Name: Meet Brijwani

Roll no: 14

Batch: S-11

Aim : : a) To study and implement non preemptive scheduling algorithm FCFS.

b) To study and implement preemptive scheduling algorithm SRTF.

Theory :

Non preemptive algorithm :

FCFS, SJF 1] FCFS : First Come

First Serve

It is a non-preemptive scheduling algorithm and the criteria for this is the arrival time of the

process CPU is allotted to the process that requires it first. Jobs arriving later are placed at the end of the queue.

FCFS (Example)

Process	Duration	Oder	Arrival Time
P1	24	1	0
P2	3	2	0
P3	4	3	0

Gantt Chart :

P1 waiting time : 0
P2 waiting time : 24
P3 waiting time : 27

The Average waiting time :

(0+24+27)/3 = 17

2] SJF : shortest job first

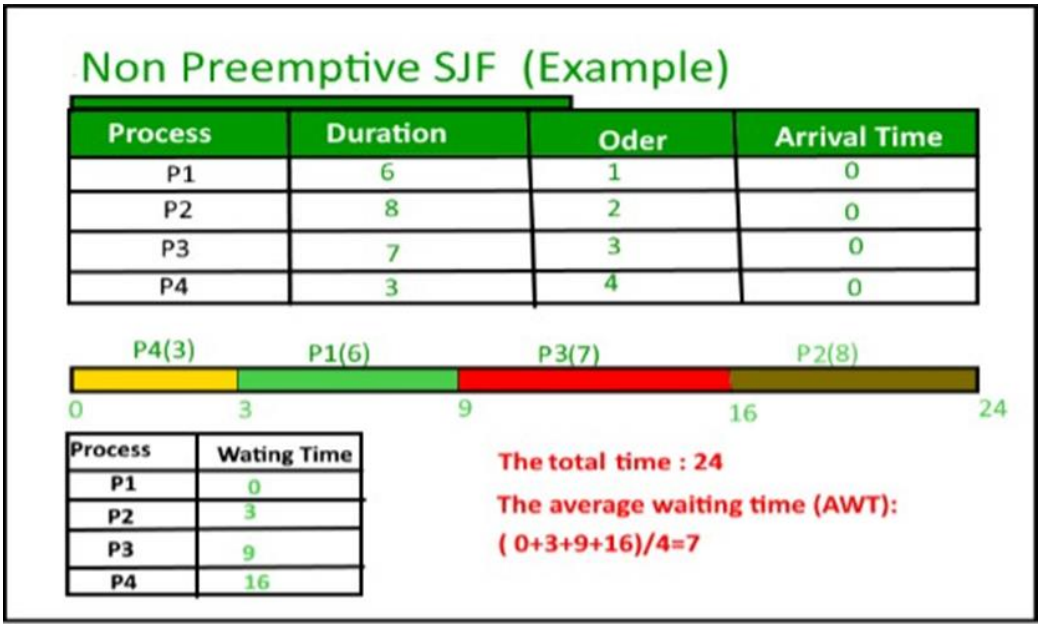
This is a non preemptive scheduling algorithm, which associates with each process the length of

the processes next CPU

first. Criteria : Burst Time.

This algorithm assigns processes according to BT if BT of two processes is the same we use

FCFS scheduling criteria.

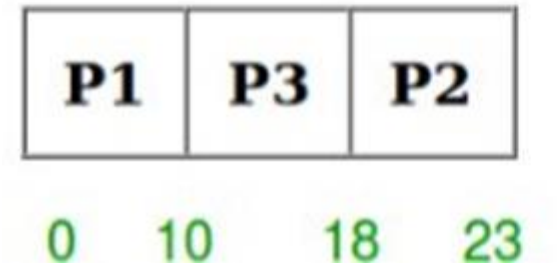


3] Priority Scheduling

This is a non preemptive scheduling algorithm. Each process here has a priority that is either

assigned already or externally done.

Process	Burst Time	Priority
P1	10	2
P2	5	0
P3	8	1



Preemptive Scheduling Algorithm : SRTF/STRN

Shortest Remaining Time First / Shortest Remaining Time Next scheduling.

It is a preemptive SJF Algorithm. The choice arrives when a new process arrives as the ready

queue. While a previous process is still executing. The next CPU burst if the newly arrived

process may be shorter than what is left of the currently executing process. A preemptive SJF

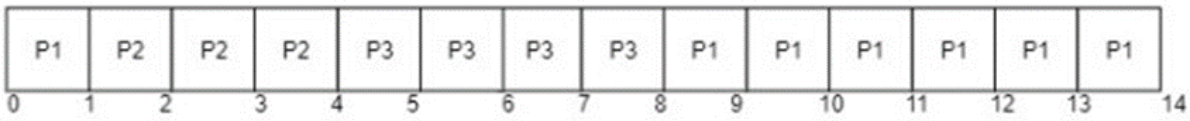
will preempt the current executing process and not allow the currently running process to finish

its CPU burst.

