Practical Concurrent and Parallel Programming 14.1

Peter Sestoft
IT University of Copenhagen

Friday 2016-12-09*

Plan for today

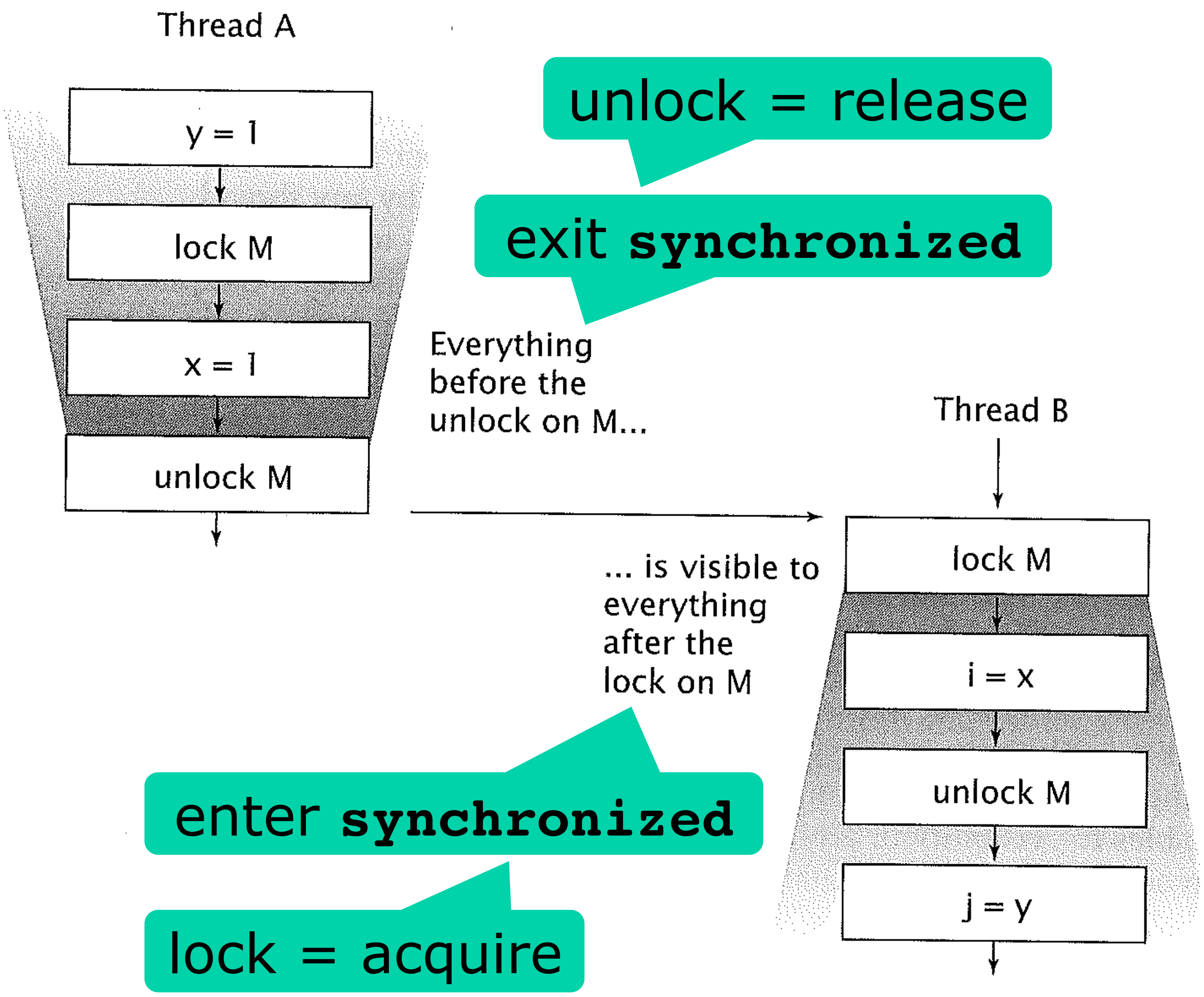
- Part 1 (Peter):
  - The Java Memory Model
  - The C# Memory Model?

- Part 2 (Ken Friis Larsen):
  - Using Rust's type system to control shared mutable memory and avoid some concurrency problems

Why do I need a memory model?

- Threads in Java and C# and C etc *communicate* via shared mutable *memory*
- We need *CPU caches* for speed
  - With caches, write-to-RAM order may seem strange
- We need *compiler optimizations* for speed
  - Compiler optimizations that are harmless in thread A may seem strange from thread B
- Disallowing strangeness gives slow software
  - So we have to live with some strangeness
- A memory model tells *how much* strangeness
- The Java Memory Model is quite well-defined
  - JLS §17.4, Goetz §16, Herlihy & Shavit §3.8

Memory model: Locks cause visibility



Goetz p. 37

The happens-before relation in Java

- A *program order* of a thread t is some total order of the thread's actions that is consistent with the intra-thread semantics of t

