Creating other types of synchronization objects that can be used with co_await, part 7: The mutex and recursive

devblogs.microsoft.com/oldnewthing/20210317-00

March 17, 2021

Raymond Chen

Our next stop in showing off our library for building awaitable synchronization objects are the mutex and recursive mutex.

The mutex and recursive mutex are strange objects when applied to coroutines, because the traditional mutex and recursive mutex apply to threads, and threads are this implied context that every function has and shares with the functions it calls, even though it may not realize it. But coroutines don't have that. When one coroutine awaits another coroutine, there is no formal connection between the two, so when you do this:

```
IAsyncAction Outer()
{
    co_await some_mutex.lock();
    co_await Inner();
}

IAsyncAction Inner()
{
    some_mutex.unlock();
}
```

the mutex has no way of knowing that the `Outer` lock matches the `Inner` unlock. It just has to trust the caller.

The lack of coroutine flow identity is particularly troublesome for the recursive mutex, because it has no way of knowing whether a second lock came from the same logical coroutine chain that performed the initial lock.

```
IAsyncAction Outer()
{
    co_await some_mutex.lock();
    co_await Inner();
    some_mutex.unlock();
}

IAsyncAction Inner()
{
    co_await some_mutex.lock();
    something();
    some_mutex.unlock();
}
```

If `some_mutex` were an asynchronous recursive mutex, it wouldn't be able to determine whether the call from `Inner` should be allowed, since it has no access to coroutine identity, nor to the dependency relationships among coroutines.

Basically, there are no coroutine mutexes or recursive mutexes. Recursive mutexes simply cannot exist, and once you strip away the thread identity features of mutexes, all you have left is an auto-reset event.

```
using awaitable_mutex = awaitable_auto_reset_event;
```

But the shared mutex, that's interesting. We'll look at that next time.

Raymond Chen

Follow