The Law of the Blind Spot

www.javaspecialists.eu/archive/Issue150.html

by Dr. Heinz M. Kabutz
Abstract:
In this fourth law of concurrency, we look at the problem with visibility of shared variable updates. Quite often, "clever" code that tries to avoid locking in order to remove contention, makes assumptions that may result in serious errors.

Welcome to the 150th issue of

The Java(tm) Specialists' Newsletter, sent to you from Marathi Beach, Crete. We have now lived here on Crete for almost a whole year. Did you know that in Greece, the children get a 14 week school holiday during summer? I think the long holiday is rooted in agriculture. During harvest time, the whole family used to work. For example, my wife spent some time living in Greece as a teenager and she had to assist in the tobacco harvest. For many Greek families, the summer is the main work time, with tourists coming in droves.

We have had the most incredible time as a family. We went to the beach on most mornings, snorkeling and swimming. I shifted my working hours to the afternoons and nights, giving us an amazing family time. It is hard to describe summer on the Greek Isles to someone who has not experienced it. I am already looking forward to the 14 weeks of school holidays in June 2008 :-)

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.

The Law of the Blind Spot

We are looking at a series of laws to make sense of Java concurrency. Just to remind you, here they are again. In this newsletter, we will look at the Law of the Blind Spot.

The Law of the Blind Spot

It is not always possible to see what other threads (cars) are doing with shared data (road).

Since I moved to Crete, the problem with the blind spot has seldom been a possibility. Almost all our roads on Crete are single-threaded (single-laned). However, in a multi-lane environment, this can cause serious accidents. The problem is with visibility: One thread (your car) cannot see what another thread (someone else's car) is doing on the shared data (the road that you are both travelling on). This typically causes problems with the check-then-act idiom. We **check** in our rear-view and side mirrors to see if there are any cars next to us. We then **act** by moving over into the slow lane. If someone was creeping up behind us, trying to overtake us on the inside, we will hear a lot of tire screeching, hooting and possibly cause a serious accident.

The Java Memory Model allows each thread to keep a local copy of fields. This flexibility allows JVM developers to add optimizations, based on the underlying architecture.

As a developer, you need to be aware that if you do not have adequate access control for your shared data, another thread might never see your change to the data.

This usually happens when we try to reason with logic in order to avoid synchronization. Unless you are synchronizing your data in some way, you can get visibility problems. Crime has become just a perception.

For example, if we look at the following code, MyThread might never end, even if we call `shutdown()`:

```
public class MyThread extends Thread
{
 private boolean running = true;
 public void run() {
 while(running) {
 // do something
 }
 }
 public void shutdown() {
 running = false;
 }
}
```

The difficulty with writing correct code is that you do not necessarily know if your threaded code is correct on all implementations of the JVM through experimentation. You cannot prove the correctness of your code by running it successfully.

There are three ways of preventing the problem of field visibility:

1. Use locks to read and write access. Either the **`synchronized`** or the new Java 5 locks would work. It is important here to lock even if you are reading, otherwise you might see an outdated version of the field from your local cache.
2. Make field **`volatile`**. This will ensure that threads always read the value from the field and do not cache it locally. It is a cheaper option than synchronizing but is more difficult to get right. Typically, you would use **`volatile`** fields where the new value is not dependent on the previous value. Operations such as `++` and `--` are not atomic so volatile will not be the correct solution.
3. The cheapest way to guarantee correct visibility is to make the field **`final`**. This puts a "freeze" on the value when the constructor has completed. Threads may then keep a local copy of the field and are guaranteed to see the correct value. This solution of course does not work if you need to change the field at some time, as in our example above.

We use `volatile` to solve the visibility problem:

```
public class MyThread extends Thread {
 private boolean volatile running =
true;
 public void run() {
 while(running) {
 // do something
 }
 }
 public void shutdown() {
 running = false;
 }
}
```

The most up-to-date value of variable `running` is now visible to all threads reading it. We do not have a problem with each thread having its own *visibility of the value*, the way we have with the blind spot.

To summarise, the lesson of this newsletter is to be careful when reducing synchronization. Your code might run correctly on your JVM, but could fail when running on a production system, during high load.

Kind regards from Greece

Heinz

P.S. This law used to be called the Law of South African Crime, but was renamed to make it more understandable to non-South Africans :-)

Concurrency Articles Related Java Course