Operating Systems with Linux Practice

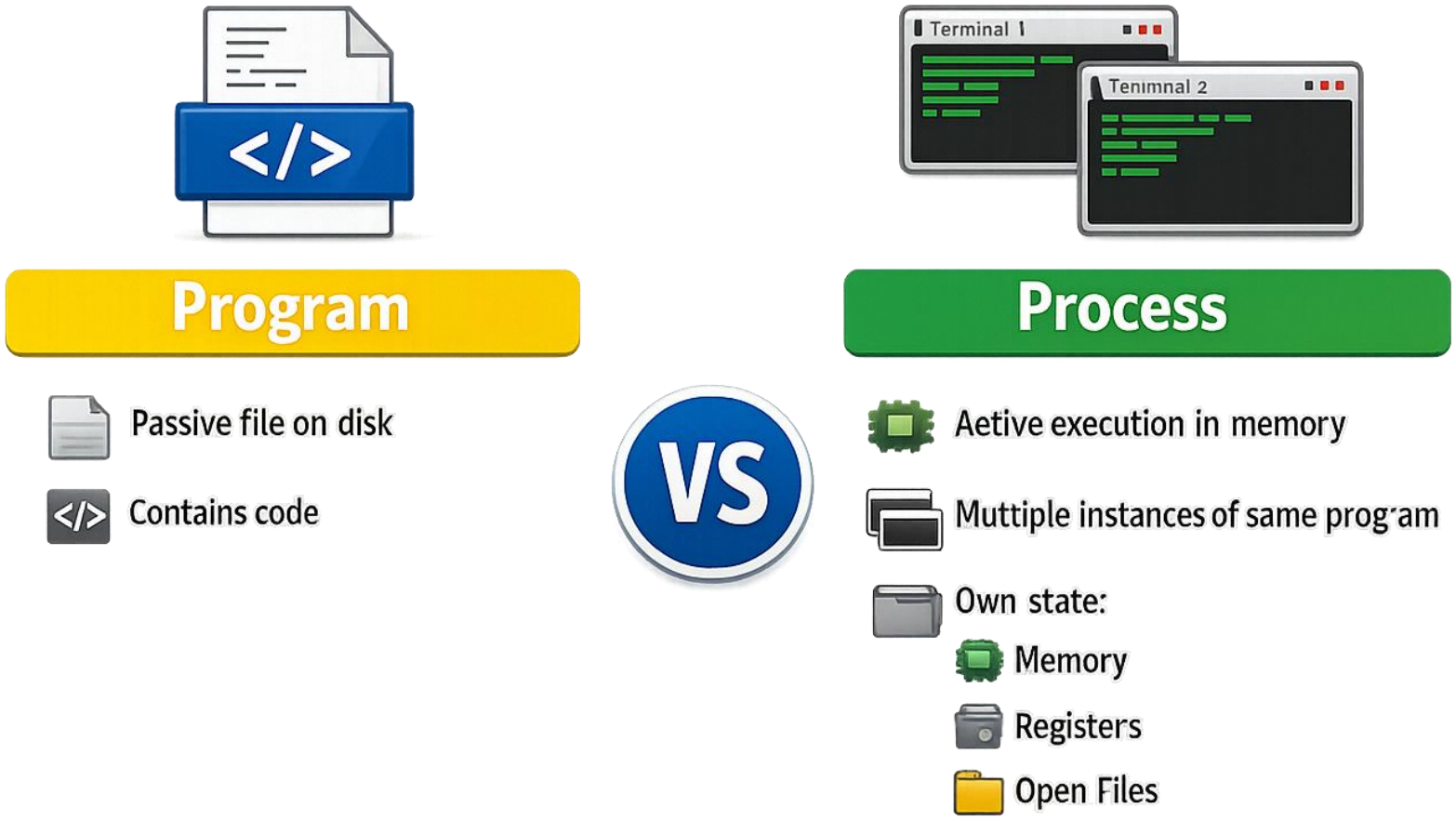
03 – Processes & Basic CPU Scheduling

Overview

This chapter focuses on processes and an introduction to CPU scheduling.

