sign up log in tour help stack overflow careers

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required. Take the 2-minute tour ×

C++ memory ordering

In some tutorial i saw such spin lock implementation

```
class spin_lock
{
    atomic<unsigned int> m_spin ;

public:
    spin_lock(): m_spin(0) {}
    ~spin_lock() { assert( m_spin.load(memory_order_relaxed) == 0);}

    void lock()
    {
        unsigned int nCur;
        do { nCur = 0; }
        while ( !m_spin.compare_exchange_weak( nCur, 1, memory_order_acquire ));
    }
    void unlock()
    {
        m_spin.store( 0, memory_order_release );
    }
};
```

Do we really need `memory_order_acquire` / `memory_order_release` tags for `compare_exchange_weak` and `store` operations respectively? Or `memory_order_relaxed` is sufficient in this case as there is no Synchronizes-With relation?

Update: Thank you for the explanations! I was considering spin_lock without context in which it is used.

c++ c++11 memory-model

edited Dec 24 '13 at 7:12

asked Dec 24 '13 at 6:33

sliser
927 4 9

3 "there is no Synchronizes-With relation" Yes, there is. Eg: Suppose there is a spin_lock protect a shared resource, and thread A use spin_lock.lock() to acquire the resource, while the thread B is unlocking the lock. There need a memory order to prevent instruction reorder. – KaiWen Dec 24 '13 at 6:41

I would implement spinlocks with `atomic_flag`, since it's the only guaranteed portable lock-free atomic type. – Casey Dec 24 '13 at 14:10

2 Answers

With regard to memory_order_acquire / memory_order_release: consider the shared data (a.k.a. memory locations, or resource) that you are using the lock to protect. I'll call that "the protected data".

The code needs to ensure that when lock() returns, any access to the protected data by the caller will read valid values (i.e. not stale values). memory_order_aquire ensures that the appropriate aquire memory barrier is inserted so that subsequent reads of the protected data (via the local CPU cache) will be valid. Similarly, when unlock() is called, memory_order_release is needed to ensure that the appropriate memory barrier is inserted so that other caches are correctly synchronised.

Some processors don't require aquire/release barriers and might, for a theoretical example, only require a full barrier in lock(). But the C++ concurrency model needs to be flexible enough to support many different processor architectures.

memory_order_relaxed is used in the destructor because this is just a sanity check to ensure that the lock isn't currently held. Destructing the lock does not confer any synchronisation

semantics to the caller.

answered Dec 24 '13 at 7:02
Ross Bencina
919 2 12

Yes `memory_order_acquire` / `memory_order_release` is needed.

You use locks to protect shared data between multiple threads. if you don't use `memory_order_release` in `unlock` method, any write on shared data can be reordered after `unlock` method. Also if you don't use `memory_order_acquire` in `lock` method any read on shared data can be reordered before `lock` method. so you need `acquire/release` to protect shared data between threads.

```
spinLock.lock() // use acquire here, so any read can't reordered before `lock`

// Writes to shared data

spinLock.unlock() // use release here, so any write can't reordered after `unlock`
```

with `acquire/release` all writes to shared data will be visible to threads that lock spinlock.

edited Dec 24 '13 at 7:06

answered Dec 24 '13 at 7:01
MohammadRB
1,083 1 4 15