# NSLocking

**Adopted By:** NSConditionLock
NSLock
NSRecursiveLock

**Declared In:** Foundation/NSLock.h

## Protocol Description

The NSLocking protocol declares the elementary methods adopted by classes that define lock objects. A lock object is used to coordinate the actions of multiple threads of execution within a single application. By using a lock object, an application can protect critical sections of code from being executed simultaneously by separate threads, thus protecting shared data and other shared resources from corruption.

For example, consider a multithreaded application in which each thread updates a shared counter. If two threads simultaneously fetch the current value into local storage, increment it, and then write the value back, the counter will be incremented only once, losing one thread's contribution. However, if the code that manipulates the shared data (the *critical section* of code) must be locked before being executed, only one thread at a time can perform the updating operation, and collisions are prevented.

A lock object is either locked or unlocked. You *acquire* a lock by sending the object a **lock** message, thus putting the object in the locked state. You *relinquish* a lock by sending an **unlock** message, and thus putting the object in the unlocked state. (The NEXTSTEP classes that adopt this protocol define additional ways to acquire and relinquish locks.)

The **lock** method as declared in this protocol is *blocking*. That is, the thread that sends a **lock** message is blocked from further execution until the lock is acquired (presumably because some other thread relinquishes its lock). Classes that adopt this protocol typically add methods that offer nonblocking alternatives.

These NEXTSTEP classes conform to the NSLocking protocol:

| Class | Adds these features to the basic protocol |
|---|---|
| NSLock | A nonblocking lock method; the ability to limit the duration of a locking attempt. |
| NSConditionLock | The ability to postpone entry to a critical section until a condition is met. |
| NSRecursiveLock | The ability for a single thread to acquire a lock more than once without deadlocking. |

The locking mechanism that these classes use causes a thread to sleep while waiting to acquire a lock rather than to poll the system constantly. Thus, lock objects can be used to lock time-consuming operations

without causing system performance to degrade. See the class specifications for these classes for further information on their behavior and usage.

There is some performance cost in acquiring a lock, so it's best to avoid the overhead if possible. An application developer has control over whether the application will execute with multiple threads, so it's clear when locking is appropriate. A library developer doesn't necessarily know whether library code will execute in a multithreaded environment. In this case, it's best to test whether the code is executing in a multithreaded environment before attempting to acquire a lock. The following example illustrates how this is done.

Assume your application uses a Counter object to record various operations. Here's one design that lets the Counter know whether it is multithreaded:

```
+ (void)initialize
{
    if ([NSThread isMultiThreaded]) {
       [self taskNowMultiThreaded:nil];
    } else {
       [[NSNotificationCenter defaultCenter] addObserver:self
          selector:@selector(taskNowMultiThreaded:)
          name:NSBecomingMultiThreaded object:nil];
    }
}
```

In the initialize method (which, by definition, is invoked before any Counter objects are created), the Counter class object first checks whether the application has already become multithreaded and if so invokes its own **taskNowMultiThreaded:** method. Otherwise, it registers as an observer of the NSBecomingMultiThreaded notification so that **taskNowMultiThreaded:** will be invoked when the application becomes multithreaded.

Counter's **taskNowMultiThreaded:** method creates a lock object that the threads use to coordinate their activities:

```
+ (void)taskNowMultiThreaded:(NSNotification *)event
{
    if (!theLock)
       theLock = [[NSLock alloc] init];
}
```

**theLock**, a static variable declared in the class implementation file, is assigned a value of **nil** until **taskNowMultiThreaded:** is invoked. Since messages sent to **nil** are permitted and have no effect, code within Counter that acts on shared data can be written like this:

```
[theLock lock];
/* Operate on shared data */
[theLock unlock];
```

With this approach, the overhead associated with lock operations is only incurred if the application is multithreaded. This code, however, raises another issue. What happens if one of the statements between the

**lock** and **unlock** messages cause the application to become multithreaded? Then the **unlock** message wouldn't be paired with the preceding **lock**.

In normal usage, locking and unlocking messages are paired. However, as in the example above, it might be convenient to unlock a lock object that hasn't yet been locked. This is permitted with two restrictions. First, you can send an unpaired unlocking message to a lock object as long as the object has never before been locked. Second, of the NEXTSTEP classes that conform to the NSLocking protocol, only NSConditionLock and NSLock allow an unpaired unlocking message. NSRecursiveLock requires locking and unlocking messages to be paired.

NEXTSTEP's locking classes are designed to work in a well-behaved, multithreaded environment: The protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without alerting the thread that the application has, in effect, become multithreaded. For this reason, it's recommended that you don't use signal handlers in multithreaded NEXTSTEP applications.

## Instance Methods

### lock

– (void)**lock**

Attempts to acquire a lock. This method blocks a thread's execution until the lock can be acquired.

An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking **unlock**.

### unlock

– (void)**unlock**

Relinquishes a previously acquired lock.