Programming Assignment 2: Priority Scheduling

Due April 29, 2024

The Basics

The goal of this assignment is to get everyone up to speed on Minix OS and to gain some familiarity with scheduling. In this assignment you are to implement a priority scheduler. A priority scheduler assigns each process some number of lottery tickets, then randomly draws a ticket among those allocated to ready processes to decide which process to run. That process is allowed to run for a set *time quantum*, after which it is interrupted by a timer interrupt and the process is repeated.

The number of tickets assigned to each process determines both the likelihood that it will run at each scheduling decision as well as the relative amount of time that it will get to execute. Processes that have more lottery tickets are more likely to get chosen each time, and thus will get more CPU time.

Note that we don't have to dole out actual tickets. We can just keep a count of how many tickets each process has $\{n_1, n_2, ..., n_m\}$ (assuming m processes), how many tickets have been doled out altogether (N), and the fraction of the total tickets each process has $\{f_1, f_2, ..., f_m\} = \{n_1/N, n_2/N, ..., n_m/N\}$. Then, to decide which process we should run at each scheduling decision, we can compute a random number p between 0 and 1. If $0 < p < f_1$ then we run process 1. Else if $f_1 < p < f_1+f_2$, we run process 2. Else if $f_1+f_2 < p < f_1+f_2+f_3$, we run process 3. Etc.

One goal of best-effort scheduling is to give I/O-bound processes both faster service and a larger percentage of the CPU when they are ready to run. Both of these things lead to better responsiveness, which is one subjective measure of computer performance for interactive applications. CPU-bound processes, on the other hand, can get by with slower service and a relatively lower percentage of the CPU when there are I/O-bound processes that want to run. Of course, CPU-bound processes need lots of CPU, but they can get most of it when there are no ready I/O-bound processes. One fairly easy way to accomplish this in a *lottery scheduler* is to give I/O-bound processes more tickets - when they are ready they will get service relatively fast, and they will get relatively more CPU than other non-I/O-bound processes.

The key question is how to determine which processes are I/O-bound and which are CPU-bound. One way to do this is to look at whether or not processes block before using up their time quantum. Processes that block before using up their time quantum are doing I/O and are therefore more I/O-bound than those that do not. On the other hand, processes that do not block before using up a time quantum are more CPU-bound than those that do. So, one way to do this is to start with every process with some specified number of tickets. If a process blocks before using up its time quantum, give it another ticket (up to some set maximum, say 10). If it does not block before using up its time quantum, take a ticket away (down to some set minimum, say 1). In this way, processes that tend to do relatively little processing after each I/O completes will

have relatively high numbers of tickets and processes that tend to run for a long time will have relatively low numbers of tickets. Those that are in the middle will have medium numbers of tickets.

This system has several important parameters: time quantum, minimum and maximum numbers of tickets, speed at which tickets are given and taken away, etc.

The Details

In this project, you will modify the scheduler for Minix. This should minimally involve modifying code in **proc.c** and **proc.h**, though you may want to look at other modules to see how they fit together (you may, of course, modify code elsewhere if you wish). To test your code, you can create additional processes in **proc.c**.

You may want to also update **system.c** and **clock.c** to help track processes creation and execution time.

For lottery scheduling, you should assign tickets using ProcessSetPriority(int n), where *n* corresponds to the number of tickets given to the process. Each time the scheduler is called, it should randomly select a ticket (by number) and run the process holding that ticket. Clearly, the random number must be between 0 and $n_{Tickets}$-1, where $n_{Tickets}$ is the sum of all the outstanding tickets. You may use the random() call to generate random numbers and the srandom() call to initialize the random number generator (these calls function the same way they do in Unix).

For dynamic priority assignment, you should modify lottery scheduling to decrease a process's priority by 1 each time it receives a full quantum, and increase its priority by 1 each time it blocks without exhausting its quantum. A process's priority should never drop below 1 and should never exceed its original (desired) priority.

You must implement lottery scheduling as follows:

1. **Basic lottery scheduling**.
   Start by implementing a lottery scheduler where every process starts with 5 tickets and the number of tickets each process has does not change.
2. **Lottery scheduling with dynamic priorities**
   Modify your scheduler to have dynamic priorities, as discussed above.
3. **Profile your scheduler**
   At each timer interrupt, see how many processes are at each priority level, average this information over some large number of interrupts (maybe 100), then print it out. Conclude something about the settings of the various parameters: are they too long, too short, just right. Justify your conclusions.
4. [Extra Credit] **Compare to Linux**
   After you have profiled the performance of I/O bound and CPU bound applications in your scheduler, read up on Linux scheduling and discuss how closely your system

matches the performance of Linux in how it handles I/O and CPU bound processes. If anyone wants to do this one, ask me for information about the Linux scheduler.

Reminder: You must check and correctly handle all return values.

What to turn in

A compressed tar file of your project directory, including your design document. The design document should indicate what code modifications you made, also indicate why you made the changes. Additionally, the design document must provide the answers/details for 1 to 3 minimally, and possibly 4.

You must do "make clean" before creating the tar file. In addition, include a README file to explain anything unusual to the TA — testing procedures, etc. Your code and other associated files must be in a single directory so they'll build properly in the submit directory.

REMEMBER: *Do not* submit object files, assembler files, or executables. Only submit source files and the design document(s) with answers/details.