# Systems and Networking I

Applied Computer Science and Artificial Intelligence

2023-2024

SAPIENZA
UNIVERSITÀ DI ROMA

Gabriele Tolomei

Dipartimento di Informatica

Sapienza Università di Roma

tolomei@di.uniroma1.it

# Who Am I?

# Who Am I?

UniPI

# Who Am I?

UniPI

UniVE

# Who Am I?

UniPI

UniVE

Yahoo! Labs

# Who Am I?

UniPI

UniVE

Yahoo! Labs

UniPD

# Who Am I?

UniPI

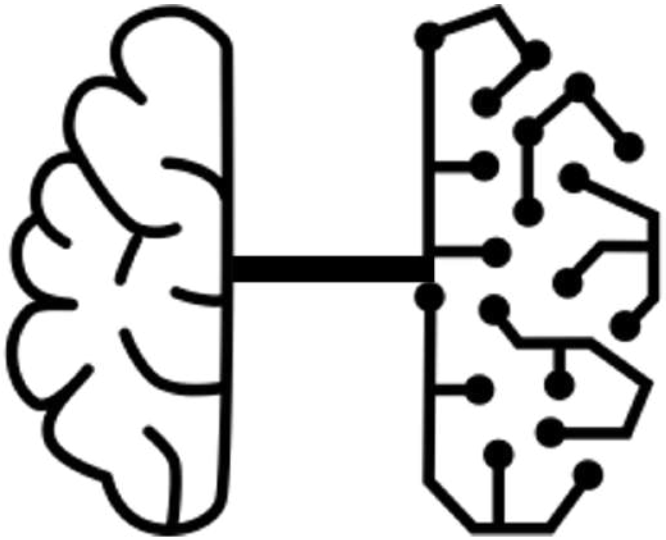
UniVE

Yahoo! Labs

UniPD

Sapienza

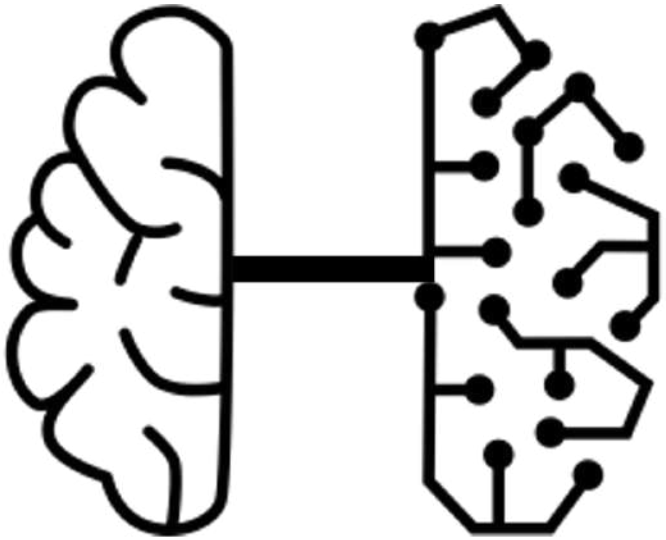
# My Research Interests



HERCOLE Lab

# My Research Interests

Human-Explainable

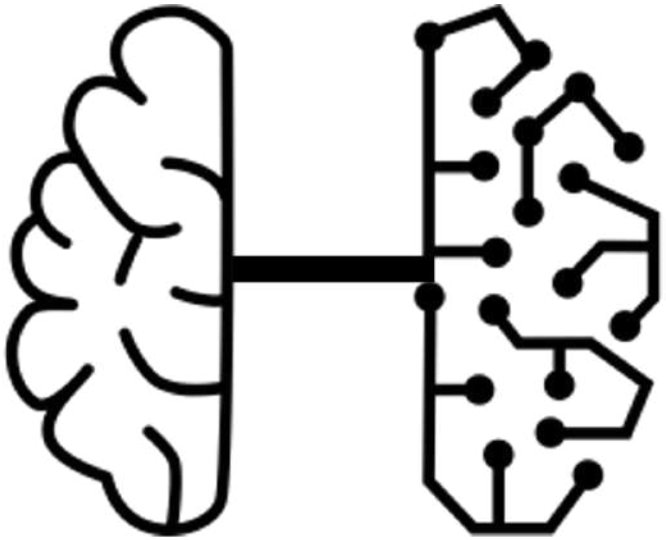


HERCOLE Lab

# My Research Interests

Robust

Human-Explainable

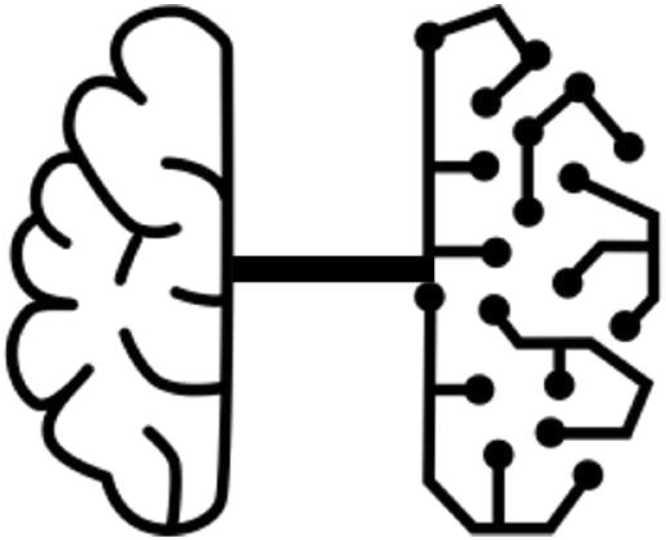


HERCOLE Lab

# My Research Interests

Robust

Human-Explainable



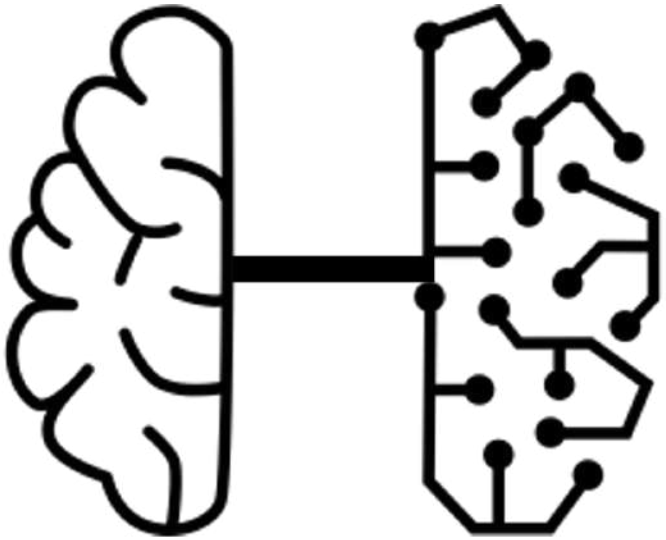
COllaborative

HERCOLE Lab

# My Research Interests

Robust

Human-Explainable



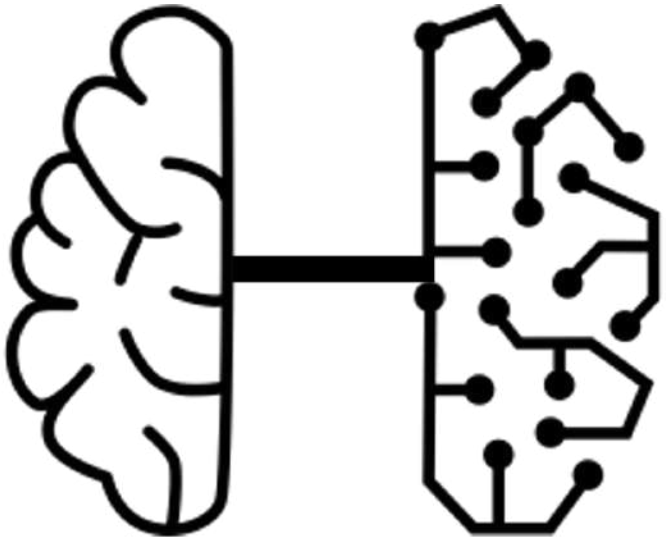
COllaborative

HERCOLE Lab

LEarning

# My Research Interests

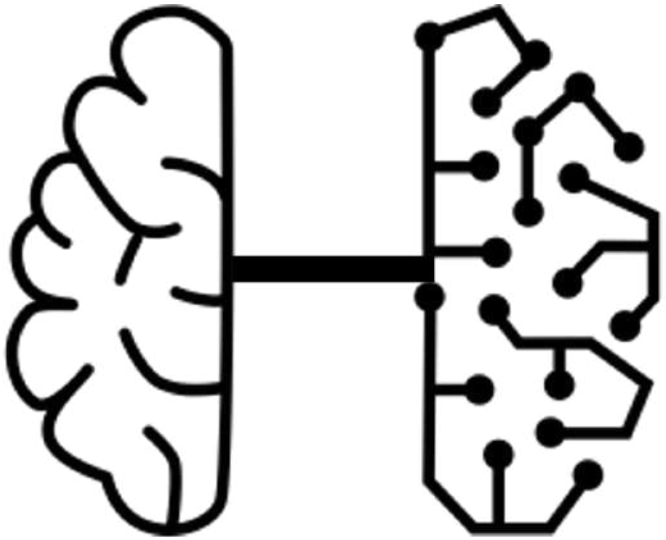
Sounds cool?



HERCOLE Lab

# My Research Interests

Sounds cool?

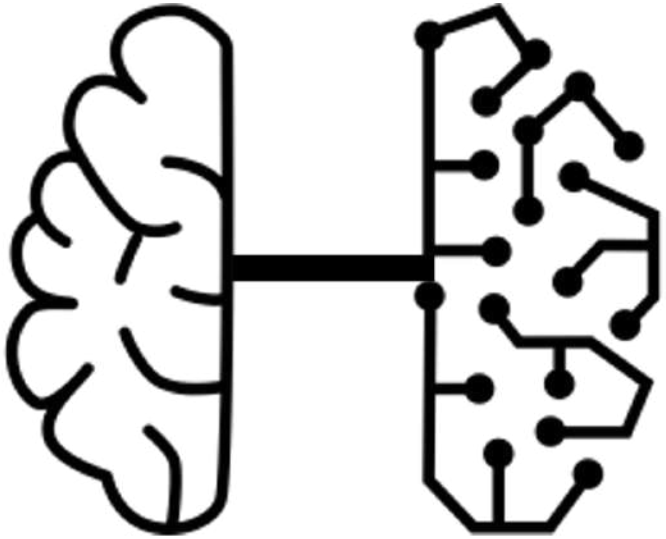


HERCOLE Lab

Check out the lab's home page

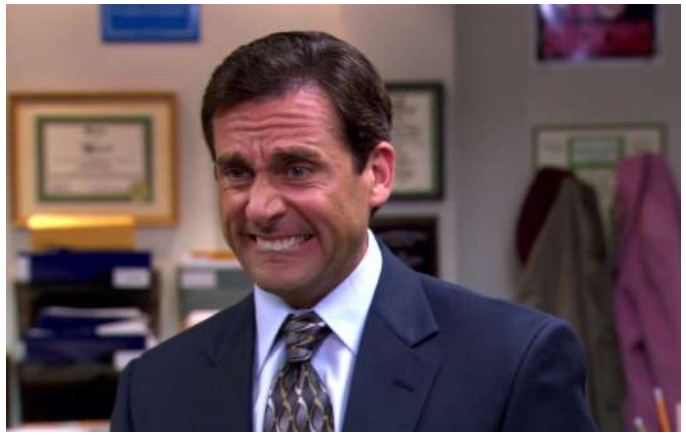
# My Research Interests

Sounds cool?



HERCOLE Lab

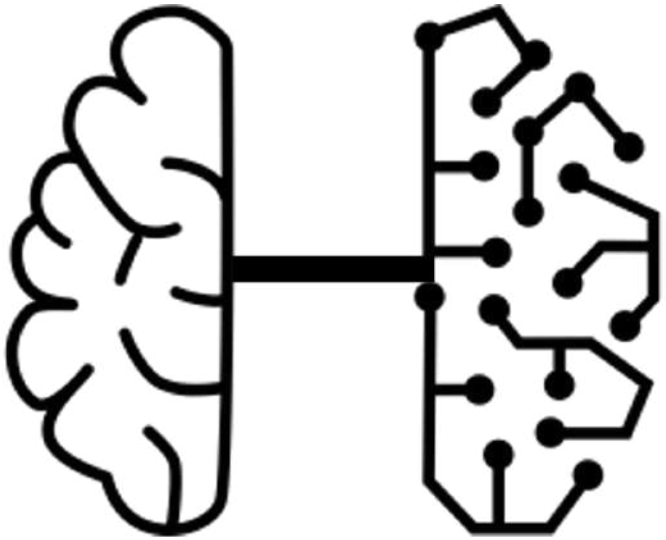
Check out the lab's home page



So hard to keep it updated!

# My Research Interests

Sounds cool?



HERCOLE Lab

Meanwhile you can follow us on Twitter (X) @HercoleLab

# Useful Information

## Class schedule

- **Tuesday:** 4 p.m. - 7 p.m.
- **Thursday:** 1 p.m. - 3 p.m.

# Useful Information

## Class schedule

- **Tuesday:** 4 p.m. - 7 p.m.
- **Thursday:** 1 p.m. - 3 p.m.

## Contacts

- email: tolomei@di.uniroma1.it
- website: https://github.com/gtolomei/operating-systems
- moodle: https://elearning.uniroma1.it/course/view.php?id=16966

# Useful Information

## Class schedule

- **Tuesday:** 4 p.m. - 7 p.m.
- **Thursday:** 1 p.m. - 3 p.m.

## Office hours

- Arranged via email
- in-person or remotely
- Room 106, 1st floor Building "E" (map)

## Contacts

- email: tolomei@di.uniroma1.it
- website: https://github.com/gtolomei/operating-systems
- moodle: https://elearning.uniroma1.it/course/view.php?id=16966

# Class Material

- Released on the class website
- Suggested books (though not mandatory!):
  - "*Operating System Concepts*" Ninth Edition – Silberschatz, Galvin, Gagne
  - "*Modern Operating Systems*" Fourth Edition – Tanenbaum, Bos
  - "*Operating Systems: Three Easy Pieces*" – Remzi and Andrea Arpaci-Dusseau [available online]
- Any additional resource available on the Web!

# Moodle

- Provides native support for:
  - Sharing news and messages (forum)
  - Additional class material (e.g., exercises)
  - Exam simulations (e.g., quizzes)
  - …

Remember to enroll in the course from the moodle web page!

# Exam

- **Moodle Quiz:**
  - 20 multiple-answer questions (max. 60 minutes)
  - Marks: +3 (correct answer), 0 (no answer), -1 (wrong answer)
    - score <= 14/30 → FAIL
    - 15/30 <= score <= 17/30 → ORAL REQUIRED
    - score >= 18/30 → PASS (oral upon request by the student)

# Exam

- **Moodle Quiz:**
  - 20 multiple-answer questions (max. 60 minutes)
  - Marks: +3 (correct answer), 0 (no answer), -1 (wrong answer)
    - score <= 14/30 → FAIL
    - 15/30 <= score <= 17/30 → ORAL REQUIRED
    - score >= 18/30 → PASS (oral upon request by the student)
- **Oral Session:**
  - Questions and exercises on the subjects covered during the whole semester

# Outline of the Course

- Part I: Introduction

# Outline of the Course

- Part I: Introduction
- Part II: Process Management

# Outline of the Course

- Part I: Introduction
- Part II: Process Management
- Part III: Process Synchronization

# Outline of the Course

- Part I: Introduction
- Part II: Process Management
- Part III: Process Synchronization
- Part IV: Memory Management

# Outline of the Course

- Part I: Introduction
- Part II: Process Management
- Part III: Process Synchronization
- Part IV: Memory Management
- Part V: Storage Management

# Outline of the Course

- Part I: Introduction
- Part II: Process Management
- Part III: Process Synchronization
- Part IV: Memory Management
- Part V: Storage Management
- Part VI: File System

# Outline of the Course

- Part I: Introduction
- Part II: Process Management
- Part III: Process Synchronization
- Part IV: Memory Management
- Part V: Storage Management
- Part VI: File System
- Part VII: Advanced Topics

# Part I: Introduction

# Language and Naming Conventions

- OS → Operating System
- HW → Hardware
- SW → Software
- VM → Virtual Machine
- …
- Other shortcuts/acronyms may appear here and there without notice! Please, ask if anything is not clear!

# What is an Operating System?

# What is an Operating System?

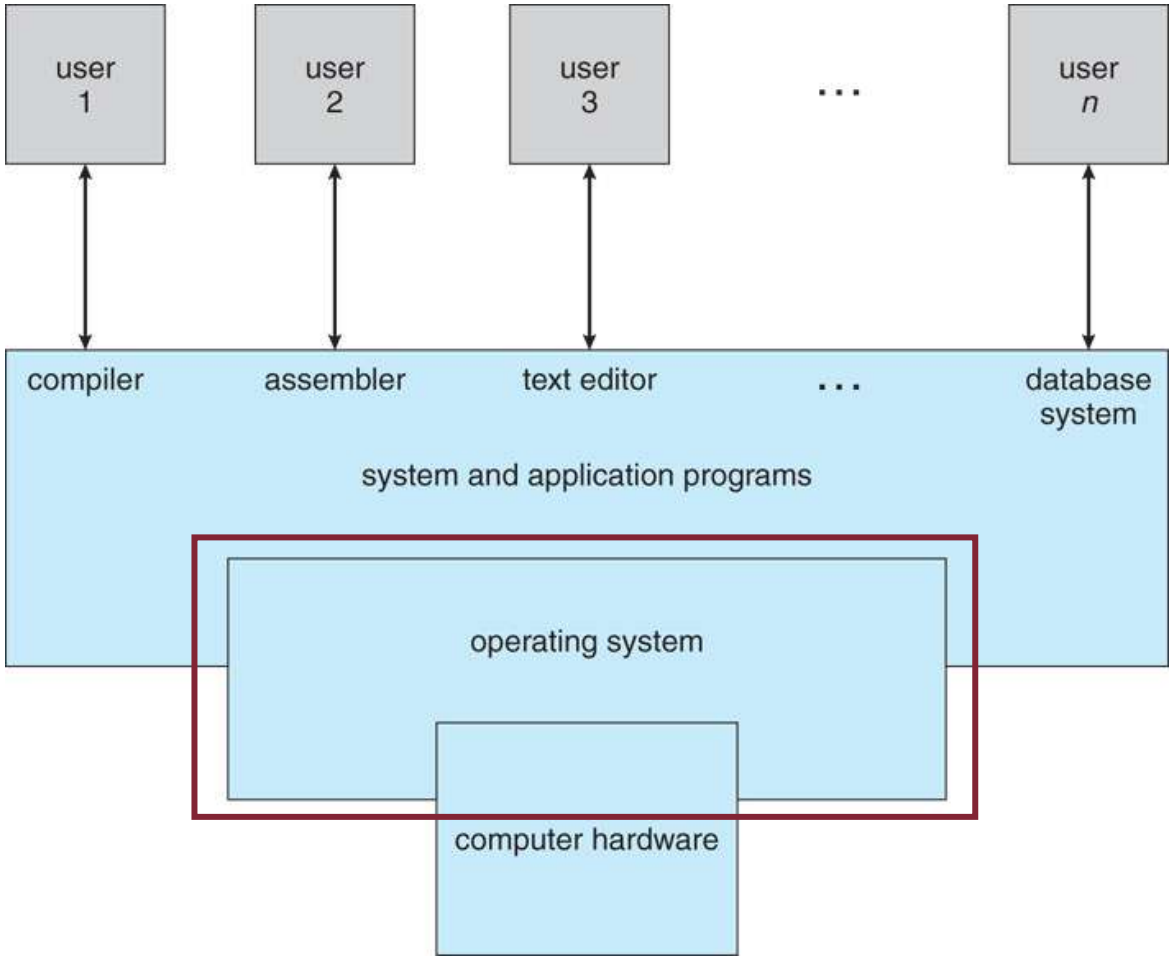
- There exists no universally accepted definition!

