Awaiting a set of handles with a timeout, part 7: Just doing it one at a time

devblogs.microsoft.com/oldnewthing/20240508-00

May 8, 2024

Raymond Chen

So far, we've been trying to wait for a set of handles in parallel, so that the same timeout applies to each handle.¹ If you aren't as picky about the timeout, you can just wait sequentially. Here's our first try:

```
template<typename Iter>
wil::task<std::vector:<bool>>
    resume_on_all_signaled(Iter first, Iter last,
        winrt::Windows::Foundation::TimeSpan timeout = {})
{
    std::vector<bool> results;
    auto deadline = winrt::clock::now() + timeout;

    for (; first != last; ++first) {
        auto remaining = deadline - winrt::clock::now();
        results.push_back(
            co_await winrt::resume_on_signal(*first, remaining));
    }
    co_return results;
}
```

The idea here is that we wait for the first handle with the full timeout, and then wait for subsequent handles with whatever amount of time still remains on the original timeout.

(Yes, we are returning to the accursed `vector<bool>`. We don't need to produce references to individual members of the vector, so the only complain is just its ugliness rather than missing functionality.)

Waiting for the handles sequentially means that if the signals are consumable (such as a semaphore or mutex) or revocable (like an event), the handles at the end of the list won't get claimed until the earlier ones have either succeeded or timed out. Maybe this is a problem if it means that your code misses out on grabbing a semaphore because the semaphore got stuck behind some other handle that was slow to signal.

But probably not.

So let's keep going.

The `resume_on_signal` function has a somewhat unfortunate interpretation of the `timeout` parameter: A timeout of zero means that the wait is infinite. This is unexpected. A timeout of zero should behave as the limit of smaller and smaller timeouts. Mathematically, a timeout of zero should mean "Don't wait at all," not "Wait forever."

Another unfortunate behavior of the `timeout` parameter is that a negative timeout causes the function to behave erratically. (It interprets the negative timeout as an absolute time.)

Our function above doesn't handle these weirdo edge cases.

Okay, so the first edge case (zero timeout meaning no timeout) we address by splitting the function in two, one for waiting with no timeout and one for waiting with a timeout. The no-timeout case we saw earlier. We can adapt it easily to the two-iterator version.

```
template<typename Iter>
wil::task<void>
    resume_on_all_signaled(Iter first, Iter last)
{
    for (; first != last; ++first) {
        co_await winrt::resume_on_signal(*first);
    }
}
```

The timeout version will take a little more work to deal with the edge cases.

```
template<typename Iter>
wil::task<std::vector:<bool>>
    resume_on_all_signaled(Iter first, Iter last,
        winrt::Windows::Foundation::TimeSpan timeout)
{
    std::vector<bool> results;
    auto deadline = winrt::clock::now() + timeout;

    for (; first != last; ++first) {
        auto remaining = deadline - winrt::clock::now();
        if (remaining.count() > 0) {
            results.push_back(
                co_await winrt::resume_on_signal(*first, remaining));
        } else {
            results.push_back(
                WaitForSingleObject(*first, 0) == WAIT_OBJECT_0);
        }
    }
    co_return results;
}
```

If there is time remaining, we use `resume_on_signal`. If there is no time remaining, then we wait with no timeout by using `WaitForSingleObject` and save the result.

¹ Primarily for the exercise of seeing how to await things in parallel.