-:OS CA3 SIMULATION BASED ASSIGNMENT ANSWERS:-

ASSIGNMENT_1:-

Q: Consider the readers and writers problem as discussed above. We wish to implement synchronization between readers and writers, while giving preference to writers, where no waiting writer.

CODE:

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

// Variables for synchronization
pthread_mutex_t rw_lock;  // Mutex for controlling access to shared data
pthread_cond_t writer_cond; // Condition variable for waiting writers
int readers_count = 0; // Number of active readers

// Shared data
int shared_data = 0;

// Initialize the locks and condition variable
void initialize() {
    pthread_mutex_init(&rw_lock, NULL);
    pthread_cond_init(&writer_cond, NULL);
}

// Read lock function
void read_lock() {
    pthread_mutex_lock(&rw_lock);

    // Wait for writers if any writer is active
    while (readers_count == -1) {
        pthread_cond_wait(&writer_cond, &rw_lock);
    }

    // Increment the reader count
    readers_count++;

    pthread_mutex_unlock(&rw_lock);
}
```

```
// Read unlock function
void read_unlock() {
    pthread_mutex_lock(&rw_lock);

    // Decrement the reader count
    readers_count--;

    // If this is the last reader, signal waiting writers to proceed
    if (readers_count == 0) {
        pthread_cond_signal(&writer_cond);
    }

    pthread_mutex_unlock(&rw_lock);
}

// Write lock function
void write_lock() {
    pthread_mutex_lock(&rw_lock);

    // Wait if there are active readers or a writer
    while (readers_count > 0 || readers_count == -1) {
        pthread_cond_wait(&writer_cond, &rw_lock);
    }

    // Set the readers count to -1 to block new readers
    readers_count = -1;

    pthread_mutex_unlock(&rw_lock);
}

// Write unlock function
void write_unlock() {
    pthread_mutex_lock(&rw_lock);

    // Reset the readers count and signal waiting writers or readers
    readers_count = 0;
    pthread_cond_signal(&writer_cond);

    pthread_mutex_unlock(&rw_lock);
}

// Reader thread function
void* reader(void* arg) {
```

```
    while (1) {
        read_lock();
        printf("Reader %ld is reading: %d\n", (long)arg, shared_data);
        read_unlock();
        usleep(100000); // Simulate some work
    }
    return NULL;
}

// Writer thread function
void* writer(void* arg) {
    while (1) {
        write_lock();
        shared_data++;
        printf("Writer %ld is writing: %d\n", (long)arg, shared_data);
        write_unlock();
        usleep(200000); // Simulate some work
    }
    return NULL;
}

int main() {
    initialize();

    pthread_t readers[3];
    pthread_t writers[2];

    for (long i = 0; i < 3; i++) {
        pthread_create(&readers[i], NULL, reader, (void*)i);
    }

    for (long i = 0; i < 2; i++) {
        pthread_create(&writers[i], NULL, writer, (void*)i);
    }

    for (int i = 0; i < 3; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(writers[i], NULL);
    }
```

