Student Name: Mohd Amaan Khan

Student ID: 11805050 (A41)

Email Address: amaankhan99845@gmail.com

GitHub Link:

Problem:

Ques16: Design a scheduler that can schedule the processes arriving system at periodical intervals. Every process is assigned with a fixed time slice t millisecond’s. If it is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and dispatcher dispatches the process to processor for execution. Compute the total time for which processes were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time quantum from the user.

**Explanation:**

1. Round Robin is the preemptive process scheduling algorithm.
2. Each process is provided a fix time to execute, it is called a quantum.
3. Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
4. Context switching is used to save states of preempted processes.

Example: Assume there are 5 processes with process ID and burst time given below PID Burst Time P1 6 P2 5 P3 2 P4 3 P5 7 – Time quantum: 2 – Assume that all process arrives at 0. Now, we will calculate average waiting time for these processes to complete. Solution – We can represent execution of above processes using GANTT chart as shown below –

Explanation: – First p1 process is picked from the ready queue and executes for 2 per unit time (time slice = 2). If arrival time is not available, it behaves like FCFS with time slice. – After P2 is executed for 2 per unit time, P3 is picked up from the ready queue. Since P3 burst time is 2 so it will finish the process execution at once. – Like P1 & P2 process execution, P4 and p5 will execute 2 time slices and then again it will start from P1 same as above. Waiting time = Turn Around Time – Burst Time P1 = 19 – 6 = 13 P2 = 20 – 5 = 15 P3 = 6 – 2 = 4 P4 = 15 – 3 = 12 P5 = 23 – 7 = 16 Average waiting time = (13+15+4+12+16) / 5 = 12

Algorithm:

1. Create an array arrival\_time[], burst\_time[] to keep track of arrival and bust time of processes.
2. Create another array temp[] to store waiting times of processes temporarily in between execution.
3. Initialize time: total = 0, counter = 0, wait time = 0, turnaround time = 0.
4. Ask user for no of process and store it in limit.
5. Repetitively ask user to give input for - arrival time, bust time up to limit.
6. Ask user to enter time quantum and store it into time\_quantum.
7. Keep traversing the all processes while all processes are not done. Do following for i'th process if it is not done yet. a. If temp[i] <= time\_quantum && temp[i] > 0 (i) total = total + temp[i]; (ii) temp[i] = 0; (iii) counter = 1; b. Else if temp[i] > 0 (i) temp[i] = temp[i] - time\_quantum; (ii) total = total + time\_quantum; c. If temp[i] == 0 && counter == 1 (i) x--; (ii) print burst\_time[i], total, arrival\_time[i], total - arrival\_time[i] - burst\_time[i]); (iii) wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i]; (iv) turnaround\_time = turnaround\_time + total - arrival\_time[i]; (v) counter = 0; d. If i == limit – 1 (i) i = 0; e. Else if arrival\_time[i + 1] <= total (i) i++; c. Else (i) i = 0;
8. Print total waiting time - wait\_time, average waiting time - wait\_time/limit, average turnaround time - average\_turnaround\_time/limit.

Complexity:

• Complexity of initialization of variables is O(1). total = 0 counter = 0 wait\_time = 0 turnaround\_time = 0 • Complexity for initializing arrays with for loop is O(limit). for(i = 0; i < limit; i++) { … } • Complexity of calculating waiting and turnaround time is O(limit). for(total = 0, i = 0; x != 0;) { …. } • Complexity of implemented algorithm is O(limit).

# Description:

Design a scheduler that can schedule the processes arriving system at periodical intervals. Every process is assigned with a fixed time slice t milliseconds. If it is not able to complete its execution within the assigned time quantum, then automated timer generates an interrupt. The scheduler will select the next process in the queue and Dispatcher dispatches the process to processor for execution. Compute the total time for which processes were in the queue waiting for the processor. Take the input for CPU burst, arrival time and time quantum from the user. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a cyclic way. A certain time slice is defined in the system which is called time quantum. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will terminate else the process will go back to the ready queue and waits for the next turn to complete the execution.

# Code:

#include<stdio.h>

int main()

{

int i,k, limit, total = 0, x, counter = 0, time\_quantum;

int wait\_time = 0, turnaround\_time = 0, arrival\_time[10], burst\_time[10], temp[10];

float average\_wait\_time, average\_turnaround\_time;

for(k=0;k<5;k++) //for five time iteration

{

printf("\nEnter Total Number of Processes:\t");

scanf("%d", &limit);

x = limit;

for(i = 0; i < limit; i++)

{

printf("\nEnter Details of Process[%d]\n", i + 1);

printf("Arrival Time:\t");

scanf("%d", &arrival\_time[i]);

printf("Burst Time:\t");

scanf("%d", &burst\_time[i]);

temp[i] = burst\_time[i];

}

printf("\nEnter Time Quantum:\t");

scanf("%d", &time\_quantum);

printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");

for(total = 0, i = 0; x != 0;)

{

if(temp[i] <= time\_quantum && temp[i] > 0)

{

total = total + temp[i];

temp[i] = 0;

counter = 1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - time\_quantum;

total = total + time\_quantum;

}

if(temp[i] == 0 && counter == 1)

{

x--;

printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst\_time[i], total - arrival\_time[i], total - arrival\_time[i] - burst\_time[i]);

wait\_time = wait\_time + total - arrival\_time[i] - burst\_time[i];

turnaround\_time = turnaround\_time + total - arrival\_time[i];

counter = 0;

}

}

average\_wait\_time = wait\_time \* 1.0 / limit;

average\_turnaround\_time = turnaround\_time \* 1.0 / limit;

printf("\n\nTotal Waiting Time:\t%d", wait\_time);

printf("\n\nAverage Waiting Time:\t%f", average\_wait\_time);

printf("\nAvg Turnaround Time:\t%f\n", average\_turnaround\_time);

}

return 0;

}

Constraints:

1. If we have large number of processes and a process with very less burst time. According to round robin it will have to wait for very long time if burst time of other processes is large and its turn comes late.
2. If we have some processes running according to round robin and we have a process whose arrival time is after the completion of those processes, then that process will not execute. If we want that all the processes should execute successfully then the arrival time of other process must be less than or equal to the completion time of the running processes.

Test Cases:

1. If we select same arrival time for 2 processes. Expected Result: The process having less burst time should have less turnaround time. Actual Result:

Test case passed successfully.

1. If p1 process arrives at 2 and burst time=4 but p2 and p3 process arrives at 6. Expected Result: Here only process p1 will execute because p2 and p3 are unable to arrive within the completion of process p1. Actual Result:

Test case passed successfully.

1. Now if process p1 arrives at 0 and two other processes p2 and p3 arrives within the completion of process p1. Expected Result: All 3 process will execute. Actual Result:
2. Test case passed successfully.

Conclusion: Hence from the test cases we conclude that if we want that all the processes should execute successfully then the arrival time of other process must be less than or equal to the completion time of the running processes.

Output: