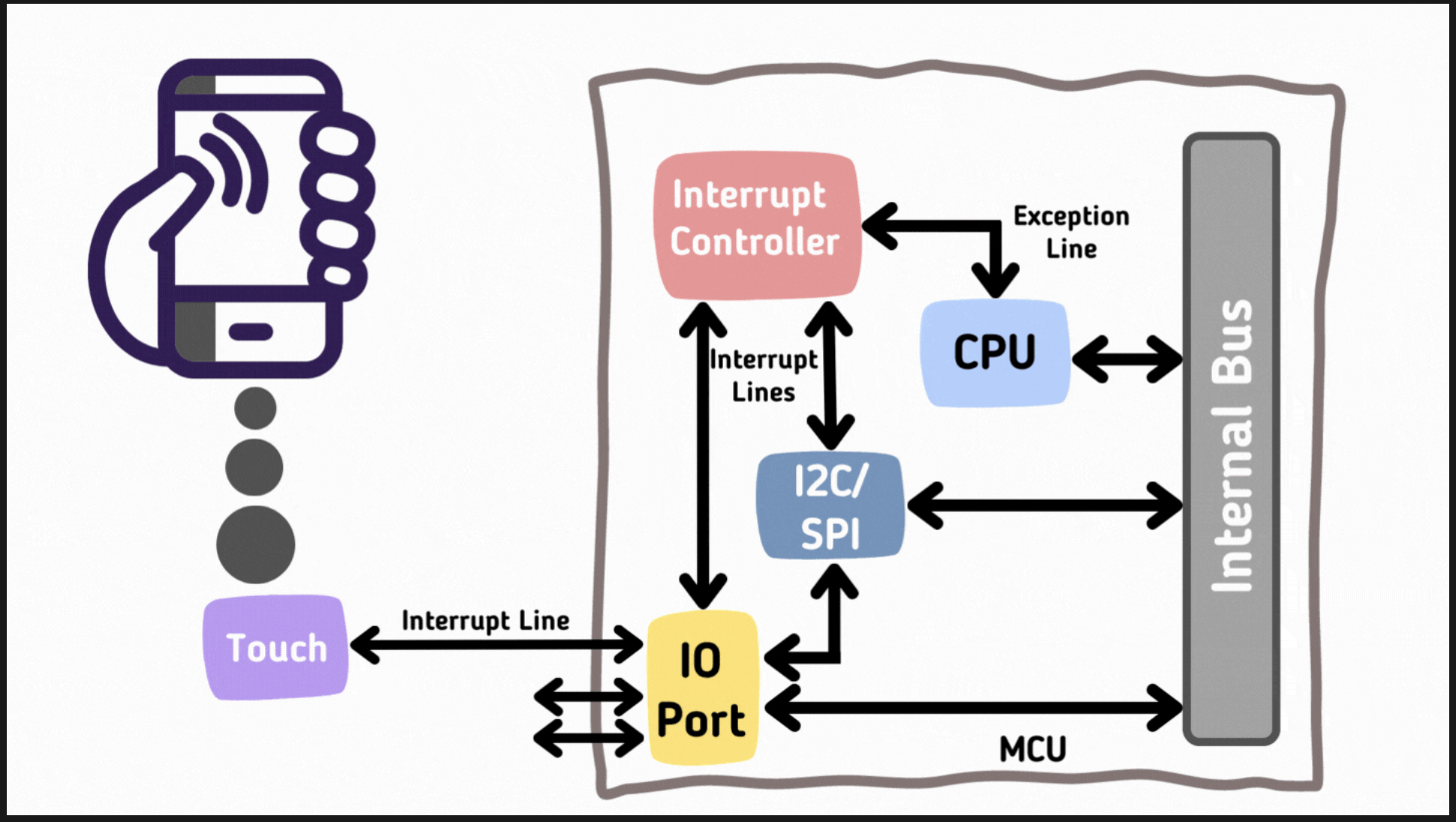
- If several jobs ready to run at the same time → **CPU scheduling**
- If processes don’t fit in memory, **swapping** moves them in and out to run
- **Virtual memory** allows execution of processes not completely in memory




OPERATING SYSTEM
CPU

# OPERATING-SYSTEM OPERATIONS

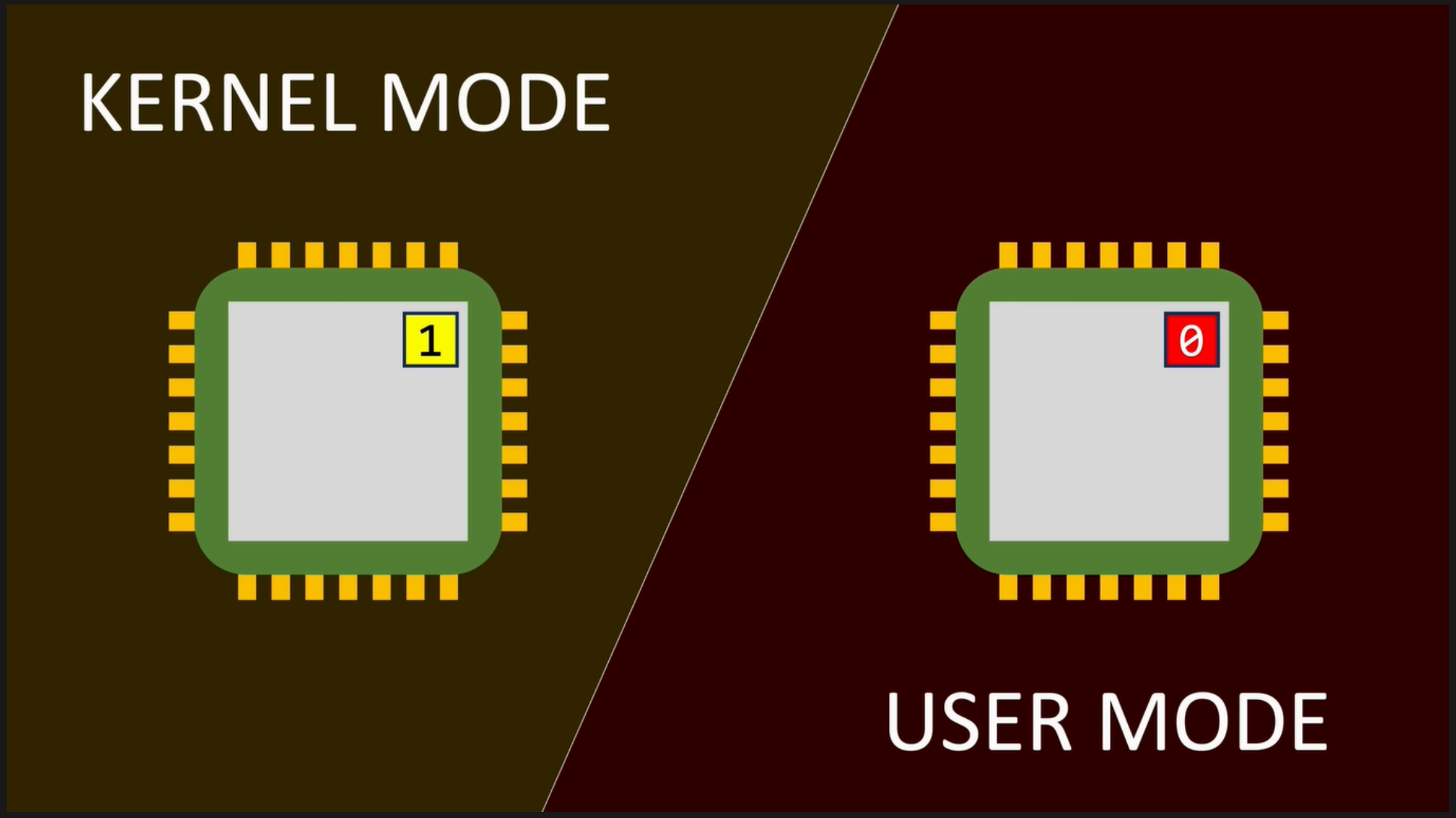
- **Interrupt-driven:** hardware and software
- **Hardware interrupt:** generated by devices
- **Software interrupt (exception/trap):**
  - Software errors (e.g., division by zero)
  - Requests for OS services
- Other process issues: infinite loops, processes modifying each other or the OS


Interrupt Controller
Exception Line
CPU
Internal Bus
Interrupt Lines
I2C/ SPI
Touch
Interrupt Line
IO Port
MCU

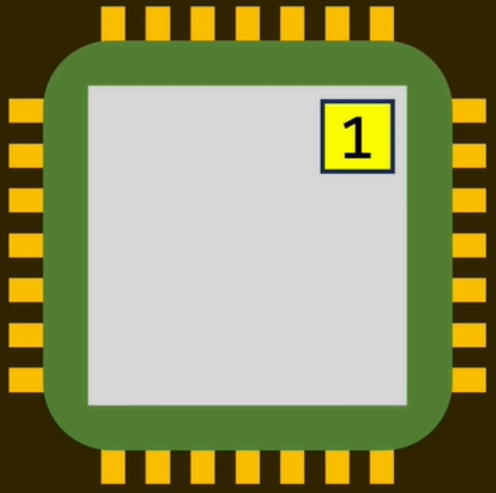
# DUAL-MODE OPERATION

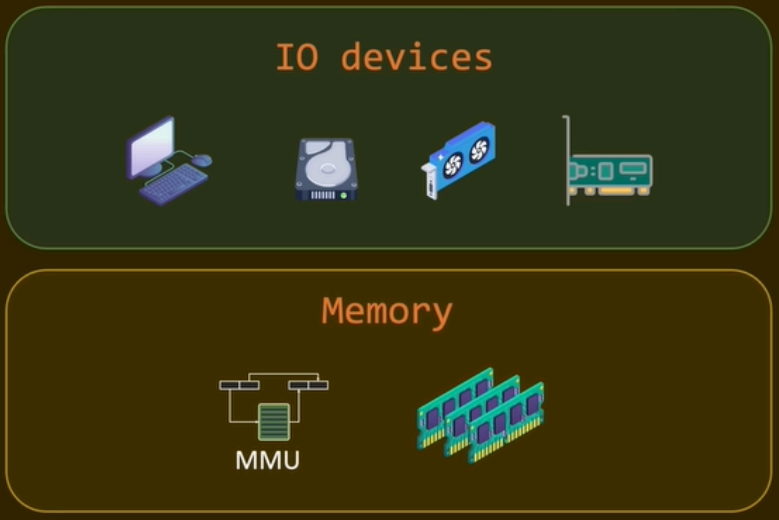
- Allows OS to **protect itself** and other system components
- Two modes: **User mode** and **Kernel mode**
- **Mode bit** provided by hardware to distinguish user vs. kernel execution
- Some instructions are **privileged**, only executable in kernel mode

- **System call** switches mode to kernel; return resets to user
- Modern CPUs support **multi-mode operations**
- Example: **Virtual Machine Manager (VMM) mode** for guest VMs


