Emmanuel Valdiva

Matthew Gyure

Jaime Alvarez

CPU Scheduling Project

The project involved understanding some basic concepts with CPU scheduling and then implementing these algorithms. This was done by first creating a program that could generate data according to the input format. Once complete, the next process was to create a main file that could run two different types of scheduling. The main file proved to be a little bit more difficult because it required to read data from a different text file, but with a little tinkering it was done using BufferredReader. This enabled us to read by every character from the input file. Another problem was reading from the command prompt. This caused problems because setting the arguments equal to a definition required to use a special method which was the equals method. This method was used to check for an objects equality rather than a reference equality. Given the length of the main file, it was also important to clearly organize the file in a manner that would be readable and efficient. One way this was accomplished was by using different files to instantiate classes because it made easier to find certain fields and methods for later use for example the SimProcess file. Finally, the algorithms themselves needed to be implemented. To accomplish this, queues were created to reference the objects from the input files. These queues were then organized by arrival times using javas collection class. For First Come First Serve each process was handled according to its service time and then was placed in a doneQueue to represent a finished process. This was a simple algorithm, but it was still lengthy as it required to keep track of CPU time and idle time. For Round Robin a little bit of more work was necessary. It was still the same format, but now processes could only run according to the time slice provided. To keep track of which process was still left a separate queue was created to keep track of the corresponding process with their new updated service time. This temporary queue also received processes as they arrived within the CPU.

Since we only got exposure to two of the algorithms if it has to be between FCFS and RR we believe that RR is the best algorithm. Since RR was designed for time sharing systems it makes sense to go with this option. Unfortunately, Round Robin is not perfect and it does have certain limitations. For instance, RR is actually FCFS but with added preemptiveness. Specifically, if the time quantum is too large, RR degenerates to FCFS policy, and thus has no additional benefit. On the other hand if the time quantum is too small, RR will be required to do a lot of switching between processes and this lowers CPU efficiency. RR isn’t perfect, but when used within certain bounds it can provide better performance than FCFS.

The effects of decreasing context switch time to 1 time unit can definitely improve the processing power of the CPU. Ideally, it is best to have a small context switching time unit because processes can be executed faster, but is impossible due to the limitations of the hardware. If this would to be increased to 10, processes would take longer to execute and would dramatically effect CPU utilization because the system would be in idle a lot. In RR scheduling, a high context would be unbeneficial due to a larger increase in idle time as well.

For the extra credit, a problem that we encountered was reading input from the new format input file. It required some more logic to know how many CPU and IO burst times each process had. After finishing the new reading method, the next step was to modify FCFS algorithm. Luckily, the modification to the algorithm was pretty easy. Also, to make these new changes easy, we simply created new files that will handle specifically the new format input file.