**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

**Operating Systems**

**(23CS4PCOPS)**

***Submitted by:***

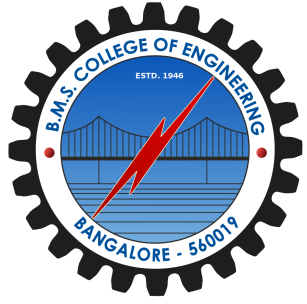
**ADITYA RAM S H(1BM22CS019)**

***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**

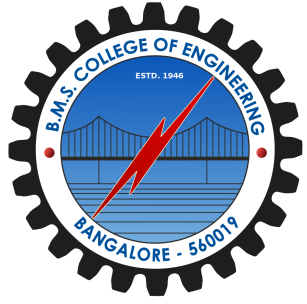
**June 2024 - August 2024**

**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 **ADITYA RAM S H(1BM22CS019),** who is a 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 - (23CS4PCOPS)** work prescribed for the said degree.

**Sneha S Bagalkot**              **Dr. Jyothi S Nayak**

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

**Table Of Contents**

| **Lab Program No.** | **Program Details** | **Page No.** |
| --- | --- | --- |
| 1 | FCFS AND SJF | 2-5 |
| 2 | PRIORITY AND ROUND ROBIN | 6-11 |
| 3 | RATE-MONOTONIC AND EARLIEST DEADLINE FIRST | 12-16 |
| 4 | PRODUCER-CONSUMER PROBLEM | 17-19 |
| 5 | DINERS-PHILOSOPHERS PROBLEM | 20-22 |
| 6 | BANKERS ALGORITHM(DEADLOCK AVOIDANCE) | 23-25 |
| 7 | DEADLOCK DETECTION | 26-30 |
| 8 | CONTIGUOUS MEMORY ALLOCATION(FIRST, BEST, WORST FIT) | 31-36 |
| 9 | PAGE REPLACEMENT(FIFO, LRU, OPTIMAL) | 37-42 |
| 10 | DISK SCHEDULING ALGORITHMS(FCFS, SCAN, C-SCAN) | 43-49 |

**Course Outcomes**

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

Systems

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

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

**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:**

#include <stdio.h>

int n, i, j, pos, temp, choice, Burst\_time[20], Waiting\_time[20],

Turn\_around\_time[20], process[20], total = 0;

float avg\_Turn\_around\_time = 0, avg\_Waiting\_time = 0;

int FCFS()

{

Waiting\_time[0] = 0;

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

{

printf("\nAverage Turnaround Time:%.2f\n", avg\_Turn\_around\_time);

return 0;

}

}

int SJF()

{

// sorting

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

{

pos = i;

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

{

if (Burst\_time[j] < Burst\_time[pos])

pos = j;

}

temp = Burst\_time[i];

Burst\_time[i] = Burst\_time[pos];

Burst\_time[pos] = temp;

temp = process[i];

process[i] = process[pos];

process[pos] = temp;

}

Waiting\_time[0] = 0;

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

{

Waiting\_time[i] = 0;

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

Waiting\_time[i] += Burst\_time[j];

total += Waiting\_time[i];

}

avg\_Waiting\_time = (float)total / n;

total = 0;

printf("\nProcess\t\tBurst Time\t\tWaiting Time\t\tTurnaround Time");

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

{

Turn\_around\_time[i] = Burst\_time[i] + Waiting\_time[i];

total += Turn\_around\_time[i];

printf("P[%d]:", i + 1);

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

process[i] = i + 1;

}

while (1)

{

printf("\n-----MAIN MENU-----\n");

printf("1. FCFS Scheduling\n2. SJF Scheduling\n");

printf("\nEnter your choice:");

scanf("%d", &choice);

switch (choice)

{

case 1:

FCFS();

break;

case 2:

SJF();

break;

default:

printf("Invalid Input!!!");

}

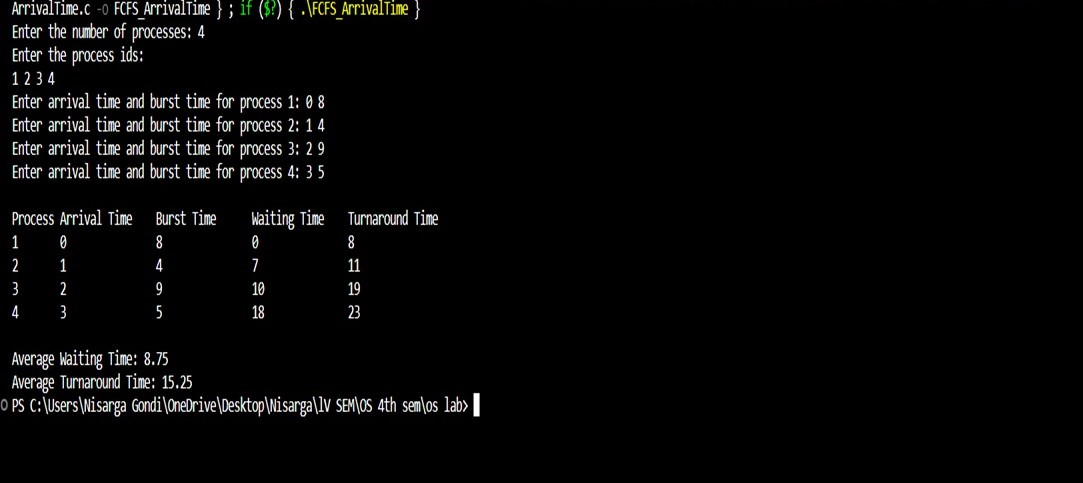
}

return 0;

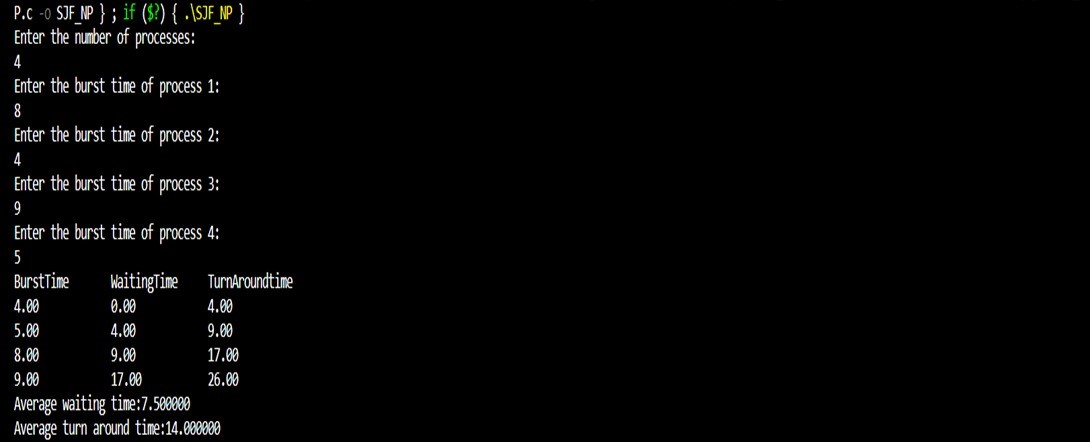
}

* + 1. **Output:**

**a.**



**b.**



**Experiment - 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)**
   * 1. **Code:**
3. **Priority (Non-pre-emptive)**

#include<stdio.h>

#include<stdlib.h>

struct Process {

int pid;

int burst\_time;

int arrival\_time;

int priority;

int waiting\_time;

int turnaround\_time;

};

void findWaitingTime(struct Process proc[], int n) {

int completed[n];

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

completed[i] = 0;

int t = 0;

int completed\_count = 0;

while (completed\_count < n) {

int min\_priority = 10000;

int idx = -1;

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

if (proc[i].arrival\_time <= t && !completed[i] && proc[i].priority < min\_priority) {

min\_priority = proc[i].priority;

idx = i;

}

}

if (idx != -1) {

t += proc[idx].burst\_time;

proc[idx].waiting\_time = t - proc[idx].burst\_time - proc[idx].arrival\_time;

if (proc[idx].waiting\_time < 0)

proc[idx].waiting\_time = 0;

completed[idx] = 1;

completed\_count++;

} else {

t++;

}

}

}

void findTurnaroundTime(struct Process proc[], int n) {

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

proc[i].turnaround\_time = proc[i].burst\_time + proc[i].waiting\_time;

}

void findAverageTime(struct Process proc[], int n) {

int total\_waiting\_time = 0, total\_turnaround\_time = 0;

findWaitingTime(proc, n);

findTurnaroundTime(proc, n);

printf("Processes Burst time Arrival time Priority Waiting time Turnaround time\n");

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

total\_waiting\_time += proc[i].waiting\_time;

total\_turnaround\_time += proc[i].turnaround\_time;

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

}

printf("Average waiting time = %.2f\n", (float)total\_waiting\_time / (float)n);

printf("Average turnaround time = %.2f\n", (float)total\_turnaround\_time / (float)n);

}

int main() {

struct Process proc[] = {{1, 6, 0, 2}, {2, 8, 1, 1}, {3, 7, 2, 3}, {4, 3, 3, 2}};

int n = sizeof(proc) / sizeof(proc[0]);

findAverageTime(proc, n);

return 0;

}

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)

{

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]);

}

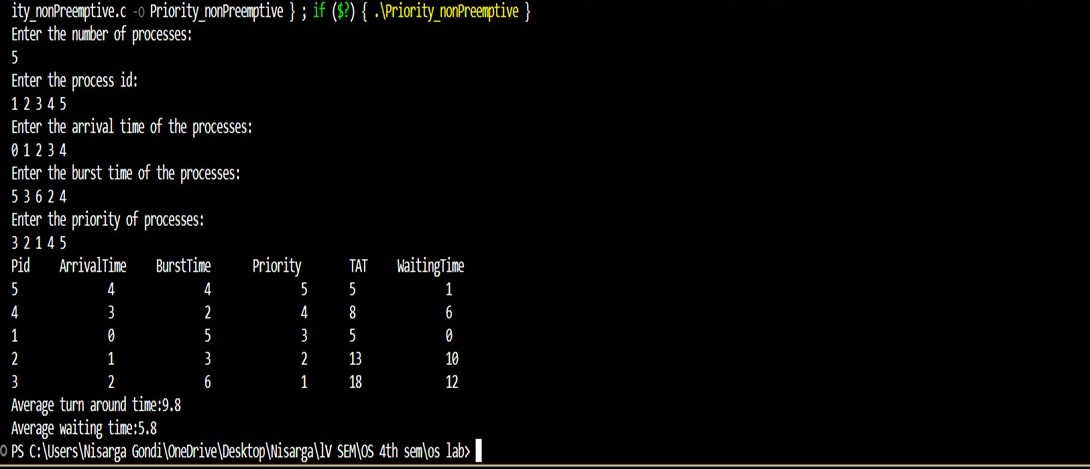
findavgTime(processes, n, burst\_time, quantum);

return 0;

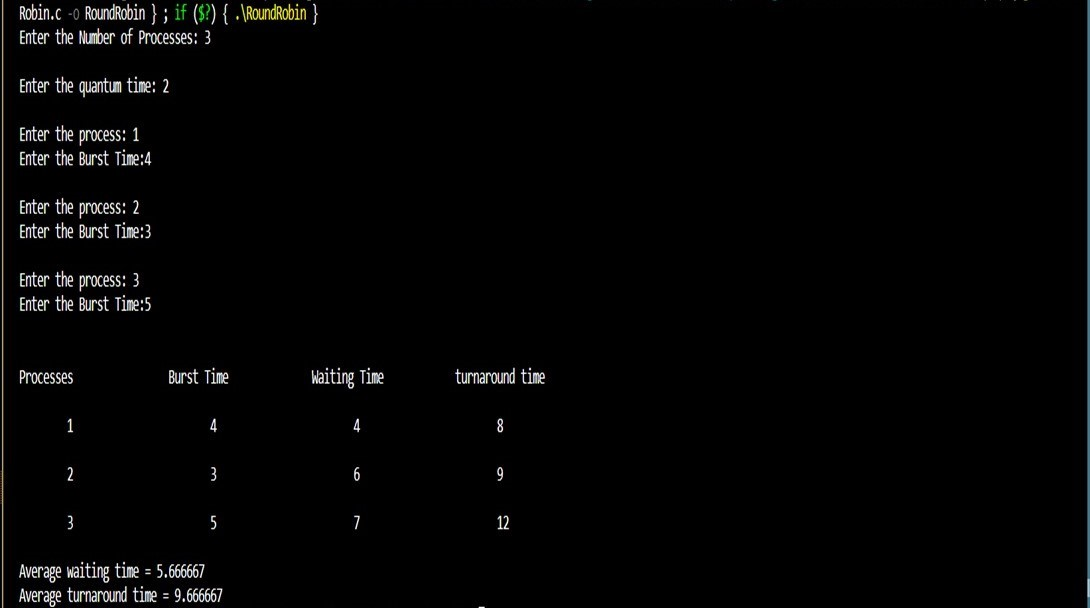
}

**2.2.3 Output:**

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



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



* 1. **Experiment - 3**
     1. **Question:**

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

1. **Rate- Monotonic**
2. **Earliest-deadline First**
   * 1. **Code:**

**(a) Rate-Monotonic**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#

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

{

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

printf("|####");

else

printf("| ");

}

printf("|\n");

}

}

void rate\_monotonic(int time)

{

int

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

{

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

next\_process = k;

}

}

}

print\_schedule(process\_list, time);

}

void

}

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];

}

print\_schedule(process\_list, time);

}

int main()

{

int option;

}

return 0;

}

**(b) Earliest Deadline FIrst**

#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int deadline[]) {

    wt[0] = 0;

    for (int i = 1; i < n; i++) {

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

    }

}

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

    for (int i = 0; i < n; i++) {

        tat[i] = bt[i] + wt[i];

    }

}

void findAvgTime(int processes[], int n, int bt[], int deadline[]) {

    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, wt, deadline);

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

    printf("Processes   Burst time   Waiting time   Turnaround time   Deadline\n");

    for (int i = 0; i < n; i++) {

        printf(" %d ", (i + 1));

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

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

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

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

    }

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

    for (int i = 0; i < n; i++) {

        total\_wt += wt[i];

        total\_tat += tat[i];

    }

    printf("Average waiting time = %.2f\n", (float)total\_wt / (float)n);

    printf("Average turnaround time = %.2f\n", (float)total\_tat / (float)n);

}

void earliestDeadlineFirstScheduling(int processes[], int n, int bt[], int deadline[]) {

    // Sort by deadline

    for (int i = 0; i < n - 1; i++) {

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

            if (deadline[j] > deadline[j + 1]) {

                int temp = deadline[j];

                deadline[j] = deadline[j + 1];

                deadline[j + 1] = temp;

                temp = bt[j];

                bt[j] = bt[j + 1];

                bt[j + 1] = temp;

                temp = processes[j];

                processes[j] = processes[j + 1];

                processes[j + 1] = temp;

            }

        }

    }

    findAvgTime(processes, n, bt, deadline);

}

int main() {

    int processes[] = {1, 2, 3};

    int n = sizeof(processes) / sizeof(processes[0]);

    int burst\_time[] = {3, 1, 2};

    int deadline[] = {7, 4, 5};

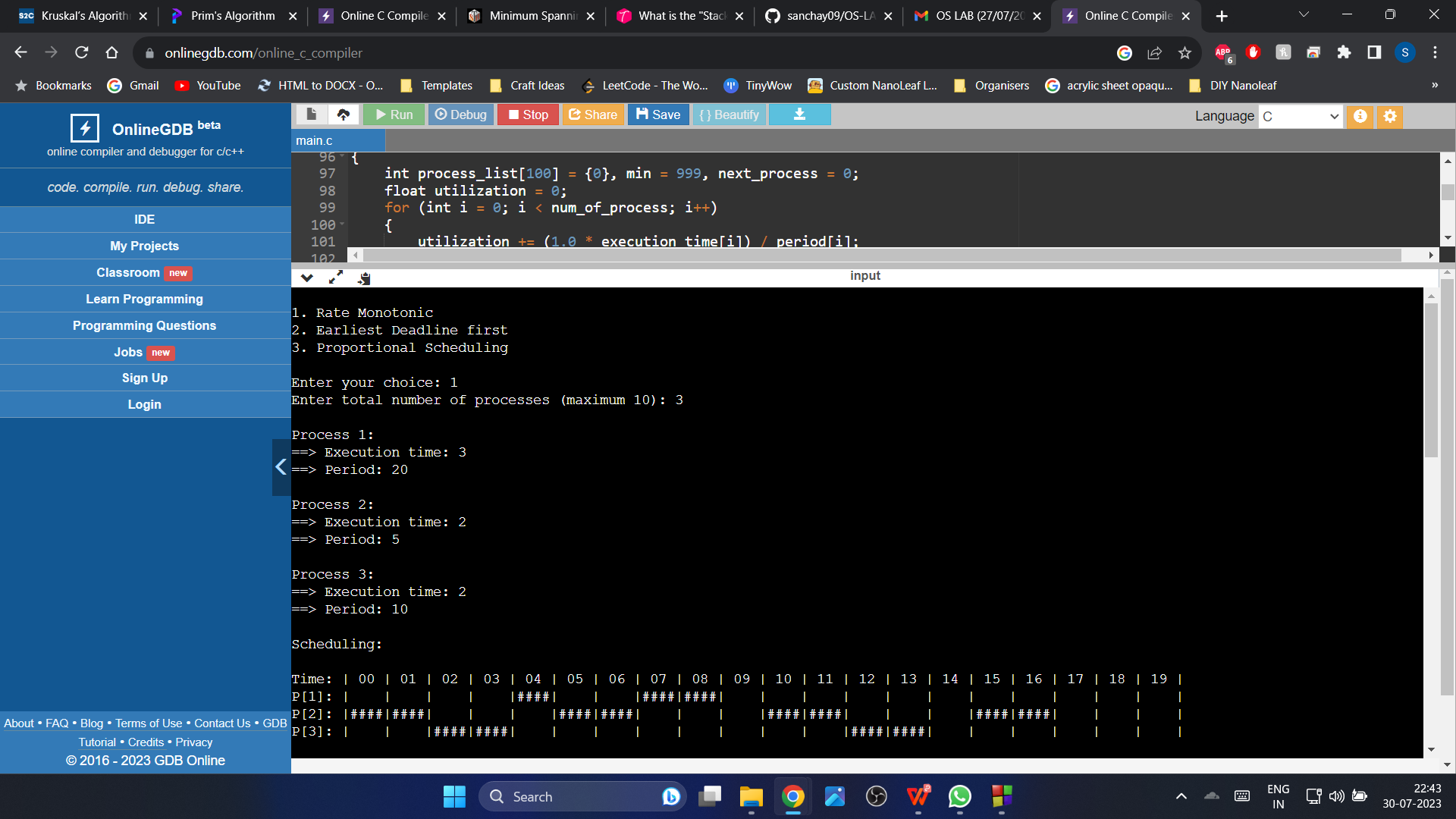
    earliestDeadlineFirstScheduling(processes, n, burst\_time, deadline);

    return 0;

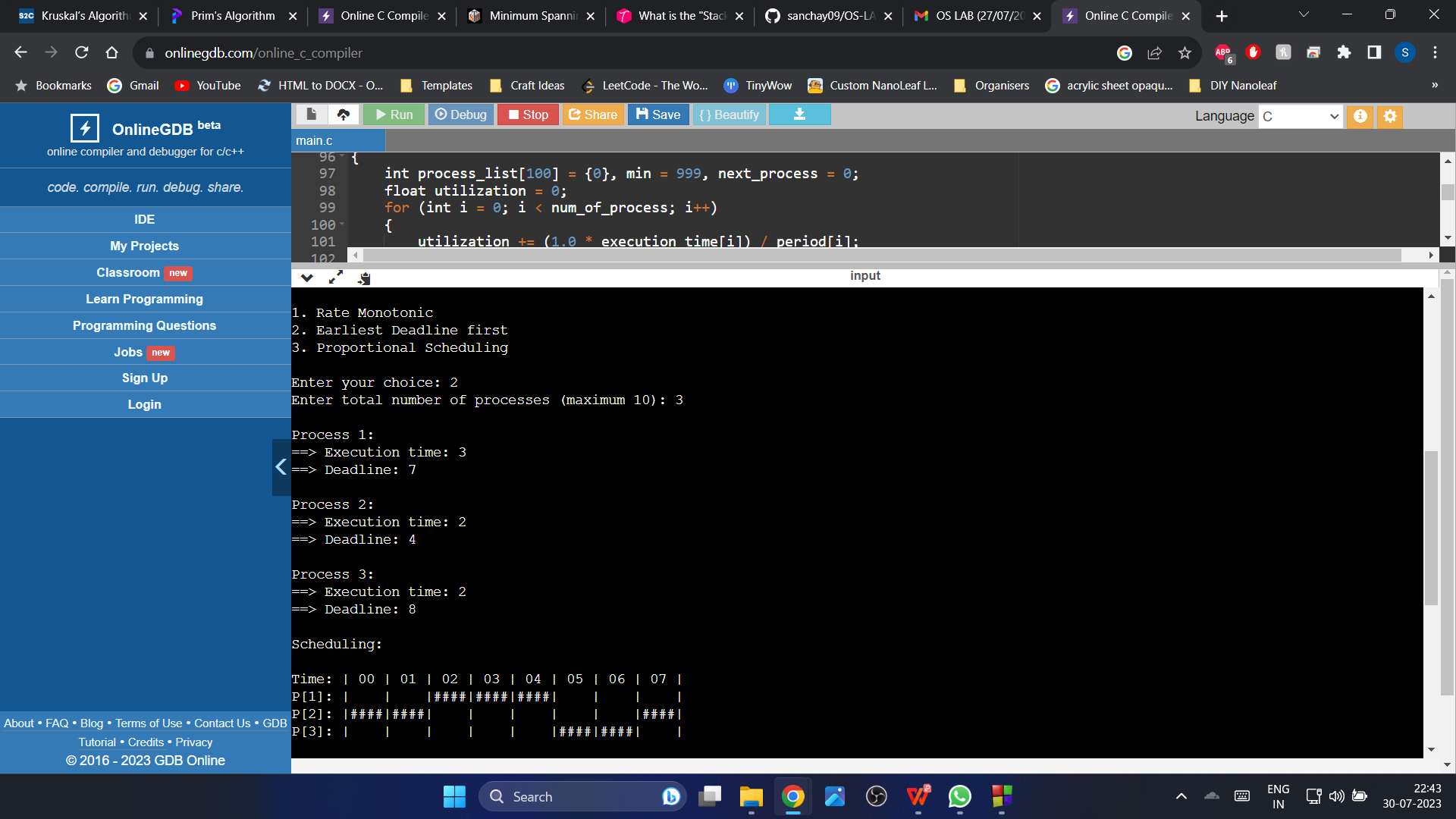
}

* + 1. **Output:**

1. **Rate Monotonic:**



1. **Earliest Deadline First:**



**Experiment - 4**

* + 1. **Question:**

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

* + 1. **Code**:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*param) {

    int item;

    while (1) {

        item = rand() % 100;

        sem\_wait(&empty);

        pthread\_mutex\_lock(&mutex);

        buffer[in] = item;

        printf("Producer produced %d at %d\n", item, in);

        in = (in + 1) % BUFFER\_SIZE;

        pthread\_mutex\_unlock(&mutex);

        sem\_post(&full);

        sleep(1);

    }

}

void \*consumer(void \*param) {

    int item;

    while (1) {

        sem\_wait(&full);

        pthread\_mutex\_lock(&mutex);

        item = buffer[out];

        printf("Consumer consumed %d from %d\n", item, out);

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

        pthread\_mutex\_unlock(&mutex);

        sem\_post(&empty);

        sleep(1);

    }

}

int main() {

    pthread\_t tid1, tid2;

    pthread\_attr\_t attr;

    pthread\_attr\_init(&attr);

    pthread\_mutex\_init(&mutex, NULL);

    sem\_init(&empty, 0, BUFFER\_SIZE);

    sem\_init(&full, 0, 0);

    pthread\_create(&tid1, &attr, producer, NULL);

    pthread\_create(&tid2, &attr, consumer, NULL);

    pthread\_join(tid1, NULL);

    pthread\_join(tid2, NULL);

    pthread\_mutex\_destroy(&mutex);

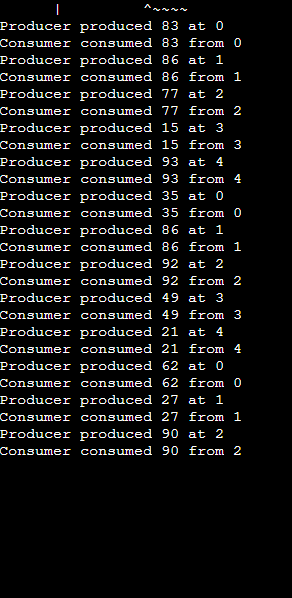
    sem\_destroy(&empty);

    sem\_destroy(&full);

    return 0;

}

* + 1. **Output:**



**EXPERIMENT - 5**

**1.2.7 Question:**

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

**1.2.8 Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#define N

sem\_t forks[N];

sem\_t mutex;

void \*philosopher(void \*num) {

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

while (1) {

printf("Philosopher %d is thinking.\n", id);

sleep(1);

sem\_wait(&mutex);

sem\_wait(&forks[id]);

sem\_wait(&forks[(id + 1) % N]);

printf("Philosopher %d is eating.\n", id);

sleep(1);

sem\_post(&forks[id]); // Put down chopsticks

sem\_post(&forks[(id + 1) % N]);

sem\_post(&mutex);

printf("Philosopher %d is done eating and starts thinking again.\n", id);

sleep(1);

}

}

int main() {

pthread\_t tid[N];

int ids[N];

sem\_init(&mutex, 0, 1);

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

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

ids[i] = i;

}

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

pthread\_create(&tid[i], NULL, philosopher, &ids[i]);

}

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

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

}

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

sem\_destroy(&forks[i]);

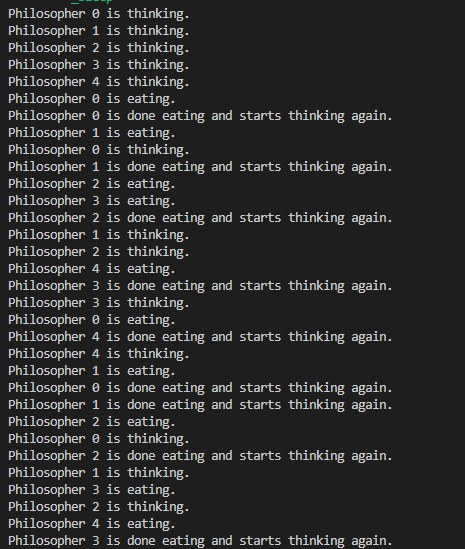
}

sem\_destroy(&mutex);

return 0;

}

**1.2.9 Output:**



**EXPERIMENT - 6**

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

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

int main() {

    int n, m, i, j, k;

    n = 5;

    m = 3;

    int alloc[MAX\_PROCESSES][MAX\_RESOURCES] = { { 0, 1, 0 },

                                                { 2, 0, 0 },

                                                { 3, 0, 2 },

                                                { 2, 1, 1 },

                                                { 0, 0, 2 } };

    int max[MAX\_PROCESSES][MAX\_RESOURCES] = { { 7, 5, 3 },

                                              { 3, 2, 2 },

                                              { 9, 0, 2 },

                                              { 2, 2, 2 },

                                              { 4, 3, 3 } };

    int avail[MAX\_RESOURCES] = { 3, 3, 2 };

    int f[MAX\_PROCESSES], ans[MAX\_PROCESSES], ind = 0;

    for (k = 0; k < n; k++) {

        f[k] = 0;

    }

    int need[MAX\_PROCESSES][MAX\_RESOURCES];

    for (i = 0; i < n; i++) {

        for (j = 0; j < m; j++) {

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

        }

    }

    printf("Need matrix:\n");

    for (i = 0; i < n; i++) {

        for (j = 0; j < m; j++) {

            printf("%d ", need[i][j]);

        }

        printf("\n");

    }

    int y = 0;

    for (k = 0; k < n; k++) {

        for (i = 0; i < n; i++) {

            if (f[i] == 0) {

                bool flag = true;

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

                    if (need[i][j] > avail[j]) {

                        flag = false;

                        break;

                    } }

                if (flag) {

                    ans[ind++] = i;

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

                        avail[y] += alloc[i][y];

                    }

                    f[i] = 1;

                }

            }

        }

    }

    printf("Following is the SAFE Sequence:\n");

    for (i = 0; i < n - 1; i++) {

        printf(" P%d ->", ans[i]);

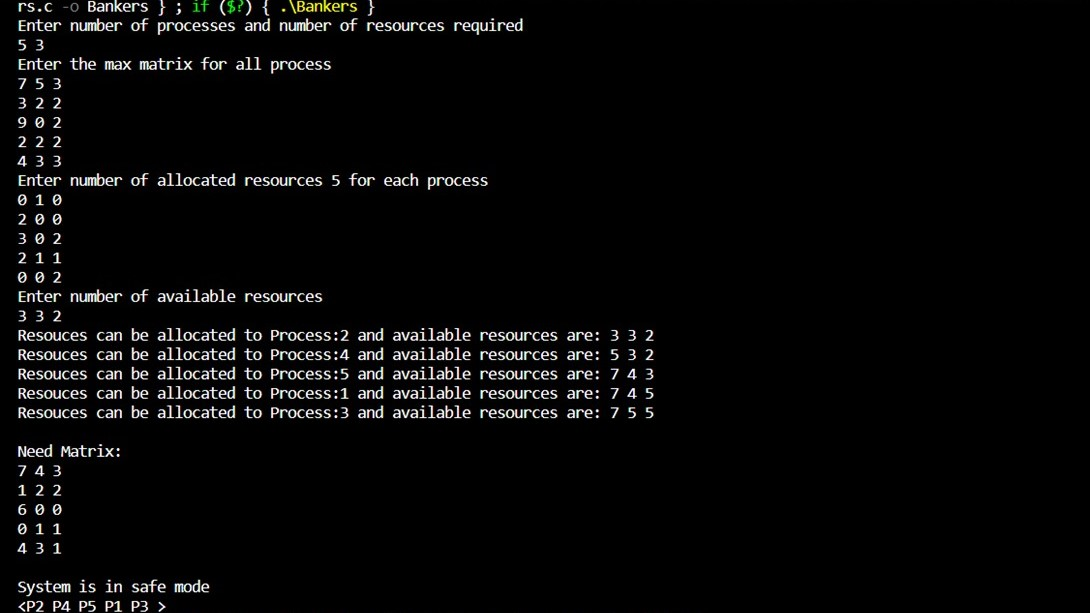
    }

    printf(" P%d\n", ans[n - 1]);

    return 0;

}

**Output:**



**EXPERIMENT - 7**

**Write a C program to simulate deadlock detection**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3

void printMatrices(int processes, int resources, int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int need[MAX\_PROCESSES][MAX\_RESOURCES], int avail[MAX\_RESOURCES]) {

printf("Allocation Matrix:\n");

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

for (int j = 0; j < resources; j++) {

printf("%d ", alloc[i][j]);

}

printf("\n");

}

printf("\nMax Matrix:\n");

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

for (int j = 0; j < resources; j++) {

printf("%d ", max[i][j]);

}

printf("\n");

}

printf("\nNeed Matrix:\n");

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

for (int j = 0; j < resources; j++) {

printf("%d ", need[i][j]);

}

printf("\n");

}

printf("\nAvailable Resources:\n");

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

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

}

printf("\n");

}

void deadlockDetection(int processes, int resources, int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int avail[MAX\_RESOURCES]) {

int need[MAX\_PROCESSES][MAX\_RESOURCES];

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES];

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

for (int j = 0; j < resources; j++) {

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

}

}

printMatrices(processes, resources, alloc, max, need, avail);

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

work[i] = avail[i];

}

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

finish[i] = false;

}

bool found;

do {

found = false;

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

if (!finish[i]) {

bool flag = true;

for (int j = 0; j < resources; j++) {

if (need[i][j] > work[j]) {

flag = false;

break;

}

}

if (flag) {

printf("\nProcess %d can be satisfied and is now finishing.\n", i);

for (int k = 0; k < resources; k++) {

work[k] += alloc[i][k];

}

finish[i] = true;

found = true;

printf("New Available Resources:\n");

for (int k = 0; k < resources; k++) {

printf("%d ", work[k]);

}

printf("\n");

}

}

}} while (found);

bool deadlock = false;

printf("\nDeadlock Check:\n");

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

if (!finish[i]) {

deadlock = true;

printf("Process %d is in a deadlock.\n", i);

}

}

if (!deadlock) {

printf("No deadlock detected.\n");

}

}

int main() {

int processes = 5;

int resources = 3;

int alloc[MAX\_PROCESSES][MAX\_RESOURCES] = {

{ 0, 1, 0 },

{ 2, 0, 0 },

{ 3, 0, 2 },

{ 2, 1, 1 },

{ 0, 0, 2 }

};

int max[MAX\_PROCESSES][MAX\_RESOURCES] = {

{ 7, 5, 3 },

{ 3, 2, 2 },

{ 9, 0, 2 },

{ 2, 2, 2 },

{ 4, 3, 3 }

};

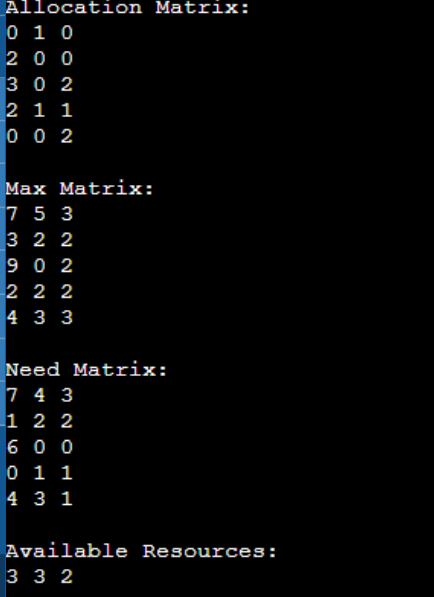
int avail[MAX\_RESOURCES] = { 3, 3, 2 }; // Available resources

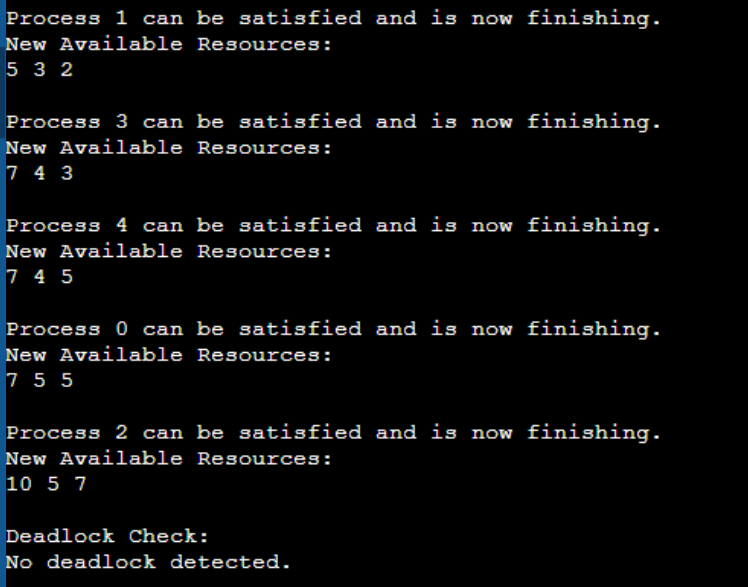
deadlockDetection(processes, resources, alloc, max, avail);

return 0;

}

**Output:**





**EXPERIMENT - 8**

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

**a) Worst-fit**

**b) Best-fit**

**c) First-fit**

#include <stdio.h>

#include <stdlib.h>

#define MAX 25

void firstFit(int nb, int nf, int b[], int f[]) {

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        allocation[i] = -1;

        for (int j = 0; j < nb; j++) {

            if (allocated[j] == 0 && b[j] >= f[i]) {

                allocation[i] = j;

                allocated[j] = 1;

                break;

            }

        }

    }

    printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:");

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}

void bestFit(int nb, int nf, int b[], int f[]) {

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        int bestIdx = -1;

        allocation[i] = -1;

        for (int j = 0; j < nb; j++) {

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (bestIdx == -1 || b[j] < b[bestIdx])

                    bestIdx = j;

            }

        }

        if (bestIdx != -1) {

            allocation[i] = bestIdx;

            allocated[bestIdx] = 1;

        }

    }

    printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:");

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}

void worstFit(int nb, int nf, int b[], int f[]) {

    int allocation[MAX];

    int allocated[MAX] = {0};

    for (int i = 0; i < nf; i++) {

        int worstIdx = -1;

        allocation[i] = -1;

        for (int j = 0; j < nb; j++) {

            if (allocated[j] == 0 && b[j] >= f[i]) {

                if (worstIdx == -1 || b[j] > b[worstIdx])

                    worstIdx = j;

            }

        }

        if (worstIdx != -1) {

            allocation[i] = worstIdx;

            allocated[worstIdx] = 1;

        }

    }

    printf("\nFile\_no:\tFile\_size:\tBlock\_no:\tBlock\_size:");

    for (int i = 0; i < nf; i++) {

        if (allocation[i] != -1)

            printf("\n%d\t\t%d\t\t%d\t\t%d", i + 1, f[i], allocation[i] + 1, b[allocation[i]]);

        else

            printf("\n%d\t\t%d\t\t-\t\t-", i + 1, f[i]);

    }

}

int main() {

    int nb, nf, choice;

    printf("Memory Management Scheme");

    printf("\nEnter the number of blocks: ");

    scanf("%d", &nb);

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

    scanf("%d", &nf);

    int b[nb], f[nf];

    printf("\nEnter the size of the blocks:\n");

    for (int i = 0; i < nb; i++) {

        printf("Block %d: ", i + 1);

        scanf("%d", &b[i]);

    }

    printf("Enter the size of the files:\n");

    for (int i = 0; i < nf; i++) {

        printf("File %d: ", i + 1);

        scanf("%d", &f[i]);

    }

    while (1) {

        printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");

        printf("Enter your choice: ");

        scanf("%d", &choice);

        switch (choice) {

            case 1:

                printf("\n\tMemory Management Scheme - First Fit\n");

                firstFit(nb, nf, b, f);

                break;

            case 2:

                printf("\n\tMemory Management Scheme - Best Fit\n");

                bestFit(nb, nf, b, f);

                break;

            case 3:

                printf("\n\tMemory Management Scheme - Worst Fit\n");

                worstFit(nb, nf, b, f);

                break;

            case 4:

                printf("\nExiting...\n");

                exit(0);

                break;

            default:

                printf("\nInvalid choice.\n");

                break;

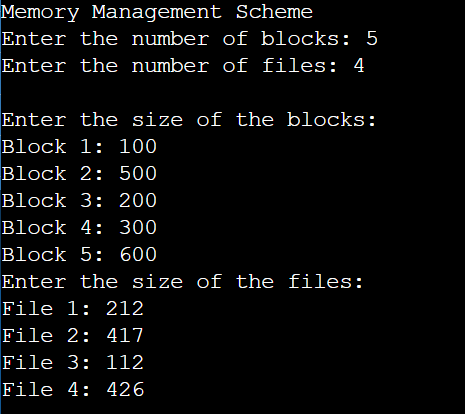
        }

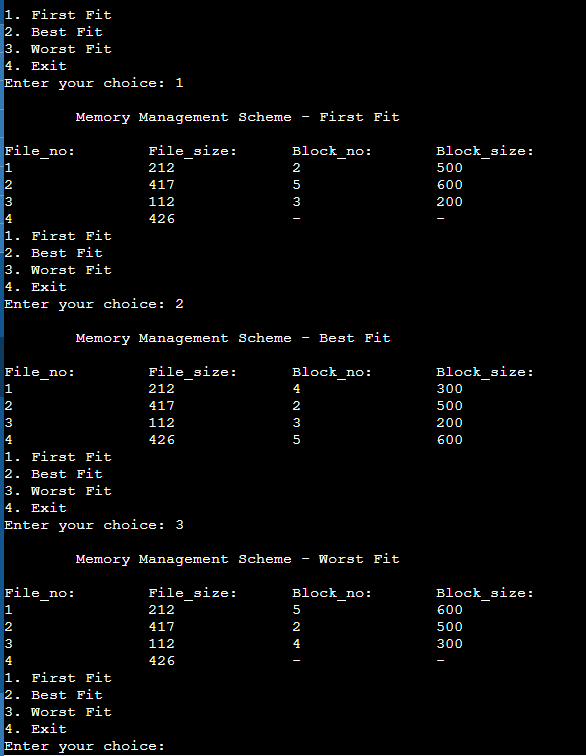
    }

    return 0;

}

**Output:**





**EXPERIMENT - 9**

**Write a C program to simulate page replacement algorithms**

**a) FIFO**

**b) LRU**

**c) Optimal**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 25

void fifo(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0, frameIndex = 0;

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

        for (int j = 0; j < frameCount; j++) {

            if (frame[j] == pages[i]) {

                isPagePresent = true;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                frameCount++;

            } else {

                frame[frameIndex] = pages[i];

                frameIndex++;

                if (frameIndex >= capacity)

                    frameIndex = 0;

            }

            pageFaults++;

        }

    }

    printf("\nFIFO Page Replacement Algorithm:\n");

    printf("Total Page Faults: %d\n", pageFaults);

}

void lru(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0, counter[MAX\_FRAMES];

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

        for (int j = 0; j < frameCount; j++) {

            if (frame[j] == pages[i]) {

                isPagePresent = true;

                counter[j] = i;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                counter[frameCount] = i;

                frameCount++;

            } else {

                int lru = 0;

                for (int j = 1; j < capacity; j++) {

                    if (counter[j] < counter[lru])

                        lru = j;

                }

                frame[lru] = pages[i];

                counter[lru] = i;

            }

            pageFaults++;

        }

    }

    printf("\nLRU Page Replacement Algorithm:\n");

    printf("Total Page Faults: %d\n", pageFaults);

}

void optimal(int pages[], int n, int capacity) {

    int frame[MAX\_FRAMES], frameCount = 0, pageFaults = 0;

    bool isPagePresent = false;

    for (int i = 0; i < n; i++) {

        isPagePresent = false;

        for (int j = 0; j < frameCount; j++) {

            if (frame[j] == pages[i]) {

                isPagePresent = true;

                break;

            }

        }

        if (isPagePresent == false) {

            if (frameCount < capacity) {

                frame[frameCount] = pages[i];

                frameCount++;

            } else {

                int future[MAX\_FRAMES] = {0};

                for (int j = 0; j < frameCount; j++) {

                    bool isFound = false;

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

                        if (pages[k] == frame[j]) {

                            future[j] = k;

                            isFound = true;

                            break;

                        }

                    }

                    if (isFound == false)

                        future[j] = n + 1;

                }

                int longest = 0;

                for (int j = 1; j < frameCount; j++) {

                    if (future[j] > future[longest])

                        longest = j;

                }

                frame[longest] = pages[i];

            }

            pageFaults++;

        }

    }

    printf("\nOptimal Page Replacement Algorithm:\n");

    printf("Total Page Faults: %d\n", pageFaults);

}

int main() {

    int pages[MAX\_PAGES], n, capacity;

    printf("Page Replacement Algorithms\n");

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

    scanf("%d", &n);

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

    for (int i = 0; i < n; i++) {

        printf("Page %d: ", i + 1);

        scanf("%d", &pages[i]);

    }

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

    scanf("%d", &capacity);

    fifo(pages, n, capacity);

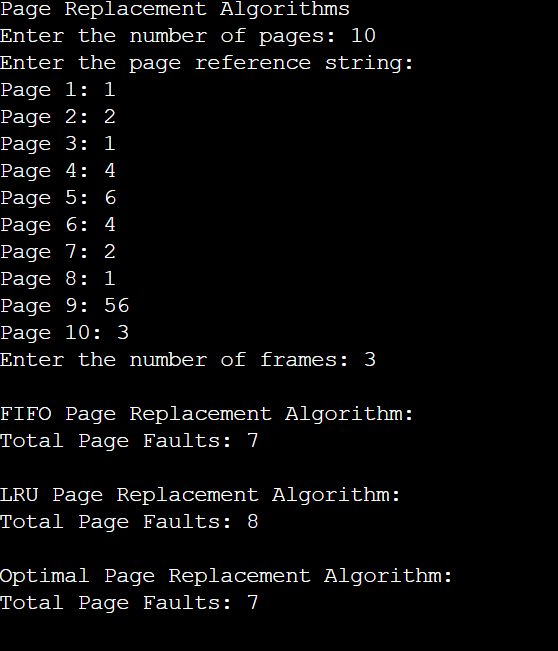
    lru(pages, n, capacity);

    optimal(pages, n, capacity);

    return 0;

}

**Output:**



**Experiment - 10**

**Write a C program to simulate disk scheduling algorithms:**

1. **FCFS**
2. **SCAN**
3. **c-SCAN**
4. **FCFS:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,n,TotalHeadMoment=0,initial;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

// logic for FCFS disk scheduling

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

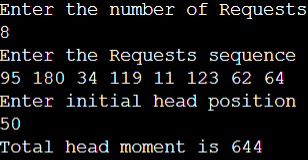
}

printf("Total head moment is %d",TotalHeadMoment);

return 0;

}

**Output:**

****

1. **SCAN:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for Scan disk scheduling

/\*logic for sort the request array \*/

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

{

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

{

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

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

//if movement is towards high value

if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

initial = size-1;

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

initial =0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

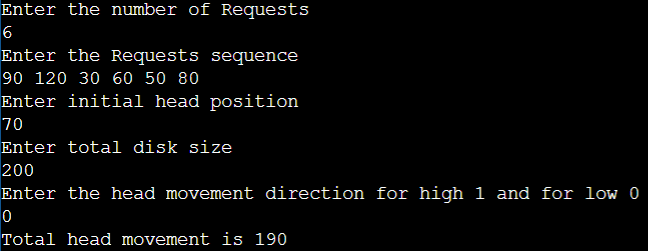
}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Output:**

****

1. **C-SCAN:**

#include<stdio.h>

#include<stdlib.h>

int main()

{

int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move;

printf("Enter the number of Requests\n");

scanf("%d",&n);

printf("Enter the Requests sequence\n");

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

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

printf("Enter initial head position\n");

scanf("%d",&initial);

printf("Enter total disk size\n");

scanf("%d",&size);

printf("Enter the head movement direction for high 1 and for low 0\n");

scanf("%d",&move);

// logic for C-Scan disk scheduling

/\*logic for sort the request array \*/

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

{

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

{

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

{

int temp;

temp=RQ[j];

RQ[j]=RQ[j+1];

RQ[j+1]=temp;

}

}

}

int index;

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

{

if(initial<RQ[i])

{

index=i;

break;

}

}

// if movement is towards high value

if(move==1)

{

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for max size

TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);

/\*movement max to min disk \*/

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial=0;

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

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

}

// if movement is towards low value

else

{

for(i=index-1;i>=0;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

// last movement for min size

TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);

/\*movement min to max disk \*/

TotalHeadMoment=TotalHeadMoment+abs(size-1-0);

initial =size-1;

for(i=n-1;i>=index;i--)

{

TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);

initial=RQ[i];

}

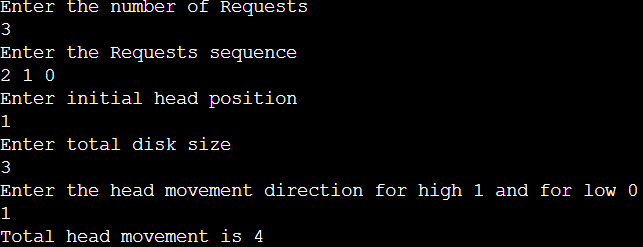
}

printf("Total head movement is %d",TotalHeadMoment);

return 0;

}

**Output:**

****