1. Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program

Program:

#include<stdio.h>

#include<unistd.h>

int main(){

printf("process id %d\n",getpid());

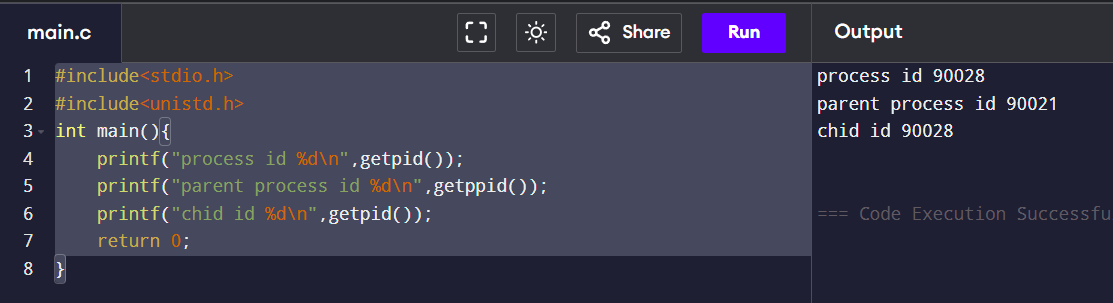
printf("parent process id %d\n",getppid());

printf("chid id %d\n",getpid());

return 0;

}

Output:



1. Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

Program:

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <stdlib.h>

#define BUFFER\_SIZE 1024

int main() {

int sourceFile, destFile, bytesRead, bytesWritten;

char buffer[BUFFER\_SIZE];

sourceFile = open("source.txt", O\_RDONLY);

if (sourceFile == -1) {

perror("Error opening source file");

exit(EXIT\_FAILURE);

}

destFile = open("destination.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (destFile == -1) {

perror("Error opening destination file");

close(sourceFile); // Don't forget to close the source file

exit(EXIT\_FAILURE);

}

while ((bytesRead = read(sourceFile, buffer, BUFFER\_SIZE)) > 0) {

bytesWritten = write(destFile, buffer, bytesRead);

if (bytesWritten != bytesRead) {

perror("Error writing to destination file");

close(sourceFile);

close(destFile);

exit(EXIT\_FAILURE);

}

}

if (bytesRead == -1) {

perror("Error reading from source file");

}

close(sourceFile);

close(destFile);

printf("File copied successfully.\n");

return 0;

}

Ouput: 

3) Design a CPU scheduling program with C using First Come First Served technique with the following considerations.

a. All processes are activated at time 0.

b. Assume that no process waits on I/O devices

Program:

#include <stdio.h>

#define MAX\_PROCESSES 20

int main() {

int n, burstTime[MAX\_PROCESSES], waitingTime[MAX\_PROCESSES], turnaroundTime[MAX\_PROCESSES];

int totalWaitingTime = 0, totalTurnaroundTime = 0;

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

scanf("%d", &n);

printf("Enter Burst Times for each process:\n");

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

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

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

}

waitingTime[0] = 0; // Waiting time for the first process is zero

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

waitingTime[i] = waitingTime[i - 1] + burstTime[i - 1];

}

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

turnaroundTime[i] = burstTime[i] + waitingTime[i];

totalWaitingTime += waitingTime[i];

totalTurnaroundTime += turnaroundTime[i];

}

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

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

printf("P[%d]\t%d\t\t%d\t\t%d\n", i + 1, burstTime[i], waitingTime[i], turnaroundTime[i]);

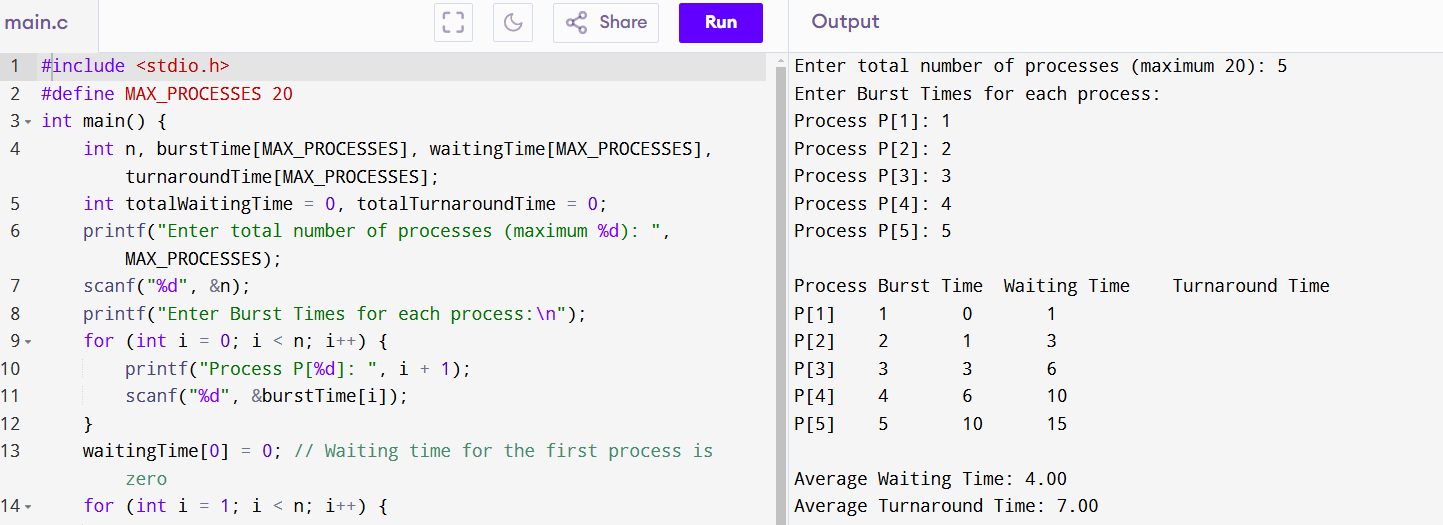
}

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

printf("\nAverage Turnaround Time: %.2f\n", (float)totalTurnaroundTime / n);

return 0;

}

Output: 

4) Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next

Program:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 10

struct Process {

int pid;

int burst\_time;

int completion\_time;

};

void sortByBurstTime(struct Process processes[], int n) {

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

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

if (processes[i].burst\_time > processes[j].burst\_time) {

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

void sjfScheduling(struct Process processes[], int n) {

int total\_time = 0;

sortByBurstTime(processes, n);

printf("Process Scheduling Order (Shortest Job First):\n");

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

processes[i].completion\_time = total\_time + processes[i].burst\_time;

total\_time = processes[i].completion\_time;

printf("Process %d: Start Time = %d, Burst Time = %d, Completion Time = %d\n",

processes[i].pid, total\_time - processes[i].burst\_time,

processes[i].burst\_time, processes[i].completion\_time);

}

}

int main() {

struct Process processes[MAX\_PROCESSES];

int n;

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

scanf("%d", &n);

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

printf("Enter Process ID and Burst Time for process %d: ", i + 1);

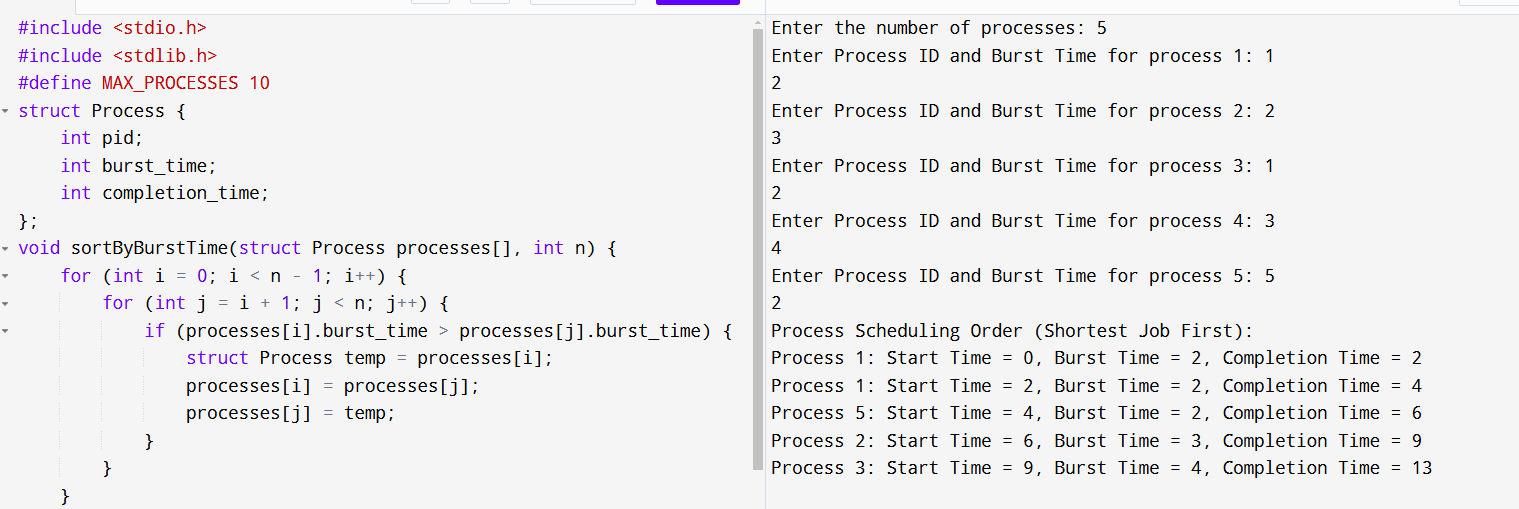
scanf("%d %d", &processes[i].pid, &processes[i].burst\_time);

}

sjfScheduling(processes, n);

return 0;

}

Output: 

5) Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

Program:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PROCESSES 10

struct Process {

int pid;

int burst\_time;

int priority;

int completion\_time;

};

void sortByPriority(struct Process processes[], int n) {

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

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

if (processes[i].priority > processes[j].priority) {

// Swap the processes

struct Process temp = processes[i];

processes[i] = processes[j];

processes[j] = temp;

}

}

}

}

void hpfScheduling(struct Process processes[], int n) {

int total\_time = 0; // To track the current time

sortByPriority(processes, n);

printf("Process Scheduling Order (Highest Priority First):\n");

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

processes[i].completion\_time = total\_time + processes[i].burst\_time;

total\_time = processes[i].completion\_time;

printf("Process %d: Start Time = %d, Burst Time = %d, Priority = %d, Completion Time = %d\n",

processes[i].pid, total\_time - processes[i].burst\_time,

processes[i].burst\_time, processes[i].priority, processes[i].completion\_time);

}

}

int main() {

struct Process processes[MAX\_PROCESSES];

int n;

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

scanf("%d", &n);

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

printf("Enter Process ID, Burst Time, and Priority for process %d: ", i + 1);

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

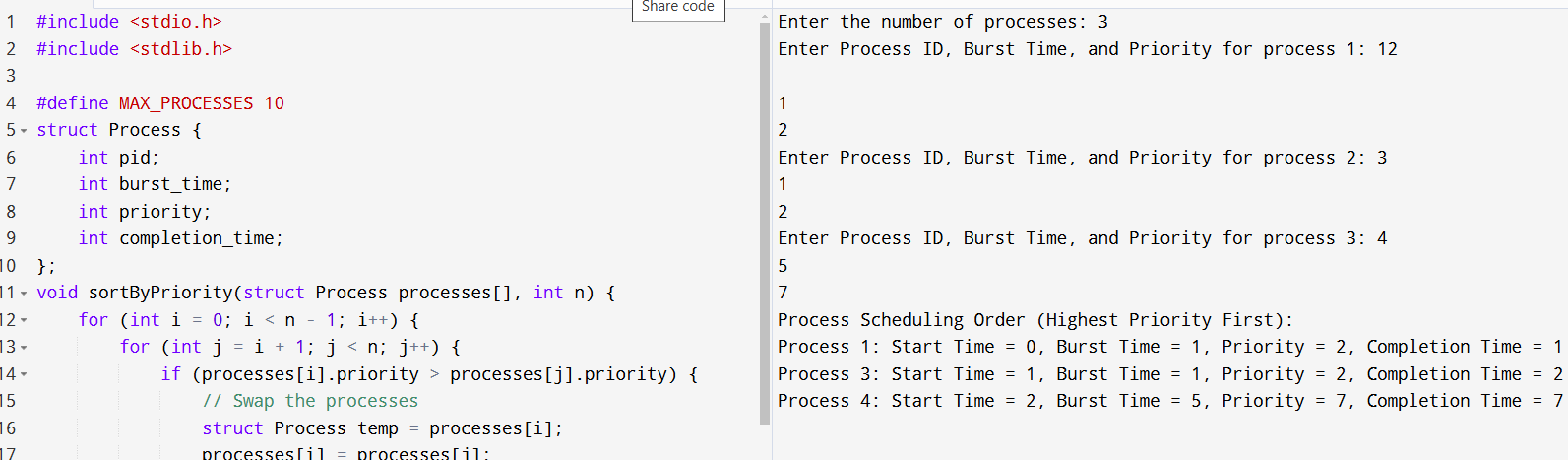
}

hpfScheduling(processes, n);

return 0;