# What is an Operating System?

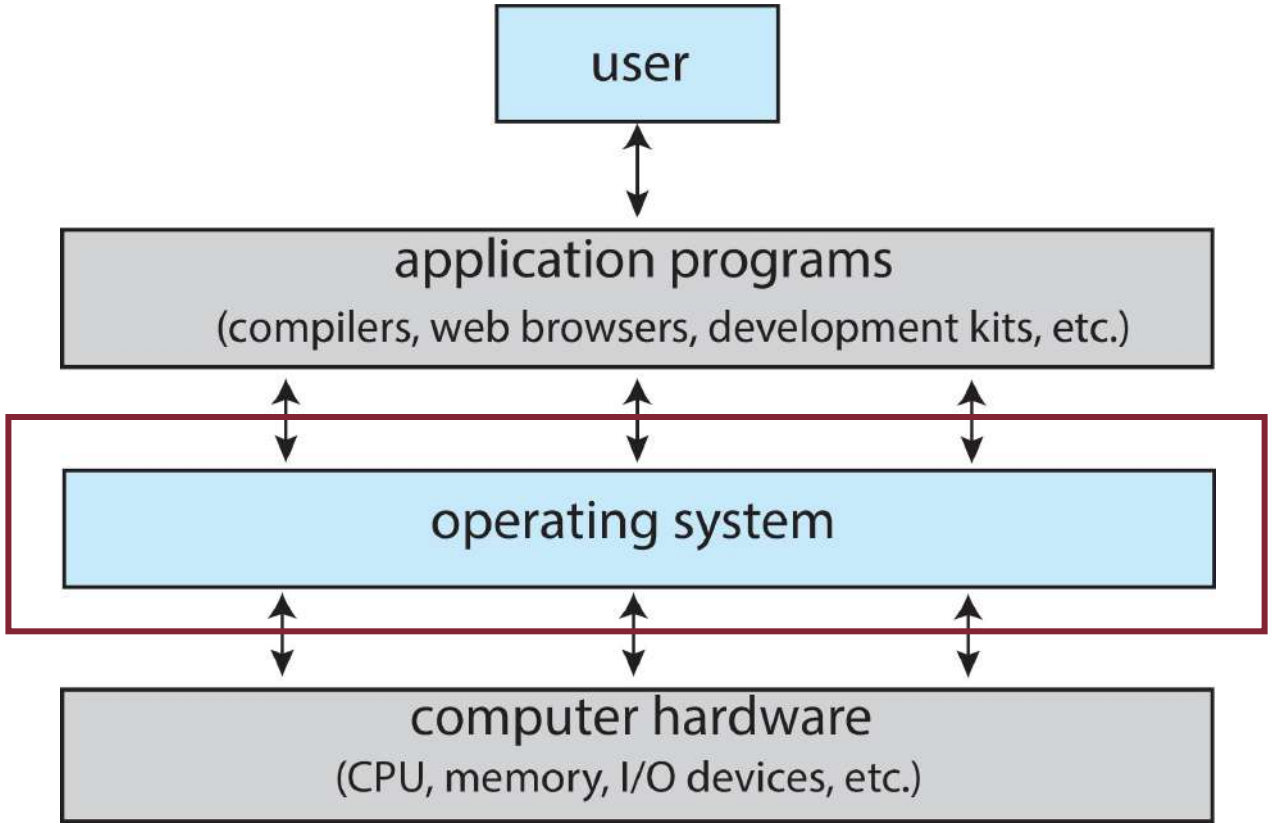
- There exists no universally accepted definition!
- However, the definition below is quite appropriate:

> Implementation of a virtual machine that is (hopefully) easier to program than bare hardware

# Computer System Overview



# Computer System Overview



# What is Inside an Operating System?

- Again, no single answer to this question!

# What is Inside an Operating System?

- Again, no single answer to this question!
- It is a **system design** choice to decide what to include in the OS

# What is Inside an Operating System?

- Again, no single answer to this question!
- It is a **system design** choice to decide what to include in the OS
- Different systems may have different requirements:
  - general-purpose, real-time, mobile, etc.

# What is Inside an Operating System?

- Again, no single answer to this question!
- It is a **system design** choice to decide what to include in the OS
- Different systems may have different requirements:
  - general-purpose, real-time, mobile, etc.
- Typically, we distinguish between:
  - **kernel** → the "core" of the OS (always up and running)
  - **system programs** → everything else which is still part of the OS

# OS Wears Many Hats

- Referee (Resource Manager)
  - Manages shared physical resources: CPUs, memory, I/O, etc.

# OS Wears Many Hats

- Referee (Resource Manager)
  - Manages shared physical resources: CPUs, memory, I/O, etc.
  - To achieve **fairness** and **efficiency**

# OS Wears Many Hats

- Illusionist (Virtual Machine)
  - Virtualize any physical resource

# OS Wears Many Hats

- **Illusionist (Virtual Machine)**
  - Virtualize any physical resource
  - To give applications/users the **illusion of infinite resources** available

# OS Wears Many Hats

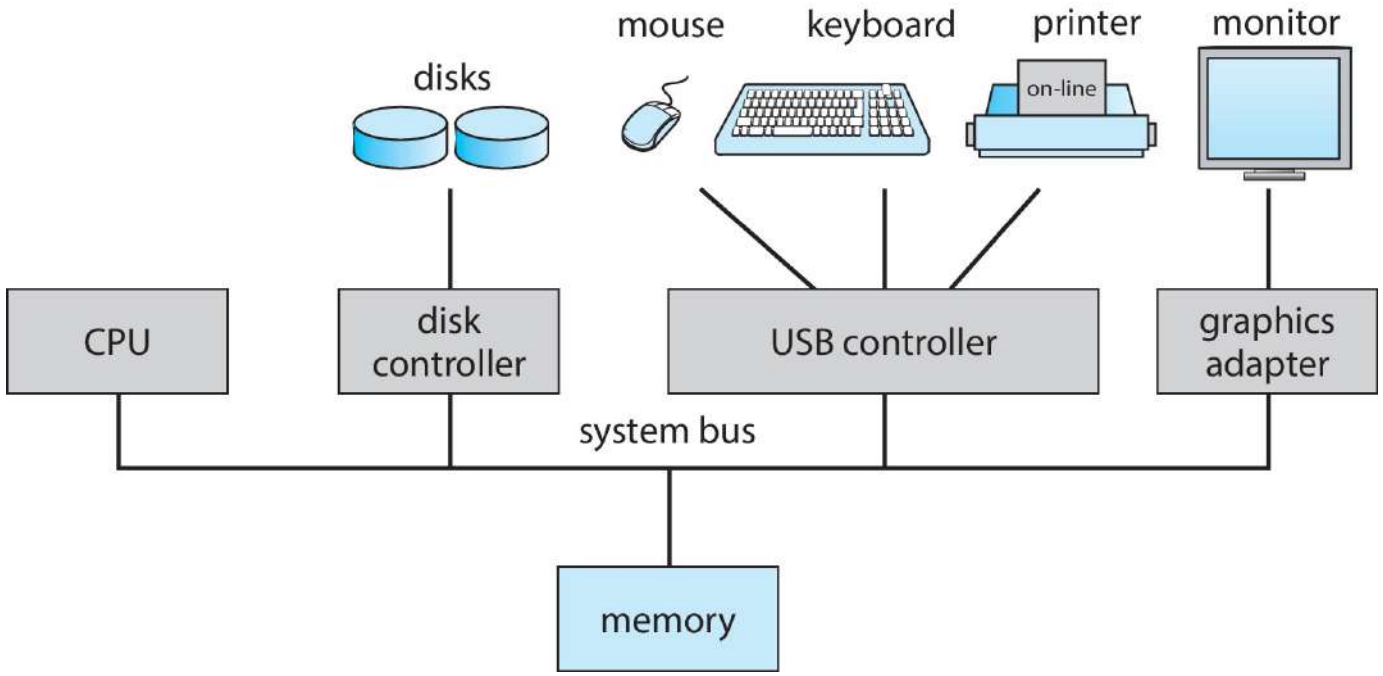
- Glue (HW/SW Interface)
  - Provides a set of common services (APIs) to separate HW from SW

# OS Wears Many Hats

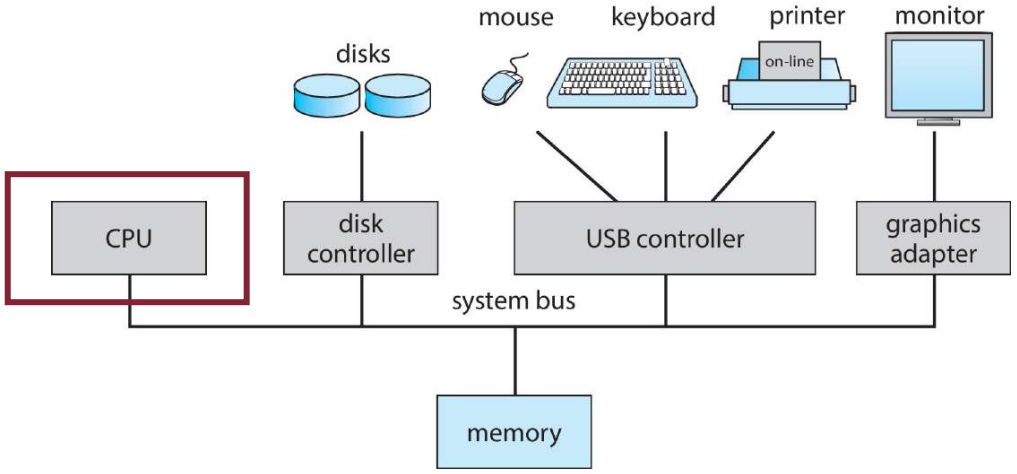
- **Glue (HW/SW Interface)**
  - Provides a set of **common services** (APIs) to separate HW from SW
  - To allow applications/users to interact with the system **without talking directly to the HW**

# Computer System Organization

# High-Level View of a Computer



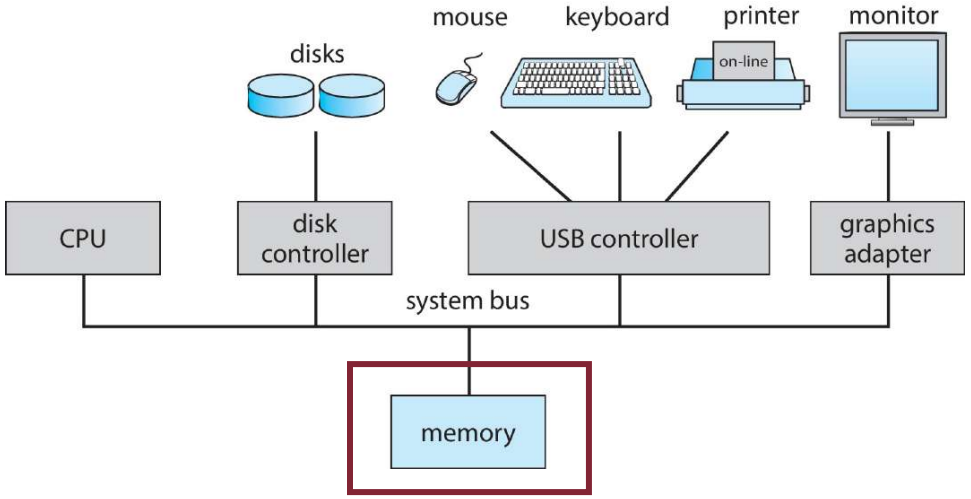
# High-Level View of a Computer




## CPU

- The processor that performs the actual computation
- Multiple cores are now common in modern architectures

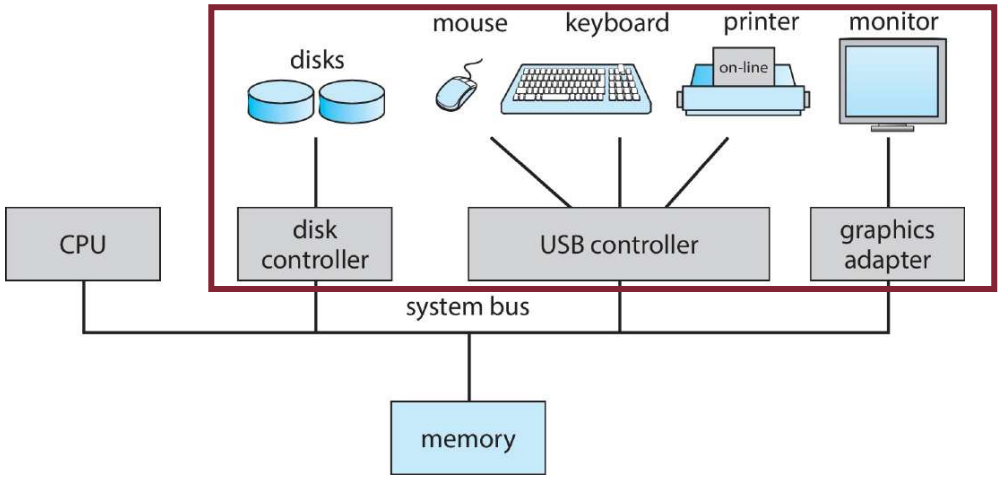
# High-Level View of a Computer




## Main Memory

- Stores data and instructions used by the CPU
- Shared between CPU and I/O

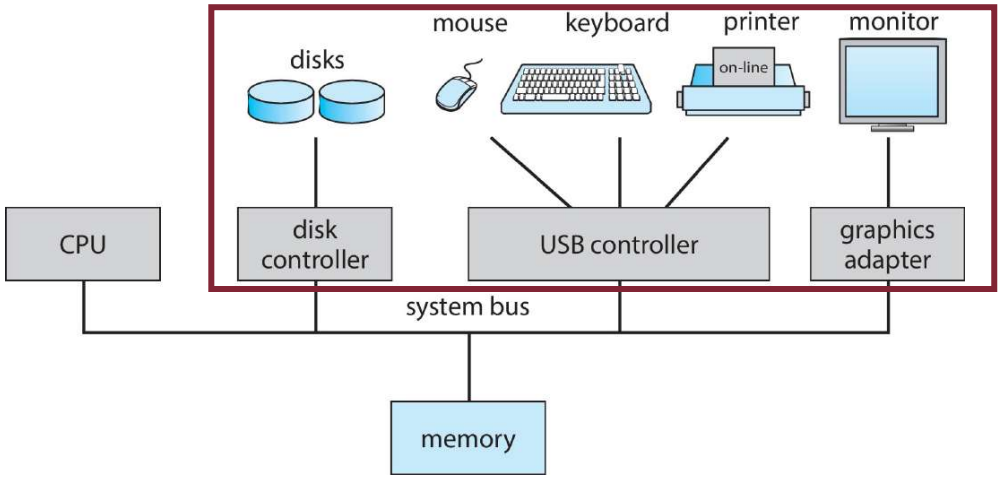
# High-Level View of a Computer




## I/O devices

- terminal, keyboard, disks, etc.
- Associated with specific device controllers

# High-Level View of a Computer




## System Bus

- Communication medium between CPU, memory, and peripherals

# Computer Architecture Model

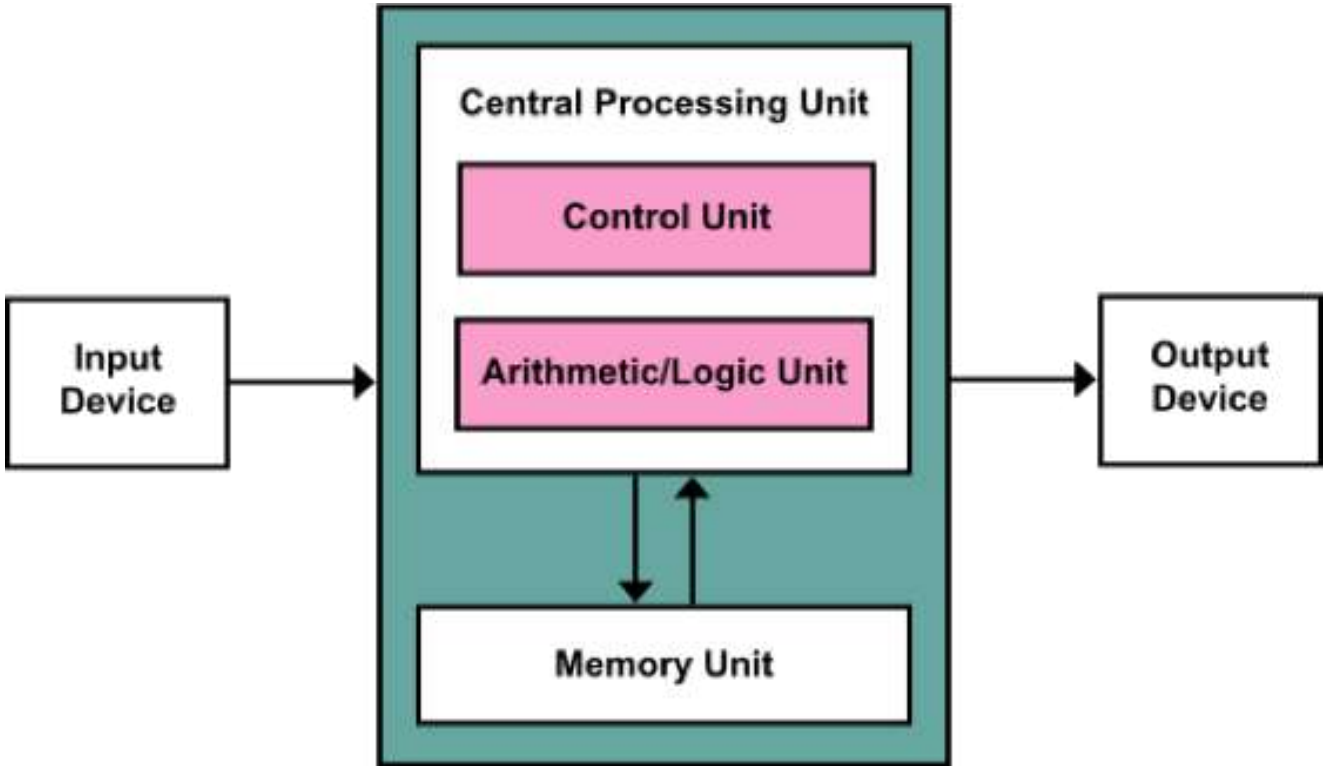
- Conceptually, the same architectural model for many computing devices:
  - PCs/laptops
  - High-end servers
  - Smartphones/Tablets
  - etc.

