Thread Data and Synchronization Issues

Synchronization Issues

Having 2 independant threads running at the same time can cause a new set of problems. Problems begin to show up when 2 or more threads are accessing a datastructure and one thread updates the datastructure.

Consider the following example of a Bank class. We have 5 customers that have both a checking account and a savings account. Note the savings and checking arrays below which indicate how much money each customer starts out with.

Execute it and observe the following sequence of events:

1. The main routine (which is actually running in a Thread that we didn't start), **`starts`** up an **`audit Thread.`**
2. The audit Thread, sleeps for **`5 seconds`** and then calls printTotals. The **`printTotals`** routine prints out the total amount of money the bank has in savings accounts, the total money in checking accounts and the grand total of all money (savings + checking).
3. The main routine (i.e. the original Thread) now makes the 2 calls:
   `ba.chk2save(2, 40);`
   `ba.chk2save(3, 50);`
4. Each call to chk2save(customer_index, amount_money) , moves the amount of money specified from the checking account to the saving account for customer index. Each call to chk2save, first adds the money to the savings account and then removes it from the checking account.
5. Suppose that our **`audit thread`** happens to get **`scheduled`** while the main Thread is in the **`middle of a chk2save`** call. Suppose the the chk2save call has added money to savings but hasn't removed the money from checking. Our audit will show that our **`bank has more total money`** than it really has. This is because savings has been increased, but checking hasn't been reduced.
6. The problem is that there is a region of code in chk2save where we hope we don't get interrupted. Sometimes regions of code like this are referred to as **`critical regions`**.
7. Normally the **`chances`** of this happening would be **`very small`**, because our Main Thread would need to lose the CPU in a bad spot and the audit thread would need execute during that time. I have **`added a sleep`** call in chk2save to make the window of vulnerability extremely large.
8. **`Small windows`** of opportunity are **`deadly`**, because you may have a large system that passed your quality assurance testing and you never hit the small window of vulnerability. Unfortunately, when the system went to production and thousands of customers started hitting your system, you hit the vulnerable window. So every so often your system does wierd things and the head of your company finds out about it and you are told that your job is on the line to solve this problem. Sound familiar.

Bank.java

```
import java.util.*;

class Bank implements Runnable
{
    int savings[] = {100, 200, 300, 400, 500};
    int checking[] = {100, 200, 300, 400, 500};

    void chk2save(int index, int amount)
    {
        System.out.println("-----Starting chk2save: " +index + "," +amount);
        if (index >= 0 && index < savings.length)
        {
            if (amount <= checking[index])
            {
                // Begin Vulnerable region
                savings[index] += amount;
                // The following sleep, makes our
                // vulnerable window huge
                try
                {
                    Thread.sleep(6000);
                }
                catch (InterruptedException e){}

                checking[index] -= amount;
                // End Vulnerable region
            }
        }
        System.out.println("-----Ending chk2save");
    }


    void printTotals()
    {
        int totalSavings=0, totalChecking=0;

        System.out.println(); // blank line
        System.out.println("**********************************");
        for (int i=0; i < savings.length; i++)
            totalSavings += savings[i];
        System.out.println("Total Savings = " + totalSavings);

        for (int i=0; i < checking.length; i++)
            totalChecking += checking[i];
        System.out.println("Total Checking = " + totalChecking);
        System.out.println("*******Grand Total = " +
                (totalSavings + totalChecking));
        System.out.println(); // blank line

    }

    public void run()
    {
        boolean continueFlag=true;
        while (continueFlag)
        {
            printTotals();
            try
            {
                Thread.sleep(5000);
            }
            catch (InterruptedException e)
            {
                System.out.println("auditor is exitting");
                continueFlag = false;
            }
        }
    }
    void delay(int num_seconds)
    {
        try {
            Thread.sleep(1000 * num_seconds);
        }
        catch (InterruptedException e){}
    }
    public static void main(String[] args)
    {
        String str="";
        Bank ba = new Bank();

        Thread audit = new Thread(ba);
        audit.start();

        ba.delay(6);        // delay for 6 seconds
        ba.chk2save(2, 40);

        ba.delay(6);        // delay for 6 seconds
        ba.chk2save(3, 50);

        ba.delay(25);       //delay for 25 seconds

        audit.interrupt();
        System.out.println("BYE");
    }

}
```

Fixing Synchronization Problems

Java provides an extremely simple solution. I have solved these problems in other languages, and the Java solution is very nice and clean relative to approaches in older languages.

The keyword **`synchronized`** provides the solution. First of all we need to add the keyword synchronized to the printTotals routine like so:

```
synchronized void printTotals()
```

We also need to **synchronize chk2save**. You can synchronize a complete method or you can just synchronize a block of code inside of a method. So for example, here are 2 possible replacements for our chk2save routine.

Fix the original Bank.java program with either of the following solutions and rerun the Bank program.

synchronized on the whole method:

```
    synchronized void chk2save(int index, int amount)
    {
        System.out.println("Doing chk2save: " +index + "," +amount);
        if (index >= 0 && index < savings.length)
        {
            if (amount <= checking[index])
            {
                savings[index] += amount;
                try
                 {
                    Thread.sleep(6000);
                 }
                catch (InterruptedException e){}

                checking[index] -= amount;
            }
        }
    }
```

synchronized on a block of code inside of a method:

```
    void chk2save(int index, int amount)
    {
        System.out.println("Doing chk2save: " +index + "," +amount);
        if (index >= 0 && index < savings.length)
        {
            if (amount <= checking[index])
            {
                synchronized(this)
                {
                    savings[index] += amount;
                    try
                    {
                        Thread.sleep(6000);
                    }
                    catch (InterruptedException e){}

                    checking[index] -= amount;
                }
            }
        }
    }
```