}

Output:



6) Construct a C program to implement preemptive priority scheduling algorithm

Program:

Program:

#include <stdio.h>

#define MAX 10

struct Process {

int pid; // Process ID

int bt; // Burst time

int art; // Arrival time

int pr; // Priority

int wt; // Waiting time

int tat; // Turnaround time

int rem\_bt; // Remaining burst time (for preemption)

};

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

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

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

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

}

}

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

struct Process temp;

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

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

// Sort first by arrival time, then by priority

if (proc[i].art > proc[j].art || (proc[i].art == proc[j].art && proc[i].pr < proc[j].pr)) {

temp = proc[i];

proc[i] = proc[j];

proc[j] = temp;

}

}

}

}

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

int total\_time = 0;

int completed = 0;

int current\_time = 0;

int min\_priority = -1;

int index = -1;

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

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

}

while (completed < n) {

min\_priority = 9999;

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

if (proc[i].art <= current\_time && proc[i].rem\_bt > 0 && proc[i].pr < min\_priority) {

min\_priority = proc[i].pr;

index = i;

}

}

if (index == -1) {

current\_time++;

continue;

}

proc[index].rem\_bt--;

if (proc[index].rem\_bt == 0) {

completed++;

proc[index].wt = current\_time - proc[index].art - proc[index].bt;

proc[index].tat = proc[index].wt + proc[index].bt;

}

current\_time++;

}

calculateWTAndTAT(proc, n);

}

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

printf("\nProcess ID | Arrival Time | Burst Time | Priority | Waiting Time | Turnaround Time\n");

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

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

printf("%9d | %12d | %10d | %8d | %12d | %15d\n", proc[i].pid, proc[i].art, proc[i].bt, proc[i].pr, proc[i].wt, proc[i].tat);

}

}

int main() {

struct Process proc[MAX];

int n;

printf("Enter number of processes: ");

scanf("%d", &n);

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

proc[i].pid = i + 1;

printf("\nEnter details for Process %d:\n", proc[i].pid);

printf("Arrival Time: ");

scanf("%d", &proc[i].art);

printf("Burst Time: ");

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

printf("Priority: ");

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

}

sortProcesses(proc, n);

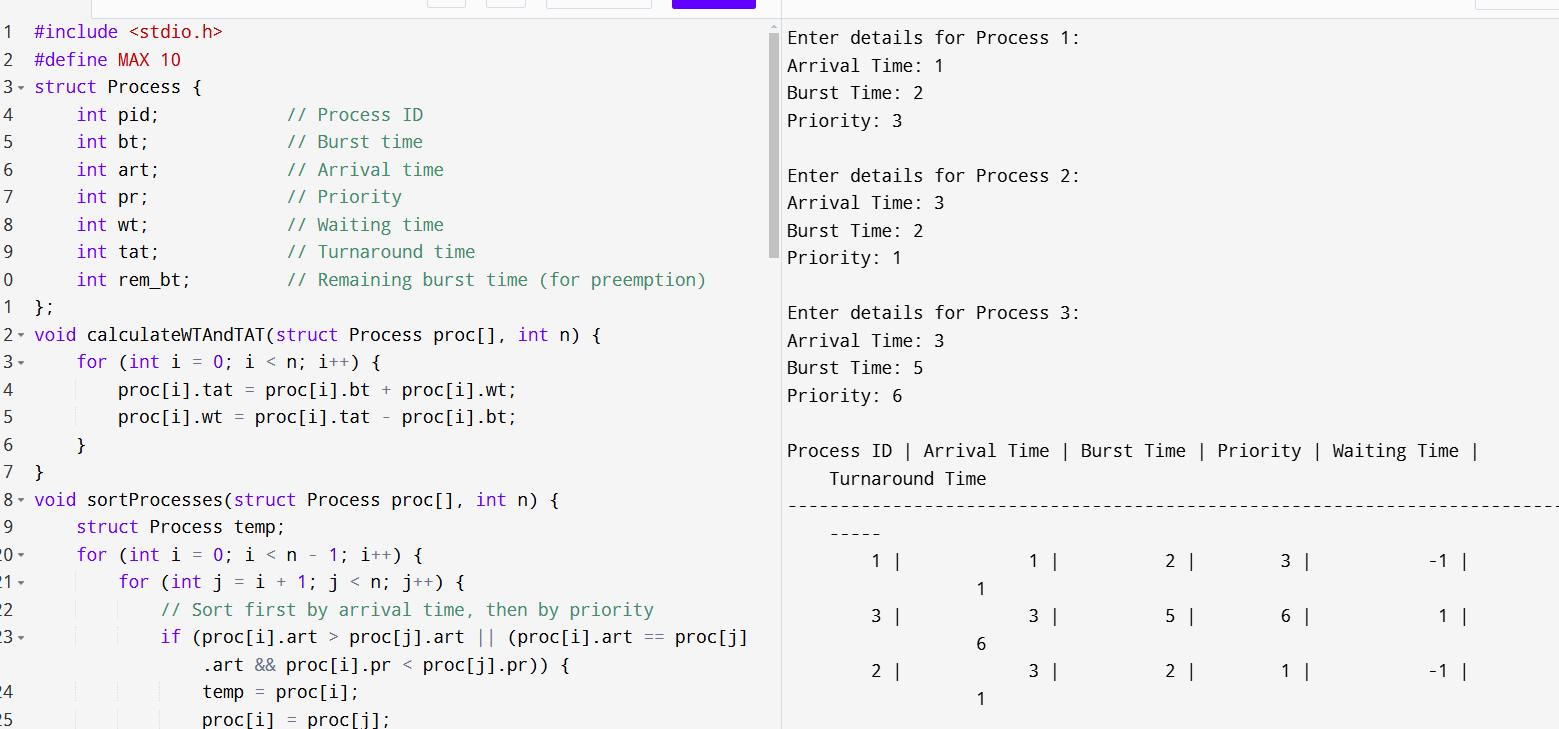
preemptivePriorityScheduling(proc, n);

displayResults(proc, n);

return 0;

}

Output:



7) Construct a C program to implement a non-preemptive SJF algorithm.

PROGRAM:

#include <stdio.h>

int main() {

int at[10], bt[10], pr[10];

int n, i, j, temp, time = 0, count, over = 0, sum\_wait = 0, sum\_turnaround = 0, start;

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

scanf("%d", &n);

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

printf("Enter the arrival time and execution time for process %d\n", i + 1);

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

pr[i] = i + 1;

}

// Sort processes based on arrival time

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

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

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

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[i];

bt[i] = bt[j];

bt[j] = temp;

temp = pr[i];

pr[i] = pr[j];

pr[j] = temp;

}

}

}

printf("Pr\t|Arri tim\t|Exeti\t|Sttime\t|Endtime\t|waitime\t|TAT\n\n");

while (over < n) {

count = 0;

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

if (at[i] <= time)

count++;

else

break;

}

if (count > 1) {

// Sort processes based on burst time

for (i = over; i < over + count - 1; i++) {

for (j = i + 1; j < over + count; j++) {

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

temp = at[i];

at[i] = at[j];

at[j] = temp;

temp = bt[i];

bt[i] = bt[j];

bt[j] = temp;

temp = pr[i];

pr[i] = pr[j];

pr[j] = temp;

}

}

}

}

start = time;

time += bt[over];

printf("p[%d]\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\t|\t%d\n", pr[over],

at[over], bt[over], start, time, time - at[over] - bt[over], time - at[over]);

sum\_wait += time - at[over] - bt[over];

sum\_turnaround += time - at[over];

over++;

}

float avgwait = (float)sum\_wait / (float)n;

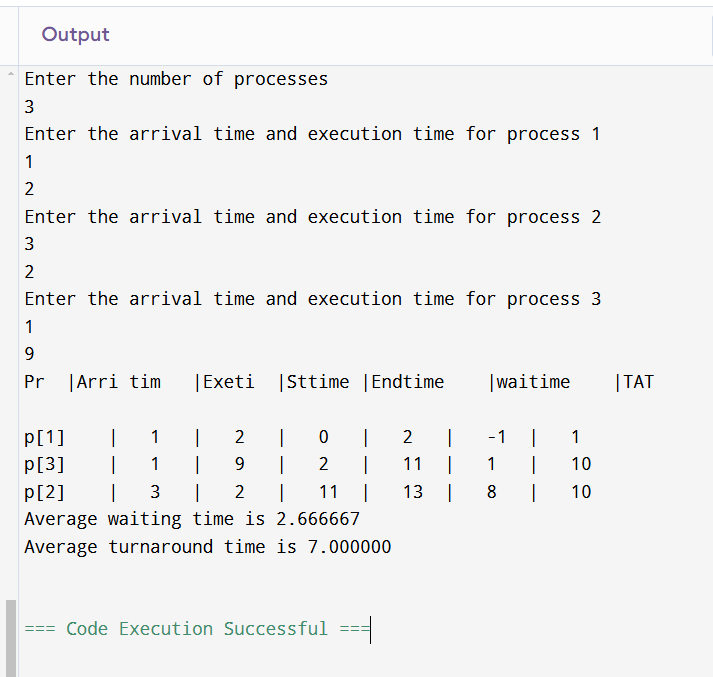
float avgturn = (float)sum\_turnaround / (float)n;

printf("Average waiting time is %f\n", avgwait);

printf("Average turnaround time is %f\n", avgturn);

return 0;

}

OUTPUT: 

8) Construct a C program to simulate Round Robin scheduling algorithm with C.

PROGRAM:

#include <stdio.h>

#define MAX 10

struct Process {

int pid; // Process ID

int bt; // Burst Time

int wt; // Waiting Time

int tat; // Turnaround Time

int rem\_bt; // Remaining Burst Time (used for round robin)

};

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

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

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

}

}

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

int completed = 0;

int current\_time = 0;

while (completed < n) {

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

if (proc[i].rem\_bt > 0) {

if (proc[i].rem\_bt <= quantum) {

current\_time += proc[i].rem\_bt;

proc[i].wt = current\_time - proc[i].bt;

proc[i].rem\_bt = 0;

completed++;

}

else {

current\_time += quantum;

proc[i].rem\_bt -= quantum;

}

}

}

}

calculateWTAndTAT(proc, n);

}

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

printf("\nProcess ID | Burst Time | Waiting Time | Turnaround Time\n");

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

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

printf("%9d | %10d | %12d | %15d\n", proc[i].pid, proc[i].bt, proc[i].wt, proc[i].tat);

}

}

int main() {

struct Process proc[MAX];

int n, quantum;

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter time quantum: ");

scanf("%d", &quantum);

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

proc[i].pid = i + 1;

printf("\nEnter burst time for Process %d: ", proc[i].pid);

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

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

}

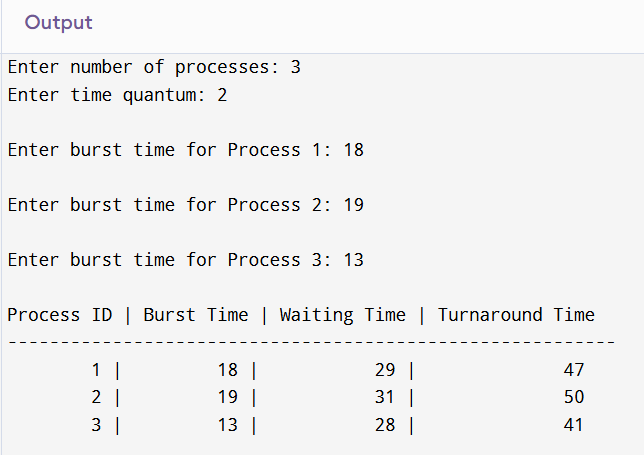
roundRobin(proc, n, quantum);

displayResults(proc, n);

return 0;

}

OUTPUT:



9) Illustrate the concept of inter-process communication using shared memory with a C program.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#define SHM\_SIZE 1024

int main() {

int shmid = shmget(IPC\_PRIVATE, SHM\_SIZE, IPC\_CREAT | 0666);

if (shmid == -1) {

perror("shmget");

exit(EXIT\_FAILURE);