```
    return 0;
}
```

ASSIGNMENT_2:-

Q: Consider a clinic with one doctor and a very large waiting room (of infinite capacity). Any patient entering the clinic will wait in the waiting room until the doctor is free to see her. Similarly,the doctor also waits for a patient to arrive to treat.

CODE:

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

// Shared memory buffer
char shared_buffer[256]; // Assuming a buffer size of 256 for treatment details

// Semaphores
sem_t patient_ready; // Initialized to 0
sem_t doctor_ready;  // Initialized to 0

// Function to simulate a patient consulting the doctor
void consultDoctor(char* symptoms) {
    // Write patient's symptoms to the shared buffer
    snprintf(shared_buffer, sizeof(shared_buffer), "Symptoms: %s", symptoms);

    // Signal the doctor that the patient is ready
    sem_post(&patient_ready);

    // Wait for the doctor to finish the treatment
    sem_wait(&doctor_ready);
}

// Function to simulate the doctor treating the patient
void treatPatient() {
    // Wait for a patient to be ready
    sem_wait(&patient_ready);

    // Process the patient's symptoms and update the treatment
    // Assuming some treatment is performed here and updating the shared buffer
```

```
    snprintf(shared_buffer, sizeof(shared_buffer), "Treatment: Some treatment details");

    // Signal the patient that the treatment details are ready
    sem_post(&doctor_ready);
}

// Function for the patient to see the treatment details
void noteTreatment() {
    // Wait for the doctor to finish updating the treatment
    sem_wait(&doctor_ready);

    // Read and print the treatment details from the shared buffer
    printf("Patient received treatment: %s\n", shared_buffer);
}

// Function for the doctor process
void* doctorThread(void* arg) {
    // Simulate the doctor treating patients
    while (1) {
        treatPatient();
        // Perform treatment
    }
    return NULL;
}

// Function for the patient process
void* patientThread(void* arg) {
    char symptoms[128]; // Simulate patient's symptoms
    while (1) {
        // Generate or receive symptoms
        snprintf(symptoms, sizeof(symptoms), "Patient %ld symptoms", (long)arg);

        consultDoctor(symptoms);

        // Perform any actions after treatment
        noteTreatment();
    }
    return NULL;
}

int main() {
    // Initialize semaphores
    sem_init(&patient_ready, 0, 0);
```

```
    sem_init(&doctor_ready, 0, 0);

    // Create a doctor thread
    pthread_t doctor;
    pthread_create(&doctor, NULL, doctorThread, NULL);

    // Create multiple patient threads
    pthread_t patients[3];
    for (long i = 0; i < 3; i++) {
        pthread_create(&patients[i], NULL, patientThread, (void*)i);
    }

    // Join doctor and patient threads (or processes)
    pthread_join(doctor, NULL);
    for (long i = 0; i < 3; i++) {
        pthread_join(patients[i], NULL);
    }

    // Cleanup: Destroy semaphores
    sem_destroy(&patient_ready);
    sem_destroy(&doctor_ready);

    return 0;
}
```

ASSIGNMENT_3:-

Q: Consider a multithreaded banking application. The main process receives requests to tranfer money from one account to the other, and each request is handled by a separate worker thread in the application.

CODE:

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

// Structure representing a simple lock
typedef struct {
    pthread_mutex_t mutex;
} mylock;

// Structure representing a bank account
```

```c
struct account {
    int accountnum;
    int balance;
    mylock lock;
};

// Function to initialize a lock
void init_lock(mylock *lock) {
    pthread_mutex_init(&lock->mutex, NULL);
}

// Function to acquire the lock
void dolock(mylock *lock) {
    pthread_mutex_lock(&lock->mutex);
}

// Function to release the lock
void unlock(mylock *lock) {
    pthread_mutex_unlock(&lock->mutex);
}

// Function to transfer money from one account to another
void transfer(struct account *from, struct account *to, int amount) {
    // Acquire the locks in the proper order to avoid deadlocks
    if (from->accountnum < to->accountnum) {
        dolock(&from->lock);
        dolock(&to->lock);
    } else {
        dolock(&to->lock);
        dolock(&from->lock);
    }

    from->balance -= amount;
    to->balance += amount;

