

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 described by Corbato et al. in 1962 [C+62] 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 (i.e., users sitting and staring at the screen, waiting for a process to finish), and thus minimize *response time*; unfortunately, algorithms like Round Robin reduce response time but are terrible for turnaround 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?

THE CRUX:

HOW TO SCHEDULE WITHOUT PERFECT KNOWLEDGE?

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

8.1 MLFQ: Basic Rules

To build such a scheduler, in this chapter we will describe the basic algorithms behind a multi-level feedback queue; although the specifics of many implemented MLFQs differ [E95], the basic approaches are all similar.

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 higher 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, 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, 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.

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

- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.

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 you can see in Figure 8.1.

In the figure, two jobs (A and B) are at the highest priority level,

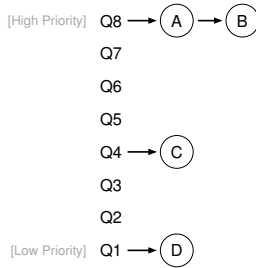


Figure 8.1: MLFQ Example

while job C is in the middle and Job D is at the lowest priority. 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.

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. To do this, 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. 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.

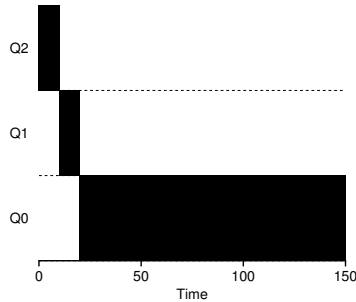


Figure 8.2: Long-running Job Over Time

Example 1: A Single Long-Running Job

To understand this better, let's look at some examples. First, we'll just look at what happens when there has been a long running job in the system for a while. Figure 8.2 shows what happens to this job over time, in a system with three queues.

As you can see in the example, the job enters at the highest priority (Q2). After a single time-slice of 10 ms, 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), where it remains. Pretty simple, no?

Example 2: Along Came A Short Job

Now let's look at a more complicated example, and hopefully see how MLFQ tries to approximate SJF. In this example, there are two jobs: A, which is a long-running CPU-intensive job, and B, which is a short-running interactive job. Assume A has been running for some time, and then B arrives. What do you think will happen? Will MLFQ approximate shortest-job first for B?

Figure 8.3 plots the results of this scenario. 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

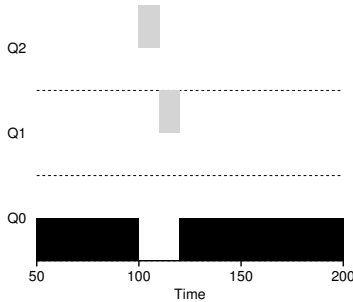


Figure 8.3: Along Came An Interactive Job

short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).

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. In this manner, MLFQ approximates SJF.

Example 3: What About I/O?

Let's now look at an example with some I/O. As Rule 4b states above, if a process gives up the processor before using up its time slice, we keep it at the same priority level. The intent of this rule is simple: if an interactive job, for example, is doing a lot of I/O (say by waiting for user input from the keyboard or mouse), 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.

Figure 8.4 shows an example of how this works, with an interactive job B (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.

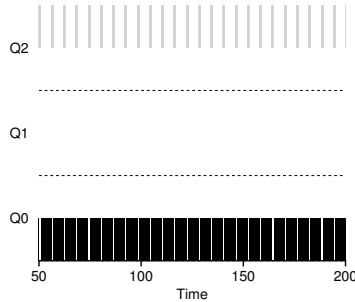


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload

Problems With Our Current MLFQ

We thus have a basic MLFQ algorithm. It seems to do a fairly good job, sharing the CPU fairly between long-running jobs, and letting short or I/O-intensive interactive jobs run quickly. Unfortunately, the approach we have developed thus far contains a few serious problems. Can you think of any? (pause and think on it a minute)

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 than 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 done just right (e.g., by running for 99% of the time slice before relinquishing the CPU), a job could get most available 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.

