**Question 2**

Write a multithreaded program that generates the Fibonacci series.

***Program:***

import java.util.Scanner;

class Fibonacci extends Thread{

    int n;

    int series[];

    public Fibonacci(int n){

        this.n = n;

        series = new int[n];

    }

public void run(){

    series[0] = 0;

    series[1] = 1;

    for(int i =2;i<n;i++){

        series[i] = series[i-1] + series[i-2];

    }

}

void printSeries(){

    for(int i =0;i<n;i++)

        System.out.print(series[i] + " ");

    }

}

public class FibThread {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number");

        int n = sc.nextInt();

        Fibonacci fib = new Fibonacci(n);

        fib.start();

        try {

            Thread.sleep(1000);

        } catch (InterruptedException e) {

        // TODO Auto-generated catch block

            e.printStackTrace();

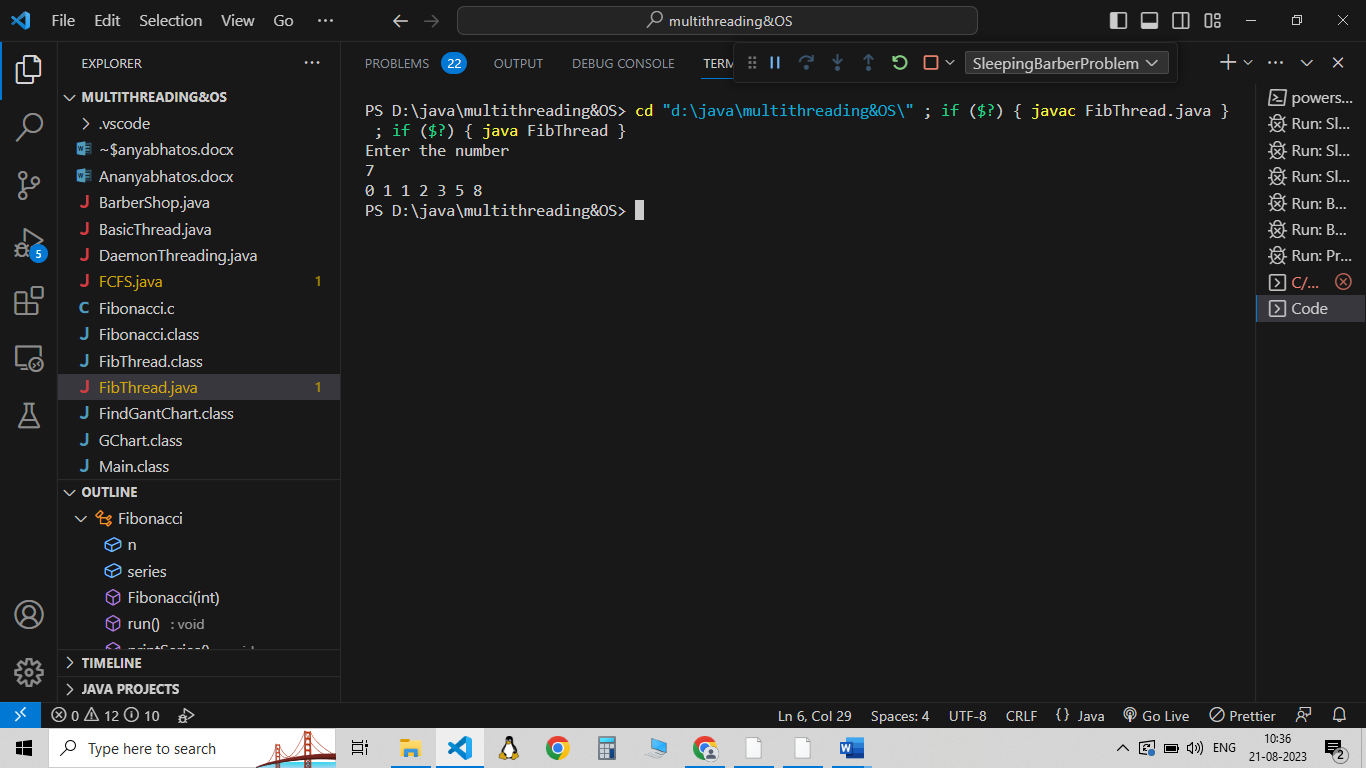
        }

        fib.printSeries();

    }

}

***Output:***



**Question 3**

Write a program that demonstrates the basic Threads API for constructing a multithreaded

program that calculates the summation of a nonnegative integer in a separate thread.

***Program:***

import java.util.Scanner;

class Sum extends Thread{

    int n,sum;

    public Sum(int n){

        this.n = n;

        sum =0;

    }

    public void run(){

        for(int i=0;i<=n;i++)

            sum += i;    }

    public void printSum(){

        Thread.currentThread().setName("Summation Thread");

        System.out.println("Executing in Thread :" + Thread.currentThread().getName());

        System.out.println("The sum of all non negative numbers upto "+ n + " is " + sum);

    }

}

public class Summation {

    public static void main(String[] args) {

    // TODO Auto-generated method stub

        Thread.currentThread().setName("Main");

        System.out.println("Executing in Thread :" + Thread.currentThread().getName());

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the number");

        int n = sc.nextInt();

        Sum obj = new Sum(n);

        obj.start();

        try {

            Thread.sleep(1000);

        } catch (InterruptedException e) {

            e.printStackTrace();

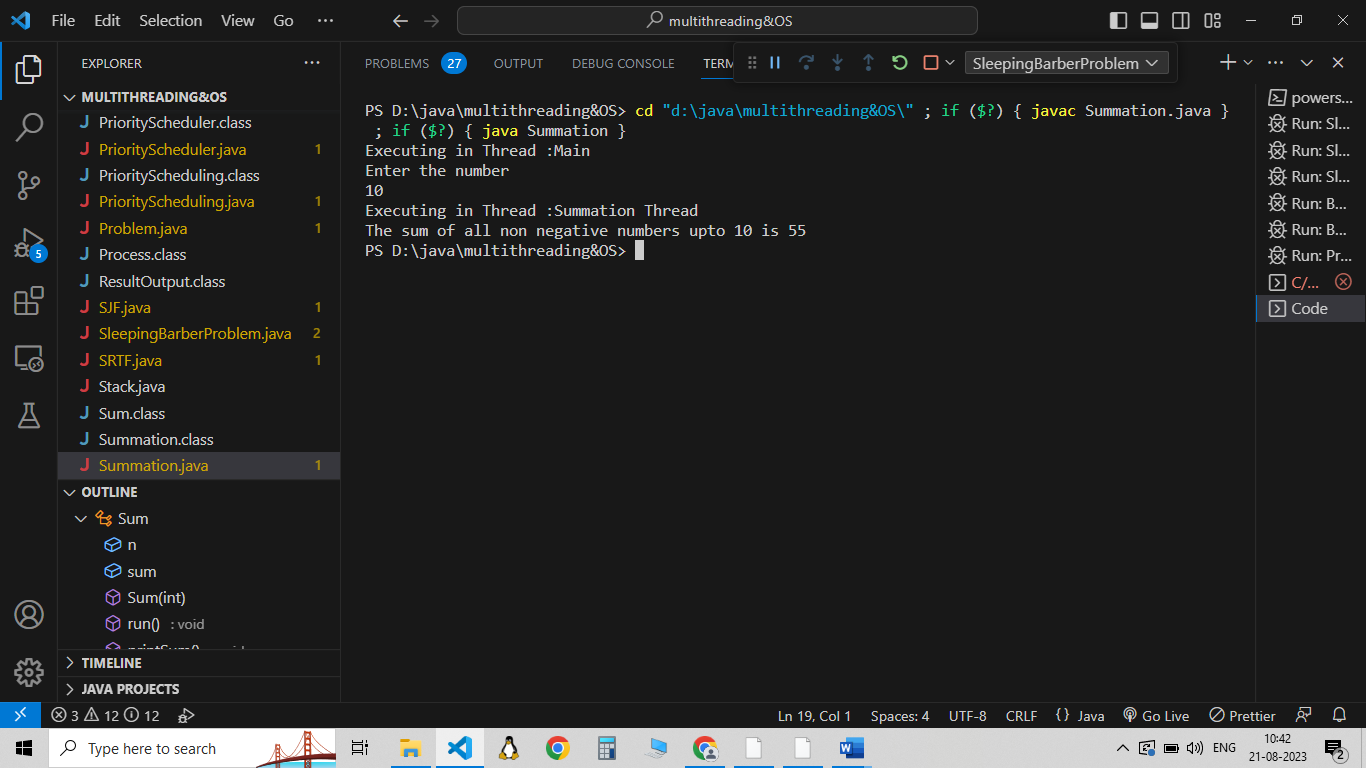
        }

        obj.printSum();

    }

}

***Output:***



**Question 4**

Given two matrices A and B is a matrix with M rows and k columns and matrix B contains k rows and N columns, the matrix product of A and B is matrix C, where C contains M rows and N columns. Implement using multithreading concept.

***Program:***

import java.util.Scanner;

class Multiplication implements Runnable{

    int [][] matA ;

    int [][] matB;

    int [][] matC;

    int max\_threads,i;

    Multiplication(int [][] matA,int [][] matB, int [][] matC, int max ,int i){

        this.matA = matA;

        this.matB = matB;

        this.matC = matC;

        max\_threads = max;

        this.i =i;

    }

    public void run(){

        int c = matA[0].length;

        for(int j =0;j<c;j++){

            for(int k =0;k<c;k++){

                matC[i][j] += matA[i][k]\*matB[k][j];

            }

        }

    }

    public int[][] input(int [][] matrix){

        Scanner sc = new Scanner(System.in);

        int n = matrix.length;

        int m = matrix[0].length;

        for(int i=0;i<n;i++){

            for(int j =0;j<m;j++){

                matrix[i][j] = sc.nextInt();

            }

        }

        return matrix;

    }

    public void output(){

        int n = matC.length;

        int m = matC[0].length;

        for(int i=0;i<n;i++){

            for(int j =0;j<m;j++){

                System.out.print(matC[i][j] + " ");

            }

            System.out.println();

        }

    }

}

public class MatrixMultiplication {

    public static void main(String[] args) {

    // TODO Auto-generated method stub

        Scanner sc = new Scanner(System.in);

        int r1,c1,r2,c2;

        System.out.println("Enter the no of rows and column of matrix A");

        r1 = sc.nextInt();

        c1 = sc.nextInt();

        int [][] matA = new int[r1][c1];

        System.out.println("Enter the no of rows and column of matrix B");

        r2 = sc.nextInt();

        c2 = sc.nextInt();

        int [][] matB = new int[r2][c2];

        int [][] matC = new int[r1][c2];

        Multiplication obj = new Multiplication(matA, matB, matC, r1, 0);

        System.out.println("Enter the elements of matrix A");

        matA = obj.input(matA);

        System.out.println("Enter the elements of matrix B");

        matB = obj.input(matB);

        Thread threads[] = new Thread[r1];

        for(int i =0;i<r1;i++){

            threads[i] = new Thread(new Multiplication(matA,matB,matC,r1,i));

            threads[i].start();

        }

        for(int i =0;i<r1;i++){

            try {

                threads[i].join();

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

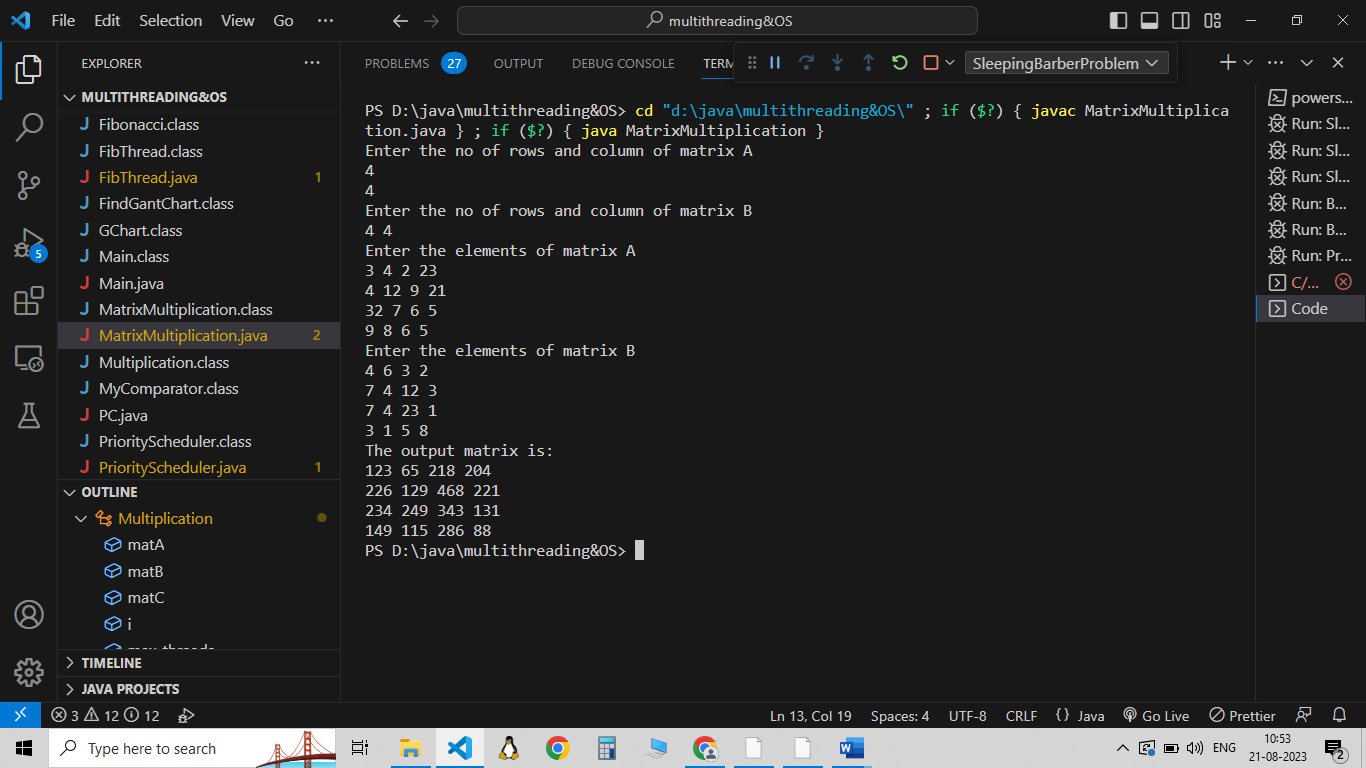
        }

        System.out.println("The output matrix is:");

        obj.output();

    }

}

***Output:***

**Question 5**

Develop a program to generate and print the n Fibonacci Series in the child process using fork(). Perform the necessary error checking to ensure that a non negative sequence number n will be provided in the command line.

***Program:***

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

void generate\_fibonacci(int n) {

    int a = 0, b = 1;

    int i;

    for (i = 0; i < n; i++) {

        printf("%d ", a);

        int next = a + b;

        a = b;

        b = next;

    }

    printf("\n");

}

int main(int argc, char\* argv[]) {

    if (argc != 2) {

        printf("Usage: %s <non-negative\_integer>\n", argv[0]);

        return 1;

    }

    int n = atoi(argv[1]);

    if (n < 0) {

        printf("Error: The input should be a non-negative integer.\n");

        return 1;

    }

    pid\_t pid = fork();

    if (pid < 0) {

        printf("Error: Forking failed.\n");

        return 1;

    }

    else if (pid == 0) {

        // Child process

        printf("Child process (PID: %d) Fibonacci Series: ", getpid());

        generate\_fibonacci(n);

    }

    else {

        // Parent process

        printf("Parent process (PID: %d) waiting for the child process (PID: %d) to finish...\n", getpid(), pid);

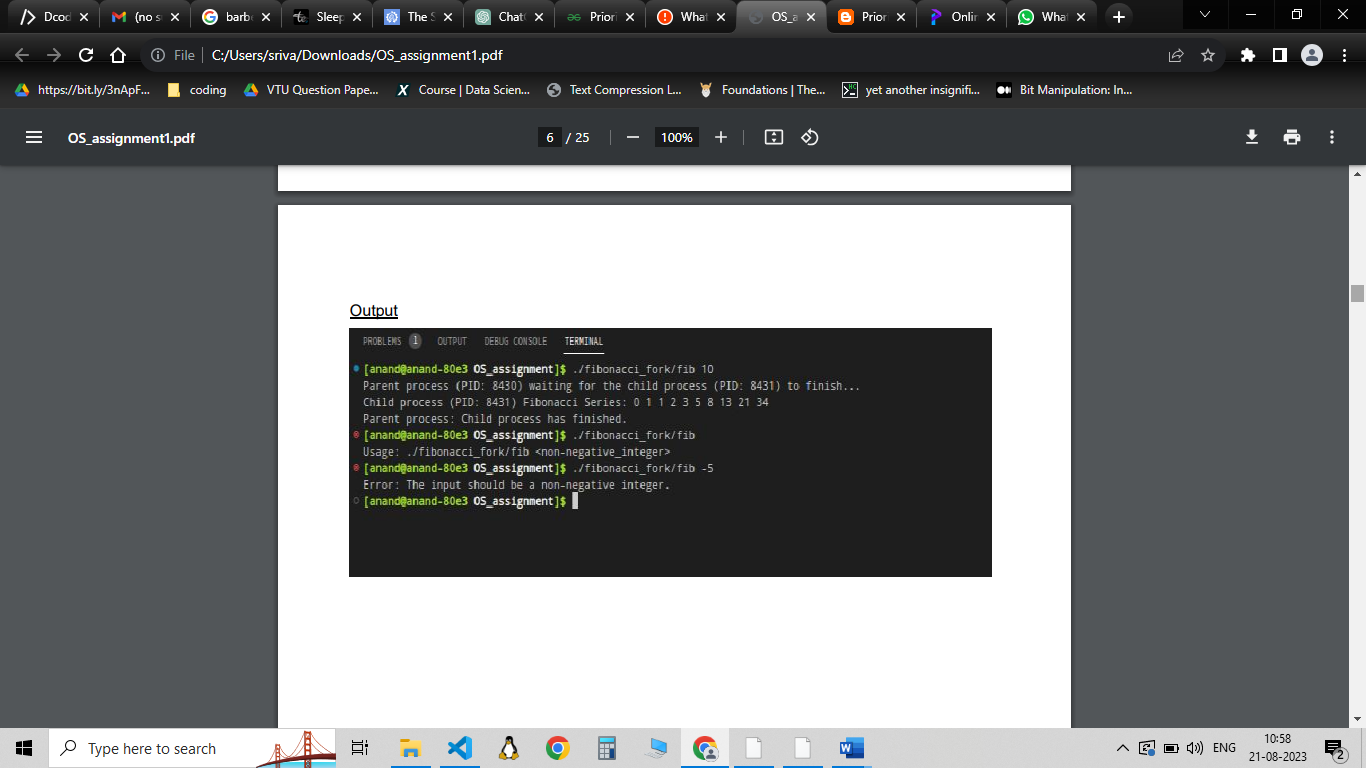
        wait(NULL);

        printf("Parent process: Child process has finished.\n");

    }

    return 0;

}

***Output:***

**Question 6**

Develop a Java /Python program to implement FCFS scheduling algorithm.

***Program:***

import java.util.Scanner;

class FCFS {

    static void findWaitingTime(int processes[], int n,

            int bt[], int wt[]) {

        // waiting time for first process is 0

        wt[0] = 0;

        // calculating waiting time

        for (int i = 1; i < n; i++) {

            wt[i] = bt[i - 1] + wt[i - 1];

        }

    }

    // Function to calculate turn around time

    static void findTurnAroundTime(int processes[], int n,

            int bt[], int wt[], int tat[]) {

        // calculating turnaround time by adding

        // bt[i] + wt[i]

        for (int i = 0; i < n; i++) {

            tat[i] = bt[i] + wt[i];

        }

    }

    //Function to calculate average time

    static void findavgTime(int processes[], int n, int bt[]) {

        int wt[] = new int[n], tat[] = new int[n];

        int total\_wt = 0, total\_tat = 0;

        //Function to find waiting time of all processes

        findWaitingTime(processes, n, bt, wt);

        //Function to find turn around time for all processes

        findTurnAroundTime(processes, n, bt, wt, tat);

        //Display processes along with all details

        System.out.printf("Processes BurstTime WaitingTime TurnAroundTime\n");

        // Calculate total waiting time and total turn

        // around time

        for (int i = 0; i < n; i++) {

            total\_wt = total\_wt + wt[i];

            total\_tat = total\_tat + tat[i];

            System.out.printf(" %d ", (i + 1));

            System.out.printf("\t   %d ", bt[i]);

            System.out.printf("\t\t%d", wt[i]);

            System.out.printf("\t\t%d\n", tat[i]);

        }

        float s = (float)total\_wt /(float) n;

        float t = (float)total\_tat / (float)n;

        System.out.printf("Average waiting time = %.2fms", s);

        System.out.printf("\n");

        System.out.printf("Average turn around time = %.2fms ", t);

    }

    // Driver code

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the no of processes");

        int n = sc.nextInt();

        //process id's

        int processes[] = new int[n];

        //Burst time of all processes

        int bursttime[] = new int[n];

        System.out.println("Enter the process no and burst time in the order they arrive:");

        for(int i =0;i<n;i++){

            processes[i] = sc.nextInt();

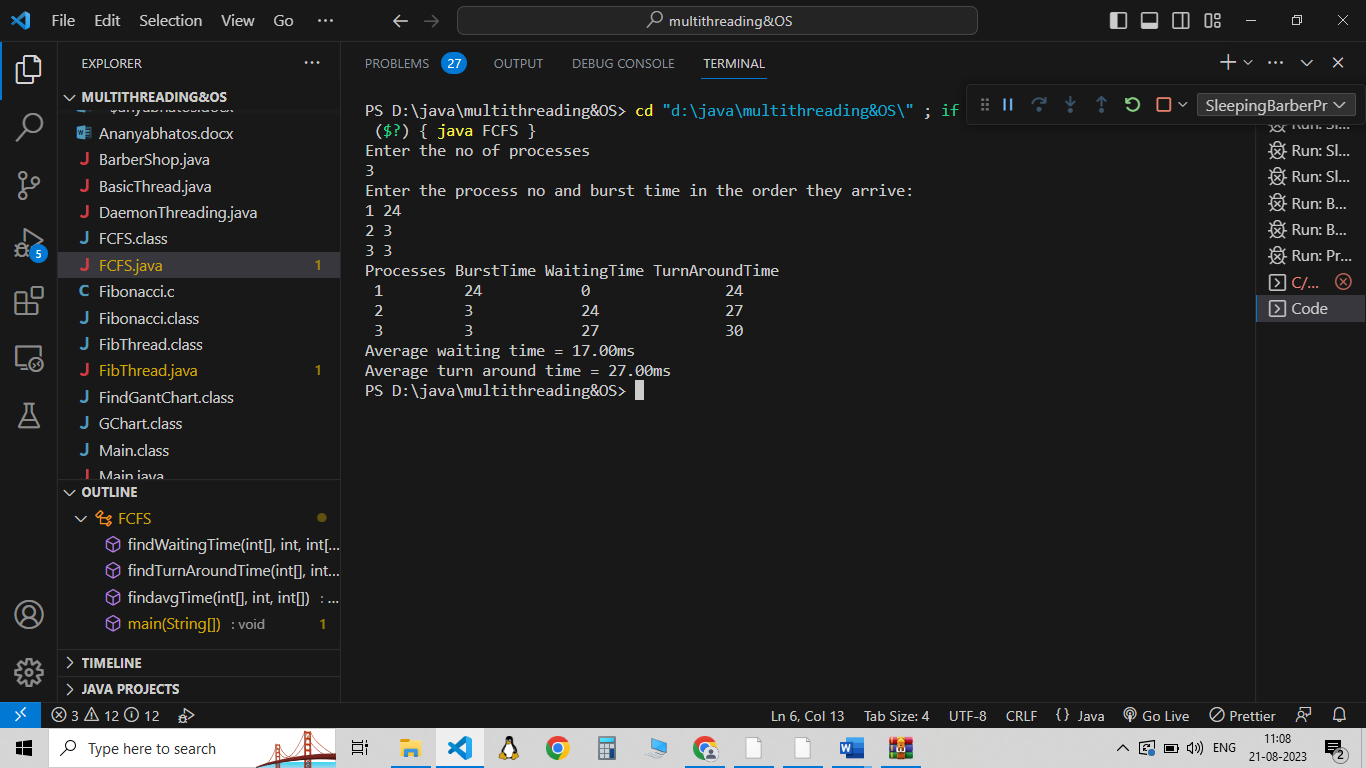
            bursttime[i] = sc.nextInt();

        }

        findavgTime(processes, n, bursttime);

    }

}

***Output:***

**Question 7**

Develop a Java /Python program to implement non- preemptive SJF scheduling Algorithm

***Program:***

import java.util.\*;

class Process{

  int id,bt, art;

  public Process(int id, int bt, int art){

    this.id= id;

    this.bt= bt;

    this.art = art;

  }

}

 class SJF

 {

   public static void main(String args[])

   {

    Scanner sc = new Scanner(System.in);

    System.out.println("Enter the no of processes! ");

    int n = sc. nextInt() ;

    Process process[] = new Process[n];

    int wt[] = new int[n];

    int tat[] = new int[n];

    System.out.println(" Enter the process no, burst time and arrival time");

    for(int i=0;i<n;i++){

      int id= sc.nextInt();

      int bt = sc.nextInt();

      int art = sc.nextInt();

      process[i]= new Process(id, bt, art);

    }

    findWaitingTime(process,n,wt);

    findtat(process,n,wt,tat);

    int totalWaitingTime =0, totalTAT= 0;

    System.out.printf("Process BurstTime ArrivalTime WaitingTime TurnAroundTime\n");

    for (int i = 0; i < n; i++) {

      totalWaitingTime += wt[i];

            totalTAT += tat[i];

      System.out.printf(" %d\t", (i + 1));

      System.out.printf("   %d ", process[i].bt);

      System.out.printf("\t\t%d ", process[i].art);

      System.out.printf("\t\t%d", wt[i]);

      System.out.printf("\t\t%d\n", tat[i]);

    }

    float s = (float)totalWaitingTime /(float) n;

    float t = (float)totalTAT / (float)n;

    System.out.printf("Average waiting time = %.2fms", s);

    System.out.printf("\n");

    System.out.printf("Average turn around time = %.2fms ",t);

   }

   public static void findWaitingTime(Process[] process, int n, int[] wt){

     int [] rt = new int[n];

     boolean check = false;

     int min= Integer.MAX\_VALUE,shortest=0,t=0,finishTime,complete=0;

     for(int i=0;i<n;i++){

       rt[i]= process[i].bt;

     }

    while(complete != n){

     for(int i=0;i<n;i++){

       if(rt[i] <= min && rt[i]>0 && process[i].art <=t){

         check=true;

         min = rt[i];

         shortest = i;

       }

     }

       if(check == false){

         t++;

         continue;

       }

       finishTime = t+ rt[shortest];

       complete++;

       wt[shortest]= finishTime - process[shortest].bt - process[shortest].art;

       System.out.println(wt[shortest] + " " + finishTime);

       rt[shortest]=0;

       check = false;

       min = Integer.MAX\_VALUE;

       t = finishTime;

     }

   }

   public static void findtat(Process[] process, int n, int[] wt,int[] tat){

     for(int i=0;i<n;i++){

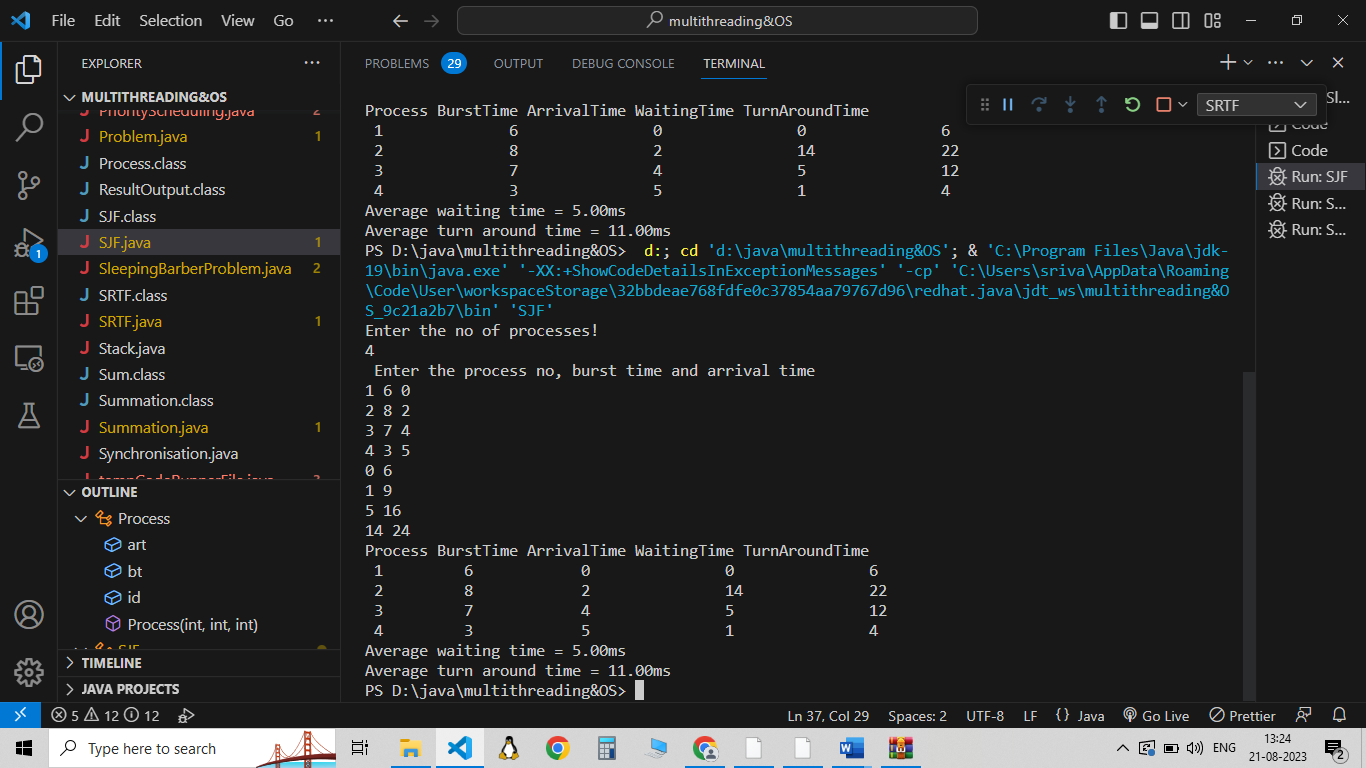
       tat[i] = wt[i]+ process[i].bt;

     }

   }

 }

***Output:***



**Question 8**

Develop a Java /Python program to implement SRTF scheduling algorithm.

***Program:***

import java.util.Scanner;

class Process{

    int pid,bt,art;

    Process(int pid,int bt,int art){

        this.pid = pid;

        this.bt = bt;

        this.art = art;

    }

}

public class SRTF {

    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the no of process");

        int n = sc.nextInt();

        Process[] process = new Process[n];

        int wt[] = new int[n];

        int tat[] = new int[n];

        int totalWaitingTime =0, totalTAT= 0;

        System.out.println("Enter the Process id , burst time and Arrival Time");

        for(int i=0;i<n;i++){

            int pid = sc.nextInt();

            int bt = sc.nextInt();

            int art = sc.nextInt();

            process[i] = new Process(pid,bt,art);

        }

        findWaitingTime(process,wt,n);

        findTurnAroundTime(process, wt,tat, n);

        System.out.printf("Processes BurstTime ArrivalTime WaitingTime TurnAroundTime\n");

        // Calculate total waiting time and total turn

        // around time

        for (int i = 0; i < n; i++) {

            totalWaitingTime += wt[i];

            totalTAT += tat[i];

            System.out.printf(" %d ", (i + 1));

            System.out.printf("\t   %d ", process[i].bt);

            System.out.printf("\t   %d ", process[i].art);

            System.out.printf("\t\t%d", wt[i]);

            System.out.printf("\t\t%d\n", tat[i]);

        }

        float s = (float)totalWaitingTime /(float) n;

        float t = (float)totalTAT / (float)n;

        System.out.printf("Average waiting time = %.2fms", s);

        System.out.printf("\n");

        System.out.printf("Average turn around time = %.2fms ", t);

}

    public static void findTurnAroundTime(Process[] process, int[] wt,int[] tat,int n){

        for(int i=0;i<n;i++)

            tat[i] = process[i].bt +wt[i];

    }

    private static void findWaitingTime(Process[] process, int[] wt,int n) {

        boolean check = false;

        int finish\_time , shortest =0 ,minm = Integer.MAX\_VALUE,t=0,complete =0;

        //To store the remaining time

        int [] rt = new int[n];

        for(int i =0;i<n;i++)

            rt[i] = process[i].bt ;

        while(complete != n){

            //To find the shortest process out of all the processes that have arrived

            for(int i=0;i<n;i++){

                if(process[i].art<=t && rt[i]<minm && rt[i] >0){

                    minm = rt[i];

                    shortest = i;

                    check = true;

                }

            }

            //If no process has arrived at the given time

            if(check == false){

                t++;

                continue;

            }

            //Update remaining time and minmalprocess

            rt[shortest]--;

            minm = rt[shortest];

            if(minm == 0){

                minm = Integer.MAX\_VALUE;

            }

            //remaining time of given process is zero,so process complete

            if(rt[shortest]==0){

                complete++;

                check = false;

                finish\_time = t+1;

                wt[shortest] = finish\_time - process[shortest].bt - process[shortest].art;

                if(wt[shortest]<0)

                    wt[shortest] =0;

            }

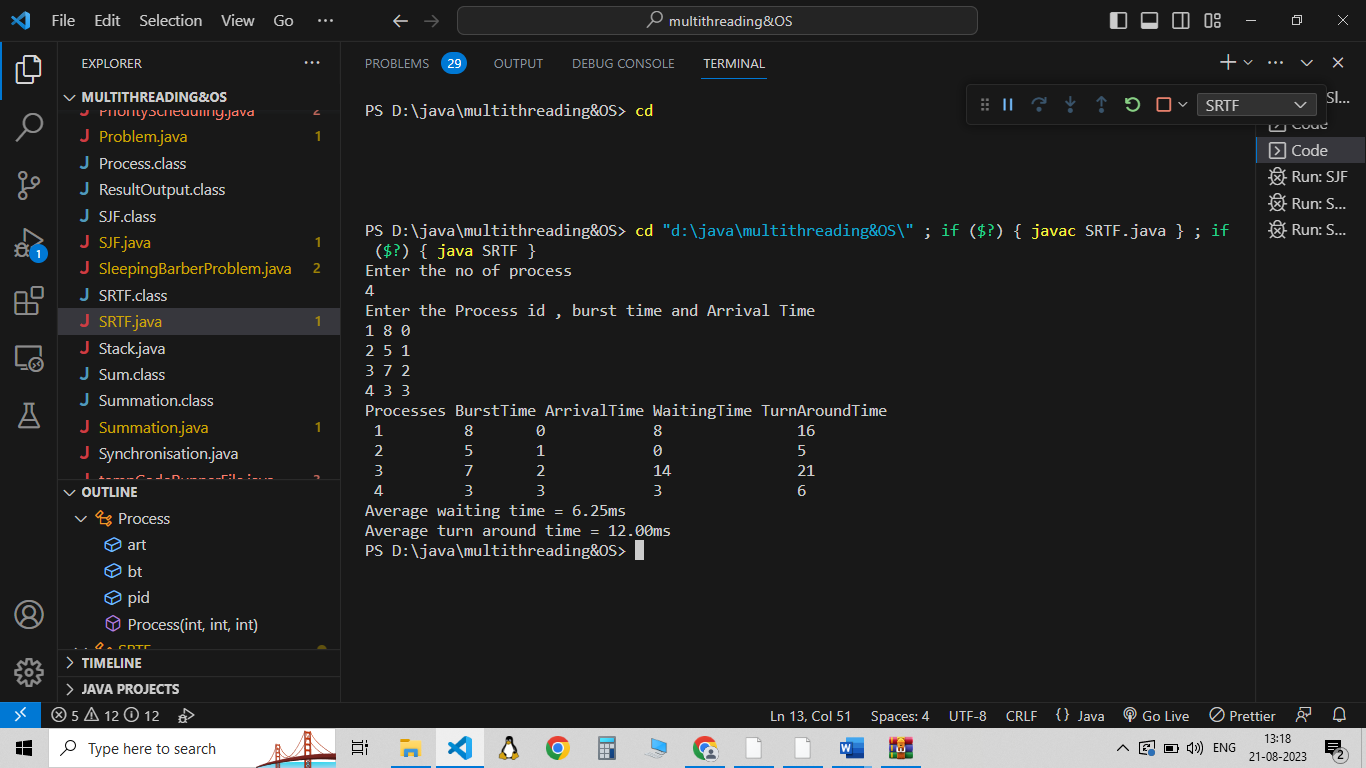
            t++;

        }

    }

}

***Output:***



**Question 9**

Develop a Java /Python program to implement priority scheduling algorithm for a given set of process, burst time and their priority.

***Program:***

import java.util.\*;

class Process {

    int at, bt, pri, pno;

    Process(int pno, int at, int bt, int pri) {

        this.pno = pno;

        this.pri = pri;

        this.at = at;

        this.bt = bt;

    }

}

class GChart {

    int pno, stime, ctime, wtime, ttime;

}

class MyComparator implements Comparator<Process> {

    public int compare(Process p1, Process p2) {

        if (p1.at < p2.at)

            return -1;

        else if (p1.at == p2.at && p1.pri > p2.pri)

            return -1;

        else

            return 1;

    }

}

class FindGantChart {

    void findGc(LinkedList<Process> queue) {

        int time = 0;

        TreeSet<Process> prique = new TreeSet<>(new MyComparator());

        LinkedList<GChart> result = new LinkedList<>();

        while (!queue.isEmpty())

            prique.add(queue.removeFirst());

        Iterator<Process> it = prique.iterator();

        time = prique.first().at;

        while (it.hasNext()) {

            Process obj = it.next();

            GChart gc1 = new GChart();

            gc1.pno = obj.pno;

            gc1.stime = time;

            time += obj.bt;

            gc1.ctime = time;

            gc1.ttime = gc1.ctime - obj.at;

            gc1.wtime = gc1.ttime - obj.bt;

            result.add(gc1);

        }

        new ResultOutput(result);

    }

}

class ResultOutput {

    ResultOutput(LinkedList<GChart> result) {

        int totalWaitingTime = 0;

        int totalTurnaroundTime = 0;

        System.out.println("\nProcess execution details:");

        System.out.println("Process\_no\tTurn\_Around\_Time\tWaiting\_Time");

        for (GChart gc : result) {

            totalWaitingTime += gc.wtime;

            totalTurnaroundTime += gc.ttime;

            System.out.println(gc.pno + "\t\t" + gc.ttime + "\t\t\t" + gc.wtime);

        }

        double averageWaitingTime = (double) totalWaitingTime / result.size();

        double averageTurnaroundTime = (double) totalTurnaroundTime / result.size();

        System.out.println("\nAverage Waiting Time is: " + averageWaitingTime);

        System.out.println("Average Turnaround Time is: " + averageTurnaroundTime);

    }

}

public class PriorityScheduler {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        LinkedList<Process> processes = new LinkedList<>();

        System.out.println("Enter the no of processes");

        int n = sc.nextInt();

        for(int i = 0; i < n; i++) {

            System.out.println("Enter the id, arrival time, burst time and priority for process " + i);

            int pid = sc.nextInt();

            int at = sc.nextInt();

            int bt = sc.nextInt();

            int pri = sc.nextInt();

            processes.add(new Process(pid, at, bt, pri));

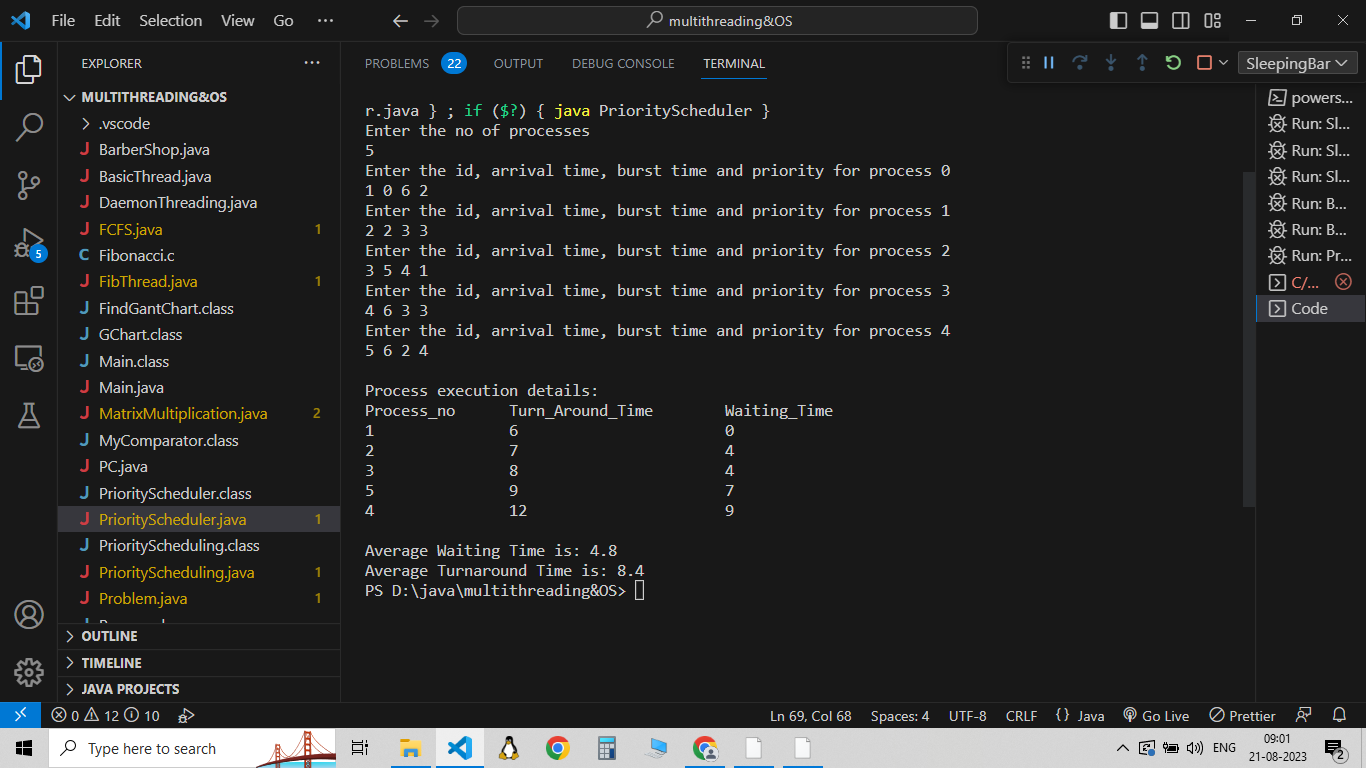
        }

        FindGantChart obj = new FindGantChart();

        obj.findGc(processes);

    }

***Output:***



**Question 12**

Consider a Sleeping-Barber Problem. A barbershop consists of a waiting room with n chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

***Program:***

import java.util.concurrent.\*;

class BarberShop {

    static final int CHAIRS = 4;

    static Semaphore barber = new Semaphore(0);

    static Semaphore customer = new Semaphore(0);

    static Semaphore mutex = new Semaphore(1);

    static int waitingCustomers = 0;

    static class Barber implements Runnable {

        public void run() {

            while (true) {

                try {

                    customer.acquire(); // Wait for a customer

                    mutex.acquire();

                    waitingCustomers--;

                    barber.release(); // Wake up the barber

                    mutex.release();

                    cutHair(); // Barber cuts hair

                } catch (InterruptedException ex) {

                    ex.printStackTrace();

                }

            }

        }

        public void cutHair() {

            System.out.println("Barber is cutting hair");

            try {

                Thread.sleep(2000);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

    }

    static class Customer implements Runnable {

        private int id;

        public Customer(int id) {

            this.id = id;

        }

        public void run() {

            try {

                mutex.acquire();

                if (waitingCustomers < CHAIRS) {

                    waitingCustomers++;

                    System.out.println("Customer " + id + " is waiting.");

                    customer.release(); // Wake up the barber if sleeping

                    mutex.release();

                    barber.acquire(); // Wait for the barber to finish

                    getHaircut(); // Get haircut

                } else {

                    mutex.release(); // No available chairs, leave

                    System.out.println("Customer " + id + " left due to no available chairs.");

                }

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

        public void getHaircut() {

            System.out.println("Customer " + id + " is getting a haircut.");

            try {

                Thread.sleep(3000);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

    }

    public static void main(String[] args) {

        Thread barberThread = new Thread(new Barber());

        barberThread.start();

        for (int i = 1; i <= 8; i++) {

            Thread customerThread = new Thread(new Customer(i));

            customerThread.start();

            try {

                Thread.sleep(1000); // Time between customer arrivals

            } catch (InterruptedException ex) {

                ex.printStackTrace();

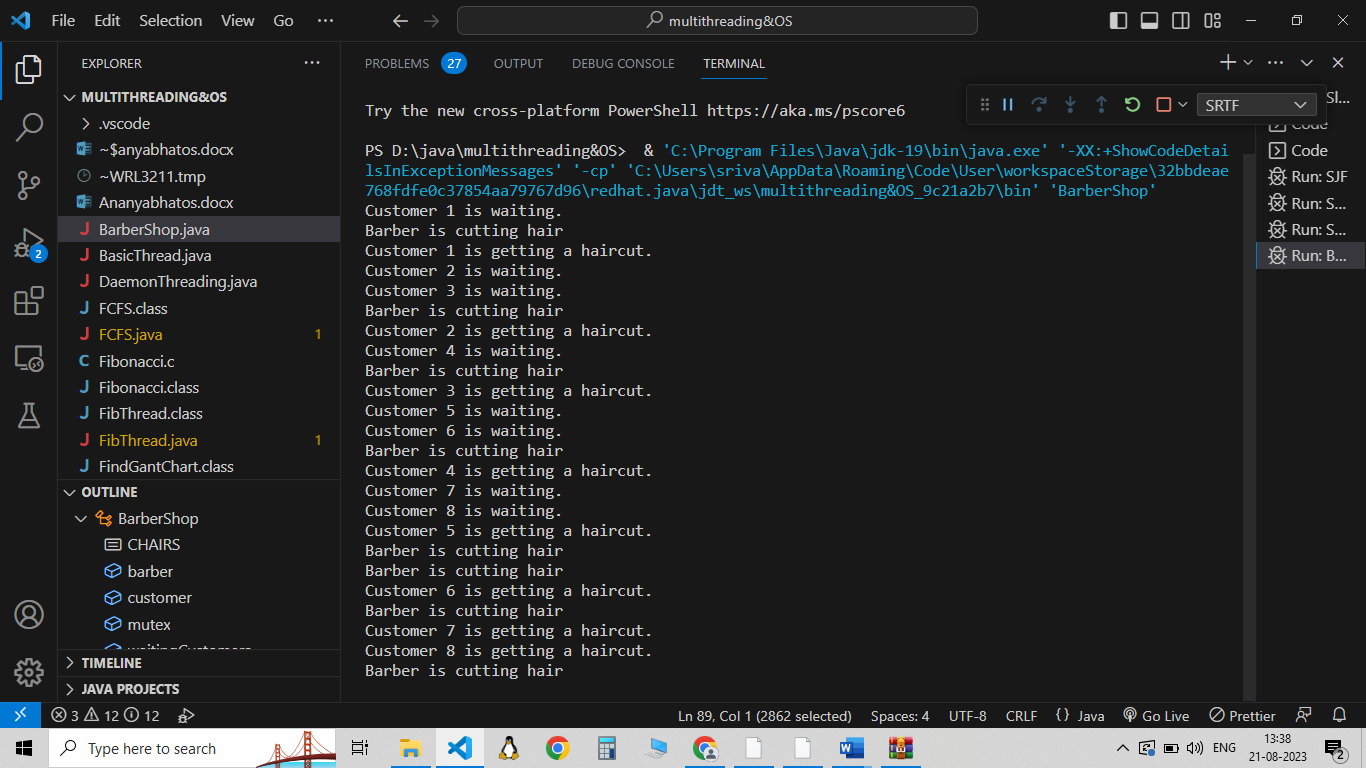
            }

        }

    }

}

***Output:***



**Question 13**

Calculate the number of times JSSATEB is printed.

#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main(){

fork();

fork();

fork();

printf(“JSSATEB\n”);

return 0;

}

***Output:***

The Fork system call is used for creating a new process in Linux, and Unix systems, which is called the child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. The number of times ‘JSSATEB’ is printed is equal to the number of processes created. Total Number of Processes = 2^n, where n is the number of fork system calls. So here n = 3, 2^3 = 8.



**Question 14**

Consider a set of 4 processes where three are smoker processes and an agent process. Each of the smoker processes will make a cigarette and smoke it. To make a cigarette requires tobacco, paper, and matches. Each smoker process has one of the three items. I.e., one process has tobacco, another has paper, and a third has matches. The agent has an infinite supply of all three. The agent places two of the three items on the table, and the smoker that has the third item makes the cigarette. Synchronize the processes and write the solution to it.

***Program:***

import java.util.concurrent.Semaphore;

public class Problem {

    public Problem() {

        currentState = TABLE\_EMPTY;

        //So that agent places the items first

        mutex = new Semaphore(1);

        Thread[] thread = new Thread[4];

        thread[0] = new Agent(this, 0);

        // Second parameter to Smoker constructor is the item the smoker \_has\_.

        thread[1] = new Smoker(this, PAPER, 1);

        thread[2] = new Smoker(this, TOBACCO, 2);

        thread[3] = new Smoker(this, MATCHES, 3);

        for (int i = 0; i < 4; i++)

            thread[i].start();

    }

    // Ignore the unused id parameters to these methods.  They were needed

    // in the busy-wait-free version from which this was ported.

    public void dispenseItems(int id) {

        int itemNotOnTable;

        while (true) {

            try {

                mutex.acquire();

                if (currentState == TABLE\_EMPTY) {

                    itemNotOnTable = 1 + (int)(Math.random() \* 3.0);

                    if (itemNotOnTable == 1) {

                        itemOnTable1 = 2;

                        itemOnTable2 = 3;

                    }

                    else if (itemNotOnTable == 2) {

                        itemOnTable1 = 1;

                        itemOnTable2 = 3;

                    }

                    else {

                        itemOnTable1 = 1;

                        itemOnTable2 = 2;

                    }

                    System.out.println("Agent puts items " + itemOnTable1

                               + " and " + itemOnTable2 + " on table");

                    currentState = ITEM\_AVAILABLE;

                }

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            } finally {

                mutex.release();

                Thread.yield(); // these are necessary!

            }

        }

    }

    public void waitForItem(int item, int id) {

        boolean gotItem = false;

        while (!gotItem) {

            try {

                mutex.acquire();

                if (currentState == ITEM\_AVAILABLE && (itemOnTable1 != item

                                                       && itemOnTable2 != item)) {

                    currentState = ITEM\_IN\_USE;

                    gotItem = true;

                }

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            } finally {

                mutex.release();

                Thread.yield();

            }

        }

    }

    public void finishSmoking(int id) {

        try {

            mutex.acquire();

            currentState = TABLE\_EMPTY;

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        } finally {

            mutex.release();

        }

    }

    public static void main(String[] args) {

        Problem p = new Problem();

    }

    protected Semaphore mutex;

    protected int currentState;

    public final int ITEM\_AVAILABLE = 1;

    public final int ITEM\_IN\_USE = 2;

    public final int TABLE\_EMPTY = 3;

    protected int itemOnTable1;

    protected int itemOnTable2;

    public final int PAPER = 1;

    public final int TOBACCO = 2;

    public final int MATCHES = 3;

}

class Agent extends Thread {

    private Problem problem;

    private int id;

    public Agent(Problem problem, int id) {

        this.problem = problem;

        this.id = id;

    }

    @Override

    public void run() {

        while (true) {

            try {

                Thread.sleep(1000); // Simulate some time before the agent dispenses items

                problem.dispenseItems(id);

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            }

        }

    }

}

class Smoker extends Thread {

    private Problem problem;

    private int item;

    private int id;

    public Smoker(Problem problem, int item, int id) {

        this.problem = problem;

        this.item = item;

        this.id = id;

    }

    @Override

    public void run() {

        while (true) {

            try {

                problem.waitForItem(item, id);

                System.out.println("Smoker " + id + " is now smoking.");

                Thread.sleep(2000); // Simulate smoking

                problem.finishSmoking(id);

            } catch (InterruptedException e) {

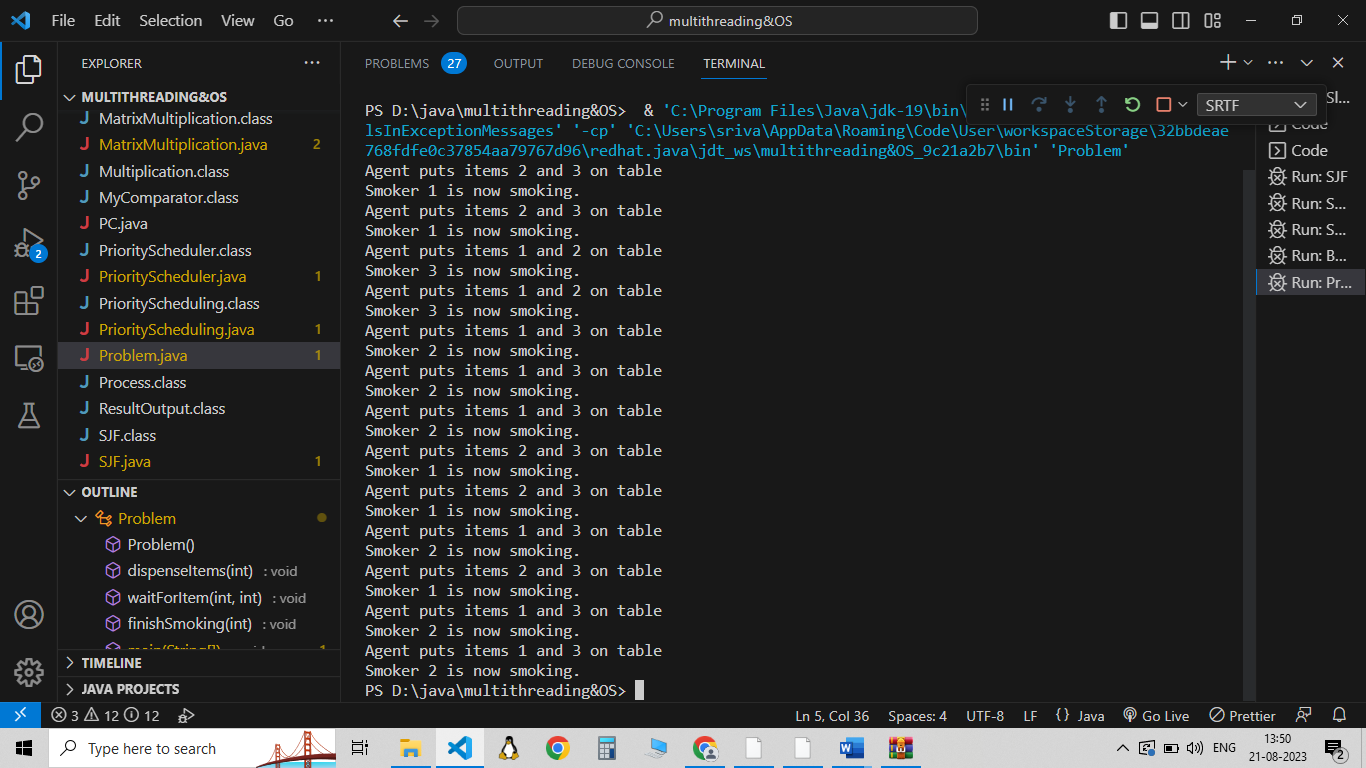
                Thread.currentThread().interrupt();

            }

        }

    }

}

***Output:***

**Question 15**

Develop a program to simulate producer consumer problem using semaphores

***Program:***

import java.util.concurrent.Semaphore;

import java.util.Random;

class SharedBuffer {

    private int buffersize;

    private int[] buffer;

    private int front,rear;

    public SharedBuffer(int size){

        buffersize = size;

        buffer = new int[size];

        front = rear = 0;

    }

    public void produce(int item){

        //Locks into the critical section and adds item into the buffer

        //instance of the curent class is used as the lock

        synchronized(this){

            buffer[front] = item;

            rear = (rear+1)%buffersize;

        }

        System.out.println("The produced item is :" + item);

    }

    public int consume(){

        int item ;

        synchronized(this){

            item = buffer[rear];

            front = (front+1)%buffersize;

        }

        System.out.println("The consumed item is :"+ item);

        return item;

    }

}

class Producer implements Runnable{

    private SharedBuffer buffer;

    private Semaphore producerSemaphore;

    private Semaphore consumerSemaphore;

    public Producer(SharedBuffer buffer , Semaphore producerSemaphore , Semaphore consumerSemaphore){

        this.buffer = buffer;

        this.producerSemaphore = producerSemaphore;

        this.consumerSemaphore = consumerSemaphore;

    }

    @Override

    public void run(){

        try{

            while(true){

                int item = produceItem();

                //blocks a permit(item) available

                producerSemaphore.acquire();

                buffer.produce(item);

                //increases the consumer permits

                consumerSemaphore.release();

                Thread.sleep(1000);

            }

        }

        catch(InterruptedException e){

            Thread.currentThread().interrupt();

        }

    }

    private int produceItem() {

        Random rand = new Random();

        return rand.nextInt(100); // Generate a random item

    }

}

class Consumer implements Runnable {

    private SharedBuffer buffer;

    private Semaphore producerSemaphore;

    private Semaphore consumerSemaphore;

    public Consumer(SharedBuffer buffer, Semaphore producerSemaphore, Semaphore consumerSemaphore) {

        this.buffer = buffer;

        this.producerSemaphore = producerSemaphore;

        this.consumerSemaphore = consumerSemaphore;

    }

    @Override

    public void run() {

        try {

            while (true) {

                consumerSemaphore.acquire();

                buffer.consume();

                producerSemaphore.release();

            }

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

}

public class PC {

    public static void main(String[] args){

        int buffersize = 5;

        SharedBuffer buffer = new SharedBuffer(buffersize);

        //to ensure that producer produces first

        Semaphore producerSemaphore = new Semaphore(buffersize);

        Semaphore consumerSemaphore = new Semaphore(0);

        Thread producerThread  = new Thread(new Producer(buffer,producerSemaphore,consumerSemaphore));

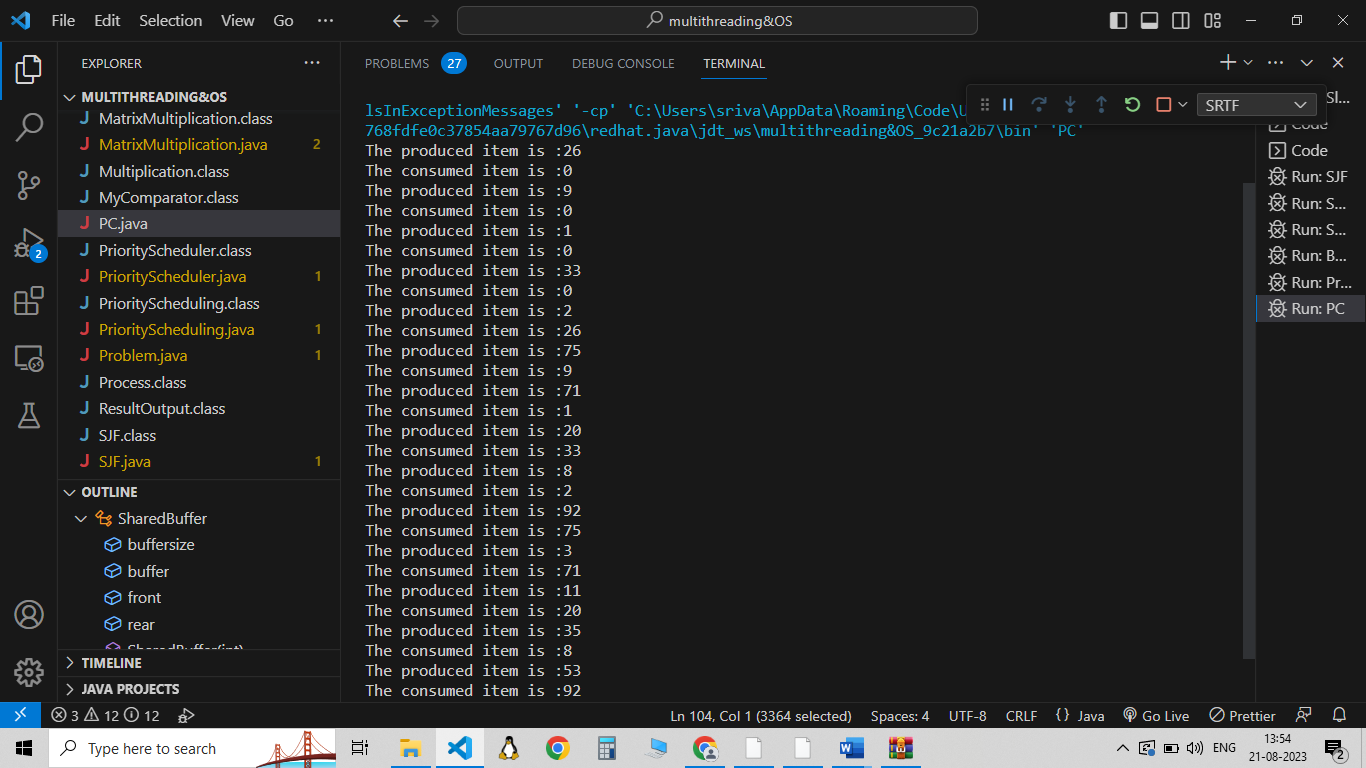
        Thread consumerThread  = new Thread(new Consumer(buffer,producerSemaphore,consumerSemaphore));

        producerThread.start();

        consumerThread.start();

    }

}

***Output*** :