End Term Project Report

On

Design of Operating System (CSE 4049)

Submitted by

Name	**: JYOTISWINI NAYAK**
Reg. No.	**: 2141014068**
Semester	**: 5th**
Section	**: K**
Session	**: 2023-2024**
Admission Batch	**: 2021**

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FACULTY OF ENGINEERING & TECHNOLOGY (ITER)

SIKSHA 'O' ANUSANDHAN DEEMED TO BE UNIVERSITY

BHUBANESWAR, ODISHA – 751030

Project Description 1: The Java program provides an interface to the user to implement the following scheduling policies as per the choice provided:

1. First Come First Served (FCFS)
2. Round Robin (RR)

Appropriate option needs to be chosen from a switch case based menu driven program with an option of "Exit from program" in case 5 and accordingly a scheduling policy will print the Gantt chart and the average waiting time, average turnaround time and average response time. The program will take Process ids, its arrival time, and its CPU burst time as input. For implementing RR scheduling, user also needs to specify the time quantum. Assume that the process ids should be unique for all processes. Each process consists of a single CPU burst (no I/O bursts), and processes are listed in order of their arrival time. Further assume that an interrupted process gets placed at the back of the Ready queue, and a newly arrived process gets placed at the back of the Ready queue as well. The output should be displayed in a formatted way for clarity of understanding and visual.

Test Cases: The program should able to produce correct answer or appropriate error message corresponding to the following test cases:

1. Consider the set of processes with arrival time (in milliseconds), CPU burst time (in milliseconds), and time quantum = 4ms as shown below.

Process	Arrival time	Burst Time
P1	0	10
P2	1	1
P3	2	2
P4	3	1
P5	6	5

- Input choice 1, and print the Gantt charts that illustrate the execution of these processes using the FCFS scheduling algorithm and then print the average turnaround time, average waiting time and average response time.
- Input choice 2, and print the Gantt charts that illustrate the execution of these processes using the RR scheduling algorithm and then print the average turnaround time, average waiting time and average response time.

• Analyze the results and determine which of the algorithms results in the minimum average waiting time over all processes?

