CS 4375 Fall 2022

Shirley Moore, Instructor

**Lab 2: MLFQ Scheduler for xv6**

**50 points**

The xv6 implementation we are using does not have process priorities, and the scheduler uses a round-robin scheduling policy. Round-robin scheduling works well for a workload consisting entirely of interactive processes, but it is terrible for compute-intensive processes since it is the worst policy for average turnaround time. A multi-level feedback queue (MLFQ) scheduler attempts to optimize both response time for interactive jobs and turnaround for compute-intensive jobs by using the history of how jobs behave to adjust their priorities.

In this lab, you will learn how to measure and report CPU time and elapsed time for a process, You will implement CPU time reporting in the ps command, and you will implement a time command that executes a command and reports the CPU time and elapsed time it uses. Then you will implement and evaluate an MLFQ scheduler for xv6.

MLFQ scheduling is described in OSTEP Chapter 8. You should also look at the relevant parts of the xv6 book on scheduling: timer interrupts are explained in section 5.4; xv6 scheduling is explained in Chapter 7 (recommend looking at 7.1-7.4) and at the relevant parts of the xv6 source code. We will go over relevant parts of the xv6 code in class. The following YouTube videos should be very helpful as well:

[xv6 Kernel-10: Context Switching](https://www.youtube.com/watch?v=1sSanF_y8FY&list=PLbtzT1TYeoMhTPzyTZboW_j7TPAnjv9XB&index=10)

[xv6 Kernel-16: Scheduling + swtch.S](https://www.youtube.com/watch?v=-O_JX5mMMHY&list=PLbtzT1TYeoMhTPzyTZboW_j7TPAnjv9XB&index=16)

Important note: For this lab, we want to use just one CPU so that all processes are scheduled on the same CPU. To make this happen, in the Makefile, change CPUS := 3 to CPUS := 1.

**Task 1**. (10 points) Devise a workload for xv6 that consists of a combination of interactive and compute-intensive processes. For example, your compute-intensive processes might carry out a time-consuming process such as multiplying two large matrices several times. Your interactive processes might periodically ask for user input, access a file, or call sleep() to simulate I/O. Try to find a workload for which the current xv6 scheduler does not work well with respect to the metrics that are important for the workload (for example, average turnaround time for a workload consisting entirely of compute-intensive jobs). Show and explain your results.

An xv6 user program that multiplies two large matrices a user-specified number of times is provided for you in the file matmul.c. You can also use already existing interactive and short-running xv6 user commands as part of your workload (e.g., ps). You can run multiple xv6 commands simultaneously by running them in the background. Remember that in order to have the matmul program be compiled and available in xv6, you need to add it to UPROGS in the xv6 Makefile.

Doing Task 1 will motivate why we want to implement a different scheduling policy for xv6.

Q1.1. If you run one instance of the xv6 matmul program with an argument of 5 and no other user processes besides init and sh, what is the turnaround time for matmul? (Note that matmul self-reports its runtime. The matmul reported runtime is completion time minus time of first run, which will be the same as turnaround time if it starts running as soon as it arrives).

Text

Description automatically generated with medium confidence

Q1.2. If you run matmul with an argument of 5 in the background and run several other user processes while matmul is running (e.g., type ps;ps;ps;ps;ps;ps;ps a few times), how is the turnaround time for matmul affected? You could also try writing a ps fork bomb program, but be careful not to overdo it!

Q1.3. If you run two instances of the matmul program in the background, each with an argument of 5, with approximately the same arrival times, what happens to the average turnaround time

?

**The average run time increases due they are running at the same time**

**Text

Description automatically generated**

To run two instances at the same time in xv6, you can type the following at the prompt:

matmul 5 &; matmul 5 &

Compare with the average turnaround time if you don’t run the matmul instances in the background, i.e.,

matmul 5; matmul 5

but note that the arrival times are now different.

Q1.4. What scheduling policy changes could improve the average turnaround time for two matmul processes that arrive at approximately the same time?

**We can us Round Robin since this method will give a shared the CPU. But if we are using different values for matmul we can use LJF since the time will differ**

**Task 2**. (10 points) Change the struct proc data structure in proc.h so that it includes a field cputime for the total number of clock ticks that the process has been in the RUNNING state since the process was created. Update the cputime for the currently running process whenever a clock interrupt occurs. Change your ps command from Lab 1 to show cputime (in clock ticks) in addition to the other fields.

Q2.1. When and where in the kernel code will you initialize the cputime field in the struct proc for a given process and what should it be initialized to? When and where will you update this field?

**Should be initialize to an int because we are talking about time**

Doing task 2 will set up some of the information you will need to access to implement a different scheduling policy that is based on how much CPU time a process has used, as well as being able to display this information to the user via the ps command.

Sample output (your does not have to be formatted exactly like this):

$ matmul &

$ ps

pid state size cputime ppid name

1 sleeping 12288 1 0 init

2 sleeping 16384 0 1 sh

6 running 12288 0 2 ps

5 runnable 12288 9 1 matmul

**Task 3**. (10 points) Implement a time command the takes a command and the command’s arguments, executes the command, and, when the command has finished, outputs the CPU time, elapsed (also called wallclock) time, and the percentage of the CPU the process had while it was in the system. Your time command output will be similar to the /usr/bin/time command in Linux, except that you do not need to keep track of the user and system portions of the CPU time separately. Type ‘man time’ on a Linux system for more information about the Linux time command.