Round Robin is also one preemptive algorithm.

Process	Burst Time	Arrival Time
P1	7	0
P2	3	1
P3	4	3

The Gantt Chart for SRTF will be:



Implementation with code FCFS:

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

int main() {
    int n;
    printf("Enter no. of processes : ");
    scanf("%d", &n);

    int *at = (int *)malloc(n * sizeof(int));
    int *bt = (int *)malloc(n * sizeof(int));
    int *ct = (int *)malloc(n * sizeof(int));
    int *wt = (int *)malloc(n * sizeof(int));
    int *tat = (int *)malloc(n * sizeof(int));

    printf("Enter arrival and burst times of the processes:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d%d", &at[i], &bt[i]);
    }
```

```
    int total_tat = 0;
    int t = at[0];
    for (int i = 0; i < n; i++) {
        t += bt[i];
        ct[i] = t;
        tat[i] = ct[i] - at[i];
        total_tat += tat[i];
        wt[i] = ct[i] - bt[i];
    }

    printf("    AT    BT    CT    TAT    WT\n");
    for (int i = 0; i < n; i++) {
        printf("%6d%6d%6d%7d%6d\n", at[i], bt[i], ct[i], tat[i], wt[i]);
    }

    printf("Average TAT = %.2f\n", (float)total_tat / n);

    free(at);
    free(bt);
    free(ct);
    free(wt);
    free(tat);

    return 0;
}
```

Output :

Implementation with code SRTF:

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

struct Process {
    int pid; // Process ID
    int bt; // Burst Time
    int art; // Arrival Time
};

void findWaitingTime(struct Process proc[], int n, int wt[]) {
    int rt[n];
    for (int i = 0; i < n; i++)
        rt[i] = proc[i].bt;

    int complete = 0, t = 0, minm = INT_MAX;
    int shortest = 0, finish_time;
    int check = 0;
```

```
    while (complete != n) {
        for (int j = 0; j < n; j++) {
            if ((proc[j].art <= t) && (rt[j] < minm) && rt[j] > 0) {
                minm = rt[j];
                shortest = j;
                check = 1;
            }
        }

        if (check == 0) {
            t++;
            continue;
        }

        rt[shortest]--;
        minm = rt[shortest];
        if (minm == 0)
            minm = INT_MAX;

        if (rt[shortest] == 0) {
            complete++;
            check = 0;
            finish_time = t + 1;
            wt[shortest] = finish_time - proc[shortest].bt - proc[shortest].art;
            if (wt[shortest] < 0)
                wt[shortest] = 0;
        }
        t++;
    }
}

void findTurnAroundTime(struct Process proc[], int n, int wt[], int tat[]) {
    for (int i = 0; i < n; i++)
        tat[i] = proc[i].bt + wt[i];
}

void findavgTime(struct Process proc[], int n) {
```

```
    int *wt = (int *)malloc(n * sizeof(int));
    int *tat = (int *)malloc(n * sizeof(int));
    int total_wt = 0, total_tat = 0;

    findWaitingTime(proc, n, wt);
    findTurnAroundTime(proc, n, wt, tat);

    printf(" P\t\tBT\t\tWT\t\tTAT\t\t\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf(" %d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].bt, wt[i], tat[i]);
    }

    printf("\nAverage waiting time = %.2f\n", (float)total_wt / (float)n);
    printf("Average turn around time = %.2f\n", (float)total_tat / (float)n);

    free(wt);
    free(tat);
}

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

    struct Process *proc = (struct Process *)malloc(n * sizeof(struct
Process));
    printf("Enter arrival time and burst time for each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d %d", &proc[i].art, &proc[i].bt);
        proc[i].pid = i + 1;
    }

    findavgTime(proc, n);
```

```
    free(proc);

    return 0;
}
```

Output:

```
Enter the number of processes: 5
Enter arrival time and burst time for each process:
Process 1: 2 6
Process 2: 5 2
Process 3: 1 8
Process 4: 0 3
Process 5: 4 4
 P              BT             WT             TAT
 1              6              7              13
 2              2              0              2
 3              8              14             22
 4              3              0              3
 5              4              2              6

Average waiting time = 4.60
Average turn around time = 9.20
```

Conclusion : Thus we have successfully implemented preemptive and non preemptive scheduling algorithms.