Thread Synchronization in Java

How to Get True Multithreading in Java

by Christopher McGee


Comment on this article

Introduction

At ITI we write both Java applications and Java applets for a variety of clients and situations. Many of the programmers working on these projects come from a C++/MFC background. The transition is remarkably smooth (much smoother than the transition in the opposite direction would be), but one thing that caused problems for us early on is the Java threading model. In working with other project teams in other companies, I have also noticed a misunderstanding of threading in Java.

When I work with other programmers, the easiest way I have found to explain the Java threading model is this: If you use "synchronized" for threading, then Java doesn't do multithreading. Java lets you think you've implemented multi-threaded processing, but in practice what Java does is disable multithreading. Instead, it queues up process threads, then executes them serially. This isn't multithreading. Rather, it is a scheme that Java uses to manage its inability to deliver true multithreading. The scheme is based on a concept called critical sections. What critical sections do is protect against more than one thread at a time entering a section of code. This concept is flawed because the code is not what you need to protect. You need to protect the underlying resource, which, if changed by more than one thread at a time, will blow up; or, if being read while being changed will yield corrupt data.

If you want to avoid serial thread execution, you need to take active steps in every program you write. In this article, my goal is to discuss some of the problems I have seen and helped debug in multi-threaded Java code. I will start by demonstrating the most common form of the problem, and then move into the different techniques you can use in the Java language to solve and prevent multi-threading bugs. In addition, I will show you how to create true multi-threading applications in Java using a little-known feature hidden in the Object class. I will demonstrate a Boolean Lock class that we use in most of our multi-threaded applications here at ITI.

- [Download the *.java files for this article]

Next Page

Previous Page Return to beginning of article

Thread Synchronization in Java

How to Get True Multithreading in Java

by Christopher McGee

Introduction

The most common manifestation of a threading problem is also the most insidious: your program locks up randomly. It does this on and off for months. No one can reproduce the problem predictably to see a pattern, so no one can figure out where the error might lie. It is very frustrating. At some point you simply attribute it to "general flakiness in the immature Java environment " and try to stop thinking about it.

In most cases that I have seen, a random problem like this is probably caused by a **race condition**. A race condition occurs when multiple threads try to access a resource simultaneously. Simultaneous access will either cause a deadlock, corrupt data, or an explosion. A deadlock means the threads get stuck on each other and they all lock up. An explosion, on the other hand, means an exception is thrown or the program GPFs (or core dumps). In the nicer form of an explosion, as when an access violation exception gets thrown, the system will kill just the offending thread (because your thread is not catching this exception), and the rest of your application will keep running. Regardless, in neither case is the occurrence pleasant for your application.

There is another nasty symptom that makes race conditions difficult to track: they are extremely timing-dependent. Because of this dependency, they may never occur when you are single-stepping through the code, but occur frequently when you are running the code at full speed. Or they may only happen in the release build of your program. If you see this sort of behavior, you almost certainly have a thread problem. (Experiencing this in the release build only could be resulting from a different memory initialization scheme, so make sure you are initializing all your variables; but this will be a consistent crash, while a race condition will not.)

Race conditions often happen because of interactions between wildly disparate parts of your program, but if you could distill the code down almost all race conditions end up looking something like this:

```
class UnsafeLogger
{
   // this is the critical resource, it could be
   //  a linked list, array, or any resource
   private File m_file;

   public void write( string s )
   {
      // file stuff simplified...
      m_file.open( "filename.log" );
      m_file.write( s );
      m_file.close();
   }
}

class Writer extends Thread
{
   private UnsafeLogger m_log;
   private String m_threadID;

   Writer( ThreadUnsafe log, String threadID )
   {
      m_log = log;
      m_threadID = threadID;
   }

   public void run()
   {
      while( true )
      {
         m_log.write( m_threadID );
         m_log.write( " is alive\n" );
      }
   }
```

```
}

class MainClass
{
   public static void main()
   {
      ThreadUnsafe log = new ThreadUnsafe();
      Writer writer1 = new Writer( log, "thread1" );
      Writer writer2 = new Writer( log, "thread2" );

      // fire up the threads
      writer1.start();
      writer2.start();
   }
}
```

Note that this code starts two identical threads at the bottom of the listing. This program will blow up when you run it because it will try to write to a file which is closed. Here is what happens:

Cycle	Thread 1	Thread 2	Note
1	Opens file		
2	Writes string		
		Opens file	Assuming that opening an already opened file is ignored
4	Closes file		
5		Writes string	Boom!
6		Closes file	

So if a thread tries to write to a file that is closed, it will blow up. However, it may not blow up if the timing is right:

Cycle	Thread 1	Thread 2	Note
1	Opens file		
2		Opens file	
3	Writes string		
4		Writes string	
5	Closes file		
6		Closes file	Assuming that closing an already closed file is ignored

In the above scenario, the thread does not blow up. Instead, what you see in the file is merged output:

```
Thread1 Thread2 is alive is alive
```

The application may even work perfectly if the timing is right:

Cycle	Thread 1	Thread 2	Note
1	Opens file		
2	Writes string		
3	Closes file		
4		Opens file	
5		Writes string	
6		Closes file	

So how do you fix this type of problem? What you want is for only one thread at a time to be able to access the critical resource (the file in this example). You need to put controls around the open-write-close cycle that allow only one thread in at a time.

Here is the fixed write method. It uses a "synchronized" function to attempt setting a "lock" on the object passed to it.

```
public void write( String s )
{
   synchronized( m_file )
   {
      m_file.open( "filename.log" );
      m_file.write( s );
      m_file.close();
    }
}
```

The **synchronized** function has two possible behaviors. If no other thread is using the resource, then the function returns immediately. If another thread already has the lock on the object then the function waits for that lock to be released before taking the lock for itself. It releases the lock at the end of the scope defined by **synchronized**.

This is what Java wants you to do. Java wants you to lock critical resources before you use them. That way, only one thread can access the critical resource at a time. Java wants all your thread-safe code to look like the following. (Soon, we'll see why this is not the best approach.)

```
CriticalResource m_criticalResource;

// lock resource before using
synchronized( m_criticalResource )
{
   // use resource
}
// end of scope unlocks resource
```

Here is what happens when you use the synchronized function in the example shown previously:

Cycle	Thread 1	Thread 2	Note
1	Get lock on file		
2	Open file		
3	Write to file		
4		Try to get lock on file	Has to wait; file is already locked
5	Close file	Still waiting	
6	Release lock on file		this happens at the end of the synchronized scope
6		Get lock on file	
6		Open file	
6		Write to file	
6		Close file	
6		Release lock on file	

The result is that while thread1 is messing with the file, thread2 has to wait. This keeps the logger from blowing up, but we may still see this in the log file:

```
Thread1Thread2 is alive
  is alive
```

What we really want is for the log messages of one thread to finish before the next thread begins writing to the log file. In this case the

programmer using the logger has to realize that more than one thread may be calling the logger at the same time. The logger can guarantee that any one message it gets goes out as one unit (called an "atomic operation"), but there is no way it can guarantee that multiple messages get sent out together. So the user needs to send the whole message to the logger at once, instead of a piece at a time. To do this, the following code:

```
m_log.write( m_threadID );
m_log.write( " is alive\n" );
```

becomes:

```
String logMessage = m_threadID + " is alive\n";
m_log.write( logMessage );
```

Here is another form of synchronized you will see:

```
public synchronized void write( String s )
{
   //...
}
```

This is the same as putting a lock on the "this" object around the whole contents of the method.

```
public void write( String s )
{
   synchronized( this )
   {
      //...
   }
}
```

Be warned, though - this can cause odd behavior in some versions of Java:

```
public static synchronized void write( String s )
{
   //...
}
```

There is no "this" object to synchronize on, so it gets converted to the following:

```
public static void write( String s )
{
   synchronized( __globalSynchronizeObject )
   {
      //...
   }
}
```

Some versions of Java use this same global object to synchronize some Java language static methods. This unexpected side-behavior results in the unfortunate situation where some calls you make from this static synchronized method will lock up. (1.2beta4 fixes this). You are probably better off being safe and using your own object to synchronize on, rather than leaving it to chance.

Next

Previous Page Return to beginning of article

Thread Synchronization in Java

How to Get True Multithreading in Java

by Christopher McGee

Locks, Spin Locks, & Critical Sections

The synchronized function alone is not enough to solve your thread problems. In many instances, you will also need to use spin locks. A spin lock is basically a thread that keeps polling continuously until a lock variable is changed.

The classical thread problem is called the "readers/writer problem." Say, for example, that you have a database in which you have some slow threads putting data into the database and you have many fast threads trying to access the database. In this case, because the readers are not changing the data, you want to allow all of them to access the database at the same time. However, the writers, which change the data, can only be allowed in one at a time. Furthermore, when a writer is writing, the readers can not be reading because they might access incomplete data.

Here is the way you solve this problem in pseudo code:

```
Lock     readLock;
Lock     writeLock;
int      readers = 0;

read()
{
   lock ( readlock );

   // if we are the first reader
   if( readers == 0 )
      // lock the writers untill we are done
      lock( writeLock );

   // we did the read lock so only one
   // reader will hit the == 0 condition and
   // lock the writers and make readers != 0
   readers++;

   unlock( readLock );

   // do reading
   // same as above, we did the read lock so
   // only one reader will unlock the writer

   lock( readLock );

   readers--;
   if( readers == 0 )
      unlock( writeLock );

   unlock( readLock );
}

write()
{
   getDataToWrite();

   lock( writeLock );

   // do writing

   unlock( writeLock );
}
```

The readLock makes sure only one reader locks and unlocks the writeLock. The writeLock makes sure that only one thread is writing at a time, and that no readers are reading during a write.

The problem is, reading this sort of functionality in Java is not intuitive. Let's try the following:

```
Object readLock = new Integer(0);
Object writeLock = new Integer(0);
int readers = 0;

public void read()
{
   synchronized( readLock )
   {
      if( readers == 0 )
      {
         synchronized( writeLock )
         {
```

Looks good on first inspection, but it doesn't work. As soon as we leave the {} scope the writeLock is released. Synchronized is not really a lock. Rather, it is what is called a "critical section." Critical sections are, in my opinion, a flawed concept.

All thread problems stem from critical resources, which is what locks provide for. Critical sections provide for critical blocks of code rather than their underlying resources. Critical sections prevent more than one thread at a time from entering a section of code, but code is not what you need to protect. Instead, you need a way to protect the resource. The resource, if changed by more than one thread at a time, will blow up (or yield corrupt data if read while being changed).

And the problem runs even deeper, since Java's synchronized is not even a pure critical section. Instead, it is this strange beast that allows critical sections to link to one another. This approach lets you protect multiple sections of code by creating mutual exclusion between them, but it still does not allow you to protect resources. Because of this flaw, if we say a block code is "critical," then we allow only one reader or writer to access the code. The reader or writer, in turn, accesses the resource at one time. You can see the problem. This scenario still lets only one reader at a time access the resource.

Using critical sections for protection, the code would look like this:

```
public void read()
{
   synchronized( m_lock )
   {
      // do reading
   }
}

public void write()
{
   getDataToWrite();
   synchronized( m_lock )
   {
      // do writing
   }
}
```

This works, in that it will not crash the application. But it does not allow multiple readers to read at the same time. In fact, if you put this on your multi-processor mega-powerful server it would not run any faster than on the single processor low-powered computer, because even though we have multiple threads, only one is allowed to do anything at one time. The fact is, this technique effectively disables multithreading. While one thread waits for the disk to access, all other threads are twiddling their thumbs waiting for this slow thread to finish reading.

Look over the CriticalResourceUsingSynchronized.java file. The file shows a multi-threaded solution to our multi-threading problem using synchronized and spin locks.

[View the code ...]

