Batch: A1 **Roll No.: 16010123012**

Experiment No. 05

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Implementation of Basic Process management algorithms - Preemptive (SRTN, RR)

AIM: To implement basic Process management algorithms (Round Robin, SRTN

Expected Outcome of Experiment:
CO 2. To understand the concept of process, thread and resource management.

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third**
3. **Edition.**
4. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
5. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

Most systems handle numerous processes with short CPU bursts interspersed with I/O requests and a few processes with long CPU bursts. To ensure good time-sharing performance, a running process may be preempted to allow another to run. The ready list, or run queue, maintains all processes ready to run and not blocked by I/O or other system requests. Entries in this list point to the process control block, which stores all process information and state.

When an I/O request completes, the process moves from the waiting state to the ready state and is placed on the run queue. The process scheduler, a key component of the operating system, decides whether the current process should continue running or if another should take over. This decision is triggered by four events:

1. The current process issues an I/O request or system request, moving it from running to waiting.

2. The current process terminates.
3. A timer interrupt indicates the process has run for its allotted time, moving it from running to ready.
4. An I/O operation completes, moving the process from waiting to ready, potentially preempting the current process.

The scheduling algorithm, or policy, determines the sequence and duration of process execution, a complex task given the limited information about ready processes.

Description of the application to be implemented:

Round Robin Algorithm:

Step 1: Input Process Details
1. Start
2. Take input for the number of processes n
3. For each process, take input:
Arrival Time (AT)
Burst Time (BT)
4. Initialize:
PID (Process ID)
Remaining Burst Time (set to Burst Time initially)
5. Take input for Time Quantum (TQ)

Step 2: Find Completion Time Using RR Scheduling:
1. Initialize current_time = 0 and completed = 0
2. Create a Ready Queue and push the first process (index 0)
3. Create an array is_completed[] of size n initialized to `false`
4. While (completed < n), repeat the following:
- Pick the first process from the Ready Queue
- Remove it from the queue
- If the process has remaining burst time:
- Execute it for min(Time Quantum, Remaining Burst Time)
- Reduce its remaining_burst_time
- Increase current_time accordingly
- If remaining_burst_time = 0 and the process is not completed:
- Set Completion Time (CT) = current_time
- Calculate Turnaround Time (TAT) = Completion Time - Arrival Time
- Calculate Waiting Time (WT) = Turnaround Time - Burst Time
- Mark the process as completed
- Increment completed count
- Check for newly arrived processes that are not completed and not in the queue. Add them to the Ready Queue
- If the current process still has remaining burst time, reinsert it into the Ready Queue

Step 3: Calculate and Display Results
1. Compute Average Turnaround Time (Avg TAT) and Average Waiting Time (Avg WT)

2. Print Process ID, Arrival Time, Burst Time, Completion Time, Turnaround Time, and Waiting Time
3. End

Shortest Remaining Time First Algorithm:

Step 1: Input Process Details
1. Start
2. Take input for the number of processes n
3. For each process, take input:
Arrival Time (AT)
Burst Time (BT)
4. Initialize:
PID (Process ID)
Remaining Burst Time (set to Burst Time initially)

Step 2: Find Completion Time Using SRTF Scheduling
1. Initialize current_time = 0 and completed = 0
2. Create an array is_completed[] of size n initialized to `false`
3. While (completed < n), repeat the following:
- Find the process with the shortest remaining burst time that has arrived (AT ≤ current_time) and is not completed.
- If such a process is found:
  - Reduce its remaining_burst_time by 1
  - Increment current_time by 1
  - If remaining_burst_time becomes 0:
   - Update completion_time = current_time
   - Calculate Turnaround Time (TAT) = Completion Time - Arrival Time
   - Calculate Waiting Time (WT) = Turnaround Time - Burst Time
   - Mark the process as completed
   - Increment completed count
- If no process is found, increment current_time by 1

Step 3: Calculate and Display Results
1. Compute Average Turnaround Time (Avg TAT) and Average Waiting Time (Avg WT)
2. Print Process ID, Arrival Time, Burst Time, Completion Time, Turnaround Time, and Waiting Time
3. End

Implementation details:

RR

```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;
```

```
struct Process
{
  int pid;
  float arrival_time;
  float burst_time;
  float remaining_burst_time;
  float completion_time;
  float turnaround_time;
  float waiting_time;
};

void findCompletionTime(Process proc[], int n, float time_quantum)
{
  float current_time = 0;
  int completed = 0;
  vector<int> ready_queue;
  vector<bool> is_completed(n, false);

  ready_queue.push_back(0);

  while (completed < n)
  {
    int index = ready_queue.front();
    ready_queue.erase(ready_queue.begin());

    if (proc[index].remaining_burst_time > 0)
    {
      float time_to_run = min(time_quantum,
proc[index].remaining_burst_time);

      proc[index].remaining_burst_time -= time_to_run;
      current_time += time_to_run;

      if (proc[index].remaining_burst_time == 0 && !is_completed[index])
      {
        proc[index].completion_time = current_time;
        proc[index].turnaround_time = proc[index].completion_time -
proc[index].arrival_time;
        proc[index].waiting_time = proc[index].turnaround_time -
proc[index].burst_time;
        is_completed[index] = true;
        completed++;
      }
    }
```

```
    for (int i = 0; i < n; i++)
    {
      if (proc[i].arrival_time <= current_time && !is_completed[i] &&
find(ready_queue.begin(), ready_queue.end(), i) == ready_queue.end())
      {
        ready_queue.push_back(i);
      }
    }

    if (proc[index].remaining_burst_time > 0)
    {
      ready_queue.push_back(index);
    }
  }
}

void findavgTime(Process proc[], int n)
{
  float total_wt = 0, total_tat = 0;
  for (int i = 0; i < n; i++)
  {
    total_wt += proc[i].waiting_time;
    total_tat += proc[i].turnaround_time;
  }
  cout << "Average turnaround time = " << total_tat / n << endl;
  cout << "Average waiting time = " << total_wt / n << endl;
}

int main()
{
  int n;
  float time_quantum;

  cout << "Enter the number of processes: ";
  cin >> n;
  Process proc[n];

  for (int i = 0; i < n; i++)
  {
    cout << "Enter the arrival time of process " << i + 1 << ": ";
    cin >> proc[i].arrival_time;
    cout << "Enter the burst time of process " << i + 1 << ": ";
    cin >> proc[i].burst_time;
    proc[i].pid = i + 1;
    proc[i].remaining_burst_time = proc[i].burst_time;
  }
```

```
  cout << "Enter the time quantum: ";
  cin >> time_quantum;

  findCompletionTime(proc, n, time_quantum);
  cout << endl;
  cout << "PN\tAT\tBT\tCT\tTAT\tWT" << endl;
  for (int i = 0; i < n; i++)
  {
    cout << proc[i].pid << "\t" << proc[i].arrival_time << "\t" <<
proc[i].burst_time << "\t" << proc[i].completion_time << "\t" <<
proc[i].turnaround_time << "\t" << proc[i].waiting_time << endl;
  }
  findavgTime(proc, n);
}
```

```
Enter the number of processes: 4
Enter the arrival time of process 1: 0
Enter the burst time of process 1: 6
Enter the arrival time of process 2: 1
Enter the burst time of process 2: 2
Enter the arrival time of process 3: 2
Enter the burst time of process 3: 8
Enter the arrival time of process 4: 3
Enter the burst time of process 4: 3
Enter the time quantum: 2

PN      AT      BT      CT      TAT     WT
1       0       6       10      10      4
2       1       2       6       5       3
3       2       8       19      17      9
4       3       3       17      14      11
Average turnaround time = 11.5
Average waiting time = 6.75
```

