**LAB EXPERIMENTS**

11.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

// Function for the first thread

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

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

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

sleep(1); // Sleep to simulate work

}

return NULL;

}

// Function for the second thread

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

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

printf("Letter: %c\n", c);

sleep(1); // Sleep to simulate work

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

// Create threads

pthread\_create(&thread1, NULL, print\_numbers, NULL);

pthread\_create(&thread2, NULL, print\_letters, NULL);

// Wait for both threads to finish

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

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

return 0;

}

OUTPUT :

Number: 1

Letter: A

Number: 2

Letter: B

Number: 3

Letter: C

Number: 4

Letter: D

Number: 5

Letter: E

Both threads have finished execution.

12.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

#define NUM\_PHILOSOPHERS 5

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

void\* philosopher(void\* num) {

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

int left = id;

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

for (int i = 0; i < 3; i++) { // Each philosopher eats 3 times

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

sleep(1); // Thinking

// Pick up forks (use lower-numbered fork first to avoid deadlock)

if (id % 2 == 0) {

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

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

} else {

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

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

}

// Eating

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

sleep(2);

// Put down forks

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

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

}

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

return NULL;

}

int main() {

pthread\_t philosophers[NUM\_PHILOSOPHERS];

int ids[NUM\_PHILOSOPHERS];

// Initialize mutexes

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

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

// Create threads

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

ids[i] = i;

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

}

// Wait for threads to finish

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

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

// Destroy mutexes

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

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

printf("All philosophers have finished.\n");

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 2 is eating...

Philosopher 4 is eating...

Philosopher 0 is thinking...

Philosopher 2 is thinking...

...

Philosopher 3 is done eating.

All philosophers have finished.

13.

#include <stdio.h>

#define MAX\_BLOCKS 50

#define MAX\_PROCESSES 50

// Global variables

int memory\_blocks[MAX\_BLOCKS], block\_count;

int processes[MAX\_PROCESSES], process\_count;

int allocation[MAX\_PROCESSES];

// Function to reset allocation array

void reset\_allocation() {

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

allocation[i] = -1;

}

}

// First Fit Allocation

void first\_fit() {

reset\_allocation();

int temp\_blocks[MAX\_BLOCKS];

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

temp\_blocks[i] = memory\_blocks[i];

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

for (int j = 0; j < block\_count; j++) {

if (temp\_blocks[j] >= processes[i]) {

allocation[i] = j;

temp\_blocks[j] -= processes[i];

break;

}

}

}

}

// Best Fit Allocation

void best\_fit() {

reset\_allocation();

int temp\_blocks[MAX\_BLOCKS];

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

temp\_blocks[i] = memory\_blocks[i];

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

int best\_index = -1;

for (int j = 0; j < block\_count; j++) {

if (temp\_blocks[j] >= processes[i]) {

if (best\_index == -1 || temp\_blocks[j] < temp\_blocks[best\_index])

best\_index = j;

}

}

if (best\_index != -1) {

allocation[i] = best\_index;

temp\_blocks[best\_index] -= processes[i];

}

}

}

// Worst Fit Allocation

void worst\_fit() {

reset\_allocation();

int temp\_blocks[MAX\_BLOCKS];

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

temp\_blocks[i] = memory\_blocks[i];

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

int worst\_index = -1;

for (int j = 0; j < block\_count; j++) {

if (temp\_blocks[j] >= processes[i]) {

if (worst\_index == -1 || temp\_blocks[j] > temp\_blocks[worst\_index])

worst\_index = j;

}

}

if (worst\_index != -1) {

allocation[i] = worst\_index;

temp\_blocks[worst\_index] -= processes[i];

}

}

}

// Display allocation

void display\_allocation() {

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

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

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

if (allocation[i] != -1)

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

else

printf("Not Allocated\n");

}

}

int main() {

int choice;

printf("Enter number of memory blocks: ");

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

printf("Enter sizes of memory blocks:\n");

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

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

scanf("%d", &memory\_blocks[i]);

}

printf("Enter number of processes: ");

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

printf("Enter sizes of processes:\n");

