Double-Checked Locking

An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently

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 will appear in the Proceedings of the 3^{rd} annual Pattern Languages of Programming conference held at Allerton Park, Illinois, September 4-6, 1996.

Abstract

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 Observer) that work well for sequential programs contain subtle assumptions that do not apply in the context of concurrency.

As an example, this paper shows how the canonical implementation of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we describe the Double-Checked Locking pattern. This pattern enables Just Once Events (events that are performed only once, such as Singleton initialization), and that unnecessary locking overhead is eliminated. Furthermore, the Double-Checked Locking pattern illustrates how changes in underlying forces (*i.e.,* the addition of multi-threading and parallelism to Singleton) can impact the form and content of patterns used to develop concurrent software.

1 Intent

The Double-Checked Locking design pattern enables atomic Just Once Events and eliminates unnecessary locking overhead on each attempt to perform the event.

2 Also Known As

Lock Hint[1]

3 Motivation

3.1 The Canonical Singleton

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [2]. 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)
  {
    static Singleton *instance_;

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

    return instance_;
  }

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

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

Unfortunately, the canonical implementation of the Singleton pattern shown above does not work in the presence of preemptive multi-tasking or true parallelism. In particular, if multiple threads executing on a parallel machine call `Singleton::instance` simultaneously *before it is initialized*, the `Singleton` constructor can be called multiple times. This, at best, will cause a memory leak and, at worst, may have disastrous consequences if initialization is not idempotent.[1]

3.3 Common Traps and Pitfalls

One way to solve this problem 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)
  {
    static Singleton *instance_;

    // 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_;
};
```

The Guard class employs a C++ idiom (described in [4]) 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_.

Although this implementation is now thread-safe, there is excessive locking overhead since every call to instance must acquire and release the lock_. One obvious (though incorrect) optimization is to place the Guard inside the conditional check of instance_:

```
static Singleton *instance (void)
{
  static Singleton *instance_;

  if (instance_ == 0) {
    Guard<Mutex> guard (lock_);

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

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

Although this reduces locking overhead it 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 Double-Checked Locking Solution

A better way to solve this problem is to use the *Double-Checked Locking* pattern. 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)
  {
    static Singleton *instance_;

    // 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_;
};
```

The first thread that acquires 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 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.

[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 other will block[3]

```
if (Flag == FALSE)
  {
    Mutex.acquire ();
    if (Flag == FALSE)
      {
        perform Just Once Event;
        Flag = TRUE;
      }
    Mutex.release ();
  }
```

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

By adding a `Mutex` and a second conditional check, the canonical Singleton implementation can be made thread-safe with minimal locking overhead. 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.

4 Applicability

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

- The application is multi-threaded and has a Just Once Event (e.g., the initialization of a Singleton);
- The event is either dynamic and/or not idempotent;
- Multiple threads can potentially attempt to perform the event simultaneously.

5 Structure and Participants

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

- `Just Once Event (JOE)` – an event that only happens once. For instance, a Singleton is only initialized once. So, the calls to `new Singleton` in Section 3 are `JOE`s.[3]
- `Mutex` – A lock that serializes access to the critical section of the `Just Once Event`. Using the Singleton example from Section 3, the `Mutex` ensures that `new Singleton` only occurs once.

[3]But, not just your average.

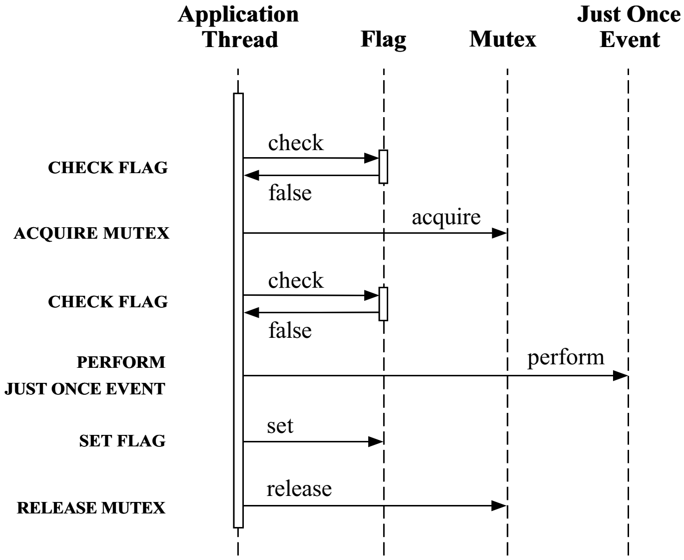

Figure 2: Participant collaborations in the Double-Checked Locking Pattern

- `Flag` – A flag that indicates whether the `JOE` has occurred. 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 structure that can be set without interruption. This issue is discussed in Section 9.
- `Application Thread` – This is the thread attempting to perform the `Just Once Event`. 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 makes sure that `Flag` is set, performs the `Just Once Event`, 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.

- *Prevents race conditions* – The second check of `Flag` ensures that the event is performed only once.

There is also a disadvantage of using this pattern:

- *Potential Deadlock* – there is potential for deadlock if circular references exist. For instance, if a Singleton constructor references other objects that in turn reference the original Singleton being constructed, this will lead to circular wait, and therefore deadlock. Changing the `Mutex` to a readers/writer lock will change the error from Deadlock to Livelock.

8 Implementation and Sample Code

The ACE toolkit [5] uses the Double-Checked Locking pattern in several library components. In order 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>
ACE_Singleton<TYPE, LOCK>
{
  // 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 instantiation:

```
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 as follows:

```
// 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...
}
```

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 a `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.

9 Variations

There are several variations to this pattern, which correspond to changes in various forces:

- *Separate Flag* – The implementation of the Double-Checked Locking pattern shown in Section 3.4 uses a pointer to `Singleton` as the `Flag`. 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 a scenario is possible on systems where memory addresses straddle alignment boundaries, thereby requiring two fetches from memory for each access. The use of a word-aligned integral `Flag` value can often alleviate this problem on multi-processors.

- *Volatile Flag* – Problems can occur with implementing Double-Checked Locking 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. One solution may be to declare `Flag` as a `volatile` data member in the `Singleton`. This ensures that the compiler will not place it into a register.

10 Known Uses

- *The ADAPTIVE Communication Environment (ACE)* – The Double-Checked Locking pattern is used extensively throughout the ACE object-oriented network programming toolkit [5].
- Andrew Birrell documents the Double-Checked Locking pattern in [1]. 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);
    top_ = 0;
  }
  stack_[top_++] = item;
  // ...
}
```

12 Acknowledgments

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

References

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

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

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

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

[5] 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.