    // Release the locks
    unlock(&from->lock);
    unlock(&to->lock);
}

int main() {
    // Your main program logic goes here
    return 0;
```

}

ASSIGNMENT_4:-

Q:Consider a server program running in an online market place firm. The program receives buy and sell orders for one type of commodity from external clients.

CODE:

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

// Declare variables for synchronization
pthread_mutex_t mutex;            // Mutex for protecting shared data
pthread_cond_t sellArrived;       // Condition variable for signaling sell thread arrival
int isSellThreadArrived = 0;      // Flag to track whether a sell thread has arrived

// Function to be executed by the buy thread
void completeBuy(void) {
   // Implementation of the completeBuy function
   printf("Buy operation completed.\n");
}

void completeSell(void) {
   // Implementation of the completeSell function
   printf("Sell operation completed.\n");
}

// Function to be executed by the buy thread
void* buy_thread_function(void* arg) {
   // Start of synchronization logic
   pthread_mutex_lock(&mutex); // Lock the mutex to protect shared data

   // Check if a matching sell thread has already arrived
   while (!isSellThreadArrived) {
      // If no matching sell thread is present, wait for a signal
      pthread_cond_wait(&sellArrived, &mutex);
   }

   // At this point, a matching sell thread has arrived
   // You can now proceed with the transaction
   completeBuy();
```

```
    // End of synchronization logic
    pthread_mutex_unlock(&mutex); // Unlock the mutex

    // Other buy thread processing (if any)
    return NULL;
}

// Function to be executed by the sell thread (symmetric logic)
void* sell_thread_function(void* arg) {
    // ... (symmetric logic to buy_thread_function)
    pthread_mutex_lock(&mutex);
    isSellThreadArrived = 1; // Set the flag to indicate the arrival of a matching sell thread
    pthread_cond_signal(&sellArrived); // Signal the buy thread that a matching sell thread has
arrived
    pthread_mutex_unlock(&mutex);

    // You can now proceed with the transaction for the sell thread
    completeSell();

    // ...
    return NULL;
}

int main() {
    // Initialize mutex and condition variable
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&sellArrived, NULL);

    // Create and start the buy and sell threads
    pthread_t buy_thread, sell_thread;
    pthread_create(&buy_thread, NULL, buy_thread_function, NULL);
    pthread_create(&sell_thread, NULL, sell_thread_function, NULL);

    // Wait for the threads to finish (in your actual code, you might use a more sophisticated
method)
    pthread_join(buy_thread, NULL);
    pthread_join(sell_thread, NULL);

    // Cleanup resources
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&sellArrived);

    return 0;
```

```
}
```

ASSIGNMENT_5:-

Q: Consider the following classical synchronization problem called the barbershop problem. A barbershop consists of a room with N chairs.

CODE:

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

#define N 5 // Number of chairs in the waiting room

// Semaphores and shared variable
sem_t mutex;        // Mutex for protecting shared variables
sem_t customers;    // Semaphore to signal that a customer has arrived
sem_t barber;       // Semaphore to signal that the barber is ready
int waiting_count = 0;  // Number of customers waiting

// Function to simulate the barber giving a haircut
void cutHair() {
   // Barber is ready, notify a customer
   sem_post(&barber);
}

// Function to simulate a customer getting a haircut
void getHairCut() {
   // A customer is ready, notify the barber
   sem_post(&customers);

   // Wait for the barber to finish the haircut
   sem_wait(&barber);
}

// Function to handle the case when the waiting room is full
void leave() {
   // Customer leaves the shop
   printf("Customer leaves the shop.\n");
}
```

```
// Barber thread function
void* barberThread(void* arg) {
   while (1) {
      // Wait until a customer arrives
      sem_wait(&customers);

      // Cut the customer's hair
      cutHair();

      // Notify the customer that the haircut is complete
      printf("Barber: Haircut complete.\n");
   }
   return NULL;
}

// Customer thread function
void* customerThread(void* arg) {
   // Customer arrives
   printf("Customer arrives.\n");

   // Try to acquire the mutex to update the waiting_count
   sem_wait(&mutex);

   if (waiting_count < N) {
      // There are available chairs in the waiting room
      waiting_count++;
      sem_post(&customers); // Signal the barber

      // Release the mutex
      sem_post(&mutex);

      // Get a haircut
      getHairCut();
   } else {
      // The waiting room is full, customer leaves
      leave();
      sem_post(&mutex);
   }

   return NULL;
}

int main() {
```

```
    // Initialize semaphores and other shared variables
    sem_init(&mutex, 0, 1);
    sem_init(&customers, 0, 0);
    sem_init(&barber, 0, 0);

    pthread_t barber;
    pthread_t customers[N];  // Create customer threads (N can be the number of chairs in the
waiting room)

    // Create and start the barber thread
    pthread_create(&barber, NULL, barberThread, NULL);

    // Create and start customer threads
    for (int i = 0; i < N; i++) {
        pthread_create(&customers[i], NULL, customerThread, NULL);
    }

    // Join threads
    pthread_join(barber, NULL);
    for (int i = 0; i < N; i++) {
        pthread_join(customers[i], NULL);
    }

    return 0;
}
```

ASSIGNMENT_6:-

Q: Consider the following synchronization problem. A group of children are picking chocolates from a box that can hold up to N chocolates.

CODE:

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

int count = 0;
pthread_mutex_t m;
pthread_cond_t fullBox, emptyBox;

// Function to get chocolate from the box
void *childThread(void *arg) {
    while (1) {
```

```
        pthread_mutex_lock(&m);
        // Check if the box is empty
        while (count == 0) {
            // If the box is empty, wait for the mother to refill it
            pthread_cond_wait(&fullBox, &m);
        }

        // Consume one chocolate from the box
        count--;

        // Signal the mother that the box is not empty anymore
        pthread_cond_signal(&emptyBox);

        pthread_mutex_unlock(&m);

        // Eat the chocolate
        printf("Child: Eating chocolate\n");
    }
    return NULL;
}

// Function to refill the chocolate box
void *motherThread(void *arg) {
    while (1) {
        pthread_mutex_lock(&m);
        // Check if the box is not empty
        while (count > 0) {
            // If the box is not empty, wait for a child to consume chocolates
            pthread_cond_wait(&emptyBox, &m);
        }

        // Refill the box with N chocolates
        count = *((int *)arg);

        // Signal all waiting children that the box is full
        pthread_cond_broadcast(&fullBox);

        pthread_mutex_unlock(&m);

        // Refill the box with chocolates
        printf("Mother: Refilling the chocolate box with %d chocolates\n", count);
    }
    return NULL;
```

```
}

int main() {
    // Initialize mutex and condition variables
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&fullBox, NULL);
    pthread_cond_init(&emptyBox, NULL);

    // Create child and mother threads
    pthread_t child, mother;
    int chocolates = 5; // Number of chocolates to refill

    pthread_create(&child, NULL, childThread, NULL);
    pthread_create(&mother, NULL, motherThread, &chocolates);

    // The threads run indefinitely; you may need to use Ctrl+C to terminate the program.

    // Cleanup: Destroy mutex and condition variables (not reachable in this program)
    // pthread_mutex_destroy(&m);
    // pthread_cond_destroy(&fullBox);
    // pthread_cond_destroy(&emptyBox);

    pthread_exit(NULL); // This is used to keep the threads running
    return 0;
}
```

ASSIGNMENT_7:-

Q: You are given a list of disk requests in the order in which they were received and need to implement the First-Come, First-Served (FCFS) disk scheduling algorithm to simulate disk scheduling.

CODE:

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

int main() {
    int head_position, total_seek_time = 0;
    int num_requests;
```

```
// Prompt the user to enter the starting disk head position

printf("Enter the starting disk head position: ");
scanf("%d", &head_position);

// Prompt the user to enter the number of disk requests

printf("Enter the number of disk requests: ");
scanf("%d", &num_requests);

// Dynamically allocate an array to store disk requests

int *requests = (int *)malloc(num_requests * sizeof(int));

// Prompt the user to enter the disk request positions

printf("Enter the disk request positions:\n");
for (int i = 0; i < num_requests; i++) {
    scanf("%d", &requests[i]);
}

// Print the disk request order

printf("\nDisk Request Order: ");
for (int i = 0; i < num_requests; i++) {
    printf("%d ", requests[i]);
}

printf("\n\nSeek Sequence:\n");

// Process disk requests in the order they were received

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

    // Calculate the distance to the current request
    int distance = abs(head_position - requests[i]);

    // Add the distance to the total seek time
    total_seek_time += distance;

    // Print the seek sequence
    printf("Move from %d to %d, Seek Time: %d\n", head_position, requests[i], distance);
```

```c
        // Update the head position to the requested position
        head_position = requests[i];
    }

    // Print the total seek time
    printf("\nTotal Seek Time: %d\n", total_seek_time);

    // Free the dynamically allocated memory
    free(requests);

    return 0;
}
```

ASSIGNMENT_8:-

Q: You are given a list of page requests and need to implement the Least Recently Used (LRU) page replacement algorithm to simulate memory allocation.

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

// Structure to represent a page frame in the page table
typedef struct {
    int page;
    int lastUsed; // Keeps track of when the page was last used
} PageFrame;

// Function to initialize the page table
void initializePageTable(PageFrame* pageTable, int tableSize) {
    for (int i = 0; i < tableSize; i++) {
        pageTable[i].page = -1;  // Initialize to an invalid page number
        pageTable[i].lastUsed = 0;
    }
}

// Function to find the least recently used page in the page table
int findLRUPage(PageFrame* pageTable, int tableSize) {
    int minIndex = 0;
    int minTime = pageTable[0].lastUsed;
```

```
    for (int i = 1; i < tableSize; i++) {
        if (pageTable[i].lastUsed < minTime) {
            minIndex = i;
            minTime = pageTable[i].lastUsed;
        }
    }

    return minIndex;
}

// Function to handle a page request
bool handlePageRequest(PageFrame* pageTable, int tableSize, int requestedPage, int*
pageFaults) {
    // Check if the requested page is already in the page table
    for (int i = 0; i < tableSize; i++) {
        if (pageTable[i].page == requestedPage) {
            pageTable[i].lastUsed = (*pageFaults)++;
            return false; // Page hit
        }
    }

    // Page fault: Requested page is not in the page table
    int lruPage = findLRUPage(pageTable, tableSize);
    pageTable[lruPage].page = requestedPage;
    pageTable[lruPage].lastUsed = (*pageFaults)++;
    return true; // Page fault
}

int main() {
    int tableSize, numRequests;
    int pageFaults = 0;

    printf("Enter the page table size: ");
    scanf("%d", &tableSize);

    PageFrame pageTable[tableSize];
    initializePageTable(pageTable, tableSize);

    printf("Enter the number of page requests: ");
    scanf("%d", &numRequests);

    printf("Enter the sequence of page requests:\n");
    for (int i = 0; i < numRequests; i++) {
```

```
        int requestedPage;
        scanf("%d", &requestedPage);

        if (handlePageRequest(pageTable, tableSize, requestedPage, &pageFaults)) {
            printf("Page fault occurred for request %d\n", requestedPage);
        }
    }

    printf("Total page faults: %d\n", pageFaults);

    return 0;
}
```

ASSIGNMENT_9:-

Q:Consider a roller coaster ride at an amusement park. The ride operator runs the ride only when there are exactly N riders on it.

CODE:

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

#define MAX_RIDERS 100

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t riders_ready = PTHREAD_COND_INITIALIZER;
int riders_count = 0;
int total_riders = 0;
int N;
int exit_program = 0; // Flag to signal program exit

void* operator_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        // Wait for N riders to accumulate or exit signal
        while (riders_count < N && !exit_program) {
            printf("Operator: Waiting for riders to queue up...\n");
            pthread_cond_wait(&riders_ready, &mutex);
        }
```

```
        if (exit_program) {
            pthread_mutex_unlock(&mutex);
            break; // Exit the loop
        }

        // Start the ride
        printf("Operator: Starting the ride!\n");
        riders_count = 0;

        // Signal riders to enter the ride
        pthread_cond_broadcast(&riders_ready);

        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

void* rider_thread(void* arg) {
    int rider_id = *(int*)arg;
    free(arg); // Free allocated memory for rider_id

    pthread_mutex_lock(&mutex);
    total_riders++;
    printf("Rider %d: Arrived at the ride.\n", rider_id);

    // Check if we have enough riders to start the ride
    riders_count++;
    if (riders_count == N) {
        printf("Rider %d: Signaling the operator.\n", rider_id);
        pthread_cond_signal(&riders_ready);
    } else {
        // Wait for more riders to arrive or exit signal
        while (riders_count < N && !exit_program) {
            printf("Rider %d: Waiting for more riders to join...\n", rider_id);
            pthread_cond_wait(&riders_ready, &mutex);
        }

        if (exit_program) {
            pthread_mutex_unlock(&mutex);
            pthread_exit(NULL);
        }
    }
```

```
    // Enter the ride
    printf("Rider %d: Entering the ride.\n", rider_id);

    pthread_mutex_unlock(&mutex);

    // Simulate the ride
    sleep(2);

