P4: xv6 Scheduling

Due Oct 22, 2019 by 5pm **Points** 100

Updates

- **Each p**roject partner should turn in their joint code to **each** of their handin directories. Each person also needs to specify a partners.txt file. (Thu 10/17)
- A user program that includes "pstat.h" must not have to also include "proc.h" (or other miscellaneous .h files) to compile correctly. To avoid the problem where **procstat** is not defined, you can either have pstat.h itself include proc.h, or you can move the definition of **enum procstate** from proc.h to pstat.h. Depending on your approach, you may need to modify which header files are included where. You may also want to use **include guard**s in your .h files. See piazza for more info about include guards. **https://piazza.com/class/jyivrc1wvcv7dh?cid=7 (https://piazza.com/class/jyivrc1wvcv7dh?cid=7)** Our test programs will not compile correctly if you do not adhere to this requirement. (Thu 10/17)
- Make sure the very first process that is added to a queue is added to the tail, thus immediately incrementing the value of qtail to 1. (Thu 10/17)
- Why might getpinfo() fail (and return -1)? If the pointer passed in as a parameter does not point to valid, allocated memory. (Thu 10/17)

This project may be performed with **one project partner**. We recommend that each of you understand each portion of your code. This project does not entail writing too many lines of code, but it does require that you get everything exactly right. Thus, if you both work together on understanding the existing code and debugging your new code, you will probably work most efficiently.

This project is technically due at **5:00 pm**, but, if you are able to connect to the handin directory, you may turn in your project up through **midnight** that day.

Objectives

- To understand existing code for performing context-switches in the xv6 kernel
- To implement a basic multi-level round-robin scheduler where priorities are statically assigned
- To implement a system call that extracts process states
- To implement a user-level process that controls the scheduling of children processes

Overview

In this project, you'll be implementing a **multi-level queue scheduler** in xv6. This multi-level queue scheduler is easier to implement than the **multi-level feedback queue (MLFQ)** we discussed in lecture because your scheduler is missing the word **"feedback";** that is, the priorities in your scheduler are

changed not based on the dynamic behavior of processes, but are based on explicit system calls to modify the priority of each process.

The basic idea is simple. Build an MLQ 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 stays at its priority level, but is moved to the back of that queue. The scheduling method in each of these queues is thus Round Robin (RR). 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` ** . Make sure in your Makefile `CPUS := 1`.

Particularly useful for this project: **Chapter 5** **(https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** in xv6 book.

Background

Before implementing anything, you should first have an understanding of how the scheduler works in xv6.

Most of the code for the scheduler is localized and can be found in **proc.c**. The header file **proc.h** describes the fields of an important structure, each of the **proc structures i**n the global **ptable**.

You should first look at the routine **scheduler()** which is essentially looping forever. After it grabs an exclusive **lock** over ptable, it then ensures that each of the RUNNABLE processes in the ptable is scheduled for a timer tick. Thus, the scheduler is implementing a simple Round Robin (RR) policy. 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.

A timer tick is about 10ms, and this timer tick is equivalent to a single iteration of the **for loop** over the ptable in the **scheduler()** code. Why 10ms? This is based on the timer interrupt frequency setup in xv6 and you may find relevant code for incrementing the value of **ticks** in **trap.c**.

When does the current scheduler make a scheduling decision? Basically, whenever a thread calls **sched().** You'll see that **sched()** switches between the current context back to the context for **scheduler()**; the actual context switching code for **swtch()** is written in assembly and you can ignore those details. So, when does a thread call **sched()** ? You'll find three places: when a process exits, when a process sleeps, and during **yield().**

When a process exits and when a process sleeps are intuitive, but when is **yield()** called? You'll want to look at **trap.c** to see how the timer interrupt is handled and control is given to scheduling code; you may or may not want to change the code in **trap()**.

Other important routines that you may need to modify include **allocproc()** and **userinit().** Of course, you may modify other routines as well.

