How to wait for multiple C++ coroutines to complete before propagating failure, memory allocation failure

devblogs.microsoft.com/oldnewthing/20230704-00

July 4, 2023

Raymond Chen

Last time, we used symmetric transfer to avoid stack build-up when calling back and forth between the outer driver function `when_all_completed` and the coroutine created from our custom promise. But we lost sight of the original reason for using the custom promise, which was to allow us to deal with memory allocations.

The concern we had was that if a memory allocation error occurred when allocating the coroutine frame, the runtime normally throws `std::bad_alloc`, and the coroutine body never runs. Furthermore, we currently do not distinguish between an allocation failure in the creation of the coroutine, as opposed to an allocation failure that was thrown from the coroutine body.

One way to deal with this is to catch the initial coroutine creation, separately from `co_await`ing it.

```
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 ensure_alloc = [&](auto& async) noexcept
    {
        return capture_exception(async);
    };

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

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

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

If a memory allocation failure occurs when trying to allocate the coroutine frame, then the `std::bad_alloc` is thrown out of the initial call to `capture_exception()`. The `noexcept` on the `ensure_alloc` converts this exception into a `std::terminate`, and the program fails fast.

We are unable to make forward progress, and throwing an exception from `when_all_completed` would be misinterpreted as propagating an exception from one of the awaitables. The only remaining option is to simply terminate the process.

An alternate solution is to encode the "terminate on `std::bad_alloc` trying to allocate the coroutine frame" in the promise itself. If the promise has a custom `operator new`, then that custom operator is used to allocate the coroutine frame. (Similarly, a custom `operator delete` is used to deallocate the coroutine frame.)

Therefore, instead of modifying `when_all_completed`, we can modify the `all_completed_promise`:[1]

```
struct all_completed_promise
{
    ...

    // Fail fast if unable to allocate the coroutine frame.
    void* operator new(std::size_t n) try {
        return ::operator new(n);
    } catch (...) { std::terminate(); }
};
```

Another option might be to invent a special exception that means “Sorry, I was unable to perform the requested operation. The awaitables have not necessarily all run to completion.”

```
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 ensure_alloc = [&](auto& async) try
    {
        return capture_exception(async);
    } catch (...) {
        throw winrt::hresult_error(
            HRESULT_FROM_WIN32(ERROR_CAN_NOT_COMPLETE));
    };

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

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

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

Or, if you prefer to apply this policy in the promise:

```
struct all_completed_promise
{
    ...

    void* operator new(std::size_t n) try {
        return ::operator new(n);
    } catch (...) {
        throw winrt::hresult_error(
            HRESULT_FROM_WIN32(ERROR_CAN_NOT_COMPLETE));
    };
};
```

And now you can document that `ERROR_CAN_NOT_COMPLETE` is the error code that means, “Sorry, I was not able to perform the required duties.”

I’m not sure this special exception is warranted, however, because there’s no way for the caller to recover. I mean, you call this function to wait for all of the operations to complete, and it says “Sorry, I couldn’t do that.” Now what are you going to do? The operations haven’t

run to completion. You can't perform the next step in your algorithm. But you can't just give up now, because you want to make sure those operations are complete and won't create race conditions.

You're stuck between the same rock and hard place that the `when_all_complete` function found itself in, and your options are the same: Either fail fast now, or tell the caller "Hi, so something is fatally, unrecoverably wrong, and there's no point continuing because if you do, there is going to be memory corruption and race conditions and all sorts of badness."

And what's your caller going to do? Either fail fast or pass the buck to its caller.

Eventually, there will be nobody left to pass the buck to, and the program will terminate with an unhandled exception. Even worse, the termination will occur at a point far, far away from the root cause, so the poor developer who gets stuck with the unhandled exception is going to have quite an unwanted adventure trying to figure out what happened.

May as well save everyone the trouble of trying to deal with a problem that they cannot recover from. Fail fast right away so you can identify the root cause quickly.

Next time, we'll try to take the dynamic memory allocation out of this entire scenario.

[1] You'd think you could write

```
// Fail fast if unable to allocate the coroutine frame.
void* operator new(std::size_t n) noexcept {
    return ::operator new(n);
}
```

so that any exception thrown by `::operator new` terminates the process. This does work, but a `noexcept operator new` tells the compiler that it will return `nullptr` on memory allocation failure, in which case the compiler will call a developer-provided `get_return_object_on_allocation_failure()` function to produce the `all_completed_result` that will be returned by the (nonexistent) coroutine. Therefore, if we apply `noexcept` to our custom `operator new`, we also have to provide a `get_return_object_on_allocation_failure()` function. Removing the `noexcept` from our `operator new` avoids this requirement.