C++ coroutines: Awaiting the simple_task

devblogs.microsoft.com/oldnewthing/20210408-00

April 8, 2021

Raymond Chen

Last time, we accepted the parameter passed to `co_return` and stored it into our promise. This time, we'll deal with the consumption side and wait for the answer to appear.

```
// ⟦awaiter support methods⟧ ≔
bool client_await_ready()
{
    assert(m_waiting == nullptr);
    return !m_holder.is_empty();
}

auto client_await_suspend(
    std::experimental::coroutine_handle<> handle)
{
    auto guard = std::lock_guard{ m_mutex };
    if (!m_holder.is_empty()) return false;
    m_waiting = handle;
    return true;
}

T client_await_resume()
{
    return m_holder.get_value();
}

auto get_awaiter() noexcept
{
    // ⟦return an awaiter that waits for the coroutine
    // to complete⟧ ≔
    struct awaiter
    {
        simple_promise_base& self;

        bool await_ready()
        {
            return self.client_await_ready();
        }

        auto await_suspend(
            std::experimental::coroutine_handle<> handle)
        {
            return self.client_await_suspend(handle);
        }

        T await_resume()
        {
            return self.client_await_resume();
        }
    };
    return awaiter{ *this };
}
```

The real work happens in the `client_` methods in the `simple_promise_base`, and the awaiter just forwards everything to those methods, so I'm going to talk about the awaiter and the `client_` methods as if they were the same thing.

Our awaiter's `await_ready` first asserts that nobody else is waiting for promise. We allow only one `co_await` because multiple `co_await` is not compatible with a move-only type: If the type is move-only, then you can't return it more than once because returning it also gives it away. There's nothing to return to the second `co_await`.

Moving the value in response to `co_await` also avoids potentially-expensive copies.

After the correctness check, we see if the awaited-for coroutine is still running by seeing if the result holder is still empty. If it's not empty, then the coroutine has already produced a result; we return `true` to go straight to `await_resume`.

If `await_ready` concludes that the awaited-for coroutine is still running, then the compiler will suspend the current coroutine and then call `await_suspend`. We use a mutex for this, because we need to avoid a race between signing up for resumption and the awaited-for coroutine reaching its `final_suspend` (which resumes the coroutine). We make one last check if the awaited-for coroutine is still running, to close the race window where the awaited-for coroutine finishes in between the `await_ready` and the acquisition of the lock in `await_suspend`.

When the coroutine resumes, `await_resume` produces the value or rethrows the exception. Note that we specify the return type explicitly as `T` rather than using `auto`. This is important in the case where `T` is a reference.

The resumption occurs when the awaited-for coroutine reaches its `final_suspend`.

```
        auto final_suspend() noexcept
        {
            // ⟦return an awaiter that decrements the reference count
            //  and resumes any awaiting coroutine⟧ ≔
            struct awaiter : std::experimental::suspend_always
            {
                simple_promise_base& self;
                void await_suspend(std::experimental::coroutine_handle<>) const
noexcept
                {
                    std::experimental::coroutine_handle<> handle;
                    {
                        auto guard = std::lock_guard{ self.m_mutex };
                        handle = self.m_waiting;
                    }
                    self.decrement_ref();
                    if (handle) handle.resume();
                }
            };
            return awaiter{ {}, *this };
        }
```

At final suspension, we first check to see if a coroutine is actively awaiting our result. This requires the mutex to avoid racing against the `get_awaiter` we saw above.

Once we capture the awaiting coroutine's handle (if any), we decrement our own reference count, since the coroutine is no longer running. The only reference count remaining belongs to the `simple_task`. (If the caller threw away the `simple_task` without awaiting it, then that decrement will destruct the coroutine state immediately.)

And then we resume the awaiting coroutine, if any. When that awaiting coroutine destructs the `simple_task`, that will drop the reference count to zero and destruct the coroutine state.

The last missing piece is the reference count management. Sadly, this is the largest single piece of the entire coroutine infrastructure, and it's almost entirely uninteresting! We'll take up the boring details next time.

Bonus chatter: It's important that we wait until `await_suspend` to decrement the reference on the promise, rather than doing it eagerly in `await_ready`. The `await_ready` method is called while the coroutine is still in the executing state, and you cannot destruct an executing coroutine. On the other hand, `await_suspend` is called after the coroutine has transitioned to the suspended state, at which point it is now safe to destroy.

Raymond Chen