Schedule Narrative

Introduction

Suppose you are in charge of a large data center. You lease CPU time to customers. For example, I may be a customer who pays $5.00 per month for a small Linux CPU. Your data center virtualizes the CPUs in your data center. Each CPU serves multiple Linux instances. Your scheduler schedules Linux instances. Some Linux instances are simple $5.00 per month Linux like Gusty's and some are complex servers used by Netflix that grow as demand on Friday night increases. You want to optimize the CPU use. Suppose analysis determines 10% of your CPU use is spent in the scheduler. Suppose your company is making $10,000,000 per month from your data center, but you have more demand than you have CPU bandwidth. If you can find a scheduler that consumes 5% of your CPU, you can increase your CPU bandwidth by 5%, which will earn an additional $500,000 per month. Your boss will give you a $25,000 monthly raise, if you can do this.

LCFS, Lottery, and Stride Schedulers

LCFS, Lottery, and Stride are fair schedulers that have simple algorithms. All processes get a chance to run. No process is starved. Additionally, the amount of time each process gets to run is determined by human interaction. In LCFS, you assign nice values to processes. Processes with a smaller nice value get to run more than processes with a higher nice value. In Lottery and Stride, you assign tickets to processes. Processes with more tickets get to run more than processes with less tickets. The remainder of this discusses LCFS and Lottery. Lottery uses random numbers. Stride is similar to Lottery, but the tickets are used to compute each process's stride value - see our book.

Solving Problems - Always Begin with Simple, Concrete Samples

When solving a problem, always begin with simple, concrete samples until you understand the problem and its underlying algorithms and data structure. Chapter 9 of OSTEP has some simple, concrete cases for Lottery and LCFS. You can begin with those and add some of your own. When working on these simple, concrete samples, you should use paper and pencil. There is something about working on simple, concrete samples with paper and pencil that transfer knowledge into your brain. After you understand the algorithm and data structures in simple, concrete samples, you can create a design, test cases, and implement the design as code.

Lottery Scheduler

Lottery uses a fixed timeslice - let's say 15ms. When a proc runs, it runs for 15ms. Processes with more tickets will (over time) run more timeslices than processes with less tickets thereby achieving the fairness defined by the user. In Lottery, the same process may run multiple times in a row. Consider two processes ready to run, P1 has 90 tickets and P2 has 10 tickets. Over time, P1 will run nine times as much as P1, but the first 5 quantum could very well be P2, P2, P2, P1, P2 just because a random number is used to select the process. However, if you examine 1000 quantum, P1 will run close to 900 times, which is 9 times more than P2 runs. A good feature of Lottery is the scheduling algorithm works when the number of ready processes vary each time the scheduler runs. If one quantum 2 procs are ready and the next quantum 10 procs are ready, the total tickets change and the algorithm runs. This works ok with compute bound and interactive processes.

LCFS Scheduler

LCFS has variable length timeslices. Processes with smaller nice values run for longer timeslices. The ready processes take turns running (mostly). The nice value is an index into a table of weights, which are used in a formula to compute the timeslice of each proc. The smaller the nice value the higher the weight. LCFS uses two values to compute the timeslice for each proc. (1) schedule latency - the total amount of time to be divided into timeslices for the ready procs. (2) minimum granularity - the smallest timeslice allocated. The minimum granularity is needed because if there are a bunch of ready procs, the divvied values of schedule latency could get smaller than the time the OS spends performing a context switch. Consider two ready processes. P1 has a weight of 2700. P2 has a weight of 300. The schedule latency is 100ms and the minimum granularity is 10ms. Notice that the total weight of ready procs is 3000. P1 has 90% of the weight. P2 has 10% of the weight. LCFS will assign P1 a timeslice of 90ms and P2 a timeslice of 10ms. In this case LCFS alternates between P1 and P2. The first four quantums are P1 - 90ms, P2 - 10ms, P1 - 90ms, P2 - 10ms. P1 and P2 alternate, but P1 runs 9 times more than P2.

LCFS translates runtime into virtual runtime - vruntime - and selects the proc in ready with the smallest vruntime as the next proc to run. The formula that computes a proc's vruntime is sort of the opposite of the formula that computes a proc's timeslice. This results in vruntimes increasing equally. Consider P1 and P2 from above. Let them each have a vruntime of 0.

```
vruntime    =  vruntime + weight0/weighti * timeslice

P1 vruntime = P1 vruntime + 1024/2700 * 90
            =      0      + .3792     * 90
            =      0      + 34.13
P2 vruntime = P2 vruntime + 1024/300  * 10
            =      0      + 3.413     * 10
            =      0      + 34.13
```

In LCFS, you have to decide how to "normalize" the vruntime of a process that has been sleeping for a while, otherwise it runs consecutive timeslices until it's vruntime catches up with others. One approach is to raise the vruntime to the average of those in ready. You did not have to wrestle with the normalization of vruntime on the project.

Lottery Implementation and Testing

Lottery Implementation

1. Add a ticket member to struct proc.
2. Update fork command (main.c) and fork() (proc.c) to assign default tickets to ticket members.
3. Create a ticket command (main.c and proc.c) to assign tickets to procs.
   Or add a ticket parameter to the fork command and fork().
4. Update ps command to show the tickets for each proc.
5. Add Lottery algorithm to scheduler.

Lottery Test 1

- Create a two proc scenario where each proc has 50% of tickets.
- Run to demonstrate the procs run equally - use schedule and timer commands.

Lottery Test 2

- Think of another test.

Lottery Test 3

- Think of another test.

Lottery Test 4

- Think of another test.

Lottery Test 5

- Think of another test.

Lottery Test 6

- Think of another test

LCFS Implementation and Testing

LCFS Implementation

1. Add to struct proc: nice, weight, timeslice, vruntime. Or lookup weight when needed.
2. Add schedule latency and minimum granularity to proc.c. Pick numbers that are easy to use - e.g., 100 and 10.
3. Update fork command (main.c) and fork() (proc.c) to assign nice value of 0.
4. Create a nice command (main.c and proc.c) to assign nice values to procs. Or add nice parameter to the fork command
5. Update ps command to show for each proc: nice, weight, timeslice.
6. Add LCFS algorithm to scheduler.

LCFS Test 1

- Create a two proc scenario where each proc has a value of 0 (weight of 1024).
- Run to demonstrate the procs run equally. The two procs should alternate.
- Use ps command to show they each have the same timeslice with is half of the schedule latency.

LCFS Test 2

- Think of another test.

LCFS Test 3

- Think of another test.

LCFS Test 4

- Think of another test.

LCFS Test 5

- Think of another test.

LCFS Test 6

- Think of another test.