Date:

SJF CPU SCHEDULING ALGORITHM

01. AIM :

Write a program to implement SJF CPU scheduling algorithms.

02. FACILITIES :

Hardware: A Linux-based system (e.g., Ubuntu, Fedora, or any other distribution).

Software: C programming language, GCC compiler for compiling and running the program.

IDE/Editor: Any text editor or IDE that supports C programming (e.g., Visual Studio Code, Vim, Sublime Text, or Eclipse).

03. SCOPE :

This experiment focuses on implementing the **Shortest Job First (SJF)** CPU scheduling algorithm. It includes:

1. Understanding how processes are scheduled based on their burst time, with the shortest job being executed first.
2. Calculating **Completion Time (CT), Turnaround Time (TAT)**, and **Waiting Time (WT)** for each process, considering their arrival and burst times.
3. Implementing the SJF algorithm in C and demonstrating how it schedules processes according to their burst times.

04 THEORY:

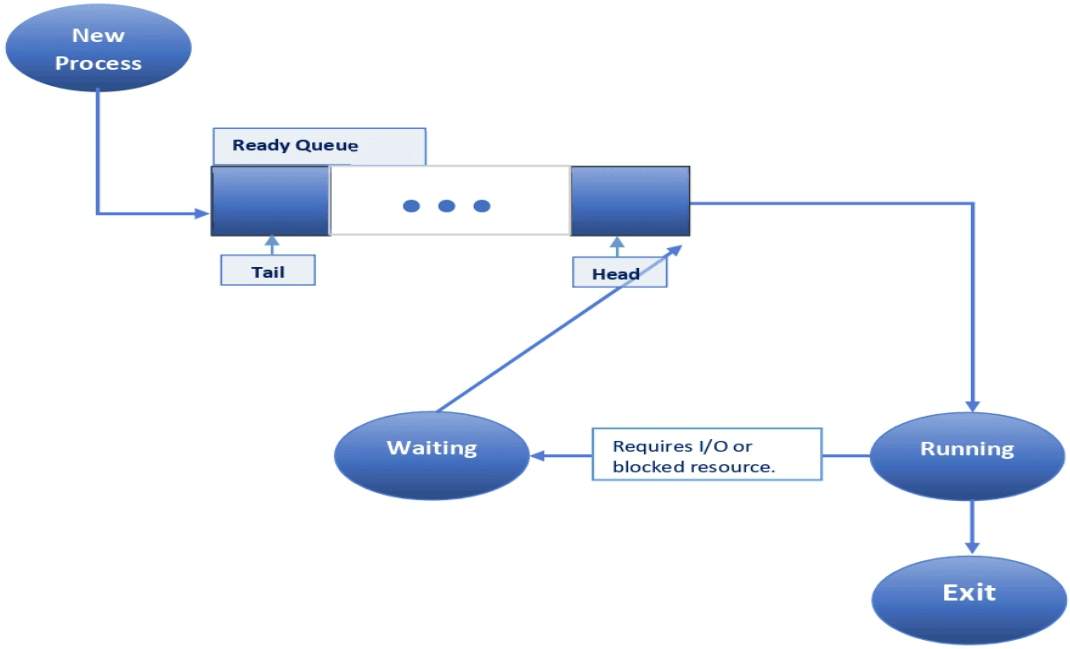
Non-Preemptive CPU Scheduling:

In **Non-preemptive CPU scheduling**, once a process is assigned to the CPU, it runs to completion without being interrupted, even if another process with a shorter burst time becomes available. This contrasts with preemptive scheduling, where a running process can be interrupted and placed back in the ready queue if a higher-priority process arrives.

The key characteristics of non-preemptive scheduling are:

1. **Process Continuity**: Once a process starts execution on the CPU, it runs until it finishes or voluntarily gives up the CPU (e.g., by waiting for I/O operations).
2. **No Interruptions**: The process does not get interrupted by other processes until it completes its CPU burst time.

3. **Scheduling**: The CPU scheduler selects the next process from the ready queue, typically based on its scheduling policy (e.g., First Come First Serve, Shortest Job First).



Shortest Job First (SJF) Scheduling Algorithm:

Shortest Job First (SJF) is a non-preemptive scheduling algorithm that selects the process with the shortest burst time (i.e., the process that requires the least CPU time to execute) from the ready queue for execution next. Since it is non-preemptive, once a process starts executing, it runs until completion.

There are two types of SJF:

1. **Preemptive SJF (also called Shortest Remaining Time First, SRTF)**: A process can be preempted if another process with a shorter remaining burst time arrives.
2. **Non-Preemptive SJF**: Once a process starts execution, it runs to completion without being preempted.

SJF aims to minimize the average waiting time and turnaround time. However, it requires knowledge of the CPU burst time of processes in advance, which is generally impractical. Nonetheless, in many practical systems, this information can be estimated.

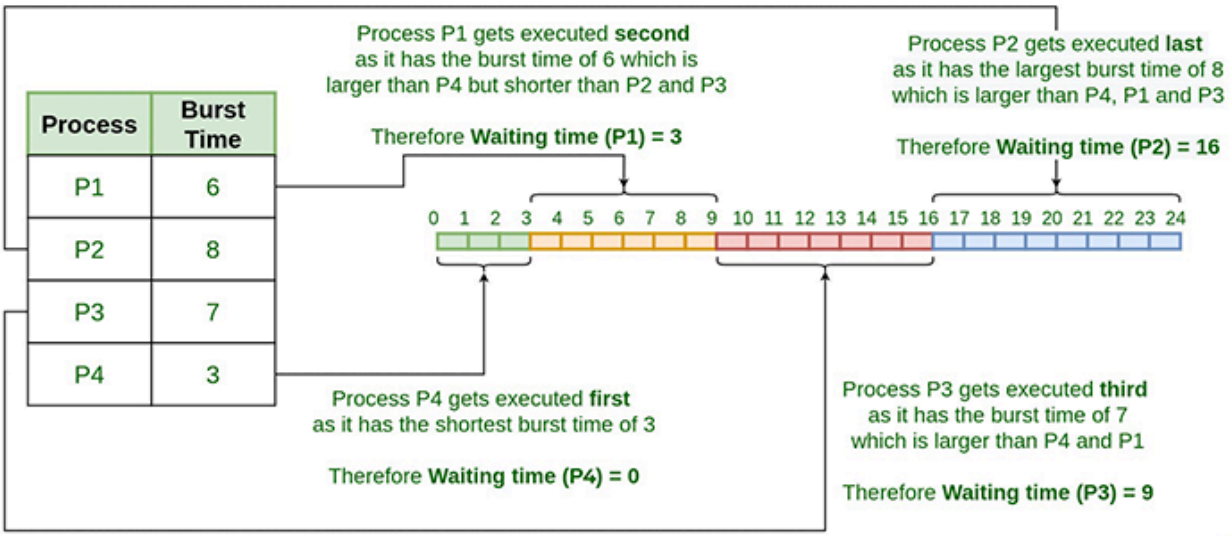
Characteristics of SJF Scheduling:

1. Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
2. It is a Greedy Algorithm.
3. It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of aging.

4. It is practically infeasible as the Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.
5. SJF can be used in specialized environments where accurate estimates of running time are available.

Working of SJF Scheduling Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.



Example:

Let's consider a set of processes with their burst times and arrival times:

Process	Arrival Time (AT)	Burst Time (BT)
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Step 1: Sort the processes based on arrival time.

At time 0, only P1 is available, so P1 will be executed first.

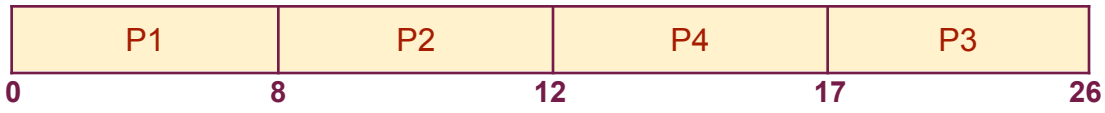
At time 1, P2 arrives, but since P1 is already running and SJF is non-preemptive, P1 continues until completion.

Step 2: Process Execution Timeline

1. **P1** starts at time 0 and runs for 8 units (from time 0 to 8).
2. **P2** arrives at time 1 but waits until P1 finishes. Once P1 finishes, P2 starts at time 8.
3. **P2** executes for 4 units (from time 8 to 12).
4. At time 2, P3 arrives and waits. At time 3, P4 arrives. Now, P4 has the shortest burst time among the remaining processes (P3 with 9 units and P4 with 5 units). Therefore, **P4** is selected next.
5. **P4** starts at time 12 and executes for 5 units (from time 12 to 17).
6. After P4 completes, **P3** is the only process left. It starts at time 17 and runs for 9 units (from time 17 to 26).

Step 3: Completion

- P1 finishes at time 8.
- P2 finishes at time 12.
- P4 finishes at time 17.
- P3 finishes at time 26.



Step 4: Calculation of Waiting Time and Turnaround Time

- **Waiting Time**: The waiting time of a process is the total time it spends waiting in the ready queue before execution starts.

```
Waiting Time = Turnaround Time - Burst Time
```

- **Turnaround Time**: The turnaround time of a process is the total time from its arrival until it finishes execution.

```
Turnaround Time = Completion Time - Arrival Time
```

For each process:

P1: Completion Time: 8
Waiting Time: 8 − 0 − 8 = 0
Turnaround Time: 8 − 0 = 8

P2: Completion Time: 12
Waiting Time: 12 − 1 − 4 = 7
Turnaround Time: 12 − 1 = 11

P3: Completion Time: 26
Waiting Time: 26 − 2 − 9 = 15
Turnaround Time: 26 − 2 = 24

P4: Completion Time: 17
Waiting Time: 17 − 3 − 5 = 9
Turnaround Time: 17 − 3 = 14

Process	AT	BT	CT	TAT	WT
P1	0	8	8	8	0
P2	1	4	12	11	7
P3	2	9	26	24	15
P4	3	5	17	14	9

Average Waiting Time: $\frac{0+7+15+9}{4} = 7.75\ units$

Average Turnaround Time: $\frac{8+9+22+16}{4} = 13.75\ units$

05 PROGRAM:

```c
#include<stdio.h>
// Structure to hold Process Details
typedef struct {
    int pid;        // Process ID
    int arrival;    // Arrival Time
    int burst;      // Burst Time
    int completion; // Completion Time
    int turnaround; // Turnaround Time
    int waiting;    // Waiting Time
} Process;

void SJFScheduling(Process p[], int n) {
    int time = 0, completed = 0;
    while (completed < n) {
        int minIndex = -1, minBurst = 9999;

        // Find the process with the shortest burst time that has arrived
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].completion == 0 && p[i].burst <
                minBurst) {
                minBurst = p[i].burst;
                minIndex = i;
            }
        }

        if (minIndex == -1) {
            // If no process is available to execute, increment time
            time++;
        } else {
            // Process is found, execute it
            time += p[minIndex].burst;
            p[minIndex].completion = time;
           p[minIndex].completion = time;
           p[minIndex].turnaround = p[minIndex].completion - p[minIndex].arrival;
           p[minIndex].waiting = p[minIndex].turnaround - p[minIndex].burst;
           completed++;
        }
    }
}
```

```c

void displayProcessTable(Process p[], int n) {
    printf("\nProcess     AT      BT      CT      TAT      WT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", p[i].pid, p[i].arrival,
            p[i].burst, p[i].completion, p[i].turnaround, p[i].waiting);
    }
}

int main() {
    int n;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    Process p[n];
    printf("Enter Arrival Time and Burst Time for each process: \n");
    for (int i = 0; i < n; i++) {
        p[i].pid = i + 1;
        printf("P%d Arrival Time: ", i + 1);
        scanf("%d", &p[i].arrival);
        printf("P%d Burst Time: ", i + 1);
        scanf("%d", &p[i].burst);
        p[i].completion = 0;  // Initializing completion time to 0
    }

    // Sorting processes by arrival time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].arrival > p[j + 1].arrival) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }

    // Execute SJFScheduling algorithm
    SJFScheduling(p, n);

    // Display the result table
    displayProcessTable(p, n);

    return 0;
}
```

06 OUTPUT

```
Enter number of processes: 4
Enter Arrival Time and Burst Time for each process:
P1 Arrival Time: 0
P1 Burst Time: 8
P2 Arrival Time: 1
P2 Burst Time: 4
P3 Arrival Time: 2
P3 Burst Time: 9
P4 Arrival Time: 3
P4 Burst Time: 5

Process     AT      BT      CT      TAT     WT
P1          0       8       8       8       0
P2          1       4       12      11      7
P3          2       9       26      24      15
P4          3       5       17      14      9
```

08 CONCLUSION:

This program implements the **Shortest Job First (SJF)** CPU scheduling algorithm, minimizing average waiting time by prioritizing shorter burst times. However, it can cause **starvation** for longer processes if shorter ones keep arriving. SJF is effective when burst times are known but less practical in real-world systems with unpredictable burst times.

09 VIVA QUESTIONS:

PREPARED BY **H. P. Amle**	**APPROVED BY** **HOD**