Processes

CS 3410: Computer System Organization and Programming

Spring 2025

Logistics

- A9 due Wed. 3/26 (Late: Sun. 3/29)
- **Spring Break!!**
  - No lab this week!
  - No assignment this week!
  - Have fun, be safe!

- Prelim 2 on Thu. 4/10 after break
  - Practice exam out later today (see website)

Logistics

- A5-A7 Grades Out!
  - A6 has been fixed
  - Submit regrade requests before you leave for break

Today's Goals

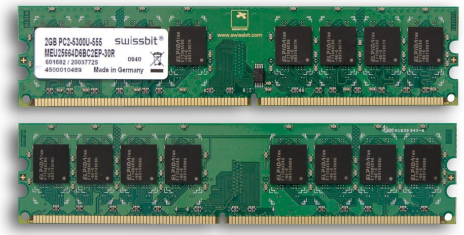
- From Hardware to Systems
- Program vs. Process
- Kernel vs. User Space
- System Calls from a C Perspective
- `fork(), exec(), waitpid()`
- Signals

From Hardware View to System View

Programs

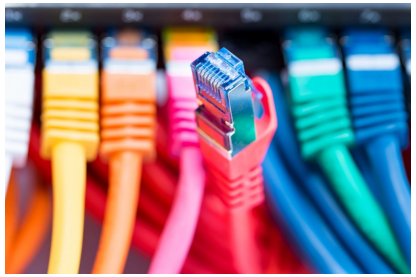
Hardware

Processor



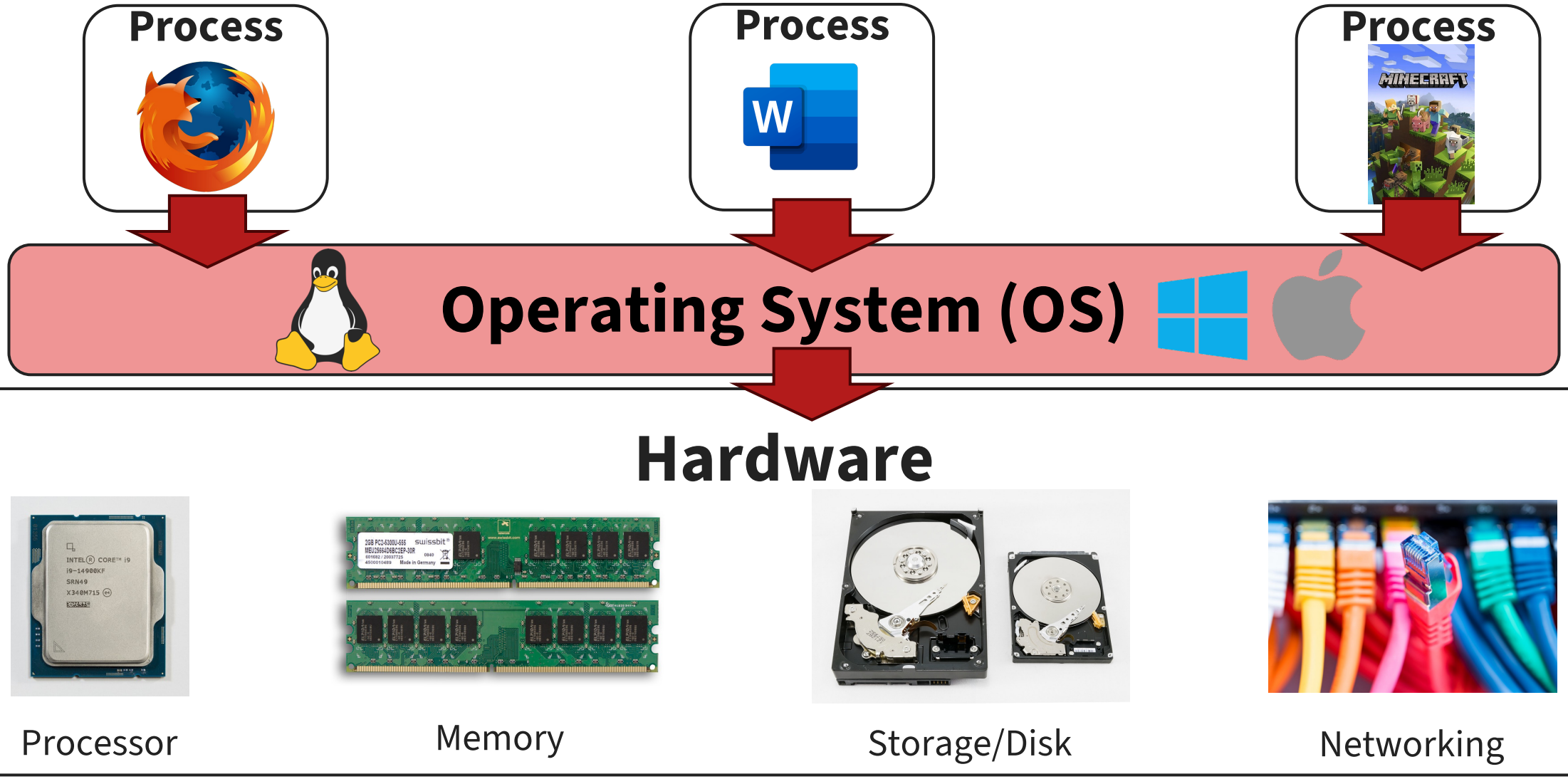
Memory

Storage/Disk



Networking

From Hardware View to System View



Operating System

The Operating System (OS) acts as an **illusionist**

- Any program we run **doesn't need to know** that the OS or other programs exist
- Any program we run **doesn't need to worry** about how syscalls actually work

Operating System

The Operating System (OS) acts as an **illusionist**

- Any program we run **doesn't need to know** that the OS or other programs exist
- Any program we run **doesn't need to worry** about how syscalls actually work

The Operating System (OS) acts as a **conductor**:

- Receive commands from the user and assigns computer resources to tasks

Program vs. Process

Program

- A program consists of **code** and **data**
  - Written in some programming language (e.g., C)
  - Typically stored as a file on disk

Process

- A process is a *currently running instance* of a program
  - **Can run a program multiple times!**

Program vs. Process

Program

- A program is **inert**
  - Just **code** and **data**
  - Doesn't *do* anything

Process

- A process is **alive**
  - Mutable data
  - Registers
  - Files
  - Packets
- Can be run simultaneously:

```
$ ./program &
$ ./program &
```

