

Understanding OrderAccess

Managing Data Races in a Hostile Environment

David Holmes
Consulting Member of Technical Staff
JVM Runtime Group

Version 1.1

Java
Your
Next
(Cloud)



Disclaimer

- This is not an academically rigorous discussion of memory models
 - In particular, terminology may be “loose” and differ from other sources
 - Will avoid extreme subjects, like causality, or things which allow/require time-travel to explain them
- This is an engineering overview for practicing software developers
 - I am not an expert on theoretical memory models or specific machine architectures
- There may be judicious use of “hand waving” and (over-)simplification

Program Agenda

- 1 Memory Models
- 2 Acquire/Release
- 3 Fences and Barriers
- 4 OrderAccess
- 5 Atomics and Memory Ordering

Program Agenda

- 1 Memory Models
- 2 Acquire/Release
- 3 Fences and Barriers
- 4 OrderAccess
- 5 Atomics and Memory Ordering

Introduction to Data Races

- Paradigm: Shared-memory based multi-threading
 - Threads communicate by modifying locations (variables) in shared memory
- Concurrent operations on a variable can **conflict** e.g. `x++`;
 - Need to provide ways to perform atomic actions with no conflict
 - Locking, atomic ops
- If two or more threads access a variable concurrently and at least one access is a write then we have a **data race**
 - Outcome depends on order of execution: it's a race
- Locking prevents concurrent access through mutual exclusion
- Programs that always use locks are data-race free (DRF)




Sequential Consistency

- If the observed memory state matches some interleaving of all the actions executing in every thread then we have a **sequentially consistent** execution
 - You can reason about the observed result by looking at the sequential code executed by each thread
- Actual executions with modern compilers and modern hardware are not sequentially consistent
 - Why not? Performance, performance, performance!
- Data-Race Free programs may be guaranteed to be sequentially consistent
 - Java, C++11

Order! Order!

- Source order – how you write the code
 - Naïve expectation that everything happens exactly as written
- Program order – generated machine code presented to the CPU
 - “Any resemblance to source order is purely coincidental”
 - It’s all about performance! As long as you can’t tell the difference
- Execution order – how the hardware actually executes the code
 - Speculative execution, instruction reordering, caching, pipeline stalls
 - It’s all about performance! As long as you can’t tell the difference
- Observation order – how things appear to happen for a given observer
 - Different observers can see things happen in a different order

Concurrent Programming is like Physics

- Data-race free programming
 - Sequential; or
 - Concurrency with locks
 - “Simple” Lock-free programming
 - Controlled data-races
 - Well-defined “synchronization” actions
 - “Relaxed” Lock-free programming
 - Uncontrolled data-races
 - “heuristic” algorithms
- 
- Newtonian mechanics
 - Easy enough to understand
 - Good enough for most purposes
 - Quantum mechanics
 - Hard to understand or reason about
 - Sometimes a necessary evil
 - Special Theory of Relativity
 - You are on your own here!
- 
- 

Memory Models

- In simplistic terms a “memory model” defines the allowed ordering of memory accesses based on how specific constructs are used
- Hardware/Architectural Memory Model
 - Defines allowed reordering of memory accesses (few: TSO; many: RMO)
 - Uses specialized instructions to enforce order: barriers, fences, acquire/release
- Java Memory Model
 - Defines program actions, synchronization actions and the happens-before ordering
 - Uses language constructs: volatile variables, synchronized blocks
- C++ Memory model (as in hotspot today)
 - No language constructs: uses OrderAccess to instruct both compiler and hardware

Data Race Example: The importance of ordering

```
int data = -1;  
bool dataReady = false;
```

```
// Thread 1
```

```
data = produce();  
dataReady = true;
```

```
// Thread 2
```

```
if (dataReady) {  
    assert(data != -1);  
    consume(data);  
}
```

- Total Store Ordering (TSO) – seems good!
 - But what about the compiler? What about load reordering?
- Relaxed Memory ordering (RMO) – not good on all fronts
- Required ordering has to be enforced using the tools provided by the memory model

Program Agenda

- 1 Memory Models
- 2 Acquire/Release**
- 3 Fences and Barriers
- 4 OrderAccess
- 5 Atomics and Memory Ordering

