Operating Systems

(Sistemas Operativos)

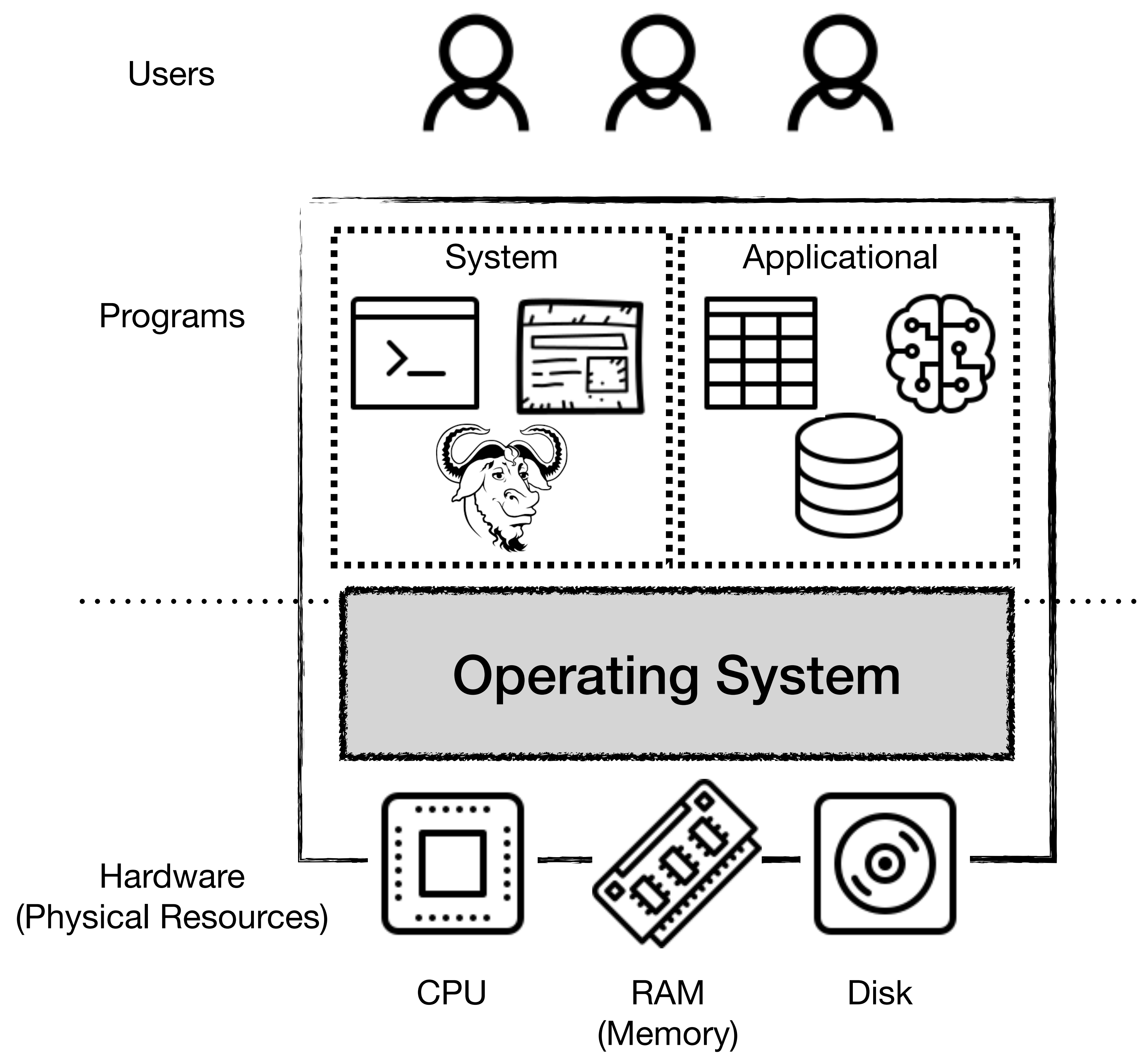
Introduction to Operating Systems

University of Minho

2024-2025

Modern Computational Systems

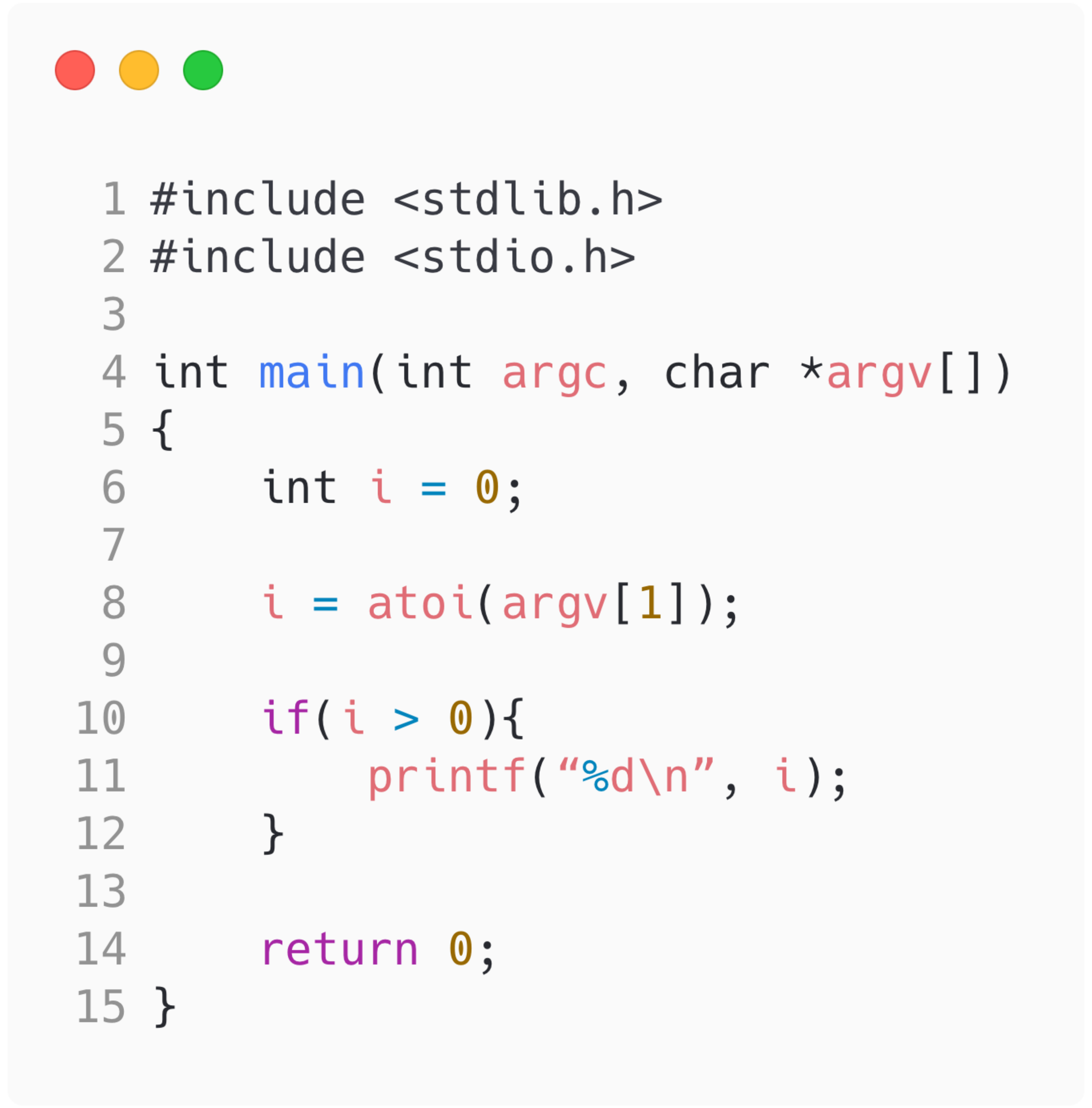
- Users
  - Humans or other computers (e.g., remote connection)
- Applicational Programs
  - Generic programs such as text editors, spreadsheets, databases, AI engines, games, …
- System Programs
  - Specialized programs such as shell (e.g., bash), graphical user interfaces (GUI), compilers, libraries
- Operating System
  - Manages the interaction between users/programs and the hardware
- Hardware
  - CPU, memory, I/O devices (e.g., disk, keyboard, display)



Operating System

Programs

- The OS main clients are **programs**
  - A running program executes instructions (billions per second in some modern programs)
- What happens when a program runs?[1]
  - The processor (CPU) **fetches** an instruction from memory
  - The instruction is **decoded** and **executed** (e.g., sum numbers, access memory, check conditions)[2]
  - This cycle is repeated until all instructions are done, i.e., the program ends
  - The CPU has a small number of **registers** to hold temporary info (e.g., Program Counter (PC), Instruction Register (IR))



```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    i = atoi(argv[1]);

    if(i > 0){
        printf(“%d\n”, i);
    }

    return 0;
}
```

[1] Simplified version of the world, modern processors can execute multiple instructions simultaneously
[2] “Complex” programs will sometimes require loading or storing data from memory, accessing devices (e.g., reading or writing to disk)

Operating System
Purpose

- The main goal of the OS is to make it **easy and efficient to run programs**
  - Enabling multiple programs to run simultaneously
    (with limited hardware resources)
  - Enabling programs to share memory and to access I/O devices (e.g., disk)
  - Efficiently managing the physical resources used by programs