MLQ Requirements

Your MLQ scheduler must follow these very precise rules:

- **Four** priority levels, numbered from 3 (**highest**) down to 0 (**lowest**).
- The time-slice associated with priority 3 is 8 timer ticks; for priority 2 it is 12 timer ticks; for priority 1 it is 16 timer ticks, and for priority 0 it is 20 timer ticks.
- There are three events which may cause a new process to be scheduled: whenever the xv6 10 ms timer tick occurs, when a process exits, or when a process sleeps. At each of those three points, the highest priority process must be run.
- You should not trigger a new scheduling event when a new process arrives, wakes, or has its priority modified through setpri(); you should wait until a timer tick to schedule the highest priority process.
- If there are more than one processes on the same priority level, then you should schedule all the processes at that particular level in a Round Robin fashion. After a process consumes its time-slice it should be moved to the back of its queue. For example, if a process is at the highest priority level, which has a time-slice of 8 timer ticks, then you should schedule this process for 8 ticks before moving to the next process.
- 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.)
- When a new process arrives, it should inherit the priority of its parent process. The first user process should start at the highest priority.
- Whenever a process wakes or moves to a new priority, it should always be added to the back of that queue with a full new time slice.
- You will not implement any mechanism for starvation; if a process is never at the highest priority, it will never run.

New system calls

You'll need to create several new system calls for this project.

The first is **int setpri(int PID, int pri).** This sets the priority of the specified **PID** to **pri**. You should check that both PID and pri are valid; if they are not, return -1. When the priority of a process is set, the process should go to the end of the queue at that level and should be given a new time-slice of the correct length. The priority of a process could be increased, decreased, or not changed (in other words, even when the priority of a process is set to its current priority, that process should still be moved to the end of its queue and given a new timeslice). Note that calling **setpri()** may cause a new process to have the highest priority in the system and thus need to be scheduled when the next timer tick occurs. (When testing your pinfo() statistics below, we will not examine how you account for the case when setpri() is applied to the calling process.)

The second is **int getpri(int PID).** This returns the current priority of the specified PID. If the PID is not valid, it returns -1.

The third is **int fork2(int pri)**. This routine is exactly like the existing **fork()** system call, except the newly created process should begin at the specified priority. Thus, **fork()** could now be implemented as **fork2(getpri())** since by default the child process inherits the priority of the parent process. If **pri** is not a valid priority, fork2() should return -1.

The last is **int getpinfo(struct pstat *)**. Because your MLQ implementation is 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 implementation works 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 {
  int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
  int pid[NPROC];   // PID of each process
  int priority[NPROC];  // current priority level of each process (0-3)
  enum procstate state[NPROC];  // current state (e.g., SLEEPING or RUNNABLE) of each process
  int ticks[NPROC][4]; // total num ticks each process has accumulated at each priority
  int qtail[NPROC][4]; // total num times moved to tail of queue (e.g., setprio, end of timeslice, w
aking)
};
```

Most of the fields should be self explanatory. To be precise, **ticks** is the total number of timer ticks each process has acquired at each of the 4 priority levels; that is, ticks should be incremented exactly once for one process when a timer tick occurs.

The **qtail** field should be incremented whenever a process is moved to the back of its queue. Remember, a process is moved to the back of its queue whenever it finishes it time-slice, wakes, or has its priority set by **setpri()**. Important: This field should be incremented even if there is only one process at a priority level (e.g., a process A finishes its time-slice and it is moved to the tail of its queue; however, there is no other process at that priority so A is scheduled again; the value of **qtail** for A would still be incremented).

You can decide if you want to update your pstat statistics whenever a change occurs, or if you have an equivalent copy of these statistics in ptable and want to copy out information from the ptable when **getpinfo()** is called.

The file should be copied from **~cs537-1/projects/scheduler/pstat.h**

Do not change the names of the fields in `pstat.h`.

User-Level Scheduling

To demonstrate that your scheduler is doing at least some of the right things, you will implement a toy user-level scheduler, **userRR**. The idea of a user-level scheduler is that a regular user-level process (probably running at high priority) controls exactly which of its children processes is scheduled by explicitly setting the priority of the chosen process to a higher level than the other processes.

Your implementation of userRR has the following arguments:

```
userRR <user-level-timeslice> <iterations> <job> <jobcount>
```

User-level-timeslice is the number of timer ticks (in 10ms intervals) that should be given to each of the child processes. **Iterations** is the number of times each job should be run. **Job** is the name of the job that userRR should start; you can assume that each job is identical and will run forever. **JobCount** is is the number of copies of **job** that should be started.

How to implement round-robin at user level? Probably start the userRR process at the highest priority. Begin by having the parent process **fork2()** and **exec()** each of the jobcount copies of job at a low priority. Then, pick one of the processes to schedule and set it to some other priority; go to sleep for the user-level timeslice; when the parent wakes, it is time to schedule a different process. When it is done with the specified number of iterations, userRR should kill all the child processes, call getpinfo, and print all the structures out.

To have jobs for userRR to run, you will also need to implement a second user-level program, **loop**. The job **loop** should not take any arguments and should just forever sleep for 10 ticks and then print out its **pid.**

For example, if the program is run as:

```
userRR 5 3 loop 2
```

Two instances of **loop** will be started; let's assume they are assigned **pid 12** and **13**. Pid 12 will be run for 5 ticks, followed by pid 13 for 5 ticks; then a second iteration with pid 12 for 5 more ticks and pid 13 for 5 ticks; finally, the 3rd iteration with pid 12 for 5 ticks and pid 13 for 5 ticks.

We will not test your implementation of userRR extensively; the purpose of userRR is primarily for you to test your implementation of your MLQ scheduler.

Tips

We recommend writing very small amounts of code and testing each change you make (e.g., by running forktest) rather than implementing too much functionality at one time.

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 (and modify). You will also want to look at **trap.c** to see how the timer interrupt is handled and control is given to scheduling code; you may or may not want to

change the code in **trap()**. 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. Somewhere you might also want to track if an event occurred that should trigger a scheduling event on the next timer tick.

It is much easier to deal with fixed-sized arrays in xv6 than allocating kernel memory. For simplicity, we recommend that you use statically-allocated memory to represent each priority queue (though you will need to move jobs within and across priority queues). Of course, your fixed-sized array may contain pointers to other structures if you find this useful; or, your statically-sized array could include pointers to act as a linked-list.

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. Good examples of how to pass arguments into the kernel are found in existing system calls. In particular, follow the path of `read()`, which will lead you to `sys_read()`, which will show you how to use `int argint(int n, int *ip)` in `syscall.c` (and related calls) to obtain a pointer that has been passed into the kernel.

Now to implement MLQ, you need to schedule the process for some time-slice, which is some multiple of timer ticks.

If learn better by listening to someone talk while they walk through code, you are welcome to watch this (old) video of another CS 537 instructor talk about how the default xv6 scheduler works. Note that the project for that semester is different than what you are implementing, but you might find the code walk through useful.

https://www.youtube.com/watch?v=eYfeOT1QYmg&feature=youtu.be (https://www.youtube.com/watch?v=eYfeOT1QYmg&feature=youtu.be)

(https://www.youtube.com/watch?v=eYfeOT1QYmg&feature=youtu.be)

Code

We suggest that you start from the initial source code of xv6 at `~cs537-1/projects` instead of your own code from p2 as bugs may propagate and affect this project.

Begin by copying this file **~cs537-1/projects/xv6-rev10.tar.gz** and gunzip and untar it. Also, copy **pstat.h.**

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.**

When debugging, remember that gdb can be quite useful!

When working with a project partner, you may want to be able to read and write the files in a shared directory on the instructional machines. The CSL machines run a distributed file system called AFS that allows you to give different file permissions to your project partner than to everyone else.

First, use the "fs" command to make sure no one can snoop about your directories. Let's say you have a directory where you are working on project 2, called "~remzi/p2". To make sure no one else can look around in there, do the following:

'cd' into the directory

```
prompt> cd ~remzi/p2
```

check the current permissions for the "." directory ("." is the current dir)

```
prompt> fs la .
```

make sure system:anyuser (that is, anybody at all) doesn't have any permissions

```
prompt> fs sa . system:anyuser ""
```

check to make sure it all worked

```
prompt> fs la .
Access list for . is
Normal rights:
system:administrators rlidwka
remzi rlidwka
```

As you can see from the output of the last "fs la ." command, only the system administrators and remzi can do anything within that directory. You can give your partner rights within the directory by using "fs sa" again:

```
prompt> fs sa . dusseau rlidwka
prompt> fs la .
Access list for . is
Normal rights:
system:administrators rlidwka
remzi rlidwka
dusseau rlidwka
```

If at any point you see the directory has permissions like this:

> prompt> fs la .
> Access list for . is
> Normal rights:
> system:administrators rlidwka
> remzi rlidwka
> system:anyuser rl

that means that **any user** in the system can read ("r") and list files ("l") in that directory, which is probably **not what you want.**

Testing

We recommend making sure that your scheduler can handle a large range of jobs. If you can run **usertests** and **forktests**, your scheduler is probably fairly robust (though unfortunately it might not be implementing the correct policy). We recommend next using your implementation of userRR to see if it has the expected behavior.

More information will be made available here later.

Handin

Each project partner should turn in their joint code to each of their handin directories.

So that we can tell who worked together on this project, each person should place a file named **partners.txt** in their handin/p4 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

To repeat, **both project partners should turn in their code and both should have this partners.txt file in their handin/p4 directory.**