Project 2b: xv6 Scheduler

You may work with a project partner from your discussion section for the Scheduler portion of this project.

Objectives

There are three objectives to this assignment:

- To understand code for performing context-switches in the xv6 kernel.
- To implement a basic MLFQ scheduler.
- To show how process behavior (i.e., how long a process uses the CPU before performing I/O or sleeping) interacts with the scheduler by creating an interesting timeline graph.

Overview

In this project, you'll be implementing a simplified **multi-level feedback queue (MLFQ)** scheduler in xv6.

The basic idea is simple. Build an MLFQ scheduler with four priority queues; the top queue (numbered 0) has the highest priority and the bottom queue (numbered 3) has the lowest priority. When a process uses up its time-slice (counted as a number of ticks), it should be downgraded to the next (lower) priority level. The time-slices for higher priorities will be shorter than lower priorities. There is also a mechanism to ensure that low priority jobs do not starve.

For this project, you should run xv6 on only a single CPU (the default is two). To do this, in your Makefile, replace `CPUS := 2` with `CPUS := 1`.

Details

You have three specific tasks for this part of the project.

Implement MLFQ

Your MLFQ scheduler must follow these very precise rules:

1. Four priority levels, numbered from 0 (highest) down to 3 (lowest).

2. Whenever the xv6 10 ms timer tick occurs, the highest priority ready process is scheduled to run.

3. The highest priority ready process is scheduled to run whenever the previously running process exits, sleeps, or otherwise yields the CPU.

4. When a timer tick occurs, whichever process was currently using the CPU should be considered to have used up an entire timer tick's worth of CPU. (Note that a timer tick is different than the time-slice.)

5. The time-slice associated with priority 0 is 5 timer ticks; for priority 1 it is also 5 timer ticks; for priority 2 it is 10 timer ticks, and for priority 3 it is 20 timer ticks.

6. When a new process arrives, it should start at priority 0.

7. If no higher priority job arrives and the running process does not relinquish the CPU, then that process is scheduled for an entire time-slice before the scheduler switches to another process.

8. At priorities 0, 1, and 2, after a process consumes its time-slice it should be downgraded one priority. After a time-slice at priority 3, the CPU should be allocated to a new process (i.e., use Round Robin with a 200~ms time-slice cross processes that are all at priority 3).

9. If a process voluntarily relinquishes the CPU before its time-slice expires, its time-slice should not be reset; the next time that process is scheduled, it will continue to use the remainder of its existing time-slice.

10. After each 1 second interval, if a runnable process has not been scheduled at all in that interval, its priority should be bumped up by one level and given a new time-slice. Note that this 1 second interval is system-wide; at this same point in time, every runnable process is evaluated for starvation.

2) Create getpinfo()

You'll need to create one new system call for this project: **int getpinfo(struct pstat *)** .

This routine returns some basic information about each process: its process ID, how many timer ticks it has acquired at each level, which queue it is currently placed on (0, 1, 2, or 3), and its current `procstate` (e.g., SLEEPING, RUNNABLE, or RUNNING).

To do this, you will need to fill in the **pstat** structure as defined here: here. Do not change the names of the fields in pstat.h

3) Make a graph

You should make a graph (or set of graphs) that show some timelines of processes running with your scheduler, including which queue each process is on, and how much CPU they received.

To obtain the info for your graph, you should use the `getpinfo()` system call. Make up a workload (or set of workloads) that vary how long each process uses the CPU before voluntarily relinquishing the CPU (e.g., by calling sleep()). Think about what types of workloads will show interesting and useful results. Use the graphs to prove to us that your scheduler is working as desired.

To get full credit for this portion of the project, your graphs will need to show that all aspects of your scheduler are working correctly under a variety of conditions. You are likely to need to show multiple interesting workloads and the results must be easy for us to interpret.

Tips

Most of the code for the scheduler is quite localized and can be found in **proc.c** ; the associated header file, **proc.h** is also quite useful to examine. Another relevant file is **trap.c**.

To change the scheduler, not too much needs to be done; study its control flow and then try some small changes.

As part of the information that you track for each process, you will probably want to know its current priority level and the number of timer ticks it has left. You'll also need to track something to determine of the process is currently starving or not.

It is much easier to deal with fixed-sized arrays in xv6 than linked-lists. For simplicity, we recommend that you use arrays to represent each priority level.

You'll need to understand how to fill in the structure **pstat** in the kernel and pass the results to user space. The structure looks like what you see in here.

To run the xv6 environment, use **make qemu-nox**. Doing so avoids the use of X windows and is generally fast and easy. However, quitting is not so easy; to quit, you have to know the shortcuts provided by the machine emulator, qemu. Type **control-a** followed by **x** to exit the emulation. There are a few other commands like this available; to see them, type **control-a** followed by an **h**.

The Code

The source code for xv6 (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there.

You may also find the following readings about xv6 useful, written by the same team that ported xv6 to x86: xv6 book.

What To Turn In

Beyond the usual code, you'll have to **make a graph** for this assignment. You can use whatever graphing tool you would like ("gnuplot" is a fine, basic choice). When you have your graph, please create a .pdf version of it and place it in a file named **graph.pdf**. If you have multiple graphs, name them **graph1.pdf**, **graph2.pdf** and so one.

Please describe the workload that you ran to create your graph and explain why your graph shows the results that it does. You can either put this explanatory text directly in **graph.pdf** or in a separate file **workload.pdf** or **workload.txt** (if you use plain text). These are the only formats and filenames you should use.