Project 2b: xv6 Scheduler

Deadline: Mar 01 11:59:59 PM

Objectives

There are three objectives to this assignment:

- To familiarize yourself with a real scheduler.
- To change that scheduler to a new algorithm.
- To make graphs to validate your implementation.

Overview

In this project, you will be replacing the current scheduler in xv6 with a **stride scheduler**, a deterministic proportional-share scheduler, that is described in this chapter.

The basic idea is that the scheduler should pick a process with a **minimum accumulated stride value** (lets call this **pass value** similar to the naming in the book) to be run next. This gets into the two major question: How to calculate the stride and pass values for each processes. All processes in the system are assigned a certain number of tickets.

A stride value is inversely proportional to the number of tickets held by the process. For example, if three processes are assigned 100, 200 and 300 tickets, their corresponding stride values are 60, 30 and 20 (dividing 6000 by the tickets for all processes; you could choose any number to divide - the idea is to be divisible by all ticket values)

For every schedule quantum a process runs, its pass value is incremented by it's stride value. In this example, after every process runs for a quantum, their pass value would look like 60, 30 and 20. Now third process has minimum pass value and so, the scheduler has to choose third process to run next.

Details

You'll need two new system calls to implement this scheduler.

The first is `int settickets(int tickets)`, which sets the number of tickets of the calling process. **By default, each process should get the minimum tickets i.e. 10.** This system call could be used to raise or lower the number of tickets received by a calling process, and thus vary the proportion of CPU cycles it gets. The tickets should be in the range of 10 and 150 (10 <= tickets <= 150) and a multiple of 10. This routine should return 0 if successful, and -1 otherwise (38 (not a multiple of 10) or 0 (less than 0) or 160 (greater than

150)).

The second is `int getpinfo(struct pstat *)`. This routine returns some **basic information about each process**, including its process ID, how many tickets assigned per process, the current pass value for each process, the stride value calculated based on the tickets and the number of times each process was scheduled. It returns -1 on failure. You can use this system call to build a variant of the command line program **ps**, which can then be called to see what is going on.

[UPDATE] Your implementation of getpinfo(struct pstat *) is supposed to gather information on all processes in the system. xv6 could support up to NPROC (64) processes and so, the system call is supposed to receive information about all 64 processes. The inuse field in the pstat should be used to identify unused processes.

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.

Assumptions

Schedule Quantum: Every time xv6 context switches to the scheduler() function in proc.c, **you may assume a complete schedule quantum (10 milliseconds) has passed.** If programs only use CPU, this assumption is actually accurate, but programs that yield often (i.e., more often than every 10 ms), perhaps by waiting for I/O, will be overcharged. For example, if process A is scheduled to run for 10 ms, but it does I/O after 2 ms (and therefore yields), A's pass value will be charged entire schedule quantum (10 ms), instead of partial quantum (just the 2 ms) it used. **This is completely fine for this project.**

Reason for above simplification: Uptime is tracked with millisecond granularity in the **ticks** variable in trap.c. This variable is protected by the **tickslock** spinlock and is readable from userspace via the **uptime** system call. If you use ticks to measure time, you need to be careful to avoid deadlock. In particular, the prior code sometimes acquires `tickslock` before `ptable.lock`. Thus, you should never acquire the locks in the opposite order in the `scheduler()` function. Correctly accessing ticks requires using locks, which we have not learned about yet.

Beware of overflow: When the system is run for a longer period of time, the variable tracking the pass value could overflow. Care must be taken by your code to handle this if this to happen.

Number of CPUS: Remember that qemu is configured to expose two cpus to xv6 (so, xv6 will be running on two cpus). This should not impact your scheduler design but understand that if there are two runnable processes in the system with tickets 10 and 150, both could be running inspite of stride difference since there are two cpus.

Cases to handle

Process Creation: A new process would get zero pass value upon its creation since it hasn't run yet. However the problem is that, the new process would always be chosen by the scheduler to run since it has the minimum pass value until it catches up with other processes in terms of pass value. This means that you could always spawn a new process and game the scheduler allowing you to run forever. Your scheduler should not allow this behavior.

Example: Process A has been running for sometime and let us assume its current pass value is 200. Assume a new process B has been created with a stride value 5 (just an assumption). So, B gets to run for 40 schedule quantums before A could get scheduled.

Process Wakeup: Similarly, when a process wakes from a long sleep, it could run forever till it catches up with other process resulting in starvation. Your scheduler should handle this as well.

The problem occurs when processes are allowed to run with their native pass value i.e. zero when a process starts or old pass value (pass value of the process before it went to sleep state) in case of process wakeup. This could be averted by not using the native pass value and instead use a new pass value closer to the minimum among the pass values of all runnable processes in the system.

Hints

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

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.

The Code

The source code for xv6 can be found in **~cs537-2/public/html/xv6.tar.gz**. 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.

Particularly useful for this project: Chapter 9

What To Turn In

Beyond the usual code, you'll have to generate one or more graphs that help

you argue your implementation is correct. In a file called graphs.txt, give the file names of your graphs, and briefly explain how the graphs show your code is correct. For example, you could show (a) that process with higher number of tickets gets proportionally more cpu time than others. ~~(b) cpu time for a process gets reduced after reducing the assigned number of tickets (c) how your scheduler avoids the gaming problem by showing that the new process does not impact other processes much.~~

[UPDATE] You could use a sample user program that makes a call to getpinfo at regular intervals (implement a loop calling getpinfo() followed by a sleep for x units of time). Run a set (say 6) of processes with different ticket values and use the above method to gather the scheduling statistics for some period of time (say 2 minutes). Copy these scheduling information into an excel file and plot a graph with x axis as time and y axis as number of times a process was scheduled (n_schedule). Your graph should contain 6 lines where every line correspond to a process in this example. Create a pdf with the graph alone with the axises named properly and the legend showing the ticket values used for every process.