**Chapter 9: Scheduling: Proportional Share**

In this chapter, we will examine a different type of scheduler known as a **proportional-share scheduler** (**fair-share scheduler**). The concept is simple. The scheduler try to guarantee that each job obtain a certain percentage of CPU time.

**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 should receive. The percent of tickets that a process has represents its share of the system resource in question.

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%.

This is achieved by holding a lottery every so often. Holding a lottery is straightforward: the scheduler must know how many total tickets there are. The scheduler then picks a winning ticket 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. Then as it generates winning tickets, A and B will get scheduled if they possess the winning tickets. For example:

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

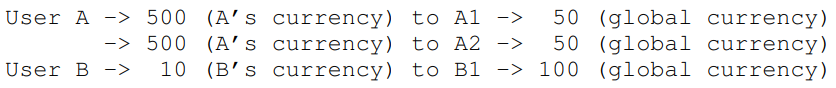
Schedule: A B A A B A A A A A A B A B A A A A A A

As we can see, the probability of B getting to run is 20%, but as it runs longer, we will achieve the desired percentage.

**9.2 Ticket Mechanisms:**

One way to manipulate tickets is using **ticket currency**. This allows the user to allocate tickets among their own jobs in whatever currency they would like.

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.



Another mechanism is **ticket transfer**, meaning that 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 inflation** can sometimes be useful where 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.

**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.

Text

Description automatically generated

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).

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:**

**Fairness metric** is simply the time the first job completes divided by the time that the second job completes. The perfect fair scheduler would achieve F = 1.

Chart, line chart

Description automatically generated

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 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 nonsolution: it really doesn’t tell you what to do.

**9.6 Stride Scheduling**

Lottery scheduling occasionally will not deliver the exact right proportions, especially over short time scales. For this reason, **stride scheduling**, a deterministic fair-share scheduler, is invented.

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.

Table

Description automatically generated

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.

**9.7 The Linux Complete Fair Scheduler (CFS):**

To achieve the 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.

**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.

The tension 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. This value grows inversely with the number of processes. If there are many processes, the scheduler prevents making too many context switches by defining **min\_granularity**. 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.

CFS utilizes a **periodic timer interrupt**

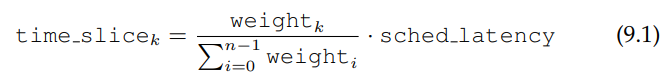
**Weighting (Niceness)**

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.

Table

Description automatically generated

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:



For example, in the table, weight of A (nice value of -5) is 3121 and weight of B (nice value of 0) is 1024. Therefore, A’s time slice is about ¾ of sched\_latency, while B’s is about ¼.

CFS uses different way to calculate vruntime. Here is the new formula, which takes the actual run time that process i has accrued (runtimei) and scales it inversely by the weight of the process, by dividing the default weight of 1024 (weight0 ) by its weight, weighti.

**Using Red-Black Trees**

When the scheduler has to find the next job to run, it should do so as quickly as possible. CFS addresses this by keeping processes in a **red-black tree**.

CFS does not keep all process in this structure; rather, only running (or runnable) processes are kept therein.

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).

**Dealing With I/O And Sleeping Processes**

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. 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.

**Other CFS fun**

It includes numerous heuristics to improve cache performance, has strategies for handling multiple CPUs effectively, can schedule across large groups of processes, etc.