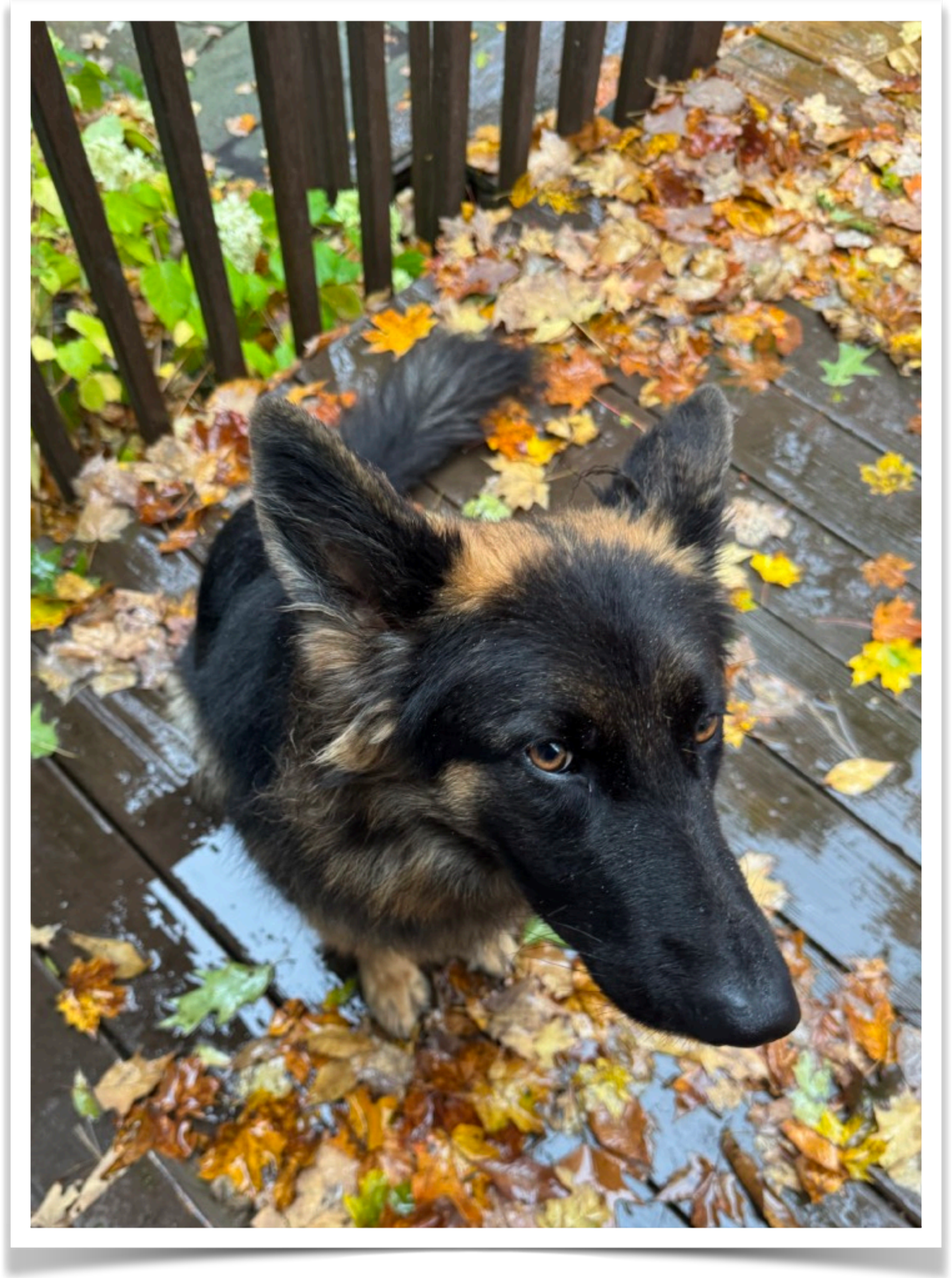


CS3410: Computer Systems and Organization

LEC18: Processes

Professor Giulia Guidi
Wednesday, October 29, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

