Operating Systems

BCS1110

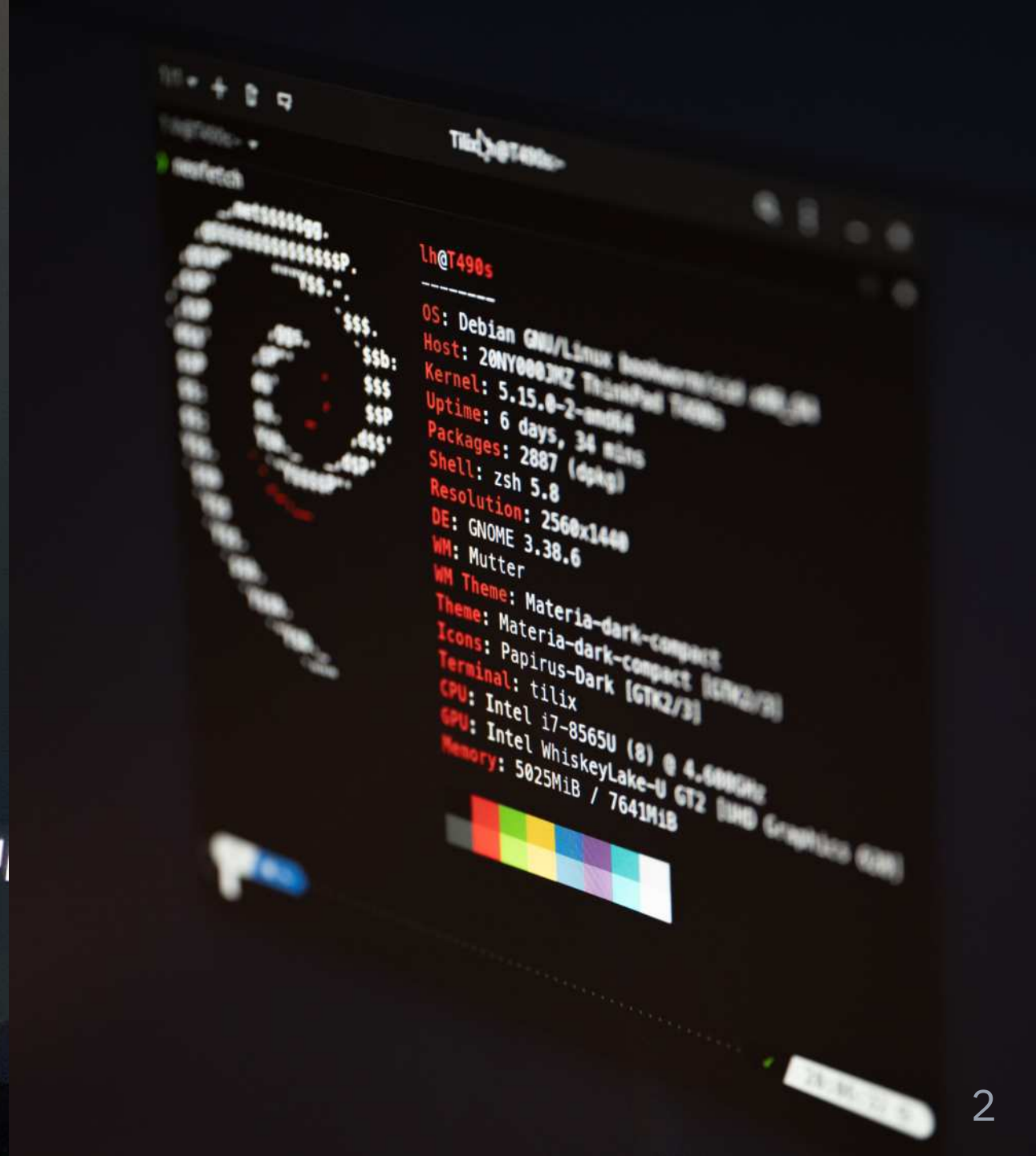
Dr. Ashish Sai

📅 Week 2 Lecture 2

💻 bcs1110.ashish.nl

📍 EPD150 MSM Conference Hall


Please wait while we install a system u


lh@T490s
OS: Debian GNU/Linux
Host: 20NY000JMZ ThinkPad T490s
Kernel: 5.15.0-2-amd64
Uptime: 6 days, 34 mins
Packages: 2887 (dpkg)
Shell: zsh 5.8
Resolution: 2560x1440
DE: GNOME 3.38.6
WM: Mutter
WM Theme: Materia-dark-compact
Theme: Materia-dark-compact [GTK2/3]
Icons: Papirus-Dark [GTK2/3]
Terminal: tilix
CPU: Intel i7-8565U (8) @ 4.600GHz
GPU: Intel WhiskeyLake-U GT2 [UHD Graphics 620]
Memory: 5025MiB / 7641MiB

Plan for today

- OS Fundamentals
- Storage and Interrupts
- OS Operations
- Process Management

Fundamentals of OS

Part 1/4

What is an Operating System?

