ABOUT PIERRE HUGUES CHARBONNEAU

Pierre-Hugues Charbonneau (nickname P-H) is working for CGI Inc. Canada for the last 10 years as a senior IT consultant. His primary area of expertise is Java EE, middleware & JVM technologies. He is a specialist in production system troubleshooting, root cause analysis, middleware, JVM tuning, scalability and capacity improvement; including internal processes improvement for IT support teams. P-H is the principal author at Java EE Support Patterns.

Java deadlock troubleshooting and resolution

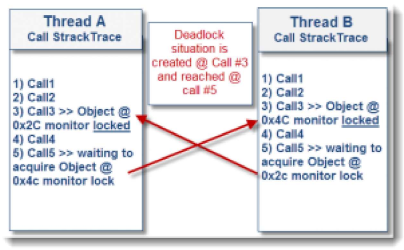
Posted by: Pierre Hugues Charbonneau in Core Java November 1st, 2012 0 22 views (0 rating, 0 votes)

One of the great things about JavaOne annual conferences is the presentation of several technical and troubleshooting labs presented by subject matter experts. One of these labs did especially capture my attention this year: "HOL6500 – Finding And Solving Java Deadlocks ", presented by Java Champion Heinz Kabutz . This is one of the best presentations I have seen on this subject. I recommend that you download, run and study the labs yourself.

This article will revisit this classic thread problem and summarize the key troubleshooting and resolution techniques presented. I will also expand the subject based on my own multi-threading troubleshooting experience.

Java deadlock: what is it?

A true Java deadlock can essentially be described as a situation where two or more threads are blocked forever, waiting for each other. This situation is very different from other more commons "day-to-day" thread problem patterns such as lock contention & thread races, threads waiting on blocking IO calls etc. Such **lock-ordering** deadlock situation can be visualized as per below:

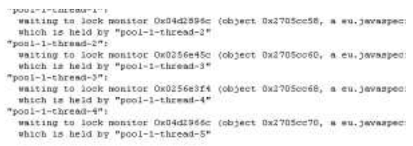

Thread A
Call StrackTrace
1) Call1
2) Call2
3) Call3 >> Object @ 0x2C monitor locked
4) Call4
5) Call5 >> waiting to acquire Object @ 0x4c monitor lock
Deadlock situation is created @ Call #3 and reached @ call #5
Thread B
Call StrackTrace
1) Call1
2) Call2
3) Call3 >> Object @ 0x4C monitor locked
4) Call4
5) Call5 >> waiting to acquire Object @ 0x2c monitor lock

In the above visual example, the attempt by Thread A & Thread B to acquire 2 locks in different orders is fatal. Once threads reached the deadlocked state, they can never recover, forcing you to restart the affected JVM process.

Heinz also describes another type of deadlock: **resource deadlock**. This is by far the most common thread problem pattern I have seen in my experience with Java EE enterprise system troubleshooting. A resource deadlock is essentially a scenario where one or multiple threads are waiting to acquire a resource which will never be available such as JDBC Pool depletions.

Lock-ordering deadlocks

You should know by now that I am a big fan of JVM thread dump analysis; crucial skill to acquire for individuals either involved in Java/Java



```
"pool-1-thread-1":
  waiting to lock monitor 0x04d2596c (object 0x2705cc58, a eu.javaspec
  which is held by "pool-1-thread-2"
"pool-1-thread-2":
  waiting to lock monitor 0x0256e45c (object 0x2705cc60, a eu.javaspec
  which is held by "pool-1-thread-3"
"pool-1-thread-3":
  waiting to lock monitor 0x0256e3f4 (object 0x2705cc68, a eu.javaspec
  which is held by "pool-1-thread-4"
"pool-1-thread-4":
  waiting to lock monitor 0x04d2966c (object 0x2705cc70, a eu.javaspec
  which is held by "pool-1-thread-5"
```

Now this is the easy part…The core of the root cause analysis effort is to understand why such threads are involved in a deadlock situation at the first place. Lock-ordering deadlocks could be triggered from your application code but unless you are involved in high concurrency programming, chances are that the culprit code is a third part API or framework that you are using or the actual Java EE container itself, when applicable.

