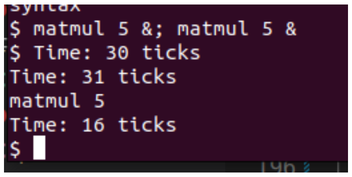
CS4375-17800 Fall 2022 Lab Report
Eric Smith Submitted on 10/05/22
essmith1@miners.utep.edu

Lab 2: MLFQ Scheduler for xv6

Task 1. Workload analysis

The main workload I used for this assignment was the given matmul function. This is a workload to use as it scales to O(n^3) time complexity and is easy to provide a heavy computational workload. Given the high turnaround times, this function should also stress the abilities of round robin scheduling and the pitfalls that come with this approach. Storing the process state and reloading the process state adds overhead to the processes and should cause them to have a higher turnaround time when compared to other scheduling policies.

Example MLFQ Scheduler:



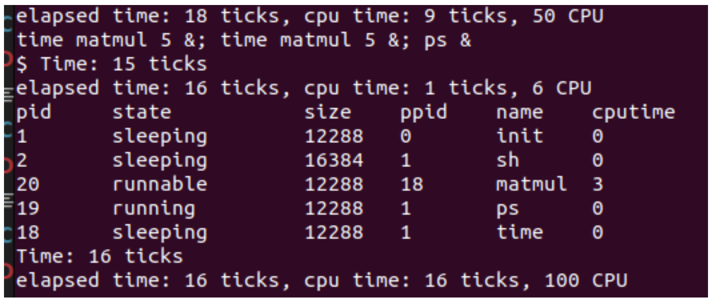
```
syntax
$ matmul 5 &; matmul 5 &
$ Time: 30 ticks
Time: 31 ticks
matmul 5
Time: 16 ticks
$ 
```

Task 2.

The main difficulty I had with implementing this method was the usage of the copyout() function. I was still confused about how to use this function from lab1, but the provided lab1 example provided with enough information to transfer the ticks field to the created pstat struct. This would allow us to store process information in user space and thus allow the information to be printed in the time command.

Once I looked at the provided reference for lab1, the solution became much clearer. From this solution, I realized my errors in copyout and was able to implement a time field relatively simply.

Example (Using MLFQ):



```
elapsed time: 18 ticks, cpu time: 9 ticks, 50 CPU
time matmul 5 &; time matmul 5 &; ps &
$ Time: 15 ticks
elapsed time: 16 ticks, cpu time: 1 ticks, 6 CPU
pid     state           size    ppid    name    cputime
1       sleeping        12288   0       init    0
2       sleeping        16384   1       sh      0
20      runnable        12288   18      matmul  3
19      running         12288   1       ps      0
18      sleeping        12288   1       time    0
Time: 16 ticks
elapsed time: 16 ticks, cpu time: 16 ticks, 100 CPU
```

Task 3.

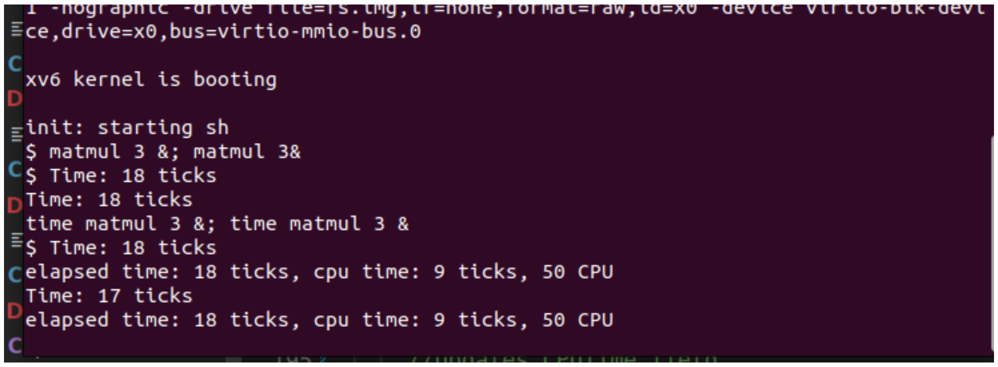
The time command outputs the name of the process, the ticks which have elapsed since the start of the process, the amount of those ticks were used on the current process, and the percentage of cpu time the process has used.

The implementation of this method went relatively smoothly. Incrementing the number of ticks a process has used can be easily implemented by adding one to the process ticks every time the process is interrupted by a user or a kernel timer trap. From this simple process, it is simple to generate the other values by computing some simple arithmetic.

As part of the creation of this function, it was necessary to create a wait function which would wait until a process has finished. This was relatively simple as the given matmul command provided reference in how to time a given process. The main modification to this was the ability to run any command instead of the static matmul command.

The hardest part of implementing a wait system call was remembering all the steps necessary to get xv6 to recognize the system call. I used the previous lab instructions as reference as I have not fully memorized the steps to add a system call in xv6.

Example(Using MLFQ):



```
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

init: starting sh
$ matmul 3 &; matmul 3&
$ Time: 18 ticks
Time: 18 ticks
time matmul 3 &; time matmul 3 &
$ Time: 18 ticks
elapsed time: 18 ticks, cpu time: 9 ticks, 50 CPU
Time: 17 ticks
elapsed time: 18 ticks, cpu time: 9 ticks, 50 CPU
```

Task 4.

Implementing a MLFQ scheduling process was the hardest part of this lab. First, it was necessary to add fields to the process and pstat structs. This would allow us to keep track of additional information such as how many tickets the process was going to run, the priority of the current process, and any other additional information.

Once I added the fields to the proc and pstat struct, I was able to get started on implementing the scheduling policy. First, I needed to add the necessary functions to get a queue working in c. I applied the initialization of the queue to where the initialization of the process page table occurred. Once a queue had been initialized, I was able to start adding processes to the queue.

Every single process in the queue would be started at the highest priority possible. In every instance in which the process became runnable in the proc file, I added the process to the end of the queue. The only exception to this would be when the process gave up it’s time slice for an I/O operation. Processes which gave up their time slice would be queued to the head of the queue.

I also needed to make sure to lower the priority of a process if it used up the entirety of its time slice and was forced to stop due to a timer interrupt. The way to detect this was to assign a temporary variable in the trap file which would be set to TSTICKS of the currently running process. This temporary variable would be iteratively subtracted from until it was eventually decided that the process had fully used it’s time slice. The priority of the process would be lowered, and the next process would replace the temporary variable.

Summary:

The MLFQ scheduler I implemented worked, but was relatively rudimentary. There are a couple of optimizations I could have made to improve the performance of the scheduler. First off, I could implement more queues for the processes to run in. This would allow the effect of the MLFQ scheduler to be more readily apparent. I could also fine tune the fields of the TSTICK values to more accurately reflect the type of behavior I would want out of a MLFQ scheduling policy.

This lab helped me learn about the variety of approaches scheduling policies. I also learned about the specifics of how timer interrupts work in xv6.