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

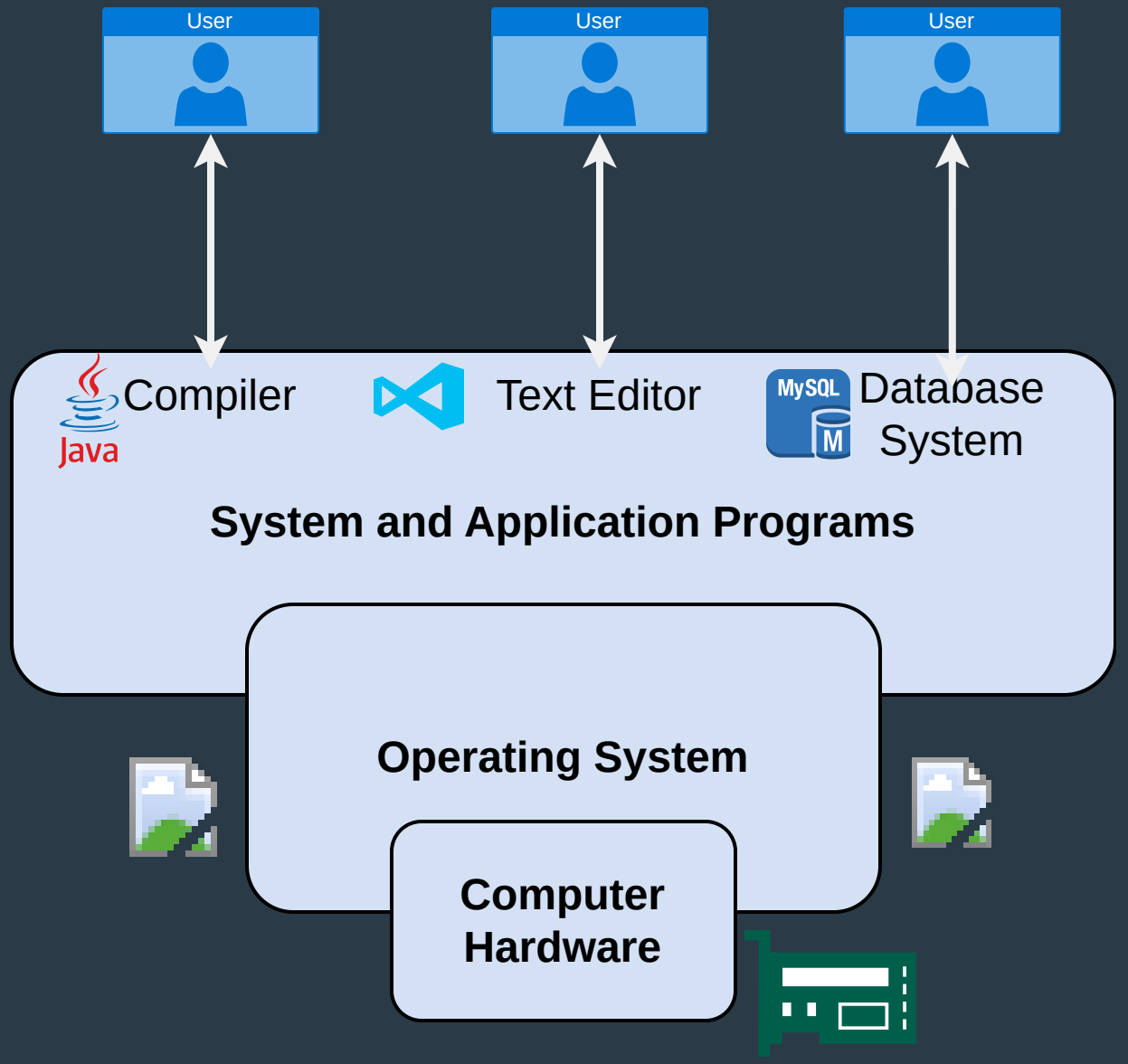
Operating System Goals

- 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 Structure

Computer system can be divided into four components:

Component	Description / Examples
1. Hardware	Provides basic computing resources - CPU, memory, I/O devices
2. Operating system	Controls and coordinates use of hardware among various applications and users
3. 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
4. Users	People, machines, other computers



Four Components of a Computer System

What Do Operating Systems Do?

