Double-Checked Locking

An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Abstract

This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.

1 Intent

The Double-Checked Locking optimization pattern reduces contention and synchronization overhead whenever "critical sections" of code need to acquire locks just once, but must be thread-safe when they do acquire locks.

2 Also Known As

Lock Hint [2]

3 Motivation

3.1 The Canonical Singleton

Developing correct and efficient concurrent applications is hard. Programmers must learn new mechanisms (such as multi-threading and synchronization APIs) and techniques (such as concurrency control and deadlock avoidance algorithms). In addition, many familiar design patterns (such as Singleton or Iterator [1]) that work well for sequential programs contain subtle assumptions that do not apply in the context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    if (instance_ == 0)
      // Critical section.
      instance_ = new Singleton;

    return instance_;
  }

  void method (void);
  // Other methods and members omitted.

private:
  static Singleton *instance_;
};
```

Application code uses the static Singleton `instance` method to retrieve a reference to the Singleton before performing operations, as follows:

```
  // ...
  Singleton::instance ()->method ();
  // ...
```

3.2 The Problem: Race Conditions

Unfortunately, the canonical implementation of the Singleton pattern shown above does not work in the presence of preemptive multi-tasking or true parallelism. For instance, if multiple threads executing on a parallel machine invoke `Singleton::instance` simultaneously *before it is initialized*, the `Singleton` constructor can be called multiple times because multiple threads will execute the `new Singleton` operation within the *critical section* shown above.

A *critical section* is a sequence of instructions that obeys the following invariant: *while one thread/process is executing in the critical section, no other thread/process may be executing in the critical section* [3]. In this example, the initialization of the Singleton is a critical section. Violating the properties of the critical section will, at best, cause a memory leak and, at worst, have disastrous consequences if initialization is not idempotent.[1]

3.3 Common Traps and Pitfalls

A common way to implement a critical section is to add a static `Mutex`[2] to the class. This `Mutex` ensures that the allocation and initialization of the Singleton occurs atomically, as follows:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    // Constructor of guard acquires
    // lock_ automatically.
    Guard<Mutex> guard (lock_);

    // Only one thread in the
    // critical section at a time.

    if (instance_ == 0)
      instance_ = new Singleton;

    return instance_;
    // Destructor of guard releases
    // lock_ automatically.
  }

private:
  static Mutex lock_;
  static Singleton *instance_;
};
```

The `Guard` class employs a C++ idiom (described in [5]) that uses the constructor to acquire a resource automatically when an object of the class is created and uses the destructor to release the resource automatically when it goes out of scope. Since `Guard` is parameterized by the type of lock (such as `Mutex`), this class can be used with a family of synchronization wrappers that conform to a uniform `acquire`/`release` interface. By using `Guard`, every access to `Singleton::instance` will automatically acquire and release the `lock_`.

Even though the critical section should be executed just once, every call to instance must acquire and release the `lock_`. Although this implementation is now thread-safe, the overhead from the excessive locking may be unacceptable. One obvious (though incorrect) optimization is to place the `Guard` inside the conditional check of `instance_`:

```
static Singleton *instance (void)
{
  if (instance_ == 0) {
    Guard<Mutex> guard (lock_);

    // Only come here if instance_
    // hasn't been initialized yet.

    instance_ = new Singleton;
  }
  return instance_;
}
```

This reduces locking overhead it, but doesn't provide thread-safe initialization. There is still a race condition in multi-threaded applications that can cause multiple initializations of `instance_`. For instance, consider two threads that simultaneously check for `instance_ == 0`. Both will succeed, one will acquire the `lock_` via the `guard`, and the other will block. After the first thread initializes the `Singleton` and releases the `lock_`, the blocked thread will obtain the `lock_` and erroneously initialize the `Singleton` for a second time.

3.4 The Solution: the Double-Checked Locking Optimization

A better way to solve this problem is to use *Double-Checked Locking*, which is a pattern for optimizing away unnecessary locking. Ironically, the Double-Checked Locking solution is almost identical to the previous one. Unnecessary locking is avoided by wrapping the call to `new` with another conditional test, as follows:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    // First check
    if (instance_ == 0)
    {
      // Ensure serialization (guard
      // constructor acquires lock_).
      Guard<Mutex> guard (lock_);

      // Double check.
      if (instance_ == 0)
        instance_ = new Singleton;
    }
    return instance_;
    // guard destructor releases lock_.
  }

private:
  static Mutex lock_;
  static Singleton *instance_;
};
```

The first thread that acquires the `lock_` will construct `Singleton` and assign the pointer to `instance_`. All threads that subsequently call `instance` will find `instance_ != 0` and skip the initialization step. The second check prevents a race condition if multiple threads try to initialize the Singleton simultaneously. This handles the case where multiple threads execute in parallel. In the code above, these threads will queue up at `lock_`. When the

[1] Object initialization is idempotent if an object can be reinitialized multiple times without ill effects.

[2] A mutex is a lock which can be acquired and released. If multiple threads attempt to acquire the lock simultaneously, only one thread will succeed; the others will block[4].

```
if (Flag == FALSE)
  {
    Mutex.acquire ();
    if (Flag == FALSE)
      {
        critical section;
        Flag = TRUE;
      }
    Mutex.release ();
  }
```

Figure 1: Structure and Participants in the Double-Checked Locking Pattern

queued threads finally obtain the mutex `lock_`, they will find `instance_ != 0` and skip the initialization of `Singleton`.

The implementation of `Singleton::instance` above only incurs locking overhead for threads that are active inside of `instance` when the `Singleton` is first initialized. In subsequent calls to `Singleton::instance`, `singleton_` is not `0` and the `lock_` is not acquired or released.

By adding a `Mutex` and a second conditional check, the canonical Singleton implementation can be made thread-safe without incurring any locking overhead after initialization has occurred. It's instructive to note how the need for Double-Checked Locking emerged from a change in forces, *i.e.*, the addition of multi-threading and parallelism to Singleton. However, the optimization is also applicable for non-Singleton use-cases, as described below.

4 Applicability

Use the Double-Checked Locking Pattern when an application has the following characteristics:

- The application has one or more critical sections of code that must execute sequentially;
- Multiple threads can potentially attempt to execute the critical section simultaneously;
- The critical section is executed just once;
- Acquiring a lock on every access to the critical section causes excessive overhead;
- It is possible to use a lightweight, yet reliable, conditional test in lieu of a lock.

5 Structure and Participants

The structure and participants of the Double-Checked Locking pattern is best shown with pseudocode. Figure 1 illustrates the following participants in the Double-Checked Locking pattern:

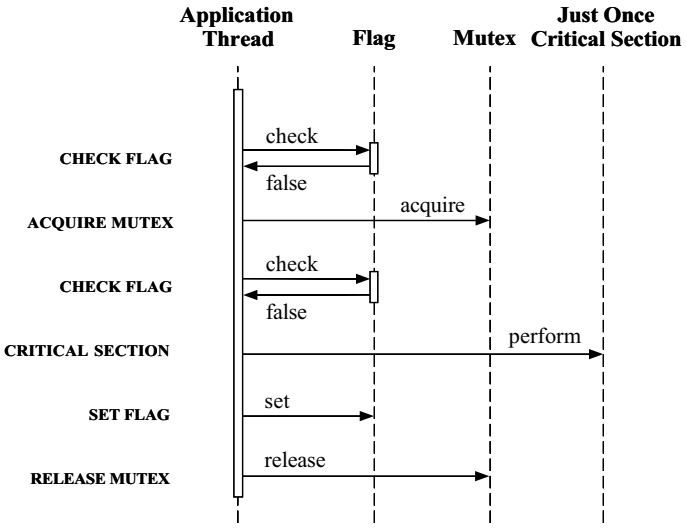

Figure 2: Participant collaborations in the Double-Checked Locking Pattern

- `Just Once Critical Section` – The critical section contains code that is executed just once. For instance, a Singleton is typically initialized only once. Thus, the call to `new Singleton` in Section 3 is executed rarely, relative to the accesses to `Singleton::instance`.
- `Mutex` – A lock that serializes access to the critical section of code. Using the Singleton example from Section 3, the `Mutex` ensures that `new Singleton` only occurs once.
- `Flag` – A flag that indicates whether the `Critical Section` has been executed already. In the Singleton example, the Singleton `instance_` pointer is used as the flag.

  If, in addition to signaling that the event occurred, the `Flag` is used for an application-specific reason (as in the Singleton example), it must be an atomic type that can be set without interruption. This issue is discussed in Section 7.
- `Application Thread` – This is the thread attempting to perform the `Critical Section`. It is implicit in the pseudocode in Figure 1.

6 Collaborations

Figure 2 illustrates the interactions between the participants of the Double-Checked Locking Pattern. The `Application Thread` first checks to see if `Flag` has been set as an optimization for the common case. If it has not been set, `Mutex` is acquired. While holding the lock, the `Application Thread` again checks that the `Flag` is set, performs the `Just Once Critical Section`,

and sets `Flag` to true. Finally, the `Application Thread` releases the lock.

7 Consequences

There are several advantages of using the Double-Checked Locking pattern:

- *Minimized locking* – By performing two `Flag` checks, the Double-Checked Locking pattern optimizes for the common case. Once `Flag` is set, the first check ensures that subsequent accesses require no locking. Section 8.2 shows how this can affect application performance.
- *Prevents race conditions* – The second check of `Flag` ensures that the event is performed only once.

There is also a disadvantage of using the Double-Checked Locking pattern:

- *Potential for subtle portability bugs* – There is a subtle portability issue that can lead to pernicious bugs if the Double-Checked Locking pattern is used in software that is ported to hardware platforms that have non-atomic pointer or integral assignment semantics. For example, if an `instance_` pointer is used as the `Flag` in the implementation of a `Singleton`, all the bits of the `Singleton` `instance_` pointer must be both read and written in single operations. If the write to memory resulting from the call to `new` is not atomic, other threads may try to read an invalid pointer. This would likely result in an illegal memory access.

  Such scenarios are possible on systems where memory addresses straddle alignment boundaries, thereby requiring two fetches from memory for each access. In this case, it may be necessary to use a separate, word-aligned integral `Flag` (assuming that the hardware supports atomic integral reads and writes), instead of using the `instance_` pointer.

  A related problem can occur if an overly-agressive compiler optimizes `Flag` by caching it in some way (*e.g.*, storing `Flag` in a register) or by removing the second check of `Flag == 0`. Section 9 explains how to solve these problems by using the `volatile` keyword.

8 Implementation and Sample Code

The ACE toolkit [6] uses the Double-Checked Locking pattern in several library components. For instance, to reduce code duplication, ACE uses a reusable adapter `ACE_Singleton` to transform "normal" classes to have Singleton-like behavior. The following code shows how the implementation of `ACE_Singleton` uses the Double-Checked Locking pattern.

```
// A Singleton Adapter: uses the Adapter
// pattern to turn ordinary classes into
// Singletons optimized with the
// Double-Checked Locking pattern.
template <class TYPE, class LOCK>
class ACE_Singleton
{
public:
  static TYPE *instance (void);

protected:
  static TYPE *instance_;
  static LOCK lock_;
};

template <class TYPE, class LOCK> TYPE *
ACE_Singleton<TYPE, LOCK>::instance ()
{
  // Perform the Double-Checked Locking to
  // ensure proper initialization.
  if (instance_ == 0) {
    ACE_Guard<LOCK> lock (lock_);
    if (instance_ == 0)
      instance_ = new TYPE;
  }
  return instance_;
}
```

`ACE_Singleton` is parameterized by `TYPE` and `LOCK`. Therefore, a class of the given `TYPE` is converted into a Singleton using a mutex of `LOCK` type.

One usage of `ACE_Singleton` is in the ACE `Token_Manager`. The `Token_Manager` performs deadlock detection for local or remote tokens (such mutexes and readers/writers locks) in multi-threaded applications. To minimize resource usage, the `Token_Manager` is created "on-demand" when first referenced through its `instance` method. To create a Singleton `Token_Manager` simply requires the following `typedef`:

```
typedef ACE_Singleton <ACE_Token_Manager,
                       ACE_Thread_Mutex>
        Token_Mgr;
```

The `Token_Mgr` Singleton is used to detect deadlock within local and remote `token` objects. Before a thread blocks waiting for a mutex, it first queries the `Token_Mgr` Singleton to test if blocking would result in a deadlock situation. For each token in the system, the `Token_Mgr` maintains a record listing the token's owning thread and all thread's blocked waiting for the token. This data is sufficient to test for a deadlock situation. The use of the `Token_Mgr` Singleton is shown below:

```
// Acquire the mutex.
int Mutex_Token::acquire (void)
{
  // ...
  // If the token is already held, we must block.
  if (mutex_in_use ()) {
    // Use the Token_Mgr Singleton to check
    // for a deadlock situation *before* blocking.
    if (Token_Mgr::instance ()->testdeadlock ()) {
      errno = EDEADLK;
      return -1;
    }
    else
      // Sleep waiting for the lock...
  }
```

```
  // Acquire lock...
}
```

Note that the ACE_Singleton can be parameterized by the type of Mutex used to acquire and release the lock (*e.g.,* Singleton<ACE_Thread_Mutex>). This allows ACE_Singleton to be parameterized with an ACE_Null_Mutex for single-threaded platforms and a regular ACE_Thread_Mutex for multi-threaded platforms.

8.1 Evaluation

The above example highlights several advantages to the ACE_Singleton implementation:

- *Avoids implementation errors* – Reusing the thread-safe algorithm for Singletons in ACE_Singleton guarantees that Double-Checked Locking pattern is applied.
- *Adapts non-Singletons* – Classes that were not originally implemented as Singletons can be adapted without altering code. This is especially useful when the source code is not accessible.

There is also a disadvantage with this implementation:

- *Intent violation* – Use of ACE_Singleton does not ensure that a class has only one instance. For instance, there is nothing to prevent multiple Token_Managers from being created. When possible, it may be safer to modify the class implementation directly rather than using the ACE_Singleton adapter.

8.2 Performance Tests

To illustrate the potential performance gains of implementing the Double-Checked Locking pattern we've profiled the access to various implementations of Singleton. For these tests, we used the following implementations:

- *Mutex Singleton* – This implementation of Singleton acquired a mutex lock for every call to the instance accessor.
- *Double-Checked Singleton* – This implementation used the Double-Checked Locking pattern to eliminate unnecessary lock acquisition.
- *ACE Singleton* – This implementation of Singleton uses ACE_Singleton (which is a template that uses the Double-Checked Locking pattern), to test for any overhead associated with the additional abstraction.

Each of the tests used the following algorithm:

```
timer.start ();
for (i = 0; i < 100000000; i++)
  My_Singleton::instance ()->do_nothing ();
timer.stop ();
```

The code for all of these tests are available at http://www.cs.wustl.edu/ schmidt/ACE_wrappers/performance-tests/Misc/test_singleton.cpp. The table below shows results from an UltraSparc 2, with two 70 MHz processors, and 256 MB memory. The following are the optimized performance results:

Singleton Implementation	Mutex	Double-Checked	ACE
real time (secs)	442.64	30.22	30.88
user time (secs)	441.47	30.12	30.86
system time (secs)	0	0	0
time per call (usecs)	4.43	0.30	0.31

These results illustrate the performance impact of using the Double-Checked Locking pattern compared with using the "standard" practice of acquiring and releasing a lock on every instance call. Both the ACE_Singleton and hand-coded implementations of the Double-Checked Locking pattern are over 15 times faster than the standard mutex version. These tests were run with only a single thread to compute the base-line overhead. If multiple threads were contending for the lock the performance of the mutex implementation would decrease even more.

9 Variations

A variation to the implementation of the Double-Checked Locking pattern may be required if the compiler optimizes Flag by caching it in some way (*e.g.,* storing Flag in a register). In this case, cache coherency may become a problem if copies of Flag held in registers in multiple threads become inconsistent. In this case, one thread's setting of the value might not be reflected in other threads' copies.

A related problem is that remove the second check of Flag == 0 since it may appear superfluous to a highly optimizing compiler. For instance, in the following code an aggressive compiler may skip the second memory read for Flag and assume that instance_ is still 0 since it is not declared volatile.

```
Singleton *Singleton::instance (void)
{
  if (Singleton::instance_ == 0)
    {
      // Only lock if instance_ isn't 0.
      Guard<Mutex> guard (lock_);

      // Dead code elimination may
      // remove the next line.
      // Perform the Double-Check.
      if (Singleton::instance_ == 0)
        // ...
```

A solution to both these problems is to declare Flag as a volatile data member in the Singleton, as follows

```
class Singleton
{
  // ...
```

```
private:
  static volatile long Flag_;  // Flag is volatile.
};
```

The use of `volatile` ensures that the compiler will not place the `Flag` into a register, nor will it optimize the second read away. The downside of using `volatile` is that all access to `Flag_` will be through memory, rather than through registers.

10 Known Uses

- The Doubled-Checked Locking pattern is a special case of a very widely used pattern in the Sequent Dynix/PTX operating system.
- The Double-Checked Locking Pattern can be used to implement POSIX `once` variables [7].
- The Double-Checked Locking pattern is used extensively throughout the ACE object-oriented network programming toolkit [6].
- Andrew Birrell describes the use of the Double-Checked Locking optimization in [2]. Birrell refers to the first check of `Flag` as a lock "hint."

11 Related Patterns

The Double-Checked Locking pattern is a thread-safe variant of the *First-Time-In* idiom. The First-Time-In idiom is often used in programming languages (like C) that lack constructors. The following code illustrates this pattern:

```
static const int STACK_SIZE = 1000;
static T *stack_;
static int top_;

void push (T *item)
{
  // First-time-in flag
  if (stack_ == 0) {
    stack_ =
      malloc (STACK_SIZE * sizeof *stack);
    assert (stack_ != 0);
    top_ = 0;
  }
  stack_[top_++] = item;
  // ...
}
```

The first time that `push` is called `stack_` is 0, which triggers its initialization via `malloc`.

Acknowledgments

Thanks to Jim Coplien, Ralph Johnson, Jaco van der Merwe, Duane Murphy, Paul McKenney, Peter Sommerlad, and John Basrai for their suggestions and comments on the Double-Checked Locking pattern.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[3] A. S. Tanenbaum, *Distributed Operating Systems*. Prentice Hall, Enflewood Cliffs, New Jersey, 1995.

[4] A. S. Tanenbaum, *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice Hall, 1992.

[5] Bjarne Stroustrup, *The C++ Programming Language, 2^{nd} Edition*. Addison-Wesley, 1991.

[6] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] IEEE, *Threads Extension for Portable Operating Systems (Draft 10)*, February 1996.