But, before we discuss how modern OSs achieves all these goals, let us first understand how the field has evolved

Evolution of Operating Systems

At the beginning (1954)

- ◉ Computers (mainframe systems) were operated manually by users
  - ‣ There was a common set of libraries (e.g., for abstracting low-level I/O code)
  - ‣ Usually, one program ran at a time with each user having an assigned time slot
- ◉ Highly inefficient and costly

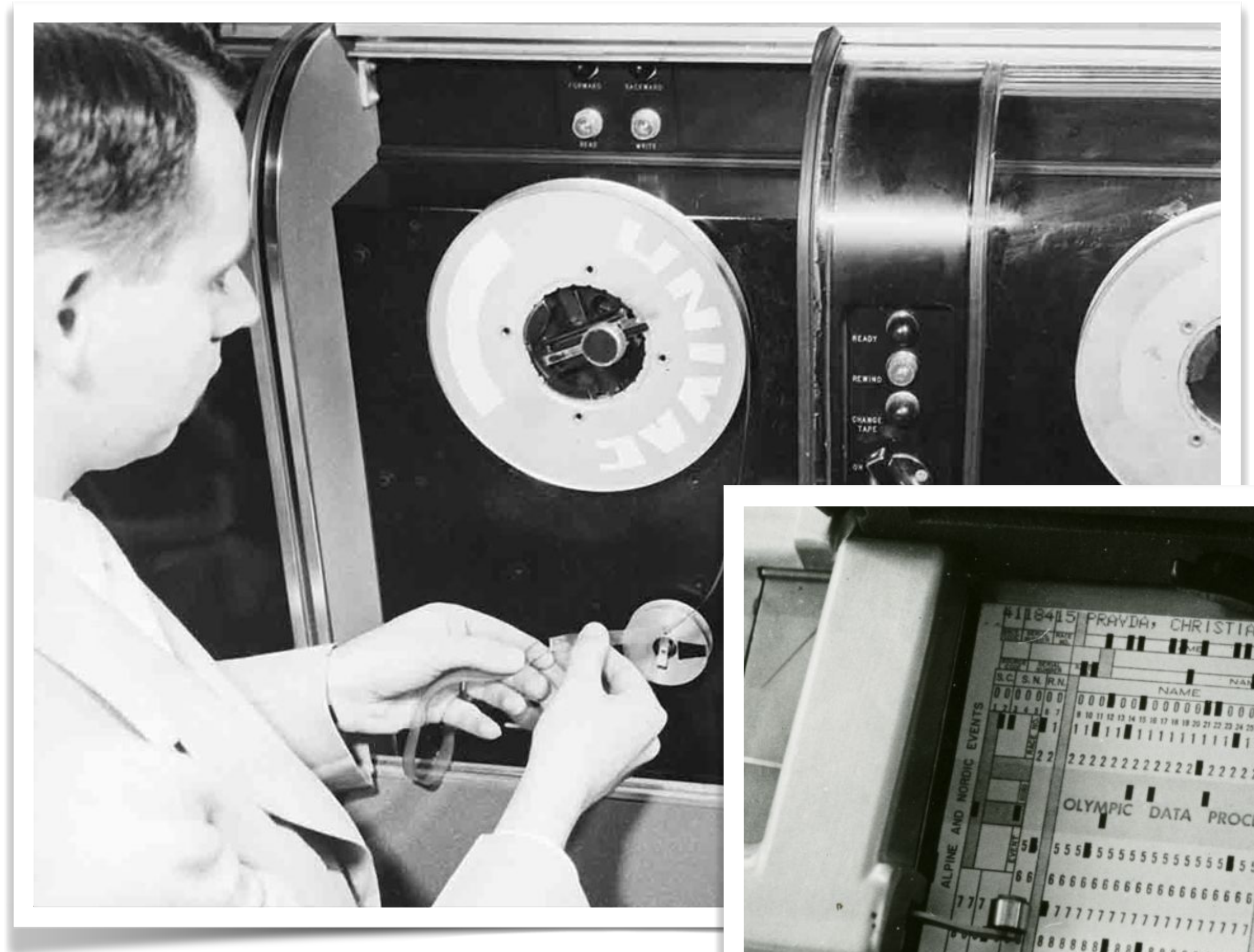
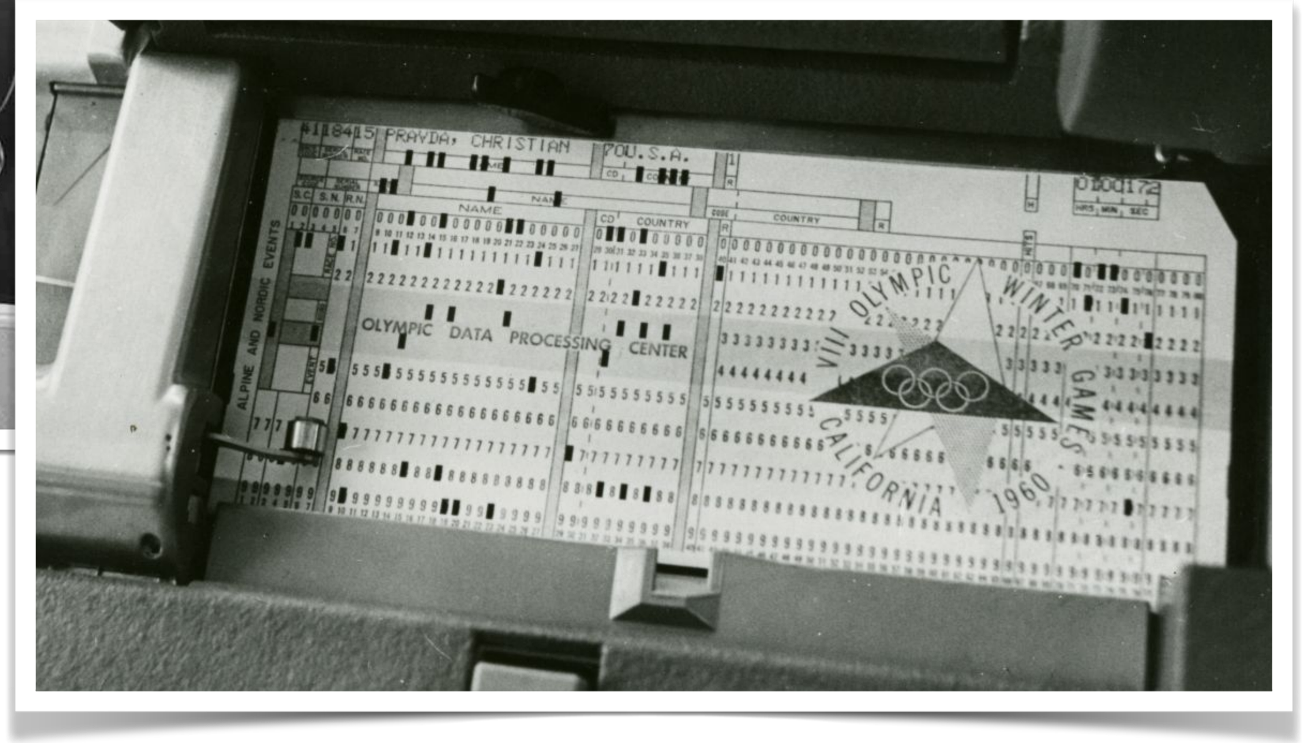
"Each user was allocated a minimum 15-minute slot, of which time he usually spent 10 minutes in setting up the equipment to do his computation"
Quote by George Ryckman regarding the IBM 701 open shop computer

