C++ and the Perils of Double-Checked Locking: Part II

Source Code Accompanies This Article. Download It Now.

- cpplock2.txt

In this installment, Scott and Andrei examine the relationship between thread safety and the `volatile` keyword.

August 01, 2004
URL:http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405772

In the first installment of this two-part article, we examined why the Singleton pattern isn't thread safe, and how the Double-Checked Locking Pattern addresses that problem. This month, we look at the role the `volatile` keyword plays in this, and why DCLP may fail on both uni- and multiprocessor architectures.

The desire for specific instruction ordering makes you wonder whether the `volatile` keyword might be of help with multithreading in general and with DCLP in particular. Consequently, we restrict our attention to the semantics of `volatile` in C++ and further restrict our discussion to its impact on DCLP.

Section 1.9 of the C++ Standard (see ISO/IEC 14882:1998(E)) includes this information (emphasis is ours):

The observable behavior of the [C++] abstract machine is its sequence of reads and writes to ***volatile*** *data and calls to library I/O functions.*
Accessing an object designated by a volatile lvalue, modifying an object, calling a library I/O function, or calling a function that does any of those operations are all ***side effects****, which are changes in the state of the execution environment.*

In conjunction with our earlier observations that the Standard guarantees that all side effects will have taken place when sequence points are reached and that a sequence point occurs at the end of each C++ statement, it would seem that all we need to do to ensure correct instruction order is to `volatile`-qualify the appropriate data and sequence our statements carefully. Our earlier analysis shows that `pInstance` needs to be declared `volatile`, and this point is made in the papers on DCLP (see Douglas C. Schmidt et al., "Double-Checked Locking" and Douglas C. Schmidt et al., *Pattern-Oriented Software Architecture,* Volume 2). However, Sherlock Holmes would certainly notice that, to ensure correct instruction order, the `Singleton` object `itself` must also be `volatile`. This is `not` noted in the original DCLP papers and that's an important oversight. To appreciate how declaring `pInstance` alone `volatile` is insufficient, consider Example 7 (Examples 1-6 appeared in Part I of this article).

```
class Singleton {
public:
static Singleton* instance();
   ...
private:
   static Singleton* volatile pInstance; // volatile added
   int x;
   Singleton() : x(5) {}
};
// from the implementation file
Singleton* Singleton::pInstance = 0;
Singleton* Singleton::instance() {
if (pInstance == 0) {
   Lock lock;
   if (pInstance == 0) {
      Singleton*volatile temp = new Singleton; // volatile added
      pInstance = temp;
   }
}
return pInstance;
}
```

Example 7: Declaring pInstance.

After inlining the constructor, the code looks like Example 8. Though `temp` is volatile, `*temp` is not, and that means that `temp->x` isn't, either. Because you now understand that assignments to nonvolatile data may sometimes be reordered, it is easy to see that compilers could reorder `temp->x`'s assignment with regard to the assignment to `pInstance`. If they did, `pInstance` would be assigned before the data it pointed to had been initialized, leading again to the possibility that a different thread would read an uninitialized `x`.

```
if (pInstance == 0) {
   Lock lock;
   if (pInstance == 0) {
      Singleton* volatile temp =
         static_cast<Singleton*>(operator new(sizeof(Singleton)));
      temp->x = 5; // inlined Singleton constructor
      pInstance = temp;
   }
}
```

Example 8: Inlining the constructor in Example 7.

An appealing treatment for this disease would be to `volatile`-qualify `*pInstance` as well as `pInstance` itself, yielding a glorified version of `Singleton` where all pawns are painted `volatile`; see Example 9.

```
class Singleton {
public:
static volatile Singleton* volatile instance();
   ...
private:
   // one more volatile added
   static volatile Singleton* volatile pInstance;
};
// from the implementation file
volatile Singleton* volatile Singleton::pInstance = 0;
volatile Singleton* volatile Singleton::instance() {
   if (pInstance == 0) {
      Lock lock;
      if (pInstance == 0) {
        // one more volatile added
        volatile Singleton* volatile temp =
           new volatile Singleton;
        pInstance = temp;
      }
   }
   return pInstance;
}
```

Example 9: A glorified version of Singleton.

At this point, you might reasonably wonder why `Lock` isn't also declared `volatile`. After all, it's critical that the lock be initialized before you try to write to `pInstance` or `temp`. Well, `Lock` comes from a threading library, so you can assume it either dictates enough restrictions in its specification or embeds enough magic in its implementation to work without needing `volatile`. This is the case with all threading libraries that we know of. In essence, use of entities (objects, functions, and the like) from threading libraries leads to the imposition of "hard sequence points" in a program—sequence points that apply to all threads. For purposes of this article, we assume that such hard sequence points act as firm barriers to instruction reordering during code optimization: Instructions corresponding to source statements preceding use of the library entity in the source code may not be moved after the instructions corresponding to use of the entity, and instructions corresponding to source statements following use of such entities in the source code may not be moved before the instructions corresponding to their use. Real threading libraries impose less draconian restrictions, but the details are not important for purposes of our discussion here.

