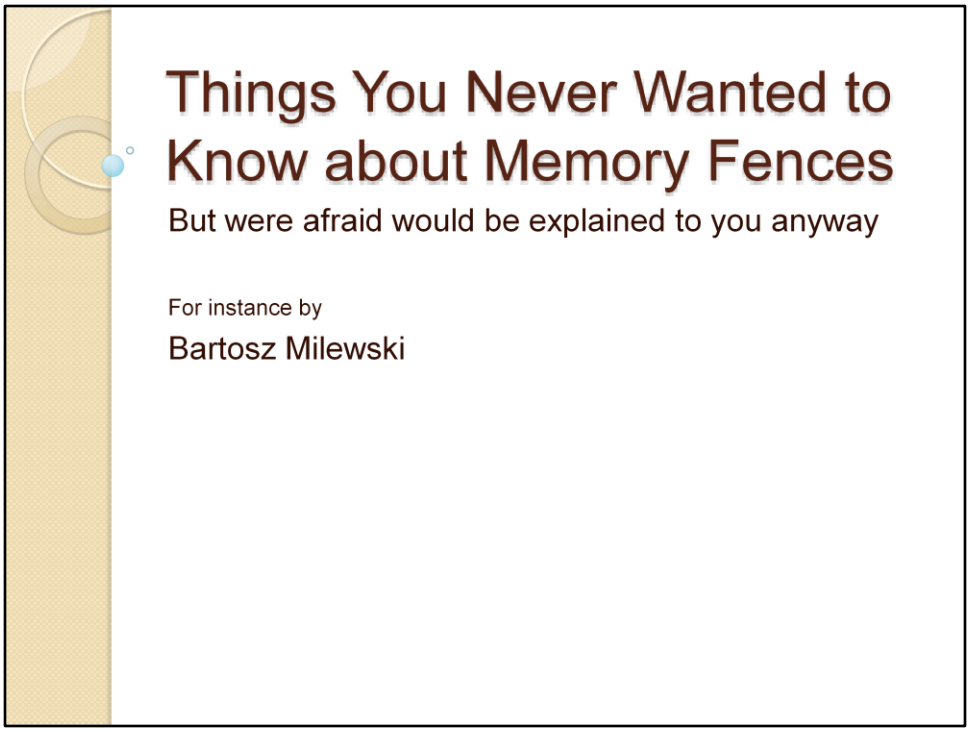
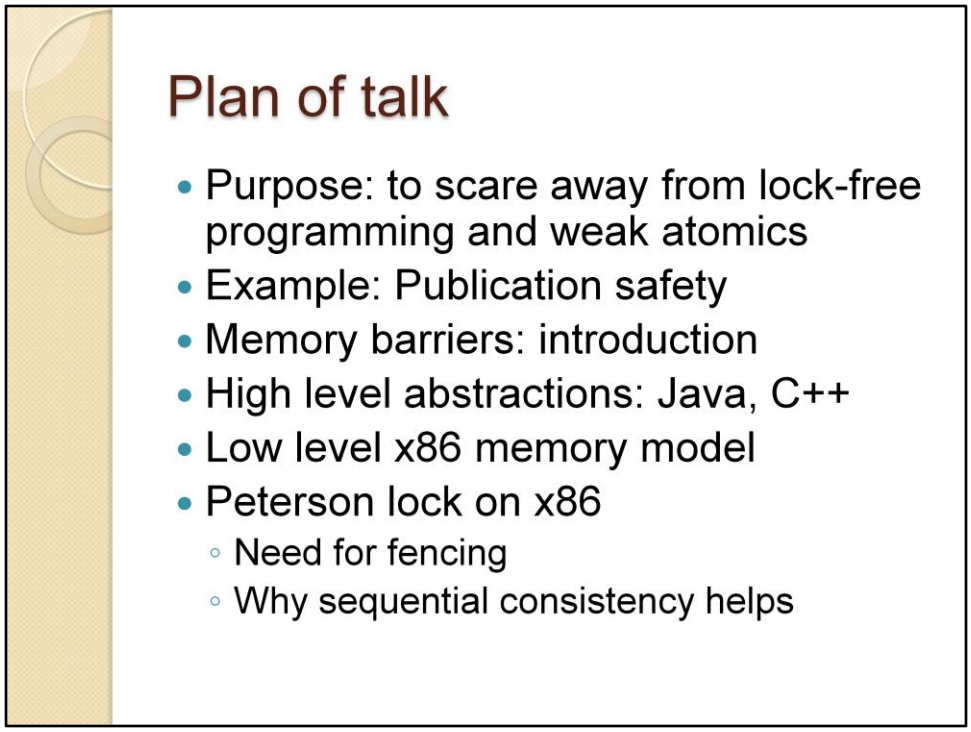
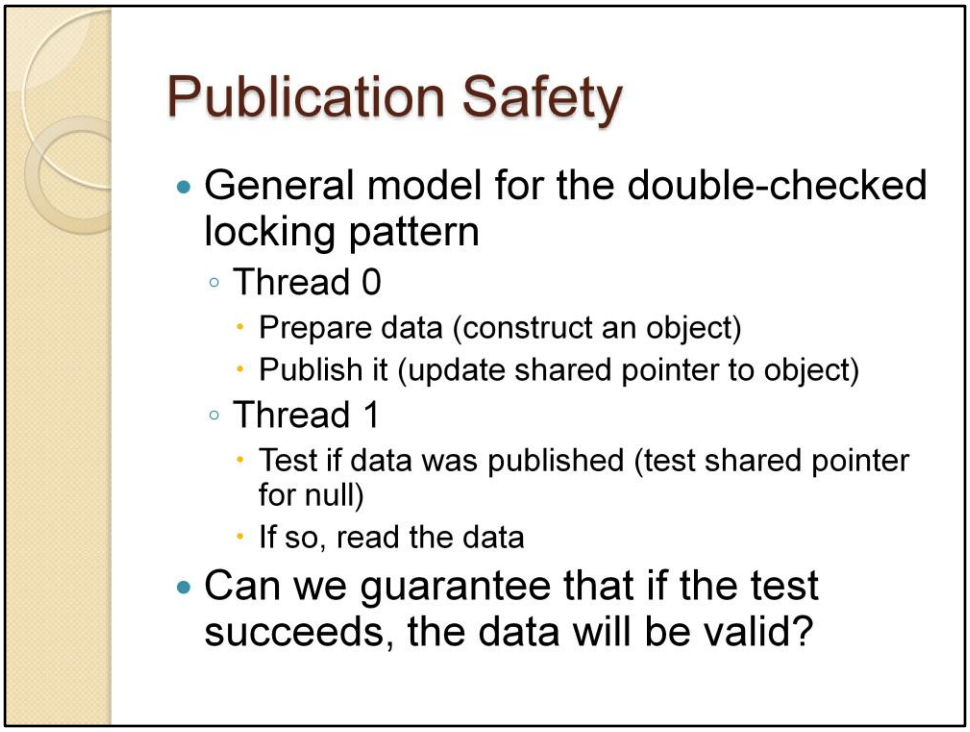


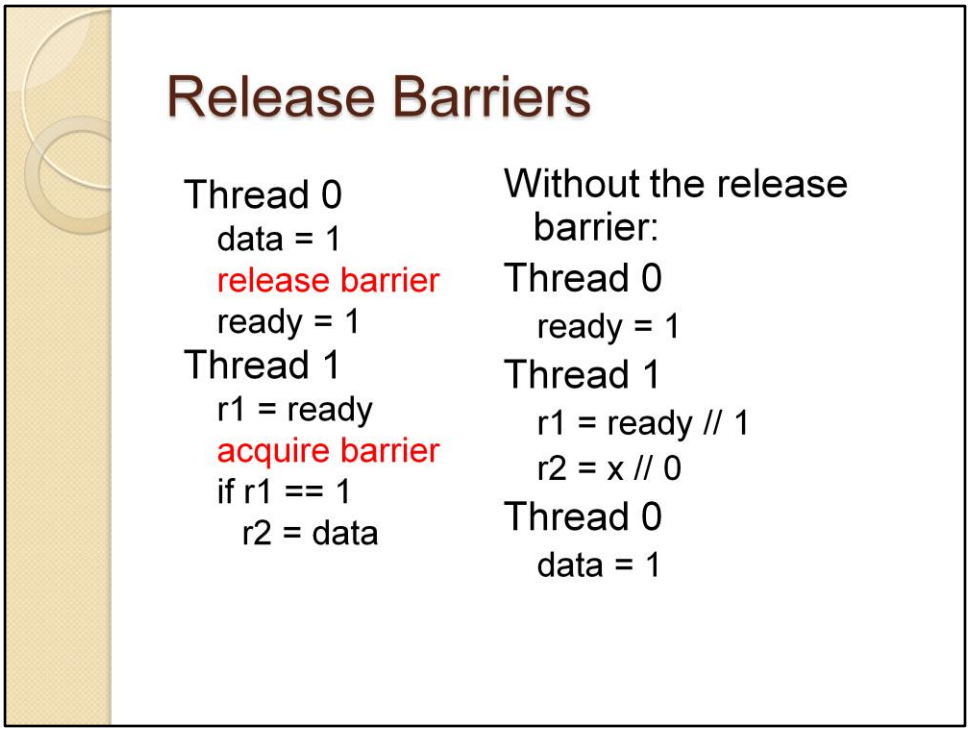
Ignorance is bliss as far as lock-free programming goes. Except when we don’t realize the depths of our ignorance and think that we can deal with it.



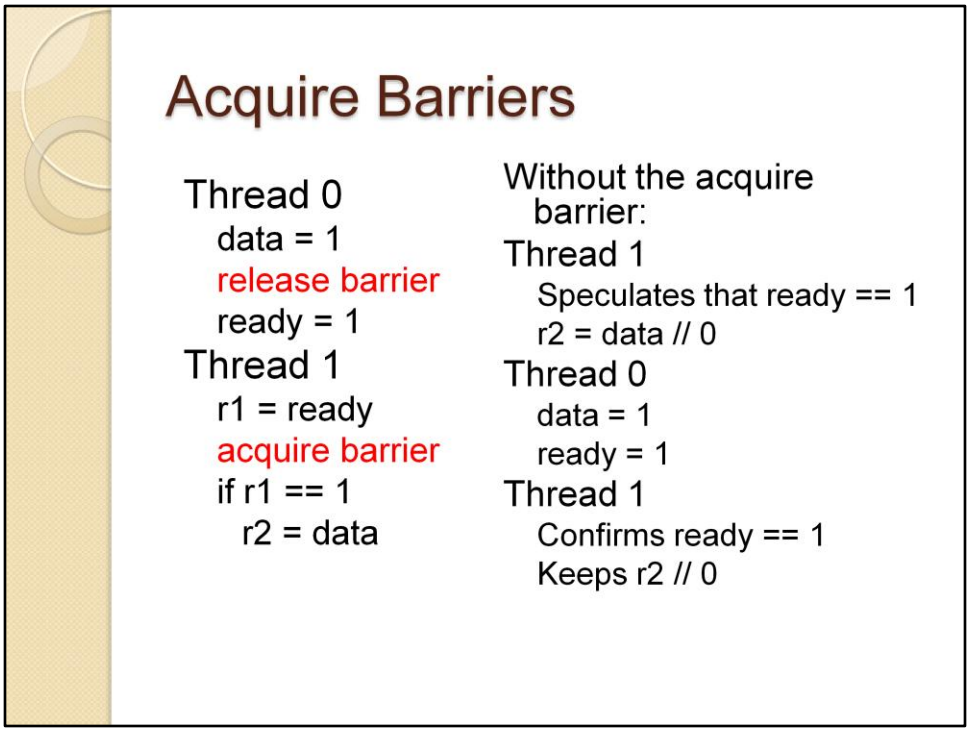
I'm trying to make the connection between high-level language constructs and the low level workings of an x86.



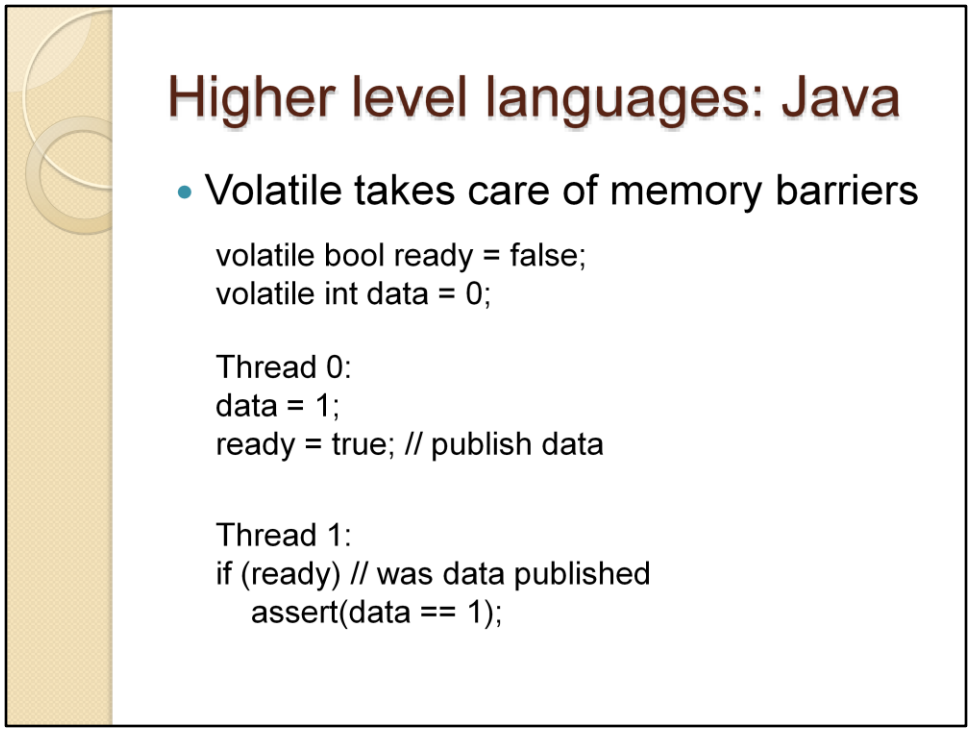
This is one of the most common patterns in lock free programming. A lot of programmers don't even realize they are doing lock-free programming when they are using this pattern. Depending on the compilers and the underlying processor, the result can be surprising—access to invalid data.