- Understand the concept of a **process**
- Learn common process **states** and **transitions**
- See what a Process Control Block (**PCB**) stores
- Get a first look at CPU **scheduling** goals and simple policies

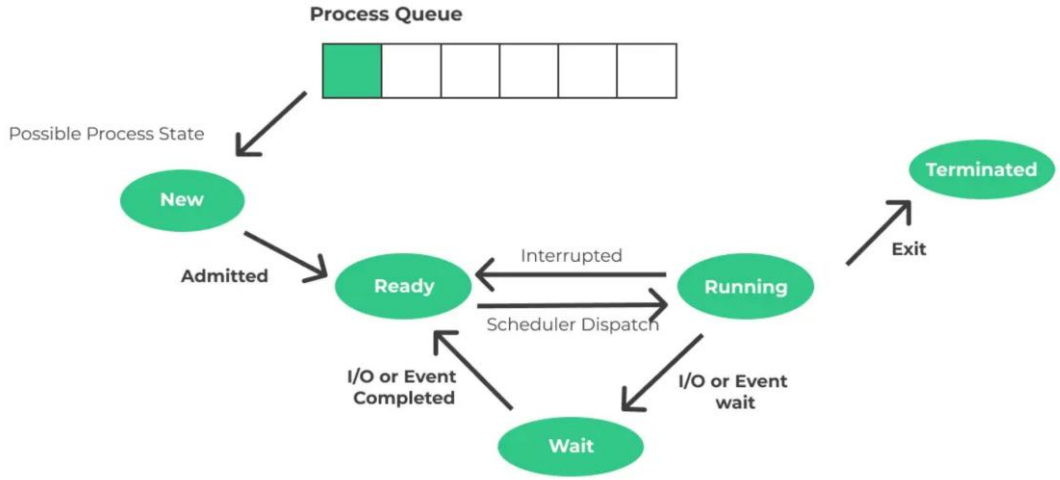
Program vs Process



Process States & Lifecycle

Typical process states in an OS:

- **New** : process is being created
- **Ready** : waiting to be assigned to the CPU
- **Running** : currently executing on the CPU
- **Waiting** (Blocked) : waiting for some event (e.g., I/O completion)
- **Terminated** : finished execution



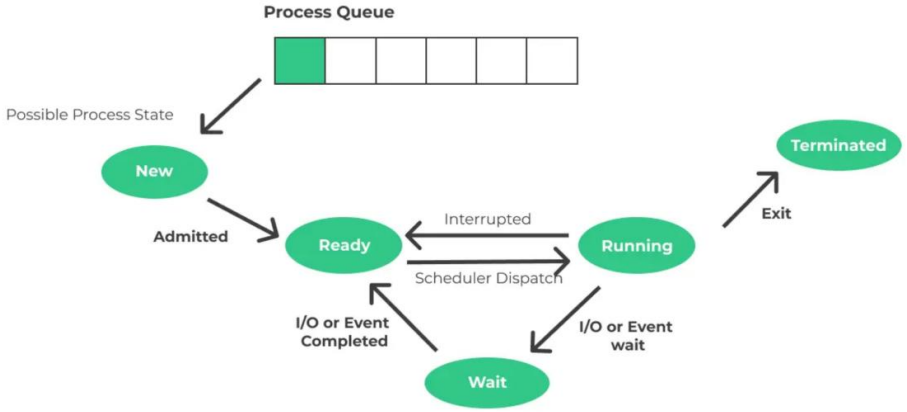
Activity 1: Process State Simulation (“State Relay”)

Purpose: Make state transitions concrete and visual

Each group uses this table and simulates events step-by-step

Process set:

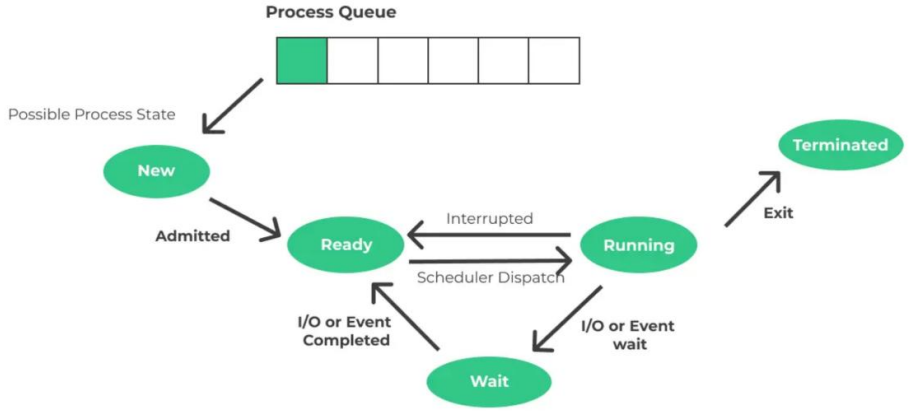
1. P1 (Running “Google Chrome”)
2. P2 (Running “Visual Studio Code”)
3. P3 (Running “Terminal”)



Activity 1: Example of event sequence

Initial state at time 0:

- P1 = Ready
- P2 = Ready
- P3 = Ready
- CPU is idle

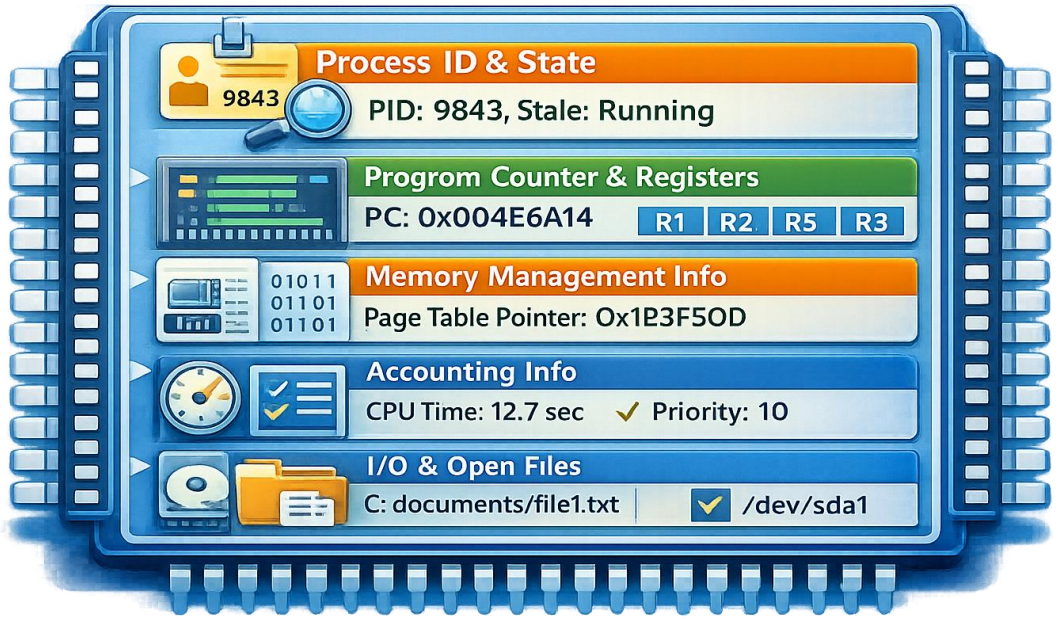


Step	Event	After event			
		Running	Ready Queue	Waiting (I/O)	Terminated
0	Initial state at time 0	-	P1, P2, P3	-	-
1	…				
2	…				

Process Control Block (PCB)

OS stores process information in a Process Control Block:

- Process ID (**PID**) and state
- **Program counter** and **CPU registers**
- **Memory management** information (e.g., page table pointer)
- **Accounting information** (CPU time used, priority)
- **I/O status** and open files



Activity 2: “PCB Definitions”

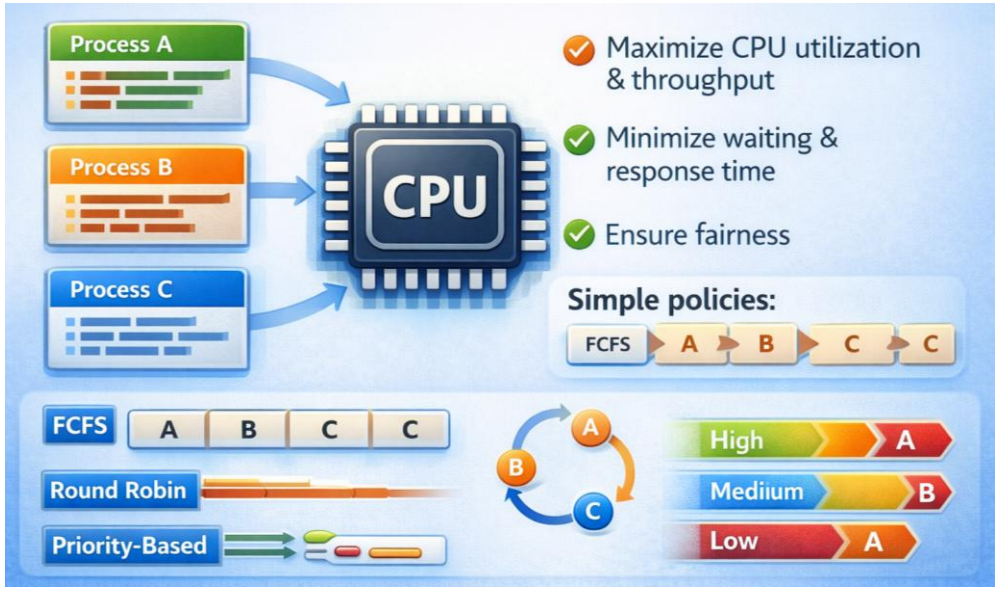
Purpose: Reinforce what the OS tracks for each process using a PCB.

