
18.5. Mutexes and Locks

A mutex, or *mutual exclusion*, is an object that helps to control the concurrent access of a resource by providing exclusive access to it. The resource might be an object or a combination of multiple objects. To get exclusive access to the resource, the corresponding thread *locks* the mutex, which prevents other threads from locking that mutex until the first thread *unlocks* the mutex.

18.5.1. Using Mutexes and Locks

Consider that we want to protect concurrent access to an object `val` that is used at various places:

```
int val;
```

A naive approach to synchronize this concurrent access is to introduce a mutex, which is used to enable and control exclusive access:

```
int val;
std::mutex valMutex;     // control exclusive access to val
```

Then, each access has to lock this mutex to get exclusive access. For example, one thread might program the following (note that this is a poor solution, which we will improve):

```
valMutex.lock();   // request exclusive access to val
if (val >= 0) {
    f(val);      // val is positive
}
else {
    f(-val);     // pass negated negative val
}
valMutex.unlock();    // release exclusive access to val
```

Another thread might access the same resource as follows:

```
valMutex.lock();    // request exclusive access to val
++val;
valMutex.unlock();    // release exclusive access to val
```

It's important that all places where concurrent access is possible use the same mutex. This applies to both read and write accesses.

This simple approach can, however, become pretty complicated. For example, you should ensure that an exception, which ends an exclusive access, also unlocks the corresponding mutex. Otherwise, a resource might become locked forever. Also, deadlock scenarios are possible, with two threads waiting for a lock of the other thread before freeing their own lock.

The C++ standard library tries to deal with these problems but can't conceptually solve them all. For example, to deal with exceptions, you should not lock and unlock a mutex yourself. You should use the RAII principle (*Resource Acquisition Is Initialization*), whereby the constructor acquires a resource so that the destructor, which is always called even when an exception causes the end of the lifetime, releases the resource automatically. For this purpose, the C++ standard library provides class `std::lock_guard`:

Click here to view code image

```
int val;
std::mutex valMutex;   // control exclusive access to val
...
std::lock_guard<std::mutex> lg(valMutex);   // lock and automatically unlock
if (val >= 0) {
    f(val);      // val is positive
}
else {
    f(-val);     // pass negated negative val
}
```

Note, however, that locks should be limited to the shortest period possible because they block other code from running in parallel. Because the destructor releases the lock, you might want to insert explicit braces so that the lock gets released before further statements are processed:

Click here to view code image

```
int val;
std::mutex valMutex;     // control exclusive access to val
...
{
    std::lock_guard<std::mutex> lg(valMutex);     // lock and automatically
unlock
```

```
        if (val >= 0) {
            f(val);       // val is positive
        }
        else {
            f(-val);      // pass negated negative val
        }
    } // ensure that lock gets released here
    ...
```

or just:

Click here to view code image

```
    ...
    {
        std::lock_guard<std::mutex> lg(valMutex);     // lock and automatically unlock
        ++val;
    } // ensure that lock gets released here
    ...
```

This is just a first simple example, but you can see that the whole topic can easily become pretty complicated. As usual, programmers should know what they program in concurrent mode. In addition, different mutexes and locks are provided, which are discussed in the upcoming subsections.

A First Complete Example for Using a Mutex and a Lock

Let's look at a first complete example:

Click here to view code image

```
// concurrency/mutex1.cpp

#include <future>
#include <mutex>
#include <iostream>
#include <string>

std::mutex printMutex;  // enable synchronized output with print()

void print (const std::string& s)
{
    std::lock_guard<std::mutex> l(printMutex);
    for (char c : s) {
        std::cout.put(c);
    }
    std::cout << std::endl;
}

int main()
{
    auto f1 = std::async (std::launch::async,
                          print, "Hello from a first thread");
    auto f2 = std::async (std::launch::async,
                          print, "Hello from a second thread");
    print("Hello from the main thread");
}
```

Here, `print()` writes all characters of a passed string to the standard output. Thus, without a lock, the output might be:[19]

[19] The fact that each character is written on its own with `put()` forces the behavior of getting interleaved characters when multiple parallel writes are performed. When writing each string as a whole, implementations often will not mix characters, but even this isn't guaranteed.

```
HHelHello from a second thread
ello from a first thread
lo from the main thread
```

or:

Click here to view code image

```
HelloHello fHello from a second ro from am th fthe main irrethreadstad
 thr
ead
```

To synchronize the output in a way that each call of `print()` exclusively writes its characters, we introduce a mutex for the print operation and a lock guard, which locks the corresponding protected section:

Click here to view code image

```
std::mutex printMutex;   // enable synchronized output with print()
...
void print (const std::string& s)
{
    std::lock_guard<std::mutex> l(printMutex);
    ...
}
```

Now the output is simply something like this:

```
Hello from a first thread
Hello from the main thread
Hello from a second thread
```

This output also is possible (but not guaranteed) when no lock is used.

Here, the `lock()` of the mutex, called by the constructor of the lock guard, blocks if the resource is acquired already. It blocks until access to the protected section is available again. However, the order of locks is still undefined. Thus, the three outputs might still be written in arbitrary order.

Recursive Locks

Sometimes, the ability to lock recursively is required. Typical examples are active objects or monitors, which contain a mutex and take a lock inside every public method to protect data races corrupting the internal state of the object. For example, a database interface might look as follows:

```
class DatabaseAccess
{
  private:
    std::mutex dbMutex;
    ... // state of database access
  public:
    void createTable (...)
    {
        std::lock_guard<std::mutex> lg(dbMutex);
        ...
    }
    void insertData (...)
    {
        std::lock_guard<std::mutex> lg(dbMutex);
        ...
    }
    ...
};
```

When we introduce a public member function that might call other public member functions, this can become complicated:

Click here to view code image

```
void createTableAndInsertData (...)
{
    std::lock_guard<std::mutex> lg(dbMutex);
    ...
    createTable(...);   // ERROR: deadlock because dbMutex is locked again
}
```

Calling `createTableAndInsertData()` will result in a deadlock because after locking `dbMutex`, the call of `createTable()` will try to lock `dbMutex` again, which will block until the lock of `dbMutex` is available, which will never happen because `createTableAndInsertData()` will block until `createTable()` is done.

The C++ standard library permits the second attempt to throw a `std::system_error` (see Section 4.3.1, page 43) with the error code `resource_deadlock_would_occur` (see Section 4.3.2, page 45) if the platform detects such a deadlock. But this is not required and is often not the case.

By using a `recursive_mutex`, this behavior is no problem. This mutex allows multiple locks by the same thread and releases the lock when the last corresponding `unlock()` call is called:

Click here to view code image

```
class DatabaseAccess
{
  private:
    std::recursive_mutex dbMutex;
    ... // state of database access
  public:
    void insertData (...)
    {
```

```
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void insertData (...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
    }
    void createTableAndinsertData (...)
    {
        std::lock_guard<std::recursive_mutex> lg(dbMutex);
        ...
        createTable(...); // OK: no deadlock
    }
    ...
};
```

Tried and Timed Locks

Sometimes a program wants to acquire a lock but doesn't want to block (forever) if this is not possible. For this situation, mutexes provide a `try_lock()` member function that *tries* to acquire a lock. If it succeeds, it returns `true`; if not, `false`.

To still be able to use a `lock_guard` so that any exit from the current scope unlocks the mutex, you can pass an additional argument `adopt_lock` to its constructor:

Click here to view code image

```
std::mutex m;

// try to acquire a lock and do other stuff while this isn't possible
while (m.try_lock() == false) {
    doSomeOtherStuff();
}
std::lock_guard<std::mutex> lg(m,std::adopt_lock);
...
```

Note that `try_lock()` might fail spuriously. That is, it might fail (return `false`) even if the lock is not taken.[20]

[20] This behavior is provided for memory-ordering reasons but is not widely known. Thanks to Hans Boehm and Bartosz Milewski for pointing it out.

To wait only for a particular amount of time, you can use a timed mutex. The special mutex classes `std::timed_mutex` and `std::recursive_timed_mutex` additionally allow calling `try_lock_for()` or `try_lock_until()` to wait for at most a specified duration of time or until a specified point in time has arrived. This, for example, might help if you have real-time requirements or want to avoid possible deadlock situations. For example:

```
std::timed_mutex m;

// try for one second to acquire a lock
if (m.try_lock_for(std::chrono::seconds(1))) {
    std::lock_guard<std::timed_mutex> lg(m,std::adopt_lock);
    ...
}
else {
    couldNotGetTheLock();
}
```

Note that `try_lock_for()` and `try_lock_until()` usually will differ when dealing with system-time adjustments (see Section 5.7.5, page 160, for details).

Dealing with Multiple Locks

Usually a thread should lock only one mutex at a time. However, it is sometimes necessary to lock more than one mutex (for example, to transfer data from one protected resource to another). In that case, dealing with the lock mechanisms introduced so far can become complicated and risky: You might get the first but not the second lock, or deadlock situations may occur if you lock the same locks in a different order.

The C++ standard library, therefore, provides convenience functions to try to lock multiple mutexes. For example:

Click here to view code image

```
std::mutex m1;
std::mutex m2;
...
{
    std::lock (m1, m2); // lock both mutexes (or none if not possible)
    std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
    std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
    ...
```

```
} // automatically unlock all mutexes
```

The global `std::lock()` locks all mutexes passed as arguments, blocking until all mutexes are locked or until an exception is thrown. In the latter case, it unlocks mutexes already successfully locked. As usual, after successful locking, you can and should use a lock guard initialized with `adopt_lock` as second argument to ensure that, in any case, the mutexes are unlocked when leaving the scope. Note that this `lock()` provides a deadlock-avoidance mechanism, which, however, means that the order of locking inside a multiple lock is undefined.

In the same way, you can *try* to acquire multiple locks without blocking if not all locks are available. The global `std::try_lock()` returns `-1` if all locks were possible. If not, the return value is the zero-based index of the first failed lock. In that case, all succeeded locks are unlocked again. For example:

Click here to view code image

```
std::mutex m1;
std::mutex m2;

int idx = std::try_lock (m1, m2);    // try to lock both mutexes
if (idx < 0) {   // both locks succeeded
      std::lock_guard<std::mutex> lockM1(m1,std::adopt_lock);
      std::lock_guard<std::mutex> lockM2(m2,std::adopt_lock);
      ...
}  // automatically unlock all mutexes
else {
   // idx has zero-based index of first failed lock
   std::cerr << "could not lock mutex m" << idx+1 << std::endl;
}
```

Note that this `try_lock()` does not provide a deadlock-avoidance mechanism. Instead, it guarantees that the locks are tried in the order of the passed arguments.

Note also that calling `lock()` or `try_lock()` without adopting the locks by a guard is usually not what was intended. Although the code looks like it creates locks that are released automatically when leaving the scope, this is not the case. The mutexes will remain locked:

Click here to view code image

```
std::mutex m1;
std::mutex m2;
...
{
    std::lock (m1, m2);     // lock both mutexes (or none if not possible)
    // no lock adopted
    ...
}
... // OOPS: mutexes are still locked !!!
```

Class `unique_lock`

Besides class `lock_guard<>`, the C++ standard library provides class `unique_lock<>`, which is a lot more flexible when dealing with locks for mutexes. Class `unique_lock<>` provides the same interface as class `lock_guard<>`, plus the ability to program explicitly when and how to lock or unlock its mutex. Thus, this lock object may or may not have a mutex locked (also known as *owning* a mutex). This differs from a `lock_guard<>`, which always has an object locked throughout its lifetime.[21] In addition, for unique locks you can query whether the mutex is currently locked by calling `owns_lock()` or `operator bool()`.

[21] The name *unique* lock explains where this behavior comes from. As with unique pointers (see Section 5.2.5, page 98), you can move locks between scopes, but it is guaranteed that only one lock at a time owns a mutex.

The major advantage of this class still is that when the mutex is locked at destruction time, the destructor automatically calls `unlock()` for it. If no mutex is locked, the destructor does nothing.

Compared to class `lock_guard`, class `unique_lock` provides the following supplementary constructors:

- You can pass `try_to_lock` for a nonblocking attempt to lock a mutex:

Click here to view code image

```
std::unique_lock<std::mutex> lock(mutex,std::try_to_lock);
...
if (lock) {   // if lock was successful
    ...
}
```

- You can pass a duration or timepoint to the constructor to try to lock for a specific period of time:

[Click here to view code image]

```
std::unique_lock<std::timed_mutex> lock(mutex,
                                        std::chrono::seconds(1));
...
```

• You can pass `defer_lock` to initialize the lock without locking the mutex (yet):

[Click here to view code image]

```
std::unique_lock<std::mutex> lock(mutex,std::defer_lock);
...
lock.lock();  // or (timed) try_lock()
...
```

The `defer_lock` flag can, for example, be used to create one or multiple locks and lock them later:

[Click here to view code image]

```
std::mutex m1;
std::mutex m2;

std::unique_lock<std::mutex> lockM1(m1,std::defer_lock);
std::unique_lock<std::mutex> lockM2(m2,std::defer_lock);
...
std::lock (m1, m2);  // lock both mutexes (or none if not possible)
```

In addition, class `unique_lock` provides the ability to `release()` its mutex or to transfer the ownership of a mutex to another lock. See Section 18.5.2, page 1000, for details.

With both a `lock_guard` and a `unique_lock`, we can now implement a naive example, where one thread waits for another by polling a *ready flag*:

[Click here to view code image]

```
#include <mutex>
...
bool readyFlag;
std::mutex readyFlagMutex;

void thread1()
{
    // do something thread2 needs as preparation
    ...
    std::lock_guard<std::mutex> lg(readyFlagMutex);
    readyFlag = true;
}

void thread2()
{
    // wait until readyFlag is true (thread1 is done)
    {
        std::unique_lock<std::mutex> ul(readyFlagMutex);
        while (!readyFlag) {
            ul.unlock();
            std::this_thread::yield();  // hint to reschedule to the next thread
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            ul.lock();
        }
    } // release lock

    // do whatever shall happen after thread1 has prepared things
    ...
}
```

Two comments on typical questions this code might raise:

• If you wonder why we use a mutex to control the access to read and write the `readyFlag`, remember the rule introduced at the beginning of this chapter: Any concurrent access with at least one write should be synchronized. See Section 18.4, page 982, and Section 18.7, page 1012, for a detailed discussion about this.

• If you wonder that no `volatile` is necessary here to declare `readyFlag` to avoid that multiple attempts in `thread2()` to read it are not optimized away note the following: These attempts to read `readyFlag` happen inside a *critical section*, defined between the setting and releasing of a lock. Such code is not allowed to get optimized in a way that the read (or a write) is moved outside the critical section. So the reads of `readyFlag` must effectively happen here:

  – At the beginning of the loop, between the declaration of `ul` and the first call of `unlock()`

– Inside the loop, between any call of `lock()` and `unlock()`

– At the end of the loop, between the last call of `lock()` and the destruction of `ul`, which unlocks the mutex if locked

Nevertheless, such a *polling* for a fulfilled condition is usually not a good solution. A better approach is to use *condition variables*. See Section 18.6.1, page 1003, for details.

18.5.2. Mutexes and Locks in Detail

Mutexes in Detail

The C++ standard library provides the following mutex classes (see Table 18.6):

- Class `std::mutex` is a simple mutex that can be locked only once by one thread at a time. If it is locked, any other `lock()` will block until the mutex is available again and `try_lock()` will fail.
- Class `std::recursive_mutex` is a mutex that allows multiple locks at the same time by the same thread. The typical application of such a lock is where functions acquire a lock and internally call another function, which also acquires the same lock again.
- Class `std::timed_mutex` is a simple mutex that additionally allows you to pass a duration or a timepoint that defines how long it tries to acquire a lock. For this, `try_lock_for()` and `try_lock_until()` are provided.
- Class `std::recursive_timed_mutex` is a mutex that allows multiple locks by the same thread with optional timeouts.

Table 18.6. Overview of Mutexes and Their Abilities

Operation	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex
lock()	Acquires mutex (blocks if not available)			
try_lock()	Acquires mutex (returns false if not available)			
unlock()	Unlocks locked mutex			
try_lock_for()	–	–	Tries to acquire a lock for a duration of time	
try_lock_until()	–	–	Tries to acquire a lock until a timepoint	
multiple locks	No	Yes (same thread)	No	Yes (same thread)

Table 18.7 lists the mutex operations, if available.

Table 18.7. Operations of Mutex Classes, If Available

Operation	Effect
mutex *m*	Default constructor; creates an unlocked mutex
m.~***mutex***()	Destroys the mutex (must not be locked)
m.lock()	Locks the mutex (blocks for lock; error if locked and not recursive)
m.try_lock()	Tries to lock the mutex (returns true if lock successful)
m.try_lock_for(*dur*)	Tries to lock for duration *dur* (returns true if lock successful)
m.try_lock_until(*tp*)	Tries to lock until timepoint *tp* (returns true if lock successful)
m.unlock()	Unlocks the mutex (undefined behavior if not locked)
m.native_handle()	Returns a platform-specific type native_handle_type for nonportable extensions

`lock()` might throw a `std::system_error` (see Section 4.3.1, page 43) with the following error codes (see Section 4.3.2, page 45):

- `operation_not_permitted`, if the thread does not have the privilege to perform the operation
- `resource_deadlock_would_occur`, if the platform detects that a deadlock would occur
- `device_or_resource_busy`, if the mutex is already locked and blocking is not possible

The behavior of a program is undefined if it unlocks a mutex object it doesn't own, destroys a mutex object owned by any thread, or if a thread terminates while owning a mutex object.

Note that `try_lock_for()` and `try_lock_until()` usually will differ when dealing with system-time adjustments (see Section 5.7.5, page 160, for details).

Class `lock_guard` in Detail

Class `std::lock_guard`, introduced in Section 18.5.1, page 989, provides a very small interface to ensure that a locked mutex gets always freed when leaving the scope (see Table 18.8). Throughout its lifetime, it is always associated with a lock either explicitly requested or adopted at construction time.

Table 18.8. Operations of Class `lock_guard`

Operation	Effect
lock_guard `lg(`*m*`)`	Creates a lock guard for the mutex *m* and locks it
lock_guard `lg(`*m*`,adopt_lock)`	Creates a lock guard for the already locked mutex *m*
`lg.~`*lock_guard*`()`	Unlocks the mutex and destroys the lock guard

Class `unique_lock` in Detail

Class `std::unique_lock`, introduced in Section 18.5.1, page 996, provides a lock guard for a mutex that does not necessarily have to be locked (*owned*). It provides the interface listed in Table 18.9. If it locks/owns a mutex at destruction time, it will `unlock()` it. But you can control explicitly whether it has an associated mutex and whether this mutex is locked. You can also try to lock the mutex with or without timeouts.

Table 18.9. Operations of Class `unique_lock`

Operation	Effect
unique_lock `l`	Default constructor; creates a lock not associated with a mutex
unique_lock `l(`*m*`)`	Creates a lock guard for the mutex *m* and locks it
unique_lock `l(`*m*`,adopt_lock)`	Creates a lock guard for the already locked mutex *m*
unique_lock `l(`*m*`,defer_lock)`	Creates a lock guard for the mutex *m* without locking it
unique_lock `l(`*m*`,try_lock)`	Creates a lock guard for the mutex *m* and tries to lock it
unique_lock `l(`*m*`,`*dur*`)`	Creates a lock guard for the mutex *m* and tries to lock it for duration *dur*
unique_lock `l(`*m*`,`*tp*`)`	Creates a lock guard for the mutex *m* and tries to lock it until timepoint *tp*
unique_lock `l(`*rv*`)`	Move constructor; moves lock state from *rv* to `l` (*rv* has no associated mutex anymore)
`l.~`*unique_lock*`()`	Unlocks the mutex, if any locked, and destroys the lock guard
unique_lock `l = `*rv*	Move assignment; moves the lock state from *rv* to `l` (*rv* has no associated mutex anymore)
`swap(l1,l2)`	Swaps locks
`l1.swap(l2)`	Swaps locks
`l.release()`	Returns a pointer to the associated mutex and releases it
`l.owns_lock()`	Returns `true` if an associated mutex is locked
`if (l)`	Checks whether an associated mutex is locked
`l.mutex()`	Returns a pointer to the associated mutex
`l.lock()`	Locks the associated mutex
`l.try_lock()`	Tries to lock the associated mutex (returns `true` if lock successful)
`l.try_lock_for(`*dur*`)`	Tries to lock the associated mutex for duration *dur* (returns `true` if lock successful)
`l.try_lock_until(`*tp*`)`	Tries to lock the associated mutex until timepoint *tp* (returns `true` if lock successful)
`l.unlock()`	Unlocks the associated mutex

`lock()` might throw a `std::system_error` (see Section 4.3.1, page 43) with the error codes listed for `lock()` for mutexes (see page 999). `unlock()` might throw a `std::system_error` with the error code `operation_not_permitted` if the unique lock isn't locked.

18.5.3. Calling Once for Multiple Threads

Sometimes multiple threads might not need some functionality that should get processed whenever the first thread needs it. A typical example is lazy initialization: The first time one of the threads needs something that has to get processed, you process it (but not before, because you want to save the time to process it if it is not needed).

The usual approach with single-threaded environments is simple: A Boolean flag signals whether the functionality was called already:

```
bool initialized = false;  // global flag
...
if (!initialized) {         // initialize if not initialized yet
    initialize();
    initialized = true;
}
```

or

```
static std::vector<std::string> staticData;

void foo()
{
    if (staticData.empty()) {
        staticData = initializeStaticData();
    }
    ...
}
```

Such code doesn't work in a multithreaded context, because data races might occur if two or more threads check whether the initialization didn't happen yet and start the initialization then. Thus, you have to protect the area for the check and the initialization against concurrent access.

As usual, you can use mutexes for it, but the C++ standard library provides a special solution for this case. You simply use a `std::once_flag` and call `std::call_once` (also provided by `<mutex>`):

Click here to view code image

```
std::once_flag oc;              // global flag
...
std::call_once(oc,initialize);  // initialize if not initialized yet
```

or:

Click here to view code image

```
static std::vector<std::string> staticData;

void foo()
{
   static std::once_flag oc;
   std::call_once(oc,[]{
                          staticData = initializeStaticData();
                      });
   ...
}
```

As you can see, the first argument passed to `call_once()` must be the corresponding `once_flag`. The further arguments are the usual arguments for *callable objects*: function, member function, function object, or lambda, plus optional arguments for the function called (see Section 4.4, page 54). Thus, lazy initialization of an object used in multiple threads might look as follows:

Click here to view code image

```
class X {
  private:
    mutable std::once_flag initDataFlag;
    void initData() const;
  public:
    data getData () const {
        std::call_once(initDataFlag,&X::initData,this);
        ...
    }
};
```

In principle, you can call different functions for the same once flag. The once flag that is passed to `call_once()` as first argument is what ensures that the passed functionality is performed only once. So, if the first call was successful, further calls with the same once flag won't call the passed functionality even if that functionality is different.

Any exception caused by the called functionality is also thrown by `call_once()`. In that case, the "first" call is considered not to be successful, so the next `call_once()` might still execute the passed functionality.[22]

[22] The standard also specifies that `call_once()` might throw a `std::system_error` if the

`once_flag` argument is no longer “valid” (i.e., destructed). However, this statement is considered to be a mistake because passing a destructed once flag anyway is either not possible or results in undefined behavior.