#### CPU Scheduling Algorithms

# Aim:

Write a menu driven program for implementing the following CPU scheduling algorithms.

(a) **FCFS** (b) **Round Robin** (c) **SJF** (d) **Priority**

For all the programs, arrival time for the processes should be considered. For

each case find the **turnaround time**, **waiting time** and display a **Gantt chart.**

Also find the **average waiting time** and **average turn around time**.

# Description:

**a) First Come First Serve (FCFS) Scheduling**

In FCFS, the process that arrives first is executed first, and so on. It is a non-preemptive scheduling algorithm, meaning once a process starts execution, it continues until it completes or gets blocked.

Working of FCFS CPU Scheduling:

When a process enters the ready queue, its arrival time is noted.

The CPU scheduler selects the process with the earliest arrival time from the ready queue for execution.

The selected process starts executing on the CPU.

If another process arrives while the CPU is busy, it joins the ready queue, but it will only execute after the current process completes.

The selected process continues to execute until it finishes its burst time or gets blocked by an I/O request.

Once a process completes its execution or gets blocked, the CPU scheduler selects the next process based on FCFS to execute.

This process repeats until all processes in the queue have completed their execution.

**Algorithm:**

1. Start

2. Sort the processes based on their arrival times.

3. Initialize `current\_time` to 0.

4. For each process in the sorted list:

- Calculate the waiting time as `current\_time - arrival\_time`.

- Update `current\_time` by adding the burst time of the current process.

5. Calculate the waiting time and turnaround time for each process.

6. Display the Gantt chart and the process table.

7. End

**b) Round Robin Scheduling:**

In Round Robin scheduling each process is executed for a small unit of time called a time quantum or time slice.

Working of Round Robin Scheduling:

The processes are placed in a circular queue called the ready queue.

Each process is assigned a fixed time quantum, typically ranging from 10 to 100 milliseconds.

The CPU scheduler selects the process at the front of the ready queue to execute.

The selected process runs for its time quantum.

If the process completes its burst time within the time quantum, it is removed from the queue.

If the time quantum elapses before the process completes, it is preempted and moved to the end of the ready queue.

The CPU scheduler then selects the next process in the queue for execution.

This process repeats until all processes have completed their execution.

**Algorithm:**

1. Start

2. Initialize `remaining\_processes` to the total number of processes.

3. Initialize `current\_time` to 0.

4. Initialize `completed` array to track completed processes.

5. While there are remaining processes:

- Find the next process that has arrived by `current\_time` and is not done.

- If the process's burst time is less than or equal to the time quantum:

- Update `current\_time` by adding the burst time.

- Set process as done and decrement `remaining\_processes`.

- Else:

- Execute the process for the time quantum.

- Update `current\_time` by adding the time quantum.

- Reduce the process's burst time by the time quantum.

6. Calculate the waiting time and turnaround time for each process.

7. Display the Gantt chart and the process table.

8. End

**c) Shortest Job First(SJF) Scheduling:**

Shortest Job First (SJF) scheduling is a CPU scheduling algorithm that selects the process with the shortest burst time to execute first. It can be both preemptive and non-preemptive, where preemptive SJF is also known as Shortest Remaining Time First (SRTF).

Working of SJF Scheduling:

The CPU scheduler maintains a ready queue containing all the processes waiting to be executed.

When a process enters the ready queue, its burst time is noted.

The CPU scheduler selects the process with the shortest burst time from the ready queue for execution.

If multiple processes have the same shortest burst time, SJF can use either FCFS or priority scheduling to break ties.

The selected process starts executing on the CPU.

If a new process arrives with a shorter burst time than the currently executing process in preemptive SJF, the CPU scheduler preemptively switches to the new process.

If the currently executing process completes its burst time or gets blocked, the CPU scheduler selects the next process with the shortest burst time from the ready queue.

This process repeats until all processes have completed their execution.

**Algorithm:**

1. Start

2. Initialize `time` to 0.

3. While there are processes to be scheduled:

- Find the process with the shortest burst time that has arrived by `time` and is not done.

- Set the process as done.

- Update `time` by adding the burst time of the selected process.

