Lock-free programming, part 1

Sonntag, 3. Juli 2022 18:07

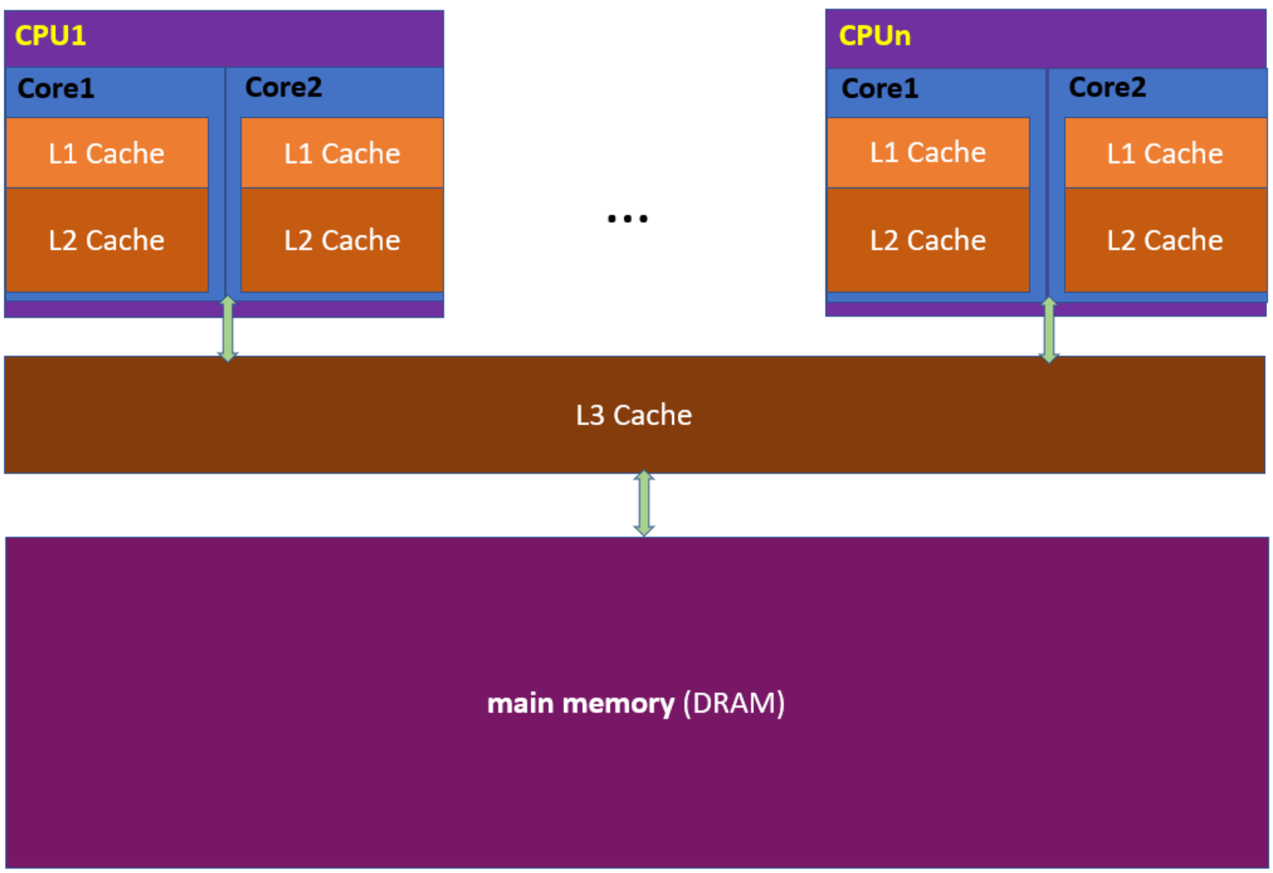
Author: Damir Ljubic
email: damirlj@yahoo.com

