C++ and The Perils of Double-Checked Locking: Part I

Source Code Accompanies This Article. Download It Now.

- dclock1.txt

In this two-part article, Scott and Andrei examine Double-Checked Locking.

July 01, 2004
URL:http://www.drdobbs.com/cpp/c-and-the-perils-of-double-checked-locki/184405726

Google the newsgroups or Web for the names of design patterns, and you're sure to find that one of the most commonly mentioned is Singleton. Try to put Singleton into practice, however, and you're all but certain to bump into a significant limitation: As traditionally implemented, Singleton isn't thread safe.

Much effort has been put into addressing this shortcoming. One of the most popular approaches is a design pattern in its own right, the Double-Checked Locking Pattern (DCLP); see Douglas C. Schmidt et al., "Double-Checked Locking" and Douglas C. Schmidt et al., *Pattern-Oriented Software Architecture, Volume 2*. DCLP is designed to add efficient thread safety to initialization of a shared resource (such as a Singleton), but it has a problem—it's not reliable. Furthermore, there's virtually no portable way to make it reliable in C++ (or in C) without substantively modifying the conventional pattern implementation. To make matters even more interesting, DCLP can fail for different reasons on uniprocessor and multiprocessor architectures.

In this two-part article, we explain why Singleton isn't thread safe, how DCLP attempts to address that problem, why DCLP may fail on both uni- and multiprocessor architectures, and why you can't (portably) do anything about it. Along the way, we clarify the relationships among statement ordering in source code, sequence points, compiler and hardware optimizations, and the actual order of statement execution. Finally, in the next installment, we conclude with some suggestions regarding how to add thread safety to Singleton (and similar constructs) such that the resulting code is both reliable and efficient.

The Singleton Pattern and Multithreading

The traditional implementation of the Singleton Pattern (see Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*) is based on making a pointer point to a new object the first time the object is requested. In a single-threaded environment, Example 1 generally works fine, though interrupts can be problematic. If you are in `Singleton::instance`, receive an interrupt, and invoke `Singleton::instance` from the handler, you can see how you'd get into trouble. Interrupts aside, however, this implementation works fine in a single-threaded environment.

```
// from the header file
class Singleton {
public:
static Singleton* instance();
   ...
private:
   static Singleton* pInstance;
};


// from the implementation file
Singleton* Singleton::pInstance = 0;

Singleton* Singleton::instance() {
   if (pInstance == 0) {
      pInstance = new Singleton;
   }
   return pInstance;
}
```

Example 1: In single-threaded environments, this code generally works okay.

Unfortunately, this implementation is not reliable in a multithreaded environment. Suppose that Thread `A` enters the instance function, executes through line 14, and is then suspended. At the point where it is suspended, it has just determined that `pInstance` is null; that is, no Singleton object has yet been created.

Thread `B` now enters `instance` and executes line 14. It sees that `pInstance` is null, so it proceeds to line 15 and creates a Singleton for `pInstance` to point to. It then returns `pInstance` to `instance`'s caller.

At some point later, Thread `A` is allowed to continue running, and the first thing it does is move to line 15, where it conjures up another `Singleton` object and makes `pInstance` point to it. It should be clear that this violates the meaning of a Singleton, as there are now two `Singleton` objects.

Technically, line 11 is where `pInstance` is initialized, but for practical purposes, it's line 15 that makes it point where we want it to, so for the remainder of this article, we'll treat line 15 as the point where `pInstance` is initialized.

Making the classic Singleton implementation thread safe is easy. Just acquire a lock before testing `pInstance`, as in Example 2. The downside to this solution is that it may be expensive. Each access to the Singleton requires acquisition of a lock, but in reality, we need a lock only when initializing `pInstance`. That should occur only the first time `instance` is called. If `instance` is called `n` times during the course of a program run, we need the lock only for the first call. Why pay for `n` lock acquisitions when you know that `n`-1 of them are unnecessary? DCLP is designed to prevent you from having to.

```
Singleton* Singleton::instance() {
   Lock lock; // acquire lock (params omitted for simplicity)
   if (pInstance == 0) {
     pInstance = new Singleton;
   }
   return pInstance;
} // release lock (via Lock destructor)
```

Example 2: Acquiring a lock before testing pInstance.

The Double-Checked Locking Pattern

The crux of DCLP is the observation that most calls to `instance` see that `pInstance` is not null, and not even try to initialize it. Therefore, DCLP tests `pInstance` for nullness before trying to acquire a lock. Only if the test succeeds (that is, if `pInstance` has not yet been initialized) is the lock acquired. After that, the test is performed again to ensure `pInstance` is still null (hence, the "double-checked" locking). The second test is necessary because it is possible that another thread happened to initialize `pInstance` between the time `pInstance` was first tested and the time the lock was acquired.