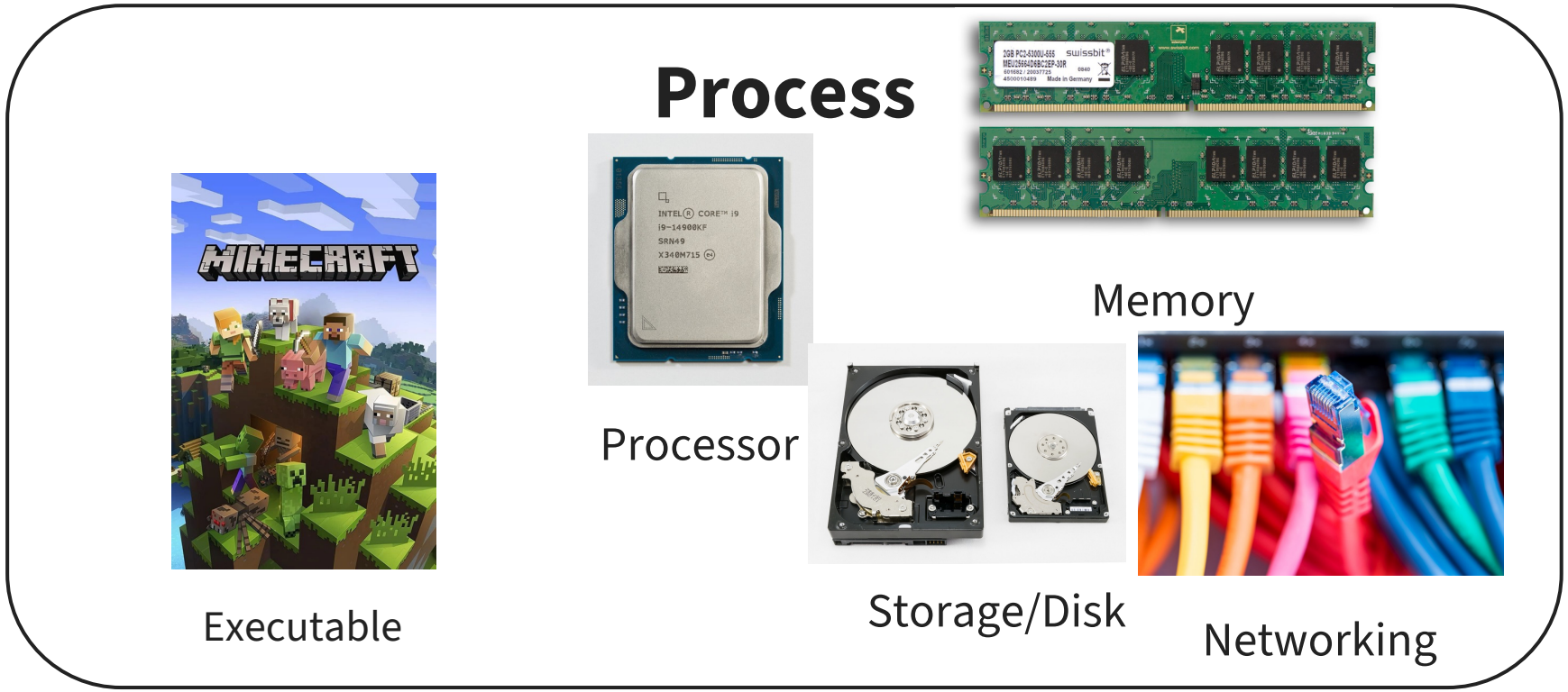
From Program to Executable

- An executable is a **file** containing:
  - The machine code (i.e., CPU instructions)
  - Data (i.e., information manipulated by the instructions)

- Obtained by compiling a program and linking with libraries

Process

An instance of an executable on an **abstraction** of a computer



Process

An instance of an executable on an **abstraction** of a computer

Process

Executable

Machine State:

Process

An instance of an executable on an **abstraction** of a computer

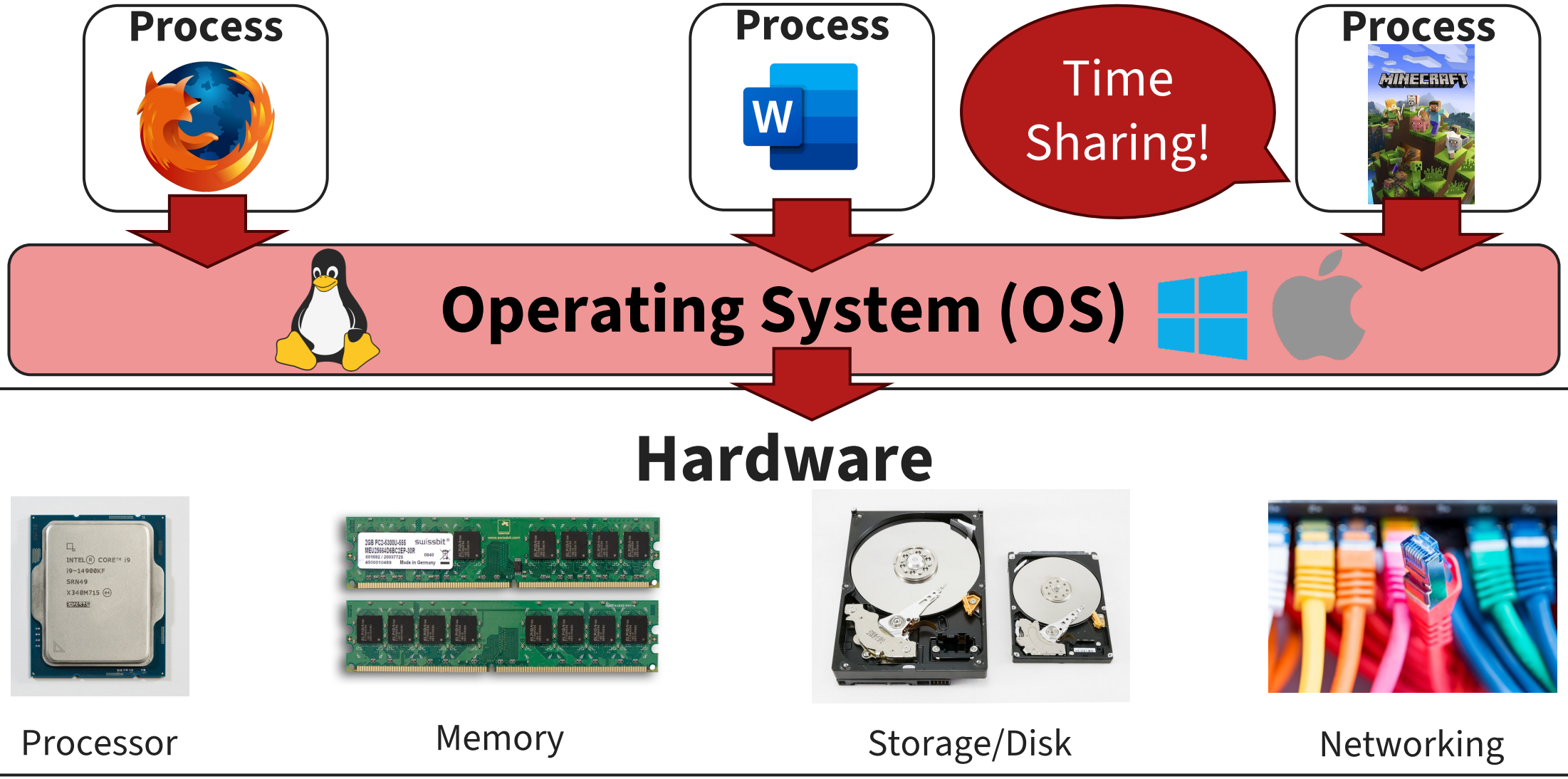
Process

Executable

Machine State:

1. Address space (memory)
2. Execution context (registers, PC, `sp`, `fp`)
3. Environment (clock, files, network)
   - Accessed via syscalls

From Hardware View to System View



Operating System

The Operating System (OS) acts as an **illusionist**

- Any program we run **doesn't need to know** that the OS or other programs exist
- Any program we run **doesn't need to worry** about how syscalls actually work

The Operating System (OS) acts as a **conductor**:

- Receive commands from the user and assigns computer resources to tasks

The Operating System (OS) acts as a **referee**:

- Keep track of what processes are running, and assign appropriate permissions

A Day in the Life of a Process

A Day in the Life of a Program

sum.c

```
#include <stdio.h>

int max = 10;

int main() {
  int sum = 0;
  add(max, &sum);
  printf("%d", sum);
   ...
}
```

Source file

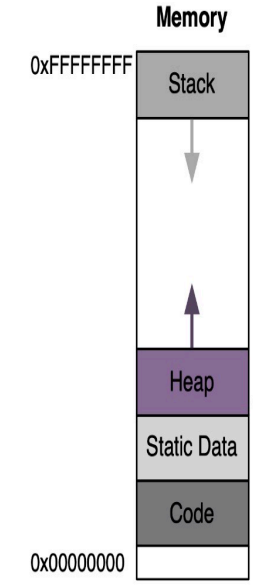
sum

```
0040 0000    …
.text        0C40023C
  main       21035000
             1B80050C
             8C048004
             21047002
             0C400020
1000 0000    …
.data        10201000 # max
             21040330
             22500102
```