Links:
https://www.youtube.com/watch?v=j2AgjFSFgRc
Concurrency in C++: A Programmer's Overview (part 1 of 2) - Fedor Pikus - CppNow 2022 - YouTube
Concurrency in C++: A Programmer's Overview (part 2 of 2) - Fedor Pikus - CppNow 2022 - YouTube
(62) CppCon 2017: Fedor Pikus "C++ atomics, from basic to advanced. What do they really do?" - YouTube
(63) Cache consistency and the C++ memory model: writing code to (...) - Yossi Moalem - code::dive 2019 - YouTube
(63) C++ and Beyond 2012: Herb Sutter - atomic Weapons 1 of 2 - YouTube
(63) C++ and Beyond 2012: Herb Sutter - atomic Weapons 2 of 2 - YouTube

Anthony Williams: Concurrency in Action, Second Edition (source code)
GitHub - anthonywilliams/ccia_code_samples: Code samples for C++ Concurrency in Action

Atomic types

Starting with C++11 edition of the standard library, we have for the first time multicores-aware memory model.
It recognizes the contemporary multicore architectures, and how the memory is organized.



This opens a whole new area of parallel (as oppose to concurrent) programming, but this is not today's topic.
With new memory model, we also got **std::atomic<T>** type, a nifty feature that supposes to ensure

- **atomicity**
  Operation on atomic types are atomic.
  That is to say, atomic operation is a single-instruction that can't be interrupted by the system, and therefore one can't see the intermediate result of operation on atomic type.
  Two threads can observe the state of the system before or after atomic operation, but not in between.
  This is the way to avoid *race condition* - when multiple threads simultaneously access the same shared memory, and at least one of them modify it, what leads to inconsistency.

  @note Since only single instruction is non-interruptible, this enforces the constraint on the size (depends on the architecture), but also some other constraints when we want to provide in place of T some user-defined type (UDT).

- **memory ordering**
  It's the way to instruct compiler/CPUs in matters of reordering atomic (but also indirectly non-atomic) operations, to have some "predictable" execution flow.
  It's about establishing certain guarantees on ordering, that would affect visibility of side-effects of one thread, in context of another threads, across cores/processors boundaries.

Atomicity

All embedded - scalar types satisfied the single-instruction size constraint, and therefore operations should be lock-free.
For user-defined types, that may not be the case.
This means, the internal locking mechanism may be surprisingly engaged, which is something that one needs to be aware.

Full example: https://godbolt.org/z/KPYzeshhs

But, there is a trick - storing a pointer into atomic placeholder ***std::atomic<T*>*** satisfies always the single-instruction size constraint, and therefore can be used in lock-free algorithms with the notion that what it points out - the underlying type, is not lock-free.

The following lines are pseudo-representation of the std::atomic<T>, to emphasize additional constraints on UDT - the type must have *trivial* copy assignment operator: the default one, provided by the compiler.
This also means:
- for aggregate types, all non-static members needs to be trivially copyable
- no virtual base classes or virtual methods

