Experiment No.5
Process Management: Scheduling a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS) b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)
Date of Performance:
Date of Submission:
Marks:
Sign:

Aim: To study and implement process scheduling algorithms FCFS and SJF

Objective:

a. Write a program to demonstrate the concept of non-preemptive scheduling algorithms. (FCFS)
b. Write a program to demonstrate the concept of preemptive scheduling algorithms (SJF)

Theory:

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms.

These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

Jobs are executed on a first come, first serve basis. It is a non-preemptive, preemptive scheduling algorithm. Easy to understand and implement. Its implementation is based on the FIFO queue. Poor in performance as average wait time is high.

Shortest Job First (SJF)

This is also known as the shortest job first, or SJF. This is a non-preemptive, preemptive scheduling algorithm. Best approach to minimize waiting time. Easy to implement in Batch systems where required CPU time is known in advance. Impossible to implement in interactive systems where required CPU time is not known. The processor should know in advance how much time the process will take.

Program 1:

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

```
int main()

{
   int n,bt[30],wait_t[30],turn_ar_t[30],av_wt_t=0,avturn_ar_t=0,i,j;
   printf("Please enter the total number of processes(maximum 30):");  // the maximum process that be used to calculate is specified.
   scanf("%d",&n);

   printf("\nEnter The Process Burst Timen");
   for(i=0;i<n;i++)  // burst time for every process will be taken as input
   {
      printf("P[%d]:",i+1);
      scanf("%d",&bt[i]);
   }

   wait_t[0]=0;

   for(i=1;i<n;i++)
   {
      wait_t[i]=0;
      for(j=0;j<i;j++)
         wait_t[i]+=bt[j];
   }

   printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");

   for(i=0;i<n;i++)
```

```
    {
        turn_ar_t[i]=bt[i]+wait_t[i];
        av_wt_t+=wait_t[i];
        avturn_ar_t+=turn_ar_t[i];
        printf("\nP[%d]\t\t%d\t\t\t%d\t\t\t\t%d",i+1,bt[i],wait_t[i],turn_ar_t[i]);
    }

    av_wt_t/=i;
    avturn_ar_t/=i;  // average calculation is done here
    printf("\nAverage Waiting Time:%d",av_wt_t);
    printf("\nAverage Turnaround Time:%d",avturn_ar_t);

    return 0;
}
```

Output:

```
Please enter the total number of processes(maximum 30):3

Enter The Process Burst TimenP[1]:7
P[2]:18
P[3]:45

Process     Burst Time  Waiting Time    Turnaround Time
P[1]        7           0               7
P[2]        18          7               25
P[3]        45          25              70
Average Waiting Time:10
Average Turnaround Time:34
```

Program 2:

```
#include <stdio.h>

int main()

{

        // Matrix for storing Process Id, Burst

        // Time, Average Waiting Time & Average

        // Turn Around Time.

        int A[100][4];
```

```
int i, j, n, total = 0, index, temp;
float avg_wt, avg_tat;
printf("Enter number of process: ");
scanf("%d", &n);
printf("Enter Burst Time:\n");
// User Input Burst Time and alloting Process Id.
for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1;
}
// Sorting process according to their Burst Time.
for (i = 0; i < n; i++) {
        index = i;
        for (j = i + 1; j < n; j++)
                if (A[j][1] < A[index][1])
                        index = j;
        temp = A[i][1];
        A[i][1] = A[index][1];
        A[index][1] = temp;

        temp = A[i][0];
        A[i][0] = A[index][0];
        A[index][0] = temp;
}
A[0][2] = 0;
```

```
        // Calculation of Waiting Times
        for (i = 1; i < n; i++) {
                A[i][2] = 0;
                for (j = 0; j < i; j++)
                        A[i][2] += A[j][1];
                total += A[i][2];
        }
        avg_wt = (float)total / n;
        total = 0;
        printf("P        BT        WT        TAT\n");
        // Calculation of Turn Around Time and printing the
        // data.
        for (i = 0; i < n; i++) {
                A[i][3] = A[i][1] + A[i][2];
                total += A[i][3];
                printf("P%d        %d        %d        %d\n", A[i][0],
                        A[i][1], A[i][2], A[i][3]);
        }
        avg_tat = (float)total / n;
        printf("Average Waiting Time= %f", avg_wt);
        printf("\nAverage Turnaround Time= %f", avg_tat);
}
```

Output:

```
enter the no of processes : 2
the arrival time for process P1 : 10
the burst time for process P1 : 5
the arrival time for process P2 : 6
the burst time for process P2 : 3
P[10]   |   9999    |   0


 average waiting time = 0.000000

 average turnaround time = 4999.500000
```

Conclusion:

What is the difference between Preemptive and Non-Preemptive algorithms?

Preemptive and non-preemptive scheduling algorithms are two different approaches used by operating systems to manage the execution of multiple processes or threads. In preemptive scheduling, the operating system can interrupt a currently running process and allocate the CPU to another process according to priority or a predefined scheduling algorithm. This means that higher-priority processes can preempt lower-priority ones, potentially leading to more equitable CPU allocation and better responsiveness. Preemptive scheduling is often associated with real-time operating systems and environments where responsiveness is critical. On the other hand, non-preemptive scheduling, also known as cooperative scheduling, allows a process to continue running until it voluntarily relinquishes the CPU or blocks for I/O. In this approach, the operating system cannot force a process to give up the CPU, and scheduling decisions are typically made based on process priorities, arrival times, or other criteria when a process voluntarily yields control. Non-preemptive scheduling can be simpler to implement and may result in less overhead compared to preemptive scheduling, but it can also lead to potential issues such as priority inversion and reduced responsiveness if a high-priority process is waiting for a low-priority one to finish. In summary, the key distinction between preemptive and non-preemptive scheduling algorithms lies in whether the operating system can forcibly interrupt a running process or if processes are allowed to continue executing until they voluntarily yield control. Each approach has its own advantages and disadvantages, and the choice between them depends on factors such as system requirements, performance goals, and the nature of the workload being managed