Ordering asynchronous updates with coroutines, part 3: Let them all compete, but only one wins

devblogs.microsoft.com/oldnewthing/20210908-00

September 8, 2021

Raymond Chen

Previously, we looked at the case where calling a method initiates some asynchronous activity, and if new activity is required, the work is handed off to the existing coroutine. A different model is to have everybody do the work in parallel, but only the last one counts. Of course, this pattern assumes that the work can safely be performed in parallel, such as perform a complex calculation.

Let's assume that the object has thread affinity, so we can assume that all accesses on the UI thread are uncontended and therefore do not require a lock.

```
std::mutex m_mutex;
winrt::hstring m_lang;
int32_t m_messageId;
winrt::hstring m_message;

winrt::IAsyncAction Widget::SetMessageAsync(int32_t messageId)
{
    auto lifetime = get_strong();
    {
        std::lock_guard guard{ m_mutex };
        m_messageId = messageId;
    }
    co_await RecalcAsync();
}

winrt::IAsyncAction Widget::SetLanguageAsync(winrt::hstring lang)
{
    auto lifetime = get_strong();
    {
        std::lock_guard guard{ m_mutex };
        m_lang = lang;
    }
    co_await RecalcAsync();
}

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

    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 = co_await library.LookupResourceAsync(messageId);

    std::lock_guard guard{ m_mutex };
    if (m_messageId == messageId && m_lang == lang) {
        m_message = message;
    }
}
```

The pattern here is that on each call to `SetMessageId()` or `SetLanguage()`, we update our local state variables and then call a common helper coroutine to recalculate the `m_message`.

The pattern in `RecalcAsync` goes like this:

- Capture the member variables you need into local variables.
- Do the asynchronous work, operating purely on the local variables.
- When finished, compare the member variables against the local variables to see if they still match.
- If so, then update the results.
- If not, then somebody else changed the `m_messageId` or `m_lang` in the meantime, so abandon the update.

In the case where the update is abandoned, we don't have to restart the calculation because whoever changed the message ID or the language is running their own `RecalcAsync`. In fact, *everyone* who changes the message ID or the language is running their own `RecalcAsync`, and only the one whose calculations match the current state gets to update the result. Note that this may not be the calculation that finishes last.

The "capture and compare" pattern assumes that the calculations are idempotent. If not, then you can use a counter to keep track of who is running the "real" computation.

```
uint32_t m_counter = 0;

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

    uint32_t counter;
    winrt::hstring messageId;
    winrt::hstring lang;
    {
        std::lock_guard guard{ m_mutex };
        counter = ++m_counter;
        messageId = m_messageId;
        lang = m_lang;
    }

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

    std::lock_guard guard{ m_mutex };
    if (m_counter == counter) {
        m_message = message;
    }
}
```

We use a counter to keep track of which instance of the recalculation we are managing, and when we finish our calculations, we check if the counter has changed since we started. If not, then we are the active recalculation and can update the result. If the counter doesn't match,

then somebody else triggered a recalculation while we were recalculating, and we'll let that other recalculation set the result.

Caller 1
`co_await RecalcAsync();`
`  counter = m_counter = 1;`
`  co_await ResolveLanguageAsync(...);`		

	Caller 2
	`co_await RecalcAsync();`
	`  counter = m_counter = 2;`
	`  co_await ResolveLanguageAsync(...);`	

`  co_await GetResourceLibraryAsync(...);`		

		Caller 3
		`co_awai`
		`  count`
		`  co_av`
		`Resolve`

	`  co_await GetResourceLibraryAsync(...);`	
		`  co_av` `GetReso`
	`  co_await LookupResourceAsync(...);`	
`  co_await LookupResourceAsync(...);`		
		`  co_av` `LookupR`

	`  m_counter is 3`
	`  counter is 2`
	`  do not update m_message`
	`  co_return;`	

		`  m_cou`
		`  count`
		`  updat`
		`  co_re`

```
    m_counter is 3
    counter is 1
    do not update m_message
    co_return;
```

Everybody tries to recalculate, but only the one that performed the most recent `RecalcAsync` gets to update the result.

You may recognize this as the coroutine version of the lock-free try/commit/abandon pattern.

This pattern solves the fairness problem we saw last time: No instance of `RecalcAsync` is being asked to calculate more than once, so it's not the case that repeated recalculation requests cause one instance to do an unbounded amount of work on behalf of others.

This pattern does however result in a lot of wasted work. Once the second `RecalcAsync` begins, we all know that the work being done by the first call is pointless, since it will end up just throwing away the result. Next time, we'll see what we can do to avoid that wasted work once we realize it's going to be wasted.

Bonus chatter: In the case where the object is single-threaded, you can get rid of the locks, which makes the code much simpler. (This relies on the C++/WinRT behavior that `co_await`'ing an `IAsyncAction` or `IAsyncOperation` resumes in the same COM context.)

```
uint32_t m_counter = 0;

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

    auto counter = ++m_counter;
    auto messageId = m_messageId;
    auto lang = m_lang;

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

    if (m_counter == counter) {
        m_message = message;
    }
}
```

You can also hop to a background thread, as long as you hop back to the main thread when accessing the member variables.

```
uint32_t m_counter = 0;

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

    auto counter = ++m_counter;
    auto messageId = m_messageId;
    auto lang = m_lang;

    co_await winrt::resume_background();

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

    co_await winrt::resume_foreground(Dispatcher());

    if (m_counter == counter) {
        m_message = message;
    }
}
```

Raymond Chen

Follow