Ordering asynchronous updates with coroutines, part 5: Bowing out via cancellation

devblogs.microsoft.com/oldnewthing/20210910-00

September 10, 2021

Raymond Chen

Last time, we showed how a coroutine could check after every `co_await` whether its work has been superseded, in which case it just gives up rather than proceeding with a calculation whose result won't be used anyway.

We noted that a coroutine provider can snoop on every `co_await` in the coroutine body by means of the `await_transform` method. C++/WinRT uses this feature to implement a few things. One of them is making every `co_await` check whether the coroutine has been cancelled and throwing `hresult_canceled` if so. We can take advantage of this by using cancellation to stop any existing instance of the coroutine.

```
winrt::IAsyncAction Widget::RecalcWorkerAsync()
{
    auto lifetime = get_strong();
    auto cancellation = co_await winrt::get_cancellation_token();

    winrt::hstring messageId;
    winrt::hstring lang;
    {
        std::lock_guard guard{ m_mutex };
        messageId = m_messageId;
        lang = m_lang;
    }

    auto resolved = co_await ResolveLanguageAsync(lang);
    auto library = co_await GetResourceLibraryAsync(resolved);
    auto message = library.LookupResourceAsync(messageId);

    std::lock_guard guard{ m_mutex };
    if (!cancellation()) {
        m_message = message;
    }
}
```

We move the recalculation into a worker function and rely on cancellation to tell us when to stop calculating. The C++/WinRT library automatically checks for cancellation at each `co_await` so the only explicit check we need is the final one.

The `IAsyncAction` produced by `RecalcWorkerAsync` is managed by the `RecalcAsync` function:

```
winrt::IAsyncAction m_pendingAction;

winrt::IAsyncAction Widget::SetPendingAction(
    winrt::IAsyncAction const& action)
{
    winrt::IAsyncAction pendingAction;
    {
        std::lock_guard guard{ m_mutex };
        pendingAction = std::exchange(m_pendingAction, nullptr);
    }
    if (pendingAction) {
        pendingAction.Cancel();
    }
}

winrt::IAsyncAction Widget::RecalcAsync()
{
    auto lifetime = get_strong();

    SetPendingAction(nullptr);

    auto currentAction = RecalcWorkerAsync();

    SetPendingAction(currentAction);

    try {
        co_await currentAction;
    } catch (winrt::hresult_canceled const&) {
        // ignore cancellation
    }
}
```

There are a few steps here.

Before we start, we cancel the previous operation. The call to `Cancel` must happen outside the lock, because the coroutine completion function is invoked synchronously from inside the `Cancel`, and we don't want to let foreign code run while inside our lock.

Next, we start the new operation.

And then we make the new operation become the current operation. There is a race here where two threads both start a new operation at the same time, in which case we have to cancel any possible interloper.

The need for the early cancel stems from this race condition that can occur if we remove the first call to `SetPendingAction`:

Thread 1	Thread 2

`RecalcWorkerAsync`:
`currentAction = RecalcAsync()`
`enter lock`
`std::exchange(`
`m_pendingAction, currentAction)`
`exit lock`
`RecalcAsync`:
`co_await`
`ResolveLanguageAsync(...);`
`co_await`
`GetResourceLibraryAsync(...);`
`co_await`
`LookupResourceAsync(...);`	

	`RecalcWorkerAsync`:
	`currentAction = RecalcAsync()`
	`enter lock`
	`pendingAction = std::exchange(`
	`m_pendingAction, currentAction)`
	`exit lock`
	`RecalcAsync`:
	`co_await`
	`ResolveLanguageAsync(...);`
	`co_await`
	`GetResourceLibraryAsync(...);`
	`co_await`
	`LookupResourceAsync(...);`
	`enter lock`
	`verify not cancelled`
	`m_message = message`
	`exit lock`

`enter lock`
`verify not cancelled`
`m_message = message`
`exit lock`	

	`RecalcWorkerAsync` continues: `pendingAction.Cancel()`

The cancellation of the previous call to `RecalcAsync` happens too late. The previous recalculation raced against the current recalculation, and the current one happened to finish first, causing the previous one to overwrite the result.

If the `Widget` is single-threaded, then we can get rid of the locks, and that also removes some of the subtle race conditions.

```
winrt::IAsyncAction Widget::RecalcWorkerAsync()
{
    auto lifetime = get_strong();
    auto cancellation = co_await winrt::get_cancellation_token();

    auto messageId = m_messageId;
    auto lang = m_lang;

    auto resolved = co_await ResolveLanguageAsync(lang);
    auto library = co_await GetResourceLibraryAsync(resolved);
    auto message = library.LookupResourceAsync(messageId);

    // std::lock_guard guard{ m_mutex };
    if (!cancellation()) {
        m_message = message;
    }
}

winrt::IAsyncAction Widget::RecalcAsync()
{
    auto lifetime = get_strong();

    auto currentAction = RecalcWorkerAsync();

    auto previousAction = std::exchange(m_pendingAction, currentAction);
    if (previousAction) previousAction.Cancel();

    try {
        co_await currentAction;
    } catch (winrt::hresult_canceled const&) {
        // ignore cancellation
    }
}
```

The race condition doesn't exist because there is no opportunity for the previous action to do any work between the time we start the new task and cancel the old one. The only race is between the final `co_await` and the cancellation, and one final check takes care of that.

One thing you might notice about this pattern is that `m_pendingAction` is never nulled out. It always holds the last successful action, even after it has completed. This means that the coroutine remains allocated, even though has ended its useful life, consuming memory (probably not too much) and keeping its inbound parameters alive (fortunately, we have none). If the coroutine frame is large, or if there are inbound parameters which you need to run down promptly,[1] you can clean it up once you've finished waiting for it.

```
winrt::IAsyncAction Widget::RecalcAsync()
{
    auto lifetime = get_strong();

    auto currentAction = RecalcWorkerAsync();

    auto previousAction = std::exchange(m_pendingAction, currentAction);
    if (previousAction) previousAction.Cancel();

    try {
        co_await currentAction;
    } catch (winrt::hresult_canceled const&) {
        // ignore cancellation
    }
    if (m_pendingAction == currentAction) {
        m_pendingAction = nullptr;
    }
}
```

[1] One example of needing to run down inbound parameters is the case where they belong to another component. You don't want to extend the lifetime of foreign objects beyond the end of the useful life of the coroutine. Not only could that create problems with that other component (say, because the other component is single-threaded and the thread that hosts it wants to exit), it could also introduce circular references between that other component and your component.

Raymond Chen

Follow