9

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?

9.1 Basic Concept: Tickets Represent Your Share

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

TIP: USE RANDOMNESS

One of the most beautiful aspects of lottery scheduling is its use of **randomness**. When you have to make a decision, using such a randomized approach is often a robust and simple way of doing so.

Random approaches have at least three advantages over more traditional decisions. First, random often avoids strange corner-case behaviors that a more traditional algorithm may have trouble handling. For example, consider the LRU replacement policy (studied in more detail in a future chapter on virtual memory); while often a good replacement algorithm, LRU attains worst-case performance for some cyclic-sequential workloads. Random, on the other hand, has no such worst case.

Second, random also is lightweight, requiring little state to track alternatives. In a traditional fair-share scheduling algorithm, tracking how much CPU each process has received requires per-process accounting, which must be updated after running each process. Doing so randomly necessitates only the most minimal of per-process state (e.g., the number of tickets each has).

Finally, random can be quite fast. As long as generating a random number is quick, making the decision is also, and thus random can be used in a number of places where speed is required. Of course, the faster the need, the more random tends towards pseudo-random.

a winning ticket, which is a number from 0 to 99[1]. 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.

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
```

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.

[1] Computer Scientists always start counting at 0. It is so odd to non-computer-types that famous people have felt obliged to write about why we do it this way [D82].

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.

9.2 Ticket Mechanisms

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.

```
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)
```

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.

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.

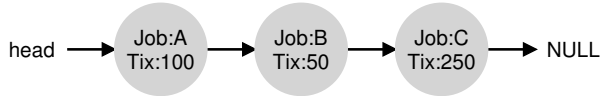
```
1  // counter: used to track if we've found the winner yet
2  int counter = 0;
3
4  // winner: use some call to a random number generator to
5  //         get a value, between 0 and the total # of tickets
6  int winner = getrandom(0, totaltickets);
7
8  // current: use this to walk through the list 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...
```

Figure 9.1: **Lottery Scheduling Decision Code**

9.3 Implementation

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.



To make a scheduling decision, we first have to pick a random number (the winner) from the total number of tickets (400)[2] 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).

[2]Surprisingly, as pointed out by Björn Lindberg, this can be challenging to do correctly; for more details, see `http://stackoverflow.com/questions/2509679/how-to-generate-a-random-number-from-within-a-range`.

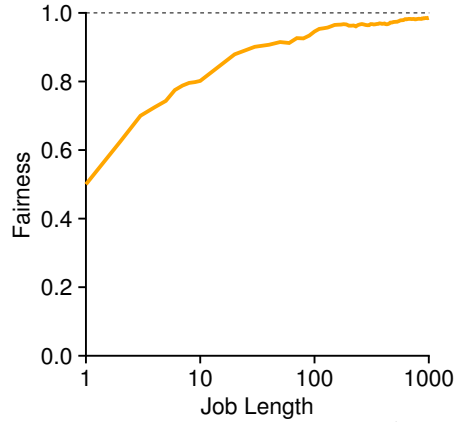


Figure 9.2: **Lottery Fairness Study**

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.

9.4 An Example

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), $F = \frac{10}{20} = 0.5$. 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 $F = 1$.

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.

9.5 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.

9.6 Stride Scheduling

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 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.

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
```

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

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**

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.

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.

9.7 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 fair-share 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 datacenters, 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.

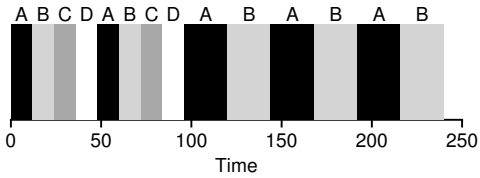


Figure 9.4: **CFS Simple Example**

Basic Operation

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.

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.

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.

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)

