Batch: A1 **Roll No.: 1711008**

Experiment No. 04

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

TITLE: Implementation of Basic Process management algorithms - Preemptive (SRTN, RR, priority)

AIM: To implement basic Process management algorithms (Round Robin,SRTN, Priority)

Expected Outcome of Experiment:

CO 3. Illustrate the working of process scheduling algorithms and compare their performance

Books/ Journals/ Websites referred:

1. **Silberschatz A., Galvin P., Gagne G. "Operating Systems Principles", Willey Eight edition.**
2. **Achyut S. Godbole , Atul Kahate "Operating Systems" McGraw Hill Third Edition.**
3. **William Stallings, "Operating System Internal & Design Principles", Pearson.**
4. **Andrew S. Tanenbaum, "Modern Operating System", Prentice Hall.**

Pre Lab/ Prior Concepts:

Most systems have a large number of processes with short CPU bursts interspersed between I/O requests and a small number of processes with long CPU bursts. To provide good time-sharing performance, we may preempt a running process to let another one run. The ready list, also known as a run queue, in the operating system keeps a list of all processes that are ready to run and not blocked on some I/O or other system request, such as a semaphore. Then entries in this list are pointers to the process control block, which stores all information and state about a process.

When an I/O request for a process is complete, the process moves from the *waiting* state to the *ready* state and gets placed on the run queue.

The process scheduler is the component of the operating system that is responsible for deciding whether the currently running process should continue running and, if not, which process should run next. There are four events that may occur where the scheduler needs to step in and make this decision:

1. The current process goes from the *running* to the *waiting* state because it issues an I/O request or some operating system request that cannot be satisfied immediately.

2. The current process terminates.

3. A timer interrupt causes the scheduler to run and decide that a process has run for its allotted interval of time and it is time to move it from the *running* to the *ready* state.

4. An I/O operation is complete for a process that requested it and the process now moves from the *waiting* to the*ready* state. The scheduler may then decide to preempt the currently-running process and move this *ready* process into the *running* state.

The decisions that the scheduler makes concerning the sequence and length of time that processes may run is called the scheduling algorithm (or scheduling policy). These decisions are not easy ones, as the scheduler has only a limited amount of information about the processes that are ready to run. A good scheduling algorithm should:

1. Be fair – give each process a fair share of the CPU, allow each process to run in a reasonable amount of time.
2. Be efficient – keep the CPU busy all the time.
3. Maximize throughput – service the largest possible number of jobs in a given amount of time; minimize the amount of time users must wait for their results.
4. Minimize response time – interactive users should see good performance
5. Minimize overhead – don't waste too many resources. Keep scheduling time and context switch time at a minimum.
6. Maximize resource use – favor processes that will use underutilized resources. There are two motives for this. Most devices are slow compared to CPU operations. We'll achieve better system throughput by keeping devices busy as often as possible. The second reason is that a process may be holding a key resource and other, possibly more important, processes

cannot use it until it is released. Giving the process more CPU time may free up the resource quicker.

7. Avoid indefinite postponement – every process should get a chance to run eventually.

Description of the application to be implemented:

Round Robin Algorithm

Round Robin scheduling algorithm is one of the most popular scheduling algorithm which can actually be implemented in most of the operating systems. This is the **preemptive version** of first come first serve scheduling. The Algorithm focuses on Time Sharing. In this algorithm, every process gets executed in a **cyclic way**. A certain time slice is defined in the system which is called time **quantum**. Each process present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will **terminate** else the process will go back to the **ready queue** and waits for the next turn to complete the execution.

Advantages

1. It can be actually implementable in the system because it is not depending on the burst time.
2. It doesn't suffer from the problem of starvation or convoy effect.
3. All the jobs get a fare allocation of CPU.

Disadvantages

1. The higher the time quantum, the higher the response time in the system.
2. The lower the time quantum, the higher the context switching overhead in the system.
3. Deciding a perfect time quantum is really a very difficult task in the system.

Shortest Remaining Time First Algorithm :

This Algorithm is the **preemptive version** of **SJF scheduling**. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**. The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

Priority scheduling

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

The difference between preemptive priority scheduling and non preemptive priority scheduling is that, in the preemptive priority scheduling, the job which is being executed can be stopped at the arrival of a higher priority job.

