The Law of the Overstocked Haberdashery

www.javaspecialists.eu/archive/Issue149.html

by Dr. Heinz M. Kabutz

Abstract:
Learn how to write correct concurrent code by understanding the Secrets of Concurrency. This is the third part of a series of laws that help explain how we should be writing concurrent code in Java. In this section, we look at why we should avoid creating unnecessary threads, even if they are not doing anything.

Welcome to the 149th issue of

The Java(tm) Specialists' Newsletter. We are continuing with the Secrets of Concurrency. Thank you for all your feedback. Enjoy!

The Law of the Overstocked Haberdashery

In this newsletter, I am continuing in looking at the laws that will help us write correct concurrent code in Java. Just to remind you, here they are again. In this newsletter, we will look at The Law of the Overstocked Haberdashery.

The Law of the Overstocked Haberdashery

Having too many threads is bad for your application. Performance will degrade and debugging will become difficult.

Before we start with the law, I need to define the word haberdashery: A shop selling sewing wares, e.g. threads and needles.

One of the services I offer companies is to help them debug complex Java problems. This can take the form of code reviews or performance tuning. When I still lived in South Africa, I was contacted by a company that wanted an experienced Java programmer to fix some minor problems in their application. During our telephone conversation, I asked them to describe their system and immediately realised that they were creating a huge number of inactive threads.

I visited the director to discuss the scope of the work. Unfortunately for everyone involved, I made a fatal mistake right at the beginning of the engagement. I completely underquoted the hourly rate. The hourly rate should have been approximately four times what I quoted. I was immediately classed as a POJP (plain old Java programmer), rather than as a world-class Java *consultant*. Since I was now a POJP, the rate I had quoted sounded too high to the director and he tried to squeeze me even on that.

During the first discussions, I again raised the problem of them creating far too many threads. The director, who had now classed me as technically way below him, insisted that there would be no problem creating hundreds of thousands of threads, as long as they were not all active at the same time.

The technical manager then showed me the system and told me that they had a memory problem, or so he thought. He had personally refactored the code, but now it did not work at all anymore. I would need to go back to the original code and reduce the memory footprint. For example, I should change Vector to ArrayList and other such fantastic optimizations.

With no unit tests, I spent a few very frustrating weeks trying to follow the instructions of my employer. Of course, none of the things I did made any difference, because the problem lay with too many inactive threads being created. The haberdashery was overstocked with too many threads.

Every few days, the director would walk in and ask if I had fixed the problem yet and my answer would be "No, but I think you are creating too many threads." He would then tell me patiently that the number of threads was theoretically unlimited, you could have hundreds of thousands of threads without it causing any problems for your system.

Eventually, after being thorougly fed up, I wrote a simple Java program that kept on creating threads, but kept them inactive by just putting them into the WAITING state. I ran it with him watching and it showed clearly that on their machine, it was limited to a few thousand.

The solution to the problem was simple and was something that I had mentioned in our very first meeting. All they had to do was to change from standard blocking IO to non-blocking New IO. Had I charged a high consulting rate, chances are that they would have listened to me in the first meeting and fixed the architecture. Instead of hiring me for three weeks, they would have had to pay for two hours of consulting. It would have saved time, money and frustration. It reminded me of the secret of consulting: The more they pay you, the more they love you.

Here is a small piece of code that you can run to find out how many *inactive* threads you can start on your JVM:

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.CountDownLatch;
public class ThreadCreationTest {
 public static void main(String[] args)
 throws InterruptedException {
 final AtomicInteger threads_created = new
AtomicInteger(0);
 while (true) {
 final CountDownLatch latch = new CountDownLatch(1);
 new Thread() {
 { start(); }
 public void run() {
 latch.countDown();
 synchronized (this) {
 System.out.println("threads created: " +
 threads_created.incrementAndGet());
 try {
 wait();
 } catch (InterruptedException e) {
 Thread.currentThread().interrupt();
 }
 }
 }
 };
 latch.await();
 }
 }
}
```

Depending on various factors (operating system, JVM version, Hotspot compiler version, JVM parameters, etc.), we will get different results. In my case, I managed to cause the actual JVM to crash:

```
Exception in "main" OutOfMemoryError: unable to create native
thread
 at java.lang.Thread.start0(Native Method)
 at java.lang.Thread.start(Thread.java:597)
 at ThreadCreationTest$1.<init>(ThreadCreationTest.java:8)
 at ThreadCreationTest.main(ThreadCreationTest.java:7)
#
# An unexpected error has been detected by Java Runtime
Environment:
#
# Internal Error (455843455054494F4E530E4350500134) #
# Java VM: Java HotSpot(TM) Client VM (1.6.0_01-b06 mixed mode)
# An error report file with more information is saved as hs_err.log
#
Aborted (core dumped)
```

We can increase our maximum number of inactive threads by decreasing the stack size. One of the limiting factors is the thread stack size, which stores local variables, parameters and the call stack. It typically does not need to be too large, unless you have very deep recursion. However, this does not necessarily solve the problem, because usually there is a problem in the architecture to start with.

You can reduce the stack size with the -Xss<size_of_stack> JVM parameter. For example, with `java ?Xss48k ThreadCreationTest`, I was able to create 32284 threads. However, when I tried to create the 32285th thread, the JVM hung up so badly that I could not stop it, except with `kill -9`.

How many Threads is Healthy?

Creating a new thread should improve the performance of your system. In the same way that you should not use exceptions to control program flow, you should not use threads in that way either.

The number of *active* threads is related to the number of physical cores that you have in your system and on the pipeline length of your CPU. Typically running 4 active threads per CPU core is acceptable. Some architectures allow you to run up to 20 active threads per core, but they are the exception.

The number of *inactive* threads that you should have is architecture specific. However, having 9000 inactive threads on one core is far too many. What would happen if they all became active at once? Also, they consume a lot of memory. This you could manage by decreasing the stack size. However, if you do reach the limit, the JVM can crash without warning.

Traffic Calming

Instead of creating new threads directly, it is advisable to use thread pools. Thread pools are now part of the JDK and can be created either as fixed or expanding pools. One of the benefits of thread pools is that you can limit the maximum number of active threads that will be used. We then submit our jobs to the pool and one of the available threads will execute it. By limiting the number of threads, we prevent our system being swamped under strain.

I have heard from various quarters that creating a thread is not as expensive as it used to be and that we should not use thread pools just to avoid the cost of creating a thread. My measurements tell a different tale. I would like to see evidence to prove the assertion that threads are cheap to create, before making a definite decision. Perhaps this is something that can be tuned with JVM parameters.

Here is a short test that creates a few thousand threads. We create a semaphore of size 10, which we will use to stop the system creating too many inactive threads:

```
import java.util.concurrent.*;
public class ThreadConstructTest {
 private static final int NUMBER_OF_THREADS = 100 * 1000;
 private Semaphore semaphore = new Semaphore(10);
 private final Runnable job = new Runnable() {
 public void run() {
 semaphore.release();
 }
 };
 public void noThreadPool() throws InterruptedException {
 for (int i = 0; i < NUMBER_OF_THREADS; i++) {
 semaphore.acquire();
 new Thread(job).start();
 }
 // wait for all jobs to finish
 semaphore.acquire(10);
 semaphore.release(10);
 }
 public void fixedThreadPool() throws InterruptedException
{
 ExecutorService pool = Executors.newFixedThreadPool(12);
 for (int i = 0; i < NUMBER_OF_THREADS; i++) {
 semaphore.acquire();
 pool.submit(job);
 }
 semaphore.acquire(10);
 semaphore.release(10);
 pool.shutdown();
 }
 public static void main(String[] args) throws Exception {
 ThreadConstructTest test = new ThreadConstructTest();
 long time = System.currentTimeMillis();
 test.noThreadPool();
 System.out.println(System.currentTimeMillis() - time);
 time = System.currentTimeMillis();
 test.fixedThreadPool();
 System.out.println(System.currentTimeMillis() - time);
 }
}
```

I will not embarrass my laptop by publishing figures :-) Besides, I am writing this newsletter on battery power whilst sitting on Tersanas Beach, so it is even slower than usual. However, please run the code on your machine and let me know if you get figures that show that creating a new thread is insignificant on your architecture.

To summarise, the lesson of this newsletter is to not create too many threads, even if they are inactive. If you do use thread pools for traffic calming, make sure that you make the number of threads configurable. You might want to one day run your program on a 768 core Azul Systems machine.

Kind regards from Greece

Heinz

Concurrency Articles Related Java Course