**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.

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.

# Using Mutexes and Locks

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

A naive approach to synchronize this concurrent access is to introduce a mutex. Then, each access has to lock this mutex to get exclusive access.

**Example**

one thread might program the following

int val;

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

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

This approach can 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.

## RAII principle

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.

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.

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:

...

{

          std::lock\_guard<std::mutex> lg(valMutex); // lock and automatically unlock

          ++val;

}        // ensure that lock gets released here

...

## Example

#include <future>

#include <mutex>

#include <iostream>

#include <string>

std::mutex printMutex;   // enable synchronized output with thr\_routine()

void thr\_routine(const std::string& s) {

          //std::lock\_guard<std::mutex> l(printMutex);       // case - 2

          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, thr\_routine, "Hello from a first thread");

          auto f2 = std::async(std::launch::async, thr\_routine, "Hello from a second thread");

          thr\_routine("Hello from the main thread");

          return 0;

}

Output:

Hello from the main thread

Hello from a second thread

Hello from a first thread

Output:

Hello froHmell o frao mfirs tt hteh rmeaaidn thread

Hello from a second thread

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

Recursive locks are reuired for 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.

### Example Deadlock

A database interface might look as follows:

class DatabaseAccess {

          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);

                   ...

          }

          ...

};

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.

C++ standard library permits the second attempt to throw a std::system\_error with the error code resource\_deadlock\_would\_occur if the platform detects such a deadlock.

### Example Recursive mutex

Recursive mutex allows multiple locks by the same thread and releases the lock when the last corresponding unlock() call is called.

class DatabaseAccess {

          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

### try\_lock

try\_lock() member function tries to acquire a lock. If it succeeds, it returns true; if not, false.

To still be able to use a lock\_guard, you can pass an additional argument adopt\_lock to its constructor.

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);

...

### 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.

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();

}

## Dealing with Multiple Locks

It is sometimes necessary to lock more than one mutex.

C++ standard library, therefore, provides convenience functions to try to lock multiple mutexes.

### Example

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

This lock() provides a deadlock-avoidance mechanism, which, however, means that the order of locking inside a multiple lock is undefined.

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.

### Example

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;

}

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:

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

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.

This differs from a lock\_guard<>, which always has an object locked throughout its lifetime.

In addition, for unique locks you can query whether the mutex is currently locked by calling owns\_lock() or operator bool().

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.

In addition, class unique\_lock provides the ability to release() its mutex or to transfer the ownership of a mutex to another lock.

Compared to class lock\_guard, class unique\_lock provides the following supplementary constructors:

1.   You can pass try\_to\_lock for a nonblocking attempt to lock a mutex:

std::unique\_lock<std::mutex> lock(mutex, std::try\_to\_lock);

...

if (lock) {     // if lock was successful

...

}

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

std::unique\_lock<std::timed\_mutex> lock(mutex, std::chrono::seconds(1));

...

3.   You can pass defer\_lock to initialize the lock without locking the mutex (yet):

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:

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)

### Example

One thread waits for another by polling a ready flag:

#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

          ...

}

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:

1.   At the beginning of the loop, between the declaration of ul and the first call of unlock()

2.   Inside the loop, between any call of lock() and unlock()

3.   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.

# Mutexes and Locks in Detail

## Mutexes in Detail

C++ standard library provides the following mutex classes:

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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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) |

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.

|  |  |
| --- | --- |
| **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 |

**Note:**

lock() might throw a std::system\_error with the following error codes:

1.   **operation\_not\_permitted**, if the thread does not have the privilege to perform the operation

2.   **resource\_deadlock\_would\_occur**, if the platform detects that a deadlock would occur

3.   **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.

## Class lock\_guard in Detail

Class std::lock\_guard provides a very small interface to ensure that a locked mutex gets always freed when leaving the scope.

|  |  |
| --- | --- |
| **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 provides a lock guard for a mutex that does not necessarily have to be locked.

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.

lock() might throw a std::system\_error.

unlock() might throw a std::system\_error with the error code operation\_not\_permitted if the unique lock isn’t locked.

|  |  |
| --- | --- |
| 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 |

# 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.

## In single-threaded environments

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.

## In multiple-threaded environments

Protect the area for the check and the initialization against concurrent access

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.

std::once\_flag oc; // global flag

...

std::call\_once(oc,initialize); // initialize if not initialized yet

or:

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

void foo() {

static std::once\_flag oc;

std::call\_once(oc, [] { staticData = initializeStaticData();});

...

}

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.

## Example

Lazy initialization of an object used in multiple threads might look as follows:

class X {

mutable std::once\_flag initDataFlag;

void initData() const;

public:

data getData () const {

std::call\_once(initDataFlag,&X::initData,this);

...

}

};

# END