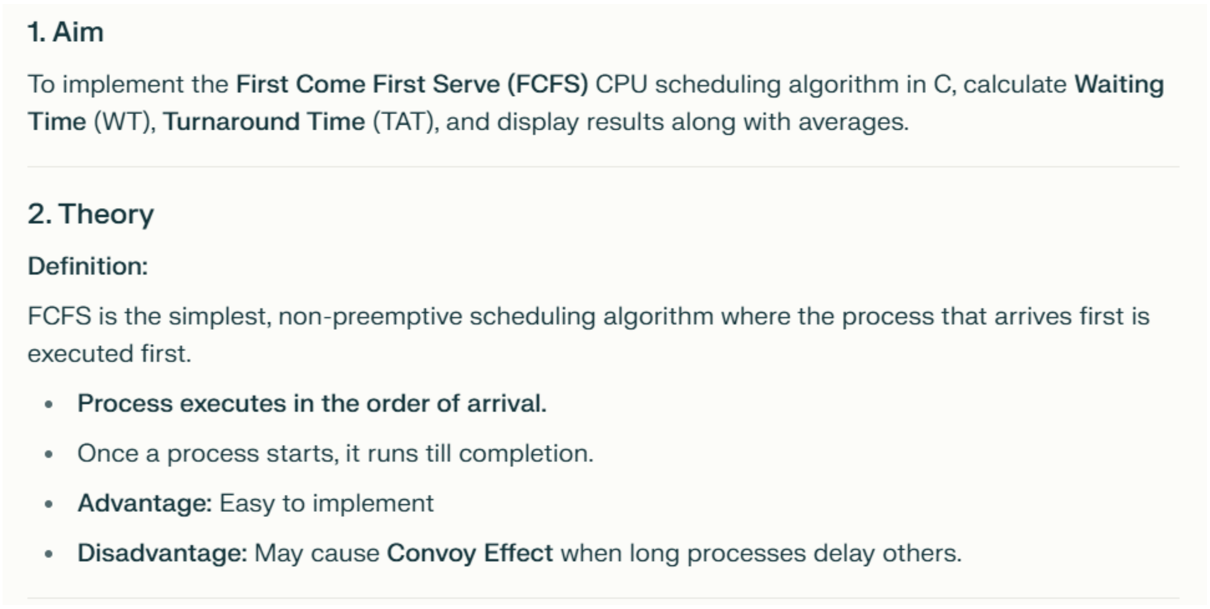
1.Write C programs to simulate the following CPU Scheduling algorithms

a) FCFS



### 1. Aim

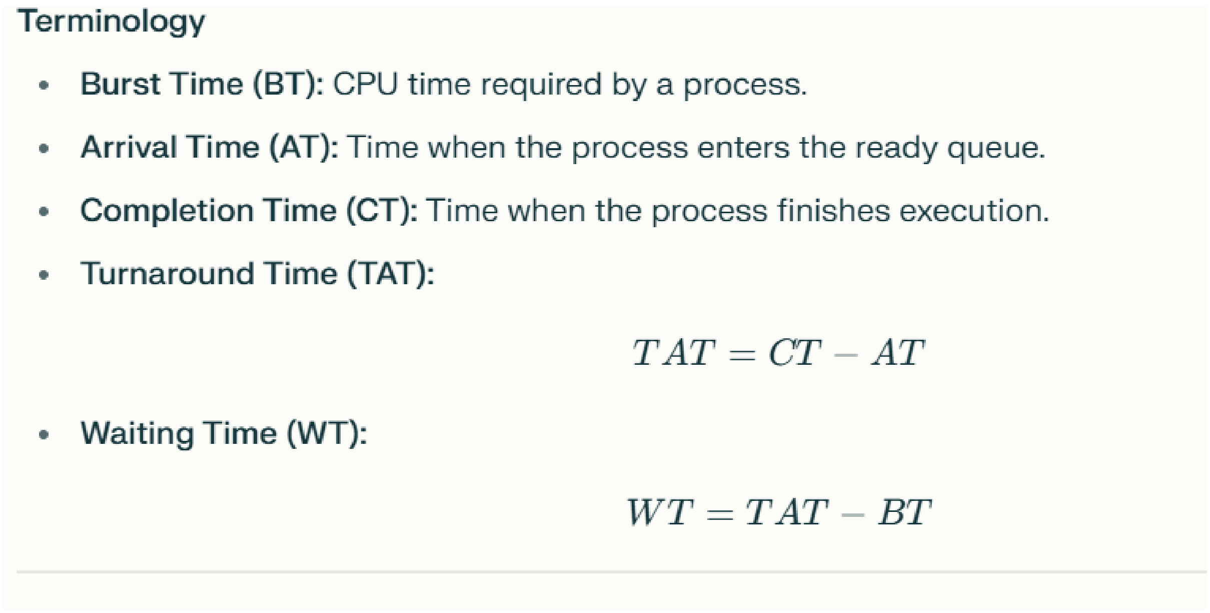
To implement the **First Come First Serve (FCFS)** CPU scheduling algorithm in C, calculate **Waiting Time** (WT), **Turnaround Time** (TAT), and display results along with averages.

### 2. Theory

**Definition:**

FCFS is the simplest, non-preemptive scheduling algorithm where the process that arrives first is executed first.

- **Process executes in the order of arrival.**
- Once a process starts, it runs till completion.
- **Advantage:** Easy to implement
- **Disadvantage:** May cause **Convoy Effect** when long processes delay others.



### Terminology

- **Burst Time (BT):** CPU time required by a process.
- **Arrival Time (AT):** Time when the process enters the ready queue.
- **Completion Time (CT):** Time when the process finishes execution.
- **Turnaround Time (TAT):**

$$TAT = CT - AT$$

- **Waiting Time (WT):**

$$WT = TAT - BT$$

## 3. Mathematical Model

For $n$ processes:

1. **Waiting Time:**
   - $WT_1 = 0$
   - $WT[i] = WT[i-1] + BT[i-1]$ for $i \geq 2$
2. **Turnaround Time:**
   - $TAT[i] = WT[i] + BT[i]$
3. **Averages:**

$$AvgWT = \frac{\sum WT}{n}$$

$$AvgTAT = \frac{\sum TAT}{n}$$

### 4. Example Calculation

Given:

| Process | BT | AT | WT | TAT |
|---|---|---|---|---|
| P1 | 5 | 0 | 0 | 5 |
| P2 | 3 | 0 | 5 | 8 |
| P3 | 8 | 0 | 8 | 16 |

- Avg WT = (0 + 5 + 8) / 3 = **4.33**
- Avg TAT = (5 + 8 + 16) / 3 = **9.67**

## 5. Algorithm

1. Input the number of processes and their **burst times**.
2. Set WT for first process = 0.
3. Calculate WT for remaining processes using cumulative BT.
4. Calculate TAT = WT + BT for each process.
5. Compute average WT and average TAT.
6. Display results.

6.C Program:

```
#include <stdio.h>

int main() {
    int n, i;
    int burst_time[20], waiting_time[20], turnaround_time[20];
    int total_wt = 0, total_tat = 0;

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

    printf("Enter burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
    }

    waiting_time[0] = 0; // First process waiting time = 0

    for(i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + burst_time[i - 1];
    }

    for(i = 0; i < n; i++) {
        turnaround_time[i] = waiting_time[i] + burst_time[i];
        total_wt += waiting_time[i];
        total_tat += turnaround_time[i];
    }

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]\t%d\t\t%d\t\t%d\n", i + 1, burst_time[i], waiting_time[i],
turnaround_time[i]);
    }

    printf("\nAverage Waiting Time = %.2f\n", (float)total_wt / n);
    printf("Average Turnaround Time = %.2f\n", (float)total_tat / n);

    return 0;
}
```

7.Output:

```
Enter number of processes: 3
Enter burst time for each process:
P[1]: 5
P[2]: 3
P[3]: 8
Process   Burst Time   Waiting Time   Turnaround Time
  P[1]         5              0               5
  P[2]         3              5               8
  P[3]         8              8              16
```

Average Waiting Time = 4.33

Average Turnaround Time = 9.67

8.Execution Flow:

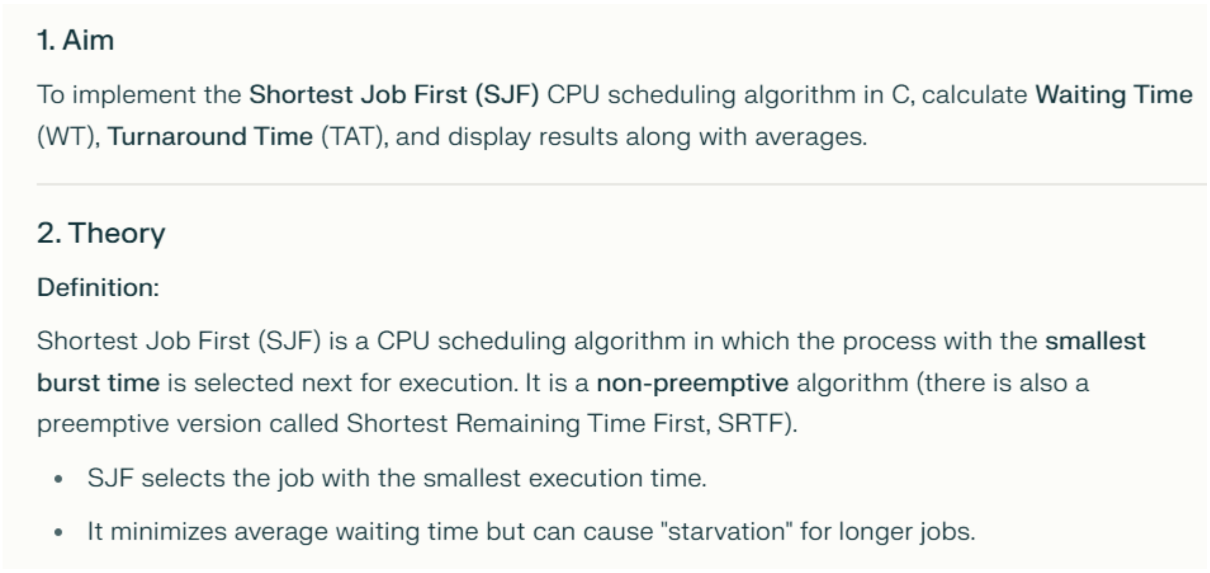
```
| P1 |----5----| P2 |----3----| P3 |--------8--------|
0     5             8                  16
```

9.Conclusion:

- FCFS executes processes in arrival order.
- It is simple but can cause delays if a large process arrives first (Convoy Effect).
- Best suited for batch processing systems.

1.Write C programs to simulate the following CPU Scheduling algorithms

B) SJF



### 1. Aim

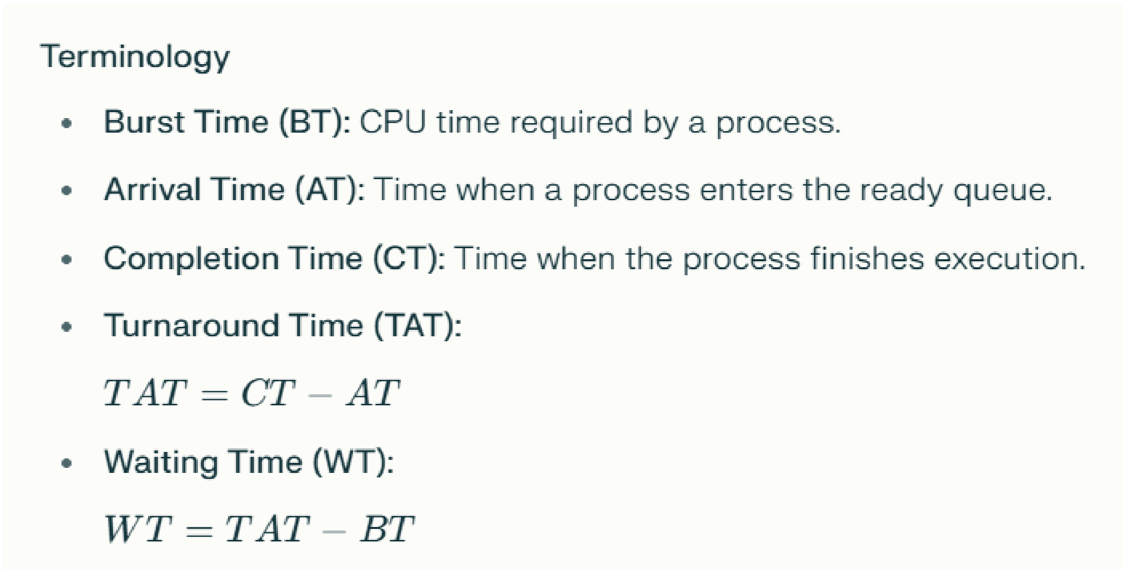
To implement the **Shortest Job First (SJF)** CPU scheduling algorithm in C, calculate **Waiting Time** (WT), **Turnaround Time** (TAT), and display results along with averages.

