# LECTURE - 2

**Recap from Lecture 1/Process Definition:**

- A **process** is defined as a program in execution.
- Operating systems allow multiple processes to run at once.
- Process states include Ready, Running, and Blocked.

## CPU Virtualization

### What Is CPU Virtualization?

**Definition and Illusion:**

- CPU virtualization is the creation of the illusion by the OS that **each process has its own CPU**.
- The OS achieves this by switching the CPU between processes (a context switch).
- This mechanism is essential because multiple programs must share one physical CPU, yet users expect smooth, uninterrupted execution.
- The setup involves 1 physical CPU serving many virtual CPUs. The ultimate objective is that the illusion includes responsiveness, making each process feel like it owns the CPU.

### Goals

1. **Efficiency:** To minimize overhead and wasted cycles, maximizing CPU use.

2. **Security:** To protect processes and the system from faulty or malicious code. Virtualization must be secure, preventing unauthorized access.

3. **Fairness:** To avoid starvation, ensuring every job gets a chance.

### Execution Methods:

- **Direct Execution (🚫 risky):** Runs directly on the CPU without OS intervention. It is fast but unsafe because misbehaving programs cannot be stopped.

- **Limited Direct Execution (LDE) (✅ safer):** Runs on the CPU but allows the OS to intervene when necessary, using traps and timer interrupts for control.

**Limited Direct Execution Steps:**

1. The OS sets up the process and switches the CPU to User Mode.

2. The user program runs until it requires an OS service, triggering a **trap** via a system call.

3. The CPU switches to Kernel Mode, saving the previous state. The OS handles the request.

4. The OS returns to the user process via a **return-from-trap**, resuming execution in User Mode.

5. Upon process termination (`exit()`), a final trap returns control to the OS for cleanup.

# System Control Mechanisms

- CPU virtualization must be secure.
- The OS enforces control mechanisms because when user code runs, OS code is paused.

## Security Environment

- Processes run in "sandboxes".
- The OS checks and validates instructions before execution to prevent harm.
- System protection ensures User Mode cannot directly access hardware or OS memory.

## CPU Modes

- Programs normally run in **User Mode**, but they must switch into **Kernel Mode** when they require operating system services.

1. **User Mode (Sandbox, Restricted):**

   ◦ **Privilege:** This mode has restricted access.

   ◦ **Access:** Processes running here cannot directly access OS memory or devices.

◦ **Function:** Programs normally execute here and must rely on **system calls** (traps) as a bridge to communicate with the OS for sensitive tasks.

2. **Kernel Mode (Full Control, Privilege):**

◦ **Privilege:** This mode has full privileges and full control.

◦ **Access:** Processes in Kernel Mode can access all memory and have direct hardware/device control.

◦ **Function:** The OS uses this mode to execute sensitive tasks and handle requests from User Mode programs.

- System protection relies on this separation, ensuring that user code running in User Mode cannot directly access hardware or OS memory.

## System Calls (Syscalls)

- Syscalls are the mandatory bridge between the user program and the OS. Example: `open("myfile", "rw")`.
- Access to sensitive resources must be via validated system calls.
- **Vuln erabilities:** Syscalls are exploitable through crafted inputs. Risks include data leakage (accessing other users' data) and parameter bugs (e.g., buffer overflow). Syscall/trap handlers are common attack targets.

## OS Enforcement and Scheduling Methods

- The OS must enforce control to ensure fairness and prevent any single process from monopolizing the CPU. The OS ensures CPU sharing using traps and timer interrupts.

**Scheduling Methods:**

1. **Cooperative Scheduling (Voluntary):** The running process yields control back to the OS, usually via system calls. **Risk:** If a process enters an infinite loop, it can freeze the system.

2. **Pre-emptive Scheduling (Forced):** The OS reclaims the CPU forcibly using **timer interrupts** after a fixed time slice. This method ensures no single process dominates and guarantees that the OS always regains control, making CPU sharing safe and fair.

**Timer-Based Process Switching (How Pre-emption Works):**

1. The OS sets a timer when a program starts running.

2. The program executes until the timer expires.

3. The hardware sends a signal called an interrupt to the OS.

4. The OS pauses the current program and saves its progress.

5. The OS then loads the next program to run.

V. CPU Scheduling: Basics and Metrics

# CPU Scheduling Basics:

- Scheduling is the OS policy that determines which process receives the CPU next.
- The choice of algorithm depends on the context (desktop, server, real-time), known information (if job length is known), and the goals (speed, fairness, responsiveness).
- **Scheduling Assumptions** often include defined arrival time (when a process starts) and defined CPU burst time (how long it needs the CPU).

# Scheduling Metrics:

- Scheduling involves balancing efficiency, responsiveness, and fairness. Note that improving one often negatively impacts another.

1. **Turnaround Time (TAT) (Efficiency):**
    - TAT (Total Time) = Completion Time – Arrival Time.
    - The goal is to minimize the average TAT across jobs. Lower TAT is better.
2. **Response Time (RT) (Responsiveness):**
    - RT (Time to first run) = Start Time – Arrival Time.
    - This metric is critical for interactive tasks.
    - The goal is to minimize the average RT.
3. **Fairness:**
    - There is no single obvious metric.
    - Fairness is achieved by avoiding **starvation** (ensuring no job waits too long).

## Scheduling Algorithms and Trade-Offs

The sources consider three simple scheduling algorithms:

1. **FIFO (First-In, First-Out):**
    - **Mechanism:** Jobs are executed strictly in the order they arrive, regardless of their required duration.
    - **Notes:** Simple.
    - **Drawback:** Short jobs may starve if a long job arrives first. This leads to the **Convoy Problem**, resulting in poor Average TAT. *Example: FIFO resulted in 103.33s average TAT in a scenario where STCF achieved 50s*.
2. **STCF (Shortest Time to Completion First):**
    - **Mechanism:** Always runs the job with the shortest remaining CPU time. Pre-emption is allowed.
    - **Benefits:** It is the optimal algorithm for minimizing average turnaround time (TAT). Short jobs do not wait behind long ones.
    - **Drawbacks:** Requires knowing or estimating burst times. Long jobs may starve. It can yield high average Response Time, leading to poor responsiveness.
3. **RR (Round-Robin):**
    - **Mechanism:** Each job receives a fixed time slice (called a quantum) to run. If the job does not finish within the slice, it is pre-empted and moved to the back of the queue.
    - **Benefits:** It is fair and highly responsive (Very low Average RT). It is ideal for interactive systems.
    - **Drawbacks:** It results in a higher average turnaround time compared to STCF.

**Scheduling Trade-Offs Summary:**

- There is no single "best" algorithm; the choice depends on priorities.
- Optimizing for **Efficiency (STCF)** means good TAT, but risks long jobs starving (poor fairness).
- Optimizing for **Fairness/Responsiveness (RR)** means good response time, but results in poor turnaround time.
- Real-world schedulers (like MLFQ, CFS) use advanced strategies to balance fairness and efficiency, often without knowing the exact job lengths