Overview

Relaxing Assumption #1

Now to relax our last assumption that **the scheduler knows the length of every job**.

We will solve this issue by creating a scheduler that predicts the future based on the recent past. The **multi-level feedback queue** scheduler is the subject of this section.

This section should help us answer the following question?

- **How can we create a scheduler that reduces both interactive job response time and turnaround time without prior knowledge of the job length?**

Introduction

The **Multi-level Feed-back Queue (MLFQ)** scheduler has been developed throughout time and can be found in several modern systems.

MLFQ seeks to address two key issues.
1. First, it would like to optimize turnaround time, which 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.
1. Second, MLFQ wants to make a system feel responsive to interactive users while minimizing reaction time.

How can we create a scheduler to meet these goals when we don't know anything about processes?

How can the scheduler learn about the jobs it executes to make better scheduling decisions?

MLFQ: Basic Rules

The MLFQ has many **queues**, each with a different **priority level**.

A single job is available to run at any time.

For each time slot, **MLFQ** chooses the job with the highest priority (i.e., in the highest queue).

Many jobs can share a queue and have the same priority. In this case, we will use round-robin scheduling to decide which to run.

So here are the first two MLFQ rules:

• Rule 1: If Priority(A) > Priority(B), then A runs (B does not).
• Rule 2: If Priority(A) = Priority(B), then A & B run in Round Robin.

Prioritization is the key to MLFQ scheduling. A job's **priority** is determined by its **observed behavior**, rather than by a fixed value.

MLFQ will keep its priority high if a job repeatedly surrenders the CPU while waiting for I/O.

- **MLFQ lowers its priority if a job uses the CPU intensively for extended time periods.**
- **MLFQ will then utilize the job's past to anticipate its future behavior.**

If we were to imagine the queues at any given time, we may see something like the figure to the left. In the illustration, jobs A and B have the highest priority, whereas jobs C and D have the lowest.

The scheduler would just alternate time slices between A and B because these are the highest priority jobs in the system. Jobs C and D would never even get to run!

Presenting a snapshot of some queues doesn't really explain how MLFQ works

We need to know how job priority shifts over time.
##
#
Fill in the blanks to complete the statement below.

If 2 jobs share the same priority level, how does the OS decide which job to run?

Attempt #1: How To Change Priority

Now we must decide how MLFQ will change a job's priority level (and queue) over time.

We must keep our workload in mind. This is a mix of:

- Interactive jobs that run quickly (and may frequently relinquish the CPU to other jobs), and
- Some longer-running "CPU-bound" jobs that require a lot of CPU time, but where response time is not an issue.

Here is our first priority-adjusting algorithm:

• Rule 3: When a job is added to the system, it is given the greatest priority (the topmost queue).

• Rule 4a: A job's priority is reduced if it uses the entire time slice (i.e., it is moved down one queue).

• Rule 4b: A job's priority is still the same if it releases the CPU before the time slice expires.

What priority is given to new jobs that enter the system?

Example: A Single, Long-Running Job

Example 1: A Single Long-Running Job

First, we'll explore what happens when a job runs for a long time. This figure shows the job's progress in a three-queue scheduler.

In the example:

- The job is, placed first in the queue (Q2).
- The scheduler reduces the job's priority by one after a time slice of 10 ms, putting it on Q1.
- After a time slice at Q1, the job's priority is lowered to Q0, where it remains.

Example: Here Comes a Short Job

Example 2: Along Came A Short Job

Let's look at a more complex case to see how MLFQ approaches SJF.
Here, we have two jobs.

- A is a long-running CPU-intensive work, and
- B is an interactive job that runs briefly.

Suppose A runs for a while before B arrives.

What happens?
Will MLFQ approximate SJF for B?

The graphic to the left shows the result.

- A is running in the lowest-priority queue (like any long-running CPU-intensive job);
- B arrives at time T = 100, and goes into the highest queue;
- B completes in two time slices, because to its short runtime (20 ms).
- Then A resumes (at low priority).

Because the algorithm doesn't know whether a job will be short or long, **it guesses it will be short and so gives it high priority**.

In any case, it will quickly finish if it is a small job, else it will go down the queues and become a long-running batch-like operation.

MLFQ comes close to SJF in this way.

Example: I/O

Now let's look at an I/O example.

