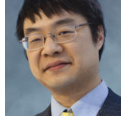
C++ coroutines: The problem of the synchronous apartment-changing callback

devblogs.microsoft.com/oldnewthing/20191220-00

December 20, 2019



Raymond Chen

Today is a puzzle you can you can try to solve with the information you've learned about C++ coroutines and C++/WinRT.

C++/WinRT uses the `IContextCallback` interface to remember the context that initiated a `co_await` operation, so it can resume execution in the original apartment when the `co_await` completes.

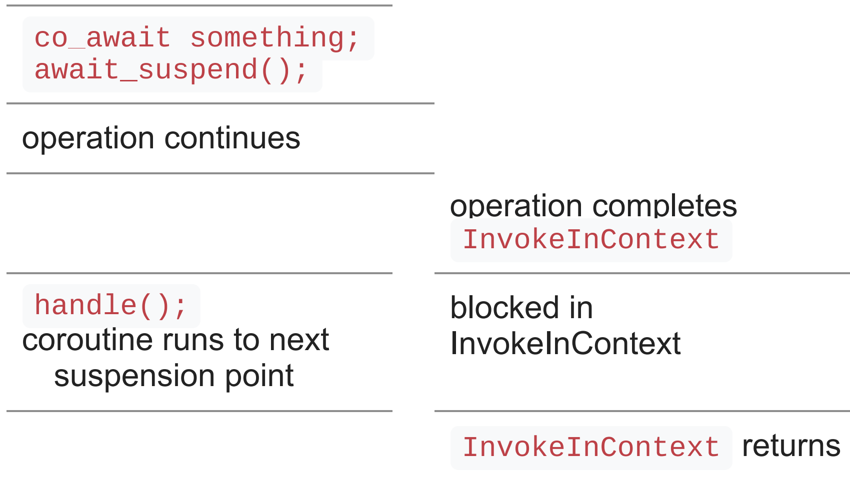
The basic idea goes like this:

```
void await_suspend(std::experimental::coroutine_handle<> handle)
{
    async.Completed([handle,
                     context = CaptureCurrentApartmentContext()]
                    (auto const&, Windows::Foundation::AsyncStatus)
    {
        // When the operation completes, get back to the
        // original apartment and resume the coroutine there.
        check_hresult(InvokeInContext(context.get(), handle));
    });
}
```

Maybe you see a problem here. I noticed a problem when I studied the C++/WinRT code and meant to do a write-up on it eventually, but then I actually ran into the problem and alerted Kenny, who promptly fixed it. (Note: Clicking through gives away the answer.)

The `IContextCallback:: ContextCallback` method invokes the callback synchronously, and the invoking apartment is stuck waiting for the result. This is good if you want to callback to do some work that you are waiting for, but it's not good if the caller just wants to fire and forget.

Thread 1	Thread 2



In the above diagram, code inside a box represents code being executed on behavior of a coroutine. If the thread does not have a box, then it is available to do other work.

A synchronous callback means that when this awaiter tries to resume execution, the thread that raised the `Completed` event is stuck until the continued coroutine reaches a suspension point or completes, because those are the things that cause the coroutine to return at the ABI. This period of time is represented by the shaded section labeled “blocked in InvokeIn-Context”. During this period, the thread is not available to do work.

This shaded period during which the thread is unresponsive may last for a long time. And that’s a problem if it’s a UI thread.

Consider the following scenario:

```
IAsyncAction SomethingAsync()
{
    co_await resume_background();

    DoBackgroundWork();

    // Get to our UI thread so we can update UI.
    co_await resume_foreground(Dispatcher());

    UpdateUIStuff();

    co_return;
}
```

This coroutine switches immediately to a background thread, does a bunch of work, and then switches back to the UI thread to update some UI.

You might decide to use this function like this:

```
IAsyncAction SomethingMoreAsync()
{
    // Do all our work on a background thread.
    co_await resume_background();

    LongBlockingOperation();
    co_await SomethingAsync();
    LongBlockingOperation();
}
```

In C++/WinRT, `co_await` of an `IAsyncAction` returns control to the same apartment that originated the operation, so all of the `LongBlockingOperation` calls occur on a background thread. Certainly it's safe to perform long blocking operations on a background thread, right?

Let's look more closely at what happens.

UI thread	Background thread
`SomethingMoreAsync` begins `co_await resume_background();`	
	`SomethingMoreAsync` resumes
	`LongBlockingOperation()`
	`SomethingAsync` begins `co_await resume_background();`
	`SomethingAsync` resumes
	`DoBackgroundWork()`
	`co_await resume_foreground();`

`SomethingAsync` resumes
`UpdateUIStuff();`
`SomethingAsync` completes
`Completed()` handler tries to resume on original context
`InvokeInContext`
	⇐ things get interesting

blocked in InvokeInContext	`handle();` `SomethingMoreAsync` resumes
	`LongBlockingOperation()`
	`SomethingMoreAsync` completes

The first part of the sequence goes as you would expect. The `SomethingMoreAsync` coroutine moves to a background thread and performs a long blocking operation. This is okay, because we're on a background thread.

Next, it calls `SomethingAsync`, which starts by moving to a background thread. (It's already on a background thread, but it doesn't know that.)

Once rescheduled (redundantly) on a background thread, it does some background work. Again, this background work can take a long time, but that's okay because we're on a background thread.

When the background work is done, `SomethingAsync` moves back to the UI thread.

Once back on the UI thread, `SomethingAsync` updates its UI and completes the coroutine.

Now things get interesting.

The awaiter for `IAsyncAction` wants to resume in the original apartment, which in this case means going back to a background thread. It does this by using `IContextCallback::ContextCallback`, which we wrapped inside `InvokeInContext` for expository purposes.

The `IContextCallback::ContextCallback` method invokes the callback synchronously, which means in our case that the call doesn't return until the resumed coroutine reaches its next suspension point. But before it can complete or perform another `co_await`, it performs a long blocking operation, believing that since it is on a background thread, long blocking operations are permitted.

And it's true that long blocking operations are permitted on a background thread. The problem is that a UI thread is waiting for the background thread.

The background thread is unwittingly holding up a UI thread.

The fix is to use `IContextCallback::ContextCallback` only in the case when we need to return to a UI thread. If we need to return to a background thread, we can use the non-blocking `resume_background` to do that.

This means that if a background thread needs to return to a UI thread, then the background thread will be held hostage by the coroutine on the UI thread until it completes or suspends. That's not so bad, because background threads can block. And besides, coroutines on UI threads are not supposed to perform long blocking operations in the first place.

It also means that if a second UI thread needs to return to an originating UI thread, then the second UI thread will be held hostage by the coroutine on the originating UI thread until it completes or suspends. But that's not so bad, because, as we noted before, coroutines on UI

threads are not supposed to perform long blocking operations in the first place.

Next time, we'll look at a coroutine bug in the C++/WinRT library and try to fix it by applying what we've learned so far.

Raymond Chen

Follow