Join on Zoom if you are remote (see website for link).

377 Operating Systems

Multi-Level Feedback Queue (MLFQ)

Last Time…

THE CRUX: HOW TO DEVELOP SCHEDULING POLICY

How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)
5. The run-time of each job is known

Note, this is unrealistic and we will relax these as we go along.

Scheduling Metrics

Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a scheduling metric.

For now, however, let us also simplify our life by having a single metric:

turnaround time

The turnaround time of a job is defined as the time at which the job *completes* minus the time at which the job *arrived* in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Response Time Definition

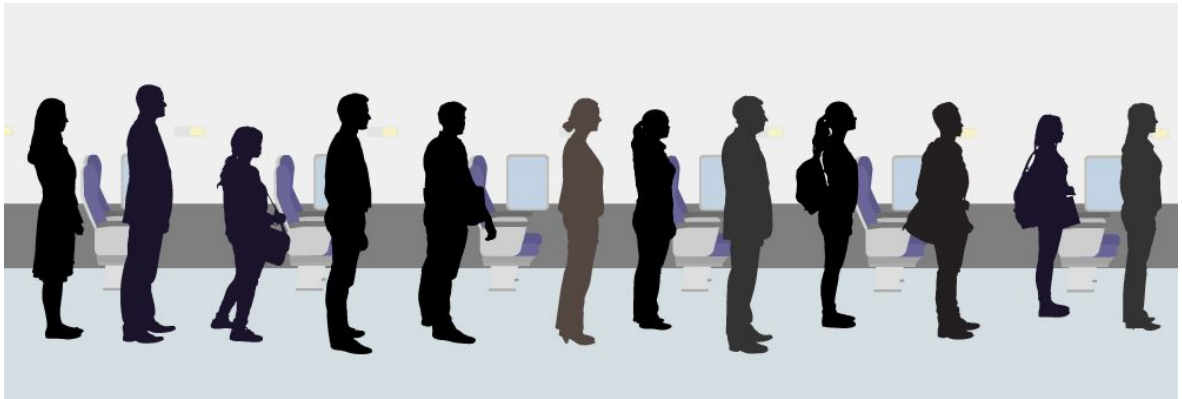
Response time is defined as the time from when the job arrives in a system to the first time it is scheduled. More formally:

$$T_{response} = T_{firstrun} - T_{arrival}$$

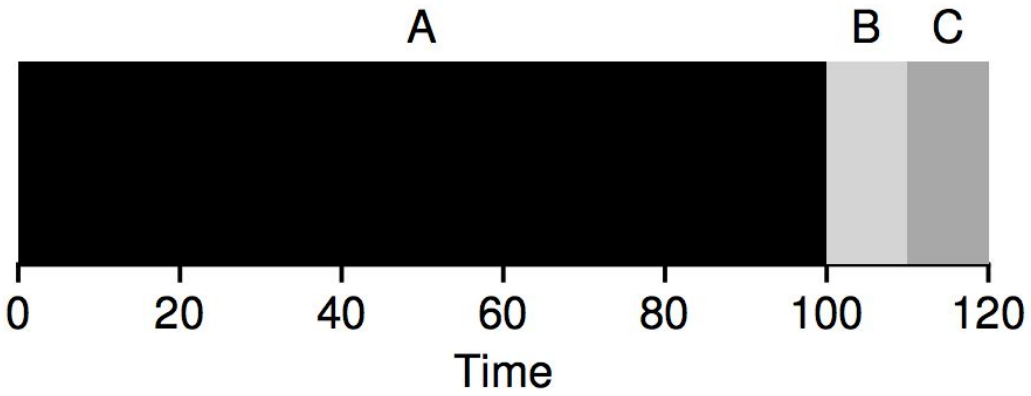
First In, First Out (FIFO)

The most basic algorithm we can implement is known as First In, First Out (FIFO) scheduling or sometimes First Come, First Served (FCFS).

FIFO has a number of positive properties: it is clearly simple and thus easy to implement. And, given our assumptions, it works pretty well.



FIFO Performance With #1 Relaxed…



Workload Assumptions

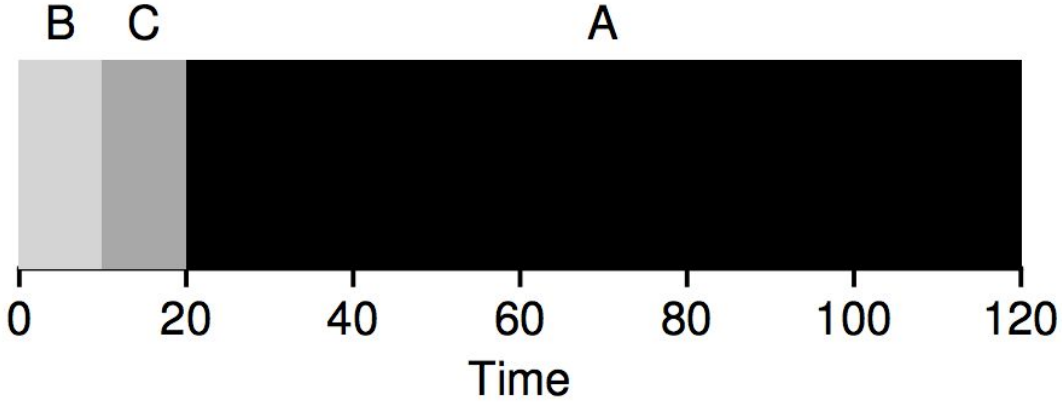
Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)
5. The run-time of each job is known

As you can see, A finished at 100, B at 110, and C at 120.

Thus, the average turnaround time is (100+110+120) / 3 = 110.

We fixed this with SJF…



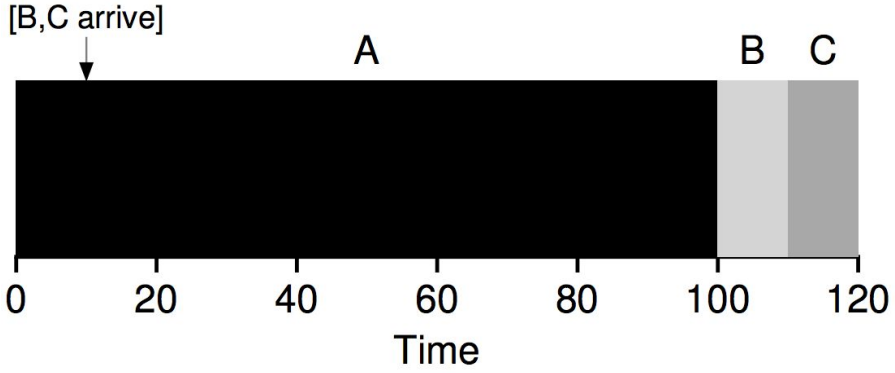
What if we applied SJF to our FIFO example from before?

What is the average turnaround time?

Thus, the average turnaround time is (10+20+120) / 3 = 50.

SJF performance when we relax #2

As you can see from the figure, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem. Average turnaround time for these three jobs is 103.33 seconds.



Workload Assumptions

Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., no I/O)
5. The run-time of each job is known

Then, we relaxed #3…

Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~Once started, each job runs to completion~~
4. All jobs only use the CPU (i.e., no I/O)
5. The run-time of each job is known

To address this concern, we need to relax assumption 3
(that jobs must run to completion).

We also need some machinery within the scheduler itself.

SJF + Preemption == :-)

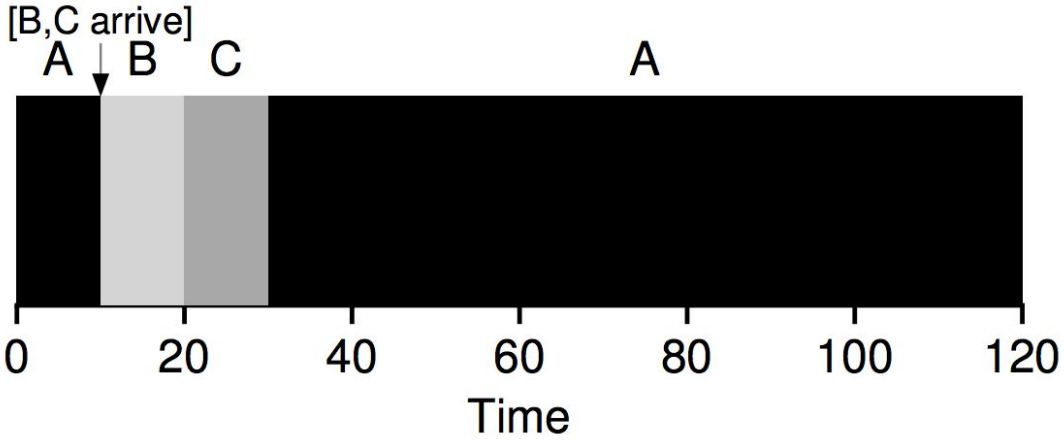
Fortunately, there is a scheduler which does exactly that:
Add preemption to SJF, known as the Shortest Time-to-Completion First (STCF) or Preemptive Shortest Job First (PSJF) scheduler.

Any time a new job enters the system:

1. STCF scheduler determines which of the remaining jobs (including the new job) has the least time left

2. STCF schedules the shortest job first.

STCF Example #1



$$\left(\frac{(120-0)+(20-10)+(30-10)}{3}\right)$$

The result is a much-improved average turnaround time: 50 seconds

Given our new assumptions, STCF is *provably optimal*; given that SJF is optimal if all jobs arrive at the same time, you should probably be able to see the intuition behind the optimality of STCF.

STCF Response Time: Not so good :-(

As you might be thinking, STCF and related disciplines are not particularly good for response time.

If three jobs arrive at the same time, l job has to wait for the previous two jobs to run in their enti eduled just once.

While great for turnaround time, this d for response time and interactivity.

Indeed, imagine sitting at a terminal, wait 10 seconds…

Thus, we are left with another problem:

how can we build a scheduler that is sensitive to response time?

Round Robin

The basic idea is simple:

Instead of running jobs to completion,

RR runs a job for a time slice (sometimes called a scheduling quantum),

then switches to the next job in the run queue.

RR: Is the grass always greener on the other side?

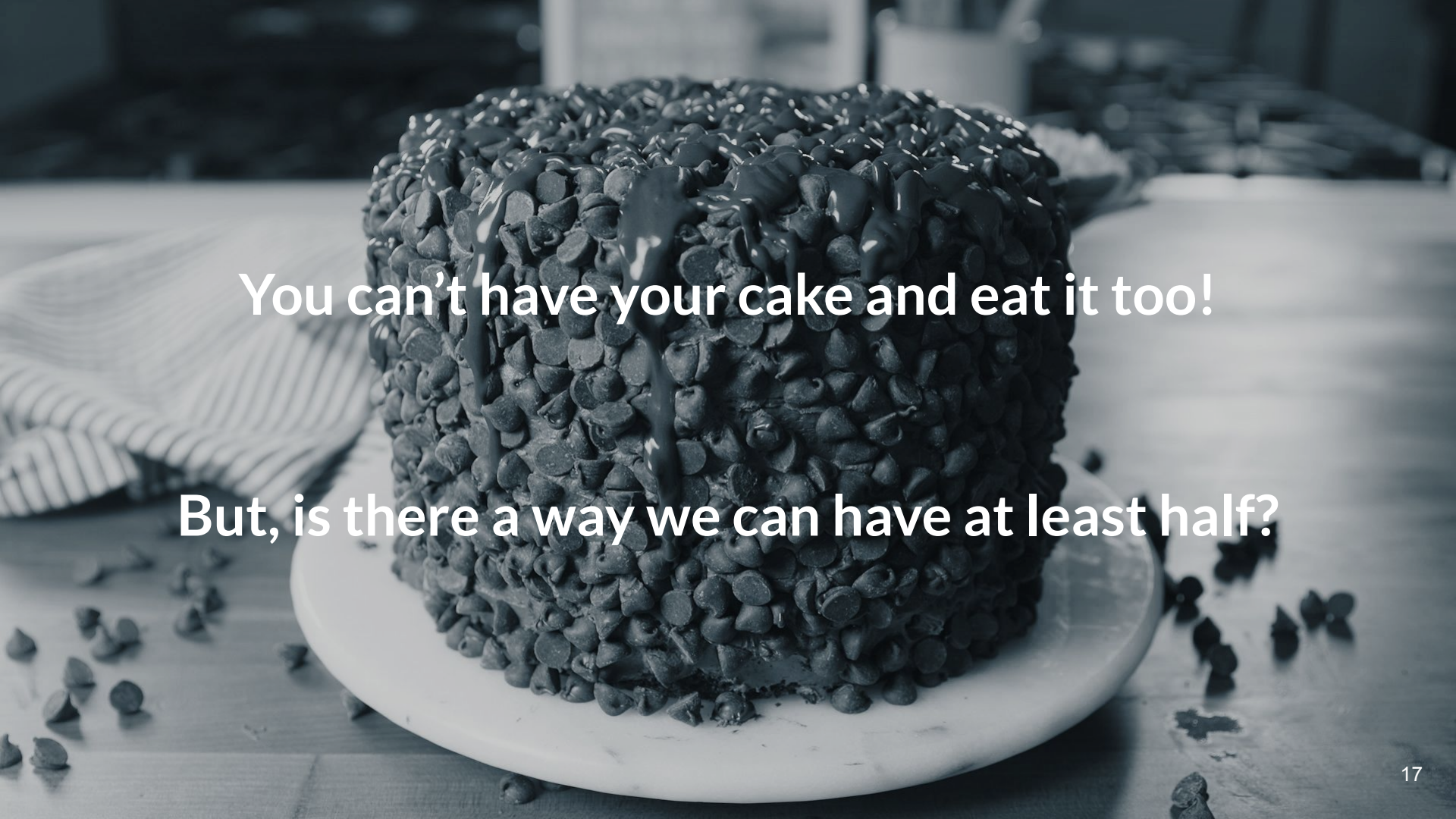
RR does well for response time.

The shorter the time slice, the better it is for RR.

But, what about the overhead of a context switch?

A context switch is expensive, so we need to be careful here.
In other words, shorter time slice means *context switch dominates performance*.

And, what about turnaround time?


You can’t have your cake and eat it too!
But, is there a way we can have at least half?

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~Once started, each job runs to completion~~
4. ~~All jobs only use the CPU (i.e., no I/O)~~
5. The run-time of each job is known

Incorporating I/O

Of course, all interesting programs perform some kind of I/O.

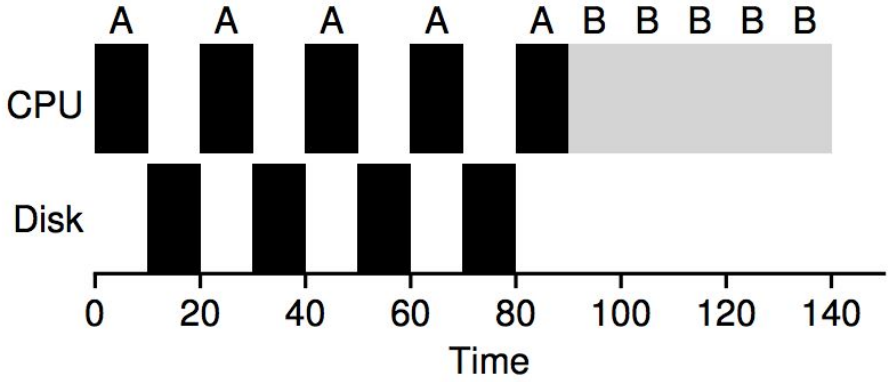
A scheduler clearly has to make a decision

- If an I/O request is made, the scheduler doesn't want to block other jobs.

If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive.

- The scheduler should probably schedule another job on the CPU at that time.

I/O Example



This is a poor use of resources!

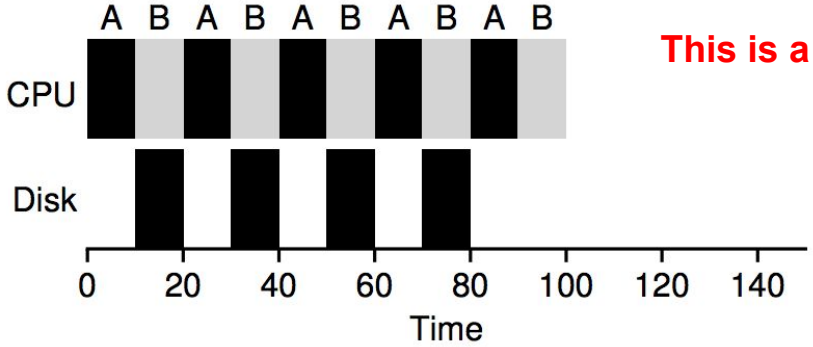
To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time.

Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.

However, there is one obvious difference:

- A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms)
- B simply uses the CPU for 50 ms and performs no I/O.
- The scheduler runs A first, then B after.

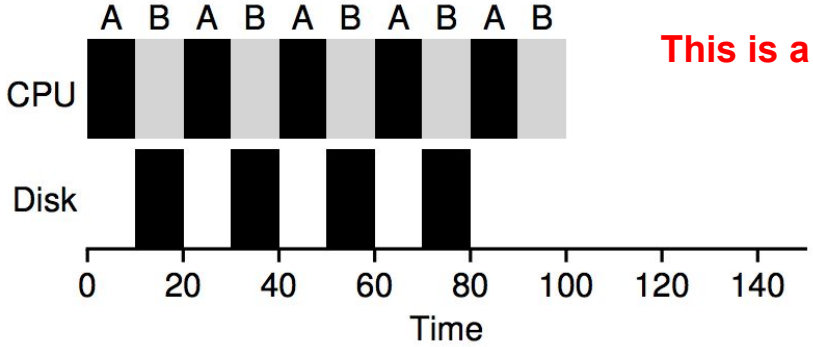
I/O Example



This is a better use of resources!

A common approach is to treat each 10-ms sub-job of A as an independent job. Thus, when the system starts, its choice is whether to schedule a 10-ms A or a 50-ms B.

I/O Example



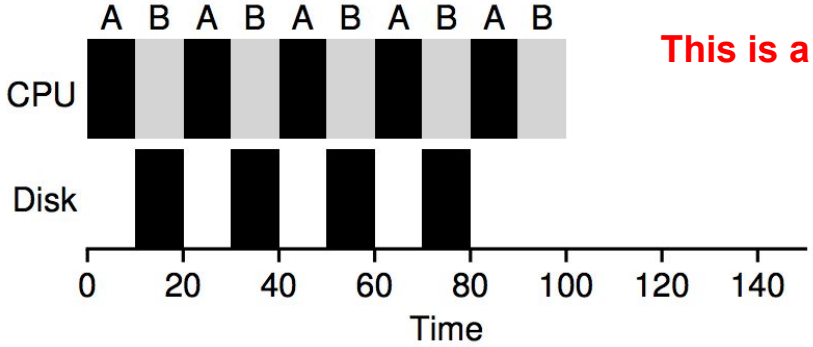
This is a better use of resources!

With STCF, the choice is clear: choose the shorter one, in this case A.

