7

Scheduling: The Multi-Level Feedback Queue

In this note, we'll tackle the problem of developing one of the most well-known approaches to scheduling, known as the **Multi-level Feedback Queue (MLFQ)**. The Multi-level Feedback Queue (MLFQ) scheduler was first developed by Corbato et al. in 1962 [1] in a system known as the Compatible Time-Sharing System (CTSS), and this work, along with later work on Multics, led the ACM to award Corbato its highest honor, the Turing Award. It has subsequently been refined throughout the years to the implementations you will encounter in modern systems.

The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize *turnaround time*, which, as we saw in the previous note, is done by running shorter jobs first; unfortunately, the OS doesn't generally know how long a job will run for, exactly the knowledge that algorithms like SJF (or STCF) require. Second, MLFQ would like to make a system feel responsive to interactive users, and thus minimize *response time*; unfortunately, algorithms like Round Robin reduce response time but are terrible for turnaround time. Thus, our problem:

THE CRUX: SCHEDULING WITHOUT PERFECT KNOWLEDGE
How can we design a scheduler that both minimizes response time for interactive jobs while also minimizing turnaround time, all without any *a priori* knowledge of job length?

7.1 MLFQ: Basic Rules

To attempt to build such a scheduler, MLFQ has a number of distinct **queues**, each assigned a different **priority level**. At any given time, a job that is ready to run can only be on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e., a job on a higher queue) is the one that will 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, the key to MLFQ scheduling 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. If, instead, a job uses the CPU intensively, 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.

Thus, we arrive at the basic rules for MLFQ:

- **Rule 1:** If Priority(A) $>$ Priority(B), A will run (and B won't).
- **Rule 2:** If Priority(A) $=$ Priority(B), both A and B will be run in round-robin fashion.

If we were to put forth a picture of what the queues might look like at a given instant, we might see something like what

DESIGN TIP: LEARNING FROM HISTORY
The multi-level feedback queue is an excellent example of a system that learns from the past to predict the future. Such approaches are common in operating systems (and many other places, including the hardware). Such approaches work when jobs have phases of behavior and are thus predictable; of course, one must be careful with such techniques, as they can easily be wrong and drive a system to make worse decisions than they would have with no knowledge at all.

you can see in Figure 7.1.

In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and Job D is at the lowest level of priority. Thus, given our current knowledge of how MLFQ works, the scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system as of now.

7.2 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. To do this, we must keep in mind our workload: a mix of interactive jobs that 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. Here is our first attempt at a 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.

```
(high priority)  Q1 -> Job A -> Job B
                 Q2 ->
                 Q3 ->
                 Q4 -> Job C
                 Q5 ->
                 Q6 ->
                 Q7 ->
 (low priority)  Q8 -> Job D
```

Figure 7.1: MLFQ: An Example

To understand this better, let's look at an example. Assume that two jobs, A and B, enter the system at the same time. Assume further that A is a CPU-bound long-running job, whereas B is an interactive job that frequently relinquishes the CPU after a short burst of usage (say, to process a key stroke).

Let's analyze what would happen in this example. At first, they are both placed in the topmost high-priority queue (Rule 3), where they will round-robin share the CPU (Rule 2). When A runs, it uses up the entire time slice, and thus moves down one level (Rule 4a). Then B runs, and because it is interactive, gives up the CPU before the time slice expires. Thus, it remains at the same high priority level (Rule 4b). At this point, any time B needs to run, it will run, because it has priority (Rule 1); however, it only can retain such a priority by releasing the CPU quickly (Rule 4b), and thus A will get many opportunities to run in-between B's bursts. A, as a CPU-bound process, always uses the entire time slice, and thus slowly works its way down the queues to the bottom-most queue. Figure 7.2 presents this example.

From the example, you can see how neat the MLFQ is in how it approaches scheduling. It dynamically learns that A is CPU-bound and thus continually demotes it; similarly, it learns that B is interactive and thus keeps it at high priority. And thus you can see that MLFQ is dynamically achieving some of our main goals: jobs with short bursts of CPU are scheduled quickly and thus remain responsive, whereas long-running CPU-bound jobs move down and thus only make progress when there is no interactive job that needs service.

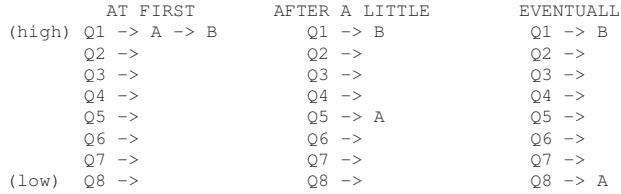

Figure 7.2: MLFQ: A and B over time

Unfortunately, while this approach works in this example, it does have a few problems. Can you think of any?

