Aaron Delahoussaye (C00449804)

Allen Sanders (C00496790)

Michael Vedol (C00459436)

Riley Young (C00445412)

Project 3 Report

Task 1: Single Core (Allen Sanders and Michael Vedol)

Runtime Table:

Five Threads: T1-18, T2-7, T3-25, T4-42, T5-21

|  |  |
| --- | --- |
| Algorithm | Completion Time (in milli-seconds) |
| FCFS | 17, 15, 17: AVG = 16 |
| RR | 27, 24, 23: AVG = 25 |
| NPSJF | 19, 20, 20: AVG = 20 |
| PSJF | 22, 19, 18: AVG = 20 |

The Fastest Algorithm: FCFS

Task 2: Multi-Core (Aaron Delahoussaye, Riley Young, Michael Vedol)

Most Efficient algorithm: Red Robin

Why: Round Robin is considered the most efficient scheduling algorithm for multi-core systems due to its fairness, effective load balancing, and prevention of process starvation, all of which contribute to improved performance and responsiveness.

Task 3: Command Line (Riley Young)

The command line accepts -S (1-4) and -C (number) inputs in any order, outputting an error if an invalid input is inputted. The most difficult part was trying to figure out how to make it accept -S and -C in any order, but other than that it was fairly easy to set up.

Task 4: Report

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

For single-core systems, the most difficult to implement was the Preemptive Shortest Job First Algorithm. For multi-core systems, the most difficult was the Round Robin Algorithm. Having the arguments being in the right position for if -C was before or after the algorithm call was tricky at first, but eventually easy to understand.

1. 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.

Single Core:

* Dispatcher\_Single () – The dispatcher single class accepts the parameter of a Queue of taskthreads, a time quantum. A semaphore, and the int represented as the algorithm decision used in a switch loop later.
* ThreadCreation() – Generates the number of threads and burst times used in the execution of the algorithm.
* DisplayQueue\_i(): Gets the the number of threads and burst times from threadCreation to print out the contents of the ready queue.
* FCFS\_Single() – While the ready queue is not empty. The algorithm selects the items in the front of the queue removes it once completed. The next item executes.
* RR\_Single() – While the ready queue is not empty, the algorithm runs the task at the front for a time quantum. If the task is completed, it is removed from the queue. Otherwise, it is added to the end of the ready queue until it is completed.
* NSJF\_Single() – While the ready queue is not empty, the algorithm selects the task with the shortest burst time. The selected task runs to completion without being interrupted, and once finished, it is removed from the queue. The next shortest task is then selected and executed until the queue is empty.
* PSJF\_Single() – While the ready queue or priority queue is not empty, the algorithm checks for new tasks arriving with shorter burst times. The task with the shortest remaining burst time is always selected for execution, and if a new task arrives that has a shorter burst time than the currently executing task, the current task is preempted and placed back into the queue. The task with the shortest burst time is then run until completion or further preemption occurs.

Multi-Core:

* Mostly the same logic for the system above
* Added multiple “CPUs” to the system to run the threads in parallel. Had clock index trackers to determine which thread went to which “CPU”.

Bugs: Some of the group members had an issue where they couldn’t read the java classes as class file, but were being read as java files. The src folder wasn’t selected as their source in the Project Structure menu and out was not showing. The members were able to open their project folder from a specific directory to get access.

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

Task 1:

* Queue – used to act as the ready queue for TaskThread objects. Used poll() and add() to remove elements from the front of the queue and adding noncompleted elements to the back of the queue
* Semaphores – used to protect access to the ready queue and access for task to run in the task thread.
* Priority Queue – implemented in PSJF algorithms

Task 2:

* Same as above