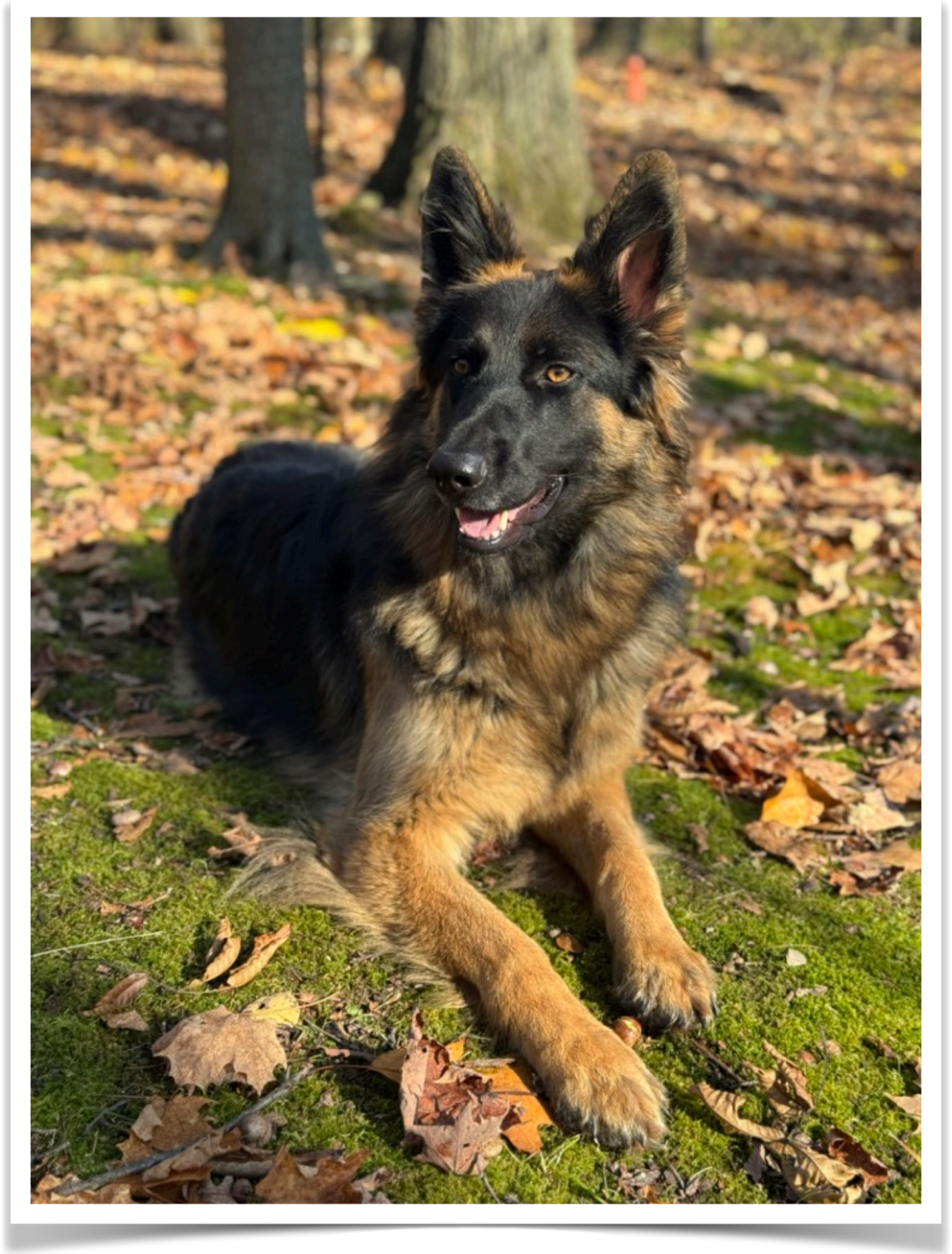
Cornell Bowers CIS
Computer Science



CS3410: Computer Systems and Organization

LEC19: System Calls

Professor Giulia Guidi
Monday, November 3, 2025

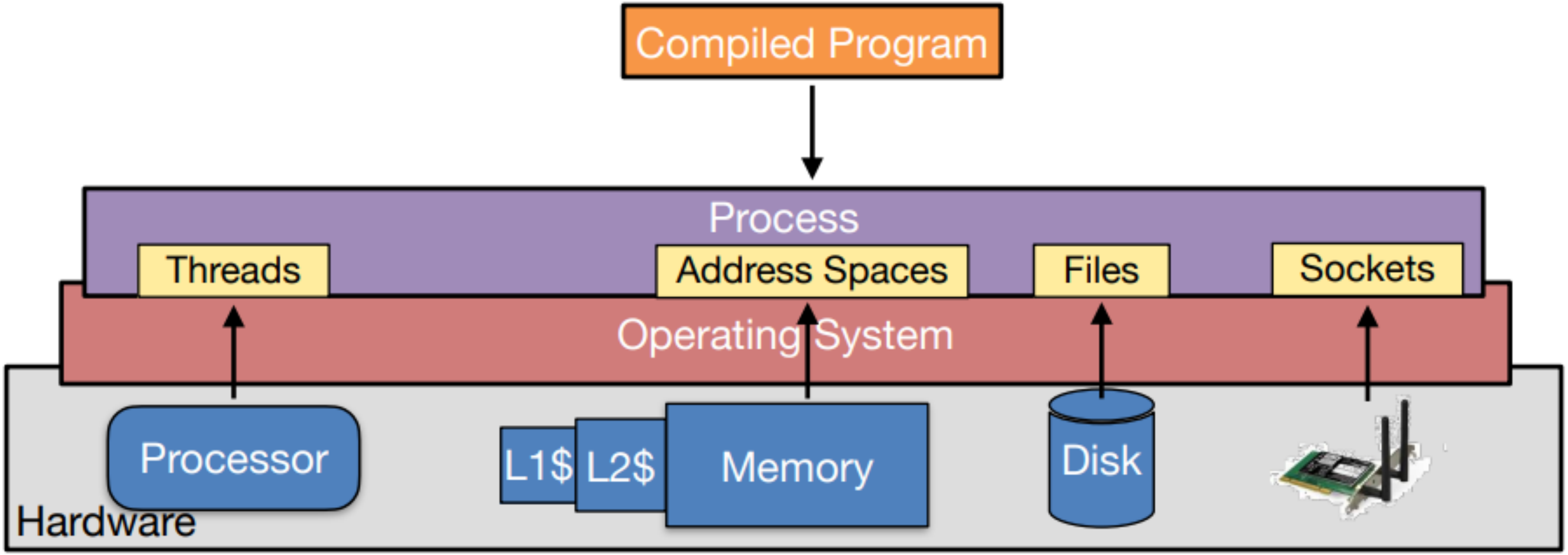
Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Plan for Today

- Review of processes
- The magic world of **syscall**

Review of OS and processes

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?

Conceptual RISC-V Print “Hello”

Conventionally, a7 holds the **system call number**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li   a7, 4             # Load system call code for 'print string'
la   a0, msg           # Load address of message
ecall                  # 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

Conceptual RISC-V Print “Hello”

Conventionally, a7 holds the **system call number**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li    a7, 4              # Load system call code for 'print string'
la    a0, msg            # Load address of message
ecall                    # Call to the OS
...
msg:  .asciiz "Hello!"
```

PC moves from **user** space to **kernel** space (more on this later)

This is a **special instruction** that triggers a **trap** to the operating system (OS)

- The OS looks at `a7` (system call number = 4) and `a0` (address of string)
- It performs the requested service: writing “`Hello!`” to the terminal

Conceptual RISC-V Print “Hello”

Conventionally, a7 holds the **system call number**: it tells the OS **which** service the program is asking for

```
# RISC assembly pseudo-code
li    a7, 4            # Load system call code for 'print string'
la    a0, msg          # Load address of message
ecall                  # Call to the OS
...
msg: .asciiz "Hello!"
```

`.asciiz` = “store the ASCII characters plus a zero byte at the end”

This is a **special instruction** that triggers a **trap** (mechanism to enter the OS) to the operating system (OS)

- The OS can read those bytes from the process’s memory and print them

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 <u>concurrently</u>

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

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

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

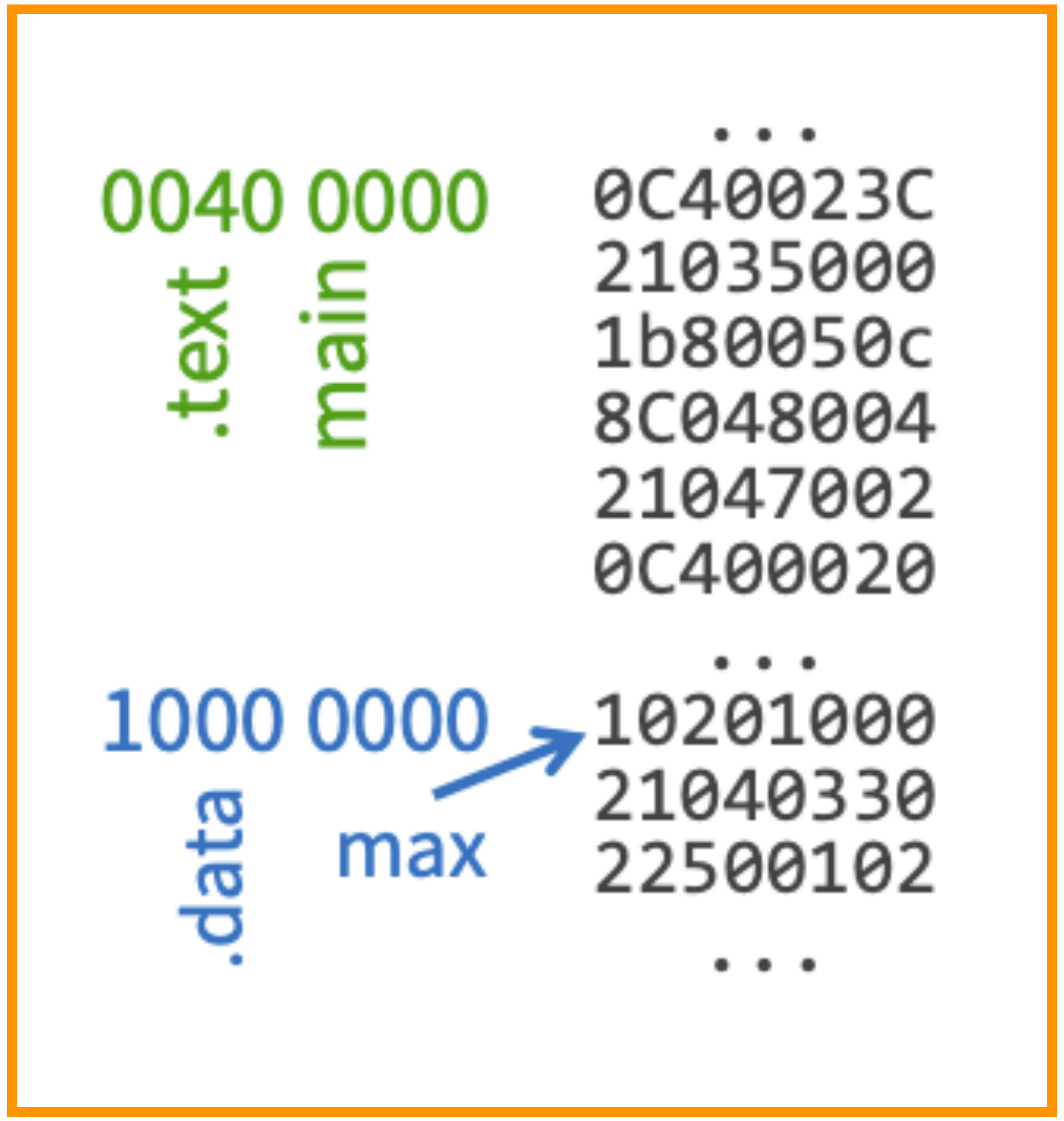
Day in the life of a process

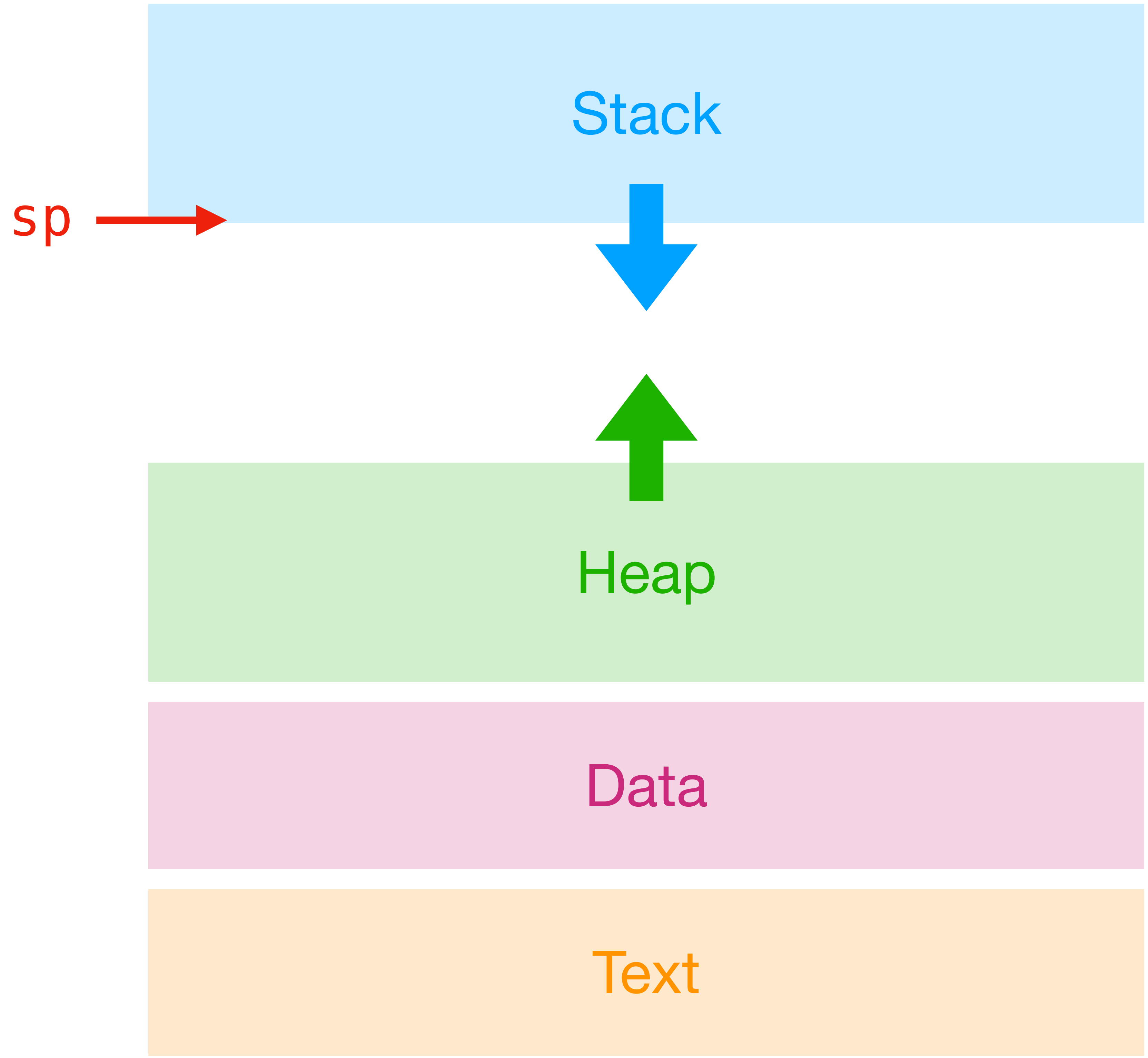
A Day in the Life of a Process

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

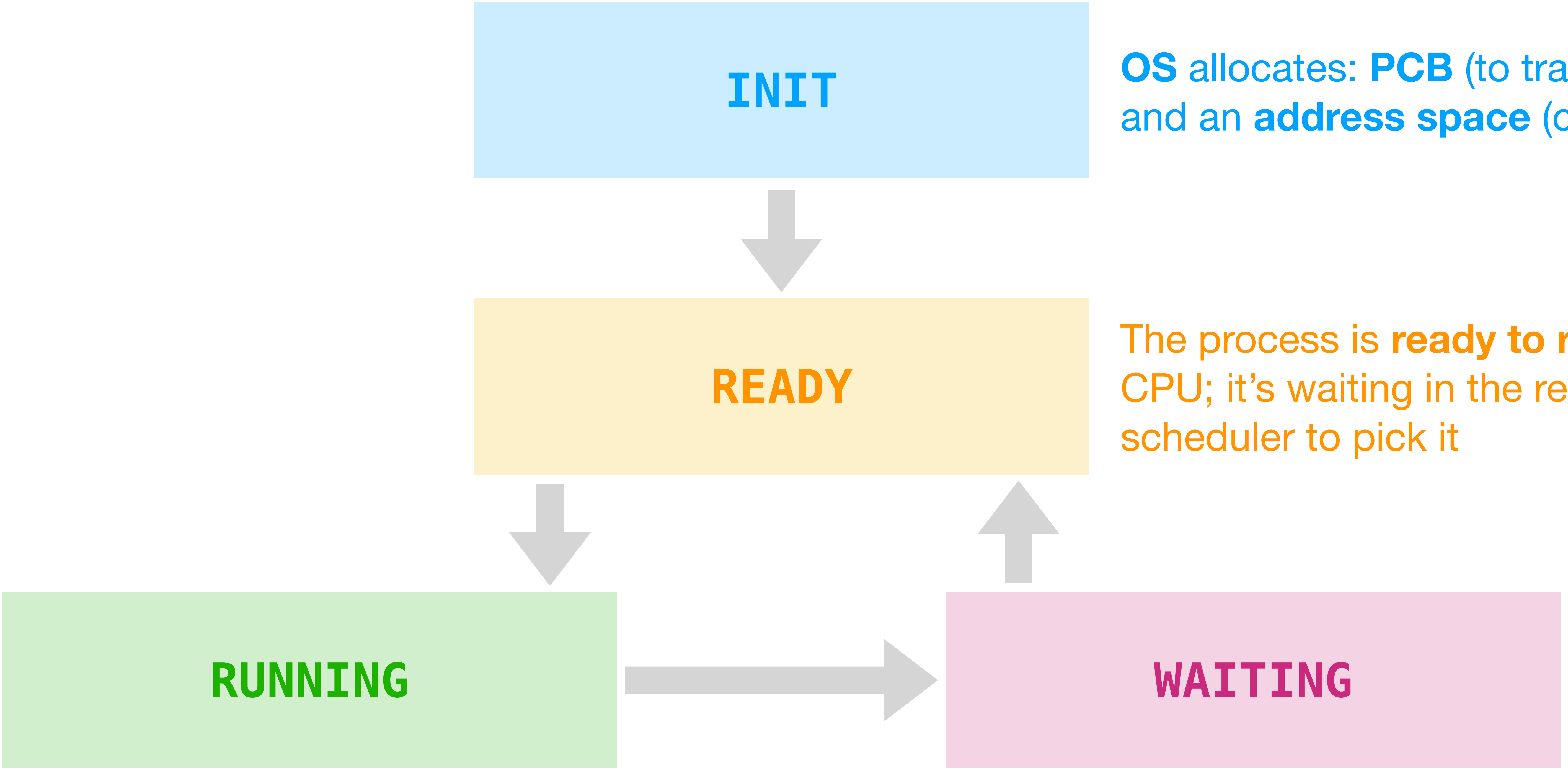




Process Control Block (PCB)

- 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



OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

The process is **now executing** on the CPU

The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

