Inside STL: Waiting for a std::atomic<:shared_ptr class="non-delete">> to change, part 1

devblogs.microsoft.com/oldnewthing/20250108-00

January 8, 2025

Like other `std::atomic` specializations, `std::atomic<std::shared_ptr<T>>` supports the `wait` and `notify_*` methods for waiting for the value to change and reporting that the value has changed. The definition of "changed" in the C++ language specification is that the value has changed if *either* the stored pointer *or* the control block pointer has changed. A shared pointer is implemented as a *pair* of pointers, but `WaitOnAddress` can wait on at most 8 bytes, and unix futexes can wait on only four bytes, so how does this work?[1]

The Microsoft implementation waits for the stored pointer to change, and the `notify_*` methods signal the stored pointer. But wait, this fails to detect the case where the stored pointer stays the same and only the control block changes.

```
std::atomic<std::shared_ptr<int>> p =
    std::make_shared<int>(42);

void change_control_block()
{
    auto old = p.load();
    auto empty = std::shared_ptr<int>();

    // Replace with an indulgent shared pointer
    // with the same stored pointer.
    p.store({ empty, old.get() });
    p.notify_all();
}

void wait_for_change()
{
    auto old = p.load();
    p.wait(old);
}
```

We updated `p` with a `shared_ptr` that has the same stored pointer but a different control block. If the stored pointer is the same, how does the `p.wait()` wake up? The implementation of `p.wait()` waits for the stored pointer to change, but we didn't change it.

The answer is that msvc doesn't wait indefinitely for the pointer to change. It waits with a timeout, and after the timeout, it checks whether either the stored pointer and control block have changed. If so, then the `wait()` method returns. Otherwise, msvc waits a little more. The wait starts at 16ms and increases exponentially until it caps at around 17 minutes.

So changes that alter only the control block pointer will still notify, thought they might be a little sluggish about it. In practice, there is very little sluggishness because `WakeByAddressSingle` and `WakeByAddressAll` will wake a `WaitOnAddress` that has entered the wait state, so the chances for a sluggish wake are rather slim.

```
lock p
decide to wait
unlock p
-------------------------------
prepare for wait ← danger zone
-------------------------------
add to wait list
prepare to block (or early-exit)
block thread
```

It's only in that danger zone, when the `notify_*` tries to wake up a thread that hasn't yet gone to sleep, where you get a sluggish wake.

And you may recall from a peek behind the curtain of `WaitOnAddress` that the system tries hard to close the gap of the "prepare to wait", so in practice, the window of sluggishness is quite small.

Next time, we'll look at the libstdc++ implementation of `wait` and `notify_*`. The wild ride continues.

[1] The proposal to add `wait` and `notify_*` to `std::atomic<std::shared_ptr<T>>` merely says that their omission was "due to oversight." There was no discussion of whether the proposed `wait` and `notify_*` methods are actually implementable, seeing as shared pointers are twice the size of normal pointers and may exceed implementation limits for atomic operations. That is merely left as an exercise for the implementation. An exercise that msvc got wrong the first time.