The cost of the wasted computer time was $146,000 per month, in 1954!

Evolution of Operating Systems

The Operator (1959)

- Users were removed from the computer's operation
  - Users prepared their programs on punched cards to be delivered to an operator
  - The operator (still a human) would load the programs into magnetic tapes
  - Programs executed sequentially (as stored in the tape) - batch processing
- More efficient but less convenient for users
  - Dependent on the scheduling order decided by the operator
  - Cannot interact with the running program (e.g., for debugging)

Evolution of Operating Systems

Control Program (1960s)

- What if tasks done by the operator could be automated?
  - A special **control program** ("OS") could manage the computer resources, schedule user programs (jobs) and execute them
  - Users (or the operator) would just load / unload programs (manage I/O)
- What if the control program loses control?
  - A job enters an infinite loop and takes control of the processor
  - A job erroneously writes to the memory region of the control program
  - A job leads to errors when accessing I/O devices

Evolution of Operating Systems

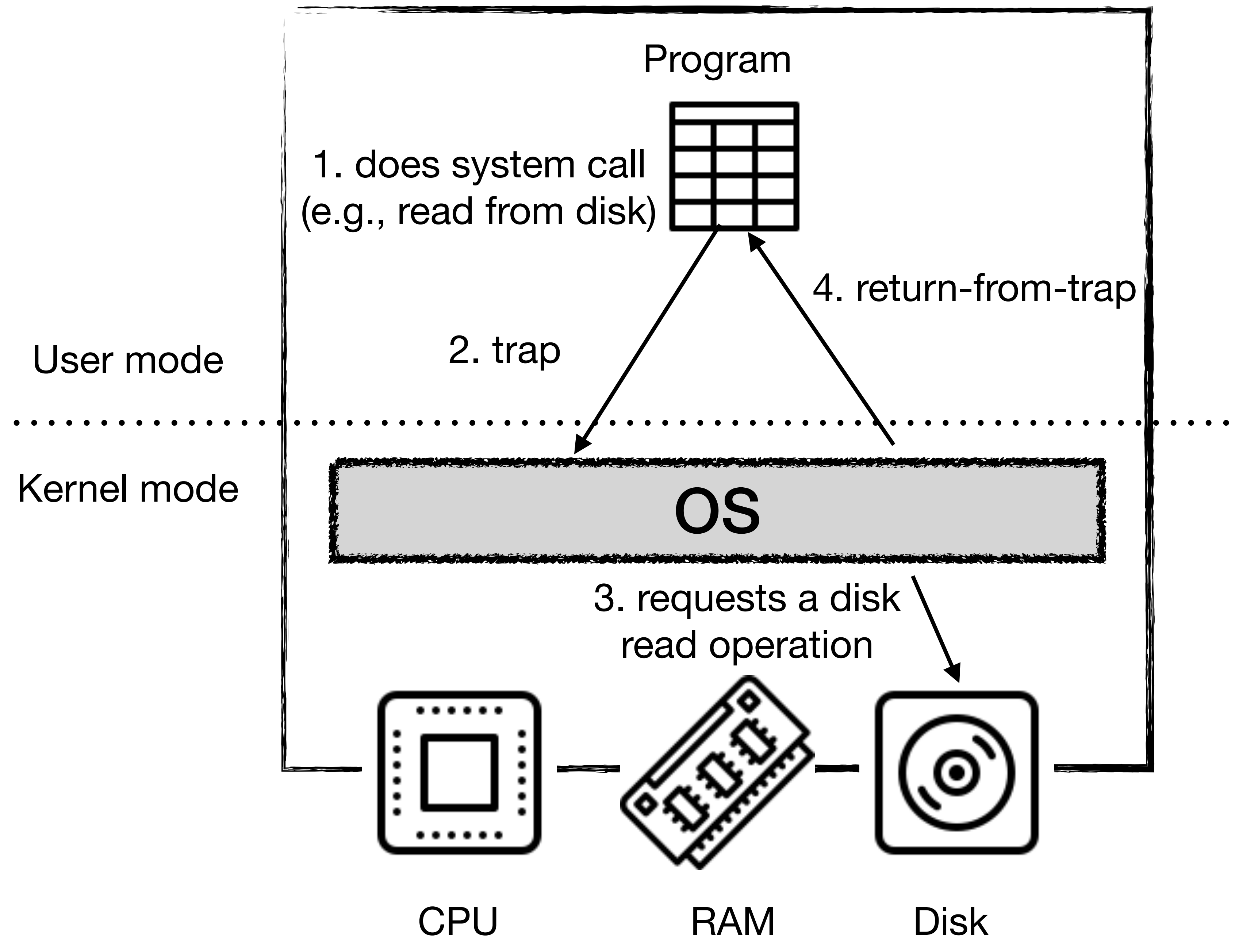
Control Program (1960s)

- The previous challenges are only addressable with the evolution of hardware
- Hardware interrupts:
  - User programs can be interrupted, even if these are on an infinite loop, giving control back to the control program
- Hardware privilege levels
  - Programs run in **user mode** (unprivileged)
  - The control program (OS) runs in **kernel mode** (privileged)
  - Privileged operations (e.g., memory, disk, network operations) only done in kernel mode
- Memory Protection
  - Avoids programs to erroneously access the control program's memory and destroy it

Evolution of Operating Systems

System Call (1960s)

- The idea of system call is born with the Atlas computing system!
- When programs, running in **user mode**, must do privileged operations (e.g., access disk, allocate memory) they issue system calls
- When a system call is initiated (**trap** hardware instruction), the hardware raises the privilege level to kernel mode
- In **kernel mode** the OS does the privileged operations
- After serving the request, the OS calls a **return-from-trap** instruction, which reverts to **user mode** and returns the control to where the program left off

