VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

OPERATING SYSTEMS

Submitted by

TARUN GOWDA V (1WA23CS014)

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

Feb-2025 to June-2025

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 – 23CS4PCOPS” carried out by TARUN GOWDA V(1WA23CS014), who is Bonafide student of B. M. S. College of Engineering. It is in partial fulfilment for the award of Bachelor of Engineering in Computer

Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year Feb 2025- June 2025. The Lab report has been approved as it satisfies the academic requirements in respect of a OPERATING SYSTEMS - (23CS4PCOPS) work prescribed for the said degree.

Faculty Incharge Name Dr. Seema Patil Dr. Kavitha Sooda

Assistant Professor Professor and Head

Department of CSE Department of CSE

BMSCE, Bengaluru BMSCE, Bengaluru

Index Sheet

|  |  |  |
| --- | --- | --- |
| Sl.  No. | Experiment Title | Page No. |
| 1. | Write a C program to simulate the following non-pre-emptive CPU scheduling algorithm to find turnaround time and waiting time.  .FCFS  . SJF (pre-emptive & Non-preemptive) | 1-10 |
| 2. | Write a C program to simulate the following CPU scheduling algorithm to find turnaround time and waiting time.  . Priority (pre-emptive & Non-preemptive)  .Round Robin (Experiment with different quantum sizes for  RR algorithm) | 11-15 |
| 3. | 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. | 16-20 |
| 4. | Write a C program to simulate Real-Time CPU Scheduling algorithms:   1. Rate- Monotonic 2. Earliest-deadline First | 21-28 |
| 5. | Write a C program to simulate producer-consumer problem using semaphores | 29-31 |
| 6. | Write a C program to simulate the concept of Dining Philosophers problem. | 32-35 |
| 7. | Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance. | 36-38 |
| 8. | Write a C program to simulate deadlock detection | 39-41 |
| 9. | Write a C program to simulate the following contiguous memory allocation techniques a) Worst-fit b) Best-fit  c) First-fit | 42-45 |
| 10. | Write a C program to simulate page replacement algorithms a) FIFO  b)LRU  c)Optimal | 46-48 |

Course Outcomes

|  |  |
| --- | --- |
| C01 | Apply the different concepts and functionalities of Operating System |
| C02 | Analyse various Operating system strategies and techniques |
| C03 | Demonstrate the different functionalities of Operating System. |
| C04 | Conduct practical experiments to implement the functionalities of Operating system. |

Program -1

Question:

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

.FCFS

. SJF (pre-emptive & Non-preemptive)

.FCFS:

#include <stdio.h>

typedef struct { int id, at, bt, wt, tat, ct,rem,rt, started;

} Process;

void sortByArrival(Process p[], int n) { for (int i = 0; i < n - 1; i++) { for (int j = 0; j < n - i - 1; j++) { if (p[j].at > p[j + 1].at) { Process temp = p[j]; p[j] = p[j + 1]; p[j + 1] = temp;

}

}

}

}

void fcfs(Process p[], int n) { sortByArrival(p, n); int time = 0;

for (int i = 0; i < n; i++) { if (time < p[i].at) time = p[i].at;

p[i].ct = time + p[i].bt; p[i].tat = p[i].ct - p[i].at; p[i].wt = p[i].tat - p[i].bt; time = p[i].ct;

}

}

void display(Process p[], int n) {

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");

float totalWT = 0, totalTAT = 0; for (int i = 0; i < n; i++) { printf("P%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].at, p[i].bt, p[i].ct, p[i].tat, p[i].wt); totalWT += p[i].wt; totalTAT += p[i].tat;

}

printf("\nAverage Waiting Time: %.2f\n", totalWT / n); printf("Average Turnaround Time: %.2f\n", totalTAT / n);

}

void main() {

int n; printf("Enter number of processes: "); scanf("%d", &n); Process p[n]; printf("Enter Arrival Time and Burst Time for each process:\n"); for (int i = 0; i < n; i++) { p[i].id = i + 1; printf("P[%d]: ", i + 1);

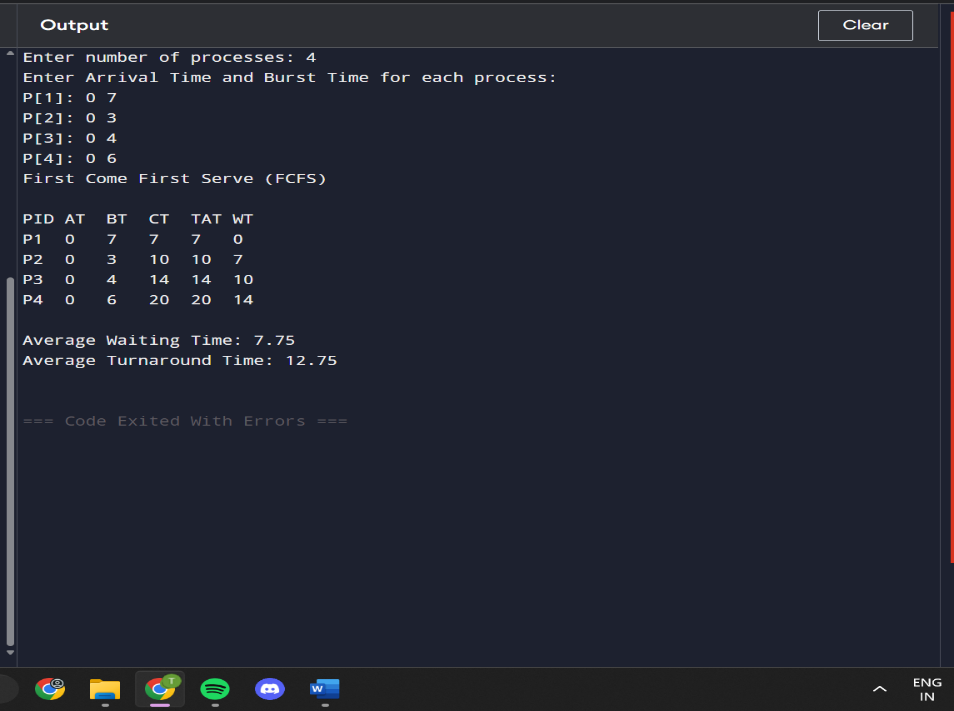
scanf("%d %d", &p[i].at, &p[i].bt);

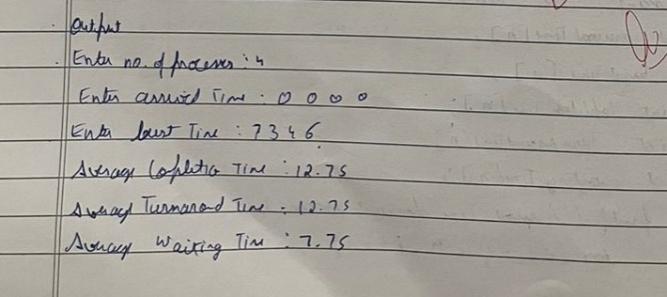
}

printf("First Come First Serve (FCFS)\n"); fcfs(p, n); display(p, n);

}

Result:





.SJF(Non-preemptive):

#include <stdio.h> #include <limits.h> typedef struct {

int id, arrival, burst, completion, turnaround, waiting;

} Process;

void sortByArrival(Process p[], int n) { for (int i = 0; i < n - 1; i++) { for (int j = 0; j < n - i - 1; j++) { if (p[j].arrival > p[j + 1].arrival) { Process temp = p[j]; p[j] = p[j + 1]; p[j + 1] = temp;

}

}

}

}

void sjf\_non\_preemptive(Process p[], int n) { int completed = 0, time = 0, minIdx; int isCompleted[n];

for (int i = 0; i < n; i++) isCompleted[i] = 0; while (completed < n) { minIdx = -1; int minBurst = INT\_MAX; for (int i = 0; i < n; i++) {

if (!isCompleted[i] && p[i].arrival <= time && p[i].burst < minBurst) { minBurst = p[i].burst; minIdx = i;

}

}

if (minIdx == -1) { time++; continue; } p[minIdx].completion = time + p[minIdx].burst;

p[minIdx].turnaround = p[minIdx].completion - p[minIdx].arrival; p[minIdx].waiting = p[minIdx].turnaround - p[minIdx].burst; time = p[minIdx].completion; isCompleted[minIdx] = 1; completed++;

}

}

void display(Process p[], int n) {

printf("\nPID Arrival Burst Completion Turnaround Waiting\n"); float totalWT = 0, totalTAT = 0; for (int i = 0; i < n; i++) {

printf("%3d %7d %6d %10d %10d %8d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion, p[i].turnaround, p[i].waiting); totalWT += p[i].waiting;

totalTAT += p[i].turnaround;

}

printf("\nAverage Waiting Time: %.2f\n", totalWT / n); printf("Average Turnaround Time: %.2f\n", totalTAT / n);

}

int main() { int n;

printf("Enter number of processes: "); scanf("%d", &n); Process p[n];

printf("Enter Arrival Time and Burst Time for each process:\n"); for (int i = 0; i < n; i++) { p[i].id = i + 1; printf("P[%d]: ", i + 1);

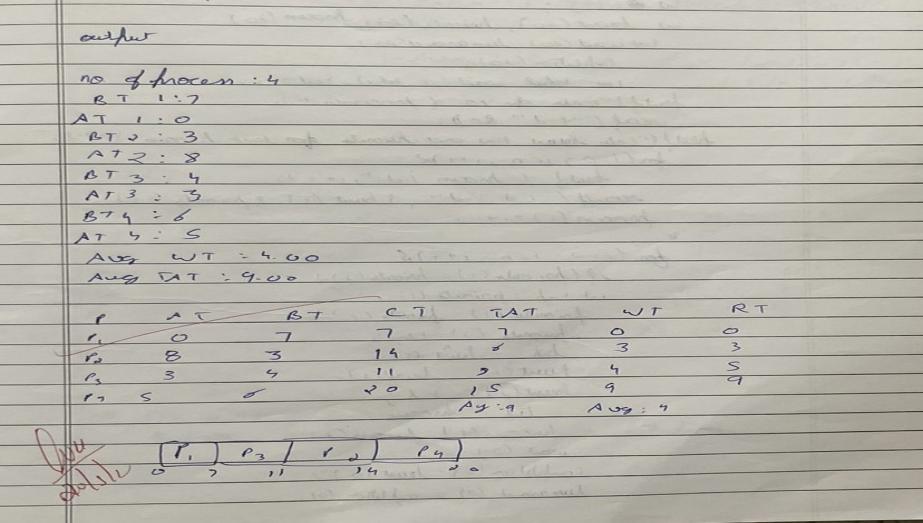
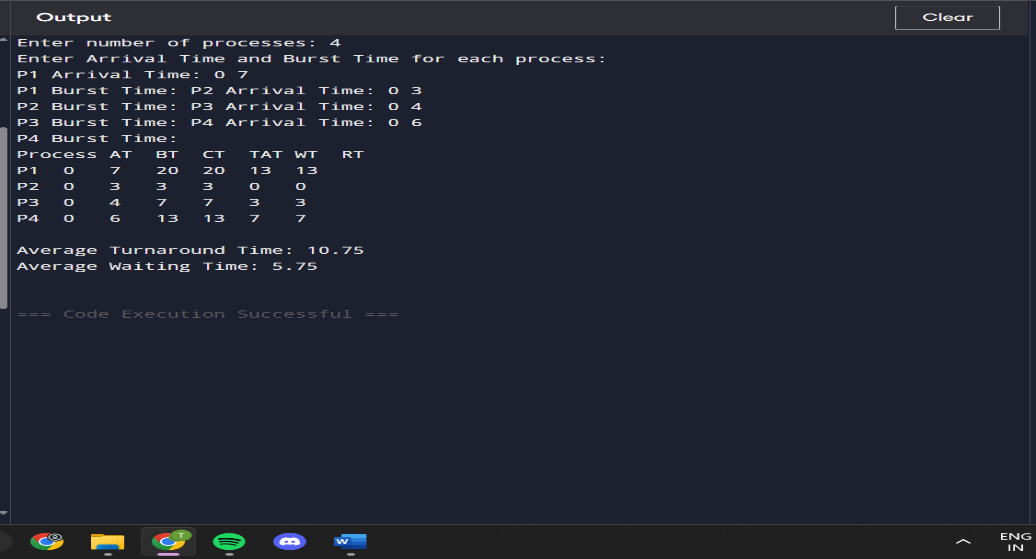
scanf("%d %d", &p[i].arrival, &p[i].burst);

}

printf("\nShortest Job First (Non-Preemptive) Scheduling\n"); sjf\_non\_preemptive(p, n); display(p, n); return 0;

}

Result:



.SJF(preemptive):

#include <stdio.h>

#include <limits.h>

typedef struct {

int id, arrival, burst, remaining, waiting, turnaround, completion,response,started; } Process;

void sortByArrival(Process p[], int n) { for (int i = 0; i < n - 1; i++) { for (int j = 0; j < n - i - 1; j++) { if (p[j].arrival > p[j + 1].arrival) { Process temp = p[j]; p[j]=p[j+1]; p[j+1]=temp;

}

}

}

}

void sjfPreemptive(Process p[], int n) { int completed = 0, time = 0, minIndex, minBurst;

while (completed < n) {

minIndex = -1, minBurst = INT\_MAX;

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

if (p[i].arrival <= time && p[i].remaining > 0) {

if (p[i].remaining < minBurst || (p[i].remaining == minBurst && p[i].arrival < p[minIndex].arrival)) {

minBurst = p[i].remaining; minIndex = i;

}

}

}

if (minIndex == -1) { time++; continue;

}

if (p[minIndex].started == 0) {

p[minIndex].response = time - p[minIndex].arrival; p[minIndex].started = 1;

}

p[minIndex].remaining--;

time++;

if (p[minIndex].remaining == 0) { p[minIndex].completion = time;

p[minIndex].turnaround = p[minIndex].completion - p[minIndex].arrival; p[minIndex].waiting = p[minIndex].turnaround - p[minIndex].burst; completed++;

}

}

}

void displayResults(Process p[], int n, const char \*title) { printf("\n--- %s ---\n", title);

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\tRT\n"); float totalWT = 0, totalTAT = 0, totalRT = 0; for (int i = 0; i < n; i++) {

printf("P%d\t%d\t%d\t%d\t%d\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].completion, p[i].turnaround, p[i].waiting, p[i].response); totalWT += p[i].waiting; totalTAT += p[i].turnaround; totalRT += p[i].response;

}

printf("Average Waiting Time: %.2f\n", totalWT / n); printf("Average Turnaround Time: %.2f\n", totalTAT / n); printf("Average Response Time: %.2f\n", totalRT / n);

}

int main() { int n, choice;

printf("Enter number of processes: "); scanf("%d", &n); Process p[n], temp[n];

printf("Enter Arrival Time and Burst Time:\n");

for (int i = 0; i < n; i++) { p[i].id = i + 1; scanf("%d %d", &p[i].arrival, &p[i].burst); p[i].remaining = p[i].burst;

p[i].waiting = p[i].turnaround = p[i].completion = p[i].response = p[i].started = 0;

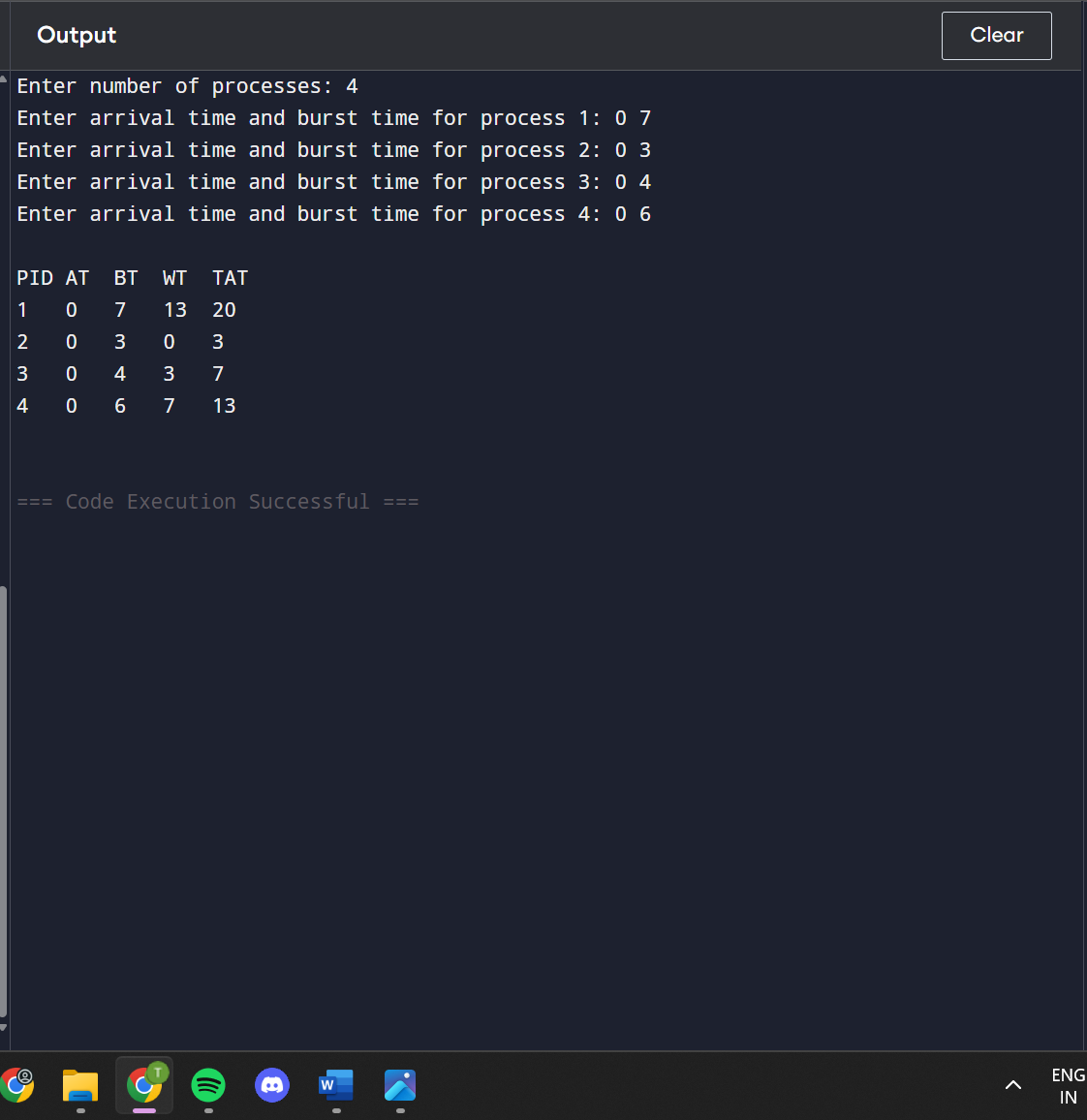
}

sjfPreemptive(p, n);

displayResults(p, n, "Shortest Job First (Preemptive)"); return 0;

}

Result:



Program - 2

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

#include <stdio.h>

#define MAX 10

typedef struct {

int pid, at, bt, pt, remaining\_bt, ct, tat, wt, rt, is\_completed, st;

} Process;

void nonPreemptivePriority(Process p[], int n) { int time = 0, completed = 0;

while (completed < n) {

int lowest\_priority = 9999, selected = -1; for (int i = 0; i < n; i++) {

if (p[i].at <= time && !p[i].is\_completed && p[i].pt < lowest\_priority) { lowest\_priority = p[i].pt; selected = i;

}

}

if (selected == -1) { time++; continue;

}

if (p[selected].rt == -1) { p[selected].st = time;

p[selected].rt = time - p[selected].at;

}

time += p[selected].bt; p[selected].ct = time;

p[selected].tat = p[selected].ct - p[selected].at; p[selected].wt = p[selected].tat - p[selected].bt; p[selected].is\_completed = 1; completed++;

}

}

void preemptivePriority(Process p[], int n) {

int time = 0, completed = 0; while (completed < n) {

int lowest\_priority = 9999, selected = -1;

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

if (p[i].at <= time && p[i].remaining\_bt > 0 && p[i].pt < lowest\_priority) { lowest\_priority = p[i].pt; selected = i;

}

}

if (selected == -1) { time++; continue;

}

if (p[selected].rt == -1) { p[selected].st = time;

p[selected].rt = time - p[selected].at;

}

p[selected].remaining\_bt--; time++;

if (p[selected].remaining\_bt == 0) { p[selected].ct = time;

p[selected].tat = p[selected].ct - p[selected].at; p[selected].wt = p[selected].tat - p[selected].bt; completed++;

}

}

}

void displayProcesses(Process p[], int n) { float avg\_tat = 0, avg\_wt = 0, avg\_rt = 0; printf("\nPID\tAT\tBT\tPriority\tCT\tTAT\tWT\tRT\n"); for (int i = 0; i < n; i++) {

printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\t%d\n",

p[i].pid, p[i].at, p[i].bt, p[i].pt, p[i].ct, p[i].tat, p[i].wt, p[i].rt); avg\_tat += p[i].tat; avg\_wt += p[i].wt; avg\_rt += p[i].rt;

}

printf("\nAverage TAT: %.2f", avg\_tat / n); printf("\nAverage WT: %.2f", avg\_wt / n); printf("\nAverage RT: %.2f\n", avg\_rt / n);

} int main() { Process p[MAX]; int n, choice;

printf("Enter the number of processes: "); scanf("%d", &n); for (int i = 0; i < n; i++) { p[i].pid = i + 1;

printf("\nEnter Arrival Time, Burst Time, and Priority for Process %d:\n", p[i].pid); printf("Arrival Time: "); scanf("%d", &p[i].at); printf("Burst Time: "); scanf("%d", &p[i].bt);

printf("Priority (lower number means higher priority): "); scanf("%d", &p[i].pt); p[i].remaining\_bt = p[i].bt; p[i].is\_completed = 0; p[i].rt = -1;

}

while (1) {

printf("\nPriority Scheduling Menu:\n"); printf("1. Non-Preemptive Priority Scheduling\n"); printf("2. Preemptive Priority Scheduling\n"); printf("3. Exit\n"); printf("Enter your choice: "); scanf("%d", &choice);

switch (choice) { case 1:

nonPreemptivePriority(p, n);

printf("Non-Preemptive Scheduling Completed!\n"); displayProcesses(p, n); break; case 2:

preemptivePriority(p, n);

printf("Preemptive Scheduling Completed!\n"); displayProcesses(p, n); break; case 3: printf("Exiting...\n"); return 0; default:

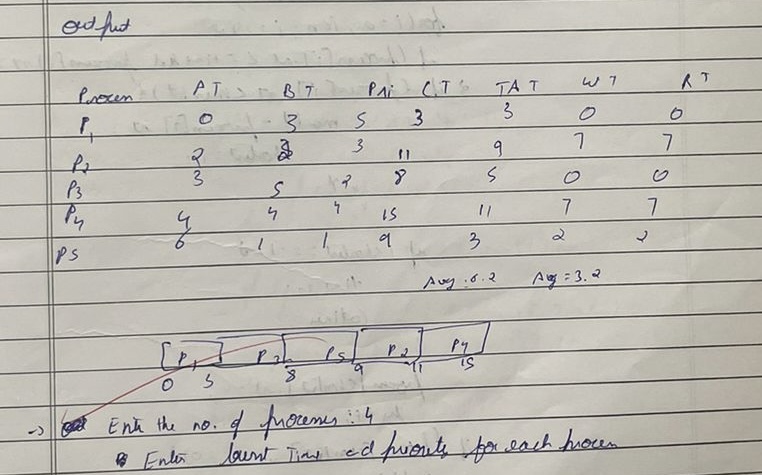
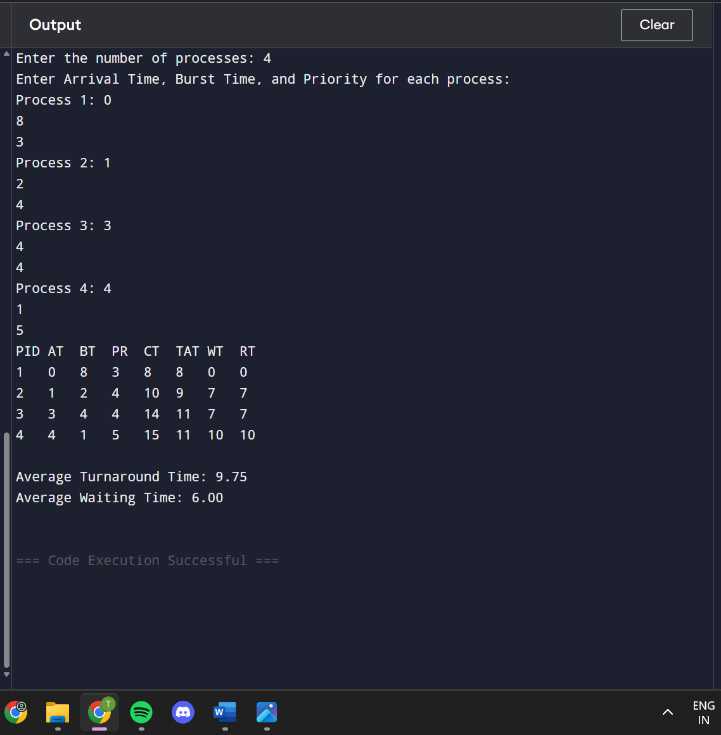
printf("Invalid choice! Try again.\n");

} } return 0;

}

Result:

r



Program - 3

Write a C program to simulate a 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.

#include <stdio.h>

#define MAX\_PROCESSES 10

#define TIME\_QUANTUM 2

typedef struct {

int burst\_time, arrival\_time, queue\_type, waiting\_time, turnaround\_time, response\_time, remaining\_time;

} Process;

void round\_robin(Process processes[], int n, int time\_quantum, int \*time) { int done, i; do { done = 1;

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

if (processes[i].remaining\_time > 0) { done = 0;

if (processes[i].remaining\_time > time\_quantum) {

\*time += time\_quantum;

processes[i].remaining\_time -= time\_quantum;

} else {

\*time += processes[i].remaining\_time;

processes[i].waiting\_time = \*time - processes[i].arrival\_time - processes[i].burst\_time; processes[i].turnaround\_time = \*time - processes[i].arrival\_time; processes[i].response\_time = processes[i].waiting\_time; processes[i].remaining\_time = 0;

}

}

}

} while (!done);

}

void fcfs(Process processes[], int n, int \*time) { for (int i = 0; i < n; i++) {

if (\*time < processes[i].arrival\_time) {

\*time = processes[i].arrival\_time;

}

processes[i].waiting\_time = \*time - processes[i].arrival\_time;

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

\*time += processes[i].burst\_time;

}

}

int main() {

Process processes[MAX\_PROCESSES], system\_queue[MAX\_PROCESSES], user\_queue[MAX\_PROCESSES];

int n, sys\_count = 0, user\_count = 0, time = 0;

float avg\_waiting = 0, avg\_turnaround = 0, avg\_response = 0, throughput; printf("Enter number of processes: "); scanf("%d", &n); for (int i = 0; i < n; i++) {

printf("Enter Burst Time, Arrival Time and Queue of P%d: ", i + 1); scanf("%d %d %d", &processes[i].burst\_time, &processes[i].arrival\_time,

&processes[i].queue\_type);

processes[i].remaining\_time = processes[i].burst\_time; if (processes[i].queue\_type == 1) { system\_queue[sys\_count++] = processes[i];

} else {

user\_queue[user\_count++] = processes[i];

}

}

for (int i = 0; i < user\_count - 1; i++) { for (int j = 0; j < user\_count - i - 1; j++) {

if (user\_queue[j].arrival\_time > user\_queue[j + 1].arrival\_time) { Process temp = user\_queue[j]; user\_queue[j] = user\_queue[j + 1]; user\_queue[j + 1] = temp;

}

}

}

printf("\nQueue 1 is System Process\nQueue 2 is User Process\n"); round\_robin(system\_queue, sys\_count, TIME\_QUANTUM, &time); fcfs(user\_queue, user\_count, &time);

printf("\nProcess Waiting Time Turn Around Time Response Time\n"); for (int i = 0; i < sys\_count; i++) {

avg\_waiting += system\_queue[i].waiting\_time; avg\_turnaround += system\_queue[i].turnaround\_time; avg\_response += system\_queue[i].response\_time;

printf("%d %d %d %d\n", i + 1, system\_queue[i].waiting\_time, system\_queue[i].turnaround\_time, system\_queue[i].response\_time);

}

for (int i = 0; i < user\_count; i++) { avg\_waiting += user\_queue[i].waiting\_time; avg\_turnaround += user\_queue[i].turnaround\_time; avg\_response += user\_queue[i].response\_time;

printf("%d %d %d %d\n", i + 1 + sys\_count, user\_queue[i].waiting\_time, user\_queue[i].turnaround\_time, user\_queue[i].response\_time);

}

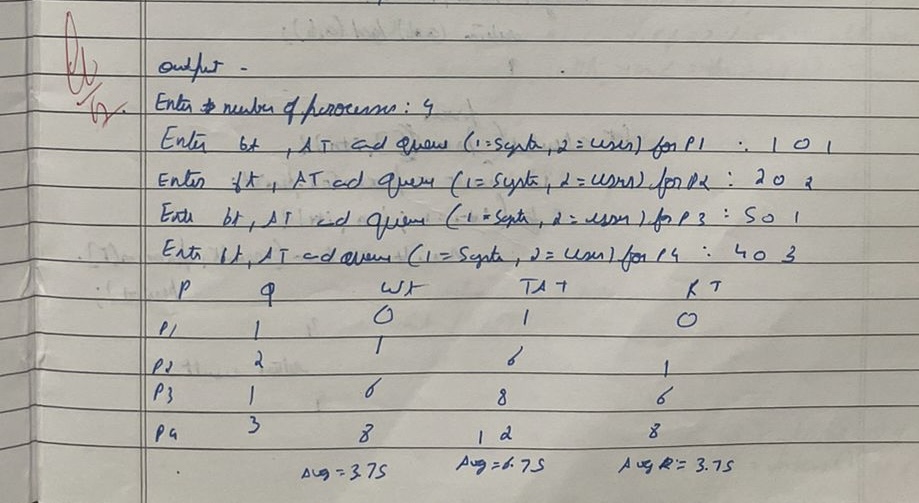
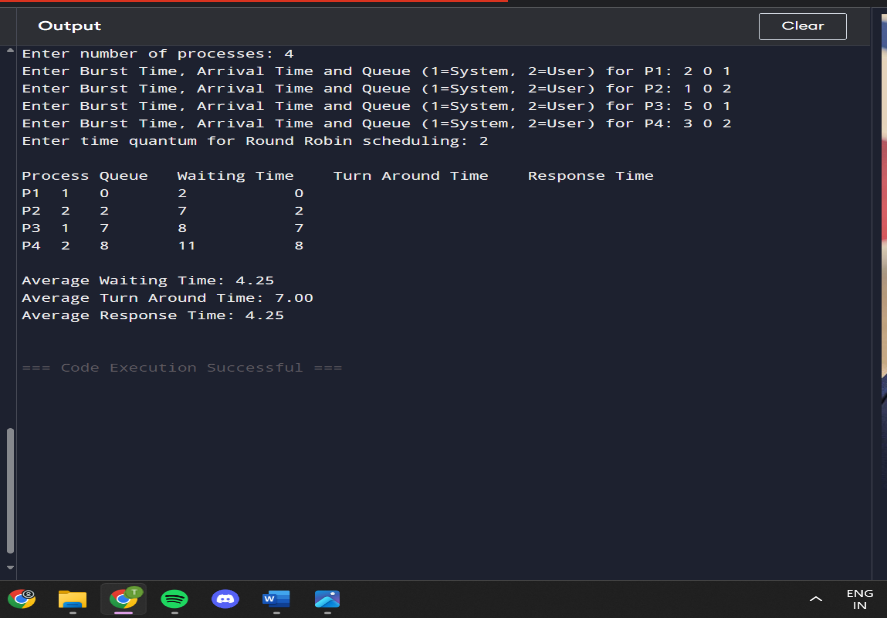
avg\_waiting /= n; avg\_turnaround /= n; avg\_response /= n; throughput = (float)n / time;

printf("\nAverage Waiting Time: %.2f", avg\_waiting); printf("\nAverage Turn Around Time: %.2f", avg\_turnaround); printf("\nAverage Response Time: %.2f", avg\_response); printf("\nThroughput: %.2f", throughput);

printf("\nProcess returned %d (0x%d) execution time: %.3f s\n", time, time, (float)time); return 0;

}

Result:



Program -4

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

1. Rate- Monotonic
2. Earliest-deadline First

. Rate Monotonic

#include <stdio.h>

#define MAX\_PROCESSES 10

typedef struct { int id; int burst\_time; int period; int remaining\_time; int next\_deadline;

} Process;

void sort\_by\_period(Process processes[], int n) { for (int i = 0; i < n - 1; i++) { for (int j = 0; j < n - i - 1; j++) {

if (processes[j].period > processes[j + 1].period) { Process temp = processes[j]; processes[j] = processes[j + 1]; processes[j + 1] = temp;

}

}

}

}

int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b);

}

int lcm(int a, int b) {

return (a \* b) / gcd(a, b);

}

int calculate\_lcm(Process processes[], int n) { int result = processes[0].period; for (int i = 1; i < n; i++) {

result = lcm(result, processes[i].period);

} return result;

}

double utilization\_factor(Process processes[], int n) { double sum = 0; for (int i = 0; i < n; i++) {

sum += (double)processes[i].burst\_time / processes[i].period;

}

return sum;

}

double rms\_threshold(int n) { return n \* (pow(2.0, 1.0 / n) - 1);

}

void rate\_monotonic\_scheduling(Process processes[], int n) { int lcm\_period = calculate\_lcm(processes, n); printf("LCM=%d\n\n", lcm\_period); printf("Rate Monotone Scheduling:\n"); printf("PID Burst Period\n"); for (int i = 0; i < n; i++) {

printf("%d %d %d\n", processes[i].id, processes[i].burst\_time, processes[i].period);

}

double utilization = utilization\_factor(processes, n);

double threshold = rms\_threshold(n);

printf("\n%.6f <= %.6f => %s\n", utilization, threshold, (utilization <= threshold) ? "true" :

"false");

if (utilization > threshold) {

printf("\nSystem may not be schedulable!\n"); return;

}

int timeline = 0, executed = 0; while (timeline < lcm\_period) { int selected = -1; for (int i = 0; i < n; i++) {

if (timeline % processes[i].period == 0) {

processes[i].remaining\_time = processes[i].burst\_time;

}

if (processes[i].remaining\_time > 0) { selected = i; break;

} }

if (selected != -1) {

printf("Time %d: Process %d is running\n", timeline, processes[selected].id); processes[selected].remaining\_time--; executed++; } else {

printf("Time %d: CPU is idle\n", timeline);

}

timeline++;

}

}

int main() {

int n;

Process processes[MAX\_PROCESSES]; printf("Enter the number of processes: "); scanf("%d", &n); printf("Enter the CPU burst times:\n"); for (int i = 0; i < n; i++) { processes[i].id = i + 1; scanf("%d", &processes[i].burst\_time); processes[i].remaining\_time = processes[i].burst\_time;

}

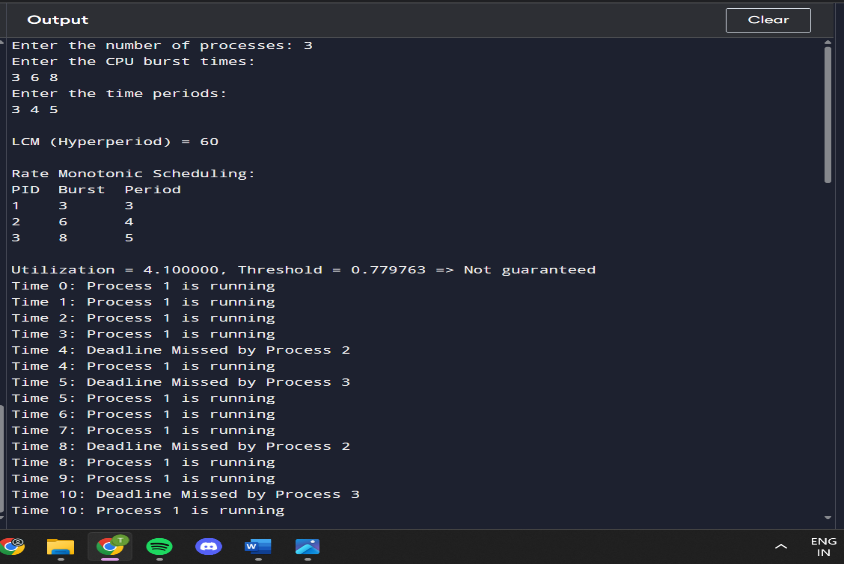
printf("Enter the time periods:\n"); for (int i = 0; i < n; i++) { scanf("%d", &processes[i].period);

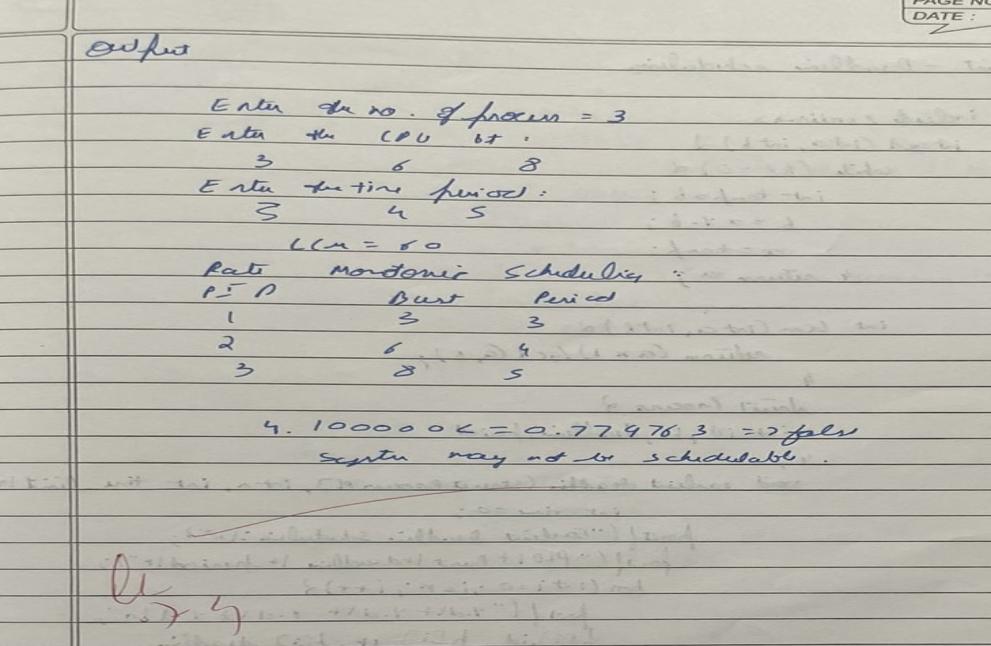
}

sort\_by\_period(processes, n); rate\_monotonic\_scheduling(processes, n); return 0;

}

Result :





. Earliest Deadline

#include <stdio.h> int gcd(int a, int b) { while (b != 0) { int temp = b; b = a % b; a = temp;

} return a;

}

int lcm(int a, int b) { return (a \* b) / gcd(a, b);

}

struct Process {

int id, burst\_time, deadline, period;

};

void earliest\_deadline\_first(struct Process p[], int n, int time\_limit) { int time = 0;

printf("Earliest Deadline Scheduling:\n"); printf("PID\tBurst\tDeadline\tPeriod\n"); for (int i = 0; i < n; i++) {

printf("%d\t%d\t\t%d\t\t%d\n", p[i].id, p[i].burst\_time, p[i].deadline, p[i].period);

}

printf("\nScheduling occurs for %d ms\n", time\_limit); while (time < time\_limit) { int earliest = -1; for (int i = 0; i < n; i++) { if (p[i].burst\_time > 0) {

if (earliest == -1 || p[i].deadline < p[earliest].deadline) { earliest = i;

}

}

}

if (earliest == -1) break;

printf("%dms: Task %d is running.\n", time, p[earliest].id); p[earliest].burst\_time--; time++;

}

}

int main() { int n;

printf("Enter the number of processes: "); scanf("%d", &n); struct Process processes[n]; printf("Enter the CPU burst times:\n"); for (int i = 0; i < n; i++) {

scanf("%d", &processes[i].burst\_time); processes[i].id = i + 1;

}

printf("Enter the deadlines:\n");

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

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

}

printf("Enter the time periods:\n"); for (int i = 0; i < n; i++) { scanf("%d", &processes[i].period);

}

int hyperperiod = processes[0].period; for (int i = 1; i < n; i++) {

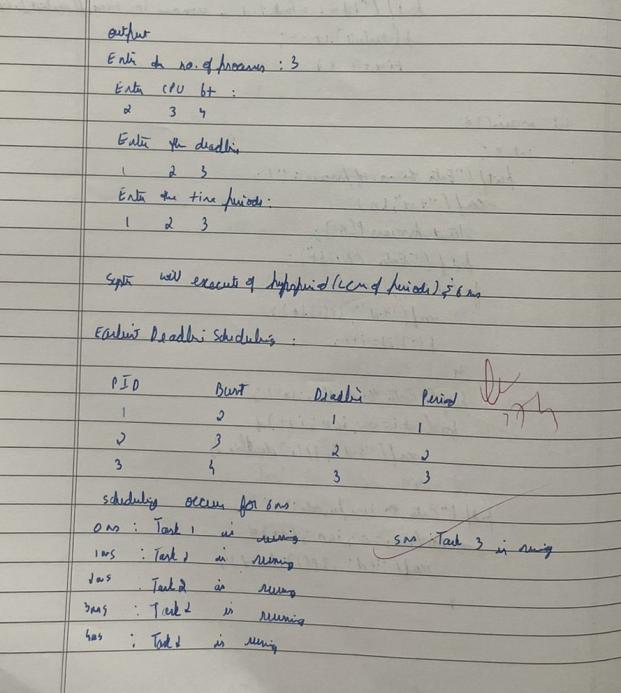
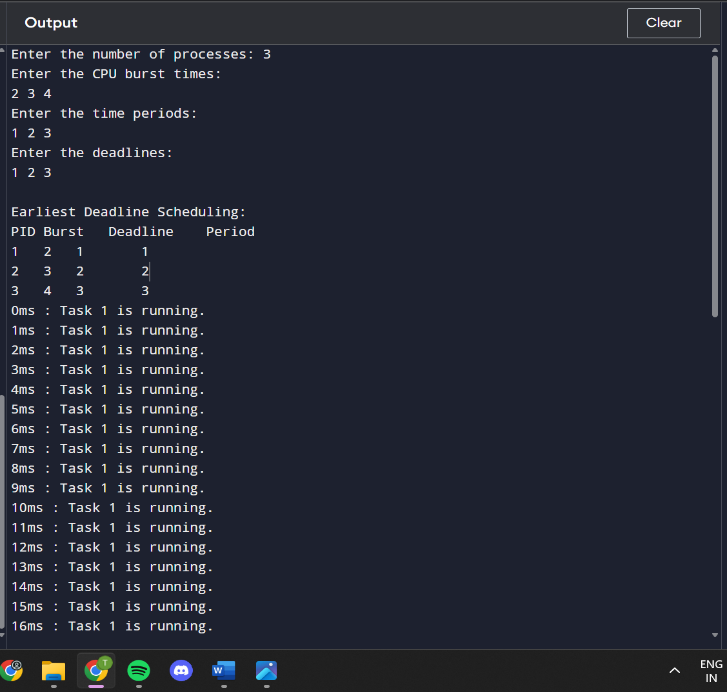
hyperperiod = lcm(hyperperiod, processes[i].period);

}

printf("\nSystem will execute for hyperperiod (LCM of periods): %d ms\n", hyperperiod); earliest\_deadline\_first(processes, n, hyperperiod); return 0;

}

Result:



Program 5

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

.Producer Consumer

#include <stdio.h>

int mutex = 1, full = 0, empty = 3, x = 0;

void wait(int \*s) {

--(\*s);

}

void signal(int \*s) {

++(\*s);

}

void producer() { wait(&empty); wait(&mutex); x++; printf("The item produced is %d\n", x); signal(&mutex); signal(&full);

}

void consumer() { wait(&full); wait(&mutex); printf("Consumed item %d\n", x); x--;

signal(&mutex); signal(&empty);

}

int main() { int choice; do { printf("\n1. Produce\n2. Consume\n3. Exit\nEnter choice: "); scanf("%d", &choice); switch (choice) {

case 1:

if ((mutex == 1) && (empty != 0)) { producer(); } else { printf("The buffer is full\n");

} break; case 2:

if ((mutex == 1) && (full != 0)) { consumer(); } else { printf("The buffer is empty\n");

} break; case 3: printf("Exiting.\n"); break; default:

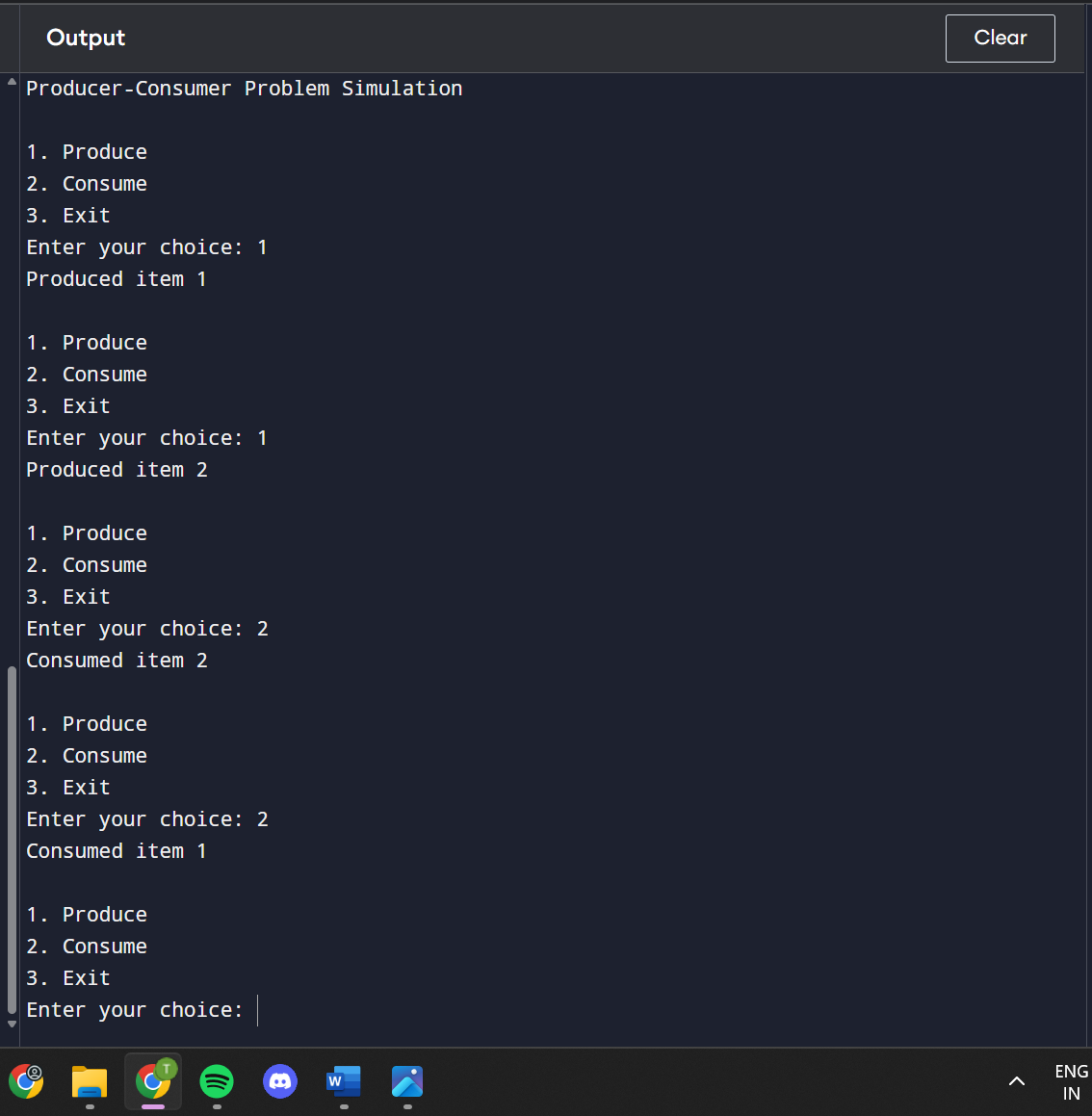
printf("Invalid choice.\n");

}

} while (choice != 3); return 0;

}

Result:



Program 6

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

#include <pthread.h>

#include <semaphore.h>

#include <stdio.h>

#include <unistd.h>

#define N 5

#define THINKING 2

#define HUNGRY 1

#define EATING 0

#define LEFT (phnum + 4) % N

#define RIGHT (phnum + 1) % N

int state[N];

int phil[N] = { 0, 1, 2, 3, 4 };

sem\_t mutex; sem\_t S[N];

void\* philosopher(void\* num); void take\_fork(int phnum); void put\_fork(int phnum); void test(int phnum);

int main() {

int i;

pthread\_t thread\_id[N];

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

for (i = 0; i < N; i++) sem\_init(&S[i], 0, 0);

for (i = 0; i < N; i++) { pthread\_create(&thread\_id[i], NULL, philosopher, &phil[i]); printf("Philosopher %d is thinking\n", i + 1);

}

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

pthread\_join(thread\_id[i], NULL);

return 0;

}

void test(int phnum) { if (state[phnum] == HUNGRY && state[LEFT] != EATING

&& state[RIGHT] != EATING) {

// state that eating state[phnum] = EATING; sleep(2); printf("Philosopher %d takes fork %d and %d\n", phnum + 1, LEFT + 1, phnum + 1); printf("Philosopher %d is Eating\n", phnum + 1); sem\_post(&S[phnum]);

}

}

take\_fork(int phnum) { sem\_wait(&mutex);

state[phnum] = HUNGRY; printf("Philosopher %d is Hungry\n", phnum + 1);

test(phnum); sem\_post(&mutex);

sem\_wait(&S[phnum]);

sleep(1);

}

put\_fork(int phnum) { sem\_wait(&mutex);

state[phnum] = THINKING;

printf("Philosopher %d putting fork %d and %d down\n", phnum + 1, LEFT + 1, phnum + 1); printf("Philosopher %d is thinking\n", phnum + 1);

test(LEFT); test(RIGHT);

sem\_post(&mutex);

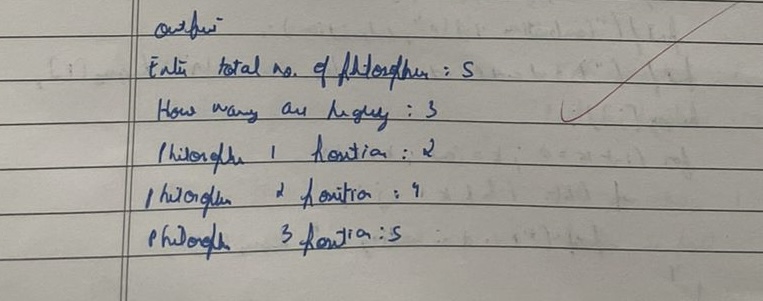
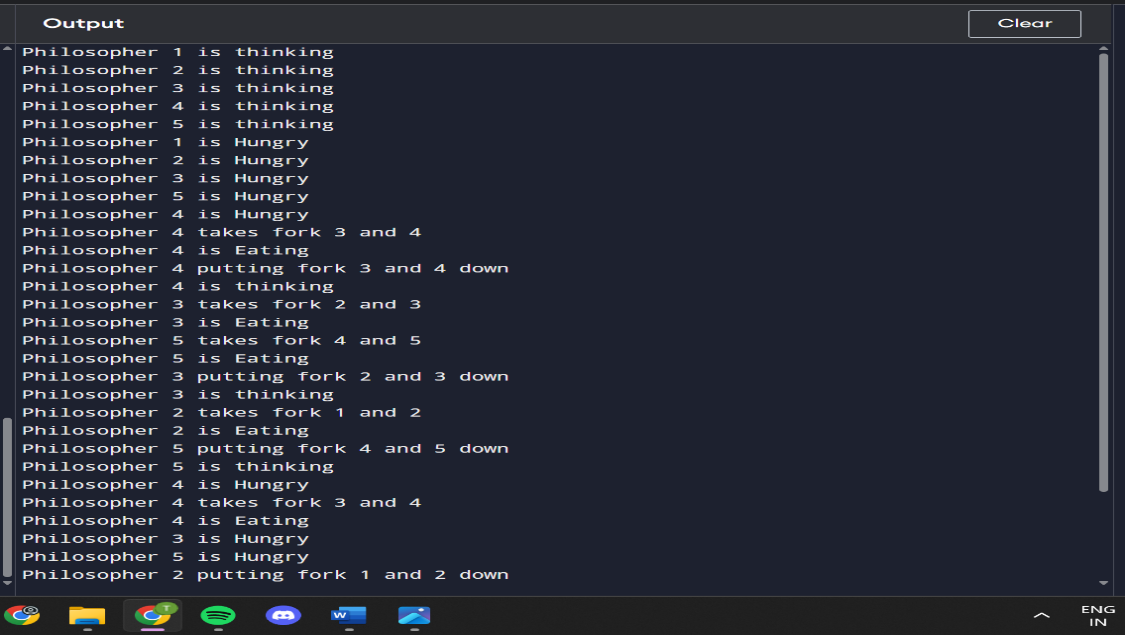
}

void\* philosopher(void\* num) { int\* i = (int\*)num; while (1) { sleep(1); take\_fork(\*i); sleep(0); put\_fork(\*i);

}

}

Result:



Program 7

Write a C program to simulate Banke’rs algorithm for the purpose of deadlock avoidance.

#include <stdio.h>

#include <stdbool.h>

int main() {

int n, m, i, j, k; printf("Enter number of processes: "); scanf("%d", &n); printf("Enter number of resources: "); scanf("%d", &m);

int alloc[n][m], max[n][m], avail[m]; int need[n][m];

printf("Enter allocation matrix (%d x %d):\n", n, m); for (i = 0; i < n; i++) { printf("Allocation for process %d: ", i); for (j = 0; j < m; j++) scanf("%d", &alloc[i][j]);

}

printf("Enter max matrix (%d x %d):\n", n, m); for (i = 0; i < n; i++) { printf("Max for process %d: ", i); for (j = 0; j < m; j++) scanf("%d", &max[i][j]);

}

printf("Enter available resources (%d values): ", m); for (i = 0; i < m; i++) scanf("%d", &avail[i]);

for (i = 0; i < n; i++) for (j = 0; j < m; j++) need[i][j] = max[i][j] - alloc[i][j];

bool finish[n]; int safeSeq[n]; int count = 0;

for (i = 0; i < n; i++) finish[i] = false;

while (count < n) { bool found = false; for (i = 0; i < n; i++) { if (!finish[i]) { for (j = 0; j < m; j++) if (need[i][j] > avail[j]) break;

if (j == m) { for (k = 0; k < m; k++) avail[k] += alloc[i][k];

safeSeq[count++] = i; finish[i] = true; found = true;

}

if (!found) { printf("System is not in safe state.\n"); return 1;

}

}

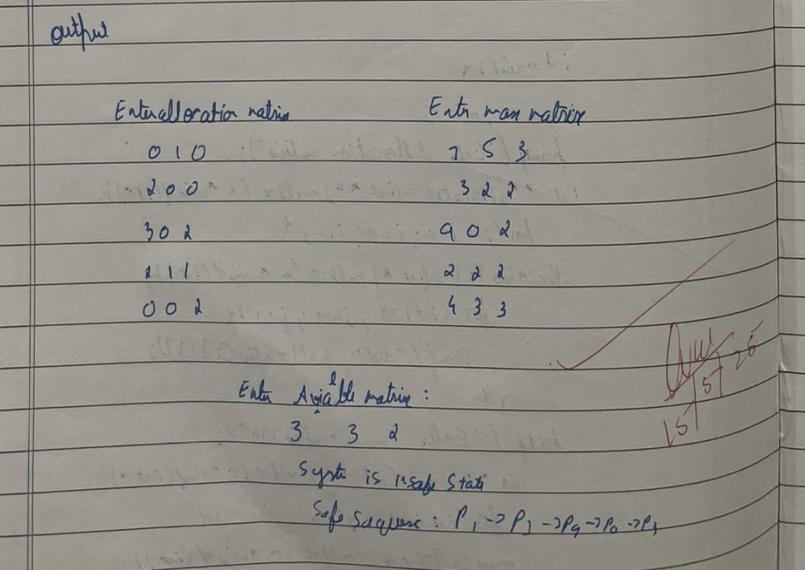
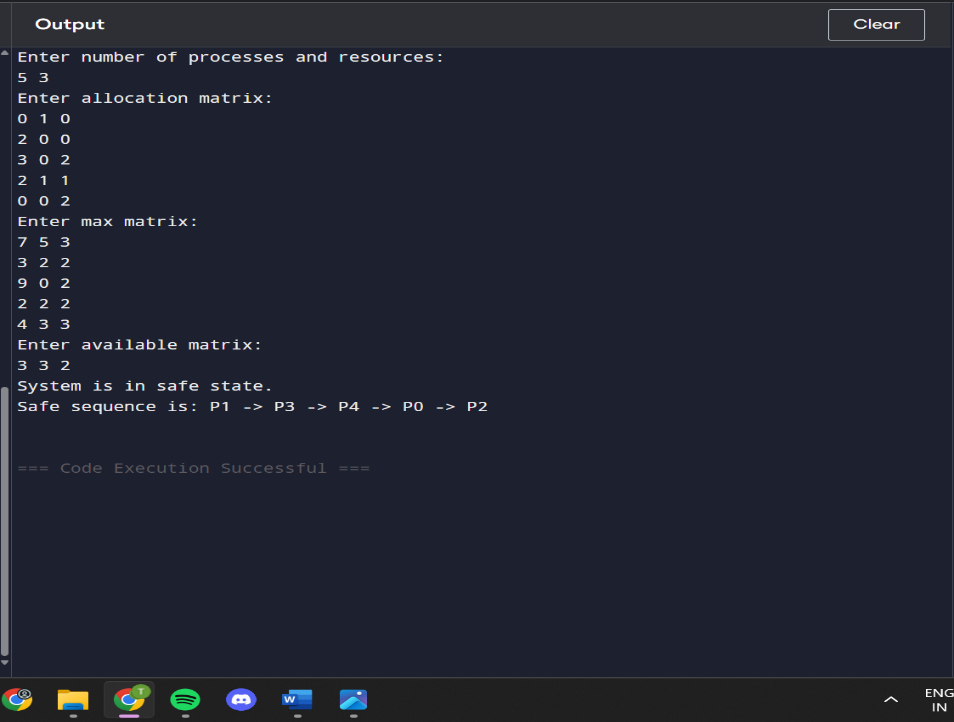
printf("System is in safe state.\n"); printf("Safe sequence is: "); for (i = 0; i < n; i++) { printf("P%d", safeSeq[i]); if (i != n - 1) printf(" -> ");

} printf("\n");

return 0;

}

Result:



Program 8

Write a C program to simulate deadlock detection

#include <stdio.h>

#include <stdbool.h>

int main() {

int n, m, i, j, k;

printf("Enter number of processes and resources:\n"); scanf("%d %d", &n, &m);

int allocation[n][m], request[n][m], available[m]; int work[m]; bool finish[n];

printf("Enter allocation matrix:\n"); for (i = 0; i < n; i++) for (j = 0; j < m; j++) scanf("%d", &allocation[i][j]);

printf("Enter request matrix:\n"); for (i = 0; i < n; i++) for (j = 0; j < m; j++) scanf("%d", &request[i][j]);

printf("Enter available matrix:\n"); for (i = 0; i < m; i++) { scanf("%d", &available[i]); work[i] = available[i];

}

for (i = 0; i < n; i++) { bool zero\_allocation = true; for (j = 0; j < m; j++) { if (allocation[i][j] != 0) { zero\_allocation = false; break;

}

}

finish[i] = zero\_allocation;

}

bool found\_process;

do { found\_process = false; for (i = 0; i < n; i++) { if (!finish[i]) { bool can\_allocate = true; for (j = 0; j < m; j++) { if (request[i][j] > work[j]) { can\_allocate = false; break;

} } if (can\_allocate) { for (k = 0; k < m; k++) work[k] += allocation[i][k]; finish[i] = true; printf("Process %d can finish.\n", i); found\_process = true;

}

}

}

} while (found\_process);

bool deadlock = false; for (i = 0; i < n; i++) { if (!finish[i]) { deadlock = true; break;

}

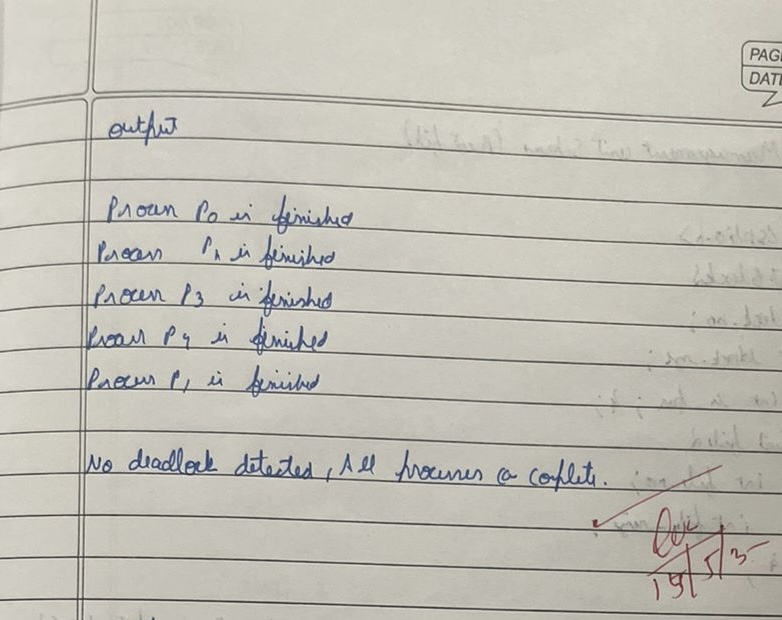
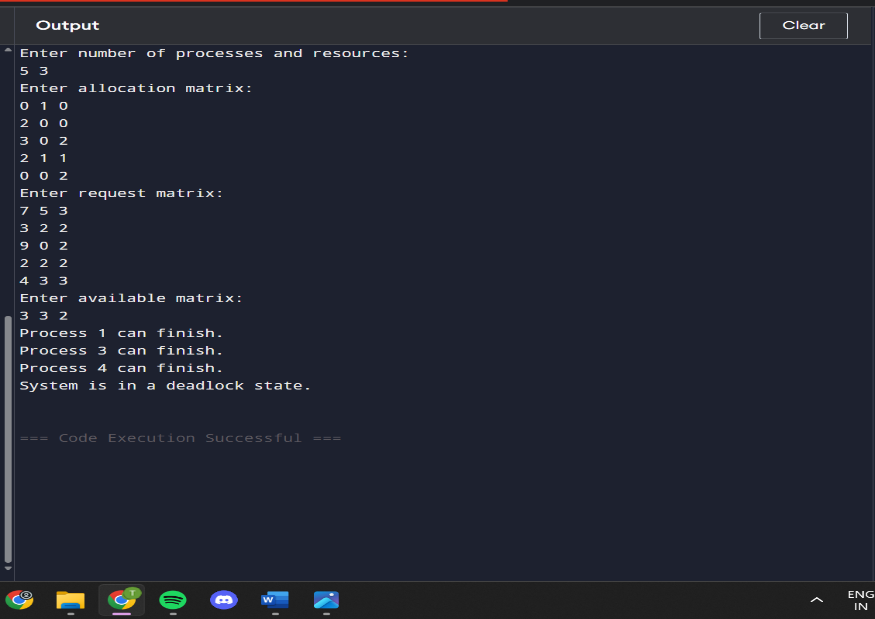
}

if (deadlock) printf("System is in a deadlock state.\n"); else printf("System is not in a deadlock state.\n");

return 0;

}

Result:



Program 9:

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

a) Worst-fit

b)Best-fit

c)First-fit

#include <stdio.h>

struct Block { int size; int allocated;

};

struct File { int size; int block\_no;

};

void resetBlocks(struct Block blocks[], int n) { for (int i = 0; i < n; i++) { blocks[i].allocated = 0;

}

}

void firstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) { printf("\n\tMemory Management Scheme – First Fit\n"); printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n"); for (int i = 0; i < n\_files; i++) { files[i].block\_no = -1; for (int j = 0; j < n\_blocks; j++) { if (!blocks[j].allocated && blocks[j].size >= files[i].size) { files[i].block\_no = j + 1; blocks[j].allocated = 1; printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, j + 1, blocks[j].size); break;

}

}

if (files[i].block\_no == -1) { printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

void bestFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) { printf("\n\tMemory Management Scheme – Best Fit\n"); printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n"); for (int i = 0; i < n\_files; i++) { int bestIdx = -1; for (int j = 0; j < n\_blocks; j++) { if (!blocks[j].allocated && blocks[j].size >= files[i].size) { if (bestIdx == -1 || blocks[j].size < blocks[bestIdx].size) { bestIdx = j;

}

}

} if (bestIdx != -1) { blocks[bestIdx].allocated = 1; files[i].block\_no = bestIdx + 1; printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, bestIdx + 1, blocks[bestIdx].size);

} else { printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size); }

}

}

void worstFit(struct Block blocks[], int n\_blocks, struct File files[], int n\_files) { printf("\n\tMemory Management Scheme – Worst Fit\n"); printf("File\_no:\tFile\_size\tBlock\_no:\tBlock\_size:\n"); for (int i = 0; i < n\_files; i++) { int worstIdx = -1; for (int j = 0; j < n\_blocks; j++) { if (!blocks[j].allocated && blocks[j].size >= files[i].size) { if (worstIdx == -1 || blocks[j].size > blocks[worstIdx].size) { worstIdx = j;

}

}

} if (worstIdx != -1) { blocks[worstIdx].allocated = 1; files[i].block\_no = worstIdx + 1; printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, files[i].size, worstIdx + 1, blocks[worstIdx].size);

