LAB#07

Implementation of CPU Scheduling Algorithm (i)Round Robin (ii)Priority based

(i) ROUND ROBIN ALGORITHM:

A round-robin **scheduling algorithm** is used to schedule the process fairly for each job a time slot or quantum and the interrupting the job if it is not completed by then the job come after the other job which is arrived in the quantum time that makes these scheduling fairly.

Note:

- Round-robin is cyclic in nature, so starvation doesn't occur
- Round-robin is a variant of first come, first served scheduling
- No priority, special importance is given to any process or task
- RR scheduling is also known as Time slicing scheduling

Advantages:

- Each process is served by CPU for a fixed time, so priority is the same for each one
- Starvation does not occur because of its cyclic nature.

Disadvantages:

- Throughput depends on quantum time.
- If the time quantum is too large RR degrades to FCFS.
- If we want to give some process priority, we cannot.

Process	Arrival Time	Burst Time	Completion time	Turn Around Time	Waiting time
P1	0	5	12	12	7
P2	1	4	11	10	6
P3	2	2	6	4	2
P4	3	1	9	6	5

Quantum time is 2 this means each process is only executing for 2 units of time at a time.

How to compute these process requests:-

1. Take the process which occurs first and start executing the process(for quantum time only).
2. Check if any other process request has arrived. If a process request arrives during the quantum time in which another process is executing, then add the new process to the Ready queue
3. After the quantum time has passed, check for any processes in the Ready queue. If the ready queue is empty then continue the current process. If the queue not empty

and the current process is not complete, then add the current process to the end of the ready queue.

4. Take the first process from the Ready queue and start executing it (same rules)
5. Repeat all steps above from 2-4
6. If the process is complete and the ready queue is empty then the task is complete

After all these we get the three times which are:

1. **Completion Time:** the time taken for a process to complete.
2. **Turn Around Time:** total time the process exists in the system. (completion time – arrival time).
3. **Waiting Time:** total time waiting for their complete execution. (turn around time – burst time).

How to implement in a programming language

```
1. Declare arrival[], burst[], wait[], turn[] arrays and initialize them. Also declare a timer
   variable and initialize it to zero. To sustain the original burst array create another
   array (temp_burst[]) and copy all the values of burst array in it.

2. To keep a check we create another array of bool type which keeps the record of whether a
   process is completed or not. we also need to maintain a queue array which contains the process
   indices (initially the array is filled with 0).

3. Now we increment the timer variable until the first process arrives and when it does, we add the
   process index to the queue array

4. Now we execute the first process until the time quanta and during that time quanta, we check
   whether any other process has arrived or not and if it has then we add the index in the queue
   (by calling the fxn. queueUpdation()).

5. Now, after doing the above steps if a process has finished, we store its exit time and
   execute the next process in the queue array. Else, we move the currently executed process at
   the end of the queue (by calling another fxn. queueMaintainence()) when the time slice expires.

6. The above steps are then repeated until all the processes have been completely executed. If a
   scenario arises where there are some processes left but they have
```

not arrived yet, then we shall wait and the CPU will remain idle during this interval.