The reason behind is that the type should be seen as a contiguous block of raw bytes, so that it could be easily, in single-instruction bitwise assigned (as with *memcopy*)

```
template <typename T>
class atomic
{
    T val_;
    std::mutex lock_; // in case that atomic operations on type T can't be lock-free

    public:
        /**
         *   Implicit c-tor
         *   Constraint on type T - it must be trivially constructible
        */
        constexpr atomic(T val) noexcept(std::is_nothrow_default_constructible_v<T>) : val_(val)
        {}

        // Copy operations are forbidden, since assigning one atomic value to another is not atomic operation
        atomic(const atomic&) = delete;
        atomic& operator = (const atomic&) = delete;

        // copy from operator
        T operator = (T val) noexcept
        {
            store(val);
            return val;
        }

        // convert to operator
        operator T () const noexcept
        {
            return load();
        }

        bool is_lock_free() const noexcept;

        // Atomic write/read operations

        void store(T val, std::memory_order ordering = std::memory_order_seq_cst) noexcept
        {
            // pseudo code
            if constexpr (is_always_lock_free) // indication that atomic type T is lock-free on all platforms
            {
                val_ = val;
            }
            else
            {
                std::guard_lock lock {lock_};
                val_ = val;
            }
        }
```

```
        T load(std::memory_order ordering = std::memory_order_seq_cst) const noexcept;

        // Atomic read-modified-write operations

        T exchange(T val, std::memory_order order = std::memory_order_seq_cst) noexcept
        {
            //pseudo code
            auto oldVal = load(order_acq);
            store(val, order_rel);

            return oldVal;
        }

        // Atomic CAS - Compare And Swap method, also known as conditional exchange
        // It comes in two versions: weak - that can spuriously fail, and the strong one
        // There are few overloaded versions, this is just one of them
        // https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange

        bool compare_exchange_*(T& expected, T val, std::memory_order success, std::memory_order failure)
        {
            // pseudo code

            // if the success: std::memory_order_acq_rel, success_acq will be turned into std::memory_order_acquire
            // In case that success: std::memory_order_release, success_acq will be std::memory_order_relaxed

            if (load(success_acq) != expected) // value meanwhile modified by other thread
            {
                expected = load(failure); // new - updated expected value
                return false;
            }

            store(val, success_rel); // if the success: std::memory_order_acq_rel, success_rel will be turned into
                                     //  std::memory_order_release
            return true;
        }
};
```

One of the most interesting atomic operations is **CAS - Compare And Swap**, which is base of many lock-free algorithms.
One also needs to be aware, that it uses bitwise comparation, so providing a comparison operators (==, !=) along with UDT, with some other comparison logic - value comparison (floating points arithmetic), would most probably yield some unpredictable results.

Here is an example of trivial spinlock implementation, that relies on CAS

```
namespace details
{
    class SpinLock
    {
        public:
            void lock()
            {
                using namespace std::chrono_literals;

                bool expected = false;
                while(!locked_.compare_exchange_weak(expected,
                                                     true,
                                                     std::memory_order_acq_rel,
                                                     std::memory_order_relaxed))
                {
                    std::this_thread::sleep_for(1ms);
                    expected = false
                }
            }

            void unlock()
            {
                locked_.store(false, std::memory_order_release);
            }

        private:
            std::atomic<bool> locked_ = false;
    };
}
```

It provides the same interface as std::mutex and therefore can be used with std::lock_guard (not that you should!)

