Adding delays to our task sequencer, part 2

devblogs.microsoft.com/oldnewthing/20250403-00/?p=111043

April 3, 2025

Last time, we added task throttling to our task sequencer by adding a 1-second wait between tasks. But maybe you want the throttling to control how frequently tasks can be issued, rather than enforcing a cooling-off period between tasks. In that case, you can calculate how much you need to wait.

For illustration purposes, say that we want to limit you to one task per second. The idea is to remember the earliest time we can start the next task, and if the previous task ends too soon, insert a delay before starting the next one.

```
struct task_sequencer
{
    〚 ... 〛

    struct completer
    {
        ~completer()
        {
            [](auto chain, auto delay) -> winrt::fire_and_forget {
                co_await winrt::resume_after(delay);
                chain->complete();
            }(std::move(chain),
              std::chrono::duration_cast<
                winrt::Windows::Foundation::TimeSpan>
              (earliest - std::chrono::steady_clock::now()));
        }
        std::shared_ptr<chained_task> chain;
        std::chrono::steady_clock::time_point earliest =
            std::chrono::steady_clock::now();
    };


public:
    template<typename Maker>
    auto QueueTaskAsync(Maker&& maker) ->decltype(maker())
    {
        auto node = std::make_shared<chained_task>();

        suspender suspend;

        using Async = decltype(maker());
        auto task = [&]() -> Async
        {
            completer completer{ current };
            auto local_maker = std::forward<Maker>(maker);
            auto context = winrt::apartment_context();

            co_await suspend;
            co_await context;
            completer.earliest =
                std::chrono::steady_clock::now() + 1s;
            co_return co_await local_maker();
        }();

        {
            winrt::slim_lock_guard guard(m_mutex);
            m_latest.swap(node);
        }

        node->continue_with(suspend.handle);

        return task;
    }

    〚 ... 〛
};
```

The idea here is that the completer remembers the earliest the next task can start. Initially, the next task can start right away (in case something goes wrong before we even get around to starting the task), but once we commit to starting the task, we update the earliest time for the next task to the current time plus the desired minimum time between task starts (one second).

When the completer destructs, we calculate how long we have to wait until the steady clock reaches `earliest`, and then ask `resume_after` to wait that amount. The `resume_after` function already handles delays that are zero or negative (by not delaying at all), so we don't need to handle that special case ourselves.

I use the steady clock instead of the wall clock to protect against clock changes caused by time synchronization or the user just going to the Time and Date control panel and manually changing the time. The steady clock always moves forward at an even rate, unaffected by any time adjustments the system or the user may impose.

There is an implicit conversion between durations if the destination duration type has at least as much resolution as the source.[1] If not, then you must use `duration_cast` to indicate that you're okay with rounding.[2]

The standard does not specify the resolution of the steady clock, so we are forced to perform a `duration_cast` to cover the case where the `TimeSpan` resolution is not at least as good as the steady clock resolution.

But wait, we're not done yet. We'll look into the situation a bit more next time.

[1] Formally, either the period of the source is a positive integer multiple of the period of the destination (so that the conversion is an integer multiplication) or destination uses a floating point type.

[2] Though if the destination uses a floating point type, you're getting rounding anyway.