

# Operating System Week 3 - CPU Scheduling, Multi-level Feedback, Lottery Scheduling

Korawit Orkphol, D.Eng



# Scheduling : Introduction (1)

- By now low-level mechanisms of running processes should be clear (e.g., context switching)
- Now we have to understand the high-level policies that an OS scheduler employs.
- By presenting a series of scheduling policies (sometimes called disciplines) that various smart and hard-working people have developed over the years.
- The origins of scheduling, in fact, predate computer systems; early approaches were taken from the field of operations management and applied to computers
- This reality should be no surprise: assembly lines and many other human endeavors (ความพยายาม, try hard to achieve a goal) also require scheduling, and many of the same concerns exist therein, including a laser-like desire for efficiency. And thus, our problem:



## Scheduling : Introduction (2)

### 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 (1)

- Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the workload.
- Determining the workload is a critical part of building policies, and the more you know about workload, the more fine-tuned your policy can be.
- The workload assumptions we make here are mostly unrealistic, but that is alright (for now), because we will relax them as we go, and eventually develop what we will refer to as a fully-operational scheduling discipline.



## Workload Assumptions (2)

- We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:
  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., they perform no I/O)
  5. The run-time of each job is known.



# Scheduling Metrics (1)

- Beyond making workload assumptions, we also need one more thing to enable us to compare different scheduling policies: a scheduling metric.
- A metric is just something that we use to measure something, and there are a number of different metrics that make sense in scheduling.
- For now, however, let us also simplify our life by simply having a single metric: turnaround time. (เวลาแล้วเสร็จ)
- 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. More formally, the turnaround time  $T_{\text{turnaround}}$  is:  $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
- Because we have assumed that all jobs arrive at the same time, for now  $T_{\text{arrival}} = 0$  and hence  $T_{\text{turnaround}} = T_{\text{completion}}$ . This fact will change as we relax the aforementioned assumptions.



## Scheduling Metrics (2)

- You should note that turnaround time is a performance metric, which will be our primary focus this chapter.
- Another metric of interest is fairness, as measured (for example) by Jain's Fairness Index [J91].
- Performance and fairness are often at odds(ที่ขัดแย้ง) in scheduling; a scheduler.
- For example, may optimize performance but at the cost of preventing a few jobs from running, thus decreasing fairness. This conundrum(ปัญหาที่ยากซับซ้อน) shows us that life isn't always perfect.



# First In, First Out (FIFO) (1)

- 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.
- Let's do a quick example together. Imagine three jobs arrive in the system, A, B, and C, at roughly the same time ( $T_{\text{arrival}} = 0$ ).
- Because FIFO has to put some job first, let's assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C.
- Assume also that each job runs for 10 seconds. What will the average turnaround time be for these jobs?



## First In, First Out (FIFO) (2)

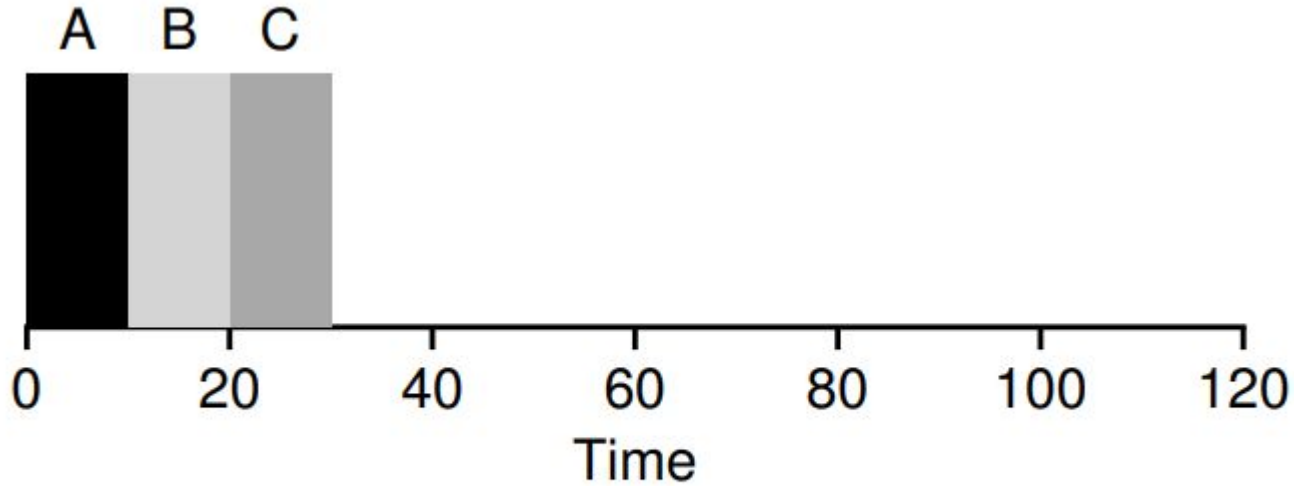


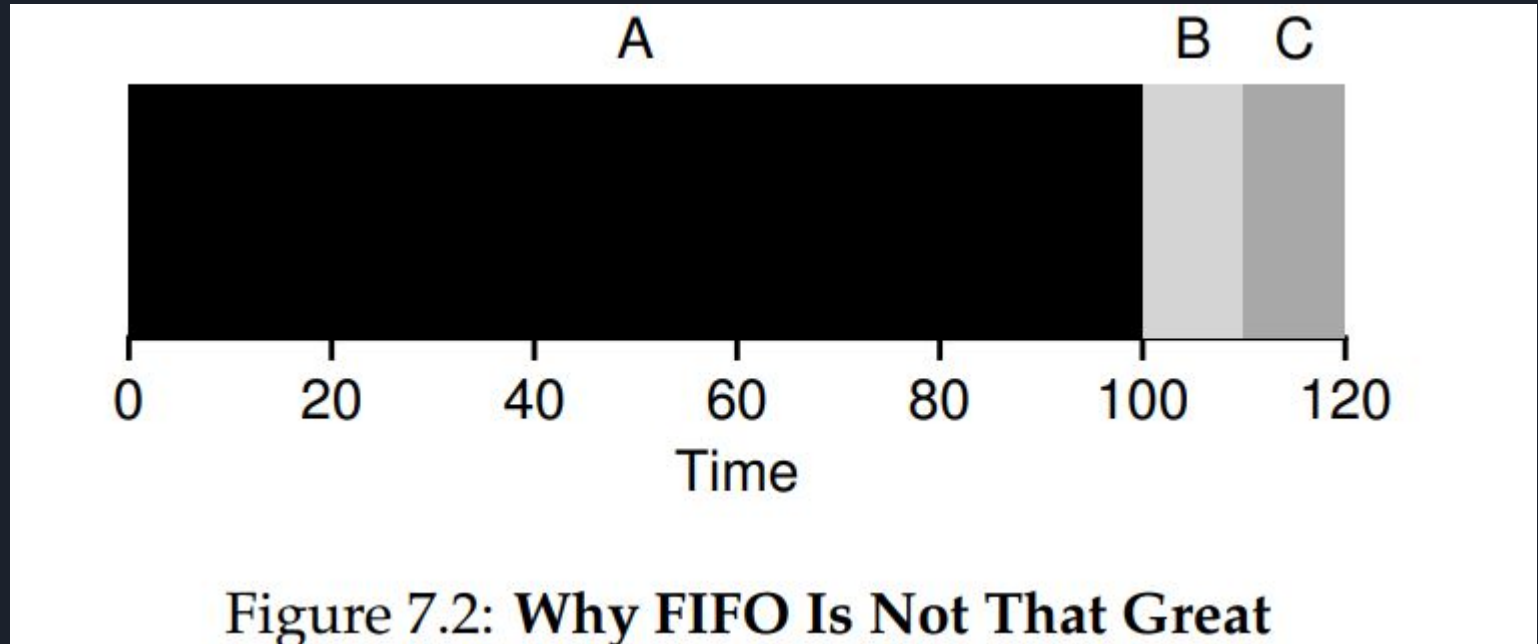
Figure 7.1: **FIFO Simple Example**



## First In, First Out (FIFO) (3)

- From Figure 7.1, you can see that A finished at 10, B at 20, and C at 30. Thus, the average turnaround time for the three jobs is simply  $(10+20+30)/3 = 20$ . Computing turnaround time is as easy as that.
- Now let's relax one of our assumptions. In particular, let's relax assumption 1, and thus no longer assume that each job runs for the same amount of time. How does FIFO perform now? What kind of workload could you construct to make FIFO perform poorly?
- let's do an example to show how jobs of different lengths can lead to trouble for FIFO scheduling. In particular, let's again assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each.

## First In, First Out (FIFO) (4)





## First In, First Out (FIFO) (5)

- As you can see in Figure 7.2, Job A runs first for the full 100 seconds before B or C even get a chance to run. Thus, the average turnaround time for the system is high: a painful 110 seconds ( $(100+110+120)/3 = 110$ ).
- This problem is generally referred to as the convoy effect [B+79], where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.
- This scheduling scenario might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with three carts full of provisions and a checkbook out; it's going to be a while . (ถ้าเป็นคุณจะทำอย่างไร)
- So what should we do? How can we develop a better algorithm to deal with our new reality of jobs that run for different amounts of time? Think about it first; then read on.



