Chapter 8: Scheduling: The Multi-Level Feedback Queue

This chapter introduces the **Multi-Level Feedback Queue (MLFQ)**, a sophisticated scheduling algorithm designed to achieve two conflicting goals simultaneously:

1. **Optimize Turnaround Time:** Prioritize short jobs to finish them quickly.
2. **Optimize Response Time:** Be responsive to interactive users.

The key challenge is achieving this *without* a priori knowledge of job lengths.

> **The Crux of the Problem: How to Schedule Without Perfect Knowledge?**
>
> How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?

MLFQ's core idea is to **learn from the past to predict the future**. It observes a process's behavior and adjusts its priority accordingly.

1. MLFQ: The Basic Rules

The MLFQ scheduler consists of a series of queues, each with a different priority level. Higher-priority queues are conceptually "above" lower-priority ones.

- **Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).** A job on a higher queue will always be chosen to run over a job on a lower queue.
- **Rule 2: If Priority(A) = Priority(B), A & B run in Round-Robin (RR).** Jobs on the same queue are scheduled using a simple RR policy.
-

The power of MLFQ comes from how it moves jobs between these queues.

2. Attempt #1: How to Change Priority

The first attempt at a priority-adjustment algorithm is based on how a process uses its **time slice** (or quantum).

- **Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).** The scheduler initially assumes every new job is short and interactive.
- **Rule 4a: If a job uses up its entire time slice while running, its priority is reduced (i.e., it moves down one queue).** This indicates the job is CPU-intensive and long-running.

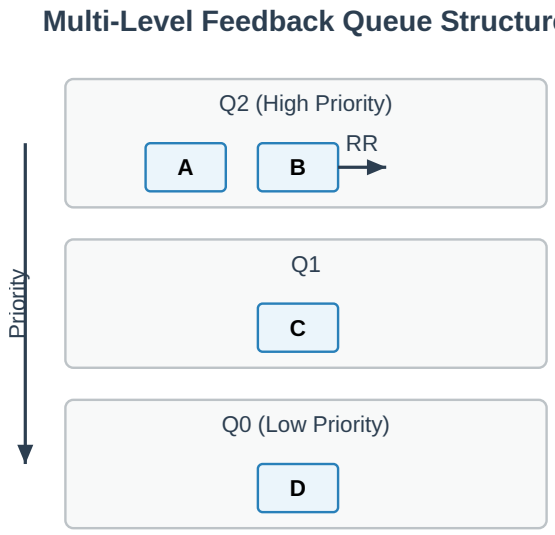

Figure 1: MLFQ Queue Structure

- **Rule 4b: If a job gives up the CPU (e.g., for I/O) before the time slice is up, it stays at the same priority level.** This rewards interactive behavior.

Example Scenarios (Attempt #1)

- **A Single Long-Running Job:** A long job enters at the highest queue, uses its full time slice, gets demoted, uses its full time slice again, gets demoted again, and so on, until it reaches the bottom-most queue.
- **An Interactive Job Arrives:** A long job (A) is running at a low priority. A new, short interactive job (B) arrives and is placed in the highest queue. B runs immediately (Rule 1), completes quickly, and exits. Then, A resumes. This successfully approximates **Shortest Job First (SJF)**.
- **A Job with I/O:** An interactive job that frequently performs I/O will repeatedly yield the CPU before its time slice expires. According to Rule 4b, it remains at a high priority, ensuring good response time.

Problems with Attempt #1

1. **Starvation:** If there are too many interactive jobs, they can monopolize the high-priority queues, causing long-running jobs in lower queues to never get CPU time.

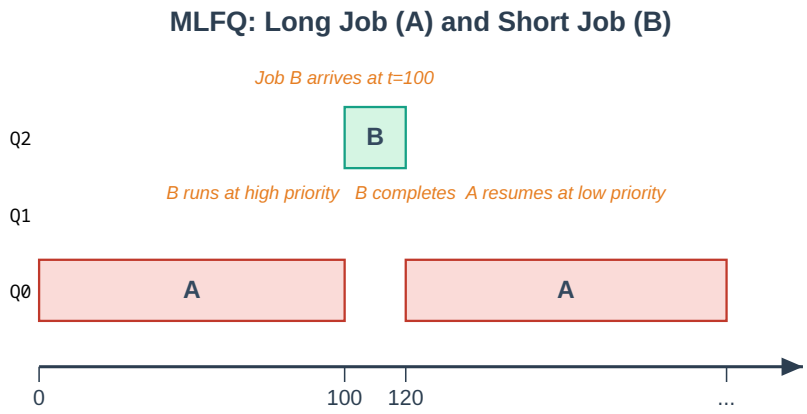

Figure 2: MLFQ Handling a Mix of Jobs

2. **Gaming the Scheduler:** A malicious or clever program can exploit Rule 4b. By running for 99% of its time slice and then issuing a trivial I/O operation, it can trick the scheduler into keeping it at a high priority, unfairly dominating the CPU.
3. **Behavior Change:** A program might be CPU-bound for a period and then become interactive. Under these rules, it would be stuck at a low priority and have poor response time.