Previous Page Return to beginning of article Next Page

CriticalResourceUsingSynchronized.java

```
// This solves the readers/writers problem only using synchronized

public class CriticalResourceUsingSynchronized
{
     private     Integer     m_readLock = new Integer(0);
     private     int     m_reading =     0;

     public int read()
     {
          int readData = 0;

          // lock readers
          synchronized( m_readLock )
          {
               // if we are the first reader lock writers

               // this is what we wanted to do from the pseudo code:
               //     if( m_reading == 0 )
               //           lock writers
               //
               //  But there is no way to do this with synchronized so
               //  the writer has to do a spin lock on m_reading

               // increment semaphore
               m_reading++;

          }     // unlock readers

          // do read  (may take significant time...)

          // lock readers
          synchronized( m_readLock )
          {
               // decrement semaphore
               m_reading--;

               // from the pseudo code we wanted to do this:
               //     if( m_reading == 0 )
               //           unlock writers
               //
               // but synchonized doesn't cut it here so the writer is
               // polling m_reading (i.e. spin lock) to see when it goes to 0

          }// unlock readers

          // return the read data
          return readData;
     }

     public void write( int x )
     {
          // we want to
```

```
          //
          // lock writers
          // write
          // unlock writers
          //
          // but since we are only using synchronized we have to spin lock;
          // because synchronized is not a lock; it is this weird linked
          // critical section thing.

          // spin lock waiting for m_readers to go to 0
          boolean succeeded = false;
          while( !succeeded )
          {
               synchronized( m_readLock )
               {
                    if( m_reading == 0 )
                    {
                         // we did it! m_readers is 0, and no readers can get in
                         // because we have locked m_readLock

                         // do write (may take significant time...)

                         // we can break out of this crazy spin lock!
                         succeeded = true;
                    }
               }

                if( succeeded == false )
               {
                    // we failed; there are still readers so let them compute
                    // so yeild the rest of our time slice so we don't check
                    // again before any other thread has a chance to do anything
                    Thread.currentThread().yield();
               }
          }
     }
}
```

Lock.java

```
// Lock.java
//
// This class implements a boolean lock object in java
//

class Lock extends Object
{
     private boolean m_bLocked = false;

     public synchronized void lock()
     {
          // if some other thread locked this object then we need to wait
          // until they release the lock
          if( m_bLocked )
          {
               do
               {
                    try
                    {
                         // this releases the synchronized that we are in
                         // then waits for a notify to be called in this object
                         // then does a synchronized again before continuing
                         wait();
                    }
                    catch( InterruptedException e )
                    {
                         e.printStackTrace();
                    }
                    catch( Exception e )
                    {
                         e.printStackTrace();
                    }
               } while( m_bLocked );     // we can't leave until we got the lock,
which
                                              // we may not have got if an exception
occured
          }

          m_bLocked = true;
     }

     public synchronized boolean lock( long milliSeconds )
     {
          if( m_bLocked )
          {
               try
               {
                    wait( milliSeconds );
               }
               catch( InterruptedException e )
               {
                    e.printStackTrace();
```

```
                }

                if( m_bLocked )
                {
                    return false;
                }
            }

            m_bLocked = true;
            return true;
        }

        public synchronized boolean lock( long milliSeconds, int nanoSeconds )
        {
            if( m_bLocked )
            {
                try
                {
                    wait( milliSeconds, nanoSeconds );
                }
                catch( InterruptedException e )
                {
                    e.printStackTrace();
                }

                if( m_bLocked )
                {
                    return false;
                }
            }

            m_bLocked = true;
            return true;
        }

        public synchronized void releaseLock()
        {
            if( m_bLocked )
            {
                m_bLocked = false;
                notify();
            }
        }

        public synchronized boolean isLocked()
        {
            return m_bLocked;
        }
}
```

CriticalResourceUsingSynchronizedAndLocks.java

```
// This solves the readers/writers problem only using synchronized and the Lock class
import Lock;

public class CriticalResourceUsingSynchronizedAndLocks
{
     private     Integer     m_readLock = new Integer(0);
     private     int     m_reading =     0;
     private Lock m_writeLock = new Lock();

     public int read()
     {
          int readData = 0;

          // lock readers
          synchronized( m_readLock )
          {
               // if we are the first reader lock writers

               if( m_reading == 0 )
                    // lock the writers
                    m_writeLock.lock();

               // increment semephore
               m_reading++;

          }     // unlock readers

          // do read  (may take significant time...)

          // lock readers
          synchronized( m_readLock )
          {
               // decrement semaphore
               m_reading--;

               if( m_reading == 0 )
                    // release the writers
                    m_writeLock.releaseLock();

          }// unlock readers

          // return read value
          return readData;
     }

     public void write( int x )
     {
          // lock writers
          m_writeLock.lock();

          // do writing

          // release writers
```

```
          m_writeLock.releaseLock();
     }
}
```

