Experiment:8- Construct a C program to simulate Round Robin scheduling algorithm with C.

Aim:

The aim of this program is to simulate the Round Robin (RR) CPU scheduling algorithm. In this algorithm, each process is assigned a fixed time slice (quantum) in which it executes. If a process does not complete within this time slice, it is preempted and moved to the back of the ready queue. This process continues until all processes have been completed.

Procedure:

1. Input:
    - Number of processes.
    - Burst time for each process.
    - Time quantum (fixed time slice).
2. Execution:
    - Execute each process in a cyclic order.
    - After each time slice, if a process is not finished, move it to the back of the queue.
    - If a process finishes within the time quantum, remove it from the ready queue.
3. Waiting Time Calculation:
    - Calculate the waiting time for each process. The waiting time is the total time a process spends in the ready queue.
4. Turnaround Time Calculation:
    - Calculate the turnaround time for each process. Turnaround time is the total time from the arrival of the process to its completion.
5. Output:
    - Output the process ID, burst time, waiting time, and turnaround time for each process.
    - Calculate and display the average waiting time and average turnaround time.

Round Robin Scheduling Algorithm:

- Round Robin is a preemptive scheduling algorithm where each process is assigned a fixed time slice or time quantum.
- If a process does not finish within its time quantum, it is put back in the ready queue, and the next process is given the CPU.
- This cycle repeats until all processes are completed.

C Program Implementation:

```
#include <stdio.h>
struct Process {
   int id;
   int burst_time;
   int remaining_time;
   int waiting_time;
   int turnaround_time;
};

void calculateWaitingAndTurnaroundTime(struct Process processes[], int n, int time_quantum) {
   int time = 0;
   int completed = 0;

   // Initially setting remaining times equal to burst times
   for (int i = 0; i < n; i++) {
      processes[i].remaining_time = processes[i].burst_time;
   }

   while (completed < n) {
      for (int i = 0; i < n; i++) {
         if (processes[i].remaining_time > 0) {
            if (processes[i].remaining_time > time_quantum) {
               // Process will execute for the time quantum
               processes[i].remaining_time -= time_quantum;
               time += time_quantum;
            } else {
               // Process finishes execution
               time += processes[i].remaining_time;
               processes[i].waiting_time = time - processes[i].burst_time;
               processes[i].turnaround_time = time;
               processes[i].remaining_time = 0;
```

```
            completed++;
        }
      }
    }
  }
}

int main() {
  int n, time_quantum;

  // Input the number of processes and time quantum
  printf("Enter the number of processes: ");
  scanf("%d", &n);
  printf("Enter time quantum: ");
  scanf("%d", &time_quantum);

  struct Process processes[n];

  // Input burst time for each process
  for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;  // Assign process ID
    printf("Enter burst time for process %d: ", i + 1);
    scanf("%d", &processes[i].burst_time);
  }

  // Calculate waiting time and turnaround time
  calculateWaitingAndTurnaroundTime(processes, n, time_quantum);

  // Output the results
  printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
  int total_waiting_time = 0, total_turnaround_time = 0;
```

```
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time,
               processes[i].waiting_time, processes[i].turnaround_time);
        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;
    }

    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);

    return 0;
}
```

Output: