Serializing asynchronous operations in C++/WinRT, gotchas and final assembly

September 16, 2022

Raymond Chen

Last time, we came up with a way of making asynchronous operations run in sequence, but I noted that there were some gotchas we need to work through.

One gotcha is cancellation.

In the C++/WinRT implementation of Windows Runtime asynchronous operations, cancellation of an asynchronous operation trigger the completion callback immediately, without waiting for the coroutine to acknowledge the cancellation. If we had used the `Completed` callback to detect the completion of the coroutine, we would have started running the next coroutine before the previous one cleaned up from its cancellation.

Good thing we aren't doing that. We trigger the next coroutine at the destruction of the `chained_task_completer`, which we arranged to destruct last, so everything else in the coroutine has already destructed. (Well, the parameters haven't been destructed yet, but they were all references, so there was nothing to destruct.)
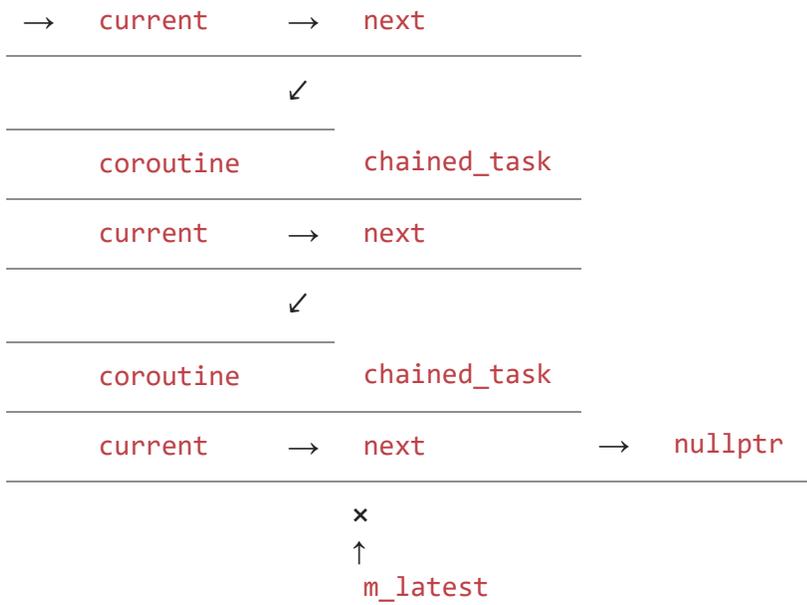
The other thing to worry about is coroutine destruction. That's where you take a suspended coroutine and call `destroy` on it. This basically rips the rug out of the coroutine and destructs everything in it without letting any of the coroutine body run anything. If that happens to our coroutine while suspended at the `co_await lazy_start`, then the `chained_task_completer` will destruct and start running the next coroutine prematurely.

Fortunately, C++/WinRT coroutines are never abandoned. They always run to completion (possibly by throwing an exception). So at least for C++/WinRT coroutines, we don't have to worry about this.[1]

What if the `task_sequencer` is destructed while there are still pending coroutines?

When the `task_sequencer` destructs, the `m_latest` shared pointer destructs, which means that the last node in the chain now has only one strong reference, namely the strong reference from the last coroutine in the chain:

```
coroutine
chained_task


  →    current      →    next
  _____

                 ↙
  _____
        coroutine          chained_task
  _____

        current      →    next
  _____

                 ↙
  _____
        coroutine          chained_task
  _____

        current      →    next          →    nullptr
  _____

                 ×
                 ↑
                 m_latest
```

As the coroutines complete, the nodes come off the head of the linked list, and when the last one completes, the last `chained_task` destructs. Everything is cleaned up.

Now we can take our `task_sequencer` for a spin.

```cpp
task_sequencer sequence;

winrt::Windows::Foundation::IAsyncOperation<winrt::hstring>
MessageAfterDelayAsync(
    winrt::hstring message,
    winrt::Windows::Foundation::TimeSpan delay)
{
    co_await winrt::resume_after(delay);
    std::wcout << message.c_str() << std::endl;
    co_return message + L" done";
}

auto AddMessageAfterDelayAsync(
    winrt::hstring message,
    winrt::Windows::Foundation::TimeSpan delay)
{
    return sequence.QueueTaskAsync(
        [=] { return MessageAfterDelayAsync(message, delay); });
}

void do_in_sequence()
{
    auto first = AddMessageAfterDelayAsync(L"first", 1s);
    auto second = AddMessageAfterDelayAsync(L"second", 1s);
    auto third = AddMessageAfterDelayAsync(L"third", 1s);

    // Cancel the second one, just for fun.
    second.Cancel();

    // wait for the third one.
    auto third_message = third.get();

    // print the results
    std::wcout << first.get().c_str() << std::endl;
    std::wcout << third_message.c_str() << std::endl;
}
```

[1] In general, abandonment of a coroutine via premature `destroy()` is not something most coroutine libraries deal with, so at least we're in good company.

Raymond Chen

Follow