On the perils of holding a lock across a coroutine suspension point, part 1: The set-up

devblogs.microsoft.com/oldnewthing/20210707-00

July 7, 2021

Raymond Chen

Say you want to perform a bunch of asynchronous operations involving some object state, but also want to make sure that no other tasks access that object state at the same time. For synchronous code, you would use a traditional synchronization object like a mutex or critical section:

```
void MyObject::RunOne()
{
  std::lock_guard guard(m_mutex);

  if (!m_list.empty()) {
    auto& item = m_list.front();
    item.Run();
    item.Cleanup();
    m_list.pop_front();
  }
}
```

The mutex ensures that only one attempt to process an item from the list is active at a time, and also to prevent any other code from mutating the `m_list` while we are using it.

But say that some of these operations are asynchronous. For simplicity, I'm eliding the traditional `auto lifetime = get_strong();` that is used to prevent the object from being destructed while awaiting. (Let's say that the rule is that you cannot release your reference to `MyObject` until `RunOneAsync` completes.)

```
IAsyncAction MyObject::RunOneAsync()
{
  std::lock_guard guard(m_mutex);

  if (!m_list.empty()) {
    auto& item = m_list.front();
    co_await item.RunAsync();
    item.Cleanup();
    m_list.pop_front();
  }
}
```

Is this okay?

One argument I've heard is that this is not okay because the `co_await` causes the original `RunOneAsync` call to to return an `IAsyncAction` to its caller, and as part of the act of returning the `IAsyncAction`, the lock is released.

This argument is incorrect. The lock remains held while the coroutine is suspended. After all, if objects were destructed at suspension, then you wouldn't be able to carry anything across a suspension point!

```
IAsyncAction WidgetManager::WhateverAsync()
{
  auto lifetime = get_strong();
  std::string name = m_widget.GetName();
  m_widget.SetName("temporary");
  co_await m_widget.SomethingAsync();
  m_widget.SetName(name); // certainly "name" is still valid, right?
  // certainly "lifetime" is still holding our object alive, right?
}
```

Don't worry. `name` and `lifetime` are still valid across the suspension because the formal parameters and local variables are kept in the coroutine frame, which remains alive while the coroutine is suspended. Indeed, the `lifetime` relies upon it!

However, it's the liveness of the lock guard that is the issue here.

Since the lock guard hasn't been destructed, the mutex remains locked while the coroutine is suspended.

Now things get exciting.

Suppose we have another coroutine that wants the lock. Heck, it could very well be another call to `RunOneAsync`!

`RunOneAsync` #1

construct lock_guard		`m_mutex.lock()`
`auto& item = m_list.front();`		
`co_await item.RunAsync();`	→	Suspended
`RunOneAsync` #1 returns `IAsyncAction`		
↓		
Thread available to do other work		
↓		
`RunOneAsync` #2		
construct lock_guard		`m_mutex.lock()`

Now we're in trouble.

If the `m_mutex` supports recursive acquisition, then what happens is that the second call to `RunOneAsync` successfully acquires the mutex (recursive acquisition), and execution continues:

`RunOneAsync` #2 continues		
`auto& item = m_list.front();`		
`co_await item.RunAsync();`	→	Suspended
`RunOneAsync` #2 returns `IAsyncAction`		
↓		
Thread available to do other work		

We are running the front element twice! I bet it's not expecting that.

The mutex failed at its intended purpose of serializing calls to `RunOneAsync`.

Okay, but wait, the disaster is still unfolding.

Eventually, the two calls will complete, in some order. Let's say that `#1` finishes first. Execution continues:

`RunOneAsync` #1 resumes	←	`RunAsync` #1 completes

`item.Cleanup();`		Cleaning up while `RunAsync` #2 still outstanding
`m_list.pop_front();`		Destructing head item while #2 is still using it
destruct lock_guard		`m_mutex.unlock()`
`RunOneAsync` #1 completes		
↓		
Thread available to do other work		
↓		
`RunOneAsync` #2 resumes	←	`RunAsync` #2 completes
`item.Cleanup();`		Cleaning up already-destructed object
`m_list.pop_front();`		Popping an item that was never run
destruct lock_guard		`m_mutex.unlock()`
`RunOneAsync` #2 completes		

When the first `RunAsync` completes, the first `RunOneAsync` resumes, and it proceeds to clean up the item that finished running, and then remove the head item from the list, thereby destructing it. All this happens even though the second `RunOneAsync` is still using it.

The first `RunOneAsync` completes, having created a right mess of things but escaping unharmed.

When the second `RunAsync` completes, the second `RunOneAsync` resumes, and it tries to clean up the item that has already been destructed. You get sent this crash dump and you scratch your head because you're looking at the code and you see that mutex right there, and you're thinking, "How can this thing get prematurely destructed? It's protected by a mutex!"

Now, maybe the `Cleanup` method happens by sheer luck not to crash. It "only" corrupts some memory. That just makes the debugging even harder.

The second `RunOneAsync` then pops the front item from the list, thinking it's popping the item that it just finished running, when in fact it's popping an item on the list *that was never run at all*.

Now the bug is that the program keeps running, but sometimes, items put onto the work list are thrown away without ever being run or cleaned up. Meanwhile, some items are run twice. This bug doesn't come with crash dumps. It's just end-user reports from the field that your program isn't doing its job.

Basically, what's going on is that thanks to coroutines sharing a thread, your recursive mutex is not doing its job of ensuring mutual exclusion. Since everything is happening on a single thread, the recursive mutex always says, "Oh, I remember you. Come on in!"

Next time, we'll look at what happens if the mutex does not support recursive acquisition.

Raymond Chen

Follow