KERNEL MODE
1
0
USER MODE

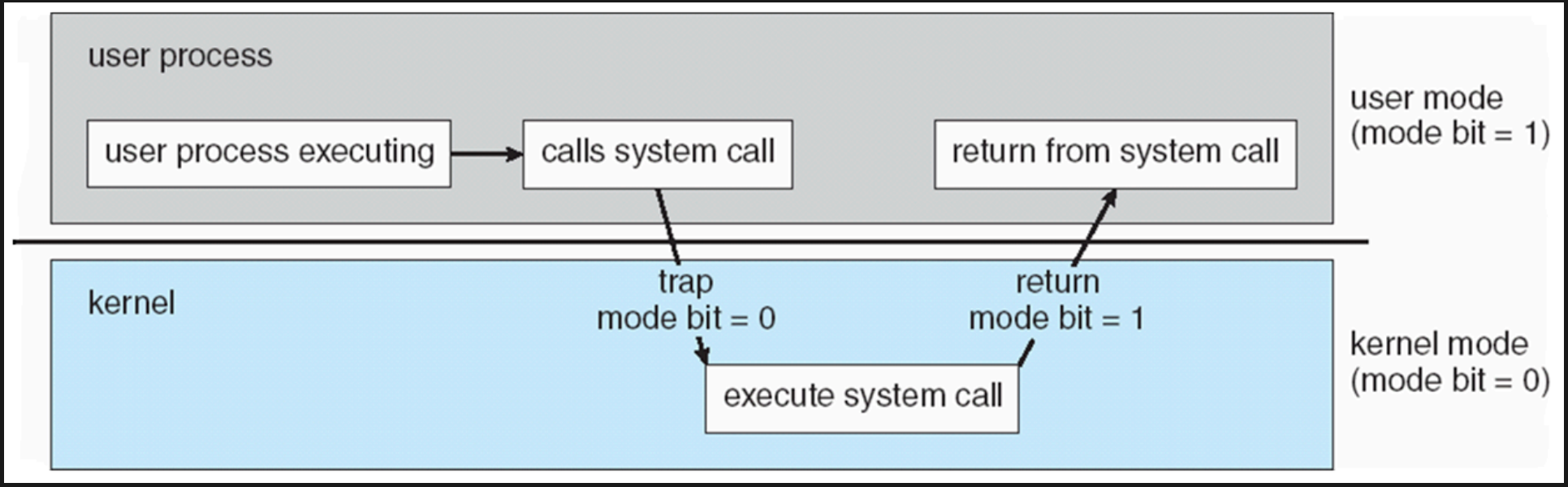
# KERNEL MODE



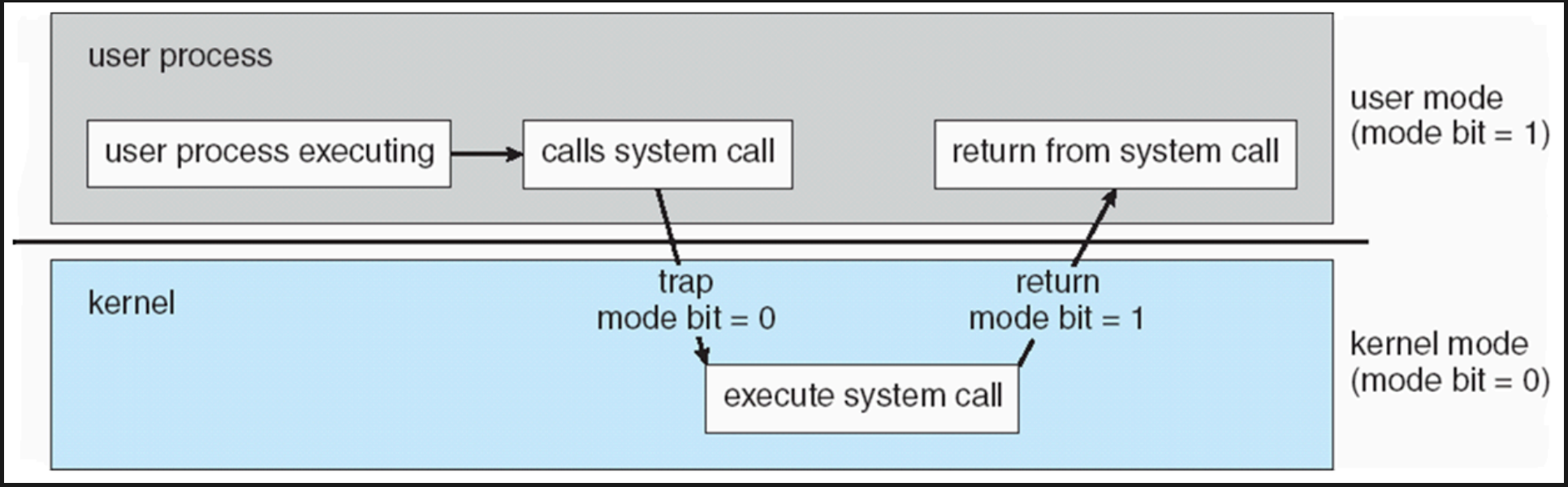

Interrupts


User mode
Kernel mode
OS


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

# TIMER IN OPERATING SYSTEMS

- Prevents **infinite loops** and processes hogging resources
- Timer set to interrupt after a defined time period
- Uses a **counter decremented by the physical clock**
- OS sets the counter (privileged instruction)
- When counter reaches zero → **interrupt generated**
- Ensures OS can **regain control** or terminate long-running processes


user process
user process executing
calls system call
return from system call
user mode
(mode bit = 1)
kernel
trap
mode bit = 0
return
mode bit = 1
execute system call
kernel mode
(mode bit = 0)

# COMPUTING ENVIRONMENTS

# TRADITIONAL COMPUTING

- Single-user or batch systems
- Centralized processing on mainframes or PCs
- Jobs executed sequentially, often non-interactive

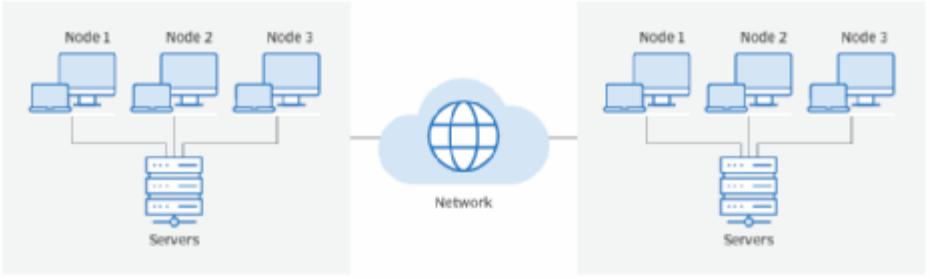
# MOBILE COMPUTING

- Supports computing on handheld or portable devices
- Enables **wireless connectivity**, mobility, and anytime access
- Applications optimized for battery and resource constraints

# DISTRIBUTED COMPUTING

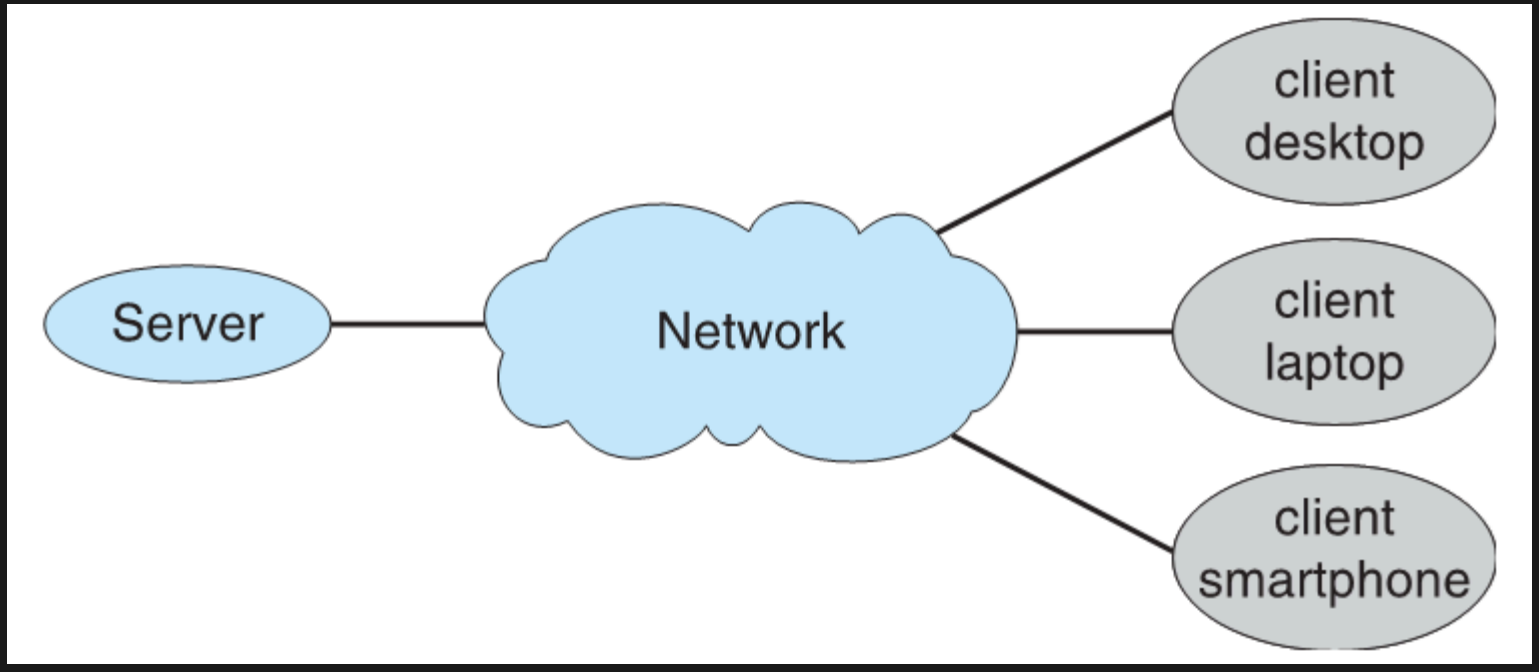
- Multiple systems work together over a network
- Share resources and collaborate on tasks
- Provides **scalability, fault tolerance, and parallel processing**

# The distributed computing process



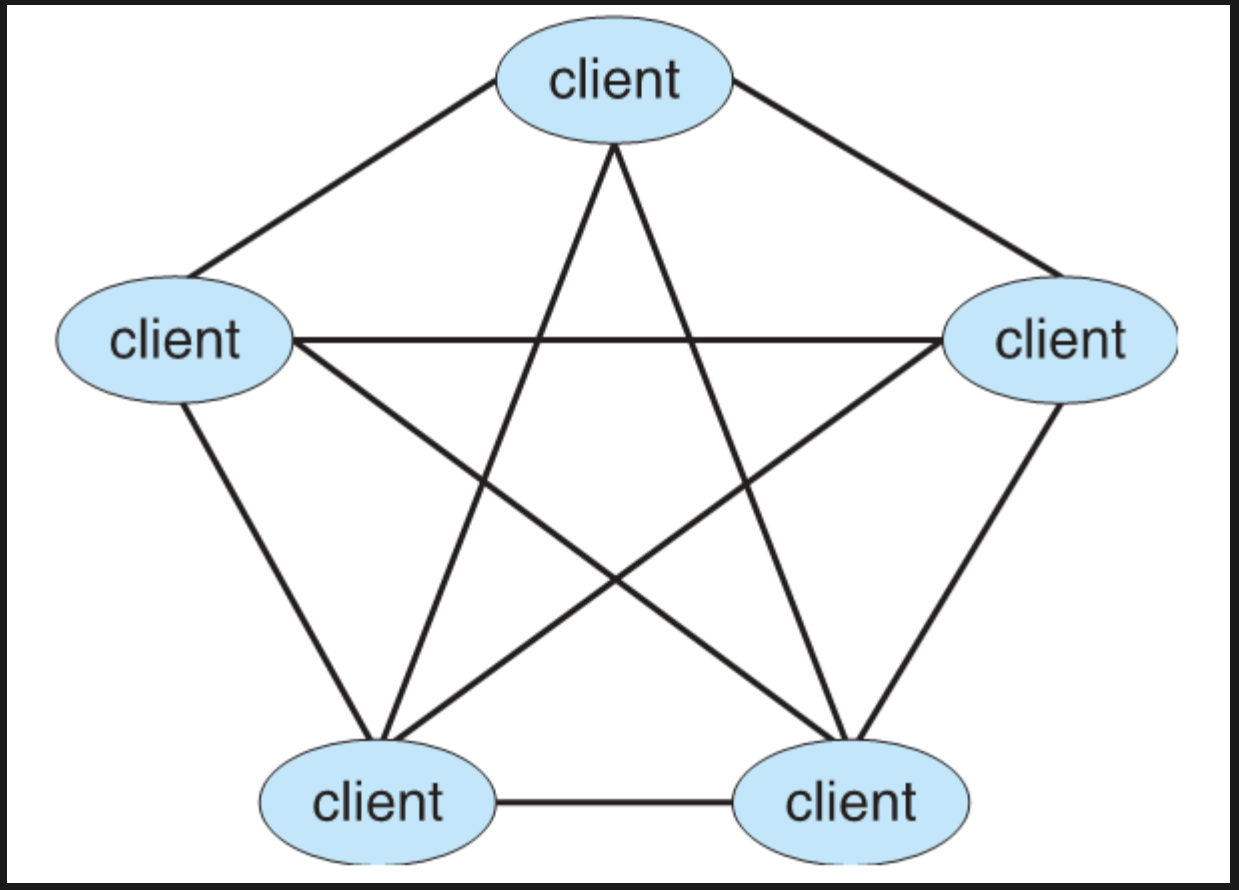
# CLIENT-SERVER COMPUTING

- **Clients** request services; **servers** provide them
- Centralized control with multiple clients accessing shared resources
- Common in business applications and web services


client
desktop
Server
Network
client
laptop
client
smartphone

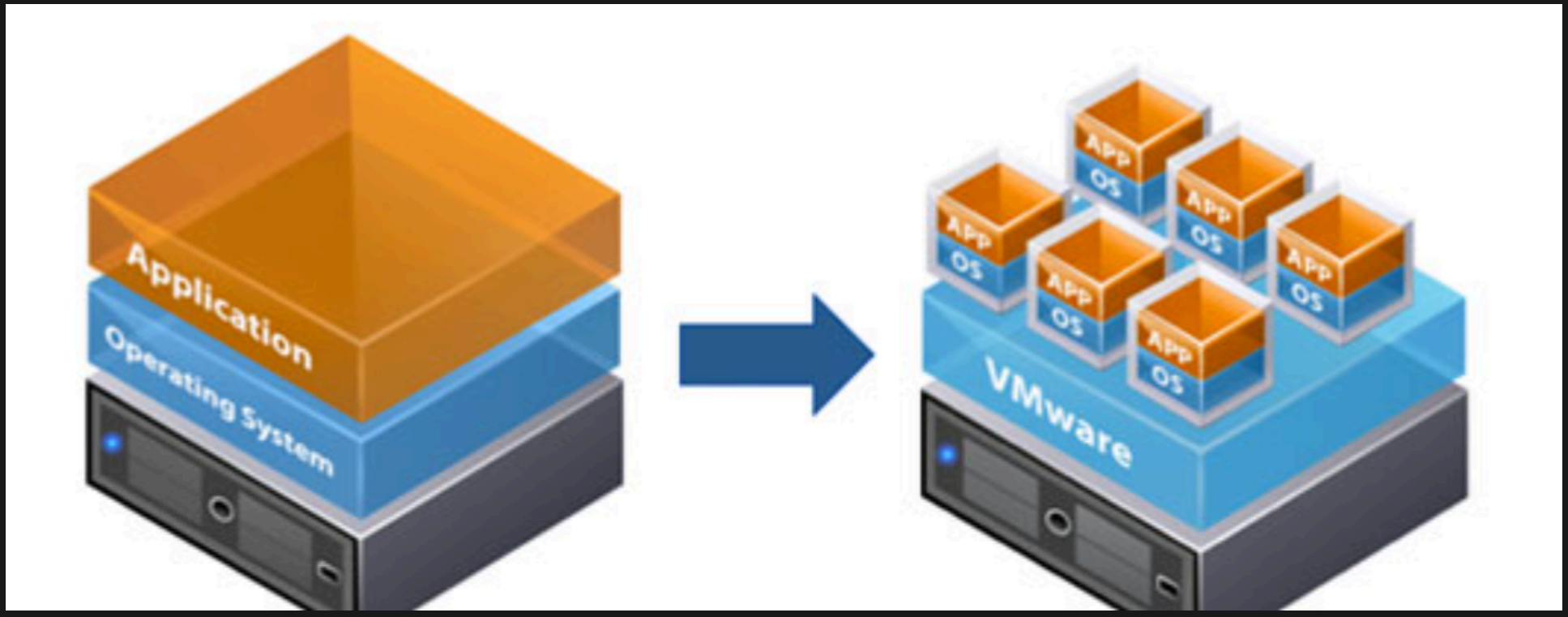
# PEER-TO-PEER (P2P) COMPUTING

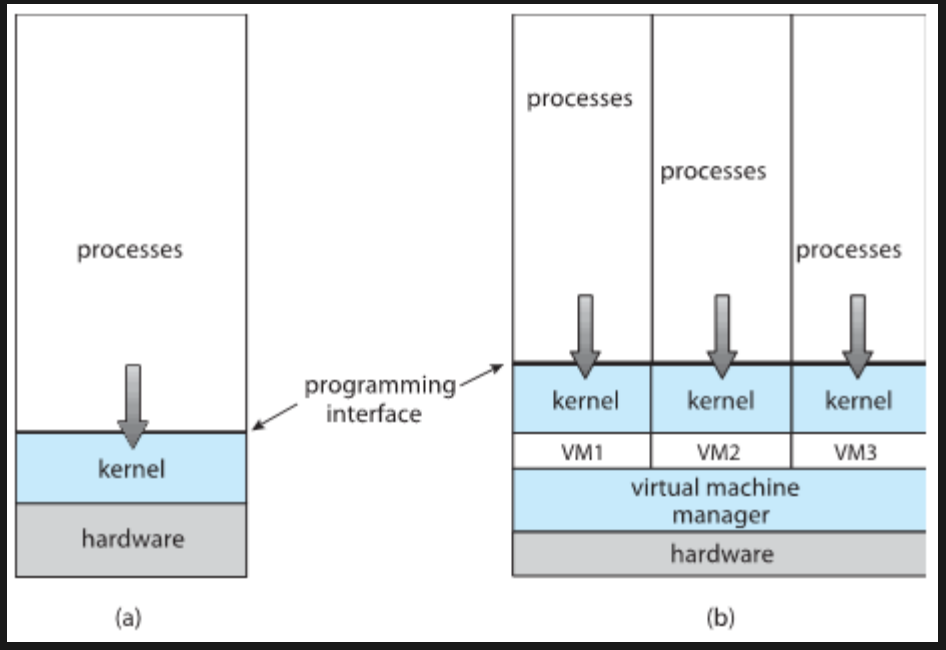
- All nodes act as **both clients and servers**
- Resources and services shared directly among peers
- Examples: file-sharing networks, blockchain networks


client
client
client
client
client

# VIRTUALIZATION

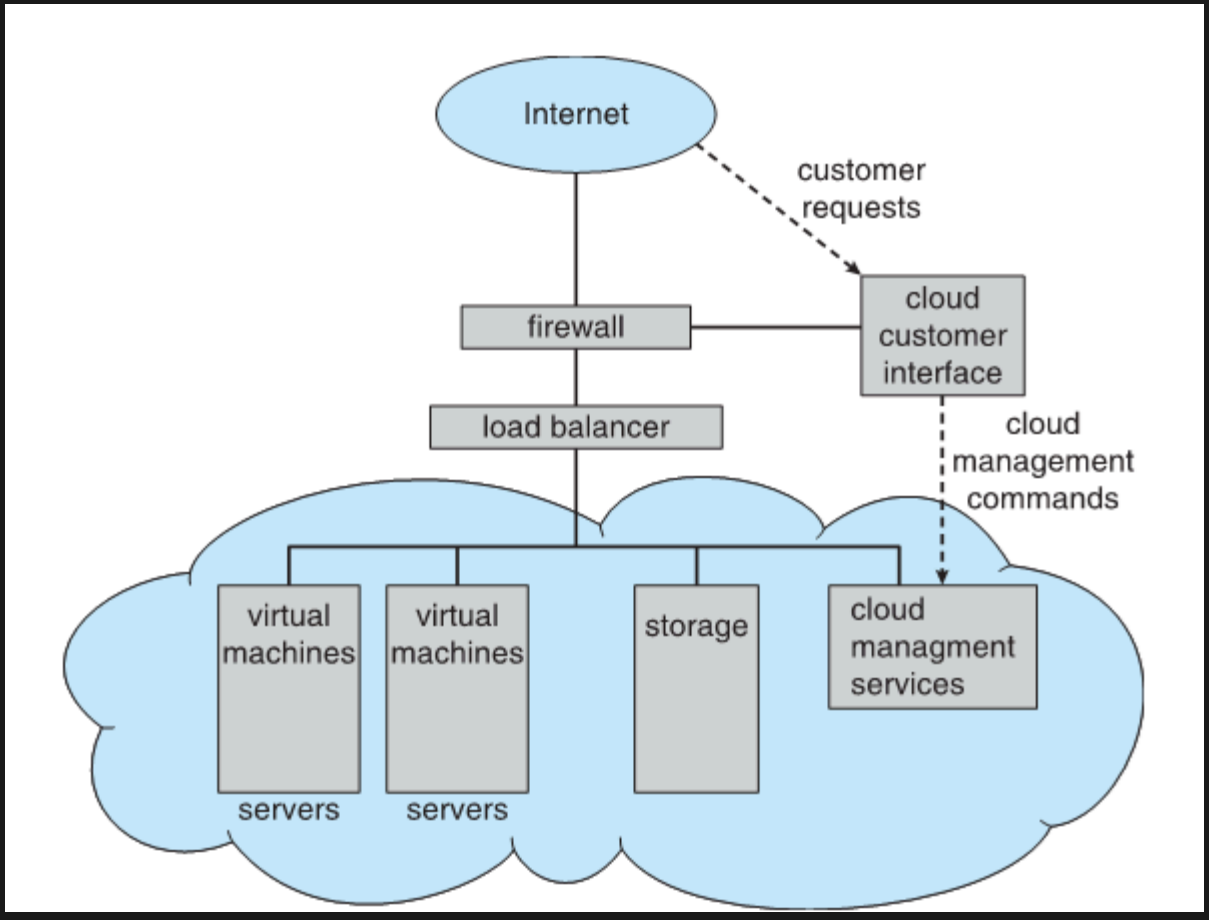
- Creates **virtual versions** of hardware, OS, storage, or networks
- Allows **multiple VMs** on a single physical host
- Improves resource utilization and isolation


Application
Operating System
APP
OS
APP
OS
APP
OS
APP
OS
APP
OS
APP
OS
VMware


processes
kernel
hardware
programming interface
processes
processes
processes
kernel
kernel
kernel
VM1
VM2
VM3
virtual machine manager
hardware
(a)
(b)

# CLOUD COMPUTING

- On-demand access to **compute, storage, and applications** over the internet
- Provides **scalability, pay-as-you-go models, and remote accessibility**
- Types: IaaS, PaaS, SaaS


Internet
customer
requests
firewall
cloud
customer
interface
load balancer
cloud
management
commands
virtual
machines
virtual
machines
storage
cloud
managment
services
servers
servers

# REAL-TIME & EMBEDDED SYSTEMS

- Designed to **respond to events within strict timing constraints**
- Embedded in devices like sensors, controllers, and industrial machines
- Critical in applications like automotive, medical, and avionics systems

# OPERATING SYSTEM SERVICES

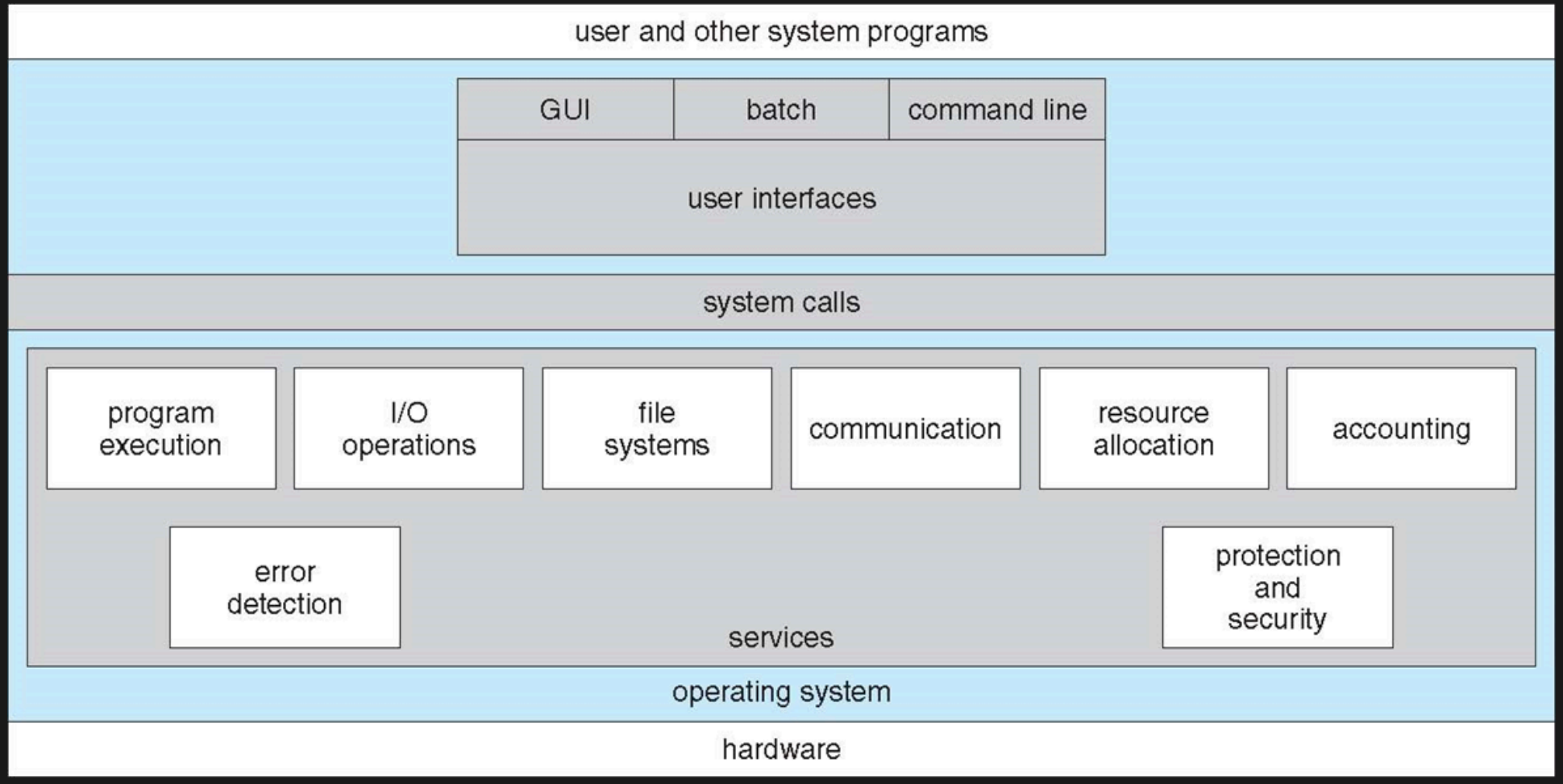
# USER-ORIENTED SERVICES

- **User Interface (UI):** CLI, GUI, or batch
- **Program Execution:** load, run, and terminate programs
- **I/O Operations:** access files or I/O devices
- **File-System Manipulation:** read/write files, create/delete directories, permissions, search, listing
- **Communications:** exchange information via shared memory or message passing
- **Error Detection:** monitor CPU, memory, I/O, and user programs; ensure consistent computing

- **Debugging Facilities:** assist users and programmers

# SYSTEM-ORIENTED SERVICES

- **Resource Allocation:** assign CPU, memory, storage, I/O devices to concurrent users/jobs
- **Accounting:** track user resource usage
- **Protection & Security:**
  - Control access to resources
  - Prevent interference among processes
  - User authentication and protection from external threats


user and other system programs
GUI
batch
command line
user interfaces
system calls
program execution
I/O operations
file systems
communication
resource allocation
accounting
error detection
protection and security
services
operating system
hardware

# SYSTEM CALLS

**Programming Interface to OS Services**

High-level languages like C/C++ provide interfaces to OS services.
Programs usually use an Application Programming Interface instead of calling system calls directly.

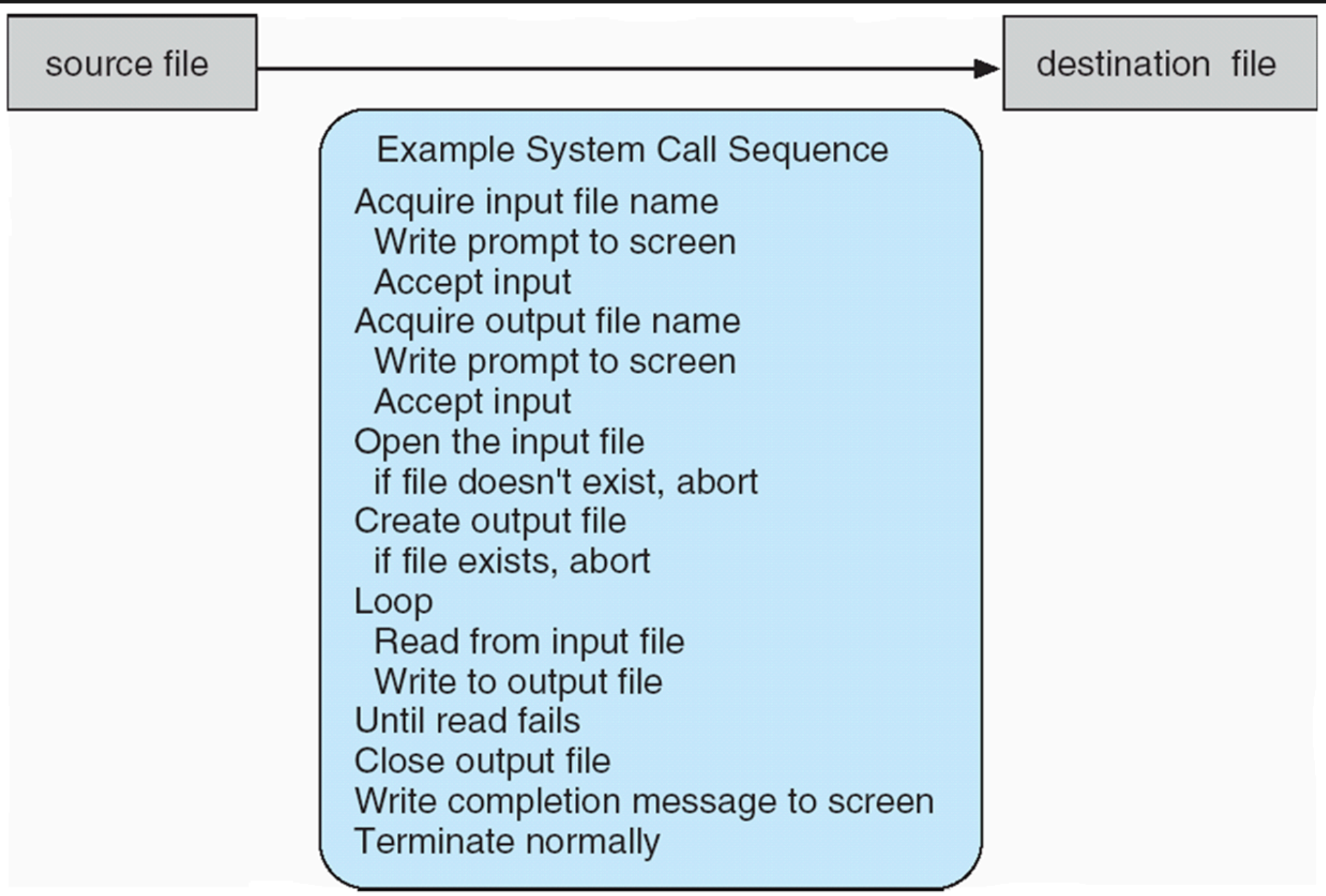
## Access via APIs

APIs offer easier function calls and abstractions, reducing complexity and improving portability across systems.

