Week - 4

CPU Scheduling

Definition : CPU Scheduling is the process of selecting one process from the ready queue and allocating the CPU to it for execution. It is a core function of the operating system's process scheduler, aiming to maximize CPU utilization, improve system throughput, and ensure fair and efficient execution of processes.

Purpose : To decide which process should be executed next by the CPU to improve system performance.

Goals : Maximize CPU utilization, throughput, and fairness while minimizing waiting time, turnaround time, and response time.

Types -

Preemptive: CPU can be taken away from a process before it finishes (e.g., SRTF, RR).

Non-preemptive: Once a process starts, it runs until completion (e.g., FCFS, non-preemptive SJF).

Scheduling Criteria

- CPU Utilization
- Throughput (processes completed per unit time)
- Turnaround Time (TAT)
- Waiting Time (WT)
- Response Time (RT)

Impact of Arrival Time – When arrival times differ, idle CPU periods may occur, affecting scheduling decisions.

Starvation – Some processes may never get CPU time if shorter jobs keep coming (possible in SJF).

Context Switching – In preemptive algorithms, switching between processes can cause overhead.

CPU Scheduling Algorithms

1. First Come, First Served (FCFS)

- **Type**: Non-preemptive.

- **How it works**: Processes are executed in the order they arrive.
- **Pros**: Simple to implement.
- **Cons**: Can cause the **Convoy Effect** (long process delays others).

2. Shortest Job First (SJF)

- **Type**: Can be **preemptive** (SRTF) or **non-preemptive**.
- **How it works**: Selects the process with the smallest burst time first.
- **Pros**: Minimizes average waiting time.
- **Cons**: Requires knowledge of burst time in advance, and **starvation** may occur for long processes.

3. Round Robin (RR)

- **Type**: Preemptive.
- **How it works**: Each process gets a fixed time quantum in cyclic order.
- **Pros**: Fair for all processes; good for time-sharing systems.
- **Cons**: Performance depends heavily on time quantum size.

Aim : To write a C program to implement First Come First Serve (FCFS) CPU scheduling algorithm, calculate Completion Time (CT), Turn Around Time (TAT), Waiting Time (WT), and display the average TAT and WT for all processes.

Algorithm :

Step 1: Start

Step 2: Input the total number of processes n.

Step 3: For each process i from 1 to n:
a. Input **Arrival Time (AT[i])**
b. Input **Burst Time (BT[i])**

Step 4: Sort all processes based on **Arrival Time (AT)** in ascending order.

Step 5: Initialize:
current_time = 0
total_tat = 0
total_wt = 0

Step 6: For each process i in sorted order:
a. If current_time < AT[i], set current_time = AT[i] (**handle CPU idle time**).
b. Compute **Completion Time (CT[i])**:
CT[i] = current_time + BT[i]
c. Update current_time = CT[i].
d. Compute **Turn Around Time (TAT[i])**:
TAT[i] = CT[i] - AT[i]
e. Compute **Waiting Time (WT[i])**:
WT[i] = TAT[i] - BT[i]
f. Add TAT[i] to total_tat and WT[i] to total_wt.

Step 7: Display the process table showing:
Process ID, AT, BT, CT, TAT, WT.

Step 8: Calculate:
Average TAT = total_tat / n
Average WT = total_wt / n

Step 9: Display Average TAT and Average WT.

Step 10: Stop

Code :

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

```
struct Process {
    int pid; // Process ID
    int at;  // Arrival Time
    int bt;  // Burst Time
    int ct;  // Completion Time
    int tat; // Turn Around Time
    int wt;  // Waiting Time
};
int main() {
    int n;
    printf("Enter total number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    // Input process details
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("\nEnter Arrival Time and Burst Time of process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].at, &proc[i].bt);
    }
    // Sort processes by Arrival Time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (proc[j].at > proc[j + 1].at) {
                struct Process temp = proc[j];
                proc[j] = proc[j + 1];
                proc[j + 1] = temp;
            }
        }
    }
    int current_time = 0;
    int total_tat = 0, total_wt = 0;
    // Calculate times
    for (int i = 0; i < n; i++) {
        if (current_time < proc[i].at) {
            current_time = proc[i].at; // CPU idle till process arrives
        }
        proc[i].ct = current_time + proc[i].bt;
        current_time = proc[i].ct;
        proc[i].tat = proc[i].ct - proc[i].at;
        proc[i].wt = proc[i].tat - proc[i].bt;
        total_tat += proc[i].tat;
        total_wt += proc[i].wt;
    }
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT");
    for (int i = 0; i < n; i++) {
        printf("\nP%d\t%d\t%d\t%d\t%d\t%d",
               proc[i].pid, proc[i].at, proc[i].bt,
               proc[i].ct, proc[i].tat, proc[i].wt);
```

```
    }
    // Display averages
    printf("\nAverage Turn Around Time: %.2f", (float)total_tat / n);
    printf("\nAverage Waiting Time: %.2f\n", (float)total_wt / n);
    return 0;
}
```

Output :

```
musaib@LAPTOP-61BAEEL0:~$ nano fcFs.c
musaib@LAPTOP-61BAEEL0:~$ gcc fcFs.c
musaib@LAPTOP-61BAEEL0:~$ ./a.out
Enter total number of processes: 5

Enter Arrival Time and Burst Time of process 1:  1 2

Enter Arrival Time and Burst Time of process 2: 2 3

Enter Arrival Time and Burst Time of process 3: 3 4

Enter Arrival Time and Burst Time of process 4: 4 5

Enter Arrival Time and Burst Time of process 5: 5 6

Process AT      BT      CT      TAT     WT
P1      1       2       3       2       0
P2      2       3       6       4       1
P3      3       4       10      7       3
P4      4       5       15      11      6
P5      5       6       21      16      10
```


Aim : To implement **Shortest Job First (SJF) Non-preemptive Scheduling** in C, calculate the **Completion Time (CT)**, **Turnaround Time (TAT)**, and **Waiting Time (WT)** for each process, and find the **average TAT** and **average WT**.

Algorithm :

Step 1: Start.

Step 2: Input the total number of processes n.

Step 3: For each process i (from 0 to n-1):

a. Assign process ID pid = i+1.

b. Input arrival time at and burst time bt.
c. Mark process as not done (done = false).

Step 4: Initialize:
- current_time = 0
- total_tat = 0
- total_wt = 0
- completed = 0

Step 5: Repeat **while** completed < n:
a. Set idx = -1 (to track selected process).
b. Set min_bt = 99999 (large value to find smallest burst time).

Step 6: For each process i from 0 to n-1:
a. If process is not done **and its arrival time ≤** current_time:
i. If its burst time < min_bt:
- Update idx = i
- Update min_bt = proc[i].bt
ii. Else if burst time equals min_bt and arrival time is smaller than the currently selected process:
- Update idx = i

Step 7: If idx != -1 (a process is found to execute):
a. Compute completion time: ct = current_time + bt
b. Update current_time = ct
c. Calculate turnaround time: tat = ct - at
d. Add tat to total_tat
e. Calculate waiting time: wt = tat - bt
f. Add wt to total_wt
g. Mark process as done
h. Increment completed

Step 8: Else (no process is ready yet):
Increment current_time by 1.

Step 9: After the loop, display process table with AT, BT, CT, TAT, WT.

Step 10: Calculate and display:
- Average TAT = total_tat / n
- Average WT = total_wt / n

Step 11: End.

Code :

```
#include <stdio.h>
#include <stdbool.h>

struct Process {
    int pid, at, bt, ct, tat, wt;
    bool done;
};

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

    struct Process proc[n];

    // Input process details
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("\nEnter Arrival time and Burst time of process %d: ", proc[i].pid);
        scanf("%d %d", &proc[i].at, &proc[i].bt);
        proc[i].done = false;
    }

    int current_time = 0, completed = 0;
    int total_tat = 0, total_wt = 0;

    while (completed < n) {
        int idx = -1, min_bt = 99999;

        // Select process with shortest burst time among arrived processes
        for (int i = 0; i < n; i++) {
            if (!proc[i].done && proc[i].at <= current_time) {
                if (proc[i].bt < min_bt) {
                    idx = i;
                    min_bt = proc[i].bt;
                } else if (proc[i].bt == min_bt) {
                    if (proc[i].at < proc[idx].at)
                        idx = i;
                }
            }
        }

        if (idx != -1) {
            proc[idx].ct = current_time + proc[idx].bt;
            current_time = proc[idx].ct;
            proc[idx].tat = proc[idx].ct - proc[idx].at;
            proc[idx].wt = proc[idx].tat - proc[idx].bt;
```

```
            total_tat += proc[idx].tat;
            total_wt += proc[idx].wt;
            proc[idx].done = true;
            completed++;
        } else {
            current_time++; // CPU idle
        }
    }

    // Output results
    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT");
    for (int i = 0; i < n; i++) {
        printf("\nP%d\t%d\t%d\t%d\t%d\t%d",
               proc[i].pid, proc[i].at, proc[i].bt,
               proc[i].ct, proc[i].tat, proc[i].wt);
    }

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

    return 0;
}
```

Output :

```
musaib@LAPTOP-61BAEEL0:~$ nano SJF.c
musaib@LAPTOP-61BAEEL0:~$
musaib@LAPTOP-61BAEEL0:~$ gcc SJF.c
musaib@LAPTOP-61BAEEL0:~$ ./a.out
Enter total number of processes: 4

Enter Arrival Time and Burst Time of process 1: 2 3

Enter Arrival Time and Burst Time of process 2: 2 5

Enter Arrival Time and Burst Time of process 3: 5 6

Enter Arrival Time and Burst Time of process 4: 6 7

Process AT      BT      CT      TAT     WT
P1      2       3       5       3       0
P2      2       5       10      8       3
P3      5       6       16      11      5
P4      6       7       23      17      10
Average Turn Around Time: 9.75
Average Waiting Time: 4.50
```