Acquire/Release Memory Ordering Semantics

- Terminology comes from requirements for lock-based synchronization
 - “A thread that **acquires** the lock, must see all stores that occurred before the previous owner **released** the lock”
 - The **acquire** is associated with the **load** that sees the lock is free: hence: **load_acquire**
 - The **release** is associated with the **store** that marks the lock free: hence: **release_store**
 - Unfortunately also referred to as store-release by some
 - Implicit constraint: no loads/stores in the locked region can move out of it!
 - But others can move in! (“roach motel”)
- Generalization: If a **load_acquire** sees the value written by a **release_store** then all values written before that **release_store** are also visible
 - **load_acquire/release_store** should **always** operate as matching pairs

Data Race Example: Using Acquire/Release

```
int data = -1;  
bool dataReady = false;
```

```
// Thread 1
```

```
data = produce();  
release_store(&dataReady, true);
```

```
// Thread 2
```

```
if (load_acquire(&dataReady)) {  
    assert(data != -1);  
    consume(data);  
}
```

- Assertion now guaranteed not to fail
- Internally load_acquire/release_store have to provide:
 - Compiler barrier
 - Hardware barrier

Architectural Support for Acquire/Release

- Itanium (IA64)
 - ld.acq, st.rel
 - Original reason for introducing OrderAccess!
- ARMv8 (Aarch64)
 - LD A_x : load with acquire semantics
 - ST L_x : store with release semantics
 - Referred to as load-acquire and store-release
 - Semantics are expressed in terms of relation of other loads/stores to the flagged load/store

Program Agenda

- 1 Memory Models
- 2 Acquire/Release
- 3 Fences and Barriers**
- 4 OrderAccess
- 5 Atomics and Memory Ordering

Fences and Barriers: what's in a name?

- Fence, barrier, memory barrier, membar
 - Used loosely and often interchangeably
 - They prevent certain kinds of code motion and so allow explicit expression of ordering constraints
 - Specific orderings imply visibility/observability of stores
- Fine-grained barriers:
 - **XY**: all **X** preceding the barrier must complete before all **Y** following it
 - I.e. **storeLoad, storeStore, loadLoad, loadStore**
- Coarse-grained:
 - “**fence**”: All four fine-grained barriers combined gives “full bi-directional **fence**”

Data Race Example: Using Barriers

```
int data = -1;  
bool dataReady = false;
```

```
// Thread 1  
data = produce();  
storeStore();  
dataReady = true;
```

```
// Thread 2  
if (dataReady) {  
    loadLoad();  
    assert(data != -1);  
    consume(data);  
}
```

- Assertion now guaranteed not to fail
- Internally the barriers have to provide:
 - Compiler barrier
 - Hardware barrier

Architectural Support for Barriers

- Details are extremely complicated!
 - Can be affected by type of memory, system configuration, caching policies etc.
 - OS can further constrain possibilities by how it configures the hardware
- SPARC
 - Membar loadLoad | loadStore | storeLoad | storeStore
 - Hmmm – isn't SPARC TSO??
 - The architecture can be TSO or RMO. Solaris executes in TSO mode. Ultra3 only supports TSO.
 - Store barriers ensure visibility of store before barrier completes
- X86
 - mfence

Architectural Support for Barriers (cont)

- ARM

- Data Memory Barrier (DMB): orders all memory accesses (fence)
 - ARMv7+ adds “dmb st”: acts like storeStore
 - ARMv8 adds “dmb ld”: acts like loadStore | loadLoad
- Data Synchronization barrier (DSB): orders all memory accesses and the instruction stream
 - Heavyweight: used by OS when need to synchronize i-cache and d-cache

- Power/PPC

- Heavy-weight sync (hwsync or sync): orders instructions and all memory accesses
- Light-weight sync (lwsync): orders instructions and some memory accesses
 - Specifically it does not provide storeLoad barrier

Implicit (partial) Barriers

- Data dependencies, control dependencies, address dependencies
 - Details differ from architecture to architecture!
 - Only guarantees order of the accesses with the dependencies!
 - Most often used to elide real barriers to get acquire semantics
 - Can introduce artificial dependencies to get desired affect e.g. address dependency on ARMv8

```
if (dataReady) { // pseudo-code: needs to happen at asm level
    int* data_addr = &data + (dataReady & 0); // fake dependency
    y = *data_addr; // can't be reordered with load of dataReady
```
- X86 locked instructions act as storeLoad barrier
- Relying on implicit barriers in shared code requires detailed analysis!
 - **Beware!** These are hardware level conditions – the compiler may have already reordered things!