4. Calculate the waiting time and turnaround time for each process.

5. Display the Gantt chart and the process table.

6. End

**d) Priority Scheduling:**

Priority Scheduling is a CPU scheduling algorithm where each process is assigned a priority. The process with the highest priority is selected for execution first. Processes with the same priority are scheduled based on another scheduling algorithm like FCFS.

Working of Priority Scheduling:

Each process is assigned a priority value, which can be based on factors such as the process's importance, deadline, or other criteria.

The CPU scheduler maintains a ready queue containing all the processes waiting to be executed.

When a process enters the ready queue, its priority is noted.

The CPU scheduler selects the process with the highest priority from the ready queue for execution.

If multiple processes have the same highest priority, Priority Scheduling may use another scheduling algorithm (like FCFS or Round Robin) to break ties.

The selected process starts executing on the CPU.

If a new process arrives with a higher priority than the currently executing process, the CPU scheduler preemptively switches to the new process.

If the currently executing process completes its execution or gets blocked, the CPU scheduler selects the next process with the highest priority from the ready queue.

This process repeats until all processes have completed their execution.

**Algorithm:**

1. Start

2. Initialize `time` to 0.

3. While there are processes to be scheduled:

- Find the highest priority process that has arrived by `time` and is not done.

- Set the process as done.

- Update `time` by adding the burst time of the selected process.

4. Calculate the waiting time and turnaround time for each process.

5. Display the Gantt chart and the process table.

6. End

**Program**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 10

typedef struct Process

{

    int id;

    int arrival\_time;

    int burst\_time;

    int priority;

    int remaining\_time;

    int turnaround\_time;

    int waiting\_time;

    int done;

} Process;

// Function prototypes

void FCFS(Process processes[], int n);

void roundRobin(Process processes[], int n, int time\_quantum);

void SJF(Process processes[], int n);

void priority(Process processes[], int n);

void calculateTurnaroundTime(Process processes[], int n);

void calculateWaitingTime(Process processes[], int n);

void displayGanttChart(Process processes[], int n);

void table(Process p[], int n);

int main()

{

    int n=5, choice, time\_quantum;

    Process processes[MAX\_PROCESSES];

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

    scanf("%d", &n);

    if (n > MAX\_PROCESSES || n <= 0)

    {

        printf("Invalid number of processes.\n");

        return 1;

    }

    // Input process details

    for (int i = 0; i < n; i++)

    {

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

        scanf("%d", &processes[i].arrival\_time);

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

        scanf("%d", &processes[i].burst\_time);

        printf("Enter priority for process %d: ", i + 1);

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

        processes[i].id = i + 1;

        processes[i].remaining\_time = processes[i].burst\_time;

        processes[i].turnaround\_time=0;

        processes[i].waiting\_time=0;

        printf("\n\n");

    }

    printf("Select the scheduling algorithm:\n");

    printf("1. FCFS (First Come First Serve)\n");

    printf("2. Round Robin\n");

    printf("3. SJF (Shortest Job First)\n");

    printf("4. Priority\n");

    printf("5. Exit\n");

    while (1)

    {

        printf("\n\nEnter your choice: ");

        scanf("%d", &choice);

        switch (choice)

        {

        case 1:

            FCFS(processes, n);

            break;

        case 2:

            printf("Enter time quantum for Round Robin: ");

            scanf("%d", &time\_quantum);

            roundRobin(processes, n, time\_quantum);

            break;

        case 3:

            SJF(processes, n);

            break;

        case 4:

            priority(processes, n);

            break;

        case 5:

            exit(0);

            break;

        default:

            printf("Invalid choice.\n");

            return 1;

        }

    }

    return 0;

}

void FCFS(Process processes[], int n)

{

    // Sort processes based on arrival time

    int current\_time = 0;

    for(int i=0;i<n-1;i++){

        for(int j=0;j<n-i-1;j++){

            if(processes[j].arrival\_time>processes[j+1].arrival\_time){

                Process p=processes[j];

                processes[j]=processes[j+1];

                processes[j+1]=p;

            }

        }

    }

    for(int i=0;i<n;i++){

        processes[i].waiting\_time=current\_time-processes[i].arrival\_time;

        current\_time+=processes[i].burst\_time;

    }

    calculateWaitingTime(processes, n);

    calculateTurnaroundTime(processes, n);

    displayGanttChart(processes, n);

    table(processes,n);

}

void roundRobin(Process processes[], int n, int qt)

{

    int remaining\_processes = n;

    int current\_time = 0;

    int completed[MAX\_PROCESSES] = {0}; // Array to track completed processes

    Process p[100],q[10];

    for(int i=0;i<n;i++){

        q[i]=processes[i];

    }

    int num=0;

    while (remaining\_processes > 0)

    {

        int index=-1;

        for(int i=0;i<n;i++){//to find earliest process

            if(processes[i].arrival\_time<=current\_time && processes[i].done==0){

                if(index==-1) index=i;

                else if(processes[i].arrival\_time<processes[index].arrival\_time)

                    index=i;

            }

        }

        if(processes[index].burst\_time<=qt){

            remaining\_processes--;

            p[num].arrival\_time=current\_time;

            p[num].burst\_time=processes[index].burst\_time;

            p[num].turnaround\_time=p[num].burst\_time;

            p[num].id=index;

            processes[index].burst\_time=0;

            current\_time+=p[num].burst\_time;

            processes[index]=q[index];

            processes[index].turnaround\_time=current\_time-processes[index].arrival\_time;

            processes[index].waiting\_time=processes[index].turnaround\_time-processes[index].burst\_time;

            processes[index].done=1;

            num++;

        }

        else{

            int temp=current\_time+qt;

            int count=0;

            for(int i=0;i<n;i++){

                if(i!=index && processes[i].arrival\_time<=temp && processes[i].done==0){

                    if(processes[i].burst\_time<=qt)

                        count+=processes[i].burst\_time;

                    else

                        count+=qt;

                }

            }

            p[num].arrival\_time=current\_time;

            p[num].burst\_time=qt;

            p[num].turnaround\_time=qt;

            p[num].id=index;

            num++;

            processes[index].arrival\_time+=count;

            processes[index].burst\_time-=qt;

            current\_time+=qt;

        }

    }

    displayGanttChart(p, num);

    table(processes,n);

    calculateWaitingTime(processes, n);

    calculateTurnaroundTime(processes, n);

}

void SJF(Process processes[], int n)

{

    Process temp[10];

    int time=0;

    for(int i=0;i<n;i++){

        int index=-1;

        for(int j=0;j<n;j++){

            if(processes[j].arrival\_time<=time && processes[j].done==0){

                if(index==-1) index=j;

                else if(processes[j].burst\_time<processes[index].burst\_time)

                    index=j;

            }

        }

        processes[index].done=1;

        time+=processes[index].burst\_time;

        temp[i]=processes[index];

        temp[i].waiting\_time=time-processes[index].burst\_time-processes[index].arrival\_time;

    }

    calculateWaitingTime(temp, n);

    calculateTurnaroundTime(temp, n);

    displayGanttChart(temp, n);

    table(temp,n);

}

void priority(Process processes[], int n)

{

    Process temp[10];

    int time=0;

    for(int i=0;i<n;i++)

        processes[i].done=0;

    for(int i=0;i<n;i++){

        int high=-1;

        for(int j=0;j<n;j++){

            if(processes[j].arrival\_time<=time && processes[j].done==0){

                if(high==-1) high=j;

                else if(processes[j].priority>processes[high].priority)

                    high=j;

            }

        }

        processes[high].done=1;

        time+=processes[high].burst\_time;

        temp[i]=processes[high];

        temp[i].waiting\_time=time-temp[i].burst\_time-temp[i].arrival\_time;

    }

    calculateWaitingTime(temp, n);

    calculateTurnaroundTime(temp, n);

    displayGanttChart(temp, n);

    table(temp,n);

}

void calculateTurnaroundTime(Process processes[], int n)

{

int sum=0;

    for (int i = 0; i < n; i++)

    {

        processes[i].turnaround\_time = processes[i].burst\_time + processes[i].waiting\_time;

        sum+=processes[i].turnaround\_time;

    }

    float avg=(sum+0.0)/n;

    printf("\nAverage TT = %f\n",avg);

}

void calculateWaitingTime(Process processes[], int n)

{

    int sum=0;

    for (int i = 1; i < n; i++)

    {

        sum+=processes[i].waiting\_time;

    }

    float wt=(sum+0.0)/n;

    printf("\nAverage WT=%f\n",wt);

}

void displayGanttChart(Process p[], int n)

{

    int i,j;

    char topLeft[]="\u250c", topRight[]="\u2510";

    char bottomLeft[]="\u2514" , bottomRight[]="\u2518";

    char horizontal[]="\u2500", vertical[]="\u2502";

    char horiDown[]="\u252c" , horiUp[]="\u2534" ;

    printf("\n\n");

    printf("%s",topLeft );

    for(i=0;i<n;i++){

        for(j=0;j<p[i].burst\_time;j++)

            printf("%s%s",horizontal,horizontal );

        if(i<n-1)

            printf("%s",horiDown);

        else

            printf("%s",topRight);

    }

    printf("\n%s",vertical);

    for(i=0;i<n;i++){

        for(j=0;j<p[i].burst\_time-1;j++)

            printf(" ");

        printf("P%d",p[i].id);

        for(j=0;j<p[i].burst\_time-1;j++)

            printf(" ");

        printf("%s",vertical);

    }

    printf("\n");

    printf("%s",bottomLeft);

    for(i=0;i<n;i++){

        for(j=0;j<p[i].burst\_time;j++)

            printf("%s%s",horizontal,horizontal);

        if(i<n-1)

            printf("%s",horiUp);

        else

            printf("%s",bottomRight);

    }

    printf("\n%d",p[0].arrival\_time);

    for(i=0;i<n;i++){

        for(j=0;j<p[i].burst\_time;j++)

            printf("  ");

        if(p[i].turnaround\_time + p[i].arrival\_time>9)

            printf("\b");

        printf("%d",p[i].turnaround\_time+p[i].arrival\_time);

    }

    printf("\n\n\n");

}

void table(Process p[], int n){

    printf("PID\tPrio\tArr\tBurst\tTT\tWT \n ");

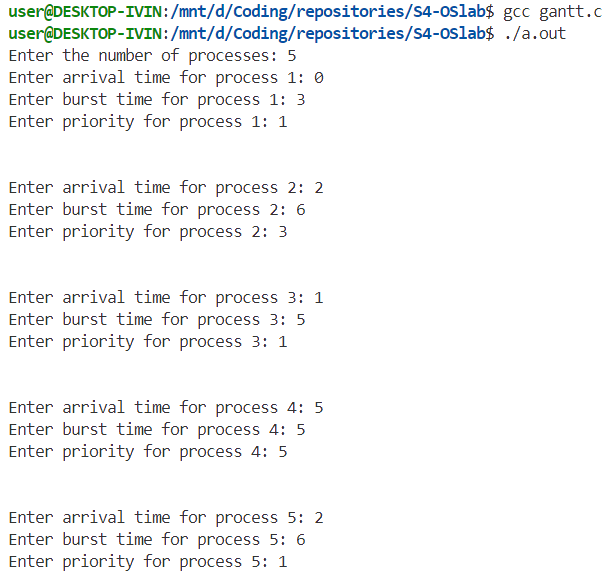
    for(int i=0;i<n;i++){

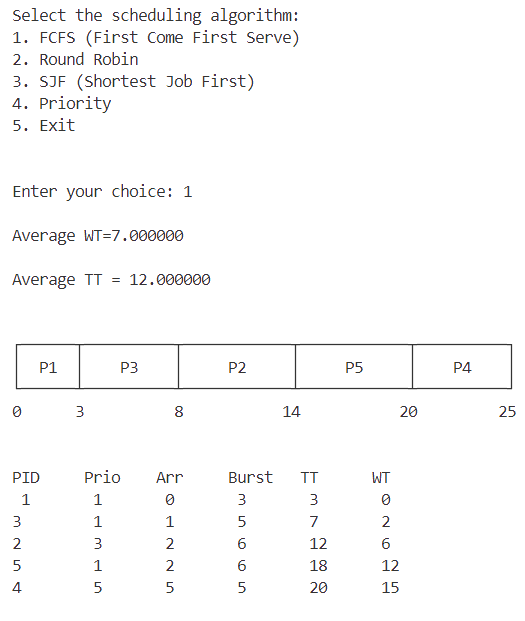
        printf("%d \t %d \t %d \t %d \t %d \t %d \n",p[i].id , p[i].priority , p[i].arrival\_time, p[i].burst\_time, p[i].turnaround\_time, p[i].waiting\_time);