    // Exit the ride
    printf("Rider %d: Exiting the ride.\n", rider_id);

    pthread_exit(NULL);
}

int main() {
    int num_riders;
    pthread_t operator_tid;
    pthread_t rider_tid[MAX_RIDERS];

    printf("Enter the number of riders needed to start the ride: ");
    scanf("%d", &N);

    if (N <= 0 || N > MAX_RIDERS) {
        printf("Invalid number of riders. Please enter a value between 1 and %d.\n",
MAX_RIDERS);
        return 1;
    }

    pthread_create(&operator_tid, NULL, operator_thread, NULL);

    while (!exit_program) {
        printf("Enter the number of riders (0 to exit): ");
        scanf("%d", &num_riders);

        if (num_riders == 0) {
            exit_program = 1; // Set the exit flag
            pthread_cond_broadcast(&riders_ready); // Wake up operator thread
        } else {
            for (int i = 0; i < num_riders; i++) {
                int* rider_id = (int*)malloc(sizeof(int));
                *rider_id = total_riders + 1;
                pthread_create(&rider_tid[total_riders], NULL, rider_thread, rider_id);
```

```
            }
        }
    }

    // Clean up
    pthread_join(operator_tid, NULL);

    for (int i = 0; i < total_riders; i++) {
        pthread_join(rider_tid[i], NULL);
    }

    return 0;
}
```

ASSIGNMENT_10:-

Q: A host of a party has invited N > 2 guests to his house. Due to fear of Covid-19 exposure, the host does not wish to open the door of his house multiple times to let guests in.

CODE:

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

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv_host = PTHREAD_COND_INITIALIZER;
pthread_cond_t cv_guest = PTHREAD_COND_INITIALIZER;
int guest_count = 0;
int N; // Total number of guests

// Function to simulate the host opening the door
void openDoor() {
    printf("Host: Opening the door.\n");
    // Signal the waiting guests that the door is open
    pthread_cond_broadcast(&cv_guest);
}

// Function for the host thread
void* hostThread(void* arg) {
    // Wait for all N guests to arrive
    pthread_mutex_lock(&m);
    while (guest_count < N) {
```

```
        printf("Host: Waiting for guests to arrive...\n");
        pthread_cond_wait(&cv_host, &m);
    }
    pthread_mutex_unlock(&m);

    // Call openDoor to let the guests in
    openDoor();

    pthread_exit(NULL);
}

// Function for the guest thread
void* guestThread(void* arg) {
    int guest_id = *(int*)arg;
    free(arg); // Free allocated memory for guest_id

    // Simulate the arrival of a guest
    printf("Guest %d: Arrived at the door.\n", guest_id);

    // Increment the guest count and notify the host
    pthread_mutex_lock(&m);
    guest_count++;
    if (guest_count == N) {
        // All guests have arrived, signal the host to open the door
        pthread_cond_signal(&cv_host);
    }
    pthread_mutex_unlock(&m);

    // Wait for the host to open the door
    pthread_mutex_lock(&m);
    while (guest_count < N) {
        printf("Guest %d: Waiting for the door to open...\n", guest_id);
        pthread_cond_wait(&cv_guest, &m);
    }
    pthread_mutex_unlock(&m);

    // Enter the house
    printf("Guest %d: Entering the house.\n", guest_id);

    pthread_exit(NULL);
}

int main() {
```

```
    // Number of guests
    printf("Enter the number of guests (N): ");
    scanf("%d", &N);

    if (N <= 2) {
        printf("Invalid number of guests. Please enter a value greater than 2.\n");
        return 1;
    }

    pthread_t host_tid;
    pthread_t guest_tid[N];

    pthread_create(&host_tid, NULL, hostThread, NULL);

    for (int i = 0; i < N; i++) {
        int* guest_id = (int*)malloc(sizeof(int));
        *guest_id = i + 1;
        pthread_create(&guest_tid[i], NULL, guestThread, guest_id);
    }

    // Wait for the host and guests to finish
    pthread_join(host_tid, NULL);

    for (int i = 0; i < N; i++) {
        pthread_join(guest_tid[i], NULL);
    }

    return 0;
}
```

ASSIGNMENT_11:-

Q: Consider the classic readers-writers synchronization problem described below. Several processes/threads wish to read and write data shared between them.