- Action x *synchronizes-with* action y is defined as follows:
  - An unlock action on monitor m *synchronizes-with* all subsequent lock actions on m
  - A write to a volatile variable v *synchronizes-with* all subsequent reads of v by any thread
  - An action that starts a thread *synchronizes-with* the first action in the thread it starts
  - The write of the default value (zero, false, or null) to each variable *synchronizes-with* the first action in every thread
  - The final action in a thread T1 *synchronizes-with* any action in another thread T2 that detects that T1 has terminated
  - If thread T1 interrupts thread T2, the interrupt by T1 *synchronizes-with* any point where any other thread (including T2) determines that T2 has been interrupted

- Action x *happens-before* action y, written hb(x,y), is defined like this:
  - If x and y are actions of the same thread and x comes before y in *program order*, then hb(x, y)
  - There is a *happens-before* edge from the end of a constructor of an object to the start of a finalizer for that object
  - If an action x *synchronizes-with* a following action y, then we also have hb(x,y)
  - If hb(x, y) and hb(y, z), then hb(x, z) – that is, hb is transitive

Java Language Specification §17.4

Goetz §16.3.1

Strange but legal behavior in Java

- Java Language Specification (JLS), sect 17.4:
  - Run these code fragments in two threads
  - Distinct shared fields A and B, initially 0
  - Local unshared variables r1, r2

Thread 1

```
r2=A;
B=1;
```

Thread 2

```
r1=B;
A=2;
```

JLS 8 Tables 17.1, 17.5

- What are the possible results?
  - Intuitively, either `r2=A` or `r1=B` is executed first
  - And therefore either r2==0 or r1==0
  - But r1==1 and r2==2 is possible, and legal by JLS
  - “Intuition”, sequential consistency, not guaranteed

Strange result, why legal?

```
r2=A;
B=1;
```

Thread 1

```
r1=B;
A=2;
```

Thread 2

- What are the possible results?
  - Result r1==1 and r2==2 is legal because consistent with *happens-before* relation

```
B=1;
A=2;
r2=A;
r1=B;
```

- Respects program order in thread 1
  - Because fields A and B are distinct
- Same for thread 2
- No synchronization order for the actions
  - Because no locking, no volatile fields

- (Probable cause: hardware cache store buffer)

JLS 8 Tables 17.1, 17.5

Another cause: compiler optimizations

- More comprehensible example from JLS 17.4
  - Assume p, q shared, p==q and p.x==0

```
r1 = p;
r2 = r1.x;
r3 = q;
r4 = r3.x;
r5 = r1.x;
```

Thread A

```
r6 = p;
r6.x = 3;
```

Thread 2

  - Compiler optimization, common subexpr. elimin.:

```
r1 = p;
r2 = r1.x;
r3 = q;
r4 = r3.x;
r5 = r2;
```

NB!

```
r6 = p;
r6.x = 3;
```

(p.x seems to switch from r2=0 to r4=3 and back to r5=0

- Using **`volatile x`** prevents this strangeness
  - But makes code slower (lecture 4) VolatileArray.java

Observing it in practice

```
class StoreBufferExample {
  volatile boolean A = false,
                   B = false;
  int A_Won = 0, B_Won = 0;
  public void ThreadA() {
    A = true;
    if (!B)
      A_Won = 1;
  }
  public void ThreadB() {
    B = true;
    if (!A)
      B_Won = 1;
  }
}
```

Executed on thread A

Executed on thread B

TestStoreBuffer.java

Ostrovsky 2013

- Without **`volatile`**, can get **`A_won = B_won = 1`**
  - Caused by CPU store buffer delay (not by compiler)
  - Memory updates are *not sequentially consistent*
- With **`volatile`**, impossible in Java (but not C#)

Sequential consistency

Herlihy & Shavit p. 52

- Principle 3.4.1: *Method calls should appear to take effect in program order*
  - *Program order* is the *order within a single thread*
- The full execution of a program is an interleaving of each all threads' executions
- A read sees the most recent write before it
- Seems natural
  - And **is** natural – on single-core computers, with no compiler optimizations
  - But not on multicore or with compiler opt.

Interleavings assuming sequentially consistent memory model

Initially: **`A = B = false`** and **`A_Won = B_Won = 0`**

A won:

```
A=true
if (!B)
A_Won=1
B=true
if (!A)
```

```
A=true
if (!B)
B=true
A_Won=1
if (!A)
```

```
A=true
if (!B)
B=true
if (!A)
A_Won=1
```

Nobody won:

```
A=true
B=true
if (!B)
if (!A)
```

```
A=true
B=true
if (!A)
if (!B)
```

B won:

```
B=true
if (!A)
B_Won=1
A=true
if (!B)
```

```
B=true
if (!A)
A=true
B_Won=1
if (!B)
```

```
B=true
if (!A)
A=true
if (!B)
B_Won=1
```

Nobody won:

```
B=true
A=true
if (!A)
if (!B)
```

```
B=true
A=true
if (!B)
if (!A)
```

Experiments on 4-core Intel i7

- Java, without volatile and with volatile:

	A loses	A wins
B loses	0	436649
B wins	550463	12888

	A loses	A wins
B loses	2668	438518
B wins	558814	0

TestStoreBuffer.java

```
if (!B)
B=true
if (!A)
B_Won=1
A=true
A_Won=1
```

Some weird executions??

