CSC 139 - Section 06

Spring 2023

By: Santiago Bermudez

Project 2: Scheduling Algorithms

FCFS.java:

For this algorithm, tasks are scheduled in the order in which they request the CPU. For the class, I would declare a private List instance variable called queue that represents the queue of tasks to be executed. I would then add an FCFS constructor, which takes in a list of tasks and initializes the queue instance variable with the passed list of tasks. Because we are executing the tasks in the order that they arrived, things are pretty simple when it comes to writing this algorithm. For the schedule() method that is implemented from the Algorithm.java file, we simply pick each task repeatedly in order and simulate the execution of a task on a CPU while the queue is not empty. For our pickNextTask() method, we would simply remove and return the first task from the queue list, using "return queue.remove(0);". Overall, this algorithm was not too difficult to create, we just need to pick the tasks in the order in which they are read.

SJF.java:

For the SJF algorithm, tasks are scheduled in the order of the length of the tasks' next CPU burst. What this means is that we try to execute the shortest time-consuming tasks first. Like the previous algorithm, we would declare a private List instance variable called queue and then add an SJF constructor. For our scheduler method, because we are trying to execute the tasks in order of their burst lengths, we are going to want to sort the queue first. We do this in the schedule() method by using Collections.sort(). We can compare the burst times of the tasks by using the getBurst() function from the Task.java file. After sorting the queue, we would then simply need to use a for loop to execute all the tasks in order. Because everything is already handled by the schedule() method, we do not really need to use the pickNextTask() method this time. While there are many equally valid ways to go about this, this is just how I decided to do things.

RR.java:

For the RR algorithm, we schedule tasks in such a way that each task is run for a time quantum or for the remainder of its burst. For this code, we will be starting off with an approach like that of FCFS, but we will add more onto it. Because we will be using time quantums, we will be declaring a private integer quantum variable apart from the List instance variable called queue.

We will also have the quantum initialized in the constructor for this algorithm. Our schedule() method will use a while loop like in FCFS, but we will keep picking tasks until there are none and we will be constantly executing them in bursts and setting their bursts along the way. We will do this until all the tasks have a remaining burst time of 0. Because some tasks need multiple executions to complete, we will be adding them back to the queue as well.

Priority.java:

For the Priority algorithm, we will schedule tasks based on priority, regardless of how long they take to complete. We will have a List instance and a constructor like before. For our schedule() method, we will have an approach just like that of FCFS, where there is a while loop that repeatedly selects and simulates the execution of tasks on a CPU until the queue is empty. However, for our pickNextTask() method, we will be comparing tasks in the queue until we find the one with the highest relative priority. We do this using a for loop with an if statement to compare tasks.

PriorityRR.java:

For the PriorityRR algorithm, we would schedule tasks in the order of their priority and use round-robin scheduling for tasks with equal priority. For this algorithm, we will use an approach like that of the round-robin algorithm, but we will add on to it. Apart from declaring a List instance and a time quantum variable, we will declare a time variable as well. Our constructor will also initialize time. For our schedule() method, we will initialize an ArrayList of completed tasks to sort things out later. While our queue is not empty, we will repeatedly pick tasks and increment the time. We will be executing tasks for certain bursts and constantly set task bursts along the way as well as add their bursts to the time. Once a task has no burst time left, we would add it to the list of completed tasks and print out a message telling the user that the task is finished. Otherwise, we will add the task back to the queue. For our pickNextTask() method, we will use a for loop and if statement to compare tasks until we find the one with the highest relative priority. We will also use this method to make sure that there are no tasks with 0 burst time left in the queue.