There is a std::chrono::high_resolution_clock, but no low_resolution_clock

devblogs.microsoft.com/oldnewthing/20250714-00/?p=111375

July 14, 2025

The C++ standard provides a `std::chrono::high_resolution_clock` which provides the implementation's clock with the smallest tick period. This gives you the best resolution available to the implementation.

But what if you don't need the best resolution available to the implementation? For example, our task sequencer wants to wait one second, but accuracy to the nearest 100 nanoseconds is not required. If it's one second or 0.999 seconds or 1.1 seconds, they're all good enough.

In that case, I manually switched to the `GetTickCount64` timer, which has only millisecond resolution, and in practice is good only to the nearest timer tick, which will be more like 10 milliseconds. We are using the value to decide how long to sleep, and sleeps are already fairly sloppy, so trying to be accurate to the nearest nanosecond is pointless.

What we need is a `cheap_steady_clock` that ticks steadily and doesn't need the shortest tick period. Low cost is more important than precision.

For Windows, this means using the `GetTickCount64` timer. Its millisecond resolution is plenty good enough for deciding how long to sleep, and the value is extremely cheap to obtain.

```
struct cheap_steady_clock
{
    using rep = int64_t;
    using period = std::milli;
    using duration = std::chrono::duration<rep, period>;
    using time_point = std::chrono::time_point<cheap_steady_clock>;

    static constexpr bool is_steady = true;

    static time_point now() noexcept
    {
        return time_point{ duration{
            static_cast<int64_t>(GetTickCount64()) } };
    }
};
```

I use a representation of `int64_t` so that negative durations are representable.

For linux systems, you probably would use `CLOCK_MONOTONIC_COARSE`, which is documented as a "faster but less precise version of `CLOCK_MONOTONIC`." In practice, it is 1ms.

```
struct cheap_steady_clock
{
    using duration = std::chrono::nanoseconds;
    using rep = duration::rep;
    using period = duration::period;
    using time_point = std::chrono::time_point<cheap_steady_clock>;

    static constexpr bool is_steady = true;

    static time_point now() noexcept
    {
        struct timespec tp;
        if (0 != clock_gettime(CLOCK_MONOTONIC_COARSE, &tp))
            throw system_error(errno, "clock_gettime(CLOCK_MONOTONIC_COARSE)
failed");
        return time_point(std::chrono::seconds(tp.tv_sec) +
                          std::chrono::nanoseconds(tp.tv_nsec));
    }
};
```

Note that the `now()` method is `noexcept` despite throwing an exception if it cannot read the time. The `noexcept` is required by the standard, so if we can't read the time, we have no choice but to terminate the program.