## Shortest Job First (SJF) (1)

- It turns out that a very simple approach solves this problem; in fact it is an idea stolen from operations research [C54,PV56] and applied to scheduling of jobs in computer systems.
- This new scheduling discipline is known as **Shortest Job First (SJF)**, and the name should be easy to remember because it describes the policy quite completely: it runs the shortest job first, then the next shortest, and so on.
- Let's take our example above but with SJF as our scheduling policy. Figure 7.3 shows the results of running A, B, and C. Hopefully the diagram makes it clear why SJF performs much better with regards to average turnaround time. Simply by running B and C before A, SJF reduces average turnaround from 110 seconds to 50 ( $(10+20+120)/3 = 50$ ), more than a factor of two improvement. (ดีกว่าเครื่องหนึ่งของอันที่แล้วอีก)

## Shortest Job First (SJF) (2)

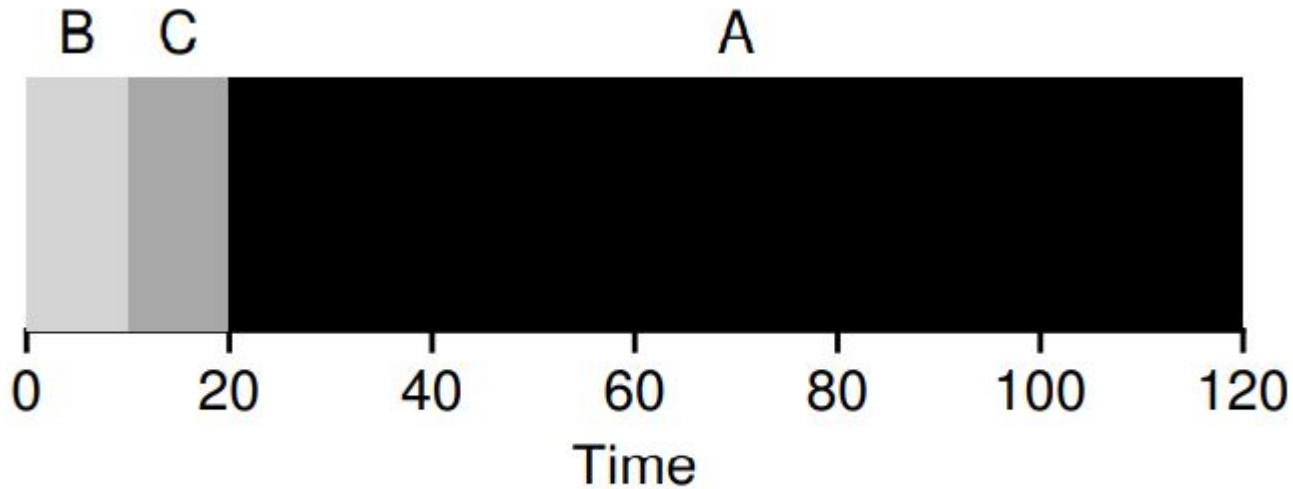


Figure 7.3: **SJF Simple Example**



## Shortest Job First (SJF) (3)

### TIP: THE PRINCIPLE OF SJF

Shortest Job First represents a general scheduling principle that can be applied to any system where the perceived turnaround time per customer (or, in our case, a job) matters. Think of any line you have waited in: if the establishment in question cares about customer satisfaction, it is likely they have taken SJF into account. For example, grocery stores commonly have a “ten-items-or-less” line to ensure that shoppers with only a few things to purchase don’t get stuck behind the family preparing for some upcoming nuclear winter.



## Shortest Job First (SJF) (4)

- In fact, given our assumptions about jobs all arriving at the same time, we could prove that SJF is indeed an optimal scheduling algorithm. However, you are in a systems class, not theory or operations research; no proofs are allowed.
- Thus we arrive upon a good approach to scheduling with SJF, but our assumptions are still fairly unrealistic. Let's relax another. In particular, we can target assumption 2, and now assume that jobs can arrive at any time instead of all at once. What problems does this lead to?
- Here we can illustrate the problem again with an example. This time, assume A arrives at  $t = 0$  and needs to run for 100 seconds, whereas B and C arrive at  $t = 10$  and each need to run for 10 seconds. With pure SJF, we'd get the schedule seen in Figure 7.4.



## Shortest Job First (SJF) (5)

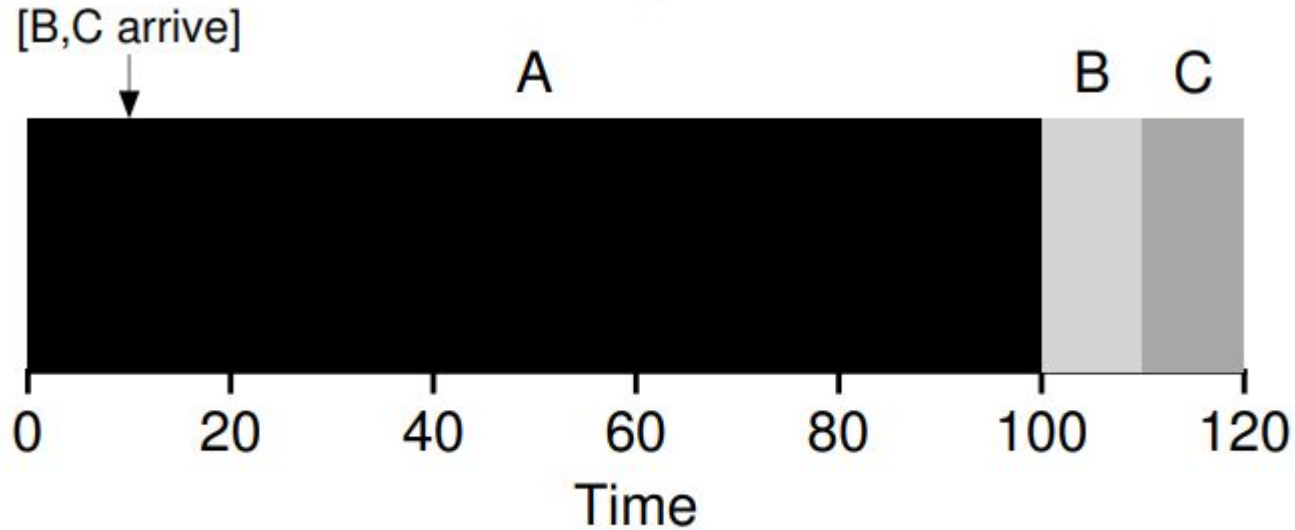


Figure 7.4: SJF With Late Arrivals From B and C



## Shortest Job First (SJF) (6)

- 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 ( $100 + (110 - 10) + (120 - 10) / 3$ ). What can a scheduler do?



# Shortest Time-to-Completion First (STCF) (1)

- To address this concern, we need to relax assumption 3 (that jobs must run to completion), so let's do that.
- We also need some machinery within the scheduler itself. As you might have guessed, given our previous discussion about timer interrupts and context switching, the scheduler can certainly do something else when B and C arrive: it can preempt job A and decide to run another job, perhaps continuing A later.
- SJF by our definition is a non-preemptive scheduler, and thus suffers from the problems described above.
- 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 [CK68].

## Shortest Time-to-Completion First (STCF) (2)

- Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.
- Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A's remaining time be scheduled. Figure 7.5 shows an example.

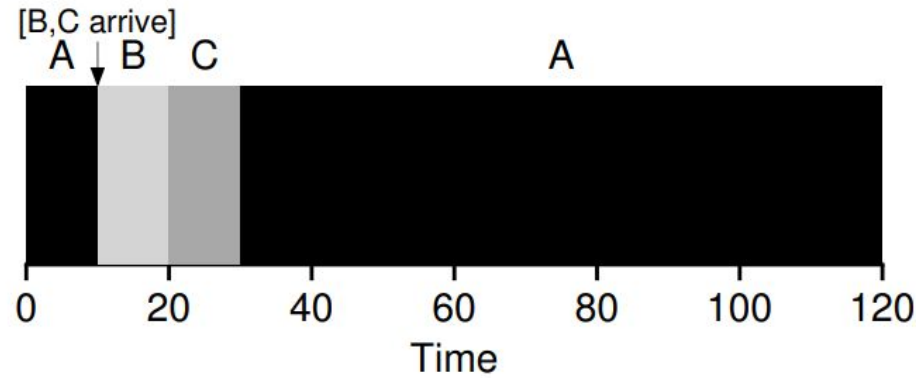


Figure 7.5: **STCF Simple Example**



## Shortest Time-to-Completion First (STCF) (3)

- The result is a much-improved average turnaround time: 50 seconds  $((120-0)+(20-10)+(30-10)/3)$ .
- And as before, 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.

### ASIDE: PREEMPTIVE SCHEDULERS

In the old days of batch computing, a number of **non-preemptive** schedulers were developed; such systems would run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, and quite willing to stop one process from running in order to run another. This implies that the scheduler employs the mechanisms we learned about previously; in particular, the scheduler can perform a **context switch**, stopping one running process temporarily and resuming (or starting) another.



# A New Metric: Response Time (1)

- Thus, if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy.
- In fact, for a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that.
- Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: response time.
- We define response time as the time from when the job arrives in a system to the first time it is scheduled. More formally:  $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$

## A New Metric: Response Time (2)

- For example, if we had the schedule from Figure 7.5 (with A arriving at time 0, and B and C at time 10), the response time of each job is as follows: 0 for job A, 0 for B, and 10 for C (average: 3.33).

