4 Scheduling Algorithms implemented in Java

Project 1

Thomas Vermeersch
BroncoID: 009936940

Date: 10/17/2017

CS408

The goal and purpose of this programming project was to learn about different scheduling algorithms an operating system could use, and understand how they could be implemented. Through doing this we could see which scheduling algorithms are most efficient and which ones are less efficient. The four algorithms we covered were First-Come-First-Serve, Shortest-Job-First, Round-Robin with two different time quantum, and finally Lottery, also with a time quantum.

The first algorithm implemented was First-Come-First-Serve, this algorithm is incredibly straightforward, as the processes simply get scheduled in the order they are received. No extra memory in terms of using a data structure is required in this process, so it is extremely memory efficient. We simply get the process id, it's burst time and its priority as they come in. We simply "let the process run" until it's burst time is 0, and swap to the next process.

The next algorithm implemented was Shortest-Job. This algorithm takes a little bit of sorting, but the algorithm itself is relatively straight forward. The file is read, and runs the processes in order of shortest burst time. The easiest way to implement this would just be nested for loops, and brute force, loop through the burst times and running the shortest one it finds. However, if we store our burst times and process Ids in a sorted tree, like a treemap, then we simply pop the top of the tree, and run the process. This implementation is a little more efficient.

The next algorithm was Round-Robin, which simply looped through the processes and gave each one a certain amount of time to run. This amount of time is called the time quantum. The function for this algorithm takes it as an argument, so the time quantums this program uses are 30 and 60 units of time. To implement this, the processes are stored in a queue, the front of the queue is popped and runs for the time quantum. If it completes, then it does not return to the queue, if it requires more time than the time quantum gives, it goes to the end of the queue. This repeats until the queue is empty.

The final algorithm implemented was the Lottery scheduling algorithm. This algorithm gives each process Id a priority, the higher the priority, the more "lottery tickets" the process gets. That way the higher priority processes get to run more often, however each process still only runs for a specified time quantum. This was implemented with a 2d array, that kept track of the process, its burst time, and how many lottery tickets it has in a range. For example, a process might have lottery numbers (100-125). The algorithm then generates a random number between the amount of lottery tickets, and checks the array to see which process "won the lottery."

Using testdata1.txt as our data, the algorithms all had different average cpu-time taken per process. First-Come-First-Serve completed with every process averaging 509 units of cpu time. Shortest-Job-First completed with every process averaging 459 units of cpu time. Round-Robin gets tested with two different time quantums to see how the time quantum affects the average cpu time taken. When time quantum is set to 30, the average time per process is 558 units of cpu time. When time quantum is set to 60, the average time per process is 533, which shows that a longer time quantum can sometimes reduce average time per process, due to less swaps. The Lottery algorithm averages 538 units of cpu time for each process to complete.

In conclusion, the data has shown that the most efficient algorithm to use is the shortest job scheduling algorithm. However, if a time quantum is necessary, the best scheduling algorithm to use would be a Round-Robin scheduling algorithm with a time quantum that works best for the specific data.