} else { printf("%d\t\t%d\t\t\_\t\t\_\n", i + 1, files[i].size);

}

}

}

int main() { int n\_blocks, n\_files, choice; printf("Memory Management Scheme\n"); printf("Enter the number of blocks: "); scanf("%d", &n\_blocks); printf("Enter the number of files: "); scanf("%d", &n\_files); struct Block blocks[n\_blocks]; struct File files[n\_files]; printf("\nEnter the size of the blocks:\n"); for (int i = 0; i < n\_blocks; i++) { printf("Block %d: ", i + 1); scanf("%d", &blocks[i].size); blocks[i].allocated = 0;

} printf("Enter the size of the files:\n"); for (int i = 0; i < n\_files; i++) { printf("File %d: ", i + 1); scanf("%d", &files[i].size);

} do { printf("\n1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n"); printf("Enter your choice: "); scanf("%d", &choice); resetBlocks(blocks, n\_blocks); // Reset block allocation before each strategy switch (choice) { case 1: firstFit(blocks, n\_blocks, files, n\_files); break; case 2: bestFit(blocks, n\_blocks, files, n\_files); break; case 3:

worstFit(blocks, n\_blocks, files, n\_files); break; case 4: printf("\nExiting...\n"); break; default:

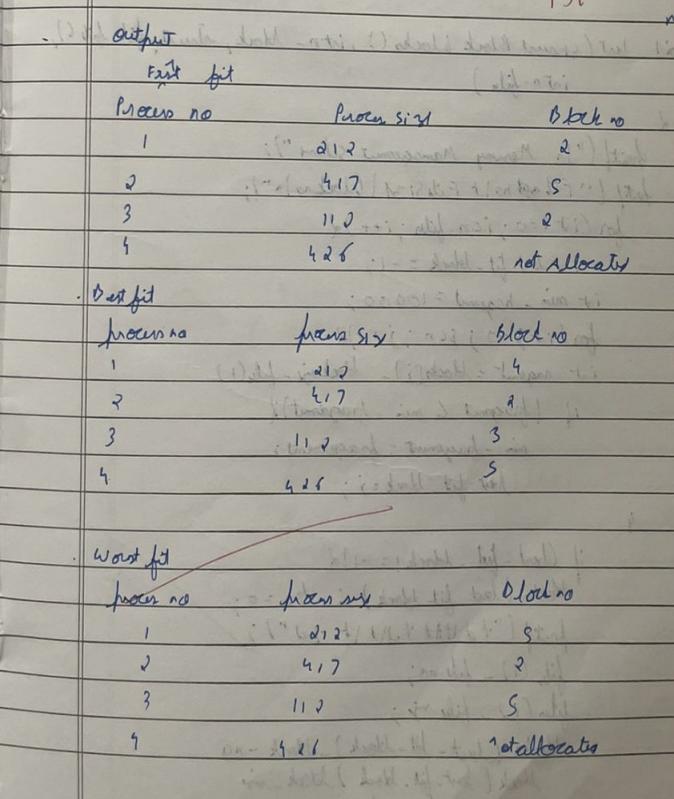
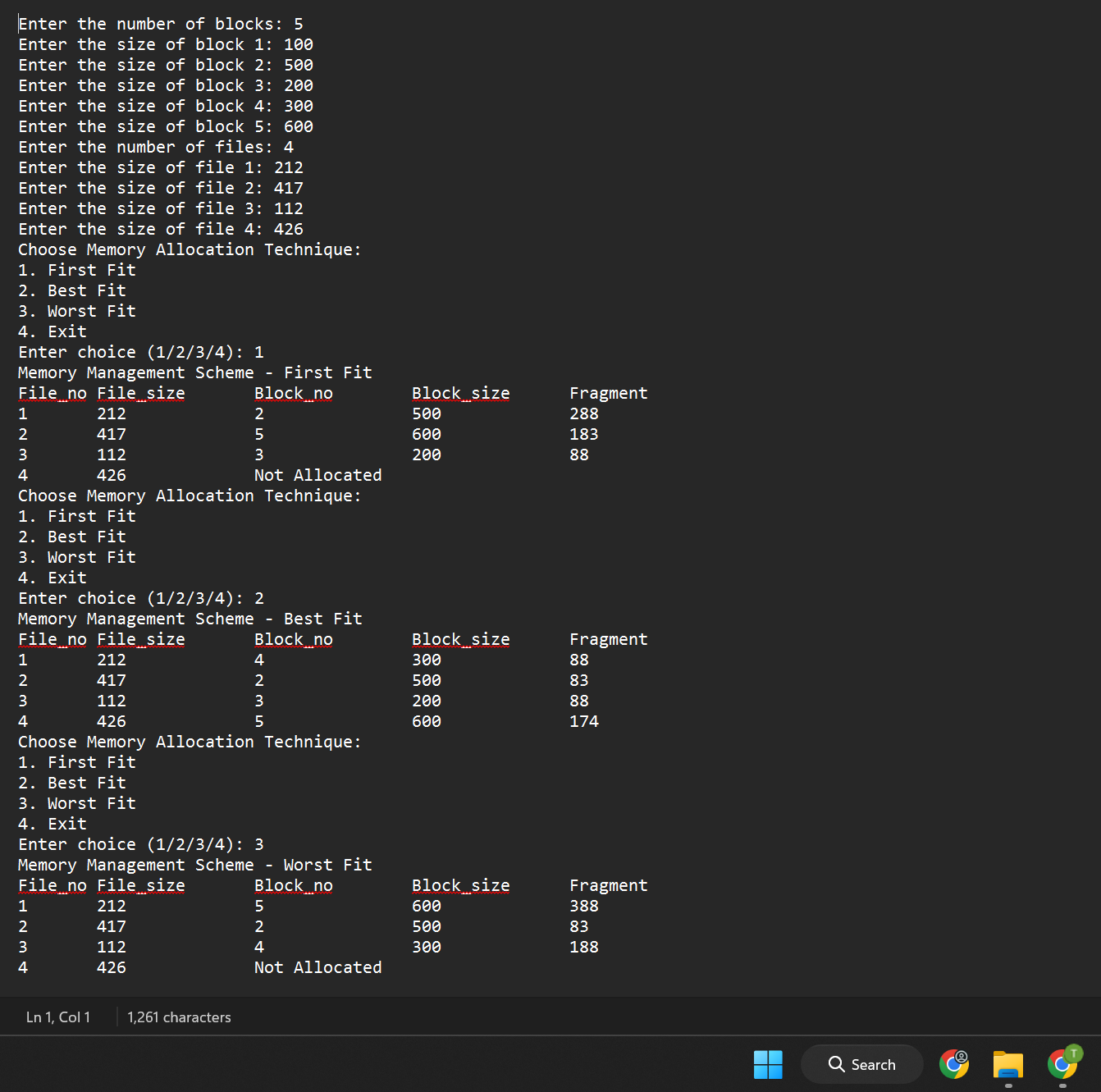
printf("Invalid choice.\n");

}

} while (choice != 4); return 0;

}

Result:



Program 10

Write a C program to simulate page replacement algorithms a) FIFO

1. LRU
2. Optimal

#include<stdio.h> int n,m; int p[10000],f[10000]; void fifo(){ for(int i=0;i<m;++i) f[i]=-1; int h=0; int s=0; for(int i=0;i<n;++i){ int k=0; for(int r=0;r<m;++r) if(p[i]==f[r]){++h;k=1;break;} if(k==1)continue; for(int r=0;r<m;++r) if(f[r]==-1){ f[r]=p[i];k=2;break;} if(k==2) continue; f[s]=p[i]; s=(s+1)%m;

}

printf("FIFO :\t\t\t Page-Hit:%d\t\t\tPage-Fault:%d\n",h,n-h);

} void lru(){ for(int i=0;i<m;++i) f[i]=-1; int h=0; int t[m]; for(int i=0;i<m;++i) t[i]=0; for(int i=0;i<n;++i){ int k=0; for(int r=0;r<m;++r) if(f[r]!=-1) ++t[r]; for(int r=0;r<m;++r) if(f[r]==p[i]){t[r]=1;k=1;break;} if(k==1){++h;continue;} for(int r=0;r<m;++r) if(f[r]==-1){f[r]=p[i];t[r]=1; k=2;break;} if(k==2) continue; int x=0; for(int r=1;r<m;++r) if(t[r]>t[x]) x=r; t[x]=1;f[x]=p[i];

}

printf("LRU :\t\t\t Page-Hit:%d\t\t\tPage-Fault:%d\n",h,n-h);

} void optimal(){ for(int i=0;i<m;++i) f[i]=-1; int h=0; for(int i=0;i<n;++i){ int k=0; for(int r=0;r<m;++r) if(f[r]==p[i]){k=1; ++h;break;} if(k==1) continue; for(int r=0;r<m;++r) if(f[r]==-1){ k=2;f[r]=p[i];break;} if(k==2) continue; int l=-1,xx=-1,y=1; for(int j=0;j<m;++j){ y=1; for(k=i+1;k<n;++k) if(f[j]==p[k]){ y=0; if(k>l){xx=j;l=k;}break;

} if(y){

f[j]=p[i];break;}

}

if(y) continue; f[xx]=p[i];

}

printf("OPTIMAL :\t\t Page-Hit:%d\t\t\tPage-Fault:%d\n",h,n-h);

} int main(){ printf("Enter No. of Page Strign and Page-Frame : "); scanf("%d%d",&n,&m);

printf("Enter the Page-String : "); for(int i=0;i<n;++i) scanf("%d",&p[i]); fifo();lru();optimal();

}

RESULT:

