【OS】Day13

【Ch8】Scheduling: The Multi-Level Feedback Queue

In this chapter, we will tackle the problem of developing one of the most well-known approaches to scheduling, known as the Multi-level Feedback Queue(MLFQ).

The fundamental problem MLFQ tries to address is two-fold:

- First, it would like to optimize turnaround time, which is done by running shorter jobs first
- Second, MLFQ would like to make a system feel responsive to interactive users and thus minimize response time

Thus, our problem: *given that we in general do not know anything about a process, how can we build a scheduler to achieve these goals? How can the scheduler learn, as the system runs, the characteristics of the jobs it is running, and thus make better scheduling decisions?*

8.1 MLFQ: Basic Rules

In our treatment, the MLFQ has a number of distinct queues, each assigned a different priority level.

At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higer priority(i.e. a job on a higher queue) is chosen to run.

Of course, more than one job may be on a given queue, and thus have the same priority. In this case, we will just use round-robin scheduling among those jobs.

Thus, we arrive at the first two basic rules for MLFQ:

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

The key to MLFQ scheduling therefore lies in *how the scheduler sets priorities*.

Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behavior.

If, for example, a job repeatedly relinquishes the CPU while waiting for input from the 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.

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

See the following figure for an example:

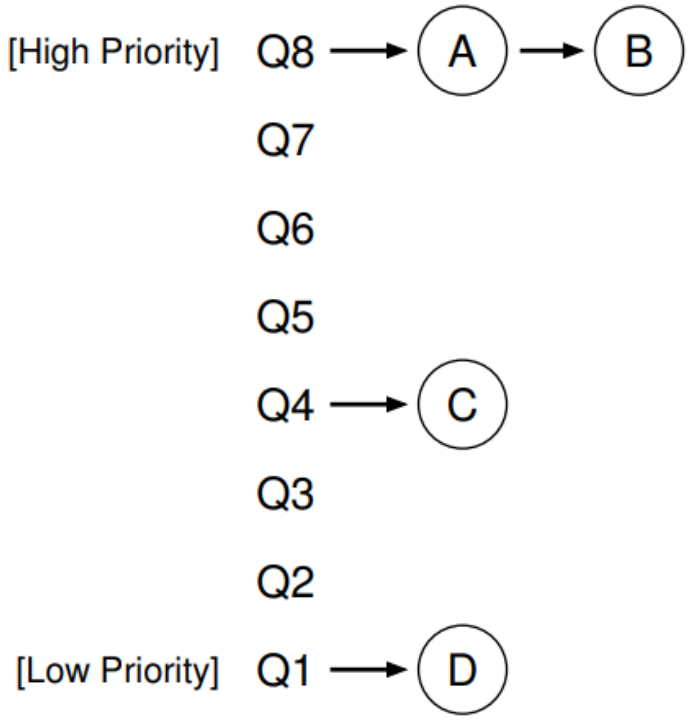

Figure 8.1: **MLFQ Example**

Process A and B has the highest priority and thus will run RR among them. Poor processes C and D have lower priorities and thus get no chance to run until A and B are finished.

8.2 Attemp#1: How to Change Priority

We now must decide how MLFQ is going to change the priority level of a job over the lifetime of a job.

We must keep in mind our workload:

- A mix of interactive jobs that are short-running(and may frequently relinquish the CPU)
- Some longer-running “CPU-bound” jobs that need a lot of CPU time but where response time isn’t important.

Here is our first attempt at a priority-adjustment algorithm:

- Rule 3: When a job enters the sytstem, it is placed at the highest priority
- Rule 4a: If a job uses up an entire time slice while running, its priority is reduced
- Rule 4b: If a job gives up the CPU before the time slice is up, it stays aet the same priority level.

Example 1: A Single Long-Running Job

The figure below shows what happens to this job over time in a three-queue scheduler

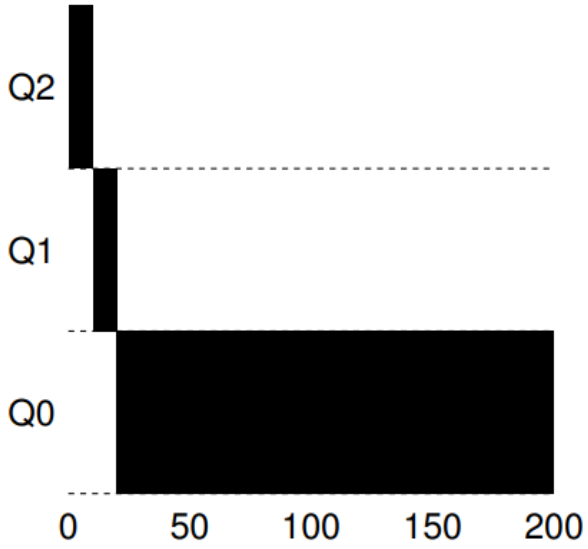

Figure 8.2: **Long-running Job Over Time**

The job enters at the highest priority queue(Q2). After a single time-slice of 10ms, the scheduler reduces the job's priority by one, and thus the job is on Q1.

After running at Q1 for a time slice, the job is finally lowered to the lowest priority in the system(Q0).

Example 2: Along Came a Short Job

In this example, we introduce another short-running interactive job B.

Assume A has been running for some time, and then B arrives. *What will happen?*

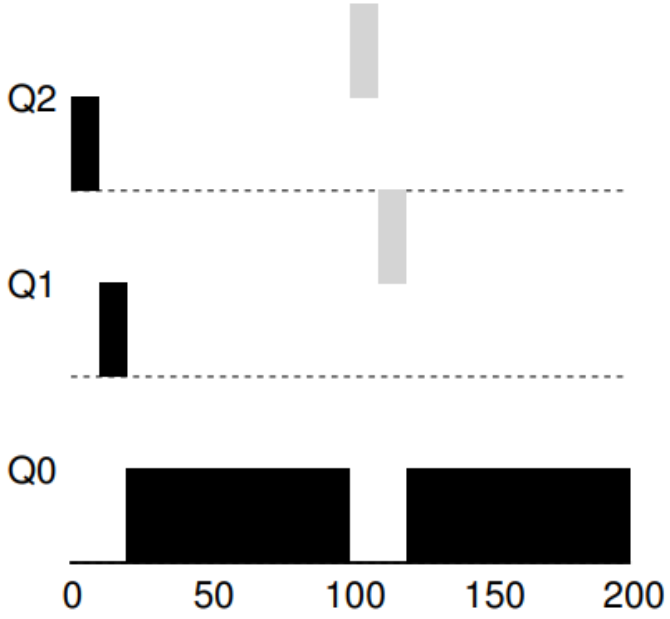

Figure 8.3: **Along Came An Interactive Job**

A(shown in black) is running along in the lowest-priority queue; B(shown in grey) arrives at Time T = 100, and thus is inserted into the highest queue; as its run-time is short(only 20ms), B completes before reaching the bottom queue, in two time slices; then A resumes running.

From this example, you can hopefully understand one of the major goals of the algorithm: 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 more batch-like process.

Example 3: What About I/O?

If an interactive job, for example, is doint a lot of I/O, it will relinquish the CPU before its time slice is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Problems With Our Current MLFQ

- First, there is the problem of starvation: if there are "too many" interactive jobs in the system, they will combine to consum all CPU time, and thus long-running jobs will never receive any CPU time(they starve).
- Second, a smart usert could rewrite their program to game the scheduler. Gaming the scheduler generally refers to the idea of donig something sneaky to trick the scheduler into giving you more than your fair share of the resource.

  The algorithm we have developed is susceptible to the following attack: before the time slice is over, issue an I/O operation and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time.
- Finally, a program may change its behavior over time; what was CPU-bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.