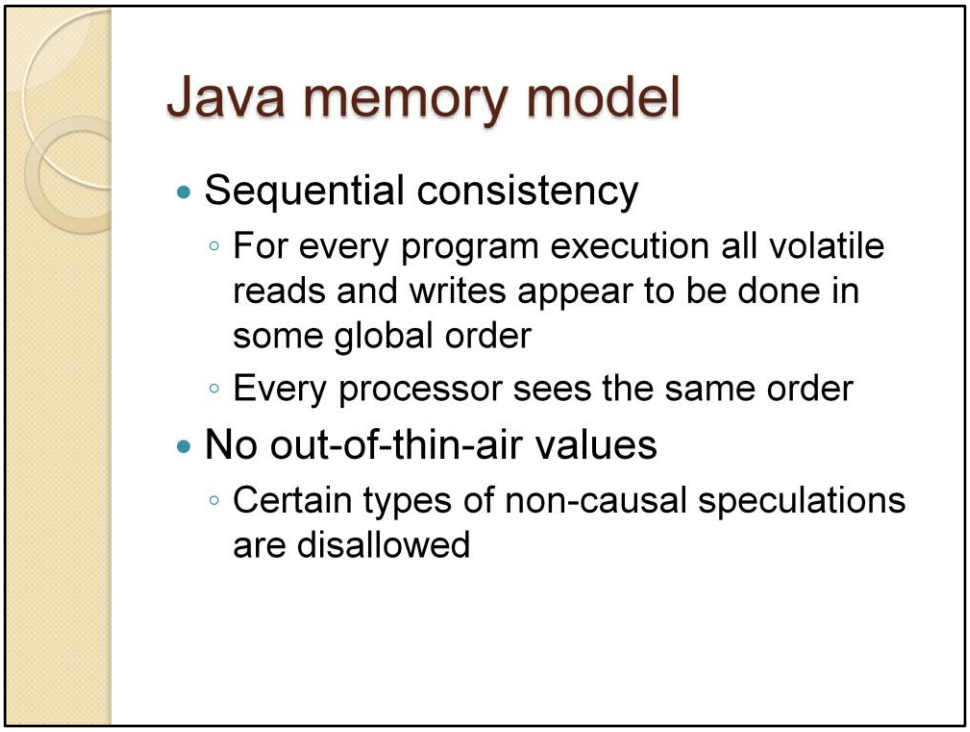
Initially data == 0, ready == 0. A release memory barrier (this is pseudo-code) separates writes. An acquire barrier separates reads. Without the release barrier the order of writes may appear switched. The right column shows a particular execution (interleaving of the two threads) in a relaxed memory model. Thread 1 ends up using invalid data.



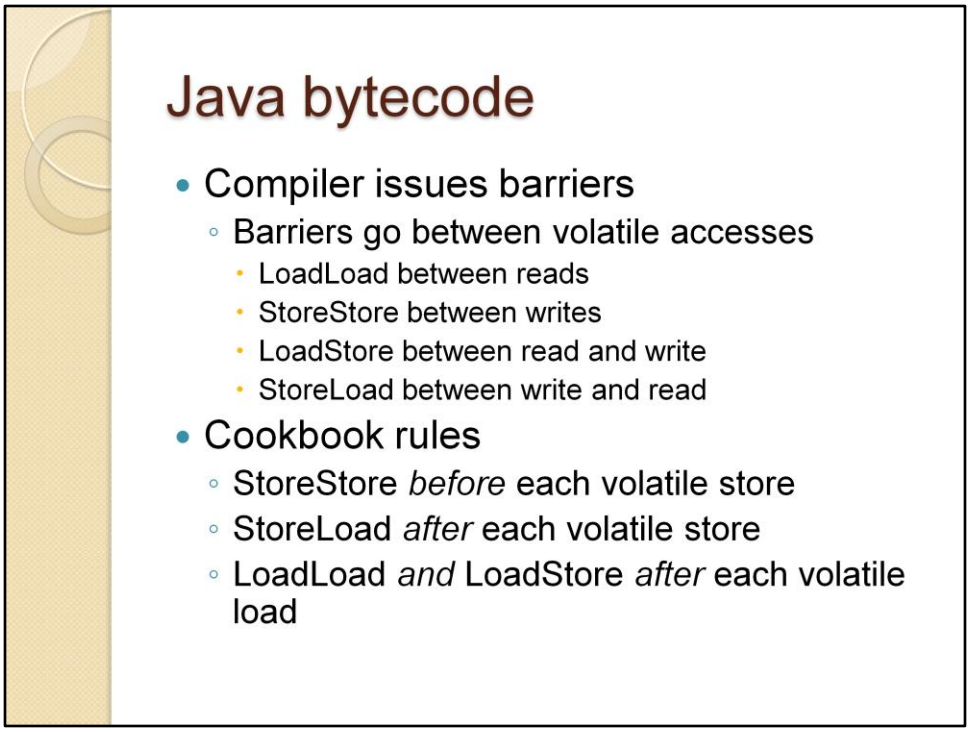
Without the acquire barrier, the reads ma be inverted. This may happen even if there is control dependency between the two reads (the reading of data happens only when ready is 1). The processor may speculate the value of ready and tentatively execute the load of data. Later the speculation is confirmed and the incorrect value of data is used.



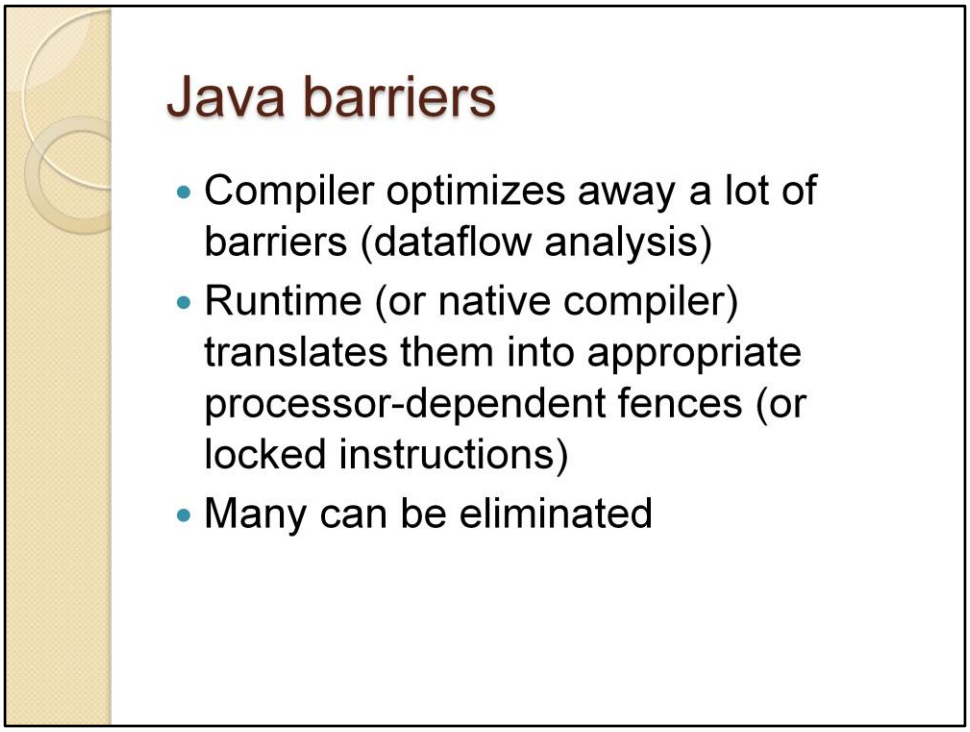
Java volatile guarantees appropriate barriers around reads and writes. Warning: the keyword “volatile” in C++ has completely different meaning. Even vendor specific extensions of the semantics of volatile might work only in some cases and fail in others (see Peterson lock later)



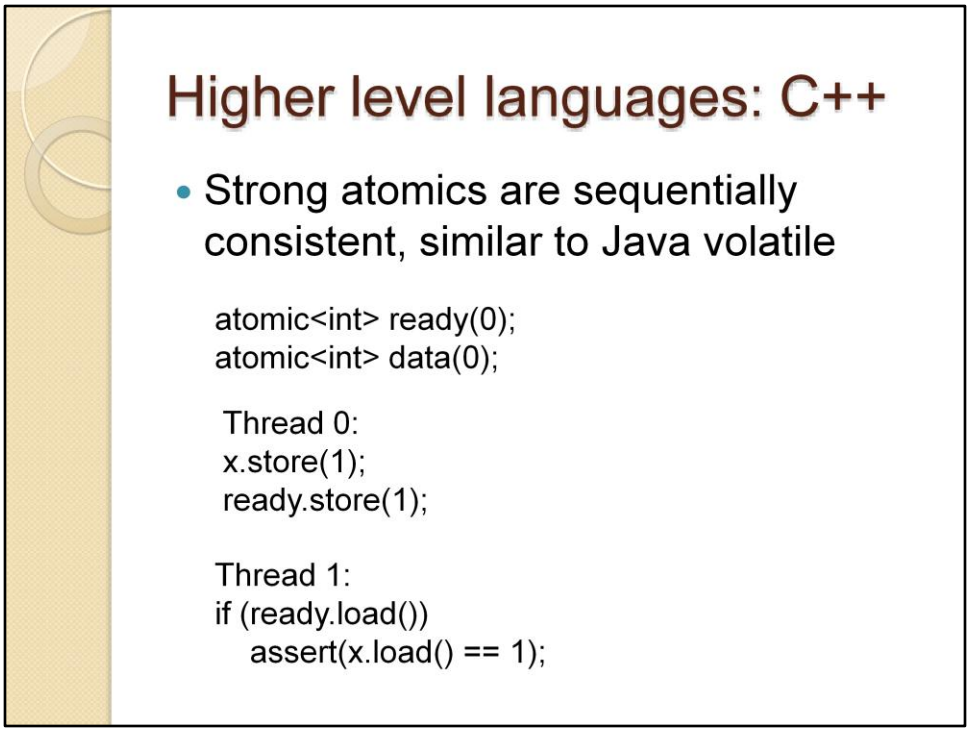
Java had a formal memory model before C++. The important part is the sequential consistency guarantee.



There are four types of barriers used by the Java compiler (they appear in the bytecode whenever volatile access is performed). Since the compiler cannot always predict what kind of access will follow the current access, the between-accesses barrier rules turn into around-each-access rules. Barriers are issued before and after each volatile write, and after each volatile read.



Cookbook rules produce a lot of barriers. They are pruned extensively.



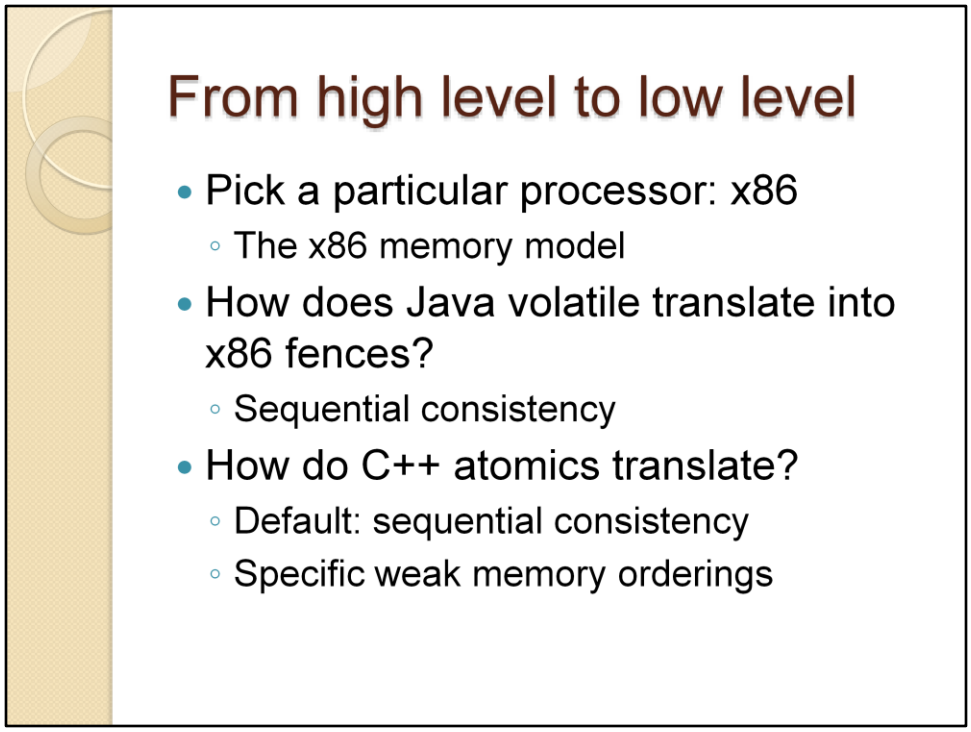
Atomics have load and store methods, among others. (They also have overloads of the assignment operator.) These atomics guarantee sequential consistency, just like Java's volatile.

C++ weak atomics: fine grain control

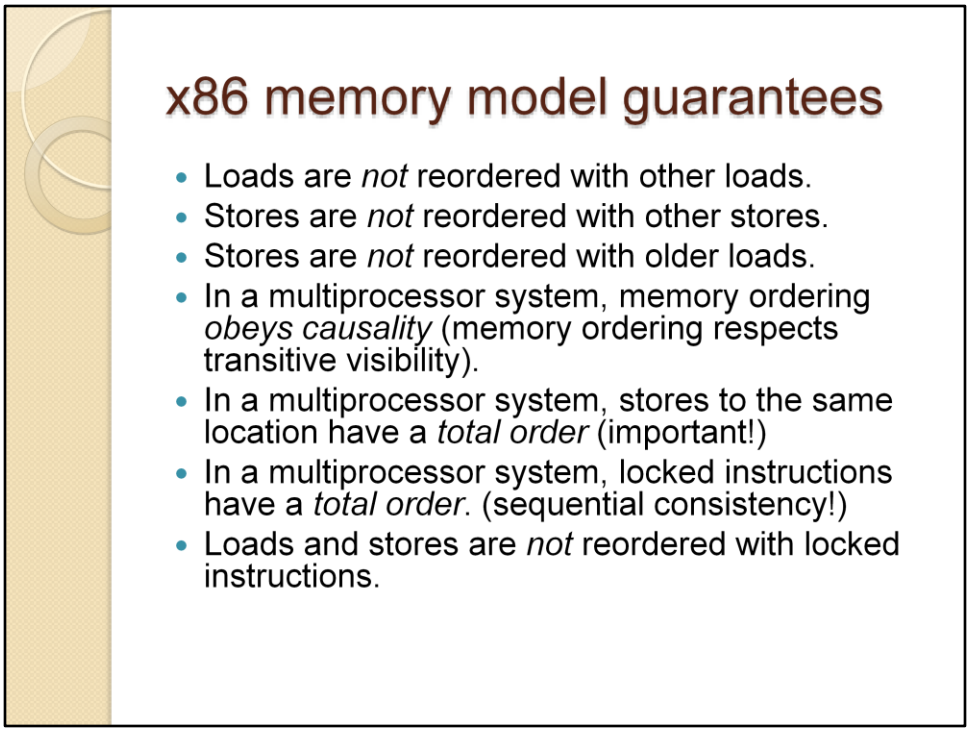
- You may specify ordering

```
atomic<int> ready(0);
atomic<int> data(0);

Thread 0:
x.store(1, memory_order_release);
ready.store(1, memory_order_release);


Thread 1:
if (ready.load(memory_order_acquire))
    assert(x.load(memory_order_acquire) == 1);
```

