1. **Illustrate the concept of multithreading using a C program**

**AIM :**

To implement the concept of multithreading using C program

## PROGRAM :

#include <stdio.h> #include <pthread.h>

void\* threadFunction(void\* arg) { char\* message = (char\*)arg; printf("%s\n", message);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

char\* message1 = "Hello from Thread 1!"; char\* message2 = "Hello from Thread 2!";

// Create threads

pthread\_create(&thread1, NULL, threadFunction, (void\*)message1); pthread\_create(&thread2, NULL, threadFunction, (void\*)message2);

// Wait for threads to complete pthread\_join(thread1, NULL); pthread\_join(thread2, NULL);

return 0;

}

## OUTPUT :

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

## AIM :

To design a C program to simulate the concept of Dining-Philosophers problem

## PROGRAM :

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <unistd.h>

#define NUM\_PHILOSOPHERS 5

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

void\* philosopherLifeCycle(void\* arg) { int id = \*((int\*)arg);

int left\_chopstick = id;

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

while (1) {

// Think

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

// Pick up chopsticks pthread\_mutex\_lock(&chopsticks[left\_chopstick]); pthread\_mutex\_lock(&chopsticks[right\_chopstick]);

// Eat

printf("Philosopher %d is eating...\n", id); sleep(rand() % 3 + 1); // Eating time

// Put down chopsticks pthread\_mutex\_unlock(&chopsticks[left\_chopstick]); pthread\_mutex\_unlock(&chopsticks[right\_chopstick]);

// Repeat the cycle

}

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS]; int philosopher\_ids[NUM\_PHILOSOPHERS];

// Initialize mutex locks

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

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

}

// Create philosopher threads

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

philosopher\_ids[i] = i;

pthread\_create(&philosophers[i], NULL, philosopherLifeCycle, (void\*)&philosopher\_ids[i]);

}

// Wait for threads to finish (although they run indefinitely) for (int i = 0; i < NUM\_PHILOSOPHERS; ++i) {

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

}

// Destroy mutex locks

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

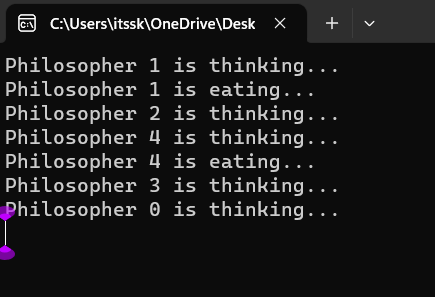
pthread\_mutex\_destroy(&chopsticks[i]);

}

return 0;

}

## OUTPUT :



1. **Construct a C program to implement various memory allocation strategies.**

## AIM :

## To construct a C program to implement 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;

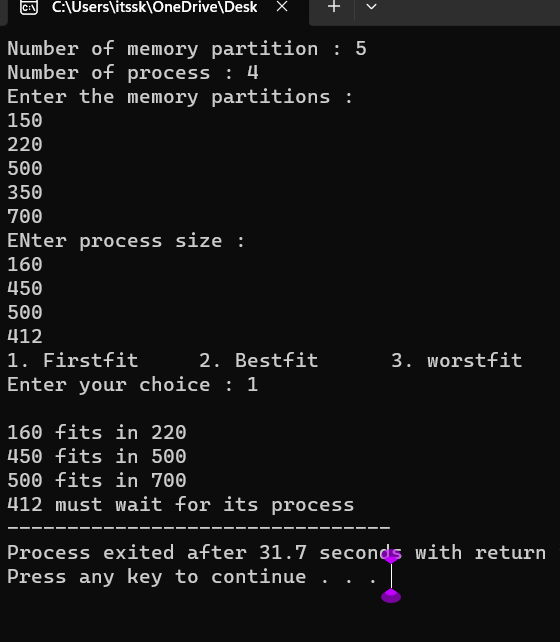
default:

printf("invalid"); break;

}

}

## OUTPUT :



1. **Construct a C program to organize the file using single level directory**

## AIM:

To construct a c program to organize the file using single level directory.

PROGRAM :

#include <stdio.h> #include <stdlib.h> #include <fcntl.h> #include <unistd.h>

#define BUFFER\_SIZE 4096 void copy(){

const char \*sourcefile= "C:/Users/itssk/OneDrive/Desktop/sasi.txt";

const char \*destination\_file="C:/Users/itssk/OneDrive/Desktop/sk.txt"; int source\_fd = open(sourcefile, O\_RDONLY);

int dest\_fd = open(destination\_file, O\_WRONLY | O\_CREAT | O\_TRUNC, 0666);

char buffer[BUFFER\_SIZE]; ssize\_t bytesRead, bytesWritten;

while ((bytesRead = read(source\_fd, buffer, BUFFER\_SIZE)) > 0) { bytesWritten = write(dest\_fd, buffer, bytesRead);

}

close(source\_fd); close(dest\_fd);

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

}

void create()

{

char path[100];

FILE \*fp; fp=fopen("C:/Users/itssk/OneDrive/Desktop/sasi.txt","w"); printf("file created successfully");

}

int main(){

int n;

printf("1. Create \t2. Copy \t3. Delete\nEnter your choice: " ); scanf("%d",&n);

switch(n){

case 1: create(); break;

case 2:

copy();

break;

case 3:

remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt"); printf("Deleted successfully");

}

}

## OUTPUT :

1. **Design a C program to organize the file using two level directory structure.**

## AIM :

To design a C program to organize the file using two level directory structure

## PROGRAM :

#include <stdio.h> #include <stdlib.h>

#include <string.h> int main() {

char mainDirectory[] = "C:/Users/itssk/OneDrive/Desktop"; char subDirectory[] = "os";

char fileName[] = "example.txt"; char filePath[200];

char mainDirPath[200];

snprintf(mainDirPath, sizeof(mainDirPath), "%s/%s/", mainDirectory, subDirectory);

snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName); FILE \*file = fopen(filePath, "w");

if (file == NULL) { printf("Error creating file.\n"); return 1;

}

fprintf(file, "This is an example file content."); printf("File created successfully: %s\n");

}

## OUTPUT :

**16. Develop a C program for implementing random access file for processing the employee details**

## AIM :

To develop a C program for implementing random access file for processing the employee details

## PROGRAM :

#include <stdio.h> #include <stdlib.h> struct Employee {

int empId;

char empName[50]; float empSalary;};

int main() { FILE \*filePtr;

struct Employee emp;

filePtr = fopen("employee.dat", "rb+"); if (filePtr == NULL) {

filePtr = fopen("employee.dat", "wb+"); if (filePtr == NULL) {

printf("Error creating the file.\n"); return 1; }

}

int choice; do {

printf("\nEmployee Database Menu:\n"); printf("1. Add Employee\n");

printf("2. Display Employee Details\n"); printf("3. Update Employee Details\n"); printf("4. Exit\n");

printf("Enter your choice: "); scanf("%d", &choice); switch (choice) {

case 1:

printf("Enter Employee ID: "); scanf("%d", &emp.empId); printf("Enter Employee Name: ");

scanf("%s", emp.empName); printf("Enter Employee Salary: "); scanf("%f", &emp.empSalary);

fseek(filePtr, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fwrite(&emp, sizeof(struct Employee), 1, filePtr); printf("Employee details added successfully.\n"); break;

case 2:

printf("Enter Employee ID to display: "); scanf("%d", &emp.empId);

fseek(filePtr, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fread(&emp, sizeof(struct Employee), 1, filePtr); printf("Employee ID: %d\n", emp.empId); printf("Employee Name: %s\n", emp.empName); printf("Employee Salary: %.2f\n", emp.empSalary); break;

case 3:

printf("Enter Employee ID to update: "); scanf("%d", &emp.empId);

fseek(filePtr, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fread(&emp, sizeof(struct Employee), 1, filePtr); printf("Enter Employee Name: ");

scanf("%s", emp.empName); printf("Enter Employee Salary: "); scanf("%f", &emp.empSalary);

fseek(filePtr, (emp.empId - 1) \* sizeof(struct Employee), SEEK\_SET);

fwrite(&emp, sizeof(struct Employee), 1, filePtr); printf("Employee details updated successfully.\n"); break;

case 4:

break; default:

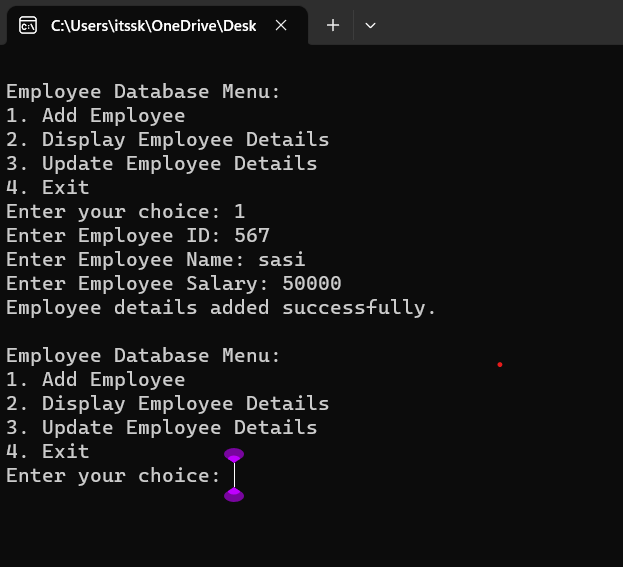
printf("Invalid choice. Please try again.\n");