}

char\* shm\_ptr = (char\*)shmat(shmid, NULL, 0);

if (shm\_ptr == (char\*)(-1)) {

perror("shmat");

exit(EXIT\_FAILURE);

}

strcpy(shm\_ptr, "Hello, shared memory!");

if (shmdt(shm\_ptr) == -1) {

perror("shmdt");

exit(EXIT\_FAILURE);

}

shm\_ptr = (char\*)shmat(shmid, NULL, 0);

if (shm\_ptr == (char\*)(-1)) {

perror("shmat");

exit(EXIT\_FAILURE);

}

printf("Data written to shared memory: %s\n", shm\_ptr);

if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl");

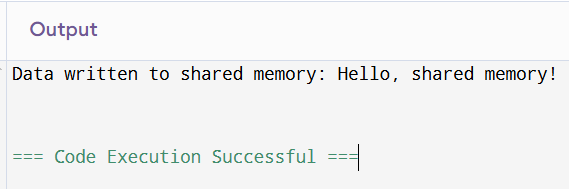
exit(EXIT\_FAILURE);

}

return 0;

}

OUTPUT:



10) Illustrate the concept of inter-process communication using message queue with a C program

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

#include <unistd.h>

#define MSG\_SIZE 100

// Define the message structure

struct msg\_buffer {

long msg\_type;

char msg\_text[MSG\_SIZE];

};

int main() {

key\_t key;

int msgid;

struct msg\_buffer message;

// Generate a unique key for the message queue

key = ftok("msg\_queue\_file", 65);

// Create a message queue and get its ID

msgid = msgget(key, 0666 | IPC\_CREAT);

if (msgid == -1) {

perror("Message queue creation failed");

exit(1);

}

// Send 5 messages to the queue

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

message.msg\_type = 1; // Message type is set to 1 (can be used for prioritization)

snprintf(message.msg\_text, MSG\_SIZE, "Hello from sender! Message %d", i + 1);

// Send the message to the queue

if (msgsnd(msgid, &message, sizeof(message.msg\_text), 0) == -1) {

perror("Message sending failed");

exit(1);

}

printf("Sent: %s\n", message.msg\_text);

sleep(1); // Sleep for a second to simulate some delay

}

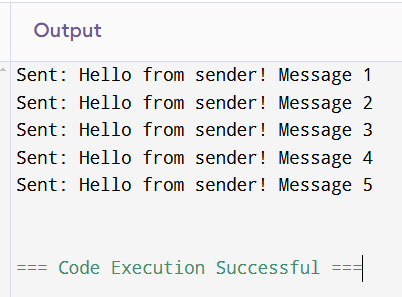
// Optionally: Remove the message queue (if you want to delete it after usage)

// msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

OUTPUT:



11) Illustrate the concept of multithreading using a C program.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

// Function to be executed by the first thread

void\* print\_numbers(void\* arg) {

for (int i = 1; i <= 5; i++) {

printf("Thread 1: %d\n", i);

sleep(1); // Simulate work

}

return NULL;

}

// Function to be executed by the second thread

void\* print\_letters(void\* arg) {

for (char c = 'A'; c <= 'E'; c++) {

printf("Thread 2: %c\n", c);

sleep(1); // Simulate work

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

// Create the first thread

if (pthread\_create(&thread1, NULL, print\_numbers, NULL) != 0) {

perror("Failed to create thread 1");

exit(EXIT\_FAILURE);

}

// Create the second thread

if (pthread\_create(&thread2, NULL, print\_letters, NULL) != 0) {

perror("Failed to create thread 2");

exit(EXIT\_FAILURE);

}

// Wait for the threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

printf("Main thread: Both threads have finished execution.\n");

return 0;

}

OUTPUT:



12) Design a C program to simulate the concept of Dining-Philosophers problem CO3

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

pthread\_mutex\_t forks[NUM\_PHILOSOPHERS];

void \*philosopher(void \*num) {

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

int left\_fork = id;

int right\_fork = (id + 1) % NUM\_PHILOSOPHERS;

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

sleep(1);

pthread\_mutex\_lock(&forks[left\_fork]);

pthread\_mutex\_lock(&forks[right\_fork]);

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

sleep(1);

pthread\_mutex\_unlock(&forks[left\_fork]);

pthread\_mutex\_unlock(&forks[right\_fork]);

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

return NULL;

}

int main() {

pthread\_t threads[NUM\_PHILOSOPHERS];

int ids[NUM\_PHILOSOPHERS];

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

pthread\_mutex\_init(&forks[i], NULL);

}

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

ids[i] = i;

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

}

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

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

}

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

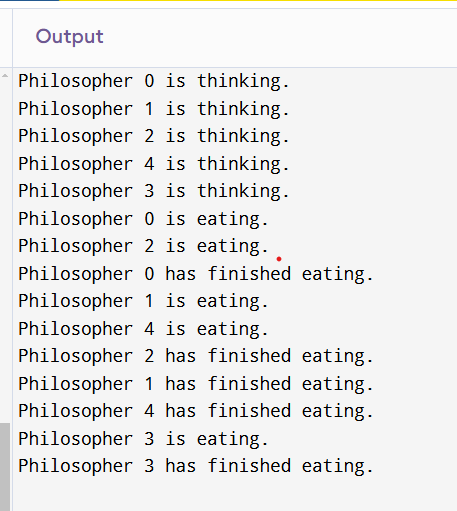
pthread\_mutex\_destroy(&forks[i]);

}

return 0;

}

OUTPUT:



13) Construct a C program for implementation of the various memory allocation strategies.

PROGRAM:

#include<stdio.h>

void bestfit(int mp[],int p[],int m,int n)

{

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

if(mp[i]>p[j]){

printf("\n%d fits in %d",p[j],mp[i]);

mp[i]=mp[i]-p[j++];

i=i-1;

}

}

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

{

printf("\n%d must wait for its process",p[i]);

}

}

void rsort(int a[],int n){

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

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

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

int t=a[i];

a[i]=a[j];

a[j]=t;

}

}

}

}

void sort(int a[],int n){

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

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

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

int t=a[i];

a[i]=a[j];

a[j]=t;

}

}

}

}

void firstfit(int mp[],int p[],int m,int n){

sort(mp,n);

sort(p,m);

bestfit(mp,p,m,n);

}

void worstfit(int mp[],int p[],int m,int n){

rsort(mp,n);

sort(p,m);

bestfit(mp,p,m,n);

}

int main(){

int m,n,mp[20],p[20],ch;

printf("Number of memory partition : ");

scanf("%d",&n);

printf("Number of process : ");

scanf("%d",&m);

printf("Enter the memory partitions : \n");

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

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

}

printf("ENter process size : \n");

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

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

}

printf("1. Firstfit\t2. Bestfit\t3. worstfit\nEnter your choice : ");

scanf("%d",&ch);

switch(ch){

case 1:

bestfit(mp,p,m,n);

break;

case 2:

firstfit(mp,p,m,n);

break;

case 3:

worstfit(mp,p,m,n);

break;

default:

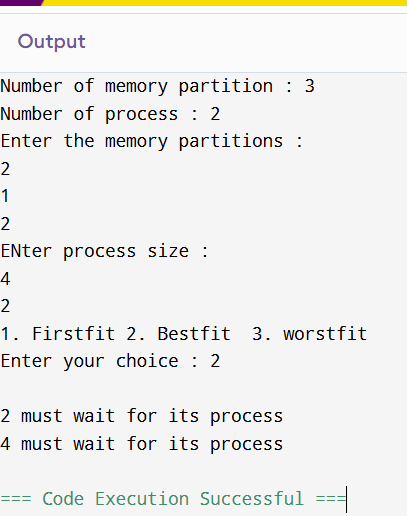
printf("invalid");

break;

}

}

OUTPUT:



14) .Construct a C program to organize the file using single leveldirectory

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_FILES 100

#define MAX\_NAME\_LEN 50

typedef struct {

char name[MAX\_NAME\_LEN];

} File;

File directory[MAX\_FILES];

int file\_count = 0;

void create\_file(const char\* filename) {

if (file\_count >= MAX\_FILES) {

printf("Error: Directory is full. Cannot create more files.\n");

return;

}

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

if (strcmp(directory[i].name, filename) == 0) {

printf("Error: File '%s' already exists.\n", filename);

return;

}

}

strcpy(directory[file\_count].name, filename);

file\_count++;

printf("File '%s' created successfully.\n", filename);

}

void delete\_file(const char\* filename) {

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

if (strcmp(directory[i].name, filename) == 0) {

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

directory[j] = directory[j + 1];

}

file\_count--;

printf("File '%s' deleted successfully.\n", filename);

return;

}

}

printf("Error: File '%s' not found.\n", filename);

}

void list\_files() {

if (file\_count == 0) {

printf("Directory is empty.\n");

return;

}

printf("Files in directory:\n");

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

printf("%d. %s\n", i + 1, directory[i].name);

}

}

int main() {

int choice;

char filename[MAX\_NAME\_LEN];

while (1) {

printf("\nSingle-Level Directory:\n");

printf("1. Create File\n");

printf("2. Delete File\n");

printf("3. List Files\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the name of the file to create: ");

scanf("%s", filename);

create\_file(filename);

break;

case 2:

printf("Enter the name of the file to delete: ");

scanf("%s", filename);

delete\_file(filename);

break;

case 3:

list\_files();

break;

case 4:

printf("Exiting program.\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

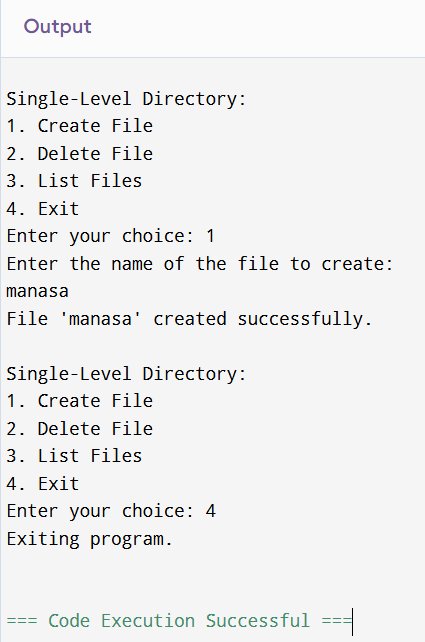
}

}

return 0;

}

OUTPUT:



15) .Design a C program to organize the file using two level directorystructure.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_USERS 10

#define MAX\_FILES 10

#define MAX\_NAME\_LEN 50

typedef struct {

char name[MAX\_NAME\_LEN];

} File;

typedef struct {

char name[MAX\_NAME\_LEN];

File files[MAX\_FILES];

int file\_count;

} Directory;

Directory users[MAX\_USERS];

int user\_count = 0;

void create\_user(const char\* username) {

if (user\_count >= MAX\_USERS) {

printf("Error: Maximum user limit reached. Cannot create more directories.\n");

return;

}

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

if (strcmp(users[i].name, username) == 0) {

printf("Error: User '%s' already exists.\n", username);

return;

}

}

strcpy(users[user\_count].name, username);

users[user\_count].file\_count = 0;

user\_count++;

printf("User '%s' created successfully.\n", username);

}

void create\_file(const char\* username, const char\* filename) {

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

if (strcmp(users[i].name, username) == 0) {

if (users[i].file\_count >= MAX\_FILES) {

printf("Error: File limit reached for user '%s'.\n", username);

return;

}

for (int j = 0; j < users[i].file\_count; j++) {

if (strcmp(users[i].files[j].name, filename) == 0) {

printf("Error: File '%s' already exists under user '%s'.\n", filename, username);

return;

}

}

strcpy(users[i].files[users[i].file\_count].name, filename);

users[i].file\_count++;

printf("File '%s' created under user '%s'.\n", filename, username);

return;

}

}

printf("Error: User '%s' not found.\n", username);

}

void delete\_file(const char\* username, const char\* filename) {

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

if (strcmp(users[i].name, username) == 0) {

for (int j = 0; j < users[i].file\_count; j++) {

if (strcmp(users[i].files[j].name, filename) == 0) {

for (int k = j; k < users[i].file\_count - 1; k++) {

users[i].files[k] = users[i].files[k + 1];

}

users[i].file\_count--;

printf("File '%s' deleted under user '%s'.\n", filename, username);

return;

}

}

printf("Error: File '%s' not found under user '%s'.\n", filename, username);

return;

}

}

printf("Error: User '%s' not found.\n", username);

}

void list\_files(const char\* username) {

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

if (strcmp(users[i].name, username) == 0) {

if (users[i].file\_count == 0) {

printf("No files found under user '%s'.\n", username);

return;

}

printf("Files under user '%s':\n", username);

for (int j = 0; j < users[i].file\_count; j++) {

printf("%d. %s\n", j + 1, users[i].files[j].name);

}

return;

}

}

printf("Error: User '%s' not found.\n", username);

}

void list\_users() {

if (user\_count == 0) {

printf("No users found.\n");

return;

}

printf("Users:\n");

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

printf("%d. %s\n", i + 1, users[i].name);

}

}

int main() {

int choice;

char username[MAX\_NAME\_LEN], filename[MAX\_NAME\_LEN];

while (1) {

printf("\nTwo-Level Directory Structure:\n");

printf("1. Create User\n");

printf("2. Create File\n");

printf("3. Delete File\n");

printf("4. List Files for a User\n");

printf("5. List All Users\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the username: ");

scanf("%s", username);

create\_user(username);

break;

case 2:

printf("Enter the username: ");

scanf("%s", username);

printf("Enter the filename: ");

scanf("%s", filename);

create\_file(username, filename);

break;

case 3:

printf("Enter the username: ");

scanf("%s", username);

printf("Enter the filename: ");

scanf("%s", filename);

delete\_file(username, filename);

break;

case 4:

printf("Enter the username: ");

scanf("%s", username);

list\_files(username);

break;

case 5:

list\_users();

break;

case 6:

printf("Exiting program.\n");

exit(0);

default:

printf("Invalid choice. Please try again.\n");

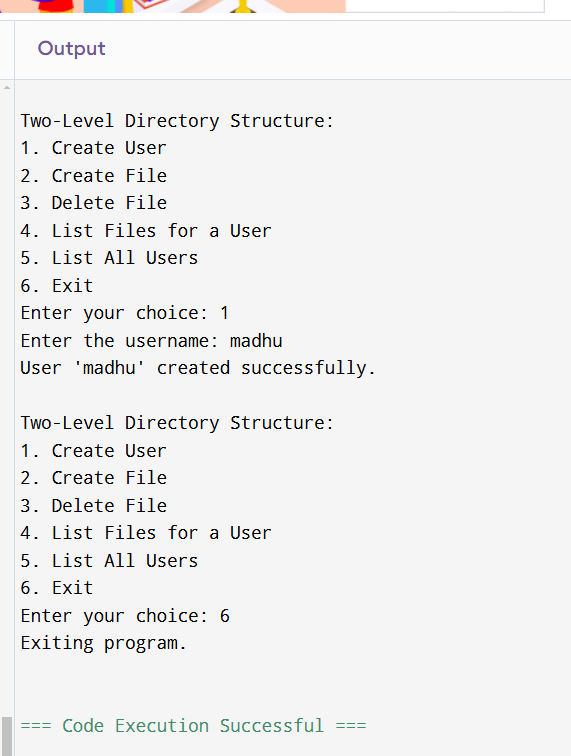
}

}

return 0;

}

OUTPUT:



16) Develop a C program for implementing random access file forprocessing the employee details

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int id;

char name[50];

float salary;

} Employee;

void addEmployee();

void viewEmployee(int id);

int main() {

int choice, id;

while (1) {

printf("\n--- Employee Management ---\n");

printf("1. Add Employee\n");

printf("2. View Employee\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

addEmployee();

break;

case 2:

printf("Enter Employee ID to view: ");

scanf("%d", &id);

viewEmployee(id);

break;

case 3:

exit(0);

default:

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

}

}

return 0;

}

void addEmployee() {

FILE \*file = fopen("employees.dat", "ab");

if (!file) {

perror("Unable to open file");

return;

}

Employee emp;

printf("Enter Employee ID: ");

scanf("%d", &emp.id);

printf("Enter Employee Name: ");

scanf(" %[^\n]", emp.name);

printf("Enter Employee Salary: ");

scanf("%f", &emp.salary);

fwrite(&emp, sizeof(Employee), 1, file);

fclose(file);

printf("Employee added successfully!\n");

}

void viewEmployee(int id) {

FILE \*file = fopen("employees.dat", "rb");

if (!file) {

perror("Unable to open file");

return;

}

Employee emp;

int found = 0;

while (fread(&emp, sizeof(Employee), 1, file)) {

if (emp.id == id) {

found = 1;

printf("\n--- Employee Details ---\n");

printf("ID: %d\n", emp.id);

printf("Name: %s\n", emp.name);

printf("Salary: %.2f\n", emp.salary);

break;

}

}

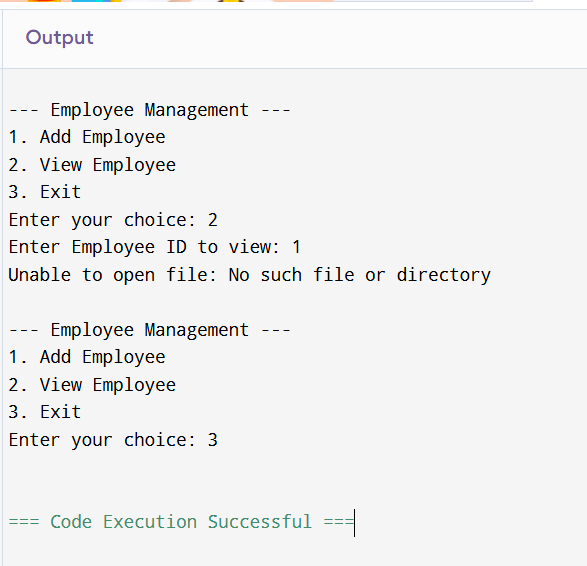
if (!found) {

printf("Employee with ID %d not found.\n", id);

}

fclose(file);

}

OUTPUT: 

17) Illustrate the deadlock avoidance concept by simulating Banker’salgorithm using C

PROGRAM:

#include <stdio.h>

int main()

{

int n, r, i, j, k;

n = 5;

r = 3;

int alloc[5][3] = { { 0, 0, 1 },

{ 3, 0, 0 },

{ 1, 0, 1 },

{ 2, 3, 2 },

{ 0, 0, 3 } };

int max[5][3] = { { 7, 6, 3 },

{ 3, 2, 2 },

{ 8, 0, 2 },

{ 2, 1, 2 },

{ 5, 2, 3 } };

int avail[3] = { 2, 3, 2 };

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

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

f[k] = 0;

}

int need[n][r];

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

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

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

}

int y = 0;

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

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

if (f[i] == 0) {

int flag = 0;

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

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

flag = 1;

break;

}

}

if (flag == 0) {

ans[ind++] = i;

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

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

f[i] = 1;

}

}

}

}

printf("Th SAFE Sequence is as follows\n");

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

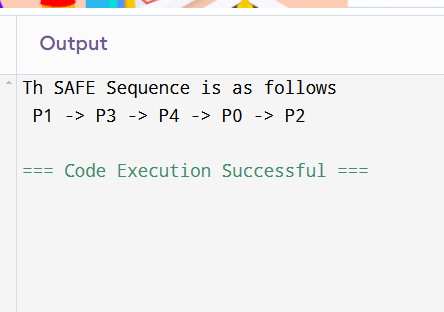
printf(" P%d ->", ans[i]);

printf(" P%d", ans[n - 1]);

return (0);

}

OUTPUT:



18) Construct a C program to simulate producer consumer problemusing semaphores.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void\* producer(void\* param) {

int item;

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

item = rand() % 100;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

sleep(1);

}

return NULL;

}

void\* consumer(void\* param) {

int item;

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

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

sleep(1);

}

return NULL;

}

int main() {

pthread\_t prod, cons;

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

sem\_init(&full, 0, 0);

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

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

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

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

sem\_destroy(&empty);

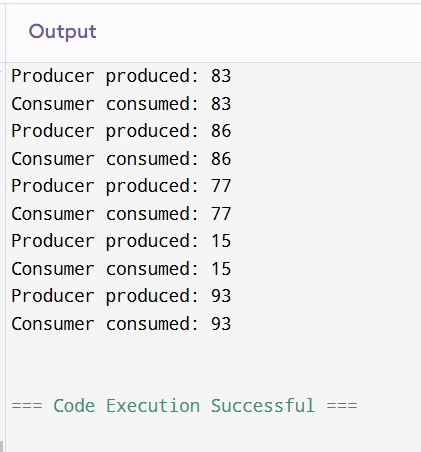
sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

OUTPUT:



19) esign a C program to implement process synchronization usingmutex locks

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#define NUM\_THREADS 5

// Shared resource (counter)

int counter = 0;

// Mutex lock for synchronization

pthread\_mutex\_t counter\_mutex;

// Function to increment the counter

void \*increment\_counter(void \*thread\_id) {

long tid = (long)thread\_id;

// Lock the mutex before modifying the shared resource

pthread\_mutex\_lock(&counter\_mutex);

printf("Thread %ld: Incrementing counter...\n", tid);

// Critical section: modify the shared counter

counter++;

printf("Thread %ld: Counter value = %d\n", tid, counter);

// Unlock the mutex after modifying the shared resource

pthread\_mutex\_unlock(&counter\_mutex);

return NULL;

}

int main() {

pthread\_t threads[NUM\_THREADS];

long i;

// Initialize the mutex lock

if (pthread\_mutex\_init(&counter\_mutex, NULL) != 0) {

printf("Mutex initialization failed\n");

return 1;

}

// Create multiple threads

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

if (pthread\_create(&threads[i], NULL, increment\_counter, (void \*)i) != 0) {

printf("Error creating thread %ld\n", i);

return 1;

}

}

// Wait for all threads to finish

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

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

}

// Destroy the mutex lock after use

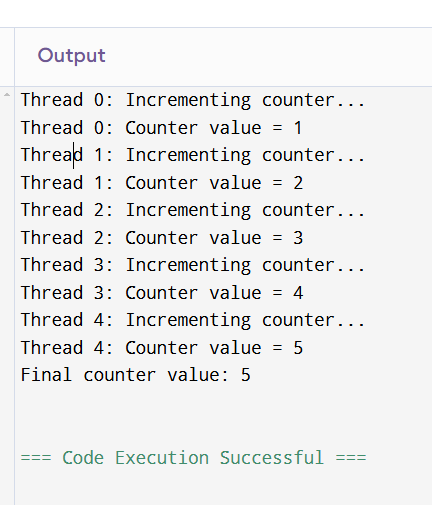
pthread\_mutex\_destroy(&counter\_mutex);

printf("Final counter value: %d\n", counter);

return 0;

}

OUTPUT:



20) Construct a C program to simulate Reader-Writer problem using semaphores

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define NUM\_READERS 5

#define NUM\_WRITERS 3

// Shared resource

int shared\_data = 0;

// Semaphores

sem\_t mutex; // Mutex for critical section

sem\_t write\_lock; // Semaphore for writer

sem\_t read\_lock; // Semaphore for reader count

int read\_count = 0; // Number of readers currently reading

// Reader thread function

void\* reader(void\* arg) {

int reader\_id = \*((int\*)arg);

// Entry section: wait for the read\_lock and mutex

sem\_wait(&read\_lock); // Enter critical section for reader count

sem\_wait(&mutex); // Lock to modify read\_count

read\_count++;

if (read\_count == 1) {

// First reader locks the write\_lock to prevent writers from accessing

sem\_wait(&write\_lock);

}

sem\_post(&mutex); // Unlock the mutex

sem\_post(&read\_lock); // Leave critical section for reader count

// Reading shared data

printf("Reader %d: Reading shared data = %d\n", reader\_id, shared\_data);

usleep(1000); // Simulate reading time

// Exit section: Decrease the reader count and release write lock if no readers are left

sem\_wait(&mutex); // Lock to modify read\_count

read\_count--;

if (read\_count == 0) {

// Last reader releases the write\_lock

sem\_post(&write\_lock);

}

sem\_post(&mutex); // Unlock the mutex

return NULL;

}

// Writer thread function

void\* writer(void\* arg) {

int writer\_id = \*((int\*)arg);

// Entry section: wait for the write\_lock

sem\_wait(&write\_lock);

// Writing to shared data

shared\_data++;

printf("Writer %d: Writing shared data = %d\n", writer\_id, shared\_data);

usleep(1000); // Simulate writing time

// Exit section: release the write\_lock

sem\_post(&write\_lock);

return NULL;

}

int main() {

pthread\_t readers[NUM\_READERS];

pthread\_t writers[NUM\_WRITERS];

int reader\_ids[NUM\_READERS];

int writer\_ids[NUM\_WRITERS];

// Initialize semaphores

sem\_init(&mutex, 0, 1); // Binary semaphore for mutex

sem\_init(&write\_lock, 0, 1); // Binary semaphore for write lock

sem\_init(&read\_lock, 0, 1); // Semaphore to count the number of readers

// Create reader threads

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

reader\_ids[i] = i + 1; // Reader IDs start from 1

pthread\_create(&readers[i], NULL, reader, &reader\_ids[i]);

}

// Create writer threads

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

writer\_ids[i] = i + 1; // Writer IDs start from 1

pthread\_create(&writers[i], NULL, writer, &writer\_ids[i]);

}

// Wait for all threads to finish

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

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

}

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

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

}

// Destroy semaphores

sem\_destroy(&mutex);

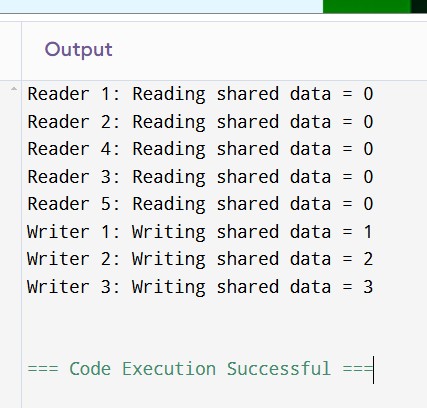
sem\_destroy(&write\_lock);

sem\_destroy(&read\_lock);

return 0;

}

OUTPUT:



21) Develop a C program to implement worst fit algorithm ofmemory management.

PROGRAM:

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

// Function to find the index of the block that has the largest size

int worstFit(int blockSize[], int m, int processSize) {

int idx = -1;

int maxBlockSize = -1;

// Find the block with the largest size that can accommodate the process

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

if (blockSize[i] >= processSize && blockSize[i] > maxBlockSize) {

maxBlockSize = blockSize[i];

idx = i;

}

}

return idx;

}

void worstFitMemoryAllocation(int blockSize[], int m, int processSize[], int n) {

// Array to store which block each process gets allocated to

int allocation[n];

// Initially, no process is allocated

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

allocation[i] = -1; // -1 means no allocation

}

// Allocate blocks to processes

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

int blockIdx = worstFit(blockSize, m, processSize[i]);

if (blockIdx != -1) {

// Process can be allocated to this block

allocation[i] = blockIdx;

// Reduce the size of the block

blockSize[blockIdx] -= processSize[i];

}

}

// Display the results

printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

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

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Define the block sizes and process sizes

int blockSize[MAX\_BLOCKS] = {100, 500, 200, 300, 600};

int processSize[MAX\_PROCESSES] = {212, 417, 112, 426};

// Number of blocks and processes

int m = 5; // Number of blocks

int n = 4; // Number of processes

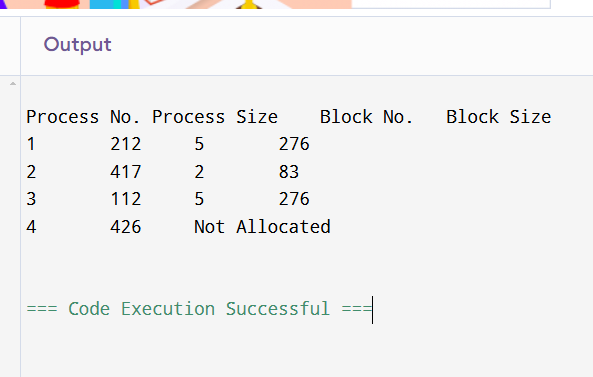
// Call the Worst Fit memory allocation function

worstFitMemoryAllocation(blockSize, m, processSize, n);

return 0;

}

OUTPUT:



22) Construct a C program to implement best fit algorithm ofmemory management.

PROGRAM:

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

// Function to find the index of the block that has the smallest size

// that is still large enough for the process

int bestFit(int blockSize[], int m, int processSize) {

int idx = -1;

int minBlockSize = 10000; // Initialize with a large value

// Find the block with the smallest size that can accommodate the process

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

if (blockSize[i] >= processSize && blockSize[i] < minBlockSize) {

minBlockSize = blockSize[i];

idx = i;

}

}

return idx;

}

void bestFitMemoryAllocation(int blockSize[], int m, int processSize[], int n) {

// Array to store which block each process gets allocated to

int allocation[n];

// Initially, no process is allocated

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

allocation[i] = -1; // -1 means no allocation

}

// Allocate blocks to processes

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

int blockIdx = bestFit(blockSize, m, processSize[i]);

if (blockIdx != -1) {

// Process can be allocated to this block

allocation[i] = blockIdx;

// Reduce the size of the block

blockSize[blockIdx] -= processSize[i];

}

}

// Display the results

printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

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

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Define the block sizes and process sizes

int blockSize[MAX\_BLOCKS] = {100, 500, 200, 300, 600};

int processSize[MAX\_PROCESSES] = {212, 417, 112, 426};

// Number of blocks and processes

int m = 5; // Number of blocks

int n = 4; // Number of processes

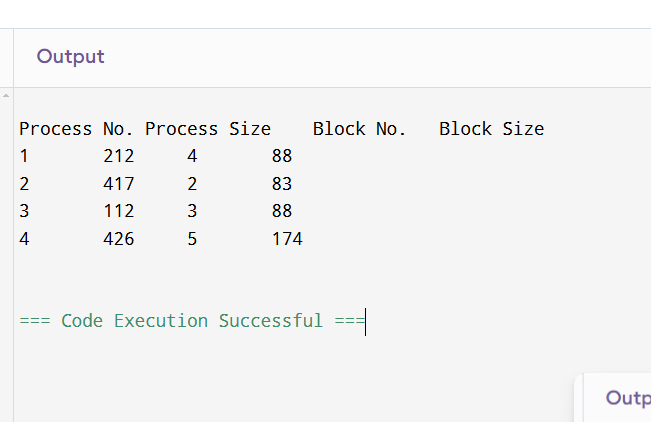
// Call the Best Fit memory allocation function

bestFitMemoryAllocation(blockSize, m, processSize, n);

return 0;

}

OUTPUT:



23) Construct a C program to implement first fit algorithm ofmemory management.

PROGRAM:

#include <stdio.h>

#define MAX\_BLOCKS 10

#define MAX\_PROCESSES 10

// Function to implement First Fit memory allocation

void firstFitMemoryAllocation(int blockSize[], int m, int processSize[], int n) {

// Array to store which block each process gets allocated to

int allocation[n];

// Initially, no process is allocated

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

allocation[i] = -1; // -1 means no allocation

}

// Allocate blocks to processes

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

// Find the first block that is large enough

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

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

// Process can be allocated to this block

allocation[i] = j;

// Reduce the size of the block

blockSize[j] -= processSize[i];

break;

}

}

}

// Display the results

printf("\nProcess No.\tProcess Size\tBlock No.\tBlock Size\n");

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

printf("%d\t\t%d\t\t", i + 1, processSize[i]);

if (allocation[i] != -1) {

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

} else {

printf("Not Allocated\n");

}

}

}

int main() {

// Define the block sizes and process sizes

int blockSize[MAX\_BLOCKS] = {100, 500, 200, 300, 600};

int processSize[MAX\_PROCESSES] = {212, 417, 112, 426};

// Number of blocks and processes

int m = 5; // Number of blocks

int n = 4; // Number of processes

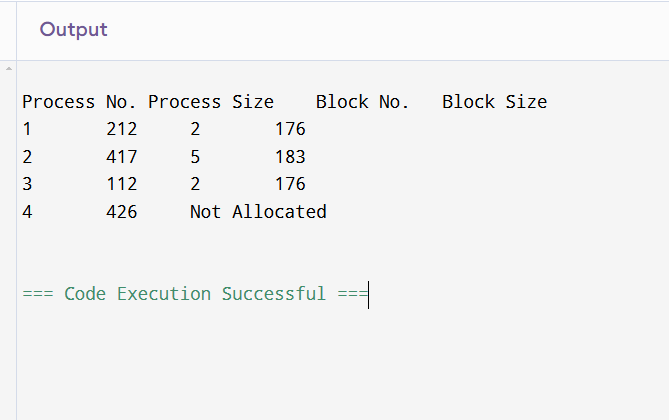
// Call the First Fit memory allocation function

firstFitMemoryAllocation(blockSize, m, processSize, n);

return 0;

}

OUTPUT:



24) Design a C program to demonstrate UNIX system calls for filemanagement.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

int main() {

int fd;

char \*filename = "example.txt";

char \*text = "Hello, UNIX System Calls!\n";

// Create a file

fd = open(filename, O\_CREAT | O\_WRONLY | O\_TRUNC, S\_IRUSR | S\_IWUSR);

if (fd == -1) {

perror("Error creating file");

exit(EXIT\_FAILURE);

}

// Write to the file

if (write(fd, text, sizeof(text)) == -1) {

perror("Error writing to file");

close(fd);

exit(EXIT\_FAILURE);

}

// Close the file

close(fd);

// Open the file for reading

fd = open(filename, O\_RDONLY);

if (fd == -1) {

perror("Error opening file");

exit(EXIT\_FAILURE);

}

// Read from the file

char buffer[100];

ssize\_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);

if (bytesRead == -1) {

perror("Error reading from file");

close(fd);

exit(EXIT\_FAILURE);

}

// Null-terminate the buffer

buffer[bytesRead] = '\0';

printf("Read from file: %s", buffer);

// Close the file

close(fd);

// Remove the file

if (remove(filename) == -1) {

perror("Error deleting file");

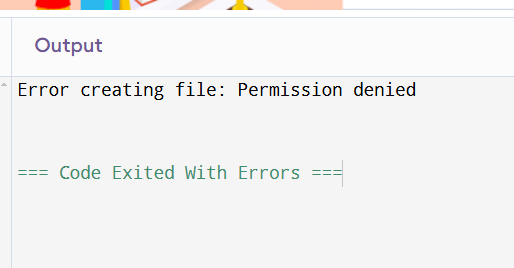
exit(EXIT\_FAILURE);

}

return 0;

}

OUTPUT:



25) Construct a C program to implement the I/O system calls ofUNIX (fcntl, seek, stat, opendir, readdir)

PROGRAM:

#include<stdio.h>

#include<fcntl.h>

#include<errno.h>

extern int errno;

int main()

{

int fd = open("foo.txt", O\_RDONLY | O\_CREAT);

printf("fd = %d\n", fd);

if (fd ==-1)

{

printf("Error Number % d\n", errno);

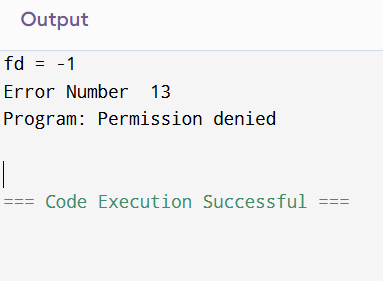
perror("Program");

}

return 0;

}

OUTPUT:



26) Construct a C program to implement the file managementoperations.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

void createFile(const char \*filename) {

FILE \*file = fopen(filename, "w");

if (file == NULL) {

perror("Error creating file");

return;

}

printf("File '%s' created successfully.\n", filename);

fclose(file);

}

void writeFile(const char \*filename, const char \*content) {

FILE \*file = fopen(filename, "w");

if (file == NULL) {

perror("Error opening file for writing");

return;

}

fprintf(file, "%s", content);

printf("Content written to file '%s'.\n", filename);

fclose(file);

}

void appendToFile(const char \*filename, const char \*content) {

FILE \*file = fopen(filename, "a");

if (file == NULL) {

perror("Error opening file for appending");

return;

}

fprintf(file, "%s", content);

printf("Content appended to file '%s'.\n", filename);

fclose(file);

}

void readFile(const char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file for reading");

return;

}

printf("Content of file '%s':\n", filename);

char ch;

while ((ch = fgetc(file)) != EOF) {

putchar(ch);

}

printf("\n");

fclose(file);

}

void deleteFile(const char \*filename) {

if (remove(filename) == 0) {

printf("File '%s' deleted successfully.\n", filename);

} else {

perror("Error deleting file");

}

}

int main() {

char filename[100], content[1000];

int choice;

do {

printf("\nFile Management Operations:\n");

printf("1. Create File\n");

printf("2. Write to File\n");

printf("3. Append to File\n");

printf("4. Read File\n");

printf("5. Delete File\n");

printf("6. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter the filename: ");

scanf("%s", filename);

createFile(filename);

break;

case 2:

printf("Enter the filename: ");

scanf("%s", filename);

printf("Enter the content: ");

getchar(); // To consume the newline character left by scanf

fgets(content, sizeof(content), stdin);

writeFile(filename, content);

break;

case 3:

printf("Enter the filename: ");

scanf("%s", filename);

printf("Enter the content to append: ");

getchar();

fgets(content, sizeof(content), stdin);

appendToFile(filename, content);

break;

case 4:

printf("Enter the filename: ");

scanf("%s", filename);

readFile(filename);

break;

case 5:

printf("Enter the filename: ");

scanf("%s", filename);

deleteFile(filename);

break;

case 6:

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

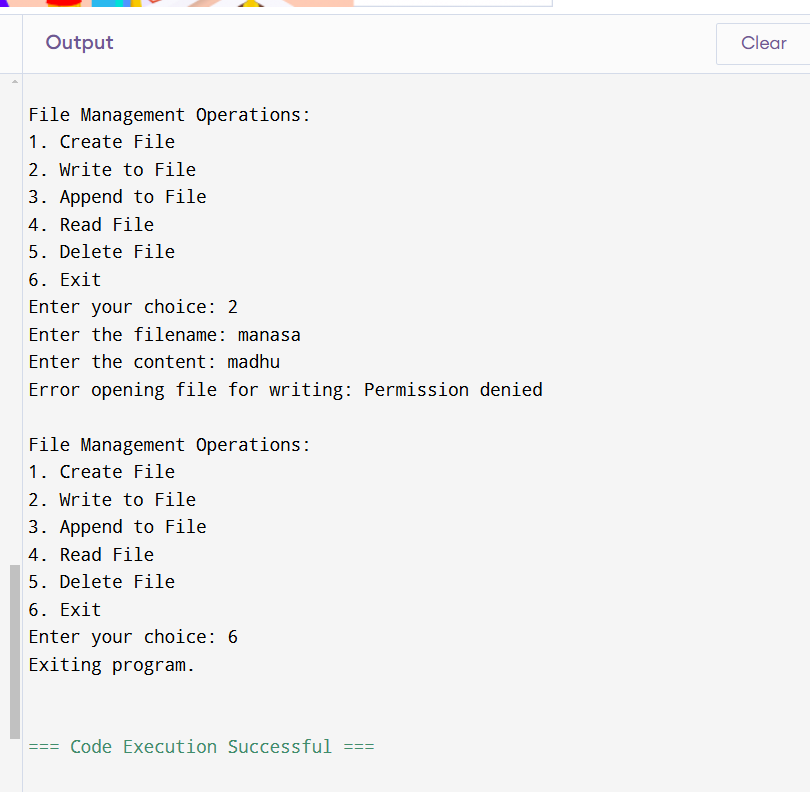
}

} while (choice != 6);

return 0;

}

OUTPUT:



27) Develop a C program for simulating the function of ls UNIXCommand.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <dirent.h>

void listDirectory(const char \*path) {

struct dirent \*entry;

DIR \*dir = opendir(path);

if (dir == NULL) {

perror("Unable to open directory");

return;

}

printf("Contents of directory '%s':\n", path);

while ((entry = readdir(dir)) != NULL) {

// Skip current and parent directory entries

if (entry->d\_name[0] == '.' && (entry->d\_name[1] == '\0' || (entry->d\_name[1] == '.' && entry->d\_name[2] == '\0'))) {

continue;

}

printf("%s\n", entry->d\_name);

}

closedir(dir);

}

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

char path[256];

if (argc > 1) {

// Path provided as command-line argument

listDirectory(argv[1]);

} else {

// Default to current directory

printf("Enter directory path (leave blank for current directory): ");

if (fgets(path, sizeof(path), stdin) == NULL) {

fprintf(stderr, "Error reading input.\n");

return 1;

}

// Remove trailing newline character from fgets

size\_t len = strlen(path);

if (len > 0 && path[len - 1] == '\n') {

path[len - 1] = '\0';

}

if (path[0] == '\0') {

listDirectory(".");

} else {

listDirectory(path);

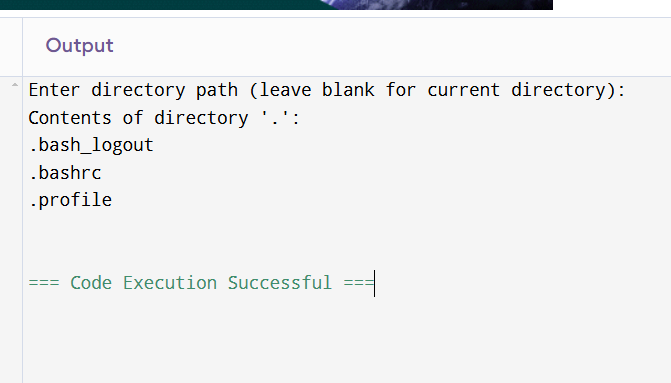
}

}

return 0;

}

OUTPUT:



28) Write a C program for simulation of GREP UNIX command

PROGRAM:

#include <stdio.h>

#include <string.h>

#define MAX\_LINE\_LENGTH 1024

// Function to simulate `grep` functionality

void grep\_pattern\_in\_file(const char \*pattern, const char \*filename) {

FILE \*file = fopen(filename, "r");

if (file == NULL) {

perror("Error opening file");

return;

}

char line[MAX\_LINE\_LENGTH];

int line\_number = 0;

int found = 0;

// Read the file line by line

while (fgets(line, sizeof(line), file) != NULL) {

line\_number++;

// Check if the pattern is present in the current line

if (strstr(line, pattern) != NULL) {

found = 1;

printf("Line %d: %s", line\_number, line);

}

}

if (!found) {

printf("Pattern '%s' not found in the file '%s'.\n", pattern, filename);

}

fclose(file);

}

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

// Check if sufficient arguments are passed (pattern and file)

if (argc != 3) {

fprintf(stderr, "Usage: %s <pattern> <filename>\n", argv[0]);

return 1;

}

const char \*pattern = argv[1]; // The pattern to search for

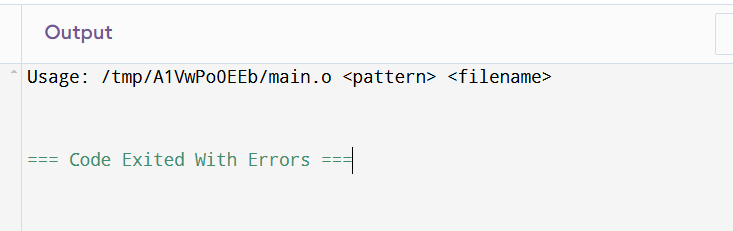
const char \*filename = argv[2]; // The file to search in

grep\_pattern\_in\_file(pattern, filename);

return 0;

}

OUTPUT:



29) Write a C program to simulate the solution of ClassicalProcess Synchronization Problem

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <stdlib.h> // for rand() and srand()

#include <time.h> // for time()

#define BUFFER\_SIZE 5 // Maximum number of items in the buffer

#define NUM\_PRODUCERS 1

#define NUM\_CONSUMERS 1

#define MAX\_OUTPUTS 5 // Limit the number of outputs to 5

// Shared buffer

int buffer[BUFFER\_SIZE];

int in = 0; // Index for the next item to be produced

int out = 0; // Index for the next item to be consumed

// Semaphores

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

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

sem\_t mutex; // Mutex for mutual exclusion when accessing the buffer

// Shared variable to track number of outputs

int produced\_count = 0;

int consumed\_count = 0;

// Producer function

void\* producer(void\* arg) {

int item;

while (produced\_count < MAX\_OUTPUTS) { // Limit the number of produced items

item = rand() % 100; // Produce an item (random number)

sem\_wait(&empty); // Wait if no empty slot is available

sem\_wait(&mutex); // Wait to access the buffer

// Add item to the buffer

buffer[in] = item;

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

printf("Produced: %d\n", item);

produced\_count++;

sem\_post(&mutex); // Release the mutex

sem\_post(&full); // Increase the full slot count

sleep(1); // Simulate some work

}

return NULL;

}

// Consumer function

void\* consumer(void\* arg) {

int item;

while (consumed\_count < MAX\_OUTPUTS) { // Limit the number of consumed items

sem\_wait(&full); // Wait if no full slot is available

sem\_wait(&mutex); // Wait to access the buffer

// Remove item from the buffer

item = buffer[out];

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

printf("Consumed: %d\n", item);

consumed\_count++;

sem\_post(&mutex); // Release the mutex

sem\_post(&empty); // Increase the empty slot count

sleep(2); // Simulate some work

}

return NULL;

}

int main() {

pthread\_t producers[NUM\_PRODUCERS];

pthread\_t consumers[NUM\_CONSUMERS];

// Seed the random number generator

srand(time(NULL));

// Initialize semaphores

sem\_init(&empty, 0, BUFFER\_SIZE); // Initially all slots are empty

sem\_init(&full, 0, 0); // Initially no full slots

sem\_init(&mutex, 0, 1); // Mutex is initially available

// Create producer and consumer threads

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

pthread\_create(&producers[i], NULL, producer, (void\*)i);

}

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

pthread\_create(&consumers[i], NULL, consumer, (void\*)i);

}

// Wait for all threads to finish

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

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

}

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

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

}

// Destroy semaphores

sem\_destroy(&empty);

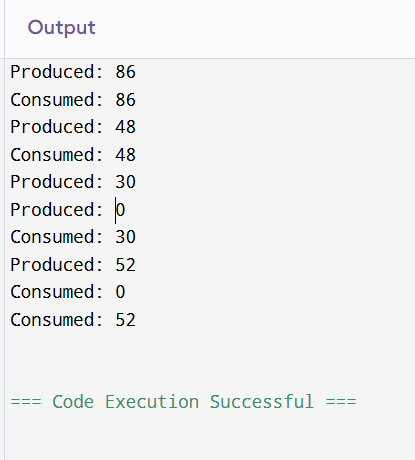
sem\_destroy(&full);

sem\_destroy(&mutex);

return 0;

}

OUTPUT:



30) Write C programs to demonstrate the following thread related concepts. (i)create (ii) join (iii) equal (iv) exit

PROGRAM:

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h> // Required for sleep function

void\* threadFunction(void\* arg) {

// Thread function that will be executed by the new thread

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

printf("Thread %d started, executing its task...\n", \*num);

// Simulating some work with sleep

sleep(1);

printf("Thread %d exiting...\n", \*num);

pthread\_exit(NULL); // Thread exit

}

int main() {

pthread\_t thread1, thread2; // Declare two thread IDs

int threadNum1 = 1, threadNum2 = 2;

// 1. Create threads

printf("Creating threads...\n");

if (pthread\_create(&thread1, NULL, threadFunction, &threadNum1) != 0) {

perror("Error creating thread 1");

return 1;

}

if (pthread\_create(&thread2, NULL, threadFunction, &threadNum2) != 0) {

perror("Error creating thread 2");

return 1;

}

// 2. Join threads

printf("Waiting for threads to finish...\n");

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

perror("Error joining thread 1");

return 1;

}

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

perror("Error joining thread 2");

return 1;

}

// 3. Thread equality check (pthread\_equal)

if (pthread\_equal(thread1, thread2)) {

printf("Thread 1 and Thread 2 are equal.\n");

} else {

printf("Thread 1 and Thread 2 are not equal.\n");

}

// 4. Exit the main thread

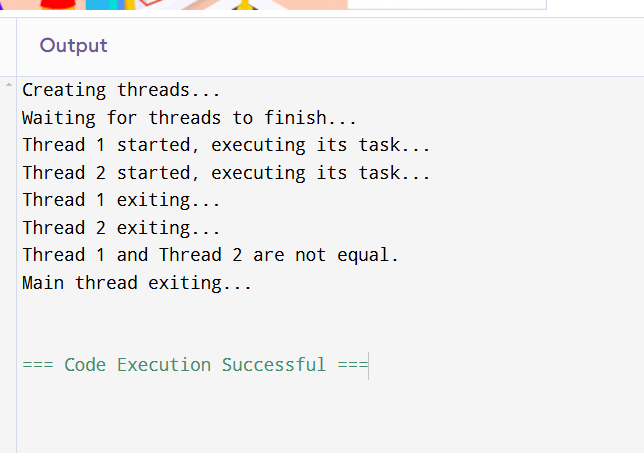
printf("Main thread exiting...\n");

pthread\_exit(NULL); // Exit the main thread

return 0;

}

OUTPUT:



31) Construct a C program to simulate the First in First Out paging technique of memory management.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100 // Maximum number of pages in memory

// Function to simulate FIFO page replacement

void fifo(int pages[], int numPages, int numFrames) {

int frames[numFrames]; // Array to store pages in memory

int pageFaults = 0; // To count the page faults

// Initialize all frames as empty (i.e., -1)

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

frames[i] = -1;

}

int front = 0; // Points to the front of the FIFO queue

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

int page = pages[i];

int pageFound = 0;

// Check if the page is already in memory

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

if (frames[j] == page) {

pageFound = 1; // Page is already in memory

break;

}

}

// If page not found in memory, it's a page fault

if (!pageFound) {

pageFaults++;

// Replace the oldest page (FIFO) by the new page

frames[front] = page;

front = (front + 1) % numFrames; // Move front pointer in a circular manner

}

// Print the current memory frame status after each page access

printf("Page %d: ", page);

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

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

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

}

}

printf("\n");

}

// Print total page faults

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

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3}; // Page reference string

int numPages = sizeof(pages) / sizeof(pages[0]); // Number of pages

int numFrames;

printf("Enter number of frames: ");

scanf("%d", &numFrames);

if (numFrames <= 0) {

printf("Number of frames must be greater than 0.\n");

return 1;

}

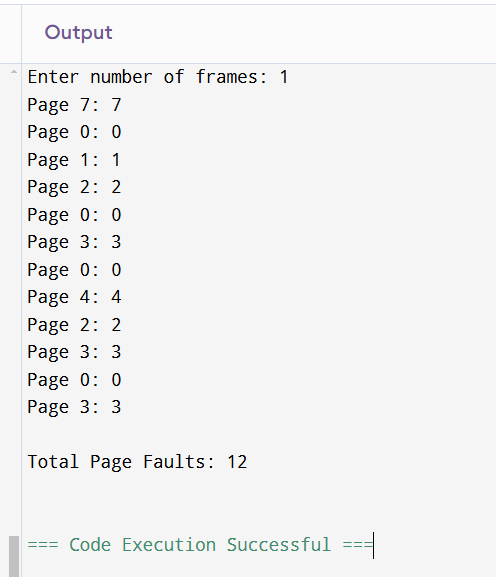
// Call the FIFO page replacement function

fifo(pages, numPages, numFrames);

return 0;

}

OUTPUT:



32) Construct a C program to simulate the Least Recently Used paging technique of memory management.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100 // Maximum number of pages in memory

// Function to simulate LRU page replacement

void lru(int pages[], int numPages, int numFrames) {

int frames[numFrames]; // Array to store pages in memory

int lastUsed[numFrames]; // Array to store the last used time of each page

int pageFaults = 0; // To count the page faults

// Initialize all frames as empty (-1) and lastUsed time to -1

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

frames[i] = -1;

lastUsed[i] = -1;

}

// To keep track of the current time (counter for accesses)

int time = 0;

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

int page = pages[i];

int pageFound = 0;

// Check if the page is already in memory

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

if (frames[j] == page) {

pageFound = 1; // Page is already in memory

lastUsed[j] = time; // Update last used time

break;

}

}

// If the page is not found in memory, it's a page fault

if (!pageFound) {

pageFaults++;

// Find the least recently used page (i.e., the page with the smallest lastUsed time)

int lruIndex = 0;

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

if (lastUsed[j] < lastUsed[lruIndex]) {

lruIndex = j;

}

}

// Replace the LRU page with the new page

frames[lruIndex] = page;

lastUsed[lruIndex] = time; // Set last used time of the replaced page to current time

}

// Print the current memory frame status after each page access

printf("Page %d: ", page);

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

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

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

}

}

printf("\n");

time++; // Increment the time (to simulate access time)

}

// Print total page faults

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

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3}; // Page reference string

int numPages = sizeof(pages) / sizeof(pages[0]); // Number of pages

int numFrames;

printf("Enter number of frames: ");

scanf("%d", &numFrames);

if (numFrames <= 0) {

printf("Number of frames must be greater than 0.\n");

return 1;

}

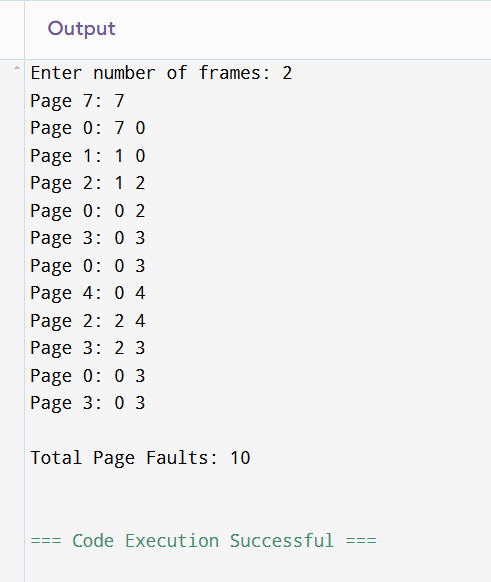
// Call the LRU page replacement function

lru(pages, numPages, numFrames);

return 0;

}

OUTPUT:



33) Construct a C program to simulate the optimal paging technique of memory management

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_PAGES 100 // Maximum number of pages in the reference string

// Function to simulate the Optimal page replacement algorithm

void optimal(int pages[], int numPages, int numFrames) {

int frames[numFrames]; // Array to store pages in memory

int pageFaults = 0; // To count the page faults

// Initialize all frames as empty (-1)

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

frames[i] = -1;

}

// To store if a page is found in the frames

int pageFound;

// Loop through each page in the reference string

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

int page = pages[i];

pageFound = 0;

// Check if the page is already in memory

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

if (frames[j] == page) {

pageFound = 1; // Page is already in memory

break;

}

}

// If the page is not found in memory, it’s a page fault

if (!pageFound) {

pageFaults++;

// If there is an empty frame, place the page there

int emptyFrame = -1;

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

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

emptyFrame = j;

break;

}

}

// If no empty frame, we need to replace a page

if (emptyFrame == -1) {

// Find the page that will not be used for the longest time in the future

int farthest = -1;

int replaceIndex = -1;

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

int nextUse = -1;

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

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

nextUse = k;

break;

}

}

// If the page is not used again, it's a candidate for replacement

if (nextUse == -1) {

replaceIndex = j;

break;

}

// Find the page that is used furthest in the future

if (nextUse > farthest) {

farthest = nextUse;

replaceIndex = j;

}

}

// Replace the page that is furthest in the future

frames[replaceIndex] = page;

} else {

// Place the page in the empty frame

frames[emptyFrame] = page;

}

}

// Print the current state of memory frames after accessing the page

printf("Page %d: ", page);

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

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

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

}

}

printf("\n");

}

// Print the total page faults

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

}

int main() {

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3}; // Page reference string

int numPages = sizeof(pages) / sizeof(pages[0]); // Number of pages

int numFrames;

printf("Enter number of frames: ");

scanf("%d", &numFrames);

if (numFrames <= 0) {

printf("Number of frames must be greater than 0.\n");

return 1;

}

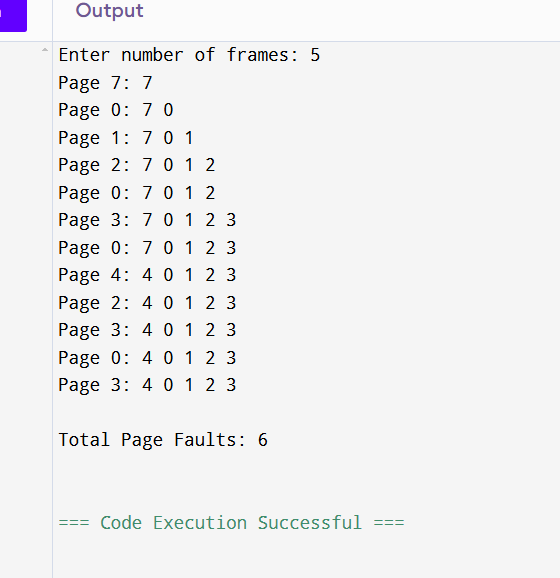
// Call the Optimal page replacement function

optimal(pages, numPages, numFrames);

return 0;

}

OUTPUT:



34) Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_RECORDS 100

typedef struct {

int id;

char data[256];

} Record;

Record file[MAX\_RECORDS];

int recordCount = 0;

void addRecord(int id, const char \*data) {

if (recordCount < MAX\_RECORDS) {

file[recordCount].id = id;

snprintf(file[recordCount].data, sizeof(file[recordCount].data), "%s", data);

recordCount++;

} else {

printf("File is full. Cannot add more records.\n");

}

}

void readRecords() {

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

printf("Record ID: %d, Data: %s\n", file[i].id, file[i].data);

}

}

int main() {

addRecord(1, "First record");

addRecord(2, "Second record");

addRecord(3, "Third record");

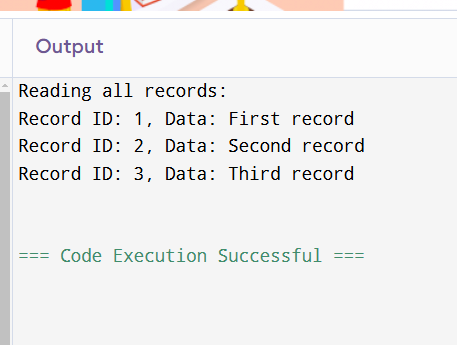
printf("Reading all records:\n");

readRecords();

return 0;

}

OUTPUT:



35) Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FILES 10

#define BLOCK\_SIZE 512

typedef struct {

int blockNumber;

char data[BLOCK\_SIZE];

} Block;

typedef struct {

Block blocks[MAX\_FILES];

int index[MAX\_FILES];

int fileCount;

} FileSystem;

void initializeFileSystem(FileSystem \*fs) {

fs->fileCount = 0;

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

fs->index[i] = -1;

}

}

void allocateFile(FileSystem \*fs, const char \*data) {

if (fs->fileCount < MAX\_FILES) {

fs->blocks[fs->fileCount].blockNumber = fs->fileCount;

snprintf(fs->blocks[fs->fileCount].data, BLOCK\_SIZE, "%s", data);

fs->index[fs->fileCount] = fs->fileCount;

fs->fileCount++;

} else {

printf("File system is full.\n");

}

}

void displayFileSystem(FileSystem \*fs) {

for (int i = 0; i < fs->fileCount; i++) {

printf("File %d: Block Number: %d, Data: %s\n", i, fs->blocks[i].blockNumber, fs->blocks[i].data);

}

}

int main() {

FileSystem fs;

initializeFileSystem(&fs);

allocateFile(&fs, "Hello, World!");

allocateFile(&fs, "File System Simulation");

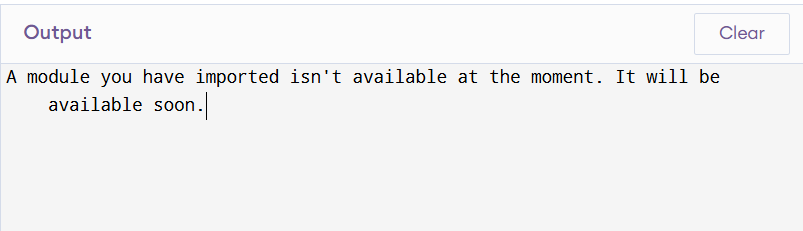
allocateFile(&fs, "C Programming");

displayFileSystem(&fs);

return 0;

}

OUTPUT:



36) With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define BLOCK\_SIZE 256 // Size of each block

// Structure to represent a data block

struct DataBlock {

char data[BLOCK\_SIZE]; // Data in the block

struct DataBlock\* next; // Pointer to the next block in the file

};

// Structure to represent a file (linked list of blocks)

struct File {

struct DataBlock\* firstBlock; // Pointer to the first block of the file

struct DataBlock\* lastBlock; // Pointer to the last block of the file

};

// Function to create a new block

struct DataBlock\* createBlock(const char\* data) {

struct DataBlock\* newBlock = (struct DataBlock\*)malloc(sizeof(struct DataBlock));

if (newBlock == NULL) {

printf("Error: Memory allocation failed for new block.\n");

exit(1);

}

strncpy(newBlock->data, data, BLOCK\_SIZE); // Copy data into the block

newBlock->next = NULL; // Initialize the next pointer to NULL

return newBlock;

}

// Function to add a block to the file

void addBlock(struct File\* file, const char\* data) {

struct DataBlock\* newBlock = createBlock(data);

if (file->firstBlock == NULL) {

// If the file is empty, the new block is both the first and last block

file->firstBlock = newBlock;

file->lastBlock = newBlock;

} else {

// Link the new block to the last block and update the last block pointer

file->lastBlock->next = newBlock;

file->lastBlock = newBlock;

}

printf("Added block: %s\n", data);

}

// Function to read and display all blocks in a file

void readFile(struct File\* file) {

if (file->firstBlock == NULL) {

printf("Error: File is empty.\n");

return;

}

struct DataBlock\* current = file->firstBlock;

printf("Reading file content:\n");

while (current != NULL) {

printf("%s\n", current->data);

current = current->next;

}

}

// Function to free the memory of the linked list blocks

void freeFile(struct File\* file) {

struct DataBlock\* current = file->firstBlock;

struct DataBlock\* nextBlock;

while (current != NULL) {

nextBlock = current->next;

free(current); // Free the current block

current = nextBlock; // Move to the next block

}

file->firstBlock = NULL; // Set the first block pointer to NULL

file->lastBlock = NULL; // Set the last block pointer to NULL

}

// Function to simulate file allocation using linked list strategy

int main() {

struct File file;

file.firstBlock = NULL; // Initialize the file (no blocks)

file.lastBlock = NULL;

// Simulate adding blocks to the file

addBlock(&file, "This is the first block of the file.");

addBlock(&file, "This is the second block of the file.");

addBlock(&file, "This is the third block of the file.");

// Simulate reading the file

readFile(&file);

// Simulate adding more blocks

addBlock(&file, "This is the fourth block of the file.");

readFile(&file);

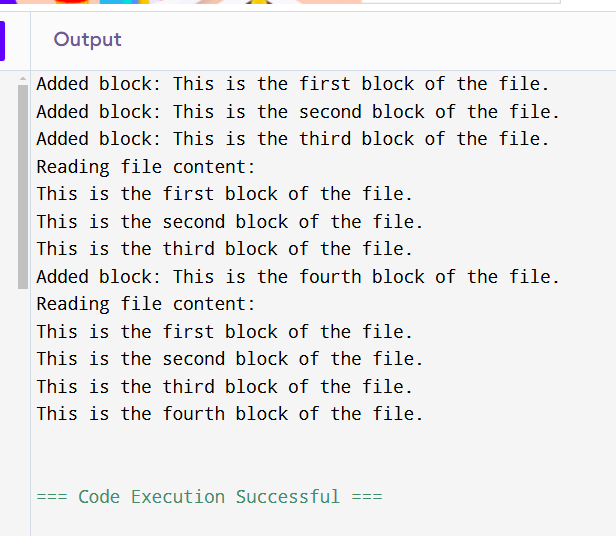
// Free the memory allocated for the file blocks

freeFile(&file);

return 0;

}

OUTPUT:



37) Construct a C program to simulate the First Come First Served disk scheduling algorithm.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

// Function to calculate and display the total and individual seek times

void fcfsDiskScheduling(int requests[], int n, int head) {

int totalSeekTime = 0; // Total seek time

int seekSequence[n]; // Sequence of seek operations

printf("Seek Sequence: %d", head); // Initial head position

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

int seekTime = abs(requests[i] - head); // Distance between head and request

totalSeekTime += seekTime; // Accumulate total seek time

head = requests[i]; // Move the head to the current request

seekSequence[i] = requests[i]; // Store the seek sequence

printf(" -> %d", head); // Print the movement

}

printf("\n\nIndividual Seek Times:\n");

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

int seekTime = (i == 0) ? abs(seekSequence[i] - head) : abs(seekSequence[i] - seekSequence[i - 1]);

printf("From %d to %d: %d\n", (i == 0 ? head : seekSequence[i - 1]), seekSequence[i], seekTime);

}

printf("\nTotal Seek Time: %d\n", totalSeekTime);

printf("Average Seek Time: %.2f\n", (float)totalSeekTime / n);

}

int main() {

int n, head;

// Input the number of disk requests

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

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests (cylinder numbers):\n");

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

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

}

// Input the initial position of the disk head

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

scanf("%d", &head);

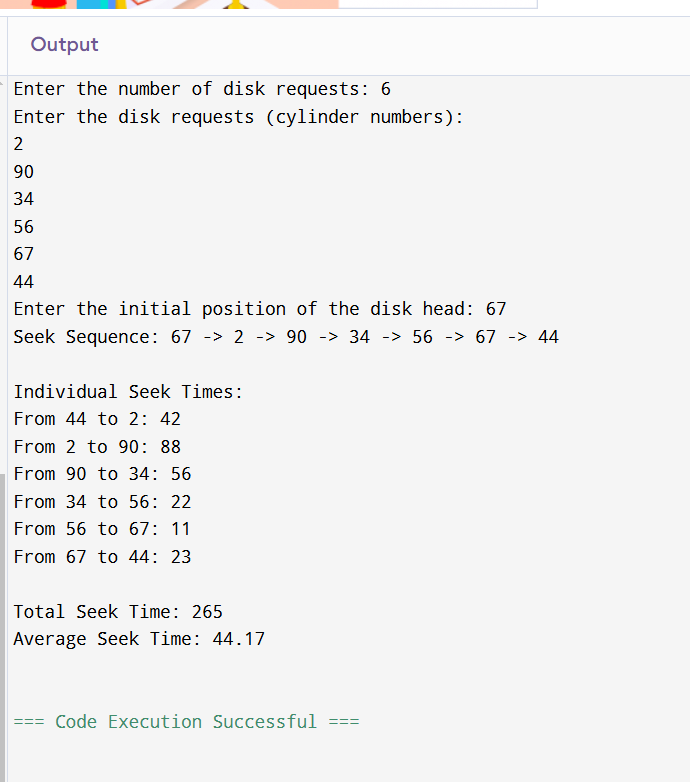
// Call the FCFS disk scheduling function

fcfsDiskScheduling(requests, n, head);

return 0;

}

OUTPUT:



38) Design a C program to simulate SCAN disk scheduling algorithm.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

// Function to sort the requests in ascending order

void sortRequests(int requests[], int n) {

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

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

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

int temp = requests[i];

requests[i] = requests[j];

requests[j] = temp;

}

}

}

}

// Function to simulate the SCAN disk scheduling algorithm

void scanDiskScheduling(int disk[], int n, int head, int direction) {

// Sort the requests in ascending order

sortRequests(disk, n);

int totalSeekCount = 0;

int distance, curTrack;

// Find the direction and service requests accordingly

if (direction == 1) {

// Move to the right first

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

if (disk[i] >= head) {

curTrack = disk[i];

distance = abs(curTrack - head);

totalSeekCount += distance;

head = curTrack;

}

}

// Then reverse direction to the left

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

if (disk[i] < head) {

curTrack = disk[i];

distance = abs(curTrack - head);

totalSeekCount += distance;

head = curTrack;

}

}

} else {

// Move to the left first

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

if (disk[i] <= head) {

curTrack = disk[i];

distance = abs(curTrack - head);

totalSeekCount += distance;

head = curTrack;

}

}

// Then reverse direction to the right

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

if (disk[i] > head) {

curTrack = disk[i];

distance = abs(curTrack - head);

totalSeekCount += distance;

head = curTrack;

}

}

}

printf("Total Seek Count: %d\n", totalSeekCount);

printf("Average Seek Count: %.2f\n", (float)totalSeekCount / n);

}

int main() {

int disk[] = { 55, 58, 60, 98, 10, 13, 23, 40 }; // Disk requests

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

int head = 50; // Initial position of the disk head

int direction = 1; // 1 for right, 0 for left

// Input the direction of the head's movement

printf("Enter the direction (1 for right, 0 for left): ");

scanf("%d", &direction);

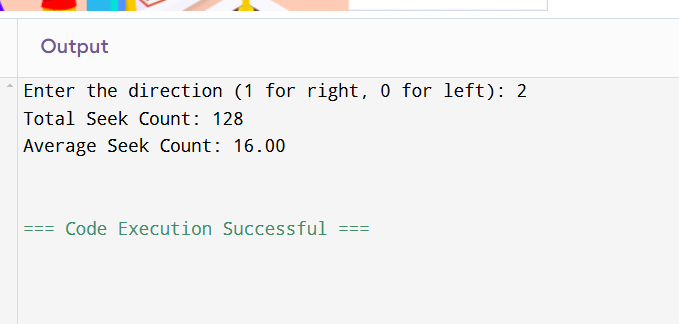
// Simulate the SCAN disk scheduling algorithm

scanDiskScheduling(disk, n, head, direction);

return 0;

}

OUTPUT:



39)Develop a C program to simulate C-SCAN disk scheduling algorithm.

PROGRAM:

#include <stdio.h>

#include <stdlib.h>

// Function to implement C-SCAN Disk Scheduling

void cScanDiskScheduling(int requests[], int n, int head, int diskSize) {

int totalSeekTime = 0;

int seekSequence[n + 2]; // +2 for boundaries (0 and diskSize - 1 if needed)

int index = 0;

// Sort the requests array

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

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

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

int temp = requests[j];

requests[j] = requests[j + 1];

requests[j + 1] = temp;

}

}

}

// Process requests from the current head to the largest request

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

if (requests[i] >= head) {

seekSequence[index++] = requests[i];

}

}

// Add the end of the disk (cylinder boundary) if necessary

seekSequence[index++] = diskSize - 1;

// Add the start of the disk (0) as the circular jump

seekSequence[index++] = 0;

// Process requests from the start of the disk to the head

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

if (requests[i] < head) {

seekSequence[index++] = requests[i];

}

}

// Calculate the total seek time

printf("Seek Sequence: %d", head);

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

printf(" -> %d", seekSequence[i]);

totalSeekTime += abs(seekSequence[i] - head);

head = seekSequence[i];

}

// Print the results

printf("\n\nTotal Seek Time: %d\n", totalSeekTime);

printf("Average Seek Time: %.2f\n", (float)totalSeekTime / n);

}

int main() {

int n, head, diskSize;

// Input the number of disk requests

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

scanf("%d", &n);

int requests[n];

// Input the disk requests

printf("Enter the disk requests (cylinder numbers):\n");

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

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

}

// Input the initial position of the disk head

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

scanf("%d", &head);

// Input the disk size

printf("Enter the total number of cylinders in the disk: ");

scanf("%d", &diskSize);

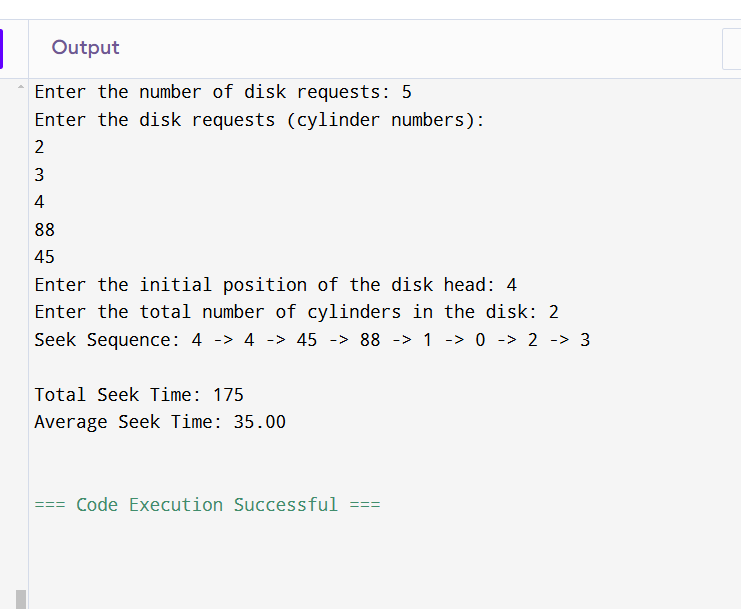
// Call the C-SCAN disk scheduling function

cScanDiskScheduling(requests, n, head, diskSize);

return 0;

}

OUTPUT:



40) Illustrate the various File Access Permission and different types of users in Linux.

PROGRAM:

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

void print\_permissions(mode\_t mode) {

// Owner (User) permissions

printf("User: ");

printf((mode & S\_IRUSR) ? "Read " : "No Read ");

printf((mode & S\_IWUSR) ? "Write " : "No Write ");

printf((mode & S\_IXUSR) ? "Execute " : "No Execute "); // Added space after "Execute"

printf("\n");

// Group permissions

printf("Group: ");

printf((mode & S\_IRGRP) ? "Read " : "No Read ");

printf((mode & S\_IWGRP) ? "Write " : "No Write ");

printf((mode & S\_IXGRP) ? "Execute " : "No Execute "); // Added space after "Execute"

printf("\n");

// Others permissions

printf("Others: ");

printf((mode & S\_IROTH) ? "Read " : "No Read ");

printf((mode & S\_IWOTH) ? "Write " : "No Write ");

printf((mode & S\_IXOTH) ? "Execute" : "No Execute"); // No space needed here

printf("\n");

}

int main() {

struct stat fileStat;

// Specify the file name

const char \*filename = "example.txt";

// Get file status

if (stat(filename, &fileStat) < 0) {

perror("Error: Unable to get file status");

return 1;

}

// Print the file's permissions

printf("File Permissions for '%s':\n", filename);

print\_permissions(fileStat.st\_mode);

return 0;

}

OUTPUT:

