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.