How to wait for multiple C++ coroutines to complete before propagating failure, peeling away at a tuple

devblogs.microsoft.com/oldnewthing/20230629-00

June 29, 2023

Raymond Chen

Last time, we used a `std::initializer_list` to hold the parameters. We chose that because we can iterate over an initializer list. Unfortunately, in our excitement, we forgot that all the items in an initializer list have to be the same type, but the `when_all_completed` function we're trying to write can be passed any mix of parameters of different types.

Aha, a `tuple` can hold values of heterogeneous types. Let's use that.

```
template<typename... Args>
IAsyncAction when_all_complete(Args&&...args)
{
    std::exception_ptr eptr;
    co_await when_all_complete_loop(
        std::make_tuple(args...), eptr);
    if (eptr) std::rethrow_exception(eptr);
}

template<typename Tuple>
IAsyncAction when_all_complete_loop(
    Tuple tuple, std::exception_ptr& eptr)
{
    if constexpr (std::tuple_size_v<Tuple>) {
        try {
            co_await std::get<0>(tuple);
        } catch (...) {
            if (!eptr) {
                eptr = std::current_exception();
            }
        }

        co_await when_all_complete_loop(
            remove_first(std::move(tuple)), eptr);
    }
    co_return;
}
```

I'm taking advantage of the `remove_first` function I wrote some time ago which produces a tuple that consists of a copy of the previous tuple, but with the first element removed.

The idea here is that we pack the awaitables into a `std::tuple` and then loop over the tuple elements, awaiting each one in turn and capturing any exception into the `std::exception_ptr` for eventual rethrow after all of the coroutines have completed.

There are a lot of subtleties here.

First, notice that we are passing a parameter *by reference* to a coroutine: The `eptr` is passed by reference into the tuple loop.

Normally, passing a reference to a coroutine is a bad idea because you normally cannot prove that the reference will remain valid for as long as you intend to use it. However, we're okay because we control the caller. What we're looking for is a way for the `eptr` to be destroyed before the `when_all_complete_loop` completes. Could the `co_await` returns prematurely? No, because the coroutine is never cancelled, so it cannot continue running after the caller is released.

The other subtlety is checking that this version `co_await`s each awaitable in the same apartment that the previous awaitable completed in. And that's true, because after `co_await`ing the first awaitable, we then await the recursive call, and the recursive call awaits the second awaitable (without any context-switching in between), and then makes a recursive call which awaits the third awaitable, and so on.

As the recursive calls unwind, each `co_await` on the recursive call will switch back to its initiating apartment, so we end with a flurry of apartment-switching, and then eventually complete back to our caller.

Now, there's a double-subtlety here: What if the `co_await` on the recursive call throws an exception because it is unable to switch back to its initiating apartment? Well, it means that an exception propagates out of the recursive chain, and any work that you thought was going to happen in the recursive chain won't actually happen.

Fortunately, there's nothing in the recursive chain left to do. The only thing happening is a bunch of `co_return` statements. So we don't actually bypass anything interesting. However, the exception propagates all the way out to the top-level `when_all_complete`, where it goes unhandled, so it propagates out of the function.

The way we structured the function, the idea is that we want to return the first exception encountered, but if an apartment switch fails, that one ends up taking precedence. But that's easy to fix: Move the recursive `co_await` inside the `try` block.

```
template<typename Tuple>
IAsyncAction when_all_complete_loop(
    Tuple tuple, std::exception_ptr& eptr)
{
    if constexpr (std::tuple_size_v<Tuple>) {
        try {
            co_await std::get<0>(tuple);
            co_await when_all_complete_loop(
                remove_first(std::move(tuple)), eptr);
        } catch (...) {
            if (!eptr) {
                eptr = std::current_exception();
            }
        }
    }
    co_return;
}
```

Okay, now that we have something that seems to work, we can try to tune the code so it doesn't make as many copies. For example, we can pass the tuple by reference to avoid copies, and instead of using `remove_first` to copy out the still-to-be-used awaitables, we keep track of which index we are recursing over.

```
template<typename... Args>
IAsyncAction when_all_complete(Args&&...args)
{
    std::exception_ptr eptr;
    co_await when_all_complete_loop<0>(
        std::make_tuple(args...), eptr);
    if (eptr) std::rethrow_exception(eptr);
}

template<int index, typename Tuple>
IAsyncAction when_all_complete_loop(
    Tuple&& tuple, std::exception_ptr& eptr)
{
    if constexpr (index < std::tuple_size_v<Tuple>) {
        try {
            co_await std::get<index>(tuple);
            co_await when_all_complete_loop<index + 1>(
                std::move(tuple), eptr);
        } catch (...) {
            if (!eptr) {
                eptr = std::current_exception();
            }
        }

    }
    co_return;
}
```

There's still an edge case we haven't taken into account, though: What if there is a `std::bad_alloc` when creating the recursive `IAsyncAction` calls? If we cannot allocate the coroutine frame for `when_all_complete_loop`, the system will throw a `std::bad_alloc`, and we dutifully save the error in the `eptr`. This exception then gets rethrown at the end, as expected, but *not all of the awaitables were awaited*.

We had one job!

Next time, we'll try to address out-of-memory exceptions that occur when allocating the coroutine frame.