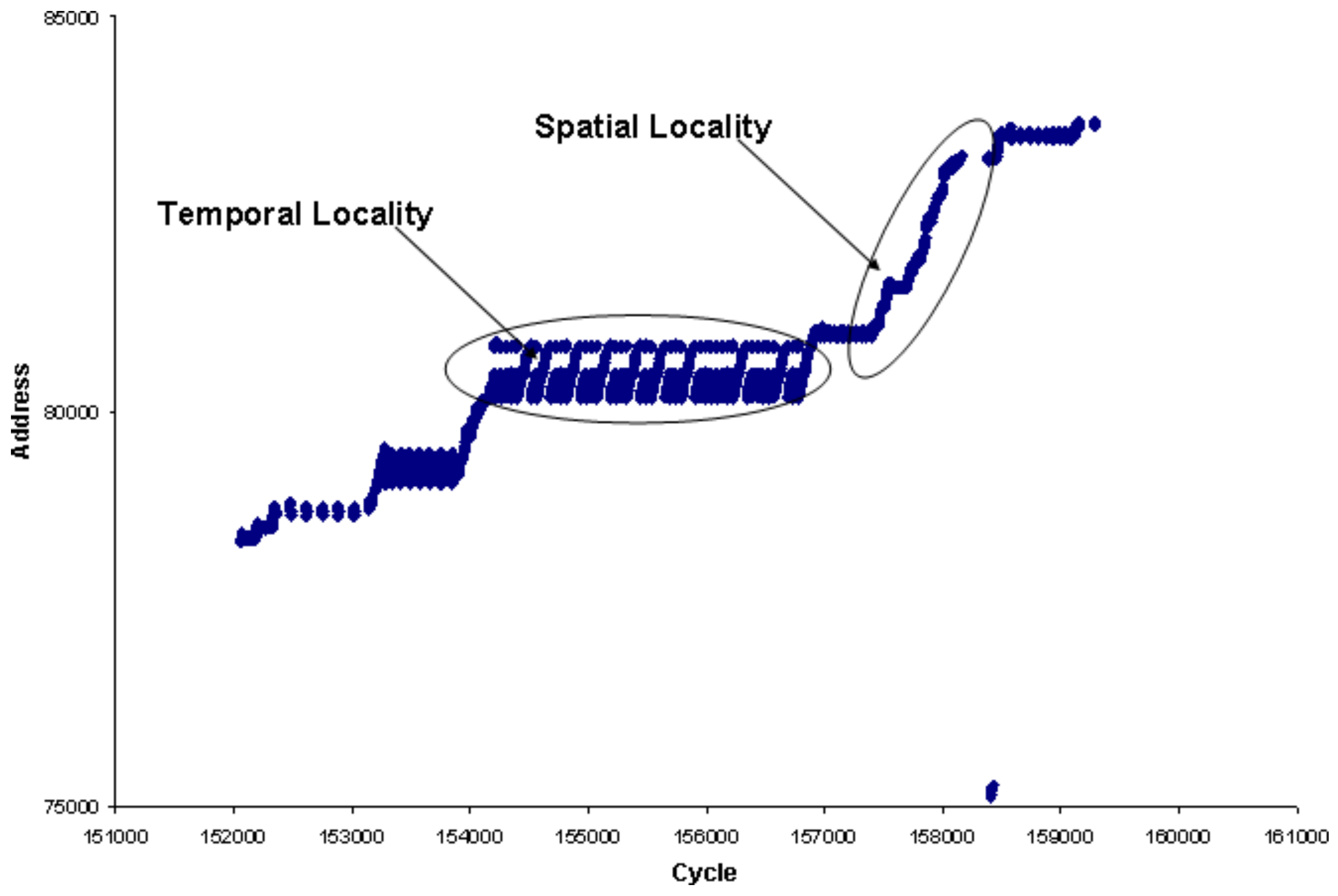
Plan for Today

- Review of caches
- A **new** topic: **processes**

Review of caches

Locality in a Nutshell

Locality is *not just about how often a variable appears*, but **about how the value is reused** over time or space **relative to the rest of the program**



https://stackoverflow.com/questions/7638932/what-is-locality-of-reference

Core Ideas and Challenges

One line per address. One chance.

A **direct mapped cache** is like an assigned seat on the plane:

- If there are empty seats, you **must** still sit in your assigned one

Good things:

- It's energy efficient
- The hardware is simple
- The lookup is super fast

Core Ideas and Challenges

One line per address. One chance.

A **direct mapped cache** is like an assigned seat on the plane:

- If there are empty seats, you **must** still sit in your assigned one

Bad things:

- Conflict misses: if two **hot** addresses map to the **exact same** line → **thrash city!**
  - Cache **thrashing** is a thing:
    - You access A → evict B → then, access B → evict A → repeat until sanity is lost
- It can lead to trashing even with **good locality**

Core Ideas and Challenges

Great flexibility, chaotic vibes.

A **fully associative cache** is like open seating on Southwest Airlines:

- Sit wherever you want

Good thing:

- It leads to significantly fewer cache misses 🎉

Core Ideas and Challenges

Great flexibility, chaotic vibes.

A **fully associative cache** is like open seating on Southwest Airlines:

- Sit wherever you want

Bad things:

- The hardware is complex and doesn't scale to large caches
- The lookup is slower
- The replacement can get complicated

Core Ideas and Challenges

Compromise between direct-mapped and fully associative caches

A **set associative cache** is like having reserved tables at a restaurant:

- You can sit at any chair at your table, but you still can't sit anywhere you want

Good things:

- Reduces conflict misses compared to direct-mapped
- It's less complex and faster than fully associative
- It's flexible enough to handle some collisions without being super expensive

Core Ideas and Challenges

Compromise between direct-mapped and fully associative caches

A **set associative cache** is like having reserved tables at a restaurant:

- You can sit at any chair at your table, but you still can't sit anywhere you want

Bad things:

- It's slightly slower than direct-mapped (must search all ways in a set)
- You need more hardware for comparators than direct-mapped
- Complexity grows as number of ways increases

Cache performance

The average access time $\mathbf{t_{avg}}$:

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

$t_{avg} = 4 + 5\% * 100$

$t_{avg} = 9$ **cycles**

The average access time $\mathbf{t_{avg}}$:

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

$t_{avg} = 1\ ns + 5\% * 50\ ns$

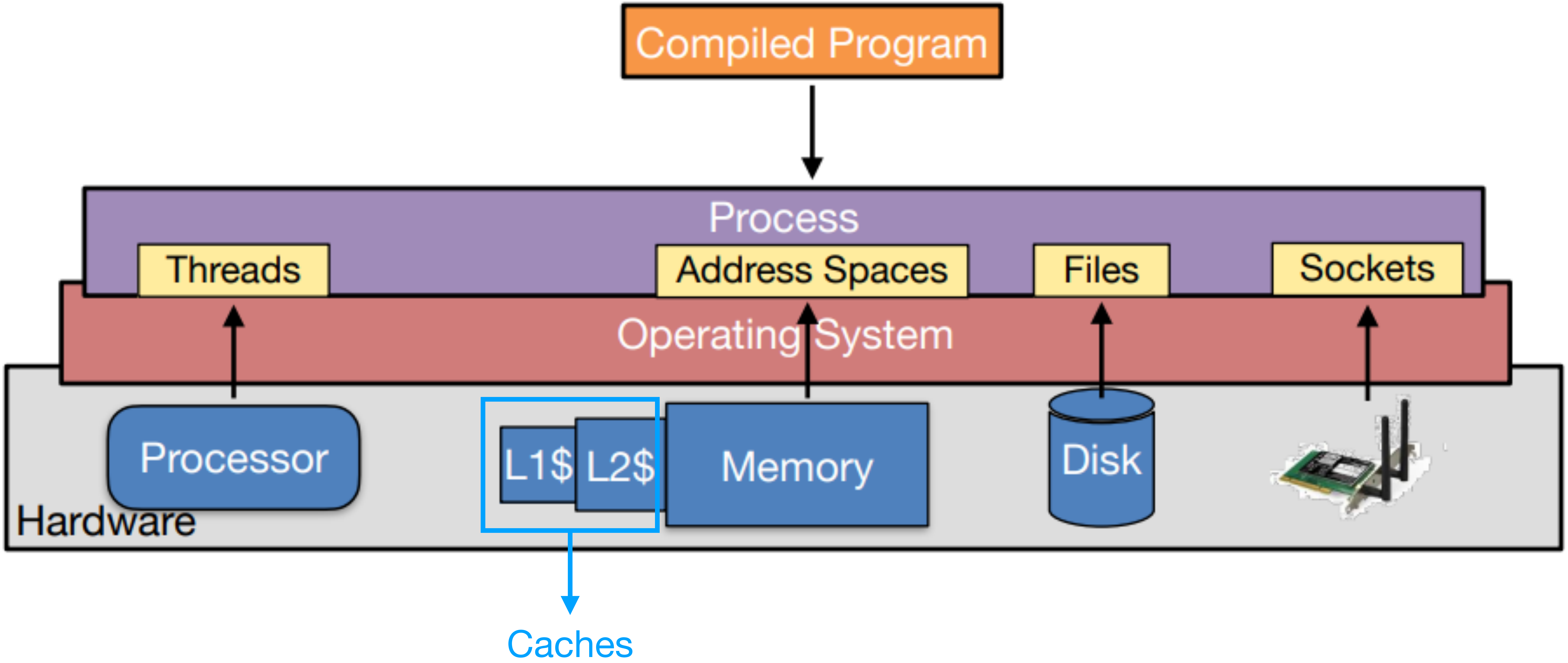
$t_{avg} = 3.5\ ns$

Three types of cache misses (3 Cs):

- **Cold** or **Compulsory:** first access ever to a block
- **Capacity:** the cache is too small
- **Conflict:** mapping collision (esp. direct mapped), the associativity is too low

The operating system (OS)

From Hardware View to System View



If we can run instructions directly on the CPU, why do we need an operating system?

