Deadlock Detection with new Locks

www.javaspecialists.eu/archive/Issue130.html

by Dr. Heinz M. Kabutz

Abstract:
Java level monitor deadlocks used to be hard to find. Then along came JDK 1.4 and showed them in CTRL+Break. In JDK 1.5, we saw the addition of the ThreadMXBean, which made it possible to continually monitor an application for deadlocks. However, the limitation was that the ThreadMXBean only worked for synchronized blocks, and not the new java.util.concurrent mechanisms. In this newsletter, we take a fresh look at the deadlock detector and show what needs to change to make it work under JDK 1.6. Also, we have a look at what asynchronous exceptions are, and how you can post them to another thread.

Welcome to the 130th edition of

The Java(tm) Specialists' Newsletter, now sent to 112 countries with the recent addition of Gibraltar. We had a fantastic meeting in Cologne earlier this month, organised by the JUG Cologne. Just a pity that Germany lost their soccer match. That was a bit of a let-down to the evening.

In October 2006 we will be moving to Europe to be closer to our customer base. This will open up possibilities where I can help companies in Europe with short 1-2 day consulting jobs, code reviews, performance reviews, short seminars, etc. Moving a whole family between continents is always challenging (2 years of planning) and I know that I will find it hardest to adjust to the new culture in Greece. Fortunately Helene is half-Greek so I have had some practice :)

NEW: Please see our new "Extreme Java" course, combining concurrency, a little bit of performance and Java 8. Extreme Java - Concurrency & Performance for Java 8.

Deadlock Detection with new Locks

One of the challenges of doing multi-threaded applications is that you need to keep an eye on both correctness and liveness. If you do not use locks, then it might happen that your objects suddenly get an invalid state (i.e. correctness is affected). It is not always easy to see this, since the problems are rarely reproduceable.

The next step is to sprinkle locks all over the place, which can then lead to liveness problems. In a recent project, we needed to find some deadlock situations in Swing code. We ended up posting a Swing event every few seconds and then raising an alarm if it was not processed within a set amount of time. This helped us root out the problems quickly.

In my newsletter # 93, I presented a class that automatically detected thread deadlocks. Bruce Eckel pointed out that this mechanism only worked for *monitor* locks (i.e. synchronized) and not for the java.util.concurrent *owned* locks.

In Java 1.6 (beta 2), the deadlocks are found for both the *monitor* and *owned* locks. However, the deadlock detector needs to be modified in order to pick up the *owned* locks that have deadlocked.

In Java 1.5, the ThreadMXBean had only the findMonitorDeadlockedThreads() method, which found deadlocks caused by the `synchronized` keyword. We now also have the findDeadlockedThreads() method that is enabled when the Java virtual machine supports monitoring of ownable synchronizer usage. (i.e. the new locks or related constructs)

New Deadlock Detector

This ThreadDeadlockDetector periodically checks whether there are threads that appear to be deadlocked. On some architectures, this could be quite resource-intensive, so we make the check interval configurable.

In Java 5, it only picked up monitor locks, not owned locks. Since Java 6, we can now see owned locks deadlocking, and a combination of owned and monitor locks.

The deadlock detection algorithm is not that accurate, in that it has trouble figuring out whether we have a deadlock when we use the tryLock(time) method. This method tries to lock for a set delay, and if it cannot manage, then it returns false. We can use this method to avoid getting deadlocks. However, the ThreadMXBean does still appear to be detecting these as real deadlocks.

The fallout is that whereas before, once you had reported on a thread being deadlocked, that thread would remain in that state forever. With the new Java 6 locks, this is not necessarily the case anymore.

