The COM static store, part 2: Race conditions in setting a singleton

devblogs.microsoft.com/oldnewthing/20210209-00

February 9, 2021

Raymond Chen

Last time, we learned about the COM static store and how you can use it to manage objects whose lifetime should be tied to the lifetime of COM, rather than to the lifetime of the process. And I noted that there were some loose ends that needed to be tidied.

If two threads both try to access the singleton at the same time, there's a race condition where both of them see that there is no object in the cache, so both threads create an object, put it in the cache, and return it. This means that one thread's object is *actually* in the cache, and the other thread's object is just hanging around. If the object has any state, then the losing thread will mutate the object state, but those changes won't be visible to anyone else.

To avoid this problem, we need a lock. Let's try this:

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

If we don't see an object in the cache, we create a fresh object, but before putting it into the cache, we check if another thread beat us to it. If so, then we throw away the one we had created and switch to the one that's already in the cache.

If there is still nothing in the cache, then our object is the one that goes into the cache.

This algorithm assumes that creating a `Thing` and throwing it away is relatively harmless. If creating a `Thing` has side effects that prevent multiple instances,[1] or if it entails excessive computation or other resources requirements, you may not want to create one if there's a chance you're just going to throw it away.

We'll try to solve that problem next time.

Bonus chatter: May as well take this time to make the function more reusable.

```
template<typename T, typename Maker>
T GetSingleton(std::wstring_view name, Maker&& maker)
{
    auto props = CoreApplication::Properties();
    if (auto found = props.TryLookup(name)) {
        return winrt::unbox_value<T>(found);
    }
    auto value = Maker();
    static winrt::slim_mutex lock;
    winrt::slim_lock_guard const guard{ lock };
    found = props.TryLookup(name);
    if (auto found = props.TryLookup(name)) {
        return winrt::unbox_value<T>(found);
    }
    props.Insert(name, winrt::box_value(value));
    return value;
}
```

We let the caller pass an arbitrary functor for making the object. The name is also a runtime parameter rather than a compile-time (template) parameter, so that you can generate unique names at runtime.

Using `box_value` and `unbox_value` allows us to store both Windows Runtime reference types and value types in the property set.

[1] For example, the object may, at construction, register with some other component, and that other component supports only one client at a time. Or the object, at construction, may attempt to access some resource that cannot be shared, like a file in exclusive mode.

Raymond Chen

Follow