C++ coroutines: Short-circuiting suspension, part 2

devblogs.microsoft.com/oldnewthing/20191216-00

December 16, 2019

Raymond Chen

There's one last section of the outline of compiler code generation for `co_await` that is marked "We're not ready to talk about this step yet." Let's talk about that step.

Before suspending the coroutine, the compiler asks the awaiter's `await_ready` method. This method returns `true` if the operation is already complete, or `false` if the coroutine should suspend.

If the operation is already complete, then the compiler can avoid having to save the coroutine's state, only to load it back up again immediately.

	calculate `x` obtain `awaiter`

`co_await`	`if (!awaiter.await_ready())` ⇐
	`{`
	save state for resumption
	`if (awaiter.await_suspend(handle))`
	`{`
	return to caller

	[Invoking the handle resumes execution here]
	`}`
	restore state after resumption
	`}` ⇐
	`result = awaiter.await_resume();`

	execution continues

In the case where `await_ready` says, "Yes, I'm ready!", the compiler skips over the code that saves the coroutine state, creates a continuation handle, suspends the coroutine, and asks the `await_ suspend` to arrange for the coroutine's continuation; and then when the continuation occurs, restoring the coroutine state. Instead, it can go straight to the "So what was the result?" This avoids a bunch of register spilling and reloading.

The C++ language comes with a predefined awaiter known as `suspend_never`. Its `await_ready` always returns `true`, which means that it never actually suspends. It always goes straight to the continuation.[1]

We can take advantage of the `await_ready` method `resume_in_any_apartment` function:

```
template<typename Async,
         typename = std::enable_if_t<
             std::is_convertible_v<
                 Async,
                 winrt::Windows::Foundation::IAsyncInfo>>>
[[nodiscard]] auto resume_in_any_apartment(Async async)
{
  struct awaiter
  {
    bool await_ready()
    {
      return async.Status() ==
                Windows::Foundation::AsyncStatus::Completed;
    }

    void await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      async.Completed([handler](auto&&...) { handler(); });
    }

    auto await_resume()
    {
        return async.GetResults();
    }
    Async async;
  };
  return awaiter{ std::move(async) };
};
```

Perhaps a clearer example of this pattern is an awaitable which detects that its work is unnecessary, such as this one which switches to the dispatcher's thread:

```
auto ensure_dispatcher_thread(CoreDispatcher dispatcher)
{
  struct awaiter : std::experimental::suspend_always
  {
    CoreDispatcher dispatcher;

    bool await_ready() { return dispatcher.HasThreadAccess(); }

    void await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      dispatcher.RunAsync(CoreDispatcherPriority::Normal,
                          [handle]{ handle(); });
    }
  };
  return awaiter{ {}, std::move(dispatcher) };
}
```

This awaitable resumes execution on the dispatcher's thread. In the `await_ready`, we check if we are already on the dispatcher's thread. If so, then we report that the `co_await` is complete even before it started, and execution will continue without ever suspending. Otherwise, the coroutine suspends, and we schedule its resumption on the dispatcher's thread.

[1] An awaiter that never suspends sounds really strange. After all, why bother even being a coroutine! But it's handy for cases in which you have to provide an awaiter even though nothing is being awaited. We'll see examples of this when we study the promise object at some unspecified point in the future.

Raymond Chen

Follow