ASSIGNMENT REPORT

on

Priority Based Scheduling

Submitted by,

Shreya Singh
Regno- 11612233
Roll No-B53
Computer Science Engineering
(B.Tech)
Dated: 19th April 19, 2018

Submitted to,

Pradeep Kumar

School of Computer Science & Engineering
Lovely Professional University, Phagwara

Table of Contents

1. Introduction:

1.1 Introduction to my topic:

Preemptive Priority Scheduling Algorithm

The preemptive priority scheduling algorithm is a popular operating system process management and job scheduling algorithm.

Every job that enters the job queue is assigned a priority based on which its execution takes place. As simple it sounds, the processes with a higher priority will be executed first and then the processes with the lower priorities will be executed.

If there are multiple processes in the queue with the same priority, then such jobs are executed in the order of their arrival often called as **first come first served**.

In this preemptive implementation of priority scheduling program in C, we consider the **arrival time** of the processes.

Since this is a preemptive job scheduling algorithm, the CPU can leave the process midway. The current state of the process will be saved by the **context switch**.

The system can then search for another process with a higher priority in the ready queue or waiting queue and start its execution.

Once the CPU comes back to the previous incomplete process, the job is resumed from where it was earlier paused.

1.2 What is the Purpose of the Problems?

The main purpose of the problem is to find the average waiting time for each process using preemptive priority scheduling algorithm.

2. Problem analysis:

3.1 Priority(How to solve priority if same for different Process id)
3.2 Arrival Time (What is the arrival time given to system)
3.3 Burst Time (What is burst time required)
3.4 Service Time (What is the amount of CPU time)
3.5 Average WaitingTime(To calculate ?)

3. Algorithm Involved:

3.1 Preemptive Priority Schedule Algorithm:

Priority Scheduling always selects the process(es) with the highest priority currently ready to run. If there is more than one process having the currently highest priority, you need a second scheduling algorithm to choose among these processes. *Non-preemptive Priority Scheduling* only selects a new process to run if the running process finished its work or yields (voluntarily) to the scheduler.

Preemptive Priority Scheduling is the same algorithm but if a new process having a higher priority than the currently running process arrives, it gets selected immediately. The new process has not to wait until the currently running process finishes or yields.

3.2 Advantages & Disadvantages:

Advantages

- Preemptive priority scheduling is much more efficient as compared to the non-preemptive version.
- This priority job scheduling algorithm is quite simple to implement.
- The aging technique is implemented to reduce the starvation of lower priority processes.
- The average turnaround time and waiting time is efficient.

Disadvantages

- Indefinite blockage of the lower priority jobs.
- For a system failure occurs, the unfinished lower priority jobs are removed from the system and cannot be recovered.

4. Boundary Criteria:

1**. Process Id (like P1,P2,P3 and so on)**- In computing, the process identifier testing (normally referred to as the process ID or PID) is a number used by operating system kernels—such as UNIX, linux and Microsoft Windows to find an active process. This number may be used as a parameter in many function calls, allowing processes to be manipulated, such as adjusting the process's priority or killing it altogether.

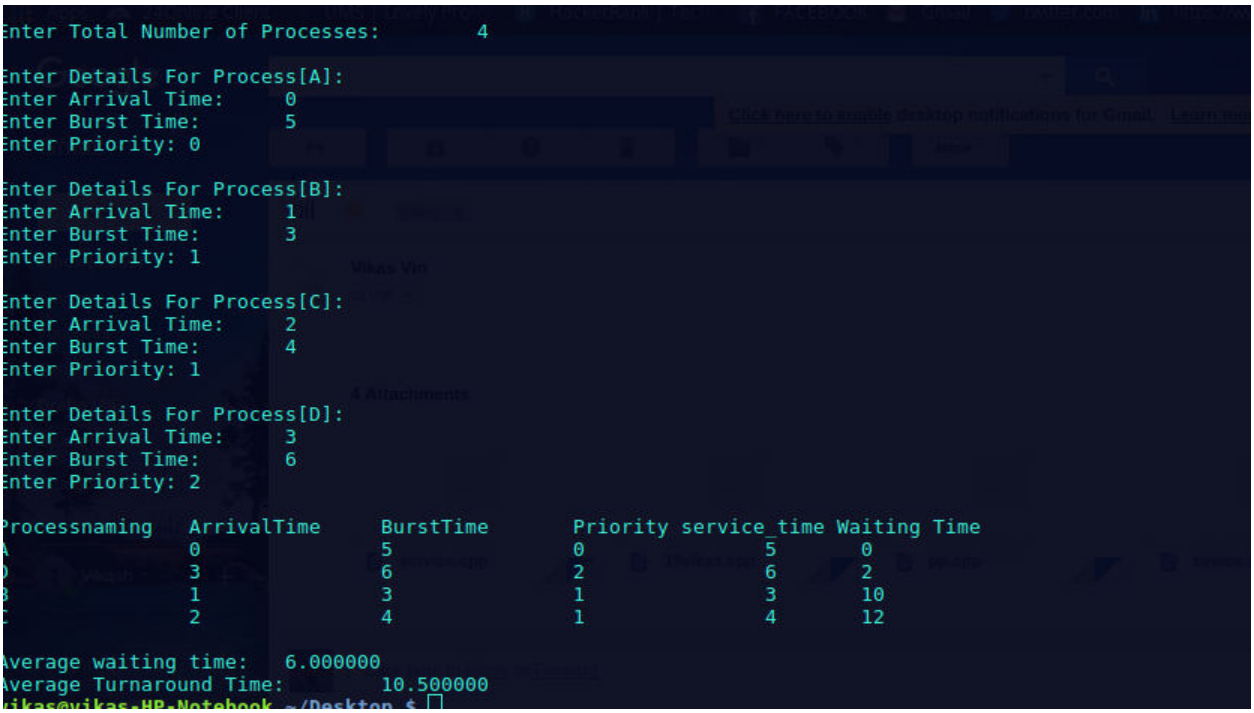
2. **Arrival time** - Time at which the process arrives in the ready queue

