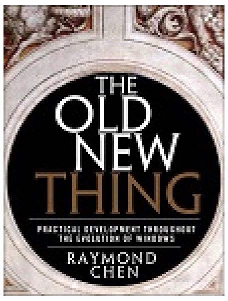
Adding delays to our task sequencer, part 3

devblogs.microsoft.com/oldnewthing/20250404-00/?p=111046

April 4, 2025



Last time, we added task throttling to our `task_sequencer` by limiting requests to one per second. To do this, we used the `steady_clock`. The standard leaves unspecified the resolution of the steady clock, but we can see how it is implemented on Windows by the three major libraries.

On Windows, stl (msvc) and libcxx (clang), and libstdc++ (gcc)/mingw use `Query-PerformanceCounter` as the source of the `steady_clock`,[1] reporting the time to nanosecond resolution. This is excellent resolution, but at a cost of significant arithmetic gymnastics.

We can go cheaper.

Our throttling delay of 1 second doesn't have to be precise down to the nanosecond. In reality, it'll only be as good as the hardware timer tick resolution, which is nowhere near 1 nanosecond. In practice, it's closer to 10 milliseconds per tick, so an error of one or two milliseconds is well under measurement error.

Therefore, we can choose to use `GetTickCount64` as our steady clock source. The `Get-TickCount64` function is quite fast (just reading a 64-bit value from memory), and it reports at millisecond resolution, which means that it converts to `TimeSpan` with multiplication and not division.

```
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::milliseconds(static_cast<int64_t>
                (earliest - GetTickCount64())));
        }
        std::shared_ptr<chained_task> chain;
        ULONGLONG earliest = GetTickCount64();
    };


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 = GetTickCount64() + 1000;
            co_return co_await local_maker();
        }();

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

        node->continue_with(suspend.handle);

        return task;
    }

    ⟦ ... ⟧
};
```

If we really wanted to minimize the number of calls to `GetTickCount64()`, we could use a `std::optional`:

```
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::milliseconds(static_cast<int64_t>
                (earliest ? *earliest - GetTickCount64() : 0)));
        }
        std::shared_ptr<chained_task> chain;
        std::optional<ULONGLONG> earliest;
    };

    ⟦ ... ⟧
};
```

Or we could reserve the sentinel value of 0 to mean “no delay”.

```
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::milliseconds(static_cast<int64_t>
                (earliest ? earliest - GetTickCount64() : 0)));
        }
        std::shared_ptr<chained_task> chain;
        ULONGLONG earliest = 0;
    };


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 = (GetTickCount64() + 1000) | 1;
            co_return co_await local_maker();
        }();

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

        node->continue_with(suspend.handle);

        return task;
    }

    ⟦ ... ⟧
};
```

We force the bottom bit of `earliest` to 1 when recording the start time, so that the value is never zero. This introduces a potential error of 1 millisecond, but one millisecond error out of 1 second is not going to be noticeable in practice for this type of work.

Bonus chatter: You may have figured out that the point of this exercise, aside from actually adding a feature to the task scheduler, is just showing the process of studying the implementation of a chunk of code, getting an idea, following through the implementation, and then refining the implementation.