Thread-Safe Interface

The *Thread-Safe Interface* design pattern minimizes locking overhead and ensures that intra-component method calls do not incur 'self-deadlock' by trying to reacquire a lock that is held by the component already.

Example When designing thread-safe components when intra-component method calls, developers must be careful to avoid self-deadlock and unnecessary locking overhead. For example, consider a more complete implementation of the File_Cache component outlined in the Strategized Locking pattern (333):

```
template <class LOCK>
class File_Cache {
public:
     // Return a pointer to the memory-mapped file
     // associated with <path> name, adding
     // it to the cache if it doesn't exist.
     const void *lookup (const string &path) const {
          // Use the Scoped Locking idiom to acquire
          // and release the <lock_> automatically.
          Guard<LOCK> guard (lock_);
          const void *file_pointer = check_cache (path);
          if (file_pointer == 0) {
               // Insert the <path> name into the cache.
               // Note the intra-class <insert> call.
               insert (path);
               file_pointer = check_cache (path);
          }
          return file_pointer;
     }
     // Add <path> name to the cache.
     void insert (const string &path) {
          // Use the Scoped Locking idiom to acquire
          // and release the <lock_> automatically.
          Guard<LOCK> guard (lock_);
          // ... insert <path> into the cache...
     }
private:
     mutable LOCK lock_;
     const void *check_cache (const string &) const;
     // ... 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 (333). If the `File_Cache` implementation is strategized with a recursive mutex, however, 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, the code will 'self-deadlock' when the `lookup()` method calls the `insert()` method. This self-deadlock occurs because `insert()` tries to reacquire the `LOCK` that has been acquired by `lookup()` already.

It is therefore 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 multi-threaded applications that contain intra-component method calls.

Problem Multi-threaded components often contain multiple publicly-accessible interface methods and private implementation methods that can alter the component states. 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 well if each method is self-contained, component methods may 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 another 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 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 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 forward immediately to an implementation method, which performs the actual method functionality. After 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. They therefore trust that they are called with the necessary lock(s) held and should never acquire or release lock(s). Implementation methods should also never call 'up' to interface methods, because these methods acquire lock(s).

Implementation The Thread-Safe Interface pattern can be implemented using two activities:

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 implementation methods for `File_Cache` can be 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 void *lookup (const string &path) const;
    void insert (const string &path);
private:
    // The following two implementation methods do not
    // acquire/release the <LOCK> and perform the actual
    // work associated with managing the <File_Cache>.
    const void *lookup_i (const string &path) const;
    void insert_i (const string &path);
    // ... 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.

➥ Our File_Cache implementation applies Thread-Safe Interface to minimize locking overhead and prevent self-deadlock in class methods:

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

     // Add <path> name to the file cache.
     void insert (const string &path) {
          // Use the Scoped Locking idiom to acquire
          // and release the <lock_> automatically.
          Guard<LOCK> guard (lock_);
          insert_i (path);
     }
private:
     mutable 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 void *lookup_i (const string &path) const {
          const void *file_pointer = check_cache_i (path);
          if (file_pointer == 0) {
               // If <path> name isn’t in the cache then
               // insert it and look it up again.
               insert_i (path);
               file_pointer = check_cache_i (path);
               // The calls to implementation methods
               // <insert_i> and <check_cache_i> assume
               // that the lock is held and perform work.
          }
          return file_pointer;
     }
```

```
        const void *check_cache_i (const string &) const
             { /* */ }

        void insert_i (const string &) { /* ... */ }

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

Variants *Thread-Safe Facade*. This variant can be used if access to a whole subsystem or coarse-grained component must be synchronized. A facade [GoF95] can be introduced as the 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 have their own internal concurrency strategies, refactoring may be needed to avoid *nested monitor lockout*[2] [JS97a].

Nested monitor lockup 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. It may not be possible to refactor the code properly to avoid nested monitor lockouts if the subsystem or component cannot be modified, for example if it is a third-party product or legacy system. In this case, Thread-Safe Facade should not be applied.

Thread-Safe Wrapper Facade. This variant helps synchronize access to a non-synchronized class or function API that cannot be modified. A wrapper facade (47) provides the interface methods, which encapsulate the corresponding implementation calls on the class or function API with actions that acquire and release a lock. The wrapper facade thus provides a synchronization proxy [POSA1] [GoF95] that serializes access to the methods of the class or function API.

Known Uses **ACE** [Sch97]. The Thread-Safe Interface pattern is used throughout the ADAPTIVE Communication Environment framework, for example in its `ACE_Message_Queue` class.

2. See the Consequences Section of the Monitor Object pattern (399) for a more detailed discussion of the nested monitor lockout problem.

The **Dynix/PTX** operating system applies the Thread-Safe Interface pattern in portions of its kernel.

Java. The hash table implementation in `java.util.Hashtable` uses the Thread-Safe Interface design pattern. `Hashtable`'s interface methods, such as `put(Object key, Object value)`, acquire a lock before changing the underlying data structure, which consists of an array of linked lists. The implementation method `rehash()` is called when the load threshold is exceeded. A new larger hash table is created, all elements are moved from the old to the new hash table, and the old table is left to the garbage collector. Note that the `rehash()` method is not protected by a lock, in contrast to the publicly accessible methods such as `put(Object key, Object value)`. Protecting `rehash()` by a lock would not deadlock the program due to Java's reentrant monitors. It would, however, diminish its performance due to the locking overhead.

A more sophisticated use case was introduced in JDK 1.2 with the `Collection` classes, which applies the Thread-Safe Wrapper Facade variant to make collection data structures thread-safe. The `java.util.Collections` takes any class implementing the `Map` interface and returns a `SynchronizedMap`, which is a different class implementing `Map`. The methods of `SynchronizedMap` do no more than synchronize on an internal monitor and then forward to the method of the original object. Developers can therefore choose between fast or thread-safe variants of data structures, which only need be implemented once.

Security checkpoints. You may encounter a real-life variation of the Thread-Safe Interface pattern when entering a country or commercial office building that has a security guard at the border or entrance. To be admitted, you must sign in. After being admitted, other people that you interact with typically trust that you are supposed to be there.

Consequences There are three **benefits** of applying the Thread-Safe Interface pattern:

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

Enhanced performance. This pattern ensures that locks are not acquired or released unnecessarily.

Simplification of software. Separating the locking and functionality concerns can help to simplify both aspects.

However, there are also four **liabilities** when applying the Thread-Safe Interface pattern:

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 implementation methods.

Potential deadlock. By itself, the Thread-Safe Interface pattern does not resolve the problem of self-deadlock completely. For example, 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.

Potential for misuse. Object-oriented programming languages, such as C++ and Java, support class-level rather than object-level access control. As a result, an object can bypass the public interface to call a private method on another object of the same class, thus bypassing that object's lock. Therefore, programmer's should be careful to avoid invoking private methods on any object of their class other than themselves.

Potential overhead. The Thread-Safe Interface pattern prevents multiple components from sharing the same lock. Therefore synchronization overhead may increase because multiple locks must be acquired, which also makes it harder to detect and avoid deadlocks. Moreover, the pattern prevents locking at a finer granularity than the component, which can increase lock contention, thereby reducing performance.

See Also The Thread-Safe Interface pattern is related to the Decorator pattern [GoF95], which extends an object transparently by attaching additional responsibilities dynamically. The intention 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 (333) should employ the Thread-Safe Interface pattern to ensure 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 (399) designated by the synchronized keyword. In Java, monitors are recursive. The problem of self-deadlock therefore 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] [Lea99a] can occur in Java if multiple nested monitors are used carelessly.

The problem of locking overhead depends on which Java Virtual Machine (JVM) is used. If a specific JVM implements monitors inefficiently and monitors are acquired recursively, the Thread-Safe Interface pattern may be able to help improve component run-time performance.

Credits Thanks to Brad Appleton for comments on this pattern. Prashant Jain provided the Thread-Safe Interface variants and the Java nested monitor lockout discussion.