CODE:

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

// Reader-writer lock functions
```

```
int readCount = 0;
int writeCount = 0;
pthread_mutex_t rwLockMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t canRead = PTHREAD_COND_INITIALIZER;
pthread_cond_t canWrite = PTHREAD_COND_INITIALIZER;

void readLock() {
    pthread_mutex_lock(&rwLockMutex);
    while (writeCount > 0) {
        pthread_cond_wait(&canRead, &rwLockMutex);
    }
    readCount++;
    pthread_mutex_unlock(&rwLockMutex);
}

void readUnlock() {
    pthread_mutex_lock(&rwLockMutex);
    readCount--;
    if (readCount == 0) {
        pthread_cond_signal(&canWrite);
    }
    pthread_mutex_unlock(&rwLockMutex);
}

void writeLock() {
    pthread_mutex_lock(&rwLockMutex);
    while (readCount > 0 || writeCount > 0) {
        pthread_cond_wait(&canWrite, &rwLockMutex);
    }
    writeCount++;
    pthread_mutex_unlock(&rwLockMutex);
}

void writeUnlock() {
    pthread_mutex_lock(&rwLockMutex);
    writeCount--;
    pthread_cond_signal(&canRead);
    pthread_cond_signal(&canWrite);
    pthread_mutex_unlock(&rwLockMutex);
}

// Shared data
int sharedData = 0;
```

```
// Function for reader thread
void* reader(void* arg) {
    int id = *((int*)arg);
    free(arg);

    while (1) {
        readLock();
        printf("Reader %d reads: %d\n", id, sharedData);
        readUnlock();
        sleep(1);
    }
    return NULL;
}

// Function for writer thread
void* writer(void* arg) {
    int id = *((int*)arg);
    free(arg);

    while (1) {
        writeLock();
        sharedData++;
        printf("Writer %d writes: %d\n", id, sharedData);
        writeUnlock();
        sleep(1);
    }
    return NULL;
}

int main() {
    // Create reader and writer threads
    pthread_t readers[3];
    pthread_t writers[2];

    for (int i = 0; i < 3; i++) {
        int* reader_id = (int*)malloc(sizeof(int));
        *reader_id = i;
        pthread_create(&readers[i], NULL, reader, reader_id);
    }

    for (int i = 0; i < 2; i++) {
        int* writer_id = (int*)malloc(sizeof(int));
```

```
        *writer_id = i;
        pthread_create(&writers[i], NULL, writer, writer_id);
    }

    // Join threads (not shown in infinite loop)

    // Cleanup and exit

    return 0;
}
```

ASSIGNMENT_12:-

Q: Consider the readers and writers problem discussed above. Recall that multiple readers can be allowed to read concurrently.

CODE:

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

// Define semaphores and variables
sem_t mutex;          // Mutex for protecting shared variables
sem_t readAccess;     // Semaphore for controlling read access
sem_t writeAccess;    // Semaphore for controlling write access
int readers = 0;      // Number of active readers

// Initialize semaphores
void init() {
    sem_init(&mutex, 0, 1);        // mutex is initialized to 1
    sem_init(&readAccess, 0, 1);   // readAccess is initialized to 1
    sem_init(&writeAccess, 0, 1);  // writeAccess is initialized to 1
}

// Function to acquire read lock
void readLock() {
    sem_wait(&mutex);
    readers++;
    if (readers == 1) {
        sem_wait(&writeAccess);
    }
```

```
    sem_post(&mutex);
    sem_wait(&readAccess);
}

// Function to release read lock
void readUnlock() {
    sem_wait(&mutex);
    readers--;
    if (readers == 0) {
        sem_post(&writeAccess);
    }
    sem_post(&mutex);
    sem_post(&readAccess);
}

// Function to acquire write lock
void writeLock() {
    sem_wait(&writeAccess);
}

// Function to release write lock
void writeUnlock() {
    sem_post(&writeAccess);
}

int main() {
    // Initialize the locks
    init();

    // Example usage of the reader-writer locks
    // You can create reader and writer threads and use the provided functions as needed

    return 0;
}
```

-: CREATED BY SHREY GARG :-