# Thread Safety Analysis

## Introduction

Clang Thread Safety Analysis is a C++ language extension which warns about potential race conditions in code. The analysis is completely static (i.e. compile-time); there is no run-time overhead. The analysis is still under active development, but it is mature enough to be deployed in an industrial setting. It is being developed by Google, in collaboration with CERT/SEI, and is used extensively in Google's internal code base.

Thread safety analysis works very much like a type system for multi-threaded programs. In addition to declaring the *type* of data (e.g. `int`, `float`, etc.), the programmer can (optionally) declare how access to that data is controlled in a multi-threaded environment. For example, if `foo` is *guarded by* the mutex `mu`, then the analysis will issue a warning whenever a piece of code reads or writes to `foo` without first locking `mu`. Similarly, if there are particular routines that should only be called by the GUI thread, then the analysis will warn if other threads call those routines.

## Getting Started

```
#include "mutex.h"

class BankAccount {
private:
  Mutex mu;
  int   balance GUARDED_BY(mu);

  void depositImpl(int amount) {
    balance += amount;       // WARNING! Cannot write balance without locking mu.
  }

  void withdrawImpl(int amount) REQUIRES(mu) {
    balance -= amount;       // OK. Caller must have locked mu.
  }

public:
  void withdraw(int amount) {
    mu.Lock();
    withdrawImpl(amount);    // OK.  We've locked mu.
  }                          // WARNING!  Failed to unlock mu.

  void transferFrom(BankAccount& b, int amount) {
    mu.Lock();
    b.withdrawImpl(amount);  // WARNING!  Calling withdrawImpl() requires locking b.mu.
    depositImpl(amount);     // OK.  depositImpl() has no requirements.
    mu.Unlock();
  }
};
```

This example demonstrates the basic concepts behind the analysis. The `GUARDED_BY` attribute declares that a thread must lock `mu` before it can read or write to `balance`, thus ensuring that the increment and decrement operations are atomic. Similarly, `REQUIRES` declares that the calling thread must lock `mu` before calling `withdrawImpl`. Because the caller is assumed to have locked `mu`, it is safe to modify `balance` within the body of the method.

The `depositImpl()` method does not have `REQUIRES`, so the analysis issues a warning. Thread safety analysis is not inter-procedural, so caller requirements must be explicitly declared. There is also a warning in `transferFrom()`, because although the method locks `this->mu`, it does not lock `b.mu`. The analysis understands that these are two separate mutexes, in two different objects.

Finally, there is a warning in the `withdraw()` method, because it fails to unlock `mu`. Every lock must have a corresponding unlock, and the analysis will detect both double locks, and double unlocks. A function is allowed to acquire a lock without releasing it, (or vice versa), but it must be annotated as such (using `ACQUIRE`/`RELEASE`).

## Running The Analysis

To run the analysis, simply compile with the `-Wthread-safety` flag, e.g.

```
clang -c -Wthread-safety example.cpp
```

Note that this example assumes the presence of a suitably annotated **mutex.h** that declares which methods perform locking, unlocking, and so on.

## Basic Concepts: Capabilities

Thread safety analysis provides a way of protecting *resources* with *capabilities*. A resource is either a data member, or a function/method that provides access to some underlying resource. The analysis ensures that the calling thread cannot access the *resource* (i.e. call the function, or read/write the data) unless it has the *capability* to do so.

Capabilities are associated with named C++ objects which declare specific methods to acquire and release the capability. The name of the object serves to identify the capability. The most common example is a mutex. For example, if `mu` is a mutex, then calling `mu.Lock()` causes the calling thread to acquire the capability to access data that is protected by `mu`. Similarly, calling `mu.Unlock()` releases that capability.

A thread may hold a capability either *exclusively* or *shared*. An exclusive capability can be held by only one thread at a time, while a shared capability can be held by many threads at the same time. This mechanism enforces a multiple-reader, single-writer pattern. Write operations to protected data require exclusive access, while read operations require only shared access.

At any given moment during program execution, a thread holds a specific set of capabilities (e.g. the set of mutexes that it has locked.) These act like keys or tokens that allow the thread to access a given resource. Just like physical security keys, a thread cannot make copy of a capability, nor can it destroy one. A thread can only release a capability to another thread, or acquire one from another thread. The annotations are deliberately agnostic about the exact mechanism used to acquire and release capabilities; it assumes that the underlying implementation (e.g. the Mutex implementation) does the handoff in an appropriate manner.

The set of capabilities that are actually held by a given thread at a given point in program execution is a run-time concept. The static analysis works by calculating an approximation of that set, called the *capability environment*. The capability environment is calculated for every program point, and describes the set of capabilities that are statically known to be held, or not held, at that particular point. This environment is a conservative approximation of the full set of capabilities that will actually held by a thread at run-time.

## Reference Guide

The thread safety analysis uses attributes to declare threading constraints. Attributes must be attached to named declarations, such as classes, methods, and data members. Users are *strongly advised* to define macros for the various attributes; example definitions can be found in **mutex.h**, below. The following documentation assumes the use of macros.

The attributes only control assumptions made by thread safety analysis and the warnings it issues. They don't affect generated code or behavior at run-time.

For historical reasons, prior versions of thread safety used macro names that were very lock-centric. These macros have since been renamed to fit a more general capability model. The prior names are still in use, and will be mentioned under the tag *previously* where appropriate.

### GUARDED_BY(c) and PT_GUARDED_BY(c)

`GUARDED_BY` is an attribute on data members, which declares that the data member is protected by the given capability. Read operations on the data require shared access, while write operations require exclusive access.

`PT_GUARDED_BY` is similar, but is intended for use on pointers and smart pointers. There is no constraint on the data member itself, but the *data that it points to* is protected by the given capability.

```
Mutex mu;
int *p1             GUARDED_BY(mu);
int *p2             PT_GUARDED_BY(mu);
unique_ptr<int> p3  PT_GUARDED_BY(mu);

void test() {
  p1 = 0;             // Warning!

  *p2 = 42;           // Warning!
  p2 = new int;       // OK.

  *p3 = 42;           // Warning!
  p3.reset(new int);  // OK.
}
```

### REQUIRES(…), REQUIRES_SHARED(…)

*Previously*: `EXCLUSIVE_LOCKS_REQUIRED`, `SHARED_LOCKS_REQUIRED`

`REQUIRES` is an attribute on functions or methods, which declares that the calling thread must have exclusive access to the given capabilities. More than one capability may be specified. The capabilities must be held on entry to the function, *and must still be held on exit*.

`REQUIRES_SHARED` is similar, but requires only shared access.

```
Mutex mu1, mu2;
int a GUARDED_BY(mu1);
int b GUARDED_BY(mu2);

void foo() REQUIRES(mu1, mu2) {
  a = 0;
  b = 0;
}
```

```
void test() {
  mu1.Lock();
  foo();          // Warning!  Requires mu2.
  mu1.Unlock();
}
```

### ACQUIRE(…), ACQUIRE_SHARED(…), RELEASE(…), RELEASE_SHARED(…), RELEASE_GENERIC(…)

*Previously*: EXCLUSIVE_LOCK_FUNCTION, SHARED_LOCK_FUNCTION, UNLOCK_FUNCTION

ACQUIRE and ACQUIRE_SHARED are attributes on functions or methods declaring that the function acquires a capability, but does not release it. The given capability must not be held on entry, and will be held on exit (exclusively for ACQUIRE, shared for ACQUIRE_SHARED).

RELEASE, RELEASE_SHARED, and RELEASE_GENERIC declare that the function releases the given capability. The capability must be held on entry (exclusively for RELEASE, shared for RELEASE_SHARED, exclusively or shared for RELEASE_GENERIC), and will no longer be held on exit.

```
Mutex mu;
MyClass myObject GUARDED_BY(mu);

void lockAndInit() ACQUIRE(mu) {
  mu.Lock();
  myObject.init();
}

void cleanupAndUnlock() RELEASE(mu) {
  myObject.cleanup();
}                          // Warning!  Need to unlock mu.

void test() {
  lockAndInit();
  myObject.doSomething();
  cleanupAndUnlock();
  myObject.doSomething();  // Warning, mu is not locked.
}
```

If no argument is passed to ACQUIRE or RELEASE, then the argument is assumed to be this, and the analysis will not check the body of the function. This pattern is intended for use by classes which hide locking details behind an abstract interface. For example:

```
template <class T>
class CAPABILITY("mutex") Container {
private:
  Mutex mu;
  T* data;

public:
  // Hide mu from public interface.
  void Lock()   ACQUIRE() { mu.Lock(); }
  void Unlock() RELEASE() { mu.Unlock(); }

  T& getElem(int i) { return data[i]; }
};

void test() {
  Container<int> c;
  c.Lock();
  int i = c.getElem(0);
  c.Unlock();
}
```

### EXCLUDES(…)

*Previously*: LOCKS_EXCLUDED

EXCLUDES is an attribute on functions or methods, which declares that the caller must *not* hold the given capabilities. This annotation is used to prevent deadlock. Many mutex implementations are not re-entrant, so deadlock can occur if the function acquires the mutex a second time.

```
Mutex mu;
int a GUARDED_BY(mu);

void clear() EXCLUDES(mu) {
  mu.Lock();
  a = 0;
  mu.Unlock();
}

void reset() {
  mu.Lock();
  clear();     // Warning!  Caller cannot hold 'mu'.
```

```
  mu.Unlock();
}
```

Unlike `REQUIRES`, `EXCLUDES` is optional. The analysis will not issue a warning if the attribute is missing, which can lead to false negatives in some cases. This issue is discussed further in **Negative Capabilities**.

### NO_THREAD_SAFETY_ANALYSIS

`NO_THREAD_SAFETY_ANALYSIS` is an attribute on functions or methods, which turns off thread safety checking for that method. It provides an escape hatch for functions which are either (1) deliberately thread-unsafe, or (2) are thread-safe, but too complicated for the analysis to understand. Reasons for (2) will be described in the **Known Limitations**, below.

```
class Counter {
  Mutex mu;
  int a GUARDED_BY(mu);

  void unsafeIncrement() NO_THREAD_SAFETY_ANALYSIS { a++; }
};
```

Unlike the other attributes, NO_THREAD_SAFETY_ANALYSIS is not part of the interface of a function, and should thus be placed on the function definition (in the `.cc` or `.cpp` file) rather than on the function declaration (in the header).

### RETURN_CAPABILITY(c)

*Previously*: `LOCK_RETURNED`

`RETURN_CAPABILITY` is an attribute on functions or methods, which declares that the function returns a reference to the given capability. It is used to annotate getter methods that return mutexes.

```
class MyClass {
private:
  Mutex mu;
  int a GUARDED_BY(mu);

public:
  Mutex* getMu() RETURN_CAPABILITY(mu) { return &mu; }

  // analysis knows that getMu() == mu
  void clear() REQUIRES(getMu()) { a = 0; }
};
```

### ACQUIRED_BEFORE(…), ACQUIRED_AFTER(…)

`ACQUIRED_BEFORE` and `ACQUIRED_AFTER` are attributes on member declarations, specifically declarations of mutexes or other capabilities. These declarations enforce a particular order in which the mutexes must be acquired, in order to prevent deadlock.

```
Mutex m1;
Mutex m2 ACQUIRED_AFTER(m1);

// Alternative declaration
// Mutex m2;
// Mutex m1 ACQUIRED_BEFORE(m2);

void foo() {
  m2.Lock();
  m1.Lock();  // Warning!  m2 must be acquired after m1.
  m1.Unlock();
  m2.Unlock();
}
```

### CAPABILITY(<string>)

*Previously*: `LOCKABLE`

`CAPABILITY` is an attribute on classes, which specifies that objects of the class can be used as a capability. The string argument specifies the kind of capability in error messages, e.g. `"mutex"`. See the `Container` example given above, or the `Mutex` class in **mutex.h**.

### SCOPED_CAPABILITY

*Previously*: `SCOPED_LOCKABLE`

`SCOPED_CAPABILITY` is an attribute on classes that implement RAII-style locking, in which a capability is acquired in the constructor, and released in the destructor. Such classes require special handling because the constructor and destructor refer to the capability via different names; see the `MutexLocker` class in **mutex.h**, below.

Scoped capabilities are treated as capabilities that are implicitly acquired on construction and released on destruction. They are associated with the set of (regular) capabilities named in thread safety attributes on the constructor. Acquire-type attributes on other member functions are treated as applying to that set of associated capabilities, while `RELEASE` implies that a function releases all associated capabilities in whatever mode they're held.

### TRY_ACQUIRE(<bool>, …), TRY_ACQUIRE_SHARED(<bool>, …)

*Previously:* `EXCLUSIVE_TRYLOCK_FUNCTION`, `SHARED_TRYLOCK_FUNCTION`

These are attributes on a function or method that tries to acquire the given capability, and returns a boolean value indicating success or failure. The first argument must be `true` or `false`, to specify which return value indicates success, and the remaining arguments are interpreted in the same way as `ACQUIRE`. See **mutex.h**, below, for example uses.

Because the analysis doesn't support conditional locking, a capability is treated as acquired after the first branch on the return value of a try-acquire function.

```
Mutex mu;
int a GUARDED_BY(mu);

void foo() {
  bool success = mu.TryLock();
  a = 0;          // Warning, mu is not locked.
  if (success) {
    a = 0;        // Ok.
    mu.Unlock();
  } else {
    a = 0;        // Warning, mu is not locked.
  }
}
```

### ASSERT_CAPABILITY(…) and ASSERT_SHARED_CAPABILITY(…)

*Previously:* `ASSERT_EXCLUSIVE_LOCK`, `ASSERT_SHARED_LOCK`

These are attributes on a function or method which asserts the calling thread already holds the given capability, for example by performing a run-time test and terminating if the capability is not held. Presence of this annotation causes the analysis to assume the capability is held after calls to the annotated function. See **mutex.h**, below, for example uses.

### GUARDED_VAR and PT_GUARDED_VAR

Use of these attributes has been deprecated.

### Warning flags

- `-Wthread-safety`: Umbrella flag which turns on the following:
  - `-Wthread-safety-attributes`: Semantic checks for thread safety attributes.
  - `-Wthread-safety-analysis`: The core analysis.
  - `-Wthread-safety-precise`: Requires that mutex expressions match precisely.
    This warning can be disabled for code which has a lot of aliases.
  - `-Wthread-safety-reference`: Checks when guarded members are passed by reference.

**Negative Capabilities** are an experimental feature, which are enabled with:

- `-Wthread-safety-negative`: Negative capabilities. Off by default.

When new features and checks are added to the analysis, they can often introduce additional warnings. Those warnings are initially released as *beta* warnings for a period of time, after which they are migrated into the standard analysis.

- `-Wthread-safety-beta`: New features. Off by default.

## Negative Capabilities

Thread Safety Analysis is designed to prevent both race conditions and deadlock. The GUARDED_BY and REQUIRES attributes prevent race conditions, by ensuring that a capability is held before reading or writing to guarded data, and the EXCLUDES attribute prevents deadlock, by making sure that a mutex is *not* held.

However, EXCLUDES is an optional attribute, and does not provide the same safety guarantee as REQUIRES. In particular:

A function which acquires a capability does not have to exclude it.
A function which calls a function that excludes a capability does not have transitively exclude that capability.

As a result, EXCLUDES can easily produce false negatives:

```
class Foo {
  Mutex mu;

  void foo() {
    mu.Lock();
    bar();           // No warning.
    baz();           // No warning.
    mu.Unlock();
  }

  void bar() {       // No warning.  (Should have EXCLUDES(mu)).
    mu.Lock();
    // ...
    mu.Unlock();
  }

  void baz() {
    bif();           // No warning.  (Should have EXCLUDES(mu)).
  }

  void bif() EXCLUDES(mu);
};
```

Negative requirements are an alternative EXCLUDES that provide a stronger safety guarantee. A negative requirement uses the REQUIRES attribute, in conjunction with the `!` operator, to indicate that a capability should *not* be held.

For example, using `REQUIRES(!mu)` instead of `EXCLUDES(mu)` will produce the appropriate warnings:

```
class FooNeg {
  Mutex mu;

  void foo() REQUIRES(!mu) {   // foo() now requires !mu.
    mu.Lock();
    bar();
    baz();
    mu.Unlock();
  }

  void bar() {
    mu.Lock();       // WARNING!  Missing REQUIRES(!mu).
    // ...
    mu.Unlock();
  }

  void baz() {
    bif();           // WARNING!  Missing REQUIRES(!mu).
  }

  void bif() REQUIRES(!mu);
};
```

Negative requirements are an experimental feature which is off by default, because it will produce many warnings in existing code. It can be enabled by passing `-Wthread-safety-negative`.

## Frequently Asked Questions

Q. Should I put attributes in the header file, or in the .cc/.cpp/.cxx file?

(A) Attributes are part of the formal interface of a function, and should always go in the header, where they are visible to anything that includes the header. Attributes in the .cpp file are not visible outside of the immediate translation unit, which leads to false negatives and false positives.

Q. "*Mutex is not locked on every path through here?*" What does that mean?

A. See **No conditionally held locks.**, below.

## Known Limitations

### Lexical scope

Thread safety attributes contain ordinary C++ expressions, and thus follow ordinary C++ scoping rules. In particular, this means that mutexes and other capabilities must be declared before they can be used in an attribute. Use-before-declaration is okay within a single class, because attributes

are parsed at the same time as method bodies. (C++ delays parsing of method bodies until the end of the class.) However, use-before-declaration is not allowed between classes, as illustrated below.

```
class Foo;

class Bar {
  void bar(Foo* f) REQUIRES(f->mu);  // Error: mu undeclared.
};

class Foo {
  Mutex mu;
};
```

### Private Mutexes

Good software engineering practice dictates that mutexes should be private members, because the locking mechanism used by a thread-safe class is part of its internal implementation. However, private mutexes can sometimes leak into the public interface of a class. Thread safety attributes follow normal C++ access restrictions, so if `mu` is a private member of `c`, then it is an error to write `c.mu` in an attribute.

One workaround is to (ab)use the `RETURN_CAPABILITY` attribute to provide a public *name* for a private mutex, without actually exposing the underlying mutex. For example:

```
class MyClass {
private:
  Mutex mu;

public:
  // For thread safety analysis only.  Does not need to be defined.
  Mutex* getMu() RETURN_CAPABILITY(mu);

  void doSomething() REQUIRES(mu);
};

void doSomethingTwice(MyClass& c) REQUIRES(c.getMu()) {
  // The analysis thinks that c.getMu() == c.mu
  c.doSomething();
  c.doSomething();
}
```

In the above example, `doSomethingTwice()` is an external routine that requires `c.mu` to be locked, which cannot be declared directly because `mu` is private. This pattern is discouraged because it violates encapsulation, but it is sometimes necessary, especially when adding annotations to an existing code base. The workaround is to define `getMu()` as a fake getter method, which is provided only for the benefit of thread safety analysis.

### No conditionally held locks.

The analysis must be able to determine whether a lock is held, or not held, at every program point. Thus, sections of code where a lock *might be held* will generate spurious warnings (false positives). For example:

```
void foo() {
  bool b = needsToLock();
  if (b) mu.Lock();
  ...  // Warning!  Mutex 'mu' is not held on every path through here.
  if (b) mu.Unlock();
}
```

### No checking inside constructors and destructors.

The analysis currently does not do any checking inside constructors or destructors. In other words, every constructor and destructor is treated as if it was annotated with `NO_THREAD_SAFETY_ANALYSIS`. The reason for this is that during initialization, only one thread typically has access to the object which is being initialized, and it is thus safe (and common practice) to initialize guarded members without acquiring any locks. The same is true of destructors.

Ideally, the analysis would allow initialization of guarded members inside the object being initialized or destroyed, while still enforcing the usual access restrictions on everything else. However, this is difficult to enforce in practice, because in complex pointer-based data structures, it is hard to determine what data is owned by the enclosing object.

### No inlining.

Thread safety analysis is strictly intra-procedural, just like ordinary type checking. It relies only on the declared attributes of a function, and will not attempt to inline any method calls. As a result, code such as the following will not work:

```
template<class T>
class AutoCleanup {
  T* object;
  void (T::*mp)();

public:
  AutoCleanup(T* obj, void (T::*imp)()) : object(obj), mp(imp) { }
  ~AutoCleanup() { (object->*mp)(); }
};

Mutex mu;
void foo() {
  mu.Lock();
  AutoCleanup<Mutex>(&mu, &Mutex::Unlock);
  // ...
}  // Warning, mu is not unlocked.
```

In this case, the destructor of `Autocleanup` calls `mu.Unlock()`, so the warning is bogus. However, thread safety analysis cannot see the unlock, because it does not attempt to inline the destructor. Moreover, there is no way to annotate the destructor, because the destructor is calling a function that is not statically known. This pattern is simply not supported.

### No alias analysis.

The analysis currently does not track pointer aliases. Thus, there can be false positives if two pointers both point to the same mutex.

```
class MutexUnlocker {
  Mutex* mu;

public:
  MutexUnlocker(Mutex* m) RELEASE(m) : mu(m)  { mu->Unlock(); }
  ~MutexUnlocker() ACQUIRE(mu) { mu->Lock(); }
};

Mutex mutex;
void test() REQUIRES(mutex) {
  {
    MutexUnlocker munl(&mutex);  // unlocks mutex
    doSomeIO();
  }                              // Warning: locks munl.mu
}
```

The MutexUnlocker class is intended to be the dual of the MutexLocker class, defined in **mutex.h**. However, it doesn't work because the analysis doesn't know that munl.mu == mutex. The SCOPED_CAPABILITY attribute handles aliasing for MutexLocker, but does so only for that particular pattern.

### ACQUIRED_BEFORE(…) and ACQUIRED_AFTER(…) are currently unimplemented.

To be fixed in a future update.

## mutex.h

Thread safety analysis can be used with any threading library, but it does require that the threading API be wrapped in classes and methods which have the appropriate annotations. The following code provides `mutex.h` as an example; these methods should be filled in to call the appropriate underlying implementation.

```
#ifndef THREAD_SAFETY_ANALYSIS_MUTEX_H
#define THREAD_SAFETY_ANALYSIS_MUTEX_H

// Enable thread safety attributes only with clang.
// The attributes can be safely erased when compiling with other compilers.
#if defined(__clang__) && (!defined(SWIG))
#define THREAD_ANNOTATION_ATTRIBUTE__(x)   __attribute__((x))
#else
#define THREAD_ANNOTATION_ATTRIBUTE__(x)   // no-op
#endif

#define CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(capability(x))

#define SCOPED_CAPABILITY \
  THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

#define GUARDED_BY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(guarded_by(x))

#define PT_GUARDED_BY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(pt_guarded_by(x))
```

```
#define ACQUIRED_BEFORE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquired_before(__VA_ARGS__))

#define ACQUIRED_AFTER(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquired_after(__VA_ARGS__))

#define REQUIRES(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(requires_capability(__VA_ARGS__))

#define REQUIRES_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(requires_shared_capability(__VA_ARGS__))

#define ACQUIRE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquire_capability(__VA_ARGS__))

#define ACQUIRE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(acquire_shared_capability(__VA_ARGS__))

#define RELEASE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_capability(__VA_ARGS__))

#define RELEASE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_shared_capability(__VA_ARGS__))

#define RELEASE_GENERIC(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(release_generic_capability(__VA_ARGS__))

#define TRY_ACQUIRE(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(try_acquire_capability(__VA_ARGS__))

#define TRY_ACQUIRE_SHARED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(try_acquire_shared_capability(__VA_ARGS__))

#define EXCLUDES(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(locks_excluded(__VA_ARGS__))

#define ASSERT_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_capability(x))

#define ASSERT_SHARED_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_shared_capability(x))

#define RETURN_CAPABILITY(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(lock_returned(x))

#define NO_THREAD_SAFETY_ANALYSIS \
  THREAD_ANNOTATION_ATTRIBUTE__(no_thread_safety_analysis)


// Defines an annotated interface for mutexes.
// These methods can be implemented to use any internal mutex implementation.
class CAPABILITY("mutex") Mutex {
public:
  // Acquire/lock this mutex exclusively.  Only one thread can have exclusive
  // access at any one time.  Write operations to guarded data require an
  // exclusive lock.
  void Lock() ACQUIRE();

  // Acquire/lock this mutex for read operations, which require only a shared
  // lock.  This assumes a multiple-reader, single writer semantics.  Multiple
  // threads may acquire the mutex simultaneously as readers, but a writer
  // must wait for all of them to release the mutex before it can acquire it
  // exclusively.
  void ReaderLock() ACQUIRE_SHARED();

  // Release/unlock an exclusive mutex.
  void Unlock() RELEASE();

  // Release/unlock a shared mutex.
  void ReaderUnlock() RELEASE_SHARED();

  // Generic unlock, can unlock exclusive and shared mutexes.
  void GenericUnlock() RELEASE_GENERIC();

  // Try to acquire the mutex.  Returns true on success, and false on failure.
  bool TryLock() TRY_ACQUIRE(true);

  // Try to acquire the mutex for read operations.
  bool ReaderTryLock() TRY_ACQUIRE_SHARED(true);

  // Assert that this mutex is currently held by the calling thread.
  void AssertHeld() ASSERT_CAPABILITY(this);

  // Assert that is mutex is currently held for read operations.
  void AssertReaderHeld() ASSERT_SHARED_CAPABILITY(this);

  // For negative capabilities.
  const Mutex& operator!() const { return *this; }
};

// Tag types for selecting a constructor.
```

```
struct adopt_lock_t {} inline constexpr adopt_lock = {};