Without further ado, here is the code:

```
import java.lang.management.*;
import java.util.*;
import java.util.concurrent.CopyOnWriteArraySet;
public class ThreadDeadlockDetector {
 private final Timer threadCheck =
 new Timer("ThreadDeadlockDetector", true);
 private final ThreadMXBean mbean =
 ManagementFactory.getThreadMXBean();
 private final Collection<Listener> listeners =
 new CopyOnWriteArraySet<Listener>();
 /**
 * The number of milliseconds between checking for deadlocks.
 * It may be expensive to check for deadlocks, and it is not
 * critical to know so quickly.
 */
 private static final int DEFAULT_DEADLOCK_CHECK_PERIOD =
10000;
 public ThreadDeadlockDetector() {
 this(DEFAULT_DEADLOCK_CHECK_PERIOD);
 }
 public ThreadDeadlockDetector(int deadlockCheckPeriod) {
 threadCheck.schedule(new TimerTask() {
 public void run() {
 checkForDeadlocks();
 }
 }, 10, deadlockCheckPeriod);
 }
 private void checkForDeadlocks() {
 long[] ids = findDeadlockedThreads();
 if (ids != null && ids.length > 0) {
 Thread[] threads = new Thread[ids.length];
 for (int i = 0; i < threads.length; i++) {
 threads[i] = findMatchingThread(
 threads[i] = findMatchingThread(
 mbean.getThreadInfo(ids[i]));
 }
 fireDeadlockDetected(threads);
 }
 }
 private long[] findDeadlockedThreads() {
 // JDK 1.5 only supports the findMonitorDeadlockedThreads()
 // method, so you need to comment out the following three
lines
 if (mbean.isSynchronizerUsageSupported())
 return mbean.findDeadlockedThreads();
 else
 return mbean.findMonitorDeadlockedThreads();
 }
 private void fireDeadlockDetected(Thread[] threads) {
 for (Listener l : listeners) {
 l.deadlockDetected(threads);
 }
 }
 private Thread findMatchingThread(ThreadInfo inf) {
 for (Thread thread : Thread.getAllStackTraces().keySet()) {
 if (thread.getId() == inf.getThreadId()) {
 return thread;
 }
 }
 throw new IllegalStateException("Deadlocked Thread not
found");
 }
 public boolean addListener(Listener l) {
 return listeners.add(l);
 }
 public boolean removeListener(Listener l) {
 return listeners.remove(l);
 }
 /**
 * This is called whenever a problem with threads is detected.
 */
 public interface Listener {
 void deadlockDetected(Thread[] deadlockedThreads);
 }
}
```

I also wrote a DefaultDeadlockListener that prints out an error message and shows the stack traces. If you get a deadlock, you probably want to rather sent a high alert to the support team.

```
public class DefaultDeadlockListener implements
 ThreadDeadlockDetector.Listener {
 public void deadlockDetected(Thread[] threads) {
 System.err.println("Deadlocked Threads:");
 System.err.println("-------------------");
 for (Thread thread : threads) {
 System.err.println(thread);
 for (StackTraceElement ste : thread.getStackTrace())
{
 System.err.println("\t" + ste);
 }
 }
 }
}
```

To test this, I defined an interface that I could implement with several different deadlock conditions.

```
public interface DeadlockingCode
{
 void f();
 void g();
}
```

The most basic error is to acquire two locks in a different sequence, such as in DeadlockingCodeSynchronized below. Our JDK 1.5 deadlock detector would have picked this one up as well:

```
public class DeadlockingCodeSynchronized implements DeadlockingCode
{
 private final Object lock = new Object();
 public synchronized void f() {
 synchronized(lock) {
 // do something
 }
 }
 public void g() {
 synchronized(lock) {
 f();
 }
 }
}
```

The output from my default deadlock listener is:

```
Deadlocked Threads:
-------------------
Thread[DeadlockingCodeSynchronized g(),5,main]

DeadlockingCodeSynchronized.f(DeadlockingCodeSynchronized.java:4)

DeadlockingCodeSynchronized.g(DeadlockingCodeSynchronized.java:9)
 DeadlockedThreadsTest$3.run(DeadlockedThreadsTest.java:42)
Thread[DeadlockingCodeSynchronized f(),5,main]

DeadlockingCodeSynchronized.f(DeadlockingCodeSynchronized.java:4)
 DeadlockedThreadsTest$2.run(DeadlockedThreadsTest.java:34)
```

The more difficult deadlock is when it is on an owned lock, such as:

```java
import java.util.concurrent.locks.*;
public class DeadlockingCodeNewLocksBasic implements DeadlockingCode
{
 private final Lock lock1 = new ReentrantLock();
 private final Lock lock2 = new ReentrantLock();
 public void f() {
 lock1.lock();
 try {
 lock2.lock();
 try {
 // do something
 } finally {
 lock2.unlock();
 }
 } finally {
 lock1.unlock();
 }
 }
 public void g() {
 lock2.lock();
 try {
 f();
 } finally {
 lock2.unlock();
 }
 }
}
```

The new mechanism also picks up mixed deadlocks, where the one side has acquired a monitor and the other an owned lock, and now a deadlock has occurred.