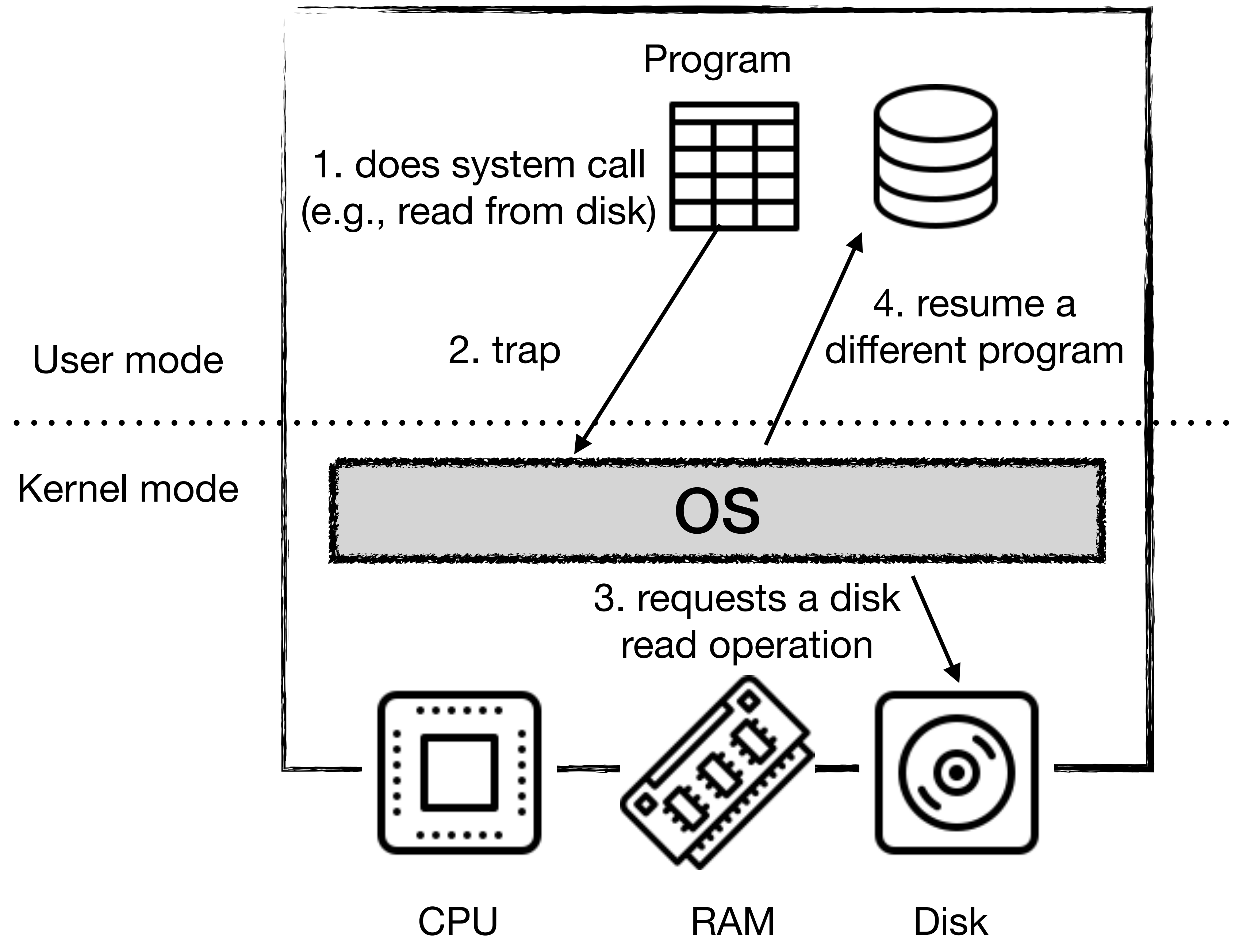


Do not worry, we will discuss these mechanisms further in the next classes!

Evolution of Operating Systems

Multiprogramming and Time-sharing (1960/70s)

- With the evolution of hardware, the OS can now load multiple programs to memory and rapidly switch between them
- However, I/O operations are slow… Programs waiting on the CPU for I/O requests to complete are wasting CPU cycles
- Switch programs waiting for I/O - **multiprogramming!**
  - After handling a system call (e.g., instructing the disk to read data), the OS switches to another program (i.e., instead of waiting for data to be read)
  - When data is read from disk and placed in a memory buffer, the OS is notified with the help of an hardware **interruption** (and can then resume the program that issued the system call)


Program
1. does system call (e.g., read from disk)
2. trap
4. resume a different program
User mode
Kernel mode
OS
3. requests a disk read operation
CPU
RAM
Disk

Do not worry, we will discuss these mechanisms further in the next classes!

Evolution of Operating Systems

Multiprogramming and Time-sharing (1960/70s)