PROGRAMMING CODE:

```
// C# program to implement Round Robin
// Scheduling with different arrival time
using System;

class GFG {
    public static void roundRobin(String[] p, int[] a,
                                  int[] b, int n)
    {
        // result of average times
        int res = 0;
        int resc = 0;

        // for sequence storage
        String seq = "";

        // copy the burst array and arrival array
        // for not effecting the actual array
        int[] res_b = new int[b.Length];
        int[] res_a = new int[a.Length];

        for (int i = 0; i < res_b.Length; i++) {
            res_b[i] = b[i];
            res_a[i] = a[i];
        }

        // critical time of system
        int t = 0;

        // for store the waiting time
        int[] w = new int[p.Length];

        // for store the Completion time
        int[] comp = new int[p.Length];

        while (true) {
            Boolean flag = true;
            for (int i = 0; i < p.Length; i++) {

                // these condition for if
                // arrival is not on zero
```

```
// check that if there come before qtime
if (res_a[i] <= t) {
    if (res_a[i] <= n) {
        if (res_b[i] > 0) {
            flag = false;
            if (res_b[i] > n) {

                // make decrease the b time
                t = t + n;
                res_b[i] = res_b[i] - n;
                res_a[i] = res_a[i] + n;
                seq += "->" + p[i];
            }
            else {

                // for last time
                t = t + res_b[i];

                // store comp time
                comp[i] = t - a[i];

                // store wait time
                w[i] = t - b[i] - a[i];
                res_b[i] = 0;

                // add sequence
                seq += "->" + p[i];
            }
        }
    }
    else if (res_a[i] > n) {

        // is any have less arrival time
        // the coming process then execute
        // them
        for (int j = 0; j < p.Length; j++) {

            // compare
            if (res_a[j] < res_a[i]) {
                if (res_b[j] > 0) {
                    flag = false;
                    if (res_b[j] > n) {
                        t = t + n;
                        res_b[j]
```

```
                            = res_b[j] - n;
                        res_a[j]
                            = res_a[j] + n;
                        seq += "->" + p[j];
                    }
                    else {
                        t = t + res_b[j];
                        comp[j] = t - a[j];
                        w[j] = t - b[j]
                               - a[j];
                        res_b[j] = 0;
                        seq += "->" + p[j];
                    }
                }
            }
        }

        // now the previous process
        // according to ith is process
        if (res_b[i] > 0) {
            flag = false;

            // Check for greaters
            if (res_b[i] > n) {
                t = t + n;
                res_b[i] = res_b[i] - n;
                res_a[i] = res_a[i] + n;
                seq += "->" + p[i];
            }
            else {
                t = t + res_b[i];
                comp[i] = t - a[i];
                w[i] = t - b[i] - a[i];
                res_b[i] = 0;
                seq += "->" + p[i];
            }
        }
    }
}

// if no process is come on the critical
else if (res_a[i] > t) {
    t++;
    i--;
}
```

```
            }

            // for exit the while loop
            if (flag) {
                break;
            }
        }

        Console.WriteLine("name   ctime   wtime");
        for (int i = 0; i < p.Length; i++) {
            Console.WriteLine(" " + p[i] + "\t" + comp[i]
                              + "\t" + w[i]);

            res = res + w[i];
            resc = resc + comp[i];
        }

        Console.WriteLine("Average waiting time is "
                          + (float)res / p.Length);
        Console.WriteLine("Average compilation time is "
                          + (float)resc / p.Length);
        Console.WriteLine("Sequence is like that " + seq);
    }

    // Driver Code
    public static void Main(String[] args)
    {
        // name of the process
        String[] name = { "p1", "p2", "p3", "p4" };

        // arrival for every process
        int[] arrivaltime = { 0, 1, 2, 3 };

        // burst time for every process
        int[] bursttime = { 10, 4, 5, 3 };

        // quantum time of each process
        int q = 3;

        // cal the function for output
        roundRobin(name, arrivaltime, bursttime, q);
    }
}
```

OUTPUT:

```
Name: Saad Hussain
RollNo:2022F-BIT-024
name    ctime    wtime
 p1      22       12
 p2      15       11
 p3      16       11
 p4      9        6
Average waiting time is 10
Average compilation time is 15.5
Sequence is like that ->p1->p2->p3->p4->p1->p2->p3->p1->p1
```

