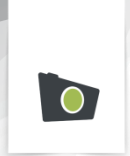
Java Memory Model for practitioners

by Vadym Kazulkin and Rodion Alukhanov, ip.labs GmbH



Topics

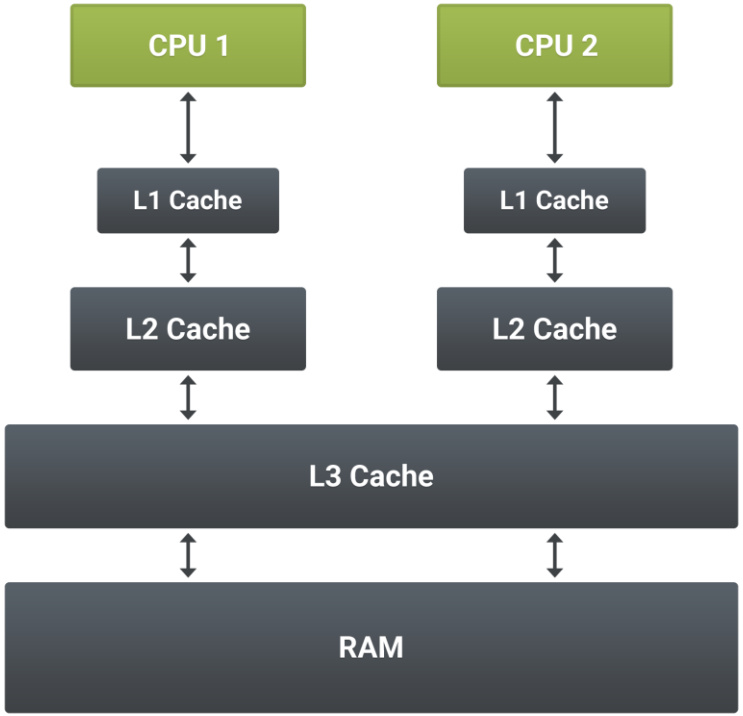
- Hardware Memory Model
- Java Memory Model
- Practical examples related to Java Memory Model
- JcStress Tool
- The future of the Java Memory Model

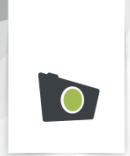
Quiz

What values of x can we see, if y=1 is printed out?

```
public class MyClass{

    int x, y=0;

    public void executeOnCPU1() {
        x = 1;
        x = 2;
        y = 1;
        x = 3;
    }

    public void executeOnCPU2() {
        System.out.println("x: "+x+ " y: "+y);
    }
}
```

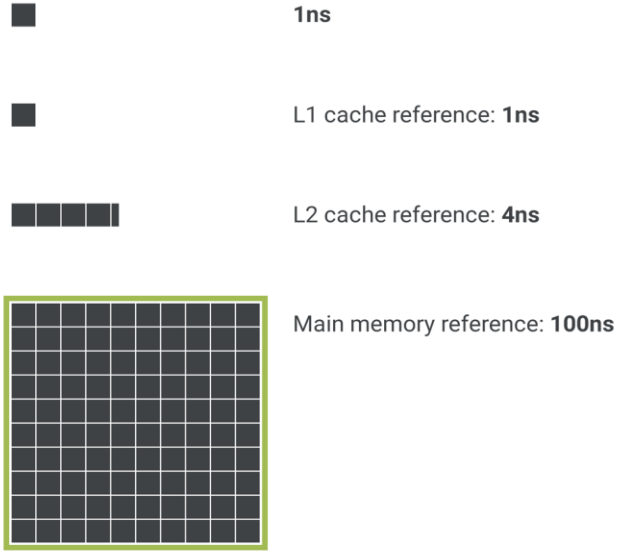
Hardware Memory Model





Hardware Memory Model

Latency Numbers Every Programmer Should Know



Source : http://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

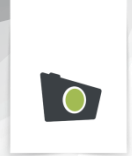
Java Memory Model for practitioners by Vadym Kazulkin and Rodion Alukhanov, ip.labs GmbH

Hardware Memory Model
Cache Coherence

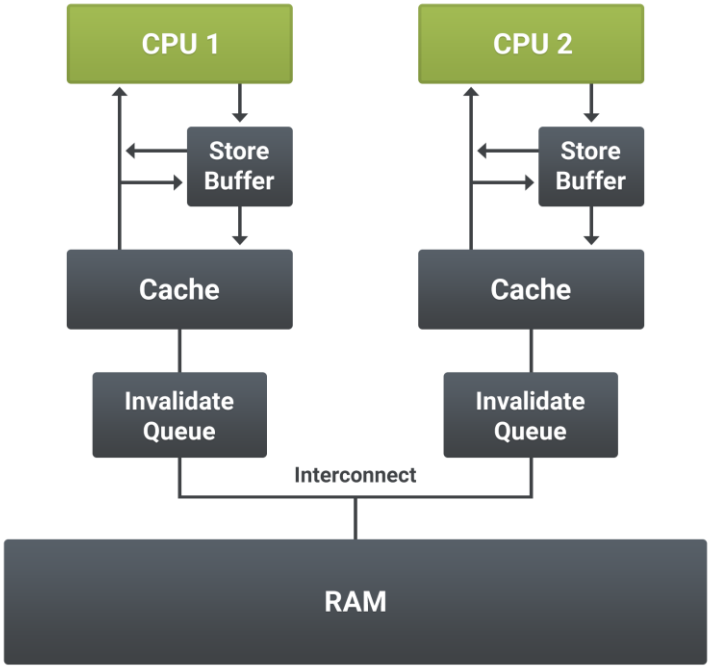
Cache coherence is the consistency of shared resource data that ends up stored in multiple local caches.

Hardware Memory Model
Cache Coherence Protocol MESI

Modified
The entry is only present in the current cache and it is modified.
Exclusive
The entry is only present in the current cache and has not been modified.
Shared
More that one processor has the entry in its cache.
Invalid
Indicates that this entry in the cache is invalid.



Hardware Memory Model
Stored Buffer & Invalidate Queues

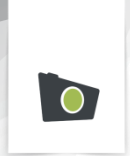


Hardware Memory Model
Memory Ordering

- A given CPU will always perceive its own memory operations as occurring in program order
- Some other CPU may observe memory operations results in a **different order** than what's written in the program

Hardware Memory Model
Memory Barrier

Memory barrier, aka membar or fence is the instruction that causes CPU to enforce an ordering constraint on memory operations issued before and after the barrier instruction.



Hardware Memory Model
Memory Barrier

For the cache coherence protocol MESI :

Store Memory Barrier - applies all the stores in the **store buffer**

Load Memory Barrier - applies all the invalidates that are already in the **invalidate queue**.

Hardware Memory Model
4 Types of Memory Barriers

LoadLoad membar: **Load** operations before the barrier must complete their execution before any **Load** operation after the barrier.

StoreStore membar: **Store** operations before the barrier must complete their execution before any **Store** operation after the barrier.

LoadStore membar : **Load** operations before the barrier must complete their execution before any **Store** operation after the barrier.

StoreLoad membar : **Store** operations before the barrier must complete their execution before any **Load** operation after the barrier.

Hardware Memory Model
Types of Memory Barrier