Write a shell script to perform the following operations on a student database: a) Insert b) Delete c) Update d) Search write code for this

#!/bin/bash

# Define the file to store student records

DB\_FILE="student\_database.txt"

# Function to display the menu

show\_menu() {

echo "Choose an option:"

echo "1) Insert a new student"

echo "2) Delete a student"

echo "3) Update a student"

echo "4) Search for a student"

echo "5) Display all students"

echo "6) Exit"

echo -n "Enter your choice: "

}

# Function to insert a new student

insert\_student() {

echo -n "Enter Student ID: "

read id

echo -n "Enter Student Name: "

read name

echo -n "Enter Student Age: "

read age

# Append the new student to the database file

echo "$id,$name,$age" >> "$DB\_FILE"

echo "Student added successfully!"

}

# Function to delete a student

delete\_student() {

echo -n "Enter Student ID to delete: "

read id

# Delete the line with the matching student ID

if grep -q "^$id," "$DB\_FILE"; then

grep -v "^$id," "$DB\_FILE" > temp\_file && mv temp\_file "$DB\_FILE"

echo "Student deleted successfully!"

else

echo "Student ID not found!"

fi

}

# Function to update a student

update\_student() {

echo -n "Enter Student ID to update: "

read id

# Check if student exists

if grep -q "^$id," "$DB\_FILE"; then

echo -n "Enter new name: "

read name

echo -n "Enter new age: "

read age

# Update the student record

grep -v "^$id," "$DB\_FILE" > temp\_file

echo "$id,$name,$age" >> temp\_file

mv temp\_file "$DB\_FILE"

echo "Student updated successfully!"

else

echo "Student ID not found!"

fi

}

# Function to search for a student

search\_student() {

echo -n "Enter Student ID to search: "

read id

# Search for the student by ID

result=$(grep "^$id," "$DB\_FILE")

if [ -n "$result" ]; then

echo "Student found: $result"

else

echo "Student ID not found!"

fi

}

# Function to display all students

display\_all\_students() {

echo "Student Database:"

if [ -s "$DB\_FILE" ]; then

cat "$DB\_FILE"

else

echo "No students found in the database."

fi

}

# Main script loop

while true; do

show\_menu

read choice

case $choice in

1) insert\_student ;;

2) delete\_student ;;

3) update\_student ;;

4) search\_student ;;

5) display\_all\_students ;;

6) echo "Exiting..."; exit ;;

\*) echo "Invalid choice, please try again." ;;

esac

echo ""

done

Write a program to compute the finish time, turnaround time, and waiting time for the First-Come, First-Serve (FCFS) scheduling algorithm.

#include <stdio.h>

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int finish\_time;

int turn\_around\_time;

int waiting\_time;

} Process;

// Function to calculate FCFS metrics

void calculate\_fcfs(Process process[], int n) {

int current\_time = 0;

for (int i = 0; i < n; i++) { // Zero-indexed

// Finish Time = Current Time + Burst Time

process[i].finish\_time = current\_time + process[i].burst\_time;

// Turnaround Time = Finish Time - Arrival Time

process[i].turn\_around\_time = process[i].finish\_time - process[i].arrival\_time;

// Waiting Time = Turnaround Time - Burst Time

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

// Update current time for the next process

current\_time = process[i].finish\_time;

}

}

// Function to display the process table

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

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

for (int i = 0; i < n; i++) { // Zero-indexed

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

process[i].id,

process[i].arrival\_time,

process[i].burst\_time,

process[i].finish\_time,

process[i].turn\_around\_time,

process[i].waiting\_time);

}

}

int main() {

int n;

// Input number of processes

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

scanf("%d", &n);

// Dynamically allocate memory for n processes

Process process[n];

// Input arrival time and burst time for each process

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

process[i].id = i + 1; // Assign process ID starting from 1

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

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

}

// Calculate the FCFS scheduling metrics

calculate\_fcfs(process, n);

// Display the process table

display(process, n);

return 0; // Return success

}

Write a program to create the following processes and demonstrate: a) Zombie process b) Orphan process c) Sum of even numbers of an array in the parent process and odd numbers in the child process

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

void create\_zombie\_process() {

pid\_t pid = fork();

if (pid == 0) {

// Child process

printf("Child Process: I will terminate soon.\n");

exit(0); // Child process exits, but parent doesn't wait yet (Zombie)

} else if (pid > 0) {

// Parent process

printf("Parent Process: I will wait for a while and not immediately collect child status.\n");

sleep(10); // Sleep to keep child in zombie state

printf("Parent Process: Finished waiting.\n");

} else {

perror("Fork failed!");

}

}

void create\_orphan\_process() {

pid\_t pid = fork();

if (pid == 0) {

// Child process

printf("Child Process: I am an orphan! My parent will die before me.\n");

sleep(5); // Sleep to show orphan state

printf("Child Process: I am still alive after my parent has died.\n");

} else if (pid > 0) {

// Parent process

printf("Parent Process: I will terminate now, causing the child to become orphan.\n");

exit(0); // Parent process terminates

} else {

perror("Fork failed!");

}

}

void sum\_of\_even\_and\_odd\_numbers() {

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

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

int even\_sum = 0, odd\_sum = 0;

pid\_t pid = fork();

if (pid == 0) {

// Child process: Sum of odd numbers

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

if (arr[i] % 2 != 0) {

odd\_sum += arr[i];

}

}

printf("Child Process: Sum of odd numbers = %d\n", odd\_sum);

exit(0);

} else if (pid > 0) {

// Parent process: Sum of even numbers

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

if (arr[i] % 2 == 0) {

even\_sum += arr[i];

}

}

// Wait for child to finish

wait(NULL);

printf("Parent Process: Sum of even numbers = %d\n", even\_sum);

} else {

perror("Fork failed!");

}

}

int main() {

printf("Demonstrating Zombie Process\n");

create\_zombie\_process();

printf("\nDemonstrating Orphan Process\n");

create\_orphan\_process();

printf("\nDemonstrating Sum of Even and Odd Numbers\n");

sum\_of\_even\_and\_odd\_numbers();

return 0;

}

Implement multi-threading for matrix operations (e.g., addition, multiplication) using Pthreads.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define MATRIX\_SIZE 3

int matrix1[MATRIX\_SIZE][MATRIX\_SIZE];

int matrix2[MATRIX\_SIZE][MATRIX\_SIZE];

int result\_add[MATRIX\_SIZE][MATRIX\_SIZE];

int result\_mult[MATRIX\_SIZE][MATRIX\_SIZE];