> Depends on the point of view

- Users want convenience, ease of use and good performance (*Don't care about resource utilization*)
  - 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

More Formal Definition of Operating System

OS is a *resource allocator*

- Manages all resources
- Decides between conflicting requests for efficient and fair resource use

OS is a *control program*

- Controls execution of programs to prevent errors and improper use of the computer

More Formal Definition of Operating System

> There is no universally accepted definition

- “Everything a vendor ships when you order an operating system” is a good approximation but varies wildly
- “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

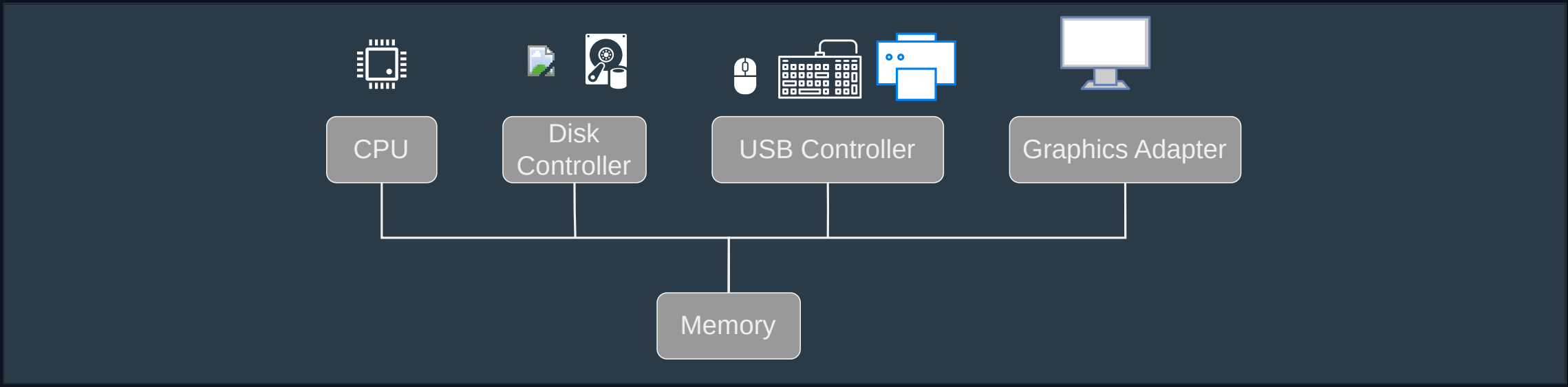
bootstrap program is loaded at power-up or reboot

- Typically stored in ROM or EPROM, generally known as **firmware**
- Initializes all aspects of system
- Loads operating system kernel and starts execution

Computer System Organization

Computer System Organization

- A computer system includes:
  - One or more **CPUs**
  - **Device controllers** (for keyboard, disk, screen, etc.)
- All are connected through a **common bus** to access **shared memory**



CPU vs Devices

- The **CPU** runs instructions from programs
- Devices (keyboard, disk, network card, …) also need to move data in/out
- Both the CPU and devices rely on **main memory** to do their work

Device Controllers

- Each device is managed by a **device controller**
  - Example: a disk controller manages the hard drive
- The controller knows the details of the device, so the CPU doesn’t have to
- Each controller has a small **local buffer** (temporary storage)

How Data Moves

- Data does **not** go straight from a device into main memory
- Instead:

1. Device writes data into its **local buffer**
2. CPU copies between **buffer** ↔️ **main memory**

- Example: typing on a keyboard → characters go to buffer → CPU moves them to memory

Concurrency

- Devices and the CPU can **both work at the same time**
- Example:
  - CPU is running a program
  - Meanwhile, disk controller is fetching data into its buffer
- This is what we mean by **concurrent execution**

The Question

- How does the CPU know **when a device has finished** its work?
  - The CPU can’t constantly stop to check each device (too slow)

> Solution: devices “signal” the CPU when they’re done

Interrupts

Part 2/4

Interrupts

- A device controller notifies the CPU by sending an **interrupt**
- An interrupt = “Tap on the shoulder”:
  - “I’m finished, please handle me now”

