Michael Liao
CSC 280
Spring 2022

CPU Scheduling Lab Report

Compiling:

No special libraries are required to compile this C program. All you need to do is use the GNU compiler command as follows:

```
gcc cpuscheduler.c -o cpuscheduler
```

Running:

Ensure that a text file named `data.txt` exists in the same directory where this program has been compiled, with the formatting as indicated in the directions. Then, simply run it by typing:

```
./cpuscheduler
```

Summary:

In order to create simulations for multiple CPU scheduling algorithms, I had to first understand how the data relating to the jobs would be formatted so I could read it in. That was addressed with a function called `read_file()` which would take an array of structs of type `job` to fill in with repeated calls of `fscanf()` accessing a file named `data.txt`, mentioned earlier (generated using a separate program which randomly generated durations and submission times); the function also returned a value representing the actual number of jobs submitted so the simulations could run efficiently and not iterate through a potentially empty array. A struct `job` has several integers associated with it, including `id`, `time_submitted`, `duration`, `completion`, and `time_finished`. These should be fairly self-explanatory; they allow the program to allocate one index in the array of structs and store all the associated data with it.

Each type of algorithm was given its own function for organization's sake (e.g. `sim_rr()` or `sim_fcfs()`) and called in the order given in the directions for this assignment. I had to write an additional function called `reset_data()` (called at the beginning of every simulation) in order to reset the `completion` and `time_finished` after a function has modified them in order to calculate the average turnaround time and ensure the sim detects when a job is completed properly.

Each function makes good use of for loops in order to iterate through the array of jobs submitted and operates by incrementing a variable called `time` and the `completion` property of the job it is currently focusing on. I check to see if a job is completed once its `completion` property (starting at 0, counting up) is equal to its `duration` property and then filter it out using an if statement checking for future jobs. I also "queue" up jobs by checking the `time_submitted` property and comparing it to the current `time` variable to see if it truly has been submitted. Each time I switch jobs (by changing the index variable `i` and swapping jobs) I also increment a `context_switches` variable that gets initialized to 1 to keep track of context switches. Then, once it's iterated through the whole jobs array and completed all of them, the function simply needs to go through and total up turnaround time (by calculating a job's `time_finished` - `time_submitted`) before dividing by the total amount of jobs. It can then print the results and exit, allowing the program to move onto the next simulation.

Results & Analysis:

With a dataset of 2000 jobs submitted, these were the results:

```
FCFS turnaround time: 5559.082520
FCFS context switches: 2000
SJF turnaround time: 3964.565918
SJF context switches: 2000
SRTN turnaround time: 3964.542969
SRTN context switches: 2036
RR turnaround time: 8256.495117
RR context switches: 13141
```

The worst-performing algorithm by far was the Round Robin algorithm, coming in nearly twice as time-consuming as the SRTN/SJF algorithms. This issue is further compounded by the about 6 times more context switches the Round Robin algorithm had with its associated overhead.

The best (shortest) turnaround time was achieved by the Shortest Remaining Time Next algorithm, although the difference between it and the Shortest Job First algorithm is quite minute. Tests with other seeds for the random data generator seem to indicate that this is the case for both these algorithms with other datasets, with the largest observed difference being 3 time units.

However, though the SRTN algorithm was indeed the fastest, it did have 36 more context switches than the SJF algorithm due to it being a preemptive algorithm, allowing jobs to be interrupted on occasion. This overhead may slow it down so the

two are on par, if not SJF even being faster than SRTN. To optimize the time the computer spends working on jobs and not on switching between them, I would recommend using the SJF algorithm for its lesser context switches and similar average turnaround time.

Shortcomings:

This conclusion being reached, this simulation is not perfect. Average turnaround time is a flawed method at best of determining the efficacy of a given algorithm; it ignores the outlier jobs like in SRTN where a job might be held off till the very end to run due to its large duration. In an environment where jobs are continually being submitted, a long enough job may never get priority to run. While averages tend to be relevant in looking at the whole of all the jobs, those outlier jobs must be accounted for as well. In an extreme example, if all jobs submitted only required 1 time unit to complete save for the first, which required 10, that job might not complete until all the others had completed under SJF and SRTN; so while the other jobs might have turnaround times of 1, the longer job would have a turnaround time in the thousands. The average turnaround time may seem quite good, but for that one job which is longer than the rest, a job that shouldn't take much more than 10 time units will take much, much longer.

A better method of analysis would be to collate all the individual turnaround times into its own dataset and graphically display the data. That way, analysts could see where outliers end up and see how the duration of a job affected its position in the scatter plot given a specific algorithm of scheduling, as well as see a general trend that might give similar information to calculating average turnaround time.