Program Agenda

- 1 Memory Models
- 2 Acquire/Release
- 3 Fences and Barriers
- 4 OrderAccess**
- 5 Atomics and Memory Ordering

OrderAccess Background

- Long history
 - Introduced as part of the IA64 port
- Mixes acquire/release semantics and barrier APIs
- Some problems with semantics over time
- Had a very recent clean up of semantics and implementation
 - 7143664: Clean up OrderAccess implementations and usage
 - Thanks to Erik Österlund

OrderAccess API

- Defines a model that allows mapping from acquire/release to barrier APIs
 - Simplifies implementation on platforms that only support one form
 - Results in more constrained behaviour in some cases
- Defines simple barriers:
 - `loadload()`, `loadstore()`, `storeload()`, `storestore()`
- Defines a full bi-directional barrier:
 - `fence()`
- Defines bound acquire/release:
 - `load_acquire()`, `release_store()`

Order Access API (cont)

- Defines unbound acquire/release in terms of barriers
 - **acquire()** == loadLoad|loadStore
 - **release()** == loadStore|storeStore
 - Allows bound forms to be implemented using unbound forms (default)
 - `release_store(&x, 1) → release(); x = 1;`
 - `y = load_acquire(&x) → y = x; acquire();`
 - Unbound forms should still be associated with specific loads and stores
 - Unbound forms exist for where we can't access raw variables directly e.g. accessors
 - But we are addressing that by defining accessors/setters with acquire/release semantics
 - Unbound forms may result in less efficient hardware level barriers
- Defines composite **release_store_fence()** for convenience & efficiency
 - `release_store_fence(&x, 1) → release(); x = 1; fence();`

OrderAccess Implementation Notes

- OS+CPU specific implementations: linux_x86, solaris_sparc, linux_arm etc
- Abstract barriers etc. mapped to:
 - Hardware instructions using inline assembly that also defines a “compiler barrier”
 - Compiler “barrier” prevents any reordering of statements around it
 - E.g. gcc: `void compiler_barrier() { __asm__ volatile ("" : : : "memory"); }`
- On strongly ordered systems many barriers are no-ops at hardware level
 - E.g. only **storeLoad** needs to be explicitly defined on x86 and SPARC
 - Beware of loose terminology: “need fence aka storeLoad” means “need fence so add in storeLoad”!
 - On many platforms **storeLoad** implementation subsumes all other barriers
 - Don’t confuse implementation with semantics!

OrderAccess Programming Guidelines for Shared Code

- Use locks where possible and keep things Data Race Free!
- Partially lock-free data structures need barriers in locked & unlocked code
 - Prefer acquire/release APIs over barrier APIs – shows link between reader and writer!
 - But only need barriers between accesses – so no `load_acquire(x)` to assert `x != NULL`
- Ensure you can identify lock-free code and understand protocols involved
 - Need to understand the big picture – can't just look at methods in isolation!
 - Understand if/where safepoints may be involved
 - If `data / dataReady` are set at a safepoint and only used by `JavaThreads` no barriers are needed
- Always think/code in terms of the most relaxed memory model possible
 - But unnecessary barriers cause confusion when reasoning about code!

Example: Monitor code recent bug fix (8166197)

– IUnlock: (slightly modified)

```
if ( _EntryList != NULL) { // unlink head
    ParkEvent * const w = _EntryList;
    _EntryList = w->ListNext;
    OrderAccess::release_store_ptr(&_OnDeck, w);
}
```

– ILock:

```
while (OrderAccess::load_ptr_acquire(&_OnDeck) != ESelf) {
    ParkCommon(ESelf, 0);
}
```

- When “w” becomes “onDeck” it must not find itself in _Entrylist!
 - NOTE: the load of _EntryList by “w” can’t even be seen locally in this method! It’s in IUnlock!

– IUnlock:

```
ParkEvent * const w = _OnDeck; // no load_acquire as we don't access
if (w != NULL)                // any other state in the monitor
    return;
```

Wait a Minute!!!

- First you said:
 - “Note the load of `_EntryList` can’t be seen locally” but we need a load-acquire
- Then you said:
 - “no need for a load-acquire as no other accesses to monitor state”
- But what if that access happens later like the first case?
- This is where you need to understand the protocols involved in the code
 - In the second case we leave the Monitor code completely and we will not access `_EntryList` again until we acquire a Monitor, execute our critical section and then start to unlock the Monitor!
 - In that code path we encounter numerous synchronization points

Wait a Minute 2: Efficiency vs. Correctness

- If the `load_acquire` guards against load reordering; and
- The load of `_onDeck` and the load of `_EntryList` are “miles apart”; then
- Surely we don’t need the `load_acquire` as they will never be reordered?
- That could well be true on current platforms*, but:
 - How far apart is far enough apart to avoid reordering?
 - How do you capture the fact they must remain “far enough apart”?
- That said:
 - Iff it were established that such a barrier was a serious performance bottleneck then we might relax it for that platform
 - With suitable commentary etc

*Imagine an architecture with software cache coherency that had to explicitly pull updates from main memory

Program Agenda with Highlight

- 1 Memory Models
- 2 Acquire/Release
- 3 Fences and Barriers
- 4 OrderAccess
- 5** **Atomics and Memory Ordering**

Atomic API

- Atomic:: **load, store, add, xchg, cmpxchg** ...
- Spec: every **read-modify-write** operation acts as a **full bi-directional fence**
 - i.e. no accesses are allowed to be reordered across such an atomic operation
 - Doesn't mean each operation must be implemented as: `fence(); op(); fence()` !
 - Atomic loads and stores require no memory ordering properties
- C++11 Atomics offer varying memory ordering semantics for operations
 - **memory_order_relaxed/consume/acquire/release/acq_rel/seq_cst**
 - Memory order kind gets passed as additional parameter to atomic ops
 - Default: `memory_order_seq_cst`

JDK-8155949: Support relaxed semantics in cmpxchg

- A first step towards compatibility with, and possible use of, C++11
- ```
intptr_t Atomic::cmpxchg_ptr(intptr_t exchange_value,
 volatile intptr_t* dest,
 intptr_t compare_value,
 cmpxchg_memory_order order);
```
- ```
enum cmpxchg_memory_order {  
    memory_order_relaxed,  
    // Use value which doesn't interfere with C++2011. We need to be more  
    conservative.  
    memory_order_conservative = 8  
};
```
- Default is **memory_order_conservative**
 - At time of writing all platforms support only the default and keep the full fence

JDK-8154736: enhancement of cmpxchg and copy_to_survivor for ppc64

- A cautionary tale!
- Any kind of relaxed ordering semantics are very hard to reason about!
- See review thread:
 - <http://mail.openjdk.java.net/pipermail/hotspot-runtime-dev/2016-October/021452.html>
- Suggests more work needed to consider role of dependent loads
 - And some means to clearly document them e.g. Linux kernel uses
 - `Q = READ_ONCE(P); smp_read_barrier_depends(); D = READ_ONCE(*Q);`
 - <http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/plain/Documentation/memory-barriers.txt?id=HEAD>

Final word: C/C++ volatile

- Exact meaning of **volatile** is compiler-specific
- Generally any variable modified concurrently and accessed lock-free should be declared **volatile**
 - At a minimum prevents optimisations like loop hoisting and general register caching
- Some compilers ensure volatile accesses maintain order w.r.t. other volatile accesses
 - But not necessarily non-volatile accesses
 - Though MSVC defines `load_acquire/release_store` semantics!
- TL;dr: Don't depend on C/C++ volatile for ordering!

References

- The JSR-133 Cookbook for Compiler Writers
 - <http://g.oswego.edu/dl/jmm/cookbook.html>
- A Tutorial Introduction to the ARM and POWER Relaxed Memory Models
 - <http://www.cl.cam.ac.uk/~pes20/ppc-supplemental/test7.pdf>
- C++11 std::memory_order
 - http://en.cppreference.com/w/cpp/atomic/memory_order



JavaYourNext (Cloud)