Executable

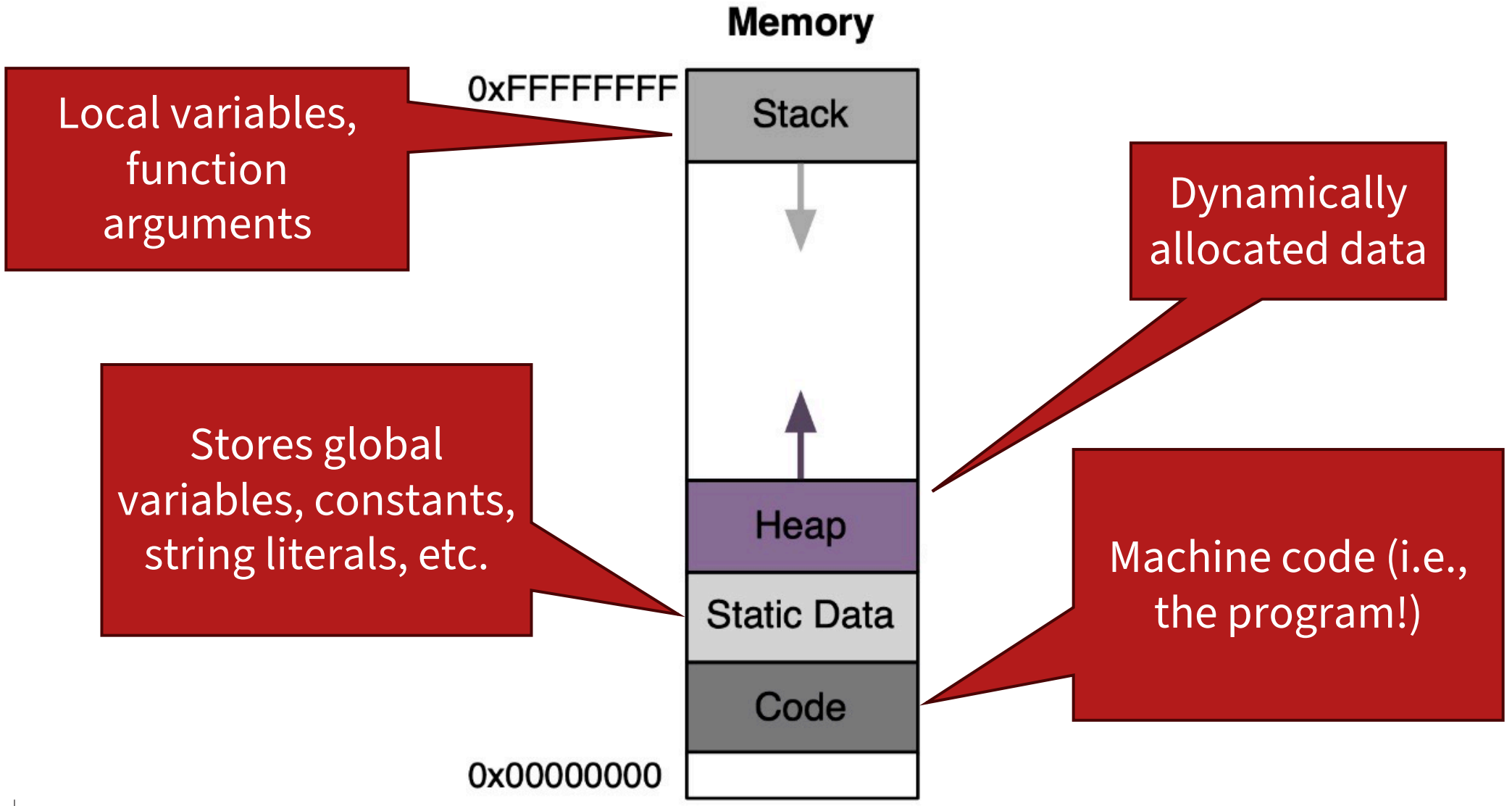
pid 042

PC
sp
fp
…



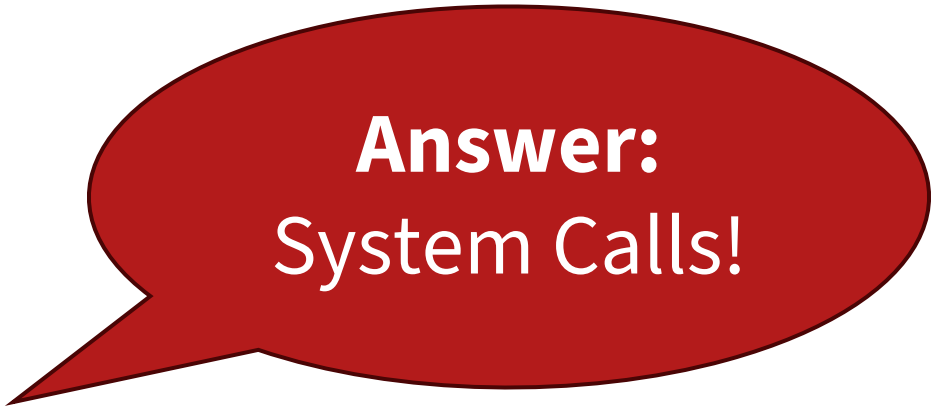
Process

Logical View of Process Memory



Environment

- Processor, registers, and memory allow you to implement algorithms
- But, how do you:
  - Read input/write to screen?
  - Create/read/write/delete files?
  - Create new processes?
  - Receive/send network packets?
  - Get the time/set alarm?
  - Terminate the current process?



A Process Physically Runs on the CPU

Each process has its own:

- Registers
- Memory
- I/O resources

Usually many, many more processes than CPU cores

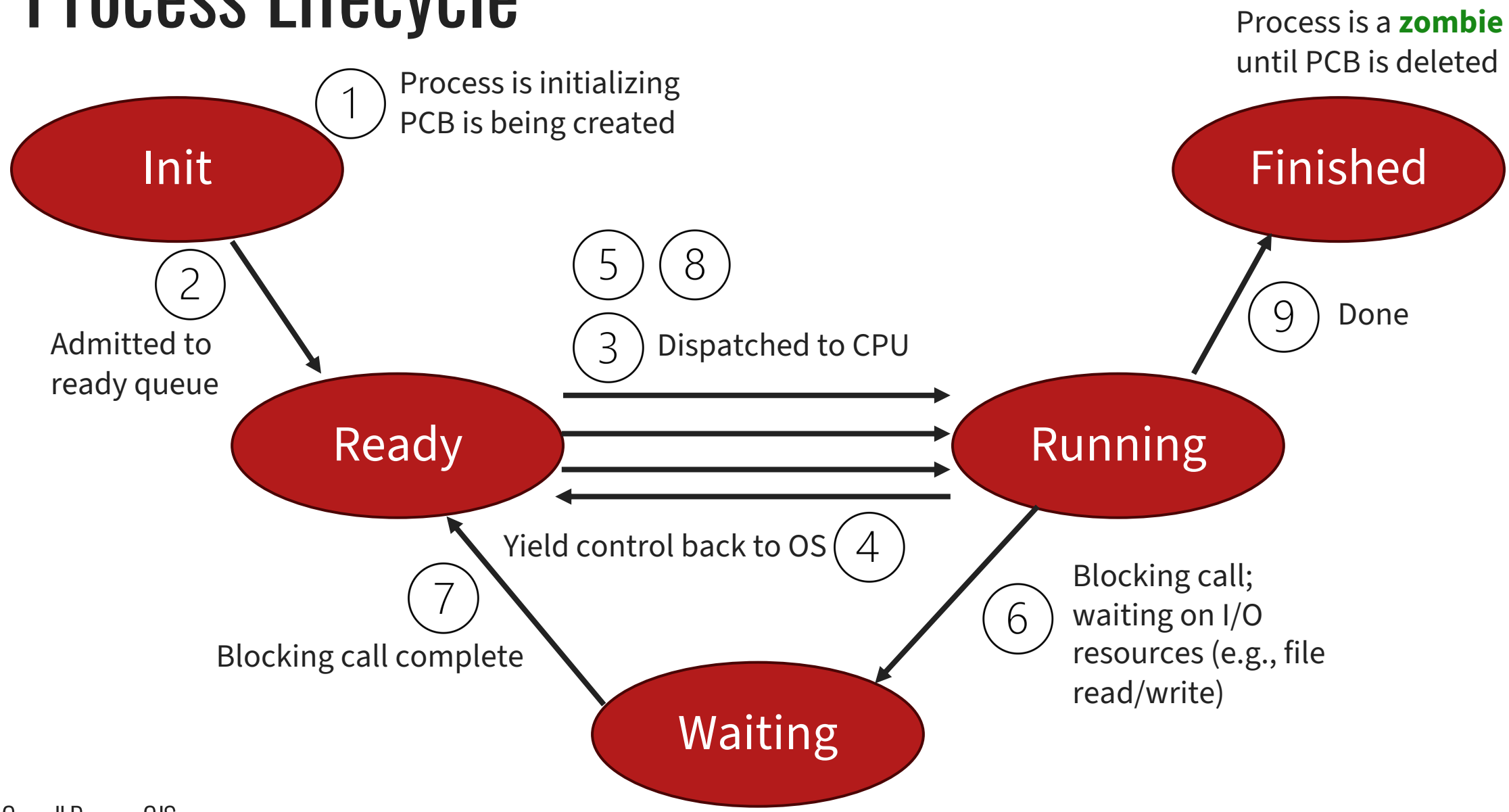
- Each process gets its own **virtual CPU**
- OS needs to **multiplex**, **schedule**, and **create** virtual CPUs for each process

Process Control Block (PCB)

OS maintains a PCB for each process, containing:

- Process ID **`pid`**
- Process State (e.g., running, waiting, ready)
- Process User **`uid`**
- Memory Management Information
- Scheduling Information
- Parent Process ID **`ppid`**
- … and more!

