Experiment 04- To write a program for the implementation of the RR scheduling algorithm

Learning Objective: Students should be able to understand the RR algorithm by using different coding languages C/C++/Java/Python.

Tools: Online compiler

Theory:

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of the First come First Serve CPU Scheduling algorithm.

Characteristics of Round Robin CPU Scheduling Algorithm with Same Arrival Time:

1. **Equal Time Allocation:** Each process gets an equal and fixed time slice (time quantum) to execute, ensuring fairness.
2. **Cyclic Execution:** Processes are scheduled in a circular order, and the CPU moves to the next process in the queue after completing the time quantum.
3. **No Process Starvation:** All processes are guaranteed CPU time at regular intervals, preventing any process from being neglected.
4. **Same Start Time:** Since all processes arrive at the same time, there is no need to consider arrival time while scheduling, simplifying the process.
5. **Context Switching:** Frequent context switching occurs as the CPU moves between processes after each time quantum, which can slightly impact performance.

Code:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void roundRobinWithArrivalTime(vector<int>& processes, vector<int>& burstTimes, vector<int>& arrivalTimes, int
quantum) {
    int n = processes.size();
    vector<int> remainingBurstTimes = burstTimes;  // Copy to track remaining burst times
    vector<int> waitingTimes(n, 0), turnAroundTimes(n, 0);
    vector<int> completionTimes(n, 0);

    queue<int> q;  // Ready queue
    int currentTime = 0;
    int completedProcesses = 0;
    int processIndex = 0;  // To track which process to add based on arrival time

    // While there are still processes to complete
    while (completedProcesses < n) {
        // Add processes that have arrived by the current time
        while (processIndex < n && arrivalTimes[processIndex] <= currentTime) {
            q.push(processIndex);
            processIndex++;
        }

        if (!q.empty()) {
            int currentProcess = q.front();
            q.pop();

            // Process gets executed for a full quantum
            if (remainingBurstTimes[currentProcess] > quantum) {
                currentTime += quantum;
                remainingBurstTimes[currentProcess] -= quantum;

                // Add the process back to the queue if it's not finished
                while (processIndex < n && arrivalTimes[processIndex] <= currentTime) {
                    q.push(processIndex);
                    processIndex++;
                }
                q.push(currentProcess);  // Add the process back to the queue
            } else {
                // Process finishes execution
                currentTime += remainingBurstTimes[currentProcess];
                completionTimes[currentProcess] = currentTime;
                waitingTimes[currentProcess] = completionTimes[currentProcess] - burstTimes[currentProcess] -
arrivalTimes[currentProcess];
                turnAroundTimes[currentProcess] = waitingTimes[currentProcess] + burstTimes[currentProcess];
                remainingBurstTimes[currentProcess] = 0;
                completedProcesses++;
            }
        } else {
            // If no process is ready to execute, advance time to the next arrival
            currentTime = arrivalTimes[processIndex];
        }
    }

    // Output the results
    cout << "Process\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n";
    int totalWaitingTime = 0, totalTurnaroundTime = 0;
    for (int i = 0; i < n; ++i) {
        cout << processes[i] << "\t\t" << arrivalTimes[i] << "\t\t"
             << burstTimes[i] << "\t\t" << waitingTimes[i] << "\t\t"
             << turnAroundTimes[i] << endl;
        totalWaitingTime += waitingTimes[i];
        totalTurnaroundTime += turnAroundTimes[i];
    }

    // Calculate and display average waiting time and turnaround time
    double avgWaitingTime = (double)totalWaitingTime / n;
    double avgTurnaroundTime = (double)totalTurnaroundTime / n;

    cout << "\nAverage Waiting Time: " << avgWaitingTime << endl;
    cout << "Average Turnaround Time: " << avgTurnaroundTime << endl;
}

int main() {
    // Example usage
    int n;
    cout << "Enter number of processes: ";
    cin >> n;
    vector<int> processes(n), burstTimes(n), arrivalTimes(n);
```

Round Robin scheduling is an effective and fair CPU scheduling algorithm, ensuring each process gets an equal opportunity to execute. The inclusion of arrival times prevents starvation, making it more realistic. However, the performance heavily depends on the choice of time quantum. A well-chosen quantum balances waiting and turnaround times, while a poor choice can lead to inefficiency. Overall, Round Robin is a good choice for systems where fairness is prioritized, but optimizations such as dynamic quantum adjustments or priority-based approaches could further improve its performance.

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance/ Learning Attitude [20%]**	
Marks Obtained				