Inside C++/WinRT: Coroutine completions: Avoiding reentrant completion

devblogs.microsoft.com/oldnewthing/20230202-00

February 2, 2023

Raymond Chen

If a Windows Runtime asynchronous operation has already completed at the point the `Completed` delegate is assigned, the implementation is permitted to invoke the delegate before returning from the assignment.

The way we have set things up so far, it means that the awaiting an already-completed Windows Runtime asynchronous operation results in a chain of calls:

```
MyAwesomeCoroutine::DoSomethingAsync$Resu
coroutine_handle<>::resume
resume_apartment
disconnect_aware_handler::Complete
Provider::FireCompletion
Provider::put_Completed
IAsyncOperation::put_Completed
await_adapter::await_suspend
MyAwesomeCoroutine::DoSomethingAsync$Resu
```

All those stack frames between the two `MyAwesomeCoroutine::DoSomethingAsync$Resu` frames are unnecessary, and if you are `co_await` ‘ing in a loop, the stack usage accumulates and can result in unexpected stack exhaustion.

What we can do is detect that the completion handler is running before `put_Completed` has returned, and in that case, we merely remember that the coroutine needs to resume, but without actually resuming it immediately. We allow execution to unwind back to `await_suspend` and then return `false` to tell the coroutine infrastructure to resume the coroutine when it unwinds.

```
template<typename Awaiter>
struct disconnect_aware_handler
{
    ⟦ ... ⟧

    void Complete()
    {
        if (m_awaiter->suspending
            .exchange(false, std::memory_order_release))
        {
            // resumption has been deferred to await_suspend
            m_handle = nullptr;
        }
        else
        {
            resume_apartment(m_context.context,
                std::exchange(m_handle, {}),
                &m_awaiter->failure);
        }
    }
};
```

If the awaiter says that it's still suspending, then don't resume immediately. Instead, reset the `suspending` to `false` to tell `await_suspend` to cancel the suspension.

```
template<typename Async>
struct await_adapter
{
    await_adapter(Async const& async) : async(async) { }

    Async const& async;
    int32_t failure = 0;
    std::atomic<bool> suspending = true;

    ⟦ ... ⟧

    auto await_suspend(coroutine_handle<> handle) const
    {
         // auto extend_lifetime = async;
        async.Completed(
            disconnect_aware_handler(this, handle));
        return suspending.exchange(false, std::memory_order_acquire);
    }

    ⟦ ... ⟧
};
```

The other half of the communication is in the `await_adapter` 's `await_suspend` method. After setting the `Completed` handler, we reset `suspending` to `false` , and return the previous value. The previous value is `true` if the completion handler hasn't run yet, and returning `true` from `await_suspend` allows the suspension to proceed. But if the completion handler has already run, then the previous value is `false`, and returning handler hasn't run yet, and returning `false` from `await_suspend` tells the coroutine infrastructure to abandon the suspension and resume the coroutine.

Note that we use atomic operations on both sides, because the completion handler might run on another thread and race against `await_suspend`. In particular, we need to watch out for the case where the completion handler is called immediately after `await_suspend` sets the `Completed` property and checks the `suspending` variable, but before it returns. In that case, we need to resume the coroutine immediately from the completion handler, because the decision to allow the coroutine to suspend has already been made.

The atomic operations use release semantics on the publishing side and acquire semantics on the consumption side so that any changes to objects immediately before completion are visible when the coroutine resumes.

Now that we defer the resumption of the coroutine until after `async.Completed()` returns, we don't need to extend its lifetime to protect against premature resumption: We never resume the coroutine while `async.Completed()` is still running.

As of this writing, C++/WinRT still supports Visual C++'s experimental coroutine support. Older versions of that coroutine support have a code generation bug (which I noted some time ago is fixed in versions 16.11 and 17.0), so we need to work around that. The way to detect the experimental coroutine support is to check for the preprocessor symbol `_RESUMABLE_FUNCTIONS_SUPPORTED`:

```
template<typename Async>
struct await_adapter
{
    ⟦ ... ⟧

    auto await_suspend(coroutine_handle<> handle) const
    {
        async.Completed(
            disconnect_aware_handler(this, handle));
#ifdef _RESUMABLE_FUNCTIONS_SUPPORTED
        if (!suspending.exchange(false, std::memory_order_acquire))
        {
            handle.resume();
        }
#else
        return suspending.exchange(false, std::memory_order_acquire);
#endif
    }

    ⟦ ... ⟧
};
```

If we are being compiled with experimental coroutine support, then we avoid the code generation bug by resuming the handle explicitly rather than returning a `bool`. This does consume a little bit or stack, but not as much as before.

Bonus chatter: The workaround is used if experimental coroutine support is detected, regardless of the Visual C++ compiler version. That's because the experimental coroutines are all ABI compatible, and I don't want to take the risk of an ODR violation if people link together object files compiled with different versions of experimental coroutine support. Visual C++ took an ABI breaking change for standard coroutines, so C++/WinRT uses that as its own signal to switch to the `bool` version of `await_suspend`. That way, there won't be any ODR violation in C++/WinRT caused by linking together object files with experimental and standard coroutines: If you try, you get a mismatch from MSVC for `_COROUTINE_ABI`. Combining experimental and standard coroutines never worked anyway, and we rely on the compiler to check for us.