End Term Project Report

On

Design of Operating System (CSE 4049)

Submitted by:

Name	**: VIKAS MENARIA**
Reg. No.	**: 2141013086**
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 DosProject1 {

    public static void main(String[] args) {
        Scanner inputScanner = new Scanner(System.in);
        boolean isFirstComeFirstServeChosen = 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 userChoice = inputScanner.nextInt();

            switch (userChoice) {
                case 1:
                    if (!isFirstComeFirstServeChosen) {
                        firstComeFirstServeAlgorithm(inputScanner);
                        isFirstComeFirstServeChosen = true;
                    } else {
                        roundRobinAlgorithm(inputScanner);
                    }
                    break;
                case 2:
                    roundRobinAlgorithm(inputScanner);
                    break;
                case 3:
                    System.out.println("Terminating the program...");
                    System.exit(0);
                    break;
                default:
                    System.out.println("Invalid choice!");
            }
        }
    }
}
```

```
private static void firstComeFirstServeAlgorithm(Scanner inputScanner) {
    System.out.print("Enter the number of processes: ");
    int processCount = inputScanner.nextInt();

    int burstDurations[] = new int[processCount];
    int arrivalMoments[] = new int[processCount];

    System.out.println("\nEnter the Burst Time for each process.");
    for (int index = 0; index < processCount; index++) {
        System.out.print("\nFor Process " + (index + 1) + ": ");
        burstDurations[index] = inputScanner.nextInt();
    }

    System.out.println("\nEnter the arrival time for each process.");
    for (int index = 0; index < processCount; index++) {
        System.out.print("\nFor Process " + (index + 1) + ": ");
        arrivalMoments[index] = inputScanner.nextInt();
    }
    calculateAndDisplayTimes(processCount, burstDurations, arrivalMoments);
}

private static void roundRobinAlgorithm(Scanner inputScanner) {
    System.out.print("Enter the number of processes: ");
    int processCount = inputScanner.nextInt();

    int processIds[] = new int[processCount];
    int burstDurations[] = new int[processCount];
    int arrivalMoments[] = new int[processCount];

    for (int index = 0; index < processCount; index++) {
        System.out.print("Enter burst time for process " + (index + 1) + ": ");
        burstDurations[index] = inputScanner.nextInt();
        processIds[index] = index + 1;
    }
    for (int index = 0; index < processCount; index++) {
        System.out.print("Enter arrival time for process " + (index + 1) + ":");
        arrivalMoments[index] = inputScanner.nextInt();
    }

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

    findAverageTime(processIds,   processCount,   burstDurations,   timeQuantum,
arrivalMoments);
}

private   static   void   calculateAndDisplayTimes(int   processCount,   int[]
burstDurations, int[] arrivalMoments) {
    int waitingTimes[] = new int[processCount];
    int responseTimes[] = new int[processCount];
    int        completionTimes[]        =        new        int[processCount];
```

```
waitingTimes[0] = 0;
completionTimes[0] = burstDurations[0];

for (int index = 1; index < processCount; index++) {
    waitingTimes[index] = completionTimes[index - 1] - arrivalMoments[index];
    if (waitingTimes[index] < 0) {
        waitingTimes[index] = 0;
    }
    responseTimes[index] = waitingTimes[index];
    completionTimes[index]     =     completionTimes[index     -     1]     +
burstDurations[index];
}
System.out.println("\nProcesses || Burst Time || Arrival Time || Waiting Time
|| Response Time || Completion Time ");

float averageWaitingTime = 0;
float averageResponseTime = 0;
float averageTurnaroundTime = 0;

for (int index = 0; index < processCount; index++) {
    System.out.println((index  +  1)  +  "\t  ||\t"  +  burstDurations[index]  +
"\t||\t" + arrivalMoments[index] + "\t||\t" + waitingTimes[index] + "\t||\t "
    + responseTimes[index] + "\t||\t " + completionTimes[index]);
    averageWaitingTime += waitingTimes[index];
    averageResponseTime += responseTimes[index];
    averageTurnaroundTime         +=         (completionTimes[index]         -
arrivalMoments[index]);
}
averageWaitingTime = averageWaitingTime / processCount;
averageResponseTime = averageResponseTime / processCount;
averageTurnaroundTime = averageTurnaroundTime / processCount;
System.out.println("\nAverage waiting time = " + averageWaitingTime);
System.out.println("\nAverage response time = " + averageResponseTime);
System.out.println("\nAverage turnaround time = " + averageTurnaroundTime);
}

static   void   findAverageTime(int   processIds[],   int   processCount,   int
burstDurations[],
int timeQuantum, int arrivalMoments[]) {
int   waitingTimes[]   =   new   int[processCount],   turnaroundTimes[]   =   new
int[processCount], completionTimes[] = new int[processCount], responseTimes[]
= new int[processCount];
double totalWaitingTime = 0, totalTurnaroundTime = 0, totalResponseTime = 0;

findWaitingTime(processIds,   processCount,   burstDurations,   waitingTimes,
timeQuantum, arrivalMoments, completionTimes, responseTimes);
findTurnAroundTime(processIds,  processCount,  burstDurations,  waitingTimes,
turnaroundTimes, completionTimes, arrivalMoments);