# Computer Architecture Model

- Conceptually, the same architectural model for many computing devices:
  - PCs/laptops
  - High-end servers
  - Smartphones/Tablets
  - etc.
- Based on **stored-program** concept (as opposed to fixed-program)

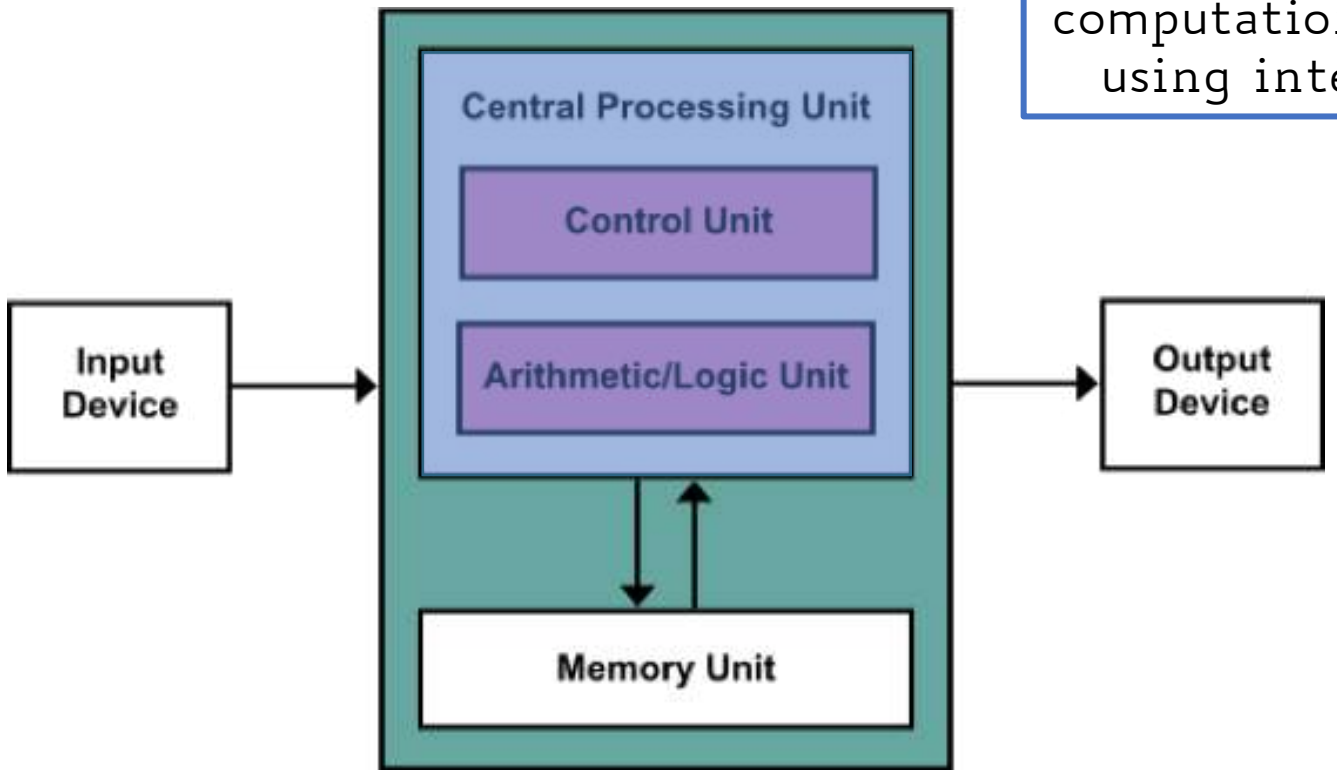
John von Neumann

# von Neumann Architecture

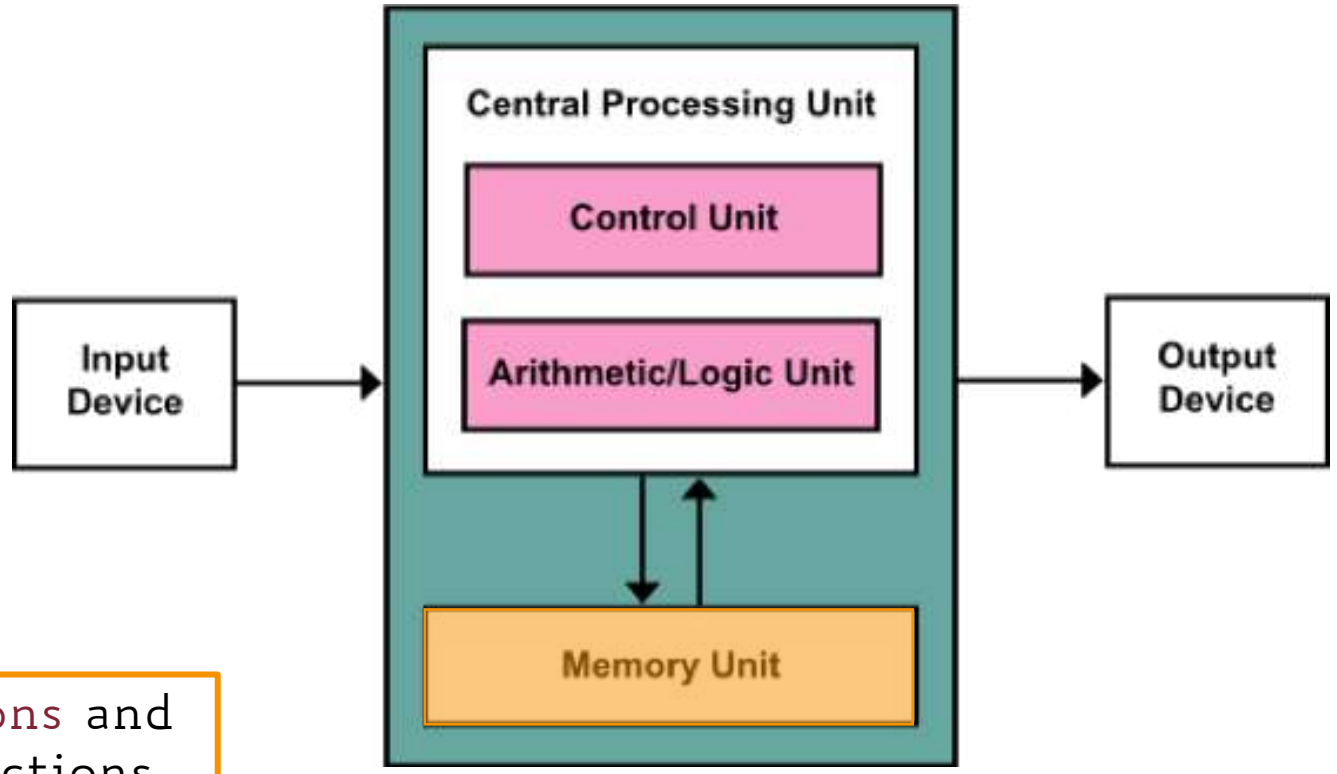


# von Neumann Architecture

Executes instructions (i.e., basic arithmetical/logical computations) sequentially, using internal registers



# von Neumann Architecture




Contains instructions and data (which instructions operate on)

# Central Processing Unit (CPU)

# Instruction Cycle: Fetch-Decode-Execute

- The CPU performs (at least) the following 3 steps cyclically:

# Instruction Cycle: Fetch-Decode-Execute

- The CPU performs (at least) the following 3 steps cyclically:
  - **Fetch:** retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)

# Instruction Cycle: Fetch-Decode-Execute

- The CPU performs (at least) the following 3 steps cyclically:
  - **Fetch:** retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)
  - **Decode:** interprets the fetched instruction

# Instruction Cycle: Fetch-Decode-Execute

- The CPU performs (at least) the following 3 steps cyclically:
  - **Fetch:** retrieves an instruction from a specific memory address whose value is contained in a special CPU register, called **Program Counter** (PC)
  - **Decode:** interprets the fetched instruction
  - **Execute:** runs the actual decoded instruction

# Machine Language

- Defines the set of **elementary instructions** the CPU is able to execute directly (i.e., on the HW)

# Machine Language

- Defines the set of **elementary instructions** the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by **binary numeral system**

# Machine Language

- Defines the set of **elementary instructions** the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by **binary numeral system**
- Each instruction is encoded as a **sequence of bits**
  - A single bit is the smallest unit of (digital) information
  - It takes on two possible values: 0 or 1

# Machine Language

- Defines the set of **elementary instructions** the CPU is able to execute directly (i.e., on the HW)
- The machine language is represented by **binary numeral system**
- Each instruction is encoded as a **sequence of bits**
  - A single bit is the smallest unit of (digital) information
  - It takes on two possible values: 0 or 1
- A **word** is the unit of data the CPU can directly operate on
  - today ranging from 32 to 64 bits

# A Side Note on Units

**Prefixes for multiples of bits (bit) or bytes (B)**

| Decimal | | | | Binary | | | | |
|---|---|---|---|---|---|---|---|---|
| Value | | SI | | Value | | IEC | | JEDEC |
| $1000$ | $10^3$ | k | kilo | $1024$ | $2^{10}$ | Ki | kibi | K kilo |
| $1000^2$ | $10^6$ | M | mega | $1024^2$ | $2^{20}$ | Mi | mebi | M mega |
| $1000^3$ | $10^9$ | G | giga | $1024^3$ | $2^{30}$ | Gi | gibi | G giga |
| $1000^4$ | $10^{12}$ | T | tera | $1024^4$ | $2^{40}$ | Ti | tebi | – |
| $1000^5$ | $10^{15}$ | P | peta | $1024^5$ | $2^{50}$ | Pi | pebi | – |
| $1000^6$ | $10^{18}$ | E | exa | $1024^6$ | $2^{60}$ | Ei | exbi | – |
| $1000^7$ | $10^{21}$ | Z | zetta | $1024^7$ | $2^{70}$ | Zi | zebi | – |
| $1000^8$ | $10^{24}$ | Y | yotta | $1024^8$ | $2^{80}$ | Yi | yobi | – |

# Instruction Set (Architecture)

- The collection of instructions defined by the machine language

# Instruction Set (Architecture)

- The collection of instructions defined by the machine language
- Machine language instructions are made of **2 parts**:
  - An **operator** (op code)
  - Zero or more **operands** representing either CPU internal registers or memory addresses

# Instruction Set (Architecture)

- The collection of instructions defined by the machine language
- Machine language instructions are made of **2 parts**:
  - An **operator** (op code)
  - Zero or more **operands** representing either CPU internal registers or memory addresses
- An **abstraction** of the underlying physical (hardware) architecture (e.g., x86, ARM, SPARC, MIPS, etc.)

# CPU Registers

- On-chip storage whose size typically coincides with the CPU word size

# CPU Registers

- On-chip storage whose size typically coincides with the CPU word size
- General-purpose (x86):
  - `eax`, `ebx`, `ecx`, etc.

# CPU Registers

- On-chip storage whose size typically coincides with the CPU word size
- General-purpose (x86):
  - `eax`, `ebx`, `ecx`, etc.
- Special-purpose (x86):
  - `esp` → Stack pointer for top address of the stack
  - `ebp` → Stack base pointer for the address of the current stack frame
  - `eip` → Instruction pointer, holds the program counter (i.e., the address of next instruction)

# Single- vs. Multi-Processor

## Single-Processor Systems

- One main CPU for executing programs
- Other dedicated processors that do not run programs (e.g., disk controllers, GPUs, etc.)

# Single- vs. Multi-Processor

## Single-Processor Systems

- One main CPU for executing programs
- Other dedicated processors that do not run programs (e.g., disk controllers, GPUs, etc.)

## Multi-Processor Systems

- Multiple CPUs to increase throughput (even if not linearly!)
- Higher resiliency to failures

# Single- vs. Multi-Processor

## Single-Processor Systems

- One main CPU for executing programs
- Other dedicated processors that do not run programs (e.g., disk controllers, GPUs, etc.)

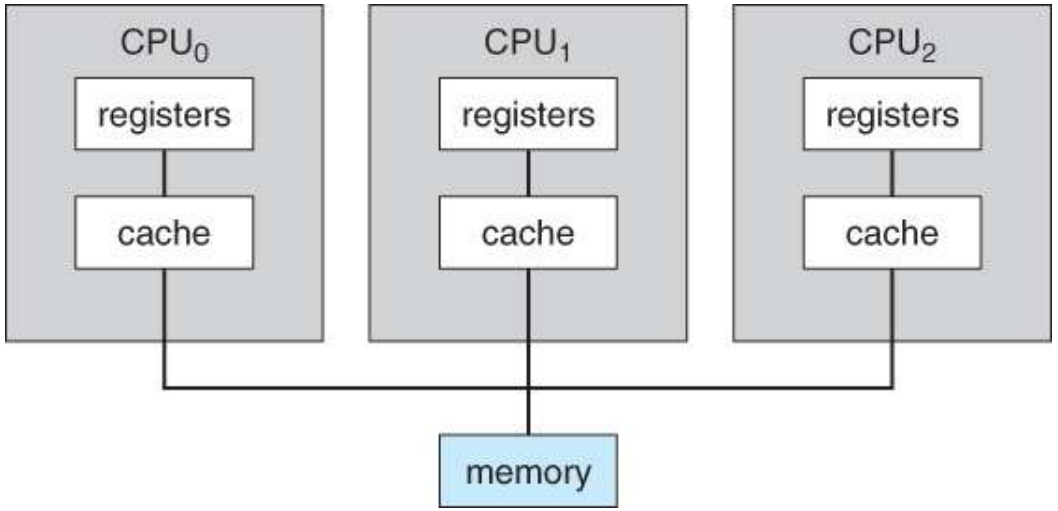
Our main focus!

## Multi-Processor Systems

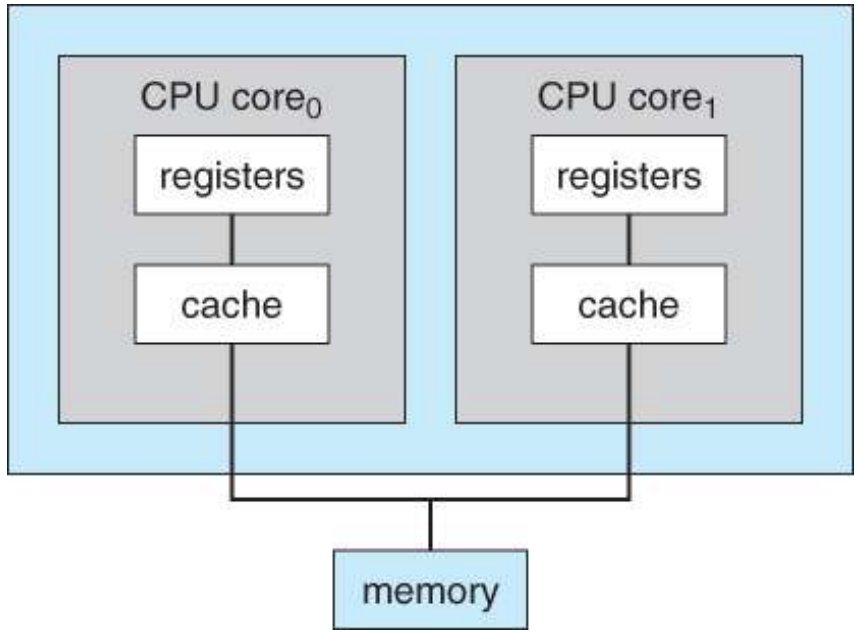
- Multiple CPUs to increase throughput (even if not linearly!)
- Higher resiliency to failures

