Project Part 1 Lab Report: CprE 308

Matt McKillip & Lee Robinson

1. Introduction

In this lab, we will explore different types of scheduling algorithms and see their results as well as create our own algorithm and test its functionality. The first part of this project is to give us experience with the core features of a real operating system. We have gone over scheduling in lecture, so we are familiar with the concepts we will be implementing during the lab. After the completion of this lab, we should have a better understanding of how scheduling algorithms work and how to test their output.

2. Results/Output

Shortest Remaining Time & Round Robin

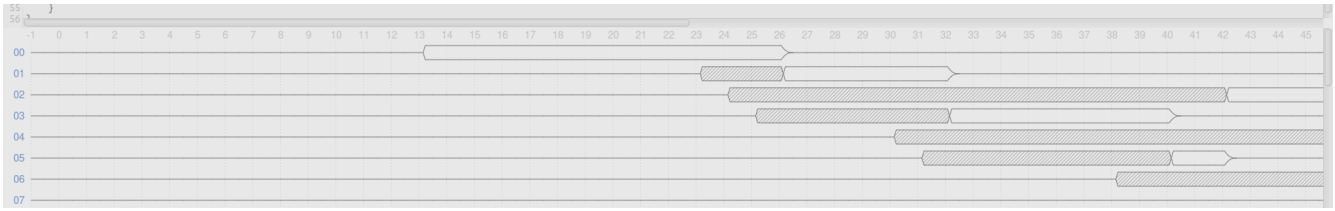
The output of our SRTN scheduler performed as expected. This algorithm is more efficient than FCFS, however more complex. The output of our RR scheduler also performed as expected. This builds on STRN and FCFC increasing the efficiency and complexity. However, it's not optimal.

Priority Round Robin & Going Further

Like the others, the output from PRR matched our expectations. This improves on the already effective RR approach and adds priority queues. We consider this scheduling algorithm to be the best. By modifying the time when processes and start and how long they run, we saw a variety of different outputs. Generally, the longer the processes the longer the overall time the scheduling algorithm took to run. If we created a lot of processes with the same start time, it made our scheduling algorithms go to work. For example, for STRN it had to check all of these processes to tell which one had the shortest remaining time. When we increased the quantum of the round robin, we noticed that the overall time was faster.

Extra Credit

The JSON output shows how the odd processes (01, 03, and 05) are being run before any of the even process despite their arrival times.



3. Design Decision

Shortest Remaining Time

For the srtn_add_task function, we first check if the method was called on a null object or if the scheduler has not been started. Next, we iterate through the task list to find where to place the task. Once we determine where to place the task, we push it into the correct place in the list. Finally, we print out that the task has been added successfully.

For the srtn_get_next function, we first repeat the same error checking. Next, we check the value of the remove parameter. If remove is 0, we pop off the first task in the list and evaluate it. If this task is null we know the running task is null. Next, we check to see if this task's remaining time is less than the running task. If this is the case, we add the running task back to the list and set the new running task to the popped off task. If remove is 1, we know to immediately pop off the head of the list and set it to the running task.

Round Robin

For the rr_add_task function, we first check if the method was called on a null object or if the scheduler has not been started. Next, we push the task parameter onto the tail end of the task_list. Finally, we print out that the task has been added successfully.

For the rr_get_next function, we once again begin with out error checking. Next, we check if the running task isn't null and that it has remaining time left. If this is the case, we add the running task into the list. After that, we set the running task to null. Finally, we pop off the head of the list and set it equal to the running task.

We based our design off the methods used in shortest remaining time, while modifying the usage to satisfy the needs of the round robin scheduler.

Priority Round Robin

For the prr_add_task function, we implemented it very similarly to the fcfs_add_task function. The only difference was the task was pushed to the task_list[priority], since there are separate queues for each priority.

For the prr_get_next function, we checked if the running task had stopped with time remaining. We needed to check this because this task needed to go onto the end of the queue to be run later. Then we looped through the queues starting at the highest priority queue checking if there is a task to run. Once we found the first task, we exited the loop and set our new running task.

The decision to keep it very similar to the given fcfs algorithm was to keep the development time down, knowing that the fcfs algorithm given was correct.

Extra Credit

For the extra credit scheduler we decided to implement a scheduling algorithm that prioritizes every other process – in our case odd processes. This is very similar to a prr, but with a only two priorities and the priorities are set based on if that is an even process or an odd one. We also decided to have a very high quantum to see the output a bit more clearly.

4. Issues

We had a few issues figuring out how to access the task object. We first tried accessing it like a pointer, but that caused a segment fault. Then we figured out that the task object was not a pointer, which solved our problem.

When implementing the priority round-robin, I had an issue accessing the running_task. I discovered that I needed to first check if the task was NULL, and if it was to skip the line of code.

While implementing the shortest remaining time, we were initially looping through the task list in both the add_task and get_next task functions. We realized that we only needed to loop once, and simply evaluate the first element of the list in get_next.

5. Conclusion

Overall, this lab was a success. We got to see the applications of scheduling algorithms and experience creating our own algorithm for extra credit. We also gained more experience using linked lists and compiling libraries. This lab gave us an insight into some of the core features of a real operating system, which helped give the lecture material some value.

6. Suggestions

We have no suggestions for this lab due to the well documented example files given and the straight-forward instructions given in the lab.