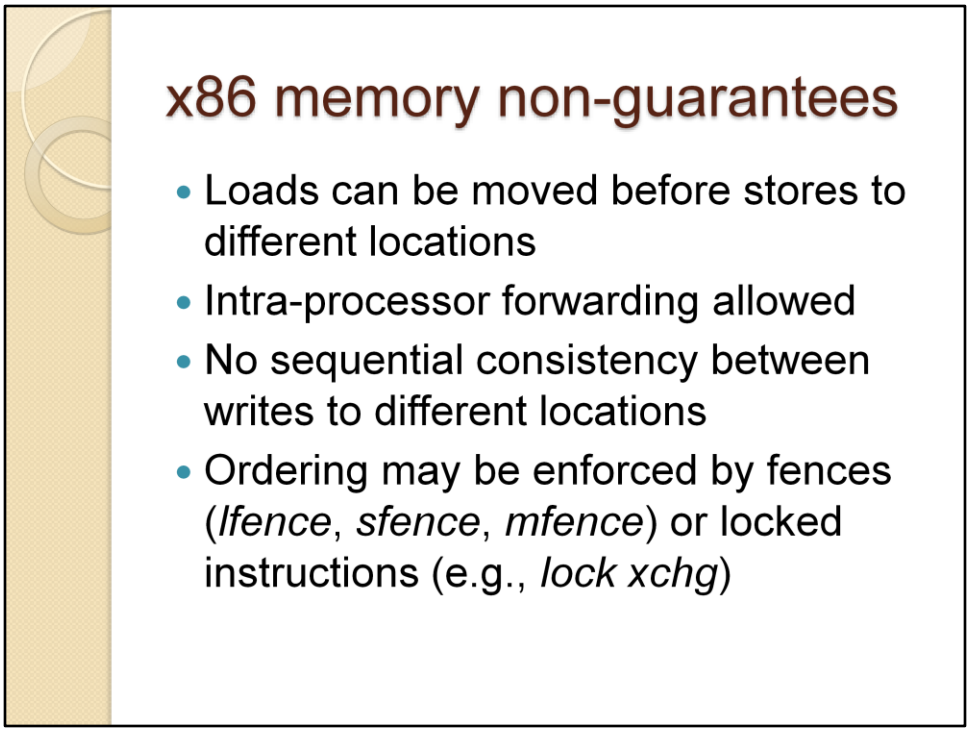
You can add an additional argument to loads and stores if you want to relax sequential consistency. In particular, publication safety only requires release and acquire ordering—not full sequential consistency.



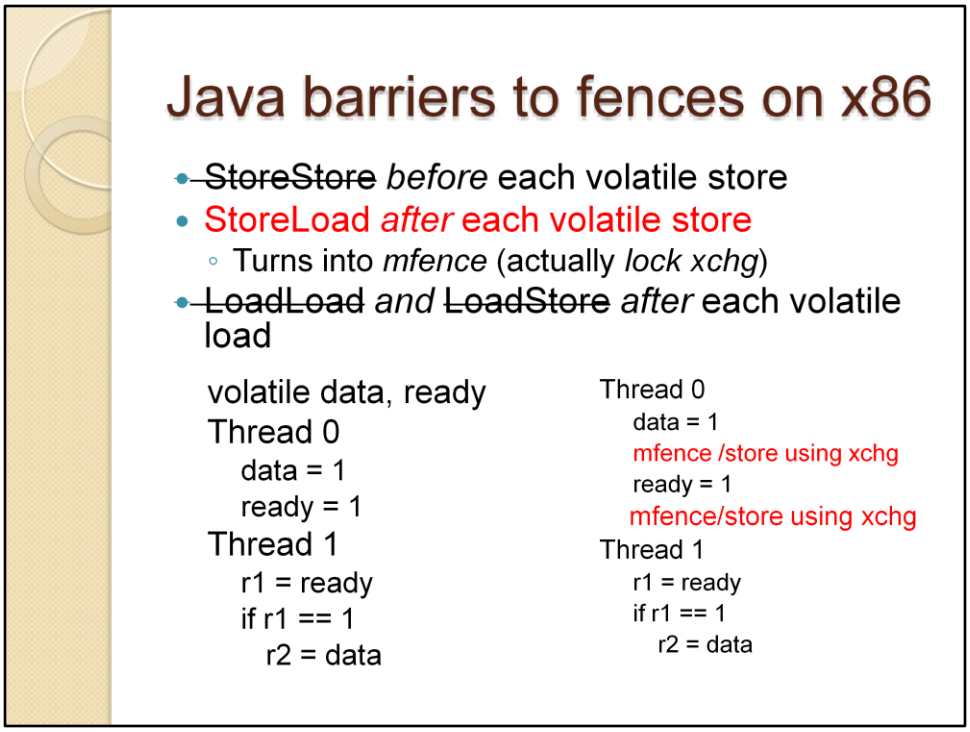
How do compilers (libraries) deal with the quirks of particular processors? In particular the x86 (starting with Pentium 4).



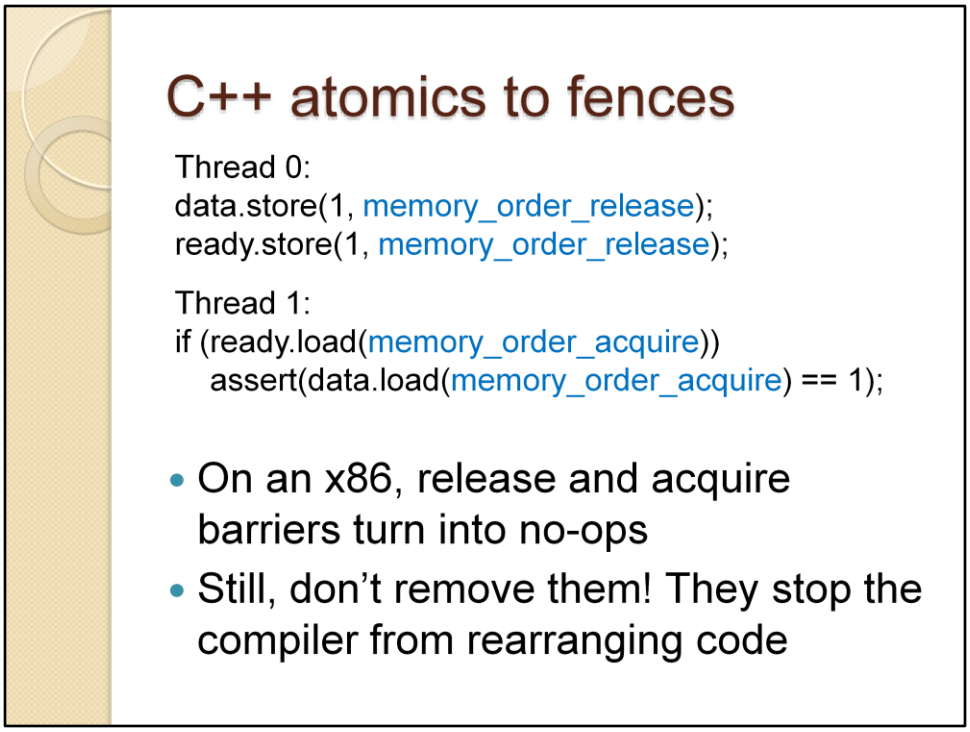
The first two guarantees tell us that the publication pattern will work on the x86 without any modifications (no fences necessary). The x86 spec gives examples for all those guarantees.