Once all the jobs get available in the ready queue, the algorithm will behave as non-preemptive priority scheduling, which means the job scheduled will run till the completion and no preemption will be done.

Implementation details: (printout of code)

A)priority

```
import java.util.*;
import java.util.LinkedList;
import java.util.Queue;
class Main {
  static Queue<Vector> q;
 static int count=0;
public static boolean find(Queue <Vector>q,char z)
{
  boolean flag=false;
for (Vector item: q) {

      if(item.contains(z))
      {
        //System.out.println("in find"+z);
        flag=true;
        break;

      }
}
```

```
    return flag;
}
  public static Queue check(Vector v,int timer,Queue q)
  {
    char a;
    Vector n=new Vector();
    for(int i=0;i<v.size();i++){
    Vector m=(Vector)v.get(i);
    if((int)m.get(1)<=timer)
    {
      if(find(q,(char)m.get(0)))
        continue;
        else{

      q.add(m);
        }


    }

    }



  return q;


  }
  public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    Vector b=new Vector();
    int count1=0,timer_temp;
    int a,bu,p,sum_wait=0,turn_wait=0,t,w,total=0,val;
    char n;
       HashMap<Character, Integer>wait = new HashMap<>();
        HashMap<Character, Integer>turn = new HashMap<>();

    while(true)
    {

      a=sc.nextInt();
      bu=sc.nextInt();
```

```
     p=sc.nextInt();
    if(a==-1 && bu==-1)
    break;
    else
    {

        n=sc.next().charAt(0);
        Vector s=new Vector(4);
        s.add(n);
        s.add(a);
        s.add(bu);
        s.add(p);
        wait.put(n,0);
        turn.put(n,bu);
        total++;
        b.add(s);

    }



}

    int timer=0,q_size,m1,count=4;
    int min=0,min2=10000;
    char a1='z';
   float wait_avg=0,turn_avg=0;
    boolean flag=true;
    Queue<Vector> q = new LinkedList<Vector>();
    Queue<Vector> q_temp = new LinkedList<Vector>();
    Vector<Character> bb=new Vector<Character>();
    q=check(b,timer,q);
    while(b.size()!=0){

       //System.out.println(q);

      if(q.size()!=0)
      {
```

```
            for (Vector item: q) {// see if there is another process
having same prio

       if((int)item.get(3)>=min)
       {
         min=(int)item.get(3);
                a1=(char)item.get(0);
                q_temp.add(item);


       }
}
if(q_temp.size()>1&&timer!=0)//check arrival is 2 have same prio
{
  for (Vector item: q_temp) {

       if((int)item.get(1)<=min2)
       {
         min2=(int)item.get(1);
                a1=(char)item.get(0);



       }
}
}


               for(int k1=0;k1<b.size();k1++)
               {

                Vector m=(Vector)b.get(k1);

                if(m.contains(a1))
                {
                  if((int)m.get(2)<=1)
                  {

                  System.out.print("from "+timer+" to ");
                  timer=timer+(int)m.get(2);
                   val=(int)turn.get(a1);
                  t=timer-(int)m.get(1);
                   w=t-val;
```

```
                    System.out.print(timer+" process  "+a1+" will run");
                    System.out.println();
                    wait.put(a1,w);
                   turn.put(a1,t);
                    turn_avg=turn_avg+t;
                    wait_avg=wait_avg+w;
                    b.remove(m);
                    q=check(b,timer,q);
                    q.remove(m);
                    min=0;
                   for (Vector item: q) {

       if((int)item.get(3)>=min)
       {
         min=(int)item.get(3);



       }
}

                    }
                    else
                    {

                      System.out.print("from "+(timer)+" to ");
                    timer=timer+1;
                    System.out.print((timer)+" process  "+a1+" will run");



                    System.out.println();
                    m1=(int)m.get(2);
                    m1=m1-1;
                    m.set(2,m1);
                    min=(int)m.get(3);
                    q=check(b,timer,q);
                    //System.out.println(q);
                    q.add(m);
                    q.remove(m);


                    }
```

```
                break;

              }

          }

        }
        else
        {
          System.out.println("idle time");
          timer=timer+1;




        }


        min2=10000;
        q_temp.clear();


      }


System.out.println("wait time of all process is ");
System.out.print(wait);
System.out.println();
System.out.println("turn time of all process is ");
System.out.print(turn);
System.out.println();
turn_avg=turn_avg/total;
   wait_avg=wait_avg/total;
System.out.println("avg t-time is "+turn_avg+" and avg w-time is
"+wait_avg);




  }
}
```

```
0   4   2 a
1   3   3 b
2   1   4 c
3   5   5 d
4 2 5 e
-1 -1 -1 o
from 0 to 1 process  a will run
from 1 to 2 process  b will run
from 2 to 3 process  c will run
from 3 to 4 process  d will run
from 4 to 5 process  d will run
from 5 to 6 process  d will run
from 6 to 7 process  d will run
from 7 to 8 process  d will run
from 8 to 9 process  e will run
from 9 to 10 process  e will run
from 10 to 11 process  b will run
from 11 to 12 process  b will run
from 12 to 13 process  a will run
from 13 to 14 process  a will run
from 14 to 15 process  a will run
wait time of all process is
{a=11, b=8, c=0, d=0, e=4}
turn time of all process is
{a=15, b=11, c=1, d=5, e=6}
avg t-time is 7.6 and avg w-time is 4.6
```