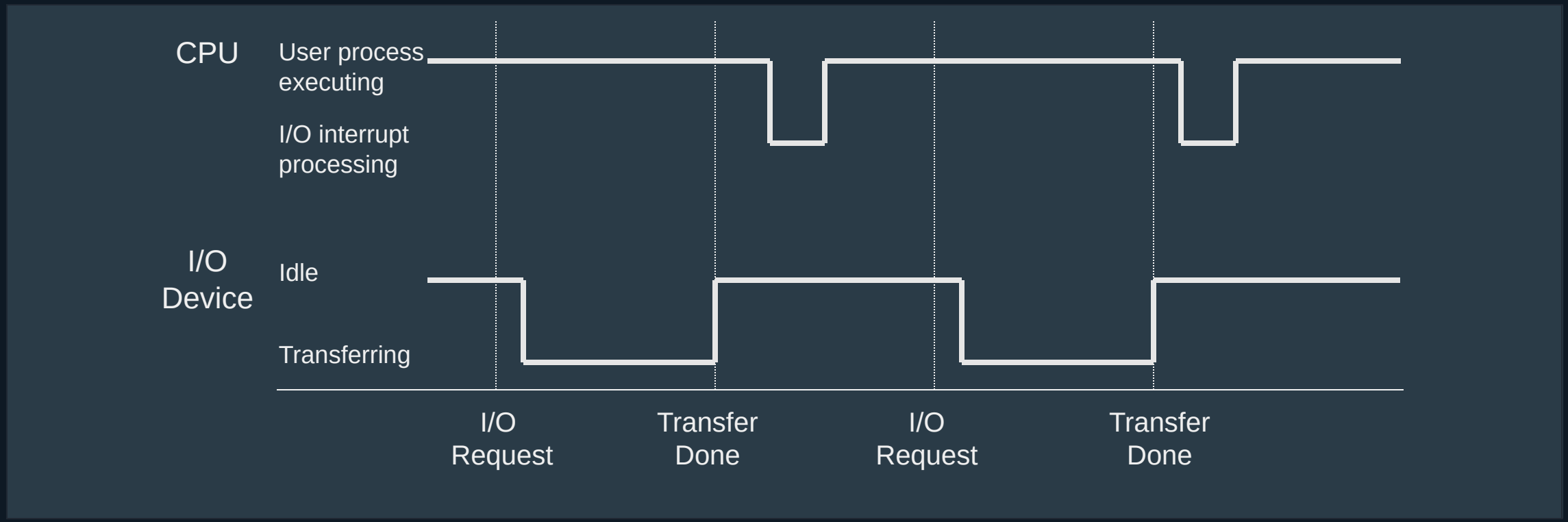
Interrupts

- A **trap** (or **exception**) = software-generated interrupt
  - Example: division by zero (error)
  - Or a user request (system call)
- Key idea: an OS is **interrupt-driven**
  - It reacts to events, instead of checking constantly

Interrupt Handling

When an interrupt occurs, the OS:

1. **Preserves CPU state** (registers, program counter)
2. **Identifies the interrupt type**
3. **Runs the correct handler**
    - Separate code segments handle each interrupt type


CPU
User process executing
I/O interrupt processing
I/O Device
Idle
Transferring
I/O Request
Transfer Done
I/O Request
Transfer Done

I/O Structure

- When a program requests I/O (e.g., read from disk), two main approaches exist:

1. **Synchronous I/O**
   - Control returns to the user program **only after I/O completes**
   - CPU may sit idle in a **wait instruction** until the next interrupt
   - Or spin in a **wait loop**, repeatedly checking device status
   - At most one I/O request can be outstanding → **no simultaneous I/O**

I/O Structure

2. **Asynchronous I/O**
    - Control returns to the user program **immediately**, without waiting
    - User can explicitly request completion check via a **system call**
    - The OS maintains a **device-status table**:
        - Each entry stores device type, address, and state
    - When an interrupt occurs, the OS looks up the device in the table, updates its state, and resumes the program

Storage

Part 2/4

Storage Definitions and Notation Review

- The basic unit of computer storage is the **bit**
- A **byte** = 8 bits

Unit	Size in Bytes
Kilobyte (KB)	$1,024$
Megabyte (MB)	$1,024^2$
Gigabyte (GB)	$1,024^3$
Terabyte (TB)	$1,024^4$
Petabyte (PB)	$1,024^5$

Storage Structure

- **Main memory** – only large storage media that the CPU can access directly
  - **Random access**
  - **Typically volatile**
- **Secondary storage** – extension of main memory that provides large nonvolatile storage capacity
  - *Hard disks* – rigid metal or glass platters covered with magnetic recording material
  - *Solid-state disks* – faster than hard disks, nonvolatile

Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility

Caching

Copying information into faster storage system; main memory can be viewed as a cache for secondary storage

