Report

Short Job First

For sjf (short job first) we used the data structures of the array it was much easier to implement and more efficient. The sjf code is an object-oriented code it takes the inputs of two arrays and one integer value the arrays are for the two arrays of arrival time and one for burst time and the integer value is for the length of the arrays. Sjf code has a total of 6 arrays and 1 integer value, the arrays are an arrival time array, a burst time array, a copy of the burst time array, finish array, completion time array, and a turnaround time array. The constructor of this object takes in two arrays and an integer value, this constructor sets all the arrays to the length of the inputted integer value and deep copies the two arrays for arrival time and burst time. The constructor also checks for incorrect input and has exceptions put into place for the integer value being a negative, the integer not matching with the lengths of the array, the array not matching lengths, and the arrays having an input that is negative. The sjf object also has a method called getAvwait that gets the average of the waiting time, and it does this by filling out the other two arrays that are for completion time and turnaround time. That method saves the value for the completion time and turnaround time for each index corresponding with its arrival time and burst time and at the end of the method it gets the total waiting time by turnaround time - burst time for each index and finally, the method returns a double value for the average waiting time. the hardest part about this whole object for Sjf was figuring out how to get the waiting time and to get waiting time we found out that it was just the formula turnaround time - burst time and that made this much easier to complete. The time complexity for the sjf was $O(n^2)$.

Round-Robin Scheduling

Round-Robin Scheduling is an algorithm in which there is a time quantum which prompts the process and cycles to the next one. If the process was not finished, then it puts it at the end and execute it after one cycle through all the processes is done. If the process was completed before a given time quantum, then it ends the process and move to the next one instead of waiting for the whole quantum period.

Implementation technique:

For any given set of processes, we loop through every process for the first time based on their arrival times from the ready queue. For looping, each process is run for a given "quantum" period. If that process finishes before that period, then the waiting time for the next process is just the burst time of this process and plus the previous waiting times, if any. If this process does not finish in this given time frame, then the scheduling stops this process and put it at the end of the ready queue and start executing the next process in this ready queue. The waiting time in this case for the next process is equal to the time quantum. This runs for 'n' number of times, where 'n' is the number of total processes. After the first loop, scheduling iterates the same way through the remaining processes in the ready queue. This algorithm is followed until all the processes are done. Time complexity for this algorithm is not constant it depends on not just 'n', that is, number of processes, but also on the other factors. It depends on the time quantum. If the time quantum is too large, then it might end in the first iteration. If the quantum is too small, it might take couple iteration to complete it. For example, if the processes have burst time as {4, 5, 6} and our time quantum is 6, then this will end in one iteration that is, O(n). However, if our time quantum is 2, then it will iterate for 2n + 3n + 3n. So, in general our time complexity for solving this problem varies between O(n) and O(n^2).

For this implementation, we have used Array Queue as the main ADT that handles all the processes. Array Queue is an easy and memory efficient way of implementing this code. We only declare the size of the Queue as much as we need. This way, we do not store and manage a large amount of memory. If we used any other data structure like a Map, it would be memory efficient in this case. Removing and added data is easy in this type of data structure as it doesn't require any iteration or anything. However, it is not as fast as a Map, but it is relatively faster than any other data structure whilst being memory efficient. Moving from that, we have declared two classes within the public class Round Robin. One of which is Array Queue, and the other is Process. Process class is used to instantiate a Process which have certain attributes assigned to it. Different processes are queued in an Array Queue based on their arrival times and are removed (dequeued) and added to the last (enqueued) based on the above algorithm. RoundRobin class declares an ArrayQueue which is all the processes and manipulates it to evaluates the average wait time.

Combined Round Robin and Priority Scheduling

In the Combined Round Robin and Priority scheduling all the processes are handled in the priority associated with process and arrival time is assumed to be zero. Processes with high priority are scheduled first (0 being the highest) . Those processes with equal priority are scheduled in Round Robin scheduling. If two processes are in equal priority, then both of them are processed using Round Robin scheduling where a process is executed for one time quantum and if the process is not completed then it puts it at end of cycle. These steps keep repeating until a process is executed with in time quantum. If process is completed the time quantum, then it ends the process and moves to next high priority process in queue

Implementation technique: -

Firstly, class Process is used to make objects of processes with parameters such as number of processes "n", burst time and priority order . At this point a condition check whether all parameters are positive if not then exception are thrown accordingly. It has several accessor methods to retrieve the parameters and has two mutator methods to change the values of parameter.

Secondly, Array Queue ADTs have been implemented to achieve Round Robin scheduling. Lastly class called CombinedRRandPriority is implemented. In this class a constructor takes in number of processes, array of burst time and priority , and time quantum. All the parameters are used to create object of class Process and store them in array list.

Then they are sorted with high priority process at the front. Now to get waiting time of processes with same priority is done by using same method to calculate waiting time in round robin scheduling as in previous section. Waiting time for the rest of the processes is calculated accordingly to their priorities using the formula "Completion time – Burst time". At the end total waiting time is calculated by adding time from round robin scheduling and priority scheduling and then it is divided by number of processes to get an average waiting time.

For this implementation we have used array list data structure to store processes because it is easy to organize data through that and it does not require much space.

We have also used ArrayQueue to calculate the waiting time of round robin scheduling. Moreover it is memory efficient and is easier to implement in this case where we have to rotate the processes in the queue.

Time complexity for worst case of the method implemented to get average wait time is

O(n^2).

Junit Testing

To check the correctness of the code, we created a Junit tester file. Here we took various cases to check if it works fine. For the sjf file we chose two test cases with the correct input values, which gave the output as expected. For one of the other cases, we took length of string i.e., n negative and we used NegativeException for the same to solve the error. For the final two cases, we made use of ArrayIndexOutOfBoundsException for the input having two arrays of different length and LengthNotMatchException for the case in which the length of the array does not match the integer n.

For the RoundRobinScheduling file, we took four different test cases which tests the code by taking an array, an integer n, an integer quantum as input and calculates the average waiting time .

For the CombinedRRandPriority file, we took two arrays for burst time and arrival time respectively, int n and int quantum. In testing we take cases with different inputs and call the avgWaitingTime method. Finally, we use assertEquals method to compare the expected and actual output to check the validity if the code.