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

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

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

}

do {

printf("\nMemory Allocation Strategies:\n");

printf("1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

first\_fit();

printf("\n--- First Fit Allocation ---");

display\_allocation();

break;

case 2:

best\_fit();

printf("\n--- Best Fit Allocation ---");

display\_allocation();

break;

case 3:

worst\_fit();

printf("\n--- Worst Fit Allocation ---");

display\_allocation();

break;

case 4:

printf("Exiting...\n");

break;

default:

printf("Invalid choice!\n");

}

} while (choice != 4);

return 0;

}

OUTPUT :

Enter number of memory blocks: 3

Block 1: 100

Block 2: 500

Block 3: 200

Enter number of processes: 3

Process 1: 212

Process 2: 417

Process 3: 112

Memory Allocation Strategies:

1. First Fit

2. Best Fit

3. Worst Fit

4. Exit

Enter your choice: 1

--- First Fit Allocation ---

Process No. Process Size Block No.

1 212 2

2 417 Not Allocated

3 112 2

14.

.model small

.stack 100h

.data

file\_count db 0

file\_names db 5 dup(11 dup('$')) ; support 5 files, 10 chars max each

menu db 13,10,"1. Create File",13,10

db "2. List Files",13,10

db "3. Exit",13,10

db "Enter choice: $"

prompt\_name db 13,10,"Enter file name: $"

file\_added db 13,10,"File created.$"

file\_limit db 13,10,"Directory full!$"

no\_files db 13,10,"No files to display.$"

input\_name db 11 dup('$')

.code

main:

mov ax, @data

mov ds, ax

menu\_loop:

lea dx, menu

call print\_str

call read\_char

cmp al, '1'

je create\_file

cmp al, '2'

je list\_files

cmp al, '3'

je exit\_program

jmp menu\_loop

; --- CREATE FILE ---

create\_file:

mov al, file\_count

cmp al, 5

jae directory\_full

lea dx, prompt\_name

call print\_str

; Read string

lea dx, input\_name

call read\_string

; Store input\_name into file\_names[file\_count]

mov si, offset input\_name

mov bl, file\_count

mov bh, 0

mov cx, 11

mov di, offset file\_names

mov ax, bx

mov bx, 11

mul bx

add di, ax

copy\_loop:

lodsb

stosb

loop copy\_loop

inc file\_count

lea dx, file\_added

call print\_str

jmp menu\_loop

directory\_full:

lea dx, file\_limit

call print\_str

jmp menu\_loop

; --- LIST FILES ---

list\_files:

cmp file\_count, 0

je no\_file\_msg

mov cx, 0

list\_loop:

mov bx, cx

mov ax, bx

mov bx, 11

mul bx

mov si, offset file\_names

add si, ax

mov dx, si

call print\_str

; Print newline

mov ah, 02h

mov dl, 13

int 21h

mov dl, 10

int 21h

inc cx

cmp cx, file\_count

jne list\_loop

jmp menu\_loop

no\_file\_msg:

lea dx, no\_files

call print\_str

jmp menu\_loop

; --- EXIT PROGRAM ---

exit\_program:

mov ah, 4ch

int 21h

; --- Helper: Print string at DX ---

print\_str:

mov ah, 09h

int 21h

ret

; --- Helper: Read string to DS:DX (terminated with enter) ---

read\_string:

; Uses simple 10-character string input with '$' terminator

mov cx, 0

read\_loop:

mov ah, 01h

int 21h

cmp al, 13 ; Enter key

je done\_read

mov [dx], al

inc dx

inc cx

cmp cx, 10

je done\_read

jmp read\_loop

done\_read:

mov al, '$'

mov [dx], al

ret

; --- Helper: Read char into AL ---

read\_char:

mov ah, 01h

output :

1. Create File

2. List Files

3. Exit

Enter choice: 1

Enter file name: file1

File created.

15.