struct defer_lock_t {} inline constexpr defer_lock = {};
struct shared_lock_t {} inline constexpr shared_lock = {};

// MutexLocker is an RAII class that acquires a mutex in its constructor, and
// releases it in its destructor.
class SCOPED_CAPABILITY MutexLocker {
private:
  Mutex* mut;
  bool locked;

public:
  // Acquire mu, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu) ACQUIRE(mu) : mut(mu), locked(true) {
    mu->Lock();
  }

  // Assume mu is held, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, adopt_lock_t) REQUIRES(mu) : mut(mu), locked(true) {}

  // Acquire mu in shared mode, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, shared_lock_t) ACQUIRE_SHARED(mu) : mut(mu), locked(true) {
    mu->ReaderLock();
  }

  // Assume mu is held in shared mode, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, adopt_lock_t, shared_lock_t) REQUIRES_SHARED(mu)
    : mut(mu), locked(true) {}

  // Assume mu is not held, implicitly acquire *this and associate it with mu.
  MutexLocker(Mutex *mu, defer_lock_t) EXCLUDES(mu) : mut(mu), locked(false) {}

  // Release *this and all associated mutexes, if they are still held.
  // There is no warning if the scope was already unlocked before.
  ~MutexLocker() RELEASE() {
    if (locked)
      mut->GenericUnlock();
  }

  // Acquire all associated mutexes exclusively.
  void Lock() ACQUIRE() {
    mut->Lock();
    locked = true;
  }

  // Try to acquire all associated mutexes exclusively.
  bool TryLock() TRY_ACQUIRE(true) {
    return locked = mut->TryLock();
  }

  // Acquire all associated mutexes in shared mode.
  void ReaderLock() ACQUIRE_SHARED() {
    mut->ReaderLock();
    locked = true;
  }

  // Try to acquire all associated mutexes in shared mode.
  bool ReaderTryLock() TRY_ACQUIRE_SHARED(true) {
    return locked = mut->ReaderTryLock();
  }

  // Release all associated mutexes. Warn on double unlock.
  void Unlock() RELEASE() {
    mut->Unlock();
    locked = false;
  }

  // Release all associated mutexes. Warn on double unlock.
  void ReaderUnlock() RELEASE() {
    mut->ReaderUnlock();
    locked = false;
  }
};


#ifdef USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES
// The original version of thread safety analysis the following attribute
// definitions.  These use a lock-based terminology.  They are still in use
// by existing thread safety code, and will continue to be supported.

// Deprecated.
#define PT_GUARDED_VAR \
  THREAD_ANNOTATION_ATTRIBUTE__(pt_guarded_var)

// Deprecated.
#define GUARDED_VAR \
  THREAD_ANNOTATION_ATTRIBUTE__(guarded_var)

// Replaced by REQUIRES
#define EXCLUSIVE_LOCKS_REQUIRED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_locks_required(__VA_ARGS__))
```

```
// Replaced by REQUIRES_SHARED
#define SHARED_LOCKS_REQUIRED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_locks_required(__VA_ARGS__))

// Replaced by CAPABILITY
#define LOCKABLE \
  THREAD_ANNOTATION_ATTRIBUTE__(lockable)

// Replaced by SCOPED_CAPABILITY
#define SCOPED_LOCKABLE \
  THREAD_ANNOTATION_ATTRIBUTE__(scoped_lockable)

// Replaced by ACQUIRE
#define EXCLUSIVE_LOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_lock_function(__VA_ARGS__))

// Replaced by ACQUIRE_SHARED
#define SHARED_LOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_lock_function(__VA_ARGS__))

// Replaced by RELEASE and RELEASE_SHARED
#define UNLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(unlock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE
#define EXCLUSIVE_TRYLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(exclusive_trylock_function(__VA_ARGS__))

// Replaced by TRY_ACQUIRE_SHARED
#define SHARED_TRYLOCK_FUNCTION(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(shared_trylock_function(__VA_ARGS__))

// Replaced by ASSERT_CAPABILITY
#define ASSERT_EXCLUSIVE_LOCK(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_exclusive_lock(__VA_ARGS__))

// Replaced by ASSERT_SHARED_CAPABILITY
#define ASSERT_SHARED_LOCK(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(assert_shared_lock(__VA_ARGS__))

// Replaced by EXCLUDE_CAPABILITY.
#define LOCKS_EXCLUDED(...) \
  THREAD_ANNOTATION_ATTRIBUTE__(locks_excluded(__VA_ARGS__))

// Replaced by RETURN_CAPABILITY
#define LOCK_RETURNED(x) \
  THREAD_ANNOTATION_ATTRIBUTE__(lock_returned(x))

#endif  // USE_LOCK_STYLE_THREAD_SAFETY_ATTRIBUTES

#endif  // THREAD_SAFETY_ANALYSIS_MUTEX_H
```