Overview

In this section, we'll look at sharing the CPU proportionally with the **proportional-share scheduler**, also known as a **fair-share scheduler**.

This section should help us answer the following questions:

- **How can we build a scheduler to share the CPU proportionally?**
- **What are the key mechanisms we should use?**
- **How effective are they?**

Introduction

Let's look at a **proportional-share scheduler**, also known as a fair-share scheduler.

Instead of optimizing for **turnaround** or **response time**, a scheduler can seek to guarantee that **each job gets a certain percentage of CPU time**.

The idea is to hold a **lottery** every now and then to choose which process should run next. Processes that should run more often should have greater chances to win.

How can we build a scheduler to share the CPU proportionally?

Tickets Represent Your Share

Lottery scheduling uses the concept of **tickets**, which represent **the share of a resource that a process should receive**. A process's percentage of tickets represents its share of a system resource.

For example, Imagine two processes, A with 75 tickets, and B with 25 tickets. So, we want A to get 75 of the CPU and B the remaining 25.

A lottery every so often does this probabilistically by holding a lottery periodically (say, every time slice).

To hold a lottery, the scheduler has to know the total number of tickets (in our example, there are 100). The scheduler then selects a winning ticket from 0 to 99.

If A has tickets 0–74 and B has tickets 75–99, the winning ticket determines whether A or B runs.

The scheduler then runs the winning process's state.

Winning tickets from a lottery scheduler and the resulting schedule might look similar to the example below:

63	85	70	39	76	17	29	41	36	39	10	99	68	83	63	62	43
A		A	A		A	A	A	A	A	A		A		A	A	A
	B			B							B		B			

Randomization in lottery scheduling results in probability correctness, but not certainty.

In our example, B is only allowed to use four out of twenty time slices (20) instead of the required 25. The longer these two jobs compete, the more likely they are to achieve the required percentages.

What does a process' percentage of tickets represent?

Ticket Mechanisms

Ticket Currency

Lottery scheduling also allows for the manipulation of tickets in various and occasionally effective ways, including the idea of a **ticket currency** Like ticket money, a user with a set of tickets can allocate tickets among their own jobs in any currency they like, and the system will automatically convert that currency into the correct global value.

Let's say users A and B each have 100 tickets.

- User A runs two jobs, A1 and A2, each worth 500 tickets (out of 1000).
- User B runs one job and assigns 10 tickets (out of 10 total).

The system converts A1 and A2's allocations from 500 to 50 in the global currency, and B1's 10 tickets to 100. The global ticket currency (200 total) is then used to determine which job runs.

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

Ticket Transfer

Using the **ticket transfer** mechanism, a process can temporarily hand off its tickets to another process.

This is useful in client/server scenarios when a client process sends messages to a server to do work for them. While the server is handling the client's request, the client might pass the tickets to the server to speed up the job. When it's finished, the server returns the tickets to the client, and everything continues.

Ticket Inflation

Ticket inflation allows a process to temporarily increase or decrease the amount of tickets it holds.

In a competitive environment with distrusting processes, this makes little sense. It's possible for one selfish process to take control of the machine by giving itself a hoard of tickets.

Inflation can be used in situations where mutual trust exists between processes. If a process needs more CPU time, it can boost its ticket value without having to communicate with any other processes.

Implementation

To implement **lottery scheduling**, you only need:
* a strong **random number generator**,
* a **data structure** to track the system's processes (like a list), and
* the **total number of tickets**.

