Race Condition, Deadlock & Synchronization

1 Objectives

The main goals of this assignment are:

- Studying race condition, deadlock and how to avoid them.
- Identifying critical sections where race condition may occur.
- Using locks for Mutual Exclusion (ME) as a synchronization tool to prevent race condition.

2 Description And Requirements

This assignment consists of two mandatory parts involving threads and synchronization, along with an optional section.

- **Part 1:** Simulate a race condition in a Banking System where multiple threads access a shared account. Identify critical sections and enforce Mutual Exclusion to resolve the issue.
- **Part 2:** Simulate a deadlock scenario in the Banking System by performing transferring transactions between two accounts. Then, implement a solution to prevent deadlocks..
- **Part 3 (Optional):** This part covers Livelock and is optional. While not mandatory, it's recommended to consider and explore potential solutions to resolve the scenario.

To pass, you must complete both mandatory parts. You may implement them as separate projects (C# in Visual Studio) or as different packages within the same Java project.

No GUI template is provided, but you may reuse GUI components from the previous assignment. Alternatively, you may implement the solution as a console-based application. In either case, ensure a well-organized project structure with clear, documented code. Use meaningful and consistent naming for classes, methods, and variables.

3 Part 1: Banking Simulation with Race Condition Analysis

Implement a multithreaded banking simulation where multiple client threads perform random deposit/withdrawal operations on a shared bank account. Initially, implement the classes **without synchronization** to observe race conditions and their impact on data integrity. Then, fix the issue using synchronization

This part of the assignment should demonstrate the following tasks:

1. **Race Conditions**: When multiple threads access shared data unsafely.
2. **Synchronization Solution**: Protect critical sections using `synchronized(lock)` (Java) **or** `lock(lockObj)` (C#)..
3. **Verification Method**: Comparing expected vs. actual balance to detect errors.

Note: The solution should avoid using Semaphore or Monitor. For mutual exclusion in critical sections, use only locks, `synchronized(lock)` in Java and `lock(lockObj)` in C.

Implement classes to represent clients, bank accounts, and a class to simulate banking transactions, along with any other necessary classes. Below are some suggested class hints, though you are free to design the project structure of your project:

- **BankAccount**: Manages account details and balance operations.
- **Client**: Represents a bank client with relevant attributes.
- **TransactionSimulator**: Simulates banking transactions (deposits, withdrawals, transfers).
- Additional helper classes as needed.

3.1 BankAccount Class (Shared Resource)

This class manages the account balance providing methods that handles deposits and withdrawals.

- Create this class and write the necessary methods to implement transactions, and return the current balance.
- Initially implement the methods without synchronization to intentionally create race conditions for testing.
- Later. ensure thread safety in critical sections using the synchronization mechanisms specified above to resolve race condition inconsistencies.

Critical Sections:

- **Phase 1** (**Unsynchronized**): Demonstrate race conditions (e.g., incorrect balance due to concurrent access).
- **Phase 2 (Synchronized):** Ensure thread-safe modifications to the account balance. Identify all critical sections (methods accessing and modifying the balance and other instance variables (deposit(), withdraw()) and add synchronization to protect the shared data using the specified locking mechanisms.

3.2 Client Class (Threads)

Represents a bank client that performs random transactions on a shared BankAccount until terminated by the main thread.

- The class maintains a unique **clientId** and **totalTransactions** which is the sum of all deposits and withdrawals for the client.
- The client performs random transactions (deposit or withdraw) in an infinite loop until receiving a stop signal from the main thread.
- Execute the selected operation on the shared **BankAccount** with a random amount.

3.3 TransactionSimulator

The purpose of this class is to simulate multiple clients performing simultaneous transactions on a shared bank account, creating real-world conditions where race conditions may occur. By running unsynchronized operations initially, the class demonstrates how concurrent access to shared resources can lead to data corruption and inconsistent financial balances.

After establishing the presence of race conditions, the class then validates thread synchronization solutions. It does this by implementing proper locking mechanisms and comparing the expected balance (calculated from transaction totals) with the actual account balance.

Hint:

- Create and initialize a **BankAccount** object with a starting balance of **1000**.
- Create and maintain a list of clients
- Create and maintain a list of threads, one thread for each client
- Start all client threads to perform random transactions.
- Run the simulation for a fixed duration (e.g., 5 seconds).
- Stops all threads and compares:
  - **Expected Balance** = Initial Balance + Sum of All Client Transactions
  - **Actual Balance** = Final BankAccount Balance

Simulating Race Conditions

The simulation runs for a predetermined duration (e.g. 5 seconds), allowing sufficient time for race conditions to occur in the unsynchronized version. This duration can be adjusted to observe how longer or shorter operation periods affect the likelihood and severity of race conditions.

- Clients access the shared BankAccount without synchronization, leading to data corruption.
- **Race condition** if Actual Balance ≠ Expected Balance.

When performing the comparison, allow a small discrepancy (e.g., ±0.01) to account for floating-point precision limitations.

Sample Output (Unsynchronized):

```
SIMULATION RESULTS:
Total transactions including all clients: 4409265,22
Expected final balance: 4410265,22
Actual final balance: 5713,01
Discrepancy: 4404552,21
FAILURE: Race condition detected - balances don't match!
```

Fixing Race-Condition.

- Synchronize critical sections (deposit(), withdraw()).
- **With Synchronization: Expected Balance == Actual Balance.**
- No race condition:

```
if (Math.abs(expectedBalance - actualBalance) < 0.01)
```

Sample Output (Synchronized):

```
SIMULATION RESULTS:
Total transactions including all clients: 192376,24
Expected final balance: 193376,24
Actual final balance: 193376,24
Discrepancy: -0
SUCCESS: No race condition detected
```

3.4 How to provoke Race Conditions for testing:

To increase the likelihood of race conditions in unsynchronized code (when tasks are too small/fast to naturally trigger them), implement these measures:

- Vary Thread Execution Timing
  - Insert delays (e.g., Thread.sleep(1)) within critical. Longer delays typically worsen discrepancies, making race conditions more observable.
- Execute Repeated Multiple Test Runs
  - Without synchronization, repeated execution should show inconsistent results due to unpredictable thread interference.
  - With proper synchronization, results must remain consistent across all runs.
- Optimize Thread Startup
  - Create all threads in one loop **without** starting them immediately.
  - Start all threads in a separate loop afterward to maximize concurrent access to shared resources.

3.5 Optional Task: Security Class

Implement a Security class to monitor transactions and detect inconsistencies in the BankAccount. The following features can be included in this class:

- **Transaction Logging:**
  - Record each transaction attempt with:

  - Client ID
  - Transaction type (deposit/withdrawal)
  - Account balance before and after the transaction

- **Inconsistency Detection:**
  - Compare "before" and "after" balance stamps to identify unauthorized or erroneous modifications.
  - Flag discrepancies (e.g., withdrawals exceeding balance, deposits not reflecting in final balance).

Implementation Notes:

- Use a thread-safe collection (e.g., ConcurrentLinkedQueue in Java, ConcurrentQueue<T> in C#) to store logs.
- Method to analyze logs and report detected anomalies.

Stamp Data Example:

```
class TransactionStamp {
    int clientId;
    String type; // "DEPOSIT" or "WITHDRAW"
    double amount;
    double balanceBefore;
    double balanceAfter;
}
```

4 PART 2: DEADLOCK (AND LIVELOCK)in ACCOUNT TRANSFERS

4.1 Part 2a: Deadlock

Deadlocks occur in multi-threaded applications when threads hold resources and wait for others to release resources that they need. In our scenario, we'll simulate a situation where two threads attempt to acquire two locks in different orders, leading to a deadlock.

Thread 1:

- Acquires lock1
- Attempts to acquire lock2

Thread 2:

- Acquires lock2
- Attempts to acquire lock1

To avoid deadlock, it is crucial for threads to acquire locks in the same order. This ensures a consistent locking sequence across all threads, preventing circular dependencies and potential deadlocks.

