WIKIPEDIA

Double-checked locking

In software engineering, **double-checked locking** (also known as "double-checked locking optimization"[1]) is a software design pattern used to reduce the overhead of acquiring a lock by testing the locking criterion (the "lock hint") before acquiring the lock. Locking occurs only if the locking criterion check indicates that locking is required.

The pattern, when implemented in some language/hardware combinations, can be unsafe. At times, it can be considered an anti-pattern.[2]

It is typically used to reduce locking overhead when implementing "lazy initialization" in a multi-threaded environment, especially as part of the Singleton pattern. Lazy initialization avoids initializing a value until the first time it is accessed.

Contents

Usage in C++11
Usage in Golang
Usage in Java
Usage in C#
See also
References
External links

Usage in C++11

For the singleton pattern, double-checked locking is not needed:

> If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.
>
> — § 6.7 [stmt.dcl] p4

```
Singleton& GetInstance() {
  static Singleton s;
  return s;
}
```

If one wished to use the double-checked idiom instead of the trivially working example above (for instance because Visual Studio before the 2015 release did not implement the C++11 standard's language about concurrent initialization quoted above [3]), one needs to use acquire and release fences:[4]

```
#include <atomic>
#include <mutex>

class Singleton {
 public:
  Singleton* GetInstance();
 private:
  Singleton() = default;

  static std::atomic<Singleton*> s_instance;
  static std::mutex s_mutex;
};

Singleton* Singleton::GetInstance() {
  Singleton* p = s_instance.load(std::memory_order_acquire);
  if (p == nullptr) {
    std::lock_guard<std::mutex> lock(s_mutex);
    p = s_instance.load(std::memory_order_relaxed);
    if (p == nullptr) {
      p = new Singleton();
      s_instance.store(p, std::memory_order_release);
    }
  }
  return p;
}
```

Usage in Golang

```
package main
import "sync"

var arrMu sync.Mutex
var arr []int

// getArr retrieves arr, lazily initializing if needed. Double-checked locking
// avoids locking the entire function, and ensures that arr will be
// initialized only once.
func getArr() *[]int {
    if arr != nil { // 1st check
        return &arr
    }

    arrMu.Lock()
    defer arrMu.Unlock()

    if arr != nil { // 2nd check
        return &arr
    }
    arr = []int{0, 1, 2}
    return &arr
}

func main() {
    // thanks to double-checked locking, two goroutines attempting to getArr()
    // will not cause double-initialization
    go getArr()
    go getArr()
}
```

Usage in Java

Consider, for example, this code segment in the Java programming language as given by [2] (as well as all other Java code segments):

```
// Single-threaded version
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }

    // other functions and members...
}
```

The problem is that this does not work when using multiple threads. A lock must be obtained in case two threads call `getHelper()` simultaneously. Otherwise, either they may both try to create the object at the same time, or one may wind up getting a reference to an incompletely initialized object.

The lock is obtained by expensive synchronizing, as is shown in the following example.

```
// Correct but possibly expensive multithreaded version
class Foo {
    private Helper helper;
    public synchronized Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }

    // other functions and members...
}
```

However, the first call to `getHelper()` will create the object and only the few threads trying to access it during that time need to be synchronized; after that all calls just get a reference to the member variable. Since synchronizing a method could in some extreme cases decrease performance by a factor of 100 or higher,[5] the overhead of acquiring and releasing a lock every time this method is called seems unnecessary: once the initialization has been completed, acquiring and releasing the locks would appear unnecessary. Many programmers have attempted to optimize this situation in the following manner:

1. Check that the variable is initialized (without obtaining the lock). If it is initialized, return it immediately.
2. Obtain the lock.
3. Double-check whether the variable has already been initialized: if another thread acquired the lock first, it may have already done the initialization. If so, return the initialized variable.
4. Otherwise, initialize and return the variable.

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

    // other functions and members...
}
```

Intuitively, this algorithm seems like an efficient solution to the problem. However, this technique has many subtle problems and should usually be avoided. For example, consider the following sequence of events:

1. Thread *A* notices that the value is not initialized, so it obtains the lock and begins to initialize the value.

2. Due to the semantics of some programming languages, the code generated by the compiler is allowed to update the shared variable to point to a **partially constructed object** before *A* has finished performing the initialization. For example, in Java if a call to a constructor has been inlined then the shared variable may immediately be updated once the storage has been allocated but before the inlined constructor initializes the object.[6]
3. Thread *B* notices that the shared variable has been initialized (or so it appears), and returns its value. Because thread *B* believes the value is already initialized, it does not acquire the lock. If *B* uses the object before all of the initialization done by *A* is seen by *B* (either because *A* has not finished initializing it or because some of the initialized values in the object have not yet percolated to the memory *B* uses (cache coherence)), the program will likely crash.

One of the dangers of using double-checked locking in J2SE 1.4 (and earlier versions) is that it will often appear to work: it is not easy to distinguish between a correct implementation of the technique and one that has subtle problems. Depending on the compiler, the interleaving of threads by the scheduler and the nature of other concurrent system activity, failures resulting from an incorrect implementation of double-checked locking may only occur intermittently. Reproducing the failures can be difficult.

As of J2SE 5.0, this problem has been fixed. The volatile keyword now ensures that multiple threads handle the singleton instance correctly. This new idiom is described in [3] (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html) and [4] (http://www.oracle.com/technetwork/articles/javase/bloch-effective-08-qa-140880.html).

```java
// Works with acquire/release semantics for volatile in Java 1.5 and later
// Broken under Java 1.4 and earlier semantics for volatile
class Foo {
    private volatile Helper helper;
    public Helper getHelper() {
        Helper localRef = helper;
        if (localRef == null) {
            synchronized (this) {
                localRef = helper;
                if (localRef == null) {
                    helper = localRef = new Helper();
                }
            }
        }
        return localRef;
    }

    // other functions and members...
}
```

Note the local variable "`localRef`", which seems unnecessary. The effect of this is that in cases where `helper` is already initialized (i.e., most of the time), the volatile field is only accessed once (due to "`return localRef;`" instead of "`return helper;`"), which can improve the method's overall performance by as much as 25 percent.[7]

Java 9 introduced the `VarHandle (https://docs.oracle.com/javase/10/docs/api/java/lang/invoke/VarHandle.html)` class, which allows use of relaxed atomics to access fields, giving somewhat faster reads on machines with weak memory models, at the cost of more difficult mechanics and loss of sequential consistency (field accesses no longer participate in the synchronization order, the global order of accesses to volatile fields).[8]

```java
// Works with acquire/release semantics for VarHandles introduced in Java 9
class Foo {
    private volatile Helper helper;

    public Helper getHelper() {
        Helper localRef = getHelperAcquire();
        if (localRef == null) {
            synchronized (this) {
                localRef = getHelperAcquire();
                if (localRef == null) {
                    localRef = new Helper();
                    setHelperRelease(localRef);
                }
            }
        }
        return localRef;
    }

    private static final VarHandle HELPER;
    private Helper getHelperAcquire() {
        return (Helper) HELPER.getAcquire(this);
    }
    private void setHelperRelease(Helper value) {
        HELPER.setRelease(this, value);
    }

    static {
        try {
            MethodHandles.Lookup lookup = MethodHandles.lookup();
            HELPER = lookup.findVarHandle(Foo.class, "helper", Helper.class);
        } catch (ReflectiveOperationException e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    // other functions and members...
}
```

If the helper object is static (one per class loader), an alternative is the initialization-on-demand holder idiom[9] (See Listing 16.6[10] from the previously cited text.)

```
// Correct lazy initialization in Java
class Foo {
    private static class HelperHolder {
       public static final Helper helper = new Helper();
    }

    public static Helper getHelper() {
        return HelperHolder.helper;
    }
}
```

This relies on the fact that nested classes are not loaded until they are referenced.

Semantics of `final` field in Java 5 can be employed to safely publish the helper object without using `volatile`:[11]

```
public class FinalWrapper<T> {
    public final T value;
    public FinalWrapper(T value) {
        this.value = value;
    }
}

public class Foo {
   private FinalWrapper<Helper> helperWrapper;

   public Helper getHelper() {
      FinalWrapper<Helper> tempWrapper = helperWrapper;

      if (tempWrapper == null) {
          synchronized (this) {
              if (helperWrapper == null) {
                  helperWrapper = new FinalWrapper<Helper>(new Helper());
              }
              tempWrapper = helperWrapper;
          }
      }
      return tempWrapper.value;
   }
}
```

The local variable `tempWrapper` is required for correctness: simply using `helperWrapper` for both null checks and the return statement could fail due to read reordering allowed under the Java Memory Model.[12] Performance of this implementation is not necessarily better than the `volatile` implementation.

Usage in C#

Double-checked locking can be implemented efficiently in .NET. A common usage pattern is to add double-checked locking to Singleton implementations:

```
public class MySingleton {
    private static object myLock = new object();
    private static volatile MySingleton mySingleton = null; // 'volatile' is unnecessary in .NET 2.0 and later

    private MySingleton() {
    }

    public static MySingleton GetInstance() {
        if (mySingleton == null) { // 1st check
            lock (myLock) {
                if (mySingleton == null) { // 2nd (double) check
                    mySingleton = new MySingleton();
                    // In .NET 1.1, write-release semantics are implicitly handled by marking mySingleton with
                    // 'volatile', which inserts the necessary memory barriers between the constructor call
                    // and the write to mySingleton. The barriers created by the lock are not sufficient
                    // because the object is made visible before the lock is released. In .NET 2.0 and later,
                    // the lock is sufficient and 'volatile' is not needed.
                }
            }
        }
        // In .NET 1.1, the barriers created by the lock are not sufficient because not all threads will
        // acquire the lock. A fence for read-acquire semantics is needed between the test of mySingleton
        // (above) and the use of its contents.This fence is automatically inserted because mySingleton is
        // marked as 'volatile'.
        // In .NET 2.0 and later, 'volatile' is not required.
        return mySingleton;
    }
}
```

In this example, the "lock hint" is the mySingleton object which is no longer null when fully constructed and ready for use.

In .NET Framework 4.0, the `Lazy<T>` class was introduced, which internally uses double-checked locking by default (ExecutionAndPublication mode) to store either the exception that was thrown during construction, or the result of the function that was passed to `Lazy<T>`:[13]

```
public class MySingleton
{
    private static readonly Lazy<MySingleton> _mySingleton = new Lazy<MySingleton>(() => new MySingleton());

    private MySingleton() { }

    public static MySingleton Instance
    {
        get
        {
            return _mySingleton.Value;
        }
    }
}
```

See also

- The Test and Test-and-set idiom for a low-level locking mechanism.

- Initialization-on-demand holder idiom for a thread-safe replacement in Java.

References

1. Schmidt, D et al. Pattern-Oriented Software Architecture Vol 2, 2000 pp353-363
2. David Bacon et al. The "Double-Checked Locking is Broken" Declaration (http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html).
3. "Support for C++11-14-17 Features (Modern C++)" (https://msdn.microsoft.com/en-au/library/hh567368.aspx#concurrencytable).
4. Double-Checked Locking is Fixed In C++11 (http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/)
5. Boehm, Hans-J (Jun 2005). "Threads cannot be implemented as a library" (http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf) (PDF). *ACM SIGPLAN Notices*. **40** (6): 261–268. doi:10.1145/1064978.1065042 (https://doi.org/10.1145%2F1064978.1065042).
6. Haggar, Peter (1 May 2002). "Double-checked locking and the Singleton pattern" (http://www.ibm.com/developerworks/java/library/j-dcl/index.html). IBM.
7. Joshua Bloch "Effective Java, Second Edition", p. 283-284
8. "Chapter 17. Threads and Locks" (https://docs.oracle.com/javase/specs/jls/se10/html/jls-17.html#jls-17.4.4). *docs.oracle.com*. Retrieved 2018-07-28.
9. Brian Goetz et al. Java Concurrency in Practice, 2006 pp348
10. Goetz, Brian; et al. "Java Concurrency in Practice – listings on website" (http://jcip.net.s3-website-us-east-1.amazonaws.com/listings.html). Retrieved 21 October 2014.
11. [1] (https://mailman.cs.umd.edu/mailman/private/javamemorymodel-discussion/2010-July/000422.html) Javamemorymodel-discussion mailing list
12. [2] (http://jeremymanson.blogspot.ru/2008/12/benign-data-races-in-java.html) Manson, Jeremy (2008-12-14). "Date-Race-Ful Lazy Initialization for Performance – Java Concurrency (&c)" (http://jeremymanson.blogspot.ru/2008/12/benign-data-races-in-java.html). Retrieved 3 December 2016.
13. Albahari, Joseph (2010). "Threading in C#: Using Threads" (http://www.albahari.com/threading/part3.aspx#_LazyT). *C# 4.0 in a Nutshell*. O'Reilly Media. ISBN 978-0-596-80095-6. "`Lazy<T>` actually implements […] double-checked locking. Double-checked locking performs an additional volatile read to avoid the cost of obtaining a lock if the object is already initialized."

External links

- Issues with the double checked locking mechanism captured in Jeu George's Blogs (https://web.archive.org/web/20060620041255/http://purevirtuals.com/blog/2006/06/16/son-of-a-bug/)
- "Double Checked Locking" Description from the Portland Pattern Repository
- "Double Checked Locking is Broken" Description from the Portland Pattern Repository
- Paper "C++ and the Perils of Double-Checked Locking (http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)" (475 KB) by Scott Meyers and Andrei Alexandrescu
- Article "Double-checked locking: Clever, but broken (http://www.javaworld.com/jw-02-2001/jw-0209-double.html)" by Brian Goetz
- Article "Warning! Threading in a multiprocessor world (http://www.javaworld.com/javaworld/jw-02-2001/jw-0209-toolbox.html)" by Allen Holub
- Double-checked locking and the Singleton pattern (http://www.ibm.com/developerworks/java/library/j-dcl/index.html)
- Singleton Pattern and Thread Safety (https://web.archive.org/web/20060412081055/http://www.oaklib.org/docs/oak/singleton.html)
- volatile keyword in VC++ 2005 (http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx)
- Java Examples and timing of double check locking solutions (http://blogs.oracle.com/cwebster/entry/double_check_locking)

- "More Effective Java With Google's Joshua Bloch" (http://www.oracle.com/technetwork/articles/javase/bloch-effective-08-qa-140880.html).
