Lock-free reference-counting a TLS slot using atomics, part 1

devblogs.microsoft.com/oldnewthing/20240612-00

June 12, 2024

Raymond Chen

Some time ago, we spent time looking at various lock-free algorithms, one of which is the lock-free singleton constructor. But suppose you want your singleton to be reference-counted?

To make things concrete, let's suppose that we want a class which manages a TLS slot, allocating it on demand, and freeing it when there are no longer any users.

Let's start with a sketch of how we want this to work, but without worrying about atomicity yet.

```
// Note: Not finished yet
struct TlsManager
{
    DWORD m_count = 0;
    DWORD m_tls = TLS_OUT_OF_INDEXES;

    void Acquire()
    {
        if (++m_count == 1) {
            m_tls = TlsAlloc();
            THROW_LAST_ERROR_IF(m_tls == TLS_OUT_OF_INDEXES);
        }
    }

    void Release()
    {
        if (--m_count == 0) {
            TlsFree(std::exchange(m_tls, TLS_OUT_OF_INDEXES));
        }
    }
};

struct TlsUsage
{
    TlsUsage() = default;

    explicit TlsUsage(TlsManager& manager) :
        m_manager(&manager) { manager.Acquire(); }

    TlsUsage(TlsUsage&& other) :
        m_manager(std::exchange(other.manager, nullptr)) {}

    TlsUsage& operator=(TlsUsage&& other) {
        std::swap(m_manager, other.m_manager);
    }

    ~TlsUsage()
    {
        if (m_manager) m_manager->Release();
    }

    void* GetValue()
    {
        return TlsGetValue(m_manager->m_tls);
    }

    void SetValue(void* value)
    {
        TlsSetValue(m_manager->m_tls, value);
    }
```

```
    TlsManager* m_manager = nullptr;
};
```

The idea here is that a `TlsManager` is the object that manages access to a TLS slot. You call `Acquire` to start using the TLS slot (allocating it on demand), and you can use that slot until you call `Release`. When the last consumer of a slot calls `Release`, the slot is freed.

Instead of talking directly to the `TlsManager`, you use a `TlsUsage`, which is an RAII type that deals with the acquire/release protocol for you.

To make the `TlsManager` thread-safe, we can add locks:

```
struct TlsManager
{
    DWORD m_count = 0;
    DWORD m_tls = TLS_OUT_OF_INDEXES;
    std::mutex m_mutex;

    void Acquire()
    {
        auto lock = std::unique_lock(m_mutex);

        if (++m_count == 1) {
            m_tls = TlsAlloc();
            THROW_LAST_ERROR_IF(m_tls == TLS_OUT_OF_INDEXES);
        }
    }

    void Release()
    {
        auto lock = std::unique_lock(m_mutex);

        if (--m_count == 0) {
            TlsFree(std::exchange(m_tls, TLS_OUT_OF_INDEXES));
        }
    }
};
```

Now, in practice, this might end up being efficient enough if `TlsUsage` objects are not frequently created and destroyed. But you might be in a case where your program is constantly creating and destroying `Widget` objects, and each `Widget` needs a `TlsUsage`. That lock might end up being a bottleneck. We'll try to address this next time.

Update: `TlsUsage` move constructor and assignment fixed.