Write a program for multilevel queue scheduling algorithm. There must be three queues generated. There must be specific range of priority associated with every queue. Now prompt the user to enter number of processes along with their priority and burst time. Each process must occupy the respective queue with specific priority range according to its priority. Apply Round Robin algorithm with quantum time 4 on queue with highest priority range. Apply priority scheduling algorithm on the queue with medium range of priority and First come first serve algorithm on the queue with lowest range of priority. Each and every queue should get a quantum time of 10 seconds. CPU will keep on shifting between queues after every 10 seconds

INTRODUCTION

Maximum CPU utilization obtained with multiprogramming n

CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait CPU burst followed by I/O burst

CPU burst distribution is of main concern

Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them

l Queue may be ordered in various ways

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state

2. Switches from running to ready state

3. Switches from waiting to ready

CPU utilization – keep the CPU as busy as possible

Throughput – # of processes that complete their execution per time unit

Turnaround time – amount of time to execute a particular process Waiting time – amount of time a process has been waiting in the ready queue

Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

ROUND ROBIN ALGORITHM

Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once. No process waits more than (n-1)q time units.

Timer interrupts every quantum to schedule next process n Performance

q large ⇒ FIFO

q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

PROIORITY SCHEDULING

Suppose that the processes arrive in the order:

P2 , P3 , P1

The Gantt chart for the schedule is:

Waiting time for P1 = 6; P2 = 0; P3 = 3

Average waiting time: (6 + 0 + 3)/3 = 3

Much better than previous case Convoy effect There is a convoy effect as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

METHODOLOGY

```
#include<stdlib.h>

#include<stdio.h>

#include<conio.h>

int main()

{

    int i, limit, total = 0, x, count = 0, time_quantum,j;


            int wait_time = 0, turnaround_time = 0,pos,z,p[10],prio[10], a_time[10], b_time[10], temp[10],b;


            float average_wait_time, average_turnaround_time;


            printf("\nEnter Total Number of Processes:");


            scanf("%d", &limit);


            x = limit;

    for(i = 0; i < limit; i++)

    {

              p[i]=i+1;
```

```
        prio[i]=0;
    printf("\nEnter total Details of Process[%d]\n", i + 1);
    printf("Arrival Time:\t");
    scanf("%d", &a_time[i]);
    printf("Burst Time:\t");
    scanf("%d", &b_time[i]);
    temp[i] = b_time[i];
}

printf("\nEnter the Time Quantum:");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\t Priority\n");
for(total = 0, i = 0; x != 0;)
{

                for(z=0;z<limit;z++)
                {
                        int temp1;
                        pos=z;
                        for(j=z+1;j<limit;j++)
                        {
                           if(prio[j]<prio[pos])
                                   pos=j;
                        }

                temp1=prio[z];

                prio[z]=prio[pos];

                prio[pos]=temp1;

                        temp1=b_time[z];
```

```
                        b_time[z]=b_time[pos];
                        b_time[pos]=temp1;
                                        temp1=a_time[z];
                                a_time[z]=a_time[pos];
                        a_time[pos]=temp1;

                        temp1=p[z];
                                p[z]=p[pos];
                        p[pos]=temp1;

                        temp1=temp[z];
                                temp[z]=temp[pos];
                        temp[pos]=temp1;
                   }
               {
               }

                        if(temp[i] <= time_quantum && temp[i] > 0)
{
    total = total + temp[i];
    temp[i] = 0;
    count = 1;
}

                        else if(temp[i] > 0)
{
    temp[i] = temp[i] - time_quantum;
    total = total + time_quantum;
}

   for(b=0;b<limit;b++)
               {
                        if(b==i)
```

```
                                prio[b]+=1;
                                else
                                prio[b]+=2;
                        }

        if(temp[i] == 0 && count == 1)
        {
            x--;
            printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d\t\t%d", p[i], b_time[i], total - a_time[i], total - a_time[i] -
b_time[i],prio[i]);
            wait_time = wait_time + total - a_time[i] - b_time[i];
            turnaround_time = turnaround_time + total - a_time[i];
            count = 0;
        }
        if(i == limit - 1)
        {
            i = 0;

                                }
        else if(a_time[i + 1] <= total)
        {
            i++;

                                }
        else
        {
            i = 0;

                                }
    }
    return 0;
}
```