Code:

```
package dos;
import java.util.Scanner;

public class dos_project1 {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        boolean fcfsChosen = false;

        while (true) {
            System.out.println();
            System.out.println("Choose the scheduling algorithm:");
            System.out.println("1. First-Come, First-Served (FCFS)");
            System.out.println("2. Round Robin (RR)");
            System.out.println("3. Terminate Program");
            int choice = sc.nextInt();

            switch (choice) {
                case 1:
                    if (!fcfsChosen) {
                        fcfsAlgorithm(sc);
                        fcfsChosen = true;
                    } else {
                        rrAlgorithm(sc);
                    }
                    break;
                case 2:
                    rrAlgorithm(sc);
                    break;
                case 3:
                    System.out.println("Terminating the program...");
                    System.exit(0);
                    break;

               default:
                    System.out.println("Invalid choice!");
            }
        }
    }
```

```
    private static void fcfsAlgorithm(Scanner sc) {
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();
        int burstTimes[] = new int[n];
        int arrivalTimes[] = new int[n];

        System.out.println("\nEnter the Burst Time for each process.");
        for (int i = 0; i < n; i++) {
            System.out.print("\nFor Process " + (i + 1) + ": ");
            burstTimes[i] = sc.nextInt();
        }

        System.out.println("\nEnter the arrival time for each process.");
        for (int j = 0; j < n; j++) {
            System.out.print("\nFor Process " + (j + 1) + ": ");
            arrivalTimes[j] = sc.nextInt();
        }
        calculateAndDisplayTimes(n, burstTimes, arrivalTimes);
    }

    private static void rrAlgorithm(Scanner sc) {
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();

        int processes[] = new int[n];
        int burstTimes[] = new int[n];
        int arrivalTimes[] = new int[n];

        for (int i = 0; i < n; i++) {
            System.out.print("Enter burst time for process " + (i + 1) + ": ");
            burstTimes[i] = sc.nextInt();
            processes[i] = i + 1;
        }
     for (int i = 0; i < n; i++) {
           System.out.print("Enter arrival time for process " + (i + 1) + ":");
            arrivalTimes[i] = sc.nextInt();
        }

        System.out.print("Enter the time quantum: ");
        int quantum = sc.nextInt();

        findAvgTime(processes, n, burstTimes, quantum, arrivalTimes);
    }
        private static void calculateAndDisplayTimes(int n, int[]
burstTimes, int[] arrivalTimes) {
         int wt[] = new int[n];
         int rt[] = new int[n];
         int ct[] = new int[n];
```

```
wt[0] = 0;
        ct[0] = burstTimes[0];

        for (int i = 1; i < n; i++) {
            wt[i] = ct[i - 1] - arrivalTimes[i];
            if (wt[i] < 0) {
                wt[i] = 0;
            }
            rt[i] = wt[i];
            ct[i] = ct[i - 1] + burstTimes[i];
        }

        System.out.println("\nProcesses || Burst Time || Arrival Time
|| Waiting Time || Response Time || Completion Time ");

        float awt = 0;
        float art = 0;
        float att = 0;

        for (int i = 0; i < n; i++) {
            System.out.println((i + 1) + "\t ||\t" + burstTimes[i] +
"\t||\t" + arrivalTimes[i] + "\t||\t" + wt[i] + "\t||\t "
                    + rt[i] + "\t||\t " + ct[i]);
            awt += wt[i];
            art += rt[i];
            att += (ct[i] - arrivalTimes[i]);
        }
        awt = awt / n;
        art = art / n;
        att = att / n;
        System.out.println("\nAverage waiting time = " + awt);
        System.out.println("\nAverage response time = " + art);
        System.out.println("\nAverage turnaround time = " + att);
    }

    static void findAvgTime(int processes[], int n, int burstTimes[],
int quantum, int arrivalTimes[]) {
        int wt[] = new int[n], tat[] = new int[n], ct[] = new int[n],
rt[] = new int[n];
        double total_wt = 0, total_tat = 0, total_rt = 0;

  findWaitingTime(processes, n, burstTimes, wt, quantum,
arrivalTimes, ct, rt);
        findTurnAroundTime(processes, n, burstTimes, wt, tat, ct,
arrivalTimes);

        System.out.println("Processes " + " Burst time " + "
Waiting time " + " Turnaround time " + " Response time");

        for (int i = 0; i < n; i++) {
            total_wt += wt[i];
            total_tat += tat[i];
```

```
total_rt += rt[i];
            System.out.println(" " + processes[i] + "\t\t" +
burstTimes[i] + "\t " + wt[i] + "\t\t " + tat[i] + "\t\t " + rt[i]);
        }

        System.out.println("Average waiting time = " + total_wt /
n);
        System.out.println("Average turnaround time = " + total_tat
/ n);
        System.out.println("Average response time = " + total_rt /
n);

        // Compare the efficiency of algorithms based on average
waiting time
        compareAlgorithmsEfficiency(total_wt / n,
calculateFCFSAvgWaitingTime(burstTimes, arrivalTimes));

    }

    static void findWaitingTime(int processes[], int n, int
burstTimes[], int wt[], int quantum, int arrivalTimes[], int ct[],
int rt[]) {
        int rem_bt[] = new int[n];
        for (int i = 0; i < n; i++)
            rem_bt[i] = burstTimes[i];

        int t = 0;
        boolean visited[] = new boolean[n];

        while (true) {
            boolean done = true;

            for (int i = 0; i < n; i++) {
                if (rem_bt[i] > 0 && arrivalTimes[i] <= t) {
                    done = false;

                    if (!visited[i]) {
                        rt[i] = t - arrivalTimes[i];
                        visited[i] = true;
                    }

                    if (rem_bt[i] > quantum) {
                        t += quantum;
                        rem_bt[i] -= quantum;
                    } else {
                        t += rem_bt[i];
                        wt[i] = t - burstTimes[i] - arrivalTimes[i];
                        rem_bt[i] = 0;
                        ct[i] = t;
                    }}}
```

```
  if (done)
                break;
        }
    }

    static void findTurnAroundTime(int processes[], int n, int
burstTimes[], int wt[], int tat[], int ct[], int arrivalTimes[]) {
        for (int i = 0; i < n; i++)
            tat[i] = ct[i] - arrivalTimes[i];
    }

    static double calculateFCFSAvgWaitingTime(int[] burstTimes,
int[] arrivalTimes) {
        int n = burstTimes.length;
        int wt[] = new int[n];
        int ct[] = new int[n];

        int prevCT = 0;
        for (int i = 0; i < n; i++) {
            wt[i] = prevCT - arrivalTimes[i];
            if (wt[i] < 0) {
                wt[i] = 0;
            }
            ct[i] = prevCT + burstTimes[i];
            prevCT = ct[i];
        }

        double total_wt = 0;
        for (int i = 0; i < n; i++) {
            total_wt += wt[i];
        }
        return total_wt / n;
    }

    static void compareAlgorithmsEfficiency(double avgWaitingTimeRR,
double avgWaitingTimeFCFS) {
        if (avgWaitingTimeRR < avgWaitingTimeFCFS) {
            System.out.println("Round Robin (RR) algorithm results
in the minimum average waiting time = "+avgWaitingTimeRR);
        } else if (avgWaitingTimeRR > avgWaitingTimeFCFS) {
            System.out.println("FCFS algorithm results in the
minimum average waiting time = "+avgWaitingTimeFCFS);
        } else {
            System.out.println("Both algorithms have the same
average waiting time.");
        }
    }
}
```

Output:

```
Choose the scheduling algorithm:
1. First-Come, First-Served (FCFS)
2. Round Robin (RR)
3. Terminate Program
1 //It is for First Come First -Served (FCFS)Enter
the number of processes: 5
Enter the Burst Time for each process.
For Process 1: 10
For Process 2: 1
For Process 3: 2
For Process 4: 1
For Process 5: 5
Enter the arrival time for each process.
For Process 1: 0
For Process 2: 1
For Process 3: 2
For Process 4: 3
For Process 5: 6
Processes || Burst Time || Arrival Time || Waiting Time || Response Time || Completion Time
1       ||  10      ||  0       ||  0       ||  0       ||  10
2       ||  1       ||  1       ||  9       ||  9       ||  11
3       ||  2       ||  2       ||  9       ||  9       ||  13
4       ||  1       ||  3       ||  10      ||  10      ||  14
5       ||  5       ||  6       ||  8       ||  8       ||  19
Average waiting time = 7.2
Average response time = 7.2
Avrage Turn Around =11.0
```

Choose the scheduling algorithm:

1. First-Come, First-Served (FCFS)

2. Round Robin (RR)

3. Terminate Program

2

Enter the number of processes: 5

Enter burst time for process 1: 10

Enter burst time for process 2: 1

Enter burst time for process 3: 2

Enter burst time for process 4: 1

Enter burst time for process 5: 5

Enter arrival time for process 1: 0

Enter arrival time for process 2: 1

Enter arrival time for process 3: 2

Enter arrival time for process 4: 3

Enter arrival time for process 5: 6

Enter the time quantum: 4

Processes	Burst time	Waiting time	Turnaround time	Response time
1	10	9	19	0
2	1	3	4	3
3	2	3	5	3
4	1	4	5	4
5	5	6	11	2

Average waiting time = 5.2

Average turnaround time = 8.8

Average response time = 2.4

Round Robin (RR) algorithm results in the minimum average waiting time = 5.2

Project Description 2:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Example: Snapshot at the initial stage:

1. Consider the following resource allocation state with 5 processes and 4 resources: There are total existing resources of 6 instances of type R1, 7 instances of type R2, 12 instance of type R3 and 12 instances of type R4.

Process	Allocation				Max			
	R1	R2	R3	R4	R1	R2	R3	R4
P_1	0	0	1	2	0	0	1	2
P_2	2	0	0	0	2	7	5	0
P_3	0	0	3	4	6	6	5	6
P_4	2	3	5	4	4	3	5	6
P_5	0	3	3	2	0	6	5	2

a) Find the content of the need matrix.

b) Is the system in a safe state? If so, give a safe sequence of the process.

c) If P3 will request for 1 more instances of type R2, Can the request be granted immediately or not

Code:

```
package dos;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

public class BankersAlgorithm {
    int[][]  max;
    int[][] allocation;
    int[][] need;
```

```
    int[]  available;
    int numOfProcesses;
    int numOfResources;

    public BankersAlgorithm(int[][] max, int[][] allocation, int[]
available) {
        this.max = max;
        this.allocation = allocation;
        this.available = available;
        numOfProcesses = max.length;
        numOfResources = available.length;
        need = new int[numOfProcesses][numOfResources];

        // Calculate the need matrix
        for (int i = 0; i < numOfProcesses; i++) {
            for (int j = 0; j < numOfResources; j++) {
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }

    public boolean isSafeState() {
        int[] work = Arrays.copyOf(available, numOfResources);
        boolean[] finish = new boolean[numOfProcesses];

        int count = 0;
        while (count < numOfProcesses) {
            boolean found = false;
            for (int i = 0; i < numOfProcesses; i++) {
                if (!finish[i]) {
                    boolean canAllocate = true;
                    for (int j = 0; j < numOfResources; j++) {
                        if (need[i][j] > work[j]) {
                            canAllocate = false;
                            break;
                        }
                    }

                    if (canAllocate) {
                        for (int j = 0; j < numOfResources; j++) {
                            work[j] += allocation[i][j];
                        }
                        finish[i] = true;
                        count++;
                        found = true;
                    }
                }
            }
```

```
if (!found) {
                    break;
                }
            }

            return count == numOfProcesses;
        }

        public int requestResources(int processNum, int[] request) {
            for (int i = 0; i < numOfResources; i++) {
                if (request[i] > available[i] || request[i] >
need[processNum][i]) {
                    return -1;
                }
            }

            for (int i = 0; i < numOfResources; i++) {
                available[i] -= request[i];
                allocation[processNum][i] += request[i];
                need[processNum][i] -= request[i];
            }

            if (!isSafeState()) {

                for (int i = 0; i < numOfResources; i++) {
                    available[i] += request[i];
                    allocation[processNum][i] -= request[i];
                    need[processNum][i] += request[i];
                }
                return 0;
            }

            return 1;
        }

        public ArrayList<Integer> getSafeSequence() {
            int[] work = Arrays.copyOf(available, numOfResources);
            boolean[] finish = new boolean[numOfProcesses];
            ArrayList<Integer> safeSeq = new ArrayList<>();

            for (int k = 0; k < numOfProcesses; k++) {
                for (int i = 0; i < numOfProcesses; i++) {
                    if (!finish[i]) {
                        boolean canAllocate = true;
                        for (int j = 0; j < numOfResources; j++) {
                            if (need[i][j] > work[j]) {
                                canAllocate = false;
                                break;
                            }}
```

```
if (canAllocate) {
                        for (int j = 0; j < numOfResources; j++) {
                            work[j] += allocation[i][j];
                        }
                        safeSeq.add(i);
                        finish[i] = true;
                    }
                }
            }
        }

        if (safeSeq.size() != numOfProcesses) {
            return null;
        }

        return safeSeq;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of processes: ");
        int numOfProcesses = scanner.nextInt();

        System.out.print("Enter the number of resources: ");
        int numOfResources = scanner.nextInt();

        int[][] max  = new  int[numOfProcesses][numOfResources];
        int[][] allocation = new int[numOfProcesses][numOfResources];
        int[] available = new int[numOfResources];

        System.out.println("Enter the Max matrix:");
        for (int i = 0; i < numOfProcesses; i++) {
            for (int j = 0; j < numOfResources; j++) {
                max[i][j] = scanner.nextInt();
            }
        }

        System.out.println("Enter the Allocation matrix:");
        for (int i = 0; i < numOfProcesses; i++) {
            for (int j = 0; j < numOfResources; j++) {
                allocation[i][j] = scanner.nextInt();
            }
        }

        System.out.println("Enter the Available resources:");
        for (int i = 0; i < numOfResources; i++) {
            available[i] = scanner.nextInt();
        }
```

```
  System.out.println("Enter the Available resources:");
        for (int i = 0; i < numOfResources; i++) {
            available[i] = scanner.nextInt();
        }

        BankersAlgorithm banker = new BankersAlgorithm(max,
allocation, available);
        System.out.println("Need Matrix:");
        for (int i = 0; i < banker.numOfProcesses; i++) {
            System.out.println(Arrays.toString(banker.need[i]));
        }
        ArrayList<Integer> safeSequence = banker.getSafeSequence();

        if (safeSequence != null) {
            System.out.println("The system is in a safe state.");
            System.out.println("Safe sequence: " + safeSequence);
        } else {
            System.out.println("The system is not in a safe
state.");
        }
        System.out.println("Enter the number of instances of type R2
that P3 wants to request:");
        int requestR2 = scanner.nextInt();

        int processNum = 2;
        int[] request = {0, requestR2, 0, 0};

        int result = banker.requestResources(processNum, request);

        if (result == 1) {
            System.out.println("Request can be granted
immediately.");
        } else if (result == 0) {
            System.out.println("Request denied as it leads to an
unsafe state.");
        } else {
            System.out.println("Requested resources exceed available
or need.");
        }

        scanner.close();
    }
}
```

Output

```
Enter the number of processes: 5
Enter the number of resources: 4
Enter the Max matrix:
0 0 1 2
2 7 5 0
6 6 5 6
4 3 5 6
0 6 5 2
Enter the Allocation matrix:
0 0 1 2
2 0 0 0
0 0 3 4
2 3 5 4
0 3 3 2
Enter the Available resources:
2 1 0 0
Need Matrix:
[0, 0, 0, 0]
[0, 7, 5, 0]
[6, 6, 2, 2]
[2, 0, 0, 2]
[0, 3, 2, 0]
The system is in a safe state.
Safe sequence: [0, 3, 4, 1, 2]
Enter the number of instances of type R2 that P3 wants to request:
0 1 0 0
Request can be granted immediately.
```

For Reference GitHub: ()