C++ coroutines: Getting rid of our mutex

devblogs.microsoft.com/oldnewthing/20210415-00

April 15, 2021

Raymond Chen

Our coroutine implementation uses a mutex to guard against the race condition where a coroutine completes at the same time another thread tries to wait for its completion. The race condition in question is this one:

Awaiter	Completer
if coroutine has not yet completed {	
	Mark coroutine as completed
	Resume anyone who is waiting
Sign up to be resumed }	

If this race condition is realized, then you have a coroutine that has signed up to be resumed, but the completion of the coroutine failed to resume it. The awaiter never wakes up.

But we can solve this race condition without a lock: We just need to have two special sentinel values: One for the initial state where the coroutine has not yet completed, nor has an awaiter registered for resumption. Another to mean that the coroutine has completed, but no awaiter has registered yet. The third case is where the awaiter has registered, but the coroutine hasn't completed: For that, we will use the awaiter's coroutine handle.

The completer exchanges the value with the "completed" sentinel value to indicate that the coroutine has completed, and if the old value was not the initial value, then it means that there is a continuation, and the completer resumes the awaiter.

The awaiter exchanges the value with the continuation, and if the old value was "completed", then it resumes itself immediately.

So now we have to find two values to use as sentinel values.

We've already been using one sentinel value: `nullptr` is a sentinel value that means that the coroutine has started but no awaiter has registered. We just need to find another one. And it turns out it was right under our noses: We can use the running coroutine itself! The running coroutine will never await itself, so we can use its own handle as our second sentinel.

Okay, now that we have our plan, let's go implement it.

```
template<typename T>
struct simple_promise_base
{
    ...
    // std::mutex m_mutex;
    // std::experimental::coroutine_handle<> m_waiting{ nullptr };
    std::atomic<std::experimental::coroutine_handle<>>
        m_waiting{ nullptr };
    simple_promise_result_holder<T> m_holder;

    ...
    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(handle,
                    std::memory_order_acq_rel);
                self.decrement_ref();
                if (waiter) waiter.resume();
            }
        };
        return awaiter{ {}, *this };
    }

    ...

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

    ...
};
```

For the `client_await_suspend`, we return `true` (meaning that we should suspend) if the coroutine hasn't completed yet, which we detect by noticing that the previous value of `m_waiting` is null, representing the fact that the coroutine is still running.

The exchange when the coroutine completes uses releases semantics because we need to ensure that the results are published before we announce that the results are ready. This covers the case where the coroutine completes before the client gotten around to calling `co_await`: In that case, the client will observe that the coroutine is complete and immediately try to read the results. We need to make sure the results are published for the client to read.

The exchange when the coroutine completes also uses acquire semantics because we need to ensure that we don't try to load any state from the handle we're about to resume until we've atomically obtained it.[1]

The exchange when suspending the awaiting coroutine uses release semantics for a similar reason: We want to make sure the suspension of the awaiting coroutine has been published before we publish the awaiting coroutine handle for the promise coroutine to use.

And the exchange when suspending the awaiting coroutine also uses acquire semantics to match the release semantics when the coroutine completes: If the coroutine has just completed, we need to make sure we read the freshly-published result.

But we finally did it. We got our simple promise and simple task to be lock-free.

It turns out our simplification of `promise_ptr` had other consequences. We'll explore them next time.

[1] This seems like an odd thing to have to protect against. How can we possibly load any state from the handle we're about to resume before we obtain the handle? Can the CPU predict the future and load a value dependent upon an address it hasn't obtained yet?

Maybe,[2] but even in the absence of time travel (or really good speculation), it's possible that the memory is cached locally on the CPU from some previous usage, and we need to make sure that the cached value is not used and new values are loaded afresh.

[2] The answer is "Yes". This behavior is permitted by the Alpha AXP memory model. And it happens for basically the reason I gave above: The value at the dependent address is locally cached.

Raymond Chen

Follow