Serializing asynchronous operations in C++/WinRT

devblogs.microsoft.com/oldnewthing/20220915-00

September 15, 2022

Raymond Chen

Armed with some insight into C++ coroutines and lazy-start coroutines, we return to the issue of making asynchronous operations run one after the other instead of concurrently.

Adapting the solution from C# comes with a frustrating complication: Unlike C# `Task`s, C++/WinRT `IAsyncAction` and `IAsyncOperation` support only one completion callback, so we cannot `co_await` them more than once. That's too bad, because the C# version relied on having two awaiters: One is the ultimate caller of the asynchronous method, and the other is the internal awaiter that we use for sequencing.

Since the caller is going to perform a `co_await`, our internal awaiter will have to use some other mechanism for sequencing. We'll do that by hooking up the continuations manually into our own data structures.

```
struct task_sequencer
{
    task_sequencer() = default;
    task_sequencer(const task_sequencer&) = delete;
    void operator=(const task_sequencer&) = delete;

private:
    using coro_handle = std::experimental::coroutine_handle<>;

    struct suspender
    {
        bool await_ready() const noexcept { return false; }
        void await_suspend(coro_handle h)
            noexcept { handle = h; }
        void await_resume() const noexcept { }

        coro_handle handle;
    };

    static void* completed()
    { return reinterpret_cast<void*>(1); }

    struct chained_task
    {
        chained_task(void* state = nullptr) : next(state) {}

        void continue_with(coro_handle h) {
            if (next.exchange(h.address(),
                        std::memory_order_acquire) != nullptr) {
                h();
            }
        }

        void complete() {
            auto resume = next.exchange(completed());
            if (resume) {
                coro_handle::from_address(resume).resume();
            }
        }

        std::atomic<void*> next;
    };

    struct completer
    {
        ~completer()
        {
            chain->complete();
        }
        std::shared_ptr<chained_task> chain;
    };
```

```
    winrt::slim_mutex m_mutex;
    std::shared_ptr<chained_task> m_latest =
        std::make_shared<chained_task>(completed());

public:
    template<typename Maker>
    auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
    {
        auto current = std::make_shared<chained_task>();
        auto previous = [&]
        {
            winrt::slim_lock_guard guard(m_mutex);
            return std::exchange(m_latest, current);
        }();

        suspender suspend;

        using Async = decltype(maker());
        auto task = [](auto&& current, auto&& makerParam,
                       auto&& contextParam, auto& suspend)
                    -> Async
        {
            completer completer{ std::move(current) };
            auto maker = std::move(makerParam);
            auto context = std::move(contextParam);

            co_await suspend;
            co_await context;
            co_return co_await maker();
        }(current, std::forward<Maker>(maker),
          winrt::apartment_context(), suspend);

        previous->continue_with(suspend.handle);

        return task;
    }
};
```

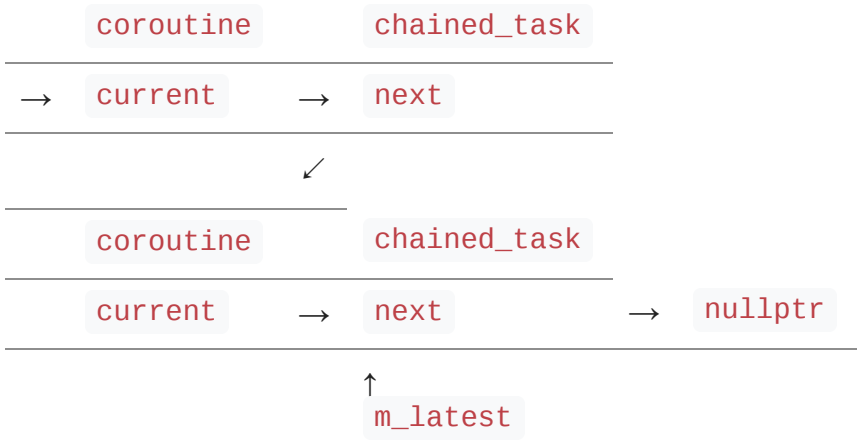
There's a lot going on here, so let's take it bit by bit.

```
struct task_sequencer
{
    task_sequencer() = default;
    task_sequencer(const task_sequencer&) = delete;
    void operator=(const task_sequencer&) = delete;
```

Our `task_sequencer` is default-constructible but is not copyable or assignable.

Inside the class, we start with the `suspender`, which we saw last time. We use this to force the lambda coroutine (coming later) to suspend and capture the coroutine handle that lets us resume it.

Next, we have the `chained_task`. This class connects the coroutines that want to run in sequence.



Each coroutine has a local variable called `current` which is a shared pointer to a `chained_task`. That `chained_task` has a member called `next` which points to the coroutine to run after the current coroutine has completed. The chain ends with a `chained_task` (also known as `m_latest`) whose `next` is null.

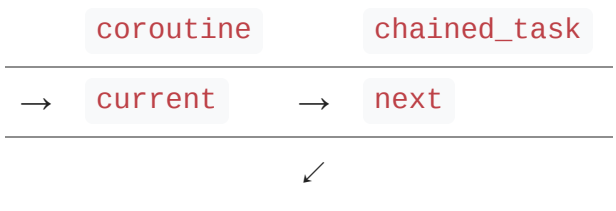
Each `chained_task` remembers the coroutine that needs to run next in its `next` member. The most recently-queued one is remembered in `m_latest`, and its `next` is `nullptr` if the coroutine is still running, or is `completed` if the coroutine has completed (and is waiting for somebody to run next).

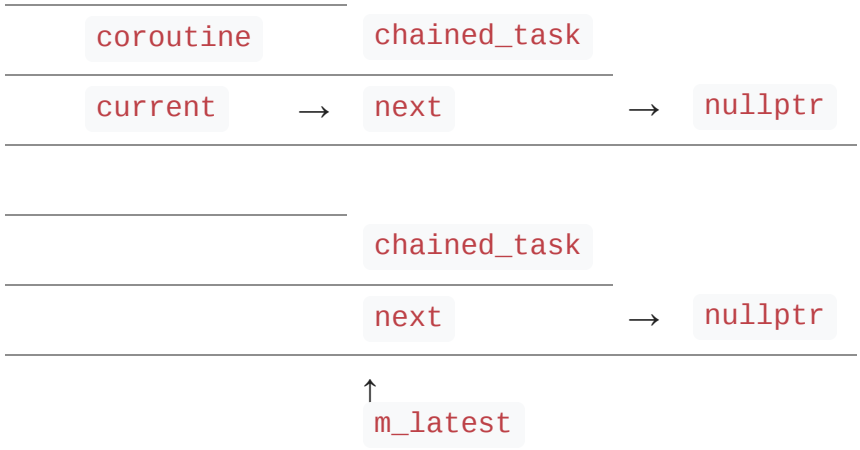
We set our initial condition by initializing `m_latest` to a `chained_task` that has already completed. That way, the next coroutine to be queued will run immediately.

First, we create a new `chained_task` node and make it the `m_latest`, while saving the previous value of `m_latest` in `previous`.

```
auto current = std::make_shared<chained_task>();
auto previous = [&] {
    winrt::slim_lock_guard guard(m_mutex);
    return std::exchange(m_latest, current);
}();
```

In pictures, we've created this:





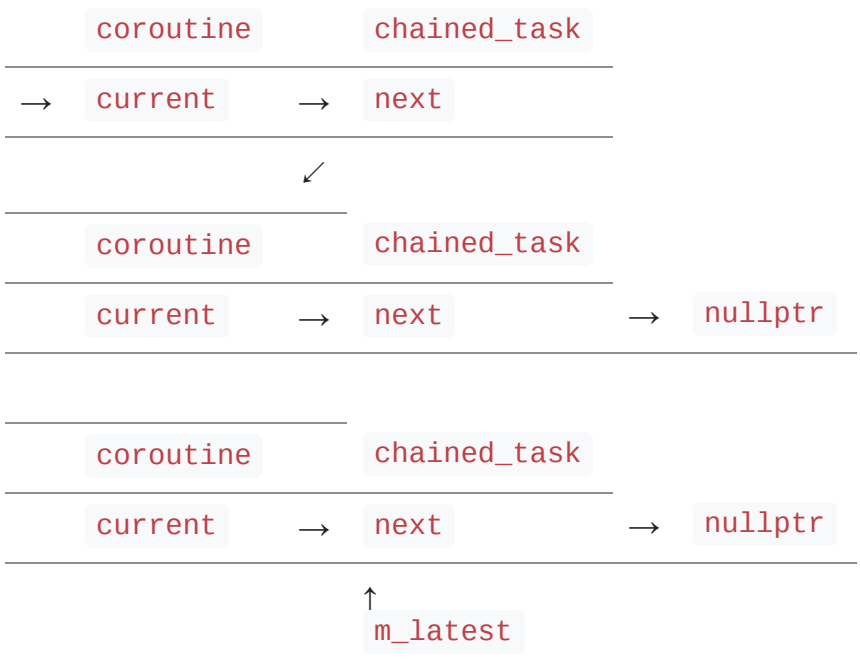
Once we've set up the new node, we capture it into a coroutine which represents the queued task, and then use the `suspender` in order to suspend the coroutine immediately and obtain its coroutine handle.

```
suspender suspend;

using Async = decltype(maker());
auto task = [](auto&& current, ..., auto& suspend)
            -> Async
{
    chained_task_completer completer{ std::move(current) };
    ...

    co_await suspend;
    ...
}(current, ..., suspend);
```

This fills in another par of the diagram:

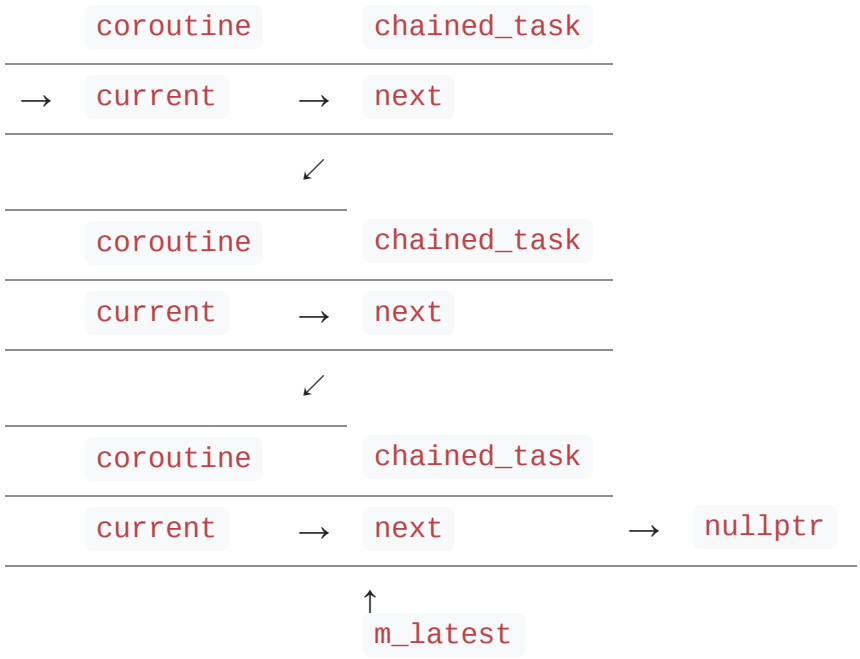


And then we take that coroutine handle and hook it up to the previous `chained_task`:

```
previous->continue_with(suspend.handle);
```

That last step links everything together again:

The linked list now looks like this:



As a special case, if the previously-final `chained_task` has already completed, then instead of hooking up the arrow from that `chained_task` to the latest coroutine, we just resume the latest coroutine immediately.

That's how nodes get added to the list. But how are they removed?

The nodes disappear from the front of the list when coroutines complete. When a coroutine completes, its `completer` destructs, which destructs the shared pointer to the `chained_task`. If the `chained_task` is not the `m_latest`, then this destroys the last shared pointer to the `chained_task`, so it too destructs. As coroutines complete, the head of the linked list gets gobbled up until only `m_latest` remains.

Now let's look inside the coroutine we created.

```
using Async = decltype(maker());
auto task = [](auto&& current, auto&& makerParam,
               auto&& contextParam, auto& suspend)
            -> Async
{
    completer completer{ std::move(current) };
    auto maker = std::move(makerParam);
    auto context = std::move(contextParam);

    co_await suspend;
    co_await context;
    co_return co_await maker();
}(current, std::forward<Maker>(maker),
  winrt::apartment_context(), suspend);
```

Again, there are a few things going on here.

This is a captureless lambda because coroutine lambdas with captures are scary. The captures are instead passed as explicit parameters so they go into the coroutine frame.

The first thing we do is move the objects out of the parameters into locals, so that they destruct as soon as the coroutine completes. We saw some time ago that coroutine parameters do not destruct until the coroutine is destroyed, but we want to resume the next coroutine in the chain as soon as the previous one completes.

We create the `completer` as the first local variable so that it destructs last. That way, we resume the next coroutine only after the current one has destructed everything it had captured. We use an object with a destructor to ensure that chaining to the next coroutine occurs even if the current coroutine exits with an exception.

The `maker` is moved into a local variable so that it (and all of its own captures) destructs as soon as the coroutine completes, rather than lingering until the coroutine is destroyed.

We also move the `apartment_context` into a local variable so that we can switch back to that context once we are resumed. The previous coroutine may have completed on a different COM context, and we need to start the next one in the original context.

When the coroutine completes (either normally or via an exception), the `completer` destructor resume the next coroutine in the chain, if one exists. If not, it just marks itself as complete so that when the next coroutine shows up, it knows it should run immediately.

The atomic operation for publishing the coroutine handle to `next` uses release semantics so that all of the coroutine state generated by the current thread are made visible before we publish the coroutine handle. Conversely, the exchange operation that obtains the coroutine handle uses acquire semantics to ensure that the processor uses the published values instead of locally-cached ones.

Note that if your object is single-threaded, and tasks can be queued only from a single thread, then you don't need the `m_mutex`, which also simplifies the updating of `m_latest`:

```
auto current = std::make_shared<chained_task>();
auto previous = std::exchange(m_latest, current);
```

But wait, we're not done yet. There are some exceptional conditions we'll look at next time.

Raymond Chen

Follow