Lock-free reference-counting a TLS slot using atomics, part 3

devblogs.microsoft.com/oldnewthing/20240614-00

June 14, 2024

Raymond Chen

Last time, we tried to remove the mutex bottleneck from our class that allocates a TLS on demand and frees the TLS slot when the last client disconnected. We figured out how to that allocate the TLS on demand, but freeing the TLS on the disconnection of the last client was a problem because of a race that can occur if an `Acquire()` occurs while the last reference is being `Release()`d.

This conflict between `Acquire()` and `Release()` arises because we are manipulating two separate atomic variables, when we really want to treat the two variables as an atomic unit.

So let's make them an atomic unit.

```
struct TlsManager
{
    struct State
    {
        DWORD count = 0;
        DWORD tls; // valid if count != 0
    };

    std::atomic<State> m_state;

    DWORD Acquire()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous;
            if (++state.count != 1) {
                if (m_state.compare_exchange_weak(previous, state)) {
                    return state.tls;
                }
            } else {
                state.tls = TlsAlloc();
                THROW_LAST_ERROR_IF(state.tls == TLS_OUT_OF_INDEXES);
                if (m_state.compare_exchange_weak(previous, state)) {
                    return state.tls;
                } else {
                    TlsFree(state.tls);
                }
            }
        }
    }
};
```

We capture the initial state and calculate what the desired new state is. We increment the reference count, and if we didn't increment to 1, then the increment is all we needed to do. Try to save this as the new state and return if successful. Otherwise, another thread won the race against us, so we restart the loop to try again. (When writing these types of lock-free algorithms, don't forget to loop back and try again if you want the operation to eventually succeed.)

If we incremented to 1, then we are also responsible for allocating the TLS slot. Allocate it and try to save the TLS slot and the incremented reference count as an atomic unit. If this succeeds, then return. Otherwise, clean up the TLS slot we mistakenly allocated and try again.

It's possible to optimize this loop a tiny bit more by caching the result of `TlsAlloc()` in case we go a second time through the `else` branch inside the loop. However, I don't think this is likely, because it means that we have to lose *two* races: While we are calling `TlsAlloc()`, another thread successfully performed an `Acquire()`, and then when we go back and try to increment the reference count, we find that another thread also successfully performed a `Release()`, forcing us to into the `TlsAlloc()` branch again.

This race would occur if another thread exactly interleaves an `Acquire()`/`Release()` pair inside our `Acquire()`. Some instrumentation would tell us whether this race is likely in practice.

I could imagine it being a possibility if `TlsAlloc()` and `TlsFree()` are slow enough that they open the necessary race window for the other thread to sneak in.

So let’s add the caching, just to see how it looks.

```
DWORD Acquire()
{
    wil::unique_tls tls;

    auto previous = m_state.load();
    while (true) {
        auto state = previous;
        if (++state.count != 1) {
            if (m_state.compare_exchange_weak(previous, state)) {
                return state.tls;
            }
        } else {
            if (!tls) {
                tls.reset(TlsAlloc());
                THROW_LAST_ERROR_IF(!tls);
            }
            state.tls = tls.get();
            if (m_state.compare_exchange_weak(previous, state)) {
                tls.release(); // owned by the TlsManager now
                return state.tls;
            }
        }
    }
}
```

We take advantage of the `wil::unique_tls` RAII type which manages a TLS slot.

With this combined state, we can now `Release()` atomically.

```
    void Release()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous;
            --state.count;
            if (m_state.compare_exchange_weak(previous, state)) {
                if (state.count == 0) {
                    TlsFree(state.tls);
                }
                return;
            }
        }
    }
```

Bonus chatter: I didn't talk about memory ordering, but the `.load()` calls can be weaked to acquire, and the `compare_exchange_weak()` calls can be weakened to release.

Bonus bonus chatter: Instead of using a structure, we can pack the values manually into a `uint64_t`. If we continue to assume that the 32-bit reference count won't overflow, we can increment and decrement the entire `uint64_t` rather than having to take it apart into two 32-bit integers.

```
// Version where the count is kept in the low-order bits.
struct TlsManager
{
    std::atomic<uint64_t> m_state;

    DWORD Acquire()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous + 1;
            if (static_cast<uint64_t>(state) != 1) {
                if (m_state.compare_exchange_weak(previous, state)) {
                    return static_cast<uint32_t>(state >> 32);
                }
            } else {
                auto tls = TlsAlloc();
                THROW_LAST_ERROR_IF(tls == TLS_OUT_OF_INDEXES);
                state = (static_cast<uint64_t>(tls) << 32) + 1;
                if (m_state.compare_exchange_weak(previous, state)) {
                    return static_cast<uint32_t>(state >> 32);
                } else {
                    TlsFree(tls);
                }
            }
        }
    }

    void Release()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous - 1;
            if (m_state.compare_exchange_weak(previous, state)) {
                if (static_cast<uint32_t>(state) == 0) {
                    TlsFree(static_cast<uint32_t>(state >> 32));
                }
                return;
            }
        }
    }
};
```

Or maybe put the count in the high-order 32 bits.

```
// Version where the count is kept in the high-order bits.
struct TlsManager
{
    std::atomic<uint64_t> m_state;
    static constexpr uint64_t unit = static_cast<uint64_t>(1) << 32;

    DWORD Acquire()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous + unit;
            if ((state >> 32) != 1) {
                if (m_state.compare_exchange_weak(previous, state)) {
                    return static_cast<uint32_t>(state);
                }
            } else {
                auto tls = TlsAlloc();
                THROW_LAST_ERROR_IF(tls == TLS_OUT_OF_INDEXES);
                state = tls + unit;
                if (m_state.compare_exchange_weak(previous, state)) {
                    return static_cast<uint32_t>(state);
                } else {
                    TlsFree(tls);
                }
            }
        }
    }

    void Release()
    {
        auto previous = m_state.load();
        while (true) {
            auto state = previous - unit;
            if (m_state.compare_exchange_weak(previous, state)) {
                if ((state >> 32) == 0) {
                    TlsFree(static_cast<uint32_t>(state));
                }
                return;
            }
        }
    }
};
```

Different compilers and different target architectures may produce better results for one formulation over another.