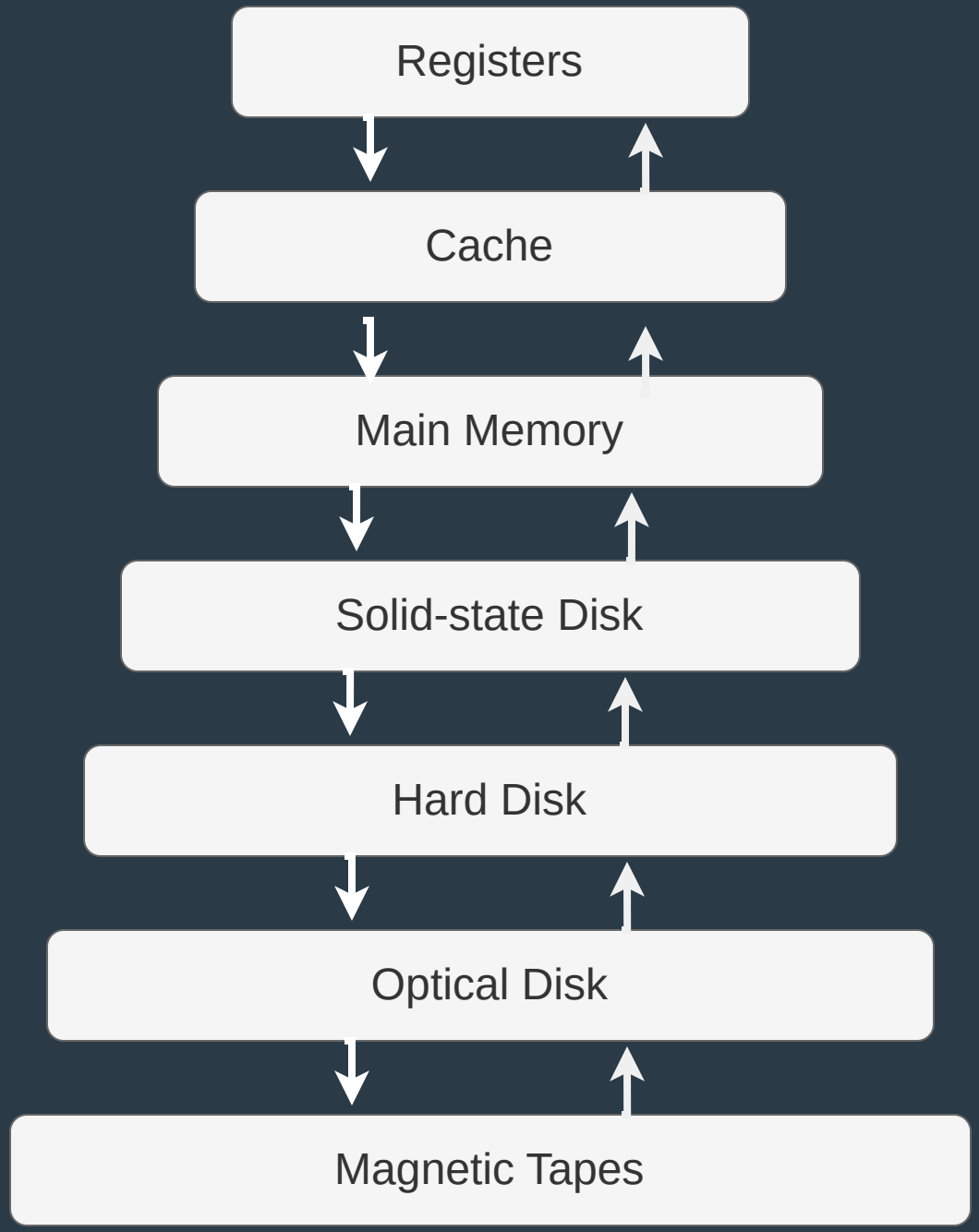
- A **cache** is a small, fast storage
- It keeps a copy of information from slower storage
- Goal: speed up access to frequently used data

Caching (how it works)

1. **Check cache first**
   - If data is there → use it (**fast**)
2. **If not in cache**
   - Copy it from slower storage into cache
   - Then use from cache

Caching (design issues)

- Cache is always **smaller** than the storage it speeds up
- Raises important design problems:
  - **Cache size** (how big should it be?)
  - **Replacement policy** (which data to remove when full)


Registers
Cache
Main Memory
Solid-state Disk
Hard Disk
Optical Disk
Magnetic Tapes

OS Operations

Part 3/4

The Kernel

- The **kernel** = the **core of the OS**
- Always running while the system is on
- Directly manages:
  - CPU scheduling
  - Memory allocation
  - I/O and devices
  - Storage

➡️ Everything else (apps, GUIs, utilities) runs **on top of the kernel**

OS Responsibilities

1. **Process Management**
   - Create, schedule, terminate processes
2. **Memory Management**
   - Track usage, allocate, swap, use virtual memory
3. **I/O and Device Management**
   - Control device communication
4. **Storage Management**
   - File systems, access control, backup
5. **Protection & Security**
   - Safe sharing of resources