A process that gives up the processor before using up its time slice is kept at the same priority level (**Rule 4b**).

This rule's objective is clear:

- **If an interactive job is doing a lot of I/O (such waiting for user input via the keyboard or mouse), it will give up the CPU early.**

In this scenario, we don't want to penalize the job doing I/O, so we keep it at the same level.

In this graphic, an interactive task B needs the CPU for 1 ms before conducting an I/O and fights for the CPU with a long-running batch job A.

Because B keeps releasing the CPU, the MLFQ approach keeps B at the top priority queue.

MLFQ further achieves its goal of running interactive jobs quickly if B is an interactive job.

Issues With Our Current Model

Our MLFQ seems to do an excellent job of balancing long-running jobs with brief or I/O-intensive interactive ones.

Sadly, our current approach has serious weaknesses.

1. The first issue is **starvation**: Too many interactive jobs in the system use up all of the CPU time, leaving long-running jobs idle (they starve). Even in these circumstances, progress on these jobs would be nice.
2. A clever user can rewrite their program to **game the scheduler**.

Gaming the scheduler refers to doing something shady to get more than your fair share of the resource.

We can leverage this weakness by doing an I/O operation (to a file you don't care about) before the time slice ends, regaining the CPU time and remaining in the same queue.

A job could practically dominate the CPU if it ran for 99% of a time slice before giving up the CPU.

An application's behavior may evolve over time, from CPU-bound to interactive. With our current approach, such a job would be out of luck and not regarded like other interactive jobs.

Attempt #2: The Priority Boost

Let's add a rule and see if we can avoid starvation.

How can we make sure that CPU-bound jobs make progress (even if it is slow)? The idea is to regularly raise the priority of all jobs in the system.

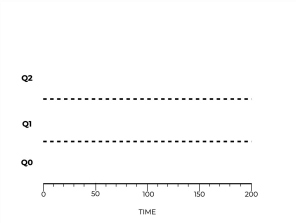
Here, let's keep it simple by putting them all at the top queue. So our new rule is:

- **Rule 5: After a certain amount of time, S, move all jobs in the system to the topmost queue.**

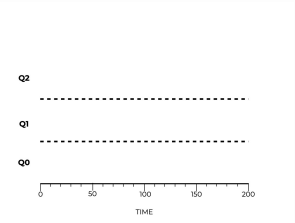
Our new rule addresses two issues.

1. By sitting at the top of the queue, a job is guaranteed not to starve, as it shares the CPU with other high-priority jobs in a round-robin fashion, making sure it eventually gets service.
2. After the priority boost, the scheduler treats a CPU-bound job properly if it becomes interactive.

For example, a long-running job contending for CPU resources with two short-running interactive jobs is shown below.



Without Priority Boost



With Priority Boost

- On the left, there is no priority boost, and so the long-running job gets starved once the two short jobs arrive.
- The example on the right provides a priority boost every 50 ms (which is probably too small), ensuring that the long-running job will at least make some progress, getting boosted to the highest priority and thus running periodically.

A time period **S** is obviously required, so what should it be?

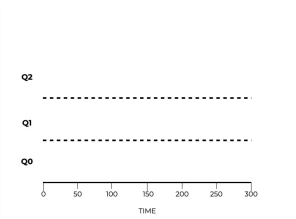
This value can be tricky to set.

With a low value, long-running jobs may suffer; a high value may cause interactive jobs to suffer.

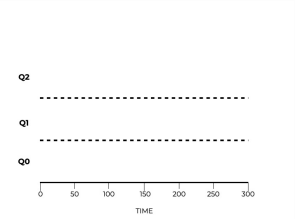
Attempt #3: Better Accounting

How can we prevent the gaming of our scheduler?

Our problem here is with Rules 4a and 4b, which allow jobs to retain priority if they free the CPU before the time slice expires.



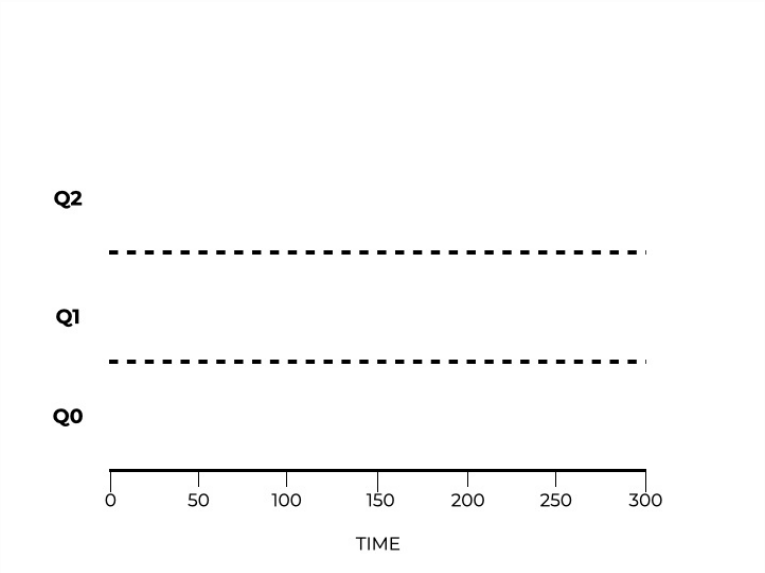
Without Gaming Tolerance



With Gaming Tolerance

Our answer is to better account for CPU time at each MLFQ level.

The scheduler could keep track of how much of a time slice a process used at each priority level, instead of forgetting. After a process has used its limit, it is demoted to the next priority queue, regardless of if it uses a long time slice or many little ones.



Lower Priority, Longer Quanta

So we can combine Rules 4a and 4b into one:

- **Rule 4: When a job exhausts its time allotment (regardless of how many times the CPU is used), its priority is lowered (i.e., it moves down one queue).**

For example, workload tries to game the scheduler with the old Rules 4a and 4b (on the left), as well as the new anti-gaming Rule 4.

- Without gaming protection, a process can dominate CPU time by issuing I/O right before a time slice finishes.
- With these protections in place, the process proceeds down the queues slowly, preventing it from gaining an unfair CPU share.

Tuning MLFQ and Other Issues

Other scheduling concerns exist with MLFQ. How do we **parameterize** such a scheduler? How long should each queue be?

There are no clear answers, and only practice with workloads and following scheduler modification will lead to a satisfying balance.

Most MLFQ variations allow changing time-slice length between queues.

A scheduler administrator can manipulate the Time-Sharing scheduling class (TS) in Solaris to change the way a process's priority is changed over time, how long each time slice is, and how often a job's priority is boosted.

Other MLFQ schedulers employ mathematical formulas to alter priorities rather than a table or the criteria given in this chapter.

Many schedulers have additional functionality, like reserving the highest priority levels for operating system work, preventing regular user processes from reaching those levels.

Summary

We described a scheduling method called Multi-Level Feedback Queue (MLFQ). It has multiple levels of queues and uses feedback to decide job priority, p Paying attention to how jobs behave over time and treating them accordingly.

The refined MLFQ rules are provided here for your convenience:

- **Rule 1**: If Priority(A) > Priority(B), then A runs (B doesn't).
- **Rule 2**: If Priority(A) = Priority(B), A & B are processed in round-robin fashion using the time slice (quantum length) given to them.
- **Rule 3**: Any job that enters the system is given the highest priority (placed at the very top of the queue).
- **Rule 4**: When a job uses up its allowance of time (regardless of how many times it gives up the CPU), the priority of the job is decreased (i.e., it drops down one queue).
- **Rule 5**: After some time period S, move all jobs to the top of the queue.

MLFQ is interesting because it **observes the execution of a job and prioritizes it accordingly without any previous knowledge**. It can give outstanding overall performance (equivalent to SJF/STCF) for short-running interactive jobs, while still being fair and progressing for long-running CPU-intensive workloads.

Lab Intro

This program, `mlfq.py`, simulates the **MLFQ scheduler** described in this section. You can use it to create problems for yourself using random seeds, or to create an experiment to test MLFQ in various situations. To run the simulator, type the following into the console and press `return`:

```
./mlfq.py
```

You can use the help flag, `-h`, to see all of the options available to you.

```
./mlfq.py -h
```

You can use the simulator to generate random jobs and try to figure out how they will behave given the MLFQ scheduler. To create a randomly generated three job workload, you would type:

```
./mlfq.py -j 3
```

The output will be a specific problem definition:

```
Here is the list of inputs:
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS allotments for queue  2 is   1
OPTIONS quantum length for queue  2 is  10
OPTIONS allotments for queue  1 is   1
OPTIONS quantum length for queue  1 is  10
OPTIONS allotments for queue  0 is   1
OPTIONS quantum length for queue  0 is  10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False

For each job, three defining characteristics are given:
startTime : at what time does the job enter the system runTime : th
```

The goal is to compute the execution trace for the given workloads.

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

After you've finished computing, use the `-c` flag to get the exact results.

```
./mlfq.py -c
```

With the default settings, this generates a random workload of three jobs (as requested). If you run it again with th `-c`), you'll see the same thing plus:

```
Execution Trace:
[ time 0 ] JOB BEGINS by JOB 0 [ time 0 ] JOB BEGINS by JOB 1 [ tim
```

The trace shows the scheduler's exact decision millisecond by millisecond.

- It begins by running `Job 0` for 7 ms until `Job 0` issues an I/O
  - This is expected, because `Job 0`'s I/O frequency, `iofreq`, is set to 7 ms. This means that every 7 ms its running, it will issue an I/O and wait for it to complete
- The scheduler then changes to `Job 1`, which runs for 2 ms before issuing an I/O.
- The scheduler then computes the response and turnaround times for each job, as well as an average.

Modifying Parameters

You can also modify the simulation's parameters.

- For further control and varying quantum length per queue, you can specify the quantum (time slice) length fo
  `-Q`.
  - For example `-Q 10,20,30` mimics a scheduler with:
    - three queues, each with a
    - 10-ms priority slice,
    - a 20-ms priority slice, and
    - a 30-ms priority slice.
- You can control how much time each queue gets.
  - `-A 20,40,60`
    sets the time allotment each queue to 20ms, 40ms, and 60ms, accordingly.
- If you are generating jobs at random, you can control their duration (`-m`) and how often they generate I/O (`-M`).
- The `-l`
  (lower-case L) or —jlist options let you define the exact set of jobs you want to simulate.
  - The list is of the pattern: `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).

To reproduce the example from an earlier graphic in this section with 2 jobs, you would build a job list as follows:

```
./mlfq.py --jlist 0,180,0:100,20,0 -q 10
```

This creates a three-level MLFQ with:
* 10-ms time slices
* Job 0 starts at 0 and runs for 180 ms, never issuing I/Os
* Job 1 starts at 100 ms and runs for 20 ms, never issuing I/Os.

There are three more variables of importance to be aware of. When run with the
`-B`
flag with a non-zero value, all jobs are moved to the top priority queue every N milliseconds. This can be executed

```
./mlfq.py -B N
```

The scheduler uses this feature to avoid starvation. But it's off by default.

The `-S`
flag executes older Rules 4a and 4b. This means that if a job issues an I/O before completing its time slice, it will re

Finally, the `-i`
flag lets you to adjust the duration of an I/O. In this simplified approach, each I/O takes 5 milliseconds by default, c

The `-I`
flag allows you to control whether jobs that just finish an I/O are shifted to the front or back of the queue.

Try to create a few examples on your own!
#

Helpful Flags

```
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)
  -a ALLOTMENT, --allotment=ALLOTMENT
                        length of allotment (if not using -A)
  -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
  -A ALLOTMENTLIST, --allotmentList=ALLOTMENTLIST
                        length of time allotment per queue
level, specified as
                        x,y,z,... where x is the # of time
slices 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 randomly
generating)
  -M MAXIO, --maxio=MAXIO
                        max I/O frequency of a job (if randomly
generating)
  -B BOOST, --boost=BOOST
                        how often to boost the priority of all
jobs back to
                        high priority
  -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
  -I, --iobump          if specified, jobs that finished I/O
move immediately
                        to front of current queue
  -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
```

Lab

1. Run the simulator to generate a few problems with:
   - 2 jobs, `-q 2`, and
   - 2 queues, `-n 2`.

Try including the `seed` flag, `-s` with different values to generate different problems.

2. Try to run the simulator to reproduce your favorite examples from this section.

   - Can you figure out which example this is?
   - `./mlfq.py --jlist 0,180,0:100,20,0 -q 10`

3. Try to configure the simulator parameters to act like a Round Robin Scheduler

4. Turn on Scheduler Gaming with the `-S` flag. Can you create a workload with:

   - 2 jobs
   - 1 job consumes 99% of the CPU