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

**Code:**

**#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;**

**}**

**Out put:**

**Hello from Thread 1!**

**Hello from Thread 2!**

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

**Code:**

**#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);**

**sleep(rand() % 3 + 1); // Thinking time**

**// Pick up chopsticks (always pick up the lower-numbered first)**

**if (id % 2 == 0) {**

**pthread\_mutex\_lock(&chopsticks[left\_chopstick]);**

**pthread\_mutex\_lock(&chopsticks[right\_chopstick]);**

**} else {**

**pthread\_mutex\_lock(&chopsticks[right\_chopstick]);**

**pthread\_mutex\_lock(&chopsticks[left\_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]);**

**}**

**}**

**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:**

**Philosopher 0 is thinking...**

**Philosopher 1 is thinking...**

**Philosopher 2 is thinking...**

**Philosopher 3 is thinking...**

**Philosopher 4 is thinking...**

**Philosopher 0 is eating...**

**Philosopher 1 is eating...**

**Philosopher 2 is eating...**

**Philosopher 3 is eating...**

**Philosopher 4 is eating...**

**Philosopher 0 is thinking...**

**Philosopher 1 is thinking...**

**Philosopher 2 is thinking...**

**Philosopher 3 is thinking...**

**Philosopher 4 is thinking...**

**Philosopher 0 is eating...**

**Philosopher 1 is eating...**

**Philosopher 2 is eating...**

**Philosopher 3 is eating...**

**Philosopher 4 is eating...**

**13. Construct a C program to implement various memory allocationstrategies.**

**Code:**

**Number of memory partitions: 3**

**Number of processes: 4**

**Enter the memory partitions:**

**100**

**500**

**200**

**Enter process sizes:**

**212**

**417**

**112**

**426**

**1. First Fit 2. Best Fit 3. Worst Fit**

**Enter your choice: 2**

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

**Code:**

**#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);**

**if (source\_fd < 0) {**

**perror("Error opening source file");**

**return;**

**}**

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

**if (dest\_fd < 0) {**

**perror("Error opening destination file");**

**close(source\_fd);**

**return;**

**}**

**char buffer[BUFFER\_SIZE];**

**ssize\_t bytesRead, bytesWritten;**

**while ((bytesRead = read(source\_fd, buffer, BUFFER\_SIZE)) > 0) {**

**bytesWritten = write(dest\_fd, buffer, bytesRead);**

**if (bytesWritten < 0) {**

**perror("Error writing to destination file");**

**close(source\_fd);**

**close(dest\_fd);**

**return;**

**}**

**}**

**if (bytesRead < 0) {**

**perror("Error reading from source file");**

**}**

**close(source\_fd);**

**close(dest\_fd);**

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

**}**

**void create() {**

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

**FILE \*fp = fopen(path, "w");**

**if (fp == NULL) {**

**perror("Error creating file");**

**return;**

**}**

**fprintf(fp, "This is a sample text file.\n"); // Write some content to the file**

**fclose(fp);**

**printf("File created successfully.\n");**

**}**

**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:**

**if (remove("C:/Users/itssk/OneDrive/Desktop/sasi.txt") == 0) {**

**printf("File deleted successfully.\n");**

**} else {**

**perror("Error deleting file");**

**}**

**break;**

**default:**

**printf("Invalid choice.\n");**

**break;**

**}**

**return 0;**

**}**

**Input:**

**1**

**Output:**

**File created successfully.**

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

**Code:**

**#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];**

**// Create the main directory path**

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

**// Create the full file path**

**snprintf(filePath, sizeof(filePath), "%s%s", mainDirPath, fileName);**

**// Create the subdirectory if it doesn't exist**

**if (mkdir(subDirectory) == -1) {**

**perror("Error creating subdirectory (it may already exist)");**

**}**

**// Open the file for writing**

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

**if (file == NULL) {**

**printf("Error creating file.\n");**

**return 1;**

**}**

**// Write content to the file**

**fprintf(file, "This is an example file content.");**

**// Close the file**

**fclose(file);**

**// Print success message**

**printf("File created successfully: %s\n", filePath);**

**return 0;**

**}**

**Output**

**Error creating subdirectory (it may already exist)**

**File created successfully: C:/Users/itssk/OneDrive/Desktop/os/example.txt**

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

**Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**struct Employee {**

**int empId;**

**char empName[50];**

**float empSalary;**

**};**

**int main() {**

**FILE \*filePtr;**

**struct Employee emp;**

**// Open the file for reading and writing in binary mode**

**filePtr = fopen("employee.dat", "rb+");**

**if (filePtr == NULL) {**

**// If the file does not exist, create it**

**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);**

**if (emp.empId <= 0) {**

**printf("Invalid Employee ID. It must be greater than 0.\n");**

**break;**

**}**

**printf("Enter Employee Name: ");**

**scanf("%s", emp.empName); // Consider using fgets for safety**

**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);**

**if (emp.empId <= 0) {**

**printf("Invalid Employee ID. It must be greater than 0.\n");**

**break;**

**}**

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

**fread(&emp, sizeof(struct Employee), 1, filePtr);**

**if (feof(filePtr)) {**

**printf("Employee ID %d does not exist.\n", emp.empId);**

**} else {**

**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);**

**if (emp.empId <= 0) {**

**printf("Invalid Employee ID. It must be greater than 0.\n");**

**break;**

**}**

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

**fread(&emp, sizeof(struct Employee), 1, filePtr);**

**if (feof(filePtr)) {**

**printf("Employee ID %d does not exist.\n", emp.empId);**

**} else {**

**printf("Enter Employee Name: ");**

**scanf("%s", emp.empName); // Consider using fgets for safety**

**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:**

**printf("Exiting the program.\n");**

**break;**

**default:**

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

**}**

**} while (choice != 4);**

**fclose(filePtr);**

**return 0;**

**}**

**Input:**

**2**

**Enter Employee ID to display: 1**

**Output:**

**Employee ID: 1**

**Employee Name: John Doe**

**Employee Salary: 55000.00**

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

**Code:**

**#include <stdio.h>**

**#include <stdlib.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];**

**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};**

**int count = 0;**

**// Initialize work array**

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

**work[i] = available[i];**

**}**

**// Check if processes can finish**

**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);**

**// Validate process number**

**if (process\_num < 0 || process\_num >= MAX\_PROCESSES) {**

**printf("Invalid process number.\n");**

**return**

**Output:**

**Enter process number (0 to 4):**

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

**Code:**

**#include <stdio.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <unistd.h> // For usleep**

**#define BUFFER\_SIZE 5**

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

**int buffer[BUFFER\_SIZE] = {0}; // Initialize buffer to zero**

**sem\_t empty, full;**

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

**void\* producer(void\* arg) {**

**while (produced\_items < MAX\_ITEMS) {**

**sem\_wait(&empty); // Wait for an empty slot**

**// Critical section: add item to buffer**

**for (int i = 0; i < BUFFER\_SIZE; ++i) {**

**if (buffer[i] == 0) { // Check for an empty slot**

**buffer[i] = produced\_items + 1; // Produce an item**

**printf("Produced: %d\n", buffer[i]);**

**produced\_items++;**

**break;**

**}**

**}**

**sem\_post(&full); // Signal that an item has been produced**

**usleep(100000); // Sleep for a while (100 ms)**

**}**

**return NULL;**

**}**

**void\* consumer(void\* arg) {**

**while (consumed\_items < MAX\_ITEMS) {**

**sem\_wait(&full); // Wait for a full slot**

**// Critical section: remove item from buffer**

**for (int i = 0; i < BUFFER\_SIZE; ++i) {**

**if (buffer[i] != 0) { // Check for a produced item**

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

