Experiment No – 4c **Date -**

(b) ROUND ROBIN SCHEDULING

Aim - To implement ROUND ROBIN scheduling algorithm

THEORY –

The Round Robin (RR) Scheduling Algorithm is a classic, preemptive CPU scheduling method designed to ensure **fairness** by giving every process an equal slice of CPU time. It is particularly well-suited for **time-sharing systems** where responsiveness is crucial.

Round Robin scheduling operates on the principle that all processes are handled in a **cyclic** and sequential manner, much like a round-robin tournament where every participant gets a turn.

Key Mechanism: The Time Quantum (Time Slice)
The core concept is the **time quantum** (or time slice), a small, fixed unit of time (typically to milliseconds) that the CPU scheduler assigns to each process.
How It Works (Preemptive FIFO)

1. **Queue Structure:** All ready processes are kept in a circular queue (a FIFO or First-In, First-Out queue).
2. **Execution:** The scheduler picks the first process from the ready queue and allocates the CPU to it for exactly **one time quantum**.
3. **Preemption Check:**
    - If the process finishes its execution (its burst time is used up) within the quantum, it is terminated, and the CPU is immediately given to the next process in the queue.
    - If the process **does not** finish within the quantum, the timer goes off, and the currently running process is **preempted** (interrupted).
4. **Rotation:** The preempted process is then moved to the **tail** (end) of the ready queue, where it waits for its next turn.
5. **Cycle:** The scheduler moves on to allocate the CPU to the next process at the head of the queue, repeating the cycle until all processes are complete.

IMPORTANT TERMS

1. Arrival Time (AT)
The time at which a process arrives in the ready queue.
Example: If Process P1 has AT = 2, it means it comes to the system at time unit 2.

2. Burst Time (BT)
The total execution time a process needs to complete on the CPU.
Example: If BT = 5, the process requires 5 units of CPU time.

3. Completion Time (CT)
The time at which a process finishes execution.
Example: If a process starts at time 4 and runs for 5 units, its CT = 9.

4. Turnaround Time (TAT)
The total time a process spends in the system (from arrival to completion).
Formula:
TAT = CT - AT

5. Waiting Time (WT)
The time a process spends waiting in the ready queue before getting CPU.
Formula:
WT = TAT - BT

6. Average Turnaround Time (Avg TAT)
Average of turnaround times of all processes.

7. Average Waiting Time (Avg WT)
Average of waiting times of all processes.

Example -

Process Name	Arrival Time	Burst Time	Time Quantum
A	2	2	1
B	0	8	1
C	3	16	1
D	7	1	1

Gantt Chart –

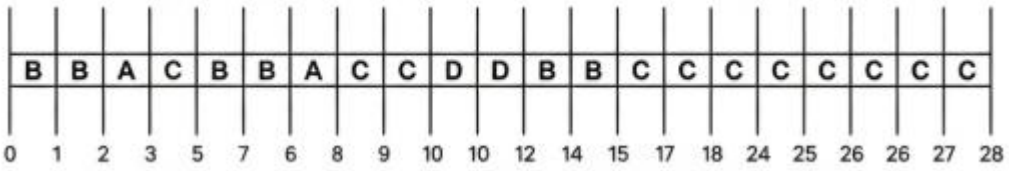

Table -

Process Name	Arrival Time	Burst Time	Time Quantum	Completion Time	Turn Around Time (TAT)	Wait Time
A	2	2	1	7	5	3
B	0	8	1	18	18	8
C	3	16	1	28	25	9
D	7	1	1	9	2	1

AVERAGE TURN AROUND TIME IS 12.5 AND AVERAGE WAIT TIME IS 5.75

PROGRAM

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

struct Process {
  string name;
  int arrival, burst,
remaining, start,
completion, turnaround,
waiting;
  bool started;
};

int main() {
  int n;
  cout << "Enter number
of processes: ";
  cin >> n;

  vector<Process>
p(n);
  for (int i = 0; i < n; i++) {
    cout << "\nEnter
Process Name, Arrival
Time, Burst Time for P"
<< i + 1 << ": ";
    cin >> p[i].name >>
p[i].arrival >> p[i].burst;
    p[i].remaining =
p[i].burst;
    p[i].started = false;
  }

  int time_quantum;
  cout << "\nEnter Time
Quantum: ";
  cin >> time_quantum;
  sort(p.begin(),
p.end(), [](const
Process& a, const
Process& b) {
    return a.arrival <
b.arrival;
  });

  vector<bool> done(n,
false);
  int time = 0,
completed = 0;
  float avgTAT = 0,
avgWT = 0;
  int totalTAT = 0;
  vector<string>
runningOrder;
  vector<int>
runningTime;
  vector<string>
readyOrder;
  vector<int>
readyTime;
  queue<int>
ready_queue;

  time = p[0].arrival;
  for (int i = 0; i < n; i++) {
    if (p[i].arrival ==
time) {
      ready_queue.push
(i);
    }
  }

  while (completed < n)
{
    if
(ready_queue.empty()) {
      int nextArrival =
INT_MAX;
      for (int i = 0; i < n;
i++) {
        if (!done[i] &&
p[i].arrival < nextArrival)
{
          nextArrival =
p[i].arrival;
        }
      }
      time = nextArrival;
      for (int i = 0; i < n;
i++) {
        if (!done[i] &&
p[i].arrival == time) {
          ready_queue.p
ush(i);
        }
      }
    } else {
      int idx =
ready_queue.front();
      ready_queue.pop(
);

      if (!p[idx].started) {
        p[idx].start =
time;
        p[idx].started =
true;
      }

      runningOrder.pus
h_back(p[idx].name);
      runningTime.push
_back(time);

      int exec_time =
min(time_quantum,
p[idx].remaining);
      time += exec_time;
      p[idx].remaining -
= exec_time;
```

```
        for (int i = 0; i < n; i++) {
            if (!done[i] && p[i].arrival > time - exec_time && p[i].arrival <= time) {
                ready_queue.push(i);
                readyOrder.push_back(p[i].name);
                readyTime.push_back(time - exec_time + (p[i].arrival - (time - exec_time)));
            }
        }

        if (p[idx].remaining == 0) {
            p[idx].completion = time;
            p[idx].turnaround = p[idx].completion - p[idx].arrival;
            p[idx].waiting = p[idx].turnaround - p[idx].burst;
            avgTAT += p[idx].turnaround;
            avgWT += p[idx].waiting;
            totalTAT += p[idx].turnaround;
            done[idx] = true;
            completed++;
        } else {
            ready_queue.push(idx);
            readyOrder.push_back(p[idx].name);
            readyTime.push_back(time);
        }
    }

    runningTime.push_back(time);
    readyTime.push_back(time);

    cout << "\n----------------------------------------------------------\n";
    cout << "Process\tAT\tBT\tCT\tTAT\tWT\n";
    cout << "----------------------------------------------------------\n";
    for (int i = 0; i < n; i++) {
        cout << p[i].name << "\t" << p[i].arrival << "\t" << p[i].burst << "\t"
             << p[i].completion << "\t" << p[i].turnaround << "\t" << p[i].waiting << "\n";
    }
    cout << "----------------------------------------------------------\n";
    cout << "Average Turnaround Time = " << avgTAT / n << endl;
    cout << "Average Waiting Time = " << avgWT / n << endl;

    auto displayChart = [&](string title, vector<string>& order, vector<int>& times) {
        cout << "\n" << title << ":\n";
        cout << "-------------------------------------------------------------\n";
        for (size_t i = 0; i < order.size(); i++) {
            cout << "| " << setw(3) << order[i] << " ";
        }
        cout << "|\n";
        cout << "-------------------------------------------------------------\n";
        cout << setw(2) << times[0];
        for (size_t i = 1; i < times.size(); i++) {
            cout << setw(6) << times[i];
        }
        cout << "\n";
    };

    displayChart("Ready Queue", readyOrder, readyTime);
    displayChart("Running Queue", runningOrder, runningTime);

    return 0;
}
```

OUTPUT –

```
PS C:\Users\Atharv\OneDrive\Documents\College\SEM5\OS\LAB\EXP4> cd "c:\Users\Atharv\OneDrive\Documents\College\SEM5\OS\LAB\EXP4\" ; if ($?) { g++ tempCodeRunnerFile.cpp -o tempCodeRunn
Enter number of processes: 4

Enter Process Name, Arrival Time, Burst Time for P1: A
1
12

Enter Process Name, Arrival Time, Burst Time for P2: B
3
3

8

Enter Process Name, Arrival Time, Burst Time for P4: D
1
6

Enter Time Quantum: 1

-------------------------------------------------------------
Process AT      BT      CT      TAT     WT
-------------------------------------------------------------
A       1       12      30      29      17
D       1       6       21      20      14
C       2       8       26      24      16
B       3       3       14      11      8
-------------------------------------------------------------
Average Turnaround Time = 21
Average Waiting Time = 13.75

Ready Queue:
-------------------------------------------------------------
|  C|  A|  B|  D|  C|  A|  B|  D|  C|  A|  B|  D|  C|  A|  D|  C|  A|  D|  C|  A|  C|  A|  C|  A|  A|  A|  A|
-------------------------------------------------------------
 2    2    3    3    4    5    6    7    8    9   10   11   12   13   15   16   17   18   19   20   22   23   24   25   27   28   29   30

Running Queue:
-------------------------------------------------------------
|  A|  D|  C|  A|  B|  D|  C|  A|  B|  D|  C|  A|  B|  D|  C|  A|  D|  C|  A|  D|  C|  A|  C|  A|  C|  A|  A|  A|  A|
-------------------------------------------------------------
 1    2    3    4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19   20   21   22   23   24   25   26   27   28   29   30
```

CONCLUSION - Round Robin scheduling prioritizes fairness and responsiveness by giving each process a fixed time quantum in a cyclic manner, effectively preventing starvation. Its performance is critically dependent on the chosen **time quantum**, balancing context switching overhead with interactive system demands. It remains a cornerstone for **time-sharing operating systems**, ensuring all users and tasks receive timely CPU access.