Ordering asynchronous updates with coroutines, part 2: Restart with hand-off

devblogs.microsoft.com/oldnewthing/20210907-00

September 7, 2021

Raymond Chen

Another serialization pattern for coroutines is where calling some method initiates some asynchronous activity, and if the method gets called again while the activity is still incomplete, you want to let the previous activity run to completion, and then run it again.

For example, maybe you have a method called `SetColor` that changes the color in private state and asynchronously propagates that color into another component. If the previous color-setting operation is still in progress when a second `SetColor` occurs, you want to let the propagation of the old color finish, and once that's done, start pushing out the new color. (Intermediate colors are not important; only the last color counts.)

```
std::mutex m_mutex;
bool m_busy = false;

winrt::fire_and_forget Widget::SetColor(Color newColor)
{
    auto lock = std::unique_lock(m_mutex);
    m_color = newColor;
    if (std::exchange(m_busy, true)) {
        co_return;
    }

    auto lifetime = get_strong();

    Color latestColor;
    do {
        latestColor = m_color;
        lock.unlock();
        try {
            co_await UpdateColorOfExternalPartner(latestColor);
        } catch (...) {
            // nowhere to report the error
            // you can choose to log it or to fail fast
        }
        lock.lock();
    } while (m_color != latestColor);
    m_busy = false;
}
```

The idea here is that after setting the private `m_color`, we check whether somebody else is already busy updating the color of the external partner. If so, then we just return immediately, knowing that the existing worker will pick up the new color eventually.

If nobody is doing the work (the previous value of `m_busy` was `false`), then we assume responsibility for the work: We capture the most recently set color, and then drop the lock while we update the external partner. Once that's done, we reacquire the lock and see if the color changed again in the meantime. If so, we go back and push the new latest color, repeating until we make it through an entire update cycle with the updated color equal to the current color.

Now, this pattern assumes that we can detect that new work is needed by inspecting the `m_color`. But that may not always be the case, in which case we need a separate flag to say "New work was requested."

```
std::mutex m_mutex;
bool m_busy = false;
bool m_refreshNeeded = false;

winrt::fire_and_forget Widget::Refresh()
{
    auto lock = std::unique_lock(m_mutex);
    m_refreshNeeded = true;
    if (std::exchange(m_busy, true)) {
        co_return;
    }
    auto lifetime = get_strong();

    while (std::exchange(m_refreshNeeded, false)) {
        lock.unlock();
        try {
            co_await RefreshExternalPartner();
        } catch (...) {
            // nowhere to report the error
            // you can choose to log it or to fail fast
        }
        lock.lock();
    }
}
```

Since we don't have a `m_color` to tell us that we need to do more work, we create an explicit `m_refreshNeeded` flag, and we use a `while` loop to keep refreshing the external partner until we manage to make it all the way to the end without another refresh request coming in.

In the case where the object has thread affinity (common for UI objects), you may already have the requirement that `SetColor` or `Refresh` be called from the UI thread. More generally, if you can arrange that all accesses to `m_color`, `m_busy`, and `m_refreshNeeded` are on the same thread, then you don't need the mutex at all, and all the uses of the `lock` object can be removed.[1]

In these examples, I made the coroutines `fire_and_forget`, so they return to their callers quickly. If we changed them to return `IAsyncAction`, then the caller could `co_await` the call to wait for the update to complete. However, the way we structured the work, it means that if the object is constantly being updated, the first call to `SetColor` or `Refresh` ends up doing all of the work, and the `IAsyncAction` doesn't complete until the final refresh, which is unfair to the first caller:

Caller 1

```
co_await Refresh();
```

```
  m_busy = true;
```

```
  co_await RefreshExternalPartner();
```

	Caller 2
	`co_await Refresh();`
	`  m_needRefresh = true;`
	`  co_return;`

`  m_needRefresh = false;` `  co_await RefreshExternalPartner();`	

	Caller 3
	`co_await Refresh();`
	`  m_needRefresh = true;`
	`  co_return;`

`  m_needRefresh = false;`
`  co_await RefreshExternalPartner();`
`  m_busy = false;`
`  co_return;`	

We'll come back to this issue after we look at some other patterns for serializing asynchronous operations.

Exercise: In the asynchronous refresh pattern, why use a `while` loop? Why couldn't we have used the previous pattern of using a `do...while` loop, like this?

```
std::mutex m_mutex;
bool m_busy = false;
bool m_refreshNeeded = false;

winrt::fire_and_forget Widget::Refresh()
{
    auto lock = std::unique_lock(m_mutex);
    if (std::exchange(m_busy, true)) {
        m_refreshNeeded = true;
        co_return;
    }
    auto cleanup = wil::scope_exit([&] { m_busy = false; });
    auto lifetime = get_strong();

    do {
        lock.unlock();
        co_await RefreshExternalPartner();
        lock.lock();
    } while (std::exchange(m_refreshNeeded, false));
}
```

Bonus chatter: You can be extra-clever and combine `m_busy` and `m_refreshNeeded` into a single atomic variable.

```
// 0 = not busy
// 1 = busy
// 2 = refresh needed

std::atomic<int> m_busy;

winrt::fire_and_forget Widget::Refresh()
{
    if (m_busy.exchange(2, std::memory_order_release)) {
        co_return;
    }
    while (m_busy.fetch_sub(1, std::memory_order_acquire) == 2) {
        ... do work ...
    }
}
```

The initial exchange publishes the request to refresh the external partner, so it uses release semantics. If the previous value is nonzero, then it means that somebody else is already working, so we can return immediately and let the existing worker pick up the refresh request.

If nobody is doing work, then we have to do it. We decrement the busy count and see if there is work to do. If so, the busy count is decremented to 1, meaning "We are doing work, and no additional work has been requested." After doing the work, we loop back and decrement again. If work has been requested in the meantime, the value will have been bumped up to 2, so our decrement drops it back to 1, and the loop continues. Eventually, we make it all the way through without anybody requesting more work, which we detect when the busy count decrements all the way to zero.

[1] Note that we are assuming that `UpdateColorOfExternalPartner` and `RefreshExternalPartner` return `IAsyncAction` or otherwise ensure that the `co_await` resumes in the same COM context in which it suspended.

Raymond Chen

Follow