CPRE 308 Lab Report
Final Project Part 1 - Process Scheduler

Written By:
Quinn Murphy
Ben Williams

Section D
Friday at 11:00 AM

Introduction

The purpose of this lab was to begin introducing fundamentals of operating systems to the students of CprE 308, and implement those concepts inside a linux environment. This week's lab focused on task scheduling, namely different task scheduling algorithms and how they are used. We programmed Shortest Time Remaining and Round Robin separately, and then wrote Priority Round Robin together.

Design

Shortest Remaining Time Next

This algorithm was all about correctly adding tasks to the task list in the proper order. As long as the task list is properly sorted by estimated time remaining, the algorithm was not terribly complex to implement.

In isu_add_task(), I used the provided iterator function to compare the task being added with each task inside of the list, and add it in such a way that it is after a shorter time but before a longer time. If the list was null, I simply pushed the task onto the head of the list. If the task was longer than every other item inside the list, I pushed the task to the tail of the list and returned.

In isu_get_next_task(), I had to update the logic for the case that the task is being run but not removed as the running task. This is because the running task will be interrupted by another task at any point in time if the queued task has a shorter estimated run time. I simply popped off the head of the list, compared it to the running task remaining time, and then pushed the higher value back onto the list and set the running task to the lesser of the two.

Round Robin

The round robin algorithm was extremely easy to implement. Because round robin just cycles through tasks, I pulled the add_task function code from the given fcfs code and just appended the new task to the end of the queue.

For the get_next_task function, I determined that there were essentially three different situations that the process could be put in which could easily be segmented as if-statement blocks. First, a process is running but has not yet hit its quantum. In this case, the process just keeps running and the scheduler does nothing. The next situation is that a process is running and is past its quantum. In this case, it gets pushed back onto the tail of the queue and a new process is pulled off of the front and begins to run. The final situation is where the process has finished. In this case, the process is not put back into the queue and it is simply forgotten about as a new one is pulled from the queue and begins to run.

Priority Round Robin

This algorithm was not terribly difficult to implement once we had code from the previous algorithm to draw upon. The concept is that tasks of a higher priority will always be run before tasks of a lower priority, and a round robin style scheduling is run on the highest 'tier' of these tasks.

For isu_add_task(), we looked at the priority property of the task being passed in, and pushed it to the tail of the corresponding task_list with that priority, just like in normal round robin.

For get_next_task(), we again pulled code from the round robin implementation, and made a few changes. Firstly, before we accessed the list in any case, we ran a for-loop to determine the highest priority row and then popped from that row. If all of the rows were empty, then the IDLE task was selected. Beyond the for-loop and adding the index to the task_list item, this code did not change much from the round robin implementation.

Results

First Come First Serve

This algorithm ran each process that came into the task list in the order that they arrived. It ran each process to completion, and didn't care about length of time. It was a good reference program for us to see how the most basic implementation was done. When you upped the frequency of tasks arriving, the output didn't change since the algorithm ignores the task_list until the running task has gone to completion. When we changed the length of input tasks, the output changed in an expected manner. Tasks would always run for the time specified, in order, and it didn't affect the efficiency or run time of the final algorithm very much, unless we really cranked up the estimated run time.

Shortest Remaining Time Next

The basic output of this algorithm would run the shortest task to completion, only interrupting that task if another shorter task entered the task list while it was running. I tried changing the input frequency of processes, and this would make the running task be interrupted more often. When tasks come in bursts, it is often common that long tasks will never get run until the very end of execution. When there are many tasks that have the same time requirements, the algorithm behaves somewhat like round robin, because each running task will run for its length uninterrupted.

Round Robin

The round robin algorithm as implemented successfully rotated through all available processes while running. Our implementation actually ended up working on the first try, so no problems arose while testing that. I tested decreasing the quantum on this algorithm, and it seems that when many processes are running, short processes tend to finish earlier than they otherwise would have. That being said, longer processes run for a little longer because they have to share equal amounts of time with the shorter ones instead of running for a long time and giving up a little amount of time to let the others complete. As the quantum gets longer, the round robin essentially turns into a first come first serve scheduler.

When the start-time of processes is short (i.e. they all come in early), round robin does a decent job of handling them at first, but then as it attempts to divide the time evenly among the processes, it all starts to go downhill. Some processes wait forever to even be ran once, and for processes that began running early on but didn't finish, they now have to wait for tens of other processes to run just to get their last

few quanta in. Longer start-times didn't pose a problem as again the scheduler essentially became first come first serve.

Round robin did really well with short process lengths, as all of them would finish in about one quanta and was acting like first come first serve. Longer processes posed the same issue as the short start-times.

Priority Round Robin

The priority round robin algorithm as implemented successfully ran all higher priority (lower numbered) processes first and rotated through each priority's queue if there were multiple processes. The only issue we ran into with this implementation was deciding when to run the loop to check which the lowest priority currently available was (which we eventually decided was after the push of the current process into the queue). Just as with the round robin scheduler, short processes finished faster and long processes finished later. An interesting experiment would be one where the quantum for high priority processes is shorter than those with lower priorities.

Priority round robin ran very similar with the start-time and length modifications as above, though there was less "global" waiting as the processes with high priority finished, left the queue, and didn't leave possible crucial operating system functions waiting. The low priority processes however remained in the queue just about as long as in the previous round robin implementation.

Conclusion

All in all, this was a very cool lab to be able to look at how operating systems schedule tasks and create some simple implementations. Having a partner was very helpful as well, as having a second set of eyes on a problem can really save some time and headache.