Operating Systems

Scheduling:
The Multi-Level Feedback Queue

Multi-Level Feedback Queue (MLFQ)

- A Scheduler that learns from the past to predict the future.
- Objective:
  - Optimize **turnaround time** → Run shorter jobs first
  - Minimize **response time** without *a priori knowledge of job length*.
  - The multi-level feedback queue is an excellent example of a system that learns from the past to predict the future.

MLFQ: Basic Rules

- MLFQ has a number of distinct **queues**.
  - Each queues is assigned a different priority level.

- A job that is ready to run is on a single queue.
  - A job **on a higher queue** is chosen to run.
  - Use round-robin scheduling among jobs in the same queue

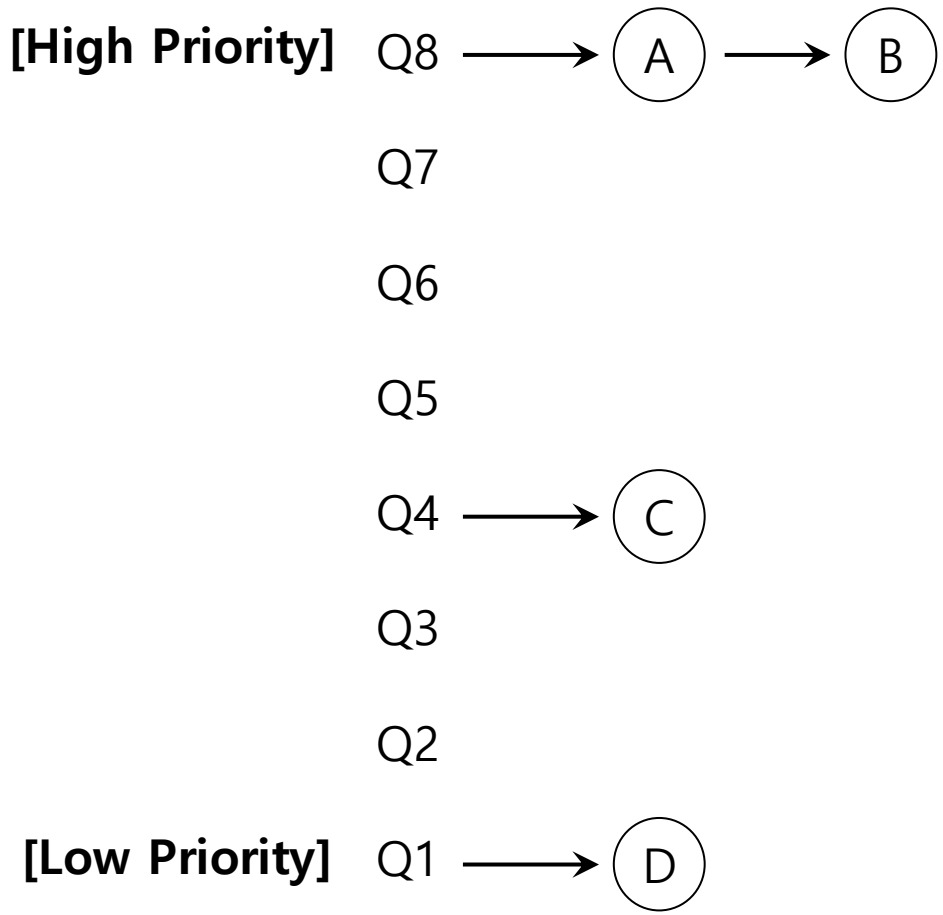
Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
Rule 2: If Priority(A) = Priority(B), A & B run in RR.

MLFQ: how the scheduler sets priorities?

- MLFQ varies the priority of a job based on its observed behavior.
- Example:
  - A job repeatedly relinquishes the CPU while waiting IOs → Keep its priority high
  - A job uses the CPU intensively for long periods of time → Reduce its priority.

- MLFQ will try to *learn* about processes as they run, and thus use the history of the job to *predict* its future behavior.

MLFQ Example



The scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system

MLFQ: How to Change Priority

- we must keep in mind our workload:
  - a mix of interactive jobs that are short-running (and may frequently relinquish the CPU), and some longer-running "CPU-bound" jobs that need a lot of CPU time but where response time isn't important

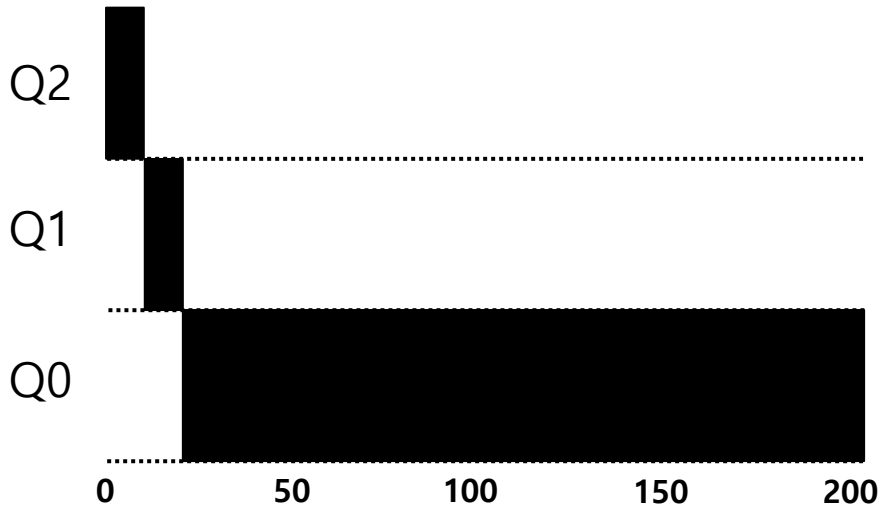
MLFQ: How to Change Priority

- MLFQ priority adjustment algorithm:
  - **Rule 3**: When a job enters the system, it is placed at the highest priority
  - **Rule 4a**: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down on queue).
  - **Rule 4b**: If a job gives up the CPU before the time slice is up, it stays at the same priority level

In this manner, MLFQ approximates SJF

Example 1: A Single Long-Running Job

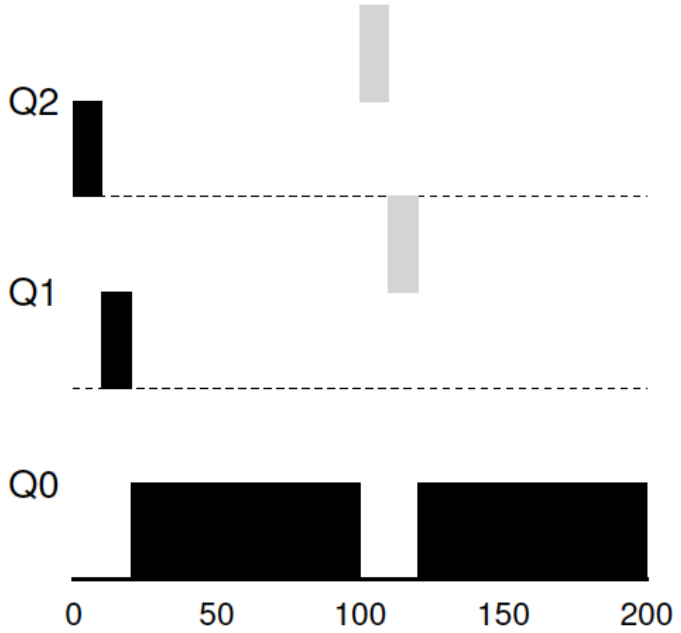
- A three-queue scheduler with time slice 10ms



Example 2: Along Came a Short Job

- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: A short-running interactive job (20ms runtime)
  - A has been running for some time, and then B arrives at time T=100.
  - In this manner, MLFQ **approximates** SJF

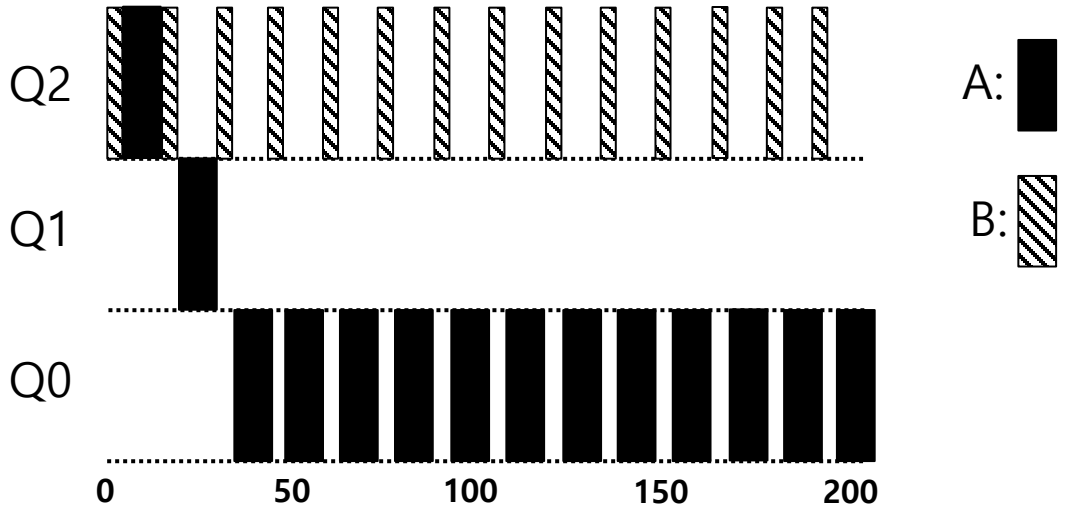
because it doesn't ***know*** whether a job will be a short job or a long-running job, it first ***assumes*** it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running process



Along Came An Interactive Job (msec)

Example 3: What About I/O?

- Assumption:
  - **Job A**: A long-running CPU-intensive job
  - **Job B**: An interactive job that need the CPU only for 1ms before performing an I/O



A Mixed I/O-intensive and CPU-intensive Workload (msec)

The MLFQ approach keeps an interactive job at the highest priority

Problems with the Basic MLFQ

- Starvation
  - If there are "too many" interactive jobs in the system.
  - Lon-running jobs will never receive any CPU time.

- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  - First, processes are guaranteed not to starve
  - Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

The Priority Boost

- Example:
  - A long-running job(A) with two short-running interactive job(B, C)