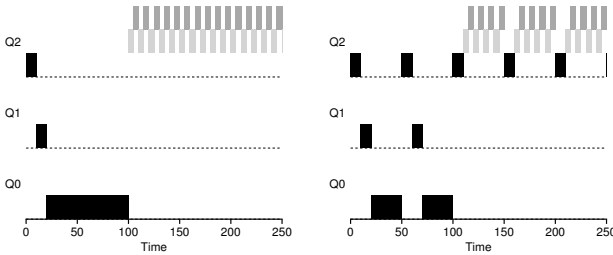


Figure 8.5: Without (Left) and With (Right) Priority Boost

8.3 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.

Let's see an example. In this scenario, we just show the behavior of a long-running job when competing for the CPU with two short-running interactive jobs. Two graphs are shown in Figure 8.5. On the left, there is no priority boost, and thus the long-running job gets starved once the two short jobs arrive; on the right, there is a priority boost every 50 ms (which is likely too small of a value, but used here for the example), and thus we at least guarantee that the long-running job will make some progress, getting boosted to the highest priority every 50 ms and thus getting to run periodically.

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. This tip brought to you by our old OS professor, John Ousterhout, and hence the other name for it: **Ousterhout's Law**.

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 [O11], 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

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 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).

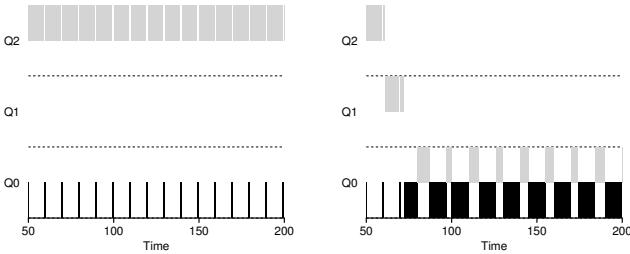


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance

Let's look once again at an example. Figure 8.6 shows what happens when a workload tries to game the scheduler with the old Rules 4a and 4b (on the left) as well the new anti-gaming Rule 4. 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 percentage of CPU time.

8.5 Tuning MLFQ and Other Issues

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 per queue? 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 workloads and subsequent tuning of the scheduler will lead to a satisfactory balance.

For example, most MLFQ variants allow 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 jobs that are CPU-bound. Thus, there is no need to switch between them frequently, and longer time slices

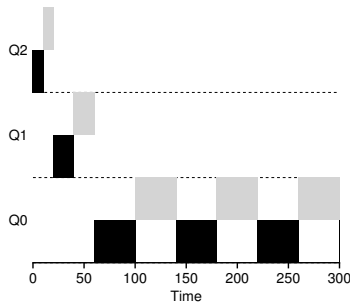


Figure 8.7: Lower Priority, Longer Quanta

make sense (e.g., hundreds of ms). Figure 8.7 shows an example in which two long-running jobs run for 10 ms at the highest queue, 20 ms at the middle queue, and 40 ms at the lowest.

The Solaris implementation of MLFQ, known as the Time Sharing scheduling class (TS), is particularly easy to configure; it 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 [AD00]; an administrator can muck with this table in order to make the scheduler behave in different ways. Default values for the table are 60 queues, with slowly increasing time-slice lengths from 20 milliseconds (highest priority) to a few hundred milliseconds (lowest), and priorities boosted around every 1 second or so.

Other MLFQ schedulers don't use a table or the exact rules described in this chapter; rather they adjust priorities using mathematical formulae. For example, the FreeBSD scheduler (version 4.3) uses a formula to calculate the current priority level of a job, basing it on how much CPU the process has used [LM+89]; in addition, usage is decayed over time, providing the desired priority boost in a different manner than described herein. See [E95] for an excellent overview of such **decay-usage** algorithms and their properties.

