
# Deadlocks

This presents deadlock material using Java examples.

## 7.1 Deadlock in a Multithreaded Java Program

Before we proceed to a discussion of handling deadlocks, let's see how deadlock can occur in a multithreaded Java program, as shown in Figure 7.1. In this example, we have two threads—`threadA` and `threadB`—as well as two reentrant locks—`first` and `second`. (Recall from Chapter 5 that a reentrant lock acts as a simple mutual exclusion lock.) In this example, `threadA` attempts to acquire the locks in the order (1) `first`, (2) `second`; `threadB` attempts using the order (1) `second`, (2) `first`. Deadlock is possible in the following scenario:

`threadA` → `second` → `threadB` → `first` → `threadA`

Note that, even though deadlock is possible, it will not occur if `threadA` is able to acquire and release the locks for `first` and `second` before `threadB` attempts to acquire the locks. This example illustrates a problem with handling deadlocks: it is difficult to identify and test for deadlocks that may occur only under certain circumstances.

## 7.2 Handling Deadlocks in Java

As noted earlier, the JVM does nothing to manage deadlocks; it is up to the application developer to write programs that are deadlock-free. In the remainder of this section, we illustrate how deadlock is possible when selected methods of the core Java API are used and how the programmer can develop programs that appropriately handle deadlock.

In Chapter 4, we introduced Java threads and some of the API that allows users to create and manipulate threads. Two additional methods of the `Thread` class are the `suspend()` and `resume()` methods, which were deprecated in later versions of the Java API because they could lead to deadlock. (A deprecated

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}

class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}

public class DeadlockExample {
       // Figure Figure 7.2
}
```

**Figure 7.1** Deadlock example.

```
public static void main(String arg[]) {
   Lock lockX = new ReentrantLock();
   Lock lockY = new ReentrantLock();

   Thread threadA = new Thread(new A(lockX,lockY));
   Thread threadB = new Thread(new B(lockX,lockY));

   threadA.start();
   threadB.start();
}
```

**Figure 7.2** Creating the threads (continuation of Figure 7.1).

method indicates that the method is still part of the Java API, but its use is discouraged.)

The `suspend()` method suspends execution of the thread currently running. The `resume()` method resumes execution of a suspended thread. Once a thread has been suspended, it can continue only if another thread resumes it. Furthermore, a suspended thread continues to hold all locks while it is blocked. Deadlock is possible if a suspended thread holds a lock on an object and the thread that can resume it must own this lock before it can resume the suspended thread.

Another method, `stop()`, has been deprecated as well, but not because it can lead to deadlock. Unlike the situation in which a thread has been suspended, when a thread has been stopped, it releases all the locks that it owns. However, locks are generally used in the following progression: (1) acquire the lock, (2) access a shared data structure, and (3) release the lock. If a thread is in the middle of step 2 when it is stopped, it will release the lock; but it may leave the shared data structure in an inconsistent state. In Section 4.5, we discussed how to terminate a thread using deferred cancellation rather than asynchronously canceling a thread using the `stop()` method. Here, we present a strategy for suspending and resuming a thread without using the deprecated `suspend()` and `resume()` methods.

The program shown in Figure 7.3 is a multithreaded applet that displays the time of day. When this applet starts, it creates a second thread (which we will call the *clock thread*) that outputs the time of day. The `run()` method of the clock thread alternates between sleeping for one second and then calling the `repaint()` method. The `repaint()` method ultimately calls the `paint()` method, which draws the current date and time in the browser's window.

This applet is designed so that the clock thread is running while the applet is visible; if the applet is not being displayed (as when the browser window has been minimized), the clock thread is suspended from execution. This is accomplished by overriding the `start()` and `stop()` methods of the `Applet` class. (Be careful not to confuse these with the `start()` and `stop()` methods of the `Thread` class.) The `start()` method of an applet is called when an applet is first created. If the user leaves the web page, if the applet scrolls off the screen, or if the browser window is minimized, the applet's `stop()`method is called. If the user returns to the applet's web page, the applet's `start()` method is called again.

The applet uses the Boolean variable ok to indicate whether the clock thread can run. This variable will be set to true in the start() method of the applet, indicating that the clock thread can run. The stop() method of the applet will set it to false. The clock thread will check the value of this Boolean variable in its run() method and will only proceed if it is true. Because the thread for the applet and the clock thread will be sharing this variable, access to it will be