### 2. Theory

**Definition:**

Shortest Job First (SJF) is a CPU scheduling algorithm in which the process with the **smallest burst time** is selected next for execution. It is a **non-preemptive** algorithm (there is also a preemptive version called Shortest Remaining Time First, SRTF).

- SJF selects the job with the smallest execution time.
- It minimizes average waiting time but can cause "starvation" for longer jobs.



### Terminology

- **Burst Time (BT):** CPU time required by a process.
- **Arrival Time (AT):** Time when a process enters the ready queue.
- **Completion Time (CT):** Time when the process finishes execution.
- **Turnaround Time (TAT):**

  $$TAT = CT - AT$$

- **Waiting Time (WT):**

  $$WT = TAT - BT$$

## 3. Mathematical Model

For $n$ processes (all arrival times zero for simple SJF):

1. Sort processes by burst time (smallest to largest).
2. Waiting Time:
   - $WT_1 = 0$
   - $WT[i] = WT[i-1] + BT[i-1]$ for $i \geq 2$
3. Turnaround Time:
   - $TAT[i] = WT[i] + BT[i]$
4. Averages:

$$AvgWT = \frac{\sum WT}{n}$$
$$AvgTAT = \frac{\sum TAT}{n}$$

### 4. Example Calculation

Given (all AT=0):

| Process | BT |
|---|---|
| P1 | 6 |
| P2 | 8 |
| P3 | 7 |
| P4 | 3 |

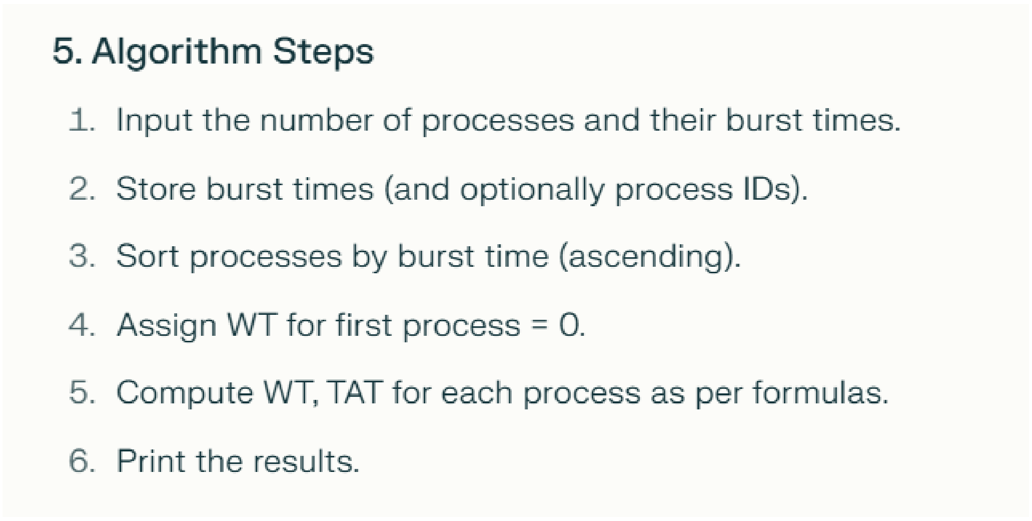
**Step 1: Sort by BT:**

P4 (3), P1 (6), P3 (7), P2 (8)

| Process | BT | WT | TAT |
|---|---|---|---|
| P4 | 3 | 0 | 3 |
| P1 | 6 | 3 | 9 |
| P3 | 7 | 9 | 16 |
| P2 | 8 | 16 | 24 |

- Avg WT = (0 + 3 + 9 + 16) / 4 = 7
- Avg TAT = (3 + 9 + 16 + 24) / 4 = 13

```
}
```



### 5. Algorithm Steps

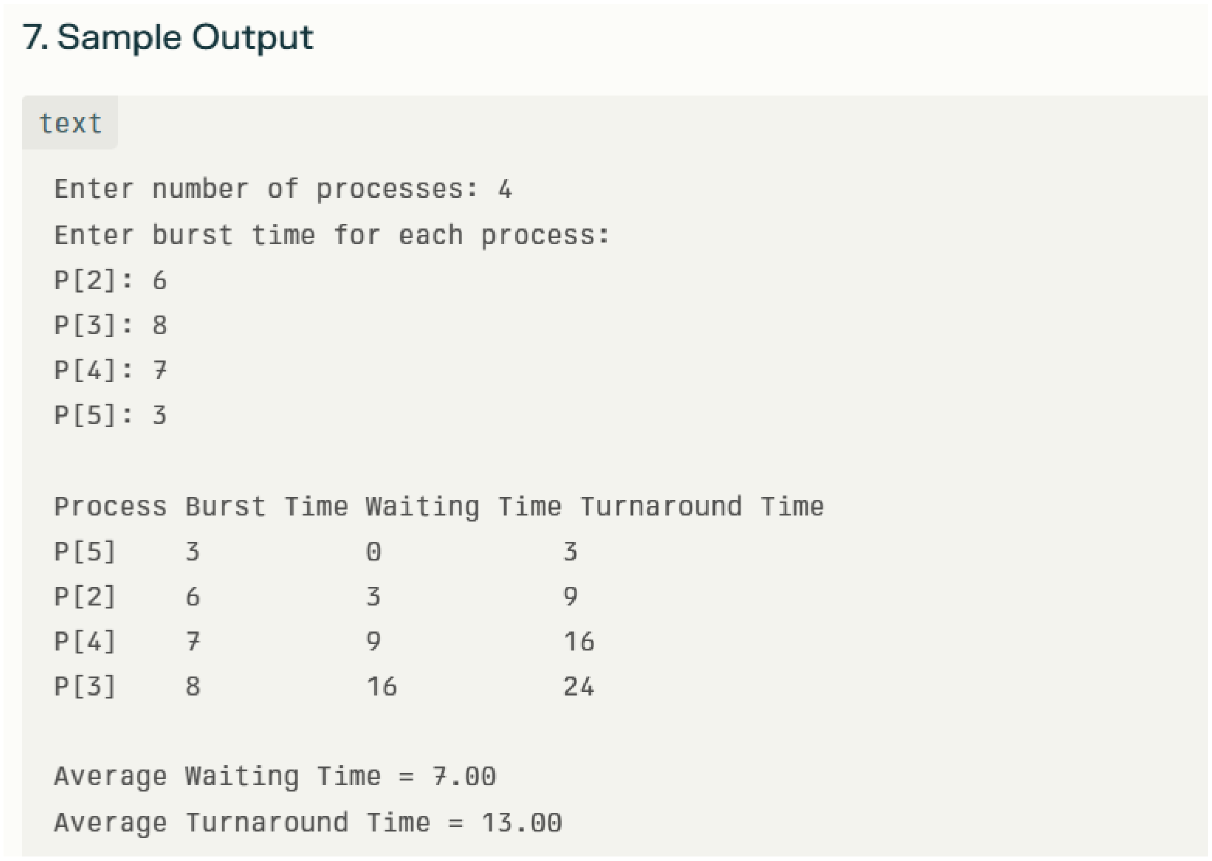
1. Input the number of processes and their burst times.
2. Store burst times (and optionally process IDs).
3. Sort processes by burst time (ascending).
4. Assign WT for first process = 0.
5. Compute WT, TAT for each process as per formulas.
6. Print the results.

**6. C Program**

```
#include <stdio.h>


int main() {
    int n, i, j, pos, temp;
    int burst_time[20], process[20], waiting_time[20], turnaround_time[20];
    int total_wt = 0, total_tat = 0;


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


    printf("Enter burst time for each process:\n");
    for(i = 0; i < n; i++) {
        printf("P[%d]: ", i + 1);
        scanf("%d", &burst_time[i]);
        process[i] = i + 1; // Store process numbers
    }
```

```
// Sort burst times with process number using selection sort
for(i = 0; i < n; i++) {
    pos = i;
    for(j = i+1; j < n; j++) {
        if(burst_time[j] < burst_time[pos])
            pos = j;
    }
    temp = burst_time[i];
    burst_time[i] = burst_time[pos];
    burst_time[pos] = temp;

    temp = process[i];
    process[i] = process[pos];
    process[pos] = temp;
}

// First process has 0 waiting time
waiting_time[0] = 0;

// Calculate waiting time
for(i = 1; i < n; i++) {
    waiting_time[i] = 0;
    for(j = 0; j < i; j++)
        waiting_time[i] += burst_time[j];
    total_wt += waiting_time[i];
}
```

```
    // Calculate turnaround time
    for(i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        total_tat += turnaround_time[i];
    }

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for(i = 0; i < n; i++)
        printf("P[%d]\t%d\t\t%d\t\t%d\n", process[i], burst_time[i],
waiting_time[i], turnaround_time[i]);

    printf("\nAverage Waiting Time = %.2f", (float)total_wt / n);
    printf("\nAverage Turnaround Time = %.2f\n", (float)total_tat / n);

    return 0;
}
```



