**Chapter 8: Scheduling: The Multi-Level Feedback Queue**

The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize **turnaround time**. Second, MLFQ would like to make a system feel responsive to interactive users, and thus minimize the **response time**.

How can we design such a system?

The multi-level feedback queue is an example of a system that learns from the past to predict the future.

**8.1 MLFQ: Basic Rules**

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 higher priority is chosen to run.

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**. For example, if a job repeatedly relinquishes the CPU while waiting for the input from the keyboard, MLFQ will keep its priority high. On the other hand, if a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. Thus, MLFQ will try to **learn** about processes as they run, and thus use the **history** of the job to predict its **future** behavior.

Picturing the queue

Diagram

Description automatically generated

The scheduler would run job A and B using RR while C and D will be run later, or never even get to run.

**8.2 Attempt #1: How To Change Priority**

We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job. We must keep in mind of our **workload**: a mix of interactive jobs that are short-running and some longer-running “CPU-bound” jobs that need a lot of CPU time but where response time isn’t important.

First attempt for priority-adjustment algorithm:

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

**Example 1: A Single Long-Running Job**

First, we will look at what happens when there has been a long running job in the system.

Chart

Description automatically generated

The job enters at highest priority Q2. Then after 10ms, the scheduler reduces the job’s priority by one to Q1. Then it continues to Q2.

Example 2: Along Came A Short Job

Chart

Description automatically generated

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

In this example, because MLFQ 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. We can see that MLFQ behaves similar to SJF.

**Example 3: What About I/O?**

Chart

Description automatically generated

An interactive job B is shown in gray that needs the CPU only for 1 ms before performing an I/O competing for the CPU with a long-running batch job A (shown in black). The MLFQ approach keeps B at the highest priority because B keeps releasing the CPU; if B is an interactive job, MLFQ further achieves its goal of running interactive jobs quickly.

**Problem with our current MLFQ**

1. If there are too many interactive jobs in the system, they will combine to consume all the CPU time. Thus, long-running jobs will never receive any CPU time (**starve**).
2. Smart user could rewrite their program to **game the scheduler**. Gaming the scheduler generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource.
3. 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

**8.3 Attempt #2: The Priority Boost**

The simple idea here is to periodically **boost** the priority of all the jobs in system:

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

Our new rule solves two problems at once. First, processes are guaranteed not to starve: by sitting in the top queue, a job will share the CPU with other high-priority jobs in a round-robin fashion, and thus eventually receive service. Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost. We can see the effect of **boosting** in the following example:

Chart

Description automatically generated

On the left figure, the long-running job gets starved once the two short jobs arrive. With boosting on the right, the long job gets to run periodically.

So, what should S be set to? John Ousterhout used to call such values in systems voo-doo constants, because they seemed to require some form of black magic to set them correctly. Unfortunately, S has that flavor. If it is set too high, long-running jobs could starve; too low, and interactive jobs may not get a proper share of the CPU.

**8.4 Attempt #3: Better Accounting**

How to prevent gaming of our scheduler?

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Once a process has used its allotment, it is demoted to the next priority queue. Whether it uses the time slice in one long burst or many small ones does not matter. Therefore, the rule 4a and 4b can be rewritten as

* **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).

A picture containing chart

Description automatically generated

Without any protection from gaming, a process can issue an I/O just before a time slice ends and thus dominate CPU time. With such protections in place, regardless of the I/O behavior of the process, it slowly moves down the queues, and thus cannot gain an unfair share of the CPU.

**8.5 Tuning MLFQ And Other Issues**

One big question is how to parameterize such a scheduler. For example, how many queues should there be? How big should the time slice be per queue? How often should priority be boosted in order to avoid starvation and account for changes in behavior?

There is no easy answers to these questions, but some experience with workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.

Most MLFQ variants allow for varying time-slice length across different queues. The high-priority queues are usually given short time slices. The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well.

**Chart

Description automatically generated with medium confidence**

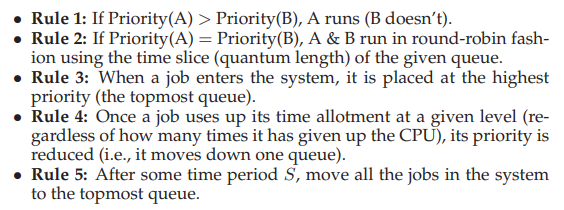
The Solaris MLFQ implementation (Time-Sharing scheduling class or TS) provides a set of tables that determine exactly how the priority of a process is altered throughout its lifetime, how long each time slice is, and how often to boost the priority of a job.

Other MLFQ schedulers don’t use a table or the exact rules described in this chapter; rather they adjust priorities using mathematical formulae.

Some schedulers reserve the highest priority levels for operating system work; thus typical user jobs can never obtain the highest levels of priority in the system. Some systems also allow some **user advice** to help set priorities; for example, by using the command-line utility **nice** you can increase or decrease the priority of a job (somewhat) and thus increase or decrease its chances of running at any given time.

**8.6 Summary:**

MLFQ rules:



MLFQ is interesting for the following reason: instead of demanding a priori knowledge of the nature of a job, it observes the execution of a job and prioritizes it accordingly.