You might hope that the aforementioned fully `volatile`-qualified code would be guaranteed by the Standard to work correctly in a multithreaded environment, but it may fail for two reasons.

First, the Standard's constraints on observable behavior are only for an abstract machine defined by the Standard, and that abstract machine has no notion of multiple threads of execution. As a result, though the Standard prevents compilers from reordering reads and writes to `volatile` data within a thread, it imposes no constraints at all on such reorderings across threads. At least that's how most compiler implementers interpret things. As a result, in practice, many compilers may generate thread-unsafe code from the aforementioned source. If your multithreaded code works properly with `volatile` and doesn't work without it, then either your C++ implementation carefully implemented `volatile` to work with threads (less likely) or you simply got lucky (more likely). Either case, your code is not portable.

Second, just as `const`-qualified objects don't become `const` until their constructors have run to completion, `volatile`-qualified objects become `volatile` only upon exit from their constructors. In the statement:

```
volatile Singleton* volatile temp =
    new volatile Singleton;
```

the object being created doesn't become `volatile` until the expression:

```
new volatile Singleton;
```

has run to completion, and that means that we're back in a situation where instructions for memory allocation and object initialization may be arbitrarily reordered.

This problem is one we can address, albeit awkwardly. Within the `Singleton` constructor, we use casts to temporarily add "volatileness" to each data member of the `Singleton` object as it is initialized, thus preventing relative movement of the instructions performing the initializations. Example 10 is the `Singleton` constructor written in this way. (To simplify the presentation, we've used an assignment to give `Singleton::x` its first value instead of a member initialization list, as in Example 10. This change has no effect on any of the issues we're addressing here.)

```
Singleton()
{
   static_cast<volatile int&>(x) = 5; // note cast to volatile
}
```

Example 10: Using casts to create the Singleton constructor.

After inlining this function in the version of `Singleton` where `pInstance` is properly `volatile` qualified, we get Example 11. Now the assignment to `x` must precede the assignment to `pInstance`, because both are `volatile`.

```
class Singleton {
public:
   static Singleton* instance();
   ...
private:
   static Singleton* volatile pInstance;
   int x;
   ...
};
Singleton* Singleton::instance()
{
   if (pInstance == 0) {
      Lock lock;
      if (pInstance == 0) {
         Singleton* volatile temp =
         static_cast<Singleton*>(operator new(sizeof(Singleton)));
            static_cast<volatile int&>(temp->x) = 5;
            pInstance = temp;
      }
   }
}
```

Example 11: Inlining a function in Singleton.

Unfortunately, all this does nothing to address the first problem—C++'s abstract machine is single threaded, and C++ compilers may choose to generate thread-unsafe code from source like that just mentioned, anyway. Otherwise, lost optimization opportunities lead to too big an efficiency hit. After all this, we're back to square one. But wait, there's more—more processors.

DCLP on Multiprocessor Machines

Suppose you're on a machine with multiple processors, each of which has its own memory cache, but all of which share a common memory space. Such an architecture needs to define exactly how and when writes performed by one processor propagate to the shared memory and thus become visible to other processors. It is easy to imagine situations where one processor has updated the value of a shared variable in its own cache, but the updated value has not yet been flushed to main memory, much less loaded into the other processors' caches. Such inter-cache inconsistencies in the value of a shared variable is known as the "cache coherency problem."

Suppose processor `A` modifies the memory for shared variable `x` and then later modifies the memory for shared variable `y`. These new values must be flushed to the main memory so that other processors see them. However, it can be more efficient to flush new cache values in increasing address order, so if `y`'s address precedes `x`'s, it is possible that `y`'s new value will be written to main memory before `x`'s is. If that happens, other processors may see `y`'s value change before `x`'s.

Such a possibility is a serious problem for DCLP. Correct `Singleton` initialization requires that the `Singleton` be initialized and that `pInstance` be updated to be non-null and that these operations be seen to occur in this order. If a thread on processor `A` performs step 1 and then step 2, but a thread on processor `B` sees step 2 as having been performed before step 1, the thread on processor `B` may again refer to an uninitialized `Singleton`.

