1.1 Synchronization Patterns

The performance, responsiveness, and design of many applications can benefit from the use of concurrency. For example, different objects or functions in an application can run concurrently in different threads in order to block independently and simplify program structure. Moreover, if multiple processors are available, these threads can exploit true parallelism.

Concurrent programming is often more challenging than sequential programming, however. Many of these challenges arise from the need to serialize access to shared resources. In particular, threads that run concurrently can access the same other objects or variables simultaneously and potentially corrupt these objects' and variables' internal state. Therefore, code that should *not* execute concurrently in objects or functions can be serialized by a *critical section*. A critical section is a sequence of instructions that obeys the following invariant: while one thread or process is executing in the critical section, no other thread or process can execute in the same critical section [Tan95].

The conventional way to implement a critical section in object-oriented programs is to hard-code some type of lock object into a class or component. For instance, a mutual exclusion (*mutex*) object is a type of lock that must be acquired and released *serially*. Thus, if multiple threads attempt to acquire the mutex simultaneously, only one thread will succeed. The others must wait until the mutex is released [Tan92], after which all waiting threads will compete again for the lock. Other types of locks, such as semaphores and readers/writer locks, use a similar acquire/release protocol [McK95b].

Unfortunately, the following two drawbacks arise from adhering to such conventional programming techniques for critical sections.

- *Error-prone*. Explicitly acquiring a lock before entering a critical section and explicitly releasing a lock when exiting a critical section can be problematic. In particular, if a critical section has multiple return paths, the lock must be released explicitly in all of them. This use case is a common source of subtle programming errors, because it is easy to forget to release a lock in one of the return paths. If this happens, deadlock will occur when subsequent threads entering the critical section block indefinitely.

- *Inflexible and inefficient.* Depending on the environment in which an application runs, performance requirements may necessitate that different types of locks be used to implement critical sections. For example, when running an application that originally used a mutex on large-scale multi-processor platforms, performance may be increased by changing the locking mechanism to a readers/writer lock. This type of lock can improve performance by allowing multiple reader threads to access a shared resource in parallel.

A number of patterns and language-specific idioms have been discovered [McK95b] that provide solutions to these and other problems related to synchronizing concurrent objects and functions. This section presents four common synchronization patterns and idioms. The first two address fundamental aspects of locking, that is, effective lock acquisition/release and flexible variation of locking strategies:

- The *Scoped Locking* C++ idiom (229), applies Bjarne Stroustrup's 'Object-Construction-is-Resource-Acquisition' [Str98] technique to ensure that a lock is acquired automatically when control enters a scope and the lock is released automatically when control leaves the scope, regardless of the return path out of the scope.
- The *Strategized Lockin*g pattern (237) is a specialization of the Strategy pattern [GHJV95] that parameterizes the synchronization mechanisms used in a component to protect its critical sections from concurrent access.

When implemented in C++, the Strategized Locking pattern typically exploits the Scoped Locking idiom. The other two patterns described in this section help improve the robustness and efficiency of synchronization mechanisms in the following ways:

- The *Thread-Safe Interface* pattern (249) minimizes locking overhead and ensures that intra-component method calls do not incur 'self-deadlock' by trying to reacquire a lock that is already held by the component.
- The *Double-Checked Locking Optimization* pattern (257) reduces contention and synchronization overhead whenever critical sections of code must acquire locks in a thread-safe manner just once during program execution.

All four patterns and idioms are used to enhance the implementations of patterns presented in Section 2.6, *Concurrency Patterns*.

Scoped Locking

The *Scoped Locking* C++ idiom ensures that a lock is acquired when control enters a scope and the lock is released automatically when control leaves the scope.

Also Known As Synchronized Block, Object-Construction-is-Resource-Acquisition [Str98][1], Guard

Example Commercial Web servers typically maintain a 'hit count' component that records how many times each URL is accessed by clients over a period of time. To reduce latency, a Web server process does not maintain the hit counts in a file on disk but rather in a memory-resident table. Moreover, to increase throughput, Web server processes are often multi-threaded [HS98]. Therefore, public methods in the hit count component must be serialized to prevent threads from corrupting the state of its internal table as hit counts are updated concurrently.

One way to serialize access to a hit count component is to acquire and release a lock in each public method explicitly. For instance, the following example uses the Thread_Mutex defined in the Wrapper Facade pattern (25) to serialize access to critical sections in the methods of a C++ Hit_Counter class that implements a Web server's hit count component.

```
class Hit_Counter {
public:
    // Increment the hit count for a URL pathname.
    int increment (const char *pathname) {
        // Acquire lock to enter critical section.
        lock_.acquire ();
        Table_Entry *entry = lookup_or_create (pathname);
        if (entry == 0) {
            // Something's gone wrong, so bail out.
            lock_.release ();
```

1. The Scoped Locking idiom is a specialization of Stroustrup's 'Object-Construction-is-Resource-Acquisition' idiom [Str98] that is applied to locking. We include this idiom here to keep the book self-contained and to illustrate how Stroustrup's idiom can be applied to concurrent programs.

```
                // Return a ‘failure’ value.
                return -1;
            }
            else
                // Increment the hit count for this pathname.
                entry->increment_hit_count ();
            // Release lock to leave critical section.
            lock_.release ();
            // ...
        }
        // Other public methods omitted.
    private:
        // Lookup the table entry that maintains the hit count
        // associated with <pathname>, creating the entry if
        // it doesn’t exist.
        Table_Entry *lookup_or_create (const char *pathname);

        // Serialize access to the critical section.
        Thread_Mutex lock_;
    };
```

Although the C++ code example shown above works, the `Hit_Count` implementation is unnecessarily hard to develop and maintain. For instance, maintenance programmers may forget to release the `lock_` on all return paths out of the `increment()` method. Moreover, because the implementation is not exception-safe, `lock_` will not be released if a later version throws an exception or calls a helper method that throws an exception [Mue96]. The first source of errors could occur if a maintenance programmer revises the else branch of the `increment()` method to check for a new failure condition, as follows:

```
    else if (entry->increment_hit_count () == -1)
        return -1; // Return a ‘failure’ value.
```

Likewise, the `lookup_or_create()` method also might be changed to throw an exception if an error occurs. Unfortunately, both of these modifications will cause the `increment()` method to return to its caller *without* releasing the `lock_`. If the `lock_` is not released, however, the Web server process will hang when other threads block indefinitely trying to acquire the `lock_`. Moreover, if these error cases occur infrequently, the problems with this code may not show up during system testing.

Context A concurrent application containing shared resources that are manipulated concurrently by multiple threads.

Problem Locks should always be acquired and released properly when control enters and leaves critical sections, respectively. If programmers must acquire and release locks explicitly, however, it is hard to ensure the locks are released in all paths through the code. For instance, in C++ control can leave a scope due to return, break, continue, or goto statements, as well as from an unhandled exception being propagated out of the scope.

Solution Define a guard class whose constructor automatically acquires a lock when control enters a scope and whose destructor automatically releases the lock when control leaves the scope. Instantiate instances of the guard class to acquire/release locks in method or block scopes that define critical sections.

Implementation The implementation of the Scoped Locking idiom is straightforward.

Define a guard class that acquires and releases a particular type of lock automatically within a method or block scope. The constructor of the guard class stores a pointer or reference to the lock and then acquires the lock before the critical section is entered. The destructor of this class uses the pointer or reference stored by the constructor to release the lock automatically when leaving the scope of the critical section. Due to the semantics of C++ destructors, guarded locks will be released even if C++ exceptions are thrown from within the critical section.

➥ The following class illustrates a guard designed for the `Thread_Mutex` developed in the implementation section of the Wrapper Facade pattern (25):

```
class Thread_Mutex_Guard {
public:
     // Store a pointer to the lock and acquire the lock.
     Thread_Mutex_Guard (Thread_Mutex &lock)
          : lock_ (&lock) { owner_= lock_->acquire (); }

     // Release the lock when the guard goes out of scope.
     ~Thread_Mutex_Guard (void) {
          // Only release the lock if it was acquired
          // successfully, i.e., -1 indicates that
          // <acquire> failed..
          if (owner_ != -1)
               lock_->release ();
     }
private:
     // Pointer to the lock we’re managing.
     Thread_Mutex *lock_;

     // Records if lock_ is currently held by this object.
     int owner_;
};                                                          ❑
```

A pointer or reference to a lock, rather than a lock object, should be used in a guard class implementation. This design prevents copying or assigning a lock, which is erroneous as discussed in the Wrapper Facade pattern (25).

In addition, it is useful to add a flag, such as the `owner_` flag in the `Thread_Mutex_Guard` example above, that indicates whether or not a guard acquired the lock successfully. The flag can also indicate failures that arise from 'order of initialization bugs' if static/global locks are used erroneously [LGS99]. By checking this flag in the guard's destructor, a subtle run-time error can be avoid that would otherwise occur if the lock was released even although it was not held by the guard.

Example Resolved The following C++ code illustrates how to apply the Scoped Locking idiom to resolve the original problems with the `Hit_Counter` class in our multi-threaded Web server.

```
class Hit_Counter {
public:
     // Increment the hit count for a URL pathname.
     int increment (const char *pathname) {
          // Use the Scoped Locking idiom to
          // automatically acquire and release the
          // <lock_>.
          Thread_Mutex_Guard guard (lock_);
          Table_Entry *entry = lookup_or_create (pathname);
          if (entry == 0)
               // Something’s gone wrong, so bail out.
               return -1;
               // Destructor releases <lock_>.
          else
               // Increment the hit count for this pathname.
               entry->increment_hit_count ();

          // Destructor for guard releases <lock_>.
     }
     // Other public methods omitted.
private:
     // Serialize access to the critical section.
     Thread_Mutex lock_;
     // ...
};
```

In this solution the guard ensures that the lock_ is acquired and released automatically as control enters and leaves the increment() method, respectively.

Variants *Explicit accessors*. One drawback with the Thread_Mutex_Guard interface described in the *Implementation* section is that it is not possible to release the lock explicitly without leaving the method or block scope. To handle these use cases, a variant of the Scoped Locking idiom can be defined to provide explicit accessors to the underlying lock.

➥ For instance, the following code fragment illustrates a use case where the lock could be released twice, depending on whether the condition in the if statement evaluates to true:

```
{
     Thread_Mutex_Guard guard (&lock);
     // Do some work ...
     if (/* a certain condition holds */)
          lock->release ()
     // Do some more work ...
     // Leave the scope.
}
```

To prevent this erroneous use case, we do not operate on the lock directly. Instead, a pair of explicit accessor methods are defined in the Thread_Mutex_Guard class, as follows:

```
class Thread_Mutex_Guard {
public:
     // Store a pointer to the lock and acquire the lock.
     Thread_Mutex_Guard (Thread_Mutex &lock)
          : lock_ (&lock) {
          acquire ();
     }

     int acquire (void) {
          // If <acquire> fails <owner_> will equal -1;
          owner_ = lock_->acquire ();
     }
```

```
        int release (void) {
            // Only release the lock if it was acquired
            // successfully and we haven't released it
            // already!
            if (owner_ != -1) {
                owner_ = -1;
                return lock_->release ();
            }
            else
                return 0;
        }

        // Release the lock when the guard goes out of scope.
        ~Thread_Mutex_Guard (void) {
            release ();
        }

    private:
        // Pointer to the lock we're managing.
        Thread_Mutex *lock_;

        // Records if the lock is held by this object.
        int owner_;
    };
```

This variant exposes `acquire()` and `release()` methods that keep track of whether the lock has been released already, and if so, it does not release the lock in `guard`'s destructor. Therefore, the following code will work correctly:

```
    {
        Thread_Mutex_Guard guard (&lock);
        // Do some work ...
        if (/* a certain condition holds */)
            guard.release ();
        // Do some more work ...
        // Leave the scope.
    }
```
❏

Strategized Scoped Locking. Defining a different guard for each type of lock is tedious, error-prone, and excessive, because it may increase the memory footprint of applications or components. Therefore, a common variant of the Scoped Locking idiom is to apply either the parameterized type or polymorphic version of the Strategized Locking pattern (237).

Known Uses **Booch Components**. The Booch Components [BV93] were one of the first C++ class libraries to use the Scoped Locking idiom for multi-threaded C++ programs.

Adaptive Communication Environment (ACE) [Sch97]. The Scoped Locking idiom is used extensively throughout the ACE object-oriented network programming toolkit.

Threads.h++. The Rogue Wave Threads.h++ library defines a set of guard classes that are modeled after the ACE Scoped Locking designs.

Java defines a programming feature called a synchronized block that implements the Scoped Locking idiom in the language.

Consequences There are two **benefits** of using the Scoped Locking idiom:

Increased robustness. By applying this idiom, locks will be acquired/released automatically when control enters/leaves critical sections defined by C++ method and block scopes. This idiom increases the robustness of concurrent applications by eliminating common programming errors related to synchronization and multi-threading.

Decreased maintenance effort. If parameterized types or polymorphism is used to implement the guard or lock classes, it is straightforward to add enhancements and bug fixes. In such cases, there is only one implementation, rather than a separate implementation for each type of guard, which eliminates version-skew.

There are two **liabilities** of applying the Scoped Locking idiom to concurrent applications and components:

Potential for deadlock when used recursively. If a method that uses the Scoped Locking idiom calls itself recursively 'self-deadlock' will occur if the lock is not a 'recursive' mutex. The Thread-Safe Interface pattern (249) describes a technique that avoids this problem. This pattern ensures that only interface methods apply the Scoped Locking idiom, whereas implementation methods do not apply this idiom.

Limitations with language-specific semantics. Because the Scoped Locking idiom is based on a C++ language feature, it may not be integrated with operating system-specific system calls. Therefore, locks may not be released automatically when threads or processes abort or exit inside of a guarded critical section.

➥ For instance, the following modification to `increment()` will prevent the Scoped Locking idiom from working:

```
Thread_Mutex_Guard guard (&lock_);
Table_Entry *entry = lookup_or_create (pathname);
if (entry == 0)
    // Something's gone wrong, so exit the thread.
    thread_exit ();
    // Destructor will not be called so the
    // <lock_> will not be released!
```
❑

As a general rule, therefore, it is inappropriate to abort or exit a thread or process within a component. Instead, some type of exception handling mechanism or error-propagation patterns should be used [Mue96].

Excessive compiler warnings. The common use case of the Scoped Locking idiom defines a guard object that is not used explicitly within the scope because its destructor releases the lock implicitly. Unfortunately, some C++ compilers print “statement has no effect” warnings when guards are defined but not used explicitly within a scope. At best, these warnings are distracting. At worst, they encourage developers to disable certain compiler warnings, which may mask other warnings that indicate actual problems with the code. An effective way to handle this problem is to define a macro that can eliminate the warnings without generating additional code.

➥ For instance, the following macro is defined in ACE [Sch97]:

```
#define UNUSED_ARG(arg) { if (&arg) /* null */; }
```

This macro can be placed after a guard to keep C++ compilers from generating spurious warnings, as follows:

```
{ // New scope.
    Thread_Mutex_Guard guard (lock_);
    UNUSED_ARG (guard);
    // ...
```
❑

See Also The Scoped Locking idiom is a special-case of a more general C++ idiom [Str98] where a constructor acquires a resource and a destructor releases the resource when a scope is entered and exited, respectively. When this idiom is applied to concurrent applications, the resource that is acquired and released is some type of lock.

Credits Thanks to Brad Appleton for comments on the Scoped Locking idiom.

Strategized Locking

The *Strategized Lockin*g pattern parameterizes the synchronization mechanisms in a component that protect its critical sections from concurrent access.

Example A key component used to implement a high-performance Web server is a *file cache*, which maps URL pathnames to memory-mapped files [HS98]. When a client requests a URL pathname that is already cached, the Web server can transfer the contents of the memory-mapped file to the client directly. Thus, the Web server need not access slower secondary storage via multiple `read()` and `write()` operations.

A file cache implementation for a portable high-performance Web server should run efficiently on various multi-threaded and single-threaded operating systems. One way of achieving this portability is to develop the following file cache classes:

```
// A single-threaded file cache implementation.
class File_Cache_ST {
public:
   // Return a pointer to the memory-mapped file
   // associated with <pathname>.
   const char *lookup (const char *pathname) {
       // No locking required because we are
       // single-threaded.


       // ... look up the file in the cache, mapping it
       // into memory if it is not currently in the cache.
       return file_pointer;
   }
   // ...
private:
   //File cache implementation...

   // No lock required because we are
   // single-threaded.
};
```

```
// A multi-threaded file cache implementation.
class File_Cache_Thread_Mutex {
public:
   // Return a pointer to the memory-mapped file
   // associated with <pathname>.
   const char *lookup (const char *pathname) {
       // Use the Scoped Locking idiom to serialize
       // access to the file cache.
       Guard<Thread_Mutex> guard (lock_);

       // ... look up the file in the cache, mapping it
       // into memory if it is not currently in the cache.
       return file_pointer;
   }
   // ...
private:
   //File cache implementation...

   // Synchronization strategy.
   Thread_Mutex lock_;
};
```

These two implementations form part of a component family whose classes differ only in their synchronization strategy. One component in the family, represented in our example by the class `File_Cache_ST`, implements a single-threaded file cache that uses no locking. The other component in the family, represented by class `File_Cache_Thread_Mutex`, implements a file cache that uses a mutex to serialize multiple threads accessing the cache concurrently. Maintaining multiple separate implementations of similar file cache components can be tedious, however. In particular, future enhancements and fixes, such as optimizing algorithms or removing defects, must be carried out consistently in each component implementation.

Context An application or system where reusable components must run efficiently in a variety of different concurrency use cases.

Problem Components that run in multi-threaded environments must protect critical sections of their code from concurrent access by multiple clients. When integrating synchronization mechanisms with component functionality the following two *forces* must be resolved:

- Different applications may require different synchronization strategies, such as mutexes, readers/writer locks, or semaphores [McK95b]. Therefore, it should be possible to customize a component's synchronization mechanisms according to the requirements of particular applications.

  ➥ In our example, the synchronization strategy is hard-coded. Therefore, a completely new class must be written to support a file cache implementation that uses a readers/writer lock instead of a thread mutex to increase performance on large-scale multi-processor platforms. It is time-consuming, however, to customize an existing file cache class to support new, more efficient synchronization strategies. ❏

- It should be straightforward to add new enhancements and bug fixes. In particular, to avoid version-skew, changes should apply consistently and automatically to all component family members.

  ➥ If there are multiple copies of the same basic file cache component, version-skew is likely to occur because changes to one component may be applied inconsistently to other component implementations. Moreover, applying each change manually is error-prone and non-scalable. ❏

Solution Parameterize a component's synchronization aspects by making them 'pluggable' types. Each type objectifies a particular synchronization strategy, such as a mutex, readers/writer lock, or semaphore. Define instances of these pluggable types as objects contained within a component, which can use these objects to synchronize its method implementations efficiently.

Implementation The Strategized Locking pattern can be implemented using the following steps.

1 *Define the component interface and implementation.* The purpose of this step is to define the component's interface and implement its methods efficiently without concern for synchronization aspects.

➥ The following class defines the `File_Cache` interface and implementation.

```
class File_Cache {
public:
    const char *lookup (const char *pathname);
    // ...

private:
    // data members and private methods go here...
};
```
❏

2 *Determine which component synchronization aspects can vary* and update the component interface and implementation to strategize these aspects. Many reusable components have relatively simple synchronization aspects that can be implemented using common locking strategies, such as mutexes and semaphores. These types of synchronization aspects can be strategized in a uniform manner.

There are two ways to strategize locking mechanisms: *polymorphism* and *parameterized types*. In general, parameterized types should be used when the locking strategy is known at compile-time. Likewise, polymorphism should be used when the locking strategy is not known until run-time. As usual, the trade-off is between the efficient run-time performance of parameterized types versus the potential for run-time extensibility with polymorphism.

Once synchronization mechanisms are strategized, components can use these mechanisms either by explicitly acquiring/releasing a lock when entering/leaving a critical section or by applying the Scoped Locking idiom (205) to acquire/release the lock automatically. If the Scoped Locking idiom (237) is applied, this implementation step involves the following three sub-steps.

2.1 *Define an abstract interface for the locking mechanisms.* To configure a component with different locking mechanisms, all concrete implementations of these mechanisms must employ an abstract interface with common signatures for acquiring and releasing locks based on either polymorphism or parameterized types.

- *Polymorphism.* In this approach, define an polymorphic lock object that contains dynamically bound `acquire()` and `release()` methods. Derive all concrete locks, such as mutexes or semaphores, from this base class and override its methods to define a concrete locking strategy, as outlined in *Implementation* step 4.

  ➥ To implement a polymorphic lock object for our file cache example, we first define an abstract locking class with virtual `acquire()` and `release()` methods, as follows:

  ```
  class Lock {
  public:
       // Acquire the lock.
       virtual int acquire (void) = 0;
       // Release the lock.
       virtual int release (void) = 0;

       // ...
  };
  ```
  ❏

- *Parameterized types.* In this approach we must ensure that all concrete locks employ the same signature for acquiring and releasing locks. The usual way to ensure this is to implement the concrete locks using the Wrapper Facade pattern (25).

2.2 *Use the Scoped Locking Idiom (229) to define a guard class that is strategized by its synchronization aspect.* This design follows the Strategy Pattern [GHJV95], where the guard class serves as the context holding a particular lock and the concrete locks provide the strategies. The Scoped Locking idiom can be implemented either with polymorphism or with parameterized types.

- *Polymorphism.* In this approach, pass a polymorphic lock object to the guard's constructor and define a instance of this lock object as a private data member. To acquire and release the lock with which it is configured, the implementation of the guard class can the polymorphic `Lock` base class defined in the previous sub-step.

➥ A `Guard` class that controls a polymorphic lock could be defined as follows:

```
class Guard {
public:
     // Store a pointer to the lock and acquire the lock.
     Guard (Lock &lock): lock_ (&lock) {
          owner_ = lock_->acquire ();
     }
     // Release the lock when the guard goes out of scope.
     ~Guard (void) {
          // Only release the lock if it was acquired
          // successfully, i.e., -1 indicates <acquire>
          // failed.
          if (owner_ != -1) lock_->release ();
     }

private:
     // Pointer to the lock we're managing.
     Lock *lock_;
     // Records if the lock was acquired successfully.
     int owner_;
};
```
❑

- *Parameterized types*. In this approach, define a template guard class that is parameterized by the type of lock that will be acquired and released automatically.

➥ The following illustrates a `Guard` class that is strategized by a LOCK template parameter:

```
template <class LOCK> class Guard {
public:
     // Store a pointer to the lock and acquire the lock.
     Guard (LOCK &lock): lock_ (&lock) {
          owner_ = lock_->acquire ();
     }
     // Release the lock when the guard goes out of scope.
     ~Guard (void) {
          // Only release the lock if it was acquired
          // successfully, i.e., -1 indicates <acquire>
          // failed.
          if (owner_ != -1) lock_->release ();
     }

private:
     // Pointer to the lock we're managing.
     LOCK *lock_;
     // Records if the lock is held by this object.
     int owner_;
};
```
❑

2.3 *Update the component interface and implementation*. The guard class is used in the implementation of the component to protect critical sections within its methods, according to the Scoped Locking Idiom (229). Depending on whether the polymorphic or parameterized type approach is used, the lock can be passed to the component either as a parameter in its constructor or by adding a lock template parameter to the component declaration. In either case, the lock passed to a component must follow the signature expected by the guard class, as discussed in the previous sub-step.

➥ The version of our file cache component that takes a polymorphic lock can be modified as follows:

```
class File_Cache {
public:
     // Constructor
     File_Cache (Lock &l)
          : lock_ (l) {}

     // A method.
     const char *lookup (const char *pathname) {
          // Use the Scoped Locking idiom to
          // acquire and release the <lock_> automatically.
          Guard guard (lock_);
          // Implement the lookup() method.
     }

     // ...
private:
     // The polymorphic strategized locking object.
     Lock *lock_;

     // Other File_Cache data members and methods go
     // here...
};
```

Likewise, the templatized version of the file cache can be defined as follows:

```
template <class LOCK> class File_Cache {
public:
     // A method.
     const char *lookup (const char *pathname) {
          // Use the Scoped Locking idiom to
          // acquire and release the <lock_> automatically.
          Guard<LOCK> guard (lock_);
          // Implement the lookup() method.
     }
     // ...
private:
     // The parameterized type strategized locking object.
     LOCK lock_;

     // Other File_Cache data members and methods go
     // here...
};
```

If your C++ compiler supports default template arguments, it may be useful to add a default LOCK to handle the most common use case. For instance, we can make the default LOCK be a readers/writer lock, as follows:

```
template <class LOCK = RW_Lock> class File_Cache
{ /* ... */ }                                          ❏
```

3 *Revise the component implementation to avoid deadlock and remove unnecessary locking overhead.* If intra-component method invocations occur developers must design their component implementation carefully to avoid self-deadlock and unnecessary synchronization overhead. A straightforward technique that prevents these problems is the Thread-Safe Interface pattern (249).

4 *Define a family of locking strategies with uniform interfaces* that can support various application-specific concurrency use cases. Common locking strategies include recursive and non-recursive mutexes, readers/writer locks, semaphores, and file locks. If the appropriate synchronization mechanism does not already exist, or the existing mechanism has an incompatible interface, use the Wrapper Facade pattern (25) to implement or adapt it to conform to the signatures expected by the component's synchronization aspects.

➥ In addition to the Thread_Mutex defined in the Wrapper Facade pattern (25), a surprisingly useful locking strategy is the Null_Mutex. This class defines an efficient locking strategy for single-threaded applications and components, as follows:

```
class Null_Mutex {
public:
     Null_Mutex (void) { }
     ~Null_Mutex (void) { }
     int acquire (void) { return 0; }
     int release (void) { return 0; }
};
```

All methods in Null_Mutex are empty inlined functions that can be removed completely by optimizing compilers. This class is an example of the Null Object pattern [PLoPD3], which simplifies applications by defining a ‘no-op’ placeholder that does not require conditional statements in the component’s implementation. An example use of `Null_Mutex` and other locking strategies appears in the *Example Resolved* section below. ❑

When applying the polymorphic approach, implement the locking strategies as subclasses of the abstract class `Lock`, as discussed in *Implementation* step 2. These subclasses can either implement a particular locking mechanism directly or they can wrap an existing non-polymorphic locking mechanism.

➥ For instance, the following class wraps the `Thread_Mutex` from the *Implementation* section of the Wrapper Facade pattern (25), thereby connecting it to our polymorphic lock hierarchy, as follows:

```
class Thread_Mutex_Lock : public Lock {
public:
     // Acquire the lock.
     virtual int acquire (void) {return lock_.acquire ();}

     // Release the lock.
     virtual int release (void) {return lock_.release ();}
private:
     // Concrete lock type.
     Thread_Mutex lock_;
};
```
❑

Example Resolved The following illustrates how to apply the parameterized type form of the Strategized Locking pattern to implement a Web server file cache that is tuned for various single-threaded and multi-threaded concurrency use cases.

- *Single-threaded file cache.*

```
typedef File_Cache<Null_Mutex> FILE_CACHE;
```

- *Multi-threaded file cache using a thread mutex.*

```
typedef File_Cache<Thread_Mutex> FILE_CACHE;
```

- *Multi-threaded file cache using a readers/writer lock.*

```
typedef File_Cache<RW_Lock> FILE_CACHE;
```

- *Multi-threaded file cache using a C++ compiler that supports default template parameters,* with the lock defaulting to a readers/writer lock.

```
typedef File_Cache<> FILE_CACHE;
```

Note how in each of these configurations the File_Cache interface and implementation require no changes. This flexibility stems from the Strategized Locking pattern, which abstracts synchronization aspects into a 'pluggable' parameterized types. Moreover, the details of locking have been strategized via a typedef. Therefore, it is straightforward to define a FILE_CACHE object that does not expose synchronization aspects to applications, as follows:

```
FILE_CACHE file_cache;
```

Variants. *Bridge strategy*. Unfortunately, configuring the polymorphic file cache in *Implementation* step 2.3 differed from configuring the templatized file cache because a polymorphic lock implemented as a pointer cannot be passed as a parameter to the templatized File_Cache and Guard classes. Instead, we need a 'real' object, rather than a pointer to an object

Fortunately, the Bridge pattern [GHJV95] can help us implement a family of locking strategies that is uniformly applicable for *both* polymorphic and parameterized type approaches. To apply this bridge strategy variant, we simply define an additional abstraction class that encapsulates, and can be configured with, a polymorphic lock. An instance of this abstraction class can then be passed uniformly to both polymorphic and templatized components.

➥ Consider the hierarchy of polymorphic locks defined in *Implementation* step 2 as being a Bridge implementor class hierarchy. The following abstraction class then can be used to encapsulate this hierarchy:

```
class Lock_Abstraction {
public:
     // Constructor stores a reference to the base class.
     Lock_Abstraction (Lock &l): lock_ (l) {};

     // Acquire the lock by forwarding to the
     // polymorphic acquire() method.
     int acquire (void) { return lock_.acquire (); }

     // Release the lock by forwarding to the
     // polymorphic release() method.
     int release (void) { return lock_.release (); }
private:
     // Maintain a reference to the polymorphic lock.
     Lock &lock_;
};
```

Note how this design allows us to initialize both our polymorphic and parameterized `File_Cache` and `Guard` classes with a single `Lock_Abstraction` class that can be configured with a concrete lock from our hierarchy of locking mechanisms. ❑

As a result of using this variation of Strategized Locking, the family of locking mechanisms becomes more reusable and easier to apply across applications. Be aware, however, that while this scheme is flexible, it is also more complicated to implement and therefore should be used with care.

Known Uses

Aspect-Oriented Programming (AOP) [KLM+97]. AOP is a general methodology for systematically strategizing aspects that vary in applications and components.

Adaptive Communication Environment (ACE) [Sch97]. The Strategized Locking pattern is used extensively throughout the ACE object-oriented network programming toolkit. Most ACE containers and collection components can be strategized by synchronization aspects using parameterized types.

Booch Components. The Booch Components [BV93] were one of the first C++ class libraries to parameterize locking strategizes with templates.

ATL Wizards. The Microsoft ATL Wizard in Visual Studio uses the parameterized type implementation of Strategized Locking, complete with default template parameters. In addition, it implements a class similar to the `Null_Mutex`. Thus, if a COM class is implemented as a single-threaded apartment a no-op lock class is used, whereas in multi-threaded apartments a 'real' recursive mutex is used.

Consequences

There are three **benefits** of applying the Strategized Locking pattern to reusable components:

Enhanced flexibility and customization. Because the synchronization aspects of components are strategized, it is straightforward to configure and customize a component for particular concurrency use cases. If no suitable locking strategy is available for a new concurrency use case, the family of locking strategies can be extended without affecting existing code.

Decreased maintenance effort for components. It is straightforward to add enhancements and bug fixes to a component because there is only one implementation, rather than a separate implementation for each concurrency use case. This centralization of concerns helps minimize version-skew.

Improved reuse. A particular component becomes less dependent on specific synchronization mechanisms. Thus, it becomes more reusable because its locking strategies can be configured.

There is a **liability** of applying the Strategized Locking pattern to reusable components:

Obtrusive locking. If templates are used to parameterize locking aspects this will expose the locking strategies to application code. This design can be obtrusive, particularly for compilers that do not support templates efficiently or correctly. One way to avoid this problem is to apply the polymorphic approach to strategize component locking behavior.

Overengineering: Externalizing the locking mechanism by placing it in a component's interface may actually provide *too* much flexibility for certain use cases. For instance, inexperienced developers may try to parameterize a component with the wrong type of lock, which can result in confusing compile-time or run-time behavior. Likewise, only a single type of synchronization mechanism may ever be used for a particular type of component, in which case the power of the Strategized Locking pattern is unnecessary. In general, this pattern is most effective when practical experience reveals that a component's behavior is orthogonal to its locking strategy, and that locking strategies can indeed vary in semantically meaningful and efficient ways in different use cases.

See Also The Scoped Locking idiom (237) uses Strategized Locking to parameterize various synchronization strategies into its guard class.

The main synchronization mechanism in Java is the monitor object (299). In particular, Java does not expose 'conventional' concurrency control mechanisms, such as mutexes and semaphores, to application developers. Therefore, the Strategized Locking pattern need not be applied to Java directly. It is possible, however, to implement different concurrency primitives, such as mutexes, semaphores, and readers/writer locks in Java. The implementation of these primitives can then be used as locking strategies to support various application-specific concurrency uses cases. Due to Java's lack of parameterized types, only the polymorphic approach of Strategized Locking pattern could be used to configure different synchronization strategies. In this case, Java implementations of this pattern will be similar to the C++ implementations described in this pattern.

Credits Thanks to Brad Appleton for comments on this pattern and Prashant Jain for his contribution to how this pattern applies to Java.

Thread-Safe Interface

The *Thread-Safe Interface* pattern ensures that intra-component method calls avoid self-deadlock and minimize locking overhead in concurrent applications whose components maintain shared state.

Example When designing thread-safe components, developers must be careful to avoid self-deadlock and unnecessary locking overhead when intra-component method calls occur. For example, consider a more complete implementation of the File_Cache component that was outlined in the Strategized Locking pattern (237):

```
template <class LOCK> class File_Cache {
public:
    // Return a pointer to the memory-mapped file
    // associated with <pathname>, adding
    // it to the cache if it doesn't exist.
    const char *lookup (const char *pathname) {
        // Use the Scoped Locking idiom to
        // automatically acquire and release the <lock_>.
        Guard<LOCK> guard (lock_);
        const char *file_pointer =
            check_cache (pathname);
        if (file_pointer == 0) {
            // Insert the <pathname> into the cache.
            // Note the intra-class <insert> call.
            insert (pathname);
            file_pointer = check_cache (pathname);
        }
        return file_pointer;
    }
    // Add <pathname> to the cache.
    void insert (const char *pathname) {
        // Use the Scoped Locking idiom to
        // automatically acquire and release the <lock_>.
        Guard<LOCK> guard (lock_);
        // ... insert <pathname> into the cache...
    }
private:
    // The strategized locking object.
    LOCK lock_;

    const char *check_cache (const char *);
    // ... other private methods and data omitted...
};
```

This implementation of `File_Cache` works efficiently only when strategized by a 'null' lock, such as the `Null_Mutex` described in the Strategized Locking pattern (237). If, however, the `File_Cache` implementation is strategized with a recursive mutex[2] it will incur unnecessary overhead when it reacquires the mutex in the `insert()` method. Even worse, if it is strategized with a non-recursive mutex, when the `lookup()` method calls the `insert()` method the code will 'self-deadlock'. This self-deadlock occurs because `insert()` tries to reacquire the `LOCK` that has already been acquired by `lookup()`.

Thus, it is effectively counter-productive to apply the Strategized Locking pattern to the implementation of `File_Cache` shown above because there are so many restrictions and subtle problems that can arise. Yet, the `File_Cache` abstraction can still benefit from the flexibility and customization provided by Strategized Locking.

Context Components in a multi-threaded application that contain intra-component method calls.

Problem Multi-threaded components typically contain multiple publically accessible interface methods and private implementation methods that can alter component state. To prevent race conditions, a lock internal to the component can be used to serialize interface method invocations that access its state. Although this design works fine if each method is self-contained, component methods often call each other to carry out their computations. If this occurs, the following *forces* will be unresolved in multi-threaded components that use improper intra-component method invocation designs:

- Thread-safe components should be designed to avoid 'self-deadlock.' Self-deadlock can occur if one component method acquires a non-recursive lock in the component and then calls a different component method that tries to reacquire the same lock.
- Thread-safe components should be designed to incur only minimal locking overhead, for example to prevent race conditions on component state. If a recursive component lock is selected to avoid the self-deadlock problem outlined above, however, unnecessary

2. A recursive mutex can be reacquired by the thread that has locked the mutex without blocking the thread.

overhead will be incurred to acquire and release the lock multiple times across intra-component method calls.

Solution Structure all components that process intra-component method invocations according to the following two design conventions:

- *Interface methods 'check'*. All interface methods, such as C++ public methods, should only acquire/release component lock(s), thereby performing synchronization checks at the 'border' of the component. After the lock is acquired, the interface method should immediately forward to an implementation method, which performs the actual method functionality. Once the implementation method returns, the Interface method should release the lock(s) before returning control to the caller.
- *Implementation methods 'trust'*. Implementation methods, such as C++ private and protected methods, should only perform work when called by interface methods. Thus, they should trust that they are called with the necessary lock(s) held and should therefore never acquire/release lock(s). Moreover, implementation methods should never call 'up' to interface methods because these methods acquire lock(s).

By following these conventions, even components designed to use the Strategized Locking pattern (237) can avoid self-deadlock and minimize locking overhead.

Implementation The Thread-Safe Interface pattern can be implemented using the following steps:

1 *Determine the interface and corresponding implementation methods*. The interface methods define the public API of the component. For each interface method, define a corresponding implementation method.

➥ The interface and corresponding implementation methods for the `File_Cache` are defined as follows:

```
template <class LOCK> class File_Cache {
public:
    // The following two interface methods just
    // acquire/release the <LOCK> and forward to
    // their corresponding implementation methods.
    const char *lookup (const char *pathname);
    void insert (const char *pathname);
private:
     // The following two implementation methods do not
     // acquire/release locks and perform the actual work
     // associated with managing the <File_Cache>.
     const char *lookup_i (const char *pathname);
     void insert_i (const char *pathname);

     // ... Other implementation methods omitted ...
};
```
❑

2 *Program the interface and implementation methods*. The bodies of the interface and implementation methods are programmed according to the design conventions described in the *Solution* section.

➥ The following implementation of the `File_Cache` class applies the Thread-Safe Interface pattern to minimize locking overhead and prevent self-deadlock in the interface and implementation methods:

```
template <class LOCK> class File_Cache {
public:
     // Return a pointer to the memory-mapped file
     // associated with <pathname>, adding
     // it to the cache if it doesn't exist.
     const char *lookup (const char *pathname) {
          // Use the Scoped Locking idiom to
          // automatically acquire and release the <lock_>.
          Guard<LOCK> guard (lock_);
          return lookup_i (pathname);
     }

     // Add <pathname> to the file cache.
     void insert (const char *pathname) {
          // Use the Scoped Locking idiom to
          // automatically acquire and release the <lock_>.
          Guard<LOCK> guard (lock_);
          insert_i (pathname);
     }

private:
     LOCK lock_; // The strategized locking object.

     // The following implementation methods do not
     // acquire or release <lock_> and perform their
     // work without calling any interface methods.
     const char *lookup_i (const char *pathname) {
          const char *file_pointer =
               check_cache_i (pathname);

          if (file_pointer == 0) {
               // If <pathname> isn't in the cache then
```

```
                    // insert it and look it up again.
                    insert_i (pathname);
                    file_pointer = check_cache_i (pathname);
                    // The calls to implementation methods
                    // <insert_i> and <check_cache_i> assume
                    // that the lock is held and perform work.
               }
               return file_pointer;
          }
          const char *check_cache_i (const char *) { /* ... */ }
          void insert_i (const char *) { /* ... */ }

          // ... other private methods and data omitted ...
     };
```

Variants *Thread-Safe Facade*. This variant is applicable if access to a whole subsystem or large component must be synchronized. A facade [GHJV95] can be introduced to serve as an entry point for all client requests. The facade's methods correspond to the interface methods. The classes that belong to the subsystem or component provide the implementation methods. If these classes already have their own internal concurrency strategies they may need to be 'refactored' to avoid the *nested monitor lockout* problem [JS97a].

The nested monitor problem occurs when a thread acquires object *X*'s monitor lock, without relinquishing the lock already held on monitor *Y*, thereby preventing a second thread from acquiring the monitor lock for *Y*. This can lead to deadlock because after acquiring monitor *X*, the first thread may wait for a condition to become true that can only change as a result of actions by the second thread after it has acquired monitor *Y*.

Note that if the subsystem or component cannot be modified, for instance if it is a 3rd party product or legacy system, it may not be possible to refactor the code properly to avoid nested monitor lockouts. In this case, Thread-Safe Facade should not be applied.

Thread-Safe Wrapper Facade. This variant helps to provide synchronized access to a class or function API that cannot be modified. A wrapper facade (25) provides the interface methods, which encapsulate the corresponding implementation calls on the class or function API with actions to acquire and release a lock. Thus, the wrapper facade provides a synchronization veneer that serializes access to the methods of the class or function API. As with the Thread-Safe Facade, if the class or function API has its own concurrency strategies, care must be taken when applying the wrapper facade to avoid the nested monitor lockout problem.

Known Uses **Adaptive Communication Environment** (ACE) [Sch97]. The Thread-Safe Interface pattern is used extensively throughout the ACE object-oriented network programming toolkit.

Consequences There are two **benefits** of applying the Thread-Safe Interface pattern to multi-threaded components:

Increased robustness. This pattern ensures that self-deadlock does not occur due to intra-component method calls.

Enhanced performance. This pattern ensures that no unnecessary locks are acquired or released.

However, there are two **liabilities** when applying the Thread-Safe Interface pattern to multi-threaded components:

Additional indirection and extra methods. Each interface method requires at least one implementation method, which increases the footprint of the component and may also add an extra level of method-call indirection for each invocation. One way to minimize this overhead is to inline the interface and/or the implementation method.

Potential deadlock. The Thread-Safe Interface pattern does not completely resolve the problem of self-deadlock by itself. For instance, consider a client that calls an interface method on component *A*, which then delegates to an implementation method that calls an interface method on another component *B*. If the implementation of component *B*'s method calls back on an interface method of component *A*, deadlock will occur when trying to reacquire the lock that was acquired by the first call in this chain. The *Variants* section describes several solutions to this problem.

See Also The Thread-Safe Interface pattern is related to the Decorator pattern [GHJV95], which extends an object transparently by dynamically attaching additional responsibilities. The intent of the Thread-Safe Interface pattern is similar, in that it attaches robust and efficient locking strategies to make components thread-safe. The primary difference is that the Decorator pattern focuses on attaching additional responsibilities to objects dynamically, whereas the

Thread-Safe Interface pattern focuses on the static partitioning of method responsibilities in component classes.

Components designed according to the Strategized Locking pattern (237) should employ the Thread-Safe Interface pattern because it ensures that the component will function robustly and efficiently regardless of the type of locking strategy that is selected.

Java implements locking at the method-level via monitor objects (299) designated by the `synchronized` keyword. In Java, monitors are recursive. Therefore, the problem of self-deadlock cannot occur as long as developers reuse the same monitor, that is, synchronize on the same object. However, the problem of nested monitor lockout [JS97a] [Lea99] is common in Java and can occur if care is not taken when using multiple nested monitors.

The problem of locking overhead depends on which virtual machine is used. If a specific virtual machine has a poor monitor implementation and monitors are acquired recursively the Thread-Safe Interface pattern can be applied to improve run-time performance in Java.

Credits Thanks to Brad Appleton for comments on this pattern. Prashant Jain provided the discussion on the Thread-Safe Interface variants, as well as the discussion of the 'nested monitor lockout' problem in Java.

Double Checked Locking Optimization

The *Double-Checked Locking Optimization* pattern reduces contention and synchronization overhead whenever critical sections of code acquire locks just once during program execution, but must be thread-safe when locks are acquired.

Also Known As Lock Hint [Bir91]

Example The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance. The following C++ code illustrates the canonical implementation of Singleton from [GHJV95]:

```
class Singleton {
public:
     static Singleton *instance (void) {
          if (instance_ == 0) {
               // Enter critical section.
               instance_ = new Singleton();
               // Leave critical section.
          }
          return instance_;
     }
     // Other methods and members omitted.
private:
     static Singleton *instance_;
     // Initialized to 0 by linker.
};
```

Applications use the static `instance()` method to retrieve a pointer to the Singleton and then invoke public methods, as follows:

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

Unfortunately, the canonical implementation of the Singleton pattern shown above is problematic in the presence of preemptive multi-tasking or true hardware parallelism. For instance, if multiple preemptive threads invoke `Singleton::instance()` simultaneously *before it is initialized*, the Singleton's constructor can be called multiple times if multiple threads execute the dynamic initialization of the Singleton within the critical section shown above. At best, this will cause a memory leak. At worst, this will have disastrous consequences, for instance, if initialization is not idempotent.[3] To protect the critical section from concurrent access we could apply the Scoped Locking idiom (249), to acquire and release a mutex lock automatically, as follows:

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

        // Only one thread is allowed in the
        // critical section at a time.
        if (instance_ == 0)
            instance_ = new Singleton;

        return instance_;
        // Destructor of <guard> releases
        // <lock_> automatically.
    }
private:
    static Thread_Mutex lock_;
    static Singleton *instance_;
};
```

Although our Singleton implementation is now thread-safe, the additional locking overhead may be excessive. In particular, every call to `instance()` now acquires and releases the lock, even though the critical section should be executed just once.

Placing the guard inside the conditional check would remove the locking overhead, as follows.

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

        // Only come here if instance_
        // hasn't been initialized yet.
        instance_ = new Singleton;
    }
    return instance_;
}
```

This solution does not provide thread-safe initialization, however, because there is still a race condition in multi-threaded applications

3. Object initialization is idempotent if an object can be reinitialized multiple times without harmful side-effects.

that can cause multiple initializations of `Singleton`. 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 `Singleton` and releases the lock, the blocked thread will obtain the lock and erroneously initialize `Singleton` a second time.

Context A concurrent application containing shared resources accessed by multiple threads.

Problem Shared resources in a concurrent application must ensure that certain portions of their code execute serially to avoid race conditions. A common way to avoid race conditions is to use locks, such as mutexes, which serialize access to the shared resources' critical sections. Every thread that wants to enter a critical section must first acquire a lock. If this lock is already acquired by another thread, the thread blocks until the lock is released and the lock can be acquired.

This serialization approach, however, can be inappropriate for objects or components, such as singletons, that require 'just once' initialization. In particular, even though the critical section code in our Singleton example must be executed just once, during its initialization, every method call on the singleton will acquire and release the mutex. As shown in [PLoPD3], this straight-forward solution can incur measurable performance penalties due to excessive locking overhead.

Solution Introduce a flag that provides a 'hint' as to whether it is necessary to execute a critical section *before* acquiring the lock that guards it. If this code need not be executed, the critical section is skipped, thereby avoiding unnecessary locking overhead. The general pseudo-code design of this code is shown below:

```
// Perform first-check to evaluate 'hint'.
if (first_time_in_flag is FALSE) {
    acquire the mutex

    // Perform double-check to avoid race condition.
    if (first_time_in_flag is FALSE) {
        execute the critical section
        set first_time_in_flag to TRUE
    }

    release the mutex
```

Implementation The Double-Checked Locking Optimization pattern can be implemented in the following three steps:

1 *Identify the critical section to be executed just once*. This critical section typically performs operations, such as initialization logic, that are executed just once in a program.

➥ For instance, a singleton is initialized just once in a program. Thus, the call to the singleton's constructor is executed only once in a critical section, regardless of the number of times the accessor method `Singleton::instance()` is called. ❑

2 *Implement the locking logic*. The locking logic serializes access to the critical section of code that is executed just once.

➥ A `Thread_Mutex` can ensure that the singleton's constructor does not execute concurrently. To implement this locking logic we can employ the Scoped Locking idiom (249) to ensure that the lock is acquired automatically when the appropriate scope is entered and released automatically when it goes out of scope. ❑

3 *Implement the first-time-in flag*. This flag indicates whether the critical section has been executed already.

➥ The `Singleton::instance_` pointer is used as the first-time-in flag. If the flag evaluates to true, the critical section is skipped. If the flag is also used for a particular application-specific purpose, as our `Singleton::instance_` pointer is used, it must be an atomic type that can be set without a partial read or write. The following code for the singleton example is thread-safe, but avoids unnecessary locking overhead by placing the call to `new` within another conditional test:

```
class Singleton {
public:
     static Singleton *instance (void) {
          // First check
          if (instance_ == 0) {
               // Ensure serialization (guard
               // constructor acquires lock_).
               Guard<Thread_Mutex> guard (lock_);
               // Double check.
               if (instance_ == 0)
                    instance_ = new Singleton;
               // guard destructor releases lock_.
          }
          return instance_;
     }
```

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

The first thread that acquires the `lock_` will construct the `Singleton` object and assign the pointer to `instance_`, which serves as the first-time-in flag in this example. All threads that call `instance()` subsequently will find `instance_ != 0` and skip the initialization step.

The second check prevents a race condition if multiple threads try to initialize `Singleton` simultaneously. This handles the case where multiple threads execute in parallel. In the code above, these threads will queue up at the `lock_` mutex. When the queued threads finally obtain the mutex lock_, they will find `instance_ != 0` and skip the initialization of `Singleton`.

This implementation of `Singleton::instance()` only incurs locking overhead for threads that are active inside of `instance()` when `Singleton` is first initialized. In subsequent calls to `instance()`, the `instance_` pointer is not `0` and therefore `lock_` is neither acquired nor released. ❏

Variants *Volatile data*. The Double-Checked Locking Optimization pattern implementation may require modifications if a compiler optimizes the first-time-in flag by caching it in some way, such as storing it in a CPU register. In this case, cache coherency may become a problem if copies of first-time-in flag held simultaneously in registers by multiple threads become inconsistent. Thus, one thread’s setting of the value might not be reflected in other threads’ copies. A related problem is that a highly optimizing compiler may consider the second check of `flag` == `0` superfluous and optimized away.

A solution to both these problems is to declare the flag as `volatile` data to ensure that a compiler will not perform aggressive optimizations that change the program’s semantics.

➥ For our Singleton example, this variant results in the following code:

```
class Singleton {
    // ...

private:
    static volatile Singleton *instance_;
    // instance_ is volatile.
};
```

The use of `volatile` ensures that a compiler will not place the `instance_` pointer into a register nor will it optimize the second check away. ❑

The downside of using `volatile` is that all access the flag will be through memory, rather than through registers, which may degrade performance.

Template adapter. Another variation for the Double-Checked Locking Optimization pattern is applicable when the pattern is implemented in C++. In this case, create a template adapter that transforms classes to have singleton-like behavior and performs the Double-Checked Locking Optimization pattern automatically. The following code illustrates how to write this template in C++:

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

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

private:
    static LOCK lock_;
    static TYPE *instance_;
};
```

The Singleton template is parameterized by `TYPE` and `LOCK`. Thus, a class of the given `TYPE` is transformed into a Singleton with the Double-Checked Locking Optimization pattern applied automatically on the `LOCK` type within the `instance()` method.

Pre-initialization of singletons. This variation is actually an alternative to using the Double-Checked Locking Optimization at all. It works by initializing all objects explicitly during program start-up, for instance, in the `main()` function of a C or C++ program. Therefore, there are no race conditions because the initialization is constrained to occur within a single thread.

This solution is inappropriate, however, when expensive calculations must be performed that may not be necessary. For instance if a, singleton is never actually created during program execution. initializing it during program start-up will simply waste resources and frustrate end-users. In addition, pre-initialization can break encapsulation by forcing application components with singletons in their implementation to expose this information somehow so that singletons can be initialized. Likewise, requiring pre-initialization makes it hard to compose applications using components that are configured dynamically using the Service Configurator pattern (203).

Known Uses

ACE. The Double-Checked Locking Optimization pattern is used extensively throughout the ACE network programming toolkit [Sch97]. For instance, to reduce code duplication, ACE uses a reusable adapter `ACE_Singleton` template to transform 'normal' classes to have singleton-like behavior [PLoPD3]. Although singletons are not the only use case of the Double-Checked Locking Optimization pattern in ACE, they are particularly easy to motivate.

Sequent Dynix/PTX. The Doubled-Checked Locking Optimization pattern is used in the Sequent Dynix/PTX operating system.

POSIX. The Double-Checked Locking Optimization pattern can be used to implement POSIX 'once' variables [IEEE96], which ensure that functions are only invoked once in a program.

Andrew Birrell describes the use of the Double-Checked Locking Optimization pattern in [Bir91]. Birrell refers to the first check of the flag as a lock 'hint.'

The Solaris 2.x documentation for the POSIX thread-specific storage functions, such as `pthread_key_create(3T)`, as discussed in the Thread-Specific Storage pattern (321) illustrates the use of the Double-Checked Locking Optimization to initialize thread-specific data.

Consequences There are two **benefits** of using the Double-Checked Locking Optimization pattern:

Minimized locking overhead. By performing two first-time-in flag checks, the Double-Checked Locking Optimization pattern minimizes overhead for the common case. Once the flag is set, the first check ensures that subsequent accesses require no further locking.

Prevents race conditions. The second check of the first-time-in flag ensures that the critical section is executed just once.

However, there are three **liabilities** of using the Double-Checked Locking Optimization pattern that can arise if the pattern is used in software that is ported to certain types of operating system, hardware, or compiler/linker platforms. Note, however, that this pattern is applicable to a large-class of platforms. Therefore, we not only describe the three liabilities but also outline techniques for overcoming them.

Non-atomic pointer or integral assignment semantics. 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 read and written atomically 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 will sporadically result in illegal memory accesses.

Such scenarios are possible on systems where memory addresses straddle word alignment boundaries, such as 32 bit pointers used on a computer with a 16 bit word bus—which requires two fetches from memory for each pointer access. In this case, it may be necessary to use a separate, word-aligned integral flag (assuming that the hardware supports atomic word-based reads and writes), rather than using an `instance_` pointer.

Multi-processor cache coherency. Certain multi-processor platforms, such as the DEC Alpha and Intel Merced, perform aggressive memory caching optimizations where read and write operations can execute 'out of order' across multiple CPU caches. In these platforms, it may not be possible to use the Double-Checked Locking Optimization pattern without further modifications because CPU cache lines will not be flushed properly if shared data is accessed without locks held. To use the Double-Checked Locking Optimization pattern correctly on these hardware platforms requires CPU-specific instructions, such as memory barriers to flush cache lines, to be inserted into the Double-Checked Locking Optimization implementation.

Note that a serendipitous side-effect of using the template adapter variation of the Double-Checked Locking Optimization pattern is that it centralizes the placement of these CPU-specific cache instructions. For instance, a memory barrier instruction can be located within the `instance()` method of the `Singleton` template adapter class, as follows:

```
template <class TYPE, class LOCK> TYPE *
Singleton<TYPE, LOCK>::instance (void) {
#if defined (ALPHA_MP)
    // Insert the CPU-specific memory barrier instruction
    // to synchronize the cache lines on multi-processor.
    asm ("mb");
#endif /* ALPHA_MP */
    // First check
    if (instance_ == 0) {
        // Ensure serialization (guard
        // constructor acquires lock_).
        Guard<LOCK> guard (lock_);

        // Double check.
        if (instance_ == 0)
            instance_ = new TYPE;
    }
    return instance_;
}
```

As long as all applications use the `Singleton` template adapter, rather than writing their own singletons and hand-crafting the Double-Checked Locking Optimization pattern at each point of use, it is straightforward to localize the placement of CPU-specific code.

Initialization of static mutexes. The various implementations of `Singleton` shown above all define `lock_` as a static data member. Unfortunately, the C++ language specification does not guarantee the order of initialization of static objects that are defined in separate compilation units. As a result, different compiler and linker platforms may behave inconsistently and lock_ may not be initialized when it is first accessed. One way to avoid this problem is to avoid global objects altogether or at least avoid defining non-trivial application logic in constructors for global objects. Another solution is to use the Object Lifetime Manager pattern [LGS99]. This pattern governs the entire lifetime of global or static objects by creating them prior to their first use and ensuring they are destroyed properly at program termination.

See Also The Double-Checked Locking Optimization pattern is a thread-safe variant of the Lazy Evaluation idiom [Mey98] [Beck97]. This idiom is often used in programming languages like C that lack constructors to ensure components are initialized before their state is accessed:

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

void push (T *item) {
    // First-time-in-check.
    if (stack_ == 0) {
        // Allocate the pointer, which implicitly
        // indicates that initialization was performed.
        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 implicit initialization via malloc().

Credits The co-author of the original version [PLoPD3] of this pattern was Tim Harrison. Thanks to Jim Coplien, Ralph Johnson, Jaco van der Merwe, Duane Murphy, Paul McKenney, and John Basrai for their suggestions and comments on the Double-Checked Locking Optimization pattern.