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

Experiment No. 04

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

Signature of the Staff In-charge with date

TITLE: CPU Scheduling – Non Preemptive

AIM: Implementation of Basic CPU Scheduling Algorithms – Non Preemptive [FCFS , SJF]

Expected Outcome of Experiment:

CO 2 Illustrate and analyse the Process, threads, process scheduling and thread scheduling

---

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 Edition.**

3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**

4. **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:

First-Come, First-Served Scheduling:
First come First served (FCFS), is the simplest scheduling algorithm. FIFO simply queues processes according to the order they arrive in the ready queue. In this algorithm, the process that comes first will be executed first and next process starts only after the previous gets fully executed.

Shortest job first:
The shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN, also known as Shortest Job Next (SJN), can be preemptive or non-preemptive

Implementation details: (printout of code)
FCFS

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

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

```
    float turnaround_time;
    float waiting_time;
};

void findCompletionTime(Process proc[], int n)
{
    float completion_time = 0;
    for (int i = 0; i < n; i++)
    {
        completion_time += proc[i].burst_time;
        proc[i].completion_time = completion_time;
    }
}

void findTurnAroundTime(Process proc[], int n)
{
    for (int i = 0; i < n; i++)
    {
        proc[i].turnaround_time = proc[i].completion_time -
proc[i].arrival_time;
    }
}

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

    sort(proc, proc + n, [](Process a, Process b)
         { return a.arrival_time < b.arrival_time; });

    findCompletionTime(proc, n);
    findTurnAroundTime(proc, n);
    findWaitingTime(proc, n);

    cout << "PN\tAT\tBT\tCT\tWT\tTAT" << 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].waiting_time << "\t" << proc[i].turnaround_time << endl;
    }

    findavgTime(proc, n);

    return 0;
}
```

Output:

```
Enter the number of processes: 4
Enter the arrival time of process 1: 0
Enter the burst time of process 1: 20
Enter the arrival time of process 2: 15
Enter the burst time of process 2: 25
Enter the arrival time of process 3: 30
Enter the burst time of process 3: 10
Enter the arrival time of process 4: 45
Enter the burst time of process 4: 15
PN      AT      BT      CT      WT      TAT
1       0       20      20      0       20
2       15      25      45      5       30
3       30      10      55      15      25
4       45      15      70      10      25
Average waiting time = 7.5
Average turnaround time = 25
```

SJN:

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

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

bool compare(Process a, Process b)
{
  if (a.arrival_time == b.arrival_time)
  {
    return a.burst_time < b.burst_time;
  }
  return a.arrival_time < b.arrival_time;
}

void sjfScheduling(Process proc[], int n)
{
  sort(proc, proc + n, compare);
  int currentTime = 0;
```

```
  for (int i = 0; i < n; i++)
  {
    int minIndex = -1;
    int minburst_time = INT_MAX;
    for (int j = 0; j < n; j++)
    {
      if (proc[j].arrival_time <= currentTime && proc[j].completion_time
== 0 && proc[j].burst_time < minburst_time)
      {
        minburst_time = proc[j].burst_time;
        minIndex = j;
      }
    }

    if (minIndex == -1)
    {
      currentTime++;
      i--;
      continue;
    }

    currentTime += proc[minIndex].burst_time;
    proc[minIndex].completion_time = currentTime;
    proc[minIndex].turnaround_time = proc[minIndex].completion_time -
proc[minIndex].arrival_time;
    proc[minIndex].waiting_time = proc[minIndex].turnaround_time -
proc[minIndex].burst_time;
  }
}

void findavgTime(Process proc[], int n)
{
  float avg_wt = 0, avg_tat = 0;
  for (int i = 0; i < n; i++)
  {
    avg_wt += proc[i].waiting_time;
    avg_tat += proc[i].turnaround_time;
  }

  cout << "Average waiting time = " << avg_wt / n << endl;
  cout << "Average turnaround time = " << avg_tat / 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;
  }

  sjfScheduling(proc, n);

  cout << "PN\tAT\tBT\tCT\tWT\tTAT" << 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].waiting_time << "\t" << proc[i].turnaround_time << endl;
  }

  findavgTime(proc, n);
}
```

OUTPUT:

```
Enter the number of processes: 4
Enter the arrival time of process 1: 0
Enter the burst time of process 1: 20
Enter the arrival time of process 2: 15
Enter the burst time of process 2: 25
Enter the arrival time of process 3: 30
Enter the burst time of process 3: 10
Enter the arrival time of process 4: 45
Enter the burst time of process 4: 15
PN      AT      BT      CT      TAT     WT
1       0       20      20      20      0
2       15      25      45      30      5
3       30      10      55      25      15
4       45      15      70      25      10
Average waiting time = 7.5
Average turnaround time = 25
```

Conclusion:

I implemented both the FCFS (First-Come-First-Serve) and SJN (Shortest Job Next) scheduling algorithms. FCFS is simple and fair, processing jobs in the order they arrive. However, I noticed that it can be inefficient, especially when longer processes are executed first, causing a "convoy effect" that increases average waiting times for shorter processes. On the other hand, SJN minimizes waiting time by prioritizing shorter tasks, making it more efficient overall. However, I found that it requires knowledge of the exact burst time for each process, which can be difficult in real-time systems, and it may lead to starvation for longer processes if shorter ones keep arriving. Both algorithms have their pros and cons, with FCFS being easier to implement but less optimal, and SJN being more efficient but harder to implement in practice.

Multiple-Choice Questions (MCQs)

1. **In FCFS scheduling, the process that arrives first:**

   A) Gets executed first

   B) Gets executed last

   C) Has the highest priority

   D) Has the shortest burst time

2. **Which scheduling algorithm can cause the "Convoy Effect"?**

   A) FCFS

   B) SJF

C) Round Robin

D) Priority Scheduling

3. **SJF scheduling algorithm is also known as:**

A) Shortest Remaining Time First

B) Shortest Time Next

C) Longest Job First

Post Lab Descriptive Questions

1. Compare FCFS and SJF.

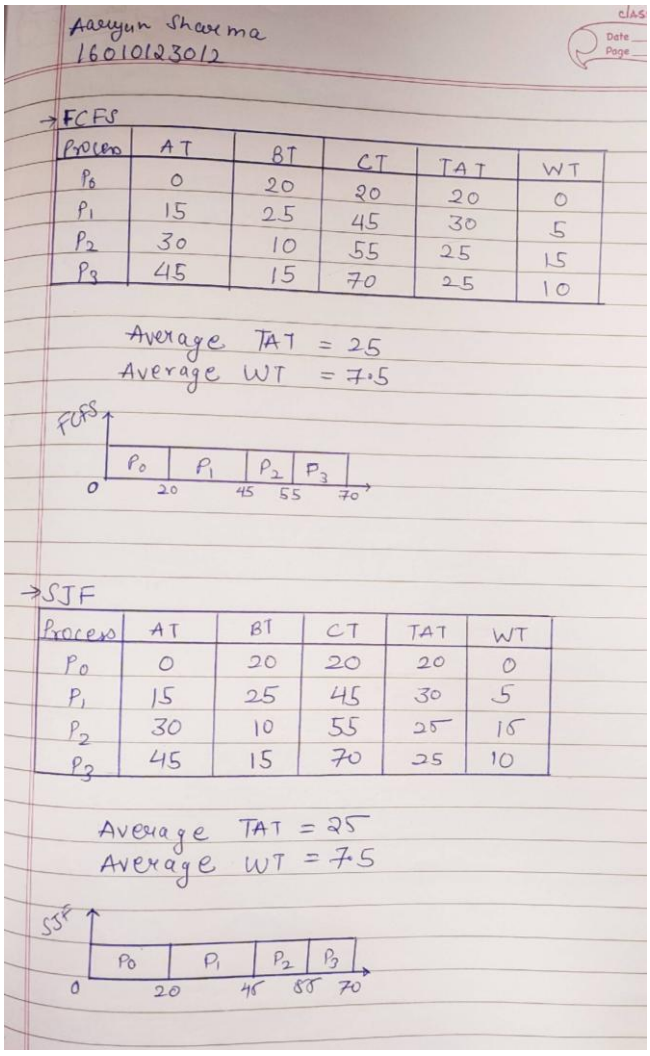
Characteristic	FCFS	SJF
Order of Execution	By arrival time	By shortest burst time
Efficiency	Can cause high waiting times due to long processes running first	Generally minimizes waiting time by prioritizing short jobs
Complexity	Simple to implement	More complex, requires knowledge of burst times
Average Waiting Time	Higher due to long processes delaying shorter ones	Lower, especially when shorter jobs arrive first
Fairness	Fair in terms of arrival order	May not be fair to longer processes, as they may starve
Starvation	Not an issue	Can lead to starvation of longer jobs

2. Discuss the impact of SJF scheduling on the average waiting time of processes.
SJF reduces average waiting time by prioritizing shorter processes, which helps complete them quickly, allowing other processes to wait less. This minimizes delays and leads to a lower average waiting time compared to FCFS. However, SJF can cause starvation for longer processes, as they may be continuously delayed by incoming shorter processes.
3. Consider a set of 4 processes with their respective arrival and burst times given in milliseconds.

Process	Arrival Time	Burst Time
P0	0	20
P1	15	25
P2	30	10
P3	45	15

A. **Apply the First-Come, First-Served (FCFS) and Shortest Job First (SJF) scheduling algorithms.**
B. **For each scheduling method:**
   - Draw the Gantt Chart.
   - Calculate the Average Turnaround Time.
   - Calculate the Average Waiting Time.

C. **Analyze the results and comment on the performance of FCFS and SJF for the given process set.**
   - SJF performs better than FCFS in reducing turnaround time and response time. However, it requires prior knowledge of burst times, making it difficult to implement in real-time systems.
   - FCFS is simple but can lead to inefficiencies, especially when long processes arrive early.



Date: 11/02/2025 **Signature of faculty in-charge**