Awaiting a set of handles with a timeout, part 3: Going beyond two

devblogs.microsoft.com/oldnewthing/20240502-00

May 2, 2024

Raymond Chen

Last time, we figured out how to await two handles with a common timeout. Armed with what we learned from that ordeal, we can try to generalize it to an arbitrary number of handles.

Our first attempt might go like this:

```
template<typename Iter>
wil::task<std::vector<bool>>
    resume_on_all_signaled(Iter first, iter last,
        winrt::Windows::Foundation::TimeSpan timeout = {})
{
    using BoolOp = winrt::Windows::Foundation::IAsyncOperation<bool>;

    std::vector<BoolOp> ops;
    std::transform(first, last, std::back_inserter(ops),
        [&](HANDLE handle) {
        return resume_on_one_signal(handle, timeout);
    });

    std::vector<bool> results;
    for (auto&& op : ops) {
        results.push_back(co_await winrt::resume_agile(op));
    }

    co_return std::move(results);
}
```

The first step is to take the incoming handles and convert them to async operations, which we gather in a vector.

Next, we await each of the awaiters and push the result onto a results vector.

Finally, we return the results.

Now, one of the really annoying things happening here is that we are using `vector<bool>`. As I noted some time ago, this specialization represents a packed bit array. This has made a lot of people very angry and has been widely regarded as a bad move.

So let's use a C-style array of `bool` instead. C++/WinRT comes with one already, known as `winrt::com_array`. We'll use that.

```
template<typename Iter>
wil::task<winrt::com_array<bool>>
    resume_on_all_signaled(Iter first, iter last,
        winrt::Windows::Foundation::TimeSpan timeout = {})
{
    using BoolOp = winrt::Windows::Foundation::IAsyncOperation<bool>;

    std::vector<BoolOp> ops;

    std::transform(first, last, std::back_inserter(ops),
        [&](HANDLE handle) {
        return resume_on_one_signal(handle, timeout);
    });

    if (ops.size() > ~0U / sizeof(bool)) {
      throw std::bad_alloc();
    }
    auto size = static_cast<uint32_t>(ops.size());
    auto results = winrt::com_array{ size };
    for (auto index = 0U; index < size; ++index) {
        results[index] =
               co_await winrt::resume_agile(awaiters[index]);
    }

    co_return std::move(results);
}
```

There is a serious issue: We behave erratically if an exception is thrown from `co_await`.

An exception is possible because the awaiter returned by `resume_on_signal` will throw an exception if it cannot set up the threadpool wait. In that case, the state of the handles is indeterminate: Some of them may have been successfully waited, and others may not have been, and you don't know which were and weren't waited because the exception prevented you from receiving the results.

Knowing which ones were awaited is important if any of the things you are waiting for are consumable, such as semaphores or auto-reset events.

We'll continue our investigation next time.