# Multi-Processor Systems: Examples

## Symmetric Multiprocessing Architecture

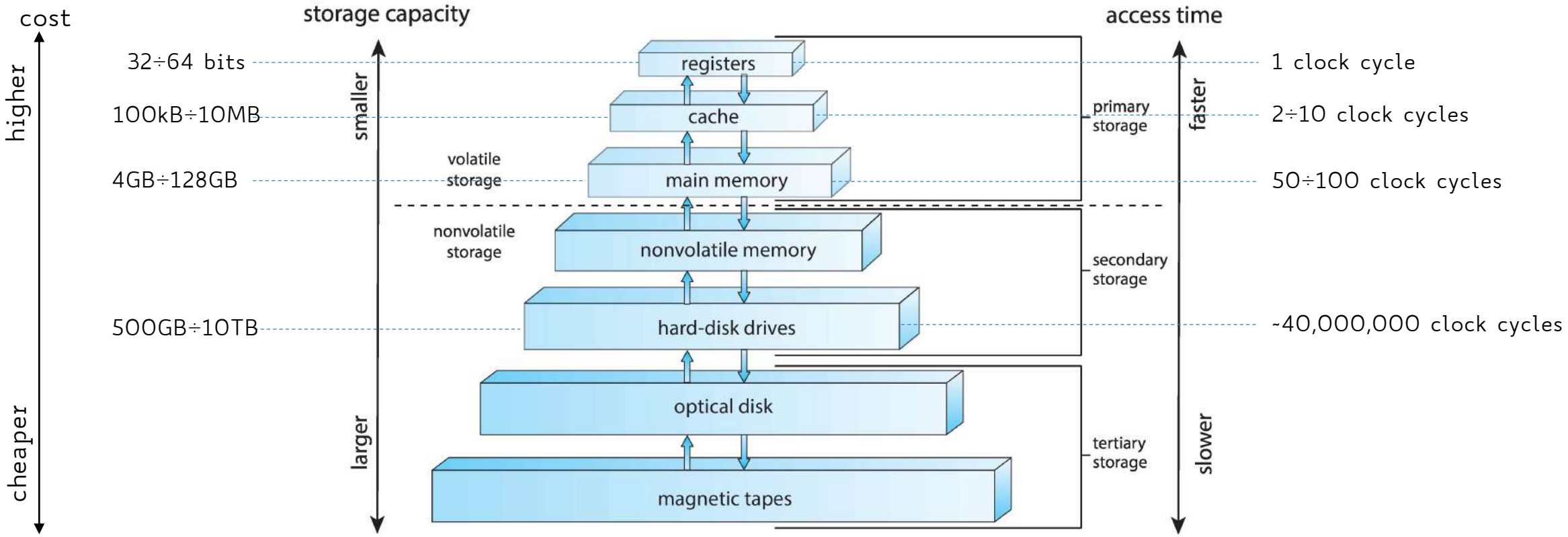



## Multicore Architecture

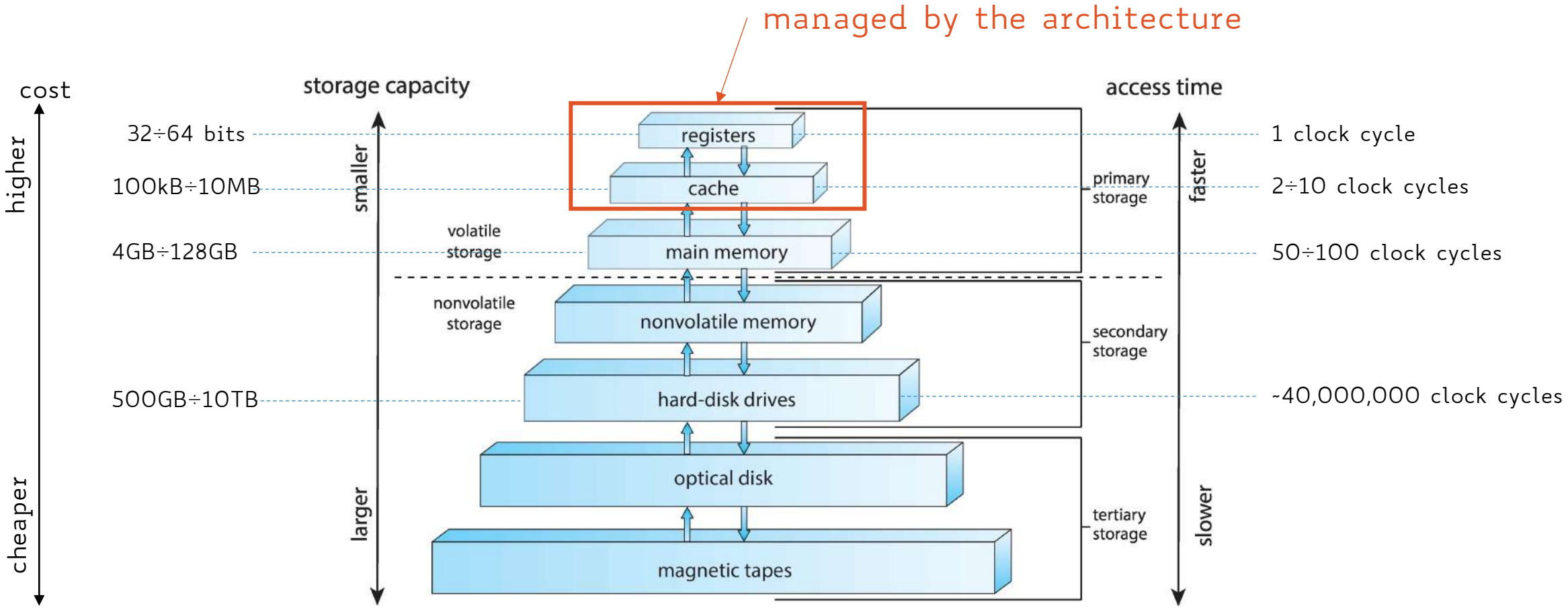


# Memory

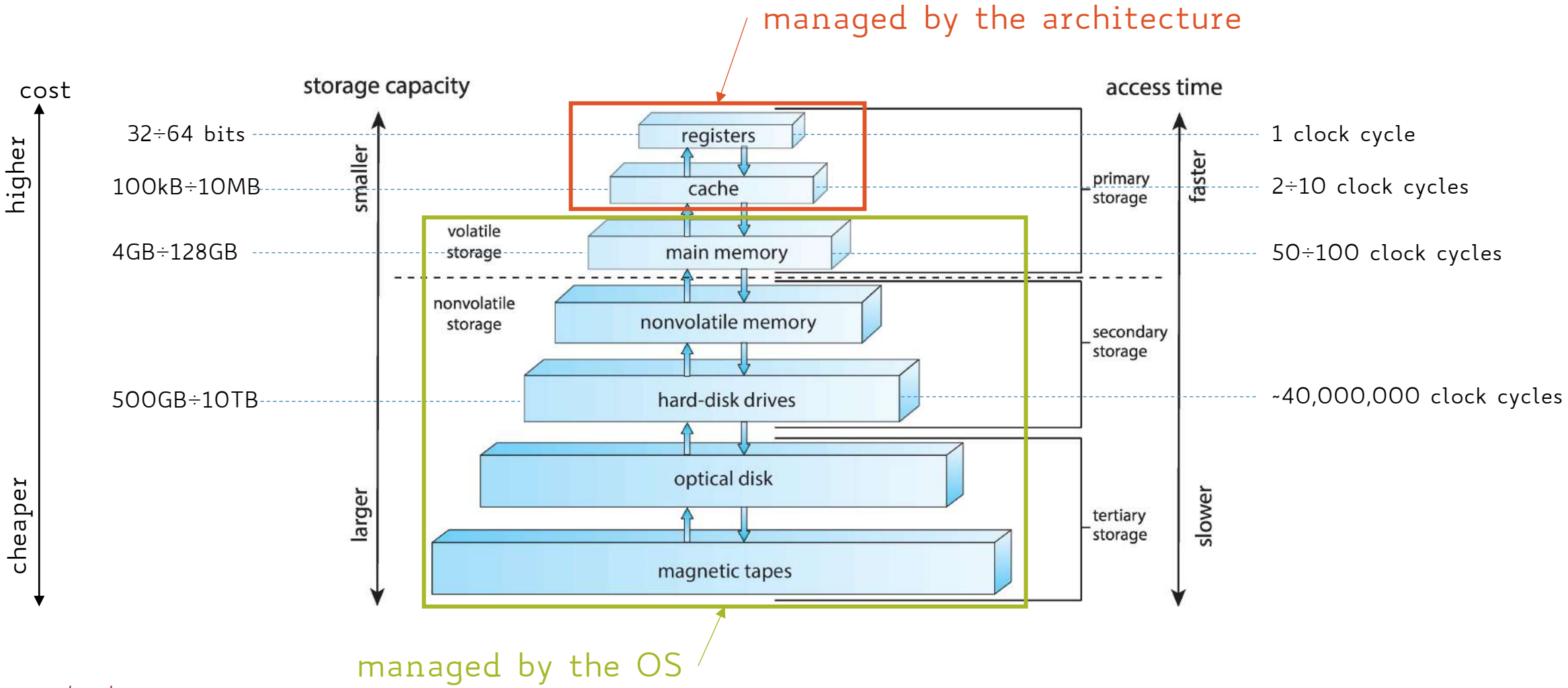
# Memory Hierarchy



# Memory Hierarchy



# Memory Hierarchy



# Main Memory Representation

- Essentially, a sequence of cells

# Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)

# Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own **address**

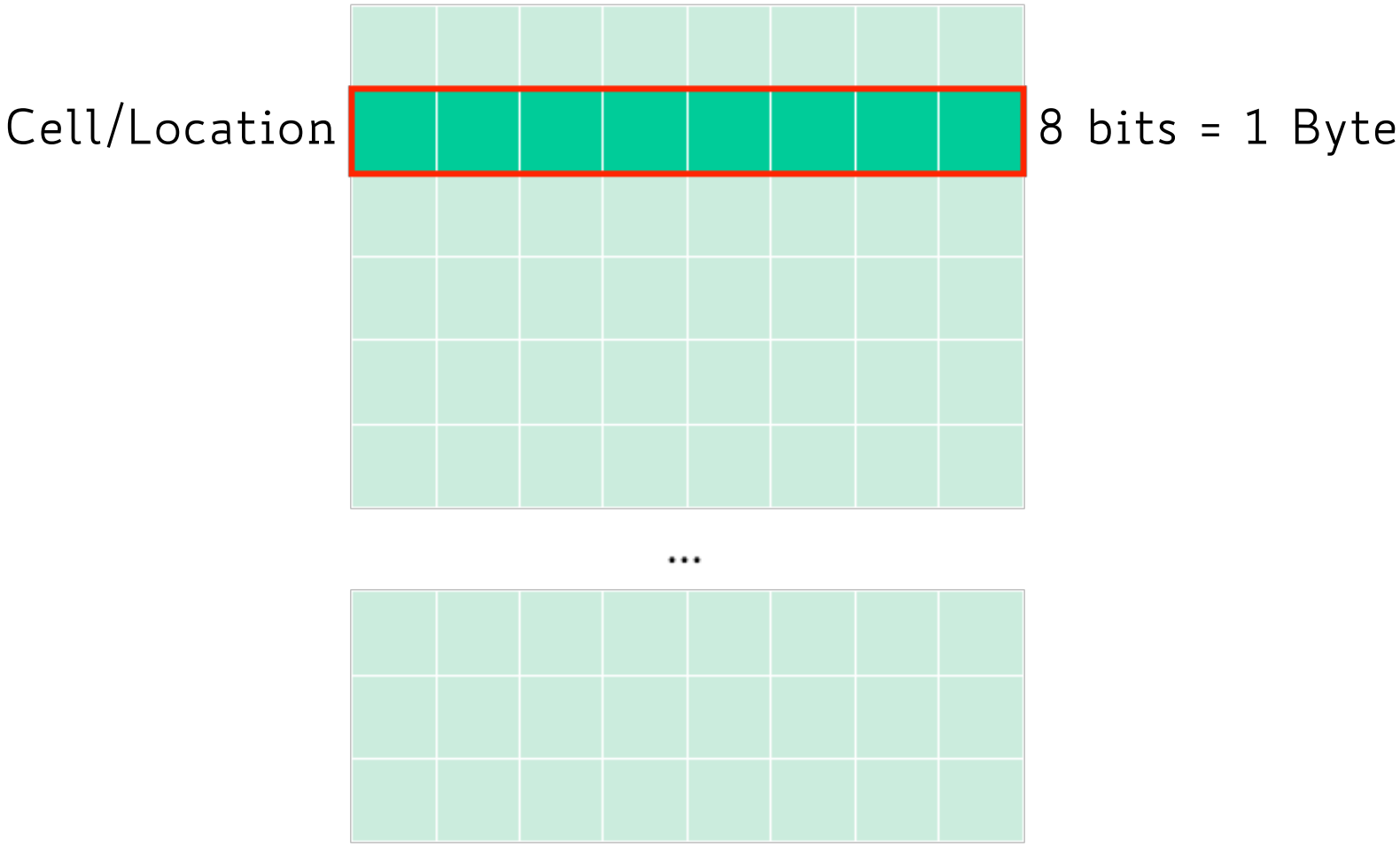
# Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own **address**
- CPU (and I/O devices) read from/write to main memory referencing memory location addresses

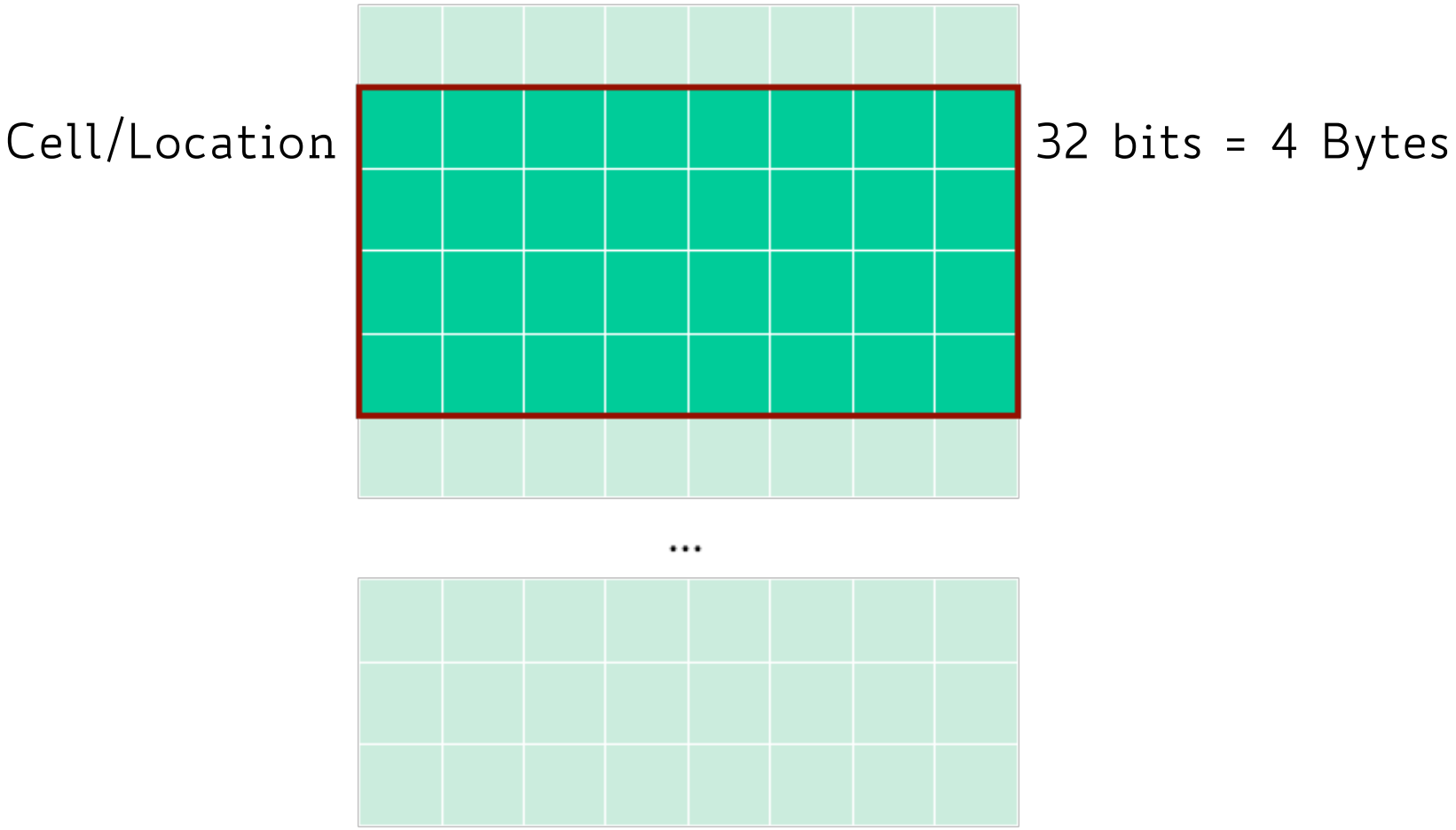
# Main Memory Representation

- Essentially, a sequence of cells
- Each cell is logically organized in groups of 8 bits (1 Byte) or multiple of it (e.g., 32 bits = 4 Bytes)
- Each cell has its own **address**
- CPU (and I/O devices) read from/write to main memory referencing memory location addresses
- The smallest addressable unit is usually 1 Byte

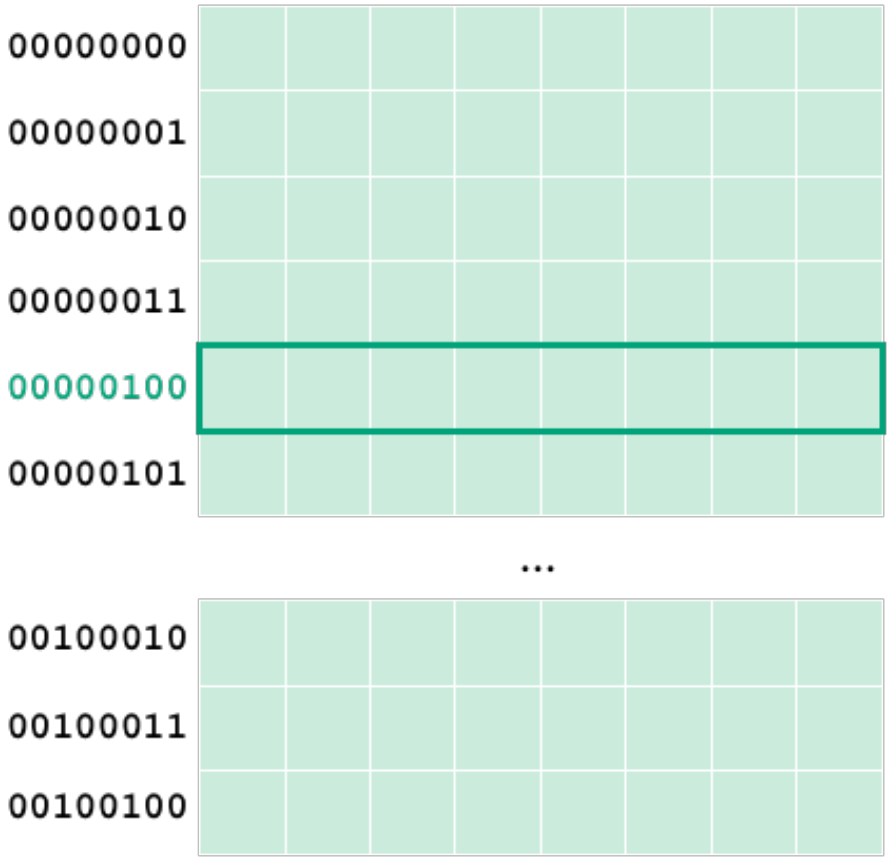
# Memory Cell (1)



# Memory Cell (2)



# Memory Address (Single Byte)



# Computer Buses

# System Bus

- Initially, a single bus to handle all the traffic

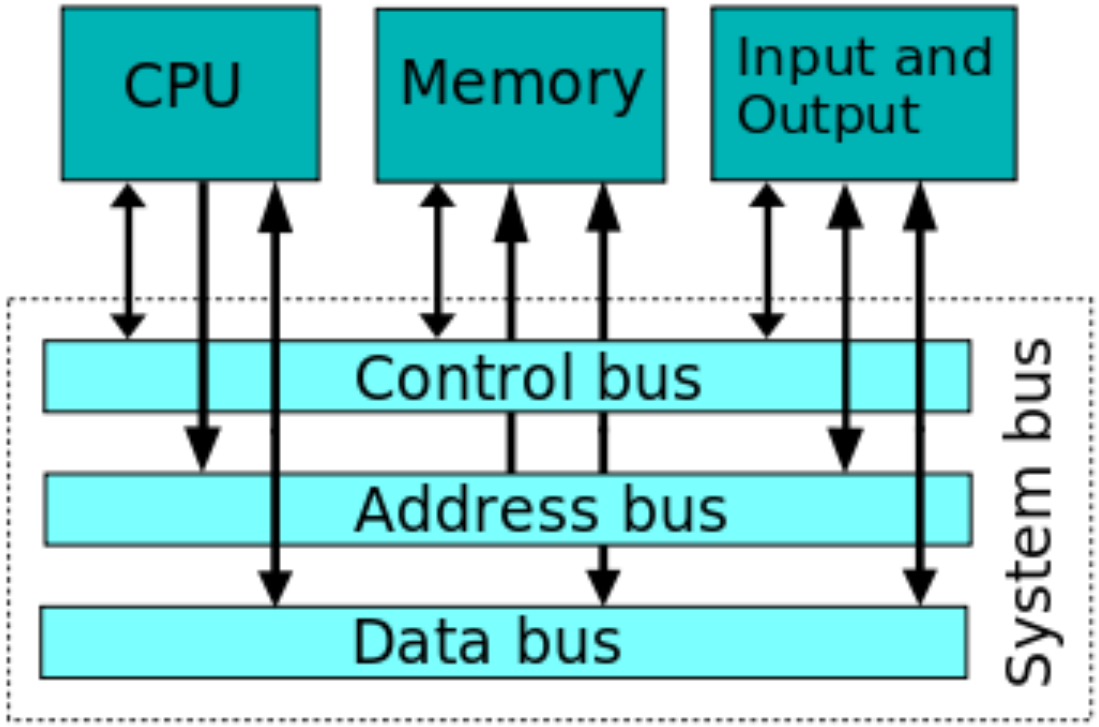
# System Bus

- Initially, a single bus to handle all the traffic
- Combines the functions of:
  - **Data bus** → to actually carry information
  - **Address bus** → to determine where information should be sent
  - **Control bus** → to indicate which operation should be performed

# System Bus

- Initially, a single bus to handle all the traffic
- Combines the functions of:
  - **Data bus** → to actually carry information
  - **Address bus** → to determine where information should be sent
  - **Control bus** → to indicate which operation should be performed
- More dedicated buses have been added to manage CPU-to-memory and I/O traffic
  - PCI, SATA, USB, etc.

# System Bus


CPU
Memory
Input and Output
Control bus
Address bus
Data bus
System bus

# I/O Devices

# Components

- Each I/O device is made of **2 parts**:

# Components

- Each I/O device is made of **2 parts**:
  - the **physical device** itself
  - the **device controller** (chip or set of chips controlling a family of physical devices)

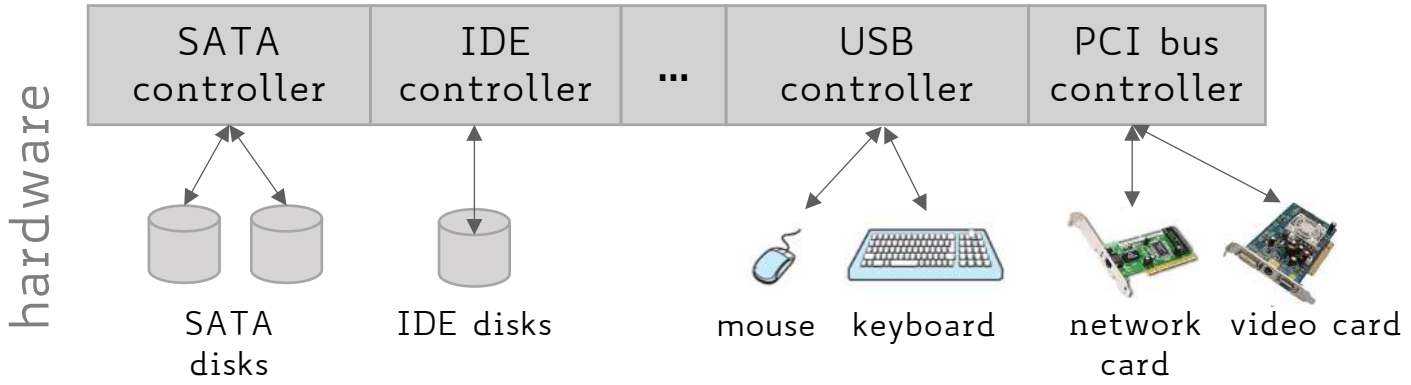
# Components

- Each I/O device is made of **2 parts**:
  - the **physical device** itself
  - the **device controller** (chip or set of chips controlling a family of physical devices)
- Can be categorized as:
  - storage, communications, user-interface, etc.

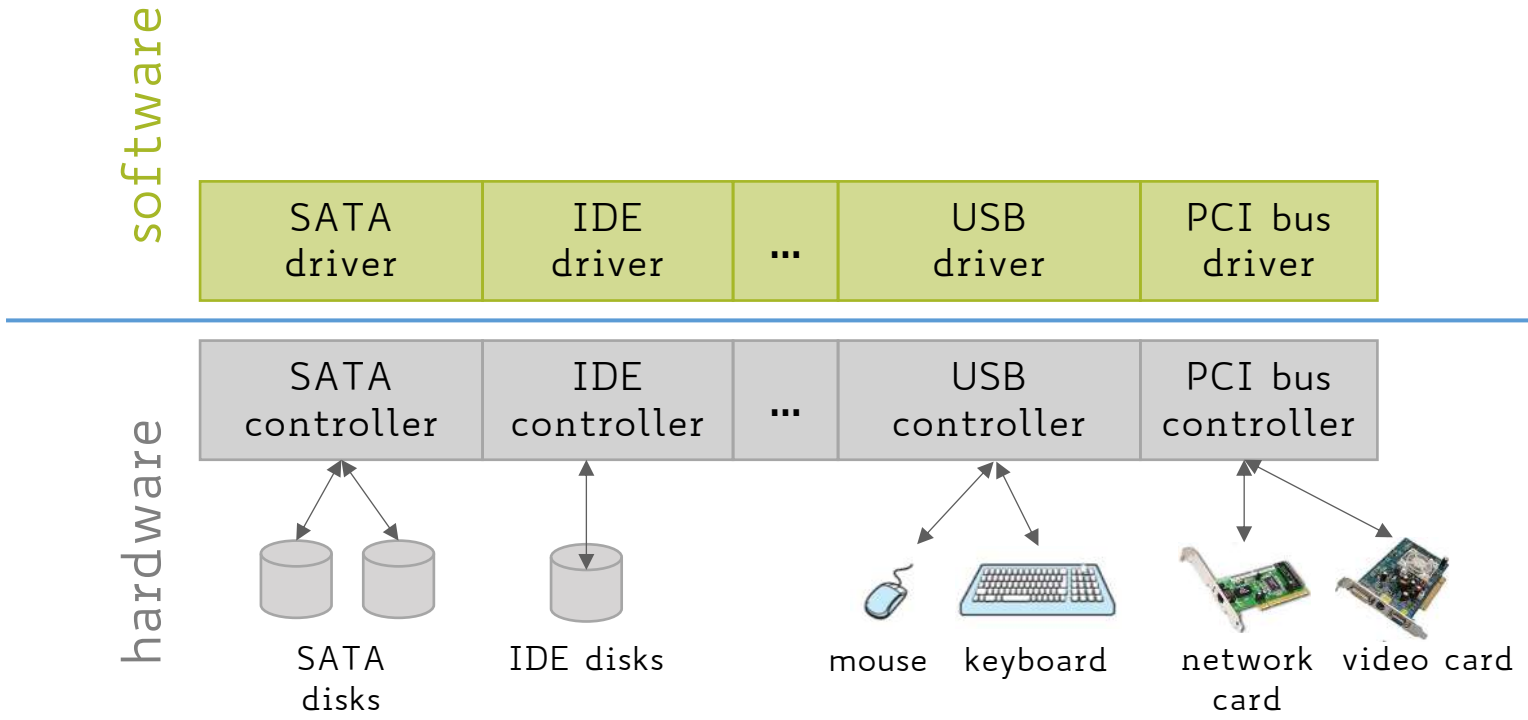
# Components

- Each I/O device is made of **2 parts**:
  - the **physical device** itself
  - the **device controller** (chip or set of chips controlling a family of physical devices)
- Can be categorized as:
  - storage, communications, user-interface, etc.
- OS talks to a device controller using a specific **device driver**

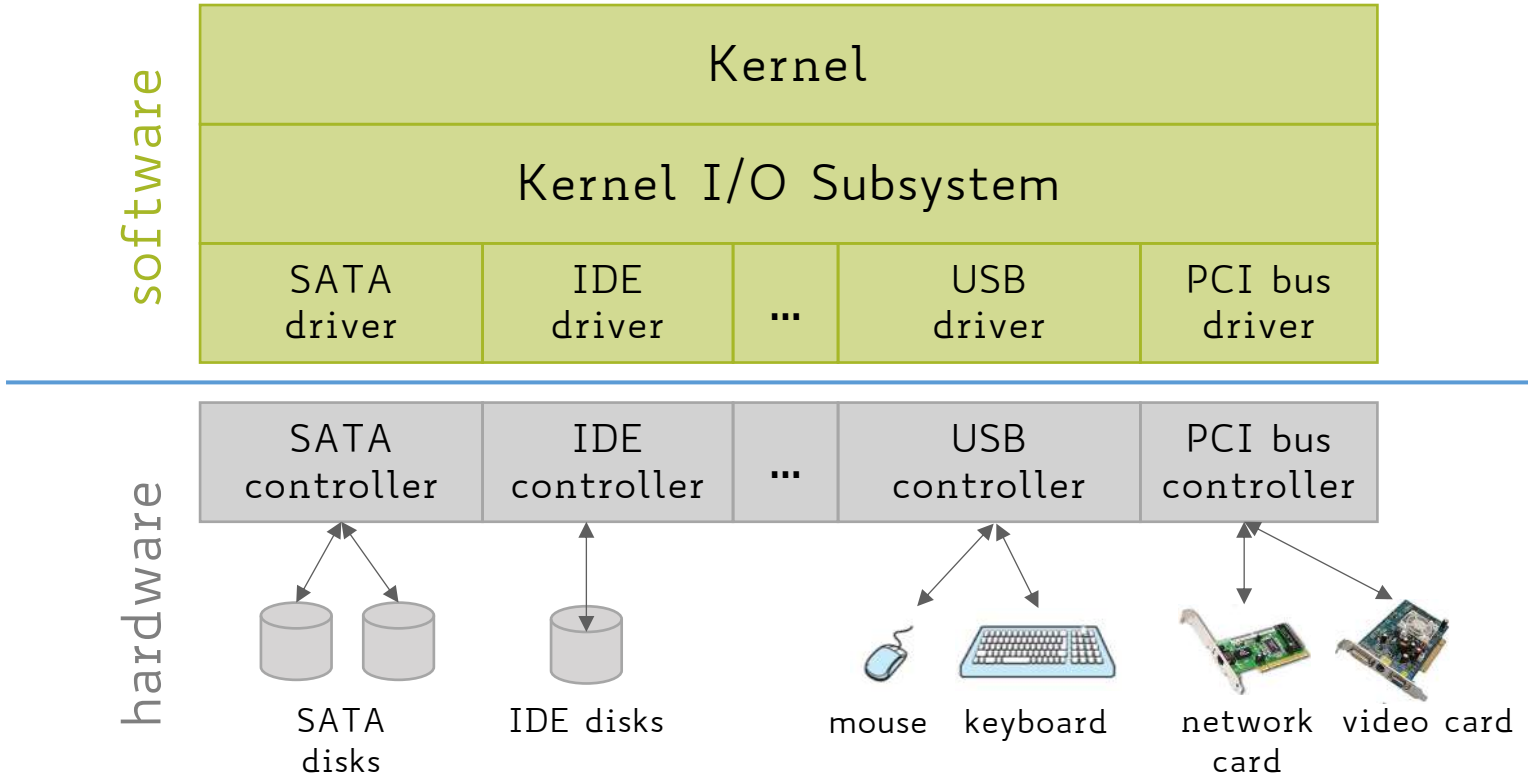
# Device Drivers: OS Abstraction



# Device Drivers: OS Abstraction



# Device Drivers: OS Abstraction



# Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - **Status registers** → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)

# Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - **Status registers** → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - **Configuration/Control registers** → used by the CPU to configure and control the device

# Device Controllers

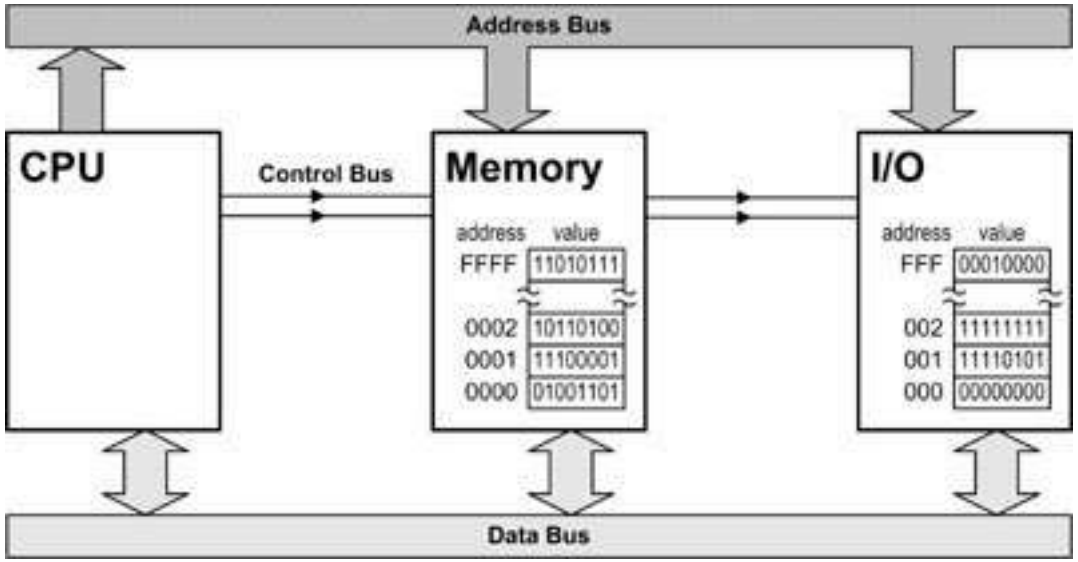
- Every device controller has a number of dedicated registers to communicate with it:
  - **Status registers** → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - **Configuration/Control registers** → used by the CPU to configure and control the device
  - **Data registers** → used to read data from or send data to the I/O device

# Device Controllers

- Every device controller has a number of dedicated registers to communicate with it:
  - **Status registers** → provide status information to the CPU about the I/O device (e.g., idle, ready for input, busy, error, transaction complete)
  - **Configuration/Control registers** → used by the CPU to configure and control the device
  - **Data registers** → used to read data from or send data to the I/O device

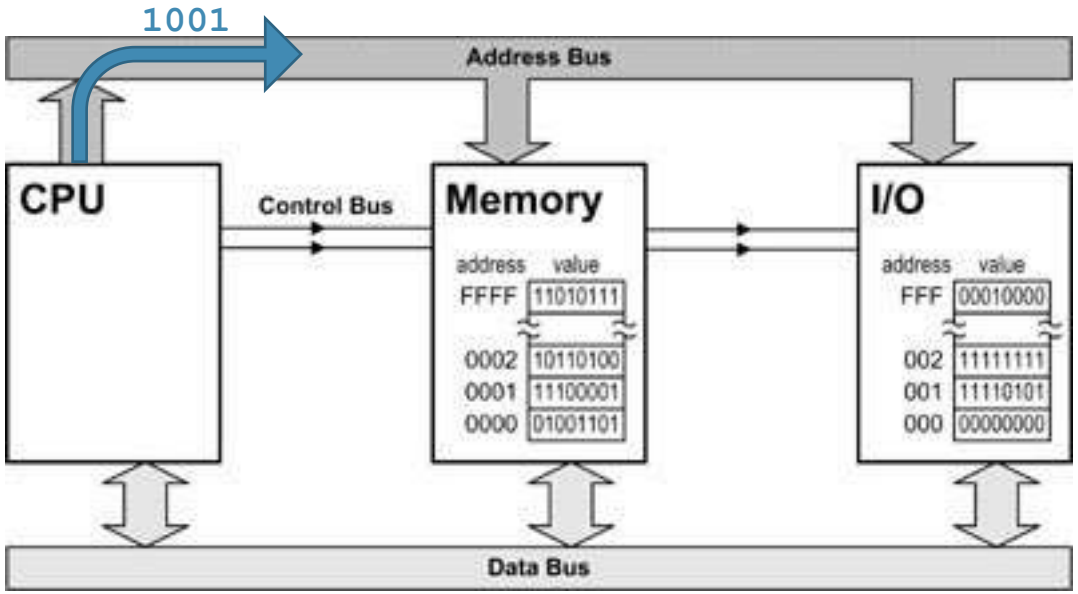
How does the CPU know how to address (registers of) I/O devices?

# Addressing Using the System Bus




How does CPU reference Memory addresses?

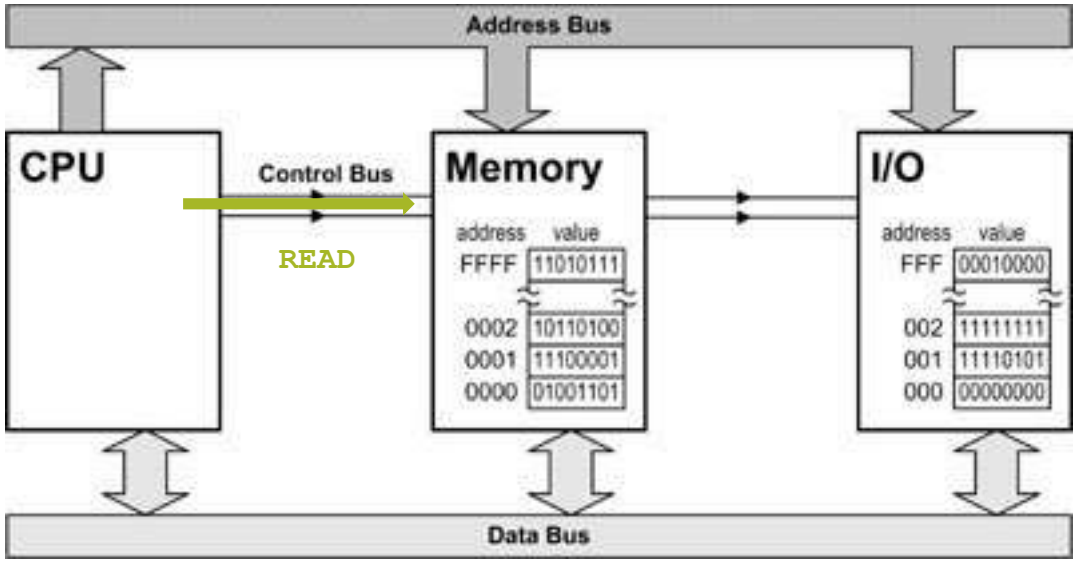
# Addressing Using the System Bus




**How does CPU reference Memory addresses?**

It puts the address of a byte of memory on the address bus

# Addressing Using the System Bus

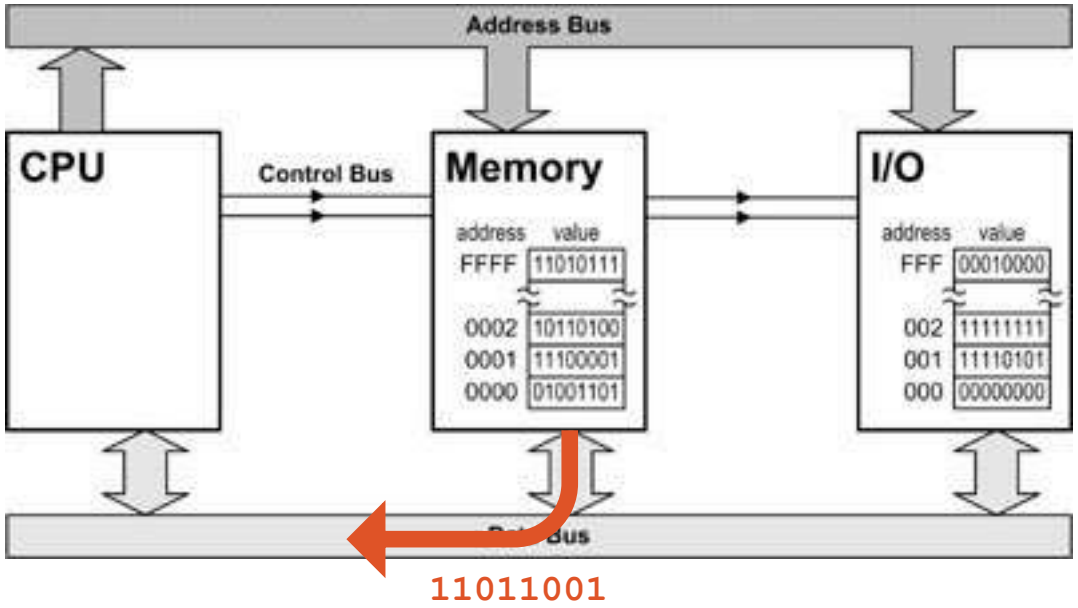



**How does CPU reference Memory addresses?**

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

# Addressing Using the System Bus



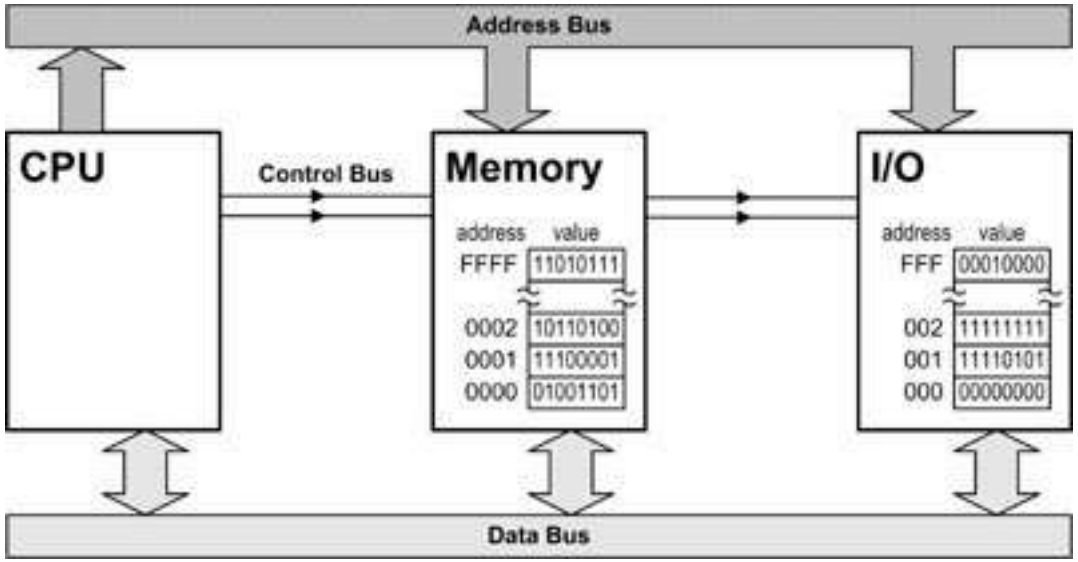

**How does CPU reference Memory addresses?**

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

# Addressing Using the System Bus




How does CPU reference Memory addresses?