OS Operations

- OS is **interrupt-driven**
- **Hardware interrupt**: from a device
- **Software interrupt** (trap/exception):
  - Error (e.g., divide by zero)
  - Request for OS service (**system call**)
  - Process misbehavior (infinite loop, modifying OS)

OS Operations — Dual Mode

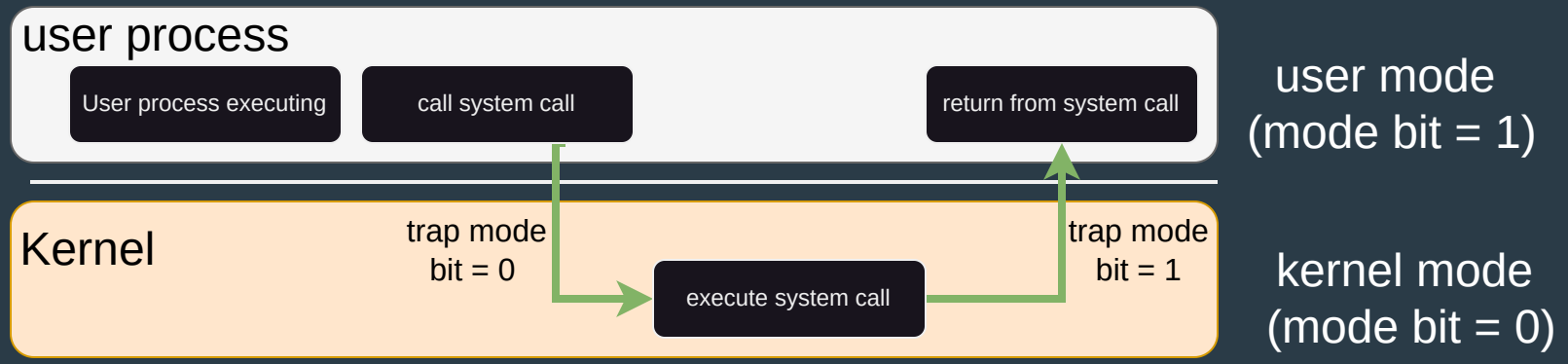
- **Dual-mode operation** protects system
  - **User mode** → normal programs
  - **Kernel mode** → OS execution
- **Privileged instructions** → only in kernel mode
  - **System call** switches to kernel
  - **Return** switches back to user

Transition from User to Kernel Mode

- OS uses a **timer** to stay in control
- Timer:
  - Set by OS (privileged)
  - Decrements with physical clock
  - When counter = 0 → **interrupt**

Why?

- Prevent infinite loops
- Stop processes from hogging CPU
- Regain control



Multiprogramming

- Needed for **efficiency**
- One user cannot keep CPU + I/O busy
- Multiprogramming ensures CPU always has work:
  - Jobs (code + data) organized in memory
  - **Job scheduling** chooses a job to run
  - If one job waits (I/O), CPU runs another

Process Management

Part 4/4

Processes

- A **process** = a **program in execution**
  - **Program** = passive (stored on disk)
  - **Process** = active (loaded in memory, running)

Needs resources:

- CPU, memory, I/O, files, input data
- At termination → OS reclaims resources

Processes in the System

- Many processes run concurrently:
  - Some belong to **users**
  - Some belong to the **operating system**
- Concurrency achieved by **multiplexing CPUs** among processes/threads

OS and Process Management

The OS is responsible for:

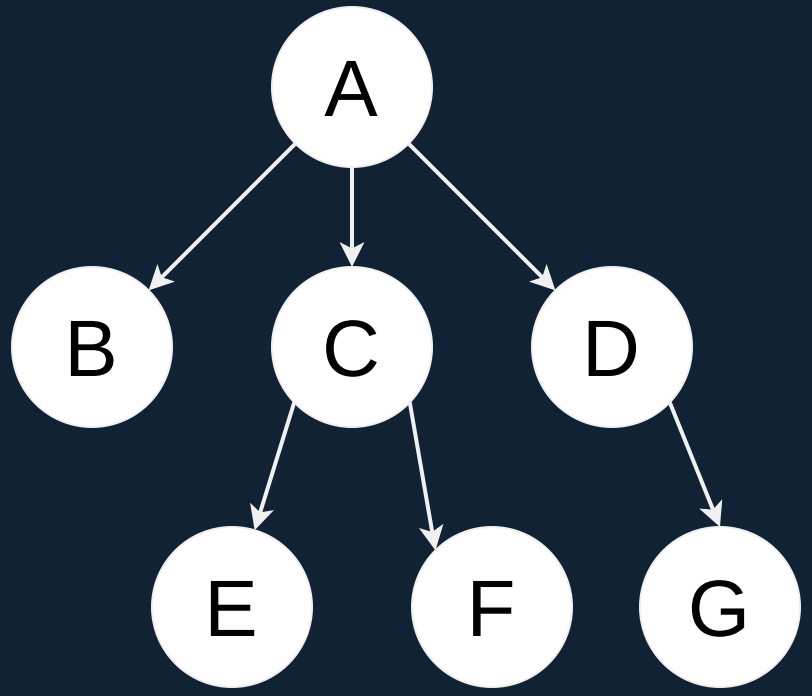
- Creating & deleting user/system processes
- Suspending & resuming processes
- Providing mechanisms for:
  - Synchronization
  - Communication
  - Deadlock handling

