Inside STL: Waiting for a std::atomic<:shared_ptr class="non-delete">> to change, part 2

devblogs.microsoft.com/oldnewthing/20250109-00

January 9, 2025

Last time, we looked at how the Microsoft C++ standard library implements `wait` and `notify_*` for `std::atomic<std::shared_ptr<T>>`. Today, we'll look at the other library that (as of this writing) implements `std::atomic<std::shared_ptr<T>>`: libstdc++.

The first thing to note is that the traditional "wait for a value to change" mechanism on unix is the futex, but futexes (futexen?) are limited to 4-byte values, which is insufficient for a 64-bit pointer, much less the *two* pointers inside a `shared_ptr`.

At this point, I will refer you to learn about how libstdc++ implements waits on atomic values, particularly the section on how it handles types that do not fit in a `__platform_wait_t`. The remainder of this discussion will treat that as an already-solved problem and focus on the shared pointer part.

Okay, back to `atomic<shared_ptr<T>>::wait()`:

```
// atomic<shared_ptr<T>>::wait
void
wait(value_type __old,
     memory_order __o = memory_order_seq_cst) const noexcept
{
    _M_impl.wait(std::move(__old), __o);
}
```

When you wait on a `shared_ptr`, the work is done by `_Sp_atomic::wait`:

```
// _Sp_atomic<shared_ptr<T>>::wait
void
wait(value_type __old, memory_order __o) const noexcept
{
    auto __pi = _M_refcount.lock(memory_order_acquire);
    if (_M_ptr == __old._M_ptr && __pi == __old._M_refcount._M_pi)
      _M_refcount._M_wait_unlock(__o);
    else
      _M_refcount.unlock(memory_order_relaxed);
}
```

The code locks the `shared_ptr` (by setting the bottom bit of the control block pointer, as we discussed earlier), then checks whether the stored pointer and control block pointer both match. If not, then the wait is satisfied, and we release the lock and return. Otherwise, we ask `_Atomic_count::_M_wait_unlock` to finish the wait.

```
// _Atomic_count::_M_wait_unlock
void
_M_wait_unlock(memory_order __o) const noexcept
{
    auto __v = _M_val.fetch_sub(1, memory_order_relaxed);
    _M_val.wait(__v & ~_S_lock_bit, __o);
}

mutable __atomic_base<uintptr_t> _M_val{0};
```

As the name suggests, `_M_wait_unlock` clears the lock bit (thereby unlocking the shared pointer) and then waits for value to change from its current value.

Meanwhile, the `notify_*` methods do something similar:

```
// atomic<shared_ptr<T>>::notify_*
void
notify_one() noexcept
{
    _M_impl.notify_one();
}

void
notify_all() noexcept
{
    _M_impl.notify_all();
}
```

They forward to `_Sp_atomic::notify_*`:

```
// _Sp_atomic<shared_ptr<T>>::notify_*
void
notify_one() noexcept
{
    _M_refcount.notify_one();
}

void
notify_all() noexcept
{
    _M_refcount.notify_all();
}
```

And those forward to `_Atomic_count::notify_*`:

```
// _Atomic_count::notify_*
void
notify_one() noexcept
{
    _M_val.notify_one();
}

void
notify_all() noexcept
{
    _M_val.notify_all();
}

mutable __atomic_base<uintptr_t> _M_val{0};
```

which forward the notify to the atomic value.

So at the end of the day, waiting on and notifying an atomic shared pointer boils down to waiting on and notifying its control block pointer.

But hang on a second. The language specification says that a wait on an atomic shared pointer is satisfied when *either* the stored pointer *or* the control block pointer changes. But this code waits only for the control block pointer to change. Do we have a bug?

Let's write a test program to see whether our theory holds up, or whether there's something else (like msvc's exponential backoff) that saves us.

```
#include <memory>
#include <chrono>
#include <thread>

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

void signaler()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    p.store({ q, nullptr });
    p.notify_one();
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::terminate();
}

int main(int, char**)
{
    std::thread(signaler).detach();
    p.wait(q);
    return 0;
}
```

This program starts a thread that waits one second to give the main thread a chance to reach `p.wait()`. It then changes the atomic shared pointer by modifying only the stored pointer and reusing the control block, and then notifies the main thread. If the program is still running after one second, then the wait was not woken, and we terminate the program.

Meanwhile, after starting the signaler thread, the main thread waits on the atomic shared pointer, and when the wait is satisfied, it exits the program.

You expect this program to exit cleanly. The signaling thread modifies the atomic shared pointer, which satisfies the wait. (Even if we didn't sleep one second before modifying the atomic shared pointer, the wait would still be satisfied because the value in the atomic shared pointer no longer matches `q`.)

In practice, this program crashes at the `std::terminate`.

So it looks like we found a bug in libstdc++. (`-dumpversion` says 14.2.0.) Waiting on an atomic shared pointer does not notify if the the shared pointer changed only its stored pointer and not its control block. The atomic wait should be a 16-byte wait that covers both the stored pointer and the control block pointer.

Bonus chatter: I find it interesting that the language specification added wait/notify support to atomic shared pointers as an afterthought, with barely any discussion or contemplation, as if it had been deemed too trivial to be worth worrying about. And two of the three major implementations messed it up. (What about the third major implementation, clang's libc++? Oh, they haven't implemented it yet!)

I was curious about this topic because the first thing that struck me about notify/wait on atomic shared pointers was "Gosh, shared pointers are twice the size of regular pointers. I wonder how the implementations manage to wait atomically on something that is larger than a register?" And when I dug into the implementations, I found that the answer was "not correctly."