## Common APIs

Win32 API for Windows systems
POSIX API for UNIX/Linux/Mac OS X
Java API for programs running on the JVM

# EXAMPLE


source file
destination file
Example System Call Sequence
Acquire input file name
Write prompt to screen
Accept input
Acquire output file name
Write prompt to screen
Accept input
Open the input file
if file doesn't exist, abort
Create output file
if file exists, abort
Loop
Read from input file
Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally

# SYSTEM CALL MAPPING

## UNIQUE CALL NUMBERS

Each system call has a specific number used to index a system-call table.

# SYSTEM CALL INTERFACE

## LOOKUP AND EXECUTION

The system-call interface checks the table and invokes the corresponding kernel function.
It returns the status and any output produced by the call.

# PROGRAMMER VIEW

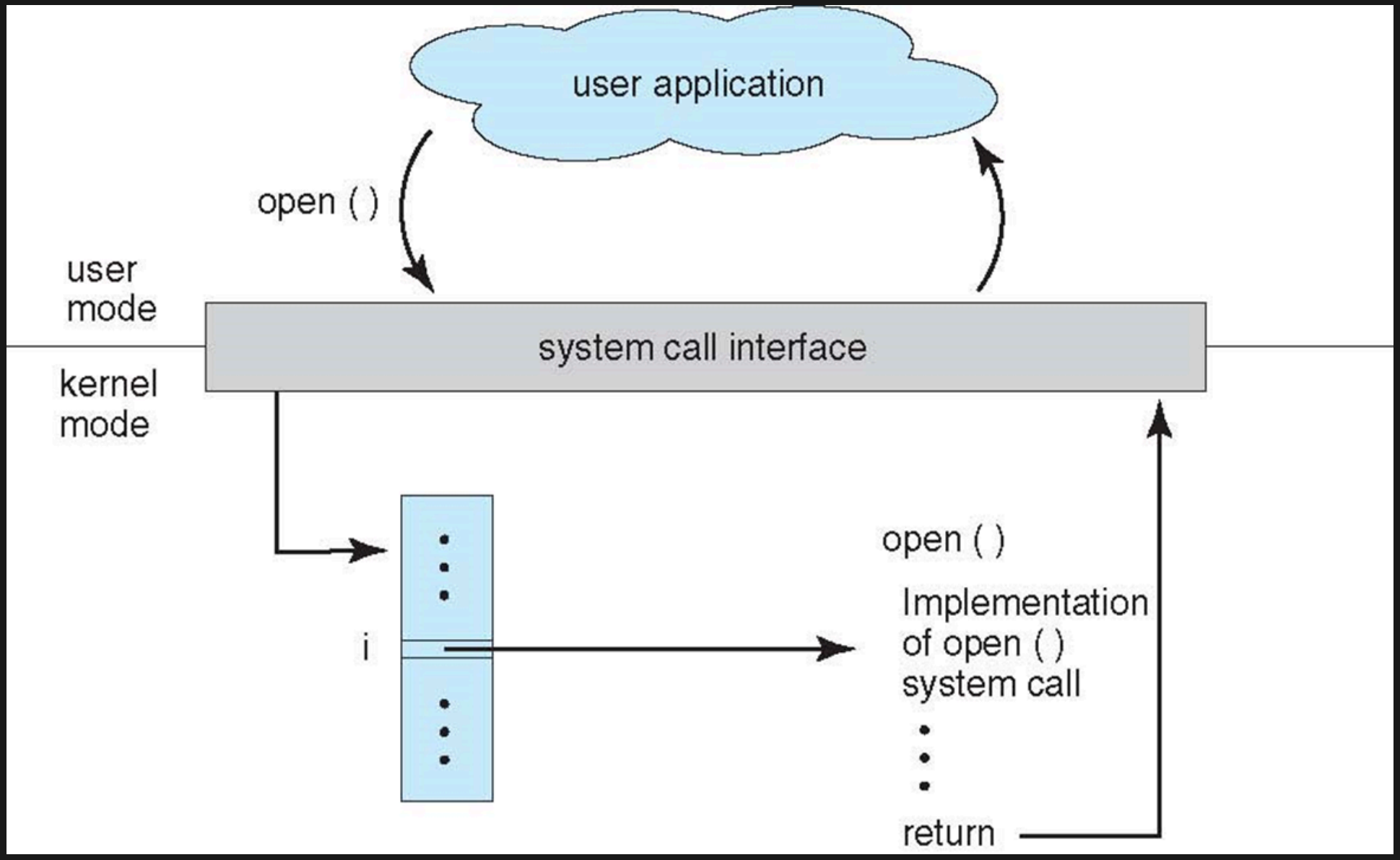
## NO NEED TO KNOW INTERNAL IMPLEMENTATION

The programmer only needs to follow the API and understand the OS behavior.
Internal mechanics of the system call remain hidden.

# API ABSTRACTION

## RUN-TIME SUPPORT LIBRARY

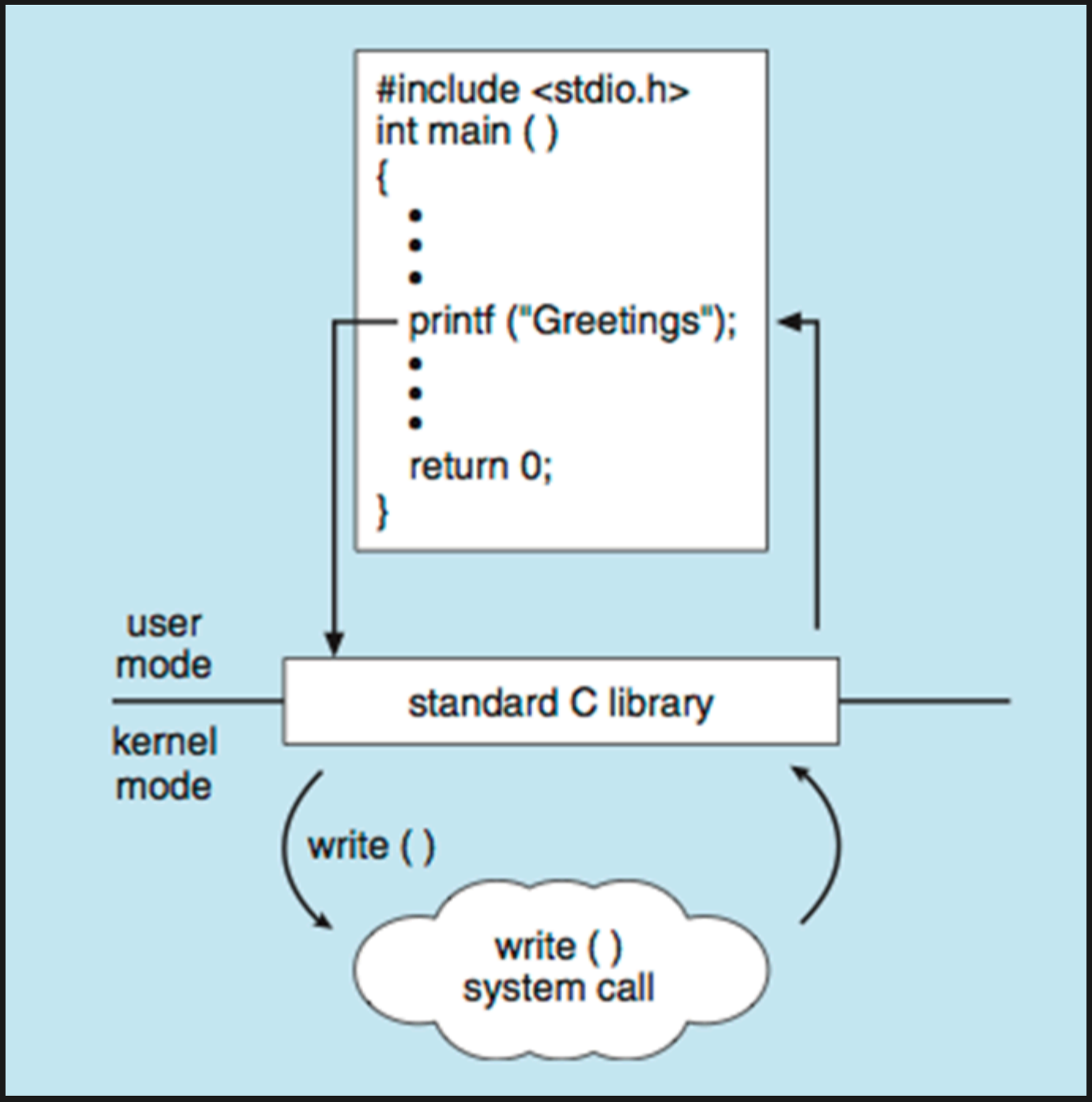
Most interaction with the OS is managed by a run-time support library that comes with the compiler.


user application
open ( )
user
mode
system call interface
kernel
mode
i
open ( )
Implementation
of open ( )
system call
return

# Examples of Windows and Unix System Calls

| | Windows | Unix |
|---|---|---|
| Process Control | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject() | fork()<br>exit()<br>wait() |
| File Manipulation | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| Device Manipulation | SetConsoleMode()<br>ReadConsole()<br>WriteConsole() | ioctl()<br>read()<br>write() |
| Information Maintenance | GetCurrentProcessID()<br>SetTimer()<br>Sleep() | getpid()<br>alarm()<br>sleep() |
| Communication | CreatePipe()<br>CreateFileMapping()<br>MapViewOfFile() | pipe()<br>shmget()<br>mmap() |
| Protection | SetFileSecurity()<br>InitlializeSecurityDescriptor()<br>SetSecurityDescriptorGroup() | chmod()<br>umask()<br>chown() |

C Program Invoking printf()


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

# OPERATING SYSTEM STRUCTURE

A general-purpose operating system is a very large program.

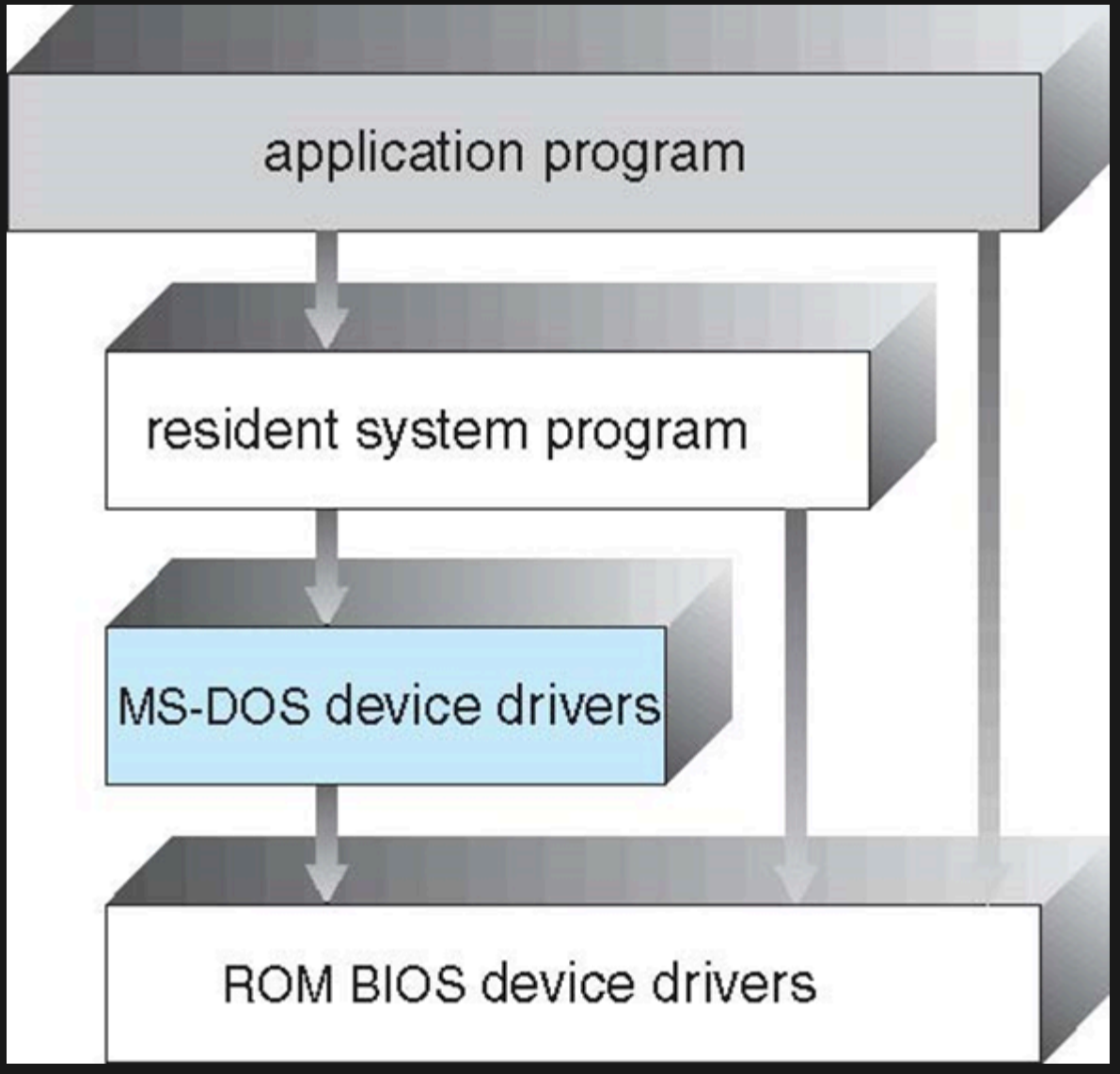
Different structural approaches include:

- Simple structure (e.g., MS-DOS)
- More complex monolithic structure (e.g., UNIX)
- Layered design providing abstraction
- Microkernel design (e.g., Mach)

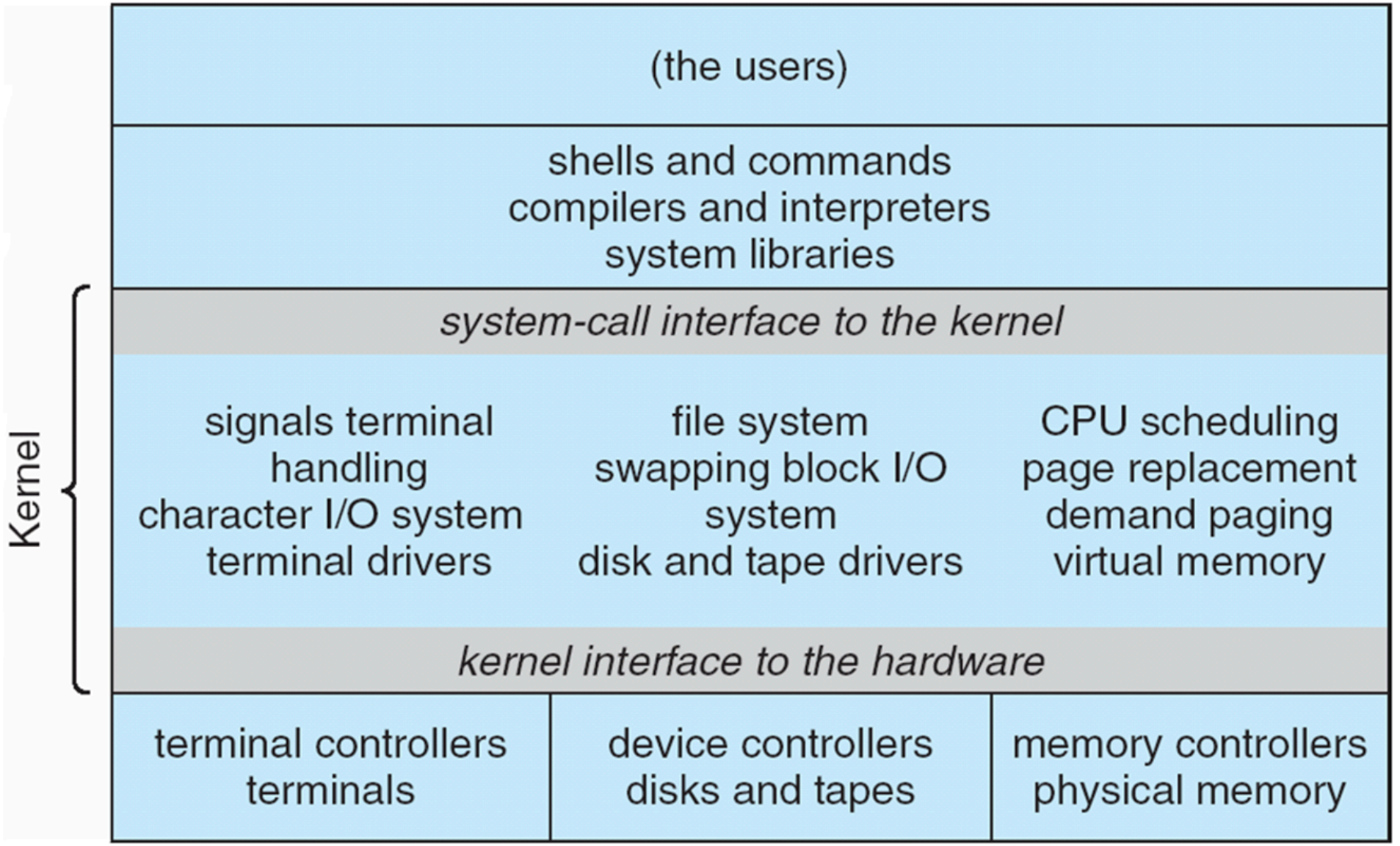
**MS-DOS Structure Design Goal** MS-DOS was created to offer maximum functionality using minimal space.

**Lack of Modularity Organization** It was not divided into separate, well-defined modules.

**Interface Characteristics Poor Separation** Although some structure exists, the interfaces and functionality levels are not clearly separated.


application program
resident system program
MS-DOS device drivers
ROM BIOS device drivers

# LINUX SYSTEM STRUCTURE / ARCHITECTURE


(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
Kernel

# GARY KILDALL

## EARLY BACKGROUND

Gary Kildall was an American computer scientist and pioneer of personal computing.
He created CP/M, the first widely adopted OS for microcomputers.

# CP/M AND INNOVATION

## BREAKTHROUGH

CP/M provided a standard OS interface for early 8-bit machines.
It inspired future OS designs, including MS-DOS.

# IBM VISITS DIGITAL RESEARCH

## MISSED MEETING

In 1980, IBM approached Digital Research to license CP/M for its upcoming PC.
A meeting failed due to scheduling and contract disagreements.

# MICROSOFT STEPS IN

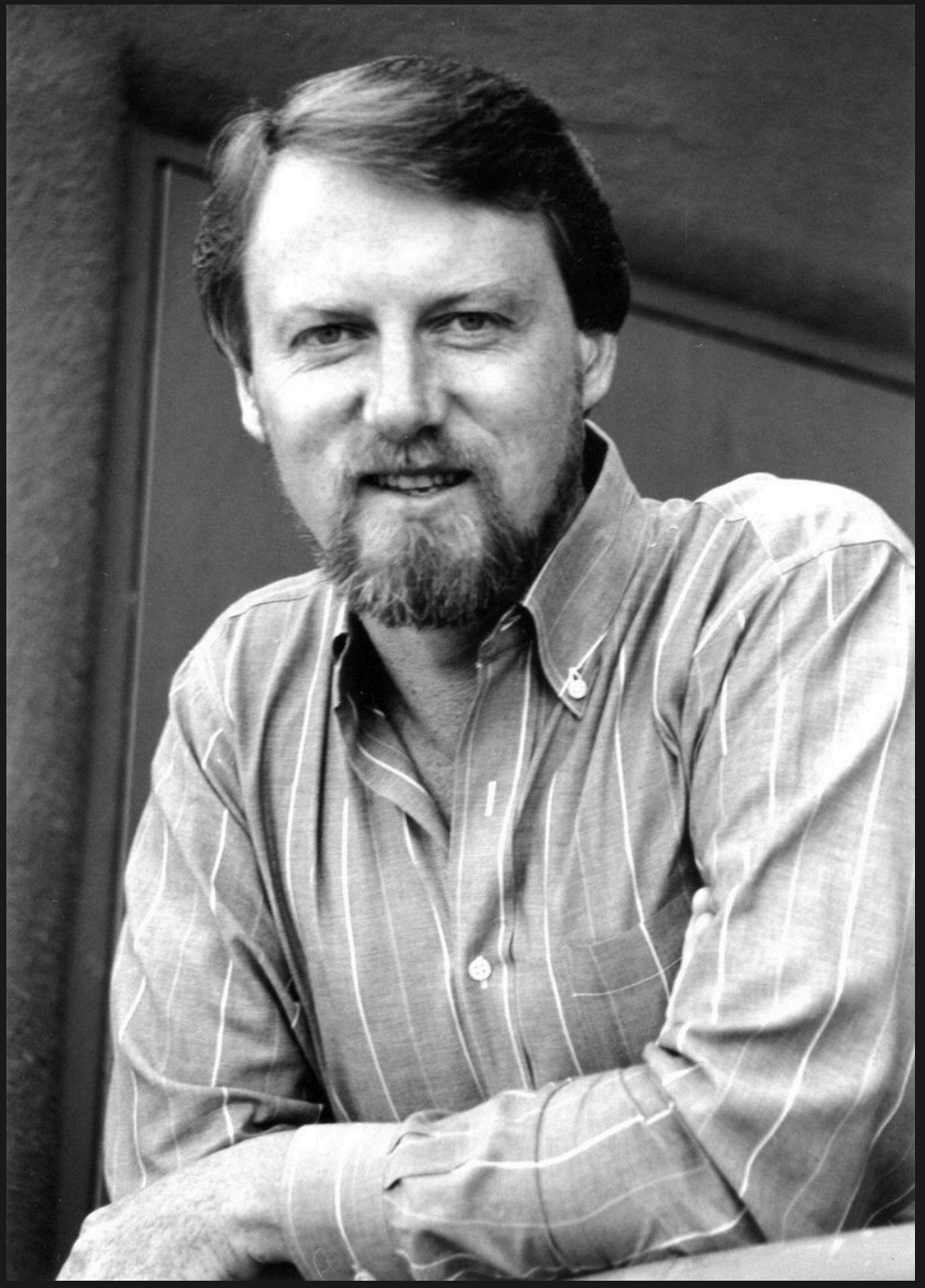
## BILL GATES' OPPORTUNITY

With no agreement from Digital Research, IBM turned to Microsoft.
Microsoft acquired QDOS, modified it, and delivered what became MS-DOS.

# WHY I MENTIONED

Kildall is often discussed when explaining MS-DOS because:

- CP/M was its inspiration
- A single failed business deal changed the course of computing history




PINBAL

# WHAT IS A PROCESS?

- A **process** is a program that is currently being executed.
- Its execution moves **step-by-step in a sequential flow**.

# PROGRAM VS PROCESS

**Program**

- Passive
- Stored on disk as an executable file

**Process**

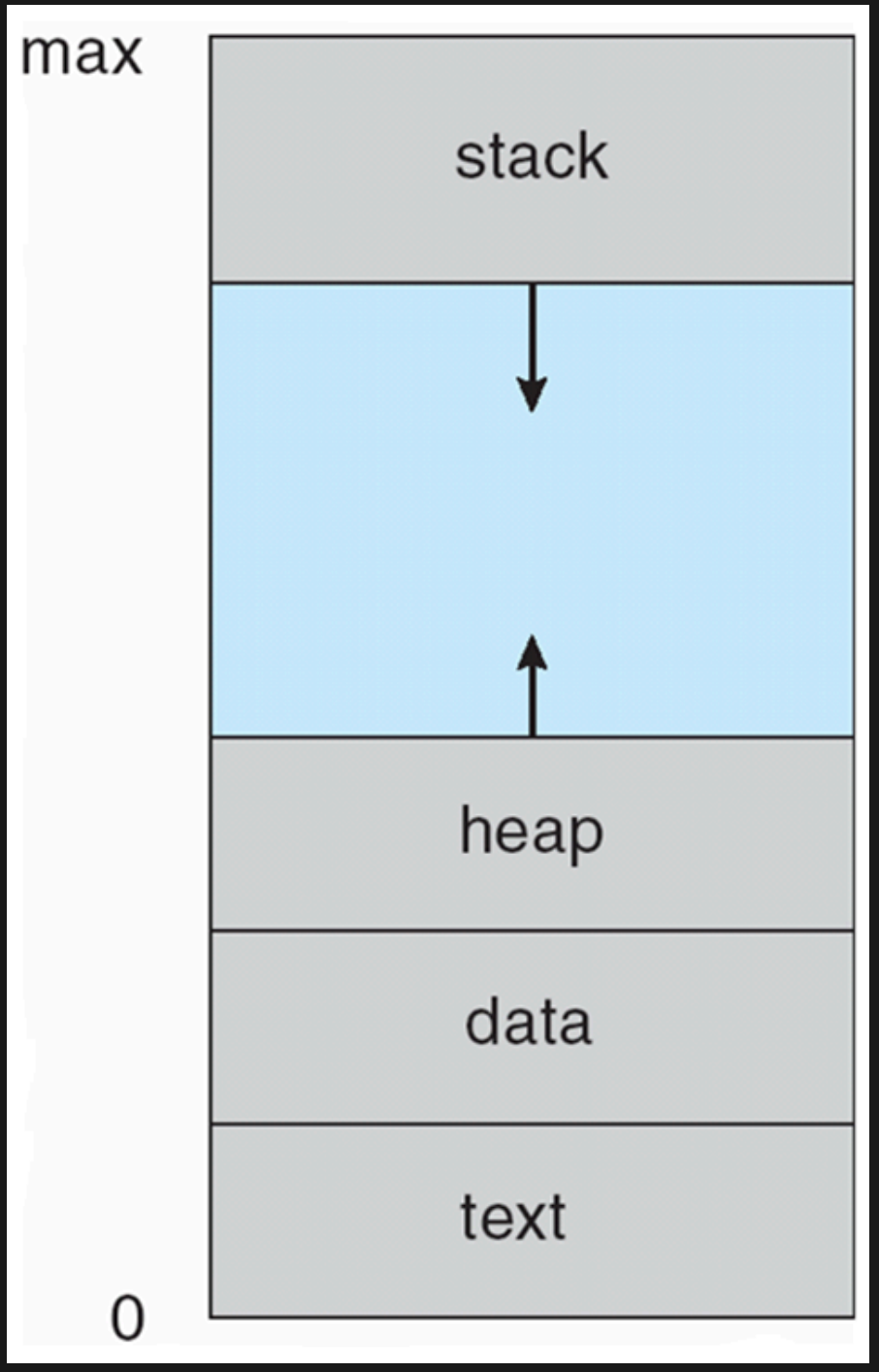
- Active in memory
- Has registers, stack, resources, etc.

# WHEN DOES A PROGRAM BECOME A PROCESS?

- When the executable file is **loaded into memory**.
- Operating system creates a process structure for it.

# MULTIPLE PROCESSES FROM ONE PROGRAM

- The same program may run **multiple times**.
- Each execution is a **different process**.
- Example: Many users running the same application independently.

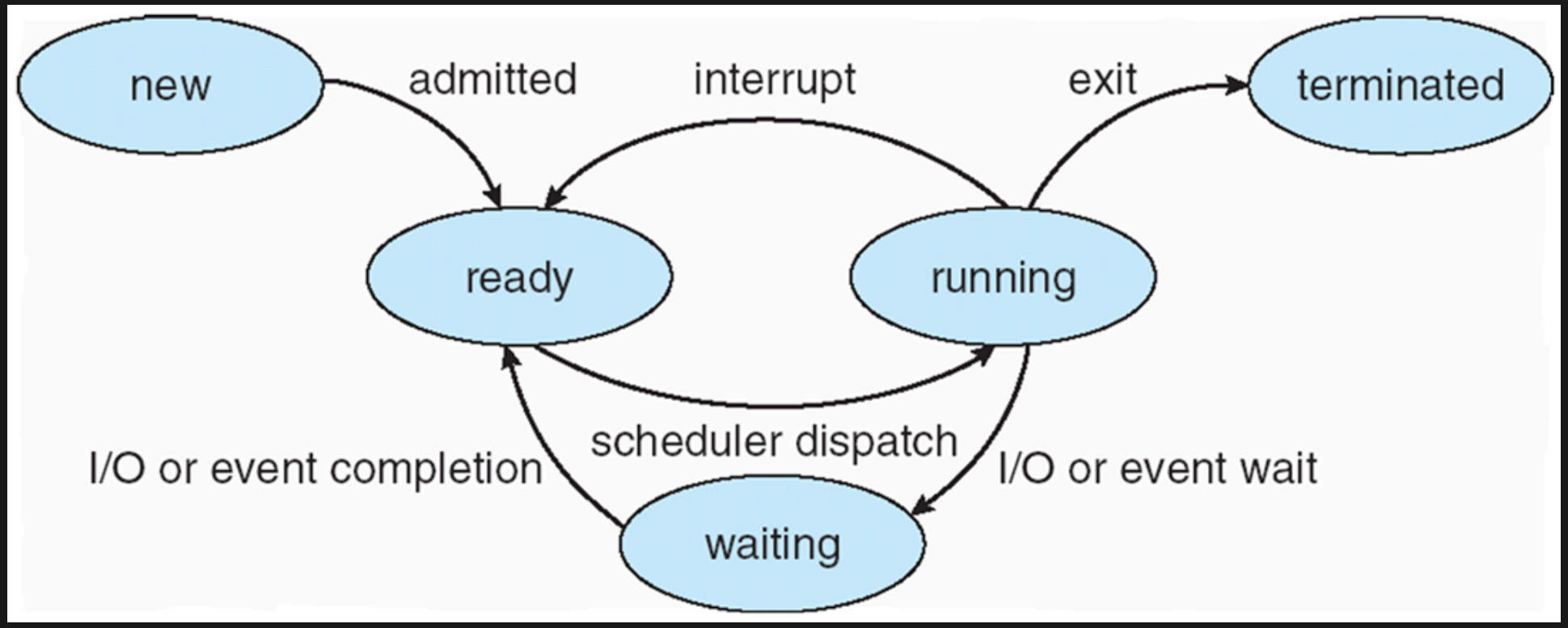

max
stack
heap
data
text
0

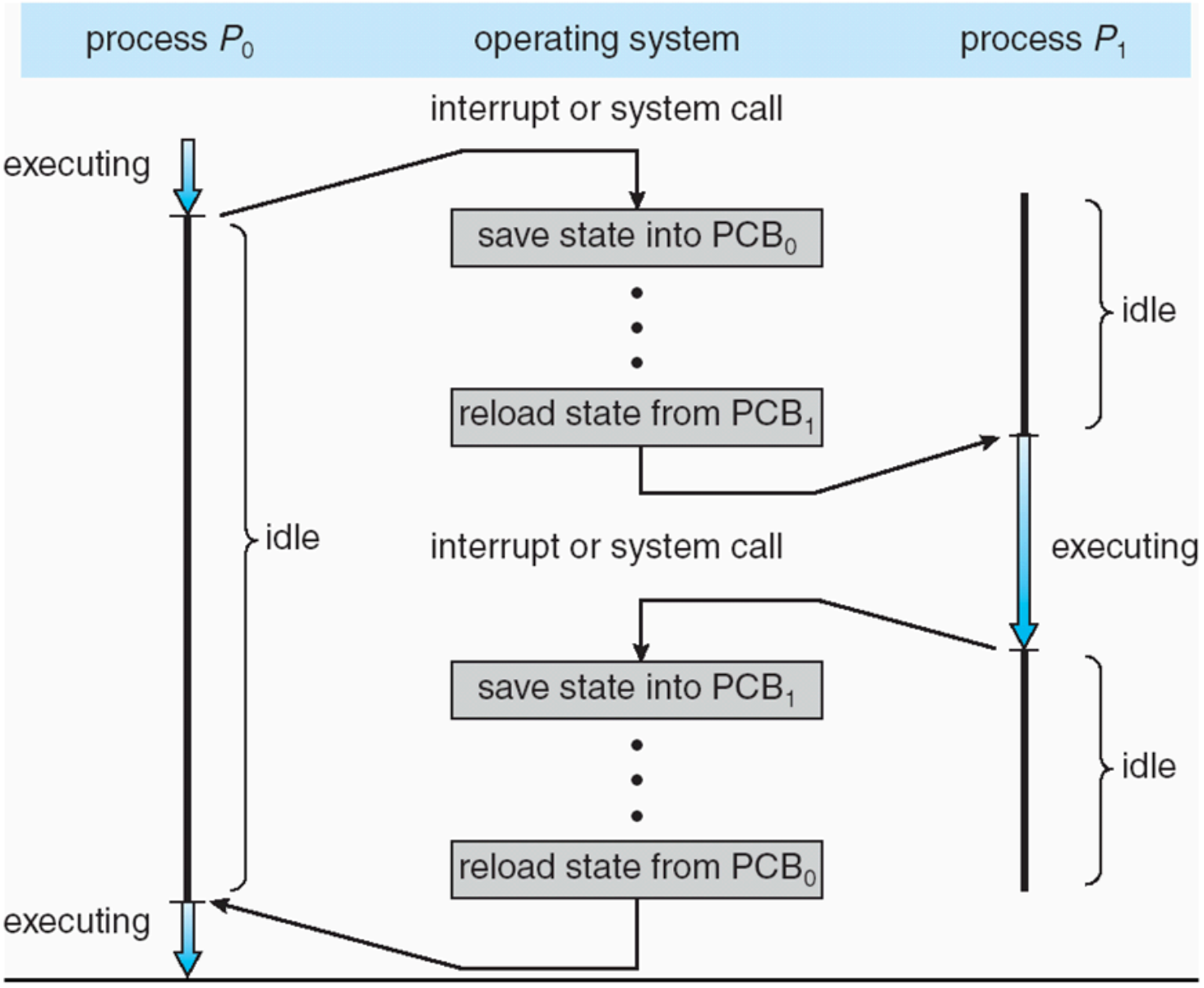
# PROCESS STATES

- As a process runs, its state changes.
- Typical states used by operating systems.

# PROCESS STATES

- **New:** Process is being created
- **Running:** Instructions are being executed
- **Waiting:** Waiting for an event
- **Ready:** Waiting for CPU assignment
- **Terminated:** Execution completed

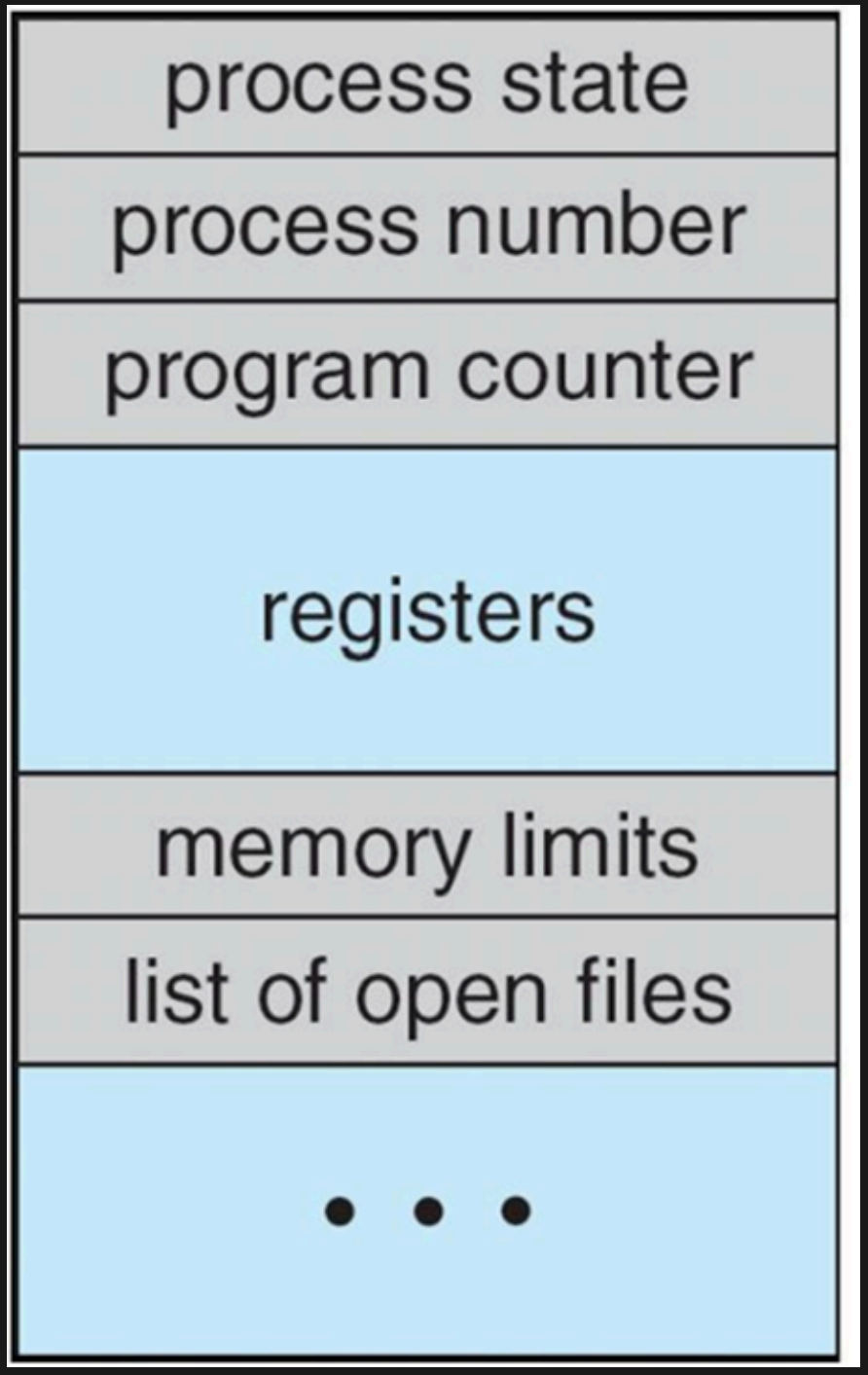

new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting


process $P_0$
operating system
process $P_1$
interrupt or system call
executing
save state into $PCB_0$
reload state from $PCB_1$
idle
idle
interrupt or system call
executing
save state into $PCB_1$
reload state from $PCB_0$
idle
executing

Process Control Block (PCB) contains  Information associated with each process (also called task control block)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process centric registers
- CPU scheduling information- priorities, scheduling queue pointers

- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

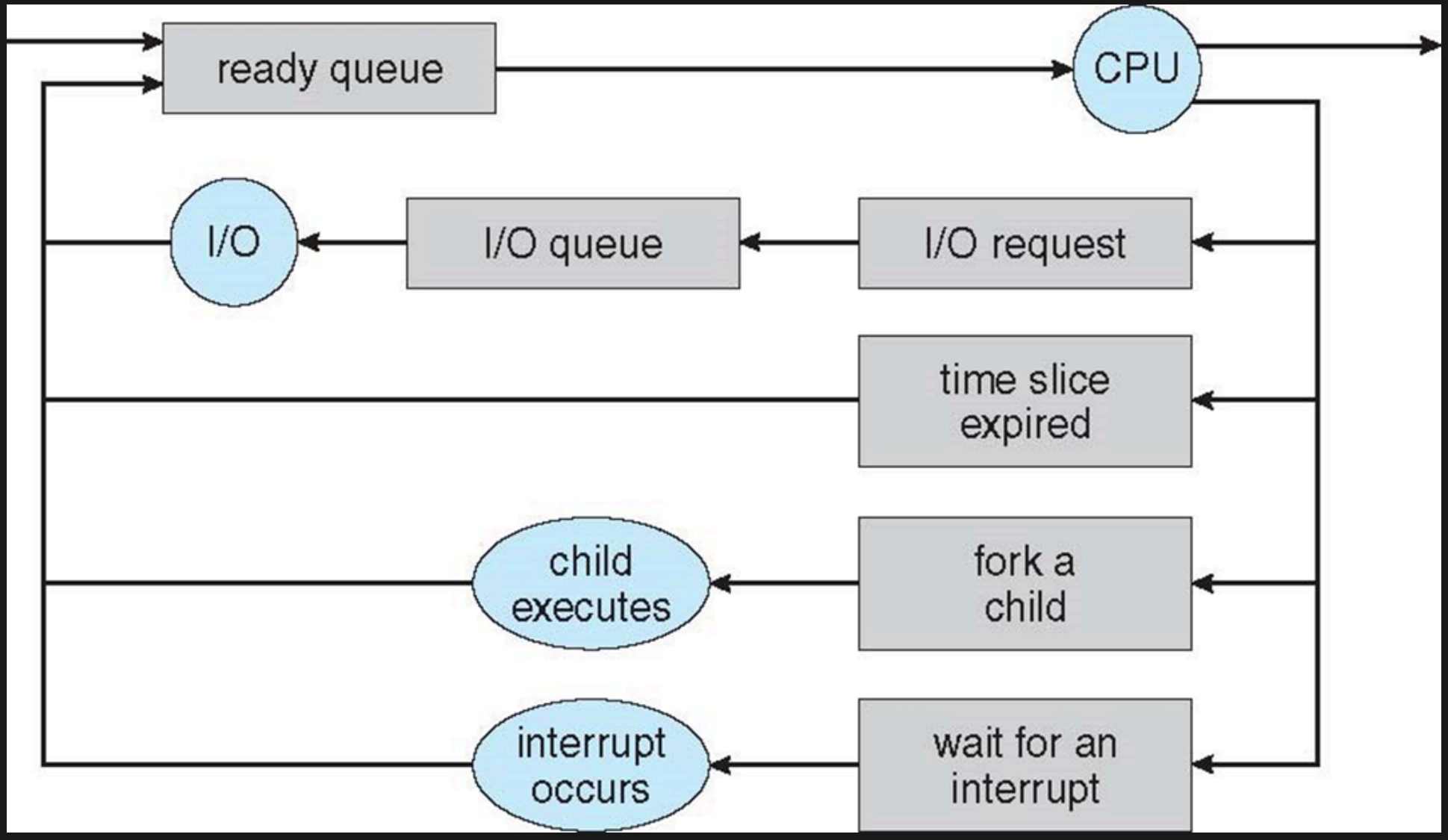

process state
process number
program counter
registers
memory limits
list of open files
• • •

