Lab 5: Liveness Hazards

In this lab, we will look into deadlock and how we can solve this by (i) ensuring that all the threads will acquire the locks in the same order, and (ii) using the timed lock feature provided by ReentrantLock. In addition to these, we will also try to replace the use of wait() and notify()/notifyAll() methods with await() and signal()/signalAll() methods, in conjunction with the use of ReentrantLock.

A. Deadlock

Although not common, it is not impossible for an application to run into deadlock. Getting a deadlock is fatal for an application, and usually we are required to restart the application to recover from it. As below is an example that demonstrates how deadlock could occur when multiple threads are trying to acquire the locks required to complete the processing in different orders.

In this example, 20 threads are created to transfer a random amount of money (<$10) from one of the 5 accounts to another account. Because all the 20 threads can run concurrently, we need to ensure that when one thread is trying to deduct certain amount of money from one account and add the money into another account, no other threads can be modifying the balance in these two accounts at the same time. Of course, this is to avoid race condition that might lead to data inconsistency.

Therefore, you can see that the transfer method (transferMoney()) will attempt to acquire two locks, one associated with the account where money will be deducted (fromAcct), and one associated with the account where the money will be transferred to (toAcct). Although it might seem like the locks will always be acquired in the same order (fromAcct then toAcct), it is depending on the actual account passed into fromAcct and toAcct. Hence, it is still possible for this to run into a deadlock under certain unfortunate circumstances, as shown in Figure 1.

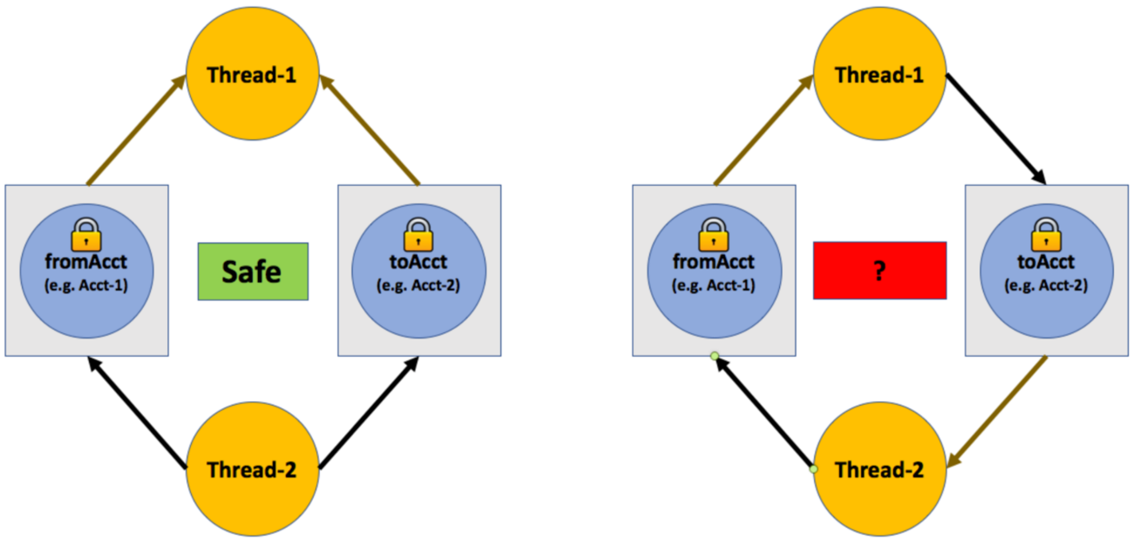

Figure 1: Different sequence in acquiring the locks.

Based on Figure 1, if Thread-1 managed to first acquire the two locks associated with account 1 (acct-1) and account 2 (acct-2), then there will be no deadlock. However, if the lock associated with account 1 is first acquired by Thread-1 and the lock associated with account 2 is first acquired by Thread-2, then it is possible to run into a deadlock when Thread-1 attempts to acquire the lock associated with account 2, and Thread-2 attempts to acquire the lock associated with account 1, all at the same time.

You can try to run the example and just leave it running for a while. Sometimes it might stop because one account has run out of money. But most of the time you should be able to see that it will stop running halfway (just stuck there), because of the unfortunate circumstances like the one described above.

```
// Example 1
import java.util.Random;

public class TestClass1 {
   private static final int numThread = 20;
   private static final int numAccount = 5;
   private static final int numIteration = 1000000;

   public static void main(String[] args) {
      Account[] accounts = new Account[numAccount];

      // Create the accounts and put them into an array.
      for (int i = 0; i < numAccount; i++) {
         accounts[i] = new Account();
      }

      // Create the threads and invoke the start() method.
      for (int i = 0; i < numThread; i++) {
         new Thread(new TransferHelper(accounts, numAccount, numIteration)).start();
      }
   }
}

class Account {
   private int balance = 10000;

   public int getBalance() {
        return balance;
   }

   public void debit(int amount) {
        balance = balance - amount;
   }

   public void credit(int amount) {
        balance = balance + amount;
   }
}

// Continue next page.
```

```
class TransferHelper implements Runnable {
   private Account[] accounts;
   private int numAccount, numIteration;
   private int fromAcctArrayIndex, toAcctArrayIndex;
   private int amount;
   Random rand = new Random();

   TransferHelper(Account[] accounts, int numAccount, int numIteration ) {
      this.accounts = accounts;
      this.numAccount = numAccount;
      this.numIteration = numIteration;
   }

   public void run() {
      for(int i = 0; i < numIteration; i++) {
         // Randomly select two accounts from the array and the amount to transfer.
         fromAcctArrayIndex = rand.nextInt(numAccount);
         toAcctArrayIndex = rand.nextInt(numAccount);
         amount = rand.nextInt(10);

         // The following statement should be in one line.
         System.out.println("Transfer $" + amount + " from Acct-" +
                                    fromAcctArrayIndex + " to " + toAcctArrayIndex);

         // Initiate the transfer.
         transferMoney(accounts[fromAcctArrayIndex], accounts[toAcctArrayIndex], amount);
      }
   }

   public void transferMoney(Account fromAcct, Account toAcct, int amount) {
      synchronized(fromAcct) {
         synchronized (toAcct) {
            if (fromAcct.getBalance() < 0) {
               System.out.println("Insufficient Funds");
               System.exit(1);
            } else {
               fromAcct.debit(amount);
               toAcct.credit(amount);
            }
         }
      }
   }
}
```