}

} while (choice != 4); fclose(filePtr);

return 0;

## OUTPUT :



**17. Illustrate the deadlock avoidance concept by simulating Banker’s algorithm using C.**

## AIM :

To illustrate the deadlock avoidance concept by simulating Banker’s algorithm using C.

## PROGRAM :

#include <stdio.h>

#define MAX\_PROCESSES 5

#define MAX\_RESOURCES 3 int is\_safe();

int available[MAX\_RESOURCES] = {3, 3, 2}; // Available instances of each resource

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

{3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};

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

{2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};

int request\_resources(int process\_num, int request[]) {

// Check if request can be granted

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

if (request[i] > available[i] || request[i] > maximum[process\_num][i]

- allocation[process\_num][i])

return 0; // Request cannot be granted

}

// Try allocating resources temporarily

for (int i = 0; i < MAX\_RESOURCES; i++) { available[i] -= request[i];

allocation[process\_num][i] += request[i];

// Update maximum and need matrix if request is granted maximum[process\_num][i] -= request[i];

}

// Check if system is in safe state after allocation if (is\_safe()) {

return 1; // Request is granted

} else {

// Roll back changes if not safe

for (int i = 0; i < MAX\_RESOURCES; i++) { available[i] += request[i]; allocation[process\_num][i] -= request[i]; maximum[process\_num][i] += request[i];

}

return 0; // Request is denied

}

}

int is\_safe() {

int work[MAX\_RESOURCES];

int finish[MAX\_PROCESSES] = {0};

// Initialize work array

for (int i = 0; i < MAX\_RESOURCES; i++) { work[i] = available[i];

}

// Check if processes can finish int count = 0;

while (count < MAX\_PROCESSES) { int found = 0;

for (int i = 0; i < MAX\_PROCESSES; i++) { if (finish[i] == 0) {

int j;

for (j = 0; j < MAX\_RESOURCES; j++) {

if (maximum[i][j] - allocation[i][j] > work[j]) break;

}

if (j == MAX\_RESOURCES) {

// Process can finish, update work and mark as finished for (int k = 0; k < MAX\_RESOURCES; k++) {

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

}

finish[i] = 1;

found = 1; count++;

}

}

}

if (found == 0) {

return 0; // No process can finish, not safe state

}

}

return 1; // All processes can finish, safe state

}

int main() {

int process\_num, request[MAX\_RESOURCES]; printf("Enter process number (0 to 4): "); scanf("%d", &process\_num);

printf("Enter resource request (e.g., 0 1 0): "); for (int i = 0; i < MAX\_RESOURCES; i++) {

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

}

if (request\_resources(process\_num, request)) { printf("Request granted.\n");

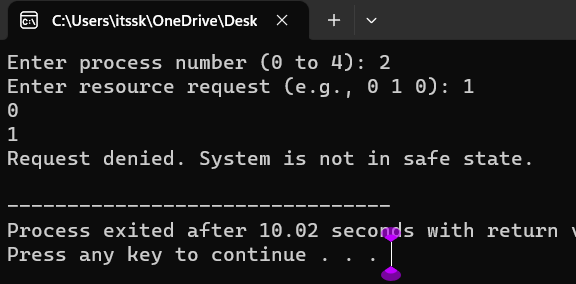
} else {

printf("Request denied. System is not in safe state.\n");

}

return 0;

## OUTPUT :



**18. Construct a C program to simulate producer consumer problem using semaphores.**

## AIM :

To construct a C program to simulate producer consumer problem using semaphores.