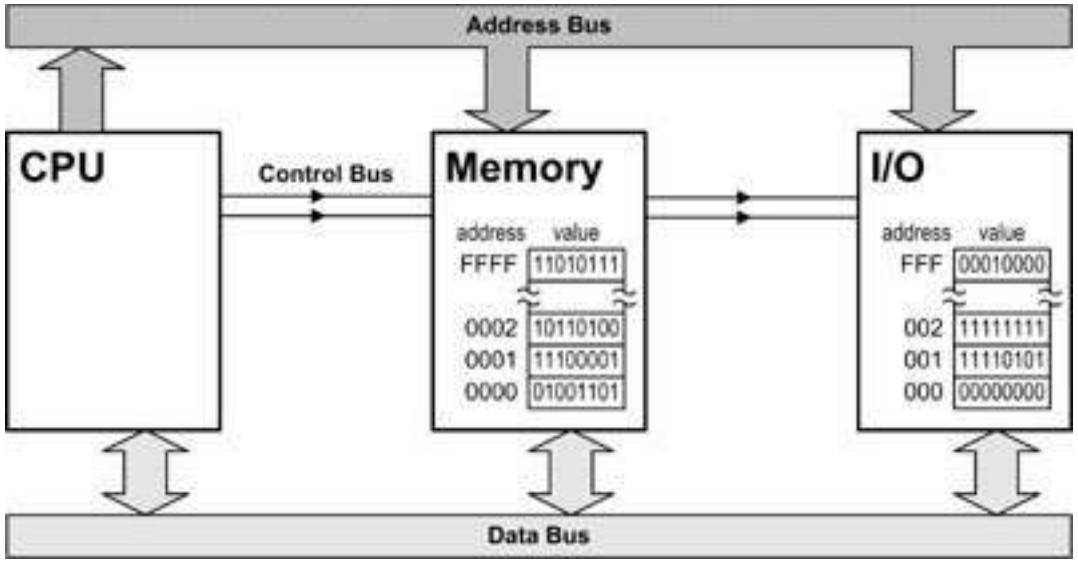
It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

How about I/O devices? How to distinguish between Memory and I/O devices?

# Addressing Using the System Bus




**How does CPU reference Memory addresses?**

It puts the address of a byte of memory on the address bus

It raises the **READ** signal on the control bus

Eventually, the RAM replies with the memory content on the data bus

If the control bus is shared between memory and I/O there is a special line called "**M/#IO**" that asserts whether the CPU wants to talk to memory or an I/O device

# Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in **2 ways**:

# Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in **2 ways**:
  - **Port-mapped I/O** → referencing controller's registers using a separate I/O address space

# Port- vs. Memory-Mapped I/O

- CPU can talk to a device controller in **2 ways**:
  - **Port-mapped I/O** → referencing controller's registers using a separate I/O address space
  - **Memory-mapped I/O** → mapping controller's registers to the same address space used for main memory

# Port-Mapped I/O

- Each I/O device controller's register is mapped to a specific port (address) at boot-up time
- Requires special class of CPU instructions (e.g., **IN**/**OUT**)
  - The **IN** instruction reads from an I/O device, **OUT** writes to it
- With the **IN** or **OUT** instructions, the **M/#IO** is not asserted, so memory does not respond and the I/O chip does

# Memory-Mapped I/O

- Memory-mapped I/O "wastes" some address space but doesn't need any special instruction
- To the CPU, I/O device ports are just like normal memory addresses mapped into RAM at boot-up time
- The CPU uses `MOV`-like instructions to access I/O device registers
- The **`M/#IO`** is always asserted indicating the address requested by the CPU refers to main memory

# Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

Both put the value **1234h** on the CPU address bus, and both assert a **READ** operation on control bus

# Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

The first one will assert **M/#IO** to indicate that the address belongs to memory address space

# Port- vs. Memory-Mapped I/O

```
MOV DX,1234h
MOV AL,[DX]     ;reads memory address 1234h (memory address space)
IN AL,DX        ;reads I/O port 1234h (I/O address space)
```

The second one will **not** assert **M/#IO** to indicate that the address belongs to I/O address space

# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status

# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status
- **Interrupt-driven**
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)

# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status
- **Interrupt-driven**
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)
- **Programmed I/O**
  - CPU does the actual work of moving data

# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status
- **Interrupt-driven**
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)
- **Programmed I/O**
  - CPU does the actual work of moving data
- **Direct Memory Access (DMA)**
  - CPU delegates off the work to a dedicated DMA controller

# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status
- **Interrupt-driven**
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)

HOW?

- **Programmed I/O**
  - CPU does the actual work of moving data
- **Direct Memory Access (DMA)**
  - CPU delegates off the work to a dedicated DMA controller

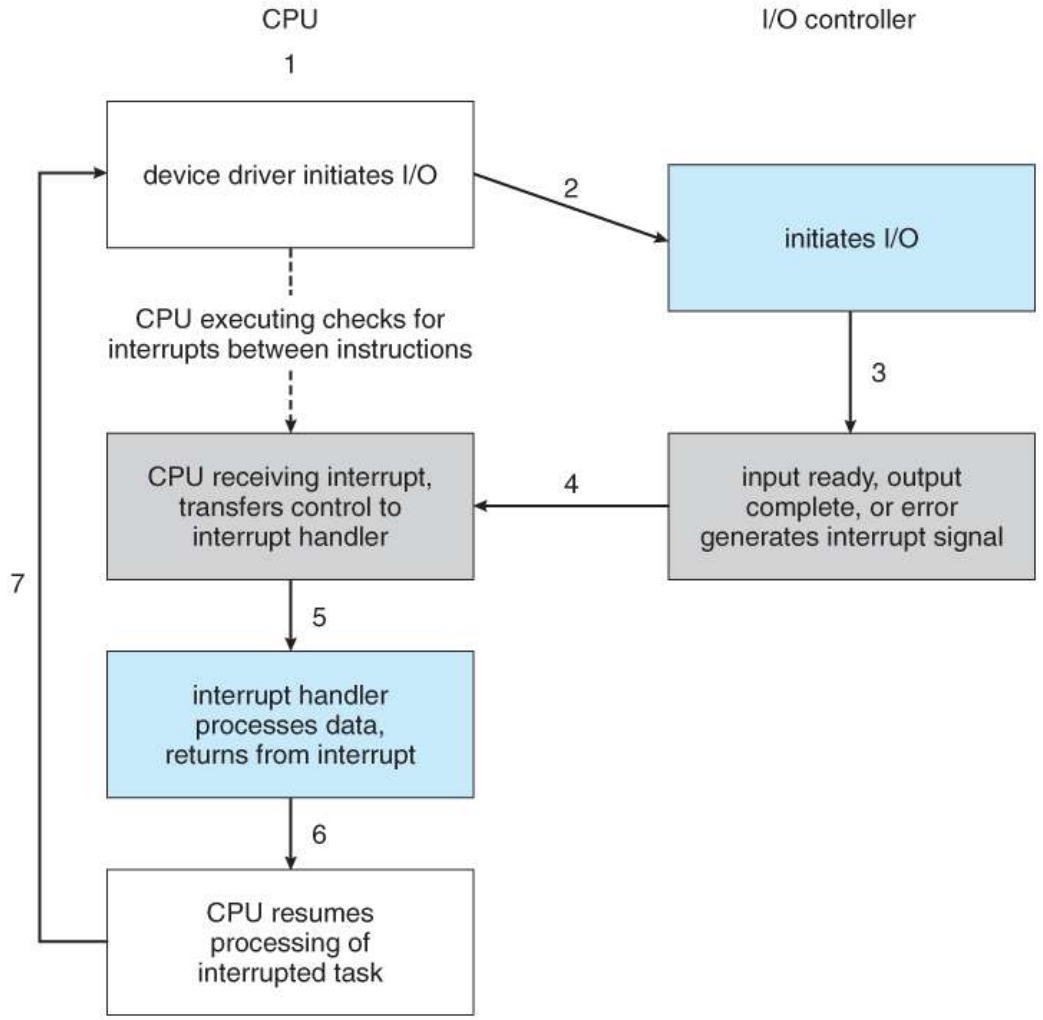
# Performing I/O Tasks

- **Polling**
  - CPU periodically checks for the I/O task status
- **Interrupt-driven**
  - CPU receives an interrupt from the controller (device or DMA) once the I/O task is done (either successfully or abnormally)

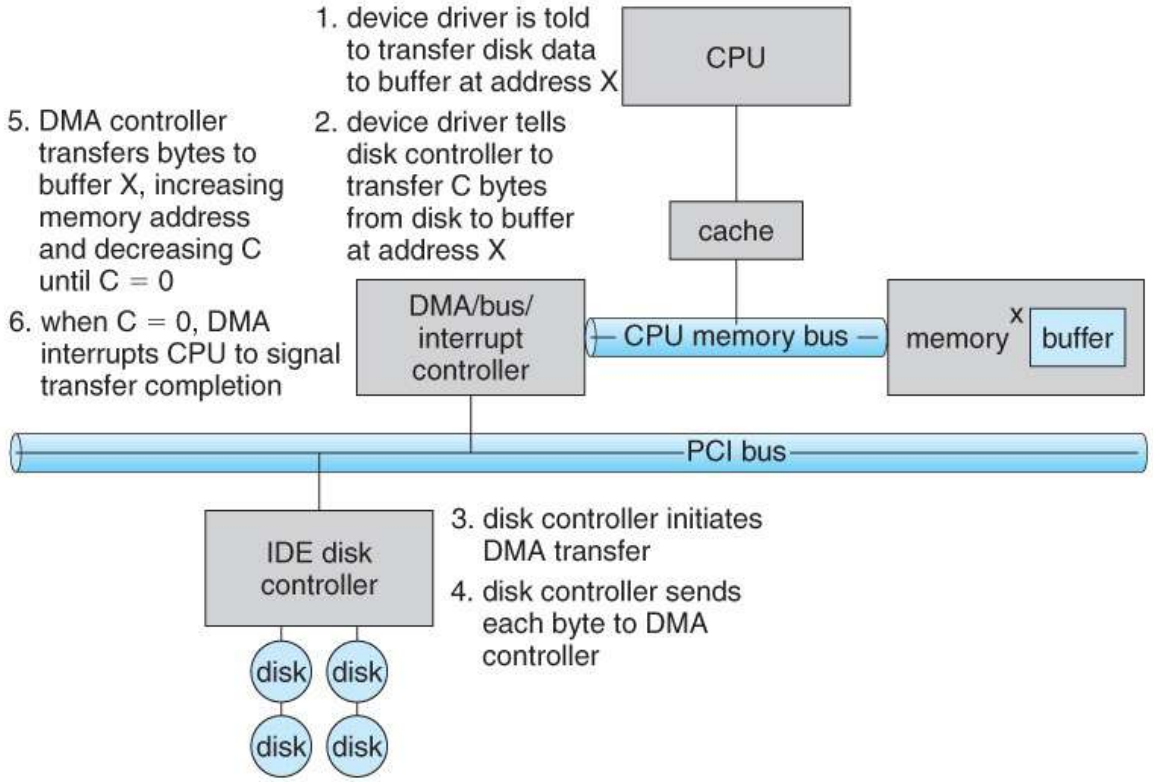
WHO?

- **Programmed I/O**
  - CPU does the actual work of moving data
- **Direct Memory Access (DMA)**
  - CPU delegates off the work to a dedicated DMA controller

# How: Interrupt-driven I/O



# Who: Direct Memory Access (DMA)




Overcome the limitation of Programmed I/O

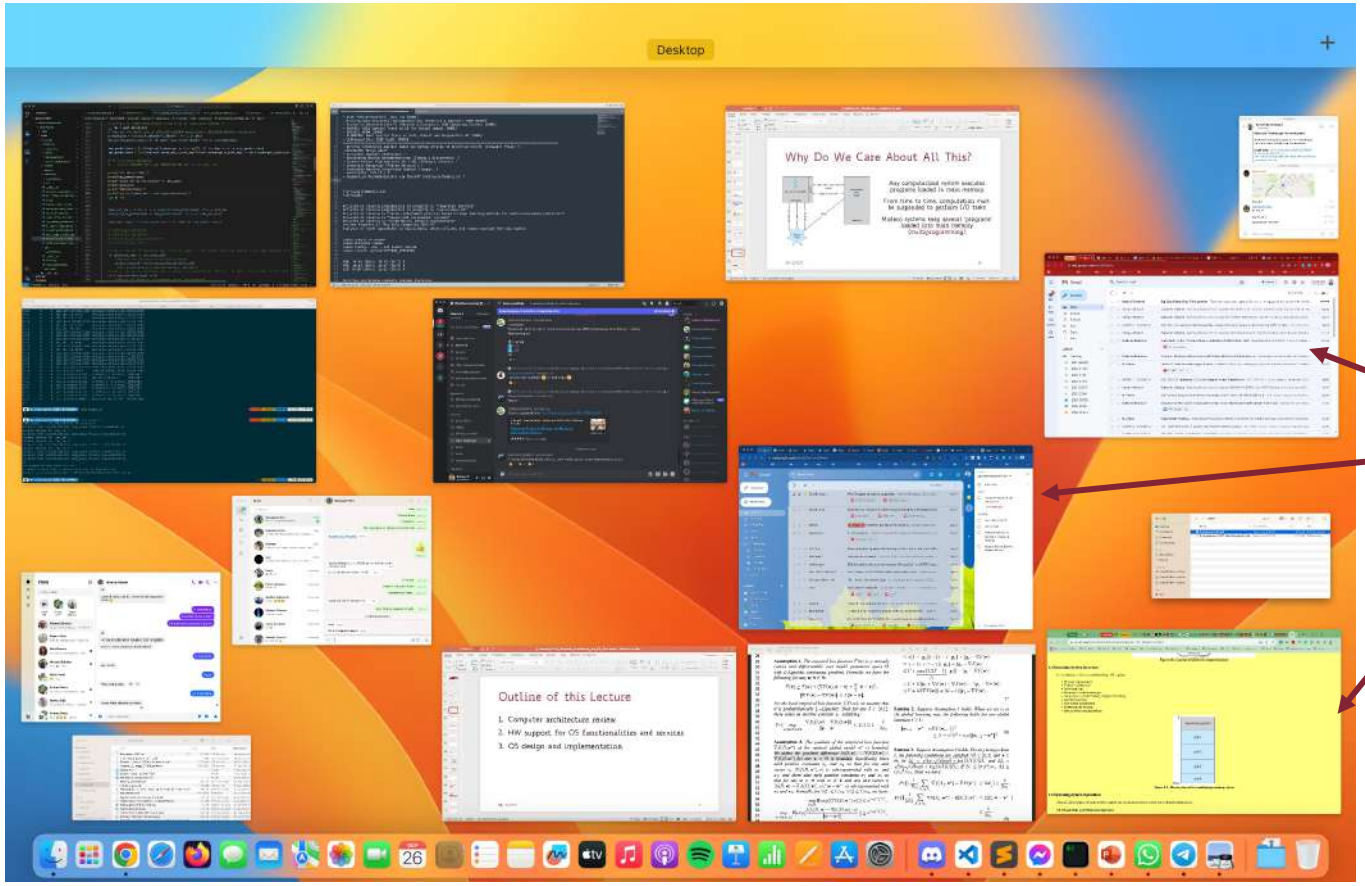
Maybe wasteful to tie up the CPU transferring data in and out of registers one byte at a time

Useful for devices that transfer large quantities of data (such as disk controllers)

Typically, used in combination with interrupt-driven I/O

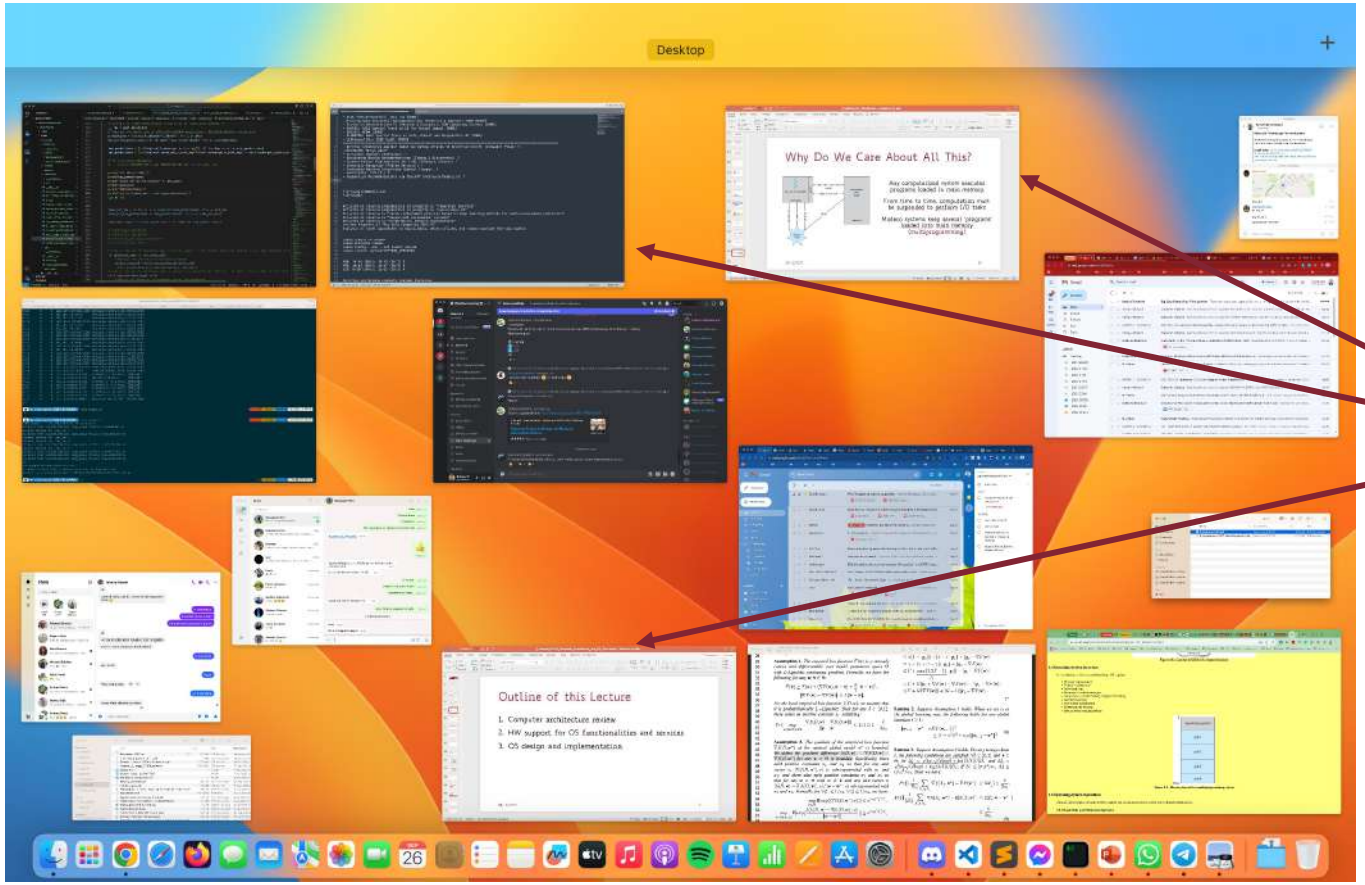
# Modern Computer Systems

# Many User Applications Running



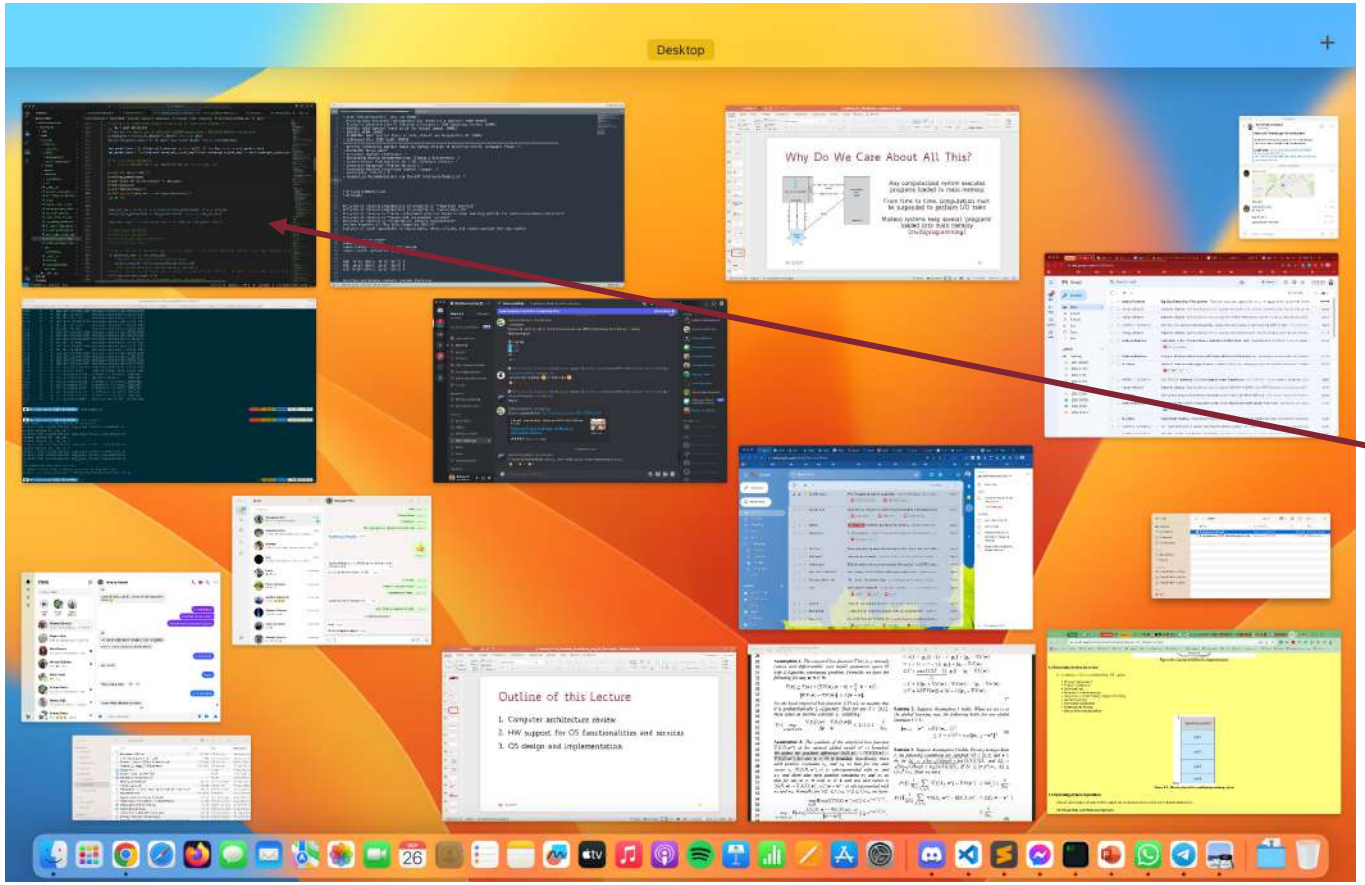

Web Browsers

# Many User Applications Running



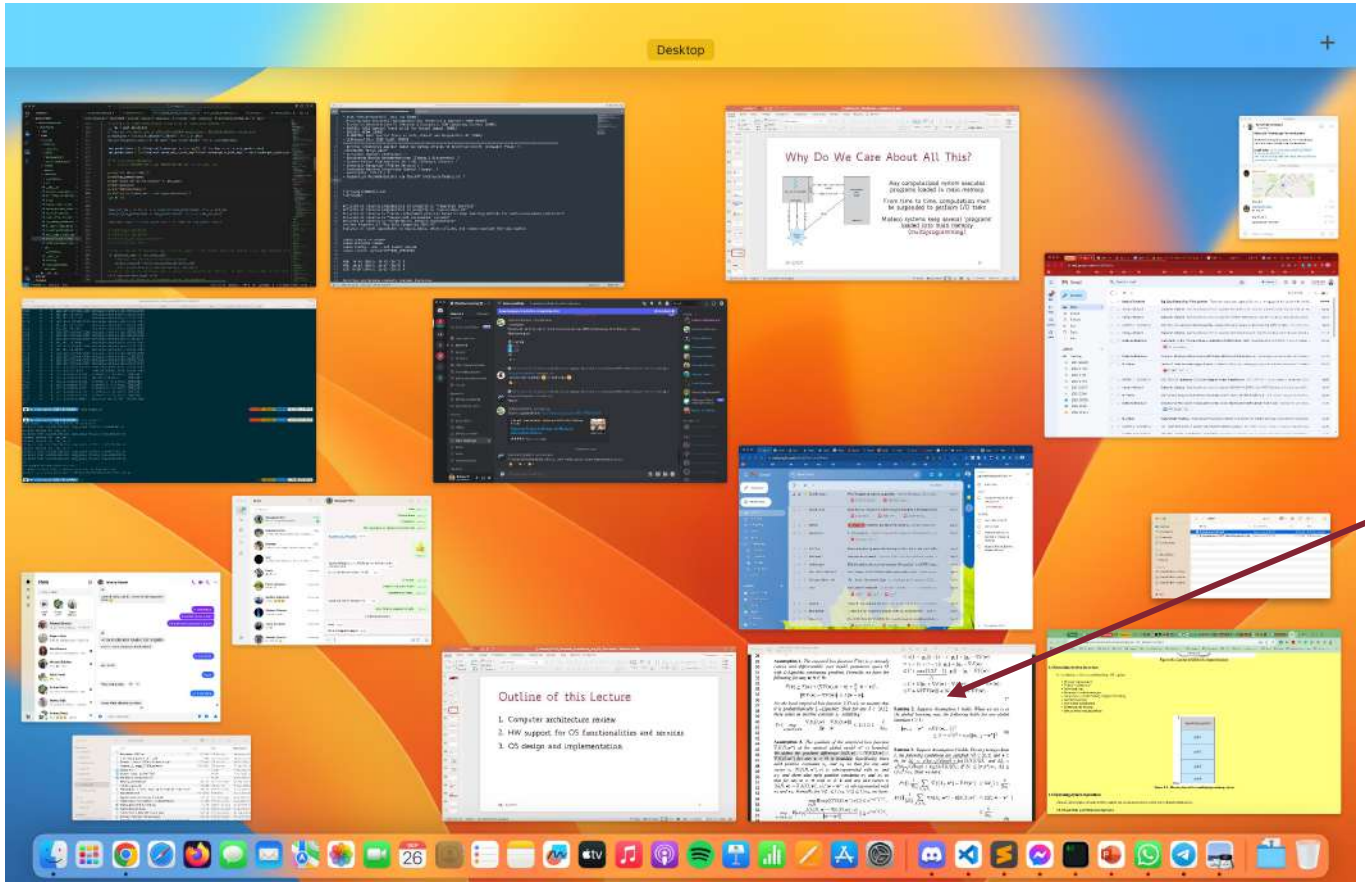

Text Editors

# Many User Applications Running



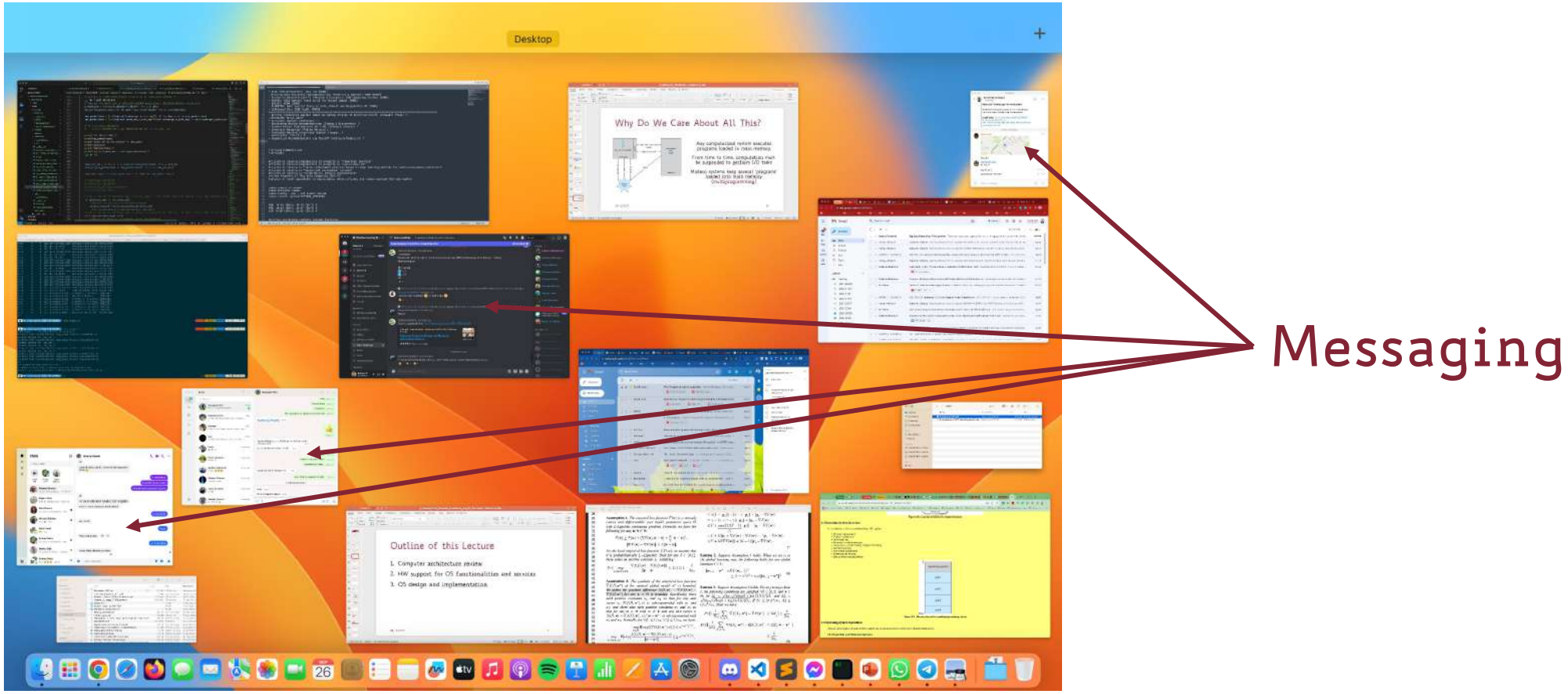

IDE
(code development)

# Many User Applications Running

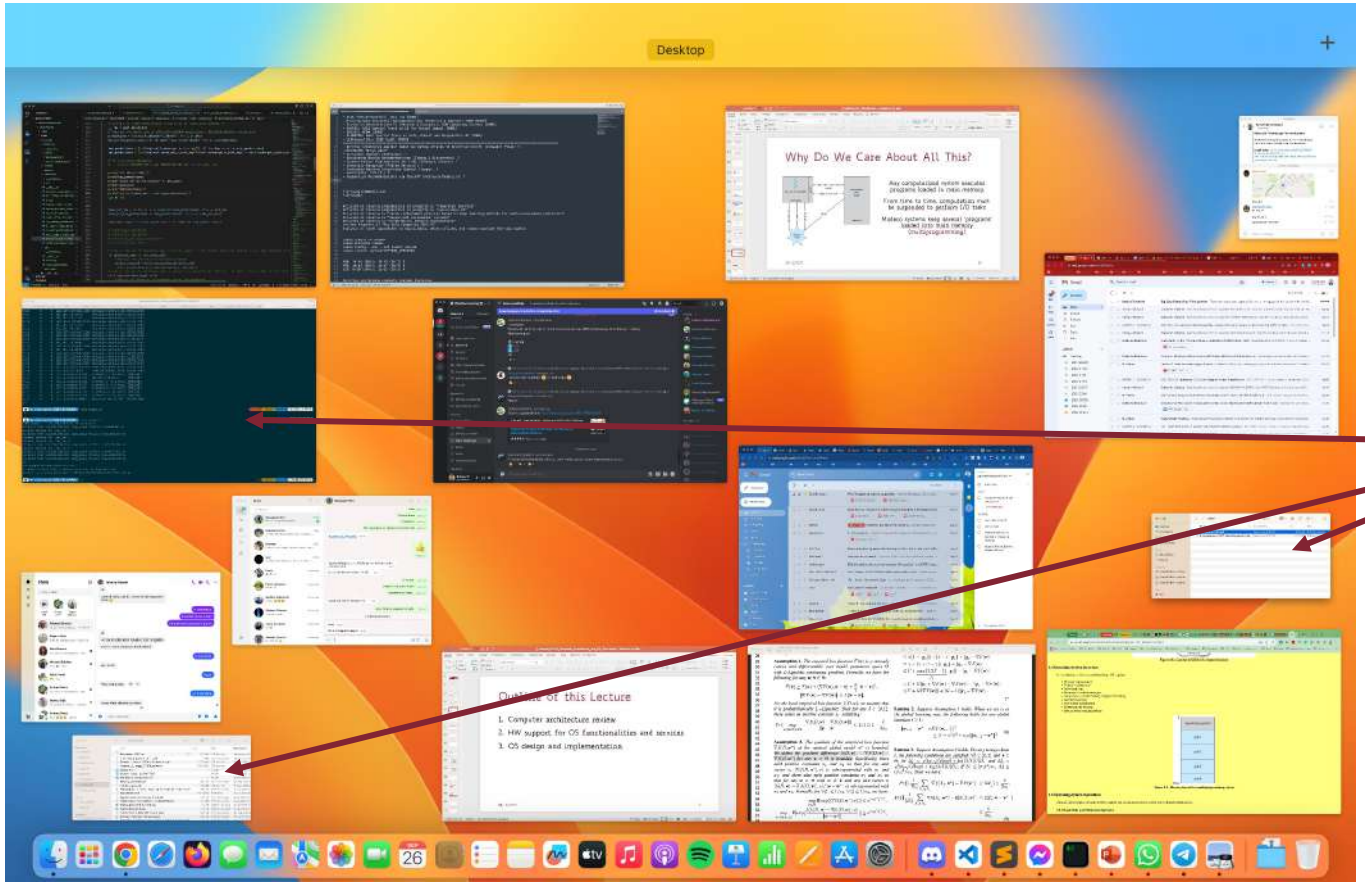



PDF Viewer

# Many User Applications Running

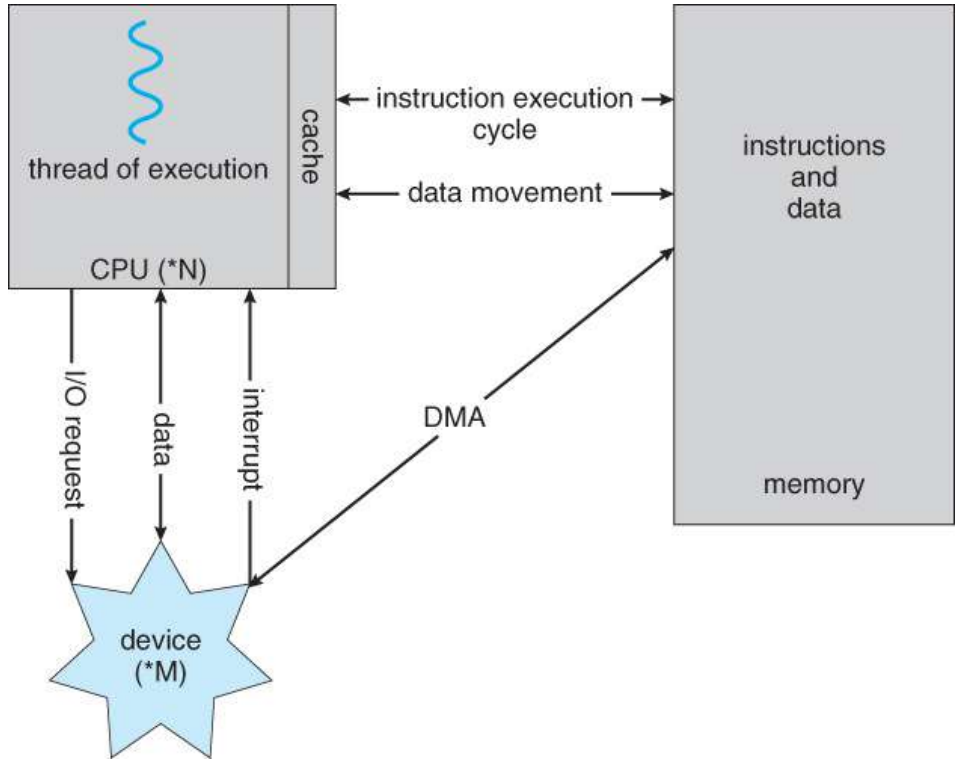


# Many System Programs Running



# Not Just Laptops/PCs…

# Why Do We Care About All This?

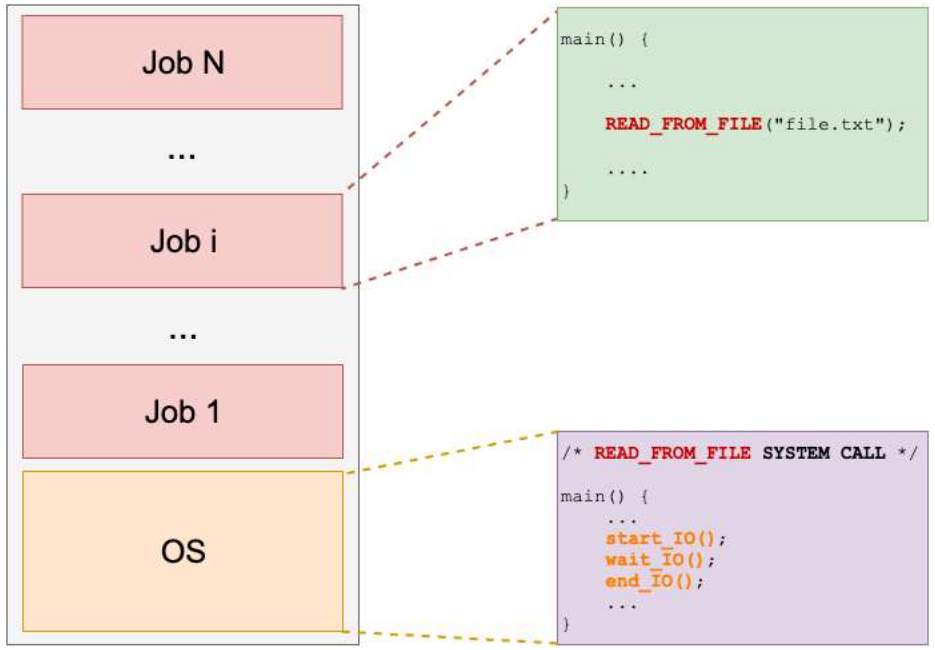



Any computerized system executes programs loaded in main memory

From time to time, computation must be suspended to perform I/O tasks

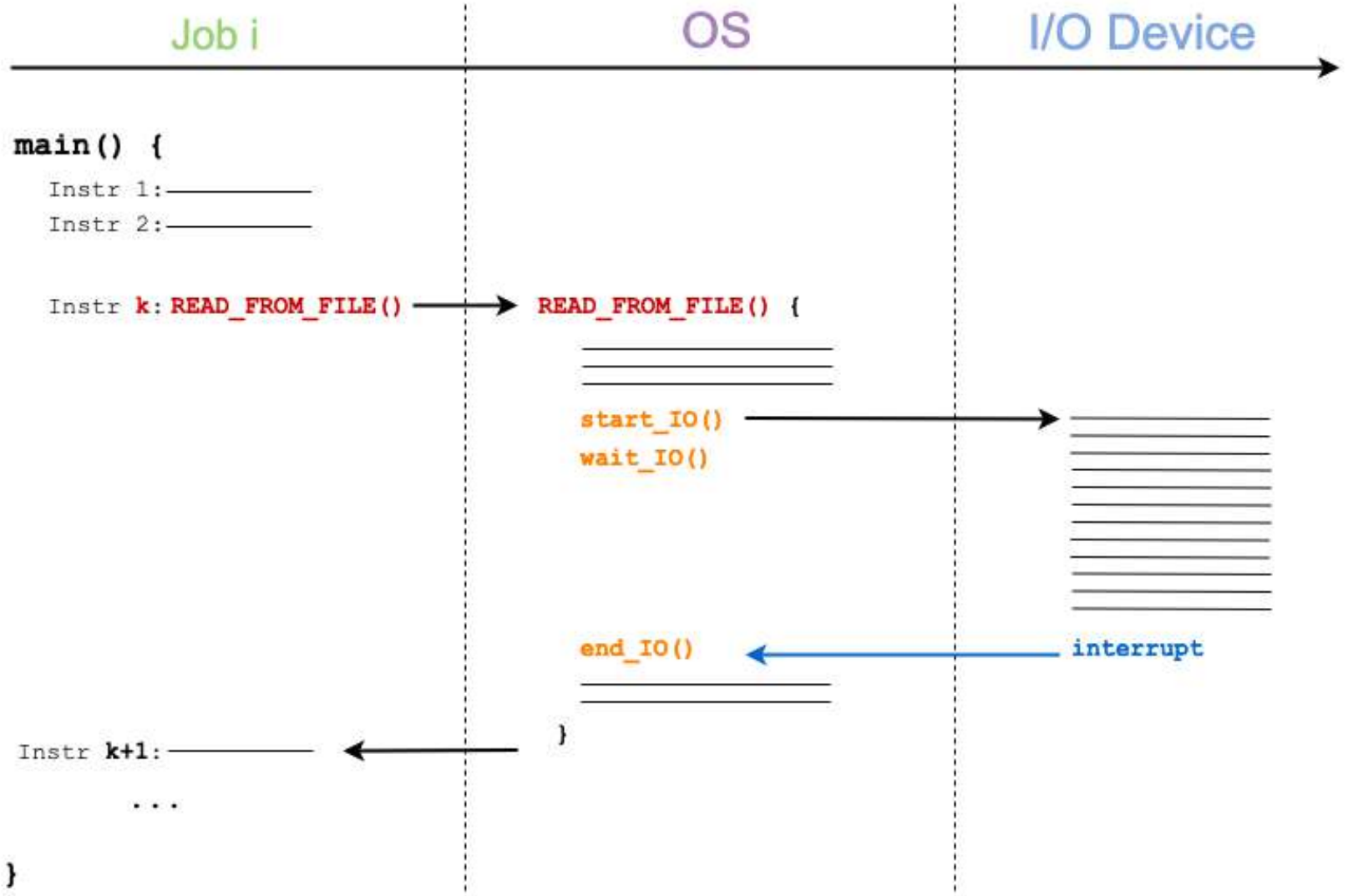
Modern systems keep several 'programs' loaded into main memory (**multiprogramming**)

