Department of Computer Engineering

Batch: E2	**Roll No.: 16010123325**
Experiment No. 04	
Grade: AA / AB / BB / BC / CC / CD /DD	

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:

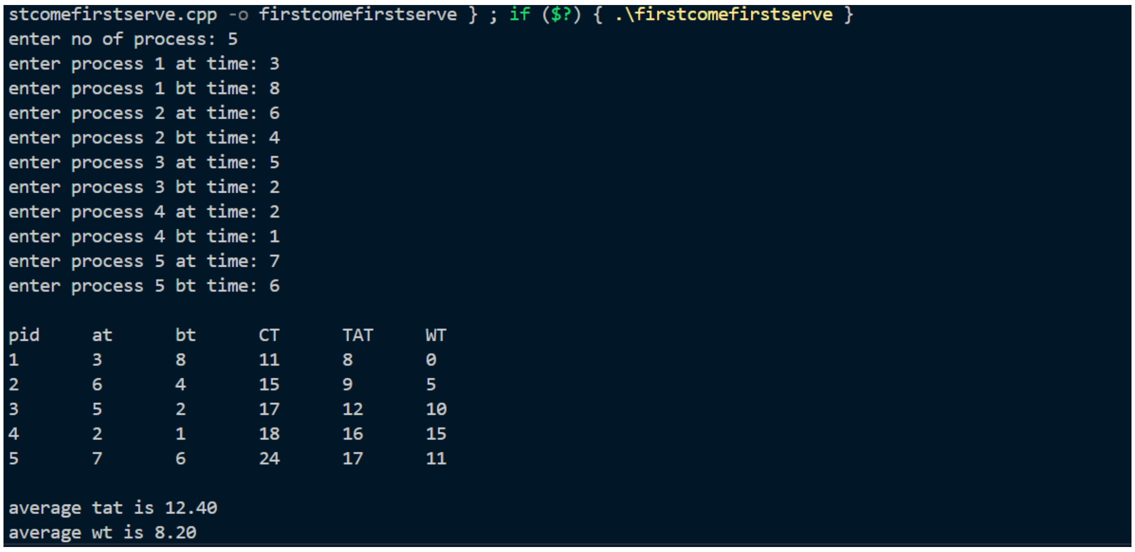
Implementation details: (printout of code)

First-Come, First-Served Scheduling:

```
#include <bits/stdc++.h>

using namespace std;

struct Process {

   int pid;  // Process ID

   int at;   // Arrival Time

   int bt;   // Burst Time

   int ct;   // Completion Time

   int tat;  // Turnaround Time

   int wt;   // Waiting Time

};
```

```
void findtime(Process proc[], int n) {
    int current_time = 0;
    double total_tat = 0, total_wt = 0;

    for (int i = 0; i < n; i++) {
        if (current_time < proc[i].at) {
            current_time = proc[i].at;
        }

        // CT = current time + BT
        proc[i].ct = current_time + proc[i].bt;
        current_time = proc[i].ct;

        // TAT = CT-AT
        proc[i].tat = proc[i].ct - proc[i].at;

        // WT = TAT-BT
        proc[i].wt = proc[i].tat - proc[i].bt;

        total_tat += proc[i].tat;
        total_wt += proc[i].wt;
    }

    cout << "\npid\tat\tbt\tCT\tTAT\tWT\n";
    for (int i = 0; i < n; i++) {
        cout << proc[i].pid << "\t" << proc[i].at << "\t"
```

```
            << proc[i].bt << "\t" << proc[i].ct << "\t"
            << proc[i].tat << "\t" << proc[i].wt << endl;
    }

    cout << "\naverage tat is " << fixed << setprecision(2) << total_tat / n << endl;
    cout << "average wt is " << fixed << setprecision(2) << total_wt / n << endl;
}

int main() {
    int n;
    cout << "enter no of process: ";
    cin >> n;

    Process proc[n];

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        cout << "enter process " << proc[i].pid << " at time: ";
        cin >> proc[i].at;
        cout << "enter process " << proc[i].pid << " bt time: ";
        cin >> proc[i].bt;
    }

    findtime(proc, n);
    return 0;
}
```



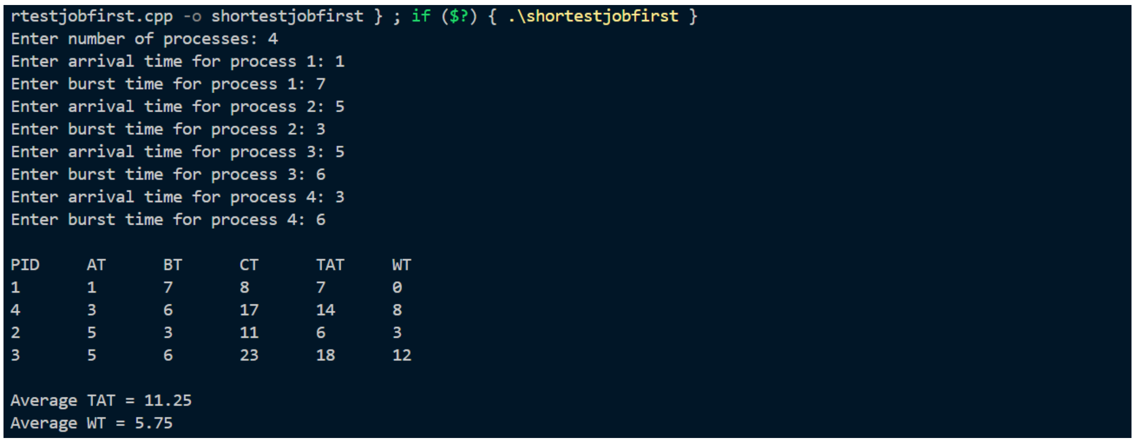
```
stcomefirstserve.cpp -o firstcomefirstserve } ; if ($?) { .\firstcomefirstserve }
enter no of process: 5
enter process 1 at time: 3
enter process 1 bt time: 8
enter process 2 at time: 6
enter process 2 bt time: 4
enter process 3 at time: 5
enter process 3 bt time: 2
enter process 4 at time: 2
enter process 4 bt time: 1
enter process 5 at time: 7
enter process 5 bt time: 6

pid     at      bt      CT      TAT     WT
1       3       8       11      8       0
2       6       4       15      9       5
3       5       2       17      12      10
4       2       1       18      16      15
5       7       6       24      17      11

average tat is 12.40
average wt is 8.20
```

Shortest job first :

```
#include <bits/stdc++.h>

using namespace std;

struct Process {

    int pid;

    int at;

    int bt;

    int ct;

    int tat;

    int wt;

};


int main() {

    int n;

    cout << "Enter number of processes: ";

    cin >> n;
```

```
vector<Process> procs(n);
for(int i = 0; i < n; i++){
    procs[i].pid = i + 1;
    cout << "Enter arrival time for process " << procs[i].pid << ": ";
    cin >> procs[i].at;
    cout << "Enter burst time for process " << procs[i].pid << ": ";
    cin >> procs[i].bt;
}

sort(procs.begin(), procs.end(), [](auto &a, auto &b){
    return a.at < b.at;
});

int completed = 0, current_time = 0;
double total_wt = 0, total_tat = 0;
vector<bool> done(n, false);

while(completed < n){
    int idx = -1, minBT = INT_MAX;
    for(int i = 0; i < n; i++){
        if(!done[i] && procs[i].at <= current_time && procs[i].bt < minBT){
            minBT = procs[i].bt;
            idx = i;
        }
    }
```

```
        if(idx == -1){
            current_time++;
        } else {
            current_time += procs[idx].bt;
            procs[idx].ct = current_time;
            procs[idx].tat = procs[idx].ct - procs[idx].at;
            procs[idx].wt = procs[idx].tat - procs[idx].bt;
            total_wt += procs[idx].wt;
            total_tat += procs[idx].tat;
            done[idx] = true;
            completed++;
        }
    }

    cout << "\nPID\tAT\tBT\tCT\tTAT\tWT\n";
    for(auto &p : procs){
        cout << p.pid << "\t" << p.at << "\t" << p.bt << "\t" << p.ct << "\t" << p.tat << "\t"
<< p.wt << "\n";
    }

    cout << "\nAverage TAT = " << total_tat / n
         << "\nAverage WT = " << total_wt / n << "\n";
    return 0;
}
```



```
rtestjobfirst.cpp -o shortestjobfirst } ; if ($?) { .\shortestjobfirst }
Enter number of processes: 4
Enter arrival time for process 1: 1
Enter burst time for process 1: 7
Enter arrival time for process 2: 5
Enter burst time for process 2: 3
Enter arrival time for process 3: 5
Enter burst time for process 3: 6
Enter arrival time for process 4: 3
Enter burst time for process 4: 6

PID     AT      BT      CT      TAT     WT
1       1       7       8       7       0
4       3       6       17      14      8
2       5       3       11      6       3
3       5       6       23      18      12

Average TAT = 11.25
Average WT = 5.75
```

Conclusion :

In this experiment, we implemented two non-preemptive CPU scheduling algorithms: **FCFS** and **SJF**.

- **FCFS** schedules processes based on arrival time, which can lead to higher waiting times due to long processes delaying shorter ones.

- **SJF** prioritizes processes with the shortest burst time, resulting in lower waiting times, but it requires knowing the burst time in advance.

Both algorithms have their strengths and limitations, with **FCFS** being simple but inefficient in certain cases, and **SJF** being more efficient but less practical in real-time scenarios.

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.

Criteria	FCFS (First-Come, First-Served)	SJF (Shortest Job First)
Execution Order	Executes processes in the order of arrival.	Executes the process with the shortest burst time first.
Waiting Time	Can be high, especially if a long process arrives first.	Lower average waiting time as shorter processes are executed first.
Fairness	Can be unfair for short jobs if long jobs arrive first.	Can be unfair if long jobs are repeatedly delayed.
Complexity	Simple to implement.	Requires knowing the burst time of processes in advance.
Starvation	No starvation issue; every process is executed in arrival order.	Starvation can occur if shorter jobs continuously arrive, delaying long jobs.
Performance	Can lead to poor performance when processes have varying burst times.	Generally more efficient in reducing waiting time.
Best Use Case	Suitable for systems where jobs arrive in a predictable sequence.	Best for systems where burst times are known and jobs are mostly short.

2. Discuss the impact of SJF scheduling on the average waiting time of processes.

SJF scheduling tends to reduce the average waiting time of processes compared to FCFS because it prioritizes shorter burst times. By executing shorter processes first, SJF minimizes the time processes spend waiting in the ready queue. This results in:

- **Lower average waiting time**: Short jobs get executed sooner, reducing the waiting time for other jobs.
- **Better response for short tasks**: Processes that require less CPU time are completed quickly, improving system responsiveness.

However, the main drawback of SJF is that it can lead to **starvation** for long processes if short jobs keep arriving. The system may continually execute shorter jobs, causing longer tasks to be delayed indefinitely if no short jobs arrive to preempt them.

Thus, while SJF improves the overall waiting time, it can cause fairness issues and difficulty in practical scenarios where burst times are not predictable.

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.**

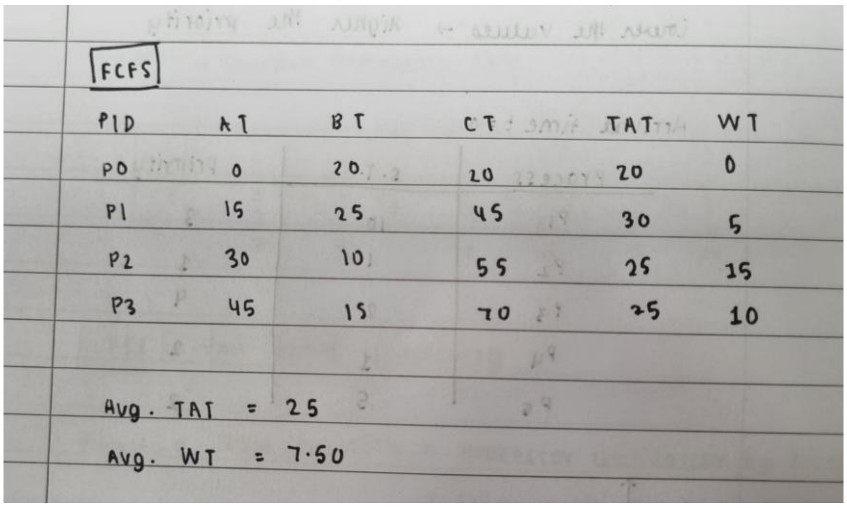
B. **For each scheduling method:**
   - Draw the Gantt Chart.
   - Calculate the Average Turnaround Time.
   - Calculate the Average Waiting Time.

C. **Analyze the results and comment on the performance of FCFS and SJF for the given process set.**

FCFS (First-Come, First-Served) Scheduling:

Execution Order:

- P0 arrives first, so it executes first.
- P1 arrives next but waits for P0 to finish.
- P2 arrives after P1 and waits for P1 to finish.
- P3 arrives last and waits for P2 to finish.

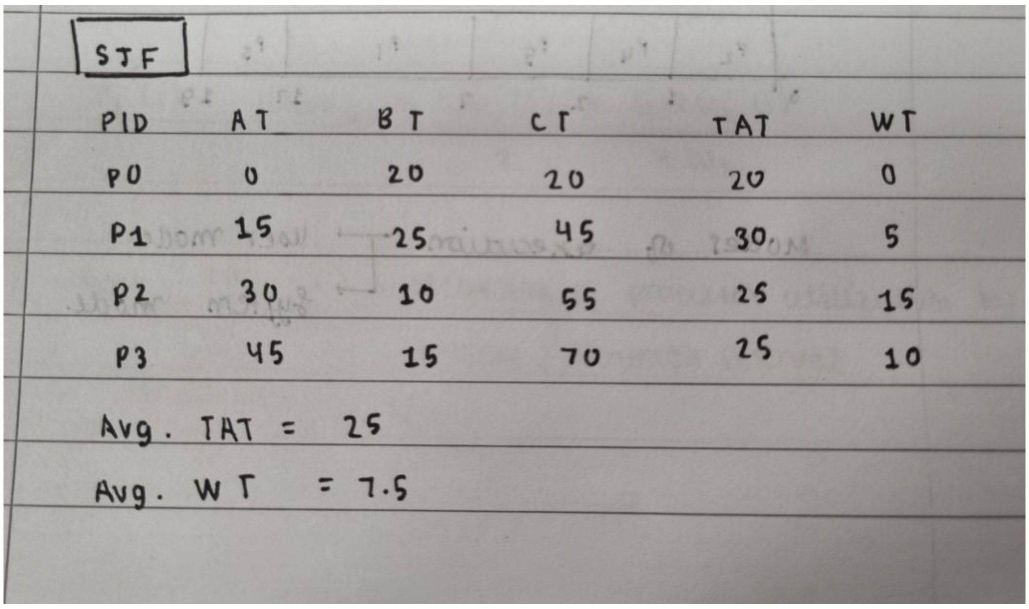


FCFS

PID	AT	BT	CT	TAT	WT
PO	0	20	20	20	0
P1	15	25	45	30	5
P2	30	10	55	25	15
P3	45	15	70	25	10

Avg. TAT = 25

Avg. WT = 7.50

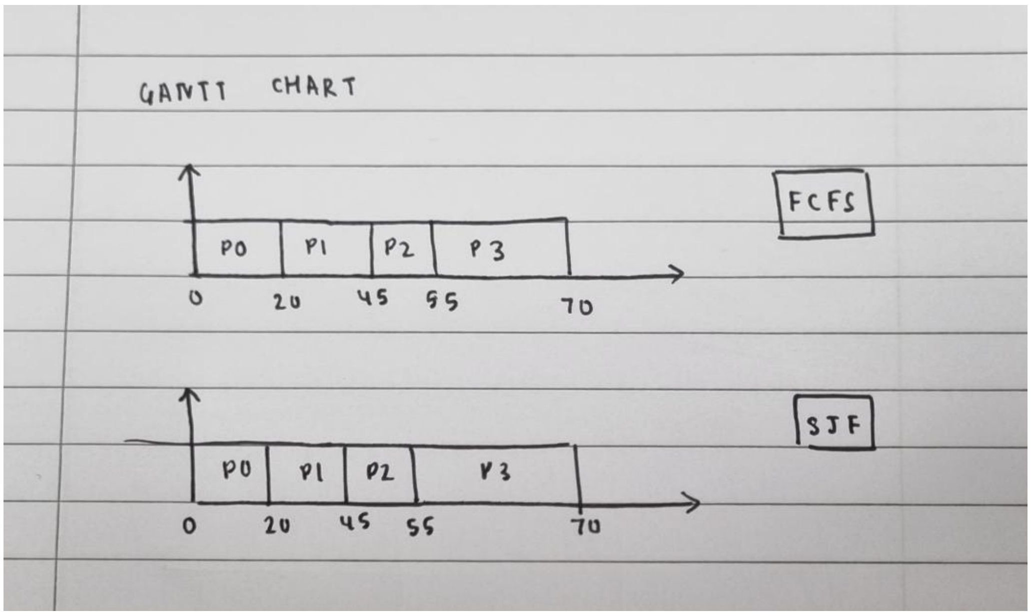


SJF

PID	AT	BT	CT	TAT	WT
P0	0	20	20	20	0
P1	15	25	45	30	5
P2	30	10	55	25	15
P3	45	15	70	25	10

Avg. TAT = 25
Avg. WT = 7.5

Gantt Chart



Here, since the burst times and arrival times are such that both algorithms result in the same execution order, the turnaround time for each process ends up being identical.

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