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,
};
```

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:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency} \tag{9.1}$$

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 pri-

ority by assigning it a nice value of -5; B, because we hates it[3], just has the default priority (nice value equal to 0). This means $\texttt{weight}_A$ (from the table) is 3121, whereas $\texttt{weight}_B$ is 1024. If you then compute the time slice of each job, you'll find that A's time slice is about $\frac{3}{4}$ of `sched_latency` (hence, 36 ms), and B's about $\frac{1}{4}$ (hence, 12 ms).

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 ($\texttt{runtime}_i$) and scales it inversely by the weight of the process, by dividing the default weight of 1024 ($\texttt{weight}_0$) by its weight, $\texttt{weight}_i$. In our running example, A's `vruntime` will accumulate at one-third the rate of B's.

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i \tag{9.2}$$

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

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.

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,

[3] Yes, yes, we are using bad grammar here on purpose, please don't send in a bug fix. Why? Well, just a most mild of references to the Lord of the Rings, and our favorite anti-hero Gollum, nothing to get too excited about.

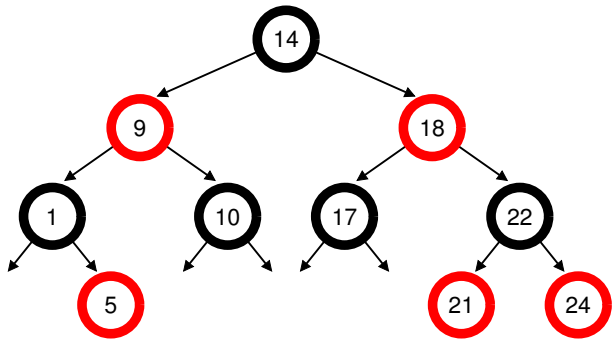


Figure 9.5: **CFS Red-Black Tree**

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.

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.

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

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.

each process as an independent entity), and many other interesting features. Read recent research, starting with Bouron [B+18], to learn more.

9.8 Summary

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.

No scheduler is a panacea, 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, and proportional-share schedulers are used to great effect. 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.

References

[AC97] "Extending Proportional-Share Scheduling to a Network of Workstations" by Andrea C. Arpaci-Dusseau and David E. Culler. PDPTA'97, June 1997. *A paper by one of the authors on how to extend proportional-share scheduling to work better in a clustered environment.*

[B+18] "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS" by J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, J. Sopena. USENIX ATC '18, July 2018, Boston, Massachusetts. *A recent, detailed work comparing Linux CFS and the FreeBSD schedulers. An excellent overview of each scheduler is also provided. The result of the comparison: inconclusive (in some cases CFS was better, and in others, ULE (the BSD scheduler), was. Sometimes in life there are no easy answers.*

[B72] "Symmetric binary B-Trees: Data Structure And Maintenance Algorithms" by Rudolf Bayer. Acta Informatica, Volume 1, Number 4, December 1972. *A cool balanced tree introduced before you were born (most likely). One of many balanced trees out there; study your algorithms book for more alternatives!*

[D82] "Why Numbering Should Start At Zero" by Edsger Dijkstra, August 1982. Available: `http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF`. *A short note from E. Dijkstra, one of the pioneers of computer science. We'll be hearing much more on this guy in the section on Concurrency. In the meanwhile, enjoy this note, which includes this motivating quote: "One of my colleagues — not a computing scientist — accused a number of younger computing scientists of 'pedantry' because they started numbering at zero." The note explains why doing so is logical.*

[K+15] "Profiling A Warehouse-scale Computer" by S. Kanev, P. Ranganathan, J. P. Darago, K. Hazelwood, T. Moseley, G. Wei, D. Brooks. ISCA '15, June, 2015, Portland, Oregon. *A fascinating study of where the cycles go in modern data centers, which are increasingly where most of computing happens. Almost 20% of CPU time is spent in the operating system, 5% in the scheduler alone!*

[J09] "Inside The Linux 2.6 Completely Fair Scheduler" by M. Tim Jones. December 15, 2009. `http://ostep.org/Citations/inside-cfs.pdf`. *A simple overview of CFS from its earlier days. CFS was created by Ingo Molnar in a short burst of creativity which led to a 100K kernel patch developed in 62 hours.*

[KL88] "A Fair Share Scheduler" by J. Kay and P. Lauder. CACM, Volume 31 Issue 1, January 1988. *An early reference to a fair-share scheduler.*

[WW94] "Lottery Scheduling: Flexible Proportional-Share Resource Management" by Carl A. Waldspurger and William E. Weihl. OSDI '94, November 1994. *The landmark paper on lottery scheduling that got the systems community re-energized about scheduling, fair sharing, and the power of simple randomized algorithms.*

[W95] "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management" by Carl A. Waldspurger. Ph.D. Thesis, MIT, 1995. *The award-winning thesis of Waldspurger's that outlines lottery and stride scheduling. If you're thinking of writing a Ph.D. dissertation at some point, you should always have a good example around, to give you something to strive for: this is such a good one.*

[W02] "Memory Resource Management in VMware ESX Server" by Carl A. Waldspurger. OSDI '02, Boston, Massachusetts. *The paper to read about memory management in VMMs (a.k.a., hypervisors). In addition to being relatively easy to read, the paper contains numerous cool ideas about this new type of VMM-level memory management.*

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., `-l 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 (`-l 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?