Now let's review below the lock-ordering deadlock resolution strategies presented by Heinz:

# Deadlock resolution by global ordering (see lab1 solution)

- Essentially involves the definition of a global ordering for the locks that would always prevent deadlock (please see lab1 solution)

# Deadlock resolution by TryLock (see lab2 solution)

- Lock the first lock
- Then try to lock the second lock
- If you can lock it, you are good to go
- If you cannot, wait and try again

The above strategy can be implemented using Java Lock & ReantrantLock which also gives you also flexibility to setup a wait timeout in order to prevent thread starvation in the event the first lock is acquired for too long.

```java
public interface Lock {

void lock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long timeout, TimeUnit unit)

throws InterruptedException;

void unlock();

Condition newCondition();

}
```

If you look at the JBoss AS7 implementation, you will notice that Lock & ReantrantLock are widely used from core implementation layers such as:

- Deployment service
- EJB3 implementation (widely used)
- Clustering and session management
- Internal cache & data structures (LRU, ConcurrentReferenceHashMap…)

Now and as per Heinz's point, the deadlock resolution strategy #2 can be quite efficient but proper care is also required such as releasing all held lock via a finally{} block otherwise you can transform your deadlock scenario into a livelock.

Resource deadlocks

Now let's move to resource deadlock scenarios. I'm glad that Heinz's lab #3 covered this since from my experience this is by far the most common "deadlock" scenario that you will see, especially if you are developing and supporting large distributed Java EE production systems.

Now let's get the facts right.

- Resource deadlocks are <u>not</u> true Java-level deadlocks
- The JVM Thread Dump will <u>not</u> magically should you these types of deadlocks. This means more work for you to analyze and understand this problem as a starting point.
- Thread dump analysis can be especially confusing when you are just starting to learn how to read Thread Dump since threads will often show up as RUNNING state vs. BLOCKED state for Java-level deadlocks. For now, it is important to keep in mind that thread state is not that important for this type of problem e.g. RUNNING state != healthy state.
- The analysis approach is very different than Java-level deadlocks. You must take multiple thread dump snapshots and identify thread problem/wait patterns between each snapshot. You will be able to see threads not moving e.g. threads waiting to acquire a resource from

Conclusion

I hope you had the chance to review, run and enjoy the labs from Heinz's presentation as much as I did. Concurrency programming and troubleshooting can be quite challenging but I still recommend that you spend some time trying to understand some of these principles since I'm confident you will face a situation in the near future that will force you to perform this deep dive and acquire those skills.

Reference: Java deadlock troubleshooting and resolution from our JCG partner Pierre-Hugues Charbonneau at the Java EE Support Patterns & Java Tutorial blog.

Tagged with: CONCURRENCY DEADLOCK

(*0 rating, 0 votes*)

You need to be a registered member to rate this. Start the discussion 22 Views Tweet it!

LIKE THIS ARTICLE? READ MORE FROM JAVA CODE GEEKS

Convenient mocking in Mockito with JUnit 5 – the official way
March 29th, 2018

Leave a Reply

Be the First to Comment!

Start the discussion

Subscribe

KNOWLEDGE BASE

- Courses
- Examples
- Minibooks
- Resources
- Tutorials

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- System Code Geeks
- Web Code Geeks

HALL OF FAME

- "Android Full Application Tutorial" series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused on crea ultimate Java to Java developers resource center; targeted at the technical a technical team lead (senior developer), project manager and junior develope JCGs serve the Java, SOA, Agile and Telecom communities with daily news w domain experts, articles, tutorials, reviews, announcements, code snippets a source projects.

DISCLAIMER

Key Skills for Microservices
DOWNLOAD OUR EBOOK
APPDYNAMI