## 7. Sample Output

```
Enter number of processes: 4
Enter burst time for each process:
P[2]: 6
P[3]: 8
P[4]: 7
P[5]: 3

Process Burst Time Waiting Time Turnaround Time
P[5]    3          0            3
P[2]    6          3            9
P[4]    7          9            16
P[3]    8          16           24

Average Waiting Time = 7.00
Average Turnaround Time = 13.00
```

## 8. Execution Flow

**| P4 |---3---| P1 |---6---| P3 |---7---| P2 |---8---|**

**0    3      9        16       24**

## 9. Conclusion

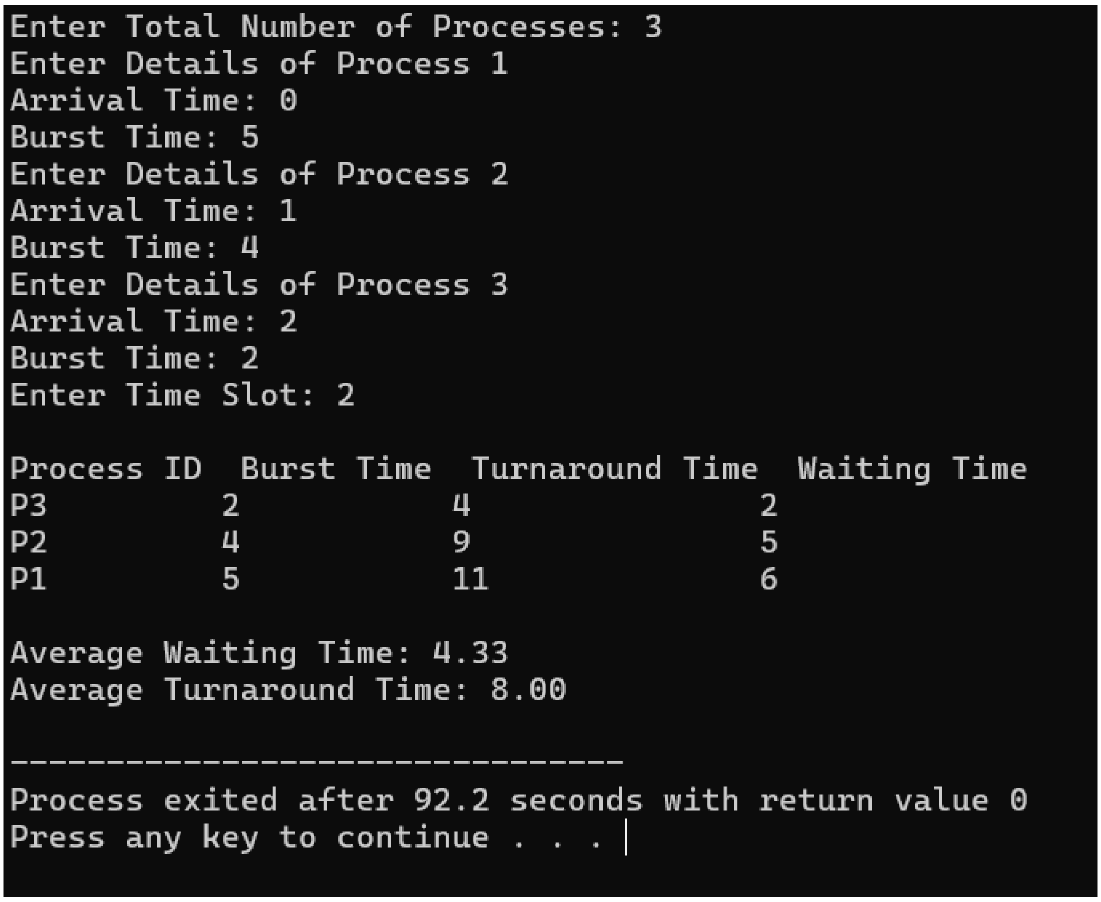
- **SJF minimizes average waiting time compared to FCFS.**
- **Risk of starvation for long jobs if short jobs continuously arrive.**
- **Useful for batch systems where job lengths are known in advance.**