#include <stdio.h>

#include <string.h>

#define MAX\_FILES 100

#define NAME\_SIZE 30

// File directory entry

struct File {

char name[NAME\_SIZE];

};

struct File directory[MAX\_FILES];

int fileCount = 0;

// Function declarations

void createFile();

void deleteFile();

void searchFile();

void listFiles();

int main() {

int choice;

while (1) {

printf("\n----- Single-Level Directory -----\n");

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

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

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

printf("4. List Files\n");

printf("5. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

getchar(); // to consume newline character

switch (choice) {

case 1: createFile(); break;

case 2: deleteFile(); break;

case 3: searchFile(); break;

case 4: listFiles(); break;

case 5:

printf("Exiting...\n");

return 0;

default:

printf("Invalid choice.\n");

}

}

return 0;

}

// Create a new file

void createFile() {

if (fileCount >= MAX\_FILES) {

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

return;

}

char fname[NAME\_SIZE];

printf("Enter file name: ");

fgets(fname, NAME\_SIZE, stdin);

fname[strcspn(fname, "\n")] = '\0'; // Remove newline

// Check if file already exists

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

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

printf("File already exists.\n");

return;

}

}

strcpy(directory[fileCount].name, fname);

fileCount++;

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

}

// Delete a file

void deleteFile() {

if (fileCount == 0) {

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

return;

}

char fname[NAME\_SIZE];

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

fgets(fname, NAME\_SIZE, stdin);

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

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

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

// Shift files to fill the gap

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

strcpy(directory[j].name, directory[j + 1].name);

}

fileCount--;

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

return;

}

}

printf("File not found.\n");

}

// Search for a file

void searchFile() {

char fname[NAME\_SIZE];

printf("Enter file name to search: ");

fgets(fname, NAME\_SIZE, stdin);

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

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

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

printf("File '%s' found at position %d.\n", fname, i + 1);

return;

}

}

printf("File not found.\n");

}

// List all files

void listFiles() {

if (fileCount == 0) {

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

return;

}

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

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

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

}

}

Output :

----- Single-Level Directory -----

1. Create File

2. Delete File

3. Search File

4. List Files

5. Exit

Enter your choice: 1

Enter file name: report.txt

File 'report.txt' created successfully.

Enter your choice: 4

Files in directory:

1. report.txt

16.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define FILE\_NAME "employee.dat"

#define MAX 100

// Structure for Employee

struct Employee {

int id;

char name[30];

float salary;

};

// Function declarations

void addEmployee();

void viewEmployee();

void updateEmployee();

int main() {

int choice;

while (1) {

printf("\n--- Employee Record System ---\n");

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

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

printf("3. Update Employee by ID\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1: addEmployee(); break;

case 2: viewEmployee(); break;

case 3: updateEmployee(); break;

case 4: exit(0);

default: printf("Invalid choice.\n");

}

}

return 0;

}

// Add employee record

void addEmployee() {

FILE \*fp;

struct Employee emp;

fp = fopen(FILE\_NAME, "ab"); // append in binary mode

if (fp == NULL) {

printf("Unable to open file.\n");

return;

}

printf("Enter Employee ID: ");

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

printf("Enter Employee Name: ");

getchar(); // consume newline

fgets(emp.name, sizeof(emp.name), stdin);

emp.name[strcspn(emp.name, "\n")] = '\0'; // remove newline

printf("Enter Employee Salary: ");

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

fwrite(&emp, sizeof(emp), 1, fp);

fclose(fp);

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

}

// View employee by ID (random access)

void viewEmployee() {

FILE \*fp;

struct Employee emp;

int id, found = 0;

fp = fopen(FILE\_NAME, "rb");

if (fp == NULL) {

printf("Unable to open file.\n");

return;

}

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

scanf("%d", &id);

while (fread(&emp, sizeof(emp), 1, fp)) {

if (emp.id == id) {

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

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

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

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

found = 1;

break;

}

}

if (!found)

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

fclose(fp);

}

// Update employee by ID (random access with fseek)

void updateEmployee() {

FILE \*fp;

struct Employee emp;

int id, found = 0;

long pos;

fp = fopen(FILE\_NAME, "rb+");

if (fp == NULL) {

printf("Unable to open file.\n");

return;

