C++ coroutines: Managing the reference count of the coroutine state

devblogs.microsoft.com/oldnewthing/20210409-00

April 9, 2021

Raymond Chen

Last time, we hooked up the `co_await` of the `simple_task` and we had a brief glimpse into the the management of the reference count on the promise (and therefore also the coroutine state). Today we'll fill in the rest of the story.

```
// ⟦simple_promise_base reference count methods⟧ ≔

void increment_ref() noexcept
{
    m_refcount.fetch_add(1, std::memory_order_relaxed);
}

auto as_handle() noexcept
{
    return std::experimental::coroutine_handle<Promise>::
        from_promise(*as_promise());
}

void decrement_ref() noexcept
{
    auto count = m_refcount.fetch_sub(1,
        std::memory_order_release) - 1;
    if (count == 0)
    {
        std::atomic_thread_fence(std::memory_order_acquire);
        as_handle().destroy();
    }
}
```

Incrementing the reference count can be done with relaxed ordering because there is no real dependency on memory accesses. The client could equally well have accessed the fields before or after incrementing the reference count. But decrementing it is trickier.

The client cannot access the object after decrementing the reference count, because once it decrements the reference count, the object could disappear. This means that we have to use release memory order on the release so that any final updates to the object are retired before

the object becomes eligible for destruction.

You might think that the release memory ordering is needed only when decrementing to zero, but that's not true. For example, the current thread might decrement to one, and another thread decrements to zero and destroys the object. If any writes from the current thread were delayed, they would be modifying memory after it was freed, corrupting the heap, and creating a very bad week for a future developer who is trying to track down a rare memory corruption bug.

If we realize that we are the one who decremented the reference count to zero, we take an acquire memory fence to ensure that the state of the coroutine is properly suspended before we destroy it. We don't want to have advanced any reads of the coroutine state because those might have occurred before the coroutine fully suspended itself.

Destroying the coroutine state uses some helper functions we haven't seen before. Let's go back to our old diagram of the coroutine state:

Compiler bookkeeping	←	`address`
Promise object	←	`promise*`
Coroutine "stack frame"		

There are three ways of referring to the coroutine state. There's the `coroutine_handle`, which is an object that represents the coroutine state. You can convert between `coroutine_handle` and a a `void*`, which is known as the `address`. And you can also convert between `coroutine_handle` and the corresponding `promise`.

	`Promise`	
`coroutine_handle<Promise>::from_promise()`	↓ ↑	`handle.prom`
	`coroutine_handle`	
`coroutine_handle<>::from_address()`	↑ ↓	`handle.addr`
	`void*`	

Converting to and from a raw address is handy when you want to pass a coroutine handle through an ABI that uses a raw pointer, such as a thread pool callback function.

And converting to and from a promise is handy in cases like this where the promise wants to talk about its own coroutine state, or conversely when you have a coroutine state and want to access data stored in the promise.

The rest of the reference counting is pretty boring. The `promise_ptr` is just a smart wrapper around the reference-counted raw promise pointer. Sadly, it's also a lot of code.

```
// ⟦implement promise_ptr⟧ ≔
template<typename T>
struct promise_ptr
{
    using promise_t = simple_promise<T>;
    promise_t* promise;
    promise_ptr(promise_t* initial = nullptr) noexcept
        : promise(initial) {}

    promise_ptr(promise_ptr const& other) noexcept
        : promise(other.promise)
    {
        increment_promise_ref(promise);
    }
    promise_ptr(promise_ptr&& other) noexcept
        : promise(std::exchange(other.promise, nullptr))
    {
    }

    ~promise_ptr()
    {
        decrement_promise_ref(promise);
    }

    promise_ptr& operator=(promise_ptr const& other)
    {
        if (promise != other.promise)
        {
            increment_promise_ref(promise);
            decrement_promise_ref(
                std::exchange(promise, other.promise));
        }
        return *this;
    }
    promise_ptr& operator=(promise_ptr&& other) noexcept
    {
        if (promise != other.promise)
        {
            decrement_promise_ref(std::exchange(promise,
                std::exchange(other.promise, nullptr)));
        }
        return *this;
    }

    void swap(promise_ptr& other) noexcept
    {
        std::swap(promise, other.promise);
    }

    promise_t* operator->() const noexcept
    {
        return promise;
```

```
        }

        static void increment_promise_ref(
            promise_t* promise) noexcept
        {
            if (promise) promise->increment_ref();
        }

        static void decrement_promise_ref(
            promise_t* promise)
        {
            if (promise) promise->decrement_ref();
        }
    };
```

This is fairly straightforward stuff. There's an amusing chain of `std::exchange` calls in the rvalue assignment operator.

`nullptr`
↓
`other.promise`
↓
`promise`
↓
`decrement_promise_ref`

The `nullptr` displaces the `other.promise`, which trickles down to `promise`, and the displaced old `promise` goes to `decrement_promise_ref` for disposal.

That fills in the last of the placeholders for our `simple_task`. We now have a promise and task that can be used to create new coroutines.

```
async_helpers::simple_task<int> Step1();
async_helpers::simple_task<int> Step2();

async_helpers::simple_task<int> CalculateAsync()
{
    auto part1 = co_await Step1();
    auto part2 = co_await Step2();
    co_return part1 + part2;
}
```

Next time, we'll discuss some of the caveats with this class and how the assumed usage pattern influenced the design.

Raymond Chen

Follow