1.Write C programs to simulate the following CPU Scheduling algorithms

C) Round Robin

```
#include <stdio.h>

int main() {
    int n;
    printf("Enter Total Number of Processes: ");
    scanf("%d", &n);

    int wait_time = 0, ta_time = 0;
    int arr_time[n], burst_time[n], temp_burst_time[n];
    int x = n;  // number of remaining processes

    for (int i = 0; i < n; i++) {
        printf("Enter Details of Process %d\n", i + 1);
        printf("Arrival Time: ");
        scanf("%d", &arr_time[i]);
        printf("Burst Time: ");
        scanf("%d", &burst_time[i]);
        temp_burst_time[i] = burst_time[i];
    }

    int time_slot;
    printf("Enter Time Slot: ");
    scanf("%d", &time_slot);
    int total = 0, counter = 0, i = 0;
    printf("\nProcess ID  Burst Time  Turnaround Time  Waiting Time\n");

    while (x != 0) {
        if (temp_burst_time[i] <= time_slot && temp_burst_time[i] > 0) {
            total += temp_burst_time[i];
            temp_burst_time[i] = 0;
            counter = 1;
        } else if (temp_burst_time[i] > 0) {
            temp_burst_time[i] -= time_slot;
            total += time_slot;
        }

        if (temp_burst_time[i] == 0 && counter == 1) {
            x--;
            printf("P%-9d %-11d %-15d %-10d\n",
                   i + 1, burst_time[i], total - arr_time[i], total - arr_time[i] -
burst_time[i]);
            wait_time += total - arr_time[i] - burst_time[i];
            ta_time += total - arr_time[i];
            counter = 0;
        }

        if (i == n - 1) {
            i = 0;
        } else if (arr_time[i + 1] <= total) {
            i++;
        } else {
            i = 0;
        }
    }

    float average_wait_time = (float)wait_time / n;
    float average_turnaround_time = (float)ta_time / n;

    printf("\nAverage Waiting Time: %.2f", average_wait_time);
    printf("\nAverage Turnaround Time: %.2f\n", average_turnaround_time);

    return 0;
}
```

Output:



```
Enter Total Number of Processes: 3
Enter Details of Process 1
Arrival Time: 0
Burst Time: 5
Enter Details of Process 2
Arrival Time: 1
Burst Time: 4
Enter Details of Process 3
Arrival Time: 2
Burst Time: 2
Enter Time Slot: 2

Process ID  Burst Time  Turnaround Time  Waiting Time
P3         2           4                2
P2         4           9                5
P1         5           11               6

Average Waiting Time: 4.33
Average Turnaround Time: 8.00

--------------------------------
Process exited after 92.2 seconds with return value 0
Press any key to continue . . .
```

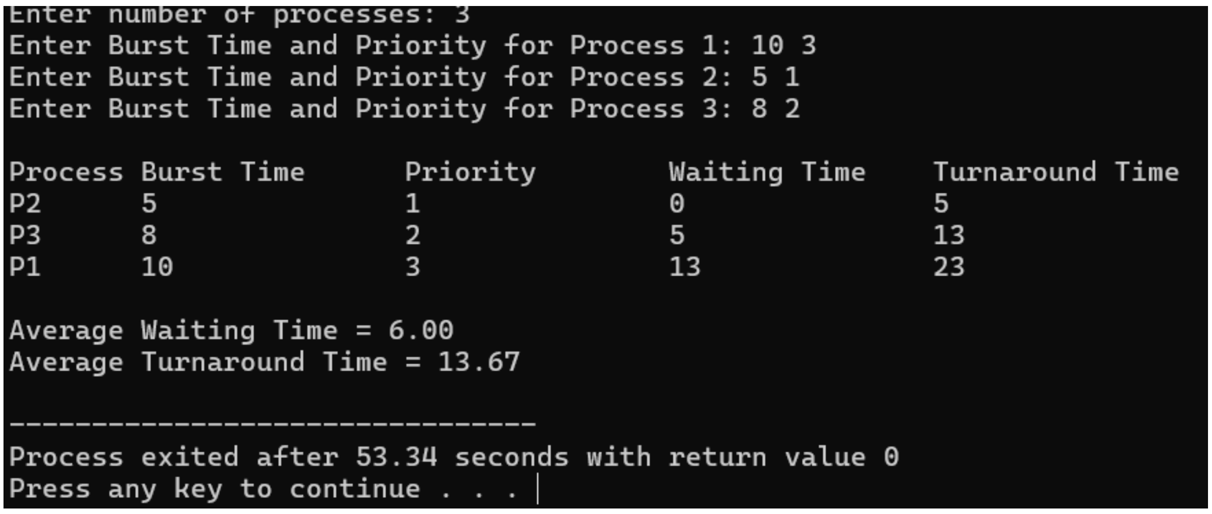
1.Write C programs to simulate the following CPU Scheduling algorithms