// Structure to hold row and column information for each thread

typedef struct {

int row;

int col;

} MatrixIndex;

// Function to add corresponding elements of two matrices

void\* add\_matrices(void\* arg) {

MatrixIndex\* index = (MatrixIndex\*)arg;

int row = index->row;

int col = index->col;

result\_add[row][col] = matrix1[row][col] + matrix2[row][col];

pthread\_exit(0);

}

// Function to perform matrix multiplication

void\* multiply\_matrices(void\* arg) {

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

for (int col = 0; col < MATRIX\_SIZE; col++) {

result\_mult[row][col] = 0;

for (int k = 0; k < MATRIX\_SIZE; k++) {

result\_mult[row][col] += matrix1[row][k] \* matrix2[k][col];

}

}

pthread\_exit(0);

}

// Function to display a matrix

void display\_matrix(int matrix[MATRIX\_SIZE][MATRIX\_SIZE]) {

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

for (int j = 0; j < MATRIX\_SIZE; j++) {

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

int main() {

pthread\_t threads[MATRIX\_SIZE \* MATRIX\_SIZE];

int thread\_count = 0;

// Initialize matrices for demonstration

printf("Enter elements for Matrix 1:\n");

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

for (int j = 0; j < MATRIX\_SIZE; j++) {

scanf("%d", &matrix1[i][j]);

}

}

printf("Enter elements for Matrix 2:\n");

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

for (int j = 0; j < MATRIX\_SIZE; j++) {

scanf("%d", &matrix2[i][j]);

}

}

// Create threads for matrix addition

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

for (int j = 0; j < MATRIX\_SIZE; j++) {

MatrixIndex\* data = (MatrixIndex\*)malloc(sizeof(MatrixIndex));

data->row = i;

data->col = j;

pthread\_create(&threads[thread\_count++], NULL, add\_matrices, data);

}

}

// Wait for all addition threads to finish

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

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

}

printf("\nMatrix Addition Result:\n");

display\_matrix(result\_add);

// Create threads for matrix multiplication

thread\_count = 0;

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

int\* row = malloc(sizeof(int));

\*row = i;

pthread\_create(&threads[thread\_count++], NULL, multiply\_matrices, row);

}

// Wait for all multiplication threads to finish

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

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

}

printf("\nMatrix Multiplication Result:\n");

display\_matrix(result\_mult);

return 0;

}

Write a program to compute the finish time, turnaround time, and waiting time for the Round Robin scheduling algorithm.

#include <stdio.h>

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int remaining\_time; // Remaining time left for the process

int finish\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void calculate\_round\_robin(Process processes[], int n, int time\_quantum) {

int current\_time = 0;

int completed = 0;

int i;

while (completed < n) {

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

// Process should only execute if it has remaining burst time and has arrived

if (processes[i].remaining\_time > 0 && processes[i].arrival\_time <= current\_time) {

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

// Process executes for time\_quantum and is preempted

current\_time += time\_quantum;

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

} else {

// Process completes within the time\_quantum

current\_time += processes[i].remaining\_time;

processes[i].finish\_time = current\_time;

processes[i].turnaround\_time = processes[i].finish\_time - processes[i].arrival\_time;

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

processes[i].remaining\_time = 0;

completed++;

}

}

}

}

}

void display\_process\_times(Process processes[], int n) {

printf("\nID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

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

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

processes[i].id,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].finish\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

int main() {

int n, time\_quantum;

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

scanf("%d", &n);

Process processes[n];

// Input arrival and burst times for each process

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

processes[i].id = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].id);

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

processes[i].remaining\_time = processes[i].burst\_time; // Initialize remaining time

}

printf("Enter time quantum: ");

scanf("%d", &time\_quantum);

// Perform Round Robin scheduling

calculate\_round\_robin(processes, n, time\_quantum);

// Display finish time, turnaround time, and waiting time for each process

display\_process\_times(processes, n);

return 0;

}

Write a program to compute the finish time, turnaround time, and waiting time for the Shortest Job First (SJF) scheduling algorithm in both preemptive and non-preemptive modes.

#include <stdio.h>

#include <limits.h>

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int remaining\_time; // For preemptive mode (SRTF)

int finish\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void calculate\_sjf\_non\_preemptive(Process processes[], int n) {

int completed = 0, current\_time = 0;

int min\_index;

while (completed < n) {

// Find the shortest process available at current\_time

min\_index = -1;

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

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0) {

if (min\_index == -1 || processes[i].remaining\_time < processes[min\_index].remaining\_time) {

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

// Execute the selected process to completion

current\_time += processes[min\_index].remaining\_time;

processes[min\_index].remaining\_time = 0;

processes[min\_index].finish\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].finish\_time - processes[min\_index].arrival\_time;

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

completed++;

}

}

}

void calculate\_sjf\_preemptive(Process processes[], int n) {

int completed = 0, current\_time = 0;

int min\_index;

while (completed < n) {

// Find the process with the shortest remaining time

min\_index = -1;

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

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0) {

if (min\_index == -1 || processes[i].remaining\_time < processes[min\_index].remaining\_time) {

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

// Execute the selected process for 1 time unit

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

current\_time++;

// Check if the process is complete

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

processes[min\_index].finish\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].finish\_time - processes[min\_index].arrival\_time;

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

completed++;

}

}

}

}

void display\_process\_times(Process processes[], int n) {

printf("\nID\tArrival\tBurst\tFinish\tTurnaround\tWaiting\n");

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

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

processes[i].id,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].finish\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

int main() {

int n, mode;

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

scanf("%d", &n);

Process processes[n];

// Input arrival and burst times for each process

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

processes[i].id = i + 1;

printf("Enter arrival time and burst time for process %d: ", processes[i].id);

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

processes[i].remaining\_time = processes[i].burst\_time; // Initialize remaining time for SRTF

}

// Choose scheduling mode

printf("Enter 1 for Non-Preemptive SJF or 2 for Preemptive SJF: ");

scanf("%d", &mode);

if (mode == 1) {

calculate\_sjf\_non\_preemptive(processes, n);

printf("\nNon-Preemptive SJF Results:\n");

} else if (mode == 2) {

calculate\_sjf\_preemptive(processes, n);

printf("\nPreemptive SJF (SRTF) Results:\n");

} else {

printf("Invalid mode selected.\n");

return 1;

}

// Display finish time, turnaround time, and waiting time for each process

display\_process\_times(processes, n);

return 0;

}

Write a program to compute the finish time, turnaround time, and waiting time for the Priority scheduling algorithm in both preemptive and non-preemptive modes.

#include <stdio.h>

#include <limits.h>

typedef struct {

int id;

int arrival\_time;

int burst\_time;

int priority;

int remaining\_time; // For preemptive mode

int finish\_time;

int turnaround\_time;

int waiting\_time;

} Process;

void calculate\_priority\_non\_preemptive(Process processes[], int n) {

int completed = 0, current\_time = 0, min\_index;

while (completed < n) {

min\_index = -1;

// Find the highest priority process that has arrived and is not completed

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

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0) {

if (min\_index == -1 || processes[i].priority < processes[min\_index].priority) {

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

// Execute the selected process to completion

current\_time += processes[min\_index].remaining\_time;

processes[min\_index].remaining\_time = 0;

processes[min\_index].finish\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].finish\_time - processes[min\_index].arrival\_time;

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

completed++;

}

}

}

void calculate\_priority\_preemptive(Process processes[], int n) {

int completed = 0, current\_time = 0, min\_index;

while (completed < n) {

min\_index = -1;

// Find the highest priority process that has arrived and is not completed

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

if (processes[i].arrival\_time <= current\_time && processes[i].remaining\_time > 0) {

if (min\_index == -1 || processes[i].priority < processes[min\_index].priority) {

min\_index = i;

}

}

}

if (min\_index == -1) {

current\_time++;

} else {

// Execute the selected process for 1 time unit

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

current\_time++;

// Check if the process is complete

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

processes[min\_index].finish\_time = current\_time;

processes[min\_index].turnaround\_time = processes[min\_index].finish\_time - processes[min\_index].arrival\_time;

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

completed++;

}

}

}

}

void display\_process\_times(Process processes[], int n) {

printf("\nID\tArrival\tBurst\tPriority\tFinish\tTurnaround\tWaiting\n");

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

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

processes[i].id,

processes[i].arrival\_time,

processes[i].burst\_time,

processes[i].priority,

processes[i].finish\_time,

processes[i].turnaround\_time,

processes[i].waiting\_time);

}

}

int main() {

int n, mode;

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

scanf("%d", &n);

Process processes[n];

// Input arrival time, burst time, and priority for each process

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

processes[i].id = i + 1;

printf("Enter arrival time, burst time, and priority for process %d: ", processes[i].id);

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

processes[i].remaining\_time = processes[i].burst\_time; // Initialize remaining time for preemptive mode

}

// Choose scheduling mode

printf("Enter 1 for Non-Preemptive Priority or 2 for Preemptive Priority: ");

scanf("%d", &mode);

if (mode == 1) {

calculate\_priority\_non\_preemptive(processes, n);

printf("\nNon-Preemptive Priority Scheduling Results:\n");

} else if (mode == 2) {

calculate\_priority\_preemptive(processes, n);

printf("\nPreemptive Priority Scheduling Results:\n");

} else {

printf("Invalid mode selected.\n");

return 1;

}

// Display finish time, turnaround time, and waiting time for each process

display\_process\_times(processes, n);

return 0;

}

Write a program to check whether a given system is in a safe state or not using the Banker’s Deadlock Avoidance algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

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

int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int n, int m) {

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

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

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

}

}

}

bool isSafe(int processes[], int avail[], int max[][MAX\_RESOURCES],

int alloc[][MAX\_RESOURCES], int n, int m) {

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

calculateNeed(need, max, alloc, n, m);

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

int safeSeq[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = avail[i];

}

int count = 0;

while (count < n) {

bool found = false;

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

if (finish[p] == 0) {

int j;

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

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

break;

}

}

if (j == m) {

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

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

}

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (!found) {

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

return false;

}

}

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

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

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

}

printf("\n");

return true;

}

int main() {

int n, m;

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

scanf("%d", &n);

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

scanf("%d", &m);

int processes[MAX\_PROCESSES];

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

processes[i] = i;

}

int avail[MAX\_RESOURCES];

printf("Enter the available resources: ");

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

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

}

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

printf("Enter the Max matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

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

printf("Enter the Allocation matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

isSafe(processes, avail, max, alloc, n, m);

return 0;

}

Write a program to check whether a given system is in a safe state or not using the Banker’s Deadlock Avoidance algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

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

int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int n, int m) {

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

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

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

}

}

}

bool isSafe(int processes[], int avail[], int max[][MAX\_RESOURCES],

int alloc[][MAX\_RESOURCES], int n, int m) {

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

calculateNeed(need, max, alloc, n, m);

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

int safeSeq[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = avail[i];

}

int count = 0;

while (count < n) {

bool found = false;

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

if (finish[p] == 0) {

int j;

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

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

break;

}

}

if (j == m) {

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

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

}

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (!found) {

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

return false;

}

}

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

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

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

}

printf("\n");

return true;

}

int main() {

int n, m;

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

scanf("%d", &n);

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

scanf("%d", &m);

int processes[MAX\_PROCESSES];

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

processes[i] = i;

}

int avail[MAX\_RESOURCES];

printf("Enter the available resources: ");

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

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

}

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

printf("Enter the Max matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

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

printf("Enter the Allocation matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

isSafe(processes, avail, max, alloc, n, m);

return 0;

}

Write a program to check whether a given system is in a safe state or not using the Banker’s Deadlock Avoidance algorithm.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

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

int alloc[MAX\_PROCESSES][MAX\_RESOURCES], int n, int m) {

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

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

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

}

}

}

bool isSafe(int processes[], int avail[], int max[][MAX\_RESOURCES],

int alloc[][MAX\_RESOURCES], int n, int m) {

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

calculateNeed(need, max, alloc, n, m);

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

int safeSeq[MAX\_PROCESSES];

int work[MAX\_RESOURCES];

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

work[i] = avail[i];

}

int count = 0;

while (count < n) {

bool found = false;

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

if (finish[p] == 0) {

int j;

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

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

break;

}

}

if (j == m) {

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

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

}

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (!found) {

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

return false;

}

}

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

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

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

}

printf("\n");

return true;

}

int main() {

int n, m;

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

scanf("%d", &n);

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

scanf("%d", &m);

int processes[MAX\_PROCESSES];

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

processes[i] = i;

}

int avail[MAX\_RESOURCES];

printf("Enter the available resources: ");

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

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

}

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

printf("Enter the Max matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

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

printf("Enter the Allocation matrix:\n");

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

printf("For process %d: ", i);

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

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

}

}

isSafe(processes, avail, max, alloc, n, m);

return 0;

}

rite a program to calculate the number of page faults for a given reference string using the FIFO page replacement algorithm.