Not sequentially consistent: seen from thread A, the if (!B) moved before A=true

- On 1-core ARM, double-wins seem impossible

C#/.NET memory model?

- Quite similar to Java
  - *C# Language Specification,* Ecma-334 standard
- But weaknesses and unclarities
  - C# **`readonly`** has no visibility effect unlike **`final`**
  - C# **`volatile`** is weaker than in Java
  - Allowed to lift variable read out of loop?
  - "Read introduction" seems downright horrible!

- If you write concurrent C# programs, read:
  - Ostrovsky: The C# Memory Model in Theory and Practice, MSDN Magazine, December 2012
  - Even though optional in this course

- Visibility effect of C#/.NET **`readonly`** fields not mentioned in C# Language Specification or Ecma-335 CLI Specification (**`initonly`**)
- In fact, no visibility guarantee is intended...

Right. The CLI doesn't give any special status to initonly fields, from a memory ordering/visibility perspective. As with ordinary fields, if they are shared between threads then some sort of fence is needed to ensure consistency. This could be in the form of a volatile write, as Carol suggests, or any of the common synchronization primitives such as releasing a lock, setting an event, etc.

Eric

-----Original Message-----
From: Carol Eidt
Sent: Tuesday, December 4, 2012 10:14 AM
To: Peter Sestoft; Mads Torgersen; Eric Eilebrecht
Cc: Carol Eidt
Subject: RE: Visibility (from other threads) of readonly fields in C#/.NET?

Hi Peter,

I apologize for not responding more quickly to your email. I am adding Eric Eilebrecht to this thread, since he is the CLR's memory ordering expert.

I believe that section I.12.6.4 Optimization addresses this, but one has to read between the lines:

"Conforming implementations of the CLI are free to execute programs using any technology that guarantees, within a single thread of execution, that side-effects and exceptions generated by a thread are visible in the order specified by the CIL. For this purpose only volatile operations (including volatile reads) constitute visible side-effects. (Note that while only volatile operations constitute visible side-effects, volatile operations also affect the visibility of non-volatile references.)"

Where it says " volatile operations also affect the visibility of non-volatile references", this implies (though vaguely) that volatile reads & writes behave as some form of memory fence, though whether it is bi-directional or acquire-release is left unstated. I also believe that the above implies that, in order to achieve the desired visibility of initonly fields, one would have to declare a volatile field that would be written last, effectively "publishing" the other fields.

I certainly wouldn't say that the Java memory model oo much fuss over this - it's just fundamentally tricky!

Carol

Works in Java, dubious in C#

C#/.NET volatile weaker than Java's

```
class StoreBufferExample {
  volatile bool A = false,
                B = false;
  int A_Won = 0, B_Won = 0;
  public void ThreadA() {
    A = true;
    if (!B)
      A_Won = 1;
  }
  public void ThreadB() {
    B = true;
    if (!A)
      B_Won = 1;
  }
}
```

```
public void ThreadA() {
  A = true;
  Thread.MemoryBarrier();
  if (!B)
      A_Won = 1;
}
```

```
public void ThreadB() {
  B = true;
  Thread.MemoryBarrier();
  if (!A)
    B_Won = 1;
}
```

TestStoreBuffer.cs

Ostrovsky 2013

- C#: possible to get **`A_Won = B_Won = 1 !!!`**
  - Even with **`volatile`**
  - To fix in C#, add MemoryBarrier call

Experiments on 4-core Intel i7

- C#/.NET 4.6, without and with volatile:

	A loses	A wins
B loses	600	874916
B wins	108249	16235

	A loses	A wins
B loses	522	912084
B wins	72290	15102

TestStoreBuffer.cs

C# volatile has no effect here!!

- Volatile in C# **not** the same as in Java
- Volatile keyword in C, C++, Java and C# has four different meanings...

C# volatile vs Java volatile

- A read of a volatile field is called a volatile read. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a volatile write. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

C# Language Spec 2006, §17.4.3

- A C# volatile read may move earlier, a volatile write may move later, hence trouble
- Not in Java:

If a programmer protects all accesses to shared data via locks or declares the fields as volatile, she can forget about the Java Memory Model and assume interleaving semantics, that is, Sequential Consistency.

Lochbihler: Making the Java memory model safe, ACM TOPLAS, December 2013

MemoryBarrier() in C#/.NET

> Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.

> MemoryBarrier is required only on multiprocessor systems with weak memory ordering (for example, a system employing multiple Intel Itanium processors).

Dubious claim

System.Threading.Thread.MemoryBarrier API docs 4.5

- But sometimes is needed anyway
  - also on x86, contradicting the API docs ...
- Java does not have MemoryBarrier, because Java **`volatile`** gives good guarantees

This week

- Reading
  - Goetz et al chapter 16
  - Java Language Specification §17.4

- No exercises