**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**



**LAB REPORT**

**on**

**Operating Systems**

**(22CS4PCOPS)**

***Submitted by:***

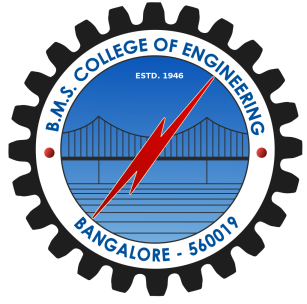
**NEHA BHASKAR KAMATH (1BM21CS113)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

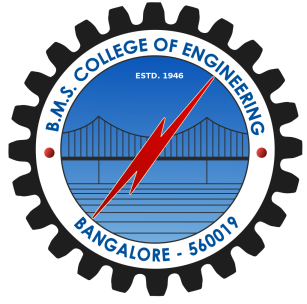
**June 2023 - August 2023**

**B. M. S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled “**Operating Systems**” carried out by **Neha Bhaskar Kamath(1BM21CS113),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Operating Systems - (22CS4PCOPS)** work prescribed for the said degree.

**M Lakshmi Neelima**  **Dr. Jyothi S Nayak**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Table Of Contents**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No.** | **Experiment Title** | | | **Page No.** |
| **1** | **Course Outcomes** | | | **5** |
| **2** | **Experiments** | | | **5-67** |
|  | **2.1** | **Experiment - 1** | | **5-11** |
| **2.1.1** | **Question:**  Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.   1. FCFS 2. SJF (pre-emptive & Non-pre-emptive) | **5** |
| **2.1.2** | **Code** | **5-10** |
| **2.1.3** | **Output** | **11** |
|  | **Experiment - 2** | | **12-16** |
| **2.2.1** | **Question:**  Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.   1. Priority (pre-emptive & Non-pre-emptive) 2. Round Robin (Experiment with different quantum sizes for RR algorithm) | **12** |
| **2.2.2** | **Code** | **12-16** |
| **2.2.3** | **Output** | **16** |
| **2.3** | **Experiment - 3** | | **17-19** |
| **2.3.1** | **Question:**  Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue. | **17** |
| **2.3.2** | **Code** | **17-19** |
| **2.3.3** | **Output** | **19** |
| **2.4.** | **Experiment - 4** | | **20-27** |
| **2.4.1** | **Question:**  Write a C program to simulate Real-Time CPU Scheduling algorithms:   1. Rate- Monotonic 2. Earliest-deadline First   **(c)** Proportional scheduling | **20** |
| **2.4.2** | **Code** | **20-25** |
| **2.4.3** | **Output** | **26-27** |
| **2.5.** | **Experiment - 5** | | **28-30** |
| **2.5.1** | **Question:**  Write a C program to simulate producer-consumer problem using semaphores. | **28** |
| **2.5.2** | **Code** | **28-30** |
| **2.5.3** | **Output** | **30** |
| **2.6** | **Experiment - 6** | | **31-32** |
| **2.6.1** | **Question:**  Write a C program to simulate the concept of Dining-Philosophers problem. | **31** |
| **2.6.2** | **Code** | **31** |
| **2.6.3** | **Output** | **32** |
| **2.7** | **Experiment - 7** | | **33-35** |
| **2.7.1** | **Question:**  Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | **33** |
| **2.7.2** | **Code** | **33-35** |
| **2.7.3** | **Output** | **35** |
| **2.8** | **Experiment - 8** | | **36-39** |
| **2.8.1** | **Question:**  Write a C program to simulate deadlock detection. | **36** |
| **2.8.2** | **Code** | **36-38** |
| **2.8.3** | **Output** | **39** |
|  | **2.9** | **Experiment - 9** | | **40-44** |
|  |  | **2.9.1** | **Question:**  Write a C program to simulate the following contiguous memory allocation techniques:   1. Worst-fit 2. Best-fit   **(c)** First-fit | **40** |
|  |  | **2.9.2** | **Code** | **40-43** |
|  |  | **2.9.3** | **Output** | **44** |
|  | **2.10** | **Experiment - 10** | | **45-46** |
|  |  | **2.10.1** | **Question:**  Write a C program to simulate paging technique of memory management. | **45** |
|  |  | **2.10.2** | **Code** | **45** |
|  |  | **2.10.3** | **Output** | **46** |
|  | **2.11** | **Experiment - 11** | | **47-56** |
|  |  | **2.11.1** | **Question:**  Write a C program to simulate page replacement algorithms:   1. FIFO 2. LRU   **(c)** Optimal | **47** |
|  |  | **2.11.2** | **Code** | **47-54** |
|  |  | **2.11.3** | **Output** | **54-56** |
|  | **2.12** | **Experiment - 12** | | **57-62** |
|  |  | **2.12.1** | **Question:**  Write a C program to simulate disk scheduling algorithms:   1. FCFS 2. SCAN   **(c)** C-SCAN | **57** |
|  |  | **2.12.2** | **Code** | **57-61** |
|  |  | **2.12.3** | **Output** | **62** |
|  | **2.13** | **Experiment - 13** | | **63-67** |
|  |  | **2.13.1** | **Question:**  Write a C program to simulate disk scheduling algorithms:   1. SSTF 2. LOOK   **(c)** c-LOOK | **63** |
|  |  | **2.13.2** | **Code** | **62-66** |
|  |  | **2.13.3** | **Output** | **67** |

1. **Course Outcomes**

**CO1:** Apply the different concepts and functionalities of Operating System.

**CO2:** Analyse various Operating system strategies and techniques.

**CO3:** Demonstrate the different functionalities of Operating System.

**CO4:** Conduct practical experiments to implement the functionalities of Operating system.

1. **Experiments**
   1. **Experiment - 1**
      1. **Question:**

Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.

1. FCFS
2. SJF
   * 1. **Code:**

**(a) FCFS**

#include <stdio.h>

int main() {

int n;

printf("Enter the number of processes: ");

scanf("%d", &n);

int pid[n], arrival[n], burst[n], waiting[n], turnaround[n];

printf("Enter the process ids:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &pid[i]);}

// Input process details

for (int i = 0; i < n; i++) {

printf("Enter arrival time and burst time for process %d: ", i + 1);

scanf("%d %d", &arrival[i], &burst[i]);

}

// Sort processes based on arrival time and then burst time

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arrival[j] == arrival[j + 1] && burst[j] > burst[j + 1]) {

int temp = burst[j];

burst[j] = burst[j + 1];

burst[j + 1] = temp;

temp = arrival[j];

arrival[j] = arrival[j + 1];

arrival[j + 1] = temp;

temp = pid[j];

pid[j] = pid[j + 1];

pid[j + 1] = temp;

}

else if (arrival[j] > arrival[j + 1]) {

int temp = arrival[j];

arrival[j] = arrival[j + 1];

arrival[j + 1] = temp;

temp = burst[j];

burst[j] = burst[j + 1];

burst[j + 1] = temp;

temp = pid[j];

pid[j] = pid[j + 1];

pid[j + 1] = temp;

}

}

}

waiting[0] = 0;

turnaround[0] = burst[0];

// Calculate waiting and turnaround times

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

waiting[i] = turnaround[i - 1] + arrival[i - 1] - arrival[i];

if (waiting[i] < 0)

waiting[i] = 0;

turnaround[i] = waiting[i] + burst[i];

}

float totalWaiting = 0, totalTurnaround = 0;

// Calculate total waiting and turnaround times

for (int i = 0; i < n; i++) {

totalWaiting += waiting[i];

totalTurnaround += turnaround[i];

}

float avgWaiting = totalWaiting / n;

float avgTurnaround = totalTurnaround / n;

printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");

for (int i = 0; i < n; i++) {

printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", pid[i], arrival[i], burst[i], waiting[i], turnaround[i]);

}

printf("\nAverage Waiting Time: %.2f\n", avgWaiting);

printf("Average Turnaround Time: %.2f\n", avgTurnaround);

return 0;

}

**(b) SJF**

#include <stdio.h>

void main()

{

int n,pid[10],bt[10],at[10],swap,tat[10],wt[10],comp=0,min,j,count=0,k;

float t\_tat=0,t\_wt=0;

printf("Enter the number of processes:\n");

scanf("%d",&n);

printf("Enter the process id:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &pid[i]);

}

printf("Enter the arrival time of the processes:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &at[i]);

}

printf("Enter the burst time of the processes:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &bt[i]);

}

//sort based on burst time

for(int i=0;i<n-1;i++)

{

for(int j=0;j<n-i-1;j++)

{

if(bt[j]>bt[j+1])

{

swap = pid[j];

pid[j] = pid[j+1];

pid[j+1] = swap;

swap= bt[j];

bt[j] = bt[j+1];

bt[j+1] = swap;

swap = at[j];

at[j] = at[j+1];

at[j+1] = swap;

}

}

}

for(int i=0;i<n;i++)

{

tat[i]=-1;

}

//find the process which has minimum arrival time because the arrays are sorted

min=at[0];

for(int i=1;i<n;i++)

{

if(at[i]<min)

{

min=at[i];

j=i;

}

}

comp+=at[j]+bt[j];

tat[j]=comp-at[j];

wt[j]=tat[j]-bt[j];

count++;

k=0;

while(count!=n)

{

if(tat[k]==-1 && at[k]<=comp)

{

comp+=bt[k];

tat[k]=comp-at[k];

wt[k]=tat[k]-bt[k];

count++;

k=(k+1)%n;

}

else if(tat[k]!=-1 || at[k]>comp)

{

k=(k+1)%n;

}

}

for(int i=0;i<n;i++)

{

t\_tat+=tat[i];

t\_wt+=wt[i];

}

printf("Pid\tArrivalTime\tBurstTime\tTAT\tWaitingTime\n");

for(int m=0;m<n;m++)

{

printf("%d\t\t%d\t\t%d\t%d\t%d\n", pid[m],at[m], bt[m],tat[m], wt[m]);

}

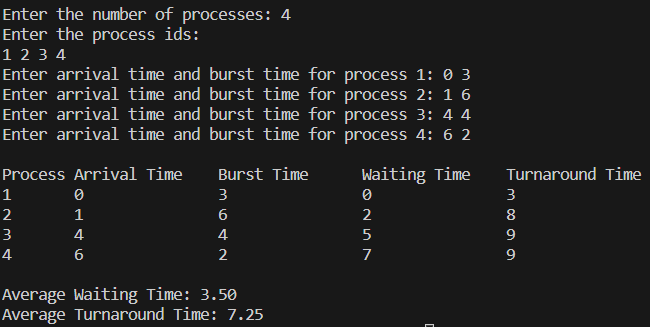
printf("Average turn around time:%0.2f\n", (t\_tat) / n);

printf("Average waiting time:%0.2f\n", (t\_wt) / n);

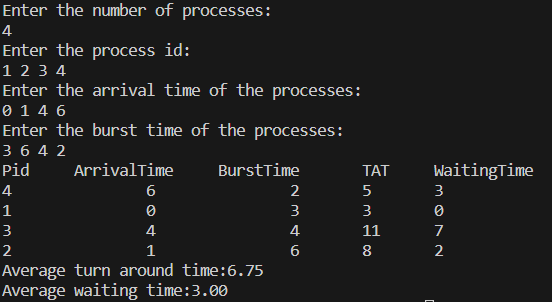
}

**2.1.3 Output:**

**(a) FCFS**

****

**(b) SJF**



* 1. **Experiment - 2**
     1. **Question:**

Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.

1. Priority (Non-pre-emptive)
2. Round Robin (Experiment with different quantum sizes for RR algorithm)
   * 1. **Code:**
3. **Priority (Non-pre-emptive)**

#include <stdio.h>

void main()

{

int n,pid[10],bt[10],at[10],pr[10],swap,tat[10],wt[10],comp=0,min,j,count=0,k;

float t\_tat=0,t\_wt=0;

printf("Enter the number of processes:\n");

scanf("%d",&n);

printf("Enter the process id:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &pid[i]);

}

printf("Enter the arrival time of the processes:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &at[i]);

}

printf("Enter the burst time of the processes:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &bt[i]);

}

printf("Enter the priority of processes:\n");

for (int i = 0; i < n; i++)

{

scanf("%d", &pr[i]);

}

// sorting based on priority, higher number means higher priority, so sorting in descending order

for(int i=0;i<n-1;i++)

{

for(int j=0;j<n-i-1;j++)

{

if(pr[j]<=pr[j+1])

{

swap = pr[j];

pr[j] = pr[j+1];

pr[j+1] = swap;

swap = pid[j];

pid[j] = pid[j+1];

pid[j+1] = swap;

swap= bt[j];

bt[j] = bt[j+1];

bt[j+1] = swap;

swap = at[j];

at[j] = at[j+1];

at[j+1] = swap;

}

}

}

for(int i=0;i<n;i++)

{

tat[i]=-1;

}

//to find which process has arrived first because we have sorted the array based on priority

min=at[0];

j=0;

for(int i=1;i<n;i++)

{

if(at[i]<min)

{

min=at[i];

j=i;

}

else if(at[i]==min) //if arrival time is the same, check which has higher priority

{

if(pr[i]>pr[j])

{

j=i;

}

else if(pr[i]==pr[j]) //if priorities are also same, check whcih one has lesser burst time.

{

if(bt[i]<bt[j])

{

j=i;

}

}

}

}

//j is the index/process which has arrived first, so compute tat for that first

comp+=at[j]+bt[j];

tat[j]=comp-at[j];

wt[j]=tat[j]-bt[j];

count++; //keeps track of number of processes computed for tat

k=0;

while(count!=n)

{

if(tat[k]==-1 && at[k]<=comp) //if tat is not yet computed and arrival time is less than completion time, then only we can compute tat

{

comp+=bt[k]; //update completion time

tat[k]=comp-at[k];

wt[k]=tat[k]-bt[k];

k=(k+1)%n;// if the process has not arrrived, we are not computing for this process rn, so we need to come back to check for those not computed

count++;

}

else if(tat[k]!=-1 || at[k]>comp)

{

k=(k+1)%n; // if tat already computed or the process has not yet arrived, just circularly increment

}

}

for(int i=0;i<n;i++)

{

t\_tat+=tat[i];

t\_wt+=wt[i];

}

printf("Pid\tArrivalTime\tBurstTime\tPriority\tTAT\tWaitingTime\n");

for(int m=0;m<n;m++)

{

printf("%d\t\t%d\t\t%d\t\t%d\t%d\t\t%d\n", pid[m],at[m], bt[m],pr[m],tat[m], wt[m]);

}

printf("Average turn around time:%0.1f\n", (t\_tat) / n);

printf("Average waiting time:%0.1f\n", (t\_wt) / n);

}

1. **Round Robin (Non-pre-emptive)**

#include <stdio.h>

#include <stdbool.h>

int turnarroundtime(int processes[], int n, int bt[], int wt[], int tat[]) {

for (int i = 0; i < n ; i++)

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

return 1;

}

int waitingtime(int processes[], int n, int bt[], int wt[], int quantum)

{

int rem\_bt[n];

for (int i = 0 ; i < n ; i++)

rem\_bt[i] = bt[i];

int t = 0;

while (1)

{

bool done = true;

for (int i = 0 ; i < n; i++)

{

if (rem\_bt[i] > 0)

{

done = false;

if (rem\_bt[i] > quantum)

{

t += quantum;

rem\_bt[i] -= quantum;

}

else

{

t = t + rem\_bt[i];

wt[i] = t - bt[i];

rem\_bt[i] = 0;

}

}

}

if (done == true)

break;

}

return 1;

}

int findavgTime(int processes[], int n, int bt[], int quantum) {

int wt[n], tat[n], total\_wt = 0, total\_tat = 0;

waitingtime(processes, n, bt, wt, quantum);

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

printf("\n\nProcesses\t\t Burst Time\t\t Waiting Time\t\t turnaround time\n");

for (int i=0; i<n; i++)

{

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

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

printf("\n\t%d\t\t\t%d\t\t\t%d\t\t\t%d\n",i+1, bt[i], wt[i], tat[i]);

}

printf("\nAverage waiting time = %f", (float)total\_wt / (float)n);

printf("\nAverage turnaround time = %f", (float)total\_tat / (float)n);

return 1;

}

int main()

{

int n, processes[n], burst\_time[n], quantum;

printf("Enter the Number of Processes: ");

scanf("%d",&n);

printf("\nEnter the quantum time: ");

scanf("%d",&quantum);

int i=0;

for(i=0;i<n;i++)

{

printf("\nEnter the process: ");

scanf("%d",&processes[i]);

printf("Enter the Burst Time:");

scanf("%d",&burst\_time[i]);

}

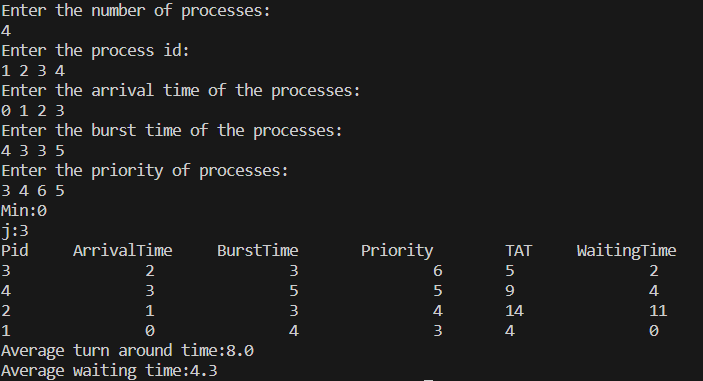
findavgTime(processes, n, burst\_time, quantum);

return 0;

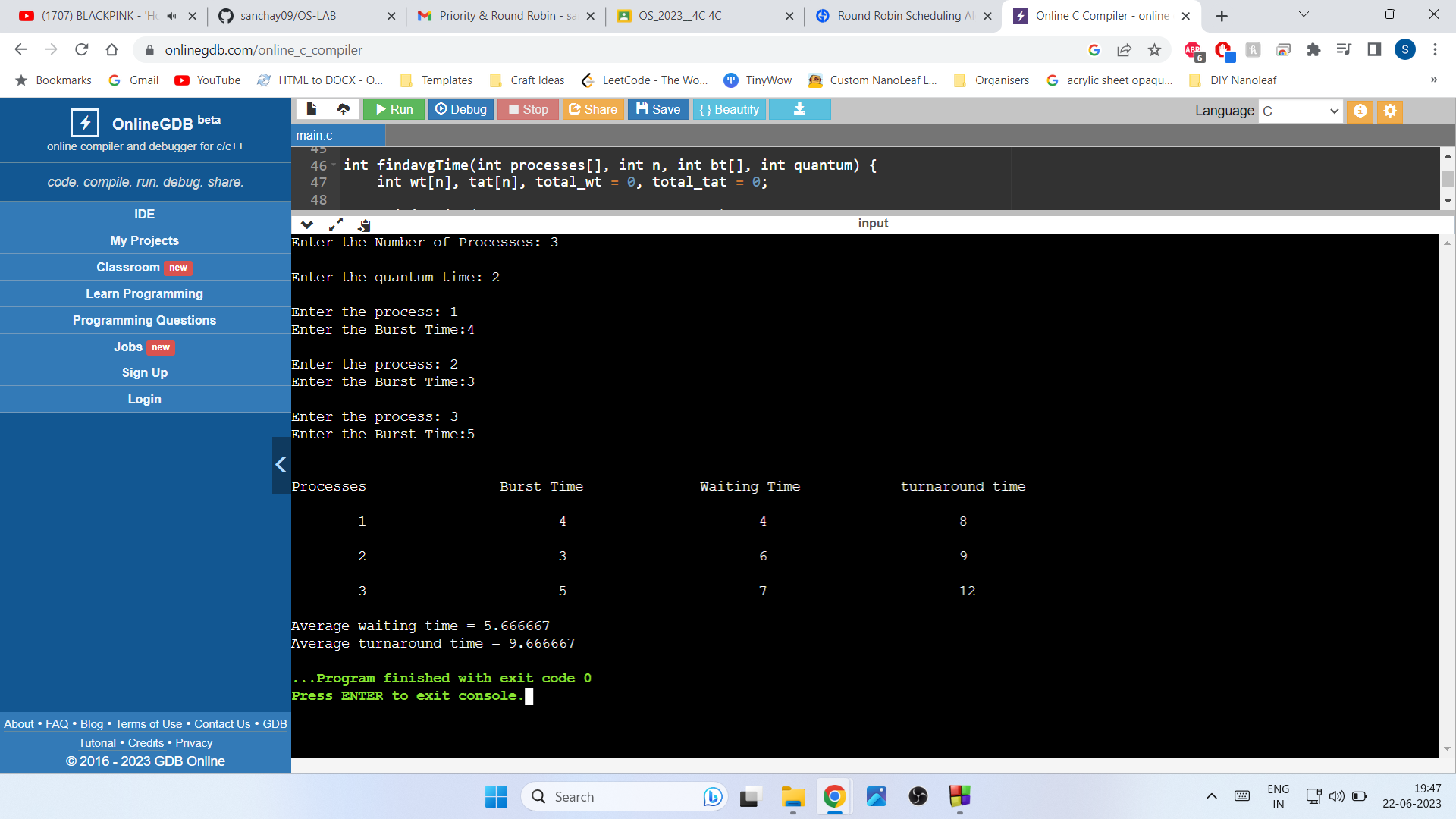
}

**2.2.3 Output:**

1. **Priority (Non-pre-emptive)**



**(b) Round Robin (Non-pre-emptive)**



**2.3 Experiment - 3**

**2.3.1 Question:**

Write a C program to simulate multi-level queue scheduling algorithm considering the following scenario. All the processes in the system are divided into two categories – system processes and user processes. System processes are to be given higher priority than user processes. Use FCFS scheduling for the processes in each queue.

**2.3.2 Code:**

#include <stdio.h>

#include <stdlib.h>

struct process {

int pid;

int arrival\_time;

int burst\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void FCFS(struct process \*queue, int n) {

int i, j;

struct process temp;

for (i = 0; i < n; i++) {

for (j = i + 1; j < n; j++) {

if (queue[i].arrival\_time > queue[j].arrival\_time) {

temp = queue[i];

queue[i] = queue[j];

queue[j] = temp;

}

}

}

}

int main() {

int n, i;

struct process \*system\_queue, \*user\_queue;

int system\_n = 0, user\_n = 0;

float avg\_waiting\_time = 0, avg\_turnaround\_time = 0;

printf("Enter the number of processes: ");

scanf("%d", &n);

system\_queue = (struct process \*) malloc(n \* sizeof(struct process));

user\_queue = (struct process \*) malloc(n \* sizeof(struct process));

for (i = 0; i < n; i++) {

struct process p;

printf("Enter arrival time, burst time, and priority (0-System/1-User) for process %d: ", i + 1);

scanf("%d %d %d", &p.arrival\_time, &p.burst\_time, &p.priority);

p.pid = i + 1;

p.waiting\_time = 0;

p.turnaround\_time = 0;

if (p.priority == 0) {

system\_queue[system\_n++] = p;

} else {

user\_queue[user\_n++] = p;

}

}

FCFS(system\_queue, system\_n);

FCFS(user\_queue, user\_n);

int time = 0;

int s=0,u=0;

while(s<system\_n || u<user\_n){

if(system\_queue[s].arrival\_time <= time){

if(user\_queue[u].arrival\_time <= time && user\_queue[u].arrival\_time < system\_queue[s].arrival\_time){

user\_queue[u].waiting\_time = time - user\_queue[u].arrival\_time;

time += user\_queue[u].burst\_time;

user\_queue[u].turnaround\_time = user\_queue[u].waiting\_time + user\_queue[u].burst\_time;

avg\_waiting\_time += user\_queue[u].waiting\_time;

avg\_turnaround\_time += user\_queue[u].turnaround\_time;

u++;

}

else{

system\_queue[s].waiting\_time = time - system\_queue[s].arrival\_time;

time += system\_queue[s].burst\_time;

system\_queue[s].turnaround\_time = system\_queue[s].waiting\_time + system\_queue[s].burst\_time;

avg\_waiting\_time += system\_queue[s].waiting\_time;

avg\_turnaround\_time += system\_queue[s].turnaround\_time;

s++;

}

}

else if(user\_queue[u].arrival\_time <= time){

user\_queue[u].waiting\_time = time - user\_queue[u].arrival\_time;

time += user\_queue[u].burst\_time;

user\_queue[u].turnaround\_time = user\_queue[u].waiting\_time + user\_queue[u].burst\_time;

avg\_waiting\_time += user\_queue[u].waiting\_time;

avg\_turnaround\_time += user\_queue[u].turnaround\_time;

u++;

}

else{

if(system\_queue[s].arrival\_time <= user\_queue[u].arrival\_time){

time = system\_queue[s].arrival\_time;

}

else{

time = user\_queue[u].arrival\_time;

}

}

}

avg\_waiting\_time /= n;

avg\_turnaround\_time /= n;

printf("PID\tBurst Time\tPriority\tQueue Type\tWaiting Time\tTurnaround Time\n");

for (i = 0; i < system\_n; i++) {

printf("%d\t%d\t\t%d\t\tSystem\t\t%d\t\t%d\n", system\_queue[i].pid, system\_queue[i].burst\_time, system\_queue[i].priority, system\_queue[i].waiting\_time, system\_queue[i].turnaround\_time);

}

for (i = 0; i < user\_n; i++) {

printf("%d\t%d\t\t%d\t\tUser\t\t%d\t\t%d\n", user\_queue[i].pid, user\_queue[i].burst\_time, user\_queue[i].priority, user\_queue[i].waiting\_time, user\_queue[i].turnaround\_time);

}

printf("Average Waiting Time: %.2f\n", avg\_waiting\_time);

printf("Average Turnaround Time: %.2f\n", avg\_turnaround\_time);

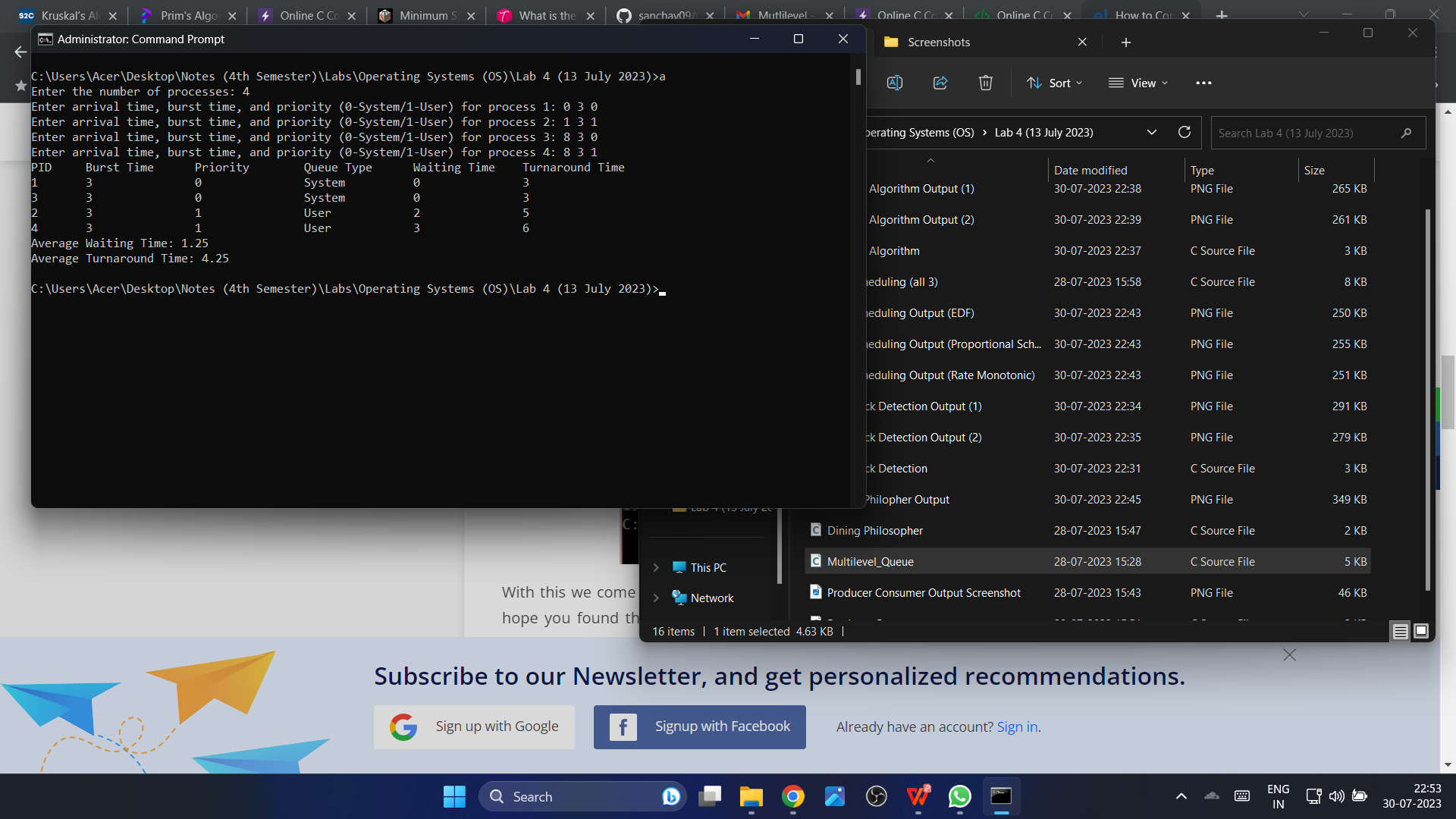
free(system\_queue);

free(user\_queue);

return 0;

}

**2.3.3 Output:**



**2.4 Experiment – 4**

**2.4.1 Question:**

Write a C program to simulate Real-Time CPU Scheduling algorithms:

1. Rate- Monotonic
2. Earliest-deadline First
3. Proportional scheduling

**2.4.2 Code:**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <stdbool.h>

#define MAX\_PROCESS 10

typedef struct {

int id;

int burst\_time;

float priority;

} Task;

int num\_of\_process;

int execution\_time[MAX\_PROCESS], period[MAX\_PROCESS], remain\_time[MAX\_PROCESS], deadline[MAX\_PROCESS], remain\_deadline[MAX\_PROCESS];

void get\_process\_info(int selected\_algo)

{

printf(“Enter total number of processes (maximum %d): “, MAX\_PROCESS);

scanf(“%d”, &num\_of\_process);

if (num\_of\_process < 1)

{

exit(0);

}

for (int i = 0; i < num\_of\_process; i++)

{

printf(“\nProcess %d:\n”, i + 1);

printf(“🡺 Execution time: “);

scanf(“%d”, &execution\_time[i]);

remain\_time[i] = execution\_time[i];

if (selected\_algo == 2)

{

printf(“🡺 Deadline: “);

scanf(“%d”, &deadline[i]);

}

else

{

printf(“🡺 Period: “);

scanf(“%d”, &period[i]);

}

}

}

int max(int a, int b, int c)

{

int max;

if (a >= b && a >= c)

max = a;

else if (b >= a && b >= c)

max = b;

else if (c >= a && c >= b)

max = c;

return max;

}

int get\_observation\_time(int selected\_algo)

{

if (selected\_algo == 1)

{

return max(period[0], period[1], period[2]);

}

else if (selected\_algo == 2)

{

return max(deadline[0], deadline[1], deadline[2]);

}

}

void print\_schedule(int process\_list[], int cycles)

{

printf(“\nScheduling:\n\n”);

printf(“Time: “);

for (int i = 0; i < cycles; i++)

{

if (i < 10)

printf(“| 0%d “, i);

else

printf(“| %d “, i);

}

printf(“|\n”);

for (int i = 0; i < num\_of\_process; i++)

{

printf(“P[%d]: “, i + 1);

for (int j = 0; j < cycles; j++)

{

if (process\_list[j] == i + 1)

printf(“|####”);

else

printf(“| “);

}

printf(“|\n”);

}

}

void rate\_monotonic(int time)

{

int process\_list[100] = {0}, min = 999, next\_process = 0;

float utilization = 0;

for (int i = 0; i < num\_of\_process; i++)

{

utilization += (1.0 \* execution\_time[i]) / period[i];

}

int n = num\_of\_process;

int m = (float) (n \* (pow(2, 1.0 / n) – 1));

if (utilization > m)

{

printf(“\nGiven problem is not schedulable under the said scheduling algorithm.\n”);

}

for (int i = 0; i < time; i++)

{

min = 1000;

for (int j = 0; j < num\_of\_process; j++)

{

if (remain\_time[j] > 0)

{

if (min > period[j])

{

min = period[j];

next\_process = j;

}

}

}

if (remain\_time[next\_process] > 0)

{

process\_list[i] = next\_process + 1;

remain\_time[next\_process] -= 1;

}

for (int k = 0; k < num\_of\_process; k++)

{

if ((i + 1) % period[k] == 0)

{

remain\_time[k] = execution\_time[k];

next\_process = k;

}

}

}

print\_schedule(process\_list, time);

}

void earliest\_deadline\_first(int time){

float utilization = 0;

for (int i = 0; i < num\_of\_process; i++){

utilization += (1.0\*execution\_time[i])/deadline[i];

}

int n = num\_of\_process;

int process[num\_of\_process];

int max\_deadline, current\_process=0, min\_deadline,process\_list[time];

bool is\_ready[num\_of\_process];

for(int i=0; i<num\_of\_process; i++){

is\_ready[i] = true;

process[i] = i+1;

}

max\_deadline=deadline[0];

for(int i=1; i<num\_of\_process; i++){

if(deadline[i] > max\_deadline)

max\_deadline = deadline[i];

}

for(int i=0; i<num\_of\_process; i++){

for(int j=i+1; j<num\_of\_process; j++){

if(deadline[j] < deadline[i]){

int temp = execution\_time[j];

execution\_time[j] = execution\_time[i];

execution\_time[i] = temp;

temp = deadline[j];

deadline[j] = deadline[i];

deadline[i] = temp;

temp = process[j];

process[j] = process[i];

process[i] = temp;

}

}

}

for(int i=0; i<num\_of\_process; i++){

remain\_time[i] = execution\_time[i];

remain\_deadline[i] = deadline[i];

}

for (int t = 0; t < time; t++){

if(current\_process != -1){

--execution\_time[current\_process];

process\_list[t] = process[current\_process];

}

else

process\_list[t] = 0;

for(int i=0;i<num\_of\_process;i++){

--deadline[i];

if((execution\_time[i] == 0) && is\_ready[i]){

deadline[i] += remain\_deadline[i];

is\_ready[i] = false;

}

if((deadline[i] <= remain\_deadline[i]) && (is\_ready[i] == false)){

execution\_time[i] = remain\_time[i];

is\_ready[i] = true;

}

}

min\_deadline = max\_deadline;

current\_process = -1;

for(int i=0;i<num\_of\_process;i++){

if((deadline[i] <= min\_deadline) && (execution\_time[i] > 0)){

current\_process = i;

min\_deadline = deadline[i];

}

}

}

print\_schedule(process\_list, time);

}

void proportionalScheduling() {

int n;

printf(“Enter the number of tasks: “);

scanf(“%d”, &n);

Task tasks[n];

printf(“Enter burst time and priority for each task:\n”);

for (int i = 0; i < n; i++) {

tasks[i].id = i + 1;

printf(“Task %d – Burst Time: “, tasks[i].id);

scanf(“%d”, &tasks[i].burst\_time);

printf(“Task %d – Priority: “, tasks[i].id);

scanf(“%f”, &tasks[i].priority);

}

// Sort tasks based on priority (ascending order)

for (int i = 0; i < n – 1; i++) {

for (int j = 0; j < n – i – 1; j++) {

if (tasks[j].priority > tasks[j + 1].priority) {

// Swap tasks

Task temp = tasks[j];

tasks[j] = tasks[j + 1];

tasks[j + 1] = temp;

}

}

}

printf(“\nProportional Scheduling:\n”);

int total\_burst\_time = 0;

float total\_priority = 0.0;

for (int i = 0; i < n; i++) {

total\_burst\_time += tasks[i].burst\_time;

total\_priority += tasks[i].priority;

}

for (int i = 0; i < n; i++) {

float time\_slice = (tasks[i].priority / total\_priority) \* total\_burst\_time;

printf(“Task %d executes for %.2f units of time\n”, tasks[i].id, time\_slice);

}

}

int main()

{

int option;

int observation\_time;

while (1)

{

printf(“\n1. Rate Monotonic\n2. Earliest Deadline first\n3. Proportional Scheduling\n\nEnter your choice: “);

scanf(“%d”, &option);

switch(option)

{

case 1: get\_process\_info(option);

observation\_time = get\_observation\_time(option);

rate\_monotonic(observation\_time);

break;

case 2: get\_process\_info(option);

observation\_time = get\_observation\_time(option);

earliest\_deadline\_first(observation\_time);

break;

case 3: proportionalScheduling();

break;

case 4: exit (0);

default: printf(“\nInvalid Statement”);

}

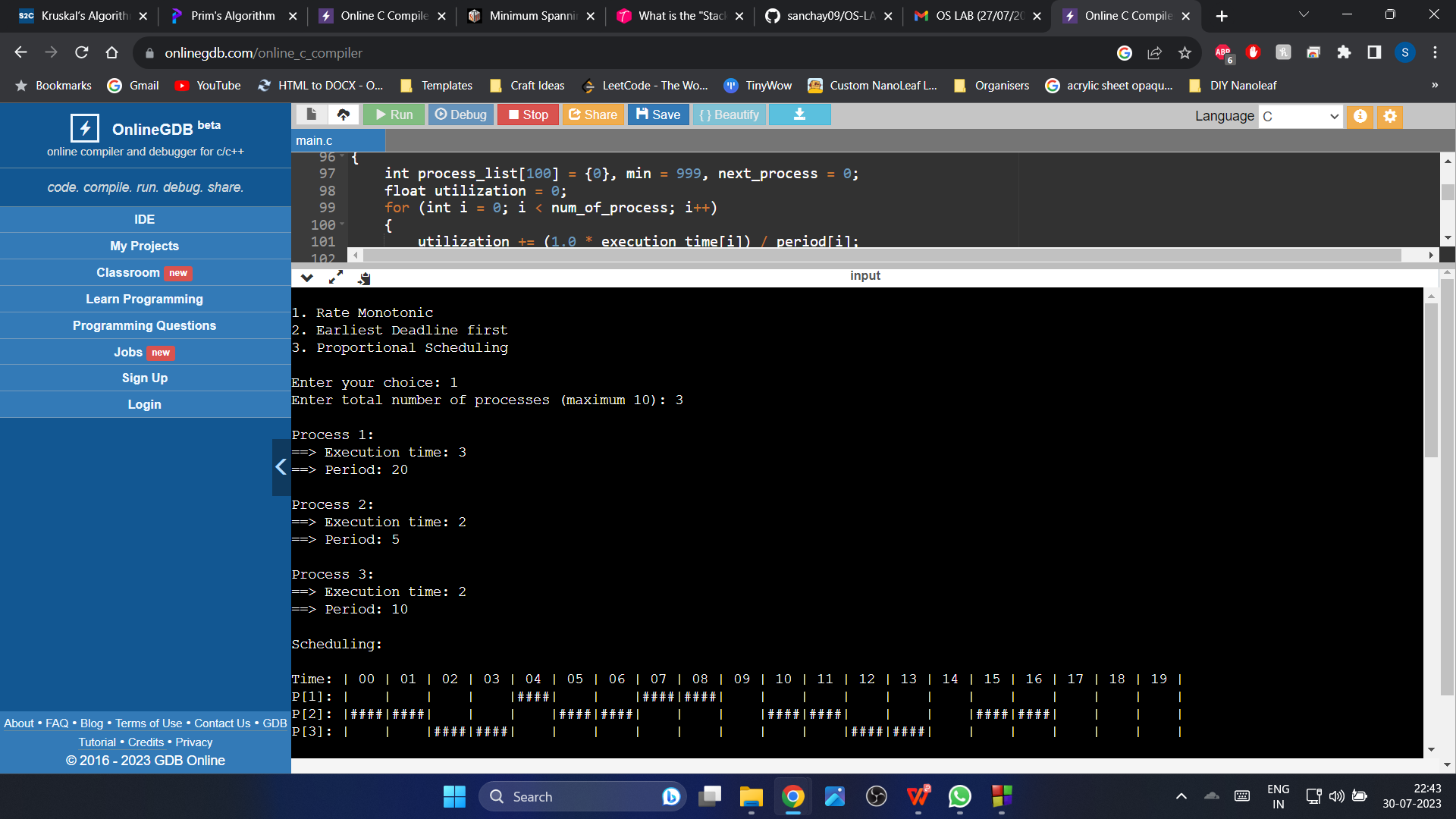
}

return 0;

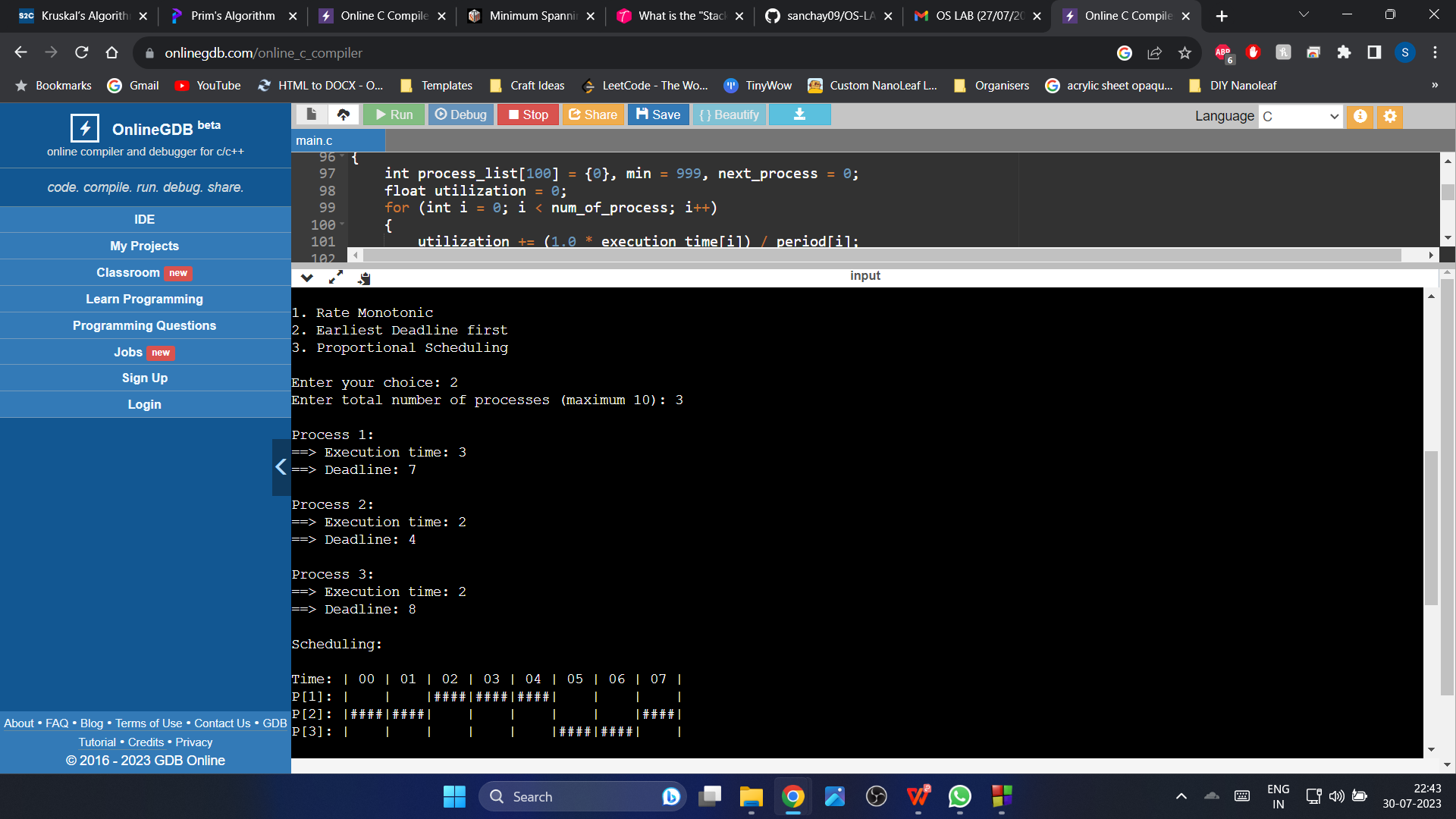
}

**2.4.3 Output:**

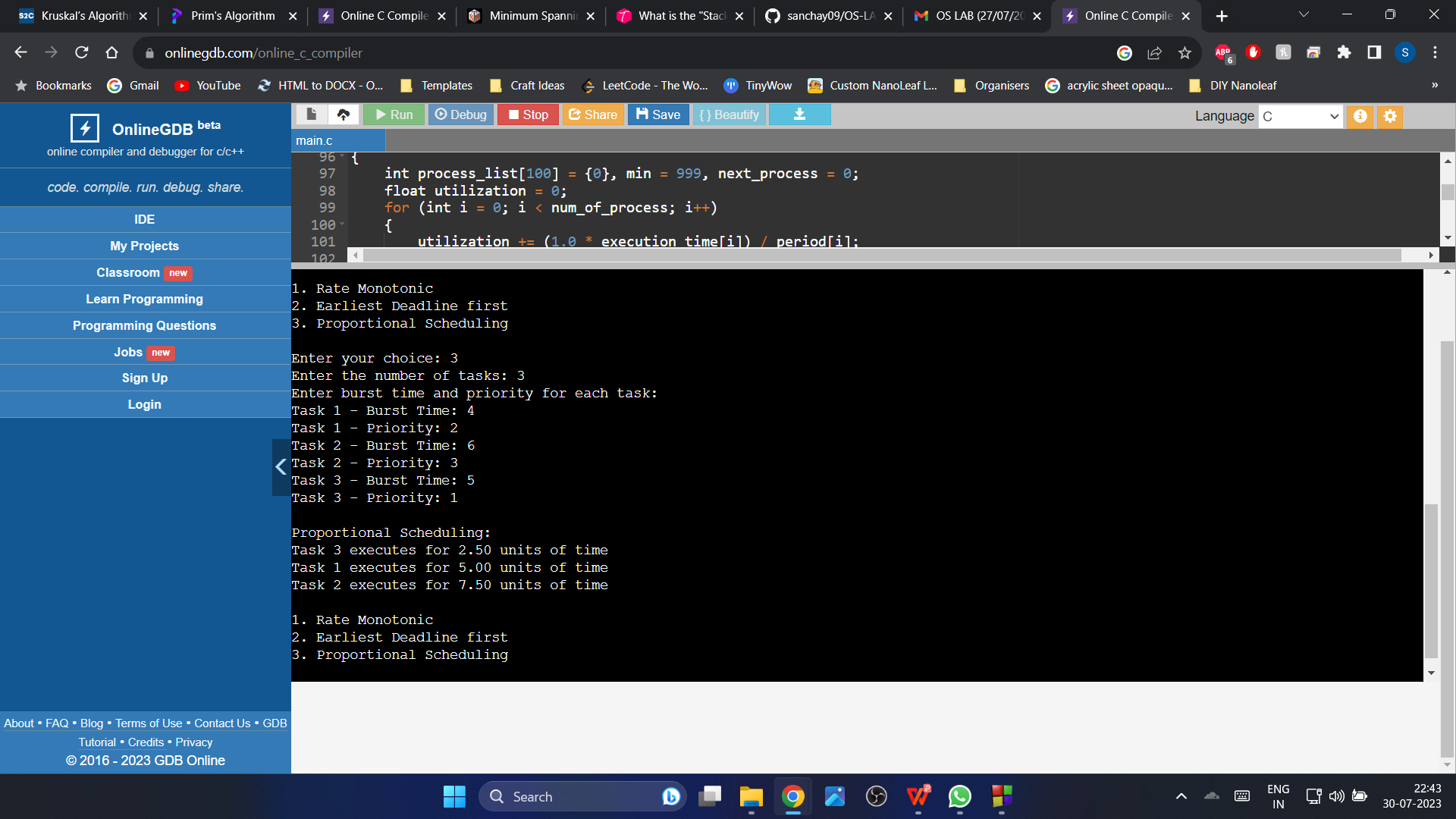
1. **Rate Monotonic:**



1. **Earliest Deadline First:**



**© Proportional Scheduling:**



**2.5 Experiment – 5**

**2.5.1 Question:**

Write a C program to simulate producer-consumer problem using semaphores.

**2.5.2 Code**:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 10

#define NUM\_ITEMS 20

int buffer[BUFFER\_SIZE];

int fill = 0; // Index to add data by producer

int use = 0; // Index to consume data by consumer

int count = 0; // Number of items in the buffer

sem\_t empty; // Semaphore to track empty slots in the buffer

sem\_t full; // Semaphore to track the number of items available for consumption

void put(int value) {

buffer[fill] = value;

fill = (fill + 1) % BUFFER\_SIZE;

count++;

}

int get() {

int tmp = buffer[use];

use = (use + 1) % BUFFER\_SIZE;

count--;

return tmp;

}

void \*producer(void \*arg) {

int i;

for (i = 0; i < NUM\_ITEMS; i++) {

sem\_wait(&empty); // Wait for an empty slot

put(i);

printf(“Produced: %d\n”, i);

sem\_post(&full); // Signal that an item is produced

}

pthread\_exit(NULL);

}

void \*consumer(void \*arg) {

int i;

for (i = 0; i < NUM\_ITEMS; i++) {

sem\_wait(&full); // Wait for an item to be produced

int value = get();

printf(“Consumed: %d\n”, value);

sem\_post(&empty); // Signal that an empty slot is available

}

pthread\_exit(NULL);

}

int main() {

// Initialize semaphores

sem\_init(&empty, 0, BUFFER\_SIZE); // Set empty slots to BUFFER\_SIZE

sem\_init(&full, 0, 0); // No items available initially

pthread\_t producer\_thread, consumer\_thread;

// Create threads

pthread\_create(&producer\_thread, NULL, producer, NULL);

pthread\_create(&consumer\_thread, NULL, consumer, NULL);

// Wait for threads to finish

pthread\_join(producer\_thread, NULL);

pthread\_join(consumer\_thread, NULL);

// Destroy semaphores

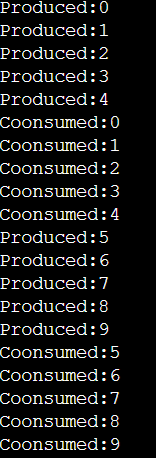
sem\_destroy(&empty);

sem\_destroy(&full);

return 0;

}

**2.5.3 Output:**



**2.6 Experiment – 6**

**2.6.1 Question:**

Write a C program to simulate the concept of Dining-Philosophers problem.

**2.6.2 Code**:

#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

#include<unistd.h>

sem\_t room;

sem\_t chopstick[5];

void \* philosopher(void \*);

void eat(int);

int main()

{

int i,a[5];

pthread\_t tid[5];

sem\_init(&room,0,4);

for(i=0;i<5;i++)

sem\_init(&chopstick[i],0,1);

for(i=0;i<5;i++){

a[i]=i;

pthread\_create(&tid[i],NULL,philosopher,(void \*)&a[i]);

}

for(i=0;i<5;i++)

pthread\_join(tid[i],NULL);

}

void \* philosopher(void \* num)

{

int phil=\*(int \*)num;

sem\_wait(&room);

printf(“\nPhilosopher %d has entered room”,phil);

sem\_wait(&chopstick[phil]);

sem\_wait(&chopstick[(phil+1)%5]);

eat(phil);

sleep(2);

printf(“\nPhilosopher %d has finished eating”,phil);

sem\_post(&chopstick[(phil+1)%5]);

sem\_post(&chopstick[phil]);

sem\_post(&room);

}

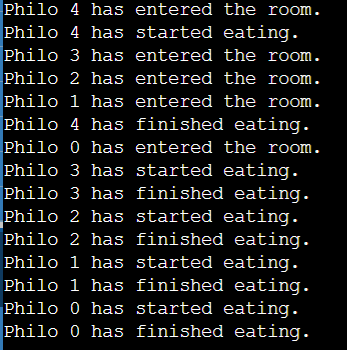
void eat(int phil)

{

printf(“\nPhilosopher %d is eating”,phil);

}

**2.6.3 Output:**

****

**2.7 Experiment – 7**

**2.7.1 Question:**

Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

**2.7.2 Code**:

#include <stdio.h>

int main()

{

int n, m, i, j, k;

printf(“Enter the number of processes: “);

scanf(“%d”, &n);

printf(“Enter the number of resources: “);

scanf(“%d”, &m);

int allocation[n][m];

printf(“Enter the Allocation Matrix:\n”);

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

{

scanf(“%d”, &allocation[i][j]);

}

}

int max[n][m];

printf(“Enter the MAX Matrix:\n”);

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

{

scanf(“%d”, &max[i][j]);

}

}

int available[m];

printf(“Enter the Available Resources:\n”);

for (i = 0; i < m; i++)

{

scanf(“%d”, &available[i]);

}

int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++)

{

f[k] = 0;

}

int need[n][m];

for (i = 0; i < n; i++)

{

for (j = 0; j < m; j++)

{

need[i][j] = max[i][j] – allocation[i][j];

}

}

int y = 0;

for (k = 0; k < n; k++)

{

for (i = 0; i < n; i++)

{

if (f[i] == 0)

{

int flag = 0;

for (j = 0; j < m; j++)

{

if (need[i][j] > available[j])

{

flag = 1;

break;

}

}

if (flag == 0)

{

ans[ind++] = i;

for (y = 0; y < m; y++)

{

available[y] += allocation[i][y];

}

f[i] = 1;

}

}

}

}

int flag = 1;

for (i = 0; i < n; i++)

{

if (f[i] == 0)

{

flag = 0;

printf(“The following system is not safe\n”);

break;

}

}

if (flag == 1)

{

printf(“Following is the SAFE Sequence\n”);

for (i = 0; i < n – 1; i++)

{

printf(“ P%d ->”, ans[i]);

}

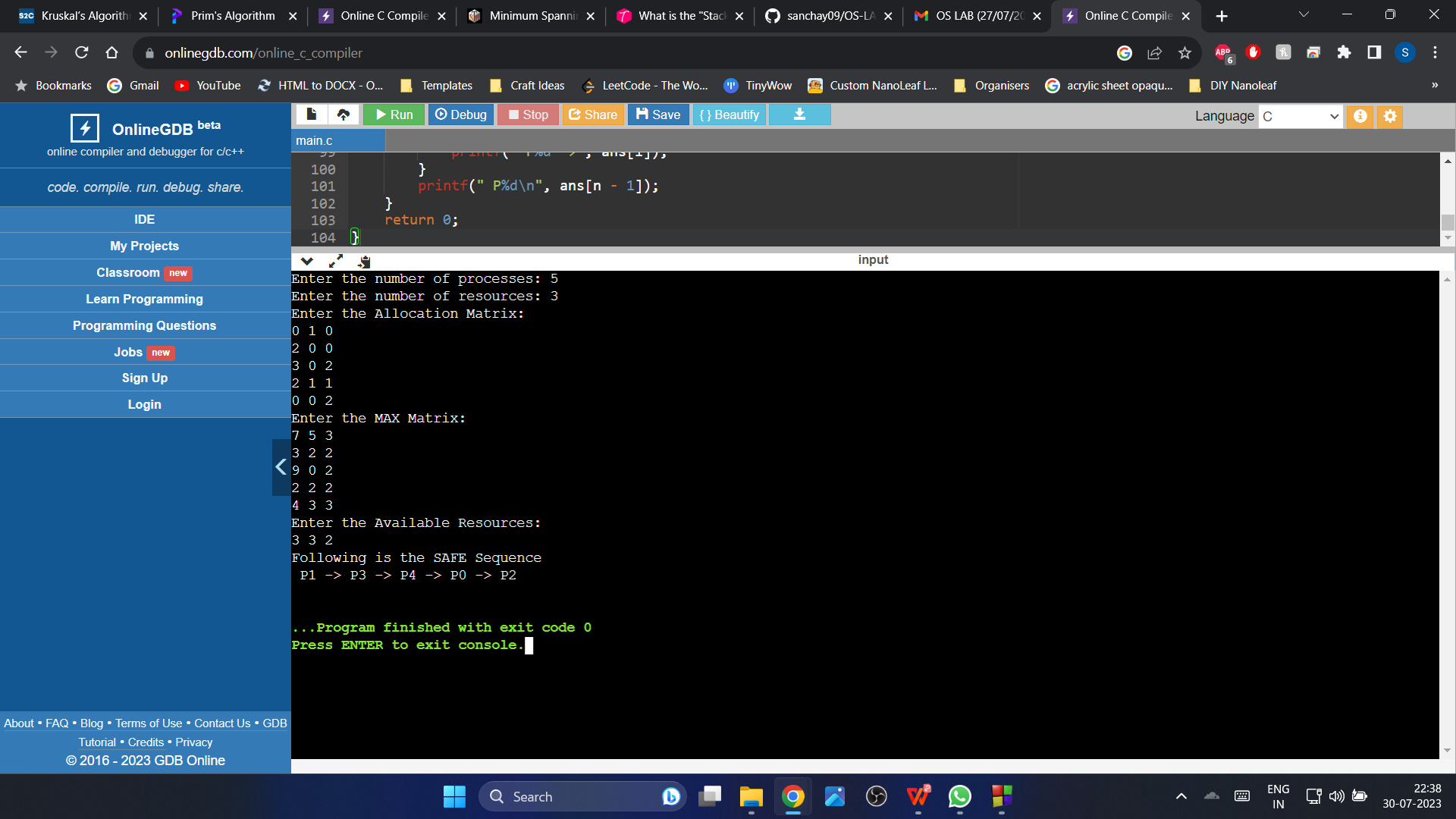
printf(“ P%d\n”, ans[n – 1]);

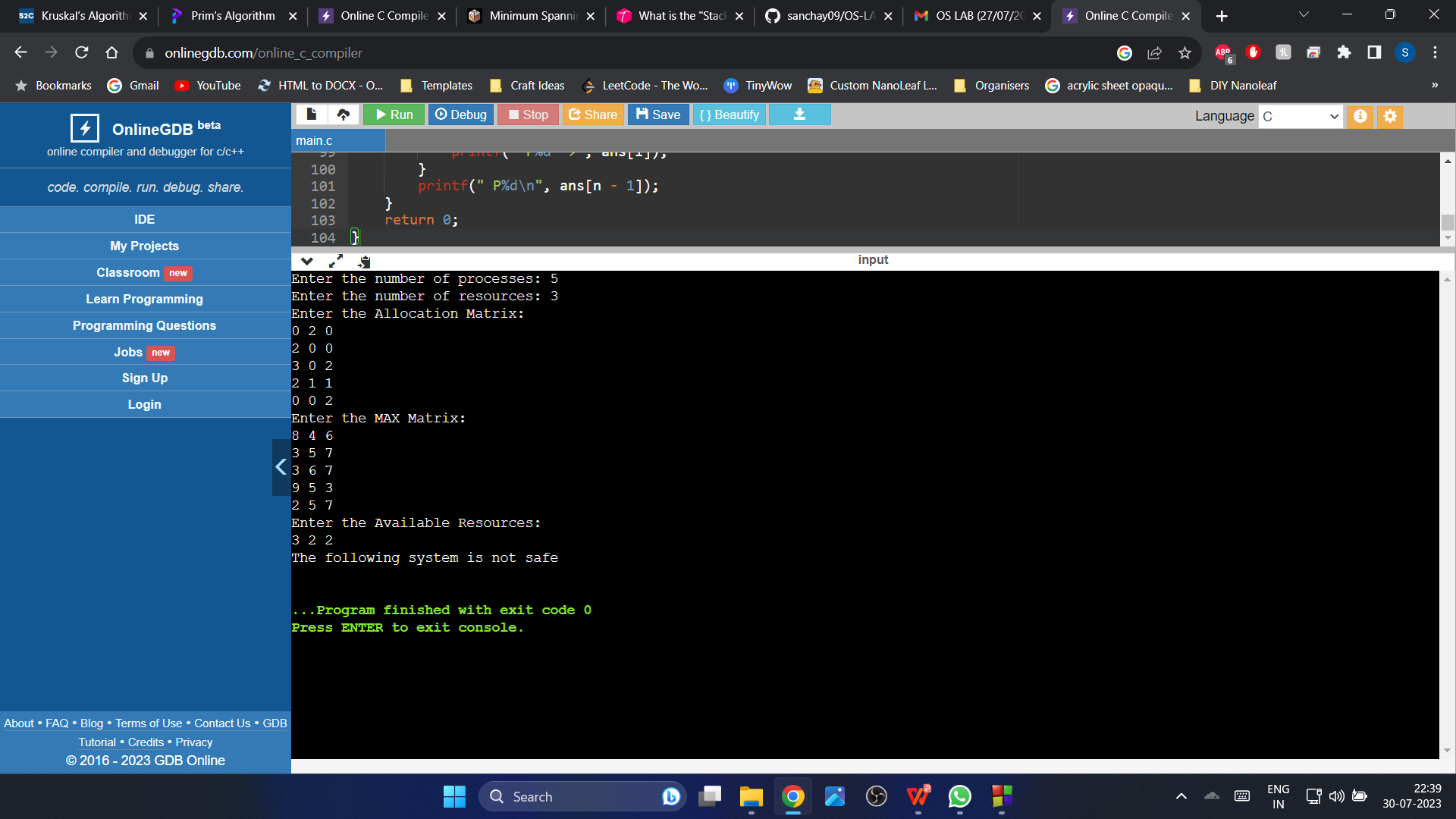
}

return 0;

}

**2.7.3 Output:**





**2.8 Experiment – 8**

**2.8.1 Question:**

Write a C program to simulate deadlock detection.

**2.8.2 Code**:

#include<stdio.h>

int max[100][100];

int allocation[100][100];

int need[100][100];

int available[100];

int n,r;

int main()

{

int i,j;

printf(“Deadlock Detection\n”);

input();

show();

cal();

return 0;

}

void input()

{

int i,j;

printf(“Enter the no of Processes: “);

scanf(“%d”,&n);

printf(“Enter the no of resource instances: “);

scanf(“%d”,&r);

printf(“Enter the Max Matrix:\n”);

for(i=0;i<n;i++)

{

for(j=0;j<r;j++)

{

scanf(“%d”,&max[i][j]);

}

}

printf(“Enter the Allocation Matrix:\n”);

for(i=0;i<n;i++)

{

for(j=0;j<r;j++)

{

scanf(“%d”,&allocation[i][j]);

}

}

printf(“Enter the available Resources:\n”);

for(j=0;j<r;j++)

{

scanf(“%d”,&available[j]);

}

}

void show()

{

int i,j;

printf(“Process\t Allocation\t Max\t Available\t”);

for(i=0;i<n;i++)

{

printf(“\nP%d\t “,i+1);

for(j=0;j<r;j++)

{

printf(“%d “,allocation[i][j]);

}

printf(“\t”);

for(j=0;j<r;j++)

{

printf(“%d “,max[i][j]);

}

printf(“\t”);

if(i==0)

{

for(j=0;j<r;j++)

printf(“%d “,available[j]);

}

}

}

void cal()

{

int finish[100],temp,need[100][100],flag=1,k,c1=0;

int dead[100];

int safe[100];

int i,j;

for(i=0;i<n;i++)

{

finish[i]=0;

}

for(i=0;i<n;i++)

{

for(j=0;j<r;j++)

{

need[i][j]=max[i][j]-allocation[i][j];

}

}

while(flag)

{

flag=0;

for(i=0;i<n;i++)

{

int c=0;

for(j=0;j<r;j++)

{

if((finish[i]==0)&&(need[i][j]<=available[j]))

{

c++;

if(c==r)

{

for(k=0;k<r;k++)

{

available[k]+=allocation[i][j];

finish[i]=1;

flag=1;

}

if(finish[i]==1)

{

i=n;

}

}

}

}

}

}

j=0;

flag=0;

for(i=0;i<n;i++)

{

if(finish[i]==0)

{

dead[j]=i;

j++;

flag=1;

}

}

if(flag==1)

{

printf(“\n\nSystem is in Deadlock and the Deadlock process are\n”);

for(i=0;i<n;i++)

{

printf(“P%d\t”,dead[i]);

}

}

else

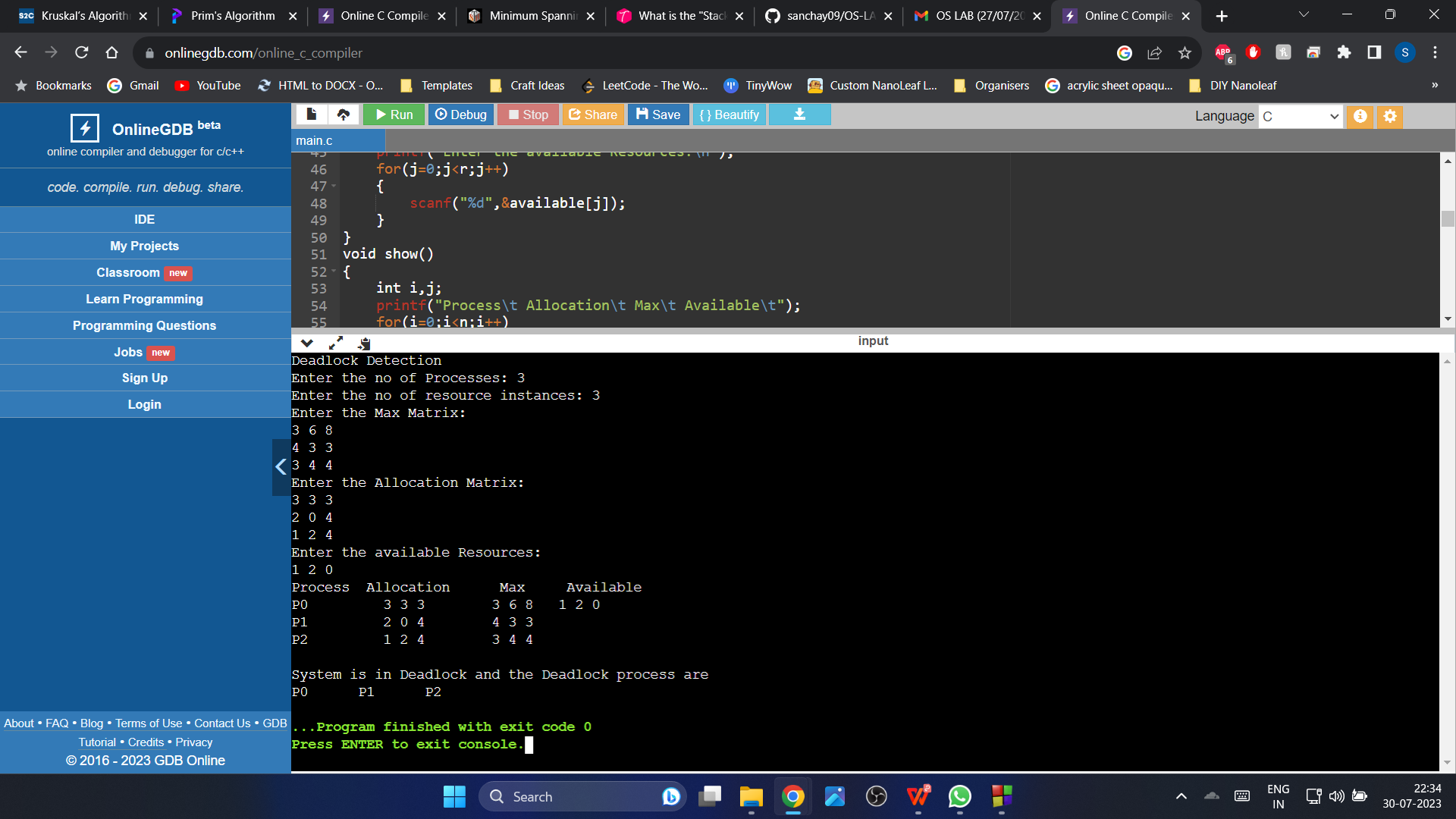
{

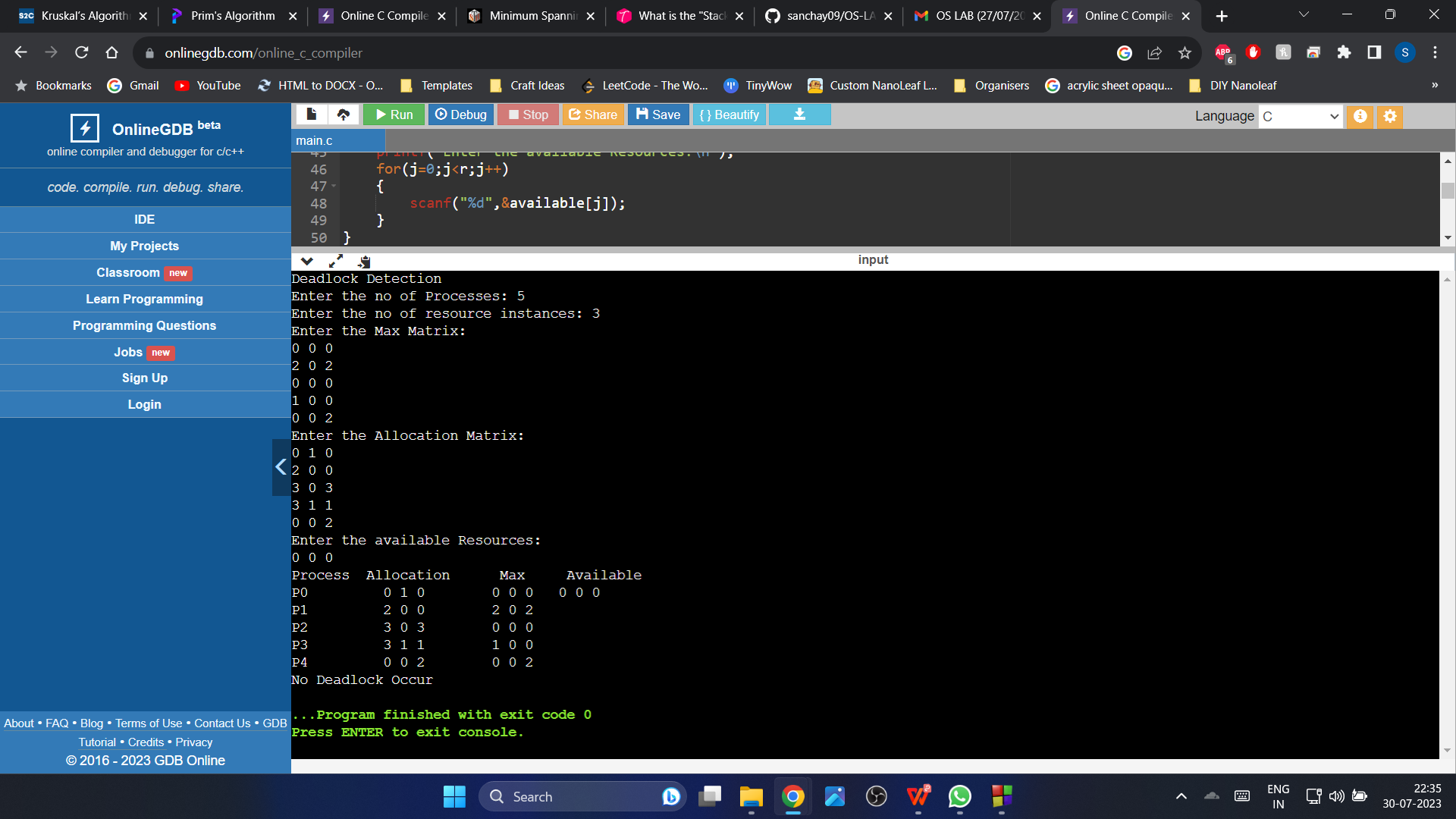
printf(“\nNo Deadlock Occur”);

}

}

**2.8.3 Output:**

****

****

**2.9 Experiment – 9**

**2.9.1 Question:**

Write a C program to simulate the following contiguous memory allocation techniques:

1. Worst-fit
2. Best-fit

**(c)** First-fit

**2.9.2 Code**:

**(a) Worst**-**fit**

#include<stdio.h>

void main()

{

int n,m,i,j;

printf("Enter the number of processes and number of blocks:\n");

scanf("%d %d",&n,&m);

int all[n],blockSize[m],processSize[n];

printf("Enter %d process sizes:\n",n);

for(i=0;i<n;i++)

{

scanf("%d",&processSize[i]);

all[i]=-1;

}

printf("Enter %d block sizes:\n",m);

for(j=0;j<m;j++)

{

scanf("%d",&blockSize[j]);

}

//Since this is worst fit, the largest available partition should be allocated. So we can sort the block sizes in descending order.

for(i=0;i<m-1;i++)

{

for(j=0;j<m-i-1;j++)

{

if(blockSize[j]<=blockSize[j+1])

{

int temp=blockSize[j];

blockSize[j]=blockSize[j+1];

blockSize[j+1]=temp;

}

}

}

for(i=0;i<n;i++)

{

for(j=0;j<m;j++)

{

if(blockSize[j]>=processSize[i])

{

all[i]=blockSize[j];

blockSize[j]=-1;

break;

}

}

}

printf("\*\*\*\*Worst fit memory allocation\*\*\*\*\n");

printf("ProcessID\tProcess\_Size\tBlock\_size\_allocated\n");

for(i=0;i<n;i++)

{

printf("P%d\t\t",(i+1));

printf("%d\t\t",processSize[i]);

if(all[i]==-1)

printf("Not allocated\n");

else

printf("%d\n",all[i]);

}

}

**(b) Best-fit**

#include<stdio.h>

void main()

{

int n,m,i,j;

printf("Enter the number of processes and number of blocks:\n");

scanf("%d %d",&n,&m);

int all[n],blockSize[m],processSize[n];

printf("Enter %d process sizes:\n",n);

for(i=0;i<n;i++)

{

scanf("%d",&processSize[i]);

all[i]=-1;

}

printf("Enter %d block sizes:\n",m);

for(j=0;j<m;j++)

{

scanf("%d",&blockSize[j]);

}

/\*Since this is best fit, the smallest partition which is adequate is allocated to the processes. So we can sort the blockSizes

in ascending order. \*/

for(i=0;i<m-1;i++)

{

for(j=0;j<m-i-1;j++)

{

if(blockSize[j]>blockSize[j+1])

{

int temp=blockSize[j];

blockSize[j]=blockSize[j+1];

blockSize[j+1]=temp;

}

}

}

for(i=0;i<n;i++)

{

for(j=0;j<m;j++)

{

if(blockSize[j]>=processSize[i])

{

all[i]=blockSize[j];

blockSize[j]=-1;

break;

}

}

}

printf("\*\*\*\*Best fit memory allocation\*\*\*\*\n");

printf("ProcessID\tProcess\_Size\tBlock\_size\_allocated\n");

for(i=0;i<n;i++)

{

printf("P%d\t\t",(i+1));

printf("%d\t\t",processSize[i]);

if(all[i]==-1)

printf("Not allocated\n");

else

printf("%d\n",all[i]);

}

}

**(c ) First-fit**

#include<stdio.h>

void main()

{

int n,m,i,j,c=0;

printf("Enter the number of processes and number of blocks:\n");

scanf("%d %d",&n,&m);

int all[n];

for(int i=0;i<n;i++)

{

all[i]=-1;

}

int blockSize[m],processSize[n];

printf("Enter the %d block sizes:\n",m);

for(j=0;j<m;j++)

{

scanf("%d",&blockSize[j]);

}

printf("Enter the %d process sizes:\n",n);

{

for(i=0;i<n;i++)

{

scanf("%d",&processSize[i]);

}

}

for(i=0;i<n;i++)

{

for(j=0;j<m;j++)

{

if(blockSize[j]>=processSize[i])

{

all[i]=blockSize[j];

blockSize[j]=-1;

break;

}

}

}

printf("\*\*\*\*First fit memory allocation\*\*\*\*\n");

printf("ProcessId\tProcessSize\tBlock\_Size\_allocated\n");

for(i=0;i<n;i++)

{

printf("P%d\t\t",(i+1));

printf("%d\t\t",processSize[i]);

if(all[i]!=-1)

printf("%d\n",all[i]);

else

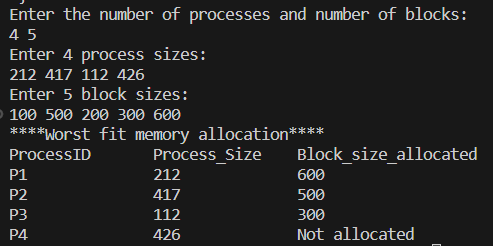
printf("Not allocated\n");

}

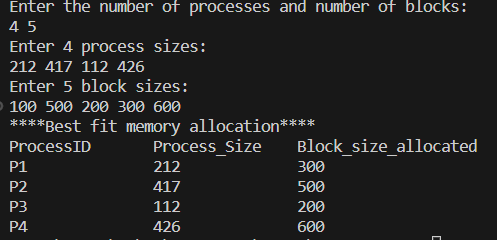
}

**2.9.3 Output:**

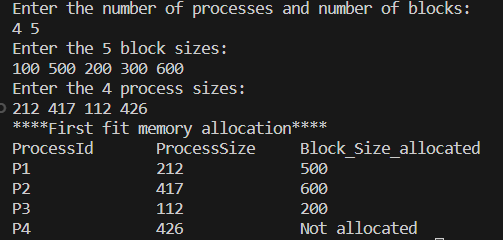
**(a) Worst-fit**



**(b) Best-fit**

****

**(c) First-fit**

****

**2.10 Experiment – 10**

**2.10.1 Question:**

Write a C program to simulate paging technique of memory management.

**2.10.2 Code**:

#include<stdio.h>

#define MAX 50

int main()

{

int page[MAX],i,n,f,ps,off,pno;

int choice=0;

printf(“Enter the number of pages in memory: “);

scanf(“%d”,&n);

printf(“\nEnter Page size: “);

scanf(“%d”,&ps);

printf(“\nEnter number of frames: “);

scanf(“%d”,&f);

for(i=0;i<n;i++)

page[i]=-1;

printf(“\nEnter the Page Table\n”);

printf(“(Enter frame no as -1 if that page is not present in any frame)\n\n”);

printf(“\nPage No\t\tFrame No\n-------\t\t-------");

for(i=0;i<n;i++)

{

printf(“\n\n%d\t\t”,i);

scanf(“%d”,&page[i]);

}

do

{

printf(“\n\nEnter the logical address(i.e,page no & offset):”);

scanf(“%d%d”,&pno,&off);

if(page[pno]==-1)

printf(“\n\nThe required page is not available in any of frames”);

else

printf(“\nPhysical address(i.e,frame no & offset):%d,%d”,page[pno],off);

printf(“\n\nDo you want to continue(1/0)?:”);

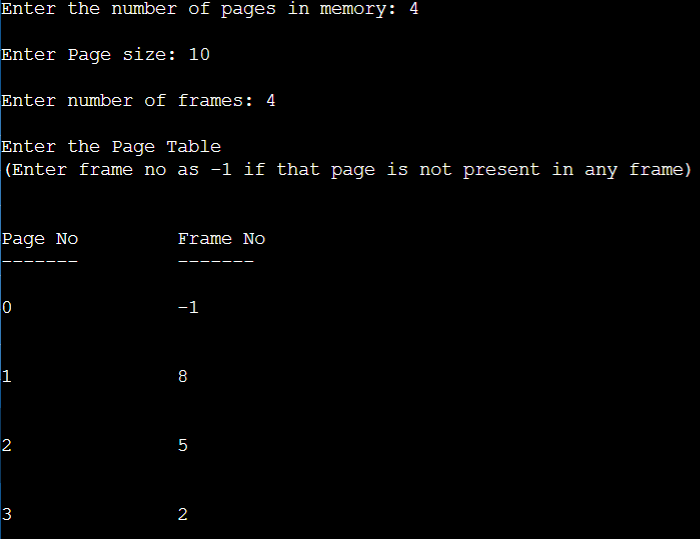
scanf(“%d”,&choice);

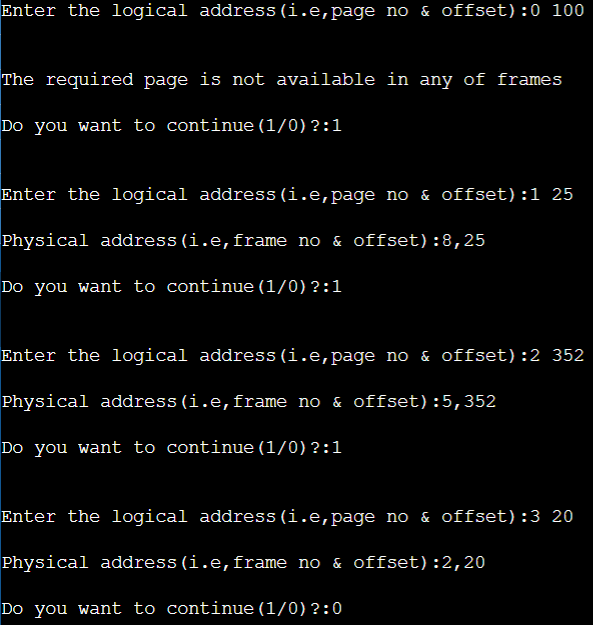
}while(choice==1);

return 1;

}

**2.10.3 Output:**





**2.11 Experiment – 11**

**2.11.1 Question:**

Write a C program to simulate page replacement algorithms:

1. FIFO
2. LRU
3. Optimal

**2.11.2 Code**:

**(a) FIFO**

#include<stdio.h>

int isHit(int fr[], int pg, int m)

{

int hit=0;

for(int i=0;i<m;i++)

{

if(fr[i]==pg)

{

hit=1;

break;

}

}

return hit;

}

void main()

{

int n,m,k=0,pagefault=0;

printf("Enter the length of reference sequence:\n");

scanf("%d",&n);

int ref[n];

printf("Enter the page reference sequence:\n");

for(int i=0;i<n;i++)

{

scanf("%d",&ref[i]);

}

printf("Enter the number of frames:\n");

scanf("%d",&m);

int fr[m];

for(int i=0;i<m;i++)

{

fr[i]=-1;

}

for(int i=0;i<n;i++)

{

//if it is not a hit

if(isHit(fr,ref[i],m)==0)

{

fr[k]=ref[i];

k=(k+1)%m; //since this is first come first serve.

pagefault++;

printf("%d:Page Fault\n",ref[i]);

}

else

printf("%d:No page fault\n",ref[i]);

}

printf("Total number of page faults:%d\n",pagefault);

}

**(b) Optimal**

#include<stdio.h>

int isHit(int fr[], int pg, int m)

{

int hit=0;

for(int i=0;i<m;i++)

{

if(fr[i]==pg)

{

hit=1;

break;

}

}

return hit;

}

void main()

{

int i,n,m,k,j,pagefault=0,max=-1,x,y,flag=0,count=0,u;

printf("Enter the length of reference sequence:\n");

scanf("%d",&n);

int ref[n];

printf("Enter the page reference sequence:\n");

for(i=0;i<n;i++)

{

scanf("%d",&ref[i]);

}

printf("Enter the number of frames:\n");

scanf("%d",&m);

int fr[m];

for(i=0;i<m;i++)

{

fr[i]=-1;

}

u=0;

y=0;

while(count<m)

{

if(isHit(fr,ref[u],m)==0)

{

fr[y]=ref[u];

printf("%d:Page fault\n",ref[u]);

u++;

y++;

count++;

pagefault++;

}

else

{

printf("%d:No page fault\n",ref[u]);

u++;

}

}

for(i=u;i<n;i++)

{

if(isHit(fr,ref[i],m)==0)

{

for(j=0;j<m;j++)

{

for(k=i+1;k<n;k++)

{

if(fr[j]==ref[k])//as soon as match happens, break.

{

flag=1;

break;

}

else if(k==n-1 && fr[j]!=ref[k])//if there is no demand of a particular page in future, just replace that.

{

flag=-1;

fr[j]=ref[i];

break;

}

}

if(flag==-1)//if there is no demand, directly replaced, no need to check other pages in the frames.

break;

else if(flag==1 && k>max)

{

max=k;

x=j;

}

}

max=-1; //reset max for other iterations

if(flag!=-1

{

fr[x]=ref[i];

}

pagefault++;

printf("%d:Page fault\n",ref[i]);

}

else

{

printf("%d:No page fault\n",ref[i]);

}

}

printf("Total no of page faults:%d\n",pagefault);

}

**(c) LRU**

#include<stdio.h>

int isHit(int fr[], int pg, int m)

{

int hit=0;

for(int i=0;i<m;i++)

{

if(fr[i]==pg)

{

hit=1;

break;

}

}

return hit;

}

void main()

{

int i,n,m,k,j,pagefault=0,min=999,x,y,count=0,u=0;

printf("Enter the length of reference sequence:\n");

scanf("%d",&n);

int ref[n];

printf("Enter the page reference sequence:\n");

for(i=0;i<n;i++)

{

scanf("%d",&ref[i]);

}

printf("Enter the number of frames:\n");

scanf("%d",&m);

int fr[m];

for(i=0;i<m;i++)

{

fr[i]=-1;

}

y=0;

u=0;

while(count<m)

{

if(isHit(fr,ref[u],m)==0)

{

fr[y]=ref[u];

printf("%d:Page Fault\n",ref[u]);

y++;

u++;

pagefault++;

count++;

}

else

{

printf("%d:No page fault\n",ref[u]);

u++;

}

}

for(i=u;i<n;i++)

{

if(isHit(fr,ref[i],m)==0)

{

for(j=0;j<m;j++)//for every element in the frames, check which index is the least.

{

for(k=i-1;k>=0;k--)//to check which index is the least, for each number in the frame, we need to start checking from i-1 only.

{

if(fr[j]==ref[k])

{

break;

}

}

if(k<min) /\*agar pg no ki index min se kam ho, iska matlab ye hai ki uski demand sabse pehele hua tha,

sirf tabhi min ko update karna\*/

{

x=j;

min=k;

}

}

min=999; //reset min for other iterations

fr[x]=ref[i];

pagefault++;

printf("%d:Page fault\n",ref[i]);

}

else

{

printf("%d:No page fault\n",ref[i]);

}

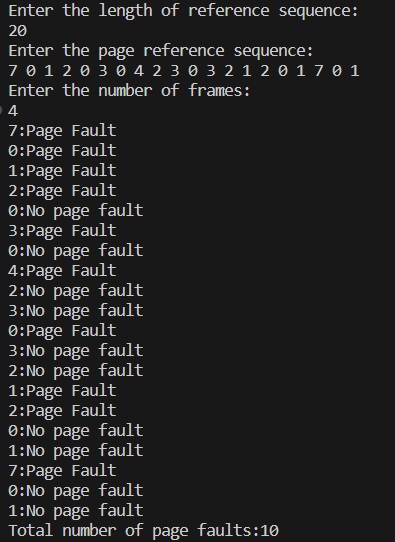
}

printf("Total number page faults:%d\n",pagefault);

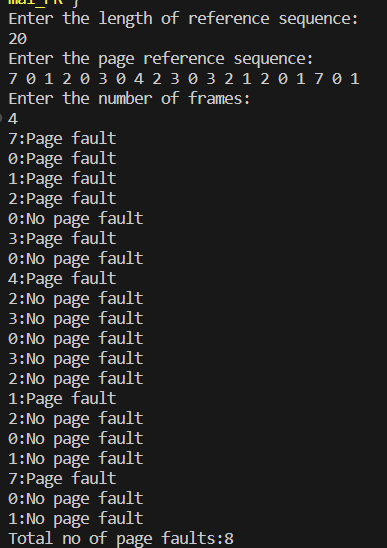
}

**2.11.3 Output:**

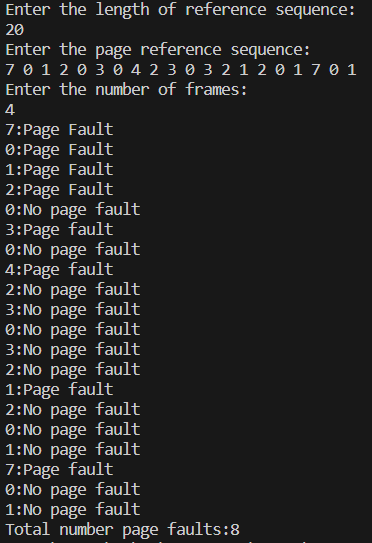
1. **FIFO:**



**(b) OPTIMAL:**



**(c) LRU:**

****

**2.12 Experiment - 12**

**2.12.1 Question:**

Write a C program to simulate disk scheduling algorithms:

1. FCFS
2. SCAN
3. c-SCAN

**2.12.2 Code**:

1. **FCFS:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int tr,n,total=0,curr;

printf("Enter the total no of tracks in the disk:\n");

scanf("%d",&tr);

printf("Enter the number of requests in the request queue:\n");

scanf("%d",&n);

int arr[n];

printf("Enter the request sequence:\n");

for(int i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("Enter the current head positon of the disk arm:\n");

scanf("%d",&curr);

for(int i=0;i<n;i++)

{

printf("The head moves from track %d to %d with seek time %d units\n",curr,arr[i], abs(arr[i]-curr));

total+=abs(arr[i]-curr);

curr=arr[i];

}

printf("The total head movements using FCFS scheduling are:%d\n",total);

}

1. **SCAN:**

#include<stdio.h>

#include<stdlib.h>

void sortAsc(int arr[], int s,int e)

{

int temp;

for(int i=s;i<e-1;i++)

{

for(int j=s;j<e-i-1;j++)

{

if(arr[j]>arr[j+1])

{

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

}

void sortDesc(int arr[], int s,int e)

{

int temp;

for(int i=s;i<e-1;i++)

{

for(int j=s;j<e-i-1;j++)

{

if(arr[j]<arr[j+1])

{

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

}

}

}

void main()

{

int tr,n,total=0,curr,dir,min,max,i,j,k;

printf("Enter the total no of tracks in the disk:\n");

scanf("%d",&tr);

printf("Enter the number of requests in the request queue:\n");

scanf("%d",&n);

int arr[n],seek[n+1];

printf("Enter the request sequence:\n");

for(int i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("Enter the current head positon of the disk arm:\n");

scanf("%d",&curr);

printf("Enter head movement direction(1 for High and 0 for Low):\n");

scanf("%d",&dir);

switch(dir)

{

case 1:

//disk fulfills all the higher requests first, so the head reaches the higher end of disk and then changes direction.

//That is why, we need to find the lower most request track.

min=arr[0];

for(i=1;i<n;i++)

{

if(arr[i]<min)

min=arr[i];

}

for(i=0;i<=n;i++)

{

seek[i]=arr[i];

}

seek[n]=curr;

printf("Seek sequence:\n");

sortDesc(seek,0,n+1); //sort in descending order

for(i=0;i<=n;i++)

{

if(seek[i]==curr)

k=i;

}

sortAsc(seek,0,k);

for(i=0;i<=n;i++)

{

printf("%d ",seek[i]);

}

printf("\n");

total=(tr-1-curr)+(tr-1-min);

printf("Total head movements using SCAN scheduling are:%d\n",total);

break;

case 0:

//disk fulfills all the lower requests first, so the head reaches the lower end of disk and then changes direction.

//That is why, we need to find the max request track.

max=arr[0];

for(i=1;i<n;i++)

{

if(arr[i]>max)

max=arr[i];

}

total=(curr-0)+(max-0);//0 is the lower most track

printf("Total head movements using SCAN scheduling are:%d\n",total);

break;

default:

printf("Invalid choice:\n");

}

}

1. **c-SCAN:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void main()

{

int tr,n,total=0,curr,dir,min,max;

printf("Enter the total no of tracks in the disk:\n");

scanf("%d",&tr);

printf("Enter the number of requests in the request queue:\n");

scanf("%d",&n);

int arr[n];

printf("Enter the request sequence:\n");

for(int i=0;i<n;i++)

{

scanf("%d",&arr[i]);

}

printf("Enter the current head positon of the disk arm:\n");

scanf("%d",&curr);

printf("Enter head movement direction(1 for High and 0 for Low):\n");

scanf("%d",&dir);

switch(dir)

{

case 1:

//head first moves to the higher end of disk while the disk fulfills all the higher requests, changes direction to

//reach the lower end when th disk does not fulill any lower request. After reaching the lower end, the head again

//changes direction when the disk starts fulfilling lower requests.

//So we need to find max request less than curr head position

max=INT\_MIN;

for(int i=0;i<n;i++)

{

if(arr[i]<curr && arr[i]>max)

{

max=arr[i];

}

}

total=(tr-1-curr)+(tr-1-0)+(max-0);

printf("The total head movements using C-SCAN scheduling are:%d\n",total);

break;

case 0:

//reverse of case 1

min=INT\_MAX;

for(int i=0;i<n;i++)

{

if(arr[i]>50 && arr[i]<min)

{

min=arr[i];

}

}

printf("Min:%d\n",min);

total=(curr-0)+(tr-1-0)+(tr-1-min);

printf("The total head movements using C-SCAN scheduling are:%d\n",total);

break;

default:

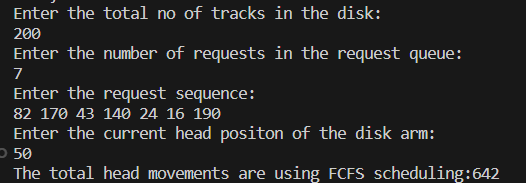
printf("Invalid choice!\n");

}

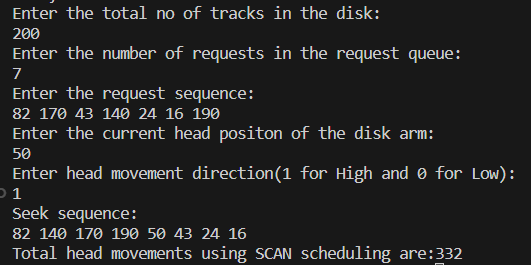
}

**2.12.3 Output:**

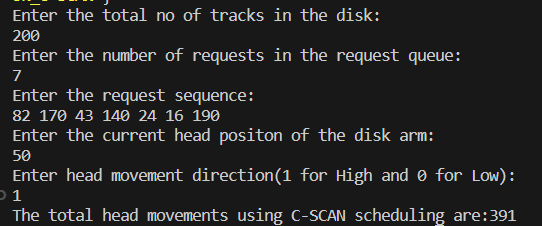
1. **FCFS:**

****

1. **SCAN:**

****

1. **C-SCAN:**

****

**2.13 Experiment - 13**

**2.13.1 Question:**

Write a C program to simulate disk scheduling algorithms:

1. SSTF
2. LOOK
3. C-LOOK

**2.13.2 Code**:

1. **SSTF:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void main()

{

int tr,n,total=0,curr,min,count=0,d,ind,i,j=0;

printf(“Enter the total no of tracks in the disk:\n”);

scanf(“%d”,&tr);

printf(“Enter the number of requests in the request queue:\n”);

scanf(“%d”,&n);

int arr[n],seek[n];

printf(“Enter the request sequence:\n”);

for(int i=0;i<n;i++)

{

scanf(“%d”,&arr[i]);

}

printf(“Enter the current head positon of the disk arm:\n”);

scanf(“%d”,&curr);

while(count!=n)

{

min=1000;

for(i=0;i<n;i++)

{

if(abs(arr[i]-curr)<min)

{

min=abs(arr[i]-curr);

ind=i;

}

}

seek[j]=arr[ind];

total+=min;

curr=arr[ind];

arr[ind]=1000;

count++;

j++;

}

printf(“Safe sequence is:\n”);

for(i=0;i<n;i++)

{

printf(“%d “,seek[i]);

}

printf(“\n”);

printf(“Total number of movements using SSTF are:%d\n”,total);

}

1. **LOOK:**

#include<stdio.h>

#include<stdlib.h>

void main()

{

int tr,n,total=0,curr,min,max,i,j=0,index;

printf(“Enter the total no of tracks in the disk:\n”);

scanf(“%d”,&tr);

printf(“Enter the number of requests in the request queue:\n”);

scanf(“%d”,&n);

int arr[n],seek[n];

printf(“Enter the request sequence:\n”);

for(int i=0;i<n;i++)

{

scanf(“%d”,&arr[i]);

}

printf(“Enter the current head positon of the disk arm:\n”);

scanf(“%d”,&curr);

//direction considered- towards larger values first

max=arr[0];

min=arr[0];

for(i=1;i<n;i++)

{

if(arr[i]>max)

{

max=arr[i];

}

if(arr[i]<min)

{

min=arr[i];

}

}

total=(max-curr)+(max-min);

printf(“Total number of movements using LOOK scheduling:%d\n”,total);

}

1. **c-LOOK:**

#include<stdio.h>

#include<stdlib.h>

#include<limits.h>

void main()

{

int tr,n,total=0,curr,min,sec\_max,i,j=0,index,max;

printf(“Enter the total no of tracks in the disk:\n”);

scanf(“%d”,&tr);

printf(“Enter the number of requests in the request queue:\n”);

scanf(“%d”,&n);

int arr[n],seek[n];

printf(“Enter the request sequence:\n”);

for(int i=0;i<n;i++)

{

scanf(“%d”,&arr[i]);

}

printf(“Enter the current head positon of the disk arm:\n”);

scanf(“%d”,&curr);

//direction considered- towards larger values first

max=arr[0];

min=arr[0];

for(i=1;i<n;i++)

{

if(arr[i]>max)

{

max=arr[i];

}

if(arr[i]<min)

{

min=arr[i];

}

}

sec\_max=INT\_MIN;

for(i=0;i<n;i++)

{

if(arr[i]<curr && arr[i]>sec\_max)

{

sec\_max=arr[i];

}

}

printf(“sec\_max:%d\n”,sec\_max);

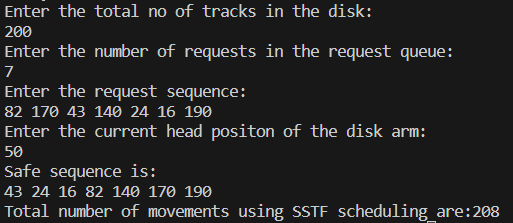
total=(max-curr)+(max-min)+(sec\_max-min);

printf(“Total number of movements using C-LOOK scheduling are:%d\n”,total);

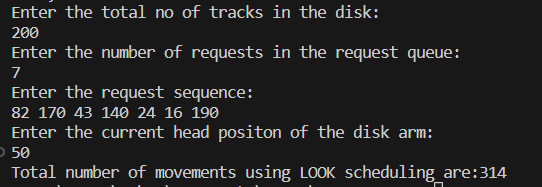
}

**2.13.3 Output:**

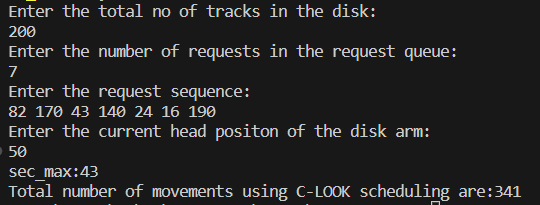
1. **SSTF:**

****

1. **LOOK:**

****

1. **c-LOOK:**

****