C++ coroutines: Getting rid of our reference count

devblogs.microsoft.com/oldnewthing/20210416-00

April 16, 2021

Raymond Chen

In an earlier installment, we simplified our `promise_ptr` type, and one of the consequences of this is that there are no remaining caller of `increment_ref`. This means that we don't need a reference count at all and can rely on the state changes to tell us when to destroy the promise: When the awaiter has obtained the result or, or when the coroutine completes and discovers that the awaiter has abandoned its effort to obtain the result.

This means that we need a new sentinel value to mean "abandoned".

If I could target C++20, then I would have `noop_coroutine_handle` available to use for another sentinel value.[1] But I want to be able to run on MSVC's coroutine implementation in C++17, so that's off the table.

I can't think of another coroutine handle that we can use for our new sentinel value, so I'm going to use a trick: Coroutine handles can be converted to and from `void*`, and the `void*` represents the "address" of the coroutine, for some unspecified meaning of *address*.

If we assume that the term *address* means that the result must be the address of *something*, then we can convert all of our coroutine handles to `void*`, and our sentinel values become pointers that are known not to point to a valid coroutine. We can't use our `this` pointer, because the coroutine might have the promise as its first member, and that would make the addresses match. Similarly, we can't use the address of the promise's first nonstatic member variable. But we could use a pointer to a second or subsequent nonstatic member variable (assuming it isn't marked as `[[no_unique_address]]`). Or we could declare some static variables for the sole purpose of providing unique addresses.

But instead, I'm going to use some hard-coded invalid pointer values, like `(void*)1`, because the code generation for them will be more efficient.

Okay, so our values are

- `nullptr` if the coroutine is running and nobody is awaiting ("started").
- `(void*)1` if the coroutine has completed ("completed").
- `address` if another coroutine is awaiting ("awaiting").

- `(void*)2` if the awaiter has been abandoned ("abandoned").

We have two life cycles running in parallel: One of the life cycle of the coroutine body, and the other is the life cycle of the task. The coroutine body will eventually complete. The task will usually (but not always) be awaited, and then will always be abandoned.

In other words, we have started → awaiting (maybe) → abandoned, and the completed step can be inserted at any arrow.

The results are

- started → completed → awaiting → abandoned: This is the case where the coroutine completes before anybody can await on it.
- started → awaiting → completed → abandoned: This is the common case where the task is awaited before it completes.
- started → awaiting → abandoned → completed: This case is not possible because the awaiting phase will always wait for completed before continuing to abandoned.
- started → completed → abandoned: This is the case where the coroutine completes before the task is abandoned without ever having been awaited.
- started → abandoned → completed: This is the case where the coroutine is abandoned before it completes.

Studying the four legal transition sequences leads to this state transition action chart:

From To	completed	awaiting	abandoned
started	Do nothing	Do nothing	Do nothing
completed		Resume awaiter	Destroy promise
awaiting	Resume awaiter		
abandoned	Destroy promise		

Okay, now we can implement the diagrams.

```
template<typename T>
struct simple_promise_base
{
    using Promise = simple_promise<T>;
    auto as_promise() noexcept
    {
        return static_cast<Promise*>(this);
    }

    std::atomic<uint32_t> m_refcount{ 2 };
    // std::mutex m_mutex;
    std::atomic<void*> m_waiting{ running_ptr };
    simple_promise_result_holder<T> m_holder;

    static constexpr void* running_ptr = nullptr;
    static constexpr void* completed_ptr = reinterpret_cast<void*>(1);
    static constexpr void* abandoned_ptr = reinterpret_cast<void*>(2);

    ...

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

    void destroy()
    {
        as_handle().destroy();
    }

    void abandon()
    {
        auto waiting = m_waiting.exchange(abandoned_ptr,
            std::memory_order_acquire);
        if (waiting != running_ptr) destroy();
    }

    ...

    auto final_suspend() noexcept
    {
        struct awaiter : std::experimental::suspend_always
```

```
            {
                simple_promise_base& self;
                void await_suspend(
                    std::experimental::coroutine_handle<> /* handle */)
                    const noexcept
                {
                    auto waiter = self.m_waiting.exchange(completed_ptr,
                        std::memory_order_acq_rel);
                    // self.decrement_ref();
                    if (waiting == abandoned_ptr) self.destroy();
                    else if (waiting != running_ptr) std::experimental::
                        coroutine_handle<>::from_address(waiting).resume();
                }
            };
            return awaiter{ {}, *this };
        }

        bool client_await_ready()
        {
            auto waiting = m_waiting.load(
                std::memory_order_acquire);
            assert(waiting == running_ptr || waiting == completed_ptr);
            return waiting != running_ptr;
        }

        auto client_await_suspend(
            std::experimental::coroutine_handle<> handle)
        {
            return m_waiting.exchange(handle.address(),
                std::memory_order_acq_rel) == running_ptr;
        }

        ...
    };

    struct promise_deleter
    {
        void operator()(simple_promise_base<T>* promise) const noexcept
        {
            promise->abandon();
        }
    };
```

We define a few constants to represent our sentinel addresses for the completed and abandoned states. The remaining methods follow our flowchart above to change to the new state and inspect the previous state to see what action needs to be taken.

Abandoning the promise needs acquire semantics so that the values used in the destruction of the promise are not fetched prematurely.

Does abandoning the promise require release semantics? That would be the case if our transition to the abandoned state could cause another thread to destroy the promise. We need to make sure our modifications to the promise (say, due to us having moved resources out of it) are visible to the other thread.

In the cases where abandonment leads directly to destruction, we don't need release semantics because we are the one who will destroy the promise. The only remaining case is where the coroutine is abandoned before it completes, and in that case, we haven't made any changes to the promise, so there is nothing that needs to get flushed out.

Therefore, my conclusion is that we do not need release semantics on abandonment. (I could be wrong.)

The next interesting point is the completion of the coroutine in the `final_suspend` awaiter. When we set the value to `completed_ptr`, we need release semantics in case there is no awaiter yet, to ensure that the result of the coroutine is properly published to the eventual awaiter. And we need acquire semantics in case there is an awaiter already registered, so that we use the values that were published when the awaiter registered itself with the promise.

In `client_await_ready`, we check whether the coroutine has already completed by peeking at the state in `m_waiting`. We do this with acquire semantics so that, in the event the answer is "Already completed", we will read the values that were published at completion.

In `client_await_suspend`, we publish the awaiter with release semantics so that the suspension of the awaiting coroutine is published to the thread that will resume it. And we also do it with acquire semantics, so that we can read the results from `m_holder` in the case that the coroutine has already completed.

And finally, the `promise_deleter` now calls `abandon` instead of `decrement_ref`.

Okay, that eliminates our reference count. Next time, we'll eliminate the atomic variable that we use to remember what's in the result holder object.

Bonus chatter: You may have noticed that getting rid of the reference count also fixes the problem of an awaiting coroutine being destroyed while suspended: If that happens, the `promise_ptr` in the awaiter is destructed, and that abandons the coroutine. This means that when the coroutine completes, it will see that nobody is awaiting and won't try to resume a destroyed awaiting coroutine.

[1] Note, however, that there is no requirement that the `noop_coroutine_handle` be unique, so we would have to keep a well-known handle in our class that everybody agrees to use. This adds another memory access to our code paths which check for the sentinel value.

Raymond Chen

Follow