B)SRTN

```
import java.util.*;
import java.util.LinkedList;
import java.util.Queue;
class Main {
  static Queue<Vector> q;
 static int count=0;
public static boolean find(Queue <Vector>q,char z)
{
  boolean flag=false;
for (Vector item: q) {

      if(item.contains(z))
      {
        //System.out.println("in find"+z);
        flag=true;
        break;

      }
}
```

```
    return flag;
}
  public static Queue check(Vector v,int timer,Queue q)
  {
    char a;
    Vector n=new Vector();
    for(int i=0;i<v.size();i++){
    Vector m=(Vector)v.get(i);
    if((int)m.get(1)<=timer)
    {
      if(find(q,(char)m.get(0)))
        continue;
        else{

      q.add(m);
        }


    }

    }



  return q;


  }
  public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    Vector b=new Vector();
    int a,bu,p,t,w,total=0,val;
      float wait_avg=0,turn_avg=0;
    char n;
       HashMap<Character, Integer>wait = new HashMap<>();
        HashMap<Character, Integer>turn = new HashMap<>();

    while(true)
    {

      a=sc.nextInt();
      bu=sc.nextInt();
```

```
    if(a==-1 && bu==-1)
    break;
    else
    {

        n=sc.next().charAt(0);
        Vector s=new Vector(3);
        s.add(n);
        s.add(a);
        s.add(bu);
        wait.put(n,0);
        turn.put(n,bu);
        total++;
        b.add(s);

    }



}

    int timer=0,q_size,m1,count=4;
    int min=100000,min2=10000;
    int ascii_z=(int)('z');
    char a1='z';
    boolean flag=true;
    Queue<Vector> q = new LinkedList<Vector>();
    Queue<Vector> q_temp = new LinkedList<Vector>();
    q=check(b,timer,q);
    while(b.size()!=0){

       //System.out.println(q);

      if(q.size()!=0)
      {



        {

         for (Vector item: q) {//check 2 process have same burst time
```

```
      if((int)item.get(2)<=min)
      {
        min=(int)item.get(2);
               a1=(char)item.get(0);
               q_temp.add(item);


      }
}
if(q_temp.size()>1&&timer!=0)//check arrival if 2 process have same burst
{
  for (Vector item: q_temp) {

      if((int)item.get(1)<=min2)
      {
        min2=(int)item.get(1);
               a1=(char)item.get(0);



      }
}
}


              for(int k1=0;k1<b.size();k1++)
              {

               Vector m=(Vector)b.get(k1);

               if(m.contains(a1))
               {
                 if((int)m.get(2)<=1)
                 {
                 System.out.print("from "+timer+" to ");
                 timer=timer+(int)m.get(2);
                  val=(int)turn.get(a1);
                 t=timer-(int)m.get(1);
                  w=t-val;
                 System.out.print(timer+" process  "+a1+" will run");
                 System.out.println();
                 wait.put(a1,w);
                turn.put(a1,t);
                 turn_avg=turn_avg+t;
```

```
                    wait_avg=wait_avg+w;
                    b.remove(m);
                    q=check(b,timer,q);
                    q.remove(m);
                    min=100000;
                   for (Vector item: q) {

        if((int)item.get(2)<=min)
        {
          min=(int)item.get(2);



        }
}

                    }
                    else
                    {
                      System.out.print("from "+timer+" to ");
                    timer=timer+1;
                    System.out.print(timer+" process  "+a1+" will run");
                    System.out.println();
                    m1=(int)m.get(2);
                    m1=m1-1;
                    m.set(2,m1);
                    min=(int)m.get(2);
                    q=check(b,timer,q);
                    //System.out.println(q);
                    q.add(m);
                    q.remove(m);


                    }


                    break;

                  }

              }
              }
            }
            else
```

```
        {
          System.out.println("idle time");
          timer=timer+1;



        }
        min2=10000;
        q_temp.clear();


      }

System.out.println("wait time of all process is ");
System.out.print(wait);
System.out.println();
System.out.println("turn time of all process is ");
System.out.print(turn);
System.out.println();
turn_avg=turn_avg/total;
   wait_avg=wait_avg/total;
System.out.println("avg t-time is "+turn_avg+" and avg w-time is
"+wait_avg);




  }
}
```

```
0 3 a
2 6 b
4 4 c
6 5 d
8 2 e
-1 -1 m
from 0 to 1 process  a will run
from 1 to 2 process  a will run
from 2 to 3 process  a will run
from 3 to 4 process  b will run
from 4 to 5 process  c will run
from 5 to 6 process  c will run
from 6 to 7 process  c will run
from 7 to 8 process  c will run
from 8 to 9 process  e will run
from 9 to 10 process  e will run
from 10 to 11 process  b will run
from 11 to 12 process  b will run
from 12 to 13 process  b will run
from 13 to 14 process  b will run
from 14 to 15 process  b will run
from 15 to 16 process  d will run
from 16 to 17 process  d will run
from 17 to 18 process  d will run
from 18 to 19 process  d will run
from 19 to 20 process  d will run
wait time of all process is
{a=0, b=7, c=0, d=9, e=0}
turn time of all process is
{a=3, b=13, c=4, d=14, e=2}
avg t-time is 7.2 and avg w-time is 3.2
```

