On the perils of holding a lock across a coroutine suspension point, part 2: Nonrecursive mutexes

devblogs.microsoft.com/oldnewthing/20210708-00

July 8, 2021

Raymond Chen

Last time, we looked at what can go wrong if you hold a recursive mutex across a coroutine suspension point. Do things get any better if you switch to a nonrecursive mutex?

Recall that we are looking at this function:

```
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();
  }
}
```

Let's walk through what happens if the mutex is nonrecursive and a call to `RunOneAsync` is made from the same thread that mad a previous not-yet-complete 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()` — blocks

During the period of suspension, anybody who wants to acquire the lock will block, since that’s how nonrecursive mutexes work.

Formally speaking, attempting to acquire a nonrecursive mutex recursively triggers *undefined behavior*, so from a compiler-theoretic point of view, the game is over and anything can happen, including time travel. In practice, what happens is that the attempted recursive acquisition blocks.

And that’s a real hard block, not a coroutine suspend. The thread that tries to acquire the lock cannot do anything while waiting for the lock to become available. In particular, it *cannot run coroutine continuations*.

Now, we don’t know much about `RunAsync`. Maybe it needs access to the originating thread in order to complete its work. Or maybe it uses another coroutine, and that *other* coroutine needs access to the originating thread. If that’s the case, then the `RunAsync` will never complete, because the originating thread is hung.

Maybe you’re lucky, and `RunAsync` can do all of its work without needing to access the originating thread. You’re still in trouble, because the `RunOneAsync` might need access to the originating thread. For example, C++/WinRT has a policy that `co_await` of an `IAsyncAction` always resumes in the same apartment context. If the original apartment is a single-threaded apartment (standard for UI threads), then it’s going to need to get back to that originating thread, but it can’t because the originating thread is hung waiting for the mutex.

Now, suppose you’re super-lucky, and the `co_await` of `RunAsync` doesn’t need to resume on the originating thread. Maybe you started in the multi-threaded apartment, so it can resume on any other thread in that apartment. Great, your code is running again, just on a different thread.

Some other thread		
↓		
`RunOneAsync` #1 resumes	←	`RunAsync` #1 completes
`item.Cleanup();`		
`m_list.pop_front();`		

destruct lock_guard	`m_mutex.unlock()` — from the wrong thread

We are unlocking a mutex from a thread that didn’t lock it. This is not a legal operation and the behavior is undefined.

So yeah, double undefined behavior.

In practice, what usually happens is that your main thread hangs unrecoverably. You dump all the stacks to try to find the owner, and you don’t see any stacks that are in code that’s holding the lock. That’s because the code that’s responsible for the lock isn’t active on any thread, so you won’t see it in any stack. The code is waiting to resume execution when its associated coroutine is resumed, and that coroutine is somewhere on the heap.

Basically, any `co_await` is a point of potential re-entrancy.

Next time, we’ll look at ways of addressing the problem.

Raymond Chen

Follow