Then, when the first sub-job of A has completed, only B is left, and it begins running.

Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms. Doing so allows for overlap, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized.

I/O Example



This is a better use of resources!

And thus we see how a scheduler might incorporate I/O.

By treating each CPU burst as a job, the scheduler makes sure processes that are “interactive” get run frequently.

While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~Once started, each job runs to completion~~
4. ~~All jobs only use the CPU (i.e., no I/O)~~
5. The run-time of each job is known

Workload Assumptions

First, we want to make some simplifying assumptions about running processes

This is typically referred to as the workload.

Simplified assumptions about running processes (often called jobs):

1. ~~Each job runs for the same amount of time~~
2. ~~All jobs arrive at the same time~~
3. ~~Once started, each job runs to completion~~
4. ~~All jobs only use the CPU (i.e., no I/O)~~
5. ~~The run-time of each job is known~~

The Multi-Level Feedback Queue

The Multi-level Feedback Queue (MLFQ) scheduler was first described by Corbato et al. in 1962 in a system known as the Compatible Time-Sharing System (CTSS).

This work, along with later work on Multics, led the ACM to award Corbato its highest honor, the Turing Award. The scheduler has subsequently been refined throughout the years to the implementations you will encounter in some modern systems

The Problem MLFQ Tries to Solve

First, MLFQ would like to optimize turnaround time.

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.

Algorithms like Round Robin reduce response time but are **terrible** for turnaround time.

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?

TIP: LEARN FROM HISTORY

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

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.

MLFQ: Basic Idea

In our treatment, the MLFQ has a number of distinct queues.

Each queue is 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

MLFQ: Basic Rules

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.

MLFQ: Intuition

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.

MLFQ: Intuition

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.

MLFQ: Intuition

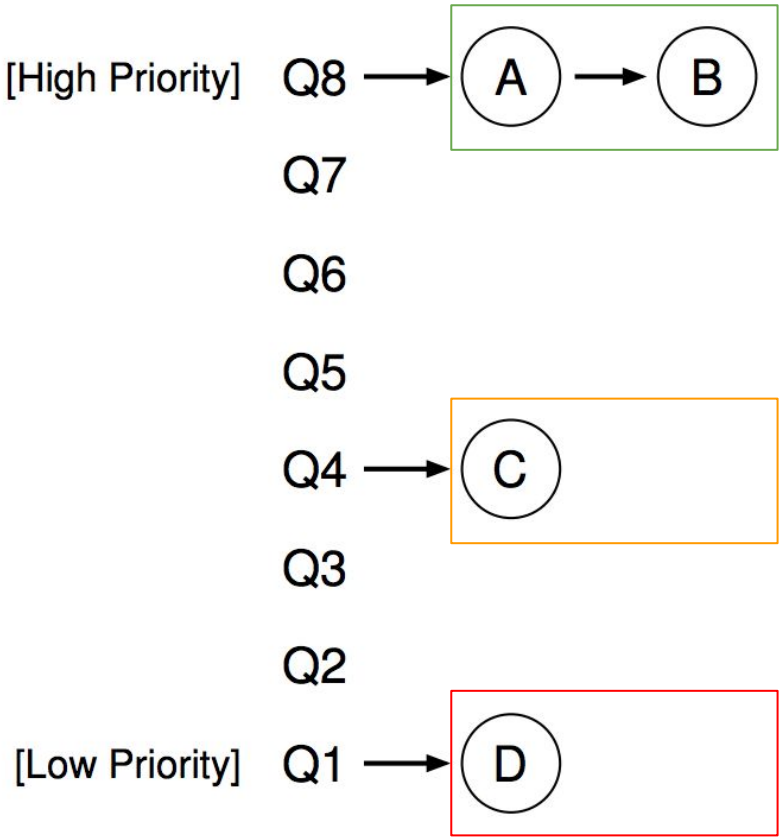
The key to MLFQ scheduling therefore lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behavior.

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**.

MLFQ: What does it look like?

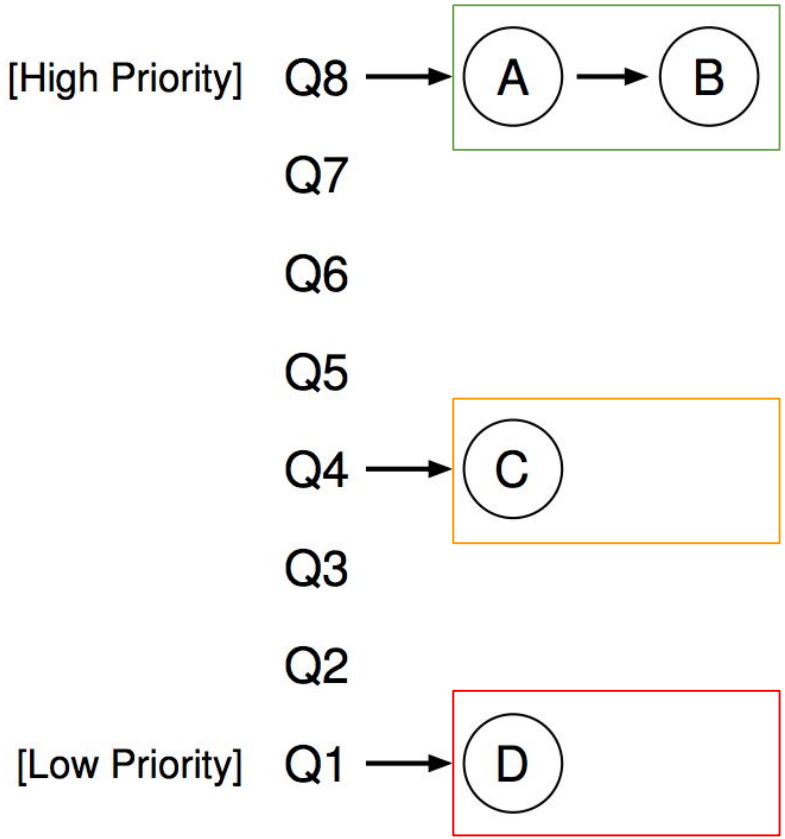
In the figure, two jobs (A and B) are at the highest priority level, while job C is in the middle and job D is at the lowest priority.



MLFQ: What does it look like?

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 priority.

The scheduler would just alternate time slices between A and B because they are the highest priority jobs in the system; poor jobs C and D would never even get to run — an outrage!