```
import java.applet.*;
import java.awt.*;


public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
       while (true) {
          try {
             // sleep for 1 second
             Thread.sleep(1000);

             // repaint the date and time
             repaint();

             // see if we need to suspend ourself
             synchronized (mutex) {
                while (ok == false)
                   mutex.wait();
             }
          }
          catch (InterruptedException e) { }
       }
    }

    public void start() {
       // Figure Figure 7.4
    }

    public void stop() {
       // Figure Figure 7.4
    }

    public void paint(Graphics g) {
      g.drawString(new java.util.Date().toString(),10,30);
    }
}
```

**Figure 7.3** Applet that displays the date and time of day.

controlled through a `synchronized` block. This program is shown in Figure 7.4.

If the clock thread sees that the Boolean value is false, it suspends itself by calling the `wait()` method for the object `mutex`. When the applet wishes to resume the clock thread, it sets the Boolean variable to true and calls `notify()` for the `mutex` object. This call to `notify()` awakens the clock thread. It checks the value of the Boolean variable and, seeing that it is now true, proceeds in its `run()` method, displaying the date and time.

## 7.3 Circular Wait Prevention Scheme

We can accomplish this scheme in an application program by developing an ordering among all synchronization objects in the system. All requests for synchronization objects must be made in increasing order. For example, if the lock ordering in the Java program shown in Figure 7.1 was

$$F(\texttt{first}) = 1$$
$$F(\texttt{second}) = 5$$

```
/**
 * This method is called when the applet is
 * started or we return to the applet.
 */
public void start() {
    ok = true;

    if (clockThread == null) {
       clockThread = new Thread(this);
       clockThread.start();
    }
    else {
       synchronized(mutex) {
          mutex.notify();
       }
    }
}

/**
 * This method is called when we
 * leave the page the applet is on.
 */
public void stop() {
    synchronized(mutex) {
       ok = false;
    }
}
```

**Figure 7.4** `start()` and `stop()` methods for the applet (continuation of Figure 7.3).

then threadB could not request the locks out of order.

Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering. Also note that the function $F$ should be defined according to the normal order of usage of the resources in a system. For example, because the tape drive is usually needed before the printer, it would be reasonable to specify that $F(\text{tape drive}) < F(\text{printer})$.

To develop a lock ordering, Java programmers are encouraged to use the method System.identityHashCode(), which returns the value that would be returned by the default value of the object's hashCode() method. For example, to obtain the value of identityHashCode() for the first and second locks in the Java program shown in Figure 7.1, you would use the following statements:

```
int firstOrderingValue = System.identityHashCode(first);
int secondOrderingValue = System.identityHashCode(second);
```

Although ensuring that resources are acquired in the proper order is the responsibility of application developers, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order and deadlock is possible. One lock-order verifier, which works on BSD versions of UNIX such as FreeBSD, is known as **witness**. Witness uses mutual-exclusion locks to protect critical sections, as described in Chapter 5; it works by dynamically maintaining the relationship of lock orders in a system. Let's use the program shown in Figure 7.1 as an example. Assume that threadA is the first to acquire the locks and does so in the order (1) first, (2) second. Witness records the relationship that first must be acquired before second. If threadB later attempts to acquire the locks out of order, witness generates a warning message on the system console.

Finally, it is important to note that imposing a lock ordering does not guarantee deadlock prevention if locks can be acquired dynamically. For example, assume we have a method that transfers funds between two accounts. To prevent a race condition, we use the object lock associated with each Account object in a synchronized block. The code appears as follows:

```
void transaction(Account from, Account to, double amount) {
   synchronized(from) {
      synchronized(to) {
         from.withdraw(amount);
         to.deposit(amount);
      }
   }
}
```

Deadlock is possible if two threads simultaneously invoke the transaction() method, transposing different accounts. That is, one thread might invoke

```
transaction(checkingAccount, savingsAccount, 25);
```

and another might invoke

