How to wait for multiple C++ coroutines to complete before propagating failure, preallocating the coroutine frame

devblogs.microsoft.com/oldnewthing/20230705-00

July 5, 2023

Raymond Chen

Last time, we dealt with memory allocation failures in our `when_all_completed` coroutine by terminating immediately. But can we avoid memory allocation failures entirely?

We learned some time ago that the coroutine frame consists of the following things:

- Promise object.
- Inbound parameters.
- Local variables.
- Temporaries.
- Compiler overhead.

This size is fixed for a given coroutine, but it varies from coroutine to coroutine depending on what goes into the coroutine body.

But if we can anticipate the necessary size for the coroutine frame, we can preallocate the memory in the caller's frame, thereby avoiding the need for dynamic allocation.

Recall that our coroutine body looks like this:

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

Let's look at the sizes of the things that contribute to the coroutine frame. Some of them are easy to calculate:

- Promise object: `sizeof(all_completed_promise)`.
- Inbound parameters: Size of a reference is `sizeof(void*)`.
- Local variables: `sizeof(std::exception_ptr)`.

Temporaries will require some thought. What temporaries are created by our lambda?

The `co_await` expression triggers the possible creation of a temporary awaiter. So we'll have to calculate the size of the awaiter associated with the `async` object.

The last piece is the compiler overhead. We will have to determine this experimentally because each compiler is welcome to implement coroutines in its own way. (Although there is a de facto ABI shared by the three major compiler vendors.)

Determining the awaiter requires us to reimplement the algorithm the compiler uses to find the awaiter. I'll adapt the code from the C++/WinRT library:

```
class awaiter_finder
{
    template<typename T>
    static void find_co_await_member(T&&, ...);
    template<typename T>
    static auto find_co_await_member(T&& value, int)
    -> decltype(static_cast<T&&>(value).operator co_await()) {
         return static_cast<T&&>(value).operator co_await();
    }
    template<typename T>
    using member_awaiter = decltype(find_co_await_member(std::declval<T>(), 0));

    template<typename T>
    static void find_co_await_free(T&&, ...);
    template<typename T>
    static auto find_co_await_free(T&& value, int)
    -> decltype(operator co_await(static_cast<T&&>(value))) {
         return operator co_await(static_cast<T&&>(value));
    }
    template<typename T>
    using free_awaiter = decltype(find_co_await_free(std::declval<T>(), 0));

public:
    template<typename T>
    static auto get_awaiter(T&& value)
    {
        if constexpr (!std::is_same_v<member_awaiter<T>, void>) {
            return find_co_await_member(static_cast<T&&>(value), 0);
        } else if constexpr (!std::is_same_v<free_awaiter<T>, void>) {
            return find_co_await_free(static_cast<T&&>(value), 0);
        } else {
            return (char)0;
        }
    }

    template<typename T>
    using type = decltype(get_awaiter(std::declval<T>()));
};
```

This uses SFINAE to detect whether a class has `co_await` as a member operator and to detect whether it supports `co_await` as a free function operator. These are the two cases where the `co_await` will create a temporary awaiter, and in those cases, we return the (suitably decayed)[1] type of that awaiter. In the case where the object is its own awaiter, there is no temporary, so the extra memory required for the awaiter is zero. There are no objects of size zero in C++, so we just use a `char`, which has size 1. This is an overestimate, but that's okay.

We can now use the `awaiter_finder` to build up the storage for holding our coroutine frames. Since only one coroutine frame is needed at a time, we just need a buffer that is big enough to hold the largest one.

```
template<typename... Types>
struct coroutine_frame_storage
{
    void* overhead[50];
    struct alignas(typename awaiter_finder::type<Types>...) {
        char buffer[(std::max)({ sizeof(
            typename awaiter_finder::type<Types>)... })];
    } awaiter;
};
```

There is no way to calculate the compiler overhead except by just playing around with the compiler. In my experiments, the compiler overhead is around 48 bytes, so I'm going to be a little generous and say it's 200 bytes.

We can preallocate the memory for the coroutine frame in the caller and pass it to the coroutine function as a bonus parameter. The custom `operator new` then knows to use that storage for the coroutine frame.

```
struct all_completed_promise
{
    ...

    template<typename Lambda, typename...Storage, typename Async>
    void* operator new(
        std::size_t n,
        Lambda&&,
        coroutine_frame_storage<Storage...>& storage,
        Async&&) {
        // If this terminates, then we need to increase the
        // extra overhead in coroutine_frame_storage.
        if (n > sizeof(storage)) std::terminate();

        return std::addressof(storage);
    }

    void operator delete(void*) {}
};

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

    auto capture_exception = [](auto& storage, 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(storage, asyncs)), ...);

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

Note that this code reuses the same `coroutine_frame_storage` for each `co_await`. This requires that the coroutine storage be deleted before the next one starts. We accomplished this by having the `all_completed_result` destroy the coroutine when it resumes. That way, the storage is no longer in use when the next `co_await` begins.

This was an awful lot of work to avoid “out of memory” errors, and it involves a little bit of chumminess with the compiler (to calculate the size of the coroutine frame). Mind you, precalculating the coroutine frame size is one of the things called out in the original coroutine specification for scenarios that must avoid dynamic memory allocation, so at least what we’re doing is implicitly acceptable to the authors of the coroutine specification.

But maybe we can avoid needing to be chummy at all.

We'll look at this next time. The secret is to avoid coroutines.

[1] The decay happens when we return the type from an `auto` method. If the return type of the `co_await` operator is a reference, the referred-to object is copied and returned.

[2] Note that we could not do

```
template<typename... Types>
struct coroutine_frame_storage
{
    void* overhead[50];
    char awaiter[std::max(
        { sizeof(typename awaiter_finder::type<Types>)... })];
};
```

This version allocates the correct number of bytes, but it does not preserve any alignment requirements awaiter.