(a) how do multiple programs share CPU and memory without stepping on each other?
(b) how does the OS decide which process gets cache, memory, or I/O?

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

A **system call** is a way for a program to ask the OS to do something on its behalf, like reading a file, printing to the screen, or creating a new program

erating System

:s as an **illusionist**:

t need to know that the OS *or other programs exist*

t need to worry about how **syscalls** actually work

A **system call** is a way for a program to ask the OS to do something on its behalf, like reading a file, printing to the screen, or creating a new program

A **system call** is like pressing a button on a vending machine:

- You (the program) want a snack (like reading a file or printing something)
- You can't reach inside to grab it yourself (you are in **user space**, the snack is in **kernel space**)
- So you press a button (make a **syscall**), and the machine (the OS) delivers the snack to you

Program's Perspective

From the **program's perspective**, the following statements are true:

- "I am the only program running on the CPU"
- "There's only one CPU, one memory, etc. on this system"
- "I have a full memory to use however I want"
- "ecalls (e.g., `printf`, `malloc`, `scanf`) just work"

this is ***not true*** anymore

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

Conceptual RISC-V Print “Hello”

Conventionally, `$v0` holds the **system call number**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li   v0, 4            # Load system call code for 'print string'
la   a0, msg          # Load address of message
syscall               # Call to the OS
...
msg: .asciiz "Hello!"
```

- The program is saying: “I want to use system call #4, i.e., print string”
- **Opcode** for the OS

recipe (on paper in the cookbook)

process versus program

person actively cooking from that recipe (ingredients, tools, stove all in use)

Process versus Program

recipe

- A **program** consists of code and data
  - It is specified in some programming language, e.g., C
  - It is typically stored in a file on disk

Process versus Program

recipe

- A **program** consists of code and data
  - It is specified in some programming language, e.g., C
  - It is typically stored in a file on disk
- "Running a program" means creating a **process**
  - **Can run a program multiple times!**
    - One after the other, or even concurrently

person actively cooking from that recipe (ingredients, tools, stove all in use)

From Program to “Executable”

- An executable is a file containing:
    - The executable code, i.e. CPU instructions
    - Data, i.e. information manipulated by these instructions

- Obtained by compiling a program and linking with libraries

What a Process Really Is

- Program = recipe (**passive**)
- Process = chef actively cooking (**active**, doing things, using tools)

- An executable running on an **abstraction** of a computer:

The address space (memory) + execution or CPU context (e.g., register, program counter, stack pointer)