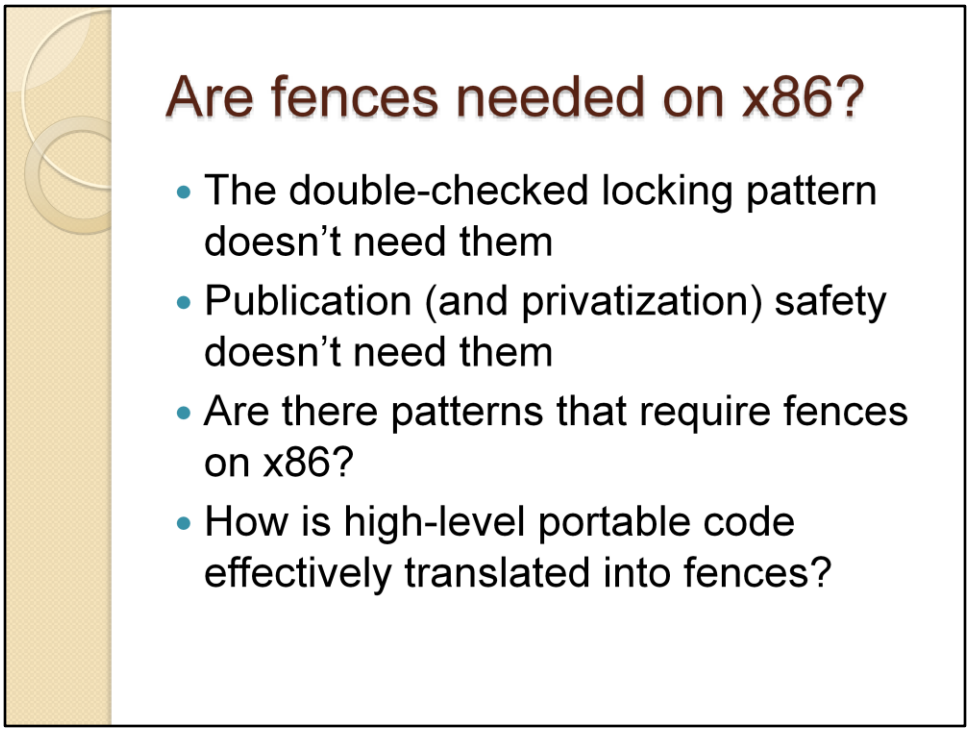
The first non-guarantee is very interesting and we're going to explore it. In practice, lfence and sfence are not used in regular programming because of the existing guarantees. They might be needed in systems programming, when dealing with different types of memory.



Three of the four types of barriers turn into no-ops on an x86. The only cookbook rule that remains is the use of StoreLoad barrier after every volatile store. This code might look over-fenced, but Java has to enforce sequential consistency, which is a stronger requirement than release/acquire semantics needed for this pattern to work.



In C++, sequential consistency ordering can be relaxed to release/acquire ordering (for this particular pattern). Since the x86 guarantees release/acquire ordering, no fences are generated in this case. Even if your code will only run on the x86, don't remove the ordering constrains!



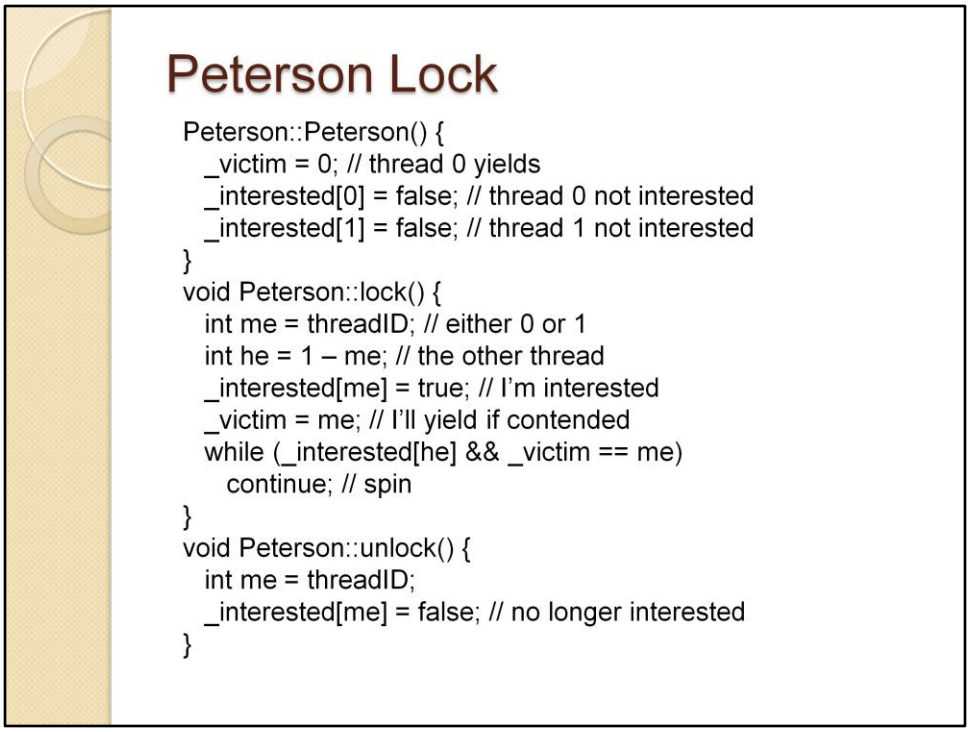
There must be algorithms that require fences even on the x86. I set myself on the quest to find one and see how this need can be expressed at high level and low level.

X86 tricky case: Peterson Lock

- An actual algorithm that won't work without fences on x86
  - Mutual exclusion for two threads
  - Thread local variable threadID, can be 0 or 1. Used as index into _interested

```
class Peterson {
private:
  bool _interested[2];
  int _victim; // who's yielding?
public:
  Peterson();
  void lock();
  void unlock();
};
```

Dekker's algorithm is one such algorithm. Peterson lock is a more modern version of it. This mutual exclusion algorithm works only for two threads. It can be useful for synchronizing exactly two threads.



The constructor does initialization.
When thread 0 tries to acquire the lock, it signals its intent and politely offers to be the one to yield in case of contention (the _victim variable). As long as the other thread is interested and I'm the victim, I'll spin. As soon as the other thread clears its _interested slot, I'm free to enter the critical section. At exit, I reset my _interested slot, so the other thread may continue.

The gist of Peterson Lock

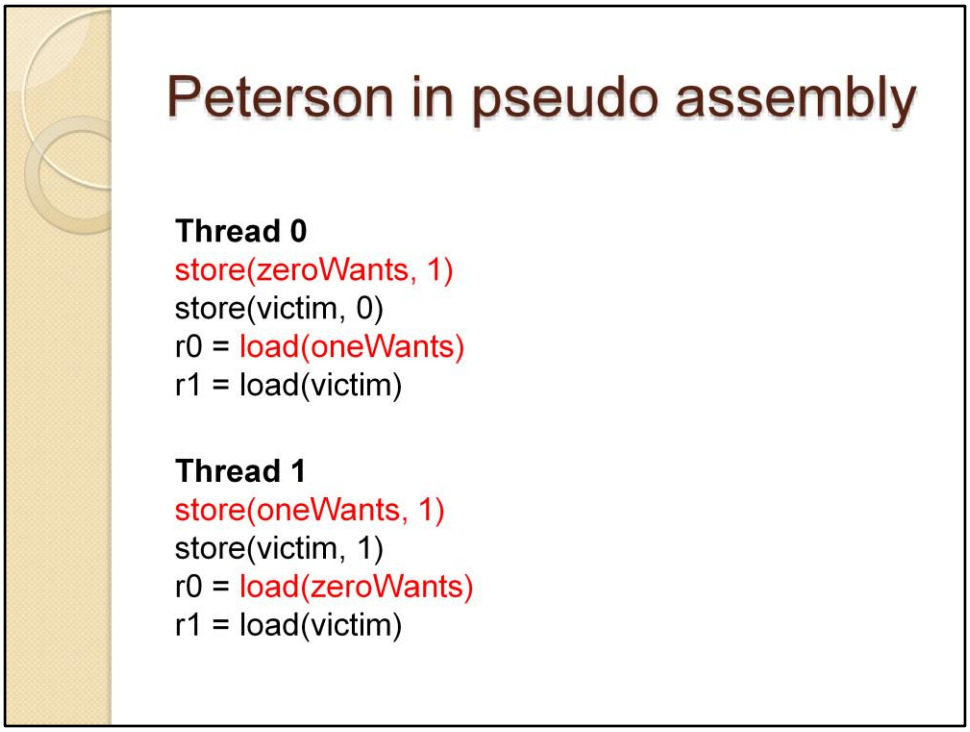
Thread 0

```
zeroWants = true;
victim = 0;
while (oneWants && victim == 0)
    continue;
// critical code
zeroWants = false;
```

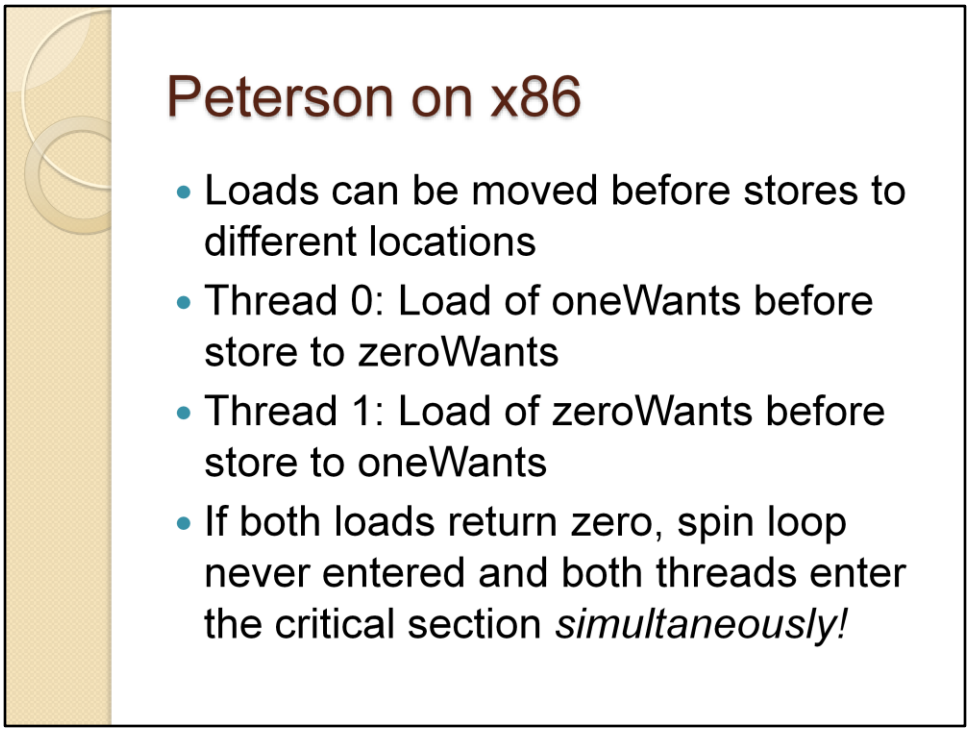
Thread 1

```
oneWants = true;
victim = 1;
while (zeroWants && victim == 1)
    continue;
// critical code
oneWants = false;
```

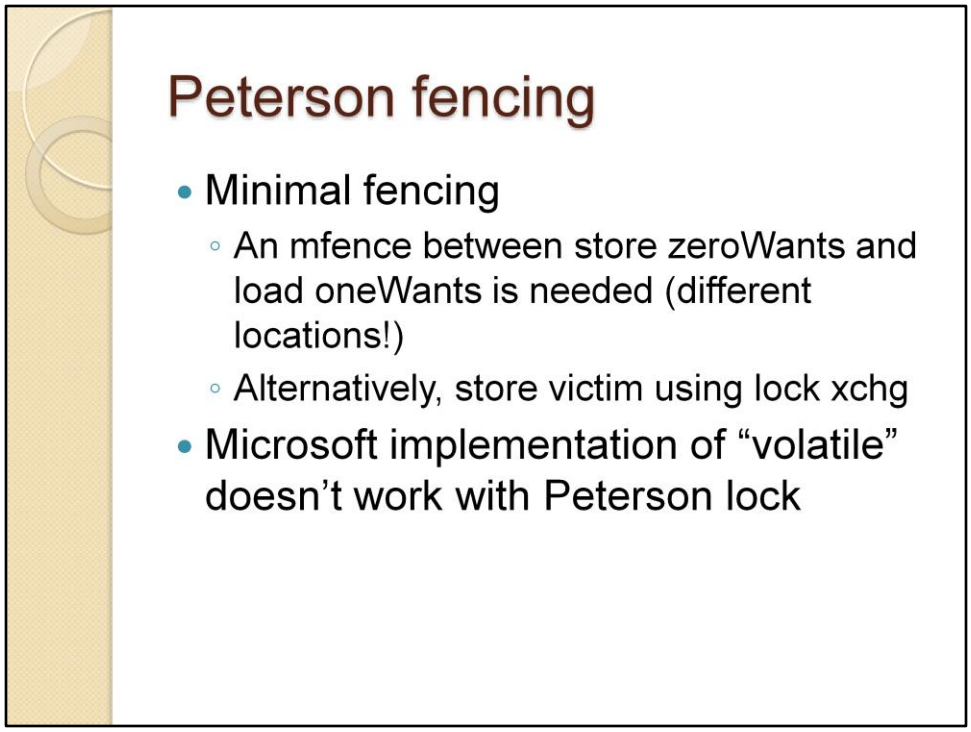
I'll go through some simplification stages, to show where exactly the fencing is necessary. Here, the array _interested was replaced by two variables zeroWants and oneWants. The two threads have their ID hardcoded. Thread 1 executes the mirror image of the code of Thread 0.



These are just the operations performed when threads enter the lock. The load of victim is actually predicated on the value of oneWants. Notice that the spin loop is skipped when the appropriate “wants” variable is zero. Can both, zeroWants and oneWants be zero?



It turns out that there is a possible interleaving on the x86 that will lead to both variables being zero.



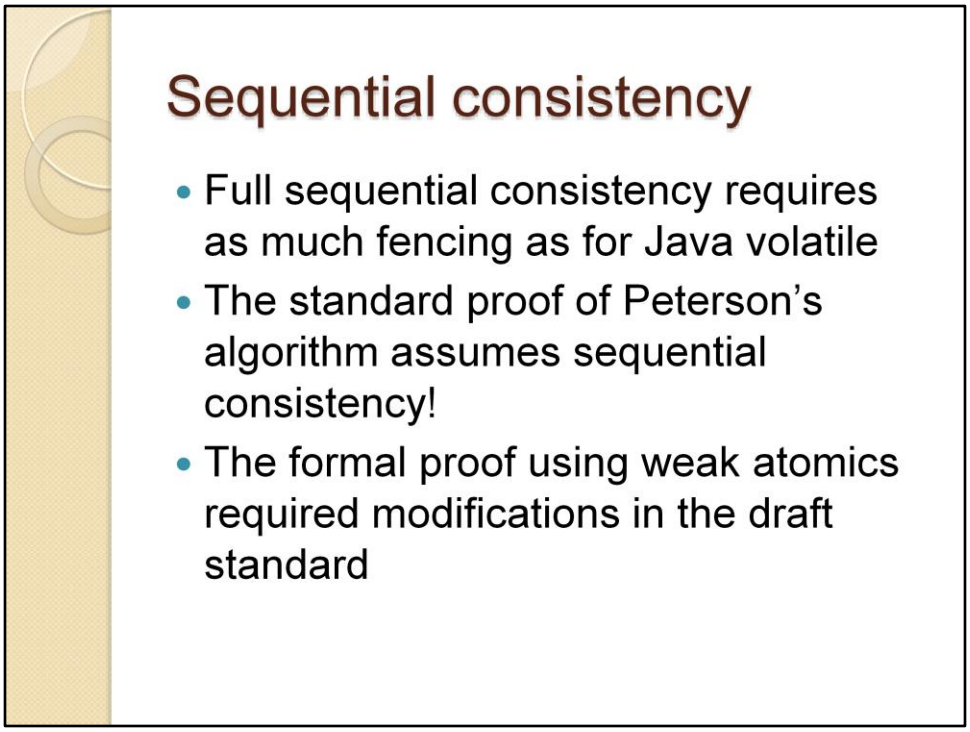
Peterson lock will not work on the x86 without an actual fence.
Note: Vendor-specific extension of C++ volatile don’t produce fences on the x86 (at least not on Microsoft compiler), so Peterson lock is broken pre C++0x.

C++ ordering for Peterson

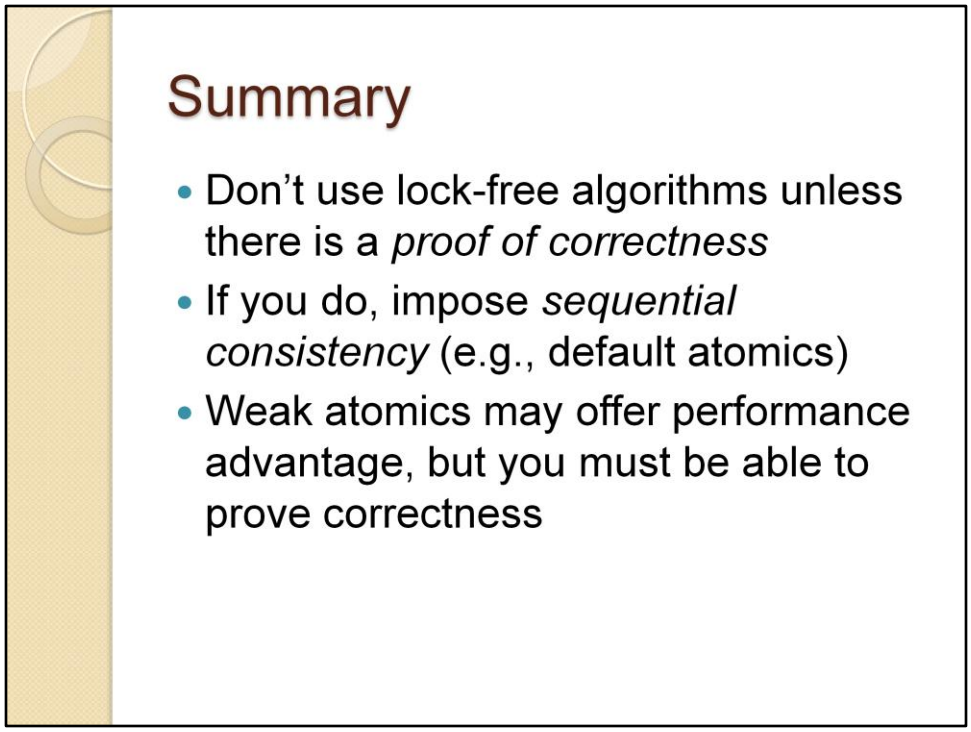
```
void Peterson::lock() {
  int me = threadID; // either 0 or 1
  int he = 1 – me; // the other thread
  _interested[me].store(true, memory_order_relaxed);
  _victim.exchange(me, memory_order_acq_rel);
  while (_interested[he].load(memory_order_acquire)
      && _victim.load(memory_order_relaxed) == me)
    continue; // spin
}
```

Produces minimal fencing on x86 (one mfence or lock xchg)

This is (almost) minimal ordering required by Peterson lock. It should work for every processor supported by C++0x compilers. On the x86 it will produce the lock xchg instruction that makes Peterson lock work. Writing this kind of code requires understanding of various processor memory models and the theory behind weak atomics. The need for memory_order_acq_rel is not at all obvious. (I would have never guessed it if I didn't go into the low-level x86 analysis.)



The safe approach is to not relax sequential consistency. Only if performance requirement reach the level of panic, should one think of relaxing it. And that requires rewriting of the proof of correctness of the Peterson's algorithm. Not to be taken lightly! I don't have such a proof, so don't quote me.



The biggest danger is when programmers don't recognize that they are doing lock-free programming. Any variable that is shared between threads must be synchronized one way or another—either through locking or the use of atomics.

Bibliography

- Bartosz Milewski's Programming Blog
- Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming (Sequential consistency, Peterson lock)
- Doug Lea, The JSR-133 Cookbook for Compiler Writers (Java barriers)
- Intel 64 Architecture Memory Ordering White Paper
- AMD64 Architecture Programmer's Manual
- Scott Meyers and Andrei Alexandrescu, Double-Checked Locking Pattern