System.out.println("Processes  "  +  "  Burst  time  "  +  "  Waiting  time  "  +  "
Turnaround time " + " Response time");
for (int index = 0; index < processCount; index++) {
    totalWaitingTime += waitingTimes[index];
    totalTurnaroundTime              +=              turnaroundTimes[index];
```

```
totalResponseTime += responseTimes[index];
System.out.println("  " + processIds[index] + "\t\t" + burstDurations[index] + "\t  " +
waitingTimes[index] + "\t\t " + turnaroundTimes[index] + "\t\t " + responseTimes[index]);
}

System.out.println("Average waiting time = " + totalWaitingTime / processCount);
System.out.println("Average turnaround time = " + totalTurnaroundTime / processCount);
System.out.println("Average response time = " + totalResponseTime / processCount);

// Compare the efficiency of algorithms based on average waiting time
compareAlgorithmsEfficiency(totalWaitingTime                /                processCount,
calculateFirstComeFirstServeAverageWaitingTime(burstDurations, arrivalMoments));

}

static void findWaitingTime(int processIds[], int processCount, int burstDurations[], int
waitingTimes[],  int  timeQuantum,  int  arrivalMoments[],  int  completionTimes[],  int
responseTimes[]) {
int remainingBurstTimes[] = new int[processCount];
for (int index = 0; index < processCount; index++)
    remainingBurstTimes[index] = burstDurations[index];

int time = 0;
boolean isVisited[] = new boolean[processCount];

while (true) {
    boolean isDone = true;

    for (int index = 0; index < processCount; index++) {
        if (remainingBurstTimes[index] > 0 && arrivalMoments[index] <= time) {
            isDone = false;

            if (!isVisited[index]) {
                responseTimes[index] = time - arrivalMoments[index];
                isVisited[index] = true;
            }

            if (remainingBurstTimes[index] > timeQuantum) {
                time += timeQuantum;
                remainingBurstTimes[index] -= timeQuantum;
            } else {
                time += remainingBurstTimes[index];
                waitingTimes[index]     =     time     -     burstDurations[index]     -
arrivalMoments[index];
                remainingBurstTimes[index] = 0;
                completionTimes[index] = time;
            }
        }
    }
```

```
if (isDone)
            break;
    }

    static void findTurnaroundTime(int processIds[], int processCount, int
    burstDurations[], int waitingTimes[], int turnaroundTimes[], int
    completionTimes[], int arrivalMoments[]) {
        for (int index = 0; index < processCount; index++)
            turnaroundTimes[index] = completionTimes[index] - arrivalMoments[index];
    }

    static double calculateFirstComeFirstServeAverageWaitingTime(int[] burstDurations,
    int[] arrivalMoments) {
        int processCount = burstDurations.length;
        int waitingTimes[] = new int[processCount];
        int completionTimes[] = new int[processCount];

        int previousCompletionTime = 0;
        for (int index = 0; index < processCount; index++) {
            waitingTimes[index] = previousCompletionTime - arrivalMoments[index];
            if (waitingTimes[index] < 0) {
                waitingTimes[index] = 0;
            }
            completionTimes[index] = previousCompletionTime + burstDurations[index];
            previousCompletionTime = completionTimes[index];
        }

        double totalWaitingTime = 0;
        for (int index = 0; index < processCount; index++) {
            totalWaitingTime += waitingTimes[index];
        }
        return totalWaitingTime / processCount;
    }

    static void compareAlgorithmsEfficiency(double averageWaitingTimeRoundRobin,
    double averageWaitingTimeFirstComeFirstServe) {
        if (averageWaitingTimeRoundRobin < averageWaitingTimeFirstComeFirstServe) {
            System.out.println("Round Robin (RR) algorithm results in the minimum
    average waiting time = "+averageWaitingTimeRoundRobin);
        } else if (averageWaitingTimeRoundRobin >
    averageWaitingTimeFirstComeFirstServe) {
            System.out.println("FCFS algorithm results in the minimum average waiting
    time = "+averageWaitingTimeFirstComeFirstServe);
        } 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 TurnAround =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**
P1	0	0	1	2	0	0	1	2
P2	2	0	0	0	2	7	5	0
P3	0	0	3	4	6	6	5	6
P4	2	3	5	4	4	3	5	6
P5	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
ResourceAllocationAlgorith
{

int[][]maximumResources;

int[][]allocatedResources;
int[] availableResources;
    int processCount;
    int resourceCount;

    public ResourceAllocationAlgorithm(int[][] maximumResources, int[][] allocatedResources, int[]
availableResources) {
        this.maximumResources = maximumResources;
        this.allocatedResources = allocatedResources;
        this.availableResources = availableResources;
        processCount = maximumResources.length;
        resourceCount = availableResources.length;
        requiredResources = new int[processCount][resourceCount];

        // Calculate the need matrix
        for (int i = 0; i < processCount; i++) {
            for (int j = 0; j < resourceCount; j++) {
                requiredResources[i][j] = maximumResources[i][j] - allocatedResources[i][j];
            }
        }
    }

    public boolean isSafeState() {
        int[] work = Arrays.copyOf(availableResources, resourceCount);
        boolean[] finish = new boolean[processCount];

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

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

            if (found)
                continue;
            else
                break;
        }
        return count == processCount;
    }
}
```

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

return count == processCount;
}

public int requestResources(int processIndex, int[] resourceRequest) {
    for (int i = 0; i < resourceCount; i++) {
        if (resourceRequest[i] > availableResources[i] || resourceRequest[i] > requiredResources[processIndex][i]) {
            return -1;
        }
    }

    for (int i = 0; i < resourceCount; i++) {
        availableResources[i] -= resourceRequest[i];
        allocatedResources[processIndex][i] += resourceRequest[i];
        requiredResources[processIndex][i] -= resourceRequest[i];
    }

    if (!isSafeState()) {

        for (int i = 0; i < resourceCount; i++) {
            availableResources[i] += resourceRequest[i];
            allocatedResources[processIndex][i] -= resourceRequest[i];
            requiredResources[processIndex][i] += resourceRequest[i];
        }
        return 0;
    }

    return 1;
}

public ArrayList<Integer> getSafeSequence() {
    int[] work = Arrays.copyOf(availableResources, resourceCount);
    boolean[] isFinished = new boolean[processCount];
    ArrayList<Integer> safeSequence = new ArrayList<>();

    for (int k = 0; k < processCount; k++) {
        for (int i = 0; i < processCount; i++) {
            if (!isFinished[i]) {
                boolean canAllocate = true;
                for (int j = 0; j < resourceCount; j++) {
                    if (requiredResources[i][j] > work[j]) {
                        canAllocate = false;
                        break;
                    }
                }

if (canAllocate) {
    for (int j = 0; j < resourceCount; j++) {
        work[j] += allocatedResources[i][j];
    }
    isFinished[i] = true;
    safeSequence.add(i);
}
}
}
return safeSequence;
}
}
```

```
if (canAllocate) {
    for (int j = 0; j < resourceCount; j++) {
        work[j] += allocatedResources[i][j];
    }
    safeSequence.add(i);
    isFinished[i] = true;
}
}
}
}

if (safeSequence.size() != processCount) {
    return null;
}

return safeSequence;
}

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

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

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

    int[][] maximumResources = new int[processCount][resourceCount];
    int[][] allocatedResources = new int[processCount][resourceCount];
    int[] availableResources = new int[resourceCount];

    System.out.println("Enter the Max matrix:");
    for (int i = 0; i < processCount; i++) {
        for (int j = 0; j < resourceCount; j++) {
            maximumResources[i][j] = inputScanner.nextInt();
        }
    }

    System.out.println("Enter the Allocation matrix:");
    for (int i = 0; i < processCount; i++) {
        for (int j = 0; j < resourceCount; j++) {
            allocatedResources[i][j] = inputScanner.nextInt();
        }
    }

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

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

ResourceAllocationAlgorithm resourceAllocator = new ResourceAllocationAlgorithm(maximumResources,
allocatedResources, availableResources);
System.out.println("Need Matrix:");
for (int i = 0; i < resourceAllocator.processCount; i++) {
    System.out.println(Arrays.toString(resourceAllocator.requiredResources[i]));
}
ArrayList<Integer> safeSequence = resourceAllocator.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 requestResource2 = inputScanner.nextInt();

int processIndex = 2;
int[] resourceRequest = {0, requestResource2, 0, 0};

int requestResult = resourceAllocator.requestResources(processIndex, resourceRequest);

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

inputScanner.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.
```