---

3. Attempt #2: The Priority Boost

To solve the starvation problem and handle processes that change their behavior, we introduce a periodic priority boost.

- **Rule 5: After some time period S, move all the jobs in the system to the topmost queue.**

This rule ensures that even long-running, CPU-bound jobs get a chance to run, preventing starvation. It also gives processes that have become interactive a chance to be treated as such again.

- **The `S` Parameter:** Choosing the value of `S` is tricky. It's a "voo-doo constant."
  - If `S` is too long, long-running jobs can starve.
  - If `S` is too short, interactive jobs might not get a proper share of the CPU, as the long jobs are too frequently boosted.
  -

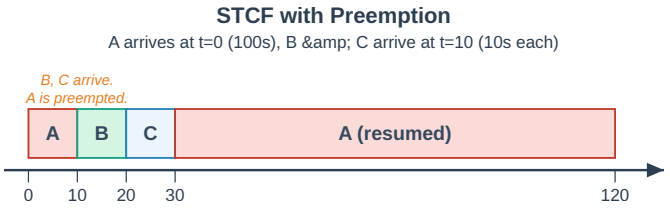

Figure 3: The Effect of a Priority Boost

4. Attempt #3: Better Accounting

To prevent gaming the scheduler, we must refine how we track CPU usage. The flaw is in treating a job that yields differently from one that uses its full time slice.

- **Revised Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).**

This rule changes the accounting. Instead of resetting a job's time slice usage every time it blocks for I/O, the scheduler tracks the *total* CPU time a job has consumed at its current priority level. Once the total time exceeds the allotment for that level, the job is demoted.

- **Result:** A process that tries to game the system by issuing frequent, short I/O calls will still have its total CPU usage accumulate. It will be demoted just like a CPU-bound job, preventing the exploit.

5. Tuning MLFQ and Other Issues

A real-world MLFQ scheduler requires careful tuning.

- **How many queues?**
- **How long is the time slice for each queue?** Often, higher-priority queues have shorter time slices (for better response time), while lower-priority queues have longer time slices (to reduce context-switching overhead for CPU-bound jobs).
- **How often to perform a priority boost?**

Real-world implementations like the Solaris Time-Sharing (TS) scheduler use configurable tables to define these parameters. Others, like the FreeBSD scheduler, use mathematical decay-usage formulas to adjust priorities dynamically.

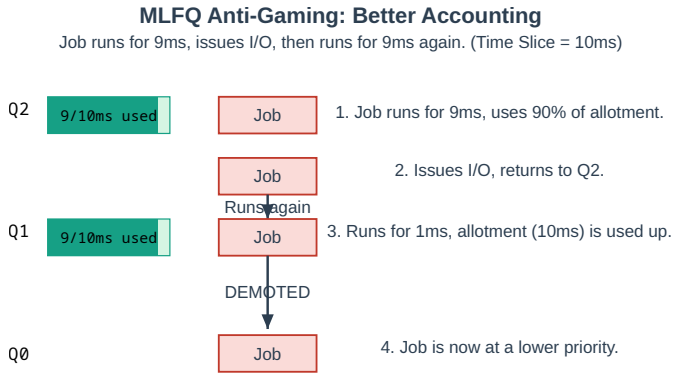

Figure 4: MLFQ with Anti-Gaming Accounting

Finally, some systems allow user advice (e.g., the `nice` command) to influence a process's base priority.

---

6. MLFQ: Final Summary

The Multi-Level Feedback Queue is an adaptive scheduling algorithm that learns from process behavior to balance the competing goals of low turnaround time and low response time without prior knowledge of job lengths.

The Complete Ruleset:

- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in Round-Robin.
- **Rule 3:** When a job enters the system, it is placed at the highest priority.
- **Rule 4 (Revised):** Once a job uses up its time allotment at a given level, its priority is reduced.
- **Rule 5:** After some time period S, move all jobs to the topmost queue (the priority boost).