3)rounrobin

```
import java.util.*;
import java.util.LinkedList;
import java.util.Queue;
class Main {
  static Queue<Vector> q;
 static int count=0;
public static boolean find(Queue <Vector>q,char z)
{
  boolean flag=false;
for (Vector item: q) {

      if(item.contains(z))
      {
        flag=true;
```

```
        break;

      }
}


    return flag;
}
  public static Queue check(Vector v,int timer,Queue q)
  {
    char a;
    Vector n=new Vector();
    for(int i=0;i<v.size();i++){
    Vector m=(Vector)v.get(i);
    if((int)m.get(1)<=timer)
    {
      if(find(q,(char)m.get(0)))
        continue;
        else{

      q.add(m);
        }


    }

    }



  return q;


  }
  public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    Vector b=new Vector();
    int a,bu,val,t,w,total=0;
    char n;
    float wait_avg=0,turn_avg=0;
    HashMap<Character, Integer>wait = new HashMap<>();
        HashMap<Character, Integer>turn = new HashMap<>();
 System.out.println("enter time slice");
     int time_slice=sc.nextInt();
```

```
while(true)
{

  a=sc.nextInt();
  bu=sc.nextInt();

  if(a==-1 && bu==-1)
  break;
  else
  {

      n=sc.next().charAt(0);
      Vector s=new Vector(3);
      s.add(n);
      s.add(a);
      s.add(bu);
      wait.put(n,0);
      turn.put(n,bu);
      total++;
      b.add(s);

  }



}

  int timer=0,q_size,m1;
  int min=100000,ratio;

  char a1='z';
  boolean flag=true;
  Queue<Vector> q = new LinkedList<Vector>();
  q=check(b,timer,q);

  while(b.size()!=0){


    if(q.size()!=0)
    {
```

```
{
  Vector x=q.remove();
   a1=(char)x.get(0);

    for(int k1=0;k1<b.size();k1++)
    {
     Vector m=(Vector)b.get(k1);

     if(m.contains(a1))
     {
       if((int)m.get(2)<=time_slice)
       {
       System.out.print("from "+timer+" to ");
       timer=timer+(int)m.get(2);
       val=(int)turn.get(a1);
       t=timer-(int)m.get(1);
        w=t-val;
       System.out.print(timer+" process  "+a1+" will run");

      wait.put(a1,w);
      turn.put(a1,t);
      turn_avg=turn_avg+t;
       wait_avg=wait_avg+w;
       System.out.println();
       b.remove(m);
       q=check(b,timer,q);
       }
       else
       {
         System.out.print("from "+timer+" to ");
       timer=timer+time_slice;
       System.out.print(timer+" process  "+a1+" will run");
       System.out.println();
       m1=(int)m.get(2);
       m1=m1-time_slice;
       m.set(2,m1);

       q=check(b,timer,q);
       q.add(m);
       q.remove(m);

       }
```

```
                break;

              }

          }
          }
        }
        else
        {
          System.out.println("idle time");
          timer=timer+1;



        }
        min=100000;
      }
System.out.println("wait time of all process is ");
System.out.print(wait);
System.out.println();
System.out.println("turn time of all process is ");
System.out.print(turn);

System.out.println();
turn_avg=turn_avg/total;
   wait_avg=wait_avg/total;
System.out.println("avg t-time is "+turn_avg+" and avg w-time is
"+wait_avg);




  }
}
```

```
enter time slice
3
0 3 a
2 6 b
4 4 c
6 5 d
8 2 e
-1 -1 m
from 0 to 3 process  a will run
from 3 to 6 process  b will run
from 6 to 9 process  c will run
from 9 to 12 process  d will run
from 12 to 15 process  b will run
from 15 to 17 process  e will run
from 17 to 18 process  c will run
from 18 to 20 process  d will run
wait time of all process is
{a=0, b=7, c=10, d=9, e=7}
turn time of all process is
{a=3, b=13, c=14, d=14, e=9}
avg t-time is 10.6 and avg w-time is 6.6
```

