Project 2b: xv6 Scheduler

Objectives

- To understand code for performing context-switches in the xv6 kernel.
- To implement a basic MLFQ scheduler.
- To create system calls that extract and update process states.

Updates

Read these updates to keep up with any small fixes in the specification.

- One of the fields in `pstat.h`, `int ticks[NPROC][4]`, keeps track of how many ticks each process have **accumulated** at each of the 4 priorities. This means that the ticks can only increase **monotonically.** If the process reaches the lowest priority level and uses up the time-slice for that level, i.e. 64 timer ticks, then you should keep on increasing this field as long as this process is scheduled at that level. If the user sets the priority of this process and moves it to a different level, then the ticks field at that level will also exceed the time-slice of the associated level. In short, **do NOT reset this field at any time**!

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 3) has the highest priority and the bottom queue (numbered 0) 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.

To make your life easier and our testing easier, 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 two specific tasks for this part of the project. However, before starting these two tasks, you need first have a high-level understanding of how scheduler works in xv6.

Most of the code for the scheduler is quite localized and can be found in `kernel/proc.c`, where you should first look at the routine `scheduler()`. It's essentially looping forever and for each iteration, it looks for a runnable process across the `ptable`. If there are multiple runnable processes, it will select one according to some policy. The vanilla xv6 does no fancy things about the scheduler; it simply schedules processes for each iteration in a round-robin fashion. For example, if there are three processes A, B and C, then the pattern under the vanilla round-robin scheduler will be `A B C A B C ...` , where each letter represents a process scheduled within a **timer tick**, which is essentially ~10ms, and you may assume that this timer tick is equivalent to a single iteration of the `for` loop in the `scheduler()` code. Why 10ms? This is based on the timer interrupt frequency setup in xv6 and you may find the code for it in `kernel/timer.c`.

Now to implement MLFQ, you need to schedule the process for some **time-slice**, which is some multiple of timer ticks. For example, if a process is on the highest priority level, which has a time-slice of 8 timer ticks, then you should schedule this process for ~80ms, or equivalently, for 8 iterations.

xv6 performs a context-switch every time a timer interrupt occurs. For example, if there are 2 processes A and B that are running at the highest priority level (queue 3), and if the round-robin time slice for each process at level 3 (highest priority) is 8 timer ticks, then if process A is chosen to be scheduled before B, A should run for a complete time slice (~80ms) before B can run. Note that even though process A runs for 8 timer ticks, every time a timer tick happens, process A will yield the CPU to the scheduler, and the scheduler will decide to run process A again (until its time slice is complete).

1) Implement MLFQ

Your MLFQ scheduler must follow these very precise rules:

1. Four priority levels, numbered from 3 (highest) down to 0 (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. If there are more than one processes on the same priority level, then you scheduler should schedule all the processes at that particular level in a round robin fashion. Your scheduler may choose any of the ready processes at that level to be scheduled first. i.e. The order in which the processes with the same priority are scheduled wouldn't be tested.
5. 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.)
6. The time-slice associated with priority 3 is 8 timer ticks; for priority 2 it is 16 timer ticks; for priority 1 it is 32 timer ticks, and for priority 0 it is 64 timer ticks.
7. When a new process arrives, it should start at priority 3 (highest priority).
8. 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.
9. At priorities 3, 2, and 1, after a process consumes its time-slice it should be downgraded one priority. After a time-slice at priority 0, the CPU should be allocated to a new process (i.e., use round robin with a ~640ms (64 timer ticks) time-slice across processes that are all at priority 0).
10. If a process voluntarily relinquishes the CPU before its time-slice expires **at a particular priority level**, 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 **at that priority level**.

11. You need NOT implement the priority boost mechanism in MLFQ for this project. Yes, a process could starve and never receive any CPU if higher-priority processes keep arriving and it's OK.

2) Create new system calls

You'll need to create two system calls for this project:

- `int getpinfo(struct pstat *)`

Because your MLFQ implementations are all in the kernel level, you need to extract useful information for each process by creating this system call so as to better test whether your implementations work as expected.

To be more specific, this system call returns 0 on success and -1 on failure. If success, 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 (3, 2, 1, or 0), and its current `procstate` (e.g., `SLEEPING`, `RUNNABLE`, or `RUNNING`) will be filled in the `pstat` structure as defined here. Do not change the names of the fields in `pstat.h`.

- `int setpriority(int pid, int priority)`

Because there is no mechanism for the priority of a process to be raised again, a process could starve and never receive any CPU if higher-priority processes keep arriving. To resolve this issue, you can manually raise/lower the priority of the process from user mode by this system call.

To be more specific, this system call sets the priority (level) of the process with the specified pid. Do not reset the timer ticks of any levels of this process when the priority is updated by this routine.

This routine should fail if the provided pid does not exist or the priority to set is not 3, 2, 1, or 0. This routine returns 0 on success and -1 on failure.

However, user can game the scheduler by setting the priority of a process such that it gets scheduled forever. You may not worry about this case.

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. 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.

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 and how to pass the arguments from user space to the kernel. You may want to stare at the routines like `int argint(int n, int *ip)` in `syscall.c` for some hints.

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

We suggest that you start from the source code of xv6 at `~cs537-1/ta/xv6/`, instead of your own code from p1b as bugs may propagate and affect this project.

Particularly useful for this project: Chapter 5 in xv6 book.

Testing

Testing is critical. Testing your code to make sure it works is crucial. Writing testing scripts for xv6 is a good exercise in itself, however, it is a bit challenging. As you may have noticed from p1b, all the tests for xv6 are essentially user programs that execute at the user level. You may refer to these C programs in `~cs537-1/ta/tests/2b/` for some guidance of how to write your own tests.

For this project, we only release part (~75%) of the tests. This is a good chance to practice creating test cases and think comprehensively! That said, we still provided some descriptions about the workload and expected behaviors for the hidden test cases, which can be found in `~cs537-1/ta/tests/2b/project2b.py`. To run the test script that are released, go to your xv6 directory and type `~cs537-1/ta/tests/2b/runtests`.