How to wait for multiple C++ coroutines to complete before propagating failure, unhelpful lambda

devblogs.microsoft.com/oldnewthing/20230627-00

June 27, 2023

Raymond Chen

Last time, we found a solution for waiting for multiple C++ coroutines to complete before propagating failure, but it relied on *expansion statements*, which weren't finished in time for C++20. We'll have to find something that uses only features available in C++20, or even better, works in C++17.[1]

The usual way to get expansion-like behavior is to use a lambda and the comma operator:

```
([&](auto& arg) {
    /* do something */
}(args), ...);
```

So let's try applying it to our "wait for all" coroutine function:

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    std::exception_ptr eptr;

    ([&] (auto& async) {
        try {
            co_await async;
        } catch (...) {
            if (!eptr) {
                eptr = std::current_exception();
            }
        }
    }(asyncs), ...);

    if (eptr) std::rethrow_exception(eptr);
}
```

Sadly, this doesn't work because the lambda performs a `co_await`, and `co_await` requires that you be in a coroutine.

I guess we have to make the lambda a coroutine.

```
template<typename... T>
IAsyncAction when_all_complete(T... asyncs)
{
    std::exception_ptr eptr;

    auto each = [&] (auto& async) -> IAsyncAction {
        try {
            co_await async;
        } catch (...) {
            if (!eptr) {
                eptr = std::current_exception();
            }
        }
    };

    (co_await each(asyncs), ...);

    if (eptr) std::rethrow_exception(eptr);
}
```

One thing you might object to is the fact that we have a lambda coroutine with a capture. This is generally frowned upon, due to the risk of the lambda being destructed while suspended, but it works here because the lambda is not destructed until `when_all_complete` finishes executing all of its `co_await`s. Even if one of the calls to `co_await async` throws an exception, that exception is caught and saved in `eptr`, and the overall coroutine completes without an exception.

It looks like we've done it, but there's a catch: The threading model.

In the original `when_all`, the parameters were each `co_await`ed directly in the main function, which means that if any of the `co_await`ed things changed threads, the thread change would remain in effect for the next `co_await`:

```
winrt::fire_and_forget example()
{
    co_await winrt::when_all(resume_background(), Something());
}
```

Inside `when_all`, this expands into

```
auto first = resume_background();
auto second = Something();
co_await first;
co_await second;
```

The `co_await first` performs a thread switch, and the `co_await second` executes on the new thread.

This is different from our `when_all_complete` because that one wraps each `co_await` inside another `IAsyncAction`, which changes the threading behavior: In C++/WinRT, `co_await`ing an `IAsyncAction` resumes on the same apartment that started the `co_await`. Since we wrapped each awaitable inside a lambda that produces an `IAsyncAction`, each `co_await` of the lambda will resume back on the original apartment, even if the original awaitable completed in a different apartment.

Here’s a comparison: With expansion statements, we just `co_await` directly from the function.

```
IAsyncAction v1(async1, async2)
{
    co_await async1;
    co_await async2;
}
```

Suppose that `async1` completes on a different apartment. (For example, it might be an apartment-switching awaitable, like `resume_background`.) Then the `co_await async1` will complete on the other apartment, and the `co_await async2` therefore begins on that other apartment.

On the other hand, with a lambda, we `co_await` from a lambda, and then `co_await` the lambda.

```
IAsyncAction v2(async1, async2)
{
    co_await [](auto& async) -> IAsyncAction {
        co_await async;
    }(async1);

    co_await [](auto& async) -> IAsyncAction {
        co_await async;
    }(async2);
}
```

First, we invoke the lambda, which does a `co_await async1`, which completes on some other apartment. However, since we `co_await` the lambda, and the lambda itself returns an `IAsyncAction`, the `co_await` of the lambda will complete back on the *original* apartment (because that’s how the `IAsyncAction` awaiter works). Next, we `co_await async2`, which now begins on the original apartment, not the apartment in which `async1` completed.

Next time, we’ll try to get all the `co_await`s to happen in the main function, so that we can preserve the apartment-switching behavior of the original `when_all` function.

Spoiler alert: Next time will be a failure.

¹ Yes, coroutines are not part of C++ until C++20, but the Microsoft Visual Studio compiler lets you opt into coroutine support while in C++17 mode, so “C++17 with optional coroutines” is the current baseline for C++/WinRT.