To address this, we'll modify the thread logic to acquire locks in a consistent order:

Thread 1:

- Acquires lock1
- Acquires lock2

Thread 2:

- Acquires lock1
- Acquires lock2

By ensuring that all threads acquire locks in the same order, we can effectively prevent deadlocks and maintain the stability of our application.

To Do:

Create a new project to demonstrate a deadlock scenario in multi-threaded banking transactions. The implementation requires:

```
void transfer(Account toAccount, int amount) { /*code*/}
```

The method should handle money transfers between accounts

Deadlock Simulation

Implement the transfer method to:

- First lock the current account (this)
- Then lock the target account (toAccount)

Test Scenario

To test the deadlock scenario, create two threads and have each thread attempt to transfer money to two different accounts (Account A and Account B) concurrently. Ensure that the test code is designed to potentially cause a deadlock by having the threads attempt to transfer money between the same two accounts, but in different order.

- Thread 1: A → B
- Thread 2: B → A

- Include output messages to indicate:
  - When threads acquire locks
  - When deadlock occurs
  - Successful completion (if no deadlock)

4.2 Part 2b: Preventing Deadlock

To eliminate potential deadlocks in account transfers, create a new class called **AccountNoDeadlock**. To prevent deadlocks in account transfers, implement a thread-safe class using the following strategies:

- **Using Lock Objects**
    - Introduce two distinct lock objects (lockObj1 and lockObj2) to separately control access to:
        - The current account instance (this) //fromAccount
        - The target account (toAccount)
- **Structured Critical Sections**
    - Implement two isolated critical sections in the transfer method:
        1. First critical section: Locks and processes the current account
        2. Second critical section: Locks and processes the target account
    - This separation ensures atomic operations while preventing circular wait conditions —the fundamental prerequisite for deadlock.
- **Consistent Lock Ordering**
    - Establish and maintain a global locking order (e.g., by account ID) when acquiring multiple locks. o prevent deadlocks when multiple locks are required, enforce a **strict global order** for lock acquisition.
    - Guarantees all threads request locks in the same predetermined sequence

Code: Java

```java
if (this.clientId < toAccount.clientId) {
    synchronized (this) {
        synchronized (toAccount) {
            // Transfer logic
        }
    }
} else {
    synchronized (toAccount) {
        synchronized (this) {
            // Transfer logic
        }
    }
}
```

Code: C#

```csharp
if (this.clientId < toAccount.clientId)
{
    lock (this)
    {
        lock (toAccount)
        {
            // Transfer logic
        }
    }
}
```

```
else
{
    lock (toAccount)
    {
        lock (this)
        {
            // Transfer logic
        }
    }
}
```

4.3 Part 2c **Optional Livelock Simulation**

Livelocks represent a concurrency scenario where threads remain active but fail to make progress, continuously retrying operations by releasing and reacquiring resources in a cyclic manner. Unlike deadlocks where threads block indefinitely, livelocked threads consume CPU resources while stuck in this unproductive loop.

To simulate this behavior, modify the Account class's Transfer method to implement a retry mechanism. When a thread cannot acquire both account locks simultaneously, it should release any held locks and reattempt the operation. You may introduce a maximum attempt limit (e.g., 20 retries) to prevent endless execution. Threads will persistently attempt transfers until either succeeding or exhausting their retry allowance.

Resolution strategies include implementing fair lock allocation, introducing randomized delays between retries, or incorporating timeouts. These approaches help break the cyclical dependency and allow threads to progress. As an optional task, this livelock simulation should be developed in a separate class or project to maintain code isolation.

5 GRADING AND SUBMISSION

You are required to present your solution to your supervisor during the lab sessions in person. Additionally, you must upload the solution to Canvas. To upload your files, compress all the folders, and subfolders into a single archive file such as **zip**, **rar**, or **7z**. Upload the archive file via the Assignment page in Canvas.

You are required to present your solution to your supervisor during the lab sessions in person. Additionally, you must upload the solution to Canvas. To upload your files, compress all the folders and subfolders into a single archive file using a format such as zip, rar, or 7z.

Good Luck!

Farid Naisan,
Course Responsible and Instructor