CriticalResourceUsingLocks.java

```
// This solves the readers/writers problem only the Lock class
import Lock;

public class CriticalResourceUsingLocks
{
     private     Lock m_readLock = new Lock();
     private     int     m_reading =     0;
     private Lock m_writeLock = new Lock();

     public int read()
     {
          int readData = 0;

          // lock readers
          m_readLock.lock();

          // if we are the first reader lock writers
          if( m_reading == 0 )
               // lock the writers
               m_writeLock.lock();

          // increment semephore
          m_reading++;

          // unlock readers
          m_readLock.releaseLock();

          // do read  (may take significant time...)

          // lock readers
          m_readLock.lock();

          // decrement semaphore
          m_reading--;

          if( m_reading == 0 )
               // release the writers
               m_writeLock.releaseLock();

          // unlock readers
          m_readLock.releaseLock();

          // return read data
          return readData;
     }
```

```
    public void write( int x )
    {
        // lock writers
        m_writeLock.lock();

        // do writing

        // release writers
        m_writeLock.releaseLock();
    }
}
```

Thread Synchronization in Java

How to Get True Multithreading in Java

by Christopher McGee

Object Monitors: wait() and notify()

Unfortunately, spin locks are not very efficient because they use processor cycles every time they poll the watched variable for change in state. To address this, Java uses a concept called a "monitor." Every object has a monitor, and you can tell this monitor to wake your thread when the object changes. This is done using two methods on Object - wait() and notify().

When you call wait() on an object, the current thread will sleep until notify() is called on that same object. But there is a caveat - you have to call synchronized on the object before you can call wait() or notify() on it.

Code that uses these methods must be in the following form:

```
Object o = new Integer(0);

synchronized( o )
{
   o.wait();
}

synchronized( o )
{
   o.notify();
}
```

Now this looks a bit odd, because the o.wait() call does not finish until a notify() is called; but in order to call notify() you have to be synchronized on o. Since the o.wait() call is already synchronized on o, won't the o.notify() not be able to synchronize until the o.wait() exits? It looks like o.wait() is waiting on o.notify(), but o.notify() is really waiting on the synchronized(o) of the wait() to exit. (In other words, the two calls appear to be deadlocking each other.)

Java's developers saw this problem, too, so they made wait() close its synchronized(o) before it starts waiting; then, when it gets the notify() signal, it calls synchronized(o) again. So while you are sitting in the o.wait() you are not holding on to the synchronized(o).

This feature is the hidden key to solving the multithreading problem - getting real locks in Java.

Next Page

Previous Page Return to beginning of article

Thread Synchronization in Java

How to Get True Multithreading in Java

by Christopher McGee

Solutions: Implementing Real Locks in Java

Using the wait() and notify() methods makes it possible to implement real locks in Java.

Look at this Boolean Lock class that is implemented using wait()/notify():

[View the code ...]

We can solve the readers/writers problem using this Lock class, without the need for spin locks. Here is the modified source:

[View the code ...]

Notice how much simpler write() is now. In fact, with the Lock class we do not really need synchronized any more at all. Here is the readers/writers problem solved using only the Lock class:

[View the code ...]

Notice, now, how similar this code is to the pseudo code we introduced above.

Conclusion

Thread problems stem from multiple threads trying to access a resource at the same time. As we have seen, the way to fix this problem is to protect the critical resource using locks or critical sections. In Java, you have both methods - synchronized for critical sections, and wait()/notify() for locks.

Make certain that only one thread is allowed to change your critical resource at a time, and make sure that the read-only methods that access the resource do not read from it while it is being changed. Also be aware that some 3rd party resources may not work correctly with multiple threads (even just reading) because of the way it their resources are coded.

I hope this article has given you some useful ammunition to help solve thread problems with your Java applications. Good luck and happy bug hunting!

Previous Page Return to beginning of article Next Page