CMPS 455 Project 3

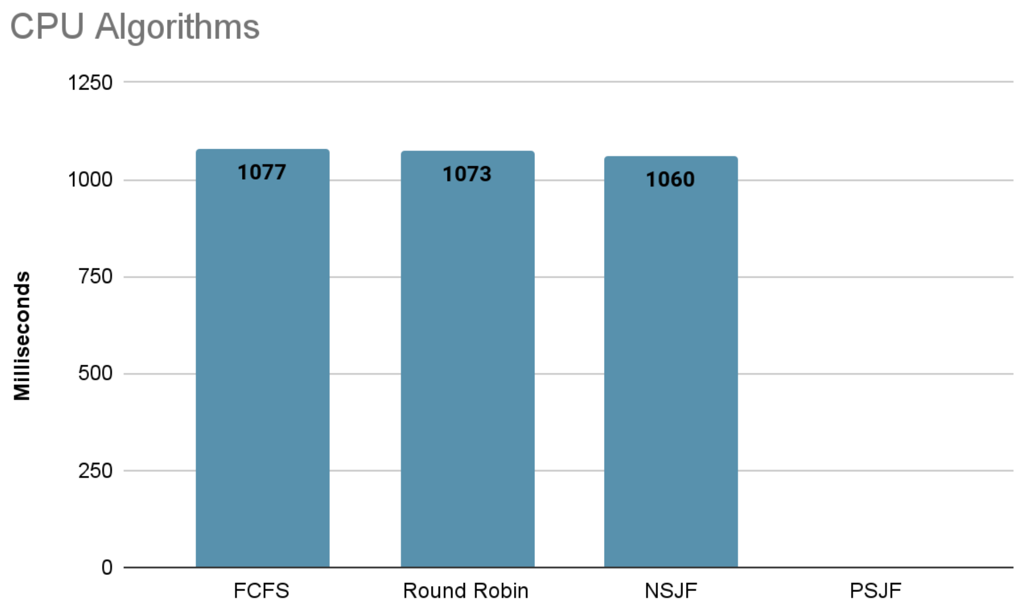
Jacob Tilmon	C00292879
Amy Canelas	C00416506
Ruby Shrestha	C00451990
Jessica Espree	C00081195

As part of your report, answer the following question about Task 1:

1. Have your program create 5 threads with burst times of 18, 7, 25, 42, and 21. Then, run your program with each of the four scheduling algorithms (for RR, use a time quantum of 5). Which scheduling algorithm was fastest? Provide a table, with columns for algorithm and runtime. (Remember: revert to random task creation and burst times before you submit the project)

According to our program NSJF was the fastest. However, in the long term round robin would terminate in a faster response time. Based on the table provided below, whenever we ran our different schedules, NSJF finished in 1060 milliseconds. Yet theoretically as time increases and more threads are introduced, the efficiency of NSJF will decrease drastically in comparison to RR. The reasoning being that in NSJF it introduces a starvation problem with the longer runtime processes, meanwhile RR will tackle the processes in cycles breaking down the burst time in a set amount of time quantum.

This logic is further proven to be correct when looking into time complexities. NSJF has a time complexity of O(N), while RR has a time complexity of O(1). Therefore the efficiency of algorithms will switch as soon as larger problem sizes are introduced, as we saw displayed in our program.



As part of your report, answer the following questions about Task 2:

1. Which scheduling algorithm was most efficient for use in a multi-core system? Why?

   As stated in task 1 question 1, round robin was the most efficient because it allowed shorter tasks to be completed quickly, while also working on the longer tasks. Round robin created a way for the burst time to be broken down in a set amount of cycle.

1. Which algorithm was the most difficult to implement for a single-core system and for a multi-core system?

   While building the project we found that PSJF was the most difficult to implement for single core and RR algorithm for multicore. The reason why PSJF was difficult was based on the logic of simulating the new threads and adding the interruptions between the threads. While this works logically the implementation is difficult compared to other schedules. The reason why RR was difficult as compared to other algorithms became apparent when implementing this specific algorithm for multi cores. Say we had N number of cores, then we would also have N number of dispatchers which need to select threads to run it in the CPU class for a given time quantum. If the thread's burst time was higher than the time quantum, the ready queue would have to be updated accordingly. So, it involved more updates to the ready queue and that made it more challenging.

2. In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them? If you failed to complete any tasks, list them here and briefly explain why.

   The implementation based on the project goes as the following; We created four arrays of semaphores named thread start, thread finish, dispatcher, and cpu. After creating the arrays we created the cpus and the dispatchers. The dispatchers are based on the number of cpus. From there we created the tasks and forked them to start the simulator. Once all the elements have been created we used the idea behind the producer and consumer problem. The dispatcher accesses the cpus and the cpus then processes the tasks.

Bugs

- Originally, the manner in which we had decided to access our program threads within the ready queue was very dangerous due to clones of tasks being created when the scheduler chooses RR. These cloned tasks would lead to a deadlock issue when the last process is in the ready queue.

  - In the new implementation, we decided to introduce a variable named Task currentTask and set it to null after it checks the condition currentTask != null and currentBurstTime<totalBurstTime. The key to making this code work was using the concept of a temporary value. We would use a placeholder value in the ready queue where it avoided the original issue of having cloned tasks.

- Another bug that we encountered was the CPU class would start new tasks before tasks were finished.
  - Our solution we decided to implement an arraylist of semaphores called “taskThreadSemsFin”, in which would only be called when the task finished within the TaskThread class. This will have a ripple effect and release our semaphore in the CPU class. Previously we had our arraylist of cpu semaphores, yet this was not enough to maintain the integrity of our desired program flow.

3. What sort of data structures and algorithms did you use for each task?

Data structures

1. Array list
    - Many are scattered through our program. This was the most prevalent data structure our group relied on to implement our idea.
        - Main
            - 1 arraylist of type TaskThread (Ready Queue)
            - 3 arraylists of type Thread (Dispatcher, Task and CPU each)
            - 4 arraylists of type semaphore

Alg

1. Producer Consumer logic
    - Between the Dispatcher, CPU and TaskThread class, we had a ripple effect among the three.
        - Dispatcher would release needed semaphore in the CPU, which would release needed semaphore in TaskThread. In TaskThread it would then release back to the CPU, and finally release Dispatcher again. Semaphores would originally be set to 0, therefore they require a release from a previous class so that they can proceed to acquire and subsequently go to sleep after doing what is needed.

2. FCFS
    a. First Come First Serve processes the tasks as they arrive, this scheduling is not optimal in regards of efficiency

3. NSJF
    a. Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
4. RR
    a. Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
5. PSJF
    a. We did not implement this algorithm.