The Java Memory Model

The meaning of concurrency in Java

Bartosz Milewski
Plan of the talk

- Motivating example
- Sequential consistency
- Data races
- The DRF guarantee
- Causality
- Out-of-thin-air guarantee
- Implementation (compiler and JVM)
Double-Checked Locking Pattern

class Singleton {
    private volatile Foo foo = null;
    public Foo getFoo() {
        if (foo == null) {
            synchronized(this) {  // lock
                if (foo == null)  // double check
                    foo = new Foo();
            }
        }
        return foo;
    }
}
Two-step new

class Foo {
    public Bar bar;
    public Foo() {
        bar = new Bar();
    }
}

• new Foo() can be split into:
  – Memory allocation (initialization of foo)
  – Constructor call (initialization of bar)
What can go wrong (without volatile)

• Thread 1 constructs Foo
  – Write to foo
  – Write to foo.bar in the constructor
• Two writes are scheduled to go to memory
• Thread 2 sees the first write
  – Returns foo
  – Tries to access foo.bar
• No guarantee that the write to bar is committed by that time: null exception!
Why could it go wrong?

• Multiprocessor optimization
  – Memory access is expensive. Caching helps, but cache coherency may be expensive too, hence
  – Relaxed processor memory models: reads and writes may be reordered/delayed/guessed in various ways

• Compiler/runtime optimization
  – Writes to different variables may be rearranged
  – This is why the DCLP didn’t work in C++, even on the x86
What do we want?

• We need a way to:
  – Tell the compiler/JVM to suspend some optimizations
  – Tell the processor to synchronize reads and writes at selected points
    • Every multiprocessor has special synchronizing instructions (fences and locked instructions)

• Present a reasonably simple (read: unlike C++) model for the programmer (sequential consistency)
Programmer’s view

• The most natural model: *Sequential Consistency*
  – Threads execute as if there was a single processor
    (good old preemptive multitasking).

• However, full sequential consistency defeats multicore optimizations--Huge performance hit!

• Realistic approach: let the programmer specify which actions must be sequentially consistent
Sequential Consistency (SC)

• Each thread/processor executes instructions in program order
• Instructions performed by different processors are arbitrarily interleaved. But all processors see the same interleaving. There is a global order.
• Each instruction is atomic and instantaneous wrt. other processors
• Conceptual switch connects one processor at a time to memory
• A read sees the result of the last write to the same location (in the particular interleaving)
Enforcing the SC illusion

• This is *not* how the programs are executed in reality!
• Compiler/JVM must enforce this illusion (even though modern processors don’t) for a reasonable subset of programs.
• Q: What subset? A: The *correctly synchronized* subset
What is a data race?

• Data **conflict**
  – Two threads access the same memory location
  – At least one access is a write

• Data **race**
  – There is no intervening *synchronization action* between two *conflicting* accesses
  – More precisely, there is no *happens before* relationship between them
Data race freedom (DRF)

- DRF guarantee: **Data race free programs** (a.k.a., *correctly synchronized*) **have sequentially consistent semantics**
- Java Memory Model provides DRF guarantee
- The proof that a program is data race free is somewhat circular.
  - Start with all possible sequentially consistent interleavings (a.k.a., SC executions)
  - Establish all happens-before relationships (next slide)
  - Show there are no conflicts that are not ordered by happens-before
Happens Before relation

• On the same processor, A happens before B if A is earlier in *program order*
• Between processors, *happens-before* defined by *synchronizing actions*
• Write to a volatile variable happens before all subsequent reads of that variable—in a *particular execution*
• Unlocking of a monitor happens before the subsequent locking of the same monitor
• Happens-before is a transitive closure of program order and synchronization order
Data races are bad for you!

- Data races are bugs (except in some dark corners of libraries)!
- Data races break sequential consistency
- Language doesn’t have to specify the behavior of buggy programs (C++ doesn’t!), but
  - The presence of data races may be used to maliciously attack an application.
  - We don’t want Java to be vulnerable
Data races and causality loops

Initially $x = y = 0$; $r_1$ and $r_2$ are thread local

$\begin{array}{ll}
\text{Thread 1} & \text{Thread 2} \\
r_1 = x; & r_2 = y; \\
y = r_1; & x = r_2;
\end{array}$

Is the result $r_1 == r_2 == 42$ possible?

- In principle, an out-of-thin-air value is possible in a self-fulfilling prophecy
- Prophecies are called *speculations* and are possible on many processors
The answer is 42

Thread 1     Thread 2
r1 = x;      r2 = y;
y = r1;      x = r2;

• T1 speculates that x may be 42. Later, when x arrives from memory and is not 42, the speculation will be discarded
• T1 sets y to 42
• T2 reads y and sees 42
• T2 sets x to 42
• T1 finally loads x from memory and sees 42. It validates the speculation
Fry is his own grandfather!

- A data race (or a series of such) may be used to pull a value out of the thin air
- A causes B and B causes A
- This behavior could be used to compromise security (speculating a reference to a sensitive object)
- **Java memory model disallows causality loops**
Preventing causality loops

• Start with a *well-behaved* execution in which reads cannot see writes through data races
• Pick a data race and *commit* it (connect reads to writes)
• Restart the execution allowing the committed read to see the committed write
• Loop until all data races committed
Phew! It’s not 42 after all

Thread 1          Thread 2
r1 = x;           r2 = y;
y = r1;           x = r2;

• Let’s commit the race on x
  – The read, r1 = x, cannot see the non-committed write, x = r2
  – It can only see the initial value 0

• Commit the race on y
  – The read, r2 = y, cannot see the non-committed write, y = r1
  – It can only see the initial value 0
From volatile to assembly

• Volatile access must be translated into a processor memory model
• Processors don’t associate synchronization actions with memory locations (variables).
• Memory fences: special instructions inserted between accesses (x86 mfence)
• Locked instructions (x86 xchg)
• Fences and locked instructions are expensive!
Compiler’s role

• Insert memory barriers—special bytecodes between volatile accesses
  – LoadLoad between volatile reads
  – StoreStore between volatile writes
  – LoadStore between a load and a store
  – StoreLoad between a store and a load

• Problem: for a given volatile access, not always possible to figure out previous and next access (even with flow analysis)
Cookbook rules

• Conservative approach:
  – StoreStore *before* volatile write
  – StoreLoad *after* volatile write
  – LoadLoad *and* LoadStore *after* volatile read

• Using flow analysis the compiler may optimize many of those away

• JVM for a particular processor translates those into fences/locked instructions

• On an x86, only StoreLoad is relevant—turns the preceding store into *xchg*
Java Memory Model

• Data-race free programs have sequentially consistent semantics—you can reason about them

• Data races don’t break causality—even a buggy program won’t compromise security
Resources


• http://www.cs.umd.edu/~pugh/java/memory

• Doug Lea’s JSR-133 compiler cookbook: http://gee.cs.oswego.edu/dl/jmm/cookbook. html