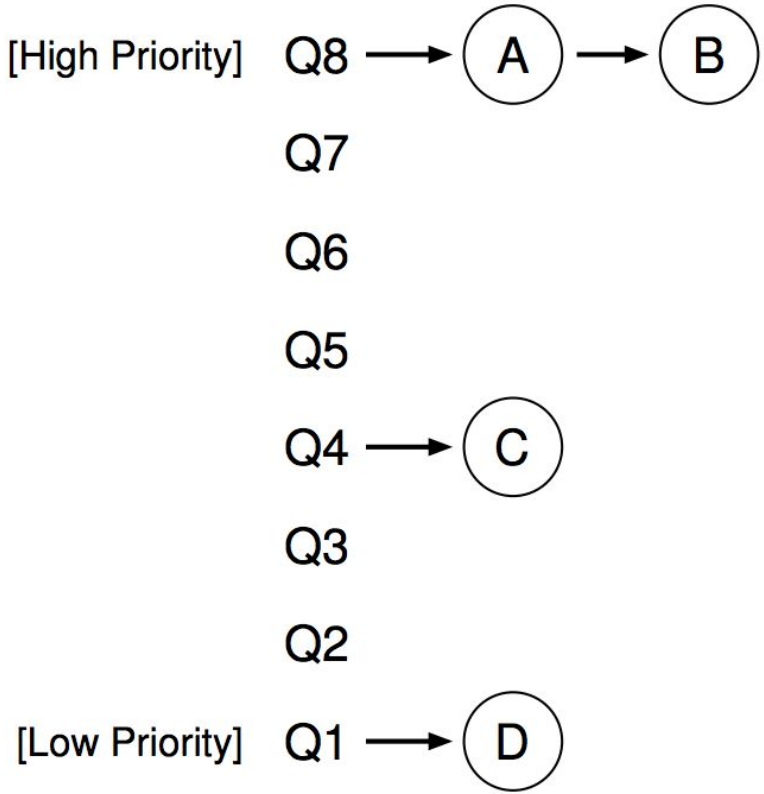


MLFQ: What does it look like?

Of course, just showing a static snapshot of some queues does not really give you an idea of how MLFQ works.

What we need is to understand how job priority **change** over time.

That is exactly what we will do next.



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.

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.

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:

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).

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).

Attempt #1: How to change priority

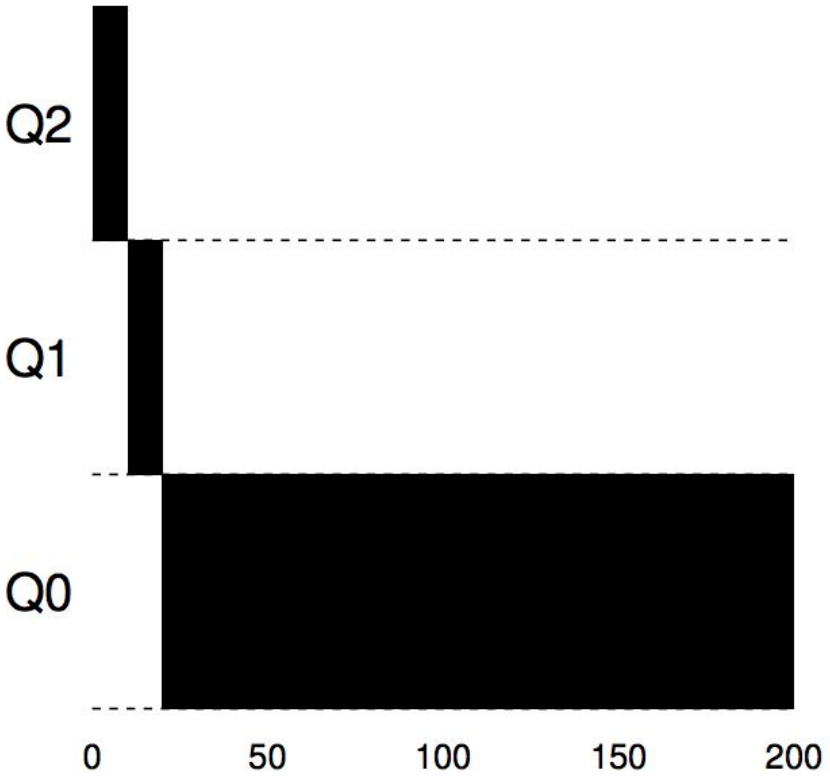
We now must decide how MLFQ is going to change the priority level of a job (and thus which queue it is on) over the lifetime of a job.

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.

Example 1: A Single Long-Running Job



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 see how MLFQ tries to approximate SJF.

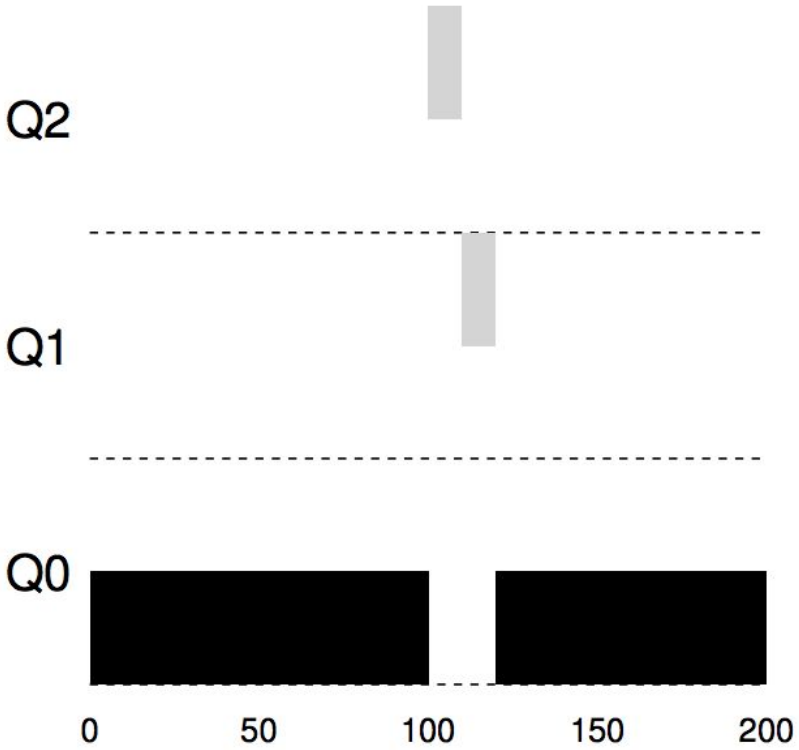
In this example, there are two jobs:

- A, which is a long-running CPU-intensive job
- B, which is a short-running interactive job.

Assume A has been running for some time, and then B arrives.

What will happen? Will MLFQ approximate SJF for B?

Example 2: Along Came a Short Job

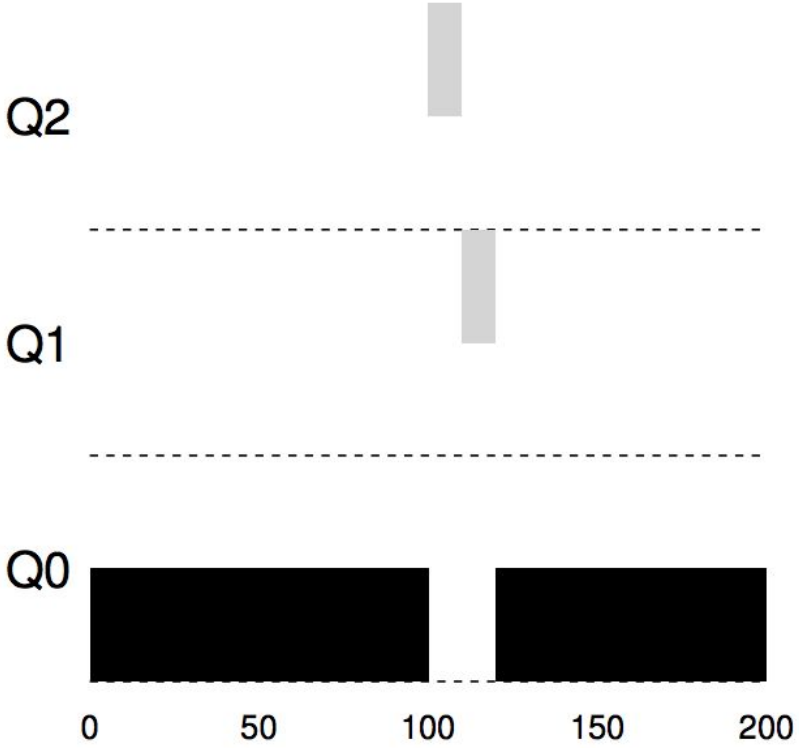


A (shown in black) is running along in the lowest-priority queue (as would any long-running CPU-intensive jobs).

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).

Example 2: Along Came a Short Job



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?

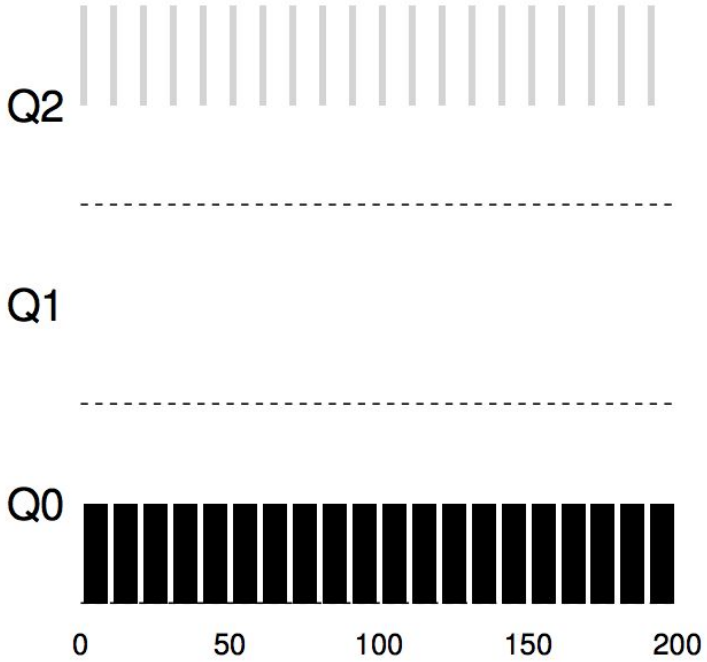
Let’s now look at an example with some I/O.

As **Rule 4b** states, 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 is doing a lot of I/O, it will relinquish the CPU before its time slice is complete
- In such case, we wish not to penalize the job and thus simply keep it at the same level.

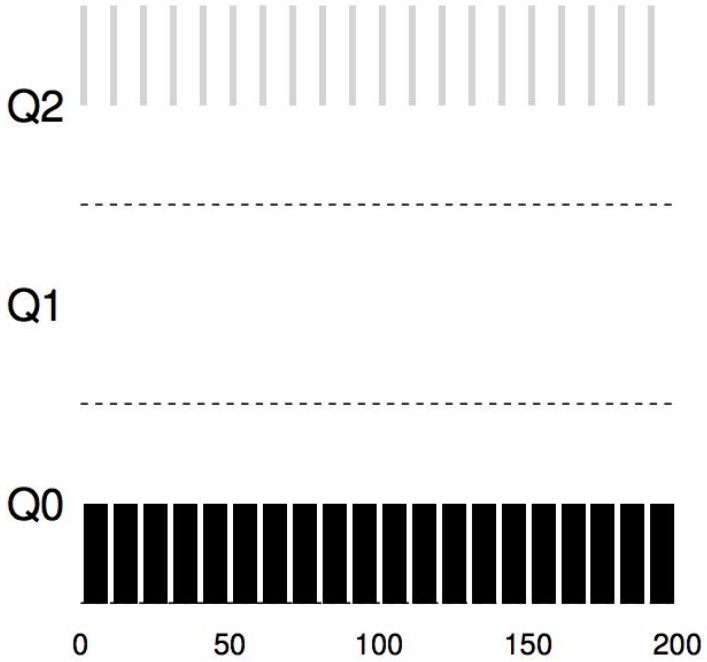
Example 3: What about I/O?



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

Example 3: What about I/O?

