The COM static store, part 3: Avoiding creation of an expensive temporary when setting a singleton

devblogs.microsoft.com/oldnewthing/20210210-06

February 10, 2021

Raymond Chen

Last time, we looked at one way to avoid a race condition when initializing a singleton in the COM static store. But it did create the possibility of creating an object that might be thrown away, and that could be a problem if the object is expensive to construct, or if other circumstances prevent you from creating more than one of those objects.

In that case, you can expand the lock to cover the construction of the `Thing`, and construct it only if you're sure you're going to need it.

```
Thing GetSingletonThing()
{
    auto props = CoreApplication::Properties();
    if (auto found = props.TryLookup(L"Thing")) {
        return found.as<Thing>();
    }
    auto guard = std::lock_guard(m_lock);
    if (auto found = props.TryLookup(L"Thing")) {
        return found.as<Thing>();
    }
    auto thing = MakeAThing();
    props.Insert(L"Thing", thing);
    return thing;
}
```

This avoids the creation of a throwaway `Thing`, but it does come at a cost: Since the `Thing` is created under the lock, its constructor is at risk of deadlocking if it acquires its own locks or performs cross-thread operations.

Suppose there's another lock *L*, and the caller of `GetSingletonThing` owns that lock. The `GetSingletonThing` function sees that there is no `Thing` yet, so it takes its own private lock, and then constructs a new `Thing`. If the `Thing` constructor also attempts to acquire lock *L*, and the lock *L* is non-recursive, then you have recursive acquisition of *L*, which is formally undefined behavior.

Even if the lock *L* allows recursive acquisition, you can still deadlock:

Thread 1	Thread 2
Acquire lock *L*	Call `GetSingletonThing`
Call `GetSingletonThing`	Object doesn’t exist yet
Object doesn’t exist yet	Acquire lock *m_lock*
Wait for lock *m_lock*	Object still doesn’t exist yet
	Construct a new `Thing`
	Wait for lock *L*

Here we hit a classic deadlock, where each thread holds one lock and is waiting for the other one.

But even if there is no lock *L*, you can still run into problems if the construction of `Thing` requires cross-thread operations.

Thread 1	Thread 2
	Call `GetSingletonThing`
Call `GetSingletonThing`	Object doesn’t exist yet
Object doesn’t exist yet	Acquire lock *m_lock*
Wait for lock *m_lock*	Object still doesn’t exist yet
	Construct a new `Thing`
	Send a request to Thread 1 to do some work

This time, Thread 2 is waiting for Thread 1 to do some work so it can finish constructing the `Thing`, but Thread 1 cannot do that work because it is waiting for the lock that protects `Thing` construction.

I’ve seen all of these types of deadlocks in production code. They hit rarely, but when they do, everybody has a bad day. Resolving the problem can be complicated because the locks or cross-thread operations are deeply embedded in the architecture, and a lot of refactoring has to be done to avoid dangerous operations while holding a lock.

So yeah, be extremely mindful about what you do while holding a lock. Don’t call out to foreign code while holding a lock.

Okay, enough about deadlocks. We'll look at some ways of optimizing the COM singleton pattern next time.

Raymond Chen

Follow