Awaiting a set of handles with a timeout, part 5: Generalizing the awaiter

devblogs.microsoft.com/oldnewthing/20240506-00

May 6, 2024

Raymond Chen

Last time, we created an awaiter that could await an array of handles. But it's often the case that you have a bunch of handles in some other form, such as a pair of iterators, which is a common currency in the C++ standard library.

First, let's generalize our version to take a pair of iterators.

```
struct resume_all_awaiter
{
    ⟦ data members unchanged ⟧

    template<typename Iter>
    awaiter(Iter first, Iter last,
        std::optional<TimeSpan> timeout) :
        // m_remaining(size),
        // m_states(size),
        // m_results(size),
        m_timeout(timeout)
    {
        std::transform(first, last, std::back_inserter(m_states),
            [](HANDLE h) { state s; s.m_handle = h; return s; });
        create_waits();
    }

    void create_waits()
    {
        if (m_states.size() > ~0U / sizeof(bool)) {
            throw std::bad_alloc();
        }

        auto size = static_cast<uint32_t>(m_states.size());
        m_remaining.store(size, std::memory_order_relaxed);
        auto results = winrt::com_array{ size };
        for (auto index = 0U; index < size; ++index) {
            auto& s = m_states[index];
            s.m_parent = this;
            s.m_result = &m_results[index];
            s.m_wait.reset(winrt::check_pointer(
                CreateThreadpoolWait(callback, &s, nullptr)));
        }
    }

    ⟦ other methods unchanged ⟧
};

template<typename Iter>
auto resume_on_all_signaled(Iter first, Iter last,
    std::optional<winrt::Windows::Foundation::TimeSpan> timeout
        = std::nullopt)
{
    return resume_all_awaiter(first, last, timeout);
}
```

First, we record all of the handles that we were given, in the same number of `resume_all_state` objects. We can't call `CreateThreadpoolWait` yet, because we want to use a pointer to the `resume_all_state` as the context pointer, which means we have to wait until all of the

elements have been pushed onto the vector before we start taking their addresses. Otherwise, they will move when the vector expands its capacity.

We build up the vector by pushing handles one at a time. We can't preallocate the vector because the iterator may support only forward iteration, so the only way to find out how many handles there are is to increment the iterator all the way to the end. And it may be only an input iterator, which means you can walk through the collection only once.

That value becomes `m_remaining`, the number of handles we are still waiting for. We also use that value to size the C-style array of `bool`s that will be used to hold the results.

After vector has been filled with handles, we can initialize the rest of the state and create the wait objects, confident that the `resume_all_state` objects won't move any more.

To reduce code size, the second half of the calculations are factored into a helper method, which can be shared among different specializations of the constructor.

We can add a helper method to simplify the case where the handles are already being held in an iterable container:

```
template<typename Container = std::initializer_list<HANDLE>>
auto resume_on_all_signaled(Container const& c,
    std::optional<winrt::Windows::Foundation::TimeSpan> timeout
        = std::nullopt)
{
    return resume_on_all_signaled(
        std::begin(c), std::end(c), timeout);
}
```

Since we are binding to a `const&`, the container parameter might be a temporary. This was the important case I mentioned last time.

We give a default type of `std::initializer_list<HANDLE>` so that you can write this:

```
co_await resume_on_all_signaled({ handle1, handle2 });
```

Next time, we'll make this code more efficient in the case that the iterators are random-access, or at least random-access-like.