THE UNIVERSITY OF SYDNEY

SOFT3410 Lab 2

Deadlock

The goal of this lab is to illustrate the problem of deadlock. By the end you will have seen examples of deadlock, and be able to understand how the misuse of locks can generate deadlock.

Exercise 1: A bank with single customer

To begin with we will use locks to make last week's single account Bank thread-safe.

```
1  public class Bank {
2      private int account = 100;
3      public static void main(String[] args) {
4          Bank account = new Bank();
5          Thread adding = new Thread(new AddMoney(account));
6          Thread subtracting = new Thread(new TakeMoney(account));
7          adding.start();
8          subtracting.start();
9      }
10
11     public void addMoney(int amount) {
12         if (amount < 1) {
13             System.out.print("Trying to add a negative amount to the account!");
14             System.out.println(" Transaction rejected.");
15             return;
16         }
17         int newBalance = account + amount;
18         System.out.print("Account balance is " + account + ". Adding " + amount + ".");
19         account = newBalance;
20         System.out.println("New balance is " + account + ".");
21         return;
22     }
23
24     public void subtractMoney(int amount) {
25         if (amount < 1) {
26             System.out.print("Trying to subtract a negative amount from the account!");
27             System.out.println(" That's generous, but the transaction is rejected.");
28             return;
29         }
30         int newBalance = account - amount;
31         System.out.print("Account balance is " + account + ". Subtracting " + amount + ".");
32         account = newBalance;
```

```
33          System.out.println("New balance is " + account + ".");
34          return;
35      }
36  }
```

Next we implement our two thread classes.

```
1   public class AddMoney implements Runnable {
2       private Bank account;
3       public AddMoney(Bank account) {
4           this->account = account;
5       }
6       public void run() {
7           for (int i = 0; i < 60; ++i) {
8               try {
9                   Thread.sleep(500);
10              } catch (Exception e) {
11                  System.err.println("Already interrupted.");
12              }
13              account.addMoney(1000);
14          }
15      }
16  }
17
18  public class TakeMoney implements Runnable {
19      private Bank account;
20      public AddMoney(Bank account) {
21          this->account = account;
22      }
23      public void run() {
24          for (int i = 0; i < 60; ++i) {
25              try {
26                  Thread.sleep(500);
27              } catch (Exception e) {
28                  System.err.println("Already interrupted.");
29              }
30              account.subtractMoney(1000);
31          }
32      }
33  }
```

To protect our bank account from race conditions, we will add a locking mechanism. Add a Lock variable to your bank:

```
1   Lock lock = new ReentrantLock();
```

Now modify the AddMoney and TakeMoney run methods to acquire the lock before modifying the account, then release the lock when done.

If you are successful you should have the correct account balance after execution.

Duration: 25 min

Exercise 2: Multiple locks

We successfully protected access to a single resource by requiring acquisition of a lock before use. Now we add a second resource - a second account.

Add a new class, TransferMoney:

```
public class TransferMoney implements Runnable {
    private Bank from;
    private Bank to;
    public AddMoney(Bank from, Bank to) {
        this->from = from;
        this->to = to;
    }
    public void run() {
        for (int i = 0; i < 60; ++i) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                System.err.println("Already interrupted.");
            }
            // lock the first account
            // lock the second account
            // subtract money from one
            // add money to the other
            // release locks
        }
    }
}
```

Modify the main method to create two bank accounts, and transfer money from the first to the second. Do any concurrency issues occur?

Duration: 15 min

Exercise 3: Deadlock

We successfully protected access to two resources by requiring acquisition of locks before use. Every access was similar, though. What happens if we also try to transfer money from the second account to the first?

You should soon see concurrency issues that halt your program. Explain the source of the problem, and why similar code worked in the previous question

Duration: 10 min

Note

In this example we ensured that the bank accounts always had valid data. However, this was not enough as transferring between accounts was blocked.