Q3.1. If you know CPU time and elapsed time, can you calculate CPU percentage from those? If so, how?

**We can follow this formula (process\_time/elapsed\_time)\*100.0**

Q3.2. How will the time command execute the command it is passed as an argument?

**Using exec**

Q3.3. Is there an existing system call that you can use to get the information needed to

calculate the elapsed time for the child command? What about the CPU time?

**We can use time since it give us reall time between invocation and the Cpu time of the system**

Q3.4. Do you need to implement any additional system call(s)? If so, what is needed? (Hint: see the Linux wait3 man page).

**We can use waitpid() to get the time of the process termination**

Implementing the time command will help you to collect data to use for evaluating a scheduling policy. It’s also useful to the user to find out the CPU time and elapsed time for their program.

Sample output (yours does not have to be formatted exactly like this):

$ matmul

Time: 35 ticks

$ time matmul

Time: 36 ticks

elapsed time: 36 ticks, cpu time: 36 ticks, 100% CPU

$ time matmul &; time matmul &

$ Time: 69 ticks

elapsed time: 70 ticks, cpu time: 35 ticks, 50% CPU

Time: 68 ticks

elapsed time: 69 ticks, cpu time: 35 ticks, 50% CPU

**Task 4**. (20 points) Implement an MLFQ scheduler for xv6 with three queues having different priorities. For this task you need not implement any policies to prevent starvation or gaming, although you may do so for extra credit (see extra credit Tasks 5 and 6 below). The detailed policies for your MLFQ scheduler are as follows:

1. When a process is first created, it should be placed at the tail end of the highest priority queue.
2. The process at the head of the highest priority nonempty queue should be selected to run next.
3. If a process becomes ready to run after voluntarily relinquishing the CPU (e.g., by performing I/O or sleeping), it should be placed at the head of the queue for its priority, but it should not preempt a running process.
4. If a process that does not have the lowest priority uses its entire time slice, it should be moved to the next lower priority queue.
5. When a process is moved to a different queue, it should be placed at the tail end of that queue.
6. Round robin scheduling should be used for processes that are at the lowest priority.

While still working with a single queue (i.e., the original process list), change the tick length (interval in start.c) so that there are multiple clock ticks (e.g., 10) per time slice so that you can do more fine-grained accounting. Let’s call the number of clock ticks per time slice TSTICKS. You may also want to shorten the tick interval in start.c. Add a tsticks field to struct proc to keep track of the number of ticks a process has used during its current time slice. You will also need to change the timer interrupt handling code so that it only does a context switch every TSTICKS ticks, rather than every tick.

Q4.1. Where will you define TSTICKS?

Q4.2. Where and when will you initialize the tsticks fields in the struct proc for a given process and what should it be initialized to? Where and when will you update this field?

Q4.3. Where and how will you change the timer interrupt handling code so that it only does a context switch every TSTICKS ticks, rather than every tick?

It is up to you to decide what to use for the number of timer ticks for the time slices for each priority, and it would be a good idea to experiment with different values. You may want to make the number of ticks per time slice for each lower priority queues be a multiple of TSTICKS so that you have to define only one constant.

You should decide how to adapt existing kernel data structures, system calls, and kernel routines, or what new ones are needed, in order to implement your solution.

Q4.4. What is the best data structure to use for the different MLFQ scheduler queues? How can you integrate the queue data structures with the process list? What kernel routines are needed to manipulate these data structures?

Q4.5. Where and how will you change the timer interrupt handling code to decide whether to do a context switch and in the case of a context switch, which process to schedule next?

Q4.6. Where in the code and how will you place a process that is being context switched out in the correct position in the correct queue?

Q4.7. How will you handle a process that goes into a sleeping state and then wakes up?

Q4.8. Do you need to keep track of the priority of a process separately from having it present in a queue? If so, how will you do this?

As part of this task, you should test and evaluate your scheduler with different workloads and show and explain your results.

Task 5. (5 points extra credit) Implement a priority boost policy so that possible starvation of compute-intensive processes is prevented. Evaluate the effect of using priority boost with different workloads and explain your results.

Task 6. (5 points extra credit) Implement a mechanism to prevent a process from gaming the scheduler by always relinquishing the CPU just before its time slice is up and thus staying at the highest priority. Evaluate the effectiveness of your gaming prevention mechanism with different workloads and explain your results.

## **Turn-in procedure and Grading/Demos**

­Convert your lab report to PDF format and push your lab1 branch with your code and report to your xv6 GitHub repository by the due date**.** Also turn in the assignment on Teams with the URL of your github repo by the due date. Give access to your repo to the instructor, TA, and IA if they do not already have it. **Late Penalty**: After the due date, 0.5 point will be taken off every hour up to the first 10 hours on the FIRST DAY ONLY. After the first 10 hours, 5 points will be taken off per day late.

This lab is worth 50 points. The breakdown of points is given in the task descriptions above. The points for each task will be evaluated based on correctness of your code, proper coding style and adequate comments, and the section of your lab report for that task.

You will be randomly selected for two lab demos during the semester. Students who are selected for demos will be informed by email after the due date, and a signup sheet for demo times will be provided. During the demo, you will be asked to explain and run your code. If you cannot explain how your code for a task works, you will not receive any credit for that task.

You may discuss the lab with other students, but do not share your code. Your code and lab report must be your own original work. Any resources you use should be credited in your lab report. If we suspect that code has been copied from an online website or github repo, from a book, or from another student, we will turn the matter over to the Office of Student Conduct for adjudication.