SRTF

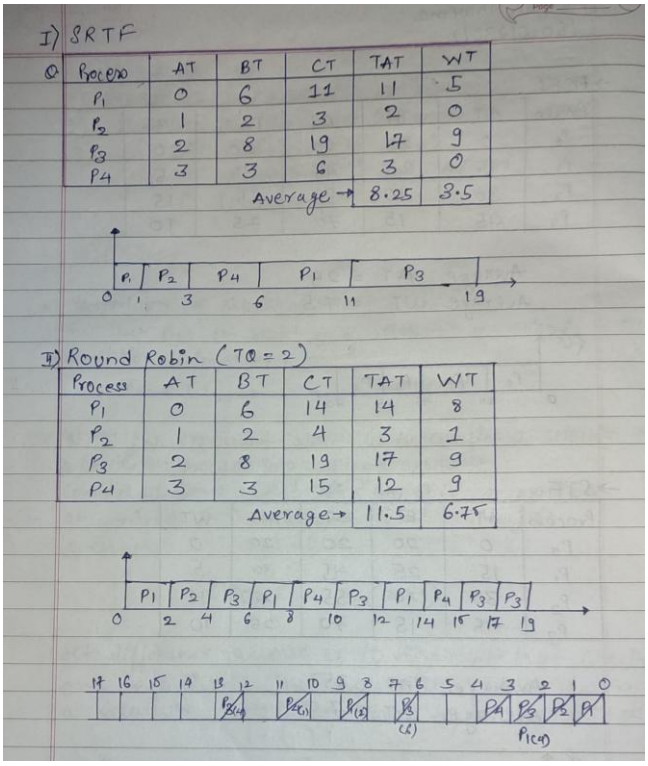
```
#include <bits/stdc++.h>
#define endl '\n'
using namespace std;

struct Process
{
  int pid;
  float arrival_time;
```

```
  float burst_time;
  float remaining_burst_time;
  float completion_time;
  float turnaround_time;
  float waiting_time;
};

void findCompletionTime(Process proc[], int n)
{
  float current_time = 0;
  int completed = 0;
  vector<bool> is_completed(n, false);

  while (completed < n)
  {
    int index = -1;
    float min_remaining_time = FLT_MAX;

    for (int i = 0; i < n; i++)
    {
      if (proc[i].arrival_time <= current_time && !is_completed[i] &&
proc[i].remaining_burst_time < min_remaining_time)
      {
        min_remaining_time = proc[i].remaining_burst_time;
        index = i;
      }
    }

    if (index != -1)
    {
      proc[index].remaining_burst_time -= 1;
      current_time += 1;

      if (proc[index].remaining_burst_time == 0)
      {
        proc[index].completion_time = current_time;
        proc[index].turnaround_time = proc[index].completion_time -
proc[index].arrival_time;
        proc[index].waiting_time = proc[index].turnaround_time -
proc[index].burst_time;
        is_completed[index] = true;
        completed++;
      }
    }
    else
    {
```

```
      current_time++;
    }
  }
}

void findavgTime(Process proc[], int n)
{
  float total_wt = 0, total_tat = 0;
  for (int i = 0; i < n; i++)
  {
    total_wt += proc[i].waiting_time;
    total_tat += proc[i].turnaround_time;
  }
  cout << "Average turnaround time = " << total_tat / n << endl;
  cout << "Average waiting time = " << total_wt / n << endl;
}

int main()
{
  int n;
  cout << "Enter the number of processes: ";
  cin >> n;
  Process proc[n];
  for (int i = 0; i < n; i++)
  {
    cout << "Enter the arrival time of process " << i + 1 << ": ";
    cin >> proc[i].arrival_time;
    cout << "Enter the burst time of process " << i + 1 << ": ";
    cin >> proc[i].burst_time;
    proc[i].pid = i + 1;
    proc[i].remaining_burst_time = proc[i].burst_time;
  }

  findCompletionTime(proc, n);
  cout << endl;
  cout << "PN\tAT\tBT\tCT\tTAT\tWT" << endl;
  for (int i = 0; i < n; i++)
  {
    cout << proc[i].pid << "\t" << proc[i].arrival_time << "\t" <<
proc[i].burst_time << "\t" << proc[i].completion_time << "\t" <<
proc[i].turnaround_time << "\t" << proc[i].waiting_time << endl;
  }
  findavgTime(proc, n);
}
```

```
Enter the number of processes: 4
Enter the arrival time of process 1: 0
Enter the burst time of process 1: 6
Enter the arrival time of process 2: 1
Enter the burst time of process 2: 2
Enter the arrival time of process 3: 2
Enter the burst time of process 3: 8
Enter the arrival time of process 4: 3
Enter the burst time of process 4: 3

PN      AT      BT      CT      TAT     WT
1       0       6       11      11      5
2       1       2       3       2       0
3       2       8       19      17      9
4       3       3       6       3       0
Average turnaround time = 8.25
Average waiting time = 3.5
```

Lab Work:



Conclusion:

I have completed the implementation of Round Robin (RR) and Shortest Remaining Time Next (SRTN) scheduling algorithms, gaining a deeper understanding of CPU scheduling and process management. Through Round Robin, I observed how time quantum affects fairness and efficiency, balancing context switching and response time. SRTN highlighted the preemptive nature of scheduling, prioritizing shorter jobs but risking starvation for longer ones. This experiment reinforced that choosing the right scheduling algorithm depends on system needs, improving my understanding of real-world OS scheduling strategies.

Post Lab Descriptive Questions

1. **Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?**

Process	Total Time	I/O (20%)	Compute (70%)	I/O (10%)
P1	10	2	7	1
P2	20	4	14	2
P3	30	6	21	3

The first I/O phase for all processes runs in parallel, so CPU is idle for max(2, 4, 6) = 6. Again, the OS overlaps I/O operations, so CPU remains idle for max(1, 2, 3) = 3. The total CPU idle time is 6 + 3 = 9.

