C++ coroutines: Improving cold-start coroutines which complete synchronously

devblogs.microsoft.com/oldnewthing/20210422-00

April 22, 2021

Raymond Chen

Last time, we learned that the naïve implementation of cold-start coroutines is susceptible to stack build-up. What we want is for `await_suspend` to return `false` if the coroutine completed synchronously. This is tricky because that reintroduces a race condition where the coroutine runs asynchronously and completes at the same time we try to transition from synchronous to asynchronous behavior in the awaiter.

For example, we could try this:

```
auto final_suspend() noexcept
{
    struct awaiter : std::experimental::suspend_always
    {
        simple_promise_base& self;
        void await_suspend(
            std::experimental::coroutine_handle<>)
            const noexcept
        {
            if (self.m_waiting()) self.m_waiting();
        }
    };
    return awaiter{ {}, *this };
}

auto client_await_suspend(
    std::experimental::coroutine_handle<> handle)
{
    as_handle().resume();
    m_waiting = handle;
    return m_holder.is_empty();
}
```

The idea here is that we don't register a resumption coroutine handle immediately. Instead, we let the coroutine resume, and if it completes synchronously, its `final_suspend` resumes the awaiter if one exists. In the case of synchronous completion, there won't be an awaiter yet. Back in the awaiter, we register the awaiting coroutine for resumption after the synchronous portion of the awaited-for coroutine finishes, and then we peek at whether it indeed completed synchronously. If so, then we return `false`.

This algorithm doesn't work because the coroutine may have continued asynchronously, and the asynchronous completion creates a data race against the awaiter trying to check whether the coroutine completed synchronously.

We'll have to bring back the atomic waiter.[1]

Return to the version where `m_waiting` was a `std::atomic<void*>`.

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

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

We now have implementations for both cold-start and hot-start coroutines. Next time, we'll unify them by using another coroutine trick.

[1] "Atomic Waiter" sounds like a failed superhero from the 1950's.

Raymond Chen

Follow