3. **Burst Time**- Time required by a process for CPU execution.

4. **Service Time**-The amount of CPU time that a process will need before it either finishes or voluntarily exits the CPU, such as to wait for input / output.

So we have to calculate the average waiting time for each process using preemptive priority scheduling algorithm.

5. Test Cases:


Enter Total Number of Processes: 4
Enter Details For Process[A]:
Enter Arrival Time: 0
Enter Burst Time: 5
Enter Priority: 0
Enter Details For Process[B]:
Enter Arrival Time: 1
Enter Burst Time: 3
Enter Priority: 1
Enter Details For Process[C]:
Enter Arrival Time: 2
Enter Burst Time: 4
Enter Priority: 1
Enter Details For Process[D]:
Enter Arrival Time: 3
Enter Burst Time: 6
Enter Priority: 2
Processnaming ArrivalTime BurstTime Priority service_time Waiting Time
A 0 5 0 5 0
3 6 2 6 2
1 3 1 3 10
2 4 1 4 12
Average waiting time: 6.000000
Average Turnaround Time: 10.500000

6. Solution Code:

```
#include<stdio.h>
struct process_schedule
{
    char process_naming;
    int arrival_timing, burst_timing, ctiming, waiting_timing, turnaround_timing,
priority,service_timing;
    int status;
}process_queue[10];

int size_limit;

void arrival_timing_Sort()
{
    struct process_schedule temp;
    int i, j;
    for(i = 0; i < size_limit - 1; i++)
    {
        for(j = i + 1; j < size_limit; j++)
        {
            if(process_queue[i].arrival_timing > process_queue[j].arrival_timing)
            {
                temp = process_queue[i];
                process_queue[i] = process_queue[j];
                process_queue[j] = temp;
            }
        }
    }
}

int  main()
{
    int i, time = 0, burst_timing = 0, largest, service_timing=0;
    char c;
    float wait_timing = 0, turnaround_timing = 0, average_waiting_timing,
average_turnaround_timing;
    printf("\nEnter Total Number of Processes:\t");
    scanf("%d", &size_limit);
    for(i = 0, c = 'A'; i < size_limit; i++, c++)
    {
```

```
        process_queue[i].process_naming = c;
        printf("\nEnter Details For Process[%C]:\n", process_queue[i].process_naming);
        printf("Enter Arrival Time:\t");
        scanf("%d", &process_queue[i].arrival_timing );
        printf("Enter Burst Time:\t");
        scanf("%d", &process_queue[i].burst_timing);
        printf("Enter Priority:\t");
        scanf("%d", &process_queue[i].priority);
        process_queue[i].status = 0;
        burst_timing = burst_timing + process_queue[i].burst_timing;

    }
    arrival_timing_Sort();
    process_queue[9].priority = -9999;
    printf("\nProcessnaming\tArrivalTime\tBurstTime\tPriority service_timing Waiting Time");
    for(time = process_queue[0].arrival_timing; time < burst_timing;)
    {
        largest = 9;
        for(i = 0; i < size_limit; i++)
        {
            if(process_queue[i].arrival_timing <= time && process_queue[i].status != 1 &&
process_queue[i].priority > process_queue[largest].priority)
            {
                largest = i;
            }
        }
        time = time + process_queue[largest].burst_timing;
        process_queue[largest].ctiming = time;
        process_queue[largest].waiting_timing = process_queue[largest].ctiming -
process_queue[largest].arrival_timing - process_queue[largest].burst_timing;
        process_queue[largest].turnaround_timing = process_queue[largest].ctiming -
process_queue[largest].arrival_timing;
        process_queue[largest].status = 1;
        wait_timing = wait_timing + process_queue[largest].waiting_timing;
        turnaround_timing = turnaround_timing + process_queue[largest].turnaround_timing;
        service_timing=service_timing+burst_timing;
        process_queue[largest].service_timing+=process_queue[largest].burst_timing;
        printf("\n%c\t\t%d\t\t%d\t\t%d\t\t%d\t%d", process_queue[largest].process_naming,
process_queue[largest].arrival_timing, process_queue[largest].burst_timing,
process_queue[largest].priority,process_queue[largest].service_timing,process_queue[largest].wa
iting_timing);
    }
    average_waiting_timing = wait_timing / size_limit;
```

```
    average_turnaround_timing = turnaround_timing / size_limit;
    printf("\n\nAverage waiting time:\t%f\n", average_waiting_timing);
    printf("Average Turnaround Time:\t%f\n", average_turnaround_timing);
}
```