C++ coroutines: The initial and final suspend, and improving our return_value method

devblogs.microsoft.com/oldnewthing/20210331-00

March 31, 2021

Raymond Chen

Last time, we had a basic implementation of a promise type but left with the question of what the `initial_suspend` and `final_suspend` are for.

When the compiler encounters a coroutine function, it performs multiple rewrite passes. One of the early passes produces the following:

```
return_type MyCoroutine(args...)
{
    create coroutine state
    copy parameters to coroutine frame
    promise_type p;
    auto return_object = p.get_return_object();

    try {
        co_await p.initial_suspend(); // ¹
        coroutine function body²
    } catch (...) {
        p.unhandled_exception();
    }
    co_await p.final_suspend();
    destruct promise p
    destruct parameters in coroutine frame
    destroy coroutine state
}
```

This rewrite is where the `initial_suspend` and `final_suspend` enter the picture.

After constructing the promise `p`, it calls `p.get_return_object()` to obtain the object that is passed back to the caller. Next, the coroutine awaits whatever `initial_suspend()` returns.

There are two common choices for `initial_suspend()`: If you return a `suspend_never` or some other awaiter that doesn't suspend the coroutine, then the coroutine keeps running until the first suspending `co_await`. This is the model for "hot-start" coroutines

which execute synchronously during their construction and don't return an object until the first suspension inside the function body.

On the other hand, if you return an awaiter which results in the coroutine suspending, then the caller gets its return object right away. Nothing in the coroutine function body has executed yet. When the caller eventually performs a `co_await` on the return object, the return object resumes the coroutine.[3] This is the model for "cold-start" coroutines which don't begin executing until they are awaited.

Conversely, there is a `final_suspend` at the end of the coroutine after the coroutine function body has finished. This gives you a chance to do any extra cleanup, as well as perform some other magic we'll look at in a future installment.

One way to improve in our implementation of the `coroutine_traits<result_holder>` promise type is our handling of `return_value`. First, there's a simple improvement of accepting the value by reference rather than value, so we can forward it into the `result_holder`, thereby avoiding some extra move operations.

```
void return_value(T&& v) const
{
    holder.set_result(std::move(v));
}

void return_value(T const& v) const
{
    holder.set_result(v);
}
```

But there's a bigger problem to fix: We release the waiting coroutines too soon.[4]

Consider a coroutine function that goes like this:

```
result_holder<int> SomethingAsync()
{
    auto guard = std::lock_guard{ mutex };
    co_return 42;
}
```

When this function reaches the `co_return`, the compiler generates a call to `return_value`, and our implementation of `return_value` sets the result into the holder, which immediately releases any waiting coroutines.

But those waiting coroutines might want to acquire the same mutex that the `Something-Async` function still owns. You end up in a hurry-up-and-wait situation, where we wake up a coroutine, only to have it block immediately.

The scenario is even more dire if we resume the waiting coroutines synchronously, because the waiting coroutine may want to acquire the mutex, but it can't because `SomethingAsync` still owns the mutex, and `SomethingAsync` won't resume execution until after `return_value` returns, which can't happen because it's waiting for the resumed coroutine to reach its next suspension point.

More generally, local variables are still alive at the point of the `co_return`, so any resources held by those local variables are still active at the point of the `return_value`.

The solution is to break the `return_value` into two steps. The first step executes immediately: Saving the value or exception into the `holder`. But we don't wake up the waiting coroutines yet. Leave them suspended for a little while longer.

After the return value or exception is captured, the local variables in the coroutine function body are destructed when we leave the scope of the `try` block that wraps the coroutine function body. Outside the `try` block, we perform a `final_suspend`, and that's where we can take the second step of waking the waiting coroutines.

To implement this, we need to add a few new methods to our `result_holder`:

```
struct result_holder_state
{
    ...

    void stage_result(T v)
    {
        if (kind.load(std::memory_order_relaxed)
            == result_kind::unset) {
            new (std::addressof(result.wrap))
                wrapper{ std::forward<T>(v) };
            kind.store(result_kind::value,
                std::memory_order_release);
        }
    }

    void stage_exception(std::exception_ptr ex) noexcept
    {
        if (kind.load(std::memory_order_relaxed)
            == result_kind::unset) {
            new (std::addressof(result.ex))
                std::exception_ptr{ std::move(ex) };
            kind.store(result_kind::exception,
                std::memory_order_release);
        }
    }

    void complete(node_list& list) noexcept
    {
        this->resume_all(list);
    }
};
```

We break the `set_result` into two parts: `stage_result` puts the result in the state, but doesn't resume anybody yet. The resumption of waiting coroutines happens when we call `complete`.

We hook up these methods to the main `result_holder` class:

```
struct result_holder
{
    ...

    void stage_result(T result) const
    {
        this->get_state().stage_result(std::move(result));
    }

    void stage_exception(std::exception_ptr ex) const noexcept
    {
        this->get_state().stage_exception(std::move(ex));
    }

    void complete() const noexcept
    {
        this->action_impl(&state::complete);
    }
};
```

And we revise our promise to use these new methods:

```
            void return_value(T&& v) const
            {
                holder.stage_result(std::move(v));
            }

            void return_value(T const& v) const
            {
                holder.stage_result(v);
            }

            suspend_never final_suspend() const noexcept
            {
                holder.complete();
                return{};
            }
```

Okay, so we've fixed the problem of resuming the waiting coroutines too soon. There's another improvement we can make, but there's another topic I want to cover first, which I'll do next time.

[1] The actual transformation is more complicated than described here, but the simplified version will suffice for now. One discrepancy that is worth noting here is that an exception that occurs when creating the initial suspend object is *not* caught by the `try` statement; instead, it propagates synchronously out of the coroutine. However, an exception that occurs *during* the `co_await` is indeed caught by the `try` statement and ends up captured into the promise.

[2] If there is a function try around the entire function, the `try` is considered to be part of the function body. See **[dcl.fct.def.general]** for the formal definition of "function body". This transformation of function try is performed even before the transformation described above.

[3] I'm assuming of course that you provided the return object a way to resume the coroutine when it is awaited. We'll explore possible ways of doing this in future installments.

[4] My thanks to Gor Nishanov for pointing out this issue to me.

Raymond Chen

Follow