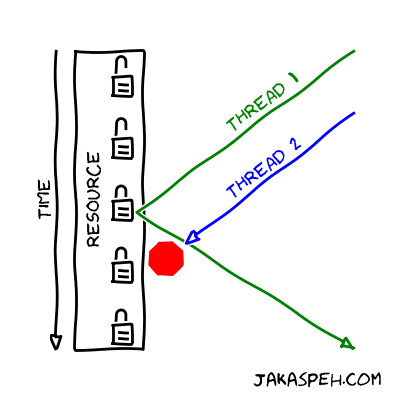
Difference between std::lock\_guard and std::unique\_lock

Feb 8, 2016

One way of preventing data races between the threads is to use mutexes.

A mutex is usually associated with a resource. The thread, which locks the mutex, has granted access to the resource. No other thread can then lock the mutex because it is already locked (look figure below). Consequently, no other thread has an access to the resource guarded by the locked mutex. This is the mutual exclusion: only one thread has access to the resource at any given time.



We already spoke about the problems which appear when using mutexes in our code: remember [Mutex and deadlock](http://jakascorner.com/blog/2016/01/deadlock.html). There, we introduced the std::lock\_guard class. But when we synchronized threads with a condition variable, we used similar class: std::unique\_lock. What is the difference between these two classes?

The difference

One of the differences between std::lock\_guard and std::unique\_lock is that the programmer **is able** to unlock std::unique\_lock, but she/he **is not able** to unlock std::lock\_guard. Let’s explain it in more detail.

std::lock\_guard

If you have an object

std**::**lock\_guard guard1(mutex);

then the constructor of guard1 locks the mutex. At the end of guard1’s life, the destructor unlocks the mutex. There is no other possibility. In fact, the std::lock\_guard class doesn’t have any other member function.

std::unique\_lock

On the other hand, we have an object of std::unique\_lock.

std**::**unique\_lock guard2(mutex);

There are similarities with std::lock\_guard class. The constructor of guard2 also locks the mutex and the destructor of guard2 also unlocks the mutex. But the std::unique\_lock has additional functionalities.

The programmer is able to unlock the mutex with the help of the guard object

guard2.unlock();

This means that the programmer can unlock the mutex before the guard2’s life ends. After the mutex was unlocked, the programmer can also lock it again

guard2.lock();

We should mention that the std::unique\_lock has also some other member functions. You can look it up [here](http://en.cppreference.com/w/cpp/thread/unique_lock).

When to use std::unique\_lock ?

There are at least two reasons for using std::unique\_lock. Sometimes we are forced to use it: other functions require it as an input. And other times using std::unique\_lock allows us to have more parallelizable code.

Higher parallelization

Let’s say that we have a long function. First part of the function accesses some shared resource and the second part locally processes the resource.

std**::**vector**<** **int** **>** vector; *// shared between threads*

...

**int** function(...)

{

...

Getting **int** from the shared vector.

...

...

Long, complicated computation with **int**.

This part does not depend on the vector.

...

}

A mutex must be locked just in the first part of the function, because we access the element of the vector. In the second part, the mutex doesn’t need to be locked anymore (because we don’t access any shared variable).

std**::**vector**<** **int** **>** vector; *// shared between threads*

std**::**mutex mutex;

...

**int** function(...)

{

...

std**::**unique\_lock guard(mutex);

Getting **int** from the shared vector.

...

...

guard.unlock();

Long, complicated computation with **int**.

This part does not depend on the vector.

...

}

In fact, it is preferable that the mutex is not locked in the second part, because then other threads can lock it. In principle, we would like that the locks last as little time as possible. This minimizes the time when threads are waiting to get a lock on the mutex and not doing any useful work. We obtain more parallelizable code.

Using functions that requires std::unique\_lock

In [Condition variable](http://jakascorner.com/blog/2016/01/condition-variable.html), we had to use the std::unique\_lock, because std::condition\_variable::wait(...) requires std::unique\_lock as an input.

The std::condition\_variable::wait(...) unlocks the mutex and waits for the std::condition\_variable.notify\_one() member function call. Then, wait(...) reacquires the lock and proceeds.

We recognize that wait(...) member function requires std::unique\_lock. The function can not use usual std::lock\_guard, because it unlocks/locks the mutex.

When to use std::lock\_guard ?

The std::unique\_lock has all of the functionalities of the std::lock\_guard. Everything which is possible to do with std::lock\_guard is also possible to do with std::unique\_lock. So, when should we use std::lock\_guard?

The rule of thumb is to always use std::lock\_guard. But if we need some higher level functionalities, which are available by std::unique\_lock, then we should use the std::unique\_lock.