CPU Idle % = (Idle Time / Total Execution Time) * 100
= (9 / 30) * 100 = 30 %

2. **What effect the time quantum has on its performance. What are the advantages and disadvantages of using a small versus a large time quantum?**
The time quantum in Round Robin (RR) scheduling is a crucial factor that influences CPU performance, response time, waiting time, and context switching overhead. It determines how long each process gets to execute before the CPU switches to the next process in the queue. The choice of time quantum directly impacts the efficiency and fairness of process scheduling.
Small Time Quantum:
- Advantages:
  - Ensures better responsiveness for all processes
  - Provides a fairer time-sharing system, preventing long processes from dominating
- Disadvantages:

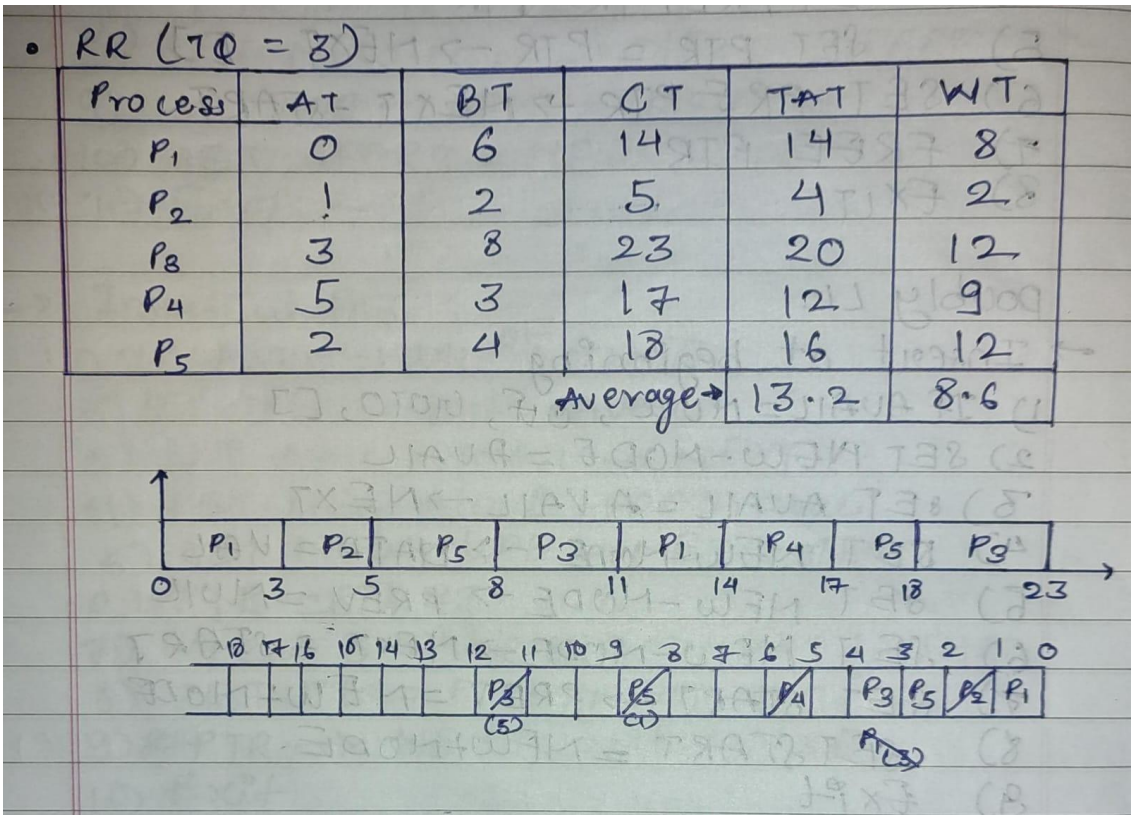
- Frequent context switching increases CPU overhead
- Reduces overall CPU efficiency

Large Time Quantum:

- Advantages:
  - Reduces context switching, improving CPU efficiency
  - Long processes get more execution time, leading to lower waiting times
- Disadvantages:
  - Response time increases for short processes
  - Can behave like FCFS (First-Come-First-Serve) if the quantum is too large, reducing fairness

3. **The following processes are scheduled using the Robin process scheduling policy with a time quantum of 3ms. Determine the average waiting time.**

Process	Arrival Time	Burst Time
P1	0	6
P2	1	2
P3	3	8
P4	5	3
P5	2	4



Date: 11 / 03 / 2025 **Signature of faculty in-charge**