...for a complete human being
Texas
International College

Lab No: 6 **Date: 2082/**

Title: Write a program to calculate the average turnaround time and waiting time for user input process parameters using RR process scheduling algorithm.

Round Robin (RR) is a preemptive CPU scheduling algorithm that assigns each process a fixed time quantum. The CPU cycles through processes in the ready queue, giving each process a fair share of CPU time. If a process doesn't finish within its time quantum, it goes back to the queue to wait for the next turn.

Algorithm:

Step 1: Start with the list of processes and note their arrival time and burst time.

Step 2: Choose a fixed time quantum.

Step 3: Arrange processes in the ready queue in the order of arrival.

Step 4: Allocate CPU to the first process for one time quantum.

Step 5:

- If the process completes, remove it from the queue.
- If not, reduce its remaining burst time and put it back at the end of the queue.

Step 6: Repeat Step 4 and Step 5 until all processes are completed.

Step 7: Calculate waiting time and turnaround time for each process, then compute averages.

Step 8: Stop.

Language: C++

IDE: VS Code

Code:

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;


struct Job {

    int jobId;

    int burst;

    int remaining;

    int arrival;

    int completion;

    int tat;   // Turn Around Time

    int wt;    // Waiting Time

};


int main() {

    int totalJobs, timeSlice;

    cout << "Enter number of processes: ";

    cin >> totalJobs;

    vector<Job> taskList(totalJobs);

    // Assume all processes arrive at time 0

    for (int j = 0; j < totalJobs; j++) {

        taskList[j].jobId = j + 1;

        taskList[j].arrival = 0;
```

```
        cout << "Enter burst time for process " << j + 1 << ": ";
        cin >> taskList[j].burst;
        taskList[j].remaining = taskList[j].burst;
    }
    cout << "Enter Time Quantum: ";
    cin >> timeSlice;
    // Round Robin Simulation
    queue<int> rrQueue;
    int clk = 0, finished = 0;
    vector<bool> inLine(totalJobs, false);
    // Initially push all processes since AT = 0
    for (int j = 0; j < totalJobs; j++) {
        rrQueue.push(j);
        inLine[j] = true;
    }
    cout << "\nGantt Chart:\n|";
    while (!rrQueue.empty()) {
        int j = rrQueue.front();
        rrQueue.pop();
        inLine[j] = false;
        int runTime = min(timeSlice, taskList[j].remaining);
        clk += runTime;
        taskList[j].remaining -= runTime;
        cout << "  P" << taskList[j].jobId << "\t|";
        if (taskList[j].remaining == 0) {
```

```
            taskList[j].completion = clk;
            finished++;
        } else {
            rrQueue.push(j);
            inLine[j] = true;
        }
    }
    // Time scale printing
    cout << "\n0";
    clk = 0;
    queue<int> tempQ;
    for (int j = 0; j < totalJobs; j++) tempQ.push(j);
    vector<int> leftBurst(totalJobs);
    for (int j = 0; j < totalJobs; j++) leftBurst[j] = taskList[j].burst;

    while (!tempQ.empty()) {
        int j = tempQ.front();
        tempQ.pop();
        int runTime = min(timeSlice, leftBurst[j]);
        clk += runTime;
        leftBurst[j] -= runTime;
        cout << "     \t" << clk;
        if (leftBurst[j] > 0) tempQ.push(j);
    }
    cout << "\n";
```

```cpp
    // Calculate TAT & WT
    cout << "\nProcess\tBT\tAT\tCT\tTAT (CT-AT)\tWT (TAT-BT)\n";
    double sumTAT = 0, sumWT = 0;
    for (int j = 0; j < totalJobs; j++) {
        taskList[j].tat = taskList[j].completion - taskList[j].arrival;
        taskList[j].wt = taskList[j].tat - taskList[j].burst;
            cout<< "  P" << taskList[j].jobId << "\t";
            cout<< taskList[j].burst << "\t";
            cout<< taskList[j].arrival << "\t;";
            cout << taskList[j].completion << "\t";
             cout<< taskList[j].completion << "-" << taskList[j].arrival;
             cout<< "=" << taskList[j].tat << "\t\t";
             cout<< taskList[j].tat << "-" << taskList[j].burst;
             cout<< "=" << taskList[j].wt << "\n";
        sumTAT += taskList[j].tat;
        sumWT += taskList[j].wt;
    }
    cout << "\nAverage Turnaround Time = " << sumTAT / totalJobs;
    cout << "\nAverage Waiting Time = " << sumWT / totalJobs << "\n";
    return 0;
}
```

Output:

```
Enter number of processes: 4
Enter burst time for process 1: 7
Enter burst time for process 2: 4
Enter burst time for process 3: 8
Enter burst time for process 4: 3
Enter Time Quantum: 3

Gantt Chart:
|  P1    |  P2    |  P3    |  P4    |  P1    |  P2    |  P3    |  P1    |  P3    |
0        3        6        9        12       15       16       19       20       22

Process BT      AT      CT      TAT (CT-AT)     WT (TAT-BT)
  P1    7       0       20      20-0=20         20-7=13
  P2    4       0       16      16-0=16         16-4=12
  P3    8       0       22      22-0=22         22-8=14
  P4    3       0       12      12-0=12         12-3=9

Average Turnaround Time = 17.5
Average Waiting Time = 12

c:\Users\Roshan\Desktop\Roshan_OS>
```

Conclusion:

Round Robin is fair and starvation-free because each process gets equal CPU time. It is widely used in time-sharing systems. However, its performance depends heavily on the time quantum:

- If too small → high context switching overhead.
- If too large → behaves like FCFS.