# Multiprogramming Systems (1960s)

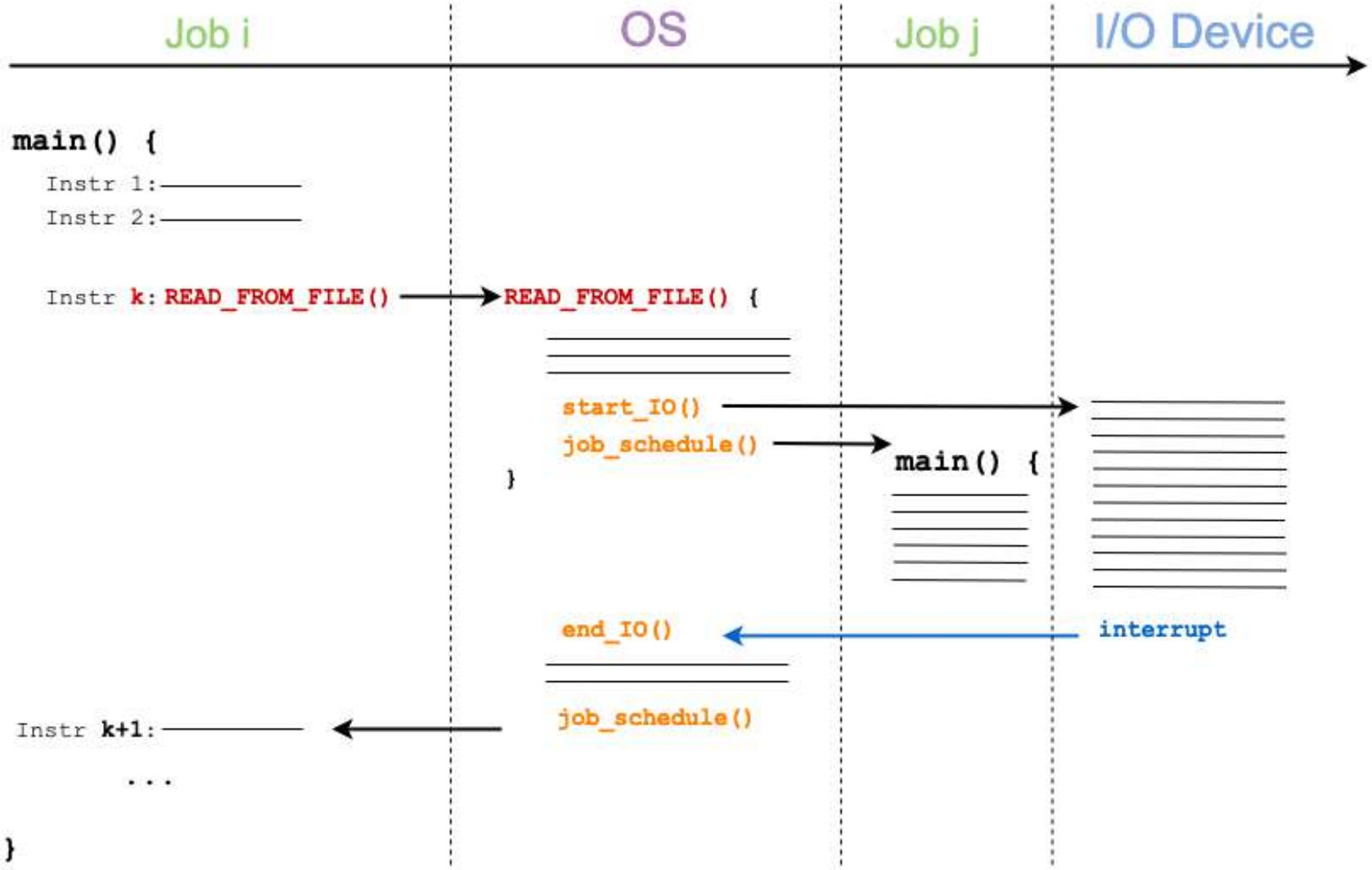



- Keep several jobs loaded in memory
- Multiplex CPU between jobs
- OS responsibilities:
  - job scheduling
  - memory protection
  - I/O operations
- **Problem:** CPU is left idle while blocking I/O operations take place

# Blocking System I/O



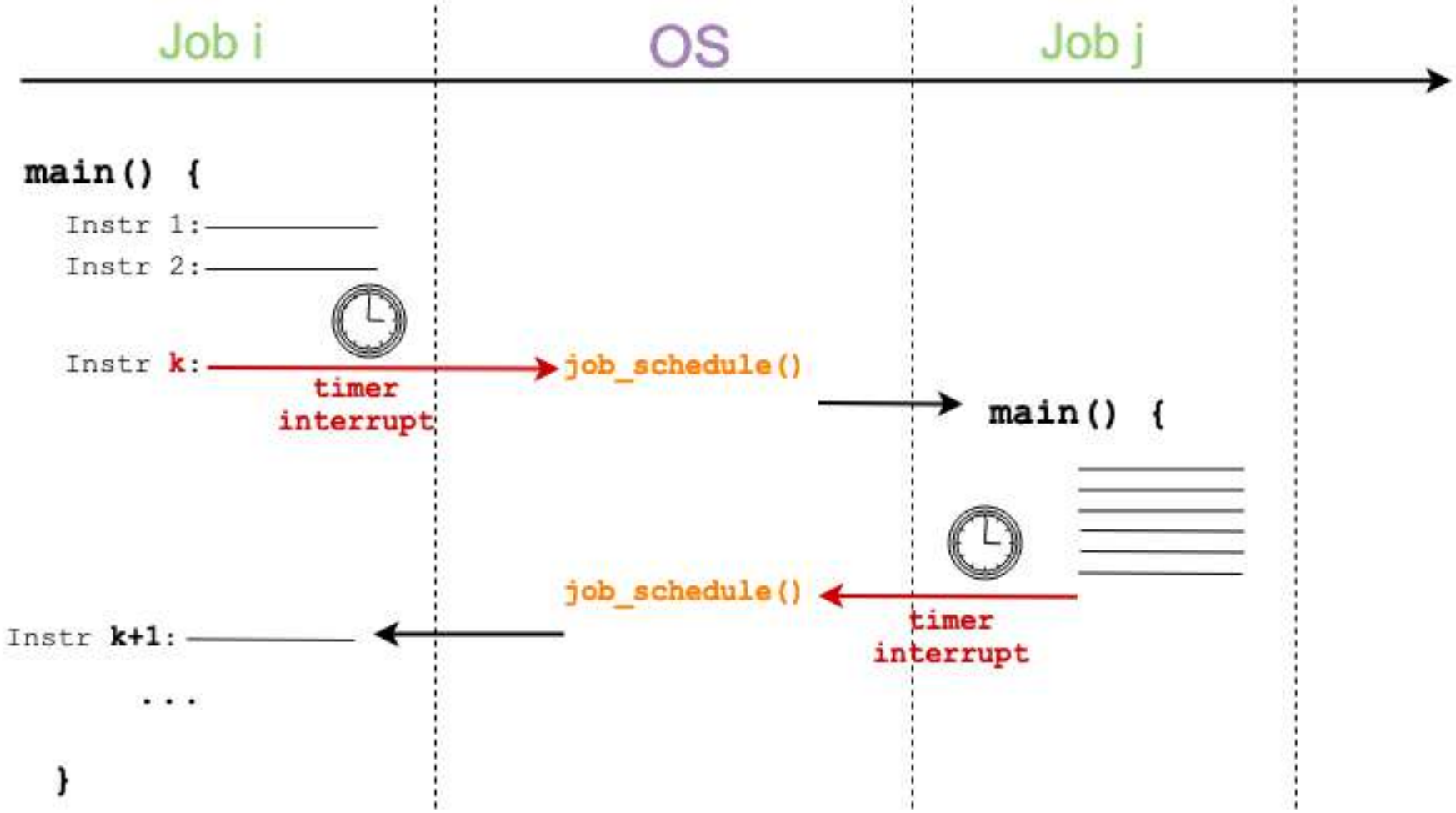
# Non-Blocking System I/O



# Time-sharing Systems (1970s)

- Many users connected to the same CPU via cheap consoles
- Timer interrupt used to multiplex CPU between jobs
- Illusion of parallelism (pseudo-parallelism)
- Ken Thompson&Dennis Ritchie → UNIX OS

# Pseudo-parallelism



# New Trends in OS Design

- Active field of research
  - OS demand is growing (many computing devices are available)
  - New application settings (Web, Cloud, mobile, cars, etc.)
  - Hardware is rapidly changing (new CPUs coming out)

# New Trends in OS Design

- Active field of research
  - OS demand is growing (many computing devices are available)
  - New application settings (Web, Cloud, mobile, cars, etc.)
  - Hardware is rapidly changing (new CPUs coming out)
- Open-source OS (Linux)
  - Allows developers to contribute to OS development
  - Excellent research platform to experiment with

# Why Study OSs?

- To learn important concepts of computer science
  - Abstraction
    - Virtualize any physical resource (CPUs, memory, I/O, etc.)

# Why Study OSs?

- To learn important concepts of computer science
  - Abstraction
    - Virtualize any physical resource (CPUs, memory, I/O, etc.)
  - **Systems Design Tradeoffs**
    - Performance vs. Cost of OS abstractions
    - Performance vs. Complexity of OS design
    - HW vs. SW implementation of key features

# Why Study OSs?

- To learn important concepts of computer science
  - Abstraction
    - Virtualize any physical resource (CPUs, memory, I/O, etc.)
  - Systems Design Tradeoffs
    - Performance vs. Cost of OS abstractions
    - Performance vs. Complexity of OS design
    - HW vs. SW implementation of key features
  - How computers work

# Large Computer Systems

- The world is increasingly dependent on computer systems
  - Large, complex, interconnected, distributed, etc.

# Large Computer Systems

- The world is increasingly dependent on computer systems
  - Large, complex, interconnected, distributed, etc.
- Huge demand for experts who deeply understand and can build such systems, which need to be:
  - Reliable, effective, efficient, secure, etc.

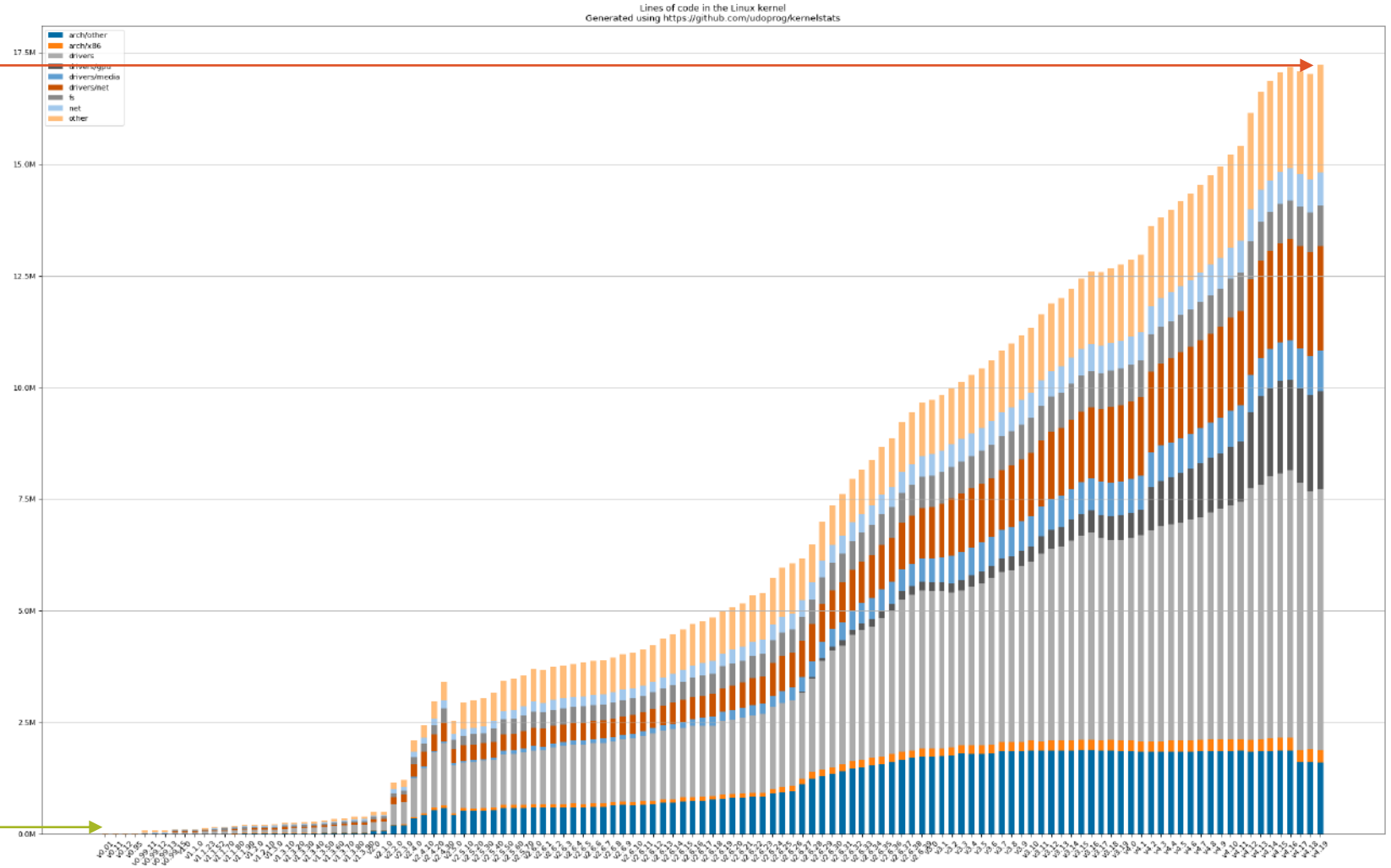
# Large Computer Systems

- The world is increasingly dependent on computer systems
  - Large, complex, interconnected, distributed, etc.
- Huge demand for experts who deeply understand and can build such systems, which need to be:
  - Reliable, effective, efficient, secure, etc.

**OS is a great example of a large computer system**

# Linux Kernel Size (Lines of Code)

~17.5M




~200k

# OS as Large Computer System

- Designing large computer systems requires you to know
  - Each computer:
    - Architectural details
    - High-level programming language (mostly, C/C++)
    - Memory management
    - Concurrency and scheduling
    - File system and I/O

# OS as Large Computer System

- Designing large computer systems requires you to know
  - Each computer:
    - Architectural details
    - High-level programming language (mostly, C/C++)
    - Memory management
    - Concurrency and scheduling
    - File system and I/O
  - **Across clusters of computers:**
    - Server architectures
    - Distributed file systems and computing frameworks

# OS Design Issues (1)

- **Structure** → How the whole system is organized
- **Concurrency** → How parallel tasks are managed
- **Sharing** → How resources are shared
- **Naming** → How resources are identified by users
- **Protection** → How critical tasks are protected from each other
- **Security** → How to authenticate, authorize, and ensure privacy
- **Performance** → How to make it more efficient (quick, compact)

# OS Design Issues (2)

- **Reliability** → How to deal with failures
- **Portability** → How to write once and run anywhere
- **Extensibility** → How to add new features/capabilities
- **Communication** → How to exchange information
- **Scalability** → How to scale up as demand increases
- **Persistency** → How to save task's status
- **Accounting** → How to claim on control resource usage

# Architectural Trends: CPU

*Million Instructions Per Second

**1 MHz = 1,000,000 clock cycles per second

| | 1971 (Intel 4004) | Today (Intel Core i9) | Δ (orders of magnitude) |
|---|---|---|---|
| MIPS* | ~0.09 | ~400,000+ | +7 |
| Instructions (fetch, decode, execute) per clock cycle | ~0.12 | ~100+ | +3 |
| Clock frequency (MHz)** | 0.74 | ~5,000 | +4 |
| Cheap size (μm) | 10 | 0.014 | -3 |

# Architectural Trends: CPU

*Million Instructions Per Second

**1 MHz = 1,000,000 clock cycles per second

| | 1971 (Intel 4004) | Today (Intel Core i9) | Δ (orders of magnitude) |
|---|---|---|---|
| MIPS* | ~0.09 | ~400,000+ | +7 |
| Instructions (fetch, decode, execute) per clock cycle | ~0.12 | ~100+ | +3 |
| Clock frequency (MHz)** | 0.74 | ~5,000 | +4 |
| Cheap size (μm) | 10 | 0.014 | -3 |

**Moore's law:** the number of transistors in a dense integrated circuit doubles about every two years

# Architectural Trends: Main Memory

| | 1973 (DEC PDP-8) | Today (Samsung DDR4) | Δ (orders of magnitude) |
|---|---|---|---|
| Capacity (kB) | 12 | 128,000,000 | +7 |
| Cost ($/MB) | ~400,000 | ~0.005 | -8 |

# Architectural Trends: Disk

| | 1956 (IBM RAMAC 305) | Today (Western Digital) | Δ (orders of magnitude) |
|---|---|---|---|
| Capacity (MB) | 5 | 15,000,000 | +7 |
| Size (inch) | 24 (x50) | 3.5 | -3 |
| Cost ($/MB) | 640 (per month) | ~0.000018 | -9 |

# What's Next?

- Moore's law has hit its limit(?)
  - chip size has physical constraints
  - power vs. heat tradeoff
  - alternatives have already pushed forward the end of it:
    - multicore-manycore processors
  - other approaches are subject of research:
    - molecular/DNA transistors
    - quantum computing

# Summary

- Basic roles of an Operating System

# Summary

- Basic roles of an Operating System
- A brief recap of how computer systems are organized

# Summary

- Basic roles of an Operating System
- A brief recap of how computer systems are organized
- Operating Systems as large and complex computer systems

# Summary

- Basic roles of an Operating System
- A brief recap of how computer systems are organized
- Operating Systems as large and complex computer systems
- New architectural trends open up novel opportunities and challenges in Operating System design