#include <stdio.h>

#include <stdbool.h>

int isPageInMemory(int frames[], int n\_frames, int page) {

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

if (frames[i] == page) {

return 1; // Page found in memory

}

}

return 0; // Page not found

}

int main() {

int n\_frames, n\_pages;

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

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

printf("Enter the number of pages in the reference string: ");

scanf("%d", &n\_pages);

int reference\_string[n\_pages];

printf("Enter the reference string:\n");

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

scanf("%d", &reference\_string[i]);

}

int frames[n\_frames];

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

frames[i] = -1; // Initialize all frames as empty (-1 means empty frame)

}

int page\_faults = 0, frame\_index = 0;

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

int page = reference\_string[i];

// Check if the page is already in memory

if (!isPageInMemory(frames, n\_frames, page)) {

// Page fault occurs

page\_faults++;

// Place the page in the current frame index (FIFO - oldest frame is replaced)

frames[frame\_index] = page;

// Move to the next frame in a circular manner

frame\_index = (frame\_index + 1) % n\_frames;

}

// Display current frame status

printf("Frames after accessing page %d: ", page);

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

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

printf("[ ] ");

} else {

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

}

}

printf("\n");

}

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

return 0;

}

Write a program to calculate the number of page faults for a given reference string using the Least Recently Used (LRU) page replacement algorithm.

#include <stdio.h>

#include <limits.h>

int findLRU(int time[], int n\_frames) {

int min\_time = time[0], min\_index = 0;

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

if (time[i] < min\_time) {

min\_time = time[i];

min\_index = i;

}

}

return min\_index;

}

int main() {

int n\_frames, n\_pages;

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

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

printf("Enter the number of pages in the reference string: ");

scanf("%d", &n\_pages);

int reference\_string[n\_pages];

printf("Enter the reference string:\n");

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

scanf("%d", &reference\_string[i]);

}

int frames[n\_frames];

int time[n\_frames];

int page\_faults = 0;

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

frames[i] = -1; // Initialize all frames as empty (-1 means empty frame)

}

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

int page = reference\_string[i];

int found = 0;

// Check if page is already in one of the frames

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

if (frames[j] == page) {

found = 1;

time[j] = i; // Update the time of use for this page

break;

}

}

// If page is not found, a page fault occurs

if (!found) {

int replace\_index;

// Check if there's an empty frame available

int empty\_frame = -1;

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

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

empty\_frame = j;

break;

}

}

if (empty\_frame != -1) {

replace\_index = empty\_frame; // Use the empty frame

} else {

replace\_index = findLRU(time, n\_frames); // Find the least recently used page

}

frames[replace\_index] = page; // Replace the page in the selected frame

time[replace\_index] = i; // Update the time of use for the new page

page\_faults++;

}

// Display current frame status

printf("Frames after accessing page %d: ", page);

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

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

printf("[ ] ");

} else {

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

}

}

printf("\n");

}

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

return 0;

}

Write a program to calculate the number of page faults for a given reference string using the Optimal page replacement algorithm.

#include <stdio.h>

int findOptimal(int frames[], int n\_frames, int reference\_string[], int n\_pages, int current\_index) {

int farthest\_index = -1;

int replace\_index = -1;

// For each frame, check when it will be used next, and select the one with the farthest future usage

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

int j;

// Find the next occurrence of frames[i]

for (j = current\_index + 1; j < n\_pages; j++) {

if (frames[i] == reference\_string[j]) {

if (j > farthest\_index) {

farthest\_index = j;

replace\_index = i;

}

break;

}

}

// If the page is not found in the future (i.e., it won't be used again), return this index immediately

if (j == n\_pages) {

return i;

}

}

// If all pages are used again in the future, replace the page with the farthest next use

return replace\_index;

}

int main() {

int n\_frames, n\_pages;

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

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

printf("Enter the number of pages in the reference string: ");

scanf("%d", &n\_pages);

int reference\_string[n\_pages];

printf("Enter the reference string:\n");

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

scanf("%d", &reference\_string[i]);

}

int frames[n\_frames];

int page\_faults = 0;

// Initialize frames to indicate they are empty

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

frames[i] = -1;

}

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

int page = reference\_string[i];

int found = 0;

// Check if the page is already in one of the frames

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

if (frames[j] == page) {

found = 1;

break;

}

}

// If the page is not found in the frames, a page fault occurs

if (!found) {

int replace\_index;

// Check if there is an empty frame

int empty\_frame = -1;

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

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

empty\_frame = j;

break;

}

}

if (empty\_frame != -1) {

replace\_index = empty\_frame; // Use the empty frame

} else {

// Use the optimal page replacement algorithm to find the page to replace

replace\_index = findOptimal(frames, n\_frames, reference\_string, n\_pages, i);

}

frames[replace\_index] = page; // Replace the page in the chosen frame

page\_faults++;

}

// Display current frame status

printf("Frames after accessing page %d: ", page);

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

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

printf("[ ] ");

} else {

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

}

}

printf("\n");

}

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

return 0;

}

Write a program to check for operand errors in a given job and raise an interrupt if an error is detected.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

// Function to simulate raising an interrupt (error handling)

void raise\_interrupt(const char \*error\_message) {

printf("Interrupt Raised: %s\n", error\_message);

// Simulate stopping the job by exiting the program

exit(1);

}

// Function to check if the operand is a valid number (integer)

int is\_valid\_operand(const char \*operand) {

// Check if the operand is empty

if (operand == NULL || strlen(operand) == 0) {

return 0; // Invalid

}

// Check if each character in the operand is a digit (for simple integer validation)

for (int i = 0; operand[i] != '\0'; i++) {

if (!isdigit(operand[i]) && operand[i] != '-' && operand[i] != '+') {

return 0; // Invalid character found

}

}

return 1; // Valid operand

}

// Function to perform arithmetic operation and check for errors

void perform\_operation(const char \*op1\_str, const char \*op2\_str, char operator) {

int op1, op2;

// Validate operands

if (!is\_valid\_operand(op1\_str) || !is\_valid\_operand(op2\_str)) {

raise\_interrupt("Invalid operand detected");

}

// Convert operands to integers

op1 = atoi(op1\_str);

op2 = atoi(op2\_str);

// Handle different operations

switch (operator) {

case '+':

printf("Result: %d + %d = %d\n", op1, op2, op1 + op2);

break;

case '-':

printf("Result: %d - %d = %d\n", op1, op2, op1 - op2);

break;

case '\*':

printf("Result: %d \* %d = %d\n", op1, op2, op1 \* op2);

break;

case '/':

// Division by zero check

if (op2 == 0) {

raise\_interrupt("Error: Division by zero");

}

printf("Result: %d / %d = %.2f\n", op1, op2, (float)op1 / op2);

break;

default:

raise\_interrupt("Invalid operator detected");

}

}

int main() {

char op1\_str[100], op2\_str[100], operator;

printf("Enter the first operand: ");

scanf("%s", op1\_str);

printf("Enter the operator (+, -, \*, /): ");

scanf(" %c", &operator); // Space before %c to ignore previous newline

printf("Enter the second operand: ");

scanf("%s", op2\_str);

// Perform the operation and handle errors

perform\_operation(op1\_str, op2\_str, operator);

return 0;

}

Write a program to check for opcode errors in a given job and raise an interrupt if an error is detected.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// List of valid opcodes

const char \*valid\_opcodes[] = {"ADD", "SUB", "MUL", "DIV"};

// Function to simulate raising an interrupt (error handling)

void raise\_interrupt(const char \*error\_message) {

printf("Interrupt Raised: %s\n", error\_message);

// Simulate stopping the job by exiting the program

exit(1);

}

// Function to check if the opcode is valid

int is\_valid\_opcode(const char \*opcode) {

// Check each valid opcode

for (int i = 0; i < sizeof(valid\_opcodes) / sizeof(valid\_opcodes[0]); i++) {

if (strcmp(opcode, valid\_opcodes[i]) == 0) {

return 1; // Valid opcode

}

}

return 0; // Invalid opcode

}

// Function to perform an operation based on the opcode

void perform\_operation(const char \*opcode, int operand1, int operand2) {

// Check if the opcode is valid

if (!is\_valid\_opcode(opcode)) {

raise\_interrupt("Invalid opcode detected");

}

// Perform the operation

if (strcmp(opcode, "ADD") == 0) {

printf("Result of %d + %d = %d\n", operand1, operand2, operand1 + operand2);

} else if (strcmp(opcode, "SUB") == 0) {

printf("Result of %d - %d = %d\n", operand1, operand2, operand1 - operand2);

} else if (strcmp(opcode, "MUL") == 0) {

printf("Result of %d \* %d = %d\n", operand1, operand2, operand1 \* operand2);

} else if (strcmp(opcode, "DIV") == 0) {

if (operand2 == 0) {

raise\_interrupt("Error: Division by zero");

}

printf("Result of %d / %d = %.2f\n", operand1, operand2, (float)operand1 / operand2);

}

}

int main() {

char opcode[10];

int operand1, operand2;

// Read the opcode and operands from the user

printf("Enter opcode (ADD, SUB, MUL, DIV): ");

scanf("%s", opcode);

printf("Enter the first operand: ");

scanf("%d", &operand1);

printf("Enter the second operand: ");

scanf("%d", &operand2);

// Perform the operation

perform\_operation(opcode, operand1, operand2);

return 0;

}

Write a program to load an Assembly Language Program (ALP) from an input file into main memory.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_MEMORY\_SIZE 1024 // Define the size of the memory

// Function to load the ALP into memory from a file

int load\_ALP(const char \*filename, char memory[MAX\_MEMORY\_SIZE][256]) {

FILE \*file = fopen(filename, "r"); // Open the file in read mode

if (file == NULL) {

perror("Failed to open file");

return -1;

}

int address = 0;

// Read each line (instruction) from the file and load into memory

while (fgets(memory[address], 256, file) != NULL) {

// Remove the newline character at the end of each line (if any)

memory[address][strcspn(memory[address], "\n")] = '\0';

// Move to the next memory address

address++;

// Check if memory exceeds the max size

if (address >= MAX\_MEMORY\_SIZE) {

printf("Memory limit exceeded. Cannot load more instructions.\n");

break;

}

}

fclose(file); // Close the file

return 0; // Successful loading

}

// Function to display the ALP loaded in memory

void display\_memory(char memory[MAX\_MEMORY\_SIZE][256]) {

printf("\nLoaded Assembly Language Program (ALP):\n");

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

if (memory[i][0] != '\0') { // If memory is not empty

printf("Memory[%d]: %s\n", i, memory[i]);

} else {

break; // Stop when memory is empty

}

}

}

int main() {

char memory[MAX\_MEMORY\_SIZE][256] = {0}; // Initialize memory to empty strings

// Specify the input ALP file

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

// Load the ALP from the file into memory

if (load\_ALP(filename, memory) == 0) {

// Display the ALP loaded into memory

display\_memory(memory);

} else {

printf("Error: Unable to load ALP from the file.\n");

}

return 0;

}

Write a program to simulate FCFS disk scheduling, calculate the total seek time, and print accepted input and output in tabular format.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 10

// Function to calculate the total seek time and display the output

void calculate\_fcfs(int requests[], int n, int initial\_position) {

int total\_seek\_time = 0;

int current\_position = initial\_position;

// Print the header for the table

printf("\nTrack Access Sequence and Seek Time (FCFS Disk Scheduling)\n");

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

printf("Request No.\tRequested Track\tSeek Time\tCumulative Seek Time\n");

// Process each request in the order of arrival (FCFS)

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

int seek\_time = abs(requests[i] - current\_position); // Calculate seek time

total\_seek\_time += seek\_time; // Add the seek time to the total

// Print the table row

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, requests[i], seek\_time, total\_seek\_time);

// Update the current position of the disk arm

current\_position = requests[i];

}

// Print the total seek time

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int requests[MAX\_REQUESTS]; // Array to store disk track requests

int n; // Number of requests

int initial\_position; // Starting position of the disk arm

// Take input for the number of requests

printf("Enter the number of disk requests (max %d): ", MAX\_REQUESTS);

scanf("%d", &n);

// Take input for the requests

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

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

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

}

// Take input for the initial position of the disk arm

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

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

// Call the FCFS function to calculate seek time and display the results

calculate\_fcfs(requests, n, initial\_position);

return 0;

}

write a program to simulate SSTF disk scheduling, calculate the total seek time, and print accepted input and output in tabular format.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 10

// Function to calculate the total seek time using SSTF

void calculate\_sstf(int requests[], int n, int initial\_position) {

int total\_seek\_time = 0;

int current\_position = initial\_position;

int visited[MAX\_REQUESTS] = {0}; // Array to keep track of visited requests

int completed\_requests = 0;

printf("\nTrack Access Sequence and Seek Time (SSTF Disk Scheduling)\n");

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

printf("Request No.\tRequested Track\tSeek Time\tCumulative Seek Time\n");

while (completed\_requests < n) {

int min\_seek\_time = \_\_INT\_MAX\_\_;

int closest\_request = -1;

int seek\_time = 0;

// Find the closest request

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

if (!visited[i]) {

seek\_time = abs(requests[i] - current\_position);

if (seek\_time < min\_seek\_time) {

min\_seek\_time = seek\_time;

closest\_request = i;

}

}

}

// Update the total seek time

total\_seek\_time += min\_seek\_time;

// Mark the current request as visited

visited[closest\_request] = 1;

// Print the result for this request

printf("%d\t\t%d\t\t%d\t\t%d\n", completed\_requests + 1, requests[closest\_request], min\_seek\_time, total\_seek\_time);

// Move the disk arm to the new position

current\_position = requests[closest\_request];

completed\_requests++;

}

// Print the total seek time

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

int main() {

int requests[MAX\_REQUESTS]; // Array to store disk track requests

int n; // Number of requests

int initial\_position; // Starting position of the disk arm

// Take input for the number of requests

printf("Enter the number of disk requests (max %d): ", MAX\_REQUESTS);

scanf("%d", &n);

// Take input for the disk track requests

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

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

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

}

// Take input for the initial position of the disk arm

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

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

// Call the SSTF function to calculate seek time and display the results

calculate\_sstf(requests, n, initial\_position);

return 0;

}

 Write a program to simulate SCAN disk scheduling, calculate the total seek time, and print accepted input and output in tabular format.

 #include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 10

// Function to calculate the total seek time using SCAN

void calculate\_scan(int requests[], int n, int initial\_position, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = initial\_position;

// Sort the requests in ascending order

int sorted\_requests[MAX\_REQUESTS];

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

sorted\_requests[i] = requests[i];

}

qsort(sorted\_requests, n, sizeof(int), (int (\*)(const void \*, const void \*))compare);

// Print the header for the table

printf("\nTrack Access Sequence and Seek Time (SCAN Disk Scheduling)\n");

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

printf("Request No.\tRequested Track\tSeek Time\tCumulative Seek Time\n");

// First handle the direction based on the initial position of the disk arm

int direction = (current\_position < sorted\_requests[0]) ? 1 : -1; // 1 -> Right, -1 -> Left

if (direction == 1) {

// Move to the right and then reverse direction

// Service all requests from current position to the maximum track

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

if (sorted\_requests[i] >= current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

// Reverse direction

int seek\_time = abs(disk\_size - 1 - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 1, disk\_size - 1, seek\_time, total\_seek\_time);

current\_position = disk\_size - 1;

// Service requests from the maximum track to the minimum track

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

if (sorted\_requests[i] < current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 2 + (n - 1 - i), sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

} else {

// Move to the left and then reverse direction

// Service all requests from current position to the minimum track

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

if (sorted\_requests[i] <= current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n - i, sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

// Reverse direction

int seek\_time = abs(0 - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 1, 0, seek\_time, total\_seek\_time);

current\_position = 0;

// Service requests from the minimum track to the maximum track

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

if (sorted\_requests[i] > current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 2 + i, sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

}

// Print the total seek time

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

// Comparator function for qsort to sort the disk requests

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int main() {

int requests[MAX\_REQUESTS]; // Array to store disk track requests

int n; // Number of requests

int initial\_position; // Starting position of the disk arm

int disk\_size; // The size of the disk (number of tracks)

// Take input for the number of requests

printf("Enter the number of disk requests (max %d): ", MAX\_REQUESTS);

scanf("%d", &n);

// Take input for the disk track requests

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

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

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

}

// Take input for the initial position of the disk arm

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

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

// Take input for the disk size (the number of tracks on the disk)

printf("Enter the size of the disk (number of tracks): ");

scanf("%d", &disk\_size);

// Call the SCAN function to calculate seek time and display the results

calculate\_scan(requests, n, initial\_position, disk\_size);

return 0;

}

Write a program to simulate C-SCAN disk scheduling, calculate the total seek time, and print accepted input and output in tabular format.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_REQUESTS 10

// Function to calculate the total seek time using C-SCAN

void calculate\_cscan(int requests[], int n, int initial\_position, int disk\_size) {

int total\_seek\_time = 0;

int current\_position = initial\_position;

// Sort the requests in ascending order

int sorted\_requests[MAX\_REQUESTS];

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

sorted\_requests[i] = requests[i];

}

qsort(sorted\_requests, n, sizeof(int), (int (\*)(const void \*, const void \*))compare);

// Print the header for the table

printf("\nTrack Access Sequence and Seek Time (C-SCAN Disk Scheduling)\n");

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

printf("Request No.\tRequested Track\tSeek Time\tCumulative Seek Time\n");

// Service all requests in the current direction (from initial\_position to the max)

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

if (sorted\_requests[i] >= current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", i + 1, sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

// If the disk arm reached the end of the disk, jump to the beginning (track 0)

int seek\_time = abs(disk\_size - 1 - current\_position); // From current position to the maximum track

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 1, disk\_size - 1, seek\_time, total\_seek\_time);

current\_position = disk\_size - 1;

// Jump to the first track (track 0)

seek\_time = abs(disk\_size - 1 - 0); // From current position (max track) to the first track (0)

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 2, 0, seek\_time, total\_seek\_time);

current\_position = 0;

// Now service all requests from the beginning (track 0) in the same direction

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

if (sorted\_requests[i] < current\_position) {

int seek\_time = abs(sorted\_requests[i] - current\_position);

total\_seek\_time += seek\_time;

printf("%d\t\t%d\t\t%d\t\t%d\n", n + 3 + i, sorted\_requests[i], seek\_time, total\_seek\_time);

current\_position = sorted\_requests[i];