Full example: https://godbolt.org/z/v8oKWdTh9

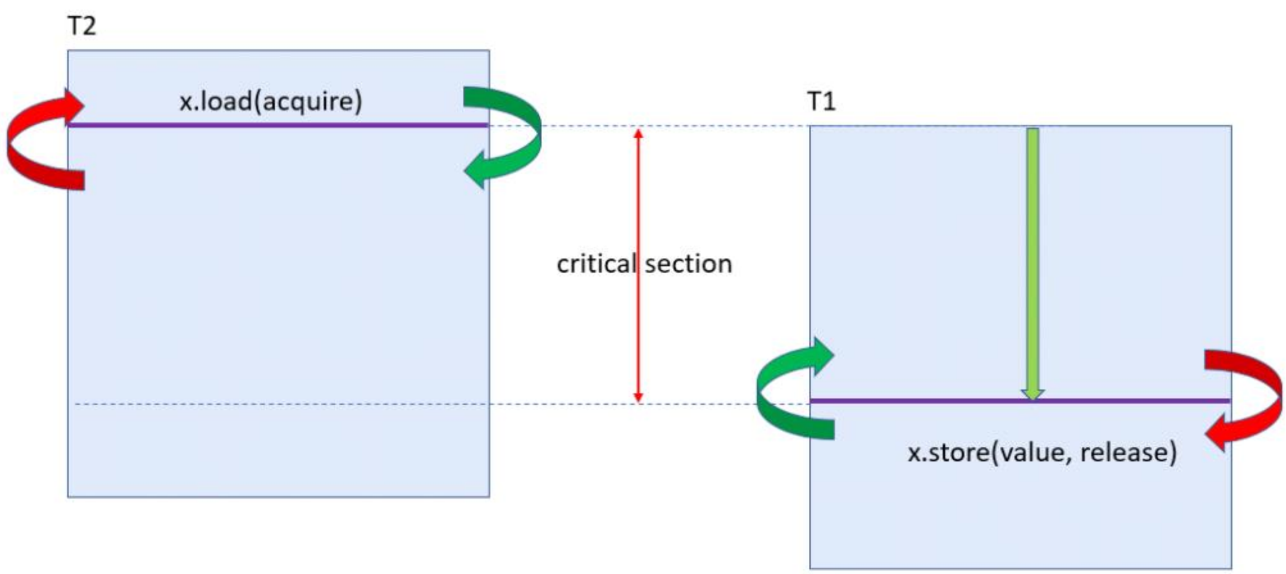
Memory ordering

There are basically two important relationships between the operations on atomic types:
"*synchronizes-with*" and "*happens-before*".
Very briefly, "*synchronizes-with*" establishes relationship between store and load on the same atomic type, where if these operations occur on different threads, we talk about inter-thread happens-before relationship.
This is also transitive. Meaning, if A is sequenced-before B in the same thread (strongly happens-before), and B inter-thread happens before C, then A also inter-thread happens before C.
But this is only conditionally true, and depends on the additional parameter of the most atomic operations: *std::memory_order*.

Default memory ordering is **sequentially consistent** one.
It's a global, total ordering between all atomics operations that guarantees basically that their order will be preserved across all CPUs, i.e. that all threads will observed (in memory) all atomic operations - all artifacts of the other threads.
This is the most rigid one, and also from performance point - the most expensive one memory ordering, since it doesn't allow any reordering and requires synchronization of all cores in system, for each operation.

There are other, more relaxed memory ordering.

std::memory_order_relaxed is the quite opposite to sequentially consistent memory ordering, since it doesn't impose any memory ordering at all.
The consequence of that is not being able to establish "*synchronizes-with*" relationship between store and load operation on the same atomic type in multi-thread context.
"*Synchronizes-with*" is the way to enforce the CPU (at runtime) to set the ***memory barriers*** between load and store, forming the "*critical section*", so that no operations (atomic and non-atomic) can't leave the fences - can't be reordered.
This is **Release-Acquire** memory ordering (std::memory_order - cppreference.com)



std::memory_order_release guarantees that all operations (non-atomic and atomic relaxed) that "*happens-before*" the barrier int the T1 thread, can't be reordered *AFTER* barrier (outside critical section).
This means, that all other threads that "*synchronizes-with*" this store - establish critical section by acquiring the same value with load, will *observed* all these operations in thread T1.
If these threads run potentially on different cores, to be observed, write operations needs to be flashed to main memory (release) and fetched (acquire) from the main memory back into cores private caches.
Similar **std::memory_order_acquire** prevents all read/write operations that happens in current thread to be reordered *BEFORE* - in front of the barrier.
Finally, for both barriers, reorder may happens only for the operations that are outside the critical section - by moving them inside.

Simple example, to demonstrate how the Release-Acquire ordering can be used to synchronize two threads, around non-atomic type

```
int x = 0;
std::atomic<bool> ready = false;

std::thread t1([&]{
    x = 1; // (1) sequenced-before and can't be reordered - can't cross the barrier
    ready.store(true, std::memory_order_release); // (2) inter-thread happens-before (3) => (1) inter-thread happens
before (3) and (4)
```

```
});

std::thread t2([&]{
    while(!ready.load(std::memory_order_acquire)){ // (3) synchronized with (2): (1) is visible
      std::this_thread::yield();
    }

    ++x; // (4) can't be reorder before the load - acquire barrier
});

t1.join();
t2.join();

assert(x == 2);
```

Full example: https://godbolt.org/z/ha3EYsdq9