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

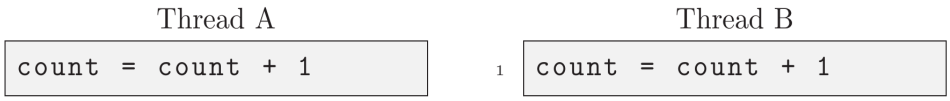
This assignment is divided into two mandatory parts, both of which involve the use of threads and synchronization, and one optional part. In Part 1, we will simulate transactions in a simple Banking System using multiple threads. In Part 2, we will simulate occurrence of a deadlock scenario in a Seat Management System. Then we will implement a solution to avoid deadlocks. In both parts, we will identify critical sections where a race condition can occur, and we will implement Mutual Exclusion to prevent it. Additionally, there is an optional part, Part 3, in the assignment that deals with Livelock. It is not mandatory but is recommended to give it a thought and think of a possible solution to avoid deadlock.

For a pass grade, you must complete both parts. You may use different application projects or integrate them into the same solution (C#, VS) or different packages in the same Java project.

No GUI template is provided for this assignment as you utilize the GUI files from the previous assignment. However, you may choose to complete the assignment using a console/terminal text-based project. In both cases, use a well-organized project and well structured and documented code. Also, use informative and appropriate names for your classes, methods and variables.

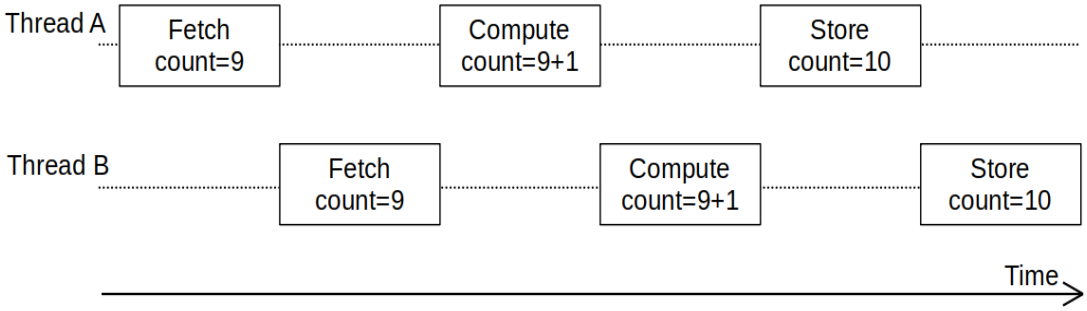
3 An Overview of Race Condition

"Read-modify-write" are atomic (indivisible) operations that CPUs can execute. Even the simplest operations in high-level programming languages are compiled into multiple CPU instructions. The problem arises when shared mutable data is accessible by multiple threads that run concurrently.



Even a simple line of code, like the one shown in the image above, is compiled into at least three operations: (1) the variable "count" is loaded from memory, (2) its value is increased by 1, and (3) the result is saved back into memory.

When multiple threads run the same code simultaneously, they may attempt to access shared data concurrently, potentially leading to data loss or corruption. The image below illustrates a possible scenario.



As the example above demonstrates, when multiple threads change the value of a shared variable without proper control, the result can be an incorrect value. In the given scenario, thread A assigns a value of 10 to the variable "count," but before it has a chance to save this value, thread B enters the code and reads the old value of 9, resulting in the final value of "count" being 10 instead of 11.

To prevent such inconsistencies, the code can be synchronized so that only one thread is permitted to execute and finish the operation before the next one is allowed to access the shared variable (Mutual Exclusion). With synchronization in place, the final value of "count" should be 11, as expected.

Mutual exclusion (ME) is a mechanism that prevents concurrent execution of code that accesses shared resources to avoid race condition. In C#, the lock keyword can be used for ME, while in Java, the synchronized keyword can be used. To apply synchronization, it is recommended to use it only around the critical sections of code, i.e., sections that access shared resources. Do not include non-critical sections and do not lock a whole object or a method.

Java: It is important to note that the synchronized statement should be used only around the critical section and not with the method definition. This helps in identifying the critical and non-critical parts of the code

C#

```
private Object lockObj = new Object();
lock (lockObj) {
        //code
}
```

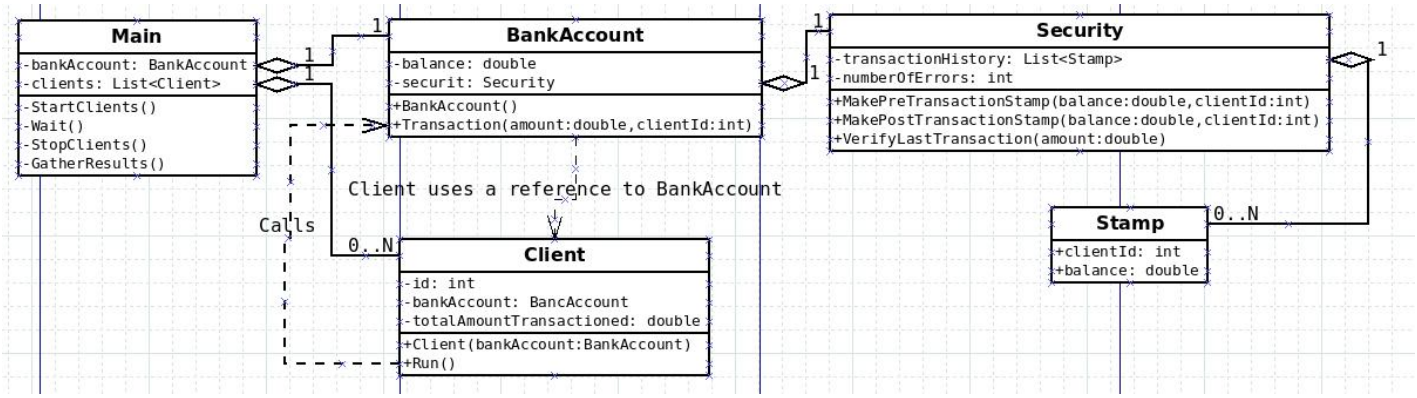
Java: Replace lock in the above code with synchronized and the rest is the same.

```
private Object lockObj = new Object();
synchronize (lockObj){
        //code
}
```

4 Part 1: Banking Transactions

Write a program that simulates banking operations in a bank, allowing multiple threads to deposit or withdraw random amounts from the same bank account. The threads should run in an infinite loop without any pauses. In this application, we intentionally create a read-modify-write type of race condition, which can result in inconsistencies or errors in the final account balance.

Write classes to represent clients, banking transactions and other classes that find necessary. The following class diagram shows a suggested structure of the project although you don't have to follow it in detail.



Each client tries to do transactions on the **BankAccount,** which is the shared resource. By accessing the account in an uncontrolled manner, threads modify the value of the balance, corrupting the shared data. This is the consequence of the race conditions that occur. To illustrate the problem, we make clients keep track of the total amount of all their transactions. With this data in hand for all clients, we can compare it with the balance of the **BankAccount** at the end of the program and analyze the results.

The Client class:

4.1 The Client class represents a client by the Id of the client. The operations expected from the class is to perform a deposit or withdrawal transaction. It represents either a thread object (Java) or has actions assigned to a client thread (C#) and simulates a client performing randomly transactions until it is ordered to stop. You can randomize a Boolean value to decide whether to deposit or withdraw an amount from the **BankAccount**.

4.2 Add a **totalAmountTransactioned** counter in this class to store the amount of each transaction.

Main operations in the client thread class:

```
private double totalAmountTransactioned;

while(operating) {
    account.deposit(amount);
    totalAmountTransactioned += amount;
```

```
        }
    }
```

The BankAccount class

Due to the occurrence of race conditions, the balance of the account after a transaction may be incorrect and this is obviously an error; we would like to track the occurrences of such errors in the **BankAccount** class.

In order to observe and store information about the correctness of each transaction performed by a client, let us create a **Security** class using another class **Stamp** to store some relevant data, as shown in the above class diagram. This can be done by using a stamp before and after a transaction. The stamp is simply data about the balance before and after the transaction. It stores data about the client making the transaction, type of transaction, e.g., deposit or withdrawal, and the balance.

The balance after a transaction is to be verified in the Security class by checking whether the balance after the transaction is completed matches the balance saved in **BankAccount**. While a transaction is being realized by a client thread, other client threads may have changed the balance of the **BankAccount**.

4.3 Write a method in the **BankAccount** class for performing a security check by using the methods of the Security class. by saving the balance before and after each transaction, as described above. You may choose to keep a history of all transactions or only save the last two values.

4.4 Add a counter that keeps track of the number of incorrect transactions. Allow the client threads to run for several seconds before stopping them and comparing the expected balance (sum of the total amount transacted in the thread classes) with the actual balance of the account. This comparison will help you identify inconsistencies or errors resulting from race conditions.

Main operations in the **BankAccount** Class:

```
public void Transaction(double amount, int clientId){
        security.MakePreTransactionStamp(balance,
 clientId);
        balance = balance + amount;
 numberOfTransactions ++;
 security.MakePostTransactionStamp(balance, clientId);
 security.VerifyLastTransaction();
 }
```

4.5 Once all client threads have been stopped, calculate the total sum of all transactions performed by the clients, and compare it with the resulting balance in the **BankAccount** class. Due to the race condition introduced in the program, it is expected that a significant number of transaction errors will occur.

4.6 The expected output of the program is to see the number of transaction errors and compare it with the expected results.

```
Number of transactions: 7526177
Number of errors: 0
All transactions of Clients sums into: -221689, balance on account -221689
```

4.7 In order to prevent race condition, it is necessary to identify all the critical sections in the code where multiple threads could access and modify shared data. Once identified, Mutual Exclusion (ME) can be used to protect these sections. As mentioned earlier, In C#, the lock(object) keyword can be used to achieve ME, while in Java, the synchronized(object) statement can be used..

4.8 After implementing ME, the expected output of the program should be accurate and consistent.

```
Number of transactions: 7526177
Number of errors: 0
All transactions of Clients sums into: -221689, balance on account -221689
```

4.9 Run your application multiple times with varying sleep times inserted at different parts of the code to observe how it affects the outcome. Analyze and explain how race conditions can impact the correctness of shared data, and be prepared to demonstrate how you protect against race conditions by implementing mutual exclusion using locks, when presenting your work.

5 PART 2: DEADLOCK AND LIVELOCK – A TRANSFER TRANSACTION

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 for this part. Implement a class named **Account**, which may be created from scratch or copied from Part 1, containing a method for transferring money to another account:

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

This method should facilitate the transfer of a specified amount of money to another account.

To simulate a deadlock scenario, implement the transfer method in a way that first thread locks the current account (**this**) and then locks the target account (**toAccount**). In Java, you can achieve this by using the **synchronized** keyword (**synchronized (toAccount) { ... }**), while in C#, you can use the **lock** keyword (**lock (toAccount) { ... }**).

To test the deadlock scenario, create two threads and have each thread attempt to transfer money to two different accounts 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.

Here's a sample test code that you may use in your code:

```
Account account1 = new Account();
Account account2 = new Account();
// Thread for transferring money from account1 to account2
Thread t1 = new Thread(() -> account1.transfer(account2, 500), "Transaction
1");
// Thread for transferring money from account2 to account1
Thread t2 = new Thread(() -> account2.transfer(account1, 300), "Transaction
2");

t1.start();
t2.start();
```

The program should enter a deadlock state showing no progress for any of the threads. To achieve this, include the following watch messages inside the transfer method:

Print: "Thread-name" is inside the transfer method." when a thread enters the critical section.

```
Transaction 1 is inside the transfer method.
Transaction 2 is inside the transfer method.
```

This setup increases the likelihood of encountering a deadlock scenario.

The program should enter a deadlock state showing no progress for any of the threads. You may of course put some watch messages inside the method like the following to get some output when threads enter the critical section:

Part 2b: Avoiding deadlock

Create a new class, such as **AccountNoDeadlock**, and implement changes to avoid deadlock. Use two separate locks, **lockObj1** and **lockObj2**, to lock the account object itself and the **toAccount** object, respectively. Additionally, utilize two separate critical sections, one for each lock.

Part 2c: Simulate a livelock scenario

Livelocks are similar to deadlocks in that they involve multiple threads and resources, but in livelocks, threads are not blocked indefinitely. Instead, they continuously retry an operation without making progress.

In a livelock scenario, threads may release resources and retry an operation, only to find that another thread has also released resources and is attempting the same operation. As a result, threads repeatedly release and reacquire resources without completing the desired task, effectively "spinning" in a loop.

To avoid livelocks, it is essential to implement fair resource allocation, timeouts or using random delays to break the circular behavior and allow threads to make progress. Additionally, using proper resource management and synchronization techniques can help to minimize the risk of livelocks in multi-threaded applications.

To Do:

Create a new class or project for this part. To simulate a livelock scenario, have threads repeatedly lock and release their lock if they cannot acquire both locks. Implement a maximum attempt limit to prevent threads from executing endlessly.

The Transfer method in the Account class is modified to retry the transfer operation while releasing and reacquiring locks if it cannot acquire both locks. The code below creates two threads, t1 and t2, representing two transactions involving the two accounts.

Each thread continuously attempts to transfer money between the accounts until the transfer is successful or the maximum attempt limit is reached.

```
Account account1 = new Account(1);  //sending 1 as id
Account account2 = new Account(2);  //sending 2 as id
```

```
int maxAttempts = 10;

Thread t1 = new Thread(() -> {
    int attempts1 = 0; // Declare outside the loop to keep track of attempts across
iterations
 while (!account1.tryTransfer(account2, 500))

   try {
        Thread.sleep(100); // Simulate action reversal or retry after a pause
       attempts1++;
       if (attempts1 >= maxAttempts) {
           System.out.println(Thread.currentThread().getName() + " reached max at-
tempts, giving up.");
                break;
            }
        } catch (InterruptedException e) {
             e.printStackTrace();
            break;
        }
    }
}, "Transaction 1");
```

Here's the equivalent C# version of the given Java code:

```
Account account1 = new Account(1);
Account account2 = new Account(2);
int maxAttempts = 10;

Thread t1 = new Thread(() =>
{
    int attempts1 = 0; // Declare outside the loop to keep track of attempts across iterations
    while (!account1.Transfer(account2, 500))
    {
      try
      {
          Thread.Sleep(100); // Simulate action reversal or retry after a pause
          attempts1++;
          if (attempts1 >= maxAttempts)
          {
               Console.WriteLine(Thread.CurrentThread.Name + " reached max
attempts, giving up.");
               break;
          }
      }
      catch (ThreadInterruptedException e)
      {
          Console.WriteLine(e.StackTrace);
          break;
      }
    }
});

t1.Name = "Transaction 1";
t1.Start();
}
```

Below is a sample output from a program execution where the maximum attempt limit is set to 10:

```
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 transfer not possible to 1
Transaction 1 transfer not possible to 2
Transaction 2 reached max attempts, giving up.
Transaction 1 reached max attempts, giving up.
```

Questions to answer when presenting your solution:

1. Where in your do the race condition, deadlock and livelock occur and why?
2. Which parts of the code form a critical section, where and how do you apply Mutual exclusion.

6 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