D) Priority

```
#include <stdio.h>

int main() {
    int n, i, j, pos, temp;
    int burst_time[20], p[20], wait_time[20], tat[20], priority[20];
    float total_wait = 0, total_tat = 0;

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

    for(i = 0; i < n; i++) {
        printf("Enter Burst Time and Priority for Process %d: ", i + 1);
        scanf("%d %d", &burst_time[i], &priority[i]);
        p[i] = i + 1;  // process ID
    }

    // Sort processes by priority (lower number = higher priority)
    for(i = 0; i < n; i++) {
        pos = i;
        for(j = i + 1; j < n; j++) {
            if(priority[j] < priority[pos])
                pos = j;
        }
        // Swap priority
    temp = priority[i];
    priority[i] = priority[pos];
    priority[pos] = temp;

    // Swap burst time
    temp = burst_time[i];
    burst_time[i] = burst_time[pos];
    burst_time[pos] = temp;

    // Swap process ID
    temp = p[i];
    p[i] = p[pos];
    p[pos] = temp;
}

// Calculate waiting times
wait_time[0] = 0;
for(i = 1; i < n; i++) {
    wait_time[i] = 0;
    for(j = 0; j < i; j++)
        wait_time[i] += burst_time[j];
    total_wait += wait_time[i];
}

// Calculate turnaround time and print table
printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
```

```
    for(i = 0; i < n; i++) {
        tat[i] = burst_time[i] + wait_time[i];
        total_tat += tat[i];
        printf("P%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i], burst_time[i], priority[i],
wait_time[i], tat[i]);
    }

    printf("\nAverage Waiting Time = %.2f", total_wait / n);
    printf("\nAverage Turnaround Time = %.2f\n", total_tat / n);

    return 0;
}
```

Output:



```
Enter number of processes: 3
Enter Burst Time and Priority for Process 1: 10 3
Enter Burst Time and Priority for Process 2: 5 1
Enter Burst Time and Priority for Process 3: 8 2

Process Burst Time      Priority        Waiting Time    Turnaround Time
P2      5               1               0               5
P3      8               2               5               13
P1      10              3               13              23

Average Waiting Time = 6.00
Average Turnaround Time = 13.67

--------------------------------
Process exited after 53.34 seconds with return value 0
Press any key to continue . . .
```

2. Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

Aim: To perform I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

/* Open GDB Online Debugger to Perform below Linux calls.--> Login First if required then perform all actions*/

/* 1.create a sample text file on pc desktop with name test.txt—usually notepad file and write some content called

Hello GDB Online Debugger.How are you.I am your friend */

/* Upload that file into GDB Online Debugger and execute the below program*/

/* On left side pane→Click →Create New Project→Write the prog→Execute*/

Program:

1. open(), read(), write(), close()

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    char buffer[100];

    // Open file in read-only mode
    fd = open("test.txt", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Read contents
    int n = read(fd, buffer, sizeof(buffer));
    if (n < 0) {
        perror("read");
        return 1;
    }
    buffer[n] = '\0';

    // Write contents to STDOUT
    write(1, buffer, n);

    // Close file
    close(fd);
    return 0;
}
```

Output:

**2. fcntl() – File control**

Program:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd = open("test.txt", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    // Get file descriptor flags
    int flags = fcntl(fd, F_GETFL);
    printf("File access mode: %d\n", flags & O_ACCMODE);

    close(fd);
    return 0;
}
```

Output: