University of Virginia
CS 2110: Software Development Methods
Prof. Nada Basit

Reading: see Piazza post & Collab announcement

Chapter 21 – Multithreading/Concurrence (Pt. III)

Topic

- Avoiding Deadlocks

Avoiding Deadlocks

- A **deadlock** occurs if **no thread can proceed** because each thread is waiting for another to do some work first
- `BankAccount` example:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
         Wait for the balance to grow
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

Avoiding Deadlocks (2)

- How can we wait for the balance to grow?
- We can’t simply call `sleep` inside `withdraw` method; thread will block all other threads that want to use `balanceChangeLock`
- In particular, no other thread can successfully execute `deposit`
- Other threads will call `deposit`, but will be blocked until `withdraw` exits
- But `withdraw` doesn’t exit until it has funds available
- ***DEADLOCK !!***

Condition Objects (1)

- To overcome problem, use a **condition object**
- **Condition objects** allow a thread to temporarily release a lock, and to regain the lock at a later time
- Each condition object belongs to a specific lock object

Condition Objects (2)

- You obtain a condition object with `newCondition` method of `Lock` interface:

```
public class BankAccount
{
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition;
   . . .
   public BankAccount()
   {
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition =
         balanceChangeLock.newCondition();
      . . .
   }
}
```

Condition object given a name that *describes* the condition

condition object belongs to a *specific* lock object

Condition Objects (3)

- It is customary to give the condition object a name that describes condition to test; e.g. “sufficient funds”

  Example from code:

```
private Condition sufficientFundsCondition;
```

- You need to implement an appropriate **test**
  - A condition needs to be checked

Condition Objects (4)

- As long as test is not fulfilled, call `await` on the condition object:

```
public void withdraw(double amount)
{
   balanceChangeLock.lock();
   try
   {
      while (balance < amount)
      {
         sufficientFundsCondition.await();
      }
      . . .
   }
   finally
   {
      balanceChangeLock.unlock();
   }
}
```

the condition needs to be checked

the condition object calls "await"

Condition Objects (5)

- Calling **`await`**
  - Makes the current thread **wait**
  - Allows another thread to acquire the lock object
- To **unblock**, another thread must execute **`signalAll`** *on the same condition object* :

```
sufficientFundsCondition.signalAll();
```

- **`signalAll`** unblocks all threads waiting on the condition

Aside:

- `signal` randomly picks just *one* thread waiting on the object and unblocks it
- `signal` can be more efficient, but you need to know that *every* waiting thread can proceed
- **Recommendation:** always call **`signalAll`**

BankAccount.java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
*/
public class BankAccount
{
   private double balance;
   private Lock balanceChangeLock;
   private Condition sufficientFundsCondition; // condition object

   /**
      Constructs a bank account with a zero balance.
   */
   public BankAccount()
   {
      balance = 0;
      balanceChangeLock = new ReentrantLock();
      sufficientFundsCondition = balanceChangeLock.newCondition();
   }
```

BankAccount.java (cont.)

```

   /**
      Deposits money into the bank account.
      @param amount the amount to deposit
   */
   public void deposit(double amount)
   {
      balanceChangeLock.lock();
      try
      {
         System.out.print("Depositing " + amount);
         double newBalance = balance + amount;
         System.out.println(", new balance is " + newBalance);
         balance = newBalance;
         sufficientFundsCondition.signalAll(); // money added!
                                               // announce it!
      }
      finally
      {
         balanceChangeLock.unlock();
      }
   }
```

BankAccount.java (cont.)

```
   /**
      Withdraws money from the bank account.
      @param amount the amount to withdraw
   */
   public void withdraw(double amount)
         throws InterruptedException
   {
      balanceChangeLock.lock();
      try
      {
         while (balance < amount)  // if not enough balance…
         {
            sufficientFundsCondition.await(); // …wait!
         }
         System.out.print("Withdrawing " + amount);
         double newBalance = balance - amount;
         System.out.println(", new balance is " + newBalance);
         balance = newBalance;
      }
   }
```

BankAccount.java (cont.)

```java
         finally
         {
            balanceChangeLock.unlock();
         }
      }

      /**
         Gets the current balance of the bank account.
         @return the current balance
      */
      public double getBalance()
      {
         return balance;
      }
   }
```

BankAccountThreadRunner.java (cont.)

Program Run:

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
...
Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
```

Notice how the balance doesn't drop below zero!
This is a more realistic situation and we can achieve this by using locks and condition objects.

Eclipse DEMO

- Watch the following demos presented in class

- Bank Example:
  [Thread Example 6 – Bank Deadlock]
- BankAccount.java
- BankAccountThreadRunner.java
- DepositRunnable.java
- WithdrawRunnable.java

REVIEW

Review: Running Threads

- A thread is a program unit that is executed concurrently with other parts of the program.
- The `start` method of the `Thread` class starts a new thread that executes the `run` method of the associated `Runnable` object.
- The `sleep` method puts the current thread to sleep for a given number of milliseconds.
- When a thread is interrupted, the most common response is to terminate the `run` method.
- The thread scheduler runs each thread for a short amount of time, called a `time slice`.

Review: Terminating Threads

- A thread terminates when its run method terminates.
- The `run` method can check whether its thread has been interrupted by calling the `interrupted` method.

Review: Race Conditions

- A **race condition** occurs if the effect of multiple threads on shared data depends on the order in which the threads are scheduled.

Review: Synchronizing Object Access

- By calling the `lock` method, a thread acquires a `Lock` object. Then no other thread can acquire the lock until the first thread releases the lock.

Review: Avoiding Deadlocks

- A **`deadlock`** occurs if no thread can proceed because each thread is waiting for another to do some work first.
- Calling **`await`** on a condition object makes the current thread wait and allows another thread to acquire the lock object.
- A waiting thread is blocked until another thread calls **`signalAll`** on the condition object for which the thread is waiting.