Process Tree

- OS keeps track of all processes in a **process table**
- Processes can **create other processes**

➡️ Relationships form a **process tree**:

- A → children B, C, D
- C → children E, F
- D → child G



Process Concept — Program vs Process

- **Program** = passive (on disk)
- **Process** = active (in memory)

➡️ A program becomes a process when loaded into memory

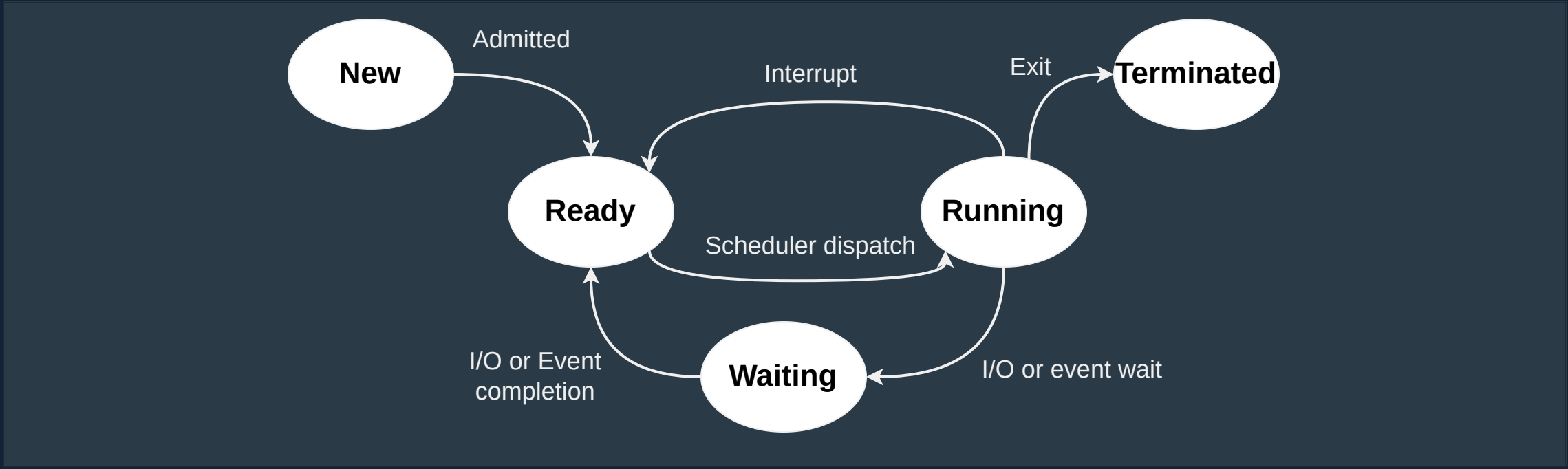
- Programs can be started by GUI clicks, command line, etc.
- One program may become **several processes** (e.g., multiple users running it)

Process States

As a process runs, it changes state:

- **New** → process is being created
- **Running** → instructions executing
- **Waiting** → waiting for event (e.g., I/O)
- **Ready** → waiting for CPU
- **Terminated** → finished execution

Diagram of Process State



CPU Scheduling

- **Scheduler** picks which ready process gets the CPU
- May occur when a process:
  1. Running → Waiting
  2. Running → Ready
  3. Waiting → Ready
  4. Terminates
- In cases (1) & (4): no choice (must switch)
- In (2) & (3): scheduler has a choice

Preemptive vs Non-Preemptive Scheduling

- **Non-preemptive**:
  - CPU given → process keeps it until it finishes or waits
- **Preemptive**:
  - CPU can be taken away at any time (switch on (2) or (3))

