



# Things You Never Wanted to Know about Memory Fences

But were afraid would be explained to you anyway

For instance by  
**Bartosz Milewski**

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.

## Plan of talk

- Purpose: to scare away from lock-free programming and weak atomics
- Example: Publication safety
- Memory barriers: introduction
- High level abstractions: Java, C++
- Low level x86 memory model
- Peterson lock on x86
  - Need for fencing
  - Why sequential consistency helps

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

## Publication Safety

- General model for the double-checked locking pattern
  - Thread 0
    - Prepare data (construct an object)
    - Publish it (update shared pointer to object)
  - Thread 1
    - Test if data was published (test shared pointer for null)
    - If so, read the data
- Can we guarantee that if the test succeeds, the data will be valid?

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.

## Release Barriers

```
Thread 0
  data = 1
  release barrier
  ready = 1
Thread 1
  r1 = ready
  acquire barrier
  if r1 == 1
    r2 = data
```

```
Without the release
barrier:
Thread 0
  ready = 1
Thread 1
  r1 = ready // 1
  r2 = x // 0
Thread 0
  data = 1
```

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.

## Acquire Barriers

Thread 0  
data = 1  
release barrier  
ready = 1

Thread 1  
r1 = ready  
acquire barrier  
if r1 == 1  
r2 = data

Without the acquire barrier:

Thread 1  
Speculates that ready == 1  
r2 = data // 0

Thread 0  
data = 1  
ready = 1

Thread 1  
Confirms ready == 1  
Keeps r2 // 0

Without the acquire barrier, the reads may 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.

## Higher level languages: Java

- Volatile takes care of memory barriers

```
volatile bool ready = false;  
volatile int data = 0;
```

```
Thread 0:  
data = 1;  
ready = true; // publish data
```

```
Thread 1:  
if (ready) // was data published  
    assert(data == 1);
```

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 memory model

- Sequential consistency
  - For every program execution all volatile reads and writes appear to be done in some global order
  - Every processor sees the same order
- No out-of-thin-air values
  - Certain types of non-causal speculations are disallowed

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

## Java bytecode

- Compiler issues barriers
  - Barriers go between volatile accesses
    - LoadLoad between reads
    - StoreStore between writes
    - LoadStore between read and write
    - StoreLoad between write and read
- Cookbook rules
  - StoreStore *before* each volatile store
  - StoreLoad *after* each volatile store
  - LoadLoad *and* LoadStore *after* each volatile load

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.



## Java barriers

- Compiler optimizes away a lot of barriers (dataflow analysis)
- Runtime (or native compiler) translates them into appropriate processor-dependent fences (or locked instructions)
- Many can be eliminated

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

## Higher level languages: C++

- Strong atomics are sequentially consistent, similar to Java volatile

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

```
Thread 0:  
x.store(1);  
ready.store(1);
```

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

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.

## From high level to low level

- Pick a particular processor: x86
  - The x86 memory model
- How does Java volatile translate into x86 fences?
  - Sequential consistency
- How do C++ atomics translate?
  - Default: sequential consistency
  - Specific weak memory orderings

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

## x86 memory model guarantees

- Loads are *not* reordered with other loads.
- Stores are *not* reordered with other stores.
- Stores are *not* reordered with older loads.
- In a multiprocessor system, memory ordering *obeys causality* (memory ordering respects transitive visibility).
- In a multiprocessor system, stores to the same location have a *total order* (important!)
- In a multiprocessor system, locked instructions have a *total order*. (sequential consistency!)
- Loads and stores are *not* reordered with locked instructions.

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.

## x86 memory non-guarantees

- Loads can be moved before stores to different locations
- Intra-processor forwarding allowed
- No sequential consistency between writes to different locations
- Ordering may be enforced by fences (*lfence*, *sfence*, *mfence*) or locked instructions (e.g., *lock xchg*)

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.

## Java barriers to fences on x86

- ~~StoreStore~~ before each volatile store
- **StoreLoad** after each volatile store
  - Turns into *mfence* (actually *lock xchg*)
- ~~LoadLoad and LoadStore~~ after each volatile load

volatile data, ready

Thread 0

data = 1

ready = 1

Thread 1

r1 = ready

if r1 == 1

r2 = data

Thread 0

data = 1

*mfence* /store using *xchg*

ready = 1

*mfence*/store using *xchg*

Thread 1

r1 = ready

if r1 == 1

r2 = data

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.

## C++ atomics to fences

Thread 0:

```
data.store(1, memory_order_release);  
ready.store(1, memory_order_release);
```

Thread 1:

```
if (ready.load(memory_order_acquire))  
    assert(data.load(memory_order_acquire) == 1);
```

- On an x86, release and acquire barriers turn into no-ops
- Still, don't remove them! They stop the compiler from rearranging code

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 constraints!



## Are fences needed on x86?

- The double-checked locking pattern doesn't need them
- Publication (and privatization) safety doesn't need them
- Are there patterns that require fences on x86?
- How is high-level portable code effectively translated into fences?

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.

## Peterson Lock

```
Peterson::Peterson() {
    _victim = 0; // thread 0 yields
    _interested[0] = false; // thread 0 not interested
    _interested[1] = false; // thread 1 not interested
}
void Peterson::lock() {
    int me = threadID; // either 0 or 1
    int he = 1 - me; // the other thread
    _interested[me] = true; // I'm interested
    _victim = me; // I'll yield if contended
    while (_interested[he] && _victim == me)
        continue; // spin
}
void Peterson::unlock() {
    int me = threadID;
    _interested[me] = false; // no longer interested
}
```

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 `_interested` array 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.

## Peterson in pseudo assembly

### Thread 0

```
store(zeroWants, 1)
store(victim, 0)
r0 = load(oneWants)
r1 = load(victim)
```

### Thread 1

```
store(oneWants, 1)
store(victim, 1)
r0 = load(zeroWants)
r1 = load(victim)
```

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?

## Peterson on x86

- Loads can be moved before stores to different locations
- Thread 0: Load of oneWants before store to zeroWants
- Thread 1: Load of zeroWants before store to oneWants
- If both loads return zero, spin loop never entered and both threads enter the critical section *simultaneously!*

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

## Peterson fencing

- Minimal fencing
  - An mfence between store zeroWants and load oneWants is needed (different locations!)
  - Alternatively, store victim using lock xchg
- Microsoft implementation of “volatile” doesn’t work with Peterson lock

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.)



## Sequential consistency

- Full sequential consistency requires as much fencing as for Java volatile
- The standard proof of Peterson's algorithm assumes sequential consistency!
- The formal proof using weak atomics required modifications in the draft standard

The safe approach is to not relax sequential consistency. Only if performance requirements 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.

## Summary

- Don't use lock-free algorithms unless there is a *proof of correctness*
- If you do, impose *sequential consistency* (e.g., default atomics)
- Weak atomics may offer performance advantage, but you must be able to prove correctness

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](#)