# 2. Process Creation and Termination

1. WAP in C to demonstrate the process creation and termination in Linux.

**Theory:**

In Linux, process creation is done using the fork() system call, which creates a new process (child) that is a duplicate of the calling process (parent). The child process gets a unique process ID and runs independently. To execute a different program within the child process, exec() family functions are used, replacing the child’s memory with the new program.

**Code:**

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main() {

pid\_t pid, mypid, myppid;

pid = getpid();

printf("Before fork: Process id is %d\n", pid);

pid = fork();

if (pid < 0) {

perror("fork() failure\n");

return 1;

}

if (pid == 0) {

printf("This is child process\n");

mypid = getpid();

myppid = getppid();

printf("Process id is %d and PPID is %d\n", mypid, myppid);

}

else {

sleep(2);

printf ("This is parent process\n");

mypid = getpid();

myppid = getppid();

printf("Process id is %d and PPID is %d\n", mypid, myppid);

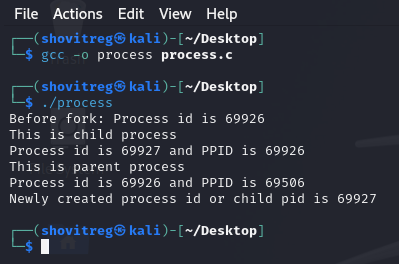
printf("Newly created process id or child pid is %d\n", pid);

}

return 0;

}

**Output:**

****

**3. Thread Creation and Termination**

## WAP in C to demonstrate the thread creation and termination in Linux.

**Theory:**

In Linux, threads are created using the POSIX pthread library. To create a thread, use pthread\_create(), which takes a thread identifier, thread attributes, the function to be executed by the thread, and arguments for that function. Threads share the same memory space and resources of the process.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

void\* threadFunction(void\* arg) {

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

printf("Thread %d: Started\n", id);

sleep(2);

printf("Thread %d: Finished\n", id);

pthread\_exit(NULL); }

int main() {

pthread\_t thread1, thread2;

int id1 = 1, id2 = 2;

if (pthread\_create(&thread1, NULL, threadFunction, &id1) != 0) {

fprintf(stderr, "Error creating thread 1\n");

return 1;

}

if (pthread\_create(&thread2, NULL, threadFunction, &id2) != 0) {

fprintf(stderr, "Error creating thread 2\n");

return 1;

}

if (pthread\_join(thread1, NULL) != 0) {

fprintf(stderr, "Error joining thread 1\n");

return 1;

}

if (pthread\_join(thread2, NULL) != 0) {

fprintf(stderr, "Error joining thread 2\n");

return 1;

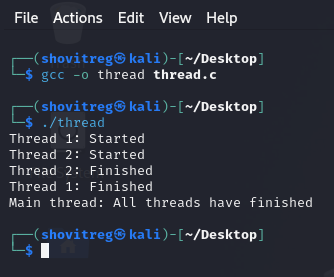
}

printf("Main thread: All threads have finished\n");

return 0;

}

**Output:**

****

# 4. Simulation of IPC Techniques

## WAP to simulate the solution of Producer-Consumer problem using semaphore.

**Theory:**

The producer consumer problem is a synchronization problem. There is a fixed size buffer and the producer produces items and enters them into the buffer. The consumer removes the items from the buffer and consumes them. A producer should not produce items into the buffer when the consumer is consuming an item from the buffer and vice versa. So the buffer should only be accessed by the producer or consumer at a time.

**Code:**

#include <stdio.h>

#include <stdlib.h>

int mutex = 1;n

int full = 0;

int empty = 10, x = 0;

void producer() {

--mutex;

++full;

--empty;

x++;

printf("\nProducer produces item %d", x);

++mutex;

}

void consumer() {

--mutex;

--full;

++empty;

printf("\nConsumer consumes item %d", x);

x--;

++mutex;

}

int main() {

int n, i;

printf("\n1. Press 1 for Producer\n2. Press 2 for Consumer\n3. Press 3 for Exit");

for (i = 1; i > 0; i++) {

printf("\nEnter your choice:");

scanf("%d", &n);

switch (n) {

case 1:

if ((mutex == 1) && (empty != 0)) {

producer();

}

else {

printf("Buffer is full!");

}

break;

case 2:

if ((mutex == 1) && (full != 0)) {

consumer();

}

else {

printf("Buffer is empty!");

}

break;

case 3:

exit(0);

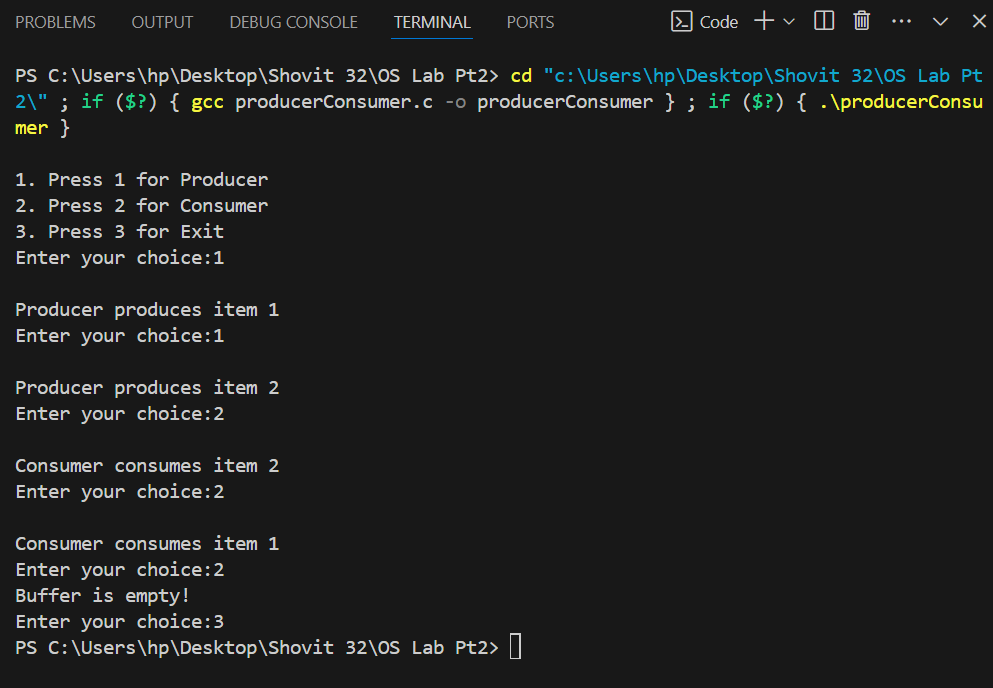
break;

}

}

}

**Output:**

****

## WAP to simulate the solution of dining philosopher's problem using semaphore.

**Theory:**

The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

**Code:**

#include <stdio.h>

#include <windows.h>

#define NUM\_PHILOSOPHERS 5

#define THINKING 0

#define HUNGRY 1

#define EATING 2

#define LEFT (philosopher\_id + NUM\_PHILOSOPHERS - 1) % NUM\_PHILOSOPHERS

#define RIGHT (philosopher\_id + 1) % NUM\_PHILOSOPHERS

HANDLE mutex;

HANDLE semaphores[NUM\_PHILOSOPHERS];

int state[NUM\_PHILOSOPHERS];

void test(int philosopher\_id) {

if (state[philosopher\_id] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {

state[philosopher\_id] = EATING;

printf("Philosopher %d is EATING\n", philosopher\_id + 1);

ReleaseSemaphore(semaphores[philosopher\_id], 1, NULL);

}

}

void take\_forks(int philosopher\_id) {

WaitForSingleObject(mutex, INFINITE);

state[philosopher\_id] = HUNGRY;

printf("Philosopher %d is HUNGRY\n", philosopher\_id + 1);

test(philosopher\_id);

ReleaseMutex(mutex);

WaitForSingleObject(semaphores[philosopher\_id], INFINITE);

}

void put\_forks(int philosopher\_id) {

WaitForSingleObject(mutex, INFINITE);

state[philosopher\_id] = THINKING;

printf("Philosopher %d is THINKING\n", philosopher\_id + 1);

test(LEFT);

test(RIGHT);

ReleaseMutex(mutex);

}

DWORD WINAPI philosopher(LPVOID num) {

int philosopher\_id = \*(int\*)num;

while (1) {

Sleep(1000);

take\_forks(philosopher\_id);

Sleep(1000);

put\_forks(philosopher\_id);

}

return 0;

}

int main() {

HANDLE threads[NUM\_PHILOSOPHERS];

int philosopher\_ids[NUM\_PHILOSOPHERS];

mutex = CreateMutex(NULL, FALSE, NULL);

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

semaphores[i] = CreateSemaphore(NULL, 0, 1, NULL);

philosopher\_ids[i] = i;

}

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

threads[i] = CreateThread(NULL, 0, philosopher, &philosopher\_ids[i], 0, NULL);

}

WaitForMultipleObjects(NUM\_PHILOSOPHERS, threads, TRUE, INFINITE);

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

CloseHandle(threads[i]);

CloseHandle(semaphores[i]);

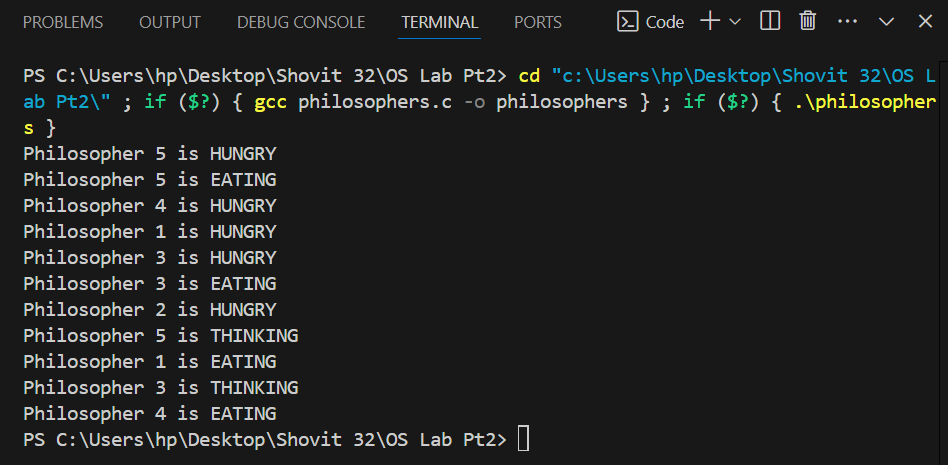
}

CloseHandle(mutex);

return 0;

}

**Output:**

****

## WAP to simulate the solution of producer consumer problem using message passing technique.

**Theory:**

The producer-consumer problem is a classic synchronization problem in operating systems. It involves two types of processes: producers, which generate data and place it into a shared resource (such as a buffer), and consumers, which take data from the shared resource for processing. The challenge is to ensure that producers don't produce data when the buffer is full, and consumers don't consume data when the buffer is empty, while avoiding race conditions and deadlocks. Using message passing, the producer-consumer problem can be implemented without the need for shared memory, relying instead on communication channels like message queues.

**Code:**

#include <stdio.h>

#include <windows.h>

#include <stdlib.h>

#include <string.h>

#define PIPE\_NAME "\\\\.\\pipe\\ProducerConsumerPipe"

#define BUFFER\_SIZE 1024

#define STOP\_MESSAGE "exit"

DWORD WINAPI producer(LPVOID lpParam);

DWORD WINAPI consumer(LPVOID lpParam);

int main() {

HANDLE hProducer, hConsumer;

hProducer = CreateThread(NULL, 0, producer, NULL, 0, NULL);

if (hProducer == NULL) {

printf("Error creating producer thread: %d\n", GetLastError());

return 1;

}

hConsumer = CreateThread(NULL, 0, consumer, NULL, 0, NULL);

if (hConsumer == NULL) {

printf("Error creating consumer thread: %d\n", GetLastError());

return 1;

}

WaitForSingleObject(hProducer, INFINITE);

WaitForSingleObject(hConsumer, INFINITE);

CloseHandle(hProducer);

CloseHandle(hConsumer);

return 0;

}

DWORD WINAPI producer(LPVOID lpParam) {

HANDLE hPipe;

char buffer[BUFFER\_SIZE];

DWORD bytesWritten;

hPipe = CreateNamedPipe(

PIPE\_NAME,

PIPE\_ACCESS\_OUTBOUND,

PIPE\_TYPE\_MESSAGE | PIPE\_WAIT,

1,

BUFFER\_SIZE,

BUFFER\_SIZE,

0,

NULL);

if (hPipe == INVALID\_HANDLE\_VALUE) {

printf("Error creating named pipe: %d\n", GetLastError());

return 1;

}

printf("Producer: Waiting for consumer to connect...\n");

ConnectNamedPipe(hPipe, NULL);

printf("Producer: Consumer connected.\n");

for (int i = 1; i <= 10; i++) {

snprintf(buffer, BUFFER\_SIZE, "Message %d", i);

if (!WriteFile(hPipe, buffer, strlen(buffer) + 1, &bytesWritten, NULL)) {

printf("Producer: Error writing to pipe: %d\n", GetLastError());

break;

}

printf("Producer: Sent '%s'\n", buffer);

Sleep(1000); // Simulate production time

}

strcpy(buffer, STOP\_MESSAGE);

WriteFile(hPipe, buffer, strlen(buffer) + 1, &bytesWritten, NULL);

printf("Producer: Sent stop message.\n");

CloseHandle(hPipe);

printf("Producer finished.\n");

return 0;

}

DWORD WINAPI consumer(LPVOID lpParam) {

HANDLE hPipe;

char buffer[BUFFER\_SIZE];

DWORD bytesRead;

hPipe = CreateFile(

PIPE\_NAME,

GENERIC\_READ,

0,

NULL,

OPEN\_EXISTING,

0,

NULL);

if (hPipe == INVALID\_HANDLE\_VALUE) {

printf("Consumer: Error opening named pipe: %d\n", GetLastError());

return 1;

}

printf("Consumer: Connected to the producer.\n");

while (1) {

if (ReadFile(hPipe, buffer, BUFFER\_SIZE, &bytesRead, NULL) && bytesRead > 0) {

buffer[bytesRead] = '\0';

printf("Consumer: Received '%s'\n", buffer);

if (strcmp(buffer, STOP\_MESSAGE) == 0) {

printf("Consumer: Stop message received. Exiting.\n");

break;

}

}

else {

printf("Consumer: Error reading from pipe: %d\n", GetLastError());

break;

}

}

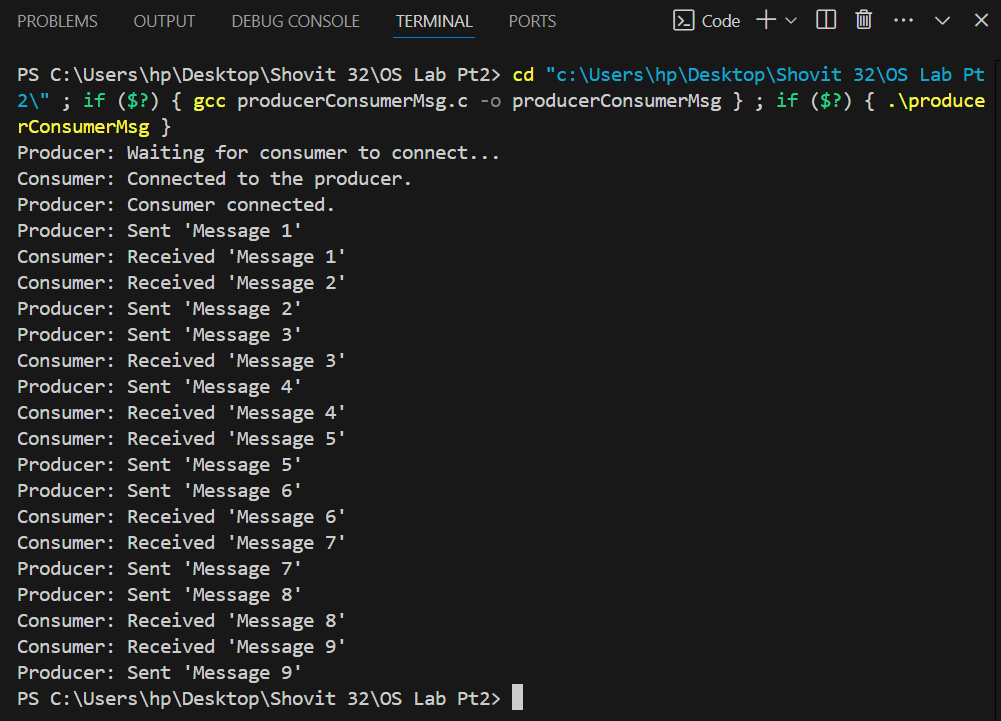
CloseHandle(hPipe);

printf("Consumer finished.\n");

return 0;

}

**Output:**

****

# 5. Simulation of Process Scheduling Algorithms

## WAP to simulate FCFS CPU Scheduling Algorithm

**Theory:**

FCFS is considered as simplest CPU-scheduling algorithm. In FCFS algorithm, the process that requests the CPU first is allocated in the CPU first. The implementation of FCFS algorithm is managed with FIFO (First in first out) queue. FCFS scheduling is non-preemptive. Nonpreemptive means, once the CPU has been allocated to a process, that process keeps the CPU until it executes a work or job or task and releases the CPU, either by requesting I/O.

**Code:**

#include<stdio.h>

int i;

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

wt[0] = 0;

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

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

}

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

{

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

findWaitingTime(processes, n, bt, wt);

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

printf("Processes Burst time Waiting time Turn around time\n");

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

{

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

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

printf(" %d ", (i+1));

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

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

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

}

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

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

printf("Average waiting time = %f", s);

printf("\n");

printf("Average turn around time = %f ", t);

}

int main()

{

int processes[] = {1, 2, 3};

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

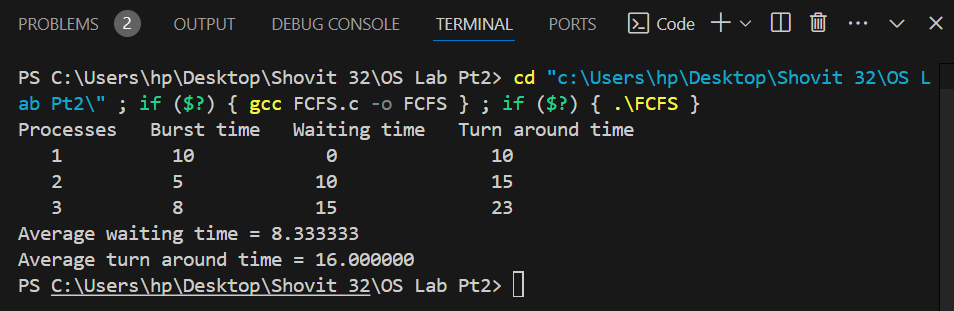
int burst\_time[] = {10, 5, 8};

findavgTime(processes, n, burst\_time);

return 0;

}

**Output:**



## WAP to simulate SJF CPU Scheduling Algorithm

**Theory:**

SJF (Shortest Job First) is a scheduling strategy that gives the process with the quickest CPU burst time to the CPU first. As this technique is non-preemptive, once a process has begun to run, it cannot be stopped until it has finished. The SJF scheduling method is ideal since it reduces the average waiting time for a set of processes.

**Code:**

#include <stdio.h>

int main()

{

int A[100][4];

int i, j, n, total = 0, index, temp;

float avg\_wt, avg\_tat;

printf("Enter number of process: ");

scanf("%d", &n);

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

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

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

scanf("%d", &A[i][1]);

A[i][0] = i + 1;

}

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

index = i;

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

if (A[j][1] < A[index][1])

index = j;

temp = A[i][1];

A[i][1] = A[index][1];

A[index][1] = temp;

temp = A[i][0];

A[i][0] = A[index][0];

A[index][0] = temp;

}

A[0][2] = 0;

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

A[i][2] = 0;

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

A[i][2] += A[j][1];

total += A[i][2];

}

avg\_wt = (float)total / n;

total = 0;

printf("P\tBT\tWT\tTAT\n");

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

A[i][3] = A[i][1] + A[i][2];

total += A[i][3];

printf("P%d\t%d\t%d\t%d\n", A[i][0], A[i][1], A[i][2], A[i][3]);

}

avg\_tat = (float)total / n;

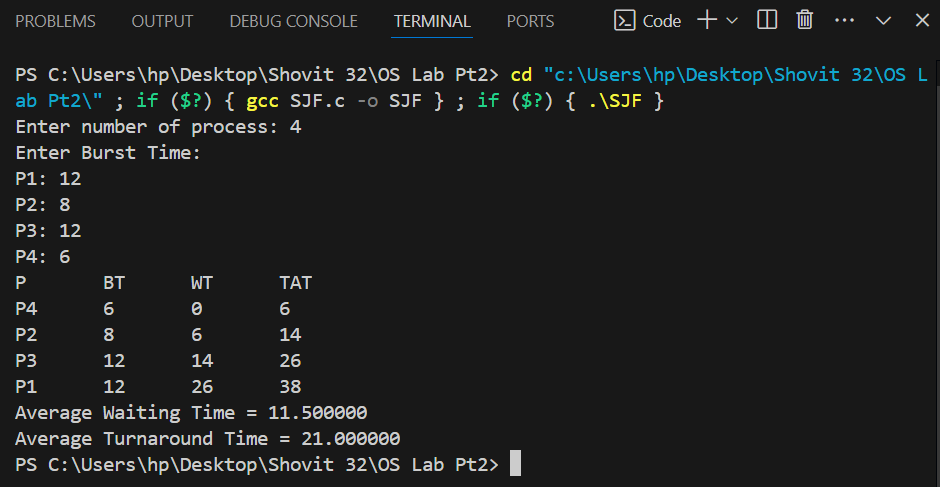
printf("Average Waiting Time = %f", avg\_wt);

printf("\nAverage Turnaround Time = %f", avg\_tat);

return 0;

}

**Output:**

****

## WAP to simulate SRTF CPU Scheduling Algorithm

**Theory:**

The algorithm described is the Shortest Remaining Time First (SRTF) scheduling, which is a preemptive version of Shortest Job First (SJF). In SRTF, the scheduler always selects the process with the least remaining burst time for execution. If a new process arrives with a shorter remaining time, it preempts the current process, saving its state in the Process Control Block (PCB) for resumption later. Once all processes are in the ready queue, SRTF behaves like SJF, as no further preemption occurs.

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 100

struct process {

int pid;

int arrival\_time;

int burst\_time;

int remaining\_time;

int waiting\_time;

int turnaround\_time;

};

int main() {

int n;

struct process processes[MAX\_PROCESSES];

bool completed[MAX\_PROCESSES] = {false};

int current\_time = 0;

int shortest\_index;

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

scanf("%d", &n);

int i;

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

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

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

processes[i].pid = i+1;

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

}

printf("\n");

int completed\_count = 0;

while (completed\_count < n) {

shortest\_index = -1;

int min\_remaining\_time = 10000;

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

if (!completed[i] && processes[i].arrival\_time <= current\_time && processes[i].remaining\_time < min\_remaining\_time) {

min\_remaining\_time = processes[i].remaining\_time;

shortest\_index = i;

}

}

if (shortest\_index != -1) {

processes[shortest\_index].remaining\_time--;

if (processes[shortest\_index].remaining\_time == 0) {

completed[shortest\_index] = true;

completed\_count++;

processes[shortest\_index].turnaround\_time = current\_time + 1 - processes[shortest\_index].arrival\_time;

processes[shortest\_index].waiting\_time = processes[shortest\_index].turnaround\_time - processes[shortest\_index].burst\_time;

}

}

current\_time++;

}

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

int total\_waiting\_time = 0;

int total\_turnaround\_time = 0;

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

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

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

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

}

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

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

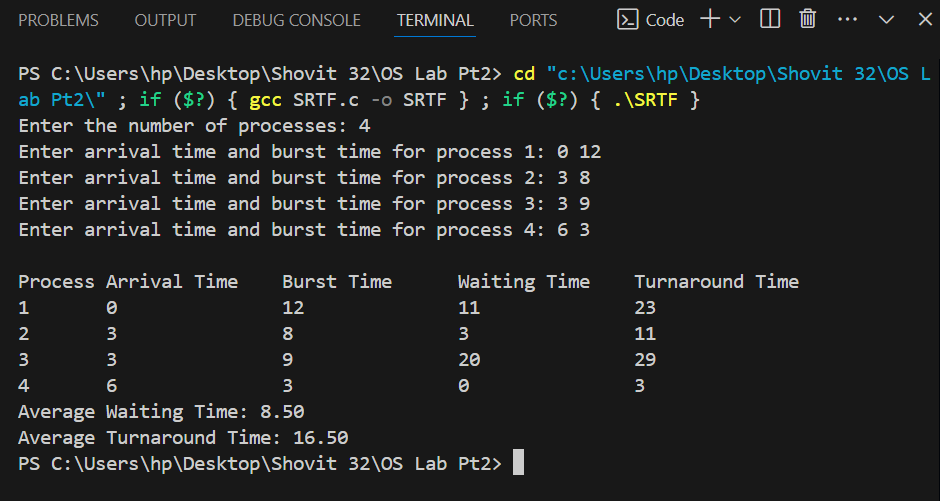
printf("Average Waiting Time: %.2f\n", avg\_waiting\_time);

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

return 0;

}

**Output:**

****

## WAP to simulate Round Robin CPU Scheduling Algorithm

**Theory:**

Round Robin is a preemptive CPU scheduling algorithm where each process gets a fixed time slot in a cyclic order, making it a time-sharing technique. It prevents starvation as each process receives equal CPU allocation, ensuring fairness. It's simple, easy to implement, and widely used due to its balanced approach.

**Code:**

#include<stdio.h>

#include<conio.h>

void main()

{

int i, NOP, sum=0,count=0, y, quant, wt=0, tat=0, at[10], bt[10], temp[10];

float avg\_wt, avg\_tat;

printf(" Total number of process in the system: ");

scanf("%d", &NOP);

y = NOP;

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

{

printf("\n Enter the Arrival and Burst time of the Process[%d]\n", i+1);

printf(" Arrival time is: \t");

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

printf(" \nBurst time is: \t");

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

temp[i] = bt[i];

}

printf("Enter the Time Quantum for the process: \t");

scanf("%d", &quant);

printf("\n Process No \t\t Burst Time \t\t TAT \t\t Waiting Time ");

for(sum=0, i = 0; y!=0; )

{

if(temp[i] <= quant && temp[i] > 0)

{

sum = sum + temp[i];

temp[i] = 0;

count=1;

}

else if(temp[i] > 0)

{

temp[i] = temp[i] - quant;

sum = sum + quant;

}

if(temp[i]==0 && count==1)

{

y--;

printf("\nProcess No[%d] \t\t %d\t\t\t\t %d\t\t\t %d", i+1, bt[i], sum-at[i], sum-at[i]-bt[i]);

wt = wt+sum-at[i]-bt[i];

tat = tat+sum-at[i];

count =0;

}

if(i==NOP-1)

{

i=0;

}

else if(at[i+1]<=sum)

{

i++;

}

else

{

i=0;

}

}

avg\_wt = wt \* 1.0/NOP;

avg\_tat = tat \* 1.0/NOP;

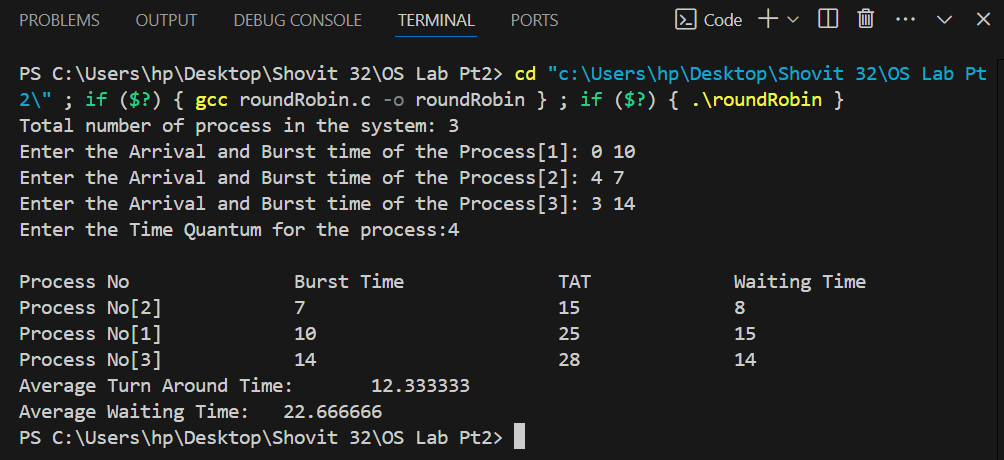
printf("\n Average Turn Around Time: \t%f", avg\_wt);

printf("\n Average Waiting Time: \t%f", avg\_tat);

getch();

}

**Output:**

****

## WAP to simulate Non-Preemptive Priority Scheduling Algorithm

**Theory:**

In the Non Preemptive Priority scheduling, the processes are scheduled according to the priority number assigned to them. Once the process gets scheduled, it will run till the completion. Generally, the lower the priority number, the higher is the priority of the process.

**Code:**

#include <stdio.h>

int i, j;

struct priority\_scheduling {

char process\_name;

int burst\_time;

int waiting\_time;

int turn\_around\_time;

int priority;

};

int main() {

int number\_of\_process;

int total = 0;

struct priority\_scheduling temp\_process;

int ASCII\_number = 65;

int position;

float average\_waiting\_time;

float average\_turnaround\_time;

printf("Enter the total number of Processes: ");

scanf("%d", &number\_of\_process);

struct priority\_scheduling process[number\_of\_process];

printf("\nPlease Enter the Burst Time and Priority of each process:\n");

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

process[i].process\_name = (char)ASCII\_number;

printf("Enter the details of the process %c \n", process[i].process\_name);

printf("Enter the burst time: ");

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

printf("Enter the priority: ");

scanf("%d", &process[i].priority);

ASCII\_number++;

}

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

position = i;

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

if (process[j].priority > process[position].priority)

position = j;

}

temp\_process = process[i];

process[i] = process[position];

process[position] = temp\_process;

}

process[0].waiting\_time = 0;

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

process[i].waiting\_time = 0;

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

process[i].waiting\_time += process[j].burst\_time;

}

total += process[i].waiting\_time;

}

average\_waiting\_time = (float)total / (float)number\_of\_process;

total = 0;

printf("\nProcess\_name \t Burst Time \t Waiting Time \t Turnaround Time\n");

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

process[i].turn\_around\_time = process[i].burst\_time + process[i].waiting\_time;

total += process[i].turn\_around\_time;

printf("\t%c \t\t%d \t\t%d \t\t%d\n", process[i].process\_name, process[i].burst\_time, process[i].waiting\_time, process[i].turn\_around\_time);

}

average\_turnaround\_time = (float)total / (float)number\_of\_process;

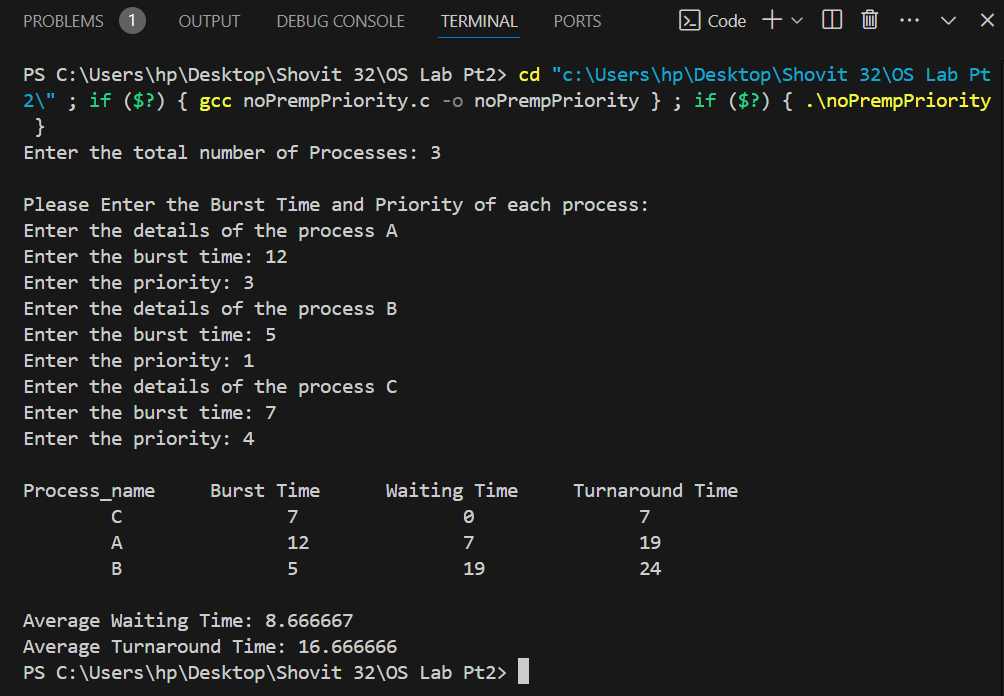
printf("\nAverage Waiting Time: %f", average\_waiting\_time);

printf("\nAverage Turnaround Time: %f\n", average\_turnaround\_time);

return 0;

}

**Output:**

****

## WAPto simulate Preemptive Priority Scheduling Algorithm

**Theory:**

In Preemptive Priority Scheduling, at the time of arrival of a process in the ready queue, its Priority is compared with the priority of the other processes present in the ready queue as well as with the one which is being executed by the CPU at that point of time. The One with the highest priority among all the available processes will be given the CPU next.

**Code:**

#include<stdio.h>

#define MIN -9999;

int i, j;

struct proc

{

int no,at,bt,rt,ct,wt,tat,pri,temp;

};

struct proc read(int i)

{

struct proc p;

printf("\nProcess No: %d\n",i);

p.no=i;

printf("Enter Arrival Time & Burst Time: ");

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

p.rt=p.bt;

printf("Enter Priority: ");

scanf("%d",&p.pri);

p.temp=p.pri;

return p;

}

void main()

{

int i,n,c,remaining,max\_val,max\_index;

struct proc p[10],temp;

float avgtat=0,avgwt=0;

printf("<--Highest Priority First Scheduling Algorithm (Preemptive)-->\n");

printf("Enter Number of Processes: ");

scanf("%d",&n);

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

p[i]=read(i+1);

remaining=n;

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

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

if(p[j].at>p[j+1].at)

{

temp=p[j];

p[j]=p[j+1];

p[j+1]=temp;

}

max\_val=p[0].temp,max\_index=0;

for( j=0;j<n&&p[j].at<=p[0].at;j++)

if(p[j].temp>max\_val)

max\_val=p[j].temp,max\_index=j;

i=max\_index;

c=p[i].ct=p[i].at+1;

p[i].rt--;

if(p[i].rt==0)

{

p[i].temp=MIN;

remaining--;

}

while(remaining>0)

{

max\_val=p[0].temp,max\_index=0;

for( j=0;j<n&&p[j].at<=c;j++)

if(p[j].temp>max\_val)

max\_val=p[j].temp,max\_index=j;

i=max\_index;

p[i].ct=c=c+1;

p[i].rt--;

if(p[i].rt==0)

{

p[i].temp=MIN;

remaining--;

}

}

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

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

{

p[i].tat=p[i].ct-p[i].at;

avgtat+=p[i].tat;

p[i].wt=p[i].tat-p[i].bt;

avgwt+=p[i].wt;

printf("P%d\t\t%d\t%d\t%d\t%d\t%d\t%d\n",p[i].no,p[i].at,p[i].bt,p[i].pri,p[i].ct,p[i].tat,p[i].wt);

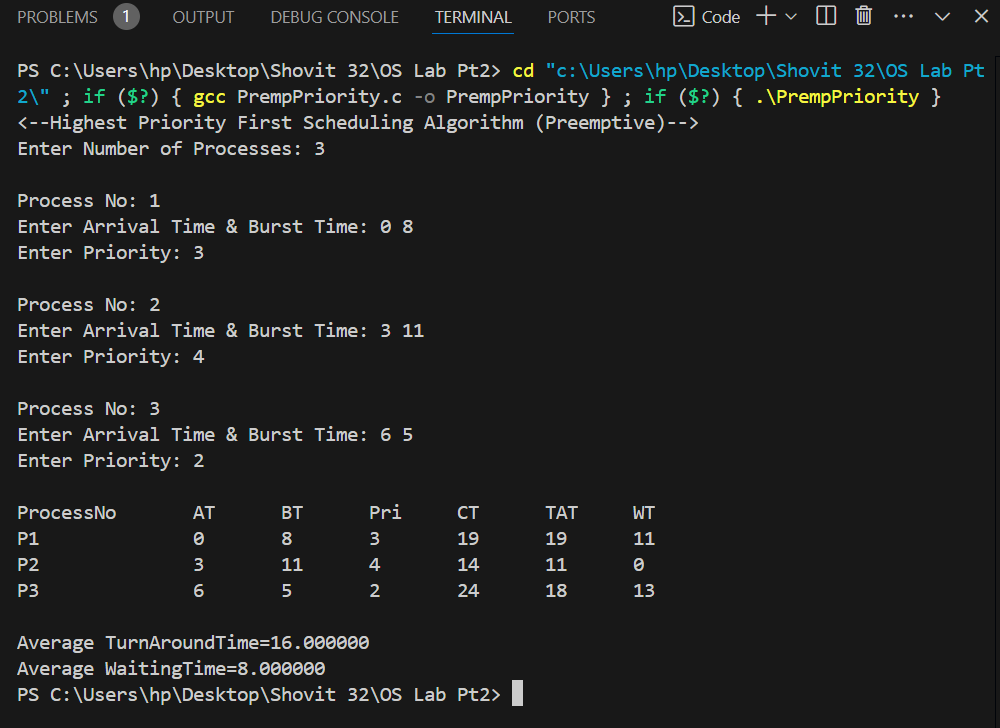
}

avgtat/=n,avgwt/=n;

printf("\nAverage TurnAroundTime=%f\nAverage WaitingTime=%f",avgtat,avgwt);

}

**Output:**



# 6. Simulation of Deadlock Avoidance and Deadlock Detection Algorithms

## WAP to implement Bankers Algorithm for multiple type of resources to decide safe/unsafe state.

**Theory:**

The Banker's Algorithm is a deadlock avoidance algorithm used to safely allocate resources to processes in a computer system. It examines all possible scenarios before allowing resource allocation, ensuring that the system remains in a safe state. Named after a banking analogy, it works similarly to assessing whether a loan can be safely granted without risking insolvency.

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void calculateNeed(int need[MAX\_PROCESSES] [MAX\_RESOURCES],

int max[MAX\_PROCESSES] [MAX\_RESOURCES],

int allocation[MAX\_PROCESSES] [MAX\_RESOURCES],

int numProcesses, int numResources) {

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

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

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

}

}

}

Bool isSafe(intprocesses[], int available[],

int max[MAX\_PROCESSES][MAX\_RESOURCES],

int allocation[MAX\_PROCESSES][MAX\_RESOURCES], int numProcesses,

int numResources) {

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

calculateNeed(need, max, allocation, numProcesses, numResources);

bool finished[MAX\_PROCESSES] = {false};

int safeSequence[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = available[i];

}

int count = 0;

while (count < numProcesses) {

bool found = false;

for (int p = 0; p < numProcesses; p++) {

if (!finished[p]) {

int j;

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

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

break;

}

}

if (j == numResources) {

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

work[k] += allocation[p][k];

}

safeSequence[count++] = p;

finished[p] = true;

found = true;

}

}

}

if (!found) {

printf("System is in an unsafe state.\n");

return false;

}

}

printf("System is in a safe state.\nSafe sequence is: ");

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

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

}

printf("\n");

return true;

}

int main() {

int numProcesses, numResources;

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

scanf("%d", &numProcesses);

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

scanf("%d", &numResources);

int processes[MAX\_PROCESSES];

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

processes[i] = i;

}

int available[MAX\_RESOURCES];

printf("Enter the available resources: ");

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

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

}

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

printf("Enter the maximum demand of each process:\n");

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

printf("Process %d: ", i);

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

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

}

}

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

printf("Enter the allocated resources for each process:\n");

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

printf("Process %d: ", i);

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

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

}

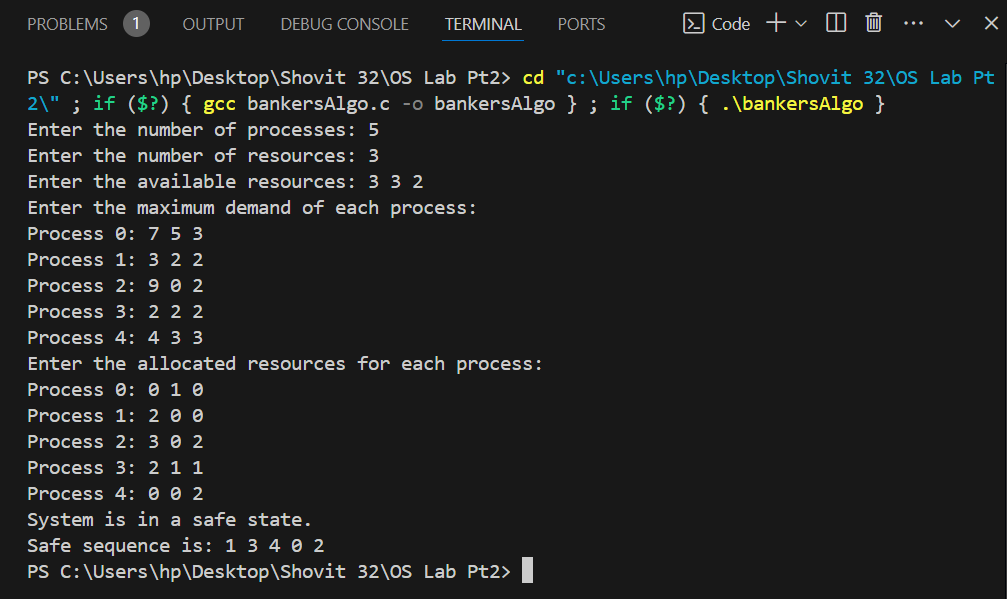
}

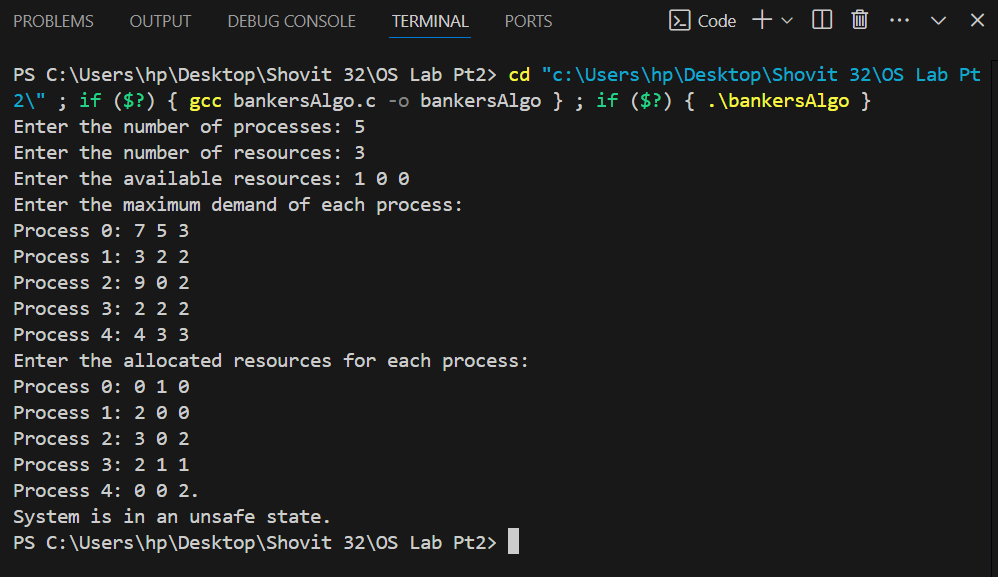
isSafe(processes, available, max, allocation, numProcesses, numResources);

return 0;

}

**Output:**





## WAP for deadlock detection in the system having multiple type of resources.

**Theory:**

In systems with multiple types of resources, deadlock detection algorithms identify processes involved in a deadlock by periodically checking for circular wait conditions among resource allocations. If deadlock is detected, the algorithm lists the deadlocked processes, allowing for corrective actions like process termination or resource preemption to resolve the deadlock and free up the resources.

**Code:**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

void calculateNeed(int need[MAX\_PROCESSES][MAX\_RESOURCES], int max[MAX\_PROCESSES][MAX\_RESOURCES], int allocation[MAX\_PROCESSES][MAX\_RESOURCES], int numProcesses, int numResources) {

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

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

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

}

}

}

bool deadlockDetection(int processes[],

int available[],

int max[MAX\_PROCESSES][MAX\_RESOURCES],

int allocation[MAX\_PROCESSES][MAX\_RESOURCES],

int numProcesses, int numResources)

{

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

calculateNeed (need, max, allocation, numProcesses, numResources);

bool finished[MAX\_PROCESSES] = {false};

int work[MAX\_RESOURCES];

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

work[i] = available[i];

}

int deadlocked[MAX\_PROCESSES];

int deadlockCount = 0;

while (true) {

bool found = false;

for (int p = 0; p < numProcesses; p++) {

if (!finished[p]) {

int j;

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

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

break;

}

}

if (j == numResources) {

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

work[k] += allocation[p][k];

}

finished[p] = true;

found = true;

}

}

}

if (!found) {

break;

}

}

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

if (!finished[i]) {

deadlocked[deadlockCount++] = i;

}

}

if (deadlockCount > 0) {

printf("Deadlock detected. Deadlocked processes are: ");

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

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

}

printf("\n");

return true;

} else {

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

return false;

}

}

int main() {

int numProcesses, numResources;

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

scanf("%d", &numProcesses);

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

scanf("%d", &numResources);

int processes[MAX\_PROCESSES];

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

processes[i] = i;

}

int available[MAX\_RESOURCES];

printf("Enter the available resources: ");

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

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

}

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

printf("Enter the maximum demand of each process:\n");

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

printf("Process %d: ", i);

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

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

}

}

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

printf("Enter the allocated resources for each process:\n");

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

printf("Process %d: ", i);

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

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

}

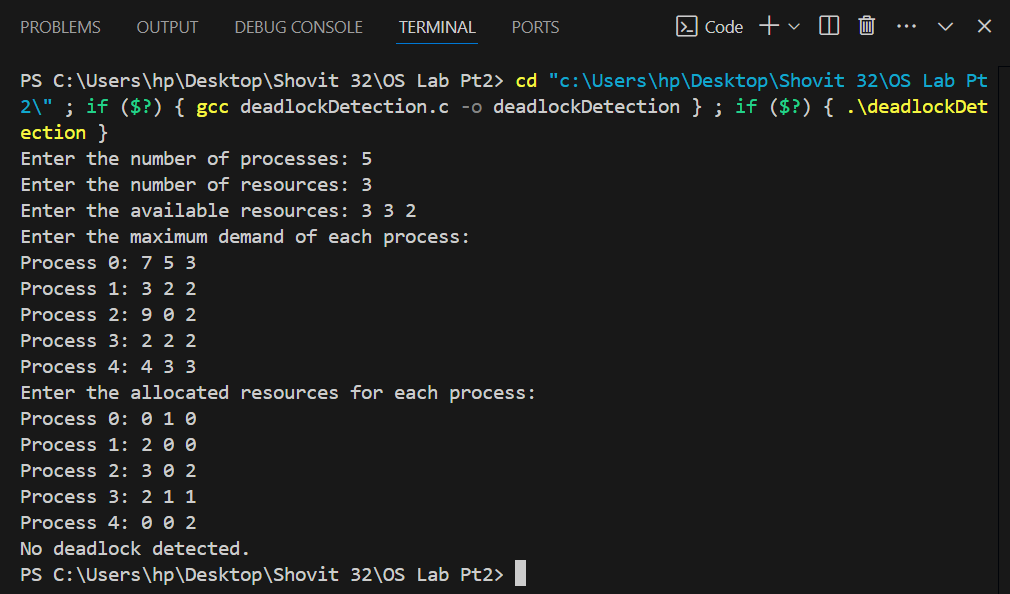
}

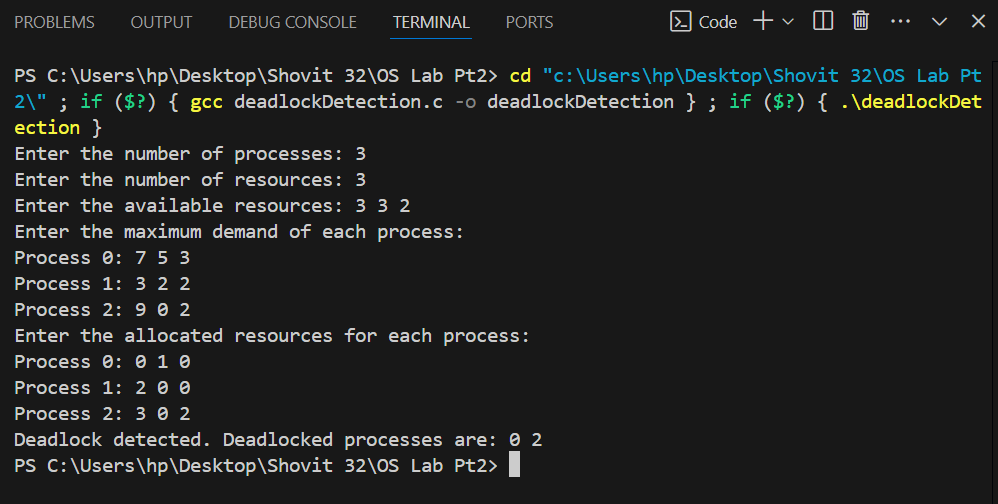
deadlockDetection(processes, available, max, allocation, numProcesses, numResources);

return 0;

}

**Output:**

****

****

# 7. Simulation of Page Replacement Algorithms

## WAP to simulate FIFO Page Replacement Algorithm

**Theory:**

The FIFO (First-In-First-Out) Page Replacement Algorithm in operating systems manages memory by replacing the oldest page in memory when a new page needs to be loaded, following the order of arrival. Pages are stored in a queue, and when a page fault occurs, the page at the front of the queue is replaced. This method is simple but may not always be the most efficient, as it doesn't consider page usage frequency.

**Code:**

#include <stdio.h>

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

int frames[capacity];

int front = 0, rear = 0, pageFaults = 0;

int isFull = 0;

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

frames[i] = -1;

}

printf("Page Frames at each step:\n");

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

int page = pages[i];

int found = 0;

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

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

frames[rear] = page;

rear = (rear + 1) % capacity;

if (rear == front) {

isFull = 1;

front = (front + 1) % capacity; // Move front if rear catches up

}

pageFaults++;

}

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

if (frames[j] == -1) {

printf(" - ");

} else {

printf("%2d ", frames[j]);

}

}

printf("\n");

}

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

}

int main() {

int n, capacity;

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

scanf("%d", &n);

int pages[n];

printf("Enter the reference string (space-separated page numbers): ");

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

{

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

}

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

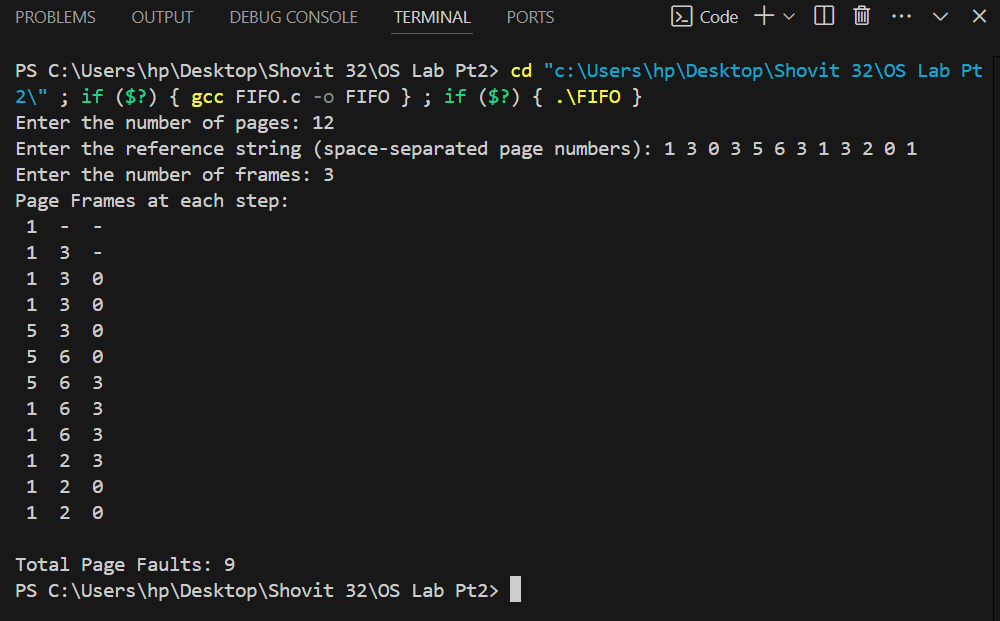
scanf("%d", &capacity);

fifoPageReplacement(pages, n, capacity);

return 0;

}

**Output:**

****

## WAP to simulate Optimal Page Replacement Algorithm

**Theory:**

The Optimal Page Replacement Algorithm replaces the page that will not be needed for the longest time in the future when a page fault occurs. It provides the best possible page fault rate by minimizing the number of page replacements, but it is not practical for real-time use because it requires future knowledge of page requests. It serves as a benchmark to compare the performance of other page replacement algorithms.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 10

void print\_frames(int frames[], int num\_frames) {

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

if (frames[i] != -1)

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

else

printf("- ");

}

printf("\n");

}

int find\_page\_index(int pages[], int page\_count, int frames[], int num\_frames, int current\_index) {

int index = -1, farthest = current\_index;

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

int j;

for (j = current\_index; j < page\_count; j++) {

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

if (j > farthest) {

farthest = j;

index = i;

}

break;

}

}

if (j == page\_count) // Page not found in future references

return i;

}

return (index == -1) ? 0 : index;

}

int main() {

int pages[MAX\_PAGES], frames[MAX\_FRAMES];

int page\_count, num\_frames;

int page\_faults = 0;

// Initialize frames

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1;

}

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

scanf("%d", &page\_count);

printf("Enter the page reference string (space-separated): ");

for (int i = 0; i < page\_count; i++) {

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

}

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

scanf("%d", &num\_frames);

printf("\nPage Reference String: ");

for (int i = 0; i < page\_count; i++) {

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

}

printf("\n");

printf("\nPage Replacement Process:\n");

for (int i = 0; i < page\_count; i++) {

int page = pages[i];

int found = 0;

for (int j = 0; j < num\_frames; j++) {

if (frames[j] == page) {

found = 1;

break;

}

}

if (!found) {

int index = find\_page\_index(pages, page\_count, frames, num\_frames, i + 1);

frames[index] = page;

page\_faults++;

printf("Page %d causes a page fault.\n", page);

print\_frames(frames, num\_frames);

}

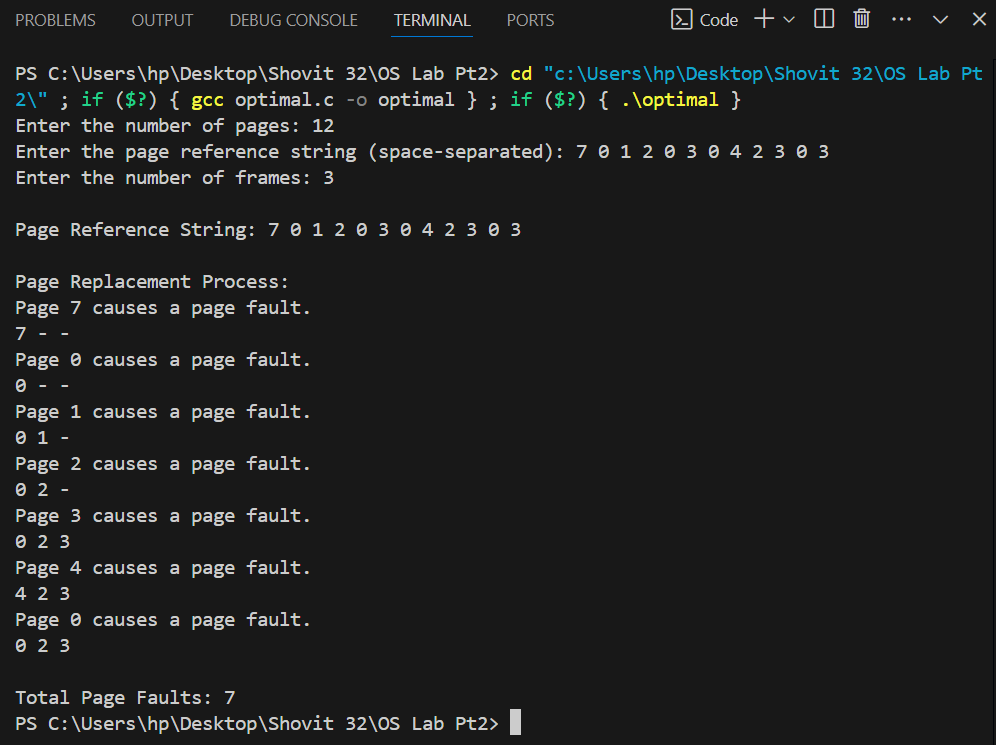
}

printf("\nTotal Page Faults: %d\n", page\_faults);

return 0;

}

**Output:**

****

## WAP to simulate LRU Page Replacement Algorithm

**Theory:**

The LRU (Least Recently Used) Page Replacement Algorithm replaces the page that has not been used for the longest time when a page fault occurs. It approximates optimal performance by keeping track of the order of page usage and selecting the least recently accessed page for replacement. LRU effectively balances performance and complexity, making it a popular choice for page management in operating systems.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 10

void print\_frames(int frames[], int num\_frames) {

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

if (frames[i] != -1)

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

else

printf("- ");

}

printf("\n");

}

int find\_page\_index(int page, int frames[], int num\_frames, int time[], int page\_count, int current\_index) {

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

if (frames[i] == page) {

time[i] = current\_index;

return -1; // Page found, no replacement needed

}

}

int oldest = 0;

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

if (time[i] < time[oldest]) {

oldest = i;

}

}

return oldest; // Page index to replace

}

int main() {

int pages[MAX\_PAGES], frames[MAX\_FRAMES];

int page\_count, num\_frames;

int page\_faults = 0;

int time[MAX\_FRAMES];

// Initialize frames and time arrays

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1;

time[i] = -1;

}

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

scanf("%d", &page\_count);

printf("Enter the page reference string (space-separated): ");

for (int i = 0; i < page\_count; i++) {

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

}

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

scanf("%d", &num\_frames);

printf("\nPage Reference String: ");

for (int i = 0; i < page\_count; i++) {

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

}

printf("\n");

printf("\nPage Replacement Process:\n");

for (int i = 0; i < page\_count; i++)

{

int page = pages[i];

int index = find\_page\_index(page, frames, num\_frames, time, page\_count, i);

if (index != -1) {

frames[index] = page;

time[index] = i;

page\_faults++;

printf("Page %d causes a page fault.\n", page);

print\_frames(frames, num\_frames);

}

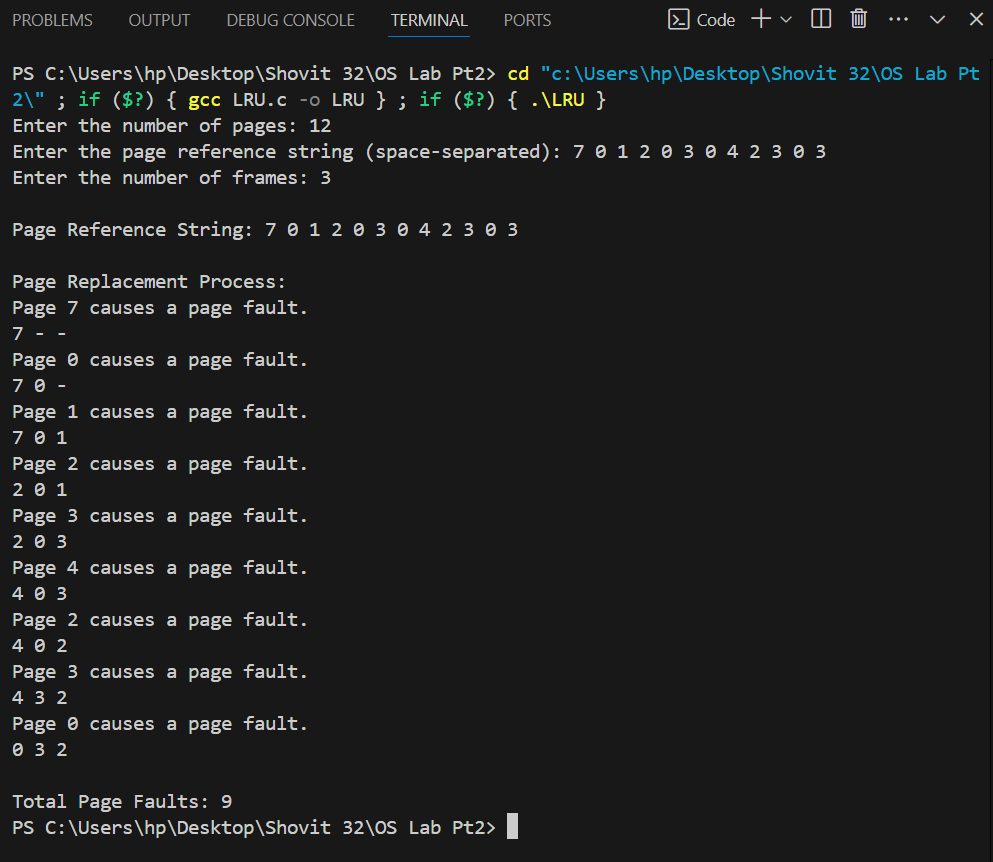
}

printf("\nTotal Page Faults: %d\n", page\_faults);

return 0;

}

**Output:**



## WAP to simulate Second Chance Page Replacement Algorithm.

**Theory:**

The Second Chance Page Replacement Algorithm is an enhancement of the FIFO algorithm that gives pages a "second chance" if they have been accessed recently. Each page has a reference bit; when a page is considered for replacement, if its reference bit is set (indicating recent use), it is cleared and the page is moved to the back of the queue, granting it a second chance.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 10

void print\_frames(int frames[], int num\_frames, int reference\_bits[]) {

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

if (frames[i] != -1)

printf("%d(%d) ", frames[i], reference\_bits[i]);

else

printf("- ");

}

printf("\n");

}

int main() {

int pages[MAX\_PAGES], frames[MAX\_FRAMES];

int page\_count, num\_frames;

int page\_faults = 0;

int reference\_bits[MAX\_FRAMES] = {0};

int front = 0, rear = 0;

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1;

}

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

scanf("%d", &page\_count);

printf("Enter the page reference string (space-separated): ");

for (int i = 0; i < page\_count; i++) {

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

}

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

scanf("%d", &num\_frames);

printf("\nPage Reference String: ");

for (int i = 0; i < page\_count; i++) {

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

}

printf("\n");

printf("\nPage Replacement Process:\n");

for (int i = 0; i < page\_count; i++) {

int page = pages[i];

int found = 0;

for (int j = 0; j < num\_frames; j++) {

if (frames[j] == page) {

reference\_bits[j] = 1;

found = 1;

break;

}

}

if (!found)

{

while (reference\_bits[front] == 1) {

reference\_bits[front] = 0;

front = (front + 1) % num\_frames;

}

frames[front] = page;

reference\_bits[front] = 1;

front = (front + 1) % num\_frames;

page\_faults++;

printf("Page %d causes a page fault.\n", page);

print\_frames(frames, num\_frames, reference\_bits);

}

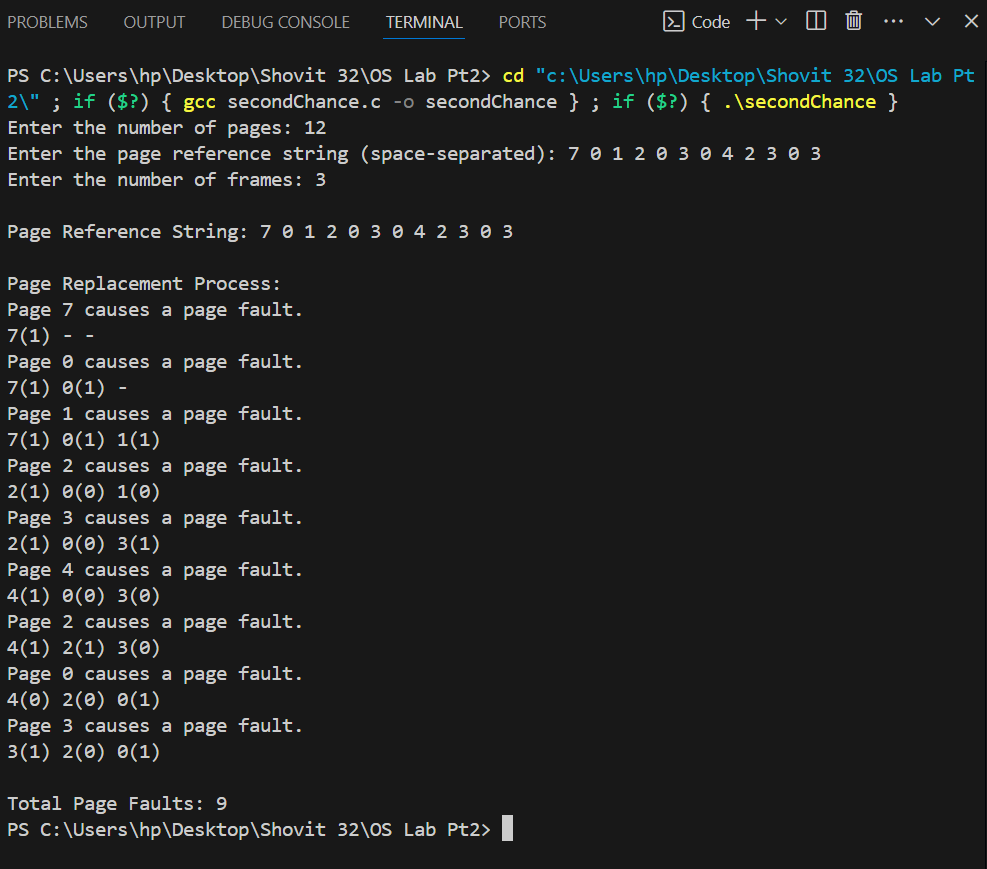
}

printf("\nTotal Page Faults: %d\n", page\_faults);

return 0;

}

**Output:**

****

## WAP to simulate LFU Page Replacement Algorithm

**Theory:**

The LFU (Least Frequently Used) Page Replacement Algorithm replaces the page with the lowest access frequency when a page fault occurs. It keeps track of how often each page is accessed, and pages with the least usage are replaced first. LFU aims to keep frequently accessed pages in memory, making it efficient for scenarios with predictable access patterns, but it can be slow to adapt to changes in access frequency over time.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100

#define MAX\_FRAMES 10

void print\_frames(int frames[], int num\_frames, int counts[]) {

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

if (frames[i] != -1)

printf("%d(%d) ", frames[i], counts[i]);

else

printf("- ");

}

printf("\n");

}

int find\_page\_index(int page, int frames[], int num\_frames, int counts[], int page\_count) {

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

if (frames[i] == page) {

counts[i]++;

return -1; // Page found, no replacement needed

}

}

int min\_count = counts[0];

int index = 0;

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

if (counts[i] < min\_count) {

min\_count = counts[i];

index = i;

}

}

return index; // Page index to replace

}

int main() {

int pages[MAX\_PAGES], frames[MAX\_FRAMES];

int page\_count, num\_frames;

int page\_faults = 0;

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

for (int i = 0; i < MAX\_FRAMES; i++) {

frames[i] = -1;

}

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

scanf("%d", &page\_count);

printf("Enter the page reference string (space-separated): ");

for (int i = 0; i < page\_count; i++) {

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

}

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

scanf("%d", &num\_frames);

printf("\nPage Reference String: ");

for (int i = 0; i < page\_count; i++) {

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

}

printf("\n");

printf("\nPage Replacement Process:\n");

for (int i = 0; i < page\_count; i++) {

int page = pages[i];

int index = find\_page\_index(page, frames, num\_frames, counts, page\_count);

if (index != -1) {

frames[index] = page;

counts[index] = 1;

page\_faults++;

printf("Page %d causes a page fault.\n", page);

print\_frames(frames, num\_frames, counts);

}

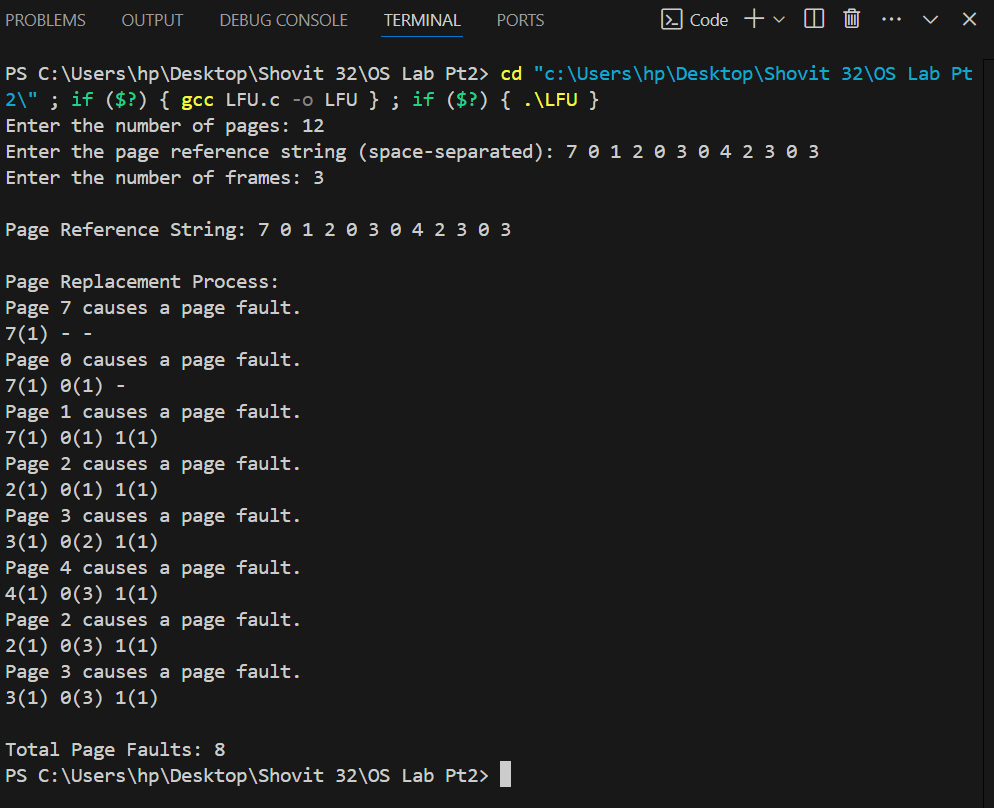
}

printf("\nTotal Page Faults: %d\n", page\_faults);

return 0;

}

**Output:**

****

# 8. Simulation of File Allocation Techniques

## WAP to simulate Contiguous File Allocation Technique

**Theory:**

The Contiguous File Allocation Technique stores a file in a single contiguous block of disk space. This method simplifies file access and improves performance by minimizing seek time and fragmentation. However, it can lead to external fragmentation and difficulties in allocating large files if sufficient contiguous space is not available.

**Code:**

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

void ContiguousAllocation(int file\_sizes[], int num\_files, int disk\_size) {

int i, j;

int disk[MAX\_BLOCKS];

int file\_start[MAX\_FILES];

for (i = 0; i < disk\_size; i++) {

disk[i] = -1;

}

printf("Contiguous File Allocation:\n");

for (i = 0; i < num\_files; i++) {

int size = file\_sizes[i];

int start\_block = -1;

for (j = 0; j < disk\_size; j++) {

if (disk[j] == -1) {

if (start\_block == -1) {

start\_block = j;

}

size--;

if (size == 0) {

break;

}

} else {

start\_block = -1;

size = file\_sizes[i];

}

}

if (size > 0) {

printf("File %d could not be allocated\n", i + 1);

continue;

}

for (j = start\_block; j < start\_block + file\_sizes[i]; j++) {

disk[j] = i + 1;

}

file\_start[i] = start\_block;

printf("File %d allocated from block %d to block %d\n", i + 1, start\_block, start\_block + file\_sizes[i] - 1);

}

printf("\nDisk allocation:\n");

for (i = 0; i < disk\_size; i++) {

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

}

printf("\n");

}

int main() {

int file\_sizes[] = {5, 10, 20}; // Example file sizes

int num\_files = 3;

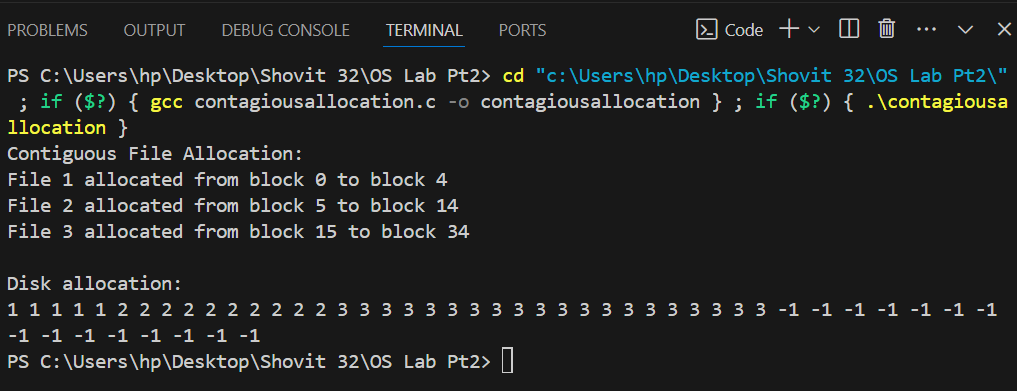
int disk\_size = 50; // Example disk size

ContiguousAllocation(file\_sizes, num\_files, disk\_size);

return 0;

}

**Output:**

****

## WAP to simulate Linked File Allocation Technique

**Theory:**

The Linked File Allocation Technique stores a file as a linked list of disk blocks, where each block contains a pointer to the next block. This method eliminates external fragmentation and allows for dynamic file growth, but can result in increased access time due to the need to follow pointers to read the entire file.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

typedef struct Node {

int block;

struct Node\* next;

}

Node;

void LinkedAllocation(int file\_sizes[], int num\_files, int disk\_size) {

int i, j;

int disk[MAX\_BLOCKS];

Node\* file\_blocks[MAX\_FILES] = {NULL};

for (i = 0; i < disk\_size; i++) {

disk[i] = -1;

}

printf("Linked File Allocation:\n");

for (i = 0; i < num\_files; i++) {

int size = file\_sizes[i];

int start\_block = -1;

int current\_block = -1;

for (j = 0; j < disk\_size; j++) {

if (disk[j] == -1) {

if (start\_block == -1) {

start\_block = j;

}

if (current\_block != -1) {

disk[current\_block] = j;

}

current\_block = j;

size--;

if (size == 0) {

break;

}

} else {

start\_block = -1;

size = file\_sizes[i];

current\_block = -1;

}

}

if (size > 0) {

printf("File %d could not be allocated\n", i + 1);

continue;

}

disk[current\_block] = -1; // End of file

Node\* head = NULL;

Node\* tail = NULL;

for (j = start\_block; j <= current\_block; j++) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->block = j;

new\_node->next = NULL;

if (head == NULL) {

head = new\_node;

tail = head;

} else {

tail->next = new\_node;

tail = new\_node;

}

}

file\_blocks[i] = head;

printf("File %d allocated from block %d to block %d\n", i + 1, start\_block, current\_block);

}

printf("\nDisk allocation:\n");

for (i = 0; i < disk\_size; i++) {

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

}

printf("\n");

for (i = 0; i < num\_files; i++) {

Node\* current = file\_blocks[i];

printf("File %d blocks: ", i + 1);

while (current != NULL) {

printf("%d ", current->block);

Node\* temp = current;

current = current->next;

free(temp);

}

printf("\n");

}

}

int main() {

int file\_sizes[] = {5, 10, 20}; // Example file sizes

int num\_files = 3;

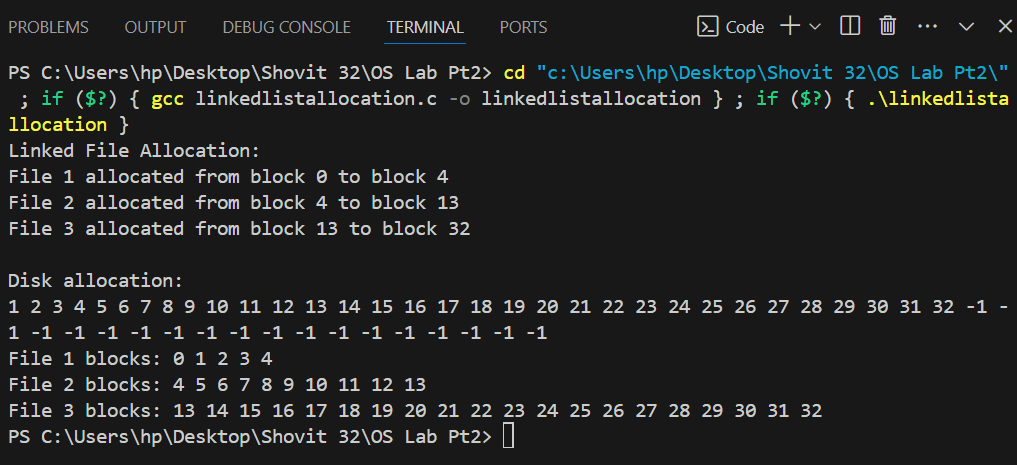
int disk\_size = 50; // Example disk size

LinkedAllocation(file\_sizes, num\_files, disk\_size);

return 0;

}

**Output:**

****

## WAP to simulate File Allocation using File Allocation Table

**Theory:**

File Allocation using the File Allocation Table (FAT) method involves a table where each entry corresponds to a disk block and points to the next block in the file. This allows for efficient file management and allocation by keeping track of which blocks are used and the order in which they are linked. It simplifies file access and supports dynamic file growth but can suffer from fragmentation over time.

**Code:**

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

void FATAllocation(int file\_sizes[], int num\_files, int disk\_size) {

int i, j;

int disk[MAX\_BLOCKS];

int FAT[MAX\_BLOCKS];

for (i = 0; i < disk\_size; i++) {

disk[i] = -1;

FAT[i] = -1;

}

printf("FAT File Allocation:\n");

for (i = 0; i < num\_files; i++) {

int size = file\_sizes[i];

int start\_block = -1;

int current\_block = -1;

for (j = 0; j < disk\_size; j++) {

if (disk[j] == -1) {

if (start\_block == -1) {

start\_block = j;

}

if (current\_block != -1) {

FAT[current\_block] = j;

}

current\_block = j;

size--;

if (size == 0) {

break;

}

} else {

start\_block = -1;

size = file\_sizes[i];

current\_block = -1;

}

}

if (size > 0) {

printf("File %d could not be allocated\n", i + 1);

continue;

}

FAT[current\_block] = -1; // End of file

for (j = start\_block; j <= current\_block; j++) {

disk[j] = i + 1;

}

printf("File %d allocated from block %d to block %d\n", i + 1, start\_block, current\_block);

}

printf("\nDisk allocation:\n");

for (i = 0; i < disk\_size; i++) {

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

}

printf("\n");

printf("FAT table:\n");

for (i = 0; i < disk\_size; i++) {

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

}

printf("\n");

}

int main() {

int file\_sizes[] = {5, 10, 20}; // Example file sizes

int num\_files = 3;

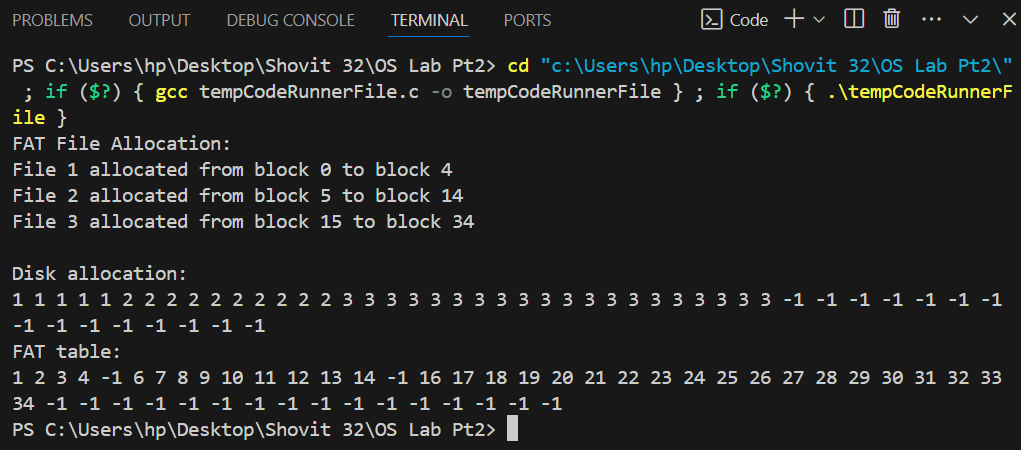
int disk\_size = 50; // Example disk size

FATAllocation(file\_sizes, num\_files, disk\_size);

return 0;

}

**Output:**

****

## WAP to implement File Allocation using Inode

**Theory:**

File Allocation using Inodes involves storing file metadata (such as file size, permissions, and pointers to data blocks) in a structure called an inode. Each file has an associated inode, which points to the data blocks where the file's content is stored. This method allows for efficient file management, supports large files, and enables flexible file system operations.

**Code:**

#include <stdio.h>

#define MAX\_BLOCKS 100

#define MAX\_FILES 10

typedef struct {

int block\_start;

int block\_end;

int size;

} Inode;

void InodeAllocation(int file\_sizes[], int num\_files, int disk\_size) {

int i, j;

int disk[MAX\_BLOCKS];

Inode inodes[MAX\_FILES];

for (i = 0; i < disk\_size; i++) {

disk[i] = -1;

}

printf("Inode File Allocation:\n");

for (i = 0; i < num\_files; i++) {

int size = file\_sizes[i];

int start\_block = -1;

int current\_block = -1;

for (j = 0; j < disk\_size; j++) {

if (disk[j] == -1) {

if (start\_block == -1) {

start\_block = j;

}

current\_block = j;

size--;

if (size == 0) {

break;

}

} else {

start\_block = -1;

size = file\_sizes[i];

current\_block = -1;

}

}

if (size > 0) {

printf("File %d could not be allocated\n", i + 1);

continue;

}

for (j = start\_block; j <= current\_block; j++) {

disk[j] = i + 1;

}

inodes[i].block\_start = start\_block;

inodes[i].block\_end = current\_block;

inodes[i].size = file\_sizes[i];

printf("File %d allocated from block %d to block %d\n", i + 1, start\_block, current\_block);

}

printf("\nDisk allocation:\n");

for (i = 0; i < disk\_size; i++) {

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

}

printf("\n");

printf("Inode table:\n");

for (i = 0; i < num\_files; i++) {

printf("File %d: Start Block %d, End Block %d, Size %d\n", i + 1, inodes[i].block\_start, inodes[i].block\_end, inodes[i].size);

}

}

int main()

{

int file\_sizes[] = {5, 10, 20}; // Example file sizes

int num\_files = 3;

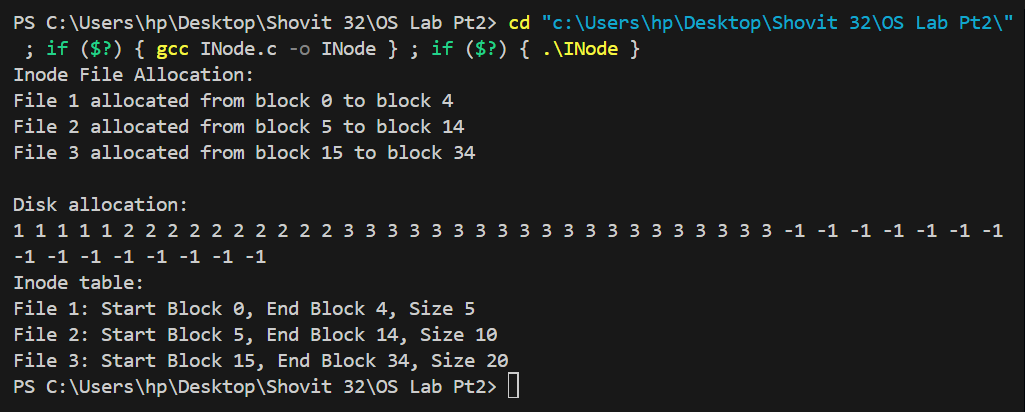
int disk\_size = 50; // Example disk size

InodeAllocation(file\_sizes, num\_files, disk\_size);

return 0;

}

**Output:**

****

# 9. Simulation of Free Space Management Techniques

## WAP to simulate Free Space Management using Bitmaps

**Theory:**

Free Space Management using Bitmaps tracks disk space availability with a bitmap where each bit represents a block on the disk. A bit set to 0 indicates a free block, while a bit set to 1 indicates an allocated block. This method efficiently manages and allocates free space, making it easy to find and allocate contiguous blocks.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_BLOCKS 100

typedef struct {

char bitmap[(MAX\_BLOCKS + 7) / 8];

} FileSystem;

void initializeFileSystem(FileSystem \*fs) {

memset(fs->bitmap, 0, sizeof(fs->bitmap));

}

void allocateBlocks(FileSystem \*fs, int num\_blocks, int allocated\_blocks[]) {

int i, count = 0;

for (i = 0; i < MAX\_BLOCKS && count < num\_blocks; i++) {

if ((fs->bitmap[i / 8] & (1 << (i % 8))) == 0) {

// Block is free

fs->bitmap[i / 8] |= (1 << (i % 8));

allocated\_blocks[count++] = i;

}

}

if (count < num\_blocks) {

printf("Not enough free blocks available.\n");

}

}

void deallocateBlocks(FileSystem \*fs, int num\_blocks, int blocks[]) {

int i;

for ( i = 0; i < num\_blocks; i++) {

int block = blocks[i];

fs->bitmap[block / 8] &= ~(1 << (block % 8));

}

}

void printBitmap(FileSystem \*fs) {

int i;

printf("Bitmap: ");

for ( i = 0; i < MAX\_BLOCKS; i++) {

if ((fs->bitmap[i / 8] & (1 << (i % 8))) == 0) {

printf("0 ");

} else {

printf("1 ");

}

}

printf("\n");

}

int main() {

FileSystem fs;

initializeFileSystem(&fs);

int allocated\_blocks[MAX\_BLOCKS];

int num\_blocks\_to\_allocate = 10;

int i;

allocateBlocks(&fs, num\_blocks\_to\_allocate, allocated\_blocks);

printf("Allocated blocks: ");

for ( i = 0; i < num\_blocks\_to\_allocate; i++) {

printf("%d ", allocated\_blocks[i]);

}

printf("\n");

printBitmap(&fs);

int num\_blocks\_to\_deallocate = 5;

int blocks\_to\_deallocate[num\_blocks\_to\_deallocate];

for ( i = 0; i < num\_blocks\_to\_deallocate; i++) {

blocks\_to\_deallocate[i] = allocated\_blocks[i];

}

deallocateBlocks(&fs, num\_blocks\_to\_deallocate, blocks\_to\_deallocate);

printf("Deallocated blocks: ");

for ( i = 0; i < num\_blocks\_to\_deallocate; i++) {

printf("%d ", blocks\_to\_deallocate[i]);

}

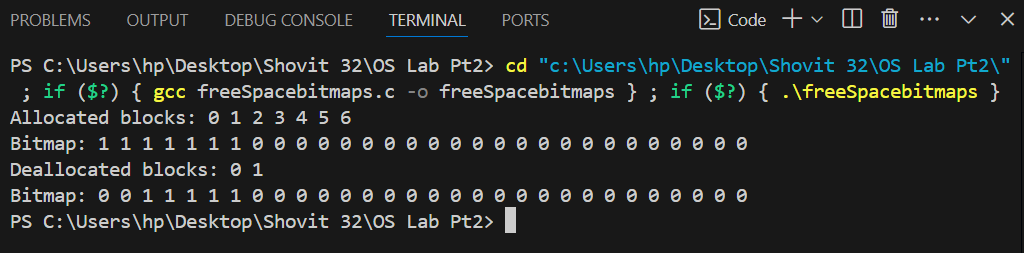
printf("\n");

printBitmap(&fs);

return 0;

}

**Output:**



## WAP to simulate Free Space Management using Linked List

**Theory:**

Free Space Management using a Linked List involves maintaining a linked list of free disk blocks. Each block contains a pointer to the next free block. This method simplifies allocation and deallocation of free space and efficiently tracks free blocks, but it can introduce overhead due to additional pointers and may be less efficient in locating large contiguous spaces.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 100

typedef struct Node {

int block\_number;

struct Node \*next;

} Node;

typedef struct {

Node \*head;

} FileSystem;

void initializeFileSystem(FileSystem \*fs) {

fs->head = NULL;

int i;

for ( i = MAX\_BLOCKS - 1; i >= 0; i--) {

Node \*new\_node = (Node \*)malloc(sizeof(Node));

new\_node->block\_number = i;

new\_node->next = fs->head;

fs->head = new\_node;

}

}

int allocateBlock(FileSystem \*fs) {

if (fs->head == NULL) {

printf("No free blocks available.\n");

return -1;

}

Node \*allocated\_node = fs->head;

int block\_number = allocated\_node->block\_number;

fs->head = allocated\_node->next;

free(allocated\_node);

return block\_number;

}

void deallocateBlock(FileSystem \*fs, int block\_number) {

Node \*new\_node = (Node \*)malloc(sizeof(Node));

new\_node->block\_number = block\_number;

new\_node->next = fs->head;

fs->head = new\_node;

}

void printFreeBlocks(FileSystem \*fs) {

Node \*current = fs->head;

printf("Free blocks: ");

while (current != NULL) {

printf("%d ", current->block\_number);

current = current->next;

}

printf("\n");

}

int main() {

FileSystem fs;

initializeFileSystem(&fs);

printf("Initial free blocks:\n");

printFreeBlocks(&fs);

int allocated\_blocks[5];

printf("Allocating 5 blocks:\n");

int i;

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

allocated\_blocks[i] = allocateBlock(&fs);

printf("Allocated block: %d\n", allocated\_blocks[i]);

}

printf("Free blocks after allocation:\n");

printFreeBlocks(&fs);

printf("Deallocating 2 blocks:\n");

deallocateBlock(&fs, allocated\_blocks[0]);

deallocateBlock(&fs, allocated\_blocks[1]);

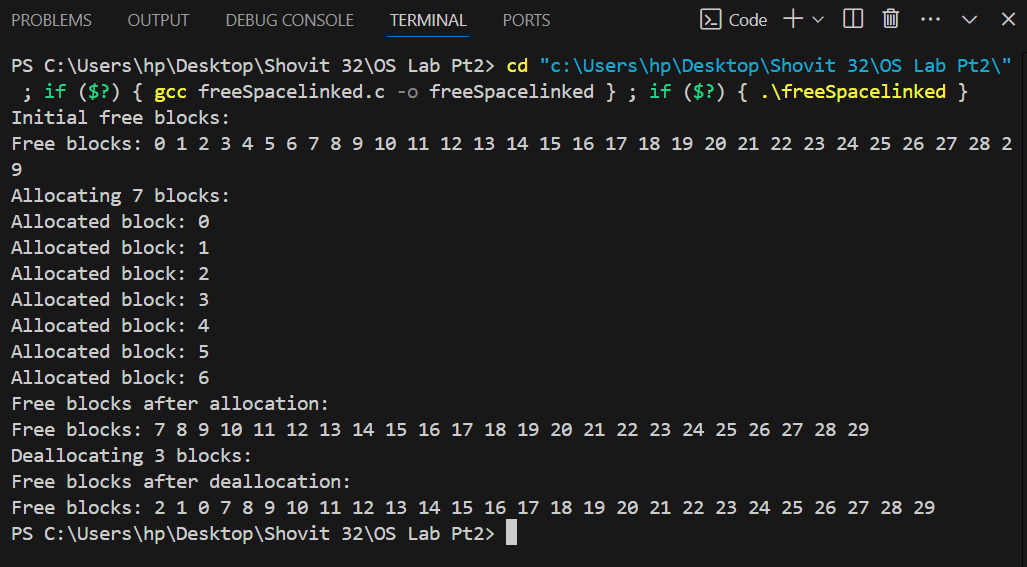
printf("Free blocks after deallocation:\n");

printFreeBlocks(&fs);

return 0;

}

**Output:**

****

# 10. Simulation of disk scheduling algorithms

## WAP to simulate FCFS Disk Scheduling Algorithm

**Theory:**

The FCFS (First-Come, First-Served) Disk Scheduling Algorithm services disk requests in the order they arrive. It is simple and easy to implement but can lead to inefficient disk head movement and longer wait times, especially if requests are scattered across the disk.

**Code:**

#include <stdio.h>

#include <stdlib.h>

void FCFS(int requests[], int num\_requests, int initial\_head) {

int i, total\_head\_movement = 0;

int current\_head = initial\_head;

printf("FCFS Disk Scheduling:\n");

for (i = 0; i < num\_requests; i++) {

printf("Move from %d to %d\n", current\_head, requests[i]);

total\_head\_movement += abs(current\_head - requests[i]);

current\_head = requests[i];

}

printf("Total Head Movement: %d\n", total\_head\_movement);

}

int main()

{

int num\_requests;

int initial\_head;

printf("Enter the number of disk requests: ");

scanf("%d", &num\_requests);

int requests[num\_requests];

printf("Enter the initial head position: ");

scanf("%d", &initial\_head);

printf("Enter the disk requests: ");

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

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

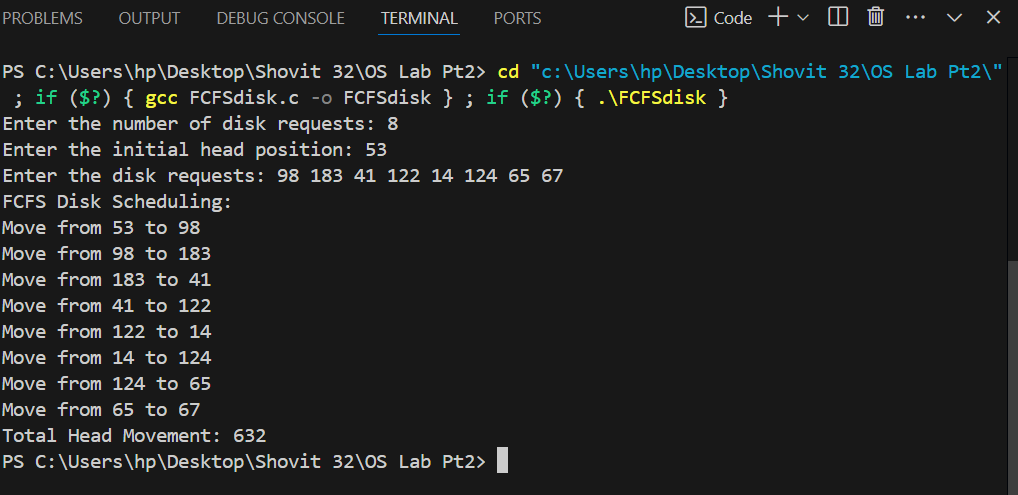
}

FCFS(requests, num\_requests, initial\_head);

return 0;

}

**Output:**

****

## WAP to simulate SSTF Disk Scheduling Algorithm

**Theory:**

The SSTF (Shortest Seek Time First) Disk Scheduling Algorithm selects the disk request closest to the current head position, minimizing seek time by always servicing the nearest request first. It improves efficiency over FCFS but can cause starvation for requests that are far from the current head position if many closer requests keep coming in.

**Code:**

#include <stdio.h>

#include <stdlib.h>

void SSTF(int requests[], int num\_requests, int initial\_head) {

int i, j, min\_dist, total\_head\_movement = 0;

int visited[num\_requests];

int current\_head = initial\_head;

int num\_visited = 0;

for (i = 0; i < num\_requests; i++) {

visited[i] = 0;

}

printf("SSTF Disk Scheduling:\n");

while (num\_visited < num\_requests) {

min\_dist = 10000;

int closest = -1;

for (i = 0; i < num\_requests; i++) {

if (!visited[i] && abs(requests[i] - current\_head) < min\_dist) {

min\_dist = abs(requests[i] - current\_head);

closest = i;

}

}

if (closest != -1) {

printf("Move from %d to %d\n", current\_head, requests[closest]);

total\_head\_movement += min\_dist;

current\_head = requests[closest];

visited[closest] = 1;

num\_visited++;

}

}

printf("Total Head Movement: %d\n", total\_head\_movement);

}

int main() {

int num\_requests;

int initial\_head;

printf("Enter the number of disk requests: ");

scanf("%d", &num\_requests);

int requests[num\_requests];

printf("Enter the initial head position: ");

scanf("%d", &initial\_head);

printf("Enter the disk requests: ");

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

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

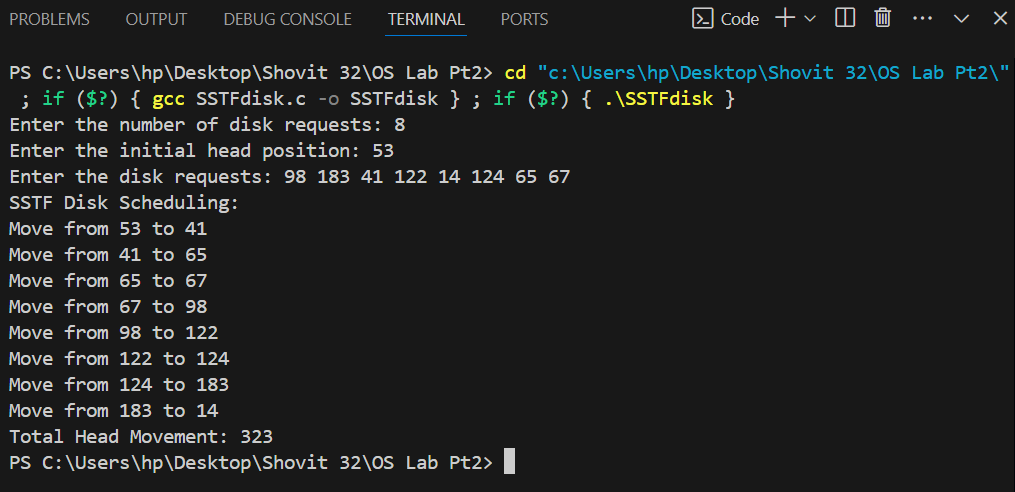
}

SSTF(requests, num\_requests, initial\_head);

return 0;

}

**Output:**

****

## WAP to simulate SCAN Disk Scheduling Algorithm

**Theory:**

The SCAN Disk Scheduling Algorithm moves the disk arm back and forth across the disk, servicing requests in one direction until it reaches the end, then reverses and services requests in the opposite direction. This "elevator" approach reduces seek time compared to FIFO and ensures more equitable servicing of requests.

**Code:**

#include<stdio.h>

int absoluteValue(int x)n{

if(x>0)

{

return x;

}

else

{

return x\*-1;

} }

void main() {

int queue[25],n,headposition,i,j,k,seek=0, maxrange,

difference,temp,queue1[20],queue2[20],temp1=0,temp2=0;

float averageSeekTime;

printf("Enter the maximum range of Disk: ");

scanf("%d",&maxrange);

printf("Enter the number of queue requests: ");

scanf("%d",&n);

printf("Enter the initial head position: ");

scanf("%d",&headposition);

printf("Enter the disk positions to be read(queue): ");

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

scanf("%d",&temp);

if(temp>headposition) {

queue1[temp1]=temp;

temp1++;

}

else {

queue2[temp2]=temp;

temp2++;

}

}

for(i=0;i<temp1-1;i++)

{

for(j=i+1;j<temp1;j++)

{

if(queue1[i]>queue1[j])

{

temp=queue1[i];

queue1[i]=queue1[j];

queue1[j]=temp;

}}

}

for(i=0;i<temp2-1;i++)

{

for(j=i+1;j<temp2;j++)

{

if(queue2[i]<queue2[j])

{

temp=queue2[i];

queue2[i]=queue2[j];

queue2[j]=temp;

}

}

}

for(i=1,j=0;j<temp1;i++,j++)

{

queue[i]=queue1[j];

}

queue[i]=maxrange;

for(i=temp1+2,j=0;j<temp2;i++,j++)

{

queue[i]=queue2[j];

}

queue[i]=0;

queue[0]=headposition;

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

{

difference = absoluteValue(queue[j+1]-queue[j]);

seek = seek + difference;

printf("Disk head moves from position %d to %d with Seek %d \n", queue[j], queue[j+1], difference);

}

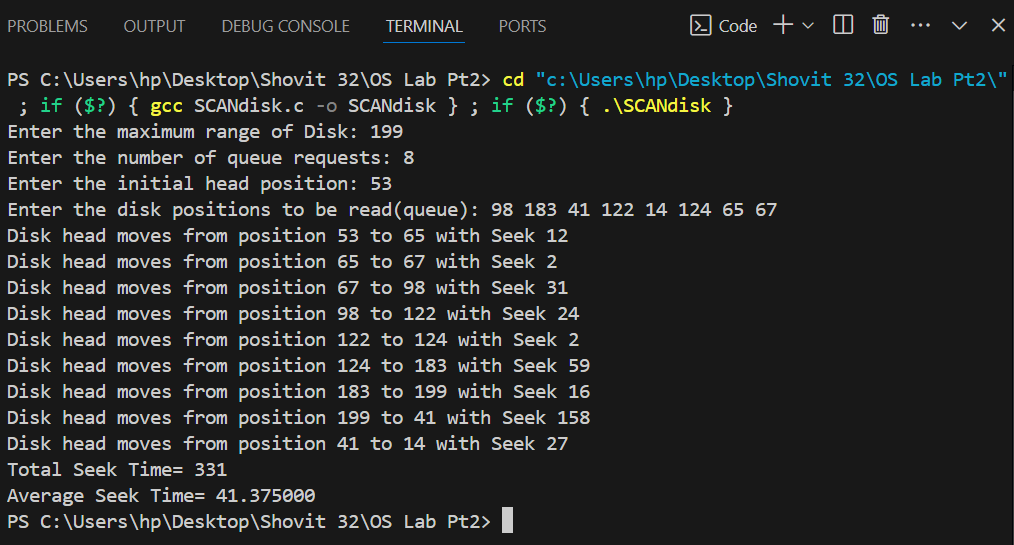
averageSeekTime = seek/(float)n;

printf("Total Seek Time= %d\n", seek);

printf("Average Seek Time= %f\n", averageSeekTime);

}

**Output:**

****

## WAP to simulate C-SCAN Disk Scheduling Algorithm

**Theory:**

C-SCAN Disk Scheduling Algorithm moves the disk arm in one direction, servicing requests until the end, then jumps back to the start without servicing on the return. It provides uniform wait times and reduces starvation by treating all requests fairly in a circular motion.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define size 8

int disk\_size = 200;

void CSCAN(int arr[], int head) {

int seek\_count = 0;

int distance, cur\_track;

int left[100], right[100];

int seek\_sequence[100];

int left\_size = 0, right\_size = 0, seek\_size = 0;

left[left\_size++] = 0;

right[right\_size++] = disk\_size - 1;

int i,j;

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

if (arr[i] < head)

left[left\_size++] = arr[i];

if (arr[i] > head)

right[right\_size++] = arr[i];

}

for ( i = 0; i < left\_size - 1; i++) {

for ( j = i + 1; j < left\_size; j++) {

if (left[i] > left[j]) {

int temp = left[i];

left[i] = left[j];

left[j] = temp;

}}}

for ( i = 0; i < right\_size - 1; i++) {

for ( j = i + 1; j < right\_size; j++) {

if (right[i] > right[j]) {

int temp = right[i];

right[i] = right[j];

right[j] = temp;

} } }

for ( i = 0; i < right\_size; i++) {

cur\_track = right[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

head = 0;

seek\_count += (disk\_size - 1);

for ( i = 0; i < left\_size; i++) {

cur\_track = left[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

printf("Total number of seek operations = %d\n", seek\_count);

printf("Seek Sequence is:\n");

for ( i = 0; i < seek\_size; i++) {

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

}}

int main() {

int arr[size] = { 98, 183, 41, 122, 14, 124, 65, 67};

int head = 53;

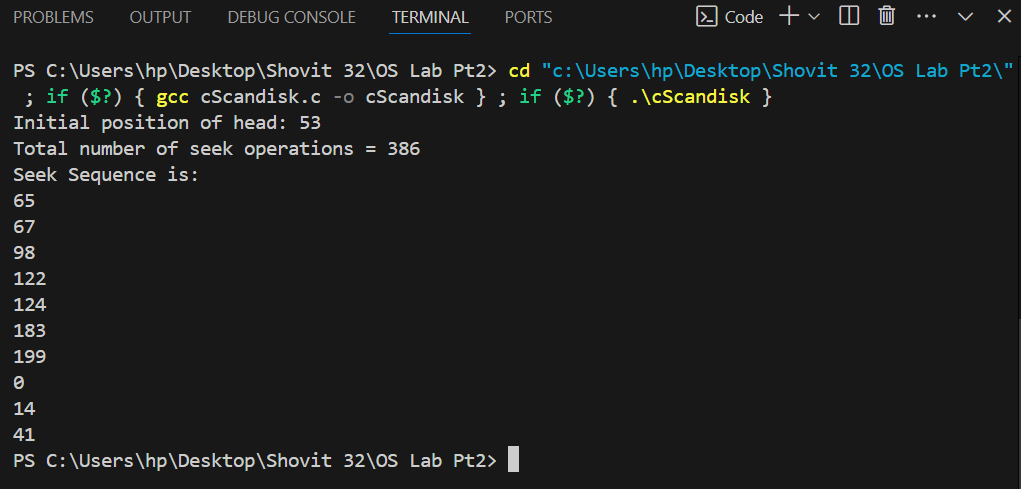
printf("Initial position of head: %d\n", head);

CSCAN(arr, head);

return 0;

}

**Output:**



## WAP to simulate LOOK Disk Scheduling Algorithm

**Theory:**

The LOOK Disk Scheduling Algorithm moves the disk arm towards the nearest end where requests exist, servicing requests along the way. When it reaches the last request in that direction, it reverses and continues servicing in the opposite direction. Unlike SCAN, it only goes as far as the last request, reducing unnecessary movement and optimizing seek time compared to algorithms that scan to the disk's edges.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define size 8

int disk\_size = 200;

void LOOK(int arr[], int head, char direction[]) {

int seek\_count = 0;

int distance, cur\_track;

int left[100], right[100];

int seek\_sequence[100];

int left\_size = 0, right\_size = 0, seek\_size = 0;

int i, j, run = 2;

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

if (arr[i] < head)

left[left\_size++] = arr[i];

if (arr[i] > head)

right[right\_size++] = arr[i];

}

for (i = 0; i < left\_size - 1; i++) {

for (j = i + 1; j < left\_size; j++) {

if (left[i] > left[j]) {

int temp = left[i];

left[i] = left[j];

left[j] = temp;

} } }

for (i = 0; i < right\_size - 1; i++) {

for (j = i + 1; j < right\_size; j++) {

if (right[i] > right[j]) {

int temp = right[i];

right[i] = right[j];

right[j] = temp;

} } }

while (run--) {

if (strcmp(direction, "left") == 0) {

for (i = left\_size - 1; i >= 0; i--) {

cur\_track = left[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

strcpy(direction, "right");

} else if (strcmp(direction, "right") == 0) {

for (i = 0; i < right\_size; i++) {

cur\_track = right[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

strcpy(direction, "left");

} }

printf("Total number of seek operations = %d\n", seek\_count);

printf("Seek Sequence is:\n");

for (i = 0; i < seek\_size; i++) {

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

} }

int main()

{

int arr[size] = { 98, 183, 41, 122, 14, 124, 65, 67};

int head = 53;

char direction[] = "right";

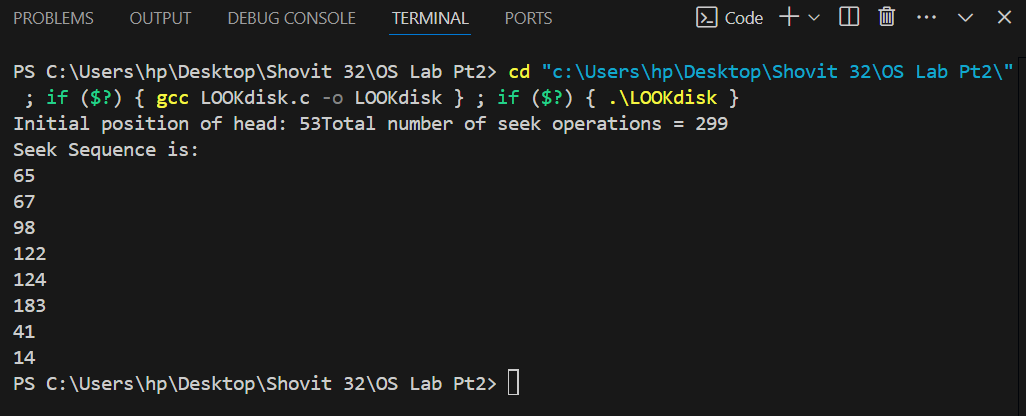
printf("Initial position of head: %d\n", head);

LOOK(arr, head, direction);

return 0;

}

**Output:**

**

## WAPto simulate C-LOOK Disk Scheduling Algorithm

**Theory:**

C-LOOK Disk Scheduling Algorithm moves the disk arm towards the end of the queue, servicing requests along the way. Upon reaching the farthest request, it jumps back to the first request in the queue without servicing in the reverse direction, minimizing movement and improving efficiency over standard LOOK.

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define size 8

int disk\_size = 200;

void CLOOK(int arr[], int head) {

int seek\_count = 0;

int distance, cur\_track;

int left[100], right[100];

int seek\_sequence[100];

int left\_size = 0, right\_size = 0, seek\_size = 0;

int i, j;

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

if (arr[i] < head)

left[left\_size++] = arr[i];

if (arr[i] > head)

right[right\_size++] = arr[i];

}

for (i = 0; i < left\_size - 1; i++) {

for (j = i + 1; j < left\_size; j++) {

if (left[i] > left[j]) {

int temp = left[i];

left[i] = left[j];

left[j] = temp;

}} }

for (i = 0; i < right\_size - 1; i++) {

for (j = i + 1; j < right\_size; j++) {

if (right[i] > right[j]) {

int temp = right[i];

right[i] = right[j];

right[j] = temp;

} } }

for (i = 0; i < right\_size; i++) {

cur\_track = right[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

seek\_count += abs(head - left[0]);

head = left[0];

for (i = 0; i < left\_size; i++) {

cur\_track = left[i];

seek\_sequence[seek\_size++] = cur\_track;

distance = abs(cur\_track - head);

seek\_count += distance;

head = cur\_track;

}

printf("Total number of seek operations = %d\n", seek\_count);

printf("Seek Sequence is:\n");

for (i = 0; i < seek\_size; i++) {

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

}}

int main()

{

int arr[size] = {98, 183, 41, 122, 14, 124, 65, 67};

int head = 53;

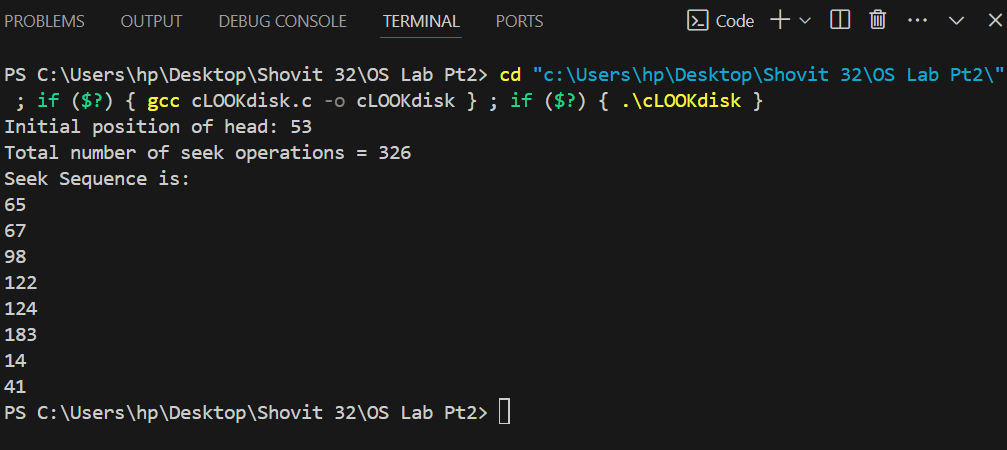
printf("Initial position of head: %d\n", head);

CLOOK(arr, head);

return 0;

}

**Output:**

****