Awaiting a set of handles with a timeout, part 6: Capturing the handles efficiently

devblogs.microsoft.com/oldnewthing/20240507-00

May 7, 2024

Raymond Chen

Last time, we created an awaiter that could await a `[first, last)` range of handles. It did so by pushing onto a vector for each such handle, but this is inefficient in the case where the number of handles is known in advance. Let's add a constructor for the case where the iterators support the subtraction operator.[1]

```
struct awaiter
{
    ⟦ data members unchanged ⟧

    template<typename Iter,
        typename = std::void_t<
            decltype(std::declval<Iter>() -
                     std::declval<Iter>())>>
    awaiter(Iter first, Iter last,
        std::optional<TimeSpan> timeout,
        int) :
        m_timeout(timeout)
    {
        auto count = last - first;
        m_states.resize(count);
        for (auto& s : m_states) {
            m_states.m_handle = *first;
            ++first;
        }
        create_waits();
    }

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

    ⟦ 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, 0);
}
```

We add another constructor that is enabled via SFINAE if the iterator type supports subtraction. If so, then we use that subtraction operator to get the number of handles, then resize the vector directly to that size, so that we can fill in the handles without having to do any reallocating. This significantly reduces code size because the compiler doesn't have to generate any resize logic.

Note that if the iterator supports subtraction, then both constructors become available, so we use an unused parameter to steer the compiler toward the `int` version if it is available, since `int` is considered a better match for `0` than `unsigned`.

[1] Why do we look for a subtraction operator, rather than checking the iterator category for `random_access_iterator_tag`? Because not all subtractable iterators satisfy the requirements of a random access iterator. In particular, dereferencing a random access iterator must produce a reference, which rules out things like `IVectorView` since the `GetAt` method returns a copy, not a reference. The C++ iterator library doesn't have a built-in way to detect a random-access output iterator, so we have to make up our own.