Critical: when the event completes, the OS moves it back to **READY**

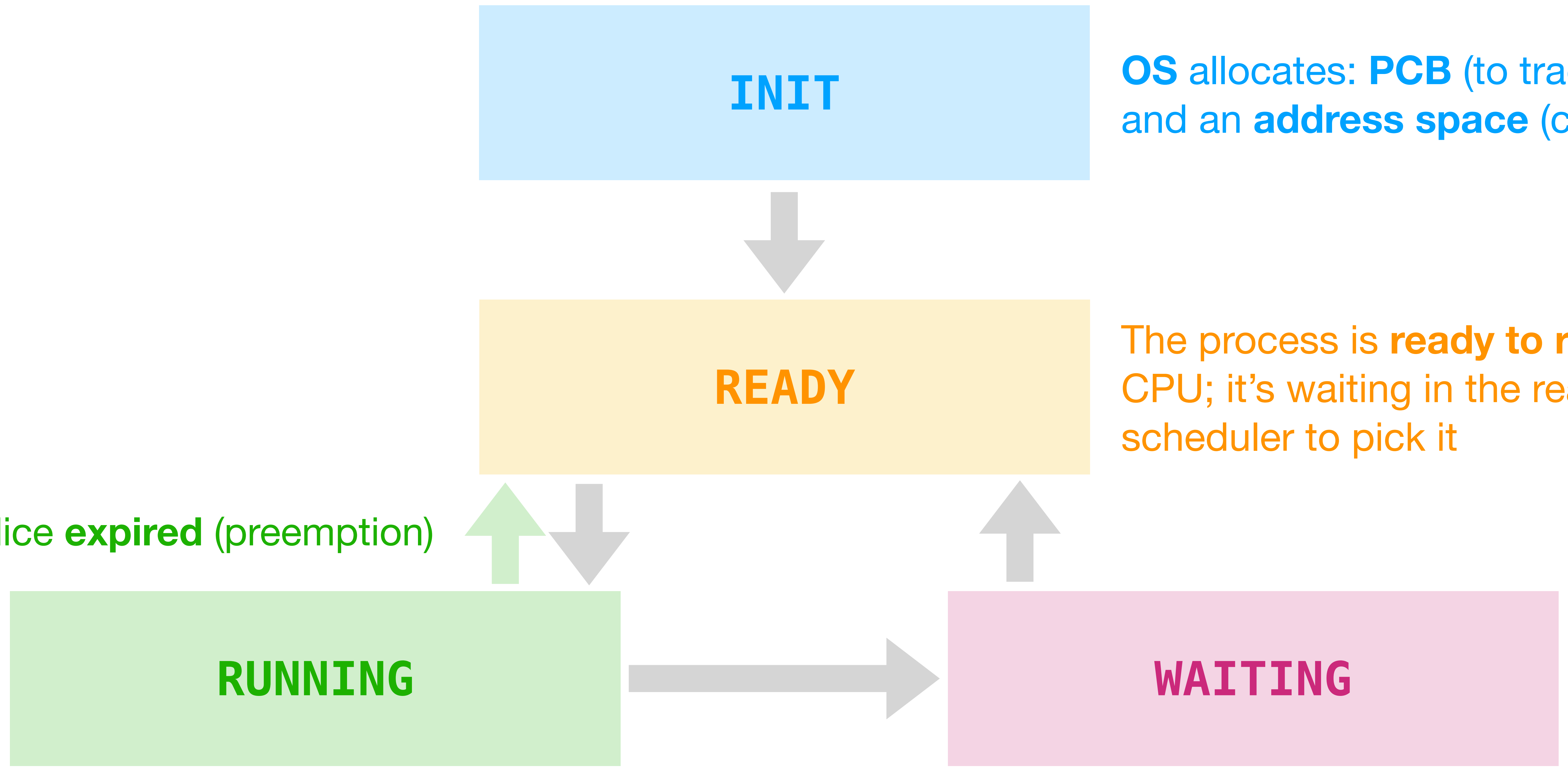
Process Life Cycle

OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

The time slice **expired** (preemption)

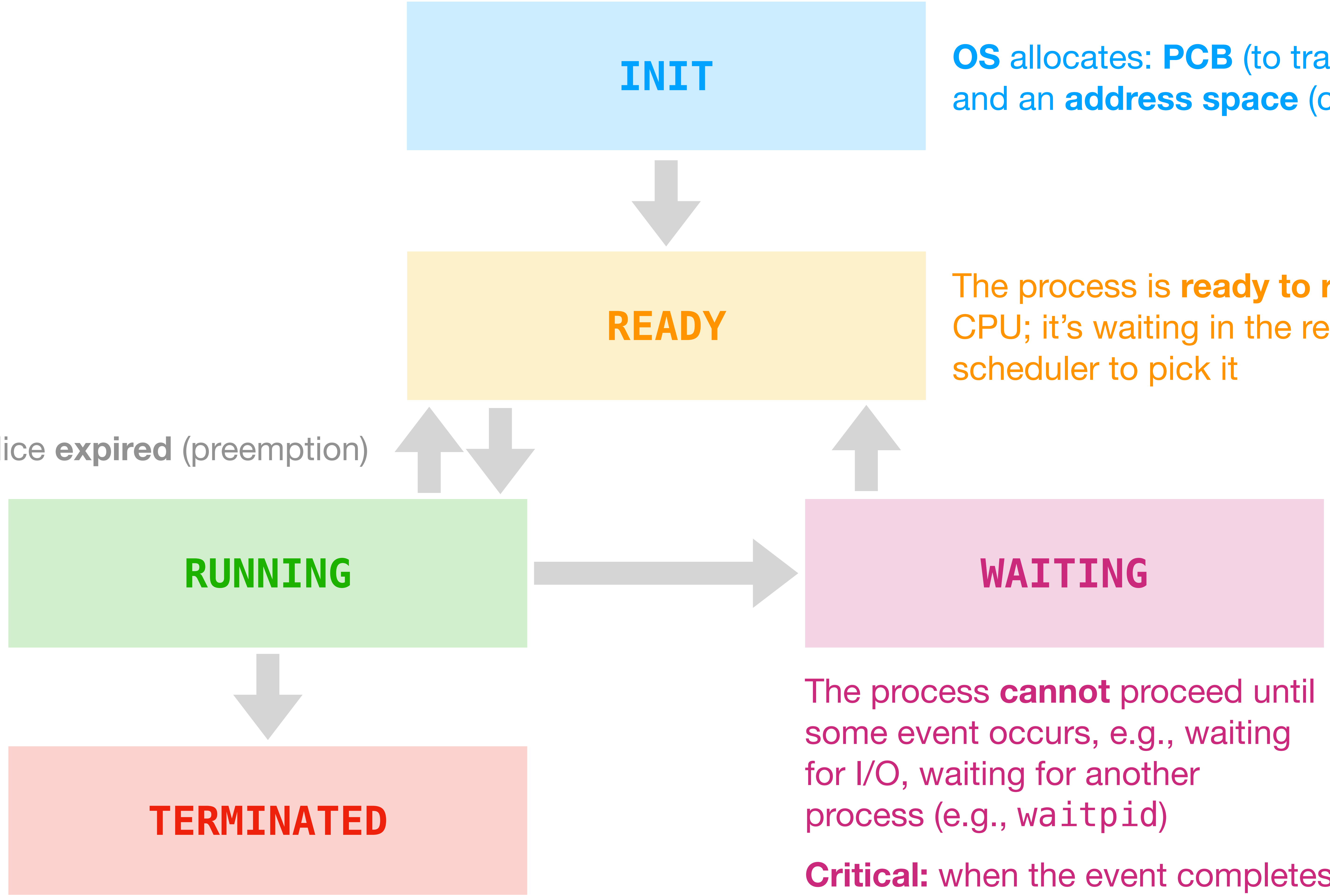
The process is **now executing** on the CPU



The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

Critical: when the event completes, the OS moves it back to **READY**

Process Life Cycle



OS allocates: **PCB** (to track its state, registers, etc.) and an **address space** (code, data, stack)

The process is **ready to run**, but **not** currently on the CPU; it's waiting in the ready **queue** for the scheduler to pick it

The time slice **expired** (preemption)

The process is **now executing** on the CPU

The process has **finished** executing, e.g., reached `exit()`, frees PCB

The process **cannot** proceed until some event occurs, e.g., waiting for I/O, waiting for another process (e.g., `waitpid`)

Critical: when the event completes, the OS moves it back to **`READY`**

Ok, new material

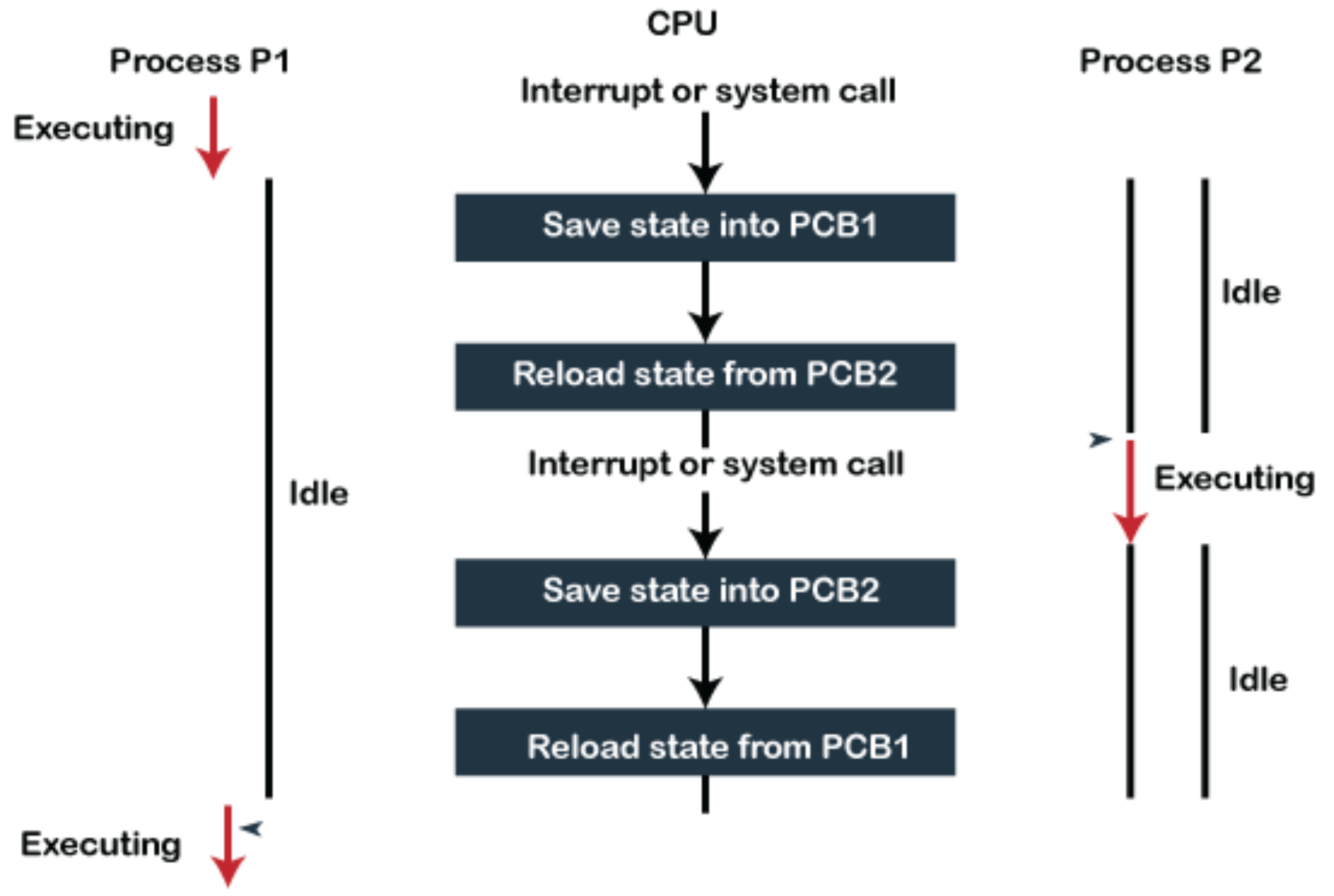
Context Switching

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

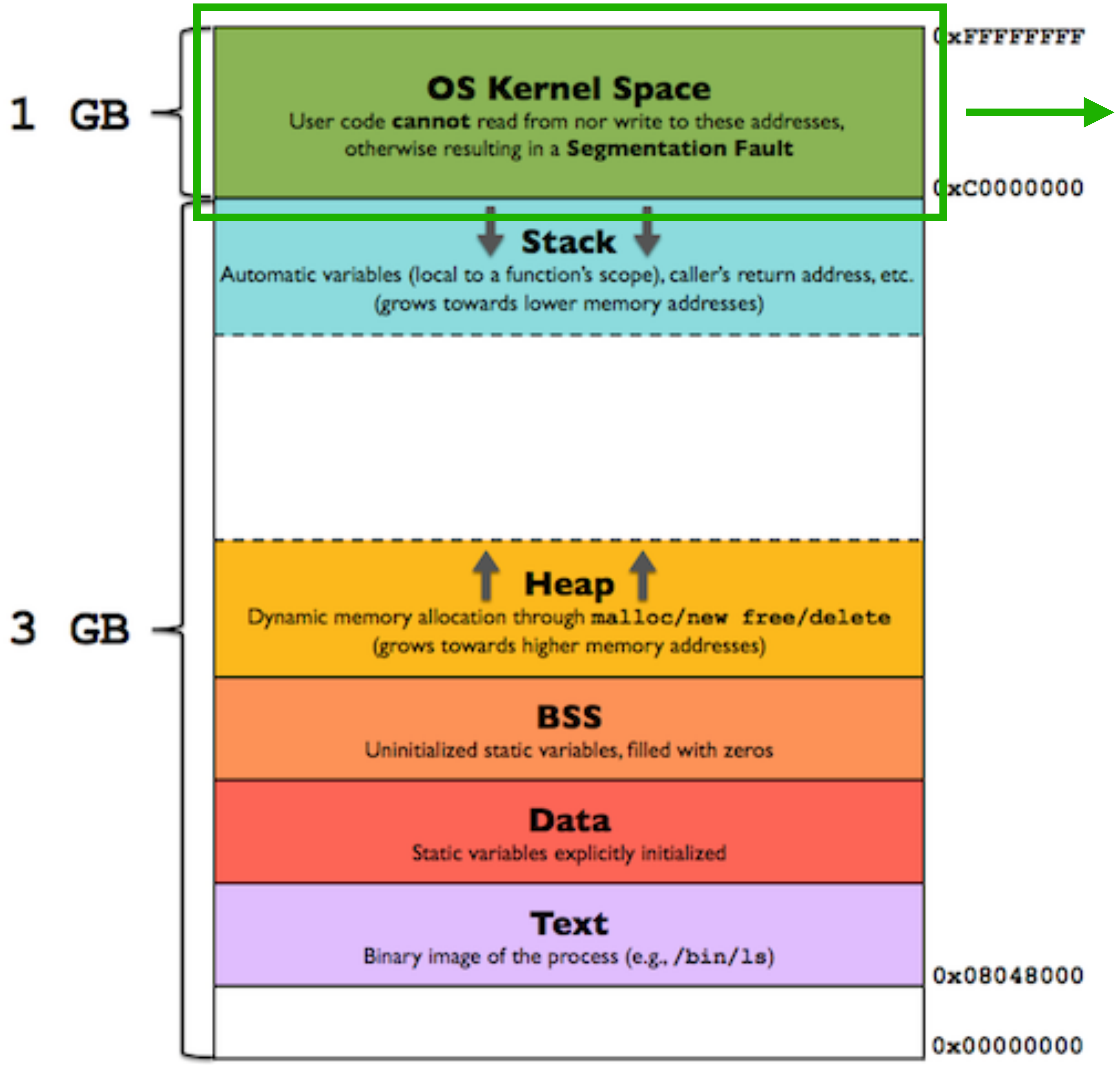
- **Kernel space** is where the core of the **operating system** (the **kernel**) runs
  - It has **full access to hardware** (e.g., CPU, memory, disks, devices)
  - Responsible for: scheduling processes, managing memory, handling I/O, enforcing security and isolation

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   a7, 4              # Load system call code for 'print string’: in user space
la   a0, msg            # Load the address of the message: in user space
ecall                   # 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



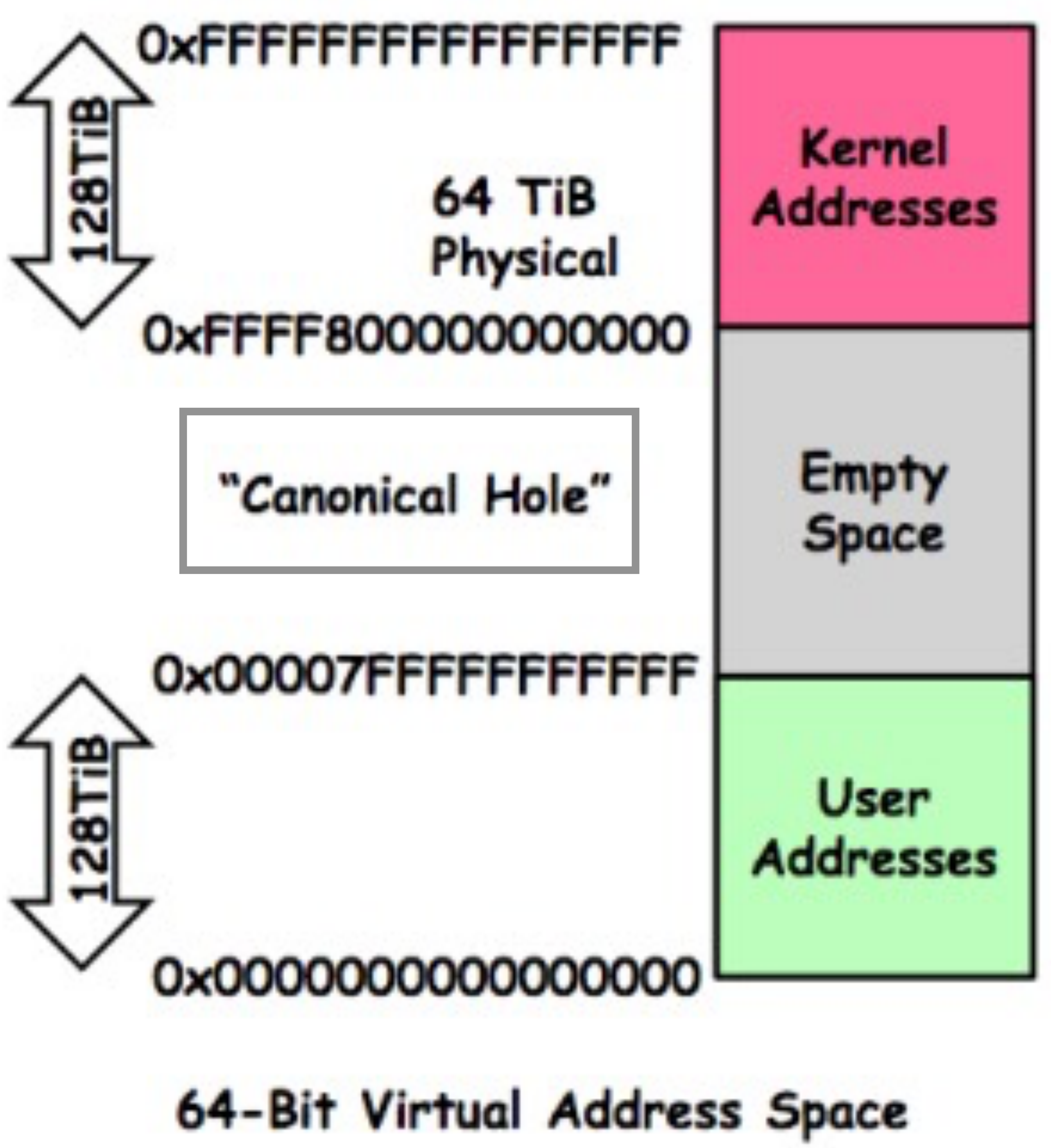
these addresses are **unavailable** in user mode

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

Common 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`**

Error Handling

```c
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

int main() {
    int fd = open("nonexistent.txt", O_RDONLY);
    if (fd == -1) {
        perror("Open failed");
        printf("errno = %d\n", errno);
    }
}
```

- `-1` → the system call error
- `perror` → prints human-readable message
- `errno` → the specific error code

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()

```c
#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;
}
```

The **`fork()`** call returns the **`pid`** of the child process to the parent process

The **`fork()`** call returns **0** to the newly created child process

If **true**, it means you're the child process

If **> 0**, it means you're the parent process

If **< 0**, it means something went wrong!

If **`fork()`** fails, it returns **-1** to the parent process

The **`getpid()`** function returns the **PID** of the calling process

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