Chapter 8 Scheduling: The Multi-Level Feedback Queue

1. Optimize turnaround time: run short jobs first
   - Don’t know how long each job is.
2. Minimize the response time: RR
   - Make system feel responsive to the user sitting at the screen
   - RR is terrible for turnaround time

   Q: How can build a scheduler that achieve these goals?

   Q: How can we design a scheduler that both minimize response time for interactive jobs while also minimizing turnaround time without a priori knowledge of job length?

Idea: make the scheduler learn from running jobs to improve its scheduling.

- learn from the past to predict the future!

8.1 MLFQ: Basic Rules

MLFQ: Several **queues**, each with different **priority level**.

At any given time, a ready-to-run job is only in a single queue.

MLFQ uses priorities to decide which job gets to run.

Rule 1: If Priority(A) > Priority(B), A runs
Rule 2: If Priority(A) = Priority(B), A&B run in Round Robin…

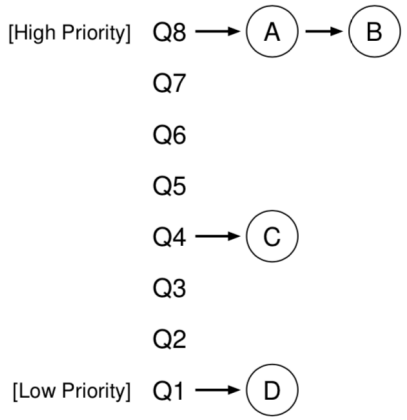

Figure 8.1: **MLFQ Example**

Fig 8.1. would be bad schedule for C and D if A and B always have need for CPU.

Therefore:

MLFQ varies the priority of a job based on its observed behavior.

Example:

- If a job repeatedly relinquishes the CPU waiting for input from keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave.
- If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority.

MLFQ learn about the processes as they run — use history to predict its future behavior.

8.2 Attempt #1: How to Change Priority

Workload: a mix of

- Interactive (short-running jobs — may frequently relinquish the CPU)
- Long-running CPU-bound jobs (may need a lot of CPU time, but response time is not important)

Rule 3: When a job arrive, it is placed at the highest priority
Rule 4a: If a job uses up an entire time slice while running, the jobs priority is reduced.
(moved one queue down)
Rule 4b: If a job gives up the CPU before the time slice is up, job keeps the same priority.

Example 1: Single long-running job (in 3-queue scheduler) Fig. 8.2.

Job enters in Q2, after one time-slice t = 10 ms, moves to Q1, and after another 10 ms, is moved to Q0, where it stays until it is finished.

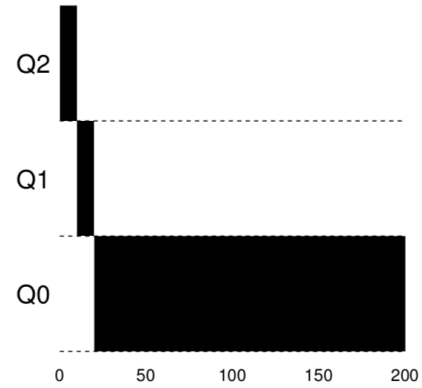

Figure 8.2: **Long-running Job Over Time**

Example 2: Along came a short job. Fig. 8.3.

Job A (long-running, CPU-intensive) arrive at t = 0.
Job B (short and interactive) arrives at t = 100 ms.

B finishes after 20 ms.
Job B never reaches Q0 but gets to run in both Q2 and Q1.
Job A resumes running after Job B finishes.

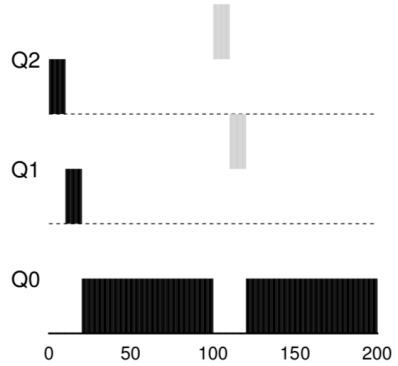

Figure 8.3: **Along Came An Interactive Job**

Example 3: What about I/O? Fig. 8.4.

If a job does a lot of IO, we don't want to penalize the job —
keep it at the same level.

Interactive job: only use the CPU for 1 ms before performing IO.
Letting the long-running job A run while it waits for IO.

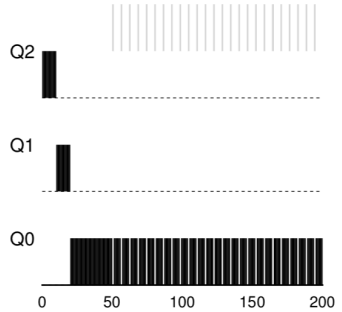

Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

MLFQ achieves its goal of running interactive jobs quickly !

Problem with our current MLFQ

Q: With this implementation of MLFQ, what can go wrong?

