Written/Programmed By: CENSORED

Date Submitted: 6/9/19

Assignment: Lab 1

Class: COMP 3500

**Abstract:**

For our first lab assignment, we were given three files: processesgenerator.o, common.h, and processesmanagement.c. We were not supposed to modify the first two files, but the last file had four methods to fill in. Three of these methods were meant to be individual CPU Scheduling Policies, and the last method is a dispatcher for these methods so that the policies would work correctly. The program was finished in jGrasp and tested using a Tux Machine, which was accessed remotely using a Unix terminal and a sftp client (the latter being used to move files to the Tux machine). Our program (after implementing the methods) was designed to run multiple processes with these policies and return very specific times. With these times, we were able to determine which of the CPU Scheduling Policies is the most efficient and which was the most inefficient. The times that we received from running the program are as follows: the average turnaround times, the average response times, the CPU busy times, the throughput times, and the average waiting times. We will explain in detail later as to what these times are and how they relate to the CPU Scheduling policies.

**Introduction:**

The turnaround time is the time it takes for a process to complete and terminate, the response time is the time a process gets in the CPU the first time, the CPU busy time is the fraction of time a CPU is used by user processes, the throughput time is the number of processes completed divided by the observation time, and the waiting time is the total time spent by a process in the ready queue. The three CPU Scheduling Policies we implemented and observed times for were First Come First Serve (FCFS), Shortest Job First (SJF), and Round Robin (RR). We implemented the RR policy with several different quantum times (a quantum is a specified amount of milliseconds that must elapse before the CPU is preempted and the previously running process is added to the end of the ready queue), namely 1ms, 5ms, 10ms, 15ms, 20ms, 25ms, and 50ms. This was done to see how the difference in the quantum time would affect the measurable times for this specific policy. We evaluated our code through careful analysis of our data output in comparison with expected estimations of outputs for each policy. For example, we would not expect a low turnaround time from the FCFS policy, and checking the data yielded from the program against this expectation proved correct. The specifics on how each policy was implemented will be covered in detail in the next few paragraphs.

**Body:**

FCFS: This policy is the easiest to implement, since whatever process arrives first is the one to receive the CPU. Only one line of code was necessary to add to this method to successfully implement it. This line created a pointer variable to an instance of the ProcessControlBlock, and then assigned the first process in the ready queue to it before returning this process from the method. The times received from running this policy when compared to the other policies showed that this policy was the most inefficient, but we will go more into detail on that later. The bar graph below shows the approximate times for all five of the important measurable times for this policy.

By this graph, it is observed that there is an incredibly high turnaround time and average waiting time, while the other three times are relatively low.

SJF: This policy receives processes on a FCFS basis but will not relinquish control of the CPU until that process completes its CPU burst. Scheduling is only done in this policy when a process finishes and relinquishes the CPU voluntarily, and every process has the length of its next CPU burst associated with it so the process with the shortest time can be scheduled first. This was the most difficult policy to implement, as it took a solid 22 lines of code. We started with the same line as the one within the FCFS implementation but realized we could not simply return, and have it efficiently implemented this policy. We decided to check if the variable contained a process with an if statement before creating a new pointer variable to the ProcessControlBlock, called newProcess, and assigning it the next process in the ready queue. We then decided to create a new pointer variable to assign our original pointer variable to, called originalProcess, so we could keep it from being changed within our code. Next, we created a while loop that would loop as long as the newProcess variable contained a process. Another if-else statement was used to determine which of the two variables (selectedProcess and newProcess) had the lowest remaining CPU burst time. If newProcess had a lower remaining CPU burst time, we would put selectedProcess onto the ready queue and reassign it the process within newProcess; otherwise, we would just put the newProcess into the ready queue. We then used two nested if statements to determine if the originalProcess and newProcess process ID’s were the same and if the selectedProcess and newProcess process ID’s were not the same. If these conditions were met, we would put the newProcess on the ready queue; but if only the first condition was met, we would break out of the outer if statement and continue along within the method. The last line of the while loop reassigned the next process from the ready queue to the newProcess variable. Once the while loop was finished iterating through the available processes, we would return the process within the selectedProcess variable from the method. Below is a graph that shows the measurable times from running this policy:

RR: This policy was implemented the same as the FCFS policy but had different results because of the usage of a quantum. The CPU is assigned to the head of the list and behaves like the FCFS policy, but CPU bursts can only be done in limited periods of time (based on the quantum provided). The bar graphs below show every measurable time of each run with a different quantum.

By the graphs above, it is easily observable that the turnaround time decreases as the quantum increases. It is also observed that the other times increase as the quantum increases. The response time is therefore better with smaller quantum’s, but at the cost of the turnaround time.