➡️ All modern OS (Windows, Linux, macOS, UNIX) use **preemptive scheduling**

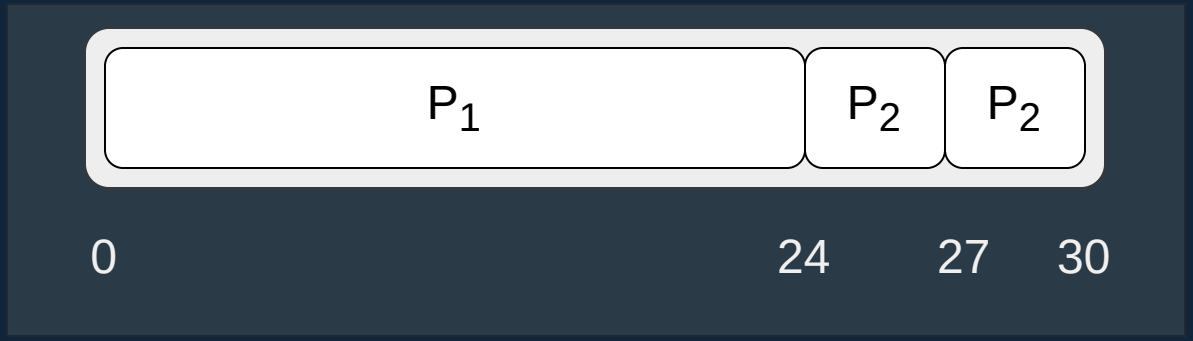
Scheduling Criteria

- **CPU utilization** → keep CPU busy
- **Throughput** → # processes completed per unit time
- **Turnaround time** → total time for a process to finish
- **Waiting time** → time spent in ready queue
- **Response time** → time from request → first response

First-Come, First-Served (FCFS)

Process	Time
P1	24
P2	3
P3	3

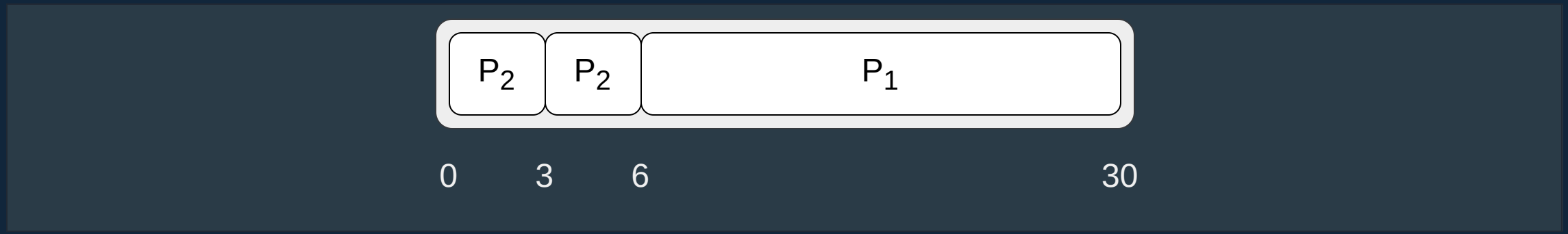
Schedule:



- Waiting times: P1=0, P2=24, P3=27
- Average = (0+24+27)/3 = 17

FCFS (different order)

Processes arrive: P2, P3, P1



- Waiting times: P1=6, P2=0, P3=3
- Average = (6+0+3)/3 = 3

➡️ **Much better than previous case**

Shortest Job First (SJF)

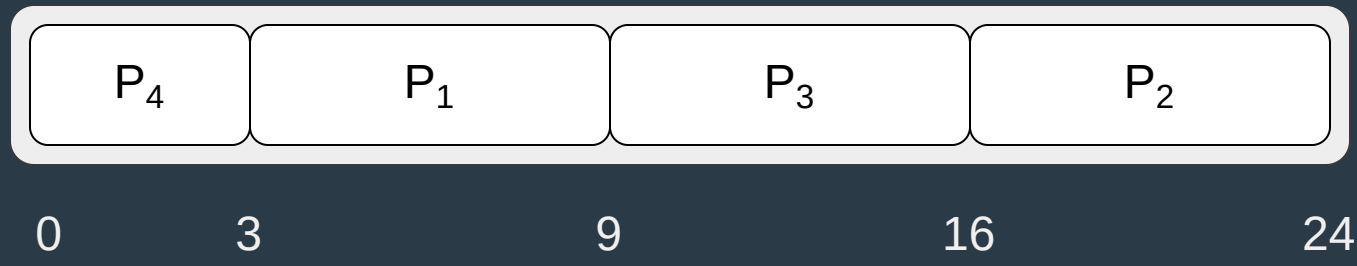
- Schedule process with **shortest CPU burst** first
- **Optimal**: minimum average waiting time
- Preemptive version = **shortest remaining time first**

But… how do we know the burst length?

- Ask user
- Estimate

Example of SJF

Process	Time
P1	6
P2	8
P3	7
P4	3



Average waiting time = (3+16+9+0)/4 = 7

See you in the lab! 👋