Starvation: If there are too many interactive jobs,
they will combine to consume all CPU-time,
preventing long-running jobs from getting any CPU-time (they starve).

Also want to make progress for these jobs too!

Game the scheduler: Trick the scheduler to give your job more than its fair share of the resource.

For example: the above MLFQ scheduler is susceptible to the following "attack":

- before the time-slice is over, e.g., after running for 99 % of the time, issue an IO operation (to some unimportant file) and thus relinquish the CPU
- Remain in the high-priority queue and get to run more often.

This can allow a job to monopolize the CPU.

Change of behavior: A job can transition from being CPU-bound to interactivity.

With our current MLFQ, such a job would be out of luck…

8.3 Attempt #2: The Priority Boost

Q: How can we ensure that CPU-bound (long-running) jobs also get some love?

Simple idea: periodically boost the priority level of all jobs.

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

In top queue, a job runs in RR-fashion with interactive jobs.

Example Fig 8.5:

Left: Job A (long-running) is starved by the two interactive jobs in Q2.

Right: With priority boost also job A gets to run for 10 ms every 50 ms.

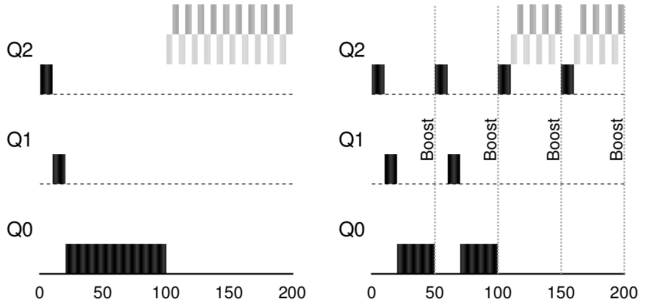

Figure 8.5: **Without (Left) and With (Right) Priority Boost**

Q: What should the time period S be set to?

Requires Black magic! To set them correctly!

Too high: long-running jobs could starve

Too low: interactive jobs may not get a proper share of the CPU.

8.4 Attempt #3: Better Accounting

Q: How to prevent gaming of our scheduler?

Rules 4a and 4b allow the job itself to control its scheduling — by relinquishing the CPU early it can stay prioritized.

Solution is better accounting of overall CPU usage.

> **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).

Example Fig. 8.6:

Left: Job B (gray) games the system and gets unfair share of CPU.

Right: Job B (gray) now only gets its fair share — since it too moves down to the lower queue.

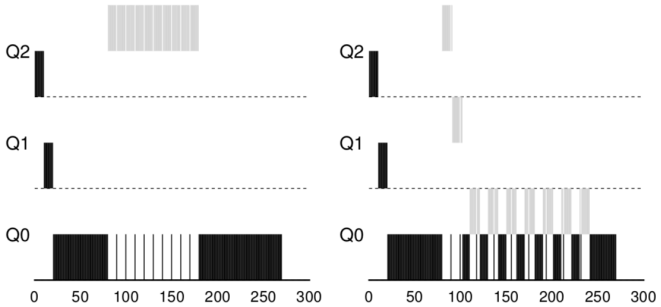

Figure 8.6: **Without (Left) and With (Right) Gaming Tolerance**

8.5 Tuning MLFQ and Other Issues

Q: How parameterize such a scheduler?

- Q1: How many queues?
- Q2: Length of the time-slice for each queue?
- Q3: How often to boost priority (to avoid starvation and account for changes in behavior)?

No easy answers!

Depends on the workload expected for the system!

Example Fig 8.7:

Q2 (ts = 10 ms)

Q1 (ts = 20 ms)

Q0 (ts = 40 ms)

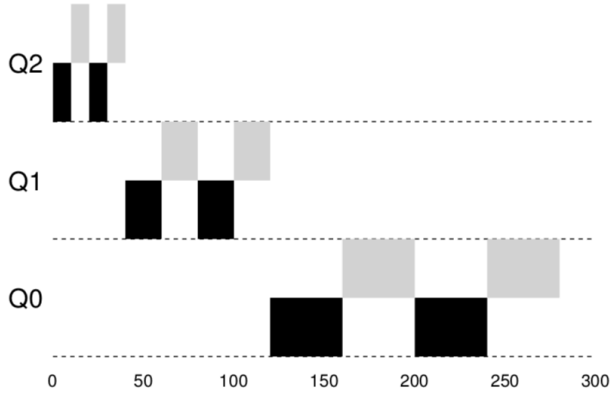

Figure 8.7: **Lower Priority, Longer Quanta**

8.6 Multi-Level Feedback Queue Summary

MLFQ achieve the best of both worlds:

It can deliver excellent overall performance (similar to SJF/STCF) for short-running interactive jobs, and it is fair and makes progress for long-running CPU-intensive workloads.

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)
Rule 2: If Priority(A) = Priority(B), A & B run in RR
Rule 3: When a job arrive, it is placed at the highest priority queue.
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)
Rule 5: After some time period S, move all jobs in the system to the topmost queue.