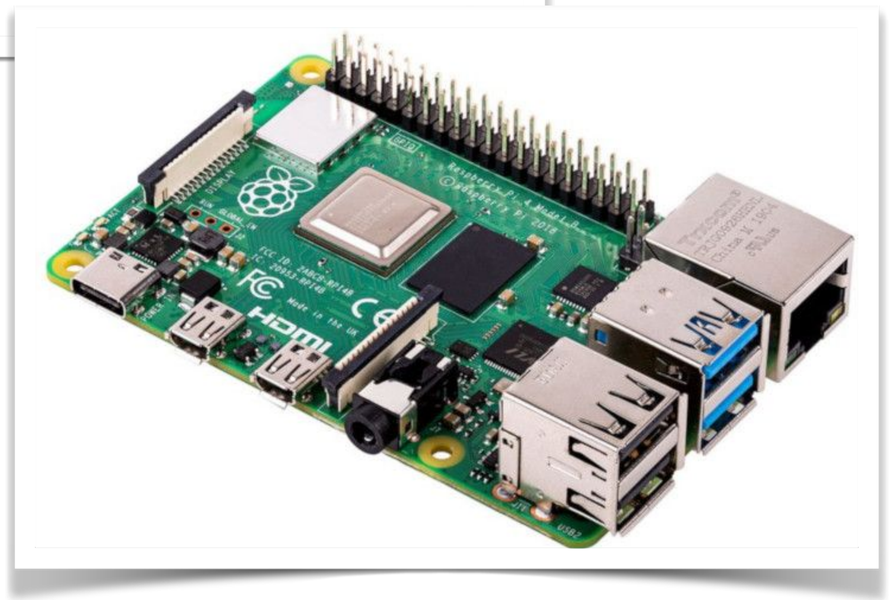
- The minicomputer emerges, it is more affordable and usable by a smaller group of users
  - CPU time is now **time-shared** across jobs from several users (with the aid of the OS). **UNIX** is born!
- In **multiprogramming**, a program holds the CPU until I/O is required.
  What if the program never does I/O?
- **Time-sharing** (or **multitasking**), is a logical extension of multiprogramming in which programs are frequently switched (with the aid of an hardware **timer interrupt**)
- Users are given the illusion of having the machine only for themselves (lower response times)
- New interesting problems arise
  - Efficient scheduling of processes
  - Memory isolation across running processes
  - Concurrency

Do not worry, we will discuss these mechanisms further in the next classes!

Evolution of Operating Systems

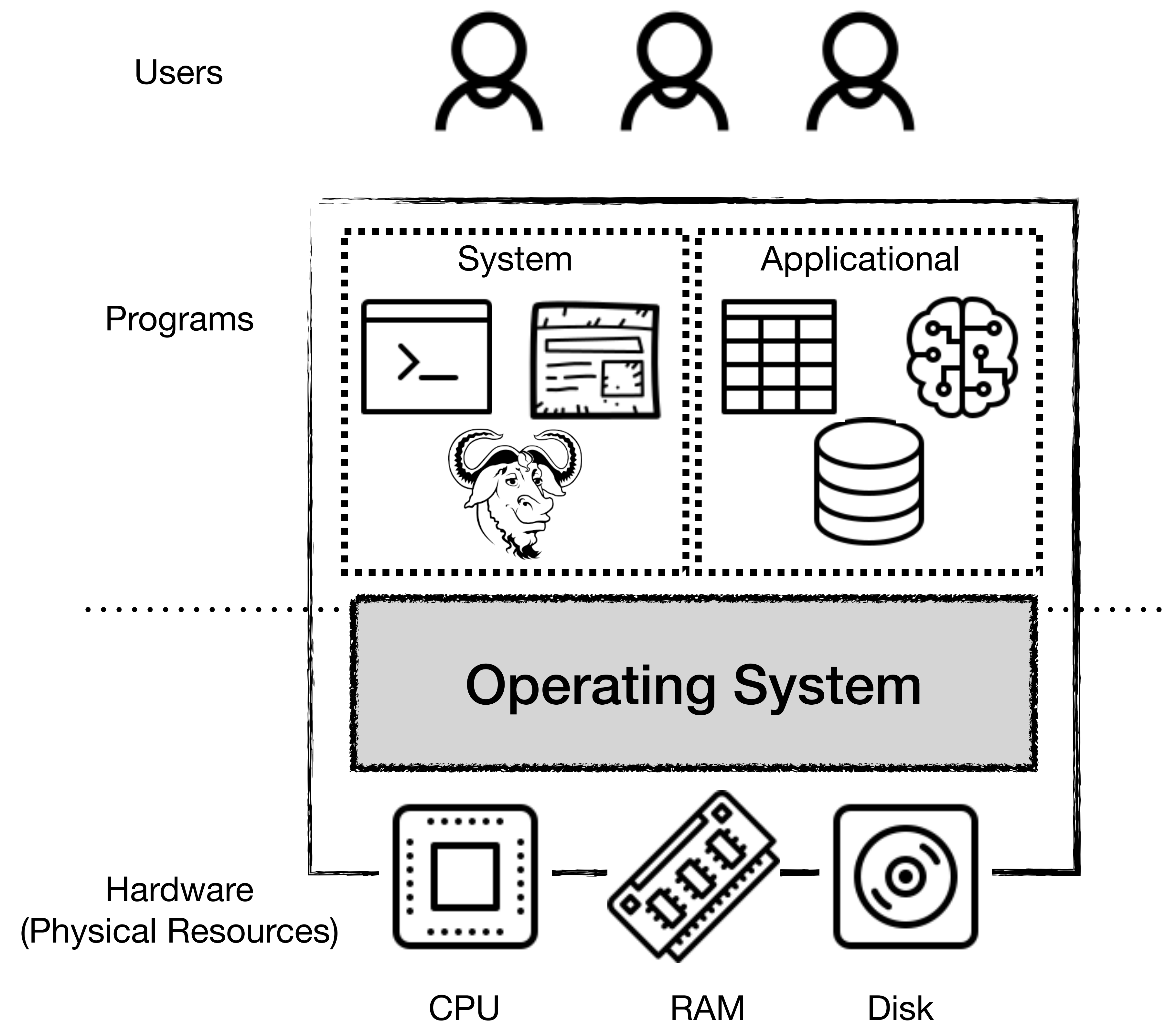
Personal Computers (1970s)