```
transaction(savingsAccount, checkingAccount, 50);
```

We leave it as an exercise for the reader to figure out a solution to fix this situation.

## Programming Problems

**7.1** In Section 7.3, we describe a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if locks are acquired dynamically and two threads simultaneously invoke the `transaction()` method. Fix the `transaction()` method to prevent deadlock from occurring. (Hint: consider using `System.identityHashCode()`.)

**7.2** A single-lane bridge connects the two Vermont villages of North Tunbridge and South Tunbridge. Farmers in the two villages use this bridge to deliver their produce to the neighboring town. The bridge can become deadlocked if both a northbound and a southbound farmer get on the bridge at the same time (Vermont farmers are stubborn and are unable to back up.) Using either Java semaphores or Java synchronization, design an algorithm that prevents deadlock.
To test your implementation, design two threads, one representing a northbound farmer and the other representing a farmer traveling southbound. Once both are on the bridge, each will sleep for a random period of time to simulate traveling across the bridge.
Initially, do not be concerned about starvation (the situation in which northbound farmers prevent southbound farmers from using the bridge, or vice versa).

**7.3** Modify your solution to Exercise 7.2 so that it is starvation-free.

## Programming Projects

In this project you will write a Java program that implements the banker's algorithm discussed in Section 7.5.3. Several customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request is denied if it leaves the system in an unsafe state.

### The Bank

The bank will employ the strategy outlined in Section 7.5.3, whereby it will consider requests from $n$ customers for $m$ resources. The bank will keep track of the resources using the following data structures:

```
public interface Bank
{
 /**
  * Add a customer
  * customerNumber - the number of the customer
  * maximumDemand - the maximum demand for this customer
  */
 public void addCustomer(int customerNum,
                         int[] maximumDemand);

 /**
  * Output the value of available, maximum,
  * allocation, and need
  */
 public void getState();

 /**
  * Request resources
  * customerNumber - the customer requesting resources
  * request - the resources being requested
  */
 public boolean requestResources(int customerNumber,
                         int[] request);

 /**
  * Release resources
  * customerNumber - the customer releasing resources
  * release - the resources being released
  */
 public void releaseResources(int customerNumber,
                         int[] release);
}
```

**Figure 7.5** Interface showing the functionality of the bank.

```
int numberOfCustomers; // the number of customers
int numberOfResources; // the number of resources

int[] available; // the available amount of each resource
int[][] maximum; // the maximum demand of each customer
int[][] allocation; // the amount currently allocated
                    // to each customer
int[][] need; // the remaining needs of each customer
```

The functionality of the bank appears in the interface shown in Figure 7.5. The implementation of this interface will require adding a constructor that is passed the number of resources initially available. For example, suppose we have three resource types with 10, 5, and 7 resources initially available. In this case, we can create an implementation of the interface using the following technique:

```
Bank theBank = new BankImpl(10,5,7);
```

The bank will grant a request if the request satisfies the safety algorithm outlined in Section 7.5.3.1. If granting the request does not leave the system in a safe state, the request is denied.

### Testing Your Implementation

You can use the file `TestHarness.java`, which is available on Wiley Plus, to test your implementation of the `Bank` interface. This program expects the implementation of the `Bank` interface to be named `BankImpl` and requires an input file containing the maximum demand of each resource type for each customer. For example, if there are five customers and three resource types, the input file might appear as follows:

```
7,5,3
3,2,2
9,0,2
2,2,2
4,3,3
```

This indicates that the maximum demand for customer$_0$ is 7, 5, 3; for customer$_1$, 3, 2, 2; and so forth. Since each line of the input file represents a separate customer, the `addCustomer()` method is to be invoked as each line is read in, initializing the value of `maximum` for each customer. (In the above example, the value of `maximum[0][]` is initialized to 7, 5, 3 for customer 0; `maximum[1][]` is initialized to 3, 2, 2; and so forth.)

Furthermore, `TestHarness.java` also requires the initial number of resources available in the bank. For example, if there are initially 10, 5, and 7 resources available, we invoke `TestHarness.java` as follows:

```
java TestHarness infile.txt 10 5 7
```

where `infile.txt` refers to a file containing the maximum demand for each customer followed by the number of resources initially available. The `available` array will be initialized to the values passed on the command line.