Strategized Locking, Thread-safe Interface, and Scoped Locking

Patterns and Idioms for Simplifying Multi-threaded C++ Components

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, USA

This paper will appear in the C++ Report magazine.

1 Introduction

Developing multi-threaded applications is hard since incorrect use of locks can cause subtle and pernicious errors. Likewise, developing multi-threaded *reusable* components is hard since it can be time-consuming to customize components to support new, more efficient locking strategies. This paper describes a pair of patterns, *Strategized Locking* and *Thread-safe Interface*, and a C++ idiom, *Scoped Locking*, that help developers avoid common problems when programming multi-threaded components and applications.

2 The Strategized Locking Pattern

2.1 Intent

The Strategized Locking pattern strategizes a component's synchronization to increase its flexibility and reusability without degrading its performance or maintainability.

2.2 Example

A key component used to implement a high-performance Web server is a file cache, which maps URL pathnames to memory-mapped files [1]. When a client requests a URL pathname that is already cached, the Web server can transfer the contents of the memory-mapped file back to the client without having to access slower secondary storage via multiple `read` and `write` operations. A file cache implementation for a highly portable high-performance Web server must 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 multi-threaded file cache implementation.
class File_Cache_Thread_Mutex {
public:
  // Return a pointer to the memory-mapped
  // file associated with <filename>.
  const char *find (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_;
};

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

These two implementations form part of a component family whose classes differ only in their synchronization strategy. One component in the family, `File_Cache_ST`, implements a single-threaded file cache that requires no locking and the other component in the family, `File_Cache_Thread_Mutex`, implements a file cache that uses a mutex to serialize multiple threads that access the cache concurrently. However, maintaining multiple separate implementations of similar file cache components can be tedious since future enhancements and fixes must be updated consistently in each component implementation.

2.3 Context

An application or system where reuseable components must operate in a variety of concurrency use-cases.

2.4 Problem

A family of reusable components whose synchronization strategies are hard-coded into their implementations fail to resolve the following *forces*:

Ease of performance tuning: It should be straightforward to tune a component for particular concurrency use-cases. If the synchronization strategy is hard-coded, however, it is time-consuming to modify the component to support new, more efficient synchronization strategies.

For instance, a completely new class must be written to add support for a `File_Cache_RW_Lock` implementation that uses a readers-writer lock instead of a thread mutex to increase performance on large-scale multi-processor platforms.

Ease of maintenance and enhancement: New enhancements and bug fixes should be straightforward. If there are multiple copies of the same basic components however, version-skew is likely to occur since changes to one component may not be applied consistently to other component implementations.

For instance, improvements to the single-threaded file cache algorithm must be explicitly inserted into all related classes in the file cache component family, which is tedious, error-prone, and non-scalable.

2.5 Solution

Strategize a component's synchronization aspects by making them "pluggable" types. Define instances of these pluggable types as data members that are used by the component's method implementations to perform synchronization strategies that are configured into the component by an application or a service.

2.6 Implementation

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

1. Define the component interface and implementation. The focus of this step is to define a concise component interface and an efficient implementation without concern for synchronization aspects.

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

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

private:
  // File_Cache data members and methods go here...
```

2. Strategize variable synchronization aspects: In this step, 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 like mutexes and semaphores. These synchronization aspects can be strategized in a uniform manner using either *parameterized types* or *polymorphism*.

- **Polymorphism:** In this approach, pass a polymorphic `Lock` object to the component's initialization method and define an instance of this `Lock` object as a private data member that performs the locking strategy in component method implementations.

A common way to implement a polymorphic `Lock` object is to use the Bridge pattern[2]. First, we define an abstract locking class with polymorphic `acquire` and `release` methods, as follows:

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

  // Release the lock.
  virtual int release (void) = 0;

  // ...
};
```

Subclasses must override the pure virtual methods in `Lockable` to define a concrete locking strategy. For instance, the the following class defines a `Thread_Mutex` lock:

```
class Thread_Mutex_Lockable
  : public Lockable
{
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_;
};
```

Finally, we apply the Bridge pattern to define a non-polymorphic interface class that holds a reference to the polymorphic `Lockable`:

```
class Lock
{
public:
  // Constructor stores a reference to the
  // base class.
  Lock (Lockable &l): lock_ (l) {};
  // Acquire the lock by forwarding to the
  // polymorphic acquire() method.
  int acquire (void) { lock_.acquire (); }
  // Release the lock by forwarding to the
  // polymorphic release() method.
  int release (void) { lock_.release (); }

private:
  // Maintain a reference to the polymorphic lock.
  Lockable &lock_;
};
```

The purpose of this class is to ensure that `Lock` can be used as an object, rather than a pointer to a base class. This design makes it possible to reuse the Scoped Locking idiom for the polymorphic locking strategies, as shown in the following `File_Cache` component:

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

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

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

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

- **Parameterized types:** In this approach, add an appropriate `LOCK` template parameter to the component and define the appropriate instances of `LOCK` as private data member(s) that perform the locking strategy used in component method implementations.

The following illustrates a `File_Cache` component that is strategized by a `LOCK` template parameter:

```
template <class LOCK>
class File_Cache
{
public:
  // A method.
  const char *find (const char *pathname)
  {
    // Use the Scoped Locking idiom to
    // acquire and release the <lock_>
    // automatically.

    Guard<LOCK> guard (lock_);
    // Implement the find() method.
  }
  // ...

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

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

Using this implementation, the template can be instantiated with any `LOCK` type that conforms to the `acquire` and `release` signature expected by the Scoped Locking idiom described in Section 4. In particular, the instantiated template parameter for `LOCK` need not inherit from an abstract base class, such as `Lockable`.

In general, the parameterized type approach should be used when the locking strategy is known at compile-time and the polymorphic approach should be used when the locking strategy is not known until run-time. As usual, the tradeoff is between the run-time performance of templates vs. the potential for run-time extensibility with polymorphism.

3. Define a family of locking strategies: Each member of this family should provide a uniform interface that can support various application-specific concurrency use-cases. If appropriate synchronization components do not already exist, or the ones that exist have incompatible interfaces, use the Wrapper Facade pattern [3] to implement or adapt them to conform to signatures expected by the component's synchronization aspects.

In addition to the `Thread_Mutex` defined in the Wrapper Facade pattern [3], other common locking strategies include readers/writer locks, semaphores, recursive mutexes, and file locks. 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 completely removed by optimizing compilers. This class is an example of the Null Object pattern [4], which simplifies applications by defining a "no-op" placeholder.

2.7 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;`

Note how in each of these configurations the `File_Cache` interface and implementation require no changes. This transparency stems from the Strategized Locking pattern, which abstracts the synchronization aspect into a "pluggable" parameterized type. Moreover, the details of locking have been strategized via a `typedef`. Therefore, it is straightforward to define a `FILE_CACHE` object without exposing the synchronization aspect to the application, as follows:

```
FILE_CACHE file_cache;
```

2.8 Known Uses

The Booch Components were one of the first C++ class libraries to parameterize locking strategizes with templates [5].

The Strategized Locking pattern is used extensively throughout the ACE OO network programming toolkit [6].

Aspect-Oriented Programming (AOP) [7] is a general methodology for systematically strategizing aspects that vary in applications and components.

2.9 See Also

The Scoped Locking idiom described in Section 4 uses Strategized Locking to parameterize various synchronization strategies into its guard class.

2.10 Consequences

There are two `benefits` that result from applying the Strategized Locking pattern to reuseable components:

1. Enhanced flexibility and performance tuning. Because the synchronization aspects of components are strategized, it is straightforward to configure and tune a component for particular concurrency use-cases.

2. 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 avoids version-skew.

There is a `liability` that results from applying the Strategized Locking pattern to reuseable 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. One way to avoid this problem is to apply the polymorphic approach to strategize component locking behavior.

3 The Thread-safe Interface Pattern

3.1 Intent

The Thread-safe Interface pattern ensures that intra-component method calls avoid self-deadlock and minimize locking overhead.

3.2 Example

When designing thread-safe components, developers must be careful to avoid self-deadlock and unnecessary locking overhead when intra-component method calls are used. To illustrate this situation, consider a more complete implementation of the `File_Cache` component that was outlined in the Strategized Locking pattern in Section 2.

```
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 *find (const char *pathname) {
    // Use the Scoped Locking idiom to
    // automatically acquire and release the
    // <lock_>.
    Guard<Thread_Mutex> guard (lock_);

    const char *file_pointer =
      check_cache (pathname);
    if (file_pointer == 0) {
      // Insert the <pathname> into the cache.
      // Note the intra-class <bind> method call.
      bind (pathname);
      file_pointer = check_cache (pathname);
    }
    return file_pointer;
  }

  // Add <pathname> to the cache.
  void bind (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 well only when strategized with a lock with recursive mutex semantics (or a `Null_Mutex`). If it is strategized with a non-recursive mutex, however, the code will "self-deadlock" when the `find` method calls the `bind` method since `bind` reacquires the `LOCK` already held by `find`. Moreover, even if this `File_Cache` implementation is strategized with a recursive mutex it will incur unnecessary overhead when it reacquires the mutex in `bind`.

3.3 Context

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

3.4 Problem

Multi-threaded components typically contain multiple interface and implementation methods that perform computations on state that is encapsulated by the component. Component state is protected by a lock that prevents race conditions by serializing methods in the component that access the state. Component methods often call each other to carry out their computations. In multi-threaded components with poorly designed intra-component method invocation behavior, however, the following *force*s will be unresolved:

Avoid Self-deadlock: Thread-safe components should be designed to avoid 'self-deadlock.' Self-deadlock will occur if one component method acquires a non-recursive component lock and calls another method that tries to reacquire the same lock.

Minimal locking overhead: Thread-safe components should be designed to incur only the minimal locking overhead necessary to prevent race conditions. However, if a recursive component lock is selected to avoid the self-deadlock problem outlined above, additional overhead will be incurred to acquire and release the lock multiple times across intra-component method calls.

3.5 Solution

Structure components with 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 locks, thereby performing the synchronization checks at the "border" of the component. Interface methods are also responsible for releasing the lock when control returns to the caller. After the lock is acquired, the interface method should forward to an implementation method, which performs the actual method functionality.

Implementation methods trust: Implementation methods, such as C++ private and protected methods, should just perform work when called by interface methods, that is, they should should trust that they are called with locks held and never acquire/release locks. Moreover, implementation methods should never call interface methods since these methods acquire locks.

As long as these design conventions are followed, components will avoid self-deadlock and minimize locking overhead.

3.6 Implementation

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

1. Determine the interface and corresponding implementation methods: These methods define the public interface to the component. For each interface method, define a corresponding implementation method.

The interface and corresponding implementation methods for the `File_Cache` is 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 *find (const char *pathname);
  void bind (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 *find_i (const char *pathname);
  void bind_i (const char *pathname);

  // ... Other implementation methods omitted ...
```

2. Define the interface and implementation methods: The interface and implementation methods are defined according to the Thread-safe Interface 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 *find (const char *pathname) {
    // Use the Scoped Locking idiom to
    // automatically acquire and release the
    // <lock_>.
    Guard<LOCK> guard (lock_);
    return find_i (pathname);
  }

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

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

  // The following implementation methods do not
  // acquire or release <lock_> and perform their
  // work without calling any interface methods.

  const char *find_i (const char *pathname) {
    const char *file_pointer =
      check_cache_i (pathname);
    if (file_pointer == 0) {
      // If the <pathname> isn’t in the cache
      // then insert it nto the cache and
      // look it up again.
      bind_i (pathname);
      file_pointer = check_cache_i (pathname);
      // The calls to implementation methods
      // <bind_i> and <check_cache_i>, which
      // assume that the lock is held and perform
      // the work.
    }
    return file_pointer;
  }

  const char *check_cache_i (const char *)
  { /* ... */ }
  void bind_i (const char *)
  { /* ... */ }

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

3.7 Known Uses

The Thread-safe Interface pattern is used extensively throughout the ACE object-oriented network programming toolkit [6].

3.8 See Also

The Thread-safe Interface pattern is related to the Decorator pattern [2], 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 thread-safe components. The primary difference is that the Decorator pattern focuses on attaching additional responsibilities to objects, whereas the Thread-safe Interface pattern focuses on classes.

Components designed according to the Strategized Locking pattern described in Section 2 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.

3.9 Consequences

There are two **benefits** that result from applying the Thread-safe Interface pattern to multi-threaded components:

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

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

There is a `liability` that results from 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 each interface method.

4 The Scoped Locking Idiom

4.1 Intent

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

4.2 Also Known As

Guard, Synchronized Block

4.3 Example

Commercial Web servers typically maintain a “hit count” component that records the number of times each URL is accessed by clients over a period of time. To reduce latency, the hit count component can be cached in a memory-resident table by each Web server process. Moreover, to increase throughput, Web server processes are often multi-threaded [1]. Therefore, public methods in the hit count component must be serialized to prevent threads from corrupting the state of its internal table by updating the hit count concurrently.

One way to serialize access to the hit count component is to explicitly acquire and release a lock in each public method:

```
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 = find_or_create (pathname);
    if (entry == 0) {
      // Something’s gone wrong, so bail out.
      lock_.release ();
      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:
  // Find the table entry that maintains the hit count
  // associated with <pathname>, creating the entry if
  // it doesn’t exist.
  Table_Entry *find_or_create (const char *pathname);

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

Although this code may work for the current `Hit_Count` component, this implementation is tedious and error-prone to develop and maintain. For instance, maintenance programmers may forget to release the `lock_` on all return paths out of the `increment` method. Moreover, since code 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. If the `lock_` is not released, however, the Web server process may hang when subsequent threads block indefinitely trying to acquire the `lock_`.

4.4 Context

A concurrent application that implements shared resources manipulated concurrently by multiple threads.

4.5 Problem

Multi-threaded applications and components that acquire and release locks explicitly fail to resolve the following *force*:

Robust locking: Locks should always be acquired and released properly when control enters and leaves critical sections, respectively. If locks are acquired and released explicitly, however, it is hard to ensure all paths through the code release locks.

A maintenance programmer may revise the `increment` method to check for a new failure condition, as follows:

```
// ...
else if (entry->increment_hit_count () == -1)
  return -1;
```

Likewise, the `find_or_create` method may be changed to throw an exception if an error occurs. Unfortunately, both modifications will cause the `increment` method to return without releasing the `lock_`. Moreover, if these error cases occur infrequently, the problems with this code may not show up during the testing process.

4.6 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 and block scope(s) that define critical sections.

4.7 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 to the lock and then acquires the lock before entering the critical section. The destructor of this class uses the pointer stored by the constructor to release the lock when the scope of the critical section is left. A pointer is used since the C++ wrapper facades for locks disallow copying and assignment.

The following class illustrates a guard designed for the `Thread_Mutex`:

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

  // Release the lock when the guard goes
  // out of scope.
  ˜Thread_Mutex_Guard (void)
  {
    // Only release the lock if it was acquired.
    if (result_ != -1)
      lock_.release ();
  }
private:
  // Reference to the lock we’re managing.
  Thread_Mutex &lock_;

  // Records if the lock was acquired successfully.
  int result_;
};
```

4.8 Example Resolved

The following C++ code illustrates how to apply the Scoped Locking idiom to resolve the original problems with the `Hit_Counter` component 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 = find_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 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 automatically acquired and released as control enters and leaves the `increment` method, respectively.

4.9 Variants

The Scoped Locking idiom has the following variant:

Strategized Scoped Locking: Defining a different guard class for each type of lock is tedious, error-prone, and may increase the memory footprint of the application or component. Therefore, a common variant of the Scoped Locking idiom is to apply the Strategized Locking pattern described in Section 2, using either *parameterized types* or *polymorphism*.

Parameterized types. In this approach, define a template guard class that is parameterized by the type `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) {
    result_ = lock_.acquire ();
  }

  // Release the lock when the guard goes out of scope.
  ˜Guard (void) {
    if (result_ != -1) lock_.release ();
  }

private:
  // Reference to the lock we're managing.
  LOCK &lock_;

  // Records if the lock was acquired successfully.
  int result_;
};
```

Using this implementation, the template can be instantiated with any `LOCK` type that conforms to the `acquire` and `release` signature expected by the Guard template.

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 that is performs the Scoped Locking idiom on `Lock`s. In this implementation, the `Lock` class uses the Bridge pattern [2] to provide an object interface to a polymorphic lock hierarchy.

The following illustrates a Guard class that controls a polymorphic lock like the one defined in the implementation section of the Strategized Locking pattern in Section 2.6:

```
class Guard {
public:
  // Store a pointer to the lock and acquire the lock.
  Guard (Lock &lock): lock_ (lock) {
    result_ = lock_.acquire ();
  }

  // Release the lock when the guard goes out of scope.
  ˜Guard (void) {
    if (result_ != -1) lock_.release ();
  }

private:
  // Reference to the lock we're managing.
  Lock &lock_;

  // Records if the lock was acquired successfully.
  int result_;
};
```

In general, the parameterized type approach should be used when the locking strategy is known at compile-time and the polymorphic approach should be used when the locking strategy is not known until run-time. As usual, the tradeoff is between the run-time performance of templates vs. the potential for run-time extensibility with polymorphism.

4.10 Known Uses

The Scoped Locking idiom is used extensively throughout the ACE object-oriented network programming toolkit [6].

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 [8] that implements the Scoped Locking idiom in the language.

4.11 See Also

The Scoped Locking idiom is an application of the general C++ idiom "resource acquisition is object initialization" [9], 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.

4.12 Consequences

There are two `benefits` of using the Scoped Locking idiom:

1. 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. Therefore, this idiom increases concurrent application robustness by eliminating a common class of synchronization programming errors.

2. 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 because there is only one implementation, rather than a separate implementation for each type of guard. This centralization of concerns avoids version-skew.

There are several `liabilities` that result from 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 then "self-deadlock" will occur if the lock is not a "recursive" mutex. The Thread-safe Interface pattern documented in Section 3 avoids this problem by ensuring that only interface methods apply the Scoped Locking idiom and the 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 is not necessarily tied into operating system-specific system calls. Therefore, it may not be able to release locks when threads or processes are aborted 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 = find_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!
```

Therefore, it is generally inappropriate to exit a thread within a component.

5 Concluding Remarks

Knowledge of patterns and idioms are an important method of alleviating the costly rediscovery and reinvention of proven concurrent software concepts and component solutions. Patterns and idioms are useful for documenting recurring micro-architectures, which are abstractions of common object-structures that expert developers apply to solve concurrent software problems, such as deadlock avoidance and low-cost locking. By studying and applying patterns and idioms, developers can often avoid traps and pitfalls that have traditionally been learned only by prolonged trial and error.

The locking patterns and idioms described in this paper are used extensively in the ACE [6] network programming framework. ACE can be obtained via the Web at the following URL:
`www.cs.wustl.edu/∼schmidt/ACE.html`
Thanks to Brad Appleton, Erik Koerber, and Tom Ziomek for comments on this article.

References

[1] D. C. Schmidt and J. Hu, "Developing Flexible and High-performance Web Servers with Frameworks and Patterns," *ACM Computing Surveys*, vol. 30, 1998.

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

[3] D. C. Schmidt, "Wrapper Facade: A Structural Pattern for Encapsulating Functions within Classes," *C++ Report*, vol. 11, February 1999.

[4] B. Woolf, "The Null Object Pattern," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[5] G. Booch and M. Vilot, "Simplifying the Booch Components," *C++ Report*, vol. 5, June 1993.

[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] G. Kiczales, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[8] J. Gosling and K. Arnold, *The Java Programming Language*. Reading, MA: Addison-Wesley, 1996.

[9] Bjarne Stroustrup, *The C++ Programming Language,* 3^{rd} *Edition*. Addison-Wesley, 1998.