Finally, many schedulers have a few other features that you might encounter. For example, some schedulers reserve the highest priority levels for operating system work; thus typical user jobs can never

DESIGN TIP: USE ADVICE WHERE POSSIBLE

As the operating system rarely knows what is best for each and every process of the system, it is often useful to provide interfaces to allow users or administrators to provide some **hints** to the OS. We often call such hints **advice**, as the OS need not necessarily pay attention to it, but rather might take the advice into account in order to make a better decision. Such hints are useful in many parts of the OS, including the scheduler (e.g., with `nice`), memory manager (e.g., `madvise`), and file system (e.g., TIP [P+95]).

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 (at least, somewhat), and thus increase or decrease its chances of running at any given time. See the `nice` man page for details.

8.6 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. The refined set of rules, spread throughout the chapter, are reproduced here for your viewing pleasure:

- **Rule 1:** If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- **Rule 2:** If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost 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 the jobs in the system to the topmost queue.

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) for short-running interactive jobs, and is fair and makes progress for long-running CPU-intensive workloads. For this reason, many systems, including BSD UNIX derivatives [LM+89,B86], Solaris [M06], and Windows NT and subsequent Windows operating systems [CS97] use a form of MLFQ as their base scheduler.

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 in Computer Science, including hardware branch predictors and caching algorithms). 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.

References

- [AD00] "Multilevel Feedback Queue Scheduling in Solaris"
Andrea Arpaci-Dusseau
Available: <http://www.cs.wisc.edu/~remzi/solaris-notes.pdf>
A great short set of notes by one of the authors on the details of the Solaris scheduler. OK, we are probably biased in this description, but the notes are pretty darn good.
- [B86] "The Design of the UNIX Operating System"
M.J. Bach
Prentice-Hall, 1986
One of the classic old books on how a real UNIX operating system is built; a definite must-read for kernel hackers.
- [C+62] "An Experimental Time-Sharing System"
F. J. Corbato, M. M. Daggett, R. C. Daley
IFIPS 1962
A bit hard to read, but the source of many of the first ideas in multi-level feedback scheduling. Much of this later went into Multics, which one could argue was the most influential operating system of all time.
- [CS97] "Inside Windows NT"
Helen Custer and David A. Solomon
Microsoft Press, 1997
The NT book, if you want to learn about something other than UNIX. Of course, why would you? OK, we're kidding; you might actually work for Microsoft some day you know.
- [E95] "An Analysis of Decay-Usage Scheduling in Multiprocessors"
D.H.J. Epema
SIGMETRICS '95
A nice paper on the state of the art of scheduling back in the mid 1990s, including a good overview of the basic approach behind decay-usage schedulers.
- [LM+89] "The Design and Implementation of the 4.3BSD UNIX Operating System"
S.J. Leffler, M.K. McKusick, M.J. Karels, J.S. Quarterman
Addison-Wesley, 1989
Another OS classic, written by four of the main people behind BSD. The later versions of this book, while more up to date, don't quite match the beauty of this one.
- [M06] "Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture"
Richard McDougall
Prentice-Hall, 2006
A good book about Solaris and how it works.

[O11] "John Ousterhout's Home Page"

John Ousterhout

Available: <http://www.stanford.edu/~ouster/>

The home page of the famous Professor Ousterhout. The two co-authors of this book had the pleasure of taking graduate operating systems from Ousterhout while in graduate school; indeed, this is where the two co-authors got to know each other, eventually leading to marriage, kids, and even this book. Thus, you really can blame Ousterhout for this entire mess you're in.

[P+95] "Informed Prefetching and Caching"

R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka

SOSP '95

A fun paper about some very cool ideas in file systems, including how applications can give the OS advice about what files it is accessing and how it plans to access them.

