Double-Checked Locking, Threads, Compiler Optimizations, and More

Scott Meyers, Ph.D.
Software Development Consultant

(Based on Work with Andrei Alexandrescu)

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614

Andrei's Summary

Multithreading is just one damn thing after, before, or simultaneous with another.

Agenda

- Lazy Initialization, the GOF Singleton Pattern, and Multithreading
- The Double-Checked Locking Pattern (DCLP)
- Compiler Optimizations and Instruction Reordering
- Sequence Points and Observable Behavior
- The impact of `volatile`
- Multiprocessors, Cache Coherency, and Memory Barriers
- Alternatives to DCLP
- Recommended Reading

Not on the Agenda

- Atomicity considerations
  - In this talk, I assume that pointer initializations/assignments are atomic.
  - That's not true for all platforms.

Lazy Initialization

Deferring an object's initialization until first use.

- Helps avoid initialization order problems.
  - ➠ Initialize things as late as possible.
- An efficiency win if an object is never used.

Examples:

```
int x = computeInitValue();            // eager initialization

...                                    // clients refer to x

int xValue() {
  static int x = computeInitValue();   // lazy initialization
  return x;
}

...                                    // clients refer to xValue()
```


GOF Singleton

The classic GOF Singleton implementation is lazy:

```
// from the .h file
class Keyboard {
public:
  static Keyboard* Instance();
  ...
private:
  static Keyboard* pInstance;            // ptr to sole Keyboard object
  ...
};

// from the .cpp file
Keyboard* Keyboard::pInstance = 0;

Keyboard* Keyboard::Instance() {
  if (pInstance == 0) {
    pInstance = new Keyboard;            // lazy initialization here
  }

  return pInstance;
}
```


Classic Singleton Isn't Thread Safe

Here's the implementation again:

```
Keyboard* Keyboard::Instance() {
  if (pInstance == 0) {               // Line 1
    pInstance = new Keyboard;         // Line 2
  }

  return pInstance;
}
```

Consider two threads, A and B:

1. Thread A executes through Line 1, see pInstance as NULL, then is suspended.
2. Thread B executes Line 1, sees pInstance as NULL, then executes Line 2.
   - ➠ At this point, the Singleton has been created.
3. Thread A starts running again and executes Line 2.
   - ➠ This creates *another* Singleton!

Clearly, this implementation of Singleton isn't thread safe.

Adding a Lock

Making it thread safe is easy:

```
Keyboard* Keyboard::Instance() {
  Lock L(args);                       // acquire mutex or other lock
  if (pInstance == 0) {
    pInstance = new Keyboard;
  }

  return pInstance;
}                                     // release lock
```

But this looks expensive:

- Each call to Instance has to acquire a lock.
  - ➠ This typically ain't cheap!
- Only the first call needs it.

We'd like a less costly solution.

The Double-Checked Locking Pattern (DCLP)

Usually, pInstance will be non-NULL, so let's test that before grabbing a lock:

```
Keyboard* Keyboard::Instance() {
  if (pInstance == 0) {                    // first check

    Lock L(args);                          // acquire lock
    if (pInstance == 0) {                  // second check
      pInstance = new Keyboard;
    }
  }

  return pInstance;
}                                          // release lock
```

This is the *Double-Checked Locking Pattern* (DCLP).

- Its goal is efficient lazy initialization of a shared resource.
- It's not reliable.

DCLP and Instruction Ordering

Consider this part of the classic DCLP implementation:

```
pInstance = new Keyboard;
```

It does three things:

1. Allocate memory (via operator new) to hold a Keyboard object.
2. Construct the Keyboard object in the memory.
3. Assign to pInstance the address of the memory.

They need not be done in this order! For example:

```
pInstance =                                // 3
  operator new(sizeof(Keyboard));          // 1
new (pInstance) Keyboard;                  // 2
```

If this code is generated, the order is 1, 3, 2.

DCLP and Instruction Ordering

In context:

```
if (pInstance == 0) {                      // Line 1
  Lock L(args);
  if (pInstance == 0) {
    pInstance =
      operator new(sizeof(Keyboard));      // Line 2
    new (pInstance) Keyboard;
  }
}
```

Consider:

1. Thread A executes through Line 2 and is suspended.
   - ➠ At this point, pInstance is non-NULL, but no singleton has been constructed.
2. Thread B executes Line 1, sees pInstance as non-NULL, returns, and dereferences the pointer returned by Instance (i.e., pInstance).
   - ➠ It attempts to reference an object that's not there yet!

DCLP and Instruction Ordering

The fundamental problem:

- You need a way to specify that Step 3 must come after Steps 1 and 2.
- There is *no way* to specify this in C++.
  - ➠ Or in C. (The rest of this talk is as true for C as it is for C++.)

Sequence Points and Observable Behavior

C++ defines *sequence points*:

- When a sequence point is reached,

  all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place.

- There's a sequence point at the end of each statement, i.e., "at the semicolon."

This suggests that careful statement ordering should allow control over the order of generated instructions.

- It doesn't.
- Compilers may reorder instructions as long as they preserve the *observable behavior* of the C++ abstract machine.
- Observable behavior consists only of
  - Reads and writes of volatile data.
  - Calls to library I/O functions.

Sequence Points and Observable Behavior

Consider:

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

- Statements 1-3 may be optimized away.
  - They're not observable behavior.
- If 1-3 are executed, 1 must precede 2-4 and 4 must follow 1-3.
- We know nothing about the relative execution order of 2 and 3.
  - Either might come first.
  - They might run simultaneously; multiple ALUs are common.

Sequence points thus offer no control over the execution of Statements 1-3.

Aside: Motivations for Instruction Reordering

- Execute operations in parallel where the hardware allows it.
- To avoid spilling data from a register.
- To perform common subexpression elimination.
- To keep the instruction pipeline full.
- To reduce the size of the generated executable.

Programmers vs. Optimizers

Some programmers try to seize control by rewriting the source code.

- E.g., they don't assign pInstance until singleton construction is complete:

```
Keyboard* Keyboard::instance() {
  if (pInstance == 0) {
    Lock L(args);
    if (pInstance == 0) {
      Keyboard* temp = new Keyboard;   //initialize to temp
      pInstance = temp;                //assign temp to pInstance
    }
  }
  return pInstance;
}
```

- But compilers can deduce that temp is unnecessary and eliminate it, thus yielding the original code.

Programmers vs. Optimizers

Analogous attempts to outfox optimizers are, in general, unreliable:

- Declaring temp extern and moving it to a new translation unit.
  - ➠ Compilers with interprocedural optimization can still detect that temp is unnecessary and eliminate it.
- Declaring the singleton constructor in a different translation unit, thus prohibiting inlining and forcing the compiler to assume that the constructor might throw an exception.
  - ➠ Build environments that support link-time inlining followed by additional optimizations can "see through" such obfuscation.
- They're *optimizing* compilers (and linkers).
  - ➠ They're supposed to eliminate unnecessary code!

The Impact of volatile

volatile prevents some compiler optimizations.

- The *order* of reads/writes to volatile data must be preserved.
- volatile memory may change outside of program control, so "redundant" reads/writes in source code must be respected.
  - ➠ Useful for e.g., memory-mapped IO

Important:

- I'm discussing the C++ meaning of volatile.
- The meaning in .NET and Java is different.
  - ➠ They do (or will) add acquire/release semantics to volatile.
  - ➠ We'll discuss acquire/release soon.

The Impact of volatile

The papers describing DCLP point out that pInstance must be volatile.

```
class Keyboard {
private:
  static Keyboard* volatile pInstance;
  ...
};
```

Otherwise the second test might be optimized away:

```
if (pInstance == 0) {                 // first check
  Lock L(args);
  if (pInstance == 0) {               // "redundant" second check
  ...
```


The Impact of volatile

This would seem to make the temp-introducing strategy viable:

```
class Keyboard {
public:
  static Keyboard* instance();
  ...
private:
  static Keyboard* volatile pInstance;
  int x;
  Keyboard() : x(5) {}                          // Note inlined ctor
};

Keyboard* Keyboard::instance() {
  if (pInstance == 0) {
    Lock L(args);
    if (pInstance == 0) {
      Keyboard* volatile temp = new Keyboard;   // these lines can't
      pInstance = temp;                         // be reordered
    }
  }
  return pInstance;
}
```

The Impact of volatile

It doesn't. Here's instance after inlining the constructor:

```
Keyboard* Keyboard::instance() {
  if (pInstance == 0) {
    Lock L(args);
    if (pInstance == 0) {
      Keyboard* volatile temp =
        static_cast<Keyboard*>(operator new(sizeof(Keyboard)));
      temp->x = 5;                          // inlined constructor
      pInstance = temp;
    }
  }
  return pInstance;
}
```

Though temp is volatile, *temp isn't, so instructions can be reordered:

```
      ...
      pInstance = temp;
      temp->x = 5;
      ...
```

If they are, pInstance again points to uninitialized memory.

The Impact of volatile

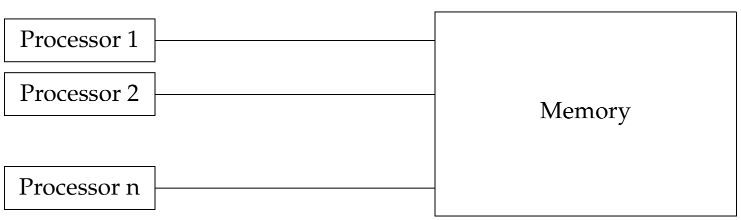
Casts can be added to the singleton constructor to prevent such reorderings, but it still isn't guaranteed to yield thread-safe code:

- Constraints on observable behavior apply only to C++'s abstract machine.
  - This abstract machine is implicitly single-threaded.
  - Use of threads puts one in the territory of undefined behavior.
- In practice, compiler vendors often choose to generate thread-unsafe code, even when volatile is used.
  - Otherwise, lost optimization opportunities lead to too big an efficiency hit.

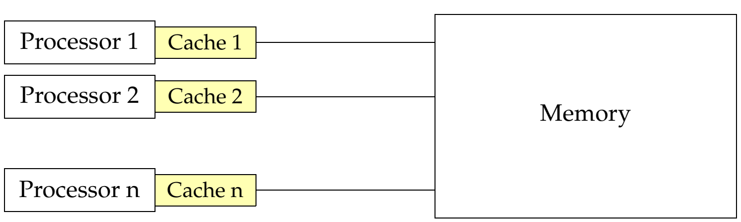
Bottom line: the road to thread-safe code isn't paved with volatile.

Multiprocessors and Cache Coherency

Conceptually, many multiprocessor systems look like this:



But each processor typically has a cache, so this is more accurate:



Multiprocessors and Cache Coherency

This gives rise to the specter of the *cache coherency problem*:

- The value of a shared memory location may have different values in different caches.

Hardware generally takes care of this problem, but there's a catch:

- Cache contents may be flushed in an order other than that in which they were written.
  - This can improve efficiency.

Consider:

- Processor n modifies shared variable x, then shared variable y.
- When the cache is flushed, y is flushed before x.
- Other processors may thus see y's value change before x's.
  - This is never a problem for processor n, only for other processors.
  - It's an issue only for multiprocessor systems.

Memory Visibility

The cache coherency problem is an aspect of a more general issue:

- **Memory visibility**: what guarantees do we have about how different threads see the contents of shared memory?

A platform's *memory model* describes its guarantees.

- C++ offers no guarantees at all.
  - ➠ Libraries callable from C++ (e.g., Posix) do offer some guarantees.
- Some other languages (e.g., Java) offer guarantees out of the box.

DCLP and Multiprocessors

Out-of-order write visibility can cause DCLP to fail. Recall that

```
pInstance = new Keyboard;
```

does three things:

1. Allocate memory (via operator new) to hold a Keyboard object.
2. Construct the Keyboard object in the memory.
3. Assign to pInstance the address of the memory.

Consider this scenario:

- Processor n does these steps in order.
- Processor m sees the results of step 3 before the results of step 2.
  - ➠ It thus thinks pInstance points to an initialized object before it does.

Memory Barriers

Such visibility problems can be prevented via the use of *memory barriers* (aka *fences*):

- Special instructions that constrain "out of order" reads and writes.

Barriers allow readers and writers to perform a handshake that gives them a consistent view of memory values:

- Readers use an *acquire* barrier to prevent subsequent source code memory accesses from moving "up in time" to before the barrier.
- Writers use a *release* barrier to prevent prior source code memory accesses from moving "down in time" to after the barrier.

The handshake is the release/acquire pair:

- The release changes the state of shared memory, and the acquire guarantees that the reader see the new state.

Memory Barriers

Note that memory barriers have both static and dynamic implications:

- Compilers may not reorder reads or writes in a way that violates memory barriers semantics.
  - ➠ This is a static constraint.
- Runtime systems (including hardware) may not do anything that violates memory barrier semantics.
  - ➠ This is a dynamic constraint.

The details of available memory barrier instructions can vary from architecture to architecture.

- E.g., on Sparc, it is possible to create barriers only for reads or for writes.

DCLP With Memory Barriers

Here's a thread-safe version of DCLP for processors supporting memory barriers:

```
Keyboard* Keyboard::instance() {
  Keyboard* temp = pInstance;     // read pInstance
  Perform acquire;                // prevent visibility of later mem. ops.
                                  // from preceding pInstance's read
  if (temp == 0) {
    Lock L(args);
    if (pInstance == 0) {
      temp = new Keyboard;
      Perform release;            // prevent visibility of earlier mem. ops
                                  // from succeeding pInstance's write
      pInstance = temp;           // write pInstance
    }
  }
  return pInstance;
}
```

Unfortunately, memory barrier instructions tend to be architecture-specific, so code like this is not portable.

A Rule for Avoiding Memory Visibility Problems

Shared data should be accessed only if one of the following is true:

- The access is inside a critical section.
- For communication via message passing, these protocols are followed:
  - ➠ Read "ready" flag; perform acquire; read message
  - ➠ Write message; perform release; write "ready" flag

A Rule for Avoiding Memory Visibility Problems

DCLP effectively uses message passing:

- pInstance is the "ready" flag (non-null => ready)
- *pInstance is the message

But DCLP fails to follow the protocol:

```
Keyboard* Keyboard::Instance() {
  if (pInstance == 0) {           // read flag
    Lock L(args);
    if (pInstance == 0) {
      pInstance = new Keyboard;
    }
  }

  return pInstance;               // allow caller to read message
}                                 // (note lack of intervening acquire
                                  // when pInstance is non-null)
```


A Rule for Avoiding Memory Visibility Problems

The pseudocode using memory barriers does follow the protocol:

```
Keyboard* Keyboard::instance() {
  Keyboard* temp = pInstance;          // read flag
  Perform acquire;                     // acquire
  if (temp == 0) {
    Lock L(args);
    if (pInstance == 0) {
      temp = new Keyboard;             // write message
      Perform release;                 // release
      pInstance = temp;                // write flag
    }
  }
  return pInstance;                    // allow caller to
}                                      // read message
```

Note that the protocol is followed regardless of whether temp is null.

A Closer Look at Locks

Look again at the code between the memory barriers:

```
if (temp == 0) {
   Lock L(args);
   if (pInstance == 0) {
      temp = new Keyboard;
```

It's critical that L be initialized before testing pInstance!

- Otherwise we have our original problem: another thread could be modifying pInstance while we're testing it.

But what keeps optimizers from rewriting the code like this?

```
if (temp == 0) {
   if (pInstance == 0) {
      Lock L(args);
      temp = new Keyboard;
```

Locks in threading libraries work, so *something* must prevent this.

A Closer Look at Locks

Threading libraries ensure that

- Lock acquisition includes the moral equivalent of an acquire
- Lock release includes the moral equivalent of a release

So the pseudocode with memory barriers is equivalent to this:

```
Keyboard* Keyboard::instance() {
   Keyboard* temp = pInstance;
   Perform acquire;
   if (temp == 0) {
      Lock L(args);
      Perform acquire;           // visibility of mem. ops. in L's ctor must
      if (pInstance == 0) {      // precede all mem ops. that follow
         temp = new Keyboard;
         Perform release;
         pInstance = temp;
      }
      Perform release;           // visibility of mem. ops. in L's dtor must
   }                             // follow all mem. ops. that precede
   return pInstance;
}
```


A Closer Look at Locks

Locks thus provide a "portable" way to insert memory barriers.

- They're as portable as the threading library.
  - E.g. Posix is pretty portable.

We can thus rewrite the previous page using locks:

```
Keyboard* Keyboard::instance() {
   Keyboard* temp = pInstance;
   Lock L1(args);                       // acquire
   if (temp == 0) {
      Lock L2(args);                    // acquire
      if (pInstance == 0) {
         {
            Lock L3(args);              // acquire
            temp = new Keyboard;
         }                              // release (L3)
         pInstance = temp;
      }                                 // release (L2)
   }
   return pInstance;
}                                       // release (L1)
```


Back Where We Started

This grabs a lock each time instance is called.

- Just like the original "easy" (and potentially inefficient) code.

We might as well eliminate the extra locks and use the obvious design!

```
Keyboard* Keyboard::Instance() {      // from page 8
   Lock L(args);
   if (pInstance == 0) {
      pInstance = new Keyboard;
   }

   return pInstance;
}
```

Note that this code also satisfies our earlier rule:

- Access to shared data is now inside a critical section.

Conclusion

There is no portable way to implement DCLP in C++.

Alternatives to DCLP

Move initialization actions into the single-threaded program startup code.

- Often an easy way to ensure both efficiency and thread safety.
- It replaces lazy evaluation with eager evaluation.

Replace global singletons with per-thread singletons:

- Each can use thread-local storage.
  - ➠ Threading concerns during initialization thus vanish.
- But now there are multiple "singletons."

Stop worrying.

- Grab a lock each time Instance is called.
  - ➠ Maybe it won't be the bottleneck you fear.
  - ➠ If it is, advise users to cache the returned pointer.

Many *Many* Thanks To

Doug Lea

Kevlin Henney

Petru Marginean

Arch Robison

James Kanze

Recommended Reading

The article on which this talk is based:

- "C++ and the Perils of Double-Checked Locking," Scott Meyers and Andrei Alexandrescu, *Dr. Dobbs Journal*, June (Part 1) and July (Part 2), 2004.

Recommended Reading

The Singleton Pattern:

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995, ISBN 0-201-63361-2. Also available as *Design Patterns CD*, Addison-Wesley, 1998, ISBN 0-201-63498-8.

Variations:

- *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1998, ISBN 0-201-92488-9, pp. 219-223.
- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X, pp. 130-138.
- *Modern C++ Design: Generic Programming and Design Patterns Applied*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5, pp. 129-156.
- *Pattern Hatching: Design Patterns Applied*, John Vlissides, Addison-Wesley, 1998, ISBN 0-201-43293-5, pp. 61-72.

Recommended Reading

The Double-Checked Locking Pattern:

- "Double-Checked Locking," Douglas Schmidt and Tim Harrison, in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998. Available at http://www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf.
  - A slightly-revised version appears in *Pattern-Oriented Software Architecture, Volume 2* (POSA2), Wiley, 2000. Tutorial notes for the book's patterns are available at http://cs.wustl.edu/~schmidt/posa2.ppt.
- *The "Double-Checked Locking is Broken" Declaration*, available at http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.
- *Synchronization and the Java Memory Model*, Doug Lea, available at http://gee.cs.oswego.edu/dl/cpj/jmm.html.
- "Warning! Threading in a Multiprocessor World", Allen Holub, *JavaWorld*, February 2001, available at http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox_p.html.
- "Multiprocessor Safety and Java", Paul Jakubik, *Visual Developer Magazine*, March/April 2000.

Recommended Reading

Memory Barriers/Fences and Memory Visibility:

- "Memory Consistency & .NET," Arch Robison, *Dr. Dobb's Journal*, April 2003.
- *Programming with POSIX Threads*, David R. Butenhof, Addison-Wesley, 1997, ISBN 0-201-63392-2, pp. 88-95.

Please Note

Scott Meyers offers consulting services in all aspects of the design and implementation of C++ software systems. For details, visit his web site:

http://www.aristeia.com/

Scott also offers a mailing list to keep you up to date on his professional publications and activities. Read about the mailing list at:

http://www.aristeia.com/MailingList/