(a) Controlled by **machine code** (instructions)

The enviroment (e.g., files, devices)

(b) Controlled by **syscalls**

What a Process Really Is

- An executable running on an **abstraction** of a computer

  (a) The address space (memory) + execution context (e.g., register)

  (b) The enviroment (files, etc.)

- A **good abstraction** (processes abstract away the CPU and registers):
  - Is portable and hides implementation details
  - Has an intuitive and easy-to-use interface
  - Can be instatiated many times
  - Is efficient to implement

Process ≠ Program

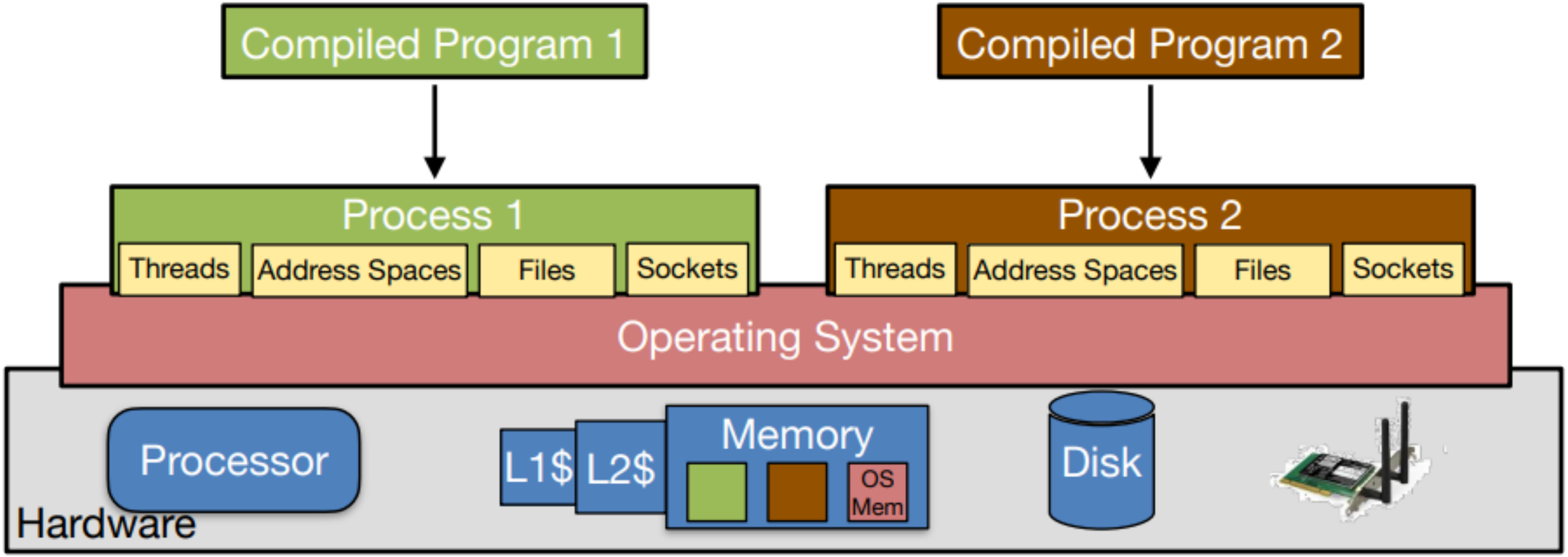
- Program = recipe (**passive**) = code + data
- Process = chef actively cooking (**active**, doing things, using tools) = mutable data, files

- The same program can be run multiple times simultaneously, e.g., 1 program, 2 processes

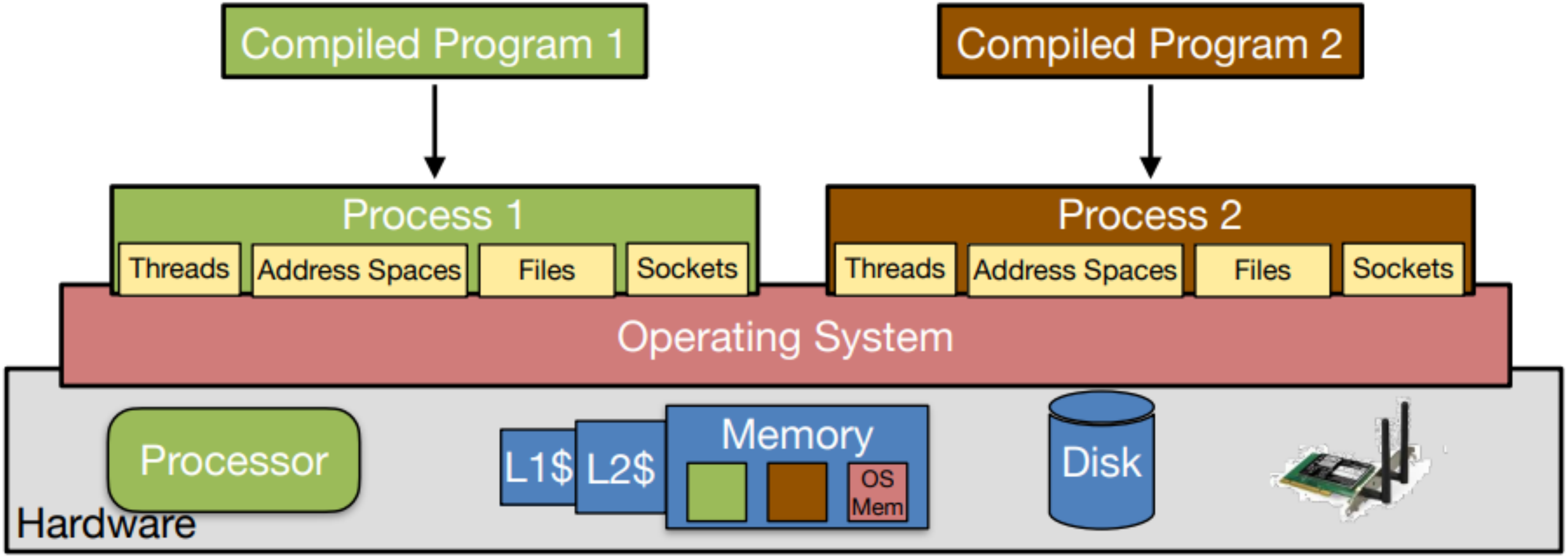
```
> ./program &
> ./program &
```

many processes can originate from the same program, just as many people can independently cook the same recipe

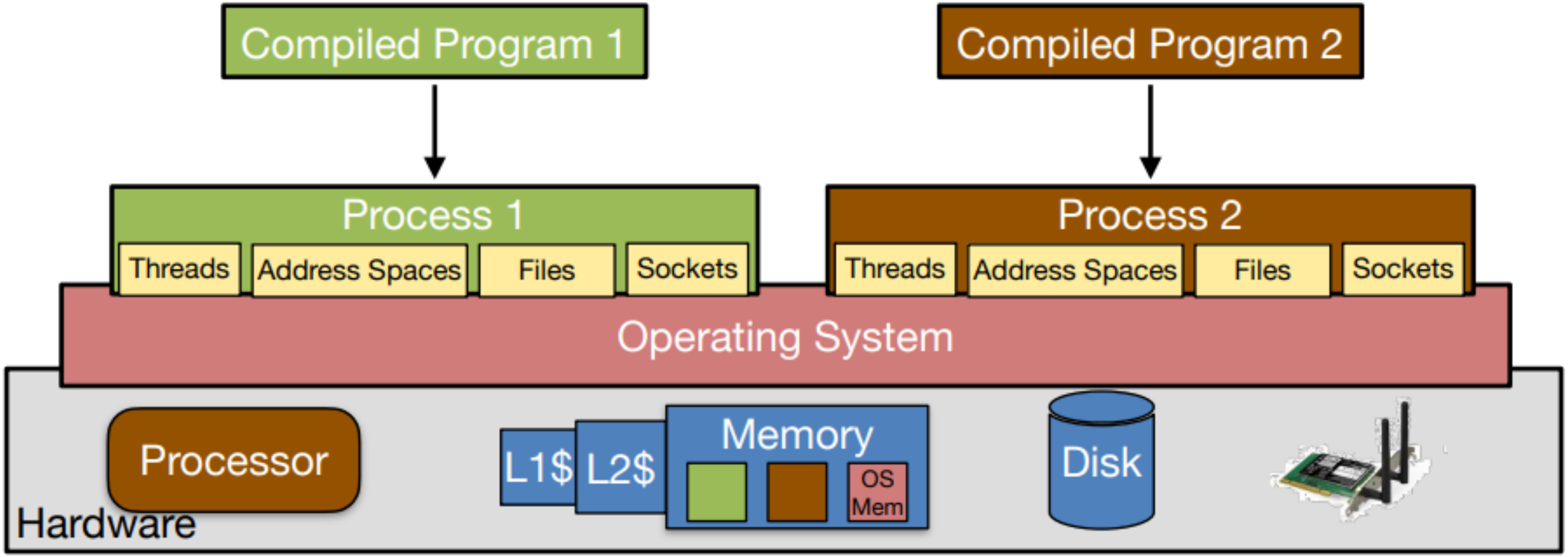
From Hardware View to System View



From Hardware View to System View



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

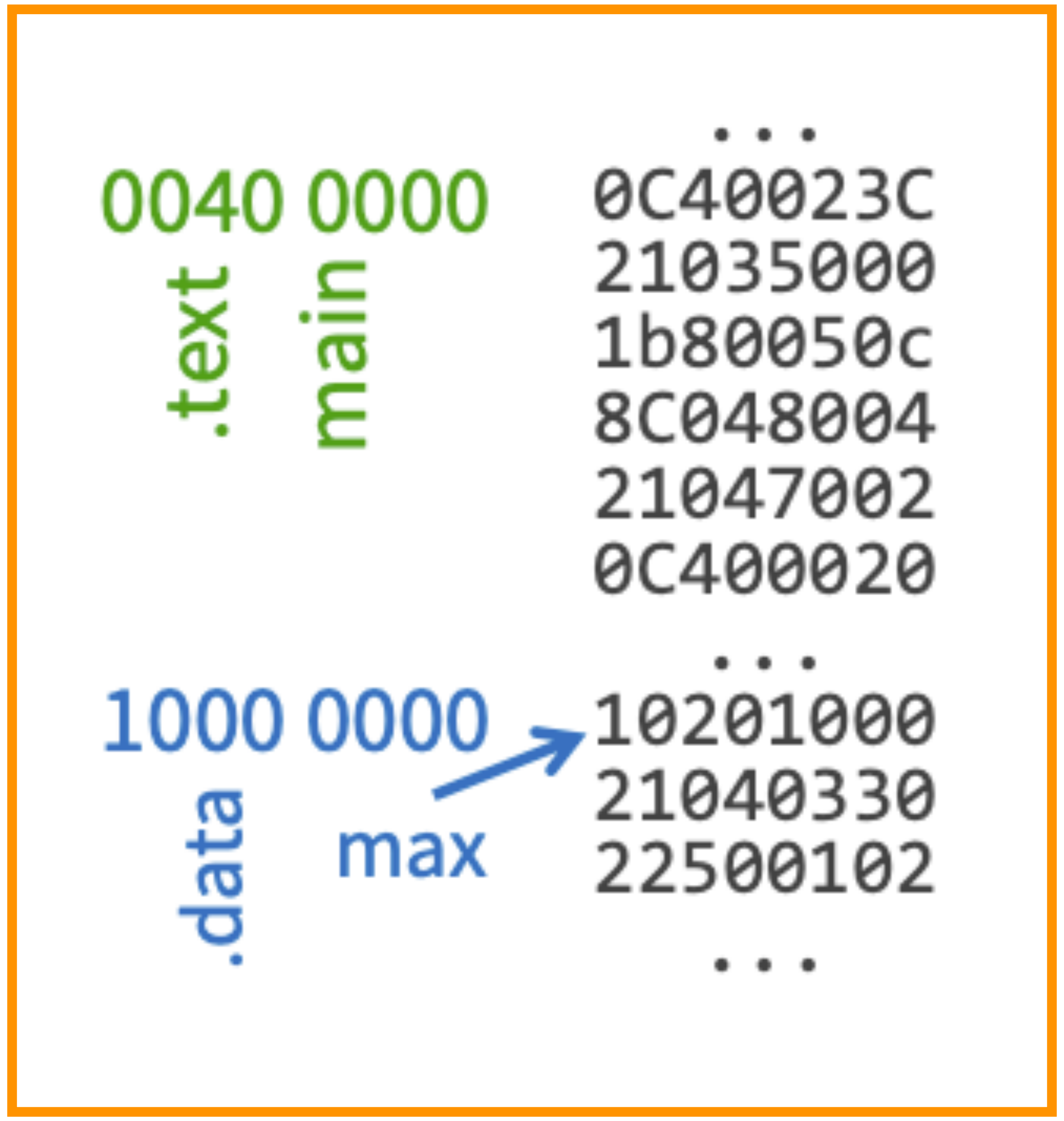
Day in the life of a process

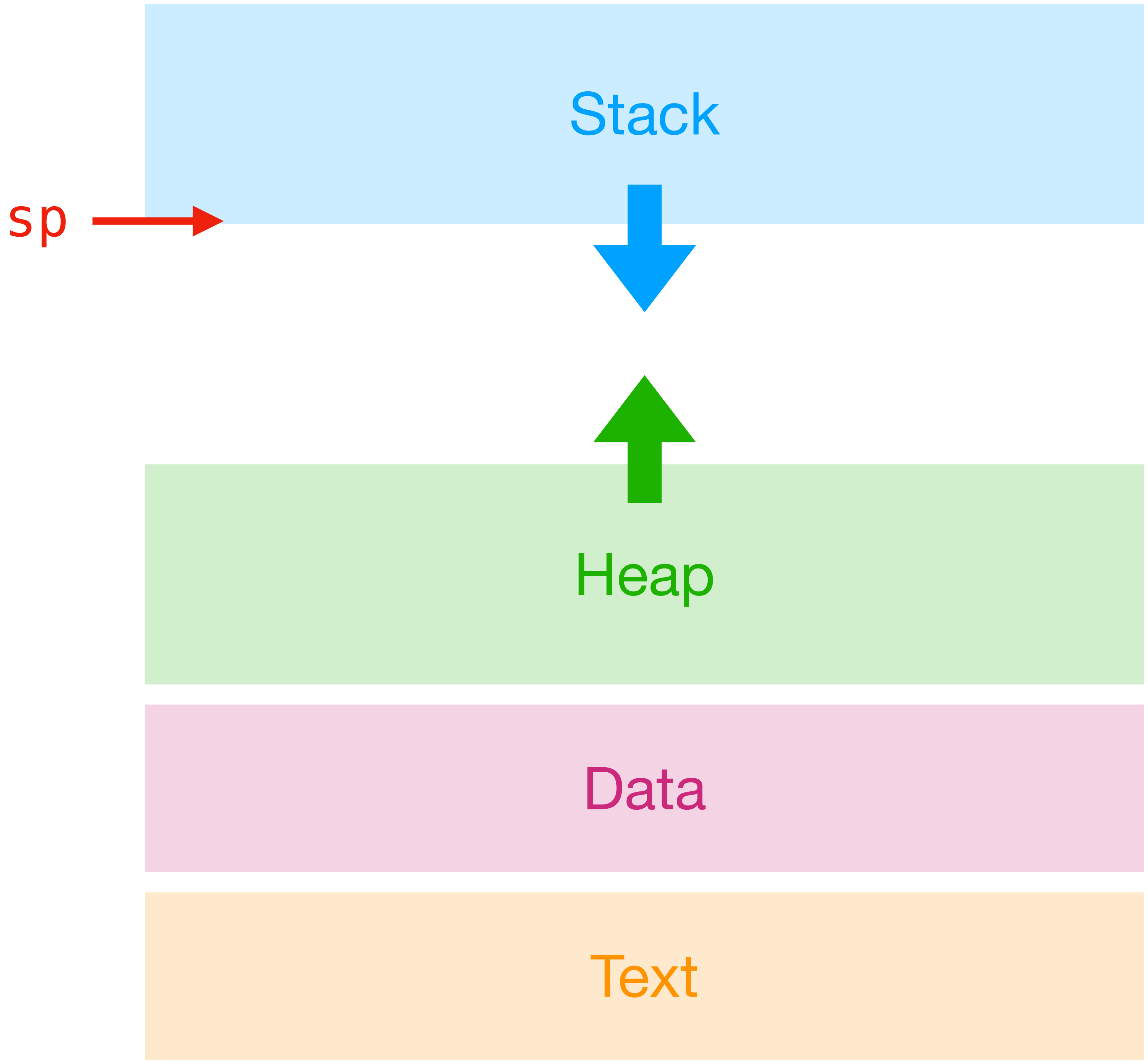
A Day in the Life of a Process

The source file: **`sum.c`** - - - - - - - - - -> The executable: **`sum`** - - - - - - - - - - -> Process is alive: **process** id `pid xxx`