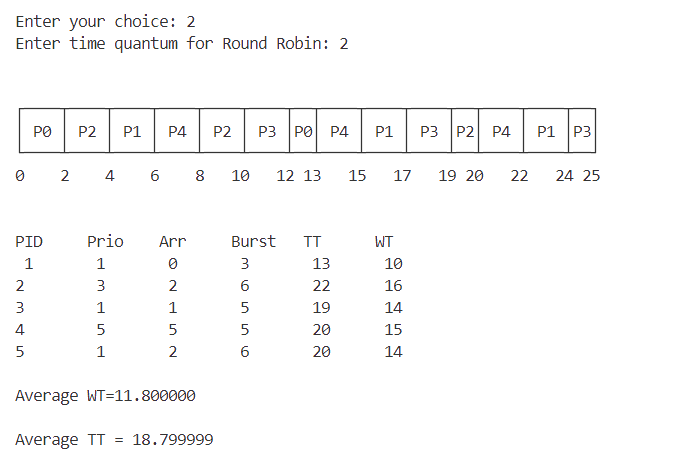
    }

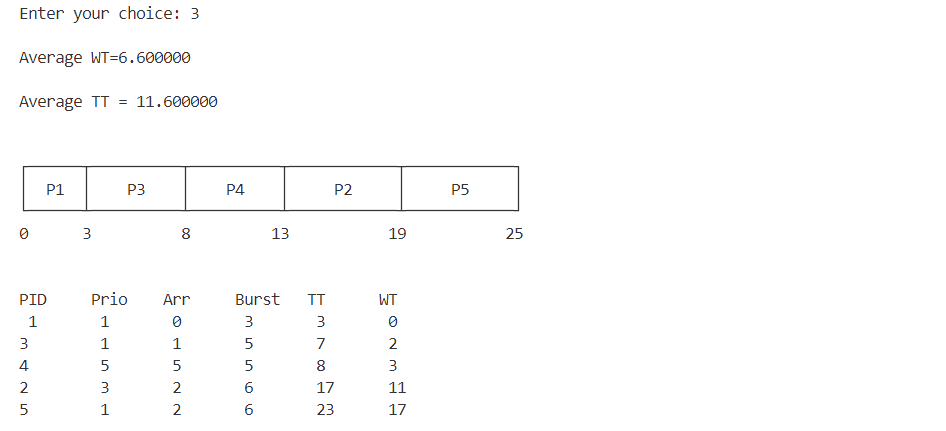
}

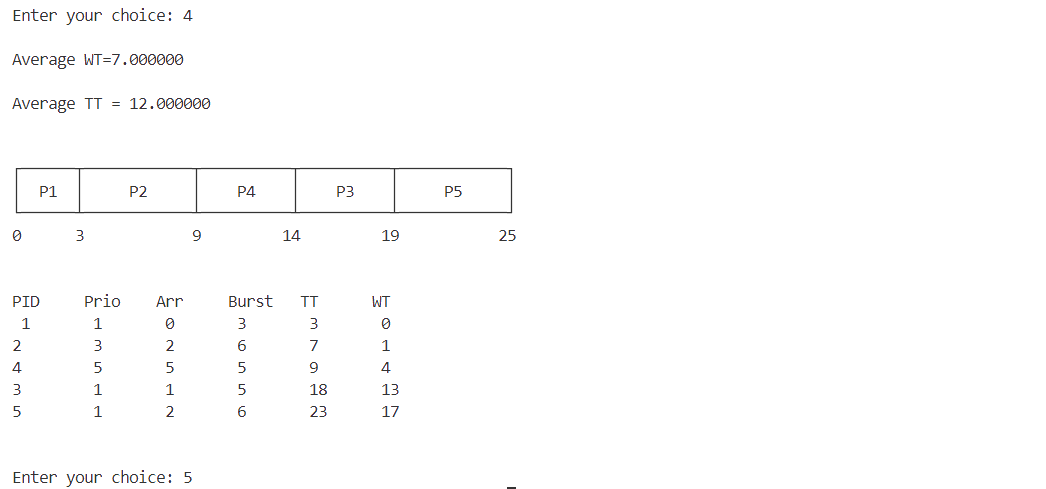
**Output**











**Result**

The program has been executed and output has been verified.