**EXPERIMENT NO: 3 b** **DATE –**

# Non-Pre-emptive CPU Scheduling Algorithm

## (a) Shortest Job First CPU Scheduling Algorithm

**Aim-**To implement Shortest Job First CPU scheduling algorithm

**THEORY –**

**Shortest Job First (SJF) Scheduling**

Shortest Job First (SJF) is a **non-preemptive scheduling algorithm** in which the process with the **smallest burst time (execution time)** is executed first. It is also known as **Shortest Job Next (SJN)**.

- **How it Works:**

1. The scheduler selects the process with the **smallest burst time** from the ready queue.
2. Once a process is allocated the CPU, it runs till completion (no preemption).
3. If two processes have the same burst time, the one that arrived first (based on Arrival Time) is executed first.

SJF gives **minimum average waiting time** among all non-preemptive scheduling algorithms. However, it may cause the **"starvation" problem**, where longer processes may be postponed indefinitely if short processes keep arriving.

**Parameters Used in SJF Scheduling**

1. **Arrival Time (AT):**
    - The time at which a process enters the ready queue (i.e., when it becomes available for execution).
2. **Burst Time (BT):**
    - The total time required by a process for execution on the CPU.

- In SJF, scheduling decisions are based on the **smallest BT**.

3. **Completion Time (CT):**
    - The time at which a process finishes execution.
    - It is calculated after all processes are scheduled.
4. **Turnaround Time (TAT):**
    - The total time a process spends in the system, from its arrival until its completion.
    - Formula:

$$TAT=CT-AT$$

**Waiting Time (WT):**

- The amount of time a process spends **waiting in the ready queue** before it gets CPU time.
- Formula:

$$WT=TAT-BT$$

EXAMPLE –

| Process | AT | BT |
|---|---|---|
| P1 | 2 | 10 |
| P2 | 0 | 6 |
| P3 | 1 | 2 |
| P4 | 7 | 8 |

**-GANTT diagram -**

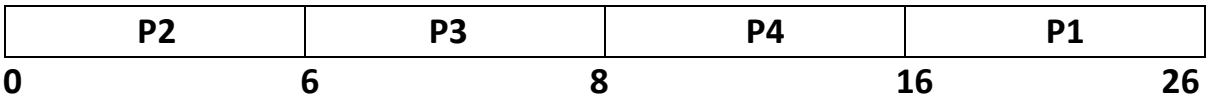


**TABLE -**

| PROCESS | AT | BT | CT | TAT | WT |
|---|---|---|---|---|---|
| P1 | 2 | 10 | 26 | 24 | 14 |
| P2 | 0 | 6 | 6 | 6 | 0 |
| P3 | 1 | 2 | 8 | 7 | 5 |
| P4 | 7 | 8 | 16 | 9 | 1 |

**PROGRAM –**

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <iomanip>

using namespace std;


struct Process {

   string name;

   int arrival, burst, start, completion, turnaround, waiting;

};


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;

   }


   vector<bool> done(n, false);

   int time = 0, completed = 0;

   float avgTAT = 0, avgWT = 0;

   int totalTAT = 0;

   vector<string> ganttOrder;

   vector<int> ganttTime;


   while (completed < n) {

      int idx = -1, minBT = 1e9;

      for (int i = 0; i < n; i++) {

         if (!done[i] && p[i].arrival <= time && p[i].burst < minBT) {

            minBT = p[i].burst;

            idx = i;

         }

      }

      if (idx == -1) {

         // No process available, add idle time

         int nextArrival = 1e9;

         for (int i = 0; i < n; i++) {

            if (!done[i] && p[i].arrival < nextArrival) {

               nextArrival = p[i].arrival;

            }

         }

         ganttOrder.push_back("Idle");

         ganttTime.push_back(nextArrival);

         time = nextArrival;

      } else {

         p[idx].start = time;

         time += p[idx].burst;

         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++;

            ganttOrder.push_back(p[idx].name);

            ganttTime.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 << "Total Turnaround Time = " << totalTAT << endl;

    cout << "Average Turnaround Time = " << avgTAT / n << endl;

    cout << "Average Waiting Time = " << avgWT / n << endl;



    cout << "\nGantt Chart:\n";

    cout << "----------------------------------------------------------\n";

    for (auto &proc : ganttOrder) {

        cout << "| " << setw(5) << proc << " ";

    }

    cout << "|\n";

    cout << "----------------------------------------------------------\n";

    cout << 0;

    for (auto &t : ganttTime) {

        cout << setw(7) << t;

    }

    cout << "\n";



    return 0;

}
```

**OUTPUT –**

```
Enter number of processes: 4

Enter Process Name, Arrival Time, Burst Time for P1: P1
2
10

Enter Process Name, Arrival Time, Burst Time for P2: P2
0
6

Enter Process Name, Arrival Time, Burst Time for P3: P3
1
2

Enter Process Name, Arrival Time, Burst Time for P4: P4
7
8

------------------------------------------------------------
Process AT      BT      CT      TAT     WT
------------------------------------------------------------
P1      2       10      26      24      14
P2      0       6       6       6       0
P3      1       2       8       7       5
P4      7       8       16      9       1
------------------------------------------------------------
Total Turnaround Time = 46
Average Turnaround Time = 11.5
Average Waiting Time = 5

Gantt Chart:
------------------------------------------------------------
|    P2 |    P3 |    P4 |    P1 |
------------------------------------------------------------
0      6      8     16     26
```

**Conclusion** – The experiment clearly demonstrates the working of the Shortest Job First (SJF) scheduling algorithm. While it ensures fairness by executing shorter processes first and reduces turnaround time for them, its non-preemptive nature often increases the average waiting time for longer processes, leading to the convoy effect.