Operating System Lab Report – Assignment 2

Name : Tanmoy Sarkar
Roll No : 002010501020
Class : BCSE 5th semester 3rd Year
Group : A1

1. Design a CPU scheduler for jobs whose execution profiles will be in a file that is to be read and appropriate scheduling algorithm to be chosen by the scheduler.

Format of the profile:
<Job id> <priority> <arrival time> <CPU burst(1) I/O burst(1) CPU burst(2) ……. >-1
(Each information is separated by blank space and each job profile ends with -1. Lesser priority number denotes higher priority process with priority number 1 being the process with highest priority.)

Example: 2 3 0 100 2 200 3 25 -1 1 1 4 60 10 ….. -1 etc.

Testing:

a. Create job profiles for 20 jobs and use three different scheduling algorithms (FCFS, preemptive Priority and Round Robin (time slice:20)).
b. Compare the average waiting time, turnaround time of each process for the different scheduling algorithms

Code -

```
import java.io.*;
import java.util.*;

class Job{
    public int id;
    public int priority;
    public int arrivalTime;
    public int completionTime;
    public List<Integer> cpuBursts;
    public List<Integer> ioBursts;
    public int totalCpuBurts;
    public int currentCpuBurstIndex;
    public int currentIoBurstIndex;
    private boolean isCompleted;
    private boolean isInCPU;

    public Job(int id, int priority, int arrivalTime, List<Integer> cpuBursts,
List<Integer> ioBursts){
        this.id = id;
        this.priority = priority;
        this.arrivalTime = arrivalTime;
        this.completionTime = 0;
        this.cpuBursts = cpuBursts;
        this.ioBursts = ioBursts;
        this.currentCpuBurstIndex = 0;
        this.currentIoBurstIndex = 0;
```

```
        this.isCompleted = false;
        this.isInCPU = true;
        this.totalCpuBurts = cpuBursts.stream().reduce(0, (a,b)->a+b);
    }

    public static Job fromString(String txt){
        txt =  txt.strip();
        String[] parts = txt.split(" ");
        if(parts.length < 3){
            throw new IllegalArgumentException("Invalid job string");
        }
        int id = Integer.parseInt(parts[0]);
        int priority = Integer.parseInt(parts[1]);
        int arrivalTime = Integer.parseInt(parts[2]);
        List<Integer> cpuBursts = new ArrayList<>();
        List<Integer> ioBursts = new ArrayList<>();
        for(int i = 3; i < parts.length; i++){
            if((i-3)%2 == 0){
                cpuBursts.add(Integer.parseInt(parts[i]));
            }else{
                ioBursts.add(Integer.parseInt(parts[i]));
            }
        }
        return new Job(id, priority, arrivalTime, cpuBursts, ioBursts);
    }

    public String toString(){
        return String.format("Job %d: priority %d, arrival time %d, cpu bursts %s, io
bursts %s", id, priority, arrivalTime, cpuBursts.toString(), ioBursts.toString());
    }

    public void decrementCPUBurstTime(){
        if(isCompleted) return;
        cpuBursts.set(currentCpuBurstIndex, cpuBursts.get(currentCpuBurstIndex)-1);
        if(cpuBursts.get(currentCpuBurstIndex) == 0){
            if(currentCpuBurstIndex >= cpuBursts.size()-1){
                isCompleted = true;
                isInCPU = false;
            }else{
                currentCpuBurstIndex++;
                isInCPU = false;
            }
        }
    }

    public void decrementIOBurstTime(){
```

```
        if(isCompleted) return;
        ioBursts.set(currentIoBurstIndex, ioBursts.get(currentIoBurstIndex)-1);
        if(ioBursts.get(currentIoBurstIndex) == 0){
            currentIoBurstIndex++;
            isInCPU = true;
        }
    }

    public boolean isProcessCompleted(){
        return isCompleted;
    }

    public boolean isInCPU(){
        return isInCPU;
    }
}

enum JobSchedulerType{
    NONE,
    FCFS,
    PRIORITY,
    RR
}

class Pair<T1, T2>{
    public T1 first;
    public T2 second;

    public Pair(T1 first, T2 second){
        this.first = first;
        this.second = second;
    }
}

class JobScheduler{
    List<Job> jobs;
    boolean debug;
    JobSchedulerType type;
    int time;
    List<Pair<Pair<Integer, Integer>, Job>> ganttChart;

    // Constructor accepts a filename to import job profiles
    public JobScheduler(String filename, boolean debug){
        this.debug = debug;
        type = JobSchedulerType.NONE;
        time = 0;
```

```
        ganttChart = new ArrayList<>();

        jobs = new ArrayList<>();
        try{
            BufferedReader br = new BufferedReader(new FileReader(filename));
            String line = br.readLine();
            String[] jobsTxt = line.split("-1");
            for(String jobTxt : jobsTxt){
                try {
                    jobs.add(Job.fromString(jobTxt));
                } catch (Exception e) {}
            }
            br.close();
            System.out.println("Imported "+Integer.toString(jobs.size())+"
profiles");
        }catch(Exception e){
            e.printStackTrace();
        }

        if(debug){
            System.out.println("----------- Imported Jobs ------------");
            for (Job job : jobs) {
                System.out.println(job);
            }
            System.out.println("-----------------------------------");
        }
    }

    // Function to gnerate random jobs and save in a file
    public static void generateRandomJobs(int n, String filename) throws IOException{
        FileWriter fw = new FileWriter(filename,false);
        BufferedWriter bw = new BufferedWriter(fw);
        int arrivalTime=0;
        for (int i = 0; i < n; i++) {
            StringBuilder sb = new StringBuilder();
            sb.append(JobScheduler.generateRandomNumber(1000, 4000));
            sb.append(" ");
            sb.append(JobScheduler.generateRandomNumber(0, 10));
            sb.append(" ");
            arrivalTime = arrivalTime + JobScheduler.generateRandomNumber(-5, 15);
            sb.append(Integer.toString(arrivalTime));
            sb.append(" ");
            int burstsCount =  JobScheduler.generateRandomNumber(1, 10);
            if(burstsCount%2 == 0){
                burstsCount--;
            }
```

```
        for (int j = 0; j < burstsCount; j++) {
            sb.append(JobScheduler.generateRandomNumber(1, 10));
            sb.append(" ");
        }
        sb.append("-1 ");
        bw.write(sb.toString());
    }
    bw.close();
    fw.close();
}

// An helper function
private static int generateRandomNumber(int min,int max){
    return (int)(Math.random()*(max-min+1)+min);
}

// Function to set the job scheduler type
public void setJobSchedulerType(JobSchedulerType type){
    this.type = type;
}

// Function to check if all jobs completed
private boolean isAllJobsCompleted(){
    for (Job job : jobs) {
        if(job.isProcessCompleted() == false) return false;
    }
    return true;
}

// Function to run the job scheduler
public void run(){
    if(type == JobSchedulerType.NONE){
        System.out.println("No job scheduler type set");
        return;
    }
    if(jobs.size() == 0){
        System.out.println("No jobs to schedule");
        return;
    }
    switch(type){
        case FCFS:
            runFcfs();
            break;
        case PRIORITY:
            runPriority();
            break;
```

```
        case RR:
            runRr();
            break;
        default:
            System.out.println("Invalid job scheduler type");
            break;
    }

}

// Function to run the FCFS job scheduler
private void runFcfs(){
    System.out.println("\nRunning FCFS job scheduler .. .. .. ");
    // Sort all the processes according to arrrival time
    Collections.sort(this.jobs, new Comparator<Job>() {
        @Override
        public int compare(Job o1, Job o2) {
            return o1.arrivalTime - o2.arrivalTime;
        }
    });

    if(debug){
        for(Job job : jobs){
            System.out.println(job);
        }
    }

    Job currentJob ;
    Queue<Job> readyProcessesQueue = new LinkedList<>();
    List<Job> blockProcesses = new ArrayList<>();

    while(isAllJobsCompleted() == false){
        // Add processes to ready queue
        for(Job job: jobs){
            if(job.arrivalTime == time){
                readyProcessesQueue.add(job);
            }
        }

        List<Job> tempBlockProcesses = new ArrayList<>();
        tempBlockProcesses.addAll(blockProcesses);

        // Check if there is any process in ready queue
        if(readyProcessesQueue.size() != 0){

            // Get the first process from the ready queue
```

```
                currentJob = readyProcessesQueue.peek();
                currentJob.decrementCPUBurstTime();

                // Add the process to gantt chart
                ganttChart.add(new Pair<>(new Pair<>(time, time+1), currentJob));

                if(!currentJob.isInCPU()){
                    // If the process is not in CPU, add it to block list

                    if(currentJob.isProcessCompleted()){
                        currentJob.completionTime = time+1;
                        // If the process has been completed then remove from the
ready queue
                        readyProcessesQueue.poll();
                    }else{
                        // If the current job is not in CPU, it means it is in IO
burst, So move it to block queue
                        readyProcessesQueue.poll();
                        blockProcesses.add(currentJob);
                    }
                }
            }

            // Do the IO burst for all the processes in block queue
            for(Job job : tempBlockProcesses){
                job.decrementIOBurstTime();
                if(job.isInCPU()){
                    // If the current job is in CPU, it means it is out of IO burst,
So move it to ready queue
                    blockProcesses.remove(job);
                    readyProcessesQueue.add(job);
                }
            }

            time+=1;
        }

    }

    // Function to run the priority job scheduler
    private void runPriority(){
        // Priority with preemptive
        System.out.println("\nRunning Priority job scheduler .. .. .. ");
        // Sort all the processes according to arrrival time
        Collections.sort(this.jobs, new Comparator<Job>() {
            @Override
```

```
            public int compare(Job o1, Job o2) {
                return o1.arrivalTime - o2.arrivalTime;
            }
        });

        PriorityQueue<Job> readyProcessesQueue = new PriorityQueue<Job>(new
Comparator<Job>() {
            @Override
            public int compare(Job o1, Job o2) {
                return o1.priority - o2.priority;
            }
        });

        List<Job> blockProcesses = new ArrayList<>();
        Job currentJob;

        while(isAllJobsCompleted() == false){
            // Add processes to ready queue
            for(Job job: jobs){
                if(job.arrivalTime == time){
                    readyProcessesQueue.add(job);
                }
            }

            List<Job> tempBlockProcesses = new ArrayList<>();
            tempBlockProcesses.addAll(blockProcesses);

            // Check if there is any process in ready queue
            if(readyProcessesQueue.size() != 0){

                // Get the first process from the ready queue
                currentJob = readyProcessesQueue.peek();
                currentJob.decrementCPUBurstTime();

                // Add the process to gantt chart
                ganttChart.add(new Pair<>(new Pair<>(time, time+1), currentJob));

                if(!currentJob.isInCPU()){
                    // If the process is not in CPU, add it to block list

                    if(currentJob.isProcessCompleted()){
                        currentJob.completionTime = time+1;
                        // If the process has been completed then remove from the
ready queue
                        readyProcessesQueue.remove(currentJob);
                    }else{
```

```
                        // If the current job is not in CPU, it means it is in IO
burst, So move it to block queue
                        readyProcessesQueue.remove(currentJob);
                        blockProcesses.add(currentJob);
                    }
                }
            }


            // Do the IO burst for all the processes in block queue
            for(Job job : tempBlockProcesses){
                job.decrementIOBurstTime();
                if(job.isInCPU()){
                    // If the current job is in CPU, it means it is out of IO burst,
So move it to ready queue
                    blockProcesses.remove(job);
                    readyProcessesQueue.add(job);
                }
            }
            time +=1;
        }



    }

    // Function to run the round robin job scheduler
    private void runRr(){
        System.out.println("\nRunning Round Robin job scheduler .. .. ..");

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

        // Sort all the processes according to arrrival time
        Collections.sort(this.jobs, new Comparator<Job>() {
            @Override
            public int compare(Job o1, Job o2) {
                return o1.arrivalTime - o2.arrivalTime;
            }
        });

        if(debug){
            for(Job job : jobs){
                System.out.println(job);
            }
```

```
        }

        Job currentJob ;
        Queue<Job> readyProcessesQueue = new LinkedList<>();
        List<Job> blockProcesses = new ArrayList<>();
        int timeQuantumCounter = 0;

        while(isAllJobsCompleted() == false){
            // Add processes to ready queue
            for(Job job: jobs){
                if(job.arrivalTime == time){
                    readyProcessesQueue.add(job);
                }
            }

            List<Job> tempBlockProcesses = new ArrayList<>();
            tempBlockProcesses.addAll(blockProcesses);

            // Check if there is any process in ready queue
            if(readyProcessesQueue.size() != 0){

                if(timeQuantumCounter == timeQuantum){
                    // If the time quantum is completed, move the current process to
the end of the queue
                    readyProcessesQueue.add(readyProcessesQueue.poll());
                    timeQuantumCounter = 0;
                }

                // Get the first process from the ready queue
                currentJob = readyProcessesQueue.peek();
                currentJob.decrementCPUBurstTime();

                // Add the process to gantt chart
                ganttChart.add(new Pair<>(new Pair<>(time, time+1), currentJob));

                if(!currentJob.isInCPU()){
                    // If the process is not in CPU, add it to block list

                    if(currentJob.isProcessCompleted()){
                        currentJob.completionTime = time+1;
                        // If the process has been completed then remove from the
ready queue
                        readyProcessesQueue.poll();
                    }else{
                        // If the current job is not in CPU, it means it is in IO
burst, So move it to block queue
```

```
                    readyProcessesQueue.poll();
                    blockProcesses.add(currentJob);
                }
            }
        }

        // Do the IO burst for all the processes in block queue
        for(Job job : tempBlockProcesses){
            job.decrementIOBurstTime();
            if(job.isInCPU()){
                // If the current job is in CPU, So move it to ready queue
                blockProcesses.remove(job);
                readyProcessesQueue.add(job);
            }
        }

        time+=1;
        timeQuantumCounter+=1;
    }
}

// Function to print gantt chart
public void printGanttChart(){
    System.out.println("------------------\nFinal Gantt
Chart\n------------------");
    for (Pair<Pair<Integer, Integer>, Job> pair : ganttChart) {
        System.out.println("Time:
"+pair.first.first+"-"+pair.first.second+"\tJob: "+pair.second.id);
    }
}

// Function to calculate the stats
public void calculateStats(){
    int waitingTime = 0;
    int turnaroundTime = 0;
    int totalCpuBursts = 0;
    for (Job job : jobs) {
        turnaroundTime += (job.completionTime - job.arrivalTime);
        waitingTime += (job.completionTime - job.arrivalTime -job.totalCpuBurts);
        totalCpuBursts += job.totalCpuBurts;
    }

    System.out.println("\n-- Stats --");
    System.out.println("Average waiting time: "+((waitingTime*1.0)/jobs.size()));
    System.out.println("Average turnaround time:
"+((turnaroundTime*1.0)/jobs.size()));
```

```
        System.out.println("CPU Utilization: "+Math.round((totalCpuBursts*1.0)/time *
100)+" %");
    }
}


public class q1 {
    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);
        int choice = 0;
        JobScheduler jobScheduler = null;
        do{
            System.out.println("\n\n");
            System.out.println("1. Generate random job profiles");
            System.out.println("2. Initaie object & Read job profiles from file");
            System.out.println("3. Schedule jobs using FCFS");
            System.out.println("4. Schedule jobs using Priority Scheduling
Preamptive");
            System.out.println("5. Schedule jobs using RR");
            System.out.println("6. Exit");
            System.out.print("Enter your choice : ");
            choice = sc.nextInt();
            sc.nextLine();
            switch(choice){
                case 1:
                    System.out.print("Enter the number of jobs to generate: ");
                    int n = sc.nextInt();
                    sc.nextLine();
                    System.out.print("Enter filename to save the job profiles: ");
                    String filename = sc.nextLine();
                    JobScheduler.generateRandomJobs(n, filename);
                    System.out.println("Job profiles generated successfully");
                    break;
                case 2:
                    System.out.print("Enter the filename: ");
                    String filename2 = sc.nextLine();
                    jobScheduler = new JobScheduler(filename2, false);
                    break;
                case 3:
                    jobScheduler.setJobSchedulerType(JobSchedulerType.FCFS);
                    jobScheduler.run();
                    jobScheduler.printGanttChart();
                    jobScheduler.calculateStats();
                    break;
                case 4:
                    jobScheduler.setJobSchedulerType(JobSchedulerType.PRIORITY);
```

```
                    jobScheduler.run();
                    jobScheduler.printGanttChart();
                    jobScheduler.calculateStats();
                    break;
                case 5:
                    jobScheduler.setJobSchedulerType(JobSchedulerType.RR);
                    jobScheduler.run();
                    jobScheduler.printGanttChart();
                    jobScheduler.calculateStats();
                    break;
                case 6:
                    System.out.println("Exiting .. .. ..");
                    break;
                default:
                    System.out.println("Invalid choice");
            }

        }while(choice != -1);
    }
}
```

Output -

1. Generate random job profiles
2. Initaie object & Read job profiles from file
3. Schedule jobs using FCFS
4. Schedule jobs using Priority Scheduling Preamptive
5. Schedule jobs using RR
6. Exit
Enter your choice : 2
Enter the filename: job_profiles_new.txt
Imported 20 profiles

1. Generate random job profiles
2. Initaie object & Read job profiles from file
3. Schedule jobs using FCFS
4. Schedule jobs using Priority Scheduling Preamptive
5. Schedule jobs using RR
6. Exit
Enter your choice : 3

Running FCFS job scheduler .. .. ..

-- Stats --

Average waiting time: 135.55
Average turnaround time: 153.4
CPU Utilisation: 98 %

Enter your choice : 4
Running Priority job scheduler .. .. ..

-- Stats --
Average waiting time: 132.15
Average turnaround time: 150.0
CPU Utilisation: 93 %

Running Round Robin job scheduler .. .. ..
Enter the time quantum: 2

-- Stats --
Average waiting time: 135.55
Average turnaround time: 153.4
CPU Utilisation: 98 %

2. Create child processes: X and Y.
   a. Each child process performs 10 iterations. The child process displays its name/id and the current iteration number, and sleeps for some random amount of time. Adjust the sleeping duration of the processes to have different outputs (i.e. another interleaving of processes' traces).
   b. Modify the program so that X is not allowed to start iteration i before process Y has terminated its own iteration i-1. Use semaphore to implement this synchronization.
   c. Modify the program so that X and Y now perform in lockstep [both perform iteration I, then iteration i+1, and so on] with the condition mentioned in Q (2b) above.
   d. Add another child process Z.

Perform the operations as mentioned in Q (2a) for all three children.
Then perform the operations as mentioned in Q (2c) [that is, 3 children in lockstep].

Code -

Part A

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10
```

```
#define MIN_RANDOM_NUMBER 50000
#define MAX_RANDOM_NUMBER 100000

int getRandomNumber(){
    return rand()%((MAX_RANDOM_NUMBER+1)-MIN_RANDOM_NUMBER) + MIN_RANDOM_NUMBER;
}

void displayData(char* processName, int iteration){
    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName, getpid(),
iteration+1);
    usleep(getRandomNumber());  // Sleep random time
}

int main()
{
    srand(time(NULL));
    int pidX = -1;
    int pidY = -1;
    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        for (int i = 0; i < ITERATIONS; i++){
            displayData("X", i);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0){
            // Inside Child Y
            for (int i = 0; i < ITERATIONS; i++)
            {
                displayData("Y", i);
            }
        }
    }

    for (int i = 1; i <= 2; i++){
        wait(NULL);
    }
    return 0;
}
```

Output -

Process : X | PID : 61672 | Iteration : 1

```
Process  : Y  | PID : 61673 | Iteration : 1
Process  : X  | PID : 61672 | Iteration : 2
Process  : Y  | PID : 61673 | Iteration : 2
Process  : X  | PID : 61672 | Iteration : 3
Process  : Y  | PID : 61673 | Iteration : 3
Process  : X  | PID : 61672 | Iteration : 4
Process  : Y  | PID : 61673 | Iteration : 4
Process  : X  | PID : 61672 | Iteration : 5
Process  : Y  | PID : 61673 | Iteration : 5
Process  : X  | PID : 61672 | Iteration : 6
Process  : Y  | PID : 61673 | Iteration : 6
Process  : X  | PID : 61672 | Iteration : 7
Process  : Y  | PID : 61673 | Iteration : 7
Process  : X  | PID : 61672 | Iteration : 8
Process  : Y  | PID : 61673 | Iteration : 8
Process  : X  | PID : 61672 | Iteration : 9
Process  : Y  | PID : 61673 | Iteration : 9
Process  : X  | PID : 61672 | Iteration : 10
Process  : Y  | PID : 61673 | Iteration : 10
```

Part B

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10
#define MIN_RANDOM_NUMBER 50000
#define MAX_RANDOM_NUMBER 100000

sem_t* mutex;

int getRandomNumber(){
    return rand()%((MAX_RANDOM_NUMBER+1)-MIN_RANDOM_NUMBER) + MIN_RANDOM_NUMBER;
}

void displayData(char* processName, int iteration){
    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName, getpid(),
iteration+1);
    usleep(getRandomNumber());  // Sleep random time
}
```

```
int main()
{
    srand(time(NULL));
    sem_unlink("mutex015");
    mutex = sem_open("mutex015", O_CREAT, 0777, 0);
    int pidX = -1;
    int pidY = -1;
    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        for (int i = 0; i < ITERATIONS; i++){
            displayData("X", i);
        }
        sem_post(mutex);
    }
    else
    {
        pidY = fork();
        if (pidY == 0){
            // Inside Child Y
            sem_wait(mutex);
            for (int i = 0; i < ITERATIONS; i++)
            {
                displayData("Y", i);
            }
        }
    }

    for (int i = 1; i <= 2; i++){
        wait(NULL);
    }

    sem_destroy(mutex);

    return 0;
}
```

Output -

```
Process  : X  | PID : 61981 | Iteration : 1
Process  : X  | PID : 61981 | Iteration : 2
Process  : X  | PID : 61981 | Iteration : 3
Process  : X  | PID : 61981 | Iteration : 4
Process  : X  | PID : 61981 | Iteration : 5
Process  : X  | PID : 61981 | Iteration : 6
Process  : X  | PID : 61981 | Iteration : 7
```

```
Process  : X  | PID : 61981 | Iteration : 8
Process  : X  | PID : 61981 | Iteration : 9
Process  : X  | PID : 61981 | Iteration : 10
Process  : Y  | PID : 61982 | Iteration : 1
Process  : Y  | PID : 61982 | Iteration : 2
Process  : Y  | PID : 61982 | Iteration : 3
Process  : Y  | PID : 61982 | Iteration : 4
Process  : Y  | PID : 61982 | Iteration : 5
Process  : Y  | PID : 61982 | Iteration : 6
Process  : Y  | PID : 61982 | Iteration : 7
Process  : Y  | PID : 61982 | Iteration : 8
Process  : Y  | PID : 61982 | Iteration : 9
Process  : Y  | PID : 61982 | Iteration : 10
```

Part C

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <time.h>

#define ITERATIONS 10
#define MIN_RANDOM_NUMBER 50000
#define MAX_RANDOM_NUMBER 100000

sem_t* mutex;

int getRandomNumber(){
    return rand()%((MAX_RANDOM_NUMBER+1)-MIN_RANDOM_NUMBER) + MIN_RANDOM_NUMBER;
}

void displayData(char* processName, int iteration){
    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName, getpid(),
iteration+1);
    usleep(getRandomNumber());  // Sleep random time
}

int main()
{
    srand(time(NULL));
    mutex = sem_open("mutex05", O_CREAT, 0777, 0);
    int pidX = -1;
```

```
    int pidY = -1;
    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        for (int i = 0; i < ITERATIONS; i++){
            if(i != 0) sem_wait(mutex);
            displayData("X", i);
            sem_post(mutex);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0){
            // Inside Child Y
            for (int i = 0; i < ITERATIONS; i++)
            {
                sem_wait(mutex);
                displayData("Y", i);
                sem_post(mutex);
            }
        }
    }
    for (int i = 1; i <= 2; i++){
        wait(NULL);
    }
    sem_destroy(mutex);
    return 0;
}
```

Output -

```
Process  : X  | PID : 62210 | Iteration : 1
Process  : X  | PID : 62210 | Iteration : 2
Process  : X  | PID : 62210 | Iteration : 3
Process  : X  | PID : 62210 | Iteration : 4
Process  : X  | PID : 62210 | Iteration : 5
Process  : X  | PID : 62210 | Iteration : 6
Process  : X  | PID : 62210 | Iteration : 7
Process  : X  | PID : 62210 | Iteration : 8
Process  : X  | PID : 62210 | Iteration : 9
Process  : X  | PID : 62210 | Iteration : 10
Process  : Y  | PID : 62211 | Iteration : 1
Process  : Y  | PID : 62211 | Iteration : 2
Process  : Y  | PID : 62211 | Iteration : 3
Process  : Y  | PID : 62211 | Iteration : 4
```

```
Process  : Y  | PID : 62211 | Iteration : 5
Process  : Y  | PID : 62211 | Iteration : 6
Process  : Y  | PID : 62211 | Iteration : 7
Process  : Y  | PID : 62211 | Iteration : 8
Process  : Y  | PID : 62211 | Iteration : 9
Process  : Y  | PID : 62211 | Iteration : 10
```

Part D

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
#include <semaphore.h>
#include <fcntl.h>

#define ITERATIONS 10
#define MIN_RANDOM_NUMBER 50000
#define MAX_RANDOM_NUMBER 100000

sem_t* mutexX;
sem_t* mutexY;
sem_t* mutexZ;

int getRandomNumber(){
    return rand()%((MAX_RANDOM_NUMBER+1)-MIN_RANDOM_NUMBER) + MIN_RANDOM_NUMBER;
}

void displayData(char* processName, int iteration){
    printf("Process  : %s  | PID : %d | Iteration : %d\n", processName, getpid(),
iteration+1);
    usleep(getRandomNumber());  // Sleep random time
}

int main()
{
    srand(time(NULL));
    sem_unlink("mutexX");
    sem_unlink("mutexY");
    sem_unlink("mutexZ");
    mutexX = sem_open("mutexX", O_CREAT, 0644, 0);
    mutexY = sem_open("mutexY", O_CREAT, 0644, 0);
    mutexZ = sem_open("mutexZ", O_CREAT, 0644, 0);
    int pidX = -1;
```

```
    int pidY = -1;
    int pidZ = -1;
    pidX = fork();
    if (pidX == 0)
    {
        // Inside child X
        for (int i = 0; i < ITERATIONS; i++){
            if(i != 0){
                sem_wait(mutexX);
            };
            displayData("X", i);
            sem_post(mutexY);
        }
    }
    else
    {
        pidY = fork();
        if (pidY == 0){
            // Inside Child Y
            for (int i = 0; i < ITERATIONS; i++)
            {
                sem_wait(mutexY);
                displayData("Y", i);
                sem_post(mutexZ);
            }
        }else{
            pidZ = fork();
            if(pidZ == 0){
                for (int i = 0; i < ITERATIONS; i++)
                {
                    sem_wait(mutexZ);
                    displayData("Z", i);
                    sem_post(mutexX);
                }
            }
        }
    }
    for (int i = 1; i <= 3; i++){
        wait(NULL);
    }
    sem_destroy(mutexX);
    sem_destroy(mutexY);
    sem_destroy(mutexY);
    return 0;
}
```

Output -

```
Process  : X  | PID : 62386 | Iteration : 1
Process  : Y  | PID : 62387 | Iteration : 1
Process  : Z  | PID : 62388 | Iteration : 1
Process  : X  | PID : 62386 | Iteration : 2
Process  : Y  | PID : 62387 | Iteration : 2
Process  : Z  | PID : 62388 | Iteration : 2
Process  : X  | PID : 62386 | Iteration : 3
Process  : Y  | PID : 62387 | Iteration : 3
Process  : Z  | PID : 62388 | Iteration : 3
Process  : X  | PID : 62386 | Iteration : 4
Process  : Y  | PID : 62387 | Iteration : 4
Process  : Z  | PID : 62388 | Iteration : 4
Process  : X  | PID : 62386 | Iteration : 5
Process  : Y  | PID : 62387 | Iteration : 5
Process  : Z  | PID : 62388 | Iteration : 5
Process  : X  | PID : 62386 | Iteration : 6
Process  : Y  | PID : 62387 | Iteration : 6
Process  : Z  | PID : 62388 | Iteration : 6
Process  : X  | PID : 62386 | Iteration : 7
Process  : Y  | PID : 62387 | Iteration : 7
Process  : Z  | PID : 62388 | Iteration : 7
Process  : X  | PID : 62386 | Iteration : 8
Process  : Y  | PID : 62387 | Iteration : 8
Process  : Z  | PID : 62388 | Iteration : 8
Process  : X  | PID : 62386 | Iteration : 9
Process  : Y  | PID : 62387 | Iteration : 9
Process  : Z  | PID : 62388 | Iteration : 9
Process  : X  | PID : 62386 | Iteration : 10
Process  : Y  | PID : 62387 | Iteration : 10
Process  : Z  | PID : 62388 | Iteration : 10
```

Explanation -

In the above program, semaphore is used to provide the synchronization.

sem_open() function is used to create the semaphore which is initialized to 0. We know that the value of semaphore denotes the number of waiting processes to access the resource. If the value is negative then the absolute value denotes the number of waiting processes. If the value is 0 or positive, it indicates that the resource can be allocated to the process.
sem_wait() function is used before executing process X in each iteration. This makes the process X wait for the negative value of semaphore.
sem_post() function is used to send a signal to process X after the completion of process Y in each iteration. The main purpose of this function is to increase the value of semaphore by 1 so that it becomes non-negative and process X can start its execution.

3. Implement the following applications using different IPC mechanisms. Your choice is restricted to Pipe, FIFO:

a. Broadcasting weather information (one broadcasting process and more than one listeners)
b. Telephonic conversation (between a caller and a receiver)

Code -

Part A

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>

#define BUFFER_SIZE 200
#define CONSUMERS 4

int main(){
    int fdarr[CONSUMERS][2];
    for (int i = 0; i < CONSUMERS; i++)
    {
        int res = pipe(fdarr[i]);
        if (res == -1)
        {
            printf("Pipe creation failed for consumer %d", i+1);
            return 1;
        }
    }
    // Create child process [1]
    int pid = fork();
    if (pid == -1)
    {
        printf("Fork failed");
        return 1;
    }
    // Child process
    if(pid == 0){
        int consumer_id = 1;
        int pid2 = fork(); // [2]
        // Child process
        if(pid2 == 0){
            consumer_id = 2;
        }
```

```
    int pid3 = fork(); // [3]
    if(pid3 == 0){
        consumer_id = consumer_id + 2;
    }
    char data[BUFFER_SIZE];
    close(fdarr[consumer_id-1][1]);
    while (1){
        int res = read(fdarr[consumer_id-1][0], data, BUFFER_SIZE);
        if (res > 0){
            if(strcmp(data, "EOF") == 0){
                printf("\nConsumer %d finished\n", consumer_id);
                break;
            }
            printf("Consumer %d: %s", consumer_id, data);
        }
    }

    close(fdarr[consumer_id-1][0]);
}
// Parent process
else{
    // Close all read ends of pipes
    for (int i = 0; i < CONSUMERS; i++){
        close(fdarr[i][0]);
    }
    // Open file
    FILE *fp = fopen("weather.txt", "r");
    if (fp == NULL){
        printf("File open failed");
        return 1;
    }
    // Read file line by line
    char buffer[BUFFER_SIZE];
    while (fgets(buffer, BUFFER_SIZE, fp) != NULL){
        // Write to all pipes
        for (int i = 0; i < CONSUMERS; i++)
        {
            write(fdarr[i][1], buffer, BUFFER_SIZE);
        }
    }
    // Send something to all pipes to indicate end of file
    for (int i = 0; i < CONSUMERS; i++){
        write(fdarr[i][1], "EOF", BUFFER_SIZE);
    }
    // Close all write ends of pipes
    for (int i = 0; i < CONSUMERS; i++){
```

```
            close(fdarr[i][1]);
        }
        // Wait for child process to finish
        wait(NULL);
    }
    return 0;
}
```

Output -

```
Consumer 1: Temp : 56C
Consumer 3: Temp : 56C
Consumer 3: Wind : 10.0 km/h
Consumer 3: Humidity : 100%
Consumer 4: Temp : 56C
Consumer 4: Wind : 10.0 km/h
Consumer 3: Pressure : 1010.0 mb
Consumer 3: Visibility : 10.0 km
Consumer 3: Cloud Cover : 100%
Consumer 3: Precipitation : 0.0 mm
Consumer 3: Weather : Overcast
Consumer 3: Weather Description : overcast clouds
Consumer 3 finished
Consumer 4: Humidity : 100%
Consumer 1: Wind : 10.0 km/h
Consumer 1: Humidity : 100%
Consumer 2: Temp : 56C
Consumer 2: Wind : 10.0 km/h
Consumer 2: Humidity : 100%
Consumer 2: Pressure : 1010.0 mb
Consumer 2: Visibility : 10.0 km
Consumer 2: Cloud Cover : 100%
Consumer 2: Precipitation : 0.0 mm
Consumer 2: Weather : Overcast
Consumer 2: Weather Description : overcast clouds
Consumer 2 finished
Consumer 1: Pressure : 1010.0 mb
Consumer 1: Visibility : 10.0 km
Consumer 1: Cloud Cover : 100%
Consumer 1: Precipitation : 0.0 mm
Consumer 1: Weather : Overcast
Consumer 1: Weather Description : overcast clouds
Consumer 1 finished
Consumer 4: Pressure : 1010.0 mb
Consumer 4: Visibility : 10.0 km
Consumer 4: Cloud Cover : 100%
Consumer 4: Precipitation : 0.0 mm
Consumer 4: Weather : Overcast
Consumer 4: Weather Description : overcast clouds
Consumer 4 finished
```

Part B

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#define BUFFER_SIZE 200
#define FIFO_NAME "telephone_fifo"

int main(){
    int pid = fork();
    if(pid == -1){
        printf("Fork creation failed");
        return 1;
    }
    // Create fifo
    if(mkfifo(FIFO_NAME, 0777) == -1){
        if(errno != EEXIST){
            printf("An error occured while creating the fifo\n");
            return 1;
        }
    }
    // Child process act as a receiver
    if(pid == 0){
        int fd = open(FIFO_NAME, O_RDONLY);
        char data[BUFFER_SIZE];
        while (1){
            if(read(fd, data, BUFFER_SIZE) > 0){
                if(strcmp(data, "end\n") == 0){
                    printf("\nReceiver: The caller has hanged up\n");
                    break;
                }else{
                    printf("Received: %s", data);
                }
            }
        }
        close(fd);
    }
    // Parent process act as a caller
    else{
        int fd = open(FIFO_NAME, O_WRONLY);
```

```
        char inputData[BUFFER_SIZE];
        while (1){
            printf("Enter message: ");
            fgets(inputData, BUFFER_SIZE, stdin);

            write(fd, inputData, BUFFER_SIZE);
            if(strcmp(inputData, "end\n") == 0){
                printf("\nCaller: I am hang up call\n");
                break;
            }
        }
        close(fd);
        wait(NULL);
    }
    return 0;
}
```

Output -

Enter message: hi
Received: hi
Enter message: hello
Enter message: bye
Received: hello
Enter message: bye
Received: bye

Explanation -

The problem states that there is one broadcaster which will broadcast a message, and on the other side, there are more than one broadcaster which will act as the receiver of broadcasted message. To implement this IPC mechanism, FIFO/Pipe can be used. In the broadcaster code, a new FIFO or named pipe is created or the existing FIFO is opened. We use system calls associated with it to store and retrieve information. The information is in character array format. The FIFO is first opened, and then a character array is entered by the user. The information, which is the message to be broadcasted, is stored in the FIFO, and then the FIFO is closed. In the receiver code, the existing FIFO is opened for retrieving the message broadcasted by the broadcaster. The message is then printed. After printing the message, we close the FIFO by making the call close().

4. Write a program for p-producer c-consumer problem, p, c >= 1. A shared circular buffer that can hold 25 items is to be used. Each producer process stores any numbers between 1 to 80 (along with the producer id)
in the buffer one by one and then exits. Each consumer process reads the numbers from the buffer and adds them to a shared variable TOTAL (initialised to 0). Though any consumer process can read any of the numbers in the buffer, the only constraint being that every number written by some producer should be read exactly once by exactly one of the consumers.

a. The program reads in the value of p and c from the user, and forks p producers and c consumers.

b. After all the producers and consumers have finished (the consumers exit after all the data produced by all producers have been read), the parent process prints the value of TOTAL.

c.

Test the program with different values of p and c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define BUFFER_MAX_ELEMENTS 25
#define MIN_RANDOM_NUMBER 1
#define MAX_RANDOM_NUMBER 80

int getRandomNumber()
{
    return rand() % ((MAX_RANDOM_NUMBER + 1) - MIN_RANDOM_NUMBER) +
MIN_RANDOM_NUMBER;
}

struct Buffer{
    int data[BUFFER_MAX_ELEMENTS + 1];
    int readCur;  // read cursor
    int writeCur; // write cursor
    int count;    // count
    sem_t *space_available_mutex;
    sem_t *data_available_mutex;
};

void initBuffer(struct Buffer *buffer){
    buffer->readCur = -1;
    buffer->writeCur = 0;
    buffer->count = 0;

    // Init mutex
    buffer->space_available_mutex = sem_open("mutex74656", O_CREAT, 0777, 1);
```

```
    buffer->data_available_mutex = sem_open("mutex7496", O_CREAT, 0777, 1);
}

void insertData(struct Buffer *buffer, int data){
    if (buffer->count == BUFFER_MAX_ELEMENTS)
        return;
    buffer->data[buffer->writeCur] = data;
    buffer->writeCur = (buffer->writeCur + 1) % BUFFER_MAX_ELEMENTS;
    buffer->count++;
    sem_post(buffer->data_available_mutex);
}

int getData(struct Buffer *buffer){
    if (buffer->count == 0)
        return -1;
    buffer->readCur = (buffer->readCur + 1) % BUFFER_MAX_ELEMENTS;
    buffer->count--;
    sem_post(buffer->space_available_mutex);
    return buffer->data[buffer->readCur];
}

struct Buffer *buffer;
int *total;
int *noOfProducersExited;

int main()
{
    srand(time(NULL));
    int noProducers, noConsumers;
    // Init buffer
    buffer = mmap(NULL, sizeof *buffer, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    total = mmap(NULL, sizeof *total, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
    noOfProducersExited = mmap(NULL, sizeof *total, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    initBuffer(buffer);
    *total = 0;
    *noOfProducersExited = 0;
    // Get number of producers and consumers
    printf("Enter number of producers: ");
    scanf("%d", &noProducers);
    printf("Enter number of consumers: ");
    scanf("%d", &noConsumers);
    if (noProducers == 0 || noConsumers == 0)    {
```

```
    printf("No producers or consumers. Exiting...\n");
    return 0;
}

int id;

// Spin producers
for (int i = 0; i < noProducers; i++){
    // Producers
    id = fork();
    if (id == 0){
        // If buffer full wait
        if (buffer->count == BUFFER_MAX_ELEMENTS){
            sem_wait(buffer->space_available_mutex);
        }
        // Generate random no
        time_t t;
        srand((int)time(&t) % getpid());
        int num = getRandomNumber();
        // Insert data
        insertData(buffer, num);
        // Print log
        printf("Producer %d generated no %d\n", getpid(), num);
        *noOfProducersExited = *noOfProducersExited + 1;
        exit(0);
    }
}

// Spin consumers

if (id > 0){
    for (int i = 0; i < noConsumers; i++){
        // Consumers
        id = fork();
        if (id == 0){
            while (1){
                // If no data please wait
                while (buffer->count == 0){
                    if (*noOfProducersExited == noProducers)
                        break;
                    // sem_wait(buffer->data_available_mutex);
                    usleep(10000);
                }
                // Get dta from buffer
                int data = getData(buffer);
                // Add
```

```
                *total = *total + data;
            }
            exit(0);
        }
    }

    for (int i = 0; i < noConsumers; i++){
        wait(NULL);
    }

    while (noProducers < *noOfProducersExited){
        usleep(10000);
    }

    printf("Total %d\n", *total);
}

return 0;
}
```

Output -

```
Enter number of producers: 5
Enter number of consumers: 5
Producer 66121 generated no 79
Producer 66120 generated no 16
Producer 66122 generated no 12
Producer 66123 generated no 2
Producer 66124 generated no 12
Total 121
```

5. Write a program for the Reader-Writer process for the following situations:
   a. Multiple readers and one writer: writer gets to write whenever it is ready (reader/s wait)
   b. Multiple readers and multiple writers: any writer gets to write whenever it is ready, provided no other writer is currently writing (reader/s wait)

Code -

Part a

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define no_of_readers 5
#define MIN_RANDOM_NUMBER 1
#define MAX_RANDOM_NUMBER 5

sem_t *write_mutex;
int *read_counter;
sem_t *read_counter_mutex;
int *resource;

int getRandomNumber(){
    return rand() % ((MAX_RANDOM_NUMBER + 1) - MIN_RANDOM_NUMBER) + MIN_RANDOM_NUMBER;
}

int main(){
    sem_unlink("mutex34");
    sem_unlink("mute56");
    write_mutex = sem_open("mutex34", O_CREAT, 0777, 1);
    read_counter_mutex = sem_open("mute56", O_CREAT, 0777, 1);
    read_counter = mmap(NULL, sizeof *read_counter, PROT_READ | PROT_WRITE, MAP_SHARED
| MAP_ANONYMOUS, -1, 0);
    resource = mmap(NULL, sizeof *resource, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);

    int id;
    id = fork();
    if (id == 0){
        // Start reader processes
        for (int i = 0; i < no_of_readers; i++){
            id = fork();
            if (id == 0){
                time_t t;
                srand((int)time(&t) % getpid());
                // Reader process
                sleep(getRandomNumber());
                // Increment read counter
                sem_wait(read_counter_mutex);
                *read_counter++;
                if (*read_counter == 1) sem_wait(write_mutex);
                sem_post(read_counter_mutex);
```

```
                // Read
                printf("Reader %d read %d\n", i + 1, *resource);
                // Decrement read counter
                sem_wait(read_counter_mutex);
                *read_counter--;
                if (*read_counter == 0)
                    sem_post(write_mutex);
                sem_post(read_counter_mutex);
                exit(1);
            }
        }
    }
    else{
        time_t t;
        srand((int)time(&t) % getpid());
        // Start writer process
        sleep(getRandomNumber());
        // Wait for all readers to finish
        sem_wait(write_mutex);
        // Write
        *resource = getRandomNumber();
        printf("Writer wrote %d\n", *resource);
        // Release write mutex
        sem_post(write_mutex);
    }
    // Wait for all childs to finish
    for (int i = 0; i < no_of_readers + 1; i++){
        wait(NULL);
    }
    return 0;
}
```

Output -

```
Reader 5 read 0
Writer wrote 2
Reader 1 read 2
Reader 3 read 2
Reader 2 read 2
Reader 4 read 2
```

Part B

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define MIN_RANDOM_NUMBER 1
#define MAX_RANDOM_NUMBER 5

sem_t *write_mutex;
sem_t *read_mutex;
int *read_counter;
int *write_counter;
sem_t *read_counter_mutex;
sem_t *write_counter_mutex;
int *resource;

int getRandomNumber(){
    return rand() % ((MAX_RANDOM_NUMBER + 1) - MIN_RANDOM_NUMBER) +
MIN_RANDOM_NUMBER;
}

int main(){
    int no_of_readers;
    int no_of_writers;

    printf("Enter no of readers: ");
    scanf("%d", &no_of_readers);
    printf("Enter no of writers: ");
    scanf("%d", &no_of_writers);
    sem_unlink("mutex34");
    sem_unlink("mute56");
    sem_unlink("mutex341");
    sem_unlink("mute561");
    write_mutex = sem_open("mutex34", O_CREAT, 0777, 1);
    read_mutex = sem_open("mutex341", O_CREAT, 0777, 1);
    read_counter_mutex = sem_open("mute56", O_CREAT, 0777, 1);
    write_counter_mutex = sem_open("mute561", O_CREAT, 0777, 1);
    read_counter = mmap(NULL, sizeof *read_counter, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    write_counter = mmap(NULL, sizeof *read_counter, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

```
    resource = mmap(NULL, sizeof *resource, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);

    int id;
    id = fork();
    if (id == 0){
        // Start reader processes
        for (int i = 0; i < no_of_readers; i++){
            id = fork();
            if (id == 0){
                time_t t;
                srand((int)time(&t) % getpid());
                // Reader process
                sleep(getRandomNumber());
                // Increment read counter
                sem_wait(read_counter_mutex);
                *read_counter++;
                if (*read_counter == 1) sem_wait(write_mutex);
                sem_post(read_counter_mutex);
                // Read
                printf("Reader %d read %d\n", i + 1, *resource);
                // Decrement read counter
                sem_wait(read_counter_mutex);
                *read_counter--;
                if (*read_counter == 0) sem_post(write_mutex);
                sem_post(read_counter_mutex);
                exit(1);
            }
        }
        // Wait for all childs to finish
        for (int i = 0; i < no_of_readers; i++){
            wait(NULL);
        }
        exit(1);
    }
    else{
        for (int i = 0; i < no_of_writers; i++){
            id = fork();
            if (id == 0){
                time_t t;
                srand((int)time(&t) % getpid());
                // Writer process
                sleep(getRandomNumber());
                // Increment write counter
                sem_wait(write_counter_mutex);
                *write_counter++;
```

```
                if (*write_counter == 1){
                    sem_wait(read_mutex);
                }
                sem_post(write_counter_mutex);
                sem_wait(write_mutex);
                // Write
                *resource = getRandomNumber();
                printf("Writer %d wrote %d\n", i + 1, *resource);
                sem_post(write_mutex);
                // Decrement write counter
                sem_wait(write_counter_mutex);
                *write_counter--;
                if (*write_counter == 0) sem_post(read_mutex);
                sem_post(write_counter_mutex);
                exit(1);
            }
        }

        // Wait for all childs to finish
        for (int i = 0; i < no_of_readers; i++){
            wait(NULL);
        }
        wait(NULL);
    }
    return 0;
}
```

Output-

Enter no of readers: 5
Enter no of writers: 6
Reader 1 read 0
Reader 2 read 0
Writer 1 wrote 5
Writer 4 wrote 3
Writer 3 wrote 1
Reader 3 read 1
Reader 4 read 1
Writer 5 wrote 2
Reader 5 read 2
Writer 2 wrote 3
Writer 6 wrote 3

6. Implement Dining Philosophers' problem using Monitor. Test the program with (a) 5 philosophers and 5 chopsticks, (b) 6 philosophers and 6 chopsticks, and (c) 7 philosophers and 7 chopsticks

Code -

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <fcntl.h>
#include <semaphore.h>
#include <time.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int N;
enum Status{
      THINKING,
      HUNGRY,
      EATING
};

enum Status *state;
sem_t *mutex;
sem_t **condition;

char *mutex_key_from_no(int n){
      char *buffer;
      buffer = (char *)malloc(sizeof(char) * 50);
      sprintf(buffer, "mutex%d", n);
      return buffer;
}

// test left and right
void test(int i){
      if (state[i] == HUNGRY && state[(i + 1) % N] != EATING && state[(i+N-1) % N] !=
EATING)     {

            // update state to eating
            sem_wait(mutex);
            state[i] = EATING;
            sem_post(mutex);

            sleep(2);
```

```
            // eating
            printf("Philosopher %d takes fork %d and %d\n", i+1, (i+N-1)%N+1, i+1);
            printf("Philosopher %d is Eating\n", i+1);
            sem_post(condition[i]); // signal
      }
}

// pickup chopsticks
void pickup(int i)
{
      // update state to hungry
      sem_wait(mutex);
      state[i] = HUNGRY;
      printf("Philosopher %d is Hungry\n", i+1);
      sem_post(mutex);
      // eat if neighbours are not eating
      test(i);
      // if unable to eat wait to be signalled
      sem_wait(condition[i]);
      sleep(1);
}

// put down chopsticks
void putdown(int i)
{

      // update state to thinking
      sem_wait(mutex);
      state[i] = THINKING;
      sem_post(mutex);
      printf("Philosopher %d putting fork %d and %d down\n", i+1, (i+N-1)%N+1, i+1);
      printf("Philosopher %d is thinking\n", i+1);

      test((i+N-1) % N);
      test((i+1) % N);
}

// start philosopher action
void startPhilosopherAction(int num){
      while (1)
      {
            sleep(1);
            pickup(num);
            sleep(0);
            putdown(num);
      }
```

```
}

int main()
{

	printf("Enter value of N : ");
	scanf("%d", &N);

	// allocate memory
	state = mmap(NULL, sizeof(enum Status) * N, PROT_READ | PROT_WRITE, MAP_SHARED
| MAP_ANONYMOUS, -1, 0);
	condition = mmap(NULL, sizeof(sem_t *) * N, PROT_READ | PROT_WRITE, MAP_SHARED
| MAP_ANONYMOUS, -1, 0);

	for (int i = 0; i < N; i++){
		state[i] = THINKING;
	}

	// initializemutex semaphore
	mutex = sem_open("mutex56", O_CREAT, 0777, 1);
	// initialize condition semaphore
	for (int i = 0; i < N; i++){
		condition[i] = sem_open(mutex_key_from_no(i), O_CREAT, 0777, 1);
	}

	int id = fork();
	if (id == 0){
		// Fork childs
		for (int i = 0; i < N; i++){
			id = fork();
			if (id == 0){
				startPhilosopherAction(i);
				exit(1);
			}
		}
		for (int i = 0; i < N; i++){
			wait(NULL);
		}
		exit(1);
	}else{
		wait(NULL);
	}

	return 0;
}
```

Output -

```
Enter value of N : 5
Philosopher 1 is Hungry
Philosopher 2 is Hungry
Philosopher 3 is Hungry
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 2 is Hungry
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 5 is Hungry
Philosopher 4 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 3 is Hungry
Philosopher 5 is thinking
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 1 is Hungry
Philosopher 1 takes fork 5 and 1
Philosopher 4 is Eating
Philosopher 1 is Eating
Philosopher 3 is Hungry
. . .
```

7. Write a program that will find out whether a system is in safe state or not with following specifications:

Command line input: name of a file - The file contains the initial state of the system as given below:
#no of resources 4 #no of instances of each resource 2 4 5 3
#no of processes 3 #no of instances of each resource that each process needs in its lifetime 1 1 1 1, 2 3 1 2, 2 2 1 3
The program waits to accept a resource allocation request to be supplied by the user or read from another file:
For example: 0 1 0 1 1 indicates that p0 has requested allocation of 1 instance of R0, R2 and R3 each.
Your program should declare the result:

1. Should this request be granted?
2. If your answer is yes, print the safe sequence in which all remaining needs can be granted one by one and also grant the request. If the requesting process's need is NIL, the program internally releases all its resources. Go back to accept another request till all processes finish with all their needs.

Testing:

a. Generate possible request sequences of each process.
b. Each such sequence must satisfy the maximum requirements of the process.

```
import java.io.*;
import java.util.Arrays;

class BankersUtil{
    int no_of_process;
    int no_of_resources;
    int safe_sequence[];
    int available_resources[];
    int allocated_resources[][];
    int max_resources_request[][];
    int need_resources[][];

    BankersUtil(int no_of_process, int no_of_resources){
        this.no_of_process = no_of_process;
        this.no_of_resources = no_of_resources;
        this.safe_sequence = new int[no_of_process];
        this.available_resources = new int[no_of_resources];
        this.allocated_resources = new int[no_of_process][no_of_resources];
        this.max_resources_request = new int[no_of_process][no_of_resources];
        this.need_resources = new int[no_of_process][no_of_resources];
    }

    public void setNoOfInstanceForResource(int resource_no, int no_of_instances){
        this.available_resources[resource_no] = no_of_instances;
    }

    public void setMaxResourceForProcess(int process_no, int resource_no, int
no_of_instances){
```

```
        this.max_resources_request[process_no][resource_no] = no_of_instances;
    }

    public void calculateNeedMatrix(){
        for(int i=0;i<no_of_process;i++){
            for(int j=0;j<no_of_resources;j++){
                need_resources[i][j] = max_resources_request[i][j] -
allocated_resources[i][j];
            }
        }
    }

    public void displayData(){
        System.out.println("Process No\tAllocated\tMax\t\tNeed\t\tAvailable");
        System.out.println("----------\t---------\t--------\t--------\t---------");
        for(int i=0;i<no_of_process;i++){
            System.out.print("P"+i+"\t\t");
            for(int j=0;j<no_of_resources;j++){
                System.out.print(allocated_resources[i][j]+" ");
            }
            System.out.print("\t");
            for(int j=0;j<no_of_resources;j++){
                System.out.print(max_resources_request[i][j]+" ");
            }
            System.out.print("\t");
            for(int j=0;j<no_of_resources;j++){
                System.out.print(need_resources[i][j]+" ");
            }
            System.out.print("\t");
            if(i==0){
                for(int j=0;j<no_of_resources;j++){
                    System.out.print(available_resources[j]+" ");
                }
            }
            System.out.println();
        }
        System.out.println();
    }

    public void displaySafeSequence(){
        System.out.print("Safe Sequence: ");
        for(int i=0;i<no_of_process;i++){
            System.out.print("P"+safe_sequence[i]+" ");
        }
        System.out.println();
    }
```

```
    public void submitRequest(int process_no, int[] request){
        for(int i=0;i<no_of_resources;i++){
            if(request[i] > need_resources[process_no][i]){
                System.out.println("Request cannot be granted as it exceeds the need
of process P"+process_no);
                return;
            }
            if(request[i] > available_resources[i]){
                System.out.println("Request cannot be granted as it exceeds the
available resources");
                return;
            }
        }

        // create copy of available resources, allocated resources and need resources
        int[] available_resources_copy = new int[no_of_resources];
        int[][] allocated_resources_copy = new int[no_of_process][no_of_resources];
        int[][] need_resources_copy =  new int[no_of_process][no_of_resources];

        // copy data
        for(int i=0;i<no_of_resources;i++){
            available_resources_copy[i] = available_resources[i];
        }
        for(int i=0;i<no_of_process;i++){
            for(int j=0;j<no_of_resources;j++){
                allocated_resources_copy[i][j] = allocated_resources[i][j];
                need_resources_copy[i][j] = need_resources[i][j];
            }
        }

        // grant request
        for(int i=0;i<no_of_resources;i++){
            available_resources_copy[i] -= request[i];
            allocated_resources_copy[process_no][i] += request[i];
            need_resources_copy[process_no][i] -= request[i];
        }

        // create variables
        boolean[] finished = new boolean[no_of_process];
        int[] work = Arrays.copyOf(available_resources_copy,
available_resources_copy.length);
        int[] safe_sequence_copy = new int[no_of_process];

        // calculate safe sequence
```

```
        int index = 0;
        boolean flag = false;


        while(true){
            for(int i=0;i<no_of_process;i++){
                if(finished[i] == false){
                    boolean canfinish = true;
                    for(int j=0;j<no_of_resources;j++){
                        if(need_resources_copy[i][j] > work[j]){
                            canfinish = false;
                            break;
                        }
                    }

                    if(canfinish){
                        for(int j=0;j<no_of_resources;j++){
                            work[j] += allocated_resources_copy[i][j];
                        }
                        finished[i] = true;
                        safe_sequence_copy[index++] = i;
                        flag = true;
                    }
                }
            }
            if(!flag) break;
            flag = false;
        }

        // check if safe sequence is valid
        for(int i=0;i<no_of_process;i++){
            if(finished[i] == false){
                System.out.println("Request cannot be granted as it will lead to
unsafe state");
                return;
            }
        }

        System.out.println("Request can be granted as it will lead to safe state");
        System.out.print("Safe Sequence: ");
        for(int i=0;i<no_of_process;i++){
            System.out.print("P"+safe_sequence_copy[i]+" ");
        }
        System.out.println();

        // update data
```

```
        for(int i=0;i<no_of_resources;i++){
            available_resources[i] -= request[i];
            allocated_resources[process_no][i] += request[i];
            need_resources[process_no][i] -= request[i];
        }

        // check if need of process becomes zero
        boolean need_becomes_zero = true;
        for(int i=0;i<no_of_resources;i++){
            if(need_resources[process_no][i] != 0){
                need_becomes_zero = false;
                break;
            }
        }

        // if need of process becomes zero, add process to safe sequence
        if(need_becomes_zero){
            System.out.println("Need of process P"+process_no+" becomes zero, so it
has finished execution");
            // free resources
            for(int i=0;i<no_of_resources;i++){
                available_resources[i] += allocated_resources[process_no][i];
                allocated_resources[process_no][i] = 0;
            }
        }
    }

}

public class Question {

  static int safe_sequence[];

  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader("input.txt"));

    int nr = Integer.parseInt(br.readLine());
    int np = Integer.parseInt(br.readLine());

    BankersUtil banker = new BankersUtil(np, nr);

    // Read no of instances for each resource
    String[] resources = br.readLine().split(" ");
    for(int i=0;i<nr;i++){
        banker.setNoOfInstanceForResource(i, Integer.parseInt(resources[i]));
    }
```

```
    // Read max need for each process
    for(int i=0;i<np;i++){
        String[] _tmp = br.readLine().split(" ");
        for(int j=0;j<nr;j++){
            banker.setMaxResourceForProcess(i, j, Integer.parseInt(_tmp[j]));
        }
    }

    banker.calculateNeedMatrix();
    banker.displayData();
    while(true){
        System.out.println("Enter process number and request for each resource");
        BufferedReader br1 = new BufferedReader(new InputStreamReader(System.in));
        String[] _tmp = br1.readLine().split(" ");
        int process_no = Integer.parseInt(_tmp[0]);
        int[] request = new int[nr];
        for(int i=0;i<nr;i++){
            request[i] = Integer.parseInt(_tmp[i+1]);
        }
        banker.submitRequest(process_no, request);
        banker.displayData();
    }
  }
}
```

Output -

```
Process No     Allocated      Max          Need         Available
----------     ---------      --------     --------     ---------
P0             0 0 0 0        1 1 1 1      1 1 1 1      2 4 5 3
P1             0 0 0 0        2 3 1 2      2 3 1 2
P2             0 0 0 0        2 2 1 3      2 2 1 3

Enter process number and request for each resource
0 1 0 1 1
Request can be granted as it will lead to safe state
Safe Sequence: P0 P1 P2
Process No     Allocated      Max          Need         Available
----------     ---------      --------     --------     ---------
P0             1 0 1 1        1 1 1 1      0 1 0 0      1 4 4 2
P1             0 0 0 0        2 3 1 2      2 3 1 2
P2             0 0 0 0        2 2 1 3      2 2 1 3

Enter process number and request for each resource
```

1 2 0 2 2
Request cannot be granted as it exceeds the available resources

Process No	Allocated	Max	Need	Available
P0	1 0 1 1	1 1 1 1	0 1 0 0	1 4 4 2
P1	0 0 0 0	2 3 1 2	2 3 1 2	
P2	0 0 0 0	2 2 1 3	2 2 1 3	

Enter process number and request for each resource
1 1 0 1 1
Request can be granted as it will lead to safe state
Safe Sequence: P0 P1 P2

Process No	Allocated	Max	Need	Available
P0	1 0 1 1	1 1 1 1	0 1 0 0	0 4 3 1
P1	1 0 1 1	2 3 1 2	1 3 0 1	
P2	0 0 0 0	2 2 1 3	2 2 1 3	

Enter process number and request for each resource
2 0 2 1 1
Request cannot be granted as it will lead to unsafe state

Process No	Allocated	Max	Need	Available
P0	1 0 1 1	1 1 1 1	0 1 0 0	0 4 3 1
P1	1 0 1 1	2 3 1 2	1 3 0 1	
P2	0 0 0 0	2 2 1 3	2 2 1 3	