Project 2b: xv6 Scheduler

Updates

Some test cases can be found here

Objectives

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

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. The scheduling method in each of these queues will be round-robin, except the bottom queue which will be implemented as FIFO.

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).

Implement MLFQ

Your MLFQ scheduler must follow these very precise rules:

- Four priority levels, numbered from 3 (highest) down to 0 (lowest). Whenever the xv6 10 ms timer tick occurs, the highest priority ready process is scheduled to run.

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

- 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. Except for priority level 0, which will be scheduled using FIFO basis.

- 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, even if it did not start at the previous tick (Note that a timer tick is different than the time-slice.)

- 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 executes the process until completion.

- When a new process arrives, it should start at priority 3 (highest priority).

- 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.

- At priorities 3, 2, and 1, after a process consumes its time-slice it should be downgraded one priority. At priority 0, the process should be executed to completion.

- 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.

- To overcome the problem of starvation, we will implement a mechanism for priority boost. If a process has waited 10x the time slice in its current priority level, it is raised to the next higher priority level at this time (unless it is already at priority level 3). For priority 0, which does not have a time slice, processes that have waited 500 ticks should be raised to priority 1.

Create new system calls

You'll need to create one system call 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 have elapsed while running in 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

```
struct pstat {
  // whether this slot of the process table is in use (1 or 0)
  int inuse[NPROC];
  // PID of each process
  int pid[NPROC];
  // current priority level of each process (0-3)
  int priority[NPROC];
  // current state (e.g., SLEEPING or RUNNABLE) of each process
  // see enum procstate
  int state[NPROC];
  // number of ticks each process has accumulated
  // RUNNING/SCHEDULED at each of 4 priorities
  int ticks[NPROC][4];
  // number of ticks each process has waited before being scheduled
  int wait_ticks[NPROC][4];
};
```

The file can be seen here. Do not change the names of the fields in `pstat.h` .

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.

Though, it is much easier to deal with fixed-sized arrays in xv6 than linked-lists. You may still try to investigate using linked-lists in your MLFQ, and it might be helpful to know how Linux kernel uses lists.

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.

Readings

- OSTEP (Chapter 8)
- xv6 book (Chapter 5)

Adapted from WISC CS537 by Adalbert Gerald Soosai Raj