Homework

This program, `scheduler-mlfq.py`, allows you to see how the MLFQ scheduler presented in this chapter behaves. As before, you can use this to generate problems for yourself using random seeds, or use it to construct a carefully-designed experiment to see how MLFQ works under different circumstances. To run the program, type:

```
prompt> ./scheduler-mlfq.py
```

Use the help flag (-h) to see the options:

```
Usage: scheduler-mlfq.py [options]
Options:
  -h, --help                show this help message and exit
  -s SEED, --seed=SEED     the random seed
  -n NUMQUEUES, --numQueues=NUMQUEUES
                           number of queues in MLFQ (if not using -Q)
  -q QUANTUM, --quantum=QUANTUM
                           length of time slice (if not using -Q)
  -Q QUANTUMLIST, --quantumList=QUANTUMLIST
                           length of time slice per queue level,
                           specified as x,y,z,... where x is the
                           quantum length for the highest-priority
                           queue, y the next highest, and so forth
  -j NUMJOBS, --numJobs=NUMJOBS
                           number of jobs in the system
  -m MAXLEN, --maxlen=MAXLEN
                           max run-time of a job (if random)
  -M MAXIO, --maxio=MAXIO
                           max I/O frequency of a job (if random)
  -B BOOST, --boost=BOOST
                           how often to boost the priority of all
                           jobs back to high priority (0 means never)
  -i IOTIME, --iotime=IOTIME
                           how long an I/O should last (fixed constant)
  -S, --stay                reset and stay at same priority level
                           when issuing I/O
  -l JLIST, --jlist=JLIST
                           a comma-separated list of jobs to run,
                           in the form x1,y1,z1:x2,y2,z2:... where
                           x is start time, y is run time, and z
                           is how often the job issues an I/O request
  -c                        compute answers for me
```

There are a few different ways to use the simulator. One way is to generate some random jobs and see if you can figure out how they will behave given the MLFQ scheduler. For example, if you wanted

to create a randomly-generated three-job workload, you would simply type:

```
prompt> ./scheduler-mlfq.py -j 3
```

What you would then see is the specific problem definition:

Here is the list of inputs:

```
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS quantum length for queue 2 is 10
OPTIONS quantum length for queue 1 is 10
OPTIONS quantum length for queue 0 is 10
OPTIONS boost 0
OPTIONS ioTime 0
OPTIONS stayAfterIO False
```

For each job, three defining characteristics are given:

```
startTime : at what time does the job enter the system
runTime   : the total CPU time needed by the job to finish
ioFreq    : every ioFreq time units, the job issues an I/O
            (the I/O takes ioTime units to complete)
```

Job List:

```
Job 0: startTime 0 - runTime 84 - ioFreq 7
Job 1: startTime 0 - runTime 42 - ioFreq 2
Job 2: startTime 0 - runTime 51 - ioFreq 4
```

Compute the execution trace for the given workloads.

If you would like, also compute the response and turnaround times for each of the jobs.

Use the `-c` flag to get the exact results when you are finished.

This generates a random workload of three jobs (as specified), on the default number of queues with a number of default settings. If you run again with the solve flag on (`-c`), you'll see the same print out as above, plus the following:

Execution Trace:

```
[time 0] JOB BEGINS by JOB 0
[time 0] JOB BEGINS by JOB 1
[time 0] JOB BEGINS by JOB 2
[time 0] Run JOB 0 at PRI 2 [TICKSLEFT 9 RUNTIME 84 TIMELEFT 83]
[time 1] Run JOB 0 at PRI 2 [TICKSLEFT 8 RUNTIME 84 TIMELEFT 82]
[time 2] Run JOB 0 at PRI 2 [TICKSLEFT 7 RUNTIME 84 TIMELEFT 81]
[time 3] Run JOB 0 at PRI 2 [TICKSLEFT 6 RUNTIME 84 TIMELEFT 80]
[time 4] Run JOB 0 at PRI 2 [TICKSLEFT 5 RUNTIME 84 TIMELEFT 79]
[time 5] Run JOB 0 at PRI 2 [TICKSLEFT 4 RUNTIME 84 TIMELEFT 78]
[time 6] Run JOB 0 at PRI 2 [TICKSLEFT 3 RUNTIME 84 TIMELEFT 77]
[time 7] IO_START by JOB 0
[time 7] Run JOB 1 at PRI 2 [TICKSLEFT 9 RUNTIME 42 TIMELEFT 41]
[time 8] Run JOB 1 at PRI 2 [TICKSLEFT 8 RUNTIME 42 TIMELEFT 40]
[time 9] IO_START by JOB 1
```

...

