How to wait for multiple C++ coroutines to complete before propagating failure, symmetric transfer

devblogs.microsoft.com/oldnewthing/20230703-00

July 3, 2023

Raymond Chen

Last time, we wrote a simple coroutine promise to help us with our `when_all_completed` function. One obvious refinement we can make is to avoid stack build-up by using symmetric transfer.

Observe that in both of the `await_suspend` flows, we resume another coroutine. In our initial implementation, we accomplished this by calling the `resume()` method on the desired coroutine handle. However, this results in stack build-up: The caller awaits the coroutine by calling `resume()` on the coroutine, and when the coroutine finishes, it returns control to the caller by calling `resume()` on the caller's coroutine handle. So we're two frames deep.

This cycle repeats for each awaitable passed to the `when_all_completed` function, and there could be quite a few of them.

We can use symmetric transfer to avoid the stack build-up, since the last thing each function does is resume some other coroutine.

First, we'll use symmetric transfer when starting the coroutine:

```
struct all_completed_result
{
    all_completed_promise& promise;
    bool await_ready() noexcept { return false; }
    auto await_suspend(
        std::coroutine_handle<> handle) noexcept;
    std::exception_ptr await_resume() noexcept;
};

auto all_completed_result::
    await_suspend(std::coroutine_handle<> handle)
    noexcept
{
    promise.awaiting_coroutine = handle;
    return promise.coroutine();
}
```

When lazy-starting the coroutine, we return the coroutine's handle instead of manually resuming it. This activates symmetric transfer, so the compiler can use a tail call to jump directly to the coroutine, avoiding a stack frame.

Doing the same thing when the coroutine finishes takes a little more work because the symmetric transfer happens in `await_suspend`, but our original version was resuming the caller in `final_suspend`. We'll have to arrange for the caller's handle to be returned from `await_suspend`.

```
struct all_completed_promise
{
    ...

    auto final_suspend() noexcept {
        struct awaiter : std::suspend_always
        {
            std::coroutine_handle<> other;
            auto await_suspend(std::coroutine_handle<>) {
                return other;
            }
        };
        return awaiter{{}, awaiting_coroutine};
    }
};
```

This is the actual symmetric transfer part: We save the coroutine handle we want to resume in the awaiter, so that the awaiter can return it from `await_suspend`. Again, symmetric transfer allows the resumption of the awaiting coroutine to happen as a tail call, avoiding a stack frame.

But we're not done yet.

We suspended the promise's coroutine, so it remains allocated in memory. We need to destroy it after we extract the `eptr` in the `await_resume` that returns it to the caller.

```
std::exception_ptr all_completed_result::
    await_resume() noexcept
{
    auto eptr = promise.eptr;
    promise.coroutine().destroy();
    return eptr;
}
```

Okay, so that reduces the likelihood of stack exhaustion issues when awaiting a whole bunch of awaitables inside `when_all_completed`.

But wait, we haven't addressed the `std::bad_alloc` problem that we identified a while back. We got distracted with all the simplifications that a custom promise offered, but forgot why why we wrote our own custom promise in the first place. Let's return to that next time.