Match PCB fields to meanings:

CPU Scheduling (Intro)

Basic idea of CPU scheduling:

- When **multiple** processes are ready, the **OS must choose** which runs next
- **Goals:**
  - Maximize CPU utilization and throughput, minimize waiting and response time, ensure fairness
- **Simple policies** (concept only for now):
  - FCFS (First come, first served)
  - Round Robin
  - Priority-based scheduling



Summary

Key takeaways from this:

- A process is a **running** instance of a program with **its own state**
- The **OS tracks** each process through a **PCB** and **states**
- **Process states** explain how processes move between **running**, **waiting**, and **ready**
- CPU **scheduling policies** determine which process uses the CPU and when

Linux (Continue)

Users, Groups, Permissions & Ownership

Overview & Learning Outcomes in Linux

This week Linux course, you will:

- Understand Linux user and group concepts (**UID**, **GID**)
- Read file permissions from "**`ls -l`**"
- Change permissions using "**`chmod`**" (symbolic & numeric)
- Change ownership with "**`chown`**" and "**`chgrp`**" (demo)

Design simple access rules using users, groups and modes

Users & Groups in Linux

- Every login account is a user with a numeric "**UID**"
- Users are members of one or more groups (**GID**s)
- **System users:**
  - Used by **services** (e.g., www-data, mail, apache)
- **Regular users:**
  - Used by **people** (e.g., group01, idtb110001, student1, cammob, admin)
- **Configuration files:**
  - `/etc/passwd` – user accounts
  - `/etc/group` – group definitions

Checking Your Identity in Linux

- `whoami` – Show current username
- `id` – Show UID, GIDs and groups
- `groups` – List groups you belong to

Example:

- `whoami` → `idtb110001`
- `id` →
  - `uid=1000(student)`
  - `gid=1000(student)`
  - `groups=1000(student)`
  - `27(sudo)`

Permission Bits: `r`, `w`, `x`

Three basic permissions:

- **`r`** – read
- **`w`** – write (modify, delete)
- **`x`** – execute (run file, enter directory)

Applied to three classes:

- User (**`u`**) – user/owner of the file
- Group (**`g`**) – members of the file's group
- Others (**`o`**) – others or everyone else

Reading Permissions with "**`ls -l`**"

Example output:

- **`-rwxr-x---  1 idtb110001 group01 2026 Jan 22 10:00 hello.txt`**

Breakdown:

- **`-`** → regular file (if "**`d`**" → directory)
- **`rwx`** → user (u) permissions
- **`r-x`** → group (g) permissions
- **`---`** → others (o) permissions

Ownership: **`idtb110001`**:`group01` (**`user`**:`group`)

Numeric (Octal) Permissions

Each permission bit has a numeric value:

- `r = 4, w = 2, x = 1`

Add them for each class:

- `rwx = 4+2+1 = 7`
- `rw- = 4+2+0 = 6`
- `r-x = 4+0+1 = 5`
- `--- = 0+0+0 = 0`

Common modes:

- `755` → `rwx r-x r-x`
- `700` → `rwx --- ---`
- `644` → `rw- r-- r--`

Changing Permissions with "**chmod**"

Symbolic mode:

- `chmod **u+x** hello.txt` → `(add execute for user)`
- `chmod **g-w** hello.txt` → `(remove write for group)`
- `chmod **o-r** hello.txt` → `(remove read for others)`

Numeric mode:

- `chmod **755** hello.txt` → `(rwx r-x r-x)`
- `chmod **600** hello.txt` → `(rw- --- ---)`

Ownership: **`chown`** & **`chgrp`** (Demo)

Owner and group of a file determine which '**`u`**' and '**`g`**' apply:

- `chown` **`idtb110001`** `hello.txt` →(change owner)
- `chgrp` **`group11`** `hello.txt` →(change group)
- `chown` **`idtb110001:group11`** `hello.txt` →(change both)

⚠ Typically requires **`root`** or **`sudo`** privileges

Designing Access Scenarios

Examples:

- Private file: `600 (rw-------)`
- Shared project directory: `770 (rwxrwx---)`
- Public web folder: `755 (rwxr-xr-x)`

Questions to ask:

- Who should **read** this?
- Who should **write** or modify it?
- Who should **NOT** see it at all?