First, there is the problem of **starvation**: if there are "too many" interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time (hence the name, starvation). Clearly, we'd like to make some progress on these jobs even in this scenario.

Second, a 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 of your fair share of the resource. The algorithm we have described is susceptible to the following attack: before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of the CPU. In fact, if this is done just right (by running for 99% of the time slice, say, before relinquishing the CPU), you could take over most of the CPU cycles of the machine.

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.

7.3 Attempt #2: The Priority Boost

Let's try to change the rules and see if we can avoid the problem of starvation. What could we do in order to guarantee that CPU-bound jobs will make some progress (even if it is not much?).

The simple idea here is to periodically **boost** the priority of all the jobs in system. There are many ways to achieve this, but let's just do something simple: throw them all in the topmost queue. Thus, we add a new rule:

- **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.

Of course, the addition of the time period S leads to the obvious question: what should S be set to? John Ousterhout, a well-regarded systems researcher, 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.

7.4 Attempt #3: Better Accounting

We now have one more problem to solve: how to prevent gaming of our scheduler? The real culprit here, as you might have guessed, are Rules 4a and 4b, which let a job retain its priority level simply by relinquishing the CPU before the time slice expires. So what should we do instead?

The solution here is to perform better **accounting** of CPU time at each level of the MLFQ. Instead of forgetting how much

DESIGN TIP: AVOID VOO-DOO CONSTANTS
It is pretty clear that avoiding voo-doo constants is a good idea whenever possible. Unfortunately, as in the example above, it is often difficult. One could try to make the system learn a good value, but that too is not straightforward. The frequent result: a configuration file filled with default parameter values that a seasoned administrator can tweak when something isn't quite working correctly. As you can imagine, these are often left unmodified, and thus we are left to hope that the default values shipped with the system work well in the field.

of a time slice a process used at a given level, the scheduler should keep track; 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. We thus rewrite Rules 4a and 4b to the following single rule:

- **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).

For example, if the time slice at a given level is 100 ms, a job that only runs for 10 ms before giving up the CPU will stay in the same queue for 10 usages of CPU time. After the tenth, the job will be demoted.

7.5 One More Thing: Time-Slice Length

MLFQ also allows for varying time-slice length across the different queues of the system. The high-priority queues are usually given short time slices; they are comprised of interactive jobs, after all, and thus quickly alternating between them makes sense (e.g., 10 or 20 milliseconds). The low-priority queues, in contrast, contain long-running CPU-bound jobs. Thus, there is

no need to switch between them frequently, and longer time slices make sense (e.g., hundreds of ms).

```
(high priority, shorter time slice)  Q1
                                     Q2
                                     ...
                                     Qn-1
 (low priority, longer time slice)   Qn
```

7.6 Tuning MLFQ

A few other issues arise with MLFQ scheduling. 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? How often should priority be boosted in order to avoid starvation and account for changes in behavior?

There are no easy answers to these questions, and thus only some experience with various workloads and subsequent tuning of the scheduler will lead to a satisfactory balance. The Solaris implementation of MLFQ, known as the Time Sharing scheduling class (TS), enables such tuning through a set of tables that determine exactly how the priority of a process is altered throughout its lifetime [2].

7.7 MLFQ: Summary

We have described a scheduling approach known as the Multi-Level Feedback Queue (MLFQ). Hopefully you can now see why it is called that: it has *multiple levels* of queues, and uses *feedback* to determine the priority of a given job.

MLFQ is interesting because instead of demanding *a priori* knowledge of the nature of a job, it instead observes the execution of a job and prioritizes it accordingly. In this way, it manages to achieve the best of both worlds: it can deliver excellent overall performance similar to SJF/STCF scheduling for turnaround time, while it can also provide a responsive system for interactive jobs just like RR. For this reason, many systems,

including BSD Unix derivatives, Solaris, and Windows NT and subsequent versions use a form of MLFQ as their base scheduler.

References

[1] "An Experimental Time-Sharing System"
F. J. Corbato, M. M. Daggett, R. C. Daley
IFIPS 1962.

[2] "Multilevel Feedback Queue Scheduling in Solaris"
Andrea Arpaci-Dusseau
Available: http://pages.cs.wisc.edu/ eli/537/lectures/Solaris.ps