Awaiting a set of handles with a timeout, part 4: Building our own awaiter

devblogs.microsoft.com/oldnewthing/20240503-00

May 3, 2024

Raymond Chen

Last time, we figured out how to await an arbitrary number of handles with a common timeout. But we found that there were two fundamental problems: The awaiter might not be movable, and we don’t want to throw an exception after some of the handles have been signaled (because that causes us to lose track of them).

Since we don’t control the awaiter used by `resume_on_signal`, we’ll have to switch to something we do control.

We’ll write our own awaiter.

Fortunately, writing an awaiter is easier than writing a coroutine promise. We just need to implement the three awaiter methods: `await_ready`, `await_suspend`, and `await_resume`.

In order to avoid the problem of throwing an exception partway through, we need to make sure we set up everything that could possibly throw an exception before we start waiting on any of the handles.

Here’s our first attempt. Let’s start with the simple case that we are given a counted array of `HANDLE`s. Our function prototype will be this:

```
auto resume_on_all_signaled(HANDLE* handles, uint32_t size,
    std::optional<winrt::Windows::Foundation::TimeSpan> timeout
        = std::nullopt);
```

I changed the `timeout` parameter to an optional `TimeSpan`, where an empty value means that there is no timeout. This avoids problems in the original code where 0 meant “no timeout (wait indefinitely)”, but a value of zero, or even a negative value, could be generated by mistake, say because the deadline has been reached or has already been passed. Making it an explicitly optional parameter avoids this edge case where a computed timeout happens to match the sentinel value. It also means that you will be able to pass a timeout of zero to probe the handles without waiting.

We start with this guy:

```
struct resume_all_state
{
    struct resume_all_awaiter* m_parent;
    HANDLE m_handle;
    bool* m_result;
    wil::unique_threadpool_wait_nowait m_wait;
    };
```

The `resume_all_state` holds the information we need about each handle. It holds a pointer to the awaiter (to be defined below), the handle we are waiting for, where we should record the result of the handle wait, and the threadpool wait that will notify us when the handle is signaled (or the timeout elapses).

```
struct resume_all_awaiter
{
```

To save ourselves some typing, we'll create a type alias.

```
    using TimeSpan = winrt::Windows::Foundation::TimeSpan;
```

And then we can declare our member variables.

```
    std::atomic<uint32_t> m_remaining;
    std::vector<resume_all_state> m_states;
    winrt::com_array<bool> m_results;
    std::coroutine_handle<> m_resume;
    std::optional<TimeSpan> m_timeout;
```

The awaiter keeps track of a few things.

- `m_remaining`: The number of handles for which we are still waiting for a result. This decreases each time a handle becomes signaled or times out, and when it reaches zero, we resume the coroutine.
- `m_states`: A vector of `state`s, one for each handle.
- `m_results`: The `com_array` which holds the results that we return as the result of the `co_await`.
- `m_resume`: The coroutine to resume once we get all the results.
- `m_timeout`: The timeout after which we give up waiting for the handles.

Okay, let's write the constructor.

```
    resume_all_awaiter(HANDLE* handles, uint32_t size,
        std::optional<TimeSpan> timeout) :
        m_remaining(size),
        m_states(size),
        m_results(size),
        m_timeout(timeout)
    {
        for (auto index = 0U; index < size; ++index) {
            auto& s = m_states[index];
            s.m_parent = this;
            s.m_handle = handles[index];
            s.m_result = &m_results[index];
            s.m_wait.reset(winrt::check_pointer(
                CreateThreadpoolWait(callback, &s, nullptr)));
        }
    }
```

We use the `size` to establish the number of `resume_all_state`s we need, the number of handles we are still waiting for (namely, all of them), and the number of `bool`s we need to return. We also save the timeout for later.

Inside the constructor body, we initialize the states with a pointer back to the `awaiter`, the handle to (eventually) wait for, a pointer to where we want to record the wait result, and a threadpool wait that uses the corresponding `resume_all_state` object as the callback data.

It is important that the vector not be reallocated once we pass a pointer to the `resume_all_state` to `CreateThreadpoolWait`, because reallocation will move the `resume_all_state` objects, leaving the pointer dangling and producing a use-after-free bug.

Note that we copy the handles into our `resume_all_state` objects rather than just saving the original pointer and size. That's because the caller might not `co_await` the awaiter immediately, and the pointer we received might have been a temporary.

```
auto awaiter = resume_on_all_signaled(std::array{ h1, h2 }.data(), 2);
co_await awaiter;
```

Yes, this is a weird-sounding corner case, but it'll be important later.

The most important thing right now is that we do all the things that could potentially fail right up front in the constructor. That way, if the `co_await` throws an exception, the caller knows that no handles have been waited on, and the states of the objects in question have not been modified.

```
    bool await_ready() noexcept { return false; }
```

The `await_ready` is easy: We are never ready. We always ask for the coroutine to be suspended. Which is what comes next:

```
    void await_suspend(std::coroutine_handle<> resume) noexcept
    {
        m_resume = resume;

        FILETIME ft;
        FILETIME* timeout = nullptr;
        if (m_timeout) {
            auto count = (std::max)(m_timeout->count(), TimeSpan::rep(0));
            ft = wil::filetime::from_int64(-count);
            timeout = &ft;
        }

        for (auto&& s : m_states) {
            SetThreadpoolWait(s.m_wait.get(), s.m_handle, timeout);
        }
    }
```

We start by saving the coroutine to be resumed when all the handles have either waited successfully or timed out.

Next, we convert the `m_timeout` to a format that `SetThreadpoolWait` expects. There are three cases.

- If the `m_timeout` is empty, then we are waiting with no timeout, and the way to specify that to `SetThreadpoolWait` is to pass `nullptr`.
- If the `m_timeout` is negative, then we clamp it to zero. This accommodates edge cases where the code tries to wait for handles just after the deadline has passed.
- We then pass that timeout (in the form of a `FILETIME`) to `SetThreadpoolWait` as a negative value, since that's the way that `SetThreadpoolWait` represents elapsed time. (Positive values represent absolute time.)

I parenthesized `std::max` to avoid `max` being recognized as a function-like macro. For historical reasons, `windows.h` defines `min` and `max` macros, and we don't want those. You can suppress those macro definitions by saying `NOMINMAX` before including `windows.h`, but it's common in library code to parenthesize `std::min` and `std::max` to avoid the problem entirely.

```
    static void CALLBACK callback(PTP_CALLBACK_INSTANCE,
        void* context, PTP_WAIT, TP_WAIT_RESULT result)
    {
        auto& s = *reinterpret_cast<resume_all_state*>(context);
        *s.m_result = (result == WAIT_OBJECT_0);
        if (s.m_parent->m_remaining.fetch_sub(1,
                 std::memory_order_release) == 1) {
            s.m_parent->m_resume();
        }
    }
```

As each handle wait completes, we recover the `resume_all_state` object for that handle and use it to record whether the wait succeeded. We then atomically decrement the number of remaining handles, and if it reaches zero, we resume the coroutine. Since we used a `unique_threadpool_wait_nowait`, the destructor of the threadpool wait won't wait for callbacks to complete, which in our case is a good thing, because waiting for the callback to complete would lead to a deadlock: The destructor of the awaiter would wait for the callback to complete, but the destructor is running as part of the coroutine resumption, which is happening *in the callback*.[1]

The `--` operator on a `std::atomic` uses sequential consistency semantics, but we need only release semantics (we are publishing a value, namely the wait result), so we use `fetch_sub`, which allows us to specify a memory order. The `fetch_sub` method returns the *previous* value, so we detect that we decremented to zero by seeing if the previous value was 1.

The last thing the awaiter needs to do is return the results when the coroutine resumes.

```
    auto await_resume() noexcept
    {
        return std::move(m_results);
    }
};
```

The `resume_on_all_signaled` function now just needs to return a properly-constructed awaiter.

```
auto resume_on_all_signaled(HANDLE* handles, uint32_t size,
    std::optional<winrt::Windows::Foundation::TimeSpan> timeout
        = std::nullopt)
{
    return resume_all_awaiter(handles, size, timeout);
}
```

Okay, now that we have a basic version, we can start fine-tuning it. Next time.

Bonus chatter: When this problem was first presented to me, I said, "Just create an awaiter that creates one threadpool wait for each handle, and which resumes when all the waits complete or time out." This is just the realization of that basic idea.

[1] This trick of using a `_nowait` threadpool wait handle works only because we never resume the coroutine until after all the waits have completed. If there were cases where the coroutine resumes before all the waits have completed, we would need to use a waiting version of the threadpool wait handle to ensure that the callback doesn't access memory after it has been freed. We could use `DissociateCurrentThreadFromCallback` just before resuming the coroutine to exempt the current callback from the wait.