## PROGRAM :

#include <stdio.h> #include <pthread.h> #include <semaphore.h> #include<Windows.h>

#define BUFFER\_SIZE 5

#define MAX\_ITEMS 10 // Maximum number of items to be produced/consumed

int buffer[BUFFER\_SIZE]; sem\_t empty, full;

int produced\_items = 0, consumed\_items = 0;

void\* producer(void\* arg) {

while (produced\_items < MAX\_ITEMS) { sem\_wait(&empty);

// Critical section: add item to buffer

for (int i = 0; i < BUFFER\_SIZE; ++i) { if (buffer[i] == 0) {

buffer[i] = produced\_items + 1; printf("Produced: %d\n", buffer[i]);

produced\_items++; break;

}

}

sem\_post(&full);

Sleep(1); // Sleep for a while

}

return NULL;

}

void\* consumer(void\* arg) {

while (consumed\_items < MAX\_ITEMS) { sem\_wait(&full);

// Critical section: remove item from buffer for (int i = 0; i < BUFFER\_SIZE; ++i) {

if (buffer[i] != 0) {

printf("Consumed: %d\n", buffer[i]); buffer[i] = 0;

consumed\_items++; break;

}

}

sem\_post(&empty); Sleep(2); // Sleep for a while

}

return NULL;

}

int main() {

pthread\_t producer\_thread, consumer\_thread;

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

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

// Create producer and consumer threads pthread\_create(&producer\_thread, NULL, producer, NULL); pthread\_create(&consumer\_thread, NULL, consumer, NULL);

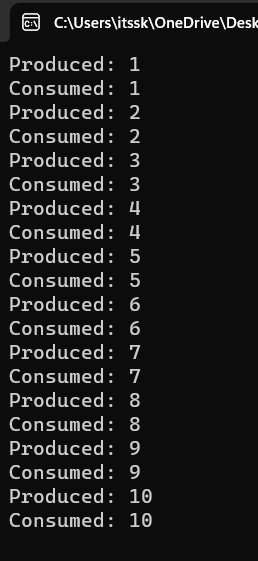
// Wait for threads to finish pthread\_join(producer\_thread, NULL); pthread\_join(consumer\_thread, NULL);

// Destroy semaphores sem\_destroy(&empty); sem\_destroy(&full);

return 0;

}

## OUTPUT :



**19. Design a C program to implement process synchronization using mutex locks.**

## AIM:

To design a C program to implement process synchronization using mutex locks.

## PROGRAM :

#include <stdio.h> #include <pthread.h>

// Shared variables int counter = 0;

pthread\_mutex\_t mutex;

// Function to be executed by threads void \*threadFunction(void \*arg) {

int i;

for (i = 0; i < 1000000; ++i) { }

return NULL;

}

int main() {

pthread\_mutex\_init(&mutex, NULL); pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, threadFunction, NULL); pthread\_create(&thread2, NULL, threadFunction, NULL);

// Wait for the threads to finish pthread\_join(thread1, NULL); pthread\_join(thread2, NULL);

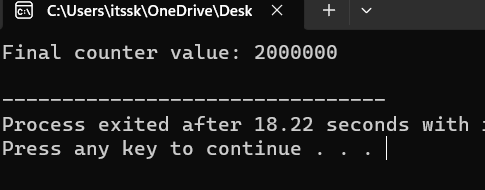
// Destroy the mutex pthread\_mutex\_destroy(&mutex);

// Print the final value of the counter printf("Final counter value: %d\n", counter);

return 0;

}

## OUTPUT :



1. **Construct a C program to simulate Reader-Writer problem using semaphores**

## AIM :

To construct a C program to simulate Reader-Writer problem using semaphores

## PROGRAM :

#include <stdio.h> #include <pthread.h> #include <semaphore.h>

sem\_t mutex, writeBlock;

int data = 0, readersCount = 0;

void \*reader(void \*arg) { int i=0;

while (i<10) { sem\_wait(&mutex); readersCount++;

if (readersCount == 1) { sem\_wait(&writeBlock);

}

sem\_post(&mutex);

// Reading operation

printf("Reader reads data: %d\n", data);

sem\_wait(&mutex); readersCount--;

if (readersCount == 0) { sem\_post(&writeBlock);

}

sem\_post(&mutex);

i++;

}

}

void \*writer(void \*arg) { int i=0;

while (i<10) { sem\_wait(&writeBlock);

// Writing operation data++;

printf("Writer writes data: %d\n", data);

sem\_post(&writeBlock); i++;

}

}

int main() {

pthread\_t readers, writers; sem\_init(&mutex, 0, 1);

sem\_init(&writeBlock, 0, 1); pthread\_create(&readers, NULL, reader, NULL); pthread\_create(&writers, NULL, writer, NULL); pthread\_join(readers, NULL); pthread\_join(writers, NULL); sem\_destroy(&mutex);

sem\_destroy(&writeBlock); return 0;

}

**OUTPUT :**

