Lock-free reference-counting a TLS slot using atomics, part 2

devblogs.microsoft.com/oldnewthing/20240613-00

June 13, 2024

Raymond Chen

Last time, we wrote a class that allocated a TLS on demand and freed the TLS slot when the last client disconnected. We finished with a version that used a mutex, and we noted that profiling might reveal that the mutex is too expensive because the code is constantly creating and destroying `TlsUsage` objects. (Then again, it might not, in which case, you may as well stick with the mutex.)

If you just follow the traditional pattern for singleton construction, you might come up with something like this for lazy allocation of the TLS slot:

```
struct TlsManager
{
    std::atomic<DWORD> m_count = 0;
    std::atomic<DWORD> m_tls = TLS_OUT_OF_INDEXES;

    // Don't use this code. See text.
    DWORD Acquire()
    {
        if (++m_count != 1) {
            return m_tls.load();
        }

        // Lazy-create the TLS slot.
        auto tls = TlsAlloc();
        THROW_LAST_ERROR_IF(tls == TLS_OUT_OF_INDEXES);
        DWORD previous = TLS_OUT_OF_INDEXES;
        if (!m_tls.compare_exchange_strong(previous, tls)) {
            // Lost the race.
            TlsFree(tls);
            tls = previous;
        }
        return tls;
    }
};
```

The thinking here is that if the previous reference count was already nonzero, then we can count on the TLS already having been allocated. Otherwise, we are the ones who incremented the count from zero to one, so we have to create it, using the standard singleton creation pattern.

You might wonder, "Wait, if we attempt to create the TLS slot only when the reference count goes from 0 to 1, then how could the `compare_exchange_strong` fail?" That's a fair question, and thinking about it leads you to discovering why this code is wrong.

The danger is that after we increment the reference count from 0 to 1, another thread may call `Acquire` and increment the count from 1 to 2. That other thread sees that the reference count was already nonzero, so it says, "Well, then clearly *I* don't need to initialize the TLS slot because the guy who incremented from 0 to 1 is responsible for doing that," and returns with the existing TLS slot.

The problem is that "the guy responsible for doing that" is not finished doing that.

One way to solve this is to make it the responsibility of *every* increment to initialize the TLS. Even if you incremented from 1 to 2, it's possible that the thread that incremented from to 0 to 1 hasn't finished the initialization yet, so you have to try to do it too, just in case.

This does mean that every increment has to attempt an initialization, which is a lot of wasted calls to `TlsAlloc`. We can avoid those wasted calls by peeking at `m_tls` and returning early if we see that its value is not `TLS_OUT_OF_INDEXES`.

```
    DWORD Acquire()
    {
        if (++m_count != 1) {
            return m_tls.load();
        }

        DWORD previous = m_tls.load();
        if (previous != TLS_OUT_OF_INDEXES) {
            // Already created
            return previous;
        }

        // Lazy-create the TLS slot.
        auto tls = TlsAlloc();
        THROW_LAST_ERROR_IF(tls == TLS_OUT_OF_INDEXES);
        auto previous = TLS_OUT_OF_INDEXES;
        if (!m_tls.compare_exchange_strong(previous, tls)) {
            // Lost the race.
            TlsFree(tls);
            tls = previous;
        }
        return tls;
    }
};
```

But wait, the adventure is only beginning. The `Release` method is much trickier. A naïve attempt might go like this:

```
    // Don't use this code. See text.
    void Release()
    {
        if (--m_count == 0) {
            TlsFree(m_tls.exchange(TLS_OUT_OF_INDEXES));
        }
    }
```

The idea here is that if we decrement to zero, then we atomically reset the `m_tls` back to its "no TLS" sentinel value and free the TLS slot that used to be there.

Unfortunately, this doesn't work because it's possible that after the `--m_count` decrements the counter to zero, but before we can free the TLS slot in `m_tls`, another thread sneaks in and calls `Acquire()`.

That concurrent call to `Acquire()` bumps the reference count back up to 1, and then sees that we already have a TLS allocated, so it just returns. The `Release()` function then proceeds to exchange and free the TLS slot that the `Acquire()` was using.

Thread 1	Thread 2

`Release()` `--m_count` (decrements to zero)	

	`Acquire()`
	`++m_count` (increments to one)
	`previous = m_tls.load()` (valid TLS) `return previous;`

`tls = m_tls.exchange(INVALID)` `TlsFree(tls)` (oops! frees in-use TLS!)	

Addressing this race between `Acquire()` and `Release()` is going to require a different approach. We'll investigate further next time.