Awaiting a set of handles with a timeout, part 1: Starting with two

devblogs.microsoft.com/oldnewthing/20240430-00

April 30, 2024

Raymond Chen

Suppose you want a C++ coroutine that waits for a bunch of kernel handles to be signaled, and gives up if a timeout is reached. And you want the coroutine to tell you which handles were signaled and which timed out.

Let's start by using something we already have, namely the C++/WinRT `resume_on_signal` coroutine that awaits a single handle with a timeout. Maybe we can use that.

```
#include <array>
#include <winrt/Windows.Foundation.h>
#include <wil/coroutine.h>

wil::task<std::array<bool, 2>>
    resume_on_both_signaled(HANDLE h1, HANDLE h2,
        winrt::Windows::Foundation::TimeSpan timeout = {})
{
    auto await1 = winrt::resume_on_signal(h1, timeout);
    auto await2 = winrt::resume_on_signal(h2, timeout);
    co_return std::array<bool, 2>
        { co_await await1, co_await await2 };
}
```

The idea is that we call `resume_on_signal` for all of the handles before we start `co_await`ing, because we want the timeout for all of the awaits to begin at the time that `resume_on_both_signaled` is called.

Unfortunately, it doesn't work.

The awaiter returned by `resume_on_signal` doesn't start the timeout timer until you `co_await` it, so what we end up doing is waiting for the first handle to become signaled with a timeout of `timeout`, and then only after that happens do we wait for the second handle to become signaled, also with a timeout of `timeout`, and the second timeout doesn't start until the first one finishes.

So it wasn't any improvement over

```
    co_return std::array<bool, 2>
        { co_await winrt::resume_on_signal(h1, timeout),
          co_await winrt::resume_on_signal(h2, timeout) };
```

Another problem is that the `resume_on_signal` returns an awaiter that expects to be immediately-awaited, so we're taking a chance by saving it into a local variable and awaiting it later.

Some awaiters save references to their parameters to avoid a copy, assuming that the awaiter will be awaited immediately before the parameters are destructed, so we have to make sure that all of the parameters we pass have their lifetime extended past the `co_await`.

Furthermore, some awaiters may not function properly if you move them. For example, the awaiter may register a callback function and pass its own `this` as the callback data. If you move the awaiter, then when the callback runs, it will try to access the awaiter at the location it *used to be*, which is a use-after-free bug.

This is a hazard of awaiters, since they are written with the expectation that they will be passed directly to `co_await`, and the possibility that they will be saved and copied or moved may not have occurred to the authors.

Let's try to repair these problems next time.