}

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

scanf("%d", &id);

while (fread(&emp, sizeof(emp), 1, fp)) {

if (emp.id == id) {

pos = ftell(fp) - sizeof(emp); // go back to start of the record

printf("Enter new name: ");

getchar(); // clear newline

fgets(emp.name, sizeof(emp.name), stdin);

emp.name[strcspn(emp.name, "\n")] = '\0';

printf("Enter new salary: ");

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

fseek(fp, pos, SEEK\_SET);

fwrite(&emp, sizeof(emp), 1, fp);

printf("Record updated successfully.\n");

found = 1;

break;

}

}

if (!found)

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

fclose(fp);

}

Return ;

}

Output :

--- Employee Record System ---

1. Add Employee

2. View Employee by ID

3. Update Employee by ID

4. Exit

Enter your choice: 1

Enter Employee ID: 101

Enter Employee Name: Alice

Enter Employee Salary: 55000

Employee added successfully.

17.

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

int n, m; // n = number of processes, m = number of resources

int available[MAX\_RESOURCES];

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

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

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

// Function declarations

bool isSafe();

void input();

int main() {

input();

if (isSafe()) {

printf("\nThe system is in a SAFE state. No deadlock.\n");

} else {

printf("\nThe system is in an UNSAFE state. Deadlock possible.\n");

}

return 0;

}

// Read inputs for available, max, allocation matrices

void input() {

printf("Enter number of processes: ");

scanf("%d", &n);

printf("Enter number of resource types: ");

scanf("%d", &m);

printf("\nEnter the Available resources:\n");

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

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

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

}

printf("\nEnter the Max Matrix (max demand of each process):\n");

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

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

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

printf(" Max for resource %d: ", j);

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

}

}

printf("\nEnter the Allocation Matrix (current allocation):\n");

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

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

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

printf(" Allocated for resource %d: ", j);

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

// Compute need matrix here itself

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

}

}

}

// Check if system is in safe state

bool isSafe() {

int work[MAX\_RESOURCES];

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

int safeSequence[MAX\_PROCESSES];

int count = 0;

// Initialize work = available

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

work[i] = available[i];

}

while (count < n) {

bool found = false;

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

if (!finish[i]) {

int j;

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

if (need[i][j] > work[j])

break;

}

if (j == m) { // If all needs <= work

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

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

safeSequence[count++] = i;

finish[i] = true;

found = true;

}

}

}

if (!found) {

return false; // System is not in a safe state

}

}

printf("\nSafe Sequence: ");

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

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

}

printf("\n");

return true;

}

Output :

Enter number of processes: 5

Enter number of resource types: 3

Available resources:

Resource 0: 3

Resource 1: 3

Resource 2: 2

Max Matrix:

P0: 7 5 3

P1: 3 2 2

P2: 9 0 2

P3: 2 2 2

P4: 4 3 3

Allocation Matrix:

P0: 0 1 0

P1: 2 0 0

P2: 3 0 2

P3: 2 1 1

P4: 0 0 2

18.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define BUFFER\_SIZE 5

int buffer[BUFFER\_SIZE];

int in = 0, out = 0;

sem\_t empty; // Counts empty slots

sem\_t full; // Counts filled slots

pthread\_mutex\_t mutex; // Ensures mutual exclusion

void\* producer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

item = rand() % 100;

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

pthread\_mutex\_lock(&mutex); // Enter critical section

buffer[in] = item;

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&full); // Signal full slot

sleep(1); // Simulate production time

}

return NULL;

}

void\* consumer(void\* arg) {

int item;

for (int i = 0; i < 10; i++) {

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

pthread\_mutex\_lock(&mutex); // Enter critical section

item = buffer[out];

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

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

pthread\_mutex\_unlock(&mutex); // Exit critical section

sem\_post(&empty); // Signal empty slot

sleep(2); // Simulate consumption time

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// Initialize semaphores and mutex

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

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

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

// Create producer and consumer threads

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

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

// Wait for threads to finish

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// Destroy semaphores and mutex

sem\_destroy(&empty);

sem\_destroy(&full);

pthread\_mutex\_destroy(&mutex);

return 0;

}

Output :

Producer produced: 42 at 0

Consumer consumed: 42 from 0

Producer produced: 13 at 1

Producer produced: 67 at 2

Consumer consumed: 13 from 1

...