```c
#include <stdio.h>

int max = 10;

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

}
```

program





Environment

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

A Process Physically Runs on the CPU

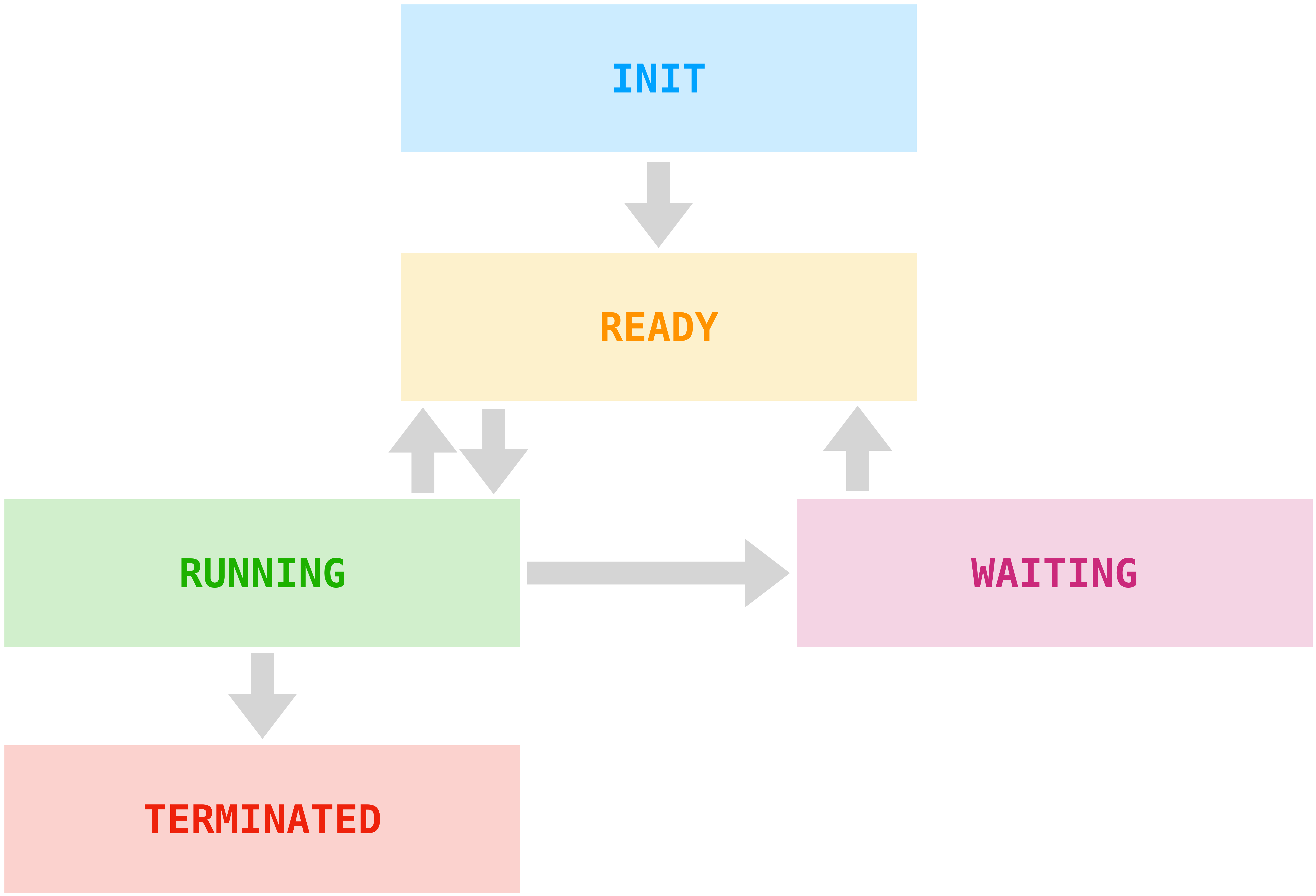
- But somehow each process has its own:
  - Registers
  - Memory
  - I/O resources

- Even though there are usually more proceses than the CPU cores
  - The need to multiplex, schedule, to create virtual CPUs for each process
  - For now, assume we have a single core CPU

Process Control Block (PCB)

- For each process, the OS has a PCB 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 Life Cycle



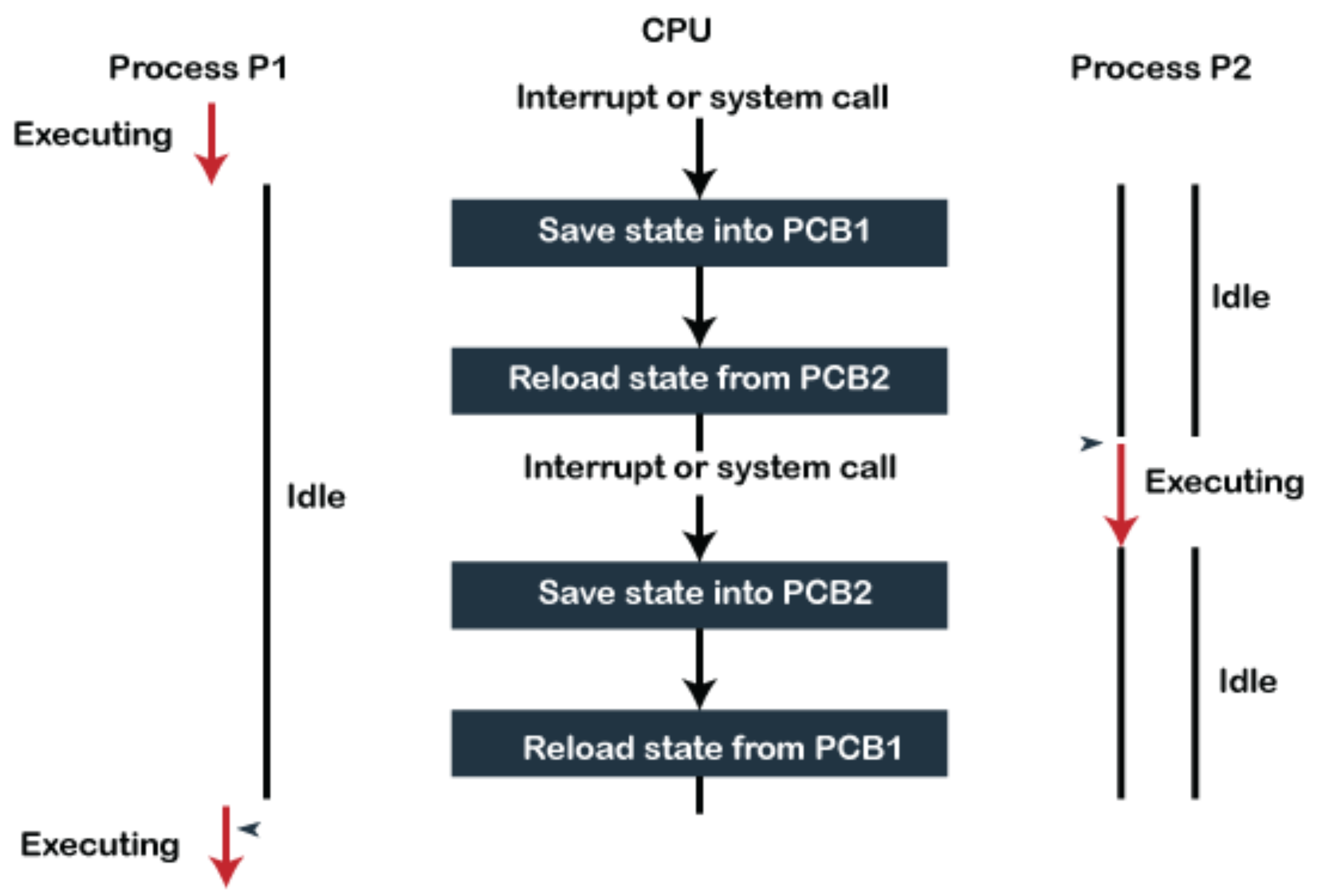
Context Switching

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

Context Switching

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

- First, save the current process state
- Update the Process Control Block (PCB)
- Then, select the next process
- Restore the next process state
- Resume execution



Performance Consideration

Overhead

- Context switching involves overhead because saving and restoring process states takes time
