The "Double-Checked Locking is Broken" Declaration

Signed by: David Bacon (IBM Research) Joshua Bloch (Javasoft), Jeff Bogda, Cliff Click (Hotspot JVM project), Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell (jGuru) Kelvin Nilsen, Bill Pugh, Emin Gun Sirer

Double-Checked Locking is widely cited and used as an efficient method for implementing lazy initialization in a multithreaded environment.

Unfortunately, it will not work reliably in a platform independent way when implemented in Java, without additional synchronization. When implemented in other languages, such as C++, it depends on the memory model of the processor, the reorderings performed by the compiler and the interaction between the compiler and the synchronization library. Since none of these are specified in a language such as C++, little can be said about the situations in which it will work. Explicit memory barriers can be used to make it work in C++, but these barriers are not available in Java.

To first explain the desired behavior, consider the following code:

```
// Single threaded version
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
        helper = new Helper();
    return helper;
    }
  // other functions and members...
  }
```

If this code was used in a multithreaded context, many things could go wrong. Most obviously, two or more `Helper` objects could be allocated. (We'll bring up other problems later). The fix to this is simply to synchronize the `getHelper()` method:

```
// Correct multithreaded version
class Foo {
  private Helper helper = null;
  public synchronized Helper getHelper() {
    if (helper == null)
        helper = new Helper();
    return helper;
    }
  // other functions and members...
  }
```

The code above performs synchronization every time `getHelper()` is called. The double-checked locking idiom tries to avoid synchronization after the helper is allocated:

```
// Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
    }
  // other functions and members...
  }
```

Unfortunately, that code just does not work in the presence of either optimizing compilers or shared memory multiprocessors.

It doesn't work

There are lots of reasons it doesn't work. The first couple of reasons we'll describe are more obvious. After understanding those, you may be tempted to try to devise a way to "fix" the double-checked locking idiom. Your fixes will not work: there are more subtle reasons why your fix won't work. Understand those reasons, come up with a better fix, and it still won't work, because there are even more subtle reasons.

Lots of very smart people have spent lots of time looking at this. There is *no way* to make it work without requiring each thread that accesses the helper object to perform synchronization.

The first reason it doesn't work

The most obvious reason it doesn't work it that the writes that initialize the `Helper` object and the write to the `helper` field can be done or perceived out of order. Thus, a thread which invokes `getHelper()` could see a non-null reference to a helper object, but see the default values for fields of the helper object, rather than the values set in the constructor.

If the compiler inlines the call to the constructor, then the writes that initialize the object and the write to the `helper` field can be freely reordered if the compiler can prove that the constructor cannot throw an exception or perform synchronization.

Even if the compiler does not reorder those writes, on a multiprocessor the processor or the memory system may reorder those writes, as perceived by a thread running on another processor.

Doug Lea has written a more detailed description of compiler-based reorderings.

A test case showing that it doesn't work

Paul Jakubik found an example of a use of double-checked locking that did not work correctly. A slightly cleaned up version of that code is available here.

When run on a system using the Symantec JIT, it doesn't work. In particular, the Symantec JIT compiles

```
singletons[i].reference = new Singleton();
```

to the following (note that the Symantec JIT using a handle-based object allocation system).

```
0206106A  mov     eax,0F97E78h
0206106F  call    01F6B210          ; allocate space for
                                    ; Singleton, return result in eax
02061074  mov     dword ptr [ebp],eax   ; EBP is &singletons[i].reference
                                    ; store the unconstructed object here.
02061077  mov     ecx,dword ptr [eax]   ; dereference the handle to
                                    ; get the raw pointer
02061079  mov     dword ptr [ecx],100h  ; Next 4 lines are
0206107F  mov     dword ptr [ecx+4],200h  ; Singleton's inlined constructor
02061086  mov     dword ptr [ecx+8],400h
0206108D  mov     dword ptr [ecx+0Ch],0F84030h
```

As you can see, the assignment to `singletons[i].reference` is performed before the constructor for Singleton is called. This is completely legal under the existing Java memory model, and also legal in C and C++ (since neither of them have a memory model).

A fix that doesn't work

Given the explanation above, a number of people have suggested the following code:

```
// (Still) Broken multithreaded version
// "Double-Checked Locking" idiom
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      Helper h;
      synchronized(this) {
        h = helper;
        if (h == null)
```

```
        synchronized (this) {
          h = new Helper();
        } // release inner synchronization lock
      helper = h;
      }
    }
  return helper;
  }
// other functions and members...
}
```

This code puts construction of the Helper object inside an inner synchronized block. The intuitive idea here is that there should be a memory barrier at the point where synchronization is released, and that should prevent the reordering of the initialization of the `Helper` object and the assignment to the field helper.

Unfortunately, that intuition is absolutely wrong. The rules for synchronization don't work that way. The rule for a monitorexit (i.e., releasing synchronization) is that actions before the monitorexit must be performed before the monitor is released. However, there is no rule which says that actions after the monitorexit may not be done before the monitor is released. It is perfectly reasonable and legal for the compiler to move the assignment `helper = h;` inside the synchronized block, in which case we are back where we were previously. Many processors offer instructions that perform this kind of one-way memory barrier. Changing the semantics to require releasing a lock to be a full memory barrier would have performance penalties.

More fixes that don't work

There is something you can do to force the writer to perform a full bidirectional memory barrier. This is gross, inefficient, and is almost guaranteed not to work once the Java Memory Model is revised. Do not use this. In the interests of science, I've put a description of this technique on a separate page. Do not use it.

However, even with a full memory barrier being performed by the thread that initializes the helper object, it still doesn't work.

The problem is that on some systems, the thread which sees a non-null value for the `helper` field also needs to perform memory barriers.

Why? Because processors have their own locally cached copies of memory. On some processors, unless the processor performs a cache coherence instruction (e.g., a memory barrier), reads can be performed out of stale locally cached copies, even if other processors used memory barriers to force their writes into global memory.

I've created a separate web page with a discussion of how this can actually happen on an Alpha processor.

Is it worth the trouble?

For most applications, the cost of simply making the getHelper() method synchronized is not high. You should only consider this kind of detailed optimizations if you know that it is causing a substantial overhead for an application.

Very often, more high level cleverness, such as using the builtin mergesort rather than handling exchange sort (see the SPECJVM DB benchmark) will have much more impact.

Making it work for static singletons

If the singleton you are creating is static (i.e., there will only be one Helper created), as opposed to a property of another object (e.g., there will be one Helper for each Foo object, there is a simple and elegant solution.

Just define the singleton as a static field in a separate class. The semantics of Java guarantee that the field will not be initialized until the field is referenced, and that any thread which accesses the field will see all of the writes resulting from initializing that field.

```
class HelperSingleton {
  static Helper singleton = new Helper();
  }
```

It will work for 32-bit primitive values

Although the double-checked locking idiom cannot be used for references to objects, it can work for 32-bit primitive values (e.g., int's or float's). Note that it does not work for long's or double's, since unsynchronized reads/writes of 64-bit primitives are not guaranteed to be atomic.

```
// Correct Double-Checked Locking for 32-bit primitives
class Foo {
  private int cachedHashCode = 0;
  public int hashCode() {
    int h = cachedHashCode;
    if (h == 0)
    synchronized(this) {
      if (cachedHashCode != 0) return cachedHashCode;
      h = computeHashCode();
      cachedHashCode = h;
      }
    return h;
    }
  // other functions and members...
```

```
}
```

In fact, assuming that the computeHashCode function always returned the same result and had no side effects (i.e., idempotent), you could even get rid of all of the synchronization.

```
// Lazy initialization 32-bit primitives
// Thread-safe if computeHashCode is idempotent
class Foo {
  private int cachedHashCode = 0;
  public int hashCode() {
    int h = cachedHashCode;
    if (h == 0) {
      h = computeHashCode();
      cachedHashCode = h;
      }
    return h;
    }
  // other functions and members...
  }
```

Making it work with explicit memory barriers

It is possible to make the double checked locking pattern work if you have explicit memory barrier instructions. For example, if you are programming in C++, you can use the code from Doug Schmidt et al.'s book:

```
// C++ implementation with explicit memory barriers
// Should work on any platform, including DEC Alphas
// From "Patterns for Concurrent and Distributed Objects",
// by Doug Schmidt
template <class TYPE, class LOCK> TYPE *
Singleton<TYPE, LOCK>::instance (void) {
    // First check
    TYPE* tmp = instance_;
    // Insert the CPU-specific memory barrier instruction
    // to synchronize the cache lines on multi-processor.
    asm ("memoryBarrier");
    if (tmp == 0) {
        // Ensure serialization (guard
        // constructor acquires lock_).
        Guard<LOCK> guard (lock_);
        // Double check.
        tmp = instance_;
        if (tmp == 0) {
            tmp = new TYPE;
            // Insert the CPU-specific memory barrier instruction
```

```
        // to synchronize the cache lines on multi-processor.
        asm ("memoryBarrier");
        instance_ = tmp;
    }
  return tmp;
  }
```

Fixing Double-Checked Locking using Thread Local Storage

Alexander Terekhov (TEREKHOV@de.ibm.com) came up clever suggestion for implementing double checked locking using thread local storage. Each thread keeps a thread local flag to determine whether that thread has done the required synchronization.

```
class Foo {
    /** If perThreadInstance.get() returns a non-null value, this thread
        has done synchronization needed to see initialization
        of helper */
    private final ThreadLocal perThreadInstance = new ThreadLocal();
    private Helper helper = null;
    public Helper getHelper() {
      if (perThreadInstance.get() == null) createHelper();
      return helper;
    }
    private final void createHelper() {
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
      // Any non-null value would do as the argument here
      perThreadInstance.set(perThreadInstance);
    }
  }
```

The performance of this technique depends quite a bit on which JDK implementation you have. In Sun's 1.2 implementation, ThreadLocal's were very slow. They are significantly faster in 1.3, and are expected to be faster still in 1.4. Doug Lea analyzed the performance of some techniques for implementing lazy initialization.

Under the new Java Memory Model

As of JDK5, there is a new Java Memory Model and Thread specification.

Fixing Double-Checked Locking using Volatile

JDK5 and later extends the semantics for volatile so that the system will not

allow a write of a volatile to be reordered with respect to any previous read or write, and a read of a volatile cannot be reordered with respect to any following read or write. See this entry in Jeremy Manson's blog for more details.

With this change, the Double-Checked Locking idiom can be made to work by declaring the helper field to be volatile. This *does not work* under JDK4 and earlier.

```
// Works with acquire/release semantics for volatile
// Broken under current semantics for volatile
 class Foo {
     private volatile Helper helper = null;
     public Helper getHelper() {
       if (helper == null) {
          synchronized(this) {
            if (helper == null)
              helper = new Helper();
          }
       }
       return helper;
    }
  }
```

Double-Checked Locking Immutable Objects

If Helper is an immutable object, such that all of the fields of Helper are final, then double-checked locking will work without having to use volatile fields. The idea is that a reference to an immutable object (such as a String or an Integer) should behave in much the same way as an int or float; reading and writing references to immutable objects are atomic.

Descriptions of double-check idiom

- Reality Check, Douglas C. Schmidt, C++ Report, SIGS, Vol. 8, No. 3, March 1996.
- Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects, Douglas Schmidt and Tim Harrison. *3rd annual Pattern Languages of Program Design conference*, 1996
- Lazy instantiation, Philip Bishop and Nigel Warren, JavaWorld Magazine
- Programming Java threads in the real world, Part 7, Allen Holub, Javaworld Magazine, April 1999.
- *Java 2 Performance and Idiom Guide*, Craig Larman and Rhett Guthrie, p100.
- *Java in Practice: Design Styles and Idioms for Effective Java*, Nigel Warren and Philip Bishop, p142.
- Rule 99, *The Elements of Java Style*, Allan Vermeulen, Scott Ambler, Greg Bumgardner, Eldon Metz, Trvor Misfeldt, Jim Shur, Patrick Thompson, SIGS Reference library
- Global Variables in Java with the Singleton Pattern, Wiebe de Jong, Gamelan