Conclusion: therefore all premtive algorithms were sucessfully implemented and studied.

Post Lab Descriptive Questions (Add questions from examination point view)

1. Consider three processes, all arriving at time zero, with total execution time of 10, 20 and 30 units, respectively. Each process spends the first 20% of execution time doing I/O, the next 70% of time doing computation, and the last 10% of time doing I/O again. The operating system uses a shortest remaining compute time first scheduling algorithm and schedules a new process either when the running process gets blocked on I/O or when the running process finishes its compute burst. Assume that all I/O operations can be overlapped as much as possible. For what percentage of time does the CPU remain idle?

Total time spent = 47
Idle time = 2 + 3 = 5(first 2 secs process 1and last 3 secs of process 3)
Percentage of idle time = (5/47)*100 = 10.6 %

2. What is Starvation?

Starvation is the name given to the indefinite postponement of a process because it requires some resource before it can run, but the resource, though available for allocation, is never allocated to this process. It is sometimes called livelock, though sometimes that name might be reserved for cases where the waiting process is doing something, but nothing useful, as in a spin lock. However it happens, starvation is self-evidently a bad thing; more formally, it's bad because we don't want a non-functional system. If starvation is possible in a system, then a process which enters the system can be held up for an arbitrary length of time. To avoid starvation, it is often said that we want the system's resources to be shared "fairly". This is an appealing suggestion, but in practice it isn't much use because it doesn't define what we mean by "fairly", so it's really more of a description of the problem than a contribution to its solution. It's more constructive to investigate the paths by which a system can reach a livelocked state, and try to control them.

Date: _____ **Signature of faculty in-charge**