Temporarily dropping a lock: The anti-lock pattern

devblogs.microsoft.com/oldnewthing/20240814-00

August 14, 2024

Raymond Chen

There is a common pattern in C++ of using an RAII type to manage a synchronization primitive. There are different versions of this, but they all have the same basic pattern:

- Creating the object from a synchronization object: Locks the synchronization object.
- Destructing the object: Unlocks the synchronization object.

These types go by various names, like `std::lock_guard`, `std::unique_lock`, or `std::coped_lock`, and specific libraries may have versions for their own types, such as C++/WinRT's `winrt::slim_lock_guard` and WIL's `wil::rwlock_release_exclusive_scope_exit` (which you thankfully never actually write out; just use `auto`).

One thing that is missing from most standard libraries, however, is the *anti*-lock.

The idea of the anti-lock is that it counteracts an active lock.

```
template<typename Mutex>
struct anti_lock
{
    anti_lock() = default;

    explicit anti_lock(Mutex& mutex)
    : m_mutex(std::addressof(mutex)) {
        if (m_mutex) m_mutex->unlock();
    }

private:
    struct anti_lock_deleter {
        void operator()(Mutex* mutex) { mutex->lock(); }
    };

    std::unique_ptr<Mutex, anti_lock_deleter> m_mutex;
};
```

The anti-lock *unlocks* a mutex at construction and locks it at destruction. Here's an example:

```
void Widget::DoSomething()
{
    auto guard = std::lock_guard(m_mutex);

    ⟦ do stuff under the lock ⟧

    int cost;
    if (m_isStandard) {
        cost = GetStandardCost();
    } else {
        // Drop the lock temporarily while we call out.
        auto anti_guard = anti_lock(m_mutex);
        cost = m_callback->GetCost();
    }();

    // We are back under the lock.
    ⟦ do more stuff under the lock ⟧
}
```

The idea here is that you know you are running some code that acquires a lock, but you need to drop the lock temporarily, and then reacquire it afterward. The reason for dropping the lock might be that you are calling out to another component and don't want to create a deadlock.

For example, commenter Joshua Hudson could have used this around all of the `co_await`s.

```
winrt::fire_and_forget DoSomething()
{
    auto guard = std::lock_guard(m_mutex);

    step1();

    // All co_awaits must be under an anti-lock.
    int cost = [&] {
        auto anti_guard = anti_lock(m_mutex);
        return co_await GetCostAsync();
    }();

    step2(cost);
}
```

For extra safety, you might require that the anti-lock be given the lock guard that it is counteracting.

```
template<typename Guard>
struct anti_lock
{
    using mutex_type = typename Guard::mutex_type;

    anti_lock() = default;

    explicit anti_lock(Guard& guard)
    : m_mutex(guard.mutex())
        if (m_mutex) m_mutex->unlock();
    }

private:
    struct anti_lock_deleter {
        void operator()(mutex_type* mutex) { m_mutex->lock(); }
    };

    std::unique_ptr<mutex_type, anti_lock_deleter> m_mutex;
};
```

Being given the lock guard means that we can also make it so that the anti-lock of a non-owning guard is a non-owning anti-lock. The negative of zero is zero.

Being given the lock guard also makes it slightly more noticeable to the caller that the anti-lock might mess with the lock state.

Now, an anti-lock sounds weird, but you could very well be using it without realizing it: `std::condition_variable`s are secretly anti-locks. They enter with the lock held, then drop the lock while blocked, then reacquire the lock when unblocked.

Here's another scenario where you may want to use an anti-lock:

```
void DoSomething()
{
    // Hold the lock while we check m_nextQuery
    std::unique_lock lock(m_mutex);
    while (auto query = std::exchange(m_nextQuery, nullptr)) {
        // Drop the lock while we do work
        anti_lock anti(lock);
        refresh_from_query(query);
        // Reacquire the lock before rechecking m_nextQuery
    }
}
```

You need to hold the mutex while checking if there is a new query (because that mutex protects the code that sets the new query), but you can drop the mutex while you process the query.

One downside of the anti-lock is that if you have an early return, the mutex is re-locked (when the anti-lock destructs) and then unlocked (when the outer guard destructs). This is hard to fix because there's no guarantee that the outer guard is going to destruct when the anti-lock destructs:

```
void DoSomething()
{
    // Hold the lock while we check m_nextQuery
    std::unique_lock lock(m_mutex);
    while (auto query = std::exchange(m_nextQuery, nullptr)) {
        try {
            // Drop the lock while we do work
            anti_lock anti(lock);
            refresh_from_query(query);
            // Reacquire the lock before rechecking m_nextQuery
        } CATCH_LOG(); // log refresh failures but don't stop
    }
}
```

If you can live with this suboptimal behavior (which presumably is infrequent), the anti-lock is pretty handy.

Bonus chatter: The anti-lock does require you to know for sure that the lock is held exactly once. If it's not held at all, then your anti-lock is unlocking a mutex that isn't even locked, which is not allowed. And if it's held twice (allowed by mutex classes such as `std::shared_mutex` and `std::recursive_mutex`), then your anti-lock only counteracts one of the locks, leave the other lock still active.

And of course anti-locks complicate lock analysis. If you use an anti-lock to counteract a lock held by the caller, then this invalidates the assumption that holding a lock across a function call protects the state guarded by the lock.