**buffer[i] = 0; // Remove the item**

**consumed\_items++;**

**break;**

**}**

**}**

**sem\_post(&empty); // Signal that an item has been consumed**

**usleep(200000); // Sleep for a while (200 ms)**

**}**

**return NULL;**

**}**

**int main() {**

**pthread\_t producer\_thread, consumer\_thread;**

**// Initialize semaphores**

**sem\_init(&empty, 0, BUFFER\_SIZE); // Initialize empty slots**

**sem\_init(&full, 0, 0); // Initialize full slots**

**// 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:**

**Produced: 1**

**Produced: 2**

**Consumed: 1**

**Produced: 3**

**Consumed: 2**

**Produced: 4**

**Consumed: 3**

**Produced: 5**

**Consumed: 4**

**Produced: 6**

**Consumed: 5**

**Produced: 7**

**Consumed: 6**

**Produced: 8**

**Consumed: 7**

**Produced: 9**

**Consumed: 8**

**Produced: 10**

**Consumed: 9**

**Consumed: 10**

**19. esign a C program to implement process synchronization usingmutex locks.**

**Code:**

**#include <stdio.h>**

**#include <pthread.h>**

**int counter = 0; // Shared variable**

**pthread\_mutex\_t mutex; // Mutex for protecting the counter**

**// Function to be executed by threads**

**void\* threadFunction(void \*arg) {**

**for (int i = 0; i < 1000000; ++i) {**

**pthread\_mutex\_lock(&mutex); // Lock the mutex**

**counter++; // Increment the counter**

**pthread\_mutex\_unlock(&mutex); // Unlock the mutex**

**}**

**return NULL;**

**}**

**int main() {**

**pthread\_mutex\_init(&mutex, NULL); // Initialize the mutex**

**pthread\_t thread1, thread2;**

**// Create two threads**

**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:**

**Final counter value: 2000000**

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

**Code:**

**#include <stdio.h>**

**#include <pthread.h>**

**#include <semaphore.h>**

**#include <stdlib.h>**

**#include <unistd.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 %ld reads data: %d\n", (long)arg, data);**

**usleep(rand() % 100); // Simulate reading time**

**sem\_wait(&mutex);**

**readersCount--;**

**if (readersCount == 0) {**

**sem\_post(&writeBlock);**

**}**

**sem\_post(&mutex);**

**i++;**

**}**

**return NULL;**

**}**

**void \*writer(void \*arg) {**

**int i = 0;**

**while (i < 10) {**

**sem\_wait(&writeBlock);**

**// Writing operation**

**data++;**

**printf("Writer %ld writes data: %d\n", (long)arg, data);**

**usleep(rand() % 100); // Simulate writing time**

**sem\_post(&writeBlock);**

**i++;**

**}**

**return NULL;**

**}**

**int main() {**

**pthread\_t readers[5], writers[2];**

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

**sem\_init(&writeBlock, 0, 1);**

**// Create multiple reader and writer threads**

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

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

**}**

**for (long i = 0; i < 2; i++) {**

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

**}**

**// Wait for all threads to finish**

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

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

**}**

**for (int i = 0; i < 2; i++) {**

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

**}**

**sem\_destroy(&mutex);**

**sem\_destroy(&writeBlock);**

**return 0;**

**}**

**Output:**

**Reader 0 reads data: 0**

**Reader 1 reads data: 0**

**Reader 2 reads data: 0**

**Reader 3 reads data: 0**

**Reader 4 reads data: 0**

**Writer 0 writes data: 1**

**Reader 0 reads data: 1**

**Reader 1 reads data: 1**

**Writer 1 writes data: 2**

**Reader 2 reads data: 2**

**Reader 3 reads data: 2**

**Reader 4 reads data: 2**

**Writer 0 writes data: 3**

**Reader 0 reads data: 3**

**Reader 1 reads data: 3**

**Writer 1 writes data: 4**

**Reader 2 reads data: 4**

**Reader 3 reads data: 4**

**Reader 4 reads data: 4**

**...**