What happens if you co_await a std::future, and why is it a bad idea?

devblogs.microsoft.com/oldnewthing/20230217-00

February 17, 2023

Raymond Chen

The C++ standard library introduced `std::future` in C++11, along with various functions and types that produce futures: `std::async`, `std::packaged_task`, and `std::promise`. The only way to known when the result of a `std::future` is ready is to poll for it, or simply block until the result is ready.

When the Visual C++ compiler implemented experimental coroutine support, it added the ability to `co_await` a `std::future`: If you do that, the coroutine suspends until the `std::future` produces a result, and the result of the `std::future` becomes the result of the `co_await`.

That sounds convenient.

A customer reported that sometimes their program would crash with an out-of-memory error. They sent us some of the crash dumps they received. The crash dumps showed that their program had created around 2000 threads before finally succumbing. And most of the threads were waiting on a condition variable.

```
ntdll!ZwWaitForAlertByThreadId+0x14
ntdll!RtlSleepConditionVariableSRW+0x137
KERNELBASE!SleepConditionVariableSRW+0x33
msvcp_win!Concurrency::details::stl_condition_variable_win7::wait_for+0x15
msvcp_win!Concurrency::details::stl_condition_variable_win7::wait+0x19
msvcp_win!_Cnd_wait+0x2a
contoso!std::condition_variable::wait+0x10
contoso!std::_Associated_state<winrt::hstring>::_Wait+0x3b
contoso!std::_State_manager<winrt::hstring>::wait+0x42
contoso!std::experimental::_Future_awaiter<winrt::hstrint>::await_suspend::__l2::
<lambda_5f42a2a4a1d632a6517852fe05159fc3>::operator()+0x45
contoso!std::invoke+0x45
contoso!std::thread::_Invoke<std::tuple<<lambda_5f42a2a4a1d632a6517852fe05159fc3>
>,0>+0x53
ucrtbase!thread_start<unsigned int (__cdecl*)(void *),1>+0x93
KERNEL32!BaseThreadInitThunk+0x14
ntdll!RtlUserThreadStart+0x28
```

From the function names on the stack, we can pull out that this code is waiting for a `std::future` to become ready. (Lots of the names are strong hints, but the giveaway is `_Future_awaiter`.)

Let's look at how `operator co_await` is implemented for `std::future`:

```
template <class _Ty>
struct _Future_awaiter {
    future<_Ty>& _Fut;

    bool await_ready() const {
        return _Fut._Is_ready();
    }

    void await_suspend(
        experimental::coroutine_handle<> _ResumeCb) {
        // TRANSITION, change to .then if and when future gets .then
        thread _WaitingThread(
            [&_Fut = _Fut, _ResumeCb]() mutable {
            _Fut.wait();
            _ResumeCb();
        });
        _WaitingThread.detach();
    }

    decltype(auto) await_resume() {
        return _Fut.get();
    }
};
```

To `co_await` a `std::future`, the code first checks if the value is already set. If not, then we create a thread and have the thread call `future.wait()`, which is a blocking wait. When the wait is satisfied, the coroutine resumes.

The stack is consistent with our analysis. We are on a dedicated thread running the lambda inside `await_suspend`, and that lambda is waiting for the `std::future` to produce the result.

Each `co_await` of a `std::future` burns a thread. Checking the customer's code showed that there's a `std::future<winrt::hstring>` that represents some calculation. The calculation itself requires asynchronous work, so each time somebody asks for the value to be calculated, a new `std::future` is created to represent the calculation, and the caller then `co_await`s for the result of the calculation.

What happened is that the calculation for some reason is taking a long time, and a lot of requests have piled up. Under normal conditions, stackless coroutines do not consume a thread while they are suspended; they just sign up to be resumed when the thing they are awaiting finally produced a result. But `std::future` has no way to register a way to be

called back when the result is ready. The only way to find out is to wait for it, and that consumes a thread. (That's what the "TRANSITION" comment is trying to say: When it becomes possible to register a callback for the readiness of a `std::future`, we should switch to it.)

The program is using `std::promise` as an implementation of a task completion source, unaware that the implementation is very expensive, burning a thread for each outstanding `co_await`. We advised the customer to switch to something lighter weight, such as the task completion source we developed as part of our study of coroutines.

Or you can build your own quick-and-dirty task completion source that has the limitation that it doesn't support exceptions. (Because I'm lazy.) For this customer's purpose, that may be sufficient.

```
template<typename T>
struct qd_completion_source
{
    void set_result(T value) {
        result = std::move(value);
        SetEvent(event.get());
    }

    auto resume_when_ready() {
        return winrt::resume_on_signal(event.get());
    }

    T& get_result() { return *result; }

private:
    std::optional<T> result;
    winrt::handle event = winrt::check_pointer(CreateEvent(nullptr, TRUE, FALSE,
nullptr));
}

// Produce the qd_completion_source
std::shared_ptr<qd_completion_source<int>>
StartSomething()
{
    auto source = std::make_shared<
            qd_completion_source<int>>();

    [](auto source) -> winrt::fire_and_forget {
        co_await step1();
        co_await step2();
        source.set_result(co_await step3());
    }(source);

    return source;
}

// Consume the qd_completion_source
winrt::fire_and_forget GetSomethingResult()
{
    auto source = StartSomething();

    co_await source->resume_when_ready();

    auto result = source->get_result();
}
```

When the result is ready, our quick-and-dirty completion source saves the answer in the `std::optional` and then signals the event. To resume when the result is ready, we resume when the event is set.

If you want to be awaitable more than once, you can return a copy of the result from `await_resume` rather than moving the result to the caller.

Like I said, this is a quick-and-dirty version. It still uses a kernel object to synchronize between the producer and consumer, but even so, a kernel event is far lighter than an entire thread! I started writing a version that used `coroutine_handle<>` but realized that I already did that.