19.

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

int shared\_counter = 0; // Shared resource

pthread\_mutex\_t lock; // Mutex lock

void\* increment(void\* arg) {

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

pthread\_mutex\_lock(&lock); // Lock before entering critical section

// --- Critical Section ---

int temp = shared\_counter;

printf("Thread %ld: read counter = %d\n", (long)pthread\_self(), temp);

temp++;

sleep(1); // Simulate work

shared\_counter = temp;

printf("Thread %ld: updated counter = %d\n", (long)pthread\_self(), shared\_counter);

// -------------------------

pthread\_mutex\_unlock(&lock); // Unlock after leaving critical section

sleep(1); // Simulate other operations

}

return NULL;

}

int main() {

pthread\_t t1, t2;

// Initialize mutex lock

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

printf("Mutex init failed\n");

return 1;

}

// Create threads

pthread\_create(&t1, NULL, increment, NULL);

pthread\_create(&t2, NULL, increment, NULL);

// Wait for threads to finish

pthread\_join(t1, NULL);

pthread\_join(t2, NULL);

// Destroy mutex

pthread\_mutex\_destroy(&lock);

printf("\nFinal value of shared\_counter = %d\n", shared\_counter);

return 0;

}

Output :

Thread 139748219955968: read counter = 0

Thread 139748219955968: updated counter = 1

Thread 139748211563264: read counter = 1

Thread 139748211563264: updated counter = 2

...

Final value of shared\_counter = 10

20 .

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define MAX\_READERS 5

sem\_t mutex; // Semaphore to protect access to read\_count

sem\_t write\_lock; // Semaphore to ensure mutual exclusion for writers

int read\_count = 0; // Shared read count to track active readers

// Function to simulate reading

void\* reader(void\* arg) {

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

// Entry section

sem\_wait(&mutex); // Lock for accessing read\_count

read\_count++; // Increment read\_count (number of readers)

if (read\_count == 1) {

sem\_wait(&write\_lock); // If it's the first reader, lock the writer

}

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

// Critical section (reading the shared resource)

printf("Reader %d is reading...\n", reader\_id);

sleep(1); // Simulating reading time

// Exit section

sem\_wait(&mutex); // Lock for accessing read\_count

read\_count--; // Decrement read\_count (one reader leaves)

if (read\_count == 0) {

sem\_post(&write\_lock); // If no readers left, allow writers to write

}

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

return NULL;

}

// Function to simulate writing

void\* writer(void\* arg) {

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

// Entry section (writer gets exclusive access)

sem\_wait(&write\_lock); // Wait for exclusive access to the resource

// Critical section (writing to the shared resource)

printf("Writer %d is writing...\n", writer\_id);

sleep(2); // Simulating writing time

// Exit section

sem\_post(&write\_lock); // Release exclusive access to the resource

return NULL;

}

int main() {

pthread\_t readers[MAX\_READERS], writers[MAX\_READERS];

int reader\_ids[MAX\_READERS], writer\_ids[MAX\_READERS];

// Initialize semaphores

sem\_init(&mutex, 0, 1); // Mutex for controlling read\_count access

sem\_init(&write\_lock, 0, 1); // Semaphore to allow only one writer at a time

// Create reader and writer threads

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

reader\_ids[i] = i + 1; // Assign unique ID to each reader

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

writer\_ids[i] = i + 1; // Assign unique ID to each writer

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

}

// Wait for all threads to complete

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

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

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

}

// Destroy semaphores

sem\_destroy(&mutex);

sem\_destroy(&write\_lock);

return 0;

}

Output :

Reader 1 is reading...

Reader 2 is reading...

Reader 3 is reading...

Reader 4 is reading...

Reader 5 is reading...

Writer 1 is writing...

Writer 2 is writing...