Threading and synchronization

Q1) Implement a multi-threaded C program that demonstrates synchronization using semaphores to protect a shared variable from concurrent access by multiple threads.

Problem Description:

A shared integer variable `shared` is accessed and modified by two threads:

- **Thread 1 (`fun1`)**: Reads the current value of `shared`, increments it locally, and updates `shared` after a short delay.
- **Thread 2 (`fun2`)**: Reads the current value of `shared`, decrements it locally, and updates `shared` after a short delay.

Since both threads modify the same variable, **race conditions** may occur if proper synchronization is not implemented. To ensure **mutual exclusion**, use a **semaphore (`sem_t`)** to allow only one thread to modify `shared` at a time.

Requirements:

1. Use **POSIX threads (`pthread`)** to create two threads.
2. Use a **semaphore (`sem_t`)** to synchronize access to the shared variable.
3. Ensure that each thread:
    - Reads the shared variable.
    - Modifies it locally.
    - Updates it back safely using **semaphores** to avoid race conditions.
4. Print the value of `shared` before and after modification in each thread.
5. After both threads complete execution, print the **final value** of `shared`.

Expected Outcome:

The program should execute in a synchronized manner, ensuring **no data corruption** occurs due to concurrent modifications. The final value of `shared` should be correctly updated, demonstrating proper **thread synchronization using semaphores**.

Q2) Implement a warehouse inventory management system where suppliers (producers) add goods to the warehouse, and customers (consumers) take goods from it. The system should ensure that:

- Producers cannot add items if the warehouse (buffer) is full.
- Consumers cannot take items if the warehouse is empty.
- Synchronization prevents race conditions and ensures smooth operations.

Problem Description:

- The warehouse is modeled as a **bounded buffer** (fixed-size array).
- **Producers (supplier threads)** add items to the warehouse. If the warehouse is full, they must wait.
- **Consumers (customer threads)** remove items from the warehouse. If it's empty, they must wait.
- **Semaphores** are used to ensure synchronization:
    - A **mutex** ensures mutual exclusion while modifying the warehouse.
    - A **full semaphore** keeps track of available items.
    - An **empty semaphore** keeps track of available storage space.

Requirements:

Use POSIX Threads (pthread):

- Implement **two threads**: one for the **supplier** (adding stock) and one for the **customer** (removing stock).

Use Semaphores (sem_t):

- Ensure **mutual exclusion** when modifying the shared inventory.

Each thread should:

- **Read** the shared inventory value.
- **Modify** it locally (add/remove stock).
- **Update** the shared inventory safely using semaphores

Expected Outcome:

- The warehouse operates efficiently without race conditions.
- No consumer takes an item when the warehouse is empty.
- No producer adds an item when the warehouse is full.
- The system ensures **proper synchronization** between producers and consumers.

Q3) Write a C program to simulate the concept of Dining-philosophers problem using semaphores.

Q4) Cigarette Smokers Problem (Synchronization Using Semaphores)

Problem Description

In a shared environment, three smokers want to roll and smoke cigarettes, but each smoker has **only one** of the required ingredients:

1. **Smoker A** has **Tobacco** but needs **Paper and Matches**.
2. **Smoker B** has **Paper** but needs **Tobacco and Matches**.
3. **Smoker C** has **Matches** but needs **Tobacco and Paper**.

A **bartender (agent)** repeatedly places **two random ingredients** on a table. The smoker who has the missing ingredient picks up the items, rolls a cigarette, smokes it, and then signals the bartender to place the next set of ingredients.

Implementation Strategy

Use **semaphores** to ensure correct synchronization:

- **Three semaphores** for ingredient pairs:
  - `tobaccoAndPaper` (triggers smoker with Matches)
  - `paperAndMatches` (triggers smoker with Tobacco)
  - `matchesAndTobacco` (triggers smoker with Paper)
- **A semaphore `doneSmoking`** to signal when the next batch of ingredients can be placed.
- **An agent thread** that randomly picks two ingredients and places them on the table.
- **Three smoker threads**, each waiting for their missing ingredients to proceed.

Expected Outcome

1. The **agent** places two ingredients at random.
2. The smoker with the missing third ingredient **picks them up**, rolls a cigarette, and smokes.
3. After smoking, the smoker signals the **agent** to provide the next set of ingredients.
4. The process **repeats indefinitely** without deadlocks or resource conflicts.

Q5) Multi-threaded Banking System with ATM Transactions.

Problem Description

In a banking system, multiple Automated Teller Machines (ATMs) allow customers to perform transactions, such as **withdrawals** and **deposits**, from a **shared bank account**. However, simultaneous transactions by multiple users can lead to **race conditions**, where multiple ATMs modify the balance at the same time, potentially resulting in:

- **Overdrawn accounts** (e.g., two users withdrawing more money than available).
- **Incorrect balance updates** (e.g., deposits and withdrawals interfering with each other).

To **prevent race conditions**, **synchronization mechanisms** like **mutex locks or semaphores** must be used to ensure that only **one ATM transaction modifies the account balance at a time**.

Implementation Strategy

Use POSIX Threads (`pthread`)

- Create multiple ATM threads, each performing **withdrawal** or **deposit** operations.

Use Mutex Locks (`pthread_mutex_t`)

- Protect access to the shared **bank account balance** so that **only one transaction is processed at a time**.

Transaction Logic

- **Withdrawal:**
  - Check if sufficient funds are available.
  - Deduct amount if possible, otherwise print an error.
- **Deposit:**
  - Add amount to the balance.
- Ensure that both transactions **lock** and **unlock** access to the shared balance.

Expected Outcome:

- Each ATM prints the balance **before and after** modification.
- The final balance reflects **all valid transactions**.