The following table compares all the times from each policy run against each other:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Policy** | **PolicyNumber** | **TAT** | **RT** | **CBT** | **T** | **AWT** |
| **FCFS** | **1** | **12.911340** | **2.647811** | **0.904162** | **0.901168** | **11.021633** |
| **SJF** | **2** | **5.428102** | **1.936115** | **0.816647** | **0.987830** | **3.170147** |
| **RR (Q = 1 ms)** | **3** | **24.639088** | **0.002406** | **0.556995** | **0.750623** | **0.686590** |
| **RR (Q = 5 ms)** | **3** | **15.993210** | **0.027666** | **0.797449** | **0.901595** | **3.377649** |
| **RR (Q = 10 ms)** | **3** | **11.307913** | **0.056179** | **0.853839** | **0.940427** | **3.528679** |
| **RR (Q = 15 ms)** | **3** | **10.787967** | **0.094811** | **0.872056** | **0.944761** | **4.271994** |
| **RR (Q = 20 ms)** | **3** | **10.407878** | **0.132701** | **0.881642** | **0.961967** | **4.963684** |
| **RR (Q = 25 ms)** | **3** | **9.838220** | **0.160980** | **0.888946** | **0.966299** | **5.076786** |
| **RR (Q = 50 ms)** | **3** | **8.677123** | **0.342765** | **0.895585** | **0.970519** | **5.672173** |

Now we will share five graphs that compare the measurable times from the above table so that it is easier to tell which of the policies is the most efficient and which is the least efficient.

The turnaround time is the time it takes for a process to complete and terminate, given by the time a process arrives subtracted from the time the process completes. The first bar graph shows the turnaround times for each policy:

The response time is the time a process gets in the CPU the first time, given by the time a process arrives subtracted from the time the process gets the CPU the first time. The second bar graph shows the response times for each policy:

The CPU busy time is the fraction of time a CPU is used by user processes. The third bar graph shows the CPU busy times for each policy:

The throughput time is the number of processes completed divided by the observation time. The fourth bar graph shows the throughputs for each policy:

The waiting time is the total time spent by a process in the ready queue. The fifth bar graph show the average waiting times for each policy:

We will use these previous five bar graphs to explain how we reached our conclusion later.

Dispatcher: This method was designed to check if a process in the running queue needs computation. If the process did need computation, it would be put on the CPU with this method; otherwise, it would be moved from the running queue to the exit queue. We started out with the same line as the last three methods; by initializing the selectedProcess variable. We then have an if statement that will execute if the selectedProcess variable contains a process. If this if statement does not execute, then the selectedProcess variable is the only process and the method returns that process control block. Another if statement is nested inside the outer if statement that will execute if the selectedProcess variable has zero as a value for the time it has spent in the CPU. If that is the case, the starting CPU time is set to the current time, a new job is added with a new CPU burst time and response time, and the sum metrics for the response time is updated to the Job Arrival Time subtracted from the current time. Once this if statement has completed execution, an if-else statement is up next. The if statement will execute if the selectedProcess time in CPU is greater or equal to its total job duration. If that is the case, the job exit time for this process is updated to the current time, the current state of the process is now done, and the sum metrics for the turnaround time and waiting time are updated. The process is then put on the exit queue and the new job that was added will have its waiting time and throughput time added. If this isn’t the case, then the else statement containing three if statements will execute. The first if statement will execute if the policy is RR, and if that is the case then the CPU burst time will now equal the given quantum. The second if statement will execute if the remaining CPU burst time is less than the CPU burst time for the selectedProcess variable. If that is the case, the CPU burst time will now be equal to the remaining CPU burst time. The third if statement will execute if the time in the CPU subtracted from the total job duration is less than the CPU burst time. If that is the case, then the CPU burst time will be equal to the time in the CPU subtracted from the total job duration for the selectedProcess variable. Once these three if statements have been executed or passed by, the last few lines of the else statement will execute. These statements will add the selectedProcess variable and CPU burst time onto the CPU, add the CPU burst time to the time in CPU, add the remaining CPU burst time subtracted from the CPU burst time to the remaining CPU burst time, move the selectedProcess onto the running queue, and the CPU burst time will be added to the new job that was previously added. The initial if statement is now completed, and the dispatcher method is finished executing.

**Conclusion:**

With the data we collected from running these policies, we have determined that the most efficient CPU scheduling policy is Round Robin (RR) with a quantum of 1ms. This is due to it having the lowest times for everything except for the turnaround times, which was the highest out of all the policies. We also determined that the most inefficient CPU scheduling policy is First Come First Serve (FCFS). This is due to it having the highest times for three of the five measurable times, which were the average waiting times, the response times, and the CPU busy times. The highest turnaround time was from the Round Robin (RR) policy with a quantum of 1ms and the highest throughput was from the Shortest Job First (SJF) policy. Since we checked our output data against what would normally be yielded from these policies, we have determined that our outputs are roughly correct and the code we implemented for this assignment worked as expected.

**References:**

* Class Powerpoints on Canvas
* Class Textbook - Operating System Concepts Essentials