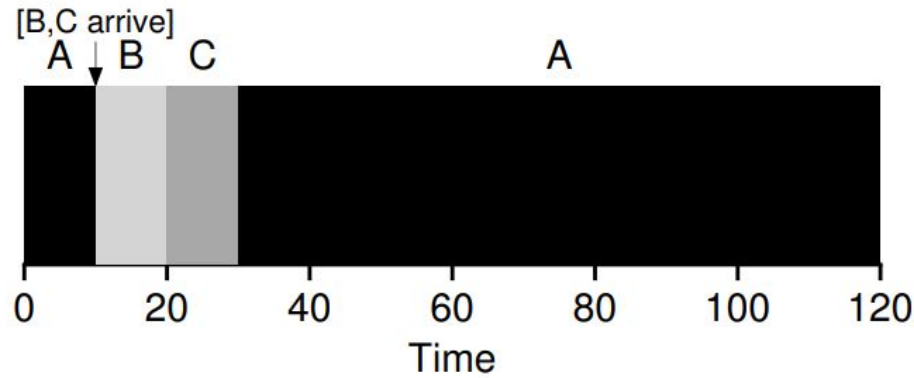


Figure 7.5: **STCF Simple Example**



## A New Metric: Response Time (3)

- As you might be thinking, STCF and related disciplines are not particularly good for response time.
- If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run in their entirety before being scheduled just once.
- While great for turnaround time, this approach is quite bad for response time and interactivity.
- Indeed, imagine sitting at a terminal, typing, and having to wait 10 seconds to see a response from the system just because some other job got scheduled in front of yours: not too pleasant.
- Thus, we are left with another problem: how can we build a scheduler that is sensitive to response time?





# Round Robin (1)

- To solve this problem, we will introduce a new scheduling algorithm, classically referred to as Round-Robin (RR) scheduling [K64].
- The basic idea is simple: instead of running jobs to completion, RR runs a job for a time slice (sometimes called a scheduling quantum) and then switches to the next job in the run queue.
- It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called time-slicing.
- Note that the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.



## Round Robin (2)

- To understand RR in more detail, let's look at an example. Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds.
- An SJF scheduler runs each job to completion before running another (Figure 7.6). In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly (Figure 7.7).
- The average response time of RR is:  $(0+1+2)/3 = 1$ ; for SJF, average response time is:  $(0+5+10)/3 = 5$ .

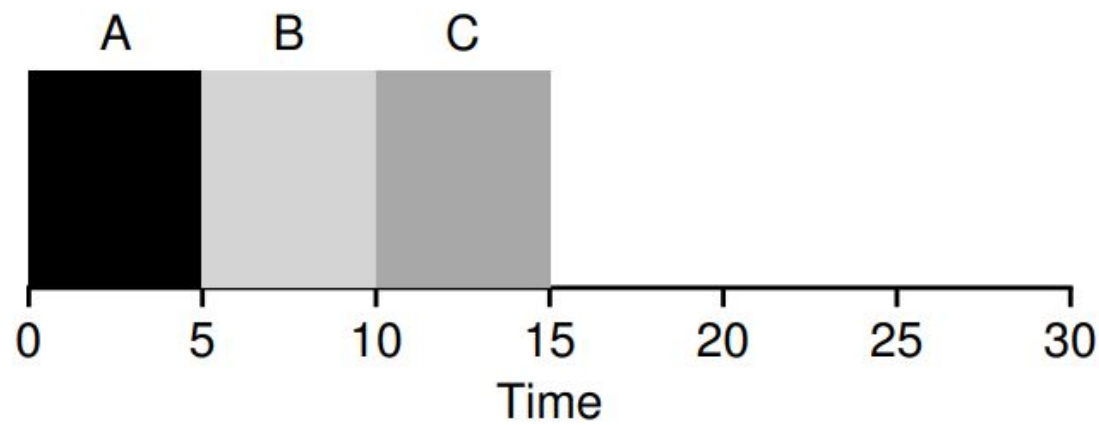


Figure 7.6: SJF Again (Bad for Response Time)

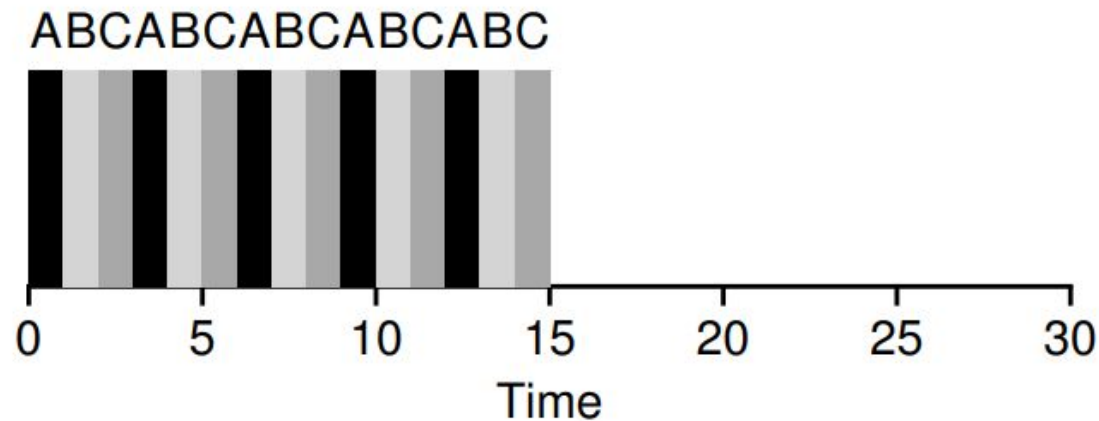


Figure 7.7: Round Robin (Good For Response Time)



## Round Robin (3)

- As you can see, the length of the time slice is critical for RR.
- The shorter it is, the better the performance of RR under the response-time metric.
- However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize (หักกลบลบล้าง) the cost of switching without making it so long that the system is no longer responsive.
- Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost [MB91].



## Round Robin (4)

- RR, with a reasonable time slice, is thus an excellent scheduler if response time is our only metric. But what about our old friend turnaround time?
- Let's look at our example above again. A, B, and C, each with running times of 5 seconds, arrive at the same time, and RR is the scheduler with a (long) 1-second time slice. We can see from the picture above that A finishes at 13, B at 14, and C at 15, for an average of 14. Pretty awful(แย่) งานทั้งสามเสร็จเกือบจะพร้อมๆกัน!
- It is not surprising, then, that RR is indeed one of the worst policies if turnaround time is our metric. Intuitively, this should make sense: what RR is doing is stretching out each job as long as it can, by only running each job for a short bit before moving to the next. Because turnaround time only cares about when jobs finish, RR is nearly pessimal(แย่สุด), even worse than simple FIFO in many cases.



## Round Robin (5)

- More generally, any policy (such as RR) that is fair, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time.
- Indeed, this is an inherent(โดยธรรมชาติ) trade-off: if you are willing to be unfair, you can run shorter jobs to completion, but at the cost of response time; if you instead value fairness, response time is lowered, but at the cost of turnaround time.
- This type of trade-off is common in systems; you can't have your cake and eat it too ได้อย่างก็ต้องเสียอย่าง ไม่สามารถจับปลาสองมือได้ การจะมีเค้กก็ต้องแลกกับการห้ามกินเค้กนั่นเอง
- We have developed two types of schedulers. The first type (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. And we still have two assumptions which need to be relaxed: assumption 4 (that jobs do no I/O), and assumption 5 (that the run-time of each job is known). Let's tackle those assumptions next.



# Incorporating I/O (1)

- First we will relax assumption 4 — of course all programs perform I/O. Imagine a program that didn't take any input: it would produce the same output each time. Imagine one without output: it is the proverbial tree falling in the forest, with no one to see it; it doesn't matter that it ran.
- A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is blocked waiting for I/O completion. 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. Thus, the scheduler should probably schedule another job on the CPU at that time
- The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state. Of course, it could even decide to run the job at that point. How should the OS treat each job?



## Incorporating I/O (2)

- To understand this issue better, let us assume we have two jobs, A and B, which each need 50 ms of CPU time. 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), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after.
- Assume we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? Clearly, just running one job and then the other without considering how to take I/O into account makes little sense.(Figure 7.8)



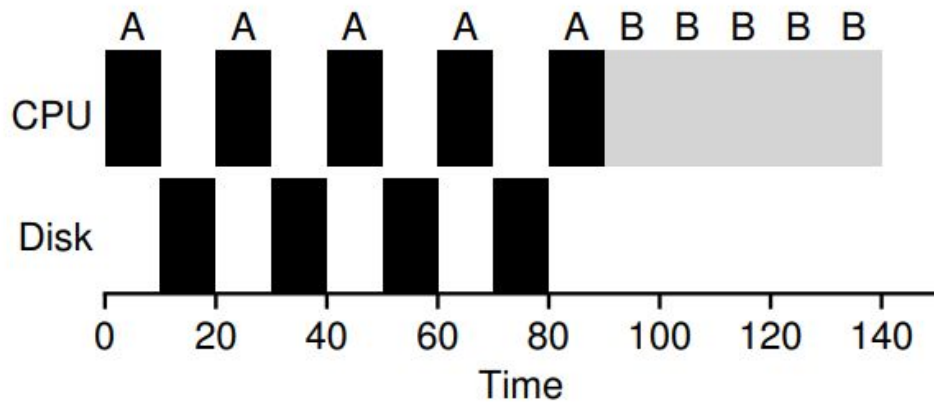


Figure 7.8: **Poor Use Of Resources**

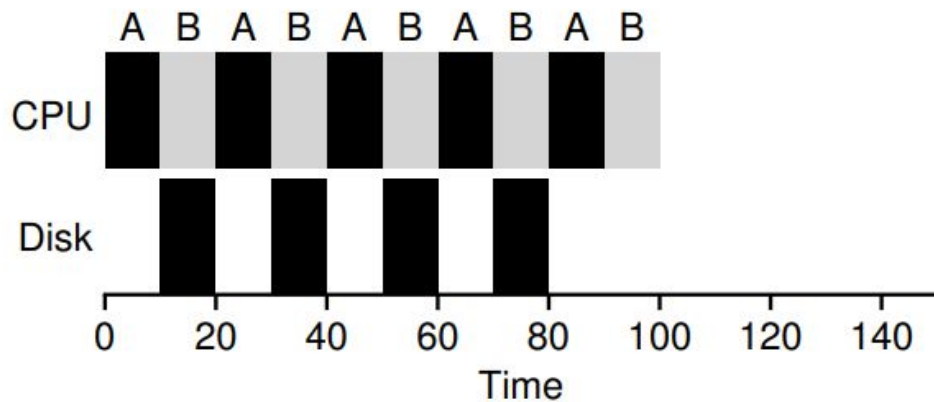


Figure 7.9: **Overlap Allows Better Use Of Resources**



## Incorporating I/O (3)

- 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. 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 (see Figure 7.9).
- 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.



# No More Oracle

- With a basic approach to I/O in place, we come to our final assumption: that the scheduler knows the length of each job. As we said before, this is likely the worst assumption we could make.
- In fact, in a general purpose OS (like the ones we care about), the OS usually knows very little about the length of each job.
- Thus, how can we build an approach that behaves like SJF/STCF without such a priori knowledge?
- Further, how can we incorporate some of the ideas we have seen with the RR scheduler so that response time is also quite good?



# Summary

- We have introduced the basic ideas behind scheduling and developed two families of approaches.
- The first runs the shortest job remaining and thus optimizes turnaround time; the second alternates between all jobs and thus optimizes response time. Both are bad where the other is good, alas, an inherent trade-off common in systems.
- We have also seen how we might incorporate I/O into the picture, but have still not solved the problem of the fundamental inability of the OS to see into the future.
- Shortly, we will see how to overcome this problem, by building a scheduler that uses the recent past to predict the future. This scheduler is known as the multi-level feedback queue, and it is the topic of the next chapter.

# The Multi-Level Feedback Queue





# The Multi-Level Feedback Queue (1)

- In this chapter, 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.
- The scheduler has subsequently been refined throughout the years to the implementations you will encounter in some modern systems.



## The Multi-Level Feedback Queue (2)

- 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? How can the scheduler learn, as the system runs, the characteristics of the jobs it is running, and thus make better scheduling decisions?



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?





# MLFQ: Basic Rules (1)

- 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], most approaches are 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, 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.



## MLFQ: Basic Rules (2)

- 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(สละ) 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: Basic Rules (3)

- If we were to put forth a picture of what the queues might look like at a given instant, we might see something like the following (Figure 8.1).
- 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 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; poor jobs C and D would never even get to run — an outrage
- 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 changes over time. And that, in a surprise only to those who are reading a chapter from this book for the first time, is exactly what we will do next.

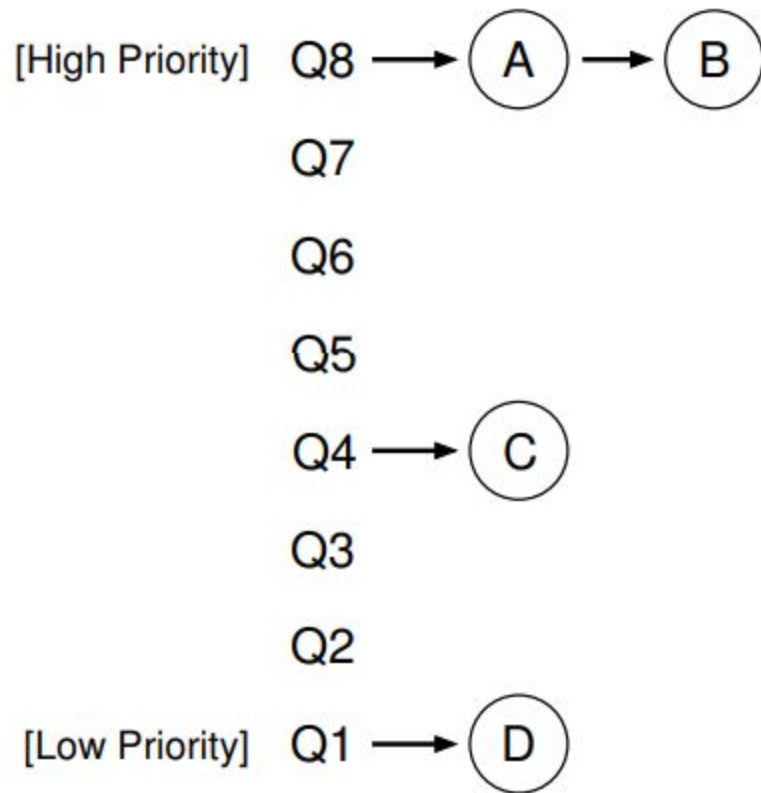


Figure 8.1: MLFQ Example



# Attempt #1: How To Change Priority (1)

- 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

- Let's look at some examples. First, we'll look at what happens when there has been a long running job in the system. Figure 8.2 shows what happens to this job over time in a three-queue scheduler.
- 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?

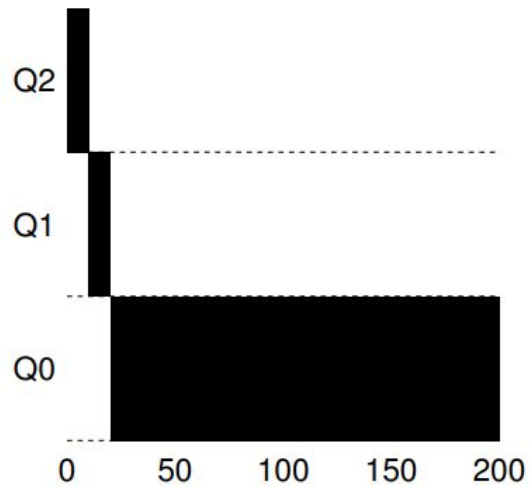


Figure 8.2: Long-running Job Over Time



## Example 2: Along Came A Short Job (1)

- 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 will happen? Will MLFQ approximate SJF 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 short (only 20 ms), B completes before reaching the bottom queue, in two time slices; then A resumes running (at low priority).



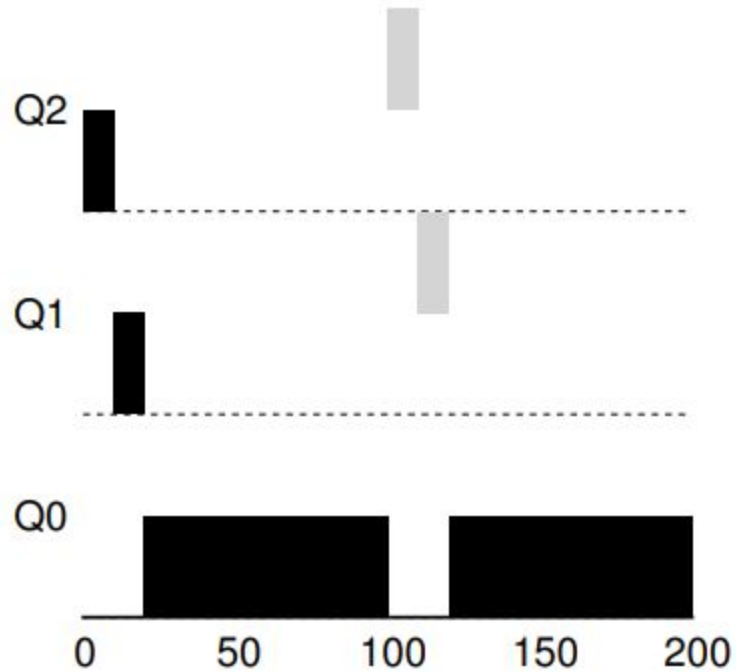


Figure 8.3: Along Came An Interactive Job



## Example 2: Along Came A Short Job (2)

- 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.

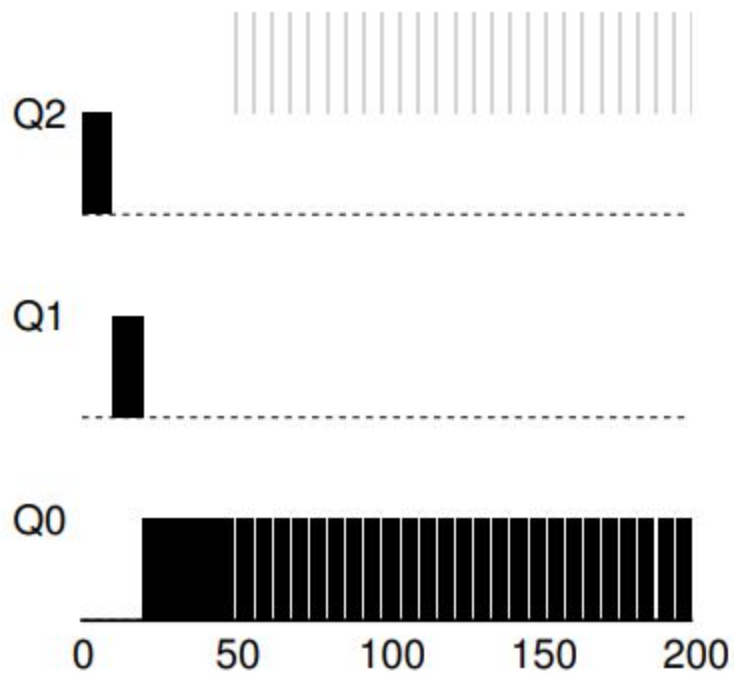


Figure 8.4: A Mixed I/O-intensive and CPU-intensive Workload



# Problems With Our Current MLFQ (1)

- We thus have a basic MLFQ. 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 serious flaws. Can you think of any? มีช่องโหว่?
- 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.
- 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.



## Problems With Our Current MLFQ (2)

- The algorithm we have described is susceptible(อ่อนไหว) 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 CPU time.
- When done right ทำเสร็จ 99% ของ timeslice (e.g., by running for 99% of a time slice before relinquishing the CPU), a job could nearly monopolize the CPU. ยึดครอง CPU
- Finally, a program may change its behavior over time; what was CPU bound may transition to a phase of interactivity. ต้องเปลี่ยนโปรแกรมเพื่อให้เป็นแบบ interactivity เพื่อได้รันมากขึ้น
- With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

### TIP: SCHEDULING MUST BE SECURE FROM ATTACK

You might think that a scheduling policy, whether inside the OS itself (as discussed herein), or in a broader context (e.g., in a distributed storage system's I/O request handling [Y+18]), is not a **security** concern, but in increasingly many cases, it is exactly that. Consider the modern datacenter, in which users from around the world share CPUs, memories, networks, and storage systems; without care in policy design and enforcement, a single user may be able to adversely harm others and gain advantage for itself. Thus, scheduling policy forms an important part of the security of a system, and should be carefully constructed.



## Attempt #2: The Priority Boost (1)

- 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.
- 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 (2)

- 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.
- 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.

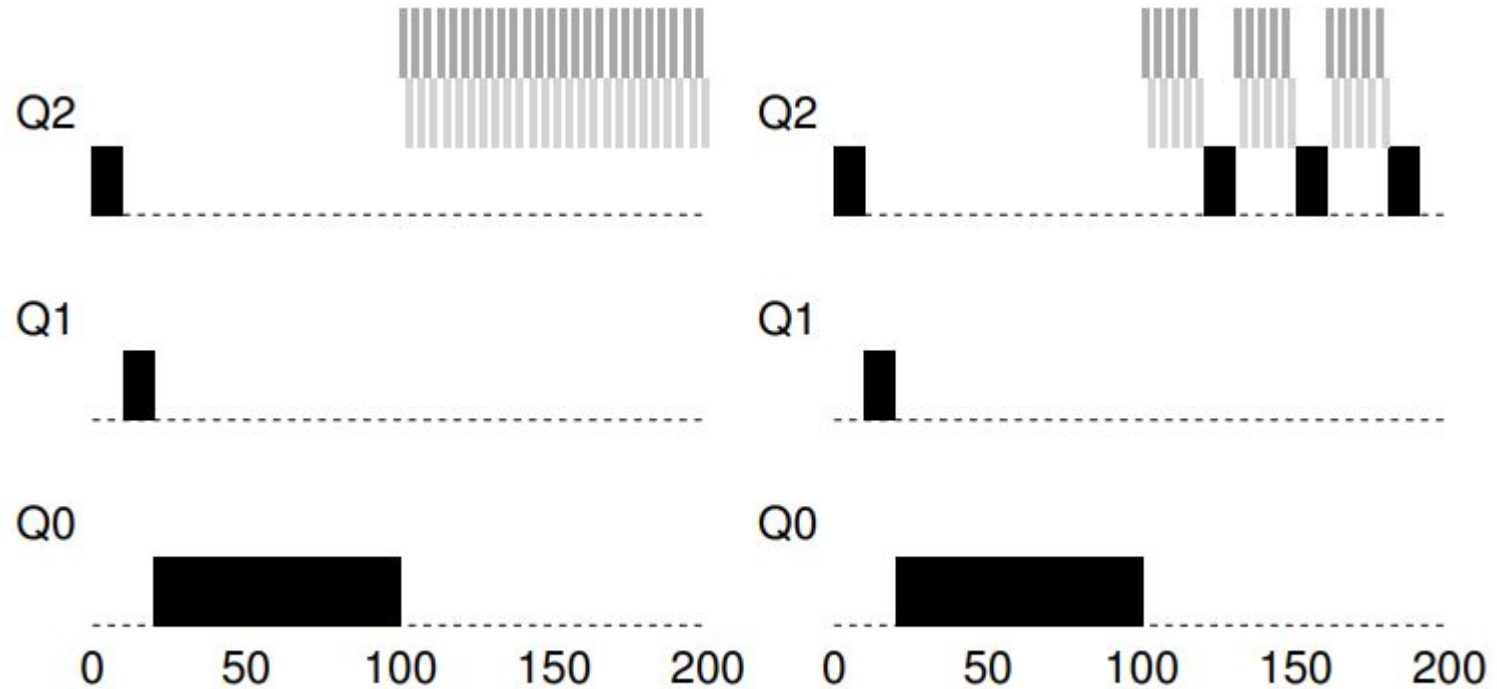


Figure 8.5: Without (Left) and With (Right) Priority Boost



## Attempt #3: Better Accounting (1)

- 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?
- 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 (2)

- Let's look 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 share of the CPU.

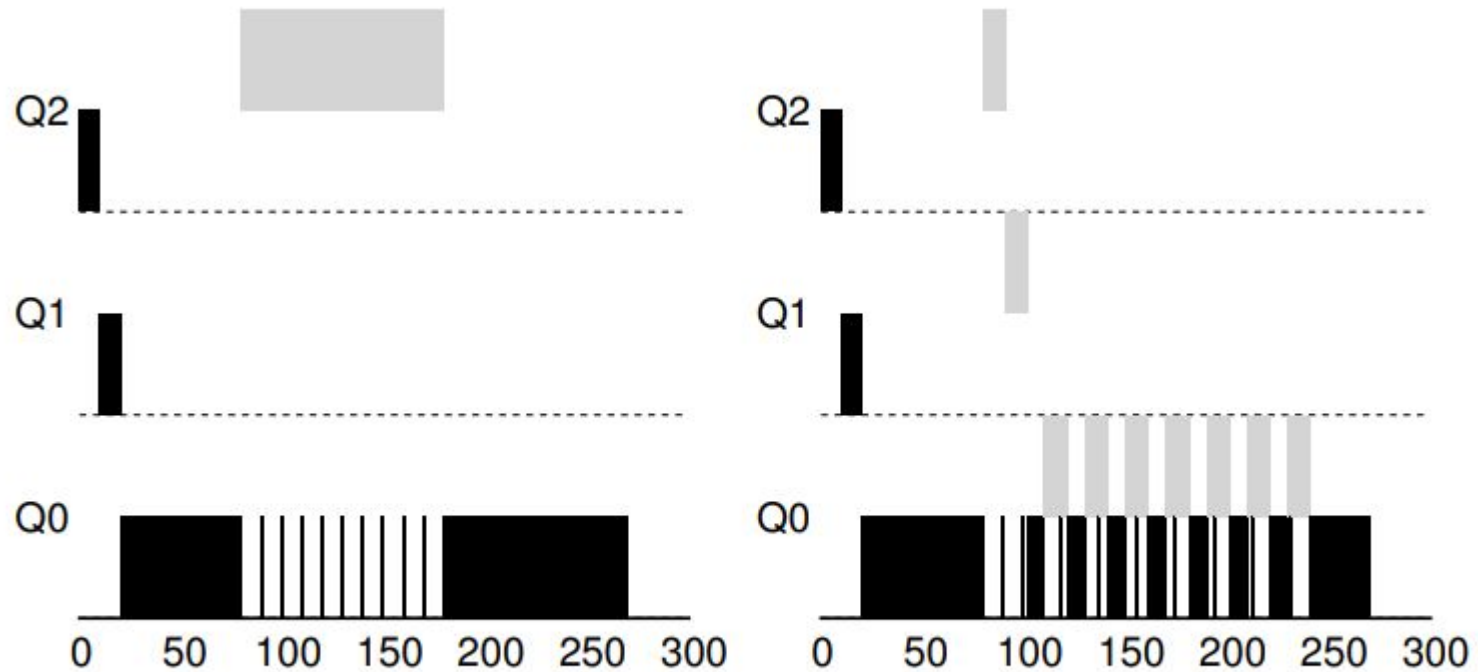


Figure 8.6: Without (Left) and With (Right) Gaming Tolerance



# Tuning MLFQ And Other Issues (1)

- 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.

### TIP: AVOID VOO-DOO CONSTANTS (OUSTERHOUT'S LAW)

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 defaults work well in the field. This tip brought to you by our old OS professor, John Ousterhout, and hence we call it **Ousterhout's Law**.



## Tuning MLFQ And Other Issues (2)

- For example, most MLFQ variants allow for varying time-slice length across different queues. 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 fewer milliseconds). The low-priority queues, in contrast, contain long-running jobs that are CPU-bound; hence, longer time slices work well (e.g., 100s of ms). Figure 8.7 shows an example in which two jobs run for 20 ms at the highest queue (with a 10-ms time slice), 40 ms in the middle (20-ms time slice), and with a 40-ms time slice at the lowest.



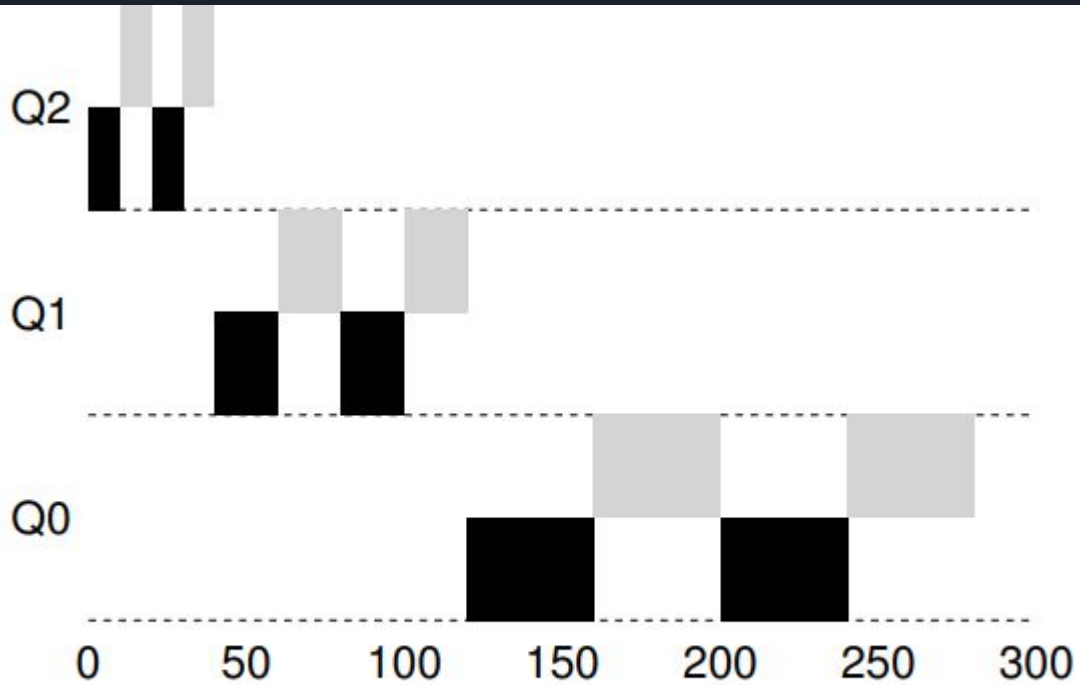


Figure 8.7: Lower Priority, Longer Quanta



## Tuning MLFQ And Other Issues (3)

- The Solaris MLFQ implementation — the Time-Sharing scheduling class, or 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.