}

}

// Print the total seek time

printf("\nTotal Seek Time: %d\n", total\_seek\_time);

}

// Comparator function for qsort to sort the disk requests

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b);

}

int main() {

int requests[MAX\_REQUESTS]; // Array to store disk track requests

int n; // Number of requests

int initial\_position; // Starting position of the disk arm

int disk\_size; // The size of the disk (number of tracks)

// Take input for the number of requests

printf("Enter the number of disk requests (max %d): ", MAX\_REQUESTS);

scanf("%d", &n);

// Take input for the disk track requests

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

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

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

}

// Take input for the initial position of the disk arm

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

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

// Take input for the disk size (the number of tracks on the disk)

printf("Enter the size of the disk (number of tracks): ");

scanf("%d", &disk\_size);

// Call the C-SCAN function to calculate seek time and display the results

calculate\_cscan(requests, n, initial\_position, disk\_size);

return 0;

}

**Implementation of Classical problem Reader-writer using Threads and Mutex**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <unistd.h>

#include <time.h>

#include <string.h>

#define NUM\_READERS 3

#define NUM\_WRITERS 2

// Shared variables

int read\_count = 0;

pthread\_mutex\_t buffer\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cv\_reader = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t cv\_writer = PTHREAD\_COND\_INITIALIZER;

// Helper function to print action, reader/writer, state, and comments

void print\_table(const char\* action, const char\* reader\_writer, const char\* state, const char\* comment) {

printf("| %-22s| %-15s| %-20s| %-35s|\n", action, reader\_writer, state, comment);

}

void\* reader(void\* arg) {

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

while (1) {

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

// Entry section

if (read\_count == 0) {

pthread\_cond\_wait(&cv\_writer, &buffer\_mutex); // First reader waits for writers

}

read\_count++; // Increment reader count

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

print\_table("Reader is reading", "Reader", "Critical Section", "First reader entered");

usleep(rand() % 500000 + 100000); // Simulate reading

pthread\_mutex\_lock(&buffer\_mutex); // Lock before updating reader count

read\_count--;

if (read\_count == 0) {

pthread\_cond\_signal(&cv\_writer); // Last reader notifies writer

}

pthread\_mutex\_unlock(&buffer\_mutex); // Unlock after updating

print\_table("Reader finished reading", "Reader", "Finished", "Reader finished reading");

usleep(rand() % 1000000 + 500000); // Simulate delay

}

return NULL;

}

void\* writer(void\* arg) {

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

while (1) {

print\_table("Writer is trying", "Writer", "Waiting", "Waiting for readers to finish");

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

// Entry section

pthread\_cond\_wait(&cv\_writer, &buffer\_mutex); // Wait until no readers are present

// Critical section

print\_table("Writer is writing", "Writer", "Critical Section", "Writer entered");

usleep(rand() % 500000 + 100000); // Simulate writing

pthread\_mutex\_unlock(&buffer\_mutex); // Unlock after writing

print\_table("Writer finished writing", "Writer", "Finished", "Writer finished writing");

usleep(rand() % 2000000 + 1000000); // Simulate delay

}

return NULL;

}

int main() {

srand(time(0));

// Print table header

printf("| %-22s| %-15s| %-20s| %-35s|\n", "Action", "Reader/Writer", "State", "Comment");

printf("%s\n", "---------------------------------------------------------------------------------------------");

// Creating reader and writer threads

pthread\_t readers[NUM\_READERS], writers[NUM\_WRITERS];

int reader\_ids[NUM\_READERS], writer\_ids[NUM\_WRITERS];

// Create reader threads

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

reader\_ids[i] = i;

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

}

// Create writer threads

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

writer\_ids[i] = i;

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

}

// Wait for threads to finish (they run indefinitely in this case)

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);

}

return 0;

}

**)Implementation of Classical problem  Reader-writer using Threads and Semaphore**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <semaphore.h>

#include <time.h>

// Shared variables

int read\_count = 0;

sem\_t read\_count\_sem; // Semaphore for protecting read\_count

sem\_t write\_sem; // Semaphore for writer access

// Helper function to print table-like formatted output

void print\_table(const char\* action, const char\* reader\_writer, const char\* state, const char\* comment) {

printf("| %-22s| %-15s| %-20s| %-35s|\n", action, reader\_writer, state, comment);

}

void\* reader(void\* arg) {

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

while (1) {

// Entry section

sem\_wait(&read\_count\_sem); // Lock to update read\_count

read\_count++;

if (read\_count == 1) {

sem\_wait(&write\_sem); // First reader locks the writer

}

sem\_post(&read\_count\_sem); // Unlock read\_count\_sem

// Critical section

char action[50];

sprintf(action, "Reader %d is reading", reader\_id);

print\_table(action, "Reader", "Critical Section", "First reader entered");

usleep((rand() % 500 + 100) \* 1000); // Simulate reading

// Exit section

sem\_wait(&read\_count\_sem); // Lock to update read\_count

read\_count--;

if (read\_count == 0) {

sem\_post(&write\_sem); // Last reader unlocks the writer

}

sem\_post(&read\_count\_sem); // Unlock read\_count\_sem

sprintf(action, "Reader %d finished reading", reader\_id);

print\_table(action, "Reader", "Finished", "Reader finished reading");

usleep((rand() % 1000 + 500) \* 1000); // Simulate delay

}

return NULL;

}

void\* writer(void\* arg) {

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

while (1) {

print\_table("Writer is trying", "Writer", "Waiting", "Waiting for readers to finish");

// Entry section

sem\_wait(&write\_sem); // Only one writer can access the critical section at a time

// Critical section

char action[50];

sprintf(action, "Writer %d is writing", writer\_id);

print\_table(action, "Writer", "Critical Section", "Writer entered");

usleep((rand() % 500 + 100) \* 1000); // Simulate writing

// Exit section

sem\_post(&write\_sem);

sprintf(action, "Writer %d finished writing", writer\_id);

print\_table(action, "Writer", "Finished", "Writer finished writing");

usleep((rand() % 2000 + 1000) \* 1000); // Simulate delay

}

return NULL;

}

int main() {

srand(time(NULL));

// Initialize semaphores

sem\_init(&read\_count\_sem, 0, 1); // Semaphore for read\_count (binary semaphore)

sem\_init(&write\_sem, 0, 1); // Semaphore for writer access

// Print table header

printf("| %-22s| %-15s| %-20s| %-35s|\n", "Action", "Reader/Writer", "State", "Comment");

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

// Creating reader and writer threads

pthread\_t readers[3], writers[2];

int reader\_ids[3], writer\_ids[2];

for (int i = 0; i < 3; i++) {

reader\_ids[i] = i;

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

}

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

writer\_ids[i] = i;

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

}

// Joining threads (for demonstration purposes, these will run indefinitely)

for (int i = 0; i < 3; i++) {

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

}

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

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

}

// Destroy semaphores

sem\_destroy(&read\_count\_sem);

sem\_destroy(&write\_sem);

return 0;

}

1. **Implementation of Classical problem Producer Consumer  using Threads and Semaphore**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <semaphore.h>

#include <time.h>

// Shared variables

int read\_count = 0;

sem\_t read\_count\_sem; // Semaphore for protecting read\_count

sem\_t write\_sem; // Semaphore for writer access

// Helper function to print table-like formatted output

void print\_table(const char\* action, const char\* reader\_writer, const char\* state, const char\* comment) {

printf("| %-22s| %-15s| %-20s| %-35s|\n", action, reader\_writer, state, comment);

}

void\* reader(void\* arg) {

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

while (1) {

// Entry section

sem\_wait(&read\_count\_sem); // Lock to update read\_count

read\_count++;

if (read\_count == 1) {

sem\_wait(&write\_sem); // First reader locks the writer

}

sem\_post(&read\_count\_sem); // Unlock read\_count\_sem

// Critical section

char action[50];

sprintf(action, "Reader %d is reading", reader\_id);

print\_table(action, "Reader", "Critical Section", "First reader entered");

usleep((rand() % 500 + 100) \* 1000); // Simulate reading

// Exit section

sem\_wait(&read\_count\_sem); // Lock to update read\_count

read\_count--;

if (read\_count == 0) {

sem\_post(&write\_sem); // Last reader unlocks the writer

}

sem\_post(&read\_count\_sem); // Unlock read\_count\_sem

sprintf(action, "Reader %d finished reading", reader\_id);

print\_table(action, "Reader", "Finished", "Reader finished reading");

usleep((rand() % 1000 + 500) \* 1000); // Simulate delay

}

return NULL;

}

void\* writer(void\* arg) {

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

while (1) {

print\_table("Writer is trying", "Writer", "Waiting", "Waiting for readers to finish");

// Entry section

sem\_wait(&write\_sem); // Only one writer can access the critical section at a time

// Critical section

char action[50];

sprintf(action, "Writer %d is writing", writer\_id);

print\_table(action, "Writer", "Critical Section", "Writer entered");

usleep((rand() % 500 + 100) \* 1000); // Simulate writing

// Exit section

sem\_post(&write\_sem);

sprintf(action, "Writer %d finished writing", writer\_id);

print\_table(action, "Writer", "Finished", "Writer finished writing");

usleep((rand() % 2000 + 1000) \* 1000); // Simulate delay

}

return NULL;

}

int main() {

srand(time(NULL));

// Initialize semaphores

sem\_init(&read\_count\_sem, 0, 1); // Semaphore for read\_count (binary semaphore)

sem\_init(&write\_sem, 0, 1); // Semaphore for writer access

// Print table header

printf("| %-22s| %-15s| %-20s| %-35s|\n", "Action", "Reader/Writer", "State", "Comment");

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

// Creating reader and writer threads

pthread\_t readers[3], writers[2];

int reader\_ids[3], writer\_ids[2];

for (int i = 0; i < 3; i++) {

reader\_ids[i] = i;

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

}

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

writer\_ids[i] = i;

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

}

// Joining threads (for demonstration purposes, these will run indefinitely)

for (int i = 0; i < 3; i++) {

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

}

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

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

}

// Destroy semaphores

sem\_destroy(&read\_count\_sem);

sem\_destroy(&write\_sem);

return 0;

}

1. **Implementation of Classical problem Producer Consumer using Threads and Mutex**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <queue>

#define BUFFER\_SIZE 5

// Shared buffer and synchronization primitives

std::queue<int> buffer;

pthread\_mutex\_t buffer\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t cv\_producer = PTHREAD\_COND\_INITIALIZER, cv\_consumer = PTHREAD\_COND\_INITIALIZER;

// Helper function to print table-like formatted output

void print\_table(const char\* action, const char\* producer\_consumer, int item) {

printf("| %-22s| %-15s| %-5d|\n", action, producer\_consumer, item);

}

void\* producer(void\* arg) {

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

int item = 0;

while (1) {

item++;

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

// Wait if the buffer is full

while (buffer.size() >= BUFFER\_SIZE) {

pthread\_cond\_wait(&cv\_producer, &buffer\_mutex);

}

// Critical section (producing)

buffer.push(item);

print\_table("Produced item", "Producer", item);

pthread\_mutex\_unlock(&buffer\_mutex); // Unlock before notifying

pthread\_cond\_signal(&cv\_consumer); // Notify consumer that an item is available

usleep((rand() % 1000 + 500) \* 1000); // Simulate delay

}

return NULL;

}

void\* consumer(void\* arg) {

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

while (1) {

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

// Wait if the buffer is empty

while (buffer.empty()) {

pthread\_cond\_wait(&cv\_consumer, &buffer\_mutex);

}

// Critical section (consuming)

int item = buffer.front();

buffer.pop();

print\_table("Consumed item", "Consumer", item);

pthread\_mutex\_unlock(&buffer\_mutex); // Unlock before notifying

pthread\_cond\_signal(&cv\_producer); // Notify producer that a slot is available

usleep((rand() % 1000 + 500) \* 1000); // Simulate delay

}

return NULL;

}

int main() {

srand(time(NULL));

// Print table header

printf("| %-22s| %-15s| %-5s|\n", "Action", "Producer/Consumer", "Item");

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

// Creating producer and consumer threads

pthread\_t producers[2], consumers[2];

int producer\_ids[2], consumer\_ids[2];

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

producer\_ids[i] = i;

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

}

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

consumer\_ids[i] = i;

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

}

// Joining threads (for demonstration purposes, these will run indefinitely)

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

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

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

}

return 0;

}