A synchronized block of code or a synchronized method works as follows:

1. **Only one thread can be in any synchronized section of an object instance at any one time.** If one or more other threads want to execute a synchronized section of code, the Java Run Time system will **Queue up the other threads** until the first thread exits the synchronized section. Once the **first thread exits**, the Java Run Time system will **schedule the next thread** waiting to execute the synchronized section.
2. Note that **synchronization** occurs on an **object instance**. In otherwords, if I have 2 instances of some object, each object can have a thread in its critical section. We will try to demonstrate this later. The Synchronization technique is a little like the "Talking Stick" we used to use in our family meetings (we had 4 kids). To avoid having multiple family members talking at the same time, we created the rule that you had to pick up the "Talking stick" when you wanted to talk. When you were done, you put the stick down. Then someone else could pick up the stick and talk. In the case of a Java object instance only one thread can be in a synchronized region of a given object instance at a time. When a thread exits a synchronized region, another thread wishing to enter a synchronized region can enter. Obtaining the privilege to enter a synchronized region is like picking up the "talking stick". Of course in Java we don't use the terminology "talking stick". We sometimes use the "lock" terminology for the the Java equivalent of the "talking stick".
3. If your Class has more than one synchronized section or method, then the lock applies to the whole object. This means that only one of your synchronized regions can be executing an any point in time.
4. Consider the following snippet of code. If one thread is in the middle of
   aThing.put(int);
   Then a second thread will be suspended if it tries to execute:
   value = aThing.get();
   However, the second thread could execute the following with no suspension:
   value = anotherThing.get();
   Also note that **aThing.do_something()** is allowed at all times because this method is not synchronized

```
class Something
{
  synchronized int get()
  { // some code
    another_synchronized_method();
  }
  synchronized void put(int)
  { // some code
  }
  synchronized void another_synchronized_method()
  {
     // some code
  }
  void do_something()
  { // some code
  }
}


Something aThing = new Something();
Something anotherThing = new Something();
```

5. Note that **`once a Thread has obtained the object lock`**, the Thread can call any number of other synchronized methods. In the above example, the get method is synchronized and it calls another synchronized method called **`another_synchronized_method`**. No problem with this.

Deadlocks

It is possible to have 2 threads that are deadlocked(i.e. neither thread can make progress and both threads will wait forever) because of a poorly designed set of synchronized methods.

1. Create the 2 classes Deadlock.java and DeadlockDemo.java and run them. Note that he way they are currently, everything runs to completion.
2. Now make a simple change. Just add the key word **`synchronized`** to the beginning of the Deadlock.java statement:
   `void printOther(Deadlock d)`
   so that it reads:
   `synchronized` `void printOther(Deadlock d)`
3. So what happened? We have 2 threads. We have the main thread and the child thread that is created and started. We have 2 instances of Deadlock ... deadlock1, and deadlock2. I have created a situation where one of the threads owns the lock to the deadlock1 object and is going to wait forever for the lock for the deadlock2 object. The other thread has the opposite situation .... it owns the lock to the deadlock2 object and is going to wait forever for the lock to the deadlock1 object.
4. How did this happen?
   - One of the threads (and we really don't know which) quickly gained the lock to one of the Deadlock objects.
   - It then quickly acquired the lock to the other Deadlock object with a call to the get method.
   - The get method did a short sleep and then exitted.
   - The vaue "red" gets printed, but in the mean time, the sleep call allowed the other thread to acquire the other Deadlock lock with a call to printOther.
   - At this point nothing can procede because each thread has one of the locks and would like to get the other. But it isn't going to happen with this stubborn ***hang onto a lock forever and wait forever for another lock strategy***.
5. This example is a bit contrived. I wanted to have my example as short and simple as possible. If you are not careful, it is quite possible to have a deadlock in a much more complicated situation. It may be real tough to see what is going on except for the fact that your system is hung.
6. Words to the wise: Be leary of staying in a synchronized region for long periods of time.
7. Be leary of situations where you actually need more than 1 synchronization lock at a time.

8. If you absolutely must have 2 locks at the same time, sometimes it is possible to establish a rule that says you must acquire lock1 first and then lock2. You will note that in our example the 2 threads are getting their 2 locks in different order.

Deadlock.java

```
import java.util.ArrayList;

public class Deadlock {
    ArrayList<String> arr = new ArrayList<String>();
    Deadlock()
    {
        arr.add("red");
        arr.add("orange");
        arr.add("green");
        arr.add("black");
        arr.add("brown");
        arr.add("white");
        arr.add("magenta");
        arr.add("yellow");
    }

    synchronized int getSize()
    {
        return arr.size();
    }
    synchronized String get(int i)
    {
        try
        {
            Thread.sleep(10);
        }
        catch (InterruptedException e){}

        return arr.get(i);
    }
    void printOther(Deadlock d)
    {
        int len = d.getSize();
        for (int i=0; i < len; i++)
        {
            System.out.println(d.get(i));
        }
    }

}
```

DeadlockDemo.java

```
public class DeadlockDemo implements Runnable{

    Deadlock deadlock1 = new Deadlock();
    Deadlock deadlock2 = new Deadlock();

    public void run()
    {
        System.out.println("Starting Thread");
        deadlock2.printOther(deadlock1);
        System.out.println("Thread exiting");
    }


    public static void main(String[] args) {
        DeadlockDemo dd = new DeadlockDemo();

        Thread t = new Thread(dd);
        t.start();

        dd.deadlock1.printOther(dd.deadlock2);
        System.out.println("Exiting main");
    }

}
```

Last Updated: October 2, 2013 9:10 PM