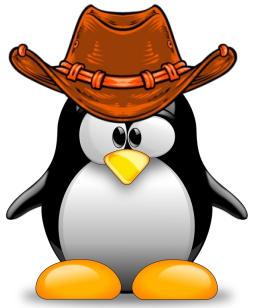


CSE 330:
Operating Systems

Adil Ahmad

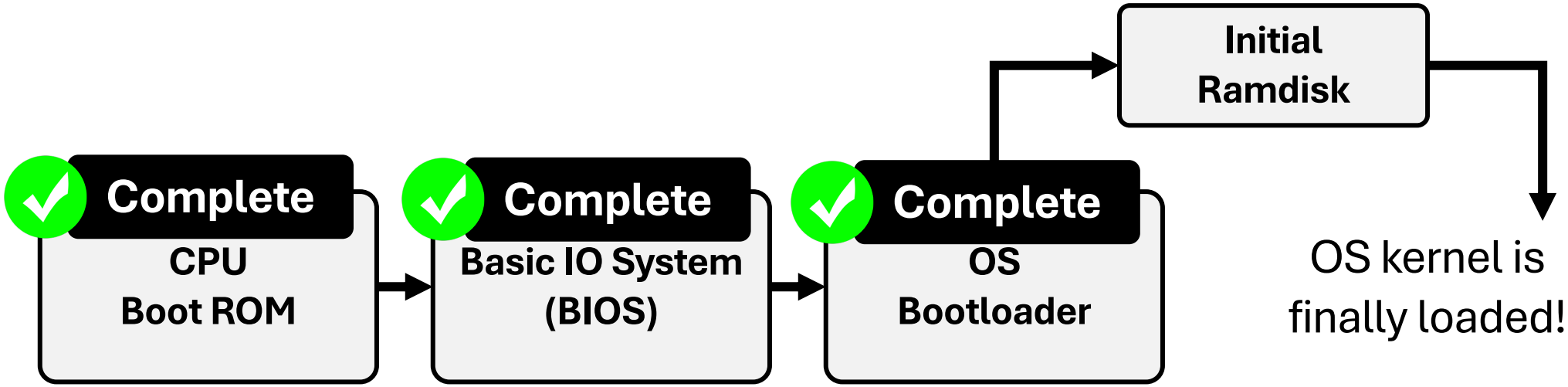
Lecture #4: Process creation and scheduling introduction

Last week, we discussed the **system boot process**

- There are many steps before an OS boots on your computer
- They are required to ensure that your OS operates correctly, e.g.,
  - Check that the CPU/motherboard is functional *(vendor-specific)*
  - Ensure the “correct” and “expected” OS kernel is booted
  - Provide the OS with initial system configuration (e.g., RAM, devices)

We further discussed the four important boot stages

~~Three~~ Four **important stages** of a typical boot, before the OS kernel is loaded



- CPU init.
- BIOS loading

- POST
- MB init/config
- Secure boot

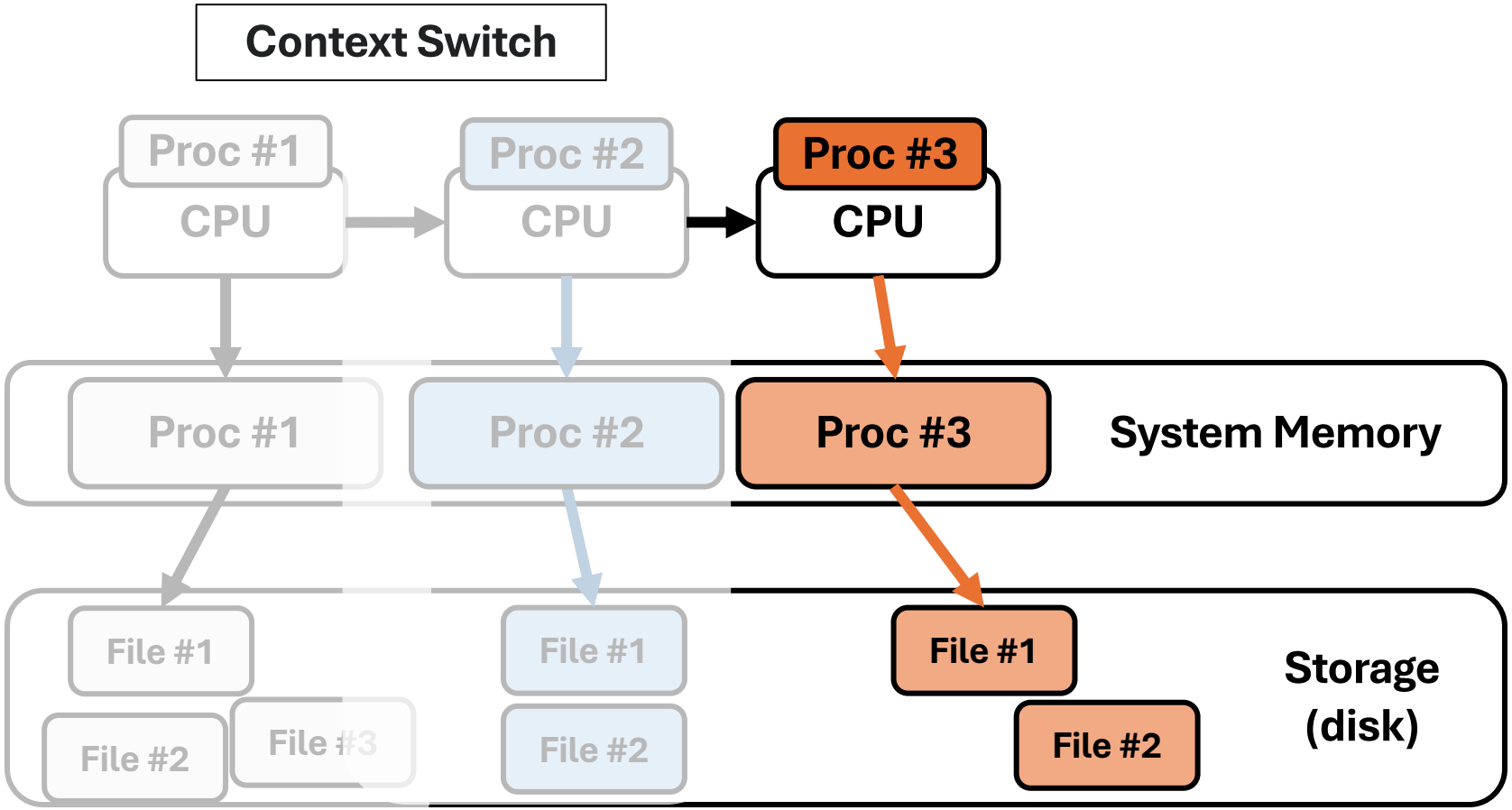
- Arch. setup
- Load/verify kernel image

After the system boots, the real fun starts!

- OS wants to start running programs for users and sometimes also for itself (also called **kernel daemon process**)

- Recall the following:
  - *Programs* are a set of codes (static entity) or executable files
    - e.g., compile a "Hello World" C program → a.out
  - *Processes* are the *running version of* programs
    - We can run the same program in multiple processes
    - One host can run multiple processes

Overview of processes after the system boots

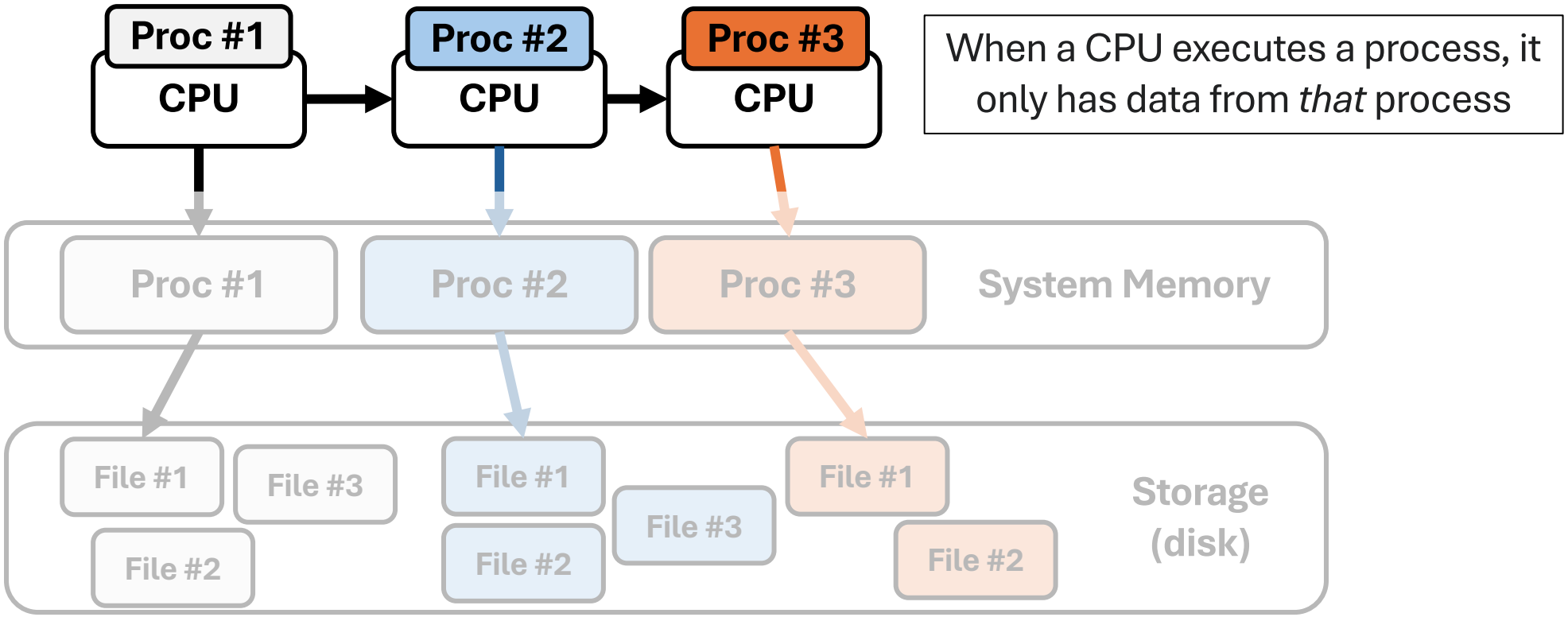


Two important concepts to grasp

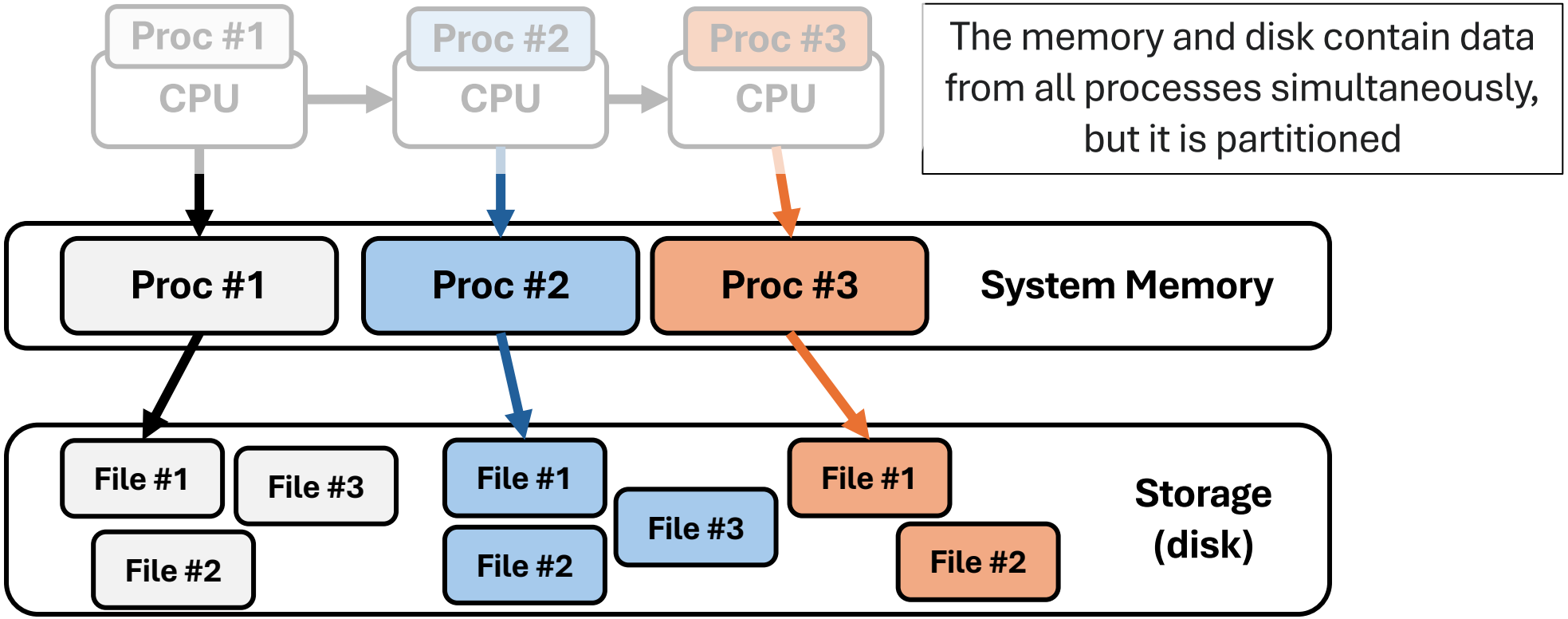
1. Time-sharing
2. Space-sharing

Can anyone tell me what these concepts are?

Understanding the concept of **time-sharing**



Understanding the concept of **space-sharing**



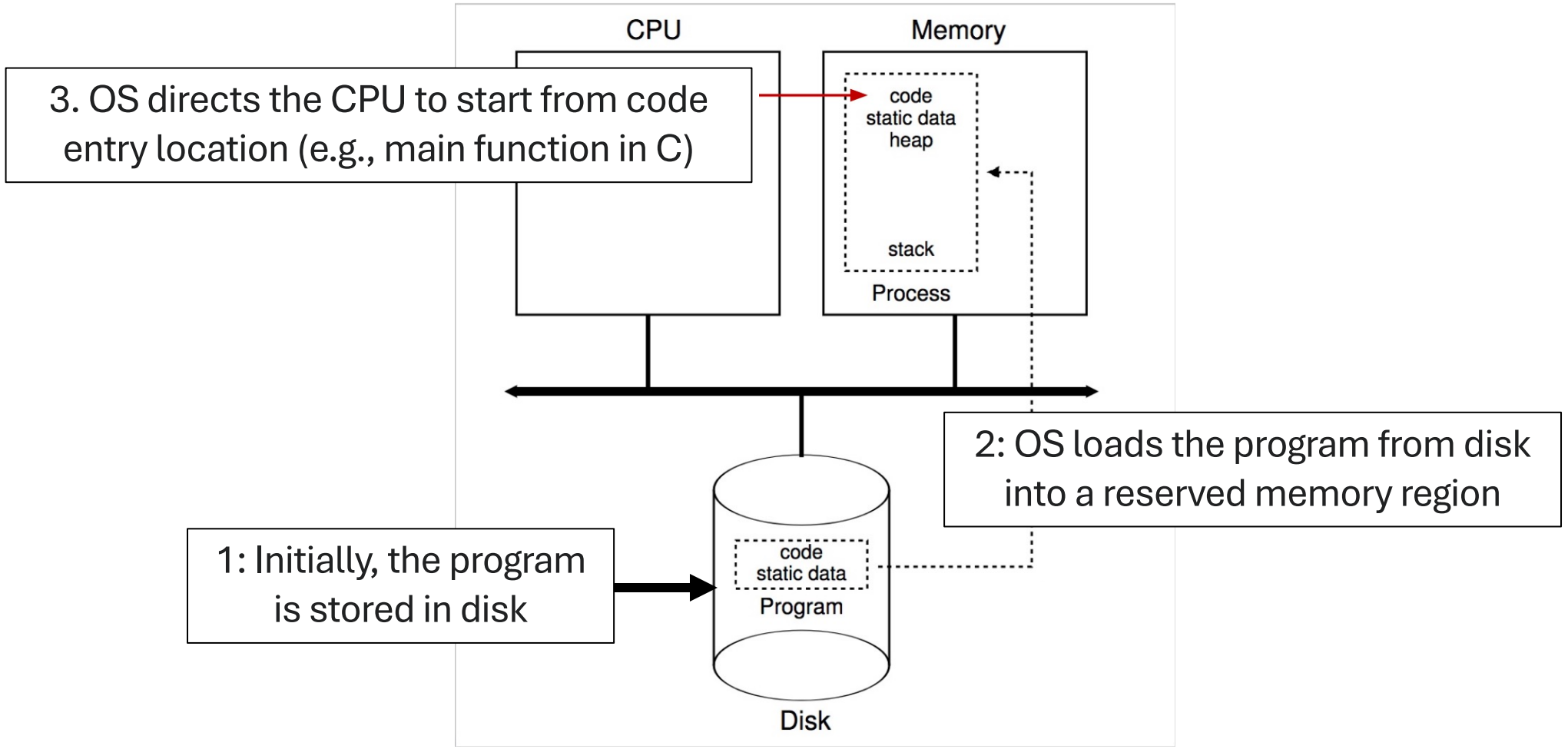
Let's start with process execution from **memory**

- OS does not show all system memory to a process, rather abstracts a portion of memory to each process
- View of memory shown to each process is called its **address space**
  - Also think as "list of addresses that can be accessed by a process"

What are the different parts (or segments) within the address space?

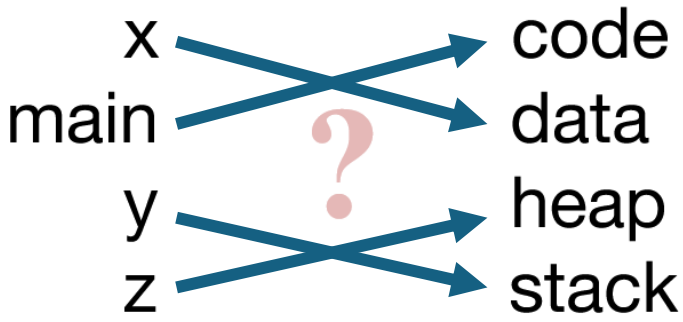
- Generally, four segments: *code*, *data (global)*, *stack*, and *heap*

An illustration of typical process creation



Can you match the provided variables to the program's segments?

```c
int x;
int main(int argc, char *argv[]) {
    int y;
    int *z = malloc(sizeof(int));
}
```



Next, let's look at process execution from the **CPU**

- Code and data from the process' memory will be loaded onto the CPU's registers when it executes, and results written back to memory

- Let's see an example:

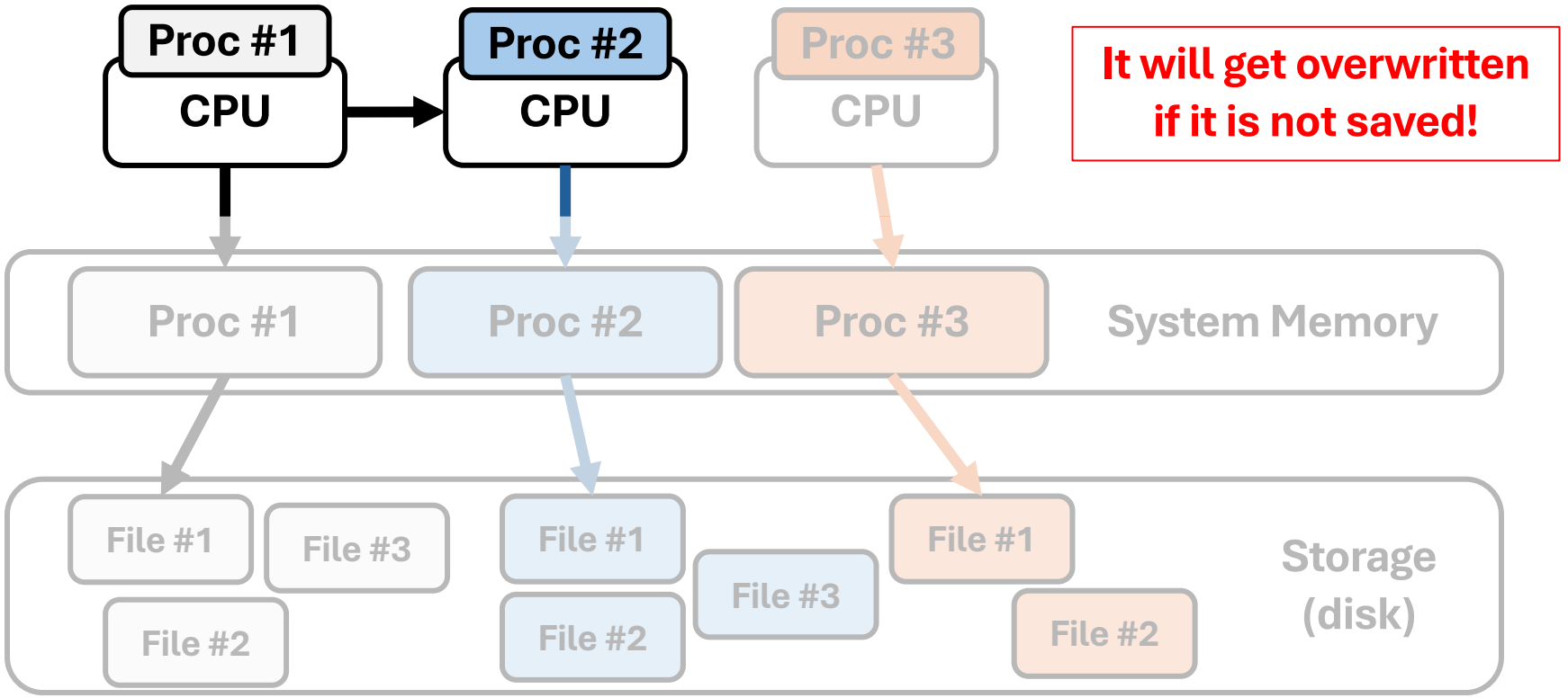
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 2;
}
```

High-level operations performed by CPU:

1. read "x" from memory into a register (*edi*)
2. add 2 to *edi*
3. write *edi* back to the memory location "x"

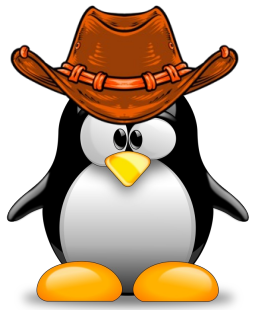
What happens to process data still in CPU registers on a switch?



That's why we need a **process control block**

- The OS saves all the *register values* of a process in a memory region reserved for the process, i.e., the **process control block**

- Not just the registers, also other kernel-specific information (e.g.,
  - Which files belong to the process?
  - Which files are currently open?
  - What is the memory range, etc.

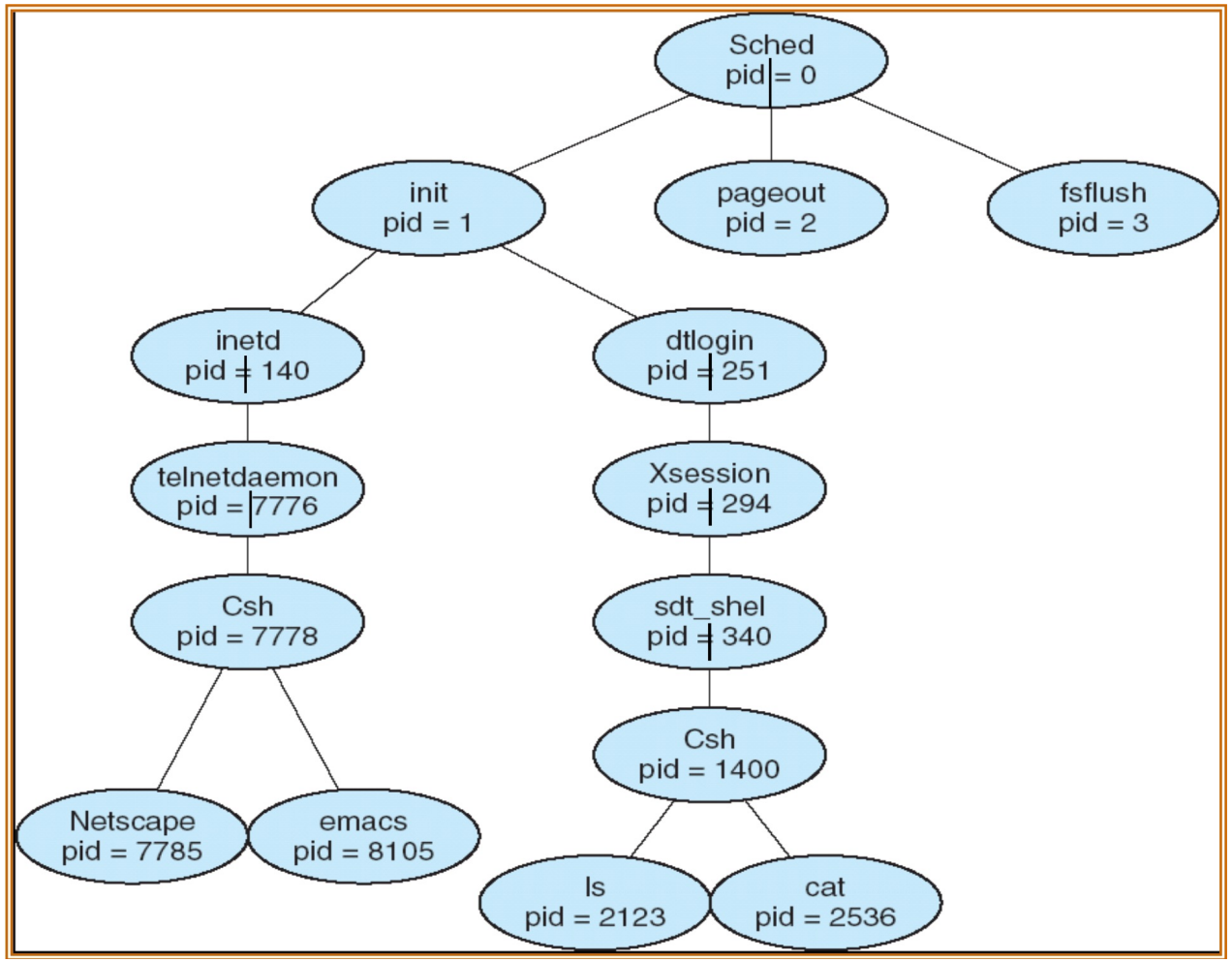


Process parents, creation, and Linux APIs

Process creation from parents

- During system boot, the kernel creates a “grandparent” process called “init” for all user-space (unprivileged) processes
- This process creates every other user-space process (e.g., your Linux terminal, etc.)
- Information about parent → child relationships is stored by Linux kernel using a tree structure

A sample Linux kernel process tree



Creating a child process: The **fork** system call in Linux

- The parent executes `fork` system call to create a child process
  - `int pid = fork()`
- The child process has a separate *copy* of the parent's address space
- Both the parent and the child continue execution at the instruction following the fork() system call
- The return value for the fork() system call is
  - *Zero value* for the new (child) process
  - *Non-zero pid* for the parent process

Example program with the fork() system call

```
void main () {
int pid = fork();
if  (pid < 0) {/* error_msg */}
else if (pid == 0)
{
      /* child process */
      execl("/bin/ls", "ls", NULL); /* execute ls */
} else {
      /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      exit(0);
    }
return;
}
```

Since pid == 0, only child process will execute this

Since pid != 0, only parent process will execute this

What happens to the value of “number” in this fork example?

```
int number = 7;

int main(void) {
        int pid = fork();
        if (pid == 0) {
                number *= number;
                printf (“number is %d\n”, number);
                return 0;
        } else if (pid > 0) {
                wait (NULL);
                printf (“number is %d\n”, number);
        }
        return 0;
}
```

7*7 = 49

7

What happens to the value of “number” in this fork example?

```
int number = 7;

int main(void) {
        int pid = fork();
        if (pid == 0) {
                number *= number;
                fork();
                printf (“number is %d\n”, number);
                return 0;
        } else if (pid > 0) {
                wait (NULL);
                printf (“number is %d\n”, number);
        }
        return 0;
}
```

7*7 = 49

49

Creating a child process: The **execl** system call in Linux

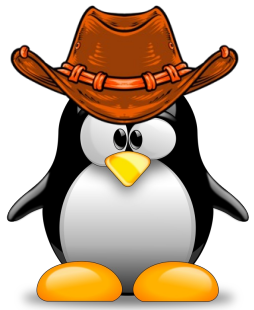
- The parent executes `execl()` system call to create a child process
- The child process has a separate “new” address space
  - No data is copied from the parent (except arguments)
- Parent is terminated and only child resumes execution
- Child process starts from its “main” function

A simple terminal shell designed with **fork** and **execl**

```
while (true) {
        type_prompt();
        read_command(cmd);
        pid = fork();
        if  (pid < 0) {/* error_msg */}
        else if (pid == 0) {
            /* child process */
                execl(cmd);
        } else {
            /* parent process */
            wait(NULL);
        }
}
```

Starts executing command (cmd) in a child process

Parent “shell” waits for child to execute command and then resumes



Process scheduling

Brief introduction

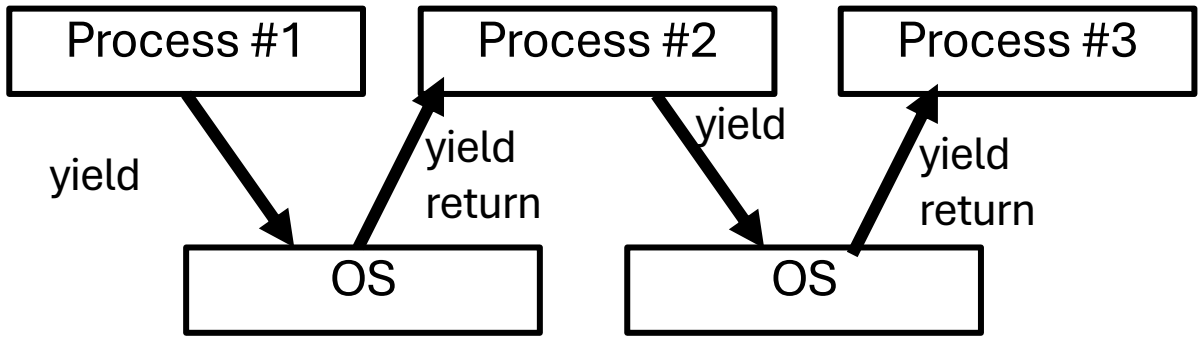
- Unlike memory regions, we don't have many CPUs (only one pre-2007)
  - Many more processes than CPUs even today!
- Hence, CPUs must be **time-shared** between different processes
  - Divide CPU time into slices (e.g., 1 millisecond, 1 second, etc.)
  - Give CPU to a certain process for the entire slice
- Which process gets how many time slices and when → **Scheduling**
  - That's what the next few lectures are all about!

One *non-trivial* problem with enforcing scheduling

- When a process starts on a CPU (i.e., it is scheduled), it “owns” the CPU, and the OS does not actually run at that time
- **Can the OS perform scheduling if it’s not executing?**
  - **No, it cannot!**
- The OS must execute on the CPU core to decide scheduling matters for the CPU
- **When does the OS execute on a certain CPU core?**
  - On system calls and interrupts

Cooperative scheduling through a system call

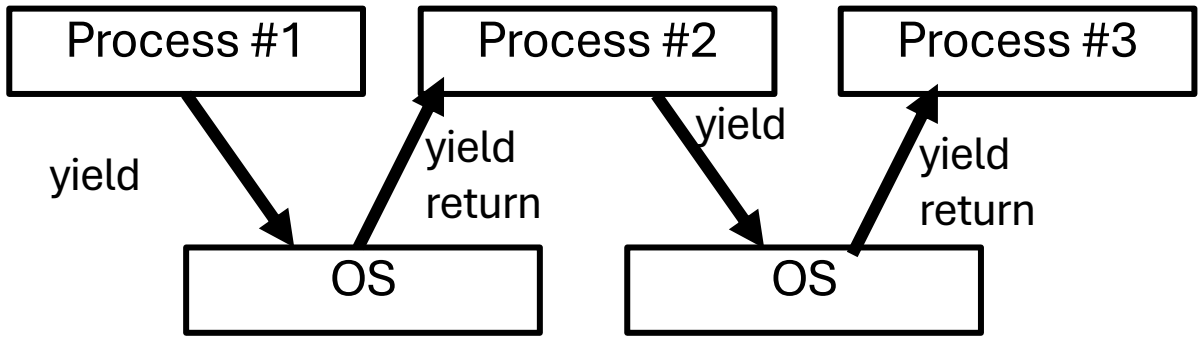
- yield() system call



- Super-simple scheduling approach and it is common on tiny “embedded” systems today

What's the potential problem with cooperative scheduling?

- Real-world programs may be malicious or buggy



- What if the program chooses not to execute yield? We must not allow that and we need to forcefully stop them

Alright, let's use an **interrupt** instead of a system call

- OS will also be executed on a CPU when a device (e.g., SSD, Bluetooth) fires an interrupt
  - In this case, it does not matter if the process "voluntarily" wants to give up its time or not. We will forcibly take away the CPU!
  - This is also called **"Preemptive Scheduling"**

What's the problem with device interrupt-based scheduling?

- OS will also be executed on a CPU when a device (e.g., SSD, Bluetooth) fires an interrupt
  - In this case, it does not matter if the process "voluntarily" wants to give up its time or not. We will forcibly take away the CPU
  - This is also called **"Preemptive Scheduling"**

- **Typical device interrupts are unpredictable.**
  - Sometimes interrupts may be fired very frequently, other times they might not be fired for 10s of seconds.
  - **Does not provide fine-grained, predictable** control to the OS

Naïve approach: reserve a CPU core for the OS

- A CPU core can send an interrupt to other CPU cores
  - **Inter-processor interrupts (IPIs)**
- When the system boots-up, the OS starts on a CPU core and never lets go (aka don't run any process on this core)
- The CPU uses its hardware counter to send an IPI after a configured interval (e.g., 1s) to each CPU core

  ➢ This gives **"predictable control", but do you notice any problems?**

Let’s not waste a CPU core and use a cheap “timer” HW

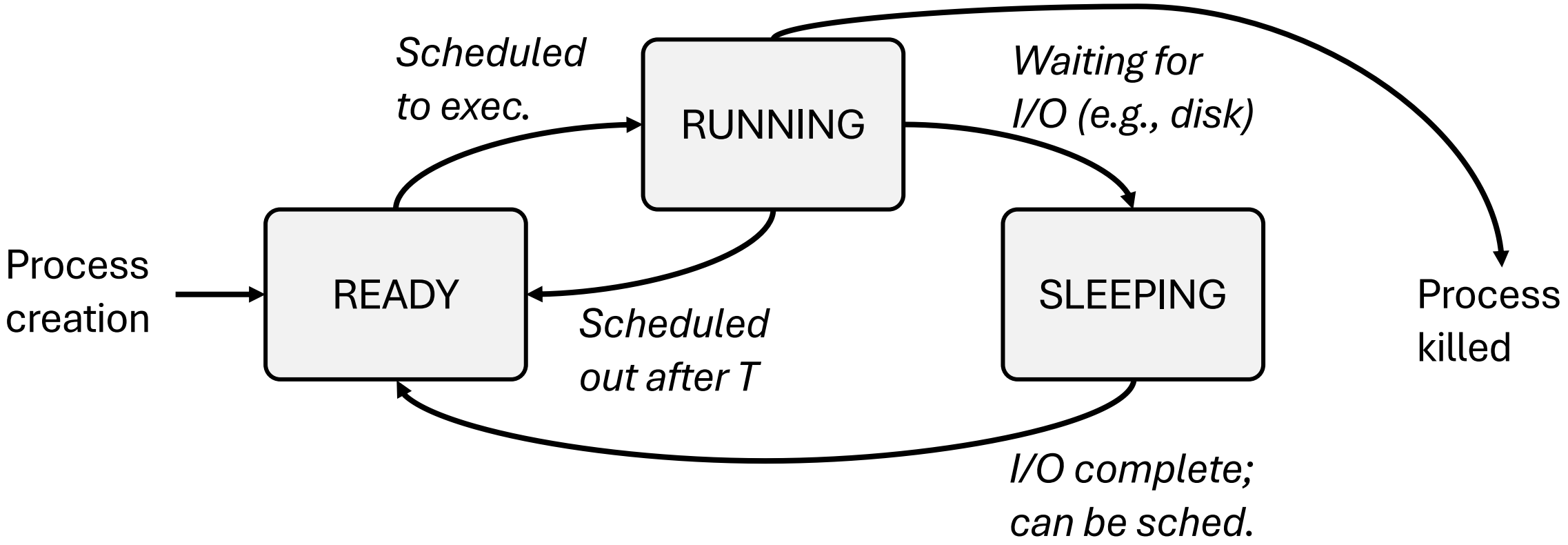
- **Timer** → a new hardware device that is provided to the OS
  - Very basic functionality; hence, it can be cheaply designed
- The OS can program it to send interrupts at fixed “timer” intervals (e.g., 1 millisecond, 1 micro-second, etc.)
- The intervals can be dynamically adjusted during runtime
  - E.g., if the process is high priority, it can get larger time slices

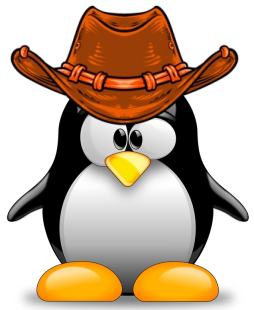
Preemptive scheduling in general-purpose OSs

- OSs programs a periodic ‘timer’ interrupt that is fired and forces a switch to the OS’ context
- OS at the interrupt decides which process should then be resumed from all the processes that are RUNNABLE
- **Is there any disadvantage to preemptive scheduling?**
  - Requires special ‘timer’ hardware
  - Sometimes spurious interrupts even when system is doing useful work

During scheduling, process can have different **states**

Three main states of a process during its execution





Questions? Otherwise see you next time!