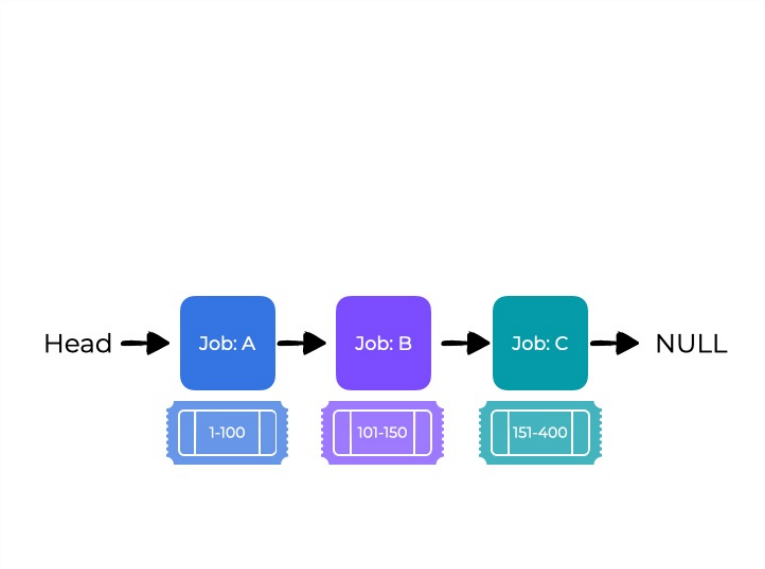
Say we have a list of processes, like the following example, having three processes with different ticket amounts.

The snippet to the left shows an example how to code a scheduling decision.

To make our scheduling decision, we have to:

- Pick a random number (the winner) from the total number of tickets(400). Let's say we chose 300.
- Find the winner by traverse the list using a counter.

In the code, each process' ticket value is added to the counter until it exceeds the value of the winner. As soon as that happens, the current element of the list wins.



.guides/img/lottery1

Using the 300 winning ticket as an example:

- First, counter is incremented to 100 to account for A's tickets
- Because 100 is less than 300, the loop continues.
- Then the counter is incremented to 150 (B's tickets)
- This is still less than 300, so we go on.

- Finally, counter is updated to 400 (definitely larger than 300), and we exit the loop at C. (the winner).

To make this process as efficient as possible, sort the list from highest to lowest number of tickets.

The ordering does not impact the algorithm's accuracy, but it makes sure that the least amount of list iterations are taken, especially if a few processes own most of the tickets.

Which of the following are necessary to implement lottery scheduling?

Example

To better understand lottery scheduling dynamics, we compare the completion time of two projects with the same amount of tickets (100) and run time (R, which we will adjust).

We want each job to finish at roughly the same time, but due to the randomness of lottery scheduling, one job may finish before the other.

To measure this difference, we'll use a **fairness metric**, F, which is the time the first job completes divided by the time that the second job completes, or:

$$F = \frac{Endtime_A}{Endtime_B}$$

So, if:
* Runtime (R) = 10
* Job A finishes at time 10 (Endtime_A = 10), and
* Job B finishes at time 20 (Endtime_B = 20), then

$$F = \frac{10}{20} = 0.5$$

When both jobs finish at nearly the same time, F will be very close to one. In this case, that's our goal: a perfectly fair scheduler would achieve $F = 1$.

Completely Fair Scheduler (CFS)

We haven't addressed **how to assign tickets to jobs** with lottery scheduling. This is a difficult problem because the system's behavior heavily depends on ticket distribution.

Assuming users know best, each user is given a certain number of tickets, which they can distribute to whatever jobs they like.

But this is not really a solution: it doesn't tell you what to do, so the "ticket-assignment problem" remains open.

With the **Completely Fair Scheduler (CFS)**, fair-share scheduling is implemented in a very efficient and scalable way. **CFS** attempts to achieve its efficiency goals through its design and creative use of data structures well suited for the task.

CFS: The Basics

The **Completely Fair Scheduler** (or **CFS**) doesn't use a set time slice. Its goal is to fairly share the CPU among all competing processes.

It does this by using a counting method called **virtual runtime (vruntime)**. Every process collects `vruntime`. In basic situations, `vruntime` increases proportionally to physical (real) time.

When a scheduling decision is made, **CFS picks the process with the lowest `vruntime` to run next**.

How does the scheduler know when to stop one process and start another?

There's a conflict here:

- If CFS switches too frequently, fairness is improved because each process receives its fair portion of CPU even during short time windows, but speed is sacrificed.
- If CFS switches less frequently, performance improves (lower context switching), but at the expense of near-term fairness.