## Tuning MLFQ And Other Issues (4)

- 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 Epema's paper for an excellent overview of such decay-usage algorithms and their properties [E95].
- 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 obtain the highest levels of priority in the system. Some systems also allow some user advice to help set priorities; for example, by using the command-line utility nice you can increase or decrease the priority of a job (somewhat) and thus increase or decrease its chances of running at any given time. See the man page for more.

### 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., informed prefetching and caching [P+95]).



# MLFQ: Summary (1)

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. History is its guide: pay attention to how jobs behave over time and treat them accordingly. The refined set of MLFQ 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 round-robin fashion using the time slice (quantum length) of the given queue.
- 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: Summary (2)

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

# Lottery Scheduling





# Scheduling: Proportional Share

- In this chapter, we'll examine a different type of scheduler known as a proportional-share scheduler, also sometimes referred to as a fair-share scheduler.
- Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.
- An excellent early example of proportional-share scheduling is found in research by Waldspurger and Weihl [WW94], and is known as lottery scheduling;
- however, the idea is certainly older [KL88]. The basic idea is quite simple: every so often, hold a lottery to determine which process should get to run next; processes that should run more often should be given more chances to win the lottery. Easy, no? Now, onto the details! But not before our crux:






## CRUX: HOW TO SHARE THE CPU PROPORTIONALLY

How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?



# Basic Concept: Tickets Represent Your Share (1)

- Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process (or user or whatever) should receive. The percent of tickets that a process has represents its share of the system resource in question.
- Let's look at an example. Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%.
- Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice).
- Holding a lottery is straightforward: the scheduler must know how many total tickets there are (in our example, there are 100).
- The scheduler then picks a winning ticket, which is a number from 0 to 99.
- Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.



81  
Here is an example output of a lottery scheduler's winning tickets:

63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43 0 49 12


Here is the resulting schedule:

A		A	A		A	A	A	A	A	A		A		A	A	A	A	A	A
	B			B							B		B						



## Basic Concept: Tickets Represent Your Share (2)

- As you can see from the example, the use of randomness in lottery scheduling leads to a probabilistic correctness in meeting the desired proportion, but no guarantee.
- In our example above, B only gets to run 4 out of 20 time slices (20%), instead of the desired 25% allocation. *However, the longer these two jobs compete, the more likely they are to achieve the desired percentages.*



### TIP: USE TICKETS TO REPRESENT SHARES

One of the most powerful (and basic) mechanisms in the design of lottery (and stride) scheduling is that of the **ticket**. The ticket is used to represent a process's share of the CPU in these examples, but can be applied much more broadly. For example, in more recent work on virtual memory management for hypervisors, Waldspurger shows how tickets can be used to represent a guest operating system's share of memory [W02]. Thus, if you are ever in need of a mechanism to represent a proportion of ownership, this concept just might be ... (wait for it) ... the ticket.



# Ticket Mechanisms (1)

- Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways.
- One way is with the concept of ticket currency.
- Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value
- For example, assume users A and B have each been given 100 tickets.
- User A is running two jobs, A1 and A2, and gives them each 500 tickets (out of 1000 total) in A's currency. User B is running only 1 job and gives it 10 tickets (out of 10 total).
- The system converts A1's and A2's allocation from 500 each in A's currency to 50 each in the global currency;
- similarly, B1's 10 tickets is converted to 100 tickets. The lottery is then held over the global ticket currency (200 total) to determine which job runs.



## Ticket Mechanisms (2)

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
        -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```



## Ticket Mechanisms (3)

- Another useful mechanism is *ticket transfer*. With transfers, a process can temporarily hand off its tickets to another process.
- This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client's behalf.
- To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client's request. When finished, the server then transfers the tickets back to the client and all is as before.



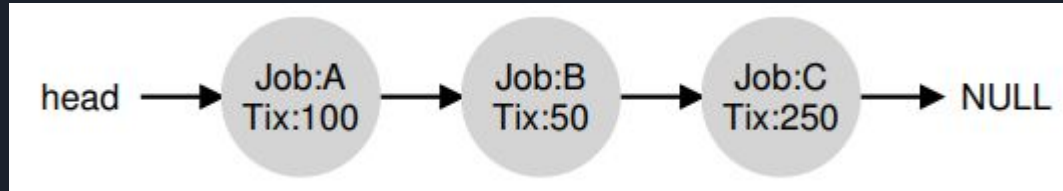


## Ticket Mechanisms (4)

- Finally, *ticket inflation* can sometimes be a useful technique.
- With inflation, a process can temporarily raise or lower the number of tickets it owns. Of course, in a competitive scenario with processes that do not trust one another, this makes little sense; one greedy process could give itself a vast number of tickets and take over the machine.
- Rather, inflation can be applied in an environment where a group of processes trust one another; in such a case, if any one process knows it needs more CPU time, it can boost its ticket value as a way to reflect that need to the system, all without communicating with any other processes.

# Implementation (1)

- Probably the most amazing thing about lottery scheduling is the simplicity of its implementation. All you need is a good random number generator to pick the winning ticket, a data structure to track the processes of the system (e.g., a list), and the total number of tickets.
- Let's assume we keep the processes in a list. Here is an example comprised of three processes, A, B, and C, each with some number of tickets





## Implementation (2)

- To make a scheduling decision, we first have to pick a random number (the winner) from the total number of tickets (400)<sup>2</sup> Let's say we pick the number 300. Then, we simply traverse the list, with a simple counter used to help us find the winner (Figure 9.1).
- The code walks the list of processes, adding each ticket value to *counter* until the value exceeds *winner*.
- Once that is the case, the current list element is the winner.
- With our example of the winning ticket being 300, the following takes place.
- First, *counter* is incremented to 100 to account for A's tickets; because 100 is less than 300, the loop continues. Then *counter* would be updated to 150 (B's tickets), still less than 300 and thus again we continue.
- Finally, *counter* is updated to 400 (clearly greater than 300), and thus we break out of the loop with *current* pointing at C (the winner)

```
1 // counter: used to track if we've found the winner yet
2 int counter = 0;
3
4 // winner: use some call to a random number generator to
5 //           get a value, between 0 and the total # of tickets
6 int winner = getrandom(0, totaltickets);
7
8 // current: use this to walk through the list of jobs
9 node_t *current = head;
10 while (current) {
11     counter = counter + current->tickets;
12     if (counter > winner)
13         break; // found the winner
14     current = current->next;
15 }
16 // 'current' is the winner: schedule it...
```



## Implementation (3)

- To make this process most efficient, it might generally be best to organize the list in sorted order, from the highest number of tickets to the lowest.
- The ordering does not affect the correctness of the algorithm; however, it does ensure in general that the fewest number of list iterations are taken, especially if there are a few processes that possess most of the tickets.



# An Example (1)

- To make the dynamics of lottery scheduling more understandable, we now perform a brief study of the completion time of two jobs competing against one another, each with the same number of tickets (100) and same run time ( $R$ , which we will vary).
- In this scenario, we'd like for each job to finish at roughly the same time, but due to the randomness of lottery scheduling, sometimes one job finishes before the other. To quantify this difference, we define a simple fairness metric,  $F$  which is simply the time the first job completes divided by the time that the second job completes.
- For example, if  $R = 10$ , and the first job finishes at time 10 (and the second job at 20),  $F = 10/20 = 0.5$ .
- When both jobs finish at nearly the same time,  $F$  will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve  $F = 1$ .

## An Example (2)

- Figure 9.2 plots the average fairness as the length of the two jobs (R) is varied from 1 to 1000 over thirty trials (results are generated via the simulator provided at the end of the chapter).
- As you can see from the graph, when the job length is not very long, average fairness can be quite low.
- Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired fair outcome.

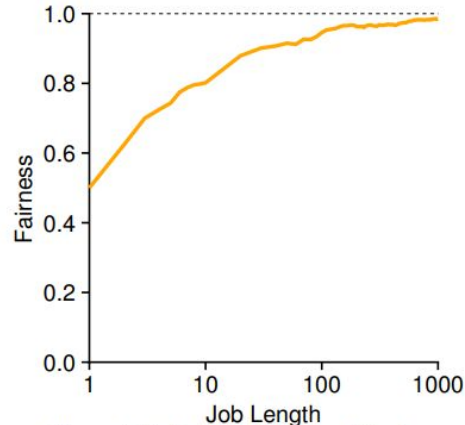


Figure 9.2: Lottery Fairness Study



# How To Assign Tickets?

- One problem we have not addressed with lottery scheduling is: how to assign tickets to jobs? This problem is a tough one, because of course how the system behaves is strongly dependent on how tickets are allocated.
- One approach is to assume that the users know best; in such a case, each user is handed some number of tickets, and a user can allocate tickets to any jobs they run as desired. However, this solution is a non solution: it really doesn't tell you what to do. Thus, given a set of jobs, the "ticket-assignment problem" remains open.





# Stride Scheduling (1)

- You might also be wondering: why use randomness at all? As we saw above, while randomness gets us a simple (and approximately correct) scheduler, it occasionally will not deliver the exact right proportions, especially over short time scales.
- For this reason, Waldspurger invented stride scheduling, a deterministic fair-share scheduler [W95].



## Stride Scheduling (2)

- Stride scheduling is also straightforward.
- Each job in the system has a stride, which is inverse in proportion to the number of tickets it has.
- In our example above, with jobs A, B, and C, with 100, 50, and 250 tickets, respectively, we can compute the stride of each by dividing some large number by the number of tickets each process has been assigned.
- For example, if we divide 10,000 by each of those ticket values, we obtain the following stride values for A, B, and C: 100, 200, and 40.
- We call this value the stride of each process; every time a process runs, we will increment a counter for it (called its pass value) by its stride to track its global progress.



## Stride Scheduling (3)


- The scheduler then uses the stride and pass to determine which process should run next. The basic idea is simple: at any given time, pick the process to run that has the lowest pass value so far; when you run a process, increment its pass counter by its stride. A pseudocode implementation is provided by Waldspurger [W95]:

```
curr = remove_min(queue);    // pick client with min pass
schedule(curr);              // run for quantum
curr->pass += curr->stride;    // update pass using stride
insert(queue, curr);         // return curr to queue
```



## Stride Scheduling (4)

- In our example, we start with three processes (A, B, and C), with stride values of 100, 200, and 40, and all with pass values initially at 0.
- Thus, at first, any of the processes might run, as their pass values are equally low. Assume we pick A (arbitrarily; any of the processes with equal low pass values can be chosen).
- A runs; when finished with the time slice, we update its pass value to 100. Then we run B, whose pass value is then set to 200.
- Finally, we run C, whose pass value is incremented to 40. At this point, the algorithm will pick the lowest pass value, which is C's, and run it, updating its pass to 80 (C's stride is 40, as you recall).
- Then C will run again (still the lowest pass value), raising its pass to 120.
- A will run now, updating its pass to 200 (now equal to B's). Then C will run twice more, updating its pass to 160 then 200. At this point, all pass values are equal again, and the process will repeat, ad infinitum.
- Figure 9.3 traces the behavior of the scheduler over time.



Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

Figure 9.3: Stride Scheduling: A Trace



## Stride Scheduling (5)

- As we can see from the figure, C ran five times, A twice, and B just once, exactly in proportion to their ticket values of 250, 100, and 50.
- Lottery scheduling achieves the proportions probabilistically over time; stride scheduling gets them exactly right at the end of each scheduling cycle.
- So you might be wondering: given the precision of stride scheduling, why use lottery scheduling at all? Well, lottery scheduling has one nice property that stride scheduling does not: no global state.
- Imagine a new job enters in the middle of our stride scheduling example above; what should its pass value be? Should it be set to 0? If so, it will monopolize the CPU.
- With lottery scheduling, there is no global state per process; we simply add a new process with whatever tickets it has, update the single global variable to track how many total tickets we have, and go from there. In this way, lottery makes it much easier to incorporate new processes in a sensible manner.



# The Linux Completely Fair Scheduler (CFS)

- Despite these earlier works in fair-share scheduling, the current Linux approach achieves similar goals in an alternate manner.
- The scheduler, entitled the Completely Fair Scheduler (or CFS) [J09], implements fairshare scheduling, but does so in a highly efficient and scalable manner
- To achieve its efficiency goals, CFS aims to spend very little time making scheduling decisions, through both its inherent design and its clever use of data structures well-suited to the task.
- Recent studies have shown that scheduler efficiency is surprisingly important; specifically, in a study of Google data centers, Kanev et al. show that even after aggressive optimization, scheduling uses about 5% of overall datacenter CPU time.
- Reducing that overhead as much as possible is thus a key goal in modern scheduler architecture.



# Basic Operation (1)

- Whereas most schedulers are based around the concept of a fixed time slice, CFS operates a bit differently. Its goal is simple: to fairly divide a CPU evenly among all competing processes. It does so through a simple counting-based technique known as **virtual runtime (vruntime)**.
- As each process runs, it accumulates vruntime. In the most basic case, each process's *vruntime* increases at the same rate, in proportion with physical (real) time. When a scheduling decision occurs, CFS will pick the process with the lowest *vruntime* to run next.
- This raises a question: how does the scheduler know when to stop the currently running process, and run the next one? The tension here is clear: if CFS switches too often, fairness is increased, as CFS will ensure that each process receives its share of CPU even over miniscule time windows, but at the cost of performance (too much context switching); if CFS switches less often, performance is increased (reduced context switching), but at the cost of near-term fairness.





## Basic Operation (2)

- CFS manages this tension through various control parameters. The first is **sched\_latency**. CFS uses this value to determine how long one process should run before considering a switch (effectively determining its time slice but in a dynamic fashion). A typical **sched\_latency** value is 48 (milliseconds)
- CFS divides this value by the number ( $n$ ) of processes running on the CPU to determine the time slice for a process, and thus ensures that over this period of time, CFS will be completely fair.
- For example, if there are  $n = 4$  processes running, CFS divides the value of sched latency by  $n$  to arrive at a per-process time slice of 12 ms.
- CFS then schedules the first job and runs it until it has used 12 ms of (virtual) runtime, and then checks to see if there is a job with lower vruntime to run instead.
- In this case, there is, and CFS would switch to one of the three other jobs, and so forth. Figure 9.4 shows an example where the four jobs (A, B, C, D) each run for two time slices in this fashion; two of them (C, D) then complete, leaving just two remaining, which then each run for 24 ms in round-robin fashion.

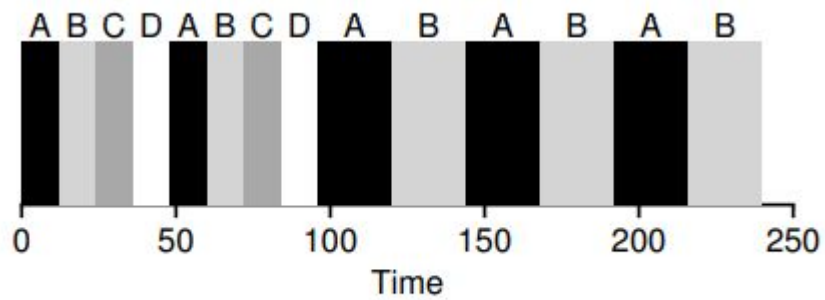


Figure 9.4: CFS Simple Example



## Basic Operation (3)

- But what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches? Good question! And the answer is yes.
- To address this issue, CFS adds another parameter, **min\_granularity**, which is usually set to a value like 6 ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overhead.
- For example, if there are ten processes running, our original calculation would divide sched latency by ten to determine the time slice (result: 4.8 ms). However, because of min granularity, CFS will set the time slice of each process to 6 ms instead.
- Although CFS won’t (quite) be perfectly fair over the target scheduling latency (sched latency) of 48 ms, it will be close, while still achieving high CPU efficiency.



## Basic Operation (4)

- Note that CFS utilizes a **periodic timer interrupt**, which means it can only make decisions at fixed time intervals. This interrupt goes off frequently (e.g., every 1 ms), giving CFS a chance to wake up and determine if the current job has reached the end of its run.
- If a job has a time slice that is not a perfect multiple of the timer interrupt interval, that is OK; CFS tracks vruntime precisely, which means that over the long haul, it will eventually approximate ideal sharing of the CPU.

# Weighting (Niceness) (1)

- CFS also enables controls over process priority, enabling users or administrators to give some processes a higher share of the CPU. It does this not with tickets, but through a classic UNIX mechanism known as the nice level of a process. The nice parameter can be set anywhere from -20 to +19 for a process, with a default of 0.
- Positive nice values imply lower priority and negative values imply higher priority; when you're too nice, you just don't get as much (scheduling) attention, alas. CFS maps the nice value of each process to a weight, as shown here:

```
static const int prio_to_weight[40] = {  
    /* -20 */ 88761, 71755, 56483, 46273, 36291,  
    /* -15 */ 29154, 23254, 18705, 14949, 11916,  
    /* -10 */ 9548, 7620, 6100, 4904, 3906,  
    /* -5 */ 3121, 2501, 1991, 1586, 1277,  
    /* 0 */ 1024, 820, 655, 526, 423,  
    /* 5 */ 335, 272, 215, 172, 137,  
    /* 10 */ 110, 87, 70, 56, 45,  
    /* 15 */ 36, 29, 23, 18, 15,  
};
```



## Weighting (Niceness) (2)

- These weights allow us to compute the effective time slice of each process (as we did before), but now accounting for their priority differences. The formula used to do so is as follows, assuming  $n$  processes:

$$\text{time\_slice}_k = \frac{\text{weight}_k}{\sum_{i=0}^{n-1} \text{weight}_i} \cdot \text{sched\_latency} \quad (9.1)$$

- Let's do an example to see how this works. Assume there are two jobs, A and B.
- A, because it's our most precious job, is given a higher priority by assigning it a nice value of -5; B, because we hate it, just has the default priority (nice value equal to 0). This means  $\text{weight}_A$  (from the table) is 3121, whereas  $\text{weight}_B$  is 1024. If you then compute the time slice of each job, you'll find that A's time slice is about 3/4 of  $\text{sched\_latency}$  (hence, 36 ms), and B's about 1/4 (hence, 12 ms).

## Weighting (Niceness) (3)

- In addition to generalizing the time slice calculation, the way CFS calculates vruntime must also be adapted.
- Here is the new formula, which takes the actual run time that process  $i$  has accrued ( $runtime_i$ ) and scales it inversely by the weight of the process, by dividing the default weight of 1024 ( $weight_0$ ) by its weight,  $weight_i$ .
- In our running example, A's vruntime will accumulate at one-third the rate of B's.

$$vruntime_i = vruntime_i + \frac{weight_0}{weight_i} \cdot runtime_i \quad (9.2)$$

$vruntime_A = vruntime_A + 1024/3121 \times runtime_A$  กล่าวคือ เวลาของ A จะสะสม 1/3 ของเวลา B นั่นเอง A ก็จะทำป๋อยขึ้นเพราะได้รับ weight (nice) -5 นั่นเอง



## Weighting (Niceness) (4)

- One smart aspect of the construction of the table of weights above is that the table preserves CPU proportionality ratios when the difference in nice values is constant.
- For example, if process A instead had a nice value of 5 (not -5), and process B had a nice value of 10 (not 0), CFS would schedule them in exactly the same manner as before. Run through the math yourself to see why





# Using Red-Black Trees (1)

- One major focus of CFS is efficiency, as stated above.
- For a scheduler, there are many facets of efficiency, but one of them is as simple as this: when the scheduler has to find the next job to run, it should do so as quickly as possible.
- Simple data structures like lists don't scale: modern systems sometimes are comprised of 1000s of processes, and thus searching through a long-list every so many milliseconds is wasteful.
- CFS addresses this by keeping processes in a red-black tree [B72].
- A red-black tree is one of many types of balanced trees; in contrast to a simple binary tree (which can degenerate to list-like performance under worst-case insertion patterns), balanced trees do a little extra work to maintain low depths, and thus ensure that operations are logarithmic (and not linear) in time.



## Using Red-Black Trees (2)

- CFS does not keep all process in this structure; rather, only running (or runnable) processes are kept therein. If a process goes to sleep (say, waiting on an I/O to complete, or for a network packet to arrive), it is removed from the tree and kept track of elsewhere.
- Let's look at an example to make this more clear. Assume there are ten jobs, and that they have the following values of vruntime: 1, 5, 9, 10, 14, 18, 17, 21, 22, and 24. If we kept these jobs in an ordered list, finding the next job to run would be simple: just remove the first element.
- However, when placing that job back into the list (in order), we would have to scan the list, looking for the right spot to insert it, an  $O(n)$  operation. Any search is also quite inefficient, also taking linear time on average. (ลิงค์ลิสต์ซ้ำ)
- Keeping the same values in a red-black tree makes most operations more efficient, as depicted in Figure 9.5. Processes are ordered in the tree by vruntime, and most operations (such as insertion and deletion) are logarithmic in time, i.e.,  $O(\log n)$ . When  $n$  is in the thousands, logarithmic is noticeably more efficient than linear. เก็บลงต้นไม้สมดุลดีกว่า

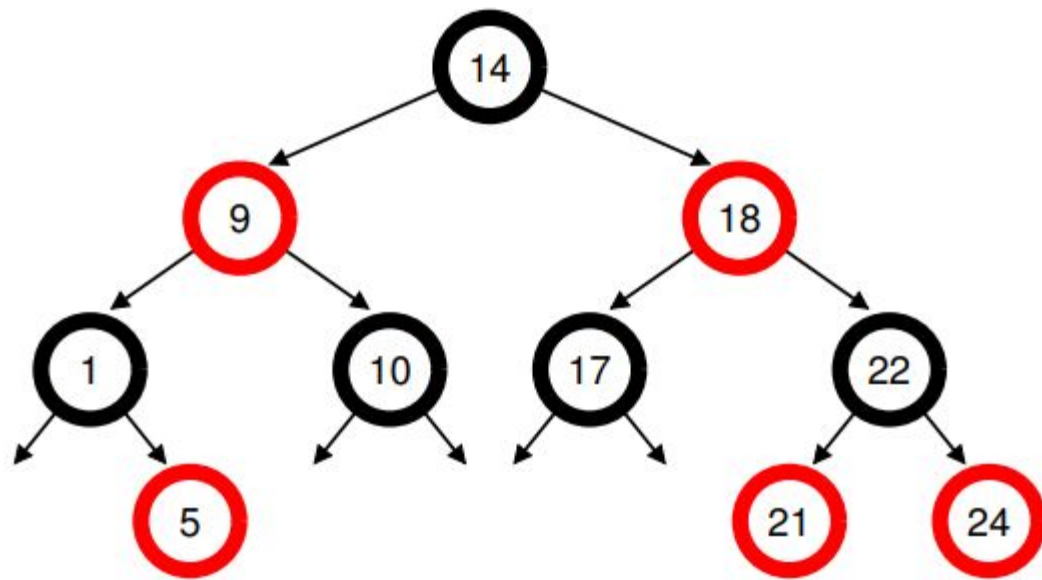


Figure 9.5: CFS Red-Black Tree



### TIP: USE EFFICIENT DATA STRUCTURES WHEN APPROPRIATE

In many cases, a list will do. In many cases, it will not. Knowing which data structure to use when is a hallmark of good engineering. In the case discussed herein, simple lists found in earlier schedulers simply do not work well on modern systems, particular in the heavily loaded servers found in datacenters. Such systems contain thousands of active processes; searching through a long list to find the next job to run on each core every few milliseconds would waste precious CPU cycles. A better structure was needed, and CFS provided one by adding an excellent implementation of a red-black tree. More generally, when picking a data structure for a system you are building, carefully consider its access patterns and its frequency of usage; by understanding these, you will be able to implement the right structure for the task at hand.



# Dealing With I/O And Sleeping Processes

- One problem with picking the lowest vruntime to run next arises with jobs that have gone to sleep for a long period of time. Imagine two processes, A and B, one of which (A) runs continuously, and the other (B) which has gone to sleep for a long period of time (say, 10 seconds). When B wakes up, its vruntime will be 10 seconds behind A's, and thus (if we're not careful), B will now monopolize the CPU for the next 10 seconds while it catches up, effectively starving A.
- CFS handles this case by altering the vruntime of a job when it wakes up. Specifically, CFS sets the vruntime of that job to the minimum value found in the tree (remember, the tree only contains running jobs) [B+18].
- In this way, CFS avoids starvation, but not without a cost: jobs that sleep for short periods of time frequently do not ever get their fair share of the CPU [AC97].



## Other CFS Fun

- CFS has many other features, too many to discuss at this point in the book. It includes numerous heuristics to improve cache performance, has strategies for handling multiple CPUs effectively (as discussed later in the book), can schedule across large groups of processes (instead of treating each process as an independent entity), and many other interesting features. Read recent research, starting with Bouron [B+18], to learn more



## Summary (1)


- We have introduced the concept of proportional-share scheduling and briefly discussed three approaches: lottery scheduling, stride scheduling, and the Completely Fair Scheduler (CFS) of Linux.
- Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. CFS, the only “real” scheduler discussed in this chapter, is a bit like weighted round-robin with dynamic time slices, but built to scale and perform well under load; to our knowledge, it is the most widely used fair-share scheduler in existence today.



## Summary (2)

- No scheduler is a panacea (ครอบจักรวาล) , and fair-share schedulers have their fair share of problems.
- One issue is that such approaches do not particularly mesh well with I/O [AC97]; as mentioned above, jobs that perform I/O occasionally may not get their fair share of CPU.
- Another issue is that they leave open the hard problem of ticket or priority assignment, i.e., how do you know how many tickets your browser should be allocated, or to what nice value to set your text editor? Other general-purpose schedulers (such as the MLFQ we discussed previously, and other similar Linux schedulers) handle these issues automatically and thus may be more easily deployed.
- The good news is that there are many domains in which these problems are not the dominant concern (มีหลายโดเมนที่ปัญหานี้ไม่ใช่ปัญหาหลัก), and proportional-share schedulers are used to great effect. สามารถนำไปใช้กับโดเมนอื่นได้เช่น memory ใน VM
- For example, in a virtualized data center (or cloud), where you might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation, proportional sharing can be simple and effective.
- The idea can also be extended to other resources; see Waldspurger [W02] for further details on how to proportionally share memory in VMWare's ESX Server.






## Homework (Simulation)

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. See the README for details.

### Questions

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.
2. Now do the same but with jobs of different lengths: 100, 200, and 300.
3. Now do the same, but also with the RR scheduler and a time-slice of 1.
4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?
6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?
7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given  $N$  jobs?




## Homework (Simulation)

This program, `mlfq.py`, allows you to see how the MLFQ scheduler presented in this chapter behaves. See the README for details.

### 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?
6. One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the `-I` flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.



## Homework (Simulation)

This program, `lottery.py`, allows you to see how a lottery scheduler works. See the README for details.

### Questions

1. Compute the solutions for simulations with 3 jobs and random seeds of 1, 2, and 3.
2. Now run with two specific jobs: each of length 10, but one (job 0) with just 1 ticket and the other (job 1) with 100 (e.g., `-1 10:1, 10:100`). What happens when the number of tickets is so imbalanced? Will job 0 ever run before job 1 completes? How often? In general, what does such a ticket imbalance do to the behavior of lottery scheduling?
3. When running with two jobs of length 100 and equal ticket allocations of 100 (`-1 100:100, 100:100`), how unfair is the scheduler? Run with some different random seeds to determine the (probabilistic) answer; let unfairness be determined by how much earlier one job finishes than the other.
4. How does your answer to the previous question change as the quantum size (`-q`) gets larger?
5. Can you make a version of the graph that is found in the chapter? What else would be worth exploring? How would the graph look with a stride scheduler?