- The goal is to **minimize** this overhead to maintain system performance
- Context switching has to be efficient for the smooth operation of a multitasking system

this is where the core of the operating systems (the kernel) runs

User space versus Kernel space

this is where regular programs live (apps, compilers, browsers, your code, etc.)

User Space versus Kernel Space

- **User space** is where programs (apps, compilers, browsers, your code, etc.) run
  - User space applications **cannot** directly access the system's hardware resources
  - It is **restricted** and **isolated** from the kernel space to ensure system stability and security

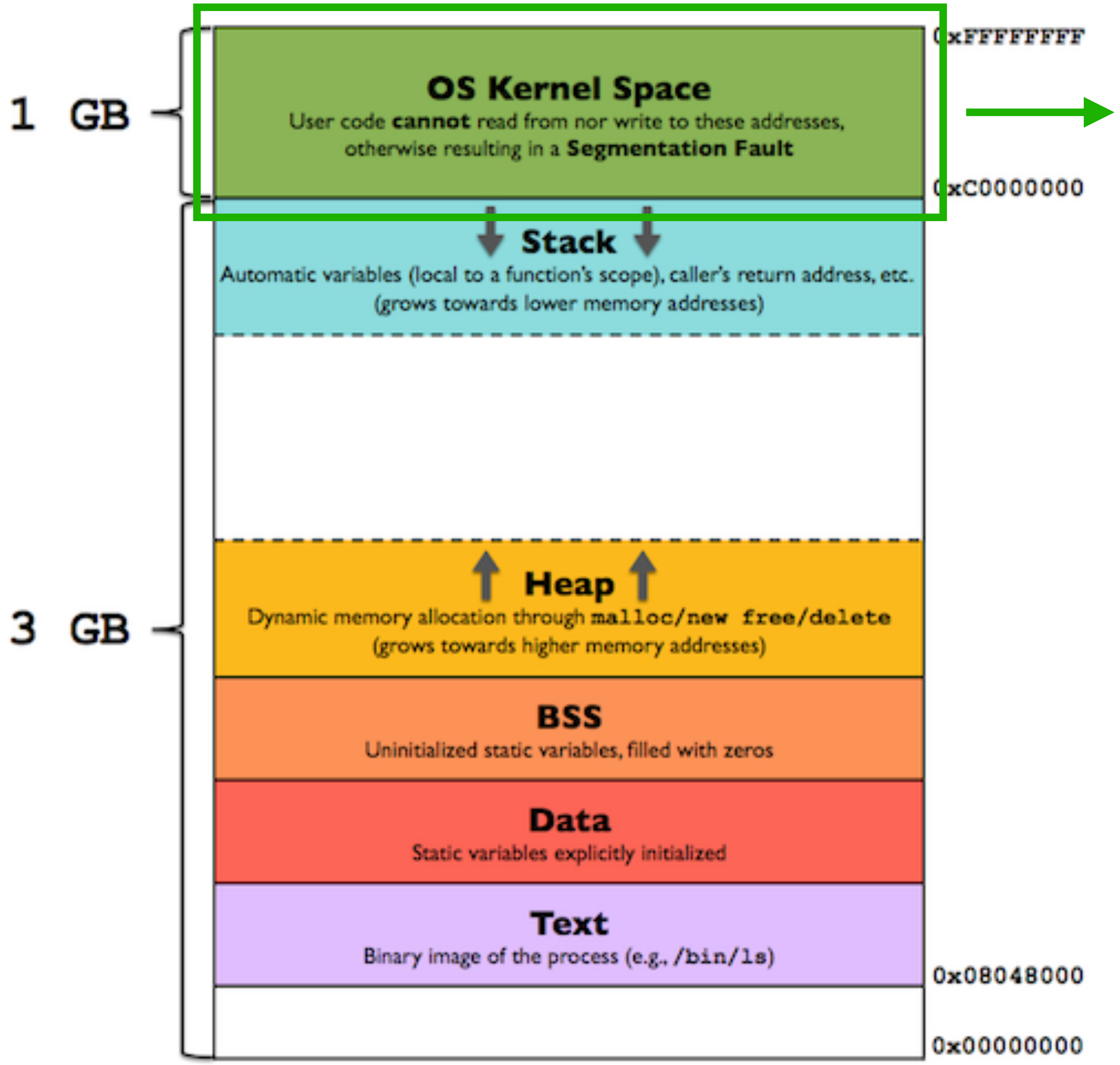
- **Kernel space** is where the core of the **operating system** (the **kernel**) runs
  - It has **full access to hardware** (e.g., CPU, memory, disks, devices)
  - Responsible for: scheduling processes, managing memory, handling I/O, enforcing security and isolation

How They Interact

User programs **cannot** just “walk” into kernel space: yhey have to ask for help through a **system call**

```
# RISC assembly pseudo-code
li   v0, 4             # Load system call code for 'print string’: in user space
la   a0, msg           # Load the address of the message: in user space
syscall                # Trap to the OS: switch from user space to kernel space
...                    # The OS examines v0 (to know which service you’re
requesting) and a0 (the argument)

msg: .asciiz “Hello!" # Data stored in user space
```

Memory Layout 32-Bit Kernel



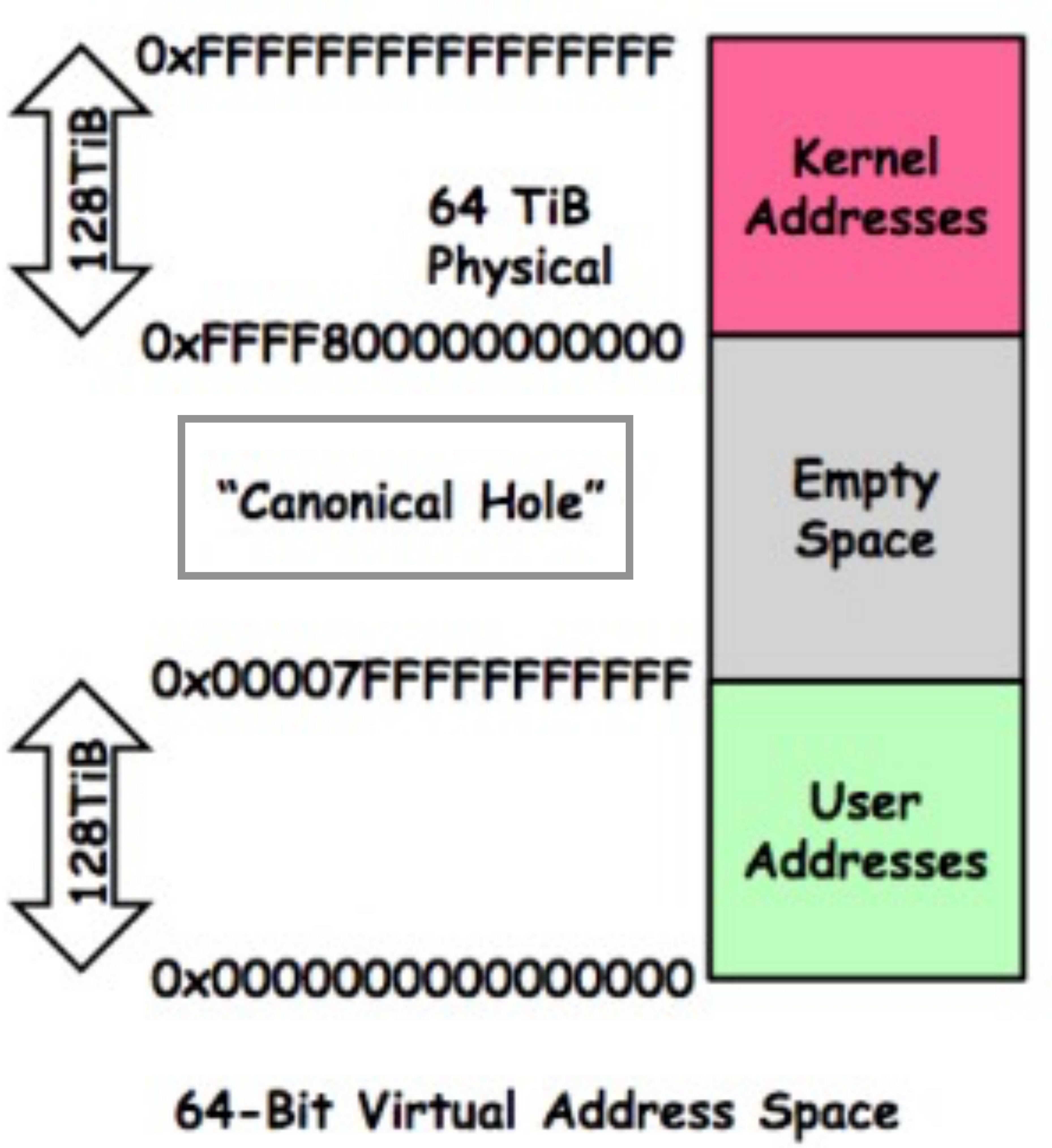
these addresses are **unavailable** in user mode

this is a **software convention**

In a 32-bit system, the **total addressable memory** is 4 GB:

- The division of memory into **1 GB for kernel space** and **3 GB for user space** is a common configuration
- It allows the OS to manage memory efficiently while providing ample space for user applications

Memory Layout 64-Bit Kernel



In a 64-bit system, the **total memory** is 16 **exabytes**:

- Current CPUs don't use all 64 bits of address lines

The available address space is **split into 2 halves separated by a very big hole** called "canonical hole"

The purpose of the **canonical hole**:

- It helps in detecting invalid memory accesses
- It enhances security and stability

System Calls

System Calls

- A process runs on a CPU
- Can access the Operating System (OS) kernel through “system calls”
- A way for the user-space application to request services from the kernel

Why a “Skinny” Interface?

- Portability
    - It’s easier to implement and maintain
- Security
    - It’s a “small attack surface”: easier to protect against vulnerabilities

It’s not just the OS interface; the Internet “IP” later is another good example of a skinny interface

Common System Calls

- **`read()`**: Reads data from a file descriptor
- **`write()`**: Writes data to a file descriptor
- **`open()`**: Opens a file and returns a file descriptor
- **`close()`**: Closes an open file descriptor
- **`fork()`**: Creates a new process
- **`exec()`**: Replaces the current process image with a new process image
- **`waitpid()`**: Waits for a specific child process to change state

Error Handling

- The system calls often return **`-1`** to indicate an error
- The global variable **`errno`** is set to indicate the specific error code
- The **`perror()`** function can be used to print a human-readable error message based on the value of **`errno`**

Fork, Exec, and Waitpid

Ex: `fork()`

- `fork()` is used to create a new process by duplicating the calling process
  - The new process is called the **child** process
  - The original process is called the **parent** process

- `fork()` function prototype:

```
pid_t fork(pid_t pid);
```

- `fork()` is called, then both processes continue executing the code **after the `fork()` call**, but they have **different PIDs**

`fork()` Return Value

- `fork()` function prototype:

```
pid_t fork(pid_t pid);
```

Process	Return value of `fork()`
Parent	`PID` of the child
Child	`0`
Error	`-1`

- If `fork()` fails, it returns `-1` in the parent and no child is created

Ex: fork()

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // child process
        printf("Hello from the child process!\n");
    } else if (pid > 0) {
        // parent process
        printf("Hello from the parent process!\n");
    } else {
        // fork failed
        perror("fork");
    }
    return 0;
}
```

Ex: fork()

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

int main() {
    pid_t pid = fork();
    if (pid == 0) {
        // child process
        printf("Hello from the child process!\n");
    } else if (pid > 0) {
        // parent process
        printf("Hello from the parent process!\n");
    } else {
        // fork failed
        perror("fork");
    }
    return 0;
}
```

Why `fork()` Would Fail?

- Common **`fork()`** failure reasons:
  - The system lacks enough memory to allocate for the new process
  - The system’s process limit has been reached
  - The process lacks the necessary permissions to create a new process
  - Other resource limits are exceeded, e.g. CPU time limit
  - Or even kernel-level issues, e.g., a bug

Ex: `exec()`

- **`exec()`** replaces the current process image with a new process image
  - Commonly used functions: **`execl()`**, **`execp()`**, **`execv()`**, etc.
- **`exec()`** function prototype:

```
int execl(const char *path, const char *arg, ...);
```

- **`exec()`** basically ***changes*** what a process does

Ex: exec()

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

int main() {
    printf("Before exec\n");
    execl("/bin/ls", "ls", NULL);
    perror("execl"); // this will only be executed if exec fails
    return 0;
}
```

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Ex: `waitpid()`

- `waitpid()` is used to wait for state changes in a child process
  - It can be used to wait for a specific child process to terminate

- `waitpid()` function prototype:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Ex: waitpid()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // child runs "ls -l"
        execlp("ls", "ls", "-l", NULL);
        perror("execlp failed");
    } else {
        // parent waits
        int status;
        waitpid(pid, &status, 0);
        printf("Child exited with %d\n", WEXITSTATUS(status));
    }
}
```

How Processes Are Created?

- `fork()`:
  - It allocates the process ID **`pid`**
  - Create and initialize PCB
  - Create and initialize a new address space
  - Then, inform the scheduler a new process is ready to run

How Processes Are Terminated?

- The system calls for termination are:
  - **`exit()`**: used by a process to terminate itself
  - **`abort()`**: used by a parent process to terminate a child process
  - **`wait()`** and **`waitpid()`**: used by a parent process to wait for the termination of a child process and retrieve its exit status

Brief Summary

- A **process** is an abstraction of a computer
- A process is **not** a program
- A **context** captures the state of the processor
- The **implementation** uses two spaces: user space and kernel space
- A **Process Control Block (PCB)** is a kernel data structure that saves context and has other information about the process