- Lower costs now enable each user to have its own computer (more convenient!)
- Unfortunately, OSs took a step back in efficiency and protection
  - DOS had several memory protection issues
  - MAC OS early versions freeze due to programs with infinite loops
- Later, the interesting ideas of minicomputers (adapted versions of UNIX) started to be ported into PCs.
  - LINUX is born!
- And then came the web, distributed systems, smartphones, IoT devices…



Modern Operating Systems

Let us get back from our “not so little” DETOURS



Modern Operating Systems

Design Principles

- **Abstract** the “low-level” details of hardware
  - Virtualizes physical resources, abstracting their complexity and heterogeneity (e.g., hardware architectures, disk models, GPU models)
  - E.g., Linux runs under multiple hardware architectures
- Provide an **interface** (System Call API) for users to tell the OS what to do
  - To run programs: *fork()*, *exec()*, …
  - Manage memory: *brk()*, …
  - Access devices: *open()*, *read()*, *write()*, …

These design principles provide Convenience to users!

System calls

Yet another detour

- On top of the OS one can build several interesting system programs
  - Graphical user interfaces (GUI)
  - Command line interfaces (shell)
  - Tools to compile, load, execute and debug programs
  - Tools to manipulate and explore files, directories
  - …
- Most users see the OS as the combination of its kernel (core code) and system programs, not realizing the existence of the system call API
  (It is debatable if system programs can be considered part of the OS)
- Fortunately, in this course, we will look into core concepts of the OS's kernel and will work with the system call API!
  - Crucial for anyone wishing to build better applications and to understand their systems

Modern Operating Systems

Design Principles

- Ensure **performance** when running multiple programs accessing shared physical resources
  - What program should be scheduled next? What program should write data to disk next?
  - **Important:** the OS must take decisions quickly and without excessive resource usage overhead
- Ensure **protection** by **isolating** running programs from each other and from the OS
  - A malicious/buggy program cannot access private memory regions from other programs (and OS)
  - Also, it should not be able to overtake the CPU and "block" the computer
- Other principles
  - **Reliability:** A failure of the OS means that all applications will fail
  - **Security:** An extension of the protection principle against malicious programs (e.g., ransomware)
  - **Energy efficiency, mobility, …**

These design principles provide Efficiency to users!

What will we learn?

CPU Management

- Although a computer has a limited number of CPUs, users get the impression that several programs (a lot more than the available CPUs) are running simultaneously.
  - How does the OS provide this illusion?
  - How can users use the OS APIs to start and stop programs?
  - How does the OS choose what programs will actually be running at each moment?

- We will study abstractions, mechanisms and policies employed by the OS
- The separation between mechanism and policy is a common design paradigm used by the OS For example, in the context of CPU management
  - **Mechanism**: how does the OS switch running programs?
  - **Policy**: which program should be scheduled next?

What will we learn?

Memory Management

- Physical memory can be seen as a simple array of bytes. Programs (running) need to store their instructions and data structures in memory
  - How does the OS abstract and manage memory accesses made by programs?
  - How is it possible to have multiple programs safely sharing the same physical memory?
  - How does the OS provide the illusion of having more memory, than the physically available one, for running programs?

- We will study structures, mechanisms and policies of the OS that answer such questions

What will we learn?

Persistence

- Memory is volatile, meaning that data is lost upon a reboot (or crash) of the computer. Disk devices (e.g., HDDs, SSDs) allow programs to store their data persistently
  - How does the OS abstract the access (API) to the low-level device drivers of these disks?
  - How does the OS make disk operations efficient when multiple programs use the same device simultaneously?
  - We typically access our data using a file-based abstraction (files, folders, …).
    How does the OS enable such abstraction?

- We will study the properties and design of disk devices and file systems to answer these questions

More Information

- **Chapter 2** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Brinch Hanse. **The evolution of Operating Systems.** In *Classic Operating Systems: From batch processing to distributed systems*. Springer-Verlag, 2000.

Questions?