Final statistics:

```
Job 0: startTime 0 - response 0 - turnaround 175
Job 1: startTime 0 - response 7 - turnaround 191
Job 2: startTime 0 - response 9 - turnaround 168

Avg 2: startTime n/a - response 5.33 - turnaround 178.00
```

The trace shows exactly, on a millisecond-by-millisecond time scale, what the scheduler decided to do. In this example, it begins by running Job 0 for 7 ms until Job 0 issues an I/O; this is entirely predictable, as Job 0's I/O frequency is set to 7 ms, meaning that every 7 ms it runs, it will issue an I/O and wait for it to complete before continuing. At that point, the scheduler switches to Job 1, which only runs 2 ms before issuing an I/O. The scheduler prints the entire execution trace in this manner, and finally also computes the response and turnaround times for each job as well as an average.

You can also control various other aspects of the simulation. For example, you can specify how many queues you'd like to have in the system (-n) and what the quantum length should be for all of those queues (-q); if you want even more control and varied quantum length per queue, you can instead specify the length of the quantum for each queue with -Q, e.g., -Q 10, 20, 30 simulates a scheduler with three queues, with the highest-priority queue having a 10-ms time slice, the next-highest a 20-ms time-slice, and the low-priority queue a 30-ms time slice.

If you are randomly generating jobs, you can also control how long they might run for ($-m$), or how often they generate I/O ($-M$). If you, however, want more control over the exact characteristics of the jobs running in the system, you can use $-l$ (lower-case "L") or $--jlist$, which allows you to specify the exact set of jobs you wish to simulate. The list is of the form: $x1,y1,z1:x2,y2,z2:...$ where x is the start time of the job, y is the run time (i.e., how much CPU time it needs), and z the I/O frequency (i.e., after running z ms, the job issues an I/O; if z is 0, no I/Os are issued).

For example, if you wanted to recreate the example in Figure 8.4, you would specify a job list as follows:

```
prompt> ./scheduler-mlfq.py --jlist 0,180,0:100,20,0 -Q 10,10,10
```

Running the simulator in this way creates a three-level MLFQ, with each level having a 10-ms time slice. Two jobs are created: Job 0 which starts at time 0, runs for 180 ms total, and never issues an I/O; Job 1 starts at 100 ms, needs only 20 ms of CPU time to complete, and also never issues I/Os.

Finally, there are three more parameters of interest. The $-B$ flag, if set to a non-zero value, boosts all jobs to the highest-priority queue every N milliseconds, when invoked as such:

```
prompt> ./scheduler-mlfq.py -B N
```

The scheduler uses this feature to avoid starvation as discussed in the chapter. However, it is off by default.

The $-S$ flag invokes older Rules 4a and 4b, which means that if a job issues an I/O before completing its time slice, it will return to that same priority queue when it resumes execution, with its full time-slice intact. This enables gaming of the scheduler.

Finally, you can easily change how long an I/O lasts by using the $-i$ flag. By default in this simplistic model, each I/O takes a fixed amount of time of 5 milliseconds or whatever you set it to with this flag.

Questions

1. Run a few randomly-generated problems with just two jobs and two queues; compute the MLFQ execution trace for each. Make your life easier by limiting the length of each job and turning off I/Os.

2. How would you run the scheduler to reproduce each of the examples in the chapter?
3. How would you configure the scheduler parameters to behave just like a round-robin scheduler?
4. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the `-S` flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.
5. Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the `-B` flag) in order to guarantee that a single long-running (and potentially-starving) job gets at least 5% of the CPU?