<u>Exercise 1</u>

a) Modify the code given in example 1 to ensure that all the threads will acquire both the locks in the same order. You can embed an account number in each account, and all the threads will acquire the locks based on the "value" of account number. For example, first acquire the lock associated with the account with a larger account number, then follow by the lock associated with the account with a smaller account number.

b) Modify the code given in example 1 to prevent deadlock by using the timed lock feature found in ReentrantLock. A thread can attempt to acquire a lock by using the tryLock() method. This method will immediately return true if the lock is available, or false otherwise. In this case, a thread will not be waiting for a lock infinitely. However, you need to put your tryLock() method in an if statement to check if the thread can acquire the lock (since the method returns either true or false). In addition to this, the if statement should be enclosed in a while loop to ensure that the thread can keep trying until it can acquire the lock and proceed from there.

B. Condition

Up to this point, we have covered how the wait() method and the notify()/notifyAll() method can be used to solve the producer-consumer problem, as well as the readers and writers problem. However, these methods can only work with the intrinsic lock. Although we should not fully abandon the intrinsic lock, we need to know the alternatives when we are required to use the extrinsic lock to solve certain problems.

The alternative to each of them is known as the await() method and the signal()/signalAll() method. But before we can call this method, we need to the extrinsic lock to create a Condition object. This is somewhat similar to create a waiting list to keep track of threads that are waiting for certain events to occur. As shown below is a simple example the demonstrates how to create a Condition object and make use of the await() method and the signal() method to coordinate the execution of two threads (boss and worker).

In this case, it is either the boss that will need to first wait for the worker to get ready, or the worker that will need to first wait for the boss to submit an order. It depends on which is selected to start first. When one is waiting, another one will notify the one that is waiting when the condition to proceed is true. As you can see from the example, it is possible to create multiple waiting lists associated with the same lock. With this, threads that are waiting for the same condition can be put onto the same list. This is more efficient than putting all the threads, which could be waiting for different conditions, onto the same list.

Take the readers and writers problem as an example, where multiple writers and readers are waiting to write and read from a shared structure. Imagine we put all the writers and readers in the same list, calling the notifyAll()/signalAll() method is going to wake up all of them. If the purpose behind the call method is to wake up only another writer to start writing, then the readers actually woke up for nothing.

```
//Example 2
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class TestClass2 {
   public static void main(String[] args) {
      Order order = new Order();
      Thread bossThread = new Thread(new Boss(order));
      Thread workerThread = new Thread(new Worker(order));
      bossThread.start();
      workerThread.start();
   }
}
```

```
class Boss implements Runnable {
   private Order order;
   Boss(Order order) {
      this.order = order;
   }

   @Override
   public void run() {
      while(true) {
         order.addOrder();
      }
   }
}

class Worker implements Runnable {
   private Order order;
   Worker(Order order) {
      this.order = order;
   }

   @Override
   public void run() {
      while(true) {
         order.processOrder();
      }
   }
}

class Order {
   private boolean hasOrder = false;
   private boolean workerReady = false;
   ReentrantLock lock = new ReentrantLock();
   Condition noOrder = lock.newCondition();
   Condition noWorker = lock.newCondition();

   public void addOrder() {
      lock.lock();
      try {
         while (workerReady == false) {
            noWorker.await();
         }
         hasOrder = true;
         System.out.println("Added New Order");
         noOrder.signal();
      } catch (InterruptedException e) {
         e.printStackTrace();
      } finally {
         lock.unlock();
      }

// Continue next page.
```

```
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

    public void processOrder() {
        lock.lock();
        try {
            workerReady = true;
            noWorker.signal();
            while (hasOrder == false) {
                noOrder.await();
            }
            workerReady = false;
            System.out.println("Process Order");
            hasOrder = false;
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

Exercise 2

a) A printer is waiting for printing tasks that could come from computer A or computer B. When there is no printing task, the printer should stay in the waiting state. The printer will only wake up to perform a printing task when computer A or computer B sends a signal to the printer. The printer takes one second to complete a task, and the printer will not accept new tasks while it is printing. After sending each printing task, the computer should also pause for half a second before sending the next task. Simulate the scenario by using the await() method and the signal()/signalAll() method.

b) Implement a queue (buffer) to keep track of the printing tasks to be completed by a printer. You are only required to add the file names of those files that need to be printed to the queue. With this, the printer should be able to continue receive printing task while it is printing. Assume the printer will take two seconds to complete a printing task. Next, create two computers that each will send 10 printing tasks to the queue. The computer should pause for half a second before sending the next printing task. Simulate the scenario by using the await() method and the signal()/signalAll() method.