Example 3 is the classic DCLP implementation (see Douglas C. Schmidt et al., "Double-Checked Locking" and Douglas C. Schmidt et al., *Pattern-Oriented Software Architecture, Volume 2*). The papers defining DCLP discuss some implementation issues (that is, the importance of `volatile`-qualifying the Singleton pointer and the impact of separate caches on multiprocessor systems, both of which we address later; as well as the need to ensure the atomicity of certain reads and writes, which we do not discuss in this article), but they fail to consider a much more fundamental problem: Ensuring that the machine instructions executed during DCLP are executed in an acceptable order. This is the fundamental problem we focus on here.

```
Singleton* Singleton::instance() {
   if (pInstance == 0) { // 1st test
      Lock lock;
      if (pInstance == 0) { // 2nd test
         pInstance = new Singleton;
      }
   }
   return pInstance;
}
```

Example 3: The classic DCLP implementation.

DCLP and Instruction Ordering

Consider again `pInstance = new Singleton;`, the line that initializes `pInstance`. This statement causes three things to happen:

- Step 1. Allocate memory to hold a `Singleton` object.
- Step 2. Construct a `Singleton` object in the allocated memory.
- Step 3. Make `pInstance` point to the allocated memory.

Of critical importance is the observation that compilers are not constrained to perform these steps in this order! In particular, compilers are sometimes allowed to swap Steps 2 and 3. Why they might want to do that is a question we'll address in a moment. For now, let's focus on what happens if they do.

Consider Example 4, where we've expanded `pInstance`'s initialization line into the three constituent tasks just mentioned and where we've merged Steps 1 (memory allocation) and 3 (`pInstance` assignment) into a single statement that precedes Step 2 (`Singleton` construction). The idea is not that a human would write this code. Rather, it's that a compiler might generate code equivalent to this in response to the conventional DCLP source code that a human would write.

```
Singleton* Singleton::instance() {
   if (pInstance == 0) {
      Lock lock;
      if (pInstance == 0) {
         pInstance = // Step 3
            operator new(sizeof(Singleton)); // Step 1
         new (pInstance) Singleton; // Step 2
      }
   }
   return pInstance;
}
```

Example 4: pInstance's initialization line expanded into three constituent tasks.

In general, this is not a valid translation of the original DCLP source code because the `Singleton` constructor called in Step 2 might throw an exception. And, if an exception is thrown, it's important that `pInstance` has not yet been modified. That's why, in general, compilers cannot move Step 3 above Step 2. However, there are conditions under which this transformation is legitimate. Perhaps the simplest such condition is when a compiler can prove that the `Singleton` constructor cannot throw (via postinlining flow analysis, for instance), but that is not the only condition. Some constructors that throw can also have their instructions reordered such that this problem arises.

Given the above translation, consider the following sequence of events:

- Thread `A` enters `instance`, performs the first test of `pInstance`, acquires the lock, and executes the statement made up of Steps 1 and 3. It is then suspended. At this point, `pInstance` is not null, but no `Singleton` object has yet been constructed in the memory `pInstance` points to.
- Thread `B` enters instance, determines that `pInstance` is not null, and returns it to `instance`'s caller. The caller then dereferences the pointer to access the `Singleton` that, oops, has not yet been constructed.

DCLP works only if Steps 1 and 2 are completed before Step 3 is performed, but there is no way to express this constraint in C or C++. That's the dagger in the heart of DCLP—you need to define a constraint on relative instruction ordering, but the languages give you no way to express the constraint.

Yes, the C and C++ Standards (see ISO/IEC 9899:1999 International Standard and ISO/IEC 14882:1998(E), respectively) do define sequence points, which define constraints on the order of evaluation. For example, paragraph 7 of Section 1.9 of the C++ Standard encouragingly states:

At certain specified points in the execution sequence called sequence points, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

Furthermore, both Standards state that a sequence point occurs at the end of each statement. So it seems that if you're just careful with how you sequence your statements, everything falls into place.

Oh, Odysseus, don't let thyself be lured by sirens' voices; for much trouble is waiting for thou and thy mates!

Both Standards define correct program behavior in terms of the "observable behavior" of an abstract machine. But not everything about this machine is observable. For example, consider function `Foo` in Example 5 (which looks silly, but might plausibly be the result of inlining some other functions called by `Foo`).

```
void Foo() {
   int x = 0, y = 0; // Statement 1
   x = 5; // Statement 2
   y = 10; // Statement 3
   printf("%d,_%d", x, y); // Statement 4
}
```

Example 5: This code could be the result of inlining some other functions called by Foo.

In both C and C++, the Standards guarantee that `Foo` prints "`5,_10`". But that's about the extent of what we're guaranteed. We don't know whether statements 1-3 will be executed at all and, in fact, a good optimizer will get rid of them. If statements 1-3 are executed, we know that statement 1 precedes statements 2-4 and—assuming that the call to `printf` isn't inlined and the result further optimized—we know about the relative ordering of statements 2 and 3. Compilers might choose to execute statement 2 first, statement 3 first, or even to execute them both in parallel, assuming the hardware has some way to do it. Which it might well have. Modern processors have a large word size and several execution units. Two or more arithmetic units are common. (For example, the Pentium 4 has three integer ALUs, PowerPC's G4e has four, and Itanium has six.) Their machine language allows compilers to generate code that yields parallel execution of two or more instructions in a single clock cycle.

Optimizing compilers carefully analyze and reorder your code so as to execute as many things at once as possible (within the constraints on observable behavior). Discovering and exploiting such parallelism in regular serial code is the single most important reason for rearranging code and introducing out-of-order execution. But it's not the only reason. Compilers (and linkers) might also reorder instructions to avoid spilling data from a register, to keep the instruction pipeline full, to perform common subexpression elimination, and reduce the size of the generated executable (see Bruno De Bus et al., "Post-pass Compaction Techniques").

When performing these kinds of optimizations, C/C++ compilers and linkers are constrained only by the dictates of observable behavior on the abstract machines defined by the language standards, and—this is the important bit—those abstract machines are implicitly single threaded. As languages, neither C nor C++ have threads, so compilers don't have to worry about breaking threaded programs when they are optimizing. It should, therefore, not surprise you that they sometimes do.

That being the case, how can you write C and C++ multithreaded programs that actually work? By using system-specific libraries defined for that purpose. Libraries such as POSIX threads (pthreads) (see ANSI/IEEE 1003.1c-1995) give precise specifications for the execution semantics of various synchronization primitives. These libraries impose restrictions on the code that library-conformant compilers are permitted to generate, thus forcing such compilers to emit code that respects the execution ordering constraints on which those libraries depend. That's why threading packages have parts written in assembler or issue system calls that are themselves written in assembler (or in some unportable language): You have to go outside Standard C and C++ to express the ordering constraints that multithreaded programs require. DCLP tries to get by using only language constructs. That's why DCLP isn't reliable.

As a rule, programmers don't like to be pushed around by their compilers. Perhaps you are such a programmer. If so, you may be tempted to try to outsmart your compiler by adjusting your source code so that `pInstance` remains unchanged until after Singleton's construction is complete. For example, you might try inserting use of a temporary variable, as in Example 6. In essence, you've just fired the opening salvo in a war of optimization. Your compiler wants to optimize. You don't want it to, at least not here. But this is not a battle you want to get into. Your foe is wiley and sophisticated, imbued with strategems dreamed up over decades by people who do nothing but think about this kind of thing all day long, day after day, year after year. Unless you write optimizing compilers yourself, they are way ahead of you. In this case, for example, it would be a simple matter for the compiler to apply dependence analysis to determine that `temp` is an unnecessary variable, hence, to eliminate it, thus treating your carefully crafted "unoptimizable" code if it had been written in the traditional DCLP manner. Game over. You lose.

```
Singleton* Singleton::instance() {
   if (pInstance == 0) {
      Lock lock;
      if (pInstance == 0) {
         Singleton* temp = new Singleton; // initialize to temp
```

```
            pInstance = temp; // assign temp to pInstance
        }
    }
    return pInstance;
}
```

Example 6: Using a temporary variable.

If you reach for bigger ammo and try moving `temp` to a larger scope (say, by making it file `static`), the compiler can still perform the same analysis and come to the same conclusion. Scope, schmope. Game over. You lose. So you call for backup. You declare `temp extern` and define it in a separate translation unit, thus preventing your compiler from seeing what you are doing. Alas, some compilers have the optimizing equivalent of night-vision goggles: They perform interprocedural analysis, discover your ruse with `temp`, and again optimize it out of existence. Remember, these are `optimizing` compilers. They're supposed to track down unnecessary code and eliminate it. Game over. You lose.

So you try to disable inlining by defining a helper function in a different file, thus forcing the compiler to assume that the constructor might throw an exception and, therefore, delay the assignment to `pInstance`. Nice try, but some build environments perform link-time inlining followed by more code optimizations (see Bruno De Bus et al., "Post-pass Compaction Techniques;" Robert Cohn et al., "Spike: An Optimizer for Alpha/NT Executables;" and Matt Pietrek, "Link-Time Code Generation"). Game over. You lose.

Nothing you do can alter the fundamental problem: You need to be able to specify a constraint on instruction ordering, and your language gives you no way to do it.

Next Month

In the next installment of this two-part article, we'll examine the role of the `volatile` keyword, see what impact DCLP has on multiprocessor machines, and conclude with a few suggestions.

References

Bruno De Bus, Daniel Kaestner, Dominique Chanet, Ludo Van Put, and Bjorn De Sutter. "Post-pass Compaction Techniques." *Communications of the ACM,* 46(8):41-46, August 2003. ISSN 0001-0782. http://doi.acm.org/10.1145/859670.859696.

Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. "Spike: An Optimizer for Alpha/NT Executables." http://www.usenix.org/publications/ library/proceedings/usenix-nt97/ presentations/goodwin/index.htm, August 1997.

ANSI/IEEE 1003.1c-1995, 1995. IEEE Standard for Information Technology. Portable Operating System Interface (POSIX)—System Application Program Interface (API) Amendment 2: Threads Extension (C Language).

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Matt Pietrek. "Link-Time Code Generation." *MSDN Magazine*, May 2002. http://msdn.microsoft.com/msdnmag/issues/02/05/Hood/.

Douglas C. Schmidt and Tim Harrison. "Double-Checked Locking." In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3,* 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.

ISO/IEC 9899:1999 International Standard. Programming languages—C. ISO/IEC, 1999.

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/.

Related Reading

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