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 simply guarantee that each job obtain a certain percentage of CPU time.

An excellent modern 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 much 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 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 49
```

Here is the resulting schedule:

```
A  B  A  A  B  A  A  A  A  A  A  B  A  B  A  A  A  A  A  A
```

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.

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.

[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 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 has 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 LRU page replacement (studied in more detail in a future chapter on virtual memory); while often a good replacement algorithm, LRU performs pessimally 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.

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 User A's own currency. User B is running only 1 job and gives it 10 tickets (out of 10 total). The system will convert 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 will be converted to 100 tickets. The lottery will then be 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

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.

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.

9.3 Implementation

Probably the most amazing thing about lottery scheduling is the simplicity of its implementation. Basically, all you need is a good random number generator to pick the winning ticket, a simple 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 list comprised of three processes, A, B, and C, each with some number of tickets.

```
head -> ( A:100 ) -> ( B:50 ) -> ( C:250 ) -> null
```

To make a scheduling decision, we first have to pick a random number (the winner) from the total number of tickets (400). 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 shows some code that will do just that.

All the code does is walk the list of processes, adding their ticket value to `counter` until the value exceeds `winner`. Once that is the case, the current list element is the winning process. With our example of the winning ticket being 300, the following would take place. First, counter would be incremented to 100 to account for A's tickets; because 100 is less than 300, the loop would continue. Then, counter would be updated to 150 (B's tickets), still less than 300 and thus again we continue. Finally, the counter is updated to 400, clearly greater than 300, and thus we would break out of the loop with `current` pointing at process C as the winner.

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 effect the correctness of the algorithm; however, it does ensure in general that the fewest number of list iterations are taken.

```
// counter: used to track if we've found the winner yet
int counter     = 0;

// winner: use some call to a random number generator to
//         get a value, between 0 and the total # of tickets
int winner      = random() % totaltickets;

// current: use this to walk through the list of jobs
node_t *current = head;

// loop until the sum of ticket values is > the winner
while (current) {
    counter = counter + current->tickets;
    if (counter > winner)
        break; // found the winner
    current = current->next;
}
// current is the winner: schedule it...
// (NOT SHOWN)
```

Figure 9.1: Lottery Scheduling Decision Code

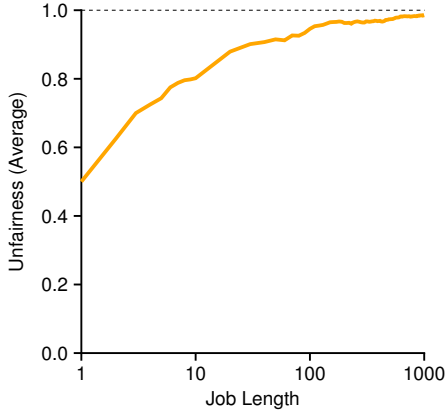

Figure 9.2: Lottery Fairness Study

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 **unfairness metric**, U 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), $U = \frac{10}{20} = 0.5$. When both jobs finish at nearly the same time, U will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve $U = 1$.

Figure 9.2 plots the average unfairness 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 unfairness can be quite severe. Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired 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 Why Not Deterministic?

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]:

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride; // compute next pass using stride
insert(queue, current);            // put back into the queue
```

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	...

Table 9.1: Stride Scheduling: A Trace

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. A runs, and 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. Table 9.1 traces the behavior of the scheduler over time.

As we can see from the table, 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.

So why use lottery 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 Summary

We have introduced the concept of proportional-share scheduling and briefly discussed two implementations: lottery and stride scheduling. Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. Although both are conceptually interesting, they have not achieved wide-spread adoption as CPU schedulers for a variety of reasons. One is that such approaches do not particularly mesh well with I/O [AC97]; another is that they leave open the hard problem of ticket assignment. General-purpose schedulers (such as the MLFQ we discussed previously) do so more gracefully and thus are more widely deployed.

As a result, proportional-share schedulers are more useful in domains where some of these problems (such as assignment of shares) are relatively easy to solve. For example, in a virtual machine environment, 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. See Waldspurger [W02] for further details on how such a scheme is used to proportionally share memory in VMWare's ESX Server.

References

[AC97] "Extending Proportional-Share Scheduling to a Network of Workstations"
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.

[D82] "Why Numbering Should Start At Zero"
Edsger Dijkstra, August 1982
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.

[KL88] "A Fair Share Scheduler"
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"
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"
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"
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

This program, `lottery.py`, allows you to see how a lottery scheduler works. As always, there are two steps to running the program. First, run without the `-c` flag: this shows you what problem to solve without revealing the answers.

```
prompt> ./lottery.py -j 2 -s 0
...
Here is the job list, with the run time of each job:
  Job 0 ( length = 8, tickets = 75 )
  Job 1 ( length = 4, tickets = 25 )

Here is the set of random numbers you will need (at most):
Random 511275
Random 404934
Random 783799
Random 303313
Random 476597
Random 583382
Random 908113
Random 504687
Random 281838
Random 755804
Random 618369
Random 250506
```

When you run the simulator in this manner, it first assigns you some random jobs (here of lengths 8, and 4), each with some number of tickets (here 75 and 25, respectively). The simulator also gives you a list of random numbers, which you will need to determine what the lottery scheduler will do. The random numbers are chosen to be between 0 and a large number; thus, you'll have to use the modulo operator to compute the lottery winner (i.e., winner should equal this random number modulo the total number of tickets in the system).

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

```
prompt> ./lottery.py -j 2 -s 0 -c
...
** 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
```

As you can see from this trace, what you are supposed to do is use the random number to figure out which ticket is the winner. Then, given the winning ticket, figure out which job should run. Repeat this until all of the jobs are finished running. It's as simple as that – you are just emulating what the lottery scheduler does, but by hand!

Just to make this absolutely clear, let's look at the first decision made in the example above. At this point, we 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 random number we are given is 511275. As there are 100 tickets in the system, 511275 % 100 is 75, and thus 75 is our winning ticket.

If ticket 75 is the winner, we simply search through the job list until we find it. The first entry, for job 0, has 75 tickets (0 through 74), and thus does not win; the next entry is for job 1, and thus we have found our winner, so we run job 1 for the quantum length (1 in this example). All of this is shown in the print out as follows:

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

As you can see, the first line summarizes what happens, and the second simply shows the entire job queue, with an $*$ denoting which job was chosen.

The simulator has a few other options, most of which should be self-explanatory. Most notably, the `-l/--jlist` flag can be used to specify an exact set of jobs and their ticket values, instead of always using randomly-generated job lists.

```
prompt> ./lottery.py -h
Usage: lottery.py [options]

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, --maxtick=MAXTICKET
      maximum ticket value, if randomly assigned
  -q QUANTUM, --quantum=QUANTUM
      length of time slice
  -c, --compute
      compute answers for me
```

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?