(B) A slightly optimized version of the above-implemented code could be done by using Queue data structure as follows:

```
using System;
using System.Collections.Generic;
using System.Linq;

class GFG
{
    // At every time quantum or when a process has been
    // executed before the time quantum, check for any new
    // arrivals and push them into the queue
    public static void
        checkForNewArrivals(Process[] processes, int n,
                            int currentTime,
                            Queue<int> readyQueue)
    {
        for (int i = 0; i < n; i++)
        {
            Process p = processes[i];
            // checking if any processes has arrived
            // if so, push them in the ready Queue.
            if (p.arrivalTime <= currentTime && !p.inQueue
                && !p.isComplete)
            {
                processes[i].inQueue = true;
                readyQueue.Enqueue(i);
            }
```

```
    }
}

// Context switching takes place at every time quantum
// At every iteration, the burst time of the processes
// in the queue are handled using this method
public static void
    updateQueue(Process[] processes, int n, int quantum,
                Queue<int> readyQueue, int currentTime,
                int programsExecuted)
{
    int i = readyQueue.Dequeue();

    // if the process is going to be finished executing,
    // ie, when it's remaining burst time is less than
    // time quantum mark it completed and increment the
    // current time and calculate its waiting time and
    // turnaround time
    if (processes[i].burstTimeRemaining <= quantum)
    {
        processes[i].isComplete = true;
        currentTime += processes[i].burstTimeRemaining;
        processes[i].completionTime = currentTime;
        processes[i].waitingTime
            = processes[i].completionTime
            - processes[i].arrivalTime
            - processes[i].burstTime;
        processes[i].turnaroundTime
            = processes[i].waitingTime
            + processes[i].burstTime;

        if (processes[i].waitingTime < 0)
            processes[i].waitingTime = 0;

        processes[i].burstTimeRemaining = 0;

        // if all the processes are not yet inserted in
        // the queue, then check for new arrivals
        if (programsExecuted != n)
        {
            checkForNewArrivals(
                processes, n, currentTime, readyQueue);
        }
    }
    else
```

```
    {
        // the process is not done yet. But it's going
        // to be pre-empted since one quantum is used
        // but first subtract the time the process used
        // so far
        processes[i].burstTimeRemaining -= quantum;
        currentTime += quantum;

        // if all the processes are not yet inserted in
        // the queue, then check for new arrivals
        if (programsExecuted != n)
        {
            checkForNewArrivals(
                processes, n, currentTime, readyQueue);
        }
        // insert the incomplete process back into the
        // queue
        readyQueue.Enqueue(i);
    }
}

// Just a function that outputs the result in terms of
// their PID.
public static void output(Process[] processes, int n)
{
    double avgWaitingTime = 0;
    double avgTurntaroundTime = 0;
    // sort the processes array by processes.PID
    processes = processes.OrderBy(p => p.pid).ToArray();

    for (int i = 0; i < n; i++)
    {
        Console.WriteLine("Process " + processes[i].pid
                          + ": Waiting Time: "
                          + processes[i].waitingTime
                          + " Turnaround Time: "
                          + processes[i].turnaroundTime);
        avgWaitingTime += processes[i].waitingTime;
        avgTurntaroundTime
            += processes[i].turnaroundTime;
    }
    Console.WriteLine("Average Waiting Time: "
                      + avgWaitingTime / n);
    Console.WriteLine("Average Turnaround Time: "
                      + avgTurntaroundTime / n);
```

```
}

/*
 * This function assumes that the processes are already
 * sorted according to their arrival time
 */
public static void roundRobin(Process[] processes,
                              int n, int quantum)
{
    Queue<int> readyQueue
        = new Queue<int>();
    readyQueue.Enqueue(0); // initially, pushing the first
    // process which arrived first
    processes[0].inQueue = true;

    int currentTime
        = 0; // holds the current time after each
    // process has been executed
    int programsExecuted
        = 0; // holds the number of programs executed so
    // far

    while (readyQueue.Count() > 0)
    {
        updateQueue(processes, n, quantum, readyQueue,
                    currentTime, programsExecuted);
    }
}

public class Process
{
    public int pid;
    public int arrivalTime;
    public int burstTime;
    public int burstTimeRemaining; // the amount of CPU time
    // remaining after each
    // execution
    public int completionTime;
    public int turnaroundTime;
    public int waitingTime;
    public bool isComplete;
    public bool inQueue;
}

public static void Main(String[] args)
```

```
    {
        int n, quantum;

        Console.WriteLine("Enter the number of processes: ");
        n = int.Parse(Console.ReadLine());
        Console.WriteLine("Enter time quantum: ");
        quantum
            = int.Parse(Console.ReadLine());

        Process[] processes = new Process[n + 1];

        for (int i = 0; i < n; i++)
        {
            Console.WriteLine("Enter arrival time and burst time of each
process "
                              + (i + 1) + ": ");
            processes[i].arrivalTime = int.Parse(
                Console.ReadLine());
            processes[i].burstTime = int.Parse(
                Console.ReadLine());
            processes[i].burstTimeRemaining
                = processes[i].burstTime;
            processes[i].pid = i + 1;
            Console.WriteLine();
        }

        // stl sort in terms of
      processes.OrderBy(p => p.arrivalTime).ToArray();
      roundRobin(processes, n, quantum);

    output(processes, n);
  }
}
```

OUTPUT:

```
Enter the arrival time and burst time of each process:
0 5
1 4
2 2
3 1Enter the number of processes: 4

Enter time quantum: 2



Process 1: Waiting Time: 7 Turnaround Time: 12
Process 2: Waiting Time: 6 Turnaround Time: 10
Process 3: Waiting Time: 2 Turnaround Time: 4
Process 4: Waiting Time: 5 Turnaround Time: 6

Average Waiting Time: 5
Average Turnaround Time: 8
```

Priority Based Algorithm:

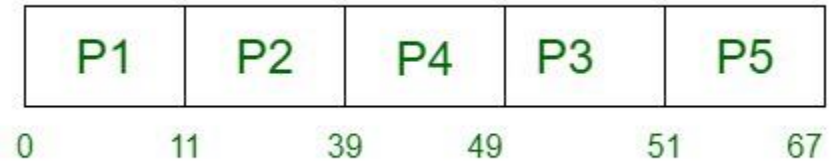
Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems. Each process is assigned first arrival time (less arrival time process first) if two processes have same arrival time, then compare to priorities (highest process first). Also, if two processes have same priority then compare to process number (less process number first). This process is repeated while all process get executed.