Recovering From Deadlocks?

As before, there is little that can be done when you encounter a deadlock situation. If you even remotely suspect that you might cause a deadlock, rather use the tryLock() method together with a rollback if unsuccessful.

It is possible to `stop()` a thread that is deadlocked on an *owned* lock, but I found situations in which it retained the locks that it had already obtained. Also, you should never use `stop()`, since it will stop your thread at any place, even in critical code. Basically, the only thing that you can do is to notify the administrator that a deadlock has occurred, let him know the threads which are deadlocked, and exit the program.

The precursor of Java was called Oak. About 4 years ago, I wrote a newsletter entitled Once upon an Oak, where I showed how Java was influenced by Oak. There were several surprises. For example, there was no **`private`** keyword.

One of the scary features of Oak was Asynchronous Exceptions. Here is an excerpt from the Oak Manual 0.2:

Margin comment: The default will probably be changed to *not* allow asynchronous exceptions except in explicitely *unprotected* sections of code.

*Heinz: Note that instead of making everything **protected** by default, they made everything **unprotected**!*

Asynchronous Exceptions

Generally, exceptions are synchronous - they are thrown by code executed sequentially by an Oak program. However, in programs that have multiple threads of execution, one thread can throw an exception (using Thread's `postException()` instance method) to another thread. The second thread can't predict exactly when it will be thrown an exception, so the exception is *asynchronous*.

By default, asynchronous exceptions can happen at any time. To prevent asynchronous exceptions from occuring in a critical section of code, you can mark the code with the **`protect`** keyword, as shown below:

```
 protect {
 /* critical section goes here
*/
 }
```

To allow asynchronous exceptions to occur in an otherwise protected section of code, use the **`unprotect`** keyword, as follows:

```
 unprotect {
 /* code that can afford asynchronous exceptions
*/
 }
```

In Java, the Thread.postException() method is called Thread.stop(Throwable). You can send any exception to another thread, provided that the SecurityManager allows you to. Here is some rather confusing code:

```
 public class ConfusingCode {
 public static long fibonacci(int n) {
 if (n < 2) return n;
 return fibonacci(n - 1) + fibonacci(n - 2);
 }
 public static void main(String[] args)
 throws InterruptedException {
 Thread fibThread = new Thread() {
 public void run() {
 try {
 for (int i = 37; i < 90; i++) {
 System.out.println("fib(" + i + ") = " +
fibonacci(i));
 }
 } catch (NullPointerException ex) {
 ex.printStackTrace();
 }
 }
 };
 fibThread.start();
 Thread.sleep(10000);
 fibThread.stop(new NullPointerException("whoops"));
 }
 }
```

When I run this code, I get the following output on my machine:

```
 fib(37) = 24157817
 fib(38) = 39088169
 fib(39) = 63245986
 fib(40) = 102334155
 java.lang.NullPointerException: whoops
 at
ConfusingCode.main(ConfusingCode.java:23)
```

As you can see, the line number does point to where `Thread.stop()` is being called. However, you could have some rather sinister fun if you were to start throwing NullPointerExceptions asynchronously at working code, especially if you subclasses it and changed the stack trace.

The problem with asynchronous exceptions in Java is that we do not have the **`protect`** keyword anymore. So therefore, critical sections can be stopped half-way through, causing serious errors in your application. It even appeared that the killed threads sometimes held onto locks.

JConsole

I have to mention JConsole, otherwise I will be flooded with emails about it :) You can also detect deadlocks with JConsole. Furthermore, you can attach JConsole onto remote processes. Since Java 1.6, you also do not need to start your program with any special system properties in order to see it locally.

However, what I like about my approach is that it automatically checks the deadlocks for you.

That's all for now - I hope that you can start using this mechanism in your code. For JDK 1.5, comment out the two lines that I marked in the ThreadDeadlockDetector class.

Kind regards from a chilly South Africa,

Heinz

Performance Articles Related Java Course