# EXPERIMENT – 1

**Aim: Write a program to implement CPU scheduling for first come first serve.**

**Theory:**

In First Come First Serve (FCFS) CPU scheduling, the process that arrives first is the one that gets executed first. It's like standing in line at a grocery store — the person who arrived first is the one who gets to check out first. By default, it assumes non-pre-emptive processes entering the scheduler.

Algorithm:

1. Arrival of Processes: When processes arrive at the CPU, they are added to the end of the ready queue. The ready queue is essentially a line of processes waiting for their turn to be executed.
2. Execution: The CPU executes the processes in the order they arrive. The first process that entered the queue is the first one to be selected for execution.
3. Completion: Once a process finishes its execution, the next process in the ready queue is picked and executed.

This scheduling algorithm is simple to understand and implement, but it has some drawbacks. For example, it may lead to "convoy effect," where small processes get stuck waiting behind a long process, causing inefficiency.

Imagine you have three processes: A, B, and C. If A arrives first, it will be executed first. Once A completes, B gets its turn, and so on. The order of execution would be A → B → C.

**Program Code:**

#include <stdio.h>

#include <stdbool.h>

struct Process

{

int id;

int arrivalTime;

int burstTime;

};

bool isEarly(struct Process \*p1, struct Process \*p2)

{

return (p1->arrivalTime <= p2->arrivalTime);

};

int main()

{

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

// Input process details

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

{

processes[i].id = i + 1;

printf("Enter arrival time for process %d:\t", i + 1);

scanf("%d", &processes[i].arrivalTime);

printf("Enter burst time for process %d:\t\t", i + 1);

scanf("%d", &processes[i].burstTime);

}

// Sorting Processes by arrival time

struct Process a;

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

{

for (int j = i + 1; j < n; ++j)

{

if (isEarly(&processes[j], &processes[i]))

{

a = processes[i];

processes[i] = processes[j];

processes[j] = a;

}

}

}

int waitingTime[n], turnaroundTime[n];

waitingTime[0] = 0;

for (int i = 1; i < n; i++)

{

waitingTime[i] = waitingTime[i - 1] + processes[i - 1].burstTime;

}

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

{

turnaroundTime[i] = waitingTime[i] + processes[i].burstTime;

}

printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

{

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime, processes[i].burstTime, waitingTime[i], turnaroundTime[i]);

}

printf("\n");

// Calculate average waiting time and average turnaround time

float averageWaitingTime = 0, averageTurnaroundTime = 0;

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

{

averageWaitingTime += waitingTime[i];

averageTurnaroundTime += turnaroundTime[i];

}

averageWaitingTime /= n;

averageTurnaroundTime /= n;

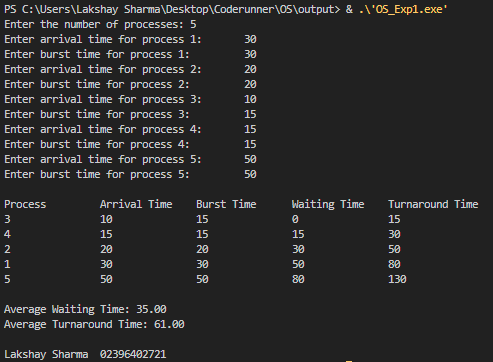
printf("Average Waiting Time: %.2f\n", averageWaitingTime);

printf("Average Turnaround Time: %.2f\n", averageTurnaroundTime);

printf("\nLakshay Sharma\t02396402721\n");

}

**Output:**

****

# EXPERIMENT – 2

**Aim: Write a program to implement CPU scheduling for shortest job first.**

**Theory:**

Shortest Job First (SJF) CPU scheduling is a non-pre-emptive scheduling algorithm where the process with the shortest burst time is selected for execution first. The burst time is the time required by a process to complete its execution.

Algorithm:

1. Arrival of Processes: When processes arrive at the CPU, they are added to the ready queue. The ready queue is ordered based on the burst time of the processes, with the shortest burst time at the front.
2. Selection of Process: The process with the shortest burst time in the ready queue is selected for execution. If two processes have the same burst time, FCFS (First Come First Serve) order is used to break the tie.
3. Execution: The selected process is executed until completion. Since SJF is non-pre-emptive, once a process starts its execution, it continues until it finishes.
4. Completion: After the current process completes its execution, the process with the next shortest burst time in the ready queue is selected for execution.
5. Repeat: Steps 2-4 are repeated until all processes are completed.

The main advantage of SJF scheduling is that it minimizes the average waiting time, making it an optimal scheduling algorithm in terms of waiting time. However, predicting the exact burst time of a process in a real system is challenging, and in practice, it may be hard to implement accurately.

There's also a variant called Shortest Remaining Time First (SRTF), which is a pre-emptive version of SJF. In SRTF, the scheduler can pre-emptively stop the currently running process if a new process arrives with a shorter burst time.

**Program Code:**

#include <stdio.h>

#include <stdbool.h>

struct Process

{

int id;

int arrivalTime;

int burstTime;

};

bool isEarly(struct Process \*p1, struct Process \*p2)

{

return (p1->burstTime <= p2->burstTime);

};

int main()

{

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

struct Process processes[n];

// Input process details

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

{

processes[i].id = i + 1;

printf("Enter arrival time for process %d:\t", i + 1);

scanf("%d", &processes[i].arrivalTime);

printf("Enter burst time for process %d:\t\t", i + 1);

scanf("%d", &processes[i].burstTime);

}

// Sorting Processes by arrival time

struct Process a;

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

{

for (int j = i + 1; j < n; ++j)

{

if (isEarly(&processes[j], &processes[i]))

{

a = processes[i];

processes[i] = processes[j];

processes[j] = a;

}

}

}

int waitingTime[n], turnaroundTime[n];

waitingTime[0] = 0;

for (int i = 1; i < n; i++)

{

waitingTime[i] = waitingTime[i - 1] + processes[i - 1].burstTime;

}

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

{

turnaroundTime[i] = waitingTime[i] + processes[i].burstTime;

}

printf("\nProcess\t\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

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

{

printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].arrivalTime, processes[i].burstTime, waitingTime[i], turnaroundTime[i]);

}

printf("\n");

// Calculate average waiting time and average turnaround time

float averageWaitingTime = 0, averageTurnaroundTime = 0;

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

{

averageWaitingTime += waitingTime[i];

averageTurnaroundTime += turnaroundTime[i];

}

averageWaitingTime /= n;

averageTurnaroundTime /= n;

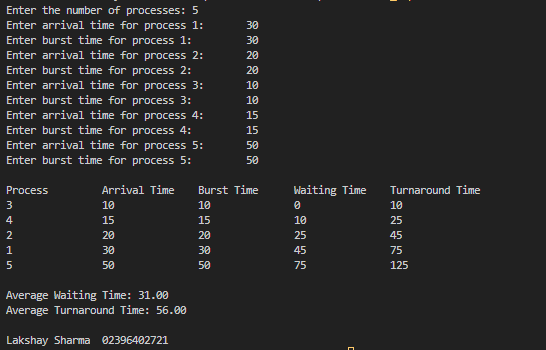
printf("Average Waiting Time: %.2f\n", averageWaitingTime);

printf("Average Turnaround Time: %.2f\n", averageTurnaroundTime);

printf("\nLakshay Sharma\t02396402721\n");

}

**Output:**

****

# EXPERIMENT – 3

**Aim: Write a program to perform priority scheduling.**

**Theory:**

Priority Scheduling in the context of CPU scheduling is an algorithm where each process is assigned a priority, and the process with the highest priority is selected for execution first. The priority of a process is typically determined based on factors such as the nature of the task, importance of the process, deadline constraints, or other criteria.

Here's how Priority Scheduling works:

Priority Assignment: Each process is assigned a priority value. The priority can be an integer or a real number, and the interpretation depends on the system's design. In some systems, a higher priority value indicates higher priority (as you mentioned in your case), while in others, lower values might indicate higher priority.

Arrival of Processes: When a process arrives at the CPU, it is added to the ready queue based on its priority.

Selection of Process: The process with the highest priority in the ready queue is selected for execution. If two processes have the same priority, other scheduling algorithms like First Come First Serve (FCFS) or Round Robin might be used as tiebreakers.

Execution: The selected process is allowed to execute until it completes its burst time or until it is preempted by a higher-priority process (preemptive priority scheduling).

Completion: After a process completes its execution, the scheduler selects the next process with the highest priority from the ready queue.

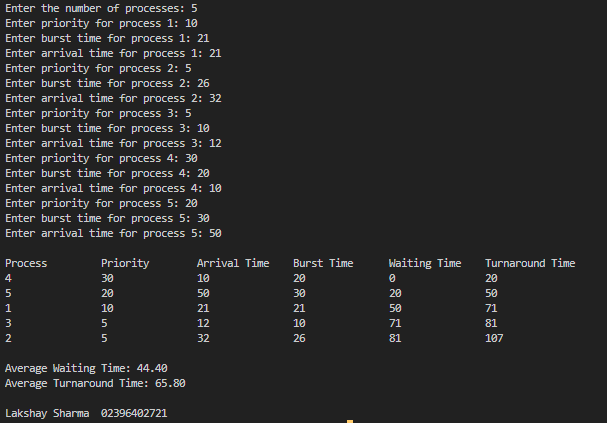
Repeat: Steps 3-5 are repeated until all processes are completed.

Preemptive Priority Scheduling: In preemptive priority scheduling, a currently running process can be preempted if a new process arrives with a higher priority. This ensures that the process with the highest priority gets the CPU immediately. The preempted process is then moved back to the ready queue.

Non-Preemptive Priority Scheduling: In non-preemptive priority scheduling, once a process starts its execution, it continues until it finishes, even if a higher-priority process arrives later. The new process with higher priority will be considered for execution only after the current process completes.

Priority Scheduling is advantageous as it allows the system to prioritize important or time-sensitive tasks. However, it can suffer from the "starvation" problem, where low-priority processes might not get a chance to execute if high-priority processes keep arriving.

**Program Code:**

****