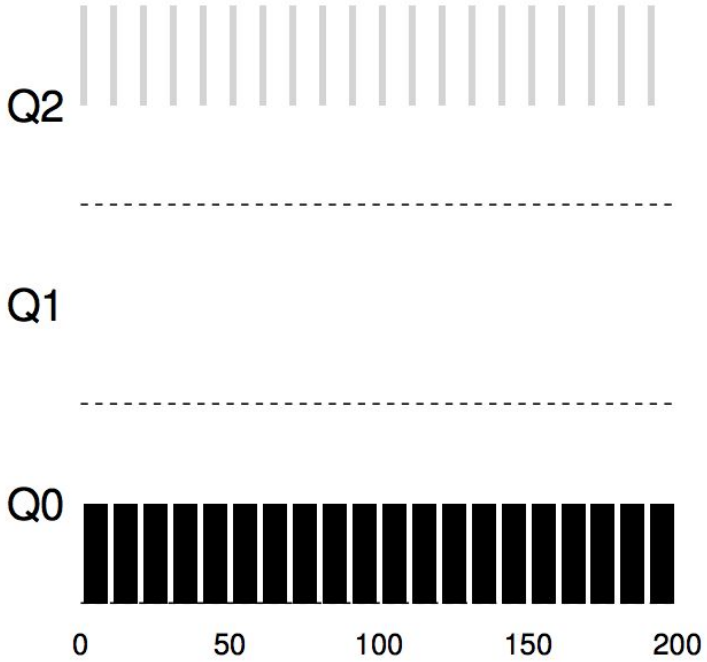


We thus have a basic MLFQ. It seems to do a good job, sharing the CPU fairly between long-running jobs, and letting short or I/O-intensive interactive jobs run quickly.

Unfortunately, the approach we have developed thus far contains **serious flaws**.

Can you think of any?

Example 3: What about I/O?

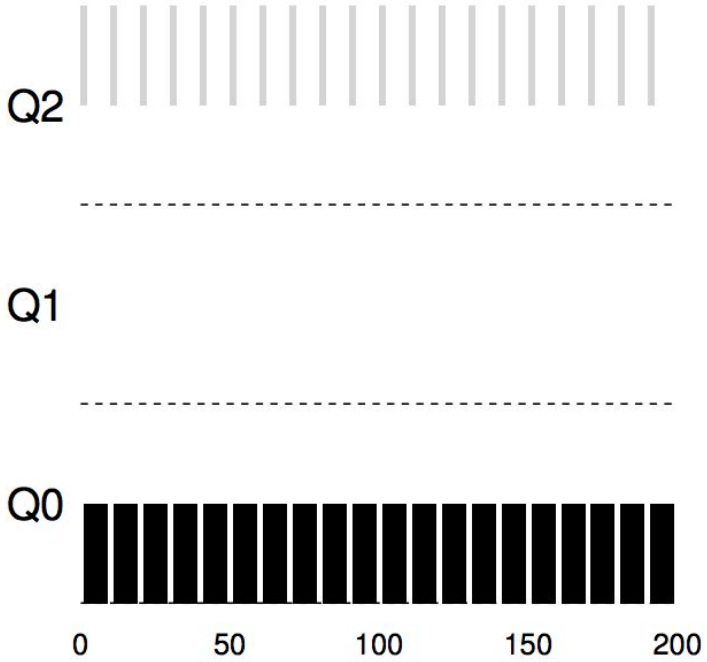


First, there is the problem of starvation.

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 (they starve).

We’d like to make some progress on these jobs even in this scenario.

Example 3: What about I/O?



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.

How might you do this?

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; hence, a new rule:

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

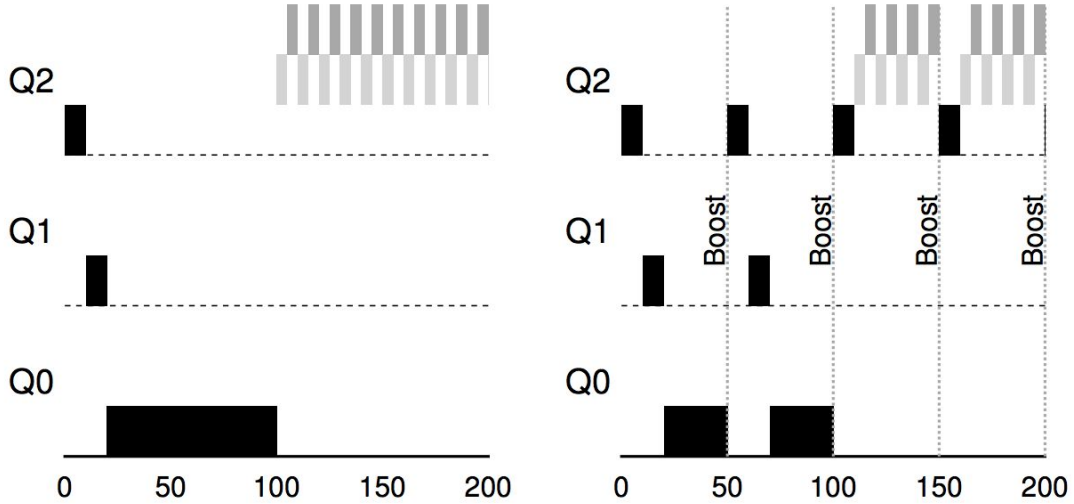
Attempt #2: The Priority Boost

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.

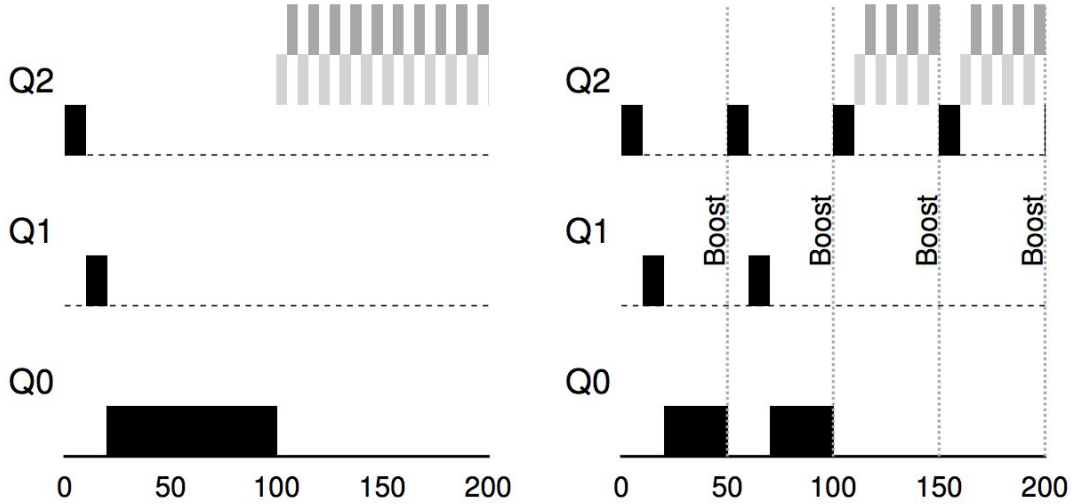
Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the **priority boost**.