# CPU SCHEDULING BASICS

- Aim: maximize CPU usage by fast process switching
- Scheduler chooses the next process for CPU
- Processes move between different queues

# SCHEDULING QUEUES

- **Job queue:** all system processes
- **Ready queue:** processes in memory waiting for CPU
- **Device queues:** processes waiting for I/O


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

# SHORT-TERM SCHEDULER

- Chooses the next process for the CPU
- Allocates processor time
- Often the only scheduler in simple systems
- Runs very frequently (milliseconds), must be fast

# LONG-TERM SCHEDULER

- Decides which processes enter the ready queue
- Runs much less often (seconds or minutes)
- Can afford slower decision-making

# TYPES OF PROCESSES

- I/O-bound: spends more time performing input/output, many short CPU bursts
- CPU-bound: spends most time computing, few long CPU burst.

# CPU SCHEDULING METRICS

# TURNAROUND TIME

Time a process takes from arrival to completion.

**Formula:**

```
Turnaround time = Completion time - Arrival time
```

# WAITING TIME

Time a process waits in the ready queue before CPU execution.

**Formula:**

`Waiting time = Turnaround time – Burst time`

# AVERAGE WAITING TIME

Average time all processes spend waiting in the ready queue.

**Formula:**

`Average waiting time = (Total waiting time of all processes) / (Number of processes)`

# RESPONSE TIME

Time from process arrival until its first CPU execution.

**Formula:**

```
Response time = Start time – Arrival time
```

# THROUGHPUT

Number of processes completed per unit time.

**Formula:**

```
Throughput = Number of processes completed / Total execution time
```

# Q1

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 5 |
| P2 | 2 | 3 |
| P3 | 4 | 7 |
| P4 | 6 | 2 |

# Q2

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 1 | 4 |
| P2 | 3 | 6 |
| P3 | 5 | 2 |
| P4 | 7 | 8 |

# Q3

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 9 |
| P2 | 1 | 5 |
| P3 | 3 | 1 |
| P4 | 4 | 4 |

# Q4

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 2 | 6 |
| P2 | 4 | 3 |
| P3 | 5 | 9 |
| P4 | 8 | 2 |

# Q5

| Process | Arrival Time | Burst Time |
|---|---|---|
| P1 | 0 | 8 |
| P2 | 1 | 2 |
| P3 | 3 | 5 |
| P4 | 6 | 4 |

# DEADLOCK

# MULTIPLE PROCESSES

Several processes run concurrently and compete for limited resources.

# RESOURCE REQUESTS

A process asks for resources.
If unavailable, it moves to a **wait state**.

# WAIT STATE ISSUE

A waiting process may stay stuck if the resources it needs are held by other waiting processes.

# RESULT

Processes may never resume — a situation that can lead to **deadlock**.

# SYSTEM MODEL

# RESOURCE REQUEST RULE

A process must **request** a resource before using it and **release** it after use.

# RESOURCE LIMITS

A process cannot request more instances of a resource than the system actually has.

# EXAMPLE

If the system has **2 printers**, no process can request **3 printers**.

# DEADLOCK CHARACTERIZATION

Deadlock occurs when processes are stuck waiting for resources held by each other, with no progress possible.

# FOUR NECESSARY CONDITIONS

1. **Mutual Exclusion** – Resources are non-shareable.
2. **Hold and Wait** – A process holds one resource and waits for another.
3. **No Preemption** – Resources cannot be forcibly taken away.
4. **Circular Wait** – A circular chain of processes exists, each waiting for a resource held by the next.

# RESOURCE ALLOCATION GRAPH

# DEFINITION

A deadlock can be represented using a **directed system resource-allocation graph**.

# GRAPH STRUCTURE

The graph has:

- **Vertices** (V)
- **Edges** (E)

# VERTEX TYPES

Vertices are divided into two sets:

- **P = {P1, P2, …, Pn}** → all active processes
- **R = {R1, R2, …, Rm}** → all resource types

# REQUEST EDGE

**Pi → Rj**

Process **Pi** has requested resource **Rj** and is waiting for it.

# ASSIGNMENT EDGE

**Rj → Pi**

An instance of resource **Rj** is allocated to process **Pi**.

