How to wait for multiple C++ coroutines to complete before propagating failure, custom promise

devblogs.microsoft.com/oldnewthing/20230630-00

June 30, 2023

Raymond Chen

Last time, we used a `std::tuple` and a recursive function to implement our `when_all_completed` function. We noted that if during the recursion, the allocation of the coroutine frame fails, then a `std::bad_alloc` is thrown, and we end up returning via exception before awaiting the completion of all of the awaitables. We failed to do our stated job.

One way to address this is to use a custom promise. If the custom promise implements `operator new`, then that is used to allocate the coroutine frame, and that's our chance to deal with the failure.

And if we're going to use a custom promise, we may as well let the promise do all the work.

Let's write the coroutine promise that returns any observed exception. This custom promise also resumes its caller in the same apartment that the coroutine finished in. The apartment-switching behavior of C++/WinRT's `IAsyncAction` awaiter was one of the frustrating parts of our `when_all_completed` implementations.[1]

```
struct all_completed_promise;

struct all_completed_result
{
    all_completed_promise& promise;
    bool await_ready() noexcept { return false; }
    void await_suspend(
        std::coroutine_handle<> handle) noexcept;
    std::exception_ptr await_resume() noexcept;
};

struct all_completed_promise
{
    std::coroutine_handle<> awaiting_coroutine;
    std::exception_ptr eptr;

    all_completed_result get_return_object() noexcept {
        return {*this};
    }
    std::suspend_always initial_suspend() noexcept { return {}; }

    auto coroutine() {
        return std::coroutine_handle<all_completed_promise>::
            from_promise(*this);
    }

    void return_void() noexcept {}
    void unhandled_exception() noexcept
    { eptr = std::current_exception(); }

    std::suspend_never final_suspend() noexcept {
        awaiting_coroutine.resume();
        return {};
    }
};

void all_completed_result::
    await_suspend(std::coroutine_handle<> handle)
    noexcept
{
    promise.awaiting_coroutine = handle;
    promise.coroutine().resume();
}

std::exception_ptr all_completed_result::
    await_resume() noexcept
{
    return promise.eptr;
}

namespace std
{
```

```
    template<typename...Args>
    struct coroutine_traits<all_completed_result, Args...>
    {
        using promise_type = all_completed_promise;
    };
}

// example
all_completed_result Sample()
{
    co_await do_something();
}

winrt::fire_and_forget Caller()
{
    // eptr = nullptr if Sample ran without exception.
    // Otherwise it holds the exception pointer which
    // can be used to rethrow the exception.
    std::exception_ptr eptr = co_await Sample();
}
```

A coroutine that returns `all_completed_result` executes its body and reports whether it encountered an exception. If so, then that exception is returned when you `co_await`.

First, let's look at the promise.

The promise holds only two things: The coroutine that is awaiting the result and the exception that occurred in the coroutine body.

The promise implements a lazy-start coroutine. This avoids having to deal with race conditions if the coroutine body completes before the caller manages to `co_await` the result. Instead, we suspend the coroutine immediately, and resume it only when manually resumed.

To make it easier to access the coroutine handle, we have a `coroutine()` function that recovers the `coroutine_handle` from the promise.

When the caller awaits the `all_completed_result`, the `await_suspend` remembers the caller's coroutine handle and then resumes the promise's coroutine from its initial suspension point.

If the coroutine runs to completion successfully, the compiler will call `return_void()`, which we ignore.

If the coroutine encounters an exception, the compiler will call `unhandled_exception()`, which saves the exception in the `eptr` member.

After the coroutine completes (either successfully or with an exception), the compiler awaits the `final_suspend`. We return a `suspend_never`, which allows the coroutine frame to be destroyed, but not before we resume the awaiting coroutine so it can see what exception occurred, if any.

The `all_completed_result` contains a reference to the associated coroutine promise. When you `co_await` it, it starts the coroutine body (and remembers the awaiting coroutine's handle for later resumption). After the coroutine completes, the compiler will call `await_resume()`, whhich returns the exception pointer that was saved in the promise.

Given this, we can go back to our lambda-coroutine-based solution, since we got rid of the apartment-switching behavior of C++/WinRTs `IAsyncAction` awaiter, and our custom coroutine promise just returns the exception directly.

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

    auto capture_exception = [](auto& async)
        -> all_completed_result {
        co_await std::move(async);
    };

    auto accumulate = [&](std::exception_ptr e) {
        if (eptr == nullptr) eptr = e;
    };

    (accumulate(co_await capture_exception(asyncs)), ...);

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

Okay, now that we have our coroutine promise, we can start making refinements to it. We'll start next time.

[1] The apartment-switching behavior is not inherent in C++ coroutines. Rather, it's a design decision of the C++/WinRT library. In theory, it would be possible to add a way to configure the C++/WinRT awaiter to disable the apartment-switching behavior, say by doing something like

```
    co_await action.resume_any_apartment();
```

where a new `resume_any_apartment()` method returns a wrapper around the original `IAsyncAction` that triggers a different custom awaiter which does not perform an apartment switch.