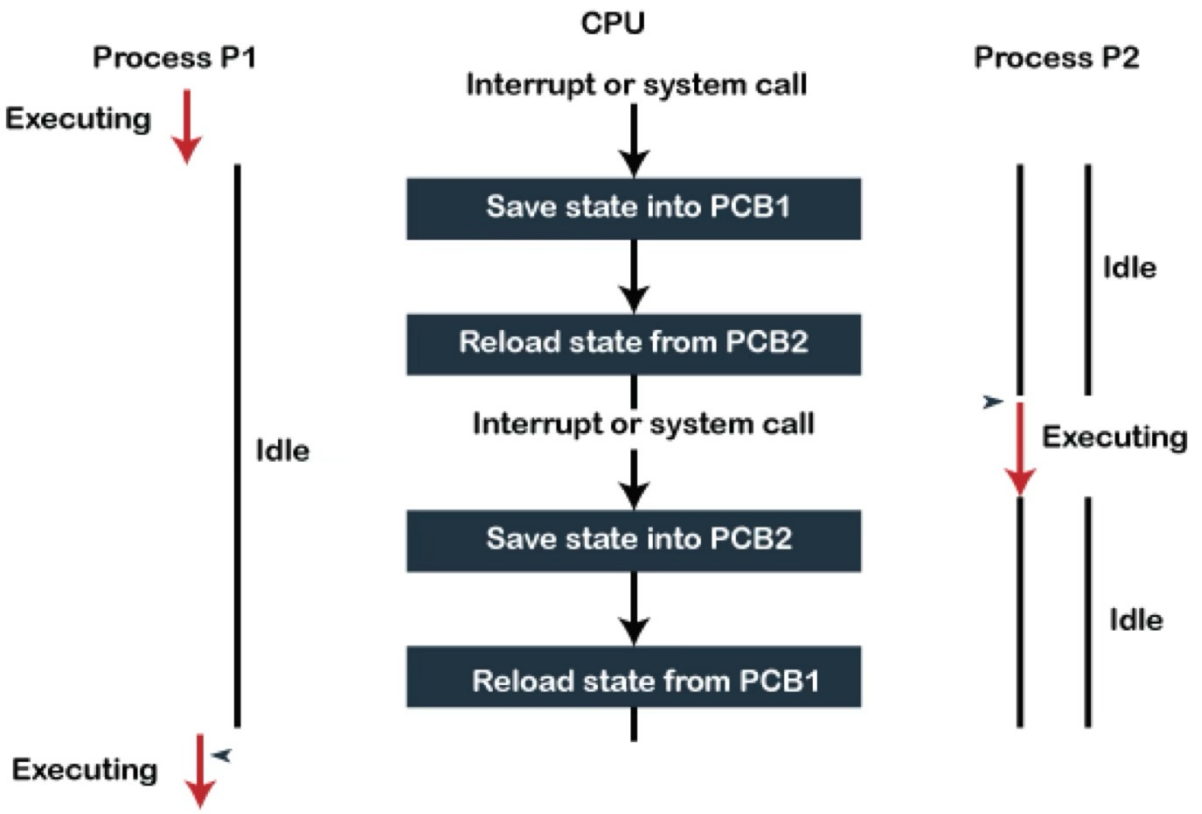
Process Lifecycle



Context Switching

The process by which an OS saves the state of a currently running process and restores the state of another process

1. Save current process state
2. Update Process Control Block (PCB)
3. Select next process
4. Restore the next process state
5. Resume execution



Performance Cost of Context Switching

- Saving and restoring process states takes time (i.e., **overhead**)
- Goal is to **minimize** overhead to maintain system performance
- Context switching needs to be efficient for the smooth operation of multitasking systems

PollEverywhere

User Space vs. Kernel Space

User Space vs. Kernel Space

- **Kernel** is a program at the core of the operating system
  - First “process”
  - Responsible for **managing the system’s resources**
- **Kernel space** is where **kernel** operates
  - Example tasks: Maintain PCBs, schedule next processes, privileged operations
  - Unrestricted access to all computer’s resources

User Space vs. Kernel Space

- **User space** is where user-facing applications run
  - Examples: web browser, text editor, music player
  - **Restricted** and **isolated** from kernel space to ensure system **stability** and **security**
  - User space applications **cannot** directly access system's hardware; must ask kernel