CFS controls this tension using various control parameters.

sched_latency

The first parameter, `sched_latency`, is a value used to determine **how long a process should run before switching**.

A common sched_latency value is 48 $(milliseconds)$.

CFS divides this value by the number of processes operating on the CPU (n) To determine a completely fair time slice for a process.

If there are $n = 4$ processes, CFS divides sched latency by n to provide a per-process time slice of $12ms$. CFS then:

- Schedules the first job
- Executes it for $12ms$ (virtual) runtime, and
- Checks for a job with less `vruntime` to run instead.

In this case, there is a job with less rent time so CFS will switch to one of the remaining jobs.

Our illustration shows an example where each of the four jobs (A, B, C, D) runs for two time slices. Two jobs (C, D) finish, leaving A and B to run for 24 ms in round-robin.

What if “too many” processes are running? Wouldn’t that mean a time slice that was too small, and a lot of context switching would happen?

Well, yes!

To solve this, CFS introduces another control parameter, `min_granularity`.

min_granularity

Our next parameter,`min_granularity` is commonly set to value like $6ms$. CFS will never set a process' time slice below this value, reducing excessive scheduling overhead.

For instance, If 10 processes are executing, we divide `sched_latency` by 10 to get the time slice (result: $4.8ms$).

Due to `min_granularity` being set to 6, CFS will instead set each process' time slice to $6ms$.

While CFS won't quite hit the desired **scheduling latency**(`sched_latency`) of 48 ms, it will get close while keeping good CPU efficiency.

Job time slices that are not perfect multiples of timer interrupt intervals are fine. CFS tracks `vruntime` exactly, so it will gradually approximate optimal CPU sharing.

Weighting (Niceness)

Using a UNIX feature known as a process' **nice** level, CFS allows users or administrators to prioritize processes, giving them a larger share of the CPU.

A process's **nice** parameter ranges from -20 to $+19$, with 0 being the default.
* **Positive** nice values denote **lower priority**
* **Negative** nice values denote **higher priority**

When you're too nice, you're just not given as much (scheduling) attention.

CFS maps the nice value of each process to a `weight`. You can see this in the code example to the left.

The `weights` allow us to compute the appropriate time slice of each process while taking priority differences into account. Assuming n processes, the formula is as follows:

$$timeSlice_k = \frac{weight_k}{\sum_{i=0}^{n-1} weight_i} * schedLatency$$

Let's look at an example.

A and B are jobs.
* A, because it's our most important job, gets a higher priority, nice value of -5
* B only gets the default priority and a nice value of 0.

If we refer to the table on the left, `WeightA` is 3121 and `weightB` is 1024.

Computing the time slice of each job, A's time slice is about $\frac{3}{4}$ `sched_latency` ($36ms$) and B's is about $\frac{1}{4}$ ($12ms$).

Aside from generalizing time slice computation, CFS `vruntime` calculation has to be changed.

The new formula takes the actual run time that process `i` has accumulated (`runtimei`) and scales it inversely by the weight of the process, by dividing the default weight of 1024 (`weight0`) by its weight, `weighti`.

In our example, A's `vruntime` accumulates at a third the rate of B's.

$$vruntime_i = nvruntime_i + \frac{weight_0}{weight_i} * runtime_i$$

The set of weights in the code retain CPU proportionality ratios when the difference in nice values is constant. In this case, CFS would schedule processes A and B like before, with nice values of 5 (not -5) and 10 (not 0).

Using Red-Black Trees

Efficiency is a major CFS focus. The scheduler should find the next job to run as fast as possible.

Lists and other simple data structures don't scale: current systems can have 1000s of processes, so searching through a big list every millisecond is inefficient.

CFS solves this problem by storing processes in a **red-black tree**, one of many types of balanced trees.

Balanced trees work harder than simple binary trees to maintain low depths, making sure operations are logarithmic (not linear) in time.

Only running (or runnable) processes are stored in CFS. A process that goes to sleep (say, waiting for an I/O or a network packet) is removed from the tree and tracked somewhere else.

For example, there are ten tasks with `vruntime` values of $1, 5, 9, 10, 14, 18, 17, 21, 22$, and 24.

If we kept these jobs in a list, finding the next job to run would be a matter of removing the first element from the list. However, placing that job back into the list (in order) would require scanning the list and looking for the right spot to put it in, an $O(n)$ operation.

Any search is inefficient, also taking linear time.

As seen our illustration, keeping the same values in a red-black tree improves efficiency. Most operations (like insertion and deletion) are logarithmic in time, i.e., $O(logn)$. When n is in the thousands, logarithmic is far more efficient.

Dealing with I/O and Sleeping Processes

One issue with choosing the lowest `vruntime` to run next is jobs that have been inactive for a long time.

Imagine two processes, A and B:
* One running continuously, and
* The other sleeping for a long time (say, 10 seconds).

Because B's `vruntime` is 10 seconds behind A's, it will hog the CPU for the next 10 seconds as it catches up, ultimately starving A CFS tackles this situation by changing a job's vruntime when it wakes up.

CFS sets the job's `vruntime` to the tree's minimal value (remember, the tree only contains running jobs).

CFS avoids starvation, but at a cost: processes that sleep quickly often do not get their fair portion of CPU.

Summary

So far, we've introduced proportional-share scheduling and briefly examined two methods: lottery scheduling and Linux's Completely Fair Scheduler.

- **Lottery scheduling** uses randomness in a clever way to achieve proportional share.
- **CFS**, the only "true" scheduler covered in this chapter, is similar to weighted round-robin with dynamic time slices, but built to scale and function effectively under stress.
- No scheduler is perfect, and even fair-share schedulers have issues. One problem is that such approaches don't work well with I/O.
- I/O jobs may not always get their fair share of CPU.
- Other general-purpose schedulers (like MLFQ and comparable Linux schedulers) may solve these concerns automatically and be easier to deploy.
- Proportional-share schedulers work well in many contexts where these issues are minor.

Lab Intro

This program, `lottery.py`, simulates a lottery scheduler. Run the program without the `-c` flag first to displays you the problem but not the solution.

```
./lottery.py -j 2 -s 0
```

When you run the simulator with the code above, it assigns you random jobs (8 and 4 lengths) with random tickets (here 75 and 25, respectively).

The simulator also gives a list of random numbers for you to determine the lottery scheduler's actions. **You have to use the modulo (%) operator to compute the lottery winner** because the random numbers are between 0 and a massive amount. **The winner should equal this random number mod (%) the total number of tickets in the system**.

Running with `-c` shows you exactly what to calculate:

```
./lottery.py -j 2 -s 0 -c
```

Your output should looks similar to the following:

```
** Solutions **
Random 511275 -> Winning ticket 75 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:8 tix:75 ) (* job:1 timeleft:4 tix:25
)
Random 404934 -> Winning ticket 34 (of 100) -> Run 0
  Jobs:  (* job:0 timeleft:8 tix:75 ) (  job:1 timeleft:3 tix:25
)
Random 783799 -> Winning ticket 99 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:7 tix:75 ) (* job:1 timeleft:3 tix:25
)
Random 303313 -> Winning ticket 13 (of 100) -> Run 0
  Jobs:  (* job:0 timeleft:7 tix:75 ) (  job:1 timeleft:2 tix:25
)
Random 476597 -> Winning ticket 97 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:6 tix:75 ) (* job:1 timeleft:2 tix:25
)
Random 583382 -> Winning ticket 82 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:6 tix:75 ) (* job:1 timeleft:1 tix:25
)
--> JOB 1 DONE at time 6
Random 908113 -> Winning ticket 13 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:6 tix:75 ) (  job:1 timeleft:0 tix:--
- )
Random 504687 -> Winning ticket 12 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:5 tix:75 ) (  job:1 timeleft:0 tix:--
- )
Random 281838 -> Winning ticket 63 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:4 tix:75 ) (  job:1 timeleft:0 tix:--
- )
Random 755804 -> Winning ticket 29 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:3 tix:75 ) (  job:1 timeleft:0 tix:--
- )
Random 618369 -> Winning ticket 69 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:2 tix:75 ) (  job:1 timeleft:0 tix:--
- )
Random 250506 -> Winning ticket 6 (of 75) -> Run 0
  Jobs:  (* job:0 timeleft:1 tix:75 ) (  job:1 timeleft:0 tix:--
- )
--> JOB 0 DONE at time 12
```

This trace shows us that we have to use the random number to find out which ticket is the winner. Then, figure out which job is holding the winning ticket and should run next. Repeat until all jobs are finished. This is exactly what the lottery scheduler does!

Consider the first decision made in the example above. We now have two jobs (`job 0` which has a runtime of `8` and `75` tickets, and `job 1` which has a runtime of `4` and `25` tickets). The first number is `511275`. The winning ticket is `75`, and there are `100` in the system.

If ticket `75` wins, we just search the job list for it. The initial entry for `job 0` has `75` tickets `(0-74)`, so it loses; the second entry is for `job 1`, who wins, so we run `job 1` for the quantum length (1 in this example). The printout shows this as follows:

```
Random 511275 -> Winning ticket 75 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:8 tix:75 ) (* job:1 timeleft:4 tix:25
)
```

This trace shows us that we have to use the random number to find out which ticket is the winner. Then, figure out which job is holding the winning ticket and should run next. Repeat until all jobs are finished. This is exactly what the lottery scheduler does!

Consider the first decision made in the example above. We now have two jobs (job 0 which has a runtime of 8 and 75 tickets, and job 1 which has a runtime of 4 and 25 tickets). The first number is `511275`. The winning ticket is 75, and there are 100 in the system.

If ticket 75 wins, we just search the job list for it. The initial entry for job 0 has 75 tickets (0-74), so it loses; the second entry is for job 1, who wins, so we run job 1 for the quantum length (1 in this example). The printout shows this as follows:

```
Random 511275 -> Winning ticket 75 (of 100) -> Run 1
  Jobs:  (  job:0 timeleft:8 tix:75 ) (* job:1 timeleft:4 tix:25
)
```

The first line describes what happens, while the second displays the whole job queue, with a * marking which job was chosen.

The `-l`/—jlist flag can be used to specify an exact list of jobs and their ticket values.

As usual, you can use the `-h` tag for a full list of options for this simulator.

```
./lottery.py -hS
```

Lab 1

1. Compute the solutions for simulations with 3 jobs and random seed 1, 2, and 3.

```
./lottery.py -j 3 -s 1
./lottery.py -j 3 -s 2
./lottery.py -j 3 -s 3
```

Can you calculate the winning ticket and processes?
##
2. Run with two specific jobs:
* Each of length 10
* Job 0 has 1 ticket
* Job 1 has 100 tickets

```
./lottery.py -l 10:1,10:100
```

What happens when the number of tickets is this imbalanced?

Will Job 0 ever run before job 1 completes?

3. When running the simulator with:

- Two jobs, both length 100
- Both with ticket allocations of 100

```
./lottery.py -l 100:100,100:100
```

How unfair is the scheduler? (Run with a few different random seeds to determine the probabilistic answer)

How does this answer change and the quantum size increases?

Useful Flags

```
Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -j JOBS, --jobs=JOBS  number of jobs in the system
  -l JLIST, --jlist=JLIST
                        instead of random jobs, provide a comma-
separated list
                        of run times and ticket values (e.g.,
10:100,20:100
                        would have two jobs with run-times of 10
and 20, each
                        with 100 tickets)
  -m MAXLEN, --maxlen=MAXLEN
                        max length of job
  -T MAXTICKET, --maxticket=MAXTICKET
                        maximum ticket value, if randomly
assigned
  -q QUANTUM, --quantum=QUANTUM
                        length of time slice
  -c, --compute         compute answers for me
```