Implementation –

1. First input the processes with their arrival time, burst time and priority.
2. First process will schedule, which have the lowest arrival time, if two or more processes will have lowest arrival time, then whoever has higher priority will schedule first.
3. Now further processes will be schedule according to the arrival time and priority of the process. (Here we are assuming that lower the priority number having higher priority). If two process priority are same then sort according to process number. **Note:** In the question, They will clearly mention, which number will have higher priority and which number will have lower priority.
4. Once all the processes have been arrived, we can schedule them based on their priority.

Process	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

Gantt Chart –



C# Programming Code:

```
using System;

class Program
{
    static int totalprocess = 5;
    static int[][] proc = new int[totalprocess][];
    static int[] arrivaltime = new int[] {1, 2, 3, 4, 5};
    static int[] bursttime = new int[] {3, 5, 1, 7, 4};
    static int[] priority = new int[] {3, 4, 1, 7, 8};


    // Driver code
    static void Main(string[] args)
    {
        for (int i = 0; i < totalprocess; i++)
        {
            proc[i] = new int[4];
            proc[i][0] = arrivaltime[i];
```

```
            proc[i][1] = bursttime[i];
            proc[i][2] = priority[i];
            proc[i][3] = i + 1;
        }

        Array.Sort(proc, (x, y) => x[2].CompareTo(y[2]));
        Array.Sort(proc, (x, y) => x[0].CompareTo(y[0]));
        Findgc();
    }

    // Using FCFS Algorithm to find Waiting time
    static void GetWtTime(int[] wt)
    {

        // declaring service array that stores
        // cumulative burst time
        int[] service = new int[totalprocess];

        // Initialising initial elements
        // of the arrays
        service[0] = 0;
        wt[0] = 0;

        for (int i = 1; i < totalprocess; i++)
        {
            service[i] = proc[i - 1][1] + service[i - 1];
            wt[i] = service[i] - proc[i][0] + 1;

            // If waiting time is negative,
            // change it o zero
            if (wt[i] < 0)
            {
                wt[i] = 0;
            }
        }
    }

    // Filling turnaroundtime array
    static void GetTatTime(int[] tat, int[] wt)
    {
        for (int i = 0; i < totalprocess; i++)
        {
            tat[i] = proc[i][1] + wt[i];
        }
    }

    static void Findgc()
    {
```

```
        // Declare waiting time and
        // turnaround time array
        int[] wt = new int[totalprocess];
        int[] tat = new int[totalprocess];
        int wavg = 0;
        int tavg = 0;

         // Function call to find waiting time array
        GetWtTime(wt);

        // Function call to find turnaround time
        GetTatTime(tat, wt);
        int[] stime = new int[totalprocess];
        int[] ctime = new int[totalprocess];
        stime[0] = 1;
        ctime[0] = stime[0] + tat[0];


Console.WriteLine("Process_no\tStart_time\tComplete_time\tTurn_Around_Time\tWai
ting_Time");

        // calculating starting and ending time
        for (int i = 0; i < totalprocess; i++)
        {
            wavg += wt[i];
            tavg += tat[i];
            Console.WriteLine(proc[i][3] + "\t\t" + stime[i] + "\t\t" +
ctime[i] + "\t\t" + tat[i] + "\t\t\t" + wt[i]);


            // display the process details
            if (i != totalprocess - 1)
            {
                stime[i + 1] = ctime[i];
                ctime[i + 1] = stime[i + 1] + tat[i + 1] - wt[i + 1];
            }
        }

        // display the average waiting time
        // and average turn around time
        Console.WriteLine("Average waiting time is: " + (double)wavg /
totalprocess);
        Console.WriteLine("Average turnaround time is: " + (double)tavg /
totalprocess);
    }
}
```

OUTPUT:

```
Process_no Start_time Complete_time Turn_Around_Time Waiting_Time
1           1          4             3               0
2           5          10            8               3
3           4          5             2               1
4          10          17            13              6
5          17          21            16              12
Average Waiting Time is : 4.4
Average Turn Around time is : 8.4
```

Lab tasks:

1. Perform the above code with 8 Number of processes by randomly setting the arrival time and burst time and priority.

(a) Use Round Robin Algorithm

(b) Use Priority Based Algorithm

2. Perform the above code with 10 Number of processes by randomly setting the arrival time and burst time and priority.

(a) Use Round Robin Algorithm

(b) Use Priority Based Algorithm