Lecture-3

CPU Scheduling Concepts

The primary goal of CPU scheduling is to virtualize the CPU by balancing **Efficiency** (turnaround time), **Responsiveness** (response time), and **Fairness**.

Prior Algorithms Recap:

- **FIFO (First-In, First-Out):** Simple, but can lead to the convoy effect.
- **STCF (Shortest Time to Completion First):** Provides optimal average turnaround time but poor response time.
- **Round-Robin (RR):** Good for response time but poor for turnaround time.

Multi-level Feedback Queue (MLFQ):

- MLFQ was developed by Fernando Corbató in 1962 and is used in Unix, Windows NT, and early Linux systems. Corbató won the Turing Award for this work. MLFQ is a blend of STCF (efficiency) and RR (responsiveness), designed to balance both.
- **Job Categorization:** MLFQ automatically learns job behavior over time and adjusts priorities.
  - **CPU-bound jobs:** Have long processing times and are de-prioritized to optimize turnaround (e.g., Key generation).
  - **I/O-bound (interactive) jobs:** Have short CPU bursts and frequent I/O waits; they are prioritized for responsiveness (e.g., Search for pattern in files).
- **MLFQ Rules:**

1. Always schedule next a job from the **highest priority queue**.

2. All jobs in the same queue are scheduled **round-robin**.

3. When a job arrives, place it in the **highest priority queue**.

4. Once a job uses up its CPU quantum (time slice) in the current queue, it **moves down one queue**. (Rules 1–3 prioritize I/O bound jobs; Rule 4 prevents monopolization by CPU-bound jobs).

5. **Priority Boost:** After a set time S, all jobs move back to the top queue. This prevents starvation of long-running jobs and helps jobs that change behavior.

Process Management

A **process** is a running instance of a program and processes are the building blocks of multitasking.

- **Process Identification:** Every process has a unique **PID** (Process ID), a **PPID** (Parent Process ID), a State (e.g., running, waiting), and assigned resources.

Key System Calls

◦ **os.fork():** Creates a new process by duplicating the current process into a parent and a child. It returns `0` in the child process and the child's PID in the parent process. The parent and child share the same code but have separate memory spaces.

◦ **os.execv() (or execvp()):** Replaces the current process with a new program; any code following the `exec()` call in the original process is never run.

◦ **os.wait() or os.waitpid():** Used by the parent process to pause and wait for the child process to finish, which is necessary for synchronization.

◦ **Avoiding "Zombies":** If a child finishes and the parent never calls `wait()`, the child becomes a **zombie** (a dead process occupying a process table entry). Parents must use `wait()` or `waitpid()` to "reap" the child.

◦ **os.kill():** Sends signals to a process (specified by PID). Common signals include `SIGTERM` (graceful termination), `SIGKILL` (force kill), `SIGSTOP` (pause), and `SIGCONT` (resume).

◦ **Exiting:** A process can terminate itself. `sys.exit()` is used for normal Python program termination (allows cleanup), while `os._exit()` exits immediately without cleanup (used in child processes after `fork()`).

- **Concurrency:** When multiple processes run concurrently, the operating system determines which process runs when, leading to mixed and potentially variable output order.

Memory Virtualization

The CPU grants 100% access to each process (briefly) with fast switching, but RAM grants only **partial access**; each process receives a portion, never all of it. The CPU communicates with RAM via the Address, Data, and Control buses.

- **Process View vs. Reality:** Every process operates under the illusion that it owns all the memory.
- **Memory Virtualization** is the technique where the OS makes each process think it has its own private memory space, isolating processes and improving security. This is achieved through **address translation**.
- **Process Address Space:** The address space is divided into three parts:
  - **Code:** Program instructions.
  - **Stack:** Used for temporary data (grows downward, fast but small).
  - **Heap:** Used for dynamic data (grows upward, bigger but slower).
- **Memory Management Approach:** Programs request contiguous memory blocks from the OS for faster access. High-level languages abstract memory management, while low-level languages (e.g., C/C++) require manual allocation (`malloc()`, `free()`).

Virtual Memory and Paging:

- **Virtual Memory** is the illusion of large, private memory created by the OS. It extends RAM by using part of the disk as extra memory, known as **swap space**.
- **Benefits (Why VM?):**

1. **Limited RAM Solution:** Allows running more programs than physical RAM permits by swapping data to disk when needed.

2. **Security Solution:** Provides process isolation, preventing data leakage or malicious overwriting.

3. **Fragmentation Solution:** Allows programs to use contiguous logical memory even if the physical memory is scattered.

Fragmentation: Fragmentation occurs when memory is used inefficiently, wasting space and slowing performance.

- **Internal Fragmentation:** Allocated memory block is not fully used.
- **External Fragmentation:** Free memory exists but is scattered in small, unusable chunks.
- The OS uses the **Page Table** to map virtual addresses to fragmented physical blocks, allowing the program to see one continuous block.

Paging Mechanism

- Virtual memory is divided into fixed-size **pages**, and physical memory (RAM) is divided into **frames**.
- The **Page Table** (PT) maps a virtual page number to a physical frame number. Each process has its own page table.
- The CPU translates the virtual address (used by the program) into a physical address using the page table.
- **Page Table Entries (PTEs):** Store the frame number and status bits like **Present** (is it in RAM?), **Protection** (read/write access), **Reference**, and **Dirty** (has it been modified?).
- **Speed Optimization:** Because every memory access requires a page table lookup, the OS uses the **Translation Lookaside Buffer (TLB)**—a small, fast cache for recent address translations—to speed up the process.
- **Locality of Access:** Virtual memory works efficiently because programs exhibit:
  - **Temporal Locality:** Accessing the same data again soon.
  - **Spatial Locality:** Accessing nearby data in memory (e.g., array elements).

Handling Pages Not in RAM:

- If a page is not present in RAM, a **page fault** is triggered.
- The OS must then fetch the required page from disk (swap space), which is very slow.
- If there is no free space in RAM, the OS must swap out another page, typically using replacement policies like **FIFO** or **LRU (Least Recently Used)**. If the swapped-out page was modified (dirty), it must first be saved back to disk.