C++ coroutines: Short-circuiting suspension, part 1

devblogs.microsoft.com/oldnewthing/20191213-00

December 13, 2019

Raymond Chen

At the start of this series, I gave the basic idea for how the compiler generates code for `co_await`, but I left out some details for expository simplicity. There are some mysterious steps called "We're not ready to talk about this step yet."

Now it's time to talk about one of those steps.

It may be the case that when you get around to doing the `await_suspend`, the thing your custom awaiter is waiting for has already completed. It could be that the operation completed synchronously, or that it was so fast that it finished even before you could schedule the completion.

You don't want to invoke the handle directly from your `await_suspend`, because that would run the coroutine continuation as a subroutine inside the suspension:

calculate `x` obtain `awaiter`	

save state for resumption
`awaiter.await_suspend(handle);`
`handle()`	

	restore state after resumption `result = awaiter.await_resume();`
	execution continues coroutine finishes
`handle()` returns return to caller	

This can result in quite a significant accumulation of stack frames if there are a lot of consecutive `co_await`s of already-completed operations.

To avoid this problem, you can change your `await_suspend` member to return `bool`. Your implementation should check whether the operation has already completed. If so, then do *not* schedule the handle for execution, but instead just return `false`, to indicate that suspension should be abandoned and that execution should resume immediately. Otherwise, schedule the handle for execution as usual, and return `true`.

Adding to our gradually-improving understanding of the compiler code generation of `co_await`:

	calculate `x` obtain `awaiter`
`co_await`	(We're not ready to talk about this step yet.) 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

Let's add fancy `await_suspend` support to our `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 : std::experimental::suspend_always
  {
    bool await_suspend(
        std::experimental::coroutine_handle<> handle)
    {
      if (async.Status() != Windows::Foundation::AsyncStatus::Started) {
        return false;
      }
      async.Completed([handler](auto&&amp...) { handler(); });
      return true;
    }

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

If at the time of suspension, the asynchronous activity is not in the `Started` state, then that means that it completed (successfully, with an error, or with cancellation). Therefore, there's no point waiting for it to complete. We can report it as already-completed and continue execution directly.[1]

This type of short-circuit is commonly seen when the `await_` `suspend` function tries to schedule the continuation, and the framework says, "Dude, it's already done!" For example, you might be performing an asynchronous read: If the `ReadFile` function returns `TRUE`, then the operation completed synchronously, and you can go straight to the resumption code.

There's one last piece of the compiler code generation that is marked "We're not ready to talk about this step yet." We're just about ready to talk about that step. Next time.

[1] There is a small race window if the asynchronous activity completes just after we check whether it has completed. Therefore, this change does not eliminate the stack accumulation completely, but it greatly reduces its likelihood.

Raymond Chen

Follow