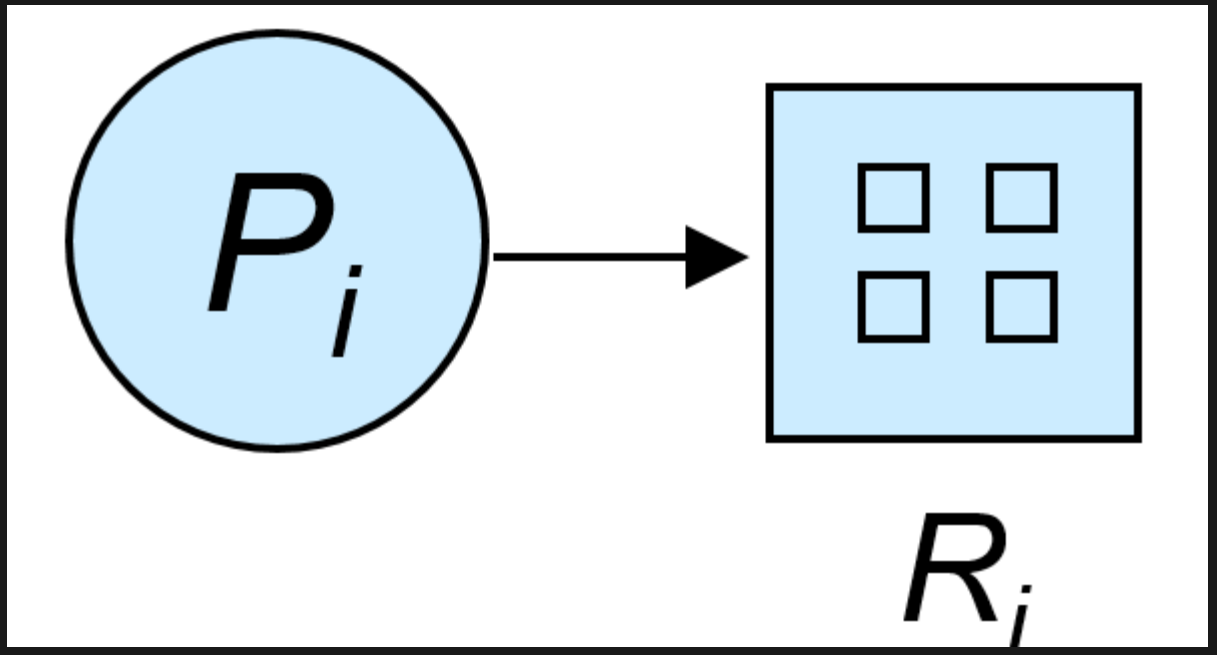
# EDGE TYPES

- **Pi → Rj** → *Request edge*
- **Rj → Pi** → *Assignment edge*

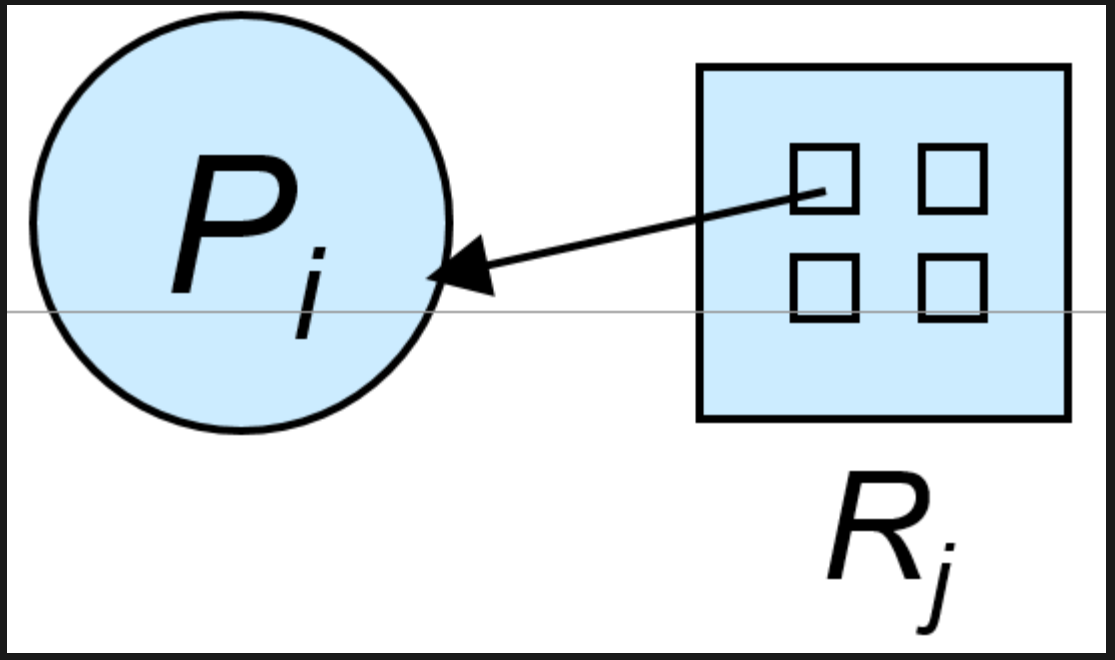

Process

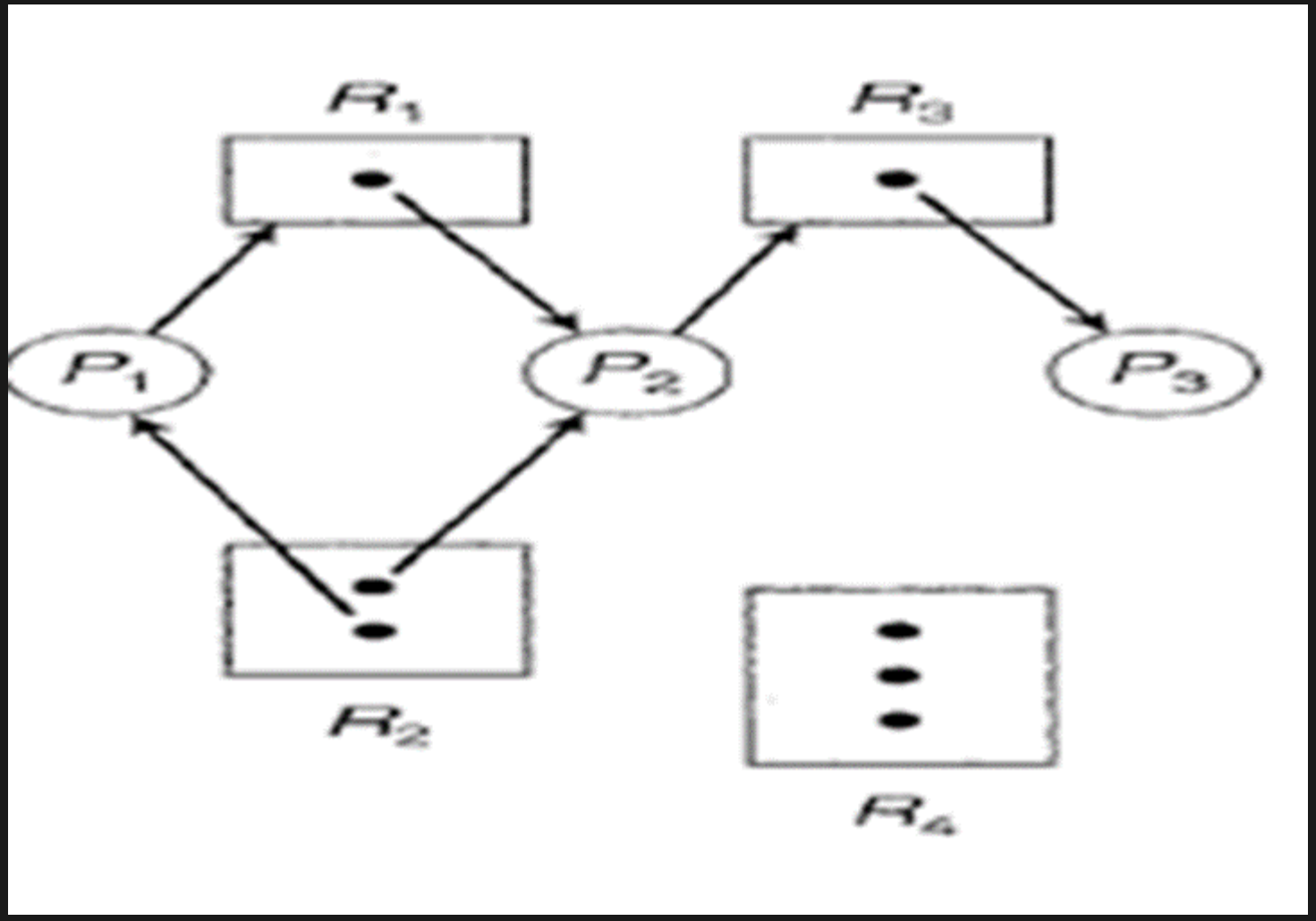
Resource Type with 4 instances 

Pi requests instance of Rj

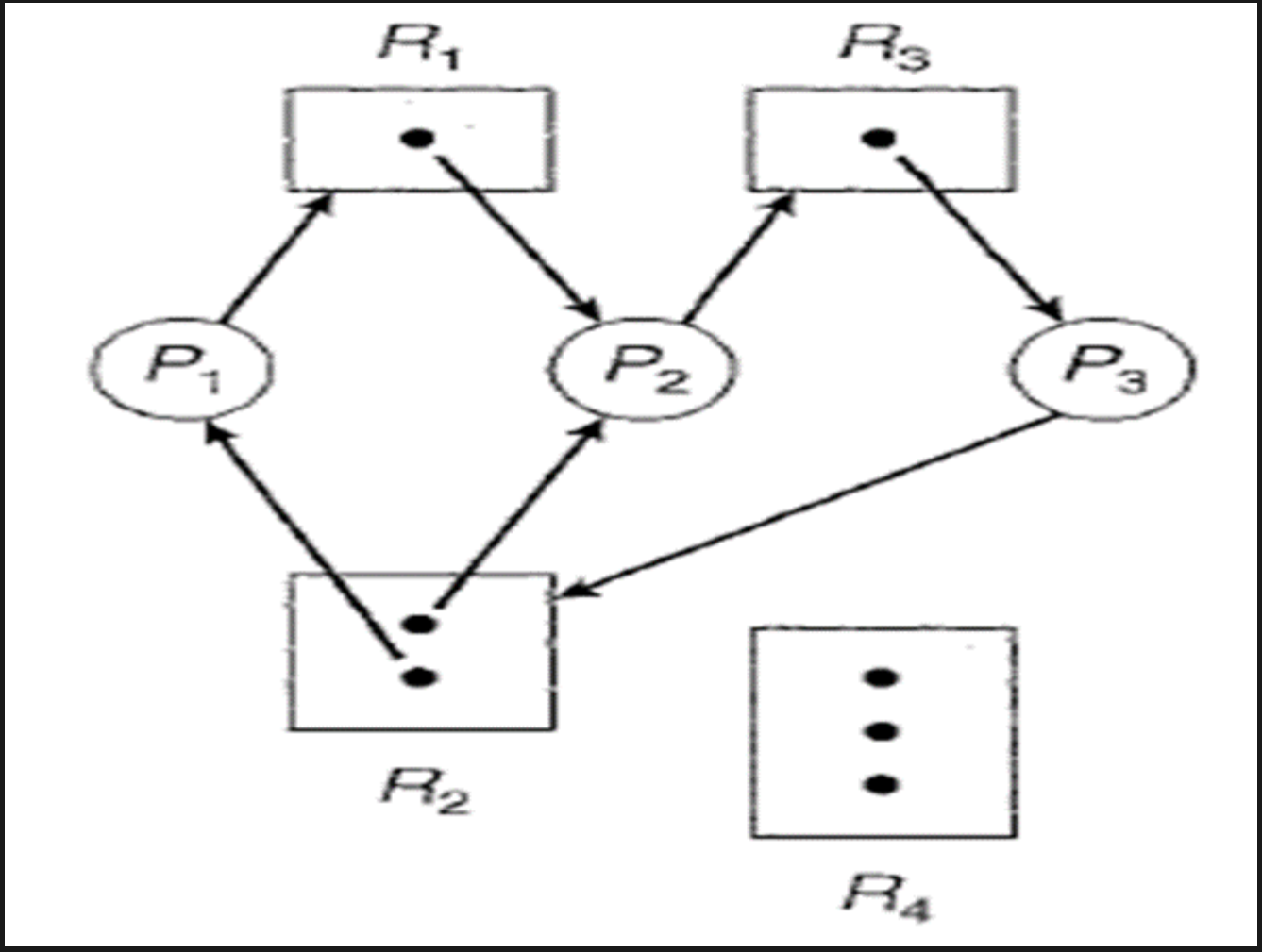


Pi is holding an instance of Rj

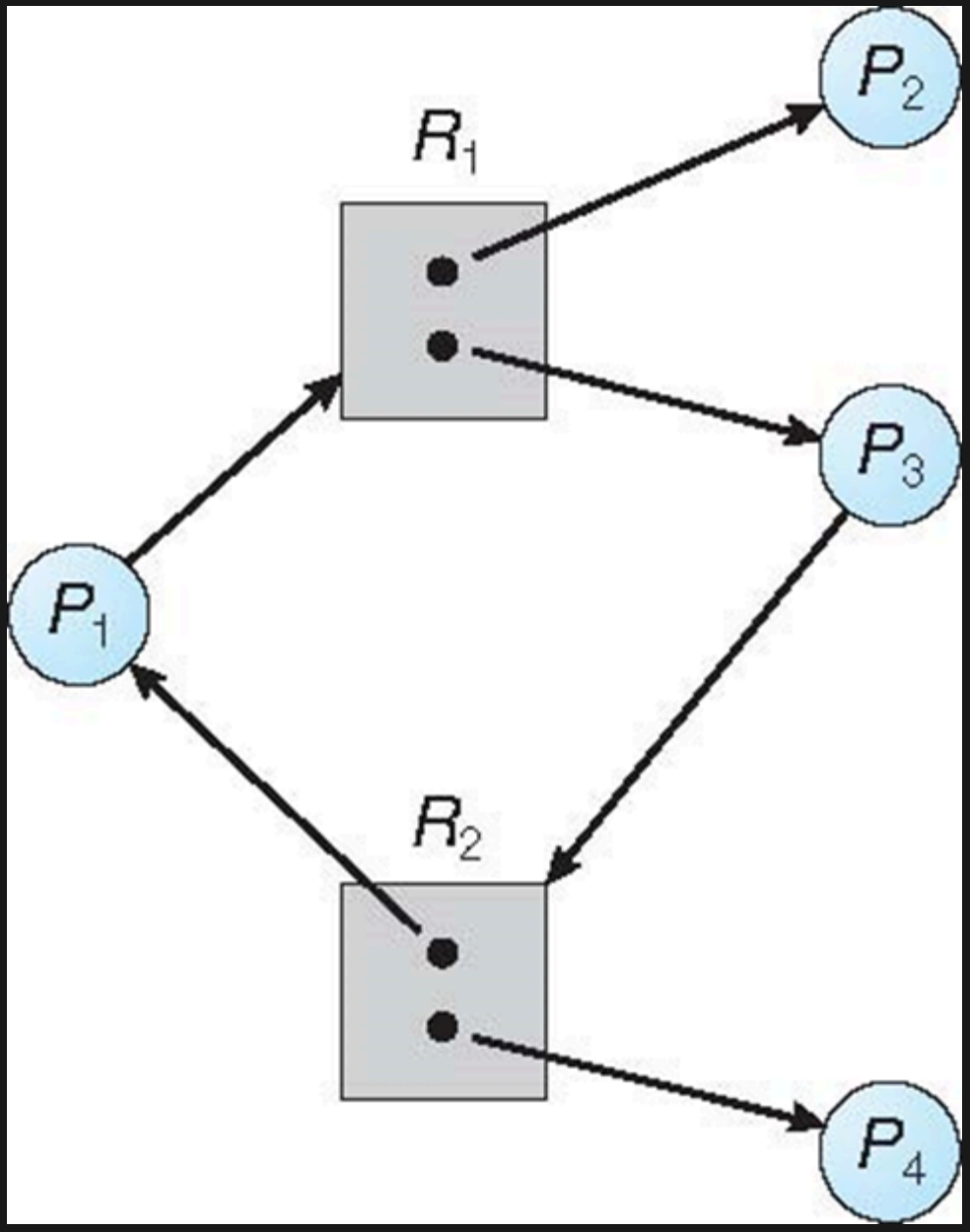



R1
R3
P1
P2
P3
R2
R4

# RAG WITH DEADLOCK



# Graph With A Cycle But No Deadlock



# METHODS FOR HANDLING DEADLOCKS

# 1. PREVENTION / AVOIDANCE

Use protocols that ensure the system **never** enters a deadlock state. (before occurance)

# 2. DETECTION & RECOVERY

Allow deadlocks to occur, then **detect** and **recover** from them. (after occurance)

# 3. IGNORE DEADLOCKS

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

# DEADLOCK PREVENTION

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
    - Low resource utilization; starvation possible

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

- Preempted resources are added to the list of resources for which the process is waiting
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# DEADLOCK AVOIDANCE

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

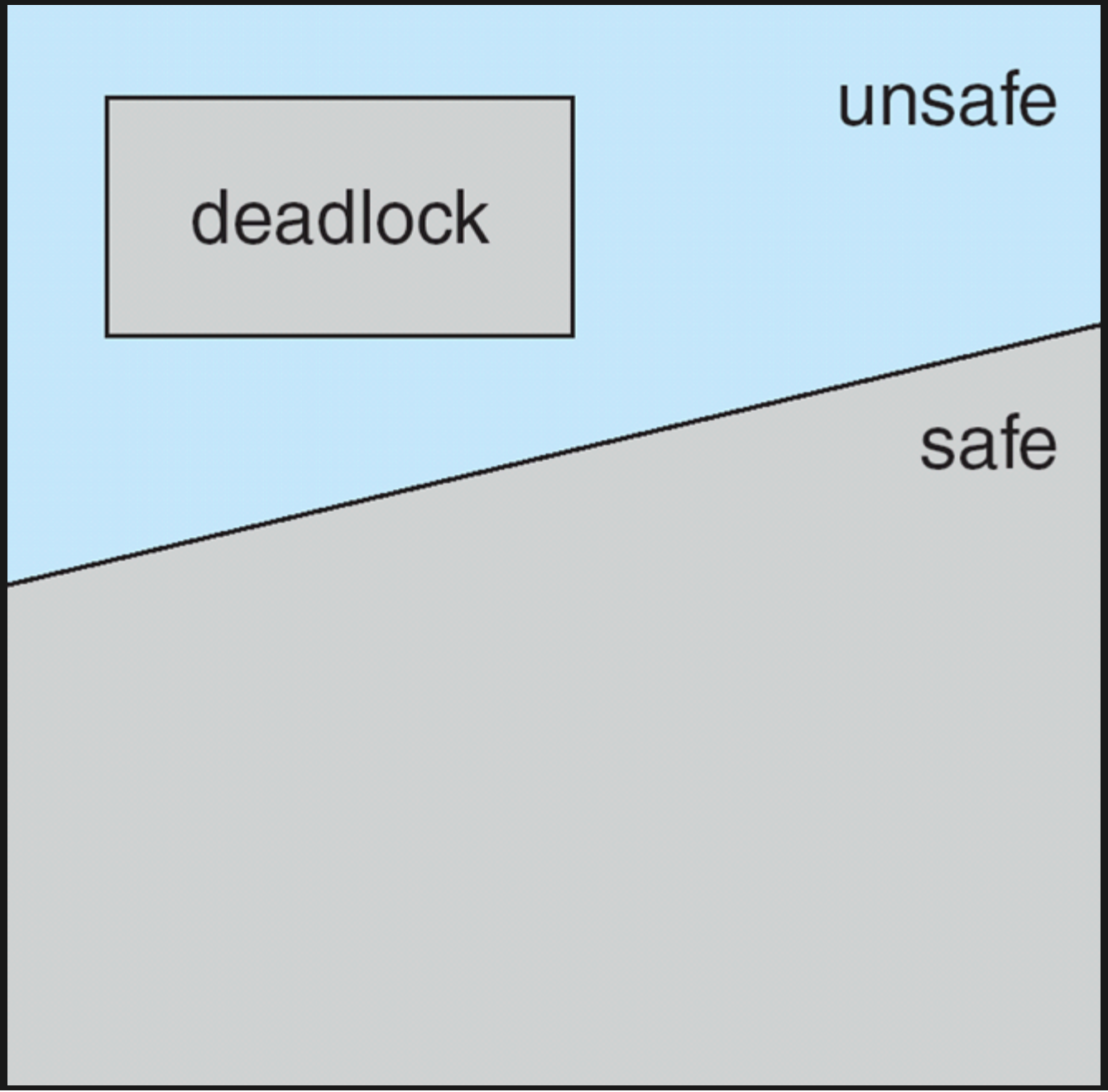
- For example, in a system with one tape drive and one printer, we might be told that process P will request first the tape drive, and later the printer, before releasing both resources. Process Q, on the other hand, will request first the printer, and then the tape drive.

# SAFE STATE

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state

- If a system is in safe state -> no deadlocks
- If a system is in unsafe state -> possibility of deadlock
- Avoidance -> ensure that a system will never enter an unsafe state.

# SAFE, UNSAFE, DEADLOCK STATE



## SINGLE INSTANCE OF A RESOURCE TYPE

- Use a resource-allocation graph

## MULTIPLE INSTANCES OF A RESOURCE TYPE

- Use the banker’s algorithm

# BANKER'S ALGORITHM

## Q1 Total resources = (A=11, B=6, C=9, D=7)

| Process | Allocated | | | Max Need | | |
|---|---|---|---|---|---|---|
| | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 5 | 3 |
| P1 | 2 | 0 | 0 | 3 | 2 | 2 |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 |

# Q2 Available = (A = 3, B = 2, C = 1, D = 2)

| Process | Allocated | | | | Max Need | | | |
|---|---|---|---|---|---|---|---|---|
| | A | B | C | D | A | B | C | D |
| P0 | 1 | 0 | 2 | 1 | 3 | 2 | 2 | 2 |
| P1 | 2 | 1 | 0 | 2 | 4 | 3 | 1 | 3 |
| P2 | 3 | 0 | 2 | 1 | 6 | 1 | 4 | 3 |
| P3 | 2 | 2 | 1 | 0 | 3 | 4 | 3 | 1 |
| P4 | 0 | 1 | 3 | 1 | 4 | 2 | 5 | 2 |

Refer https://os.surajgowda.in/scheduling-algo to verify answers