Attempt #2: The Priority Boost



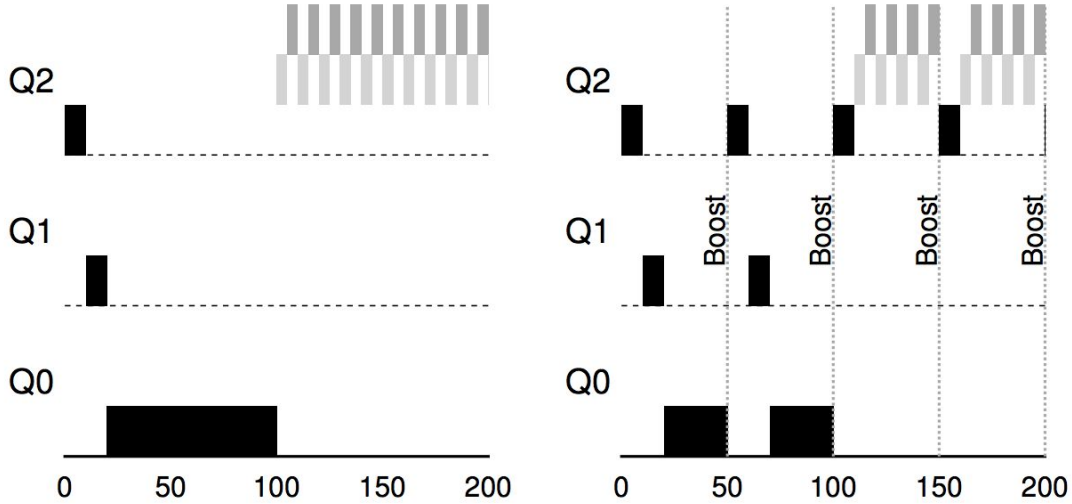
In this example, we show the behavior of a long-running job when competing for the CPU with two short-running interactive jobs.

Attempt #2: The Priority Boost



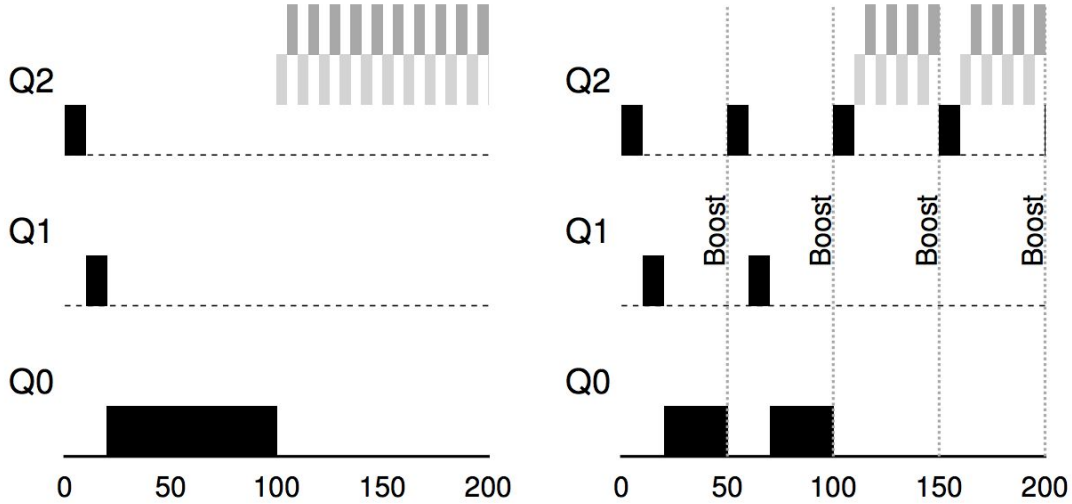
On the left, there is no priority boost, and thus the long-running job gets starved once the two short jobs arrive.

Attempt #2: The Priority Boost



On the right, there is a priority boost every 50 ms, 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

Attempt #2: The Priority Boost



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.

Of course, the addition of the time period S leads to the obvious question: what should S be set to? 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.

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 by relinquishing the CPU before the time slice expires.

So what should we do?

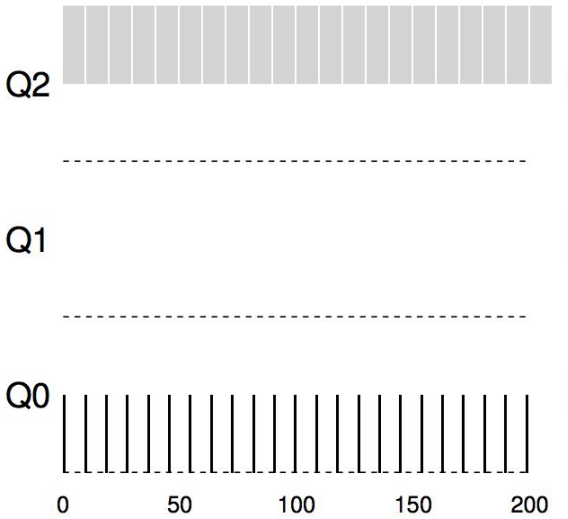
Attempt #3: Better Accounting

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).

Attempt #3: Better Accounting



This example 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**.

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.

Case study: Linux Completely Fair Scheduler

https://en.wikipedia.org/wiki/Completely_Fair_Scheduler

The data structure used for the scheduling algorithm is a red-black tree in which the nodes are scheduler-specific `sched_entity` structures. These are derived from the general `task_struct`process descriptor, with added scheduler elements.

The nodes are indexed by processor "*execution time*" in nanoseconds.[3]

A "*maximum execution time*" is also calculated for each process. This time is based upon the idea that an "ideal processor" would equally share processing power amongst all processes. Thus, the *maximum execution time* is the time the process has been waiting to run, divided by the total number of processes, or in other words, the *maximum execution time* is the time the process would have expected to run on an "ideal processor".

When the scheduler is invoked to run a new process, the operation of the scheduler is as follows:

1. The leftmost node of the scheduling tree is chosen (as it will have the lowest spent *execution time*), and sent for execution.
2. If the process simply completes execution, it is removed from the system and scheduling tree.
3. If the process reaches its *maximum execution time* or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent *execution time*.
4. The new leftmost node will then be selected from the tree, repeating the iteration.

If the process spends a lot of its time sleeping, then its spent time value is low and it automatically gets the priority boost when it finally needs it. Hence such tasks do not get less processor time than the tasks that are constantly running.


The End