The general solution to cache coherency problems is to use memory barriers: instructions recognized by compilers, linkers, and other optimizing entities that constrain the kinds of reorderings that may be performed on read/writes of shared memory in multiprocessor systems. In the case of DCLP, we need to use memory barriers to ensure that `pInstance` isn't seen to be nonnull until writes to the `Singleton` have been completed. Example 12 is pseudocode that closely follows an example presented by David Bacon et al. (see the "Double-Checked Locking Pattern is Broken"). We show only placeholders for the statements that insert memory barriers because the actual code is platform specific (typically in assembler).

```
Singleton* Singleton::instance () {
   Singleton* tmp = pInstance;
   ... // insert memory barrier
   if (tmp == 0) {
      Lock lock;
      tmp = pInstance;
      if (tmp == 0) {
         tmp = new Singleton;
         ... // insert memory barrier
         pInstance = tmp;
      }
   }
   return tmp;
}
```

Example 12: Pseudocode that follows an example presented by David Bacon.

This is overkill, as Arch Robison points out (in personal communication):

Technically, you don't need full bidirectional barriers. The first barrier must prevent downwards migration of Singleton's construction (by another thread); the second barrier must prevent upwards migration of `pInstance`'s initialization. These are called "acquire" and "release" operations, and may yield better performance than full barriers on hardware (such as Itainum) that makes the distinction.

Still, this is an approach to implementing DCLP that should be reliable, provided you're running on a machine that supports memory barriers. All machines that can reorder writes to shared memory support memory barriers in one form or another. Interestingly, this same approach works just as well in a uniprocessor setting. This is because memory barriers also act as hard sequence points that prevent the kinds of instruction reorderings that can be so troublesome.

Conclusion

There are several lessons to be learned here. First, remember that timeslice-based parallelism on uniprocessors is not the same as true parallelism across multiple processors. That's why a thread-safe solution for a particular compiler on a uniprocessor architecture may not be thread safe on a multiprocessor architecture, not even if you stick with the same compiler. (This is a general observation—it's not specific to DCLP.)

Second, although DCLP isn't intrinsically tied to `Singleton`, the use of `Singleton` tends to lead to a desire to "optimize" thread-safe access via DCLP. You should therefore be sure to avoid implementing `Singleton` with DCLP. If you (or your clients) are concerned about the cost of locking a synchronization object every time `instance` is called, you can advise clients to minimize such calls by caching the pointer that instance returns. For example, suggest that instead of writing code like Example 13(a), clients do things like Example 13(b). Before making such a recommendation, it's generally a good idea to verify that this really leads to a significant performance gain. Use a lock from a threading library to ensure thread-safe `Singleton` initialization, then do timing studies to see if the cost is truly something worth worrying about.

(a)

```
Singleton::instance()->transmogrify();
Singleton::instance()->metamorphose();
Singleton::instance()->transmute();
```

(b)

```
Singleton* const instance =
Singleton::instance(); // cache instance pointer
instance->transmogrify();
instance->metamorphose();
instance->transmute();
```

Example 13: Instead of writing code like (a), clients should use something like (b).

Third, avoid using a lazily initialized `Singleton` unless you really need it. The classic `Singleton` implementation is based on not initializing a resource until that resource is requested. An alternative is to use eager initialization; that is, to initialize a resource at the beginning of the program run. Because multithreaded programs typically start running as a single thread, this approach can push some object initializations into the single-threaded startup portion of the code, thus eliminating the need to worry about threading during the initialization. In many cases, initializing a `Singleton` resource during single-threaded program startup (that is, prior to executing main) is the simplest way to offer fast, thread-safe `Singleton` access.

A different way to employ eager initialization is to replace the Singleton Pattern with the Monostate Pattern (see Steve Ball et al., "Monostate Classes: The Power of One"). Monostate, however, has different problems, especially when it comes to controlling the order of initialization of the nonlocal static objects that make up its state. *Effective C++* (see "References") describes these problems and, ironically, suggests using a variant of `Singleton` to escape them. (The variant is not guaranteed to be thread safe; see *Pattern Hatching: Design Patterns Applied* by John Vlissides.)

Another possibility is to replace a global `Singleton` with one `Singleton` per thread, then use thread-local storage for `Singleton` data. This allows for lazy initialization without worrying about threading issues, but it also means that there may be more than one "`Singleton`" in a multithreaded program.

Finally, DCLP and its problems in C++ and C exemplify the inherent difficulty in writing thread-safe code in a language with no notion of threading (or any other form of concurrency). Multithreading considerations are pervasive because they affect the very core of code generation. As Peter Buhr pointed out in "Are Safe Concurrency Libraries Possible?" (see "References"), the desire to keep multi-threading out of the language and tucked away in libraries is a chimera. Do that, and either the libraries will end up putting constraints on the way compilers generate code (as `Pthreads` already does), or compilers and other code-generation tools will be prohibited from performing useful optimizations, even on single-threaded code. You can pick only two of the troika formed by multithreading, a thread-unaware language, and optimized code generation. Java and the .NET CLI, for example, address the tension by introducing thread awareness into the language and language infrastructure, respectively (see Doug Lea's *Concurrent Programming in Java* and Arch D. Robison's "Memory Consistency & .NET").

Acknowledgments

Drafts of this article were read by Doug Lea, Kevlin Henney, Doug Schmidt, Chuck Allison, Petru Marginean, Hendrik Schober, David Brownell, Arch Robison, Bruce Leasure, and James Kanze. Their comments, insights, and explanations greatly improved the article and led us to our current understanding of DCLP, multithreading, instruction ordering, and compiler optimizations.

References

David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Hahr, Doug Lea, Tom May, Jan-Willem Maessen, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "Double-Checked Locking Pattern is Broken" Declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

Steve Ball and John Crawford. "Monostate Classes: The Power of One." *C++ Report*, May 1997. Reprinted in *More C++ Gems*, Robert C. Martin, ed., Cambridge University Press, 2000.

Peter A. Buhr. "Are Safe Concurrency Libraries Possible?" *Communications of the ACM*, 38(2):117-120, 1995. http://citeseer.nj.nec.com/buhr95are.html.

Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1999. Excerpts relevant to this article can be found at http://gee.cs.oswego.edu/dl/cpj/jmm.html.

Scott Meyers. *Effective C++*, Second Edition. Addison-Wesley, 1998. Item 47 discusses the initialization problems that can arise when using nonlocal static objects in C++.

Arch D. Robison. "Memory Consistency & .NET." *Dr. Dobb's Journal*, April 2003.

Douglas C. Schmidt and Tim Harrison. "Double-Checked Locking." In *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle, and Frank Buschmann, editors. Addison-Wesley, 1998. http://www.cs .wustl.edu/~schmidt/PDF/DC-Locking.pdf.

Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture,* Volume 2. Wiley, 2000. Tutorial notes based on the patterns in this book are available at http://cs.wustl.edu/~schmidt/posa2.ppt.

ISO/IEC 14882:1998(E) International Standard. Programming languages—C++. ISO/IEC, 1998.

John Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, 1998. The discussion of the "Meyers Singleton" is on page 69.

Scott is author of Effective C++ and consulting editor for the Addison-Wesley Effective Software Development Series. He can be contacted at http://aristeia.com/. Andrei is the author of Modern C++ Design *and a columnist for the C/C++ Users Journal. He can be contacted at http://moderncppdesign.com/.*

volatile: A Brief History

To find the roots of volatile, let's go back to the 1970s, when Gordon Bell (of PDP-11 fame) introduced the concept of memory-mapped I/O (MMIO). Before that, processors allocated pins and defined special instructions for performing port I/O. The idea behind MMIO is to use the same pins and instructions for both memory and port access. Hardware outside the processor intercepts specific memory addresses and transforms them into I/O requests; so dealing with ports became simply reading from and writing to machine-specific memory addresses.

What a great idea. Reducing pin count is good—pins slow down signal, increase defect rate, and complicate packaging. Also, MMIO doesn't require special instructions for ports. Programs just use the memory, and the hardware takes care of the rest.

Or almost.

To see why MMIO needs volatile variables, consider the following code:

```
unsigned int *p = GetMagicAddress();
unsigned int a, b;
a = *p;
b = *p;
```

If p refers to a port, a and b should receive two consecutive words read from that port. However, if p points to a bona fide memory location, then a and b load the same location twice and, hence, will compare equal. Compilers exploit this assumption in the copy propagation optimization that transforms b=*p; into the more efficient b = a;. Similarly, for the same p, a, and b, consider:

```
*p = a;
*p = b;
```

The code writes two words to *p, but the optimizer might assume that *p is memory and perform the dead assignment elimination optimization by eliminating the first assignment.

So, when dealing with ports, some optimizations must be suspended. volatile exists for specifying special treatment for ports, specifically: The content of a volatile variable is unstable (can change by means unknown to the compiler); all writes to volatile data are observable, so they must be executed religiously; and all operations on volatile data are executed in the sequence in which they appear in the source code. The first two rules ensure proper reading and writing. The last one allows implementation of I/O protocols that mix input and output.

This is informally what C and C++'s volatile guarantees. Java took volatile a step further by guaranteeing the aforementioned properties across multiple threads. This was an important step, but it wasn't enough to make volatile usable for thread synchronization: The relative ordering of volatile and nonvolatile operations remained unspecified. This omission forces many variables to be volatile to ensure proper ordering.

Java 1.5's volatile has the more restrictive, but simpler, acquire/release semantics: Any read of a volatile is guaranteed to occur prior to any memory reference (volatile or not) in the statements that follow, and any write to a volatile is guaranteed to occur after all memory references in the statements preceding it. .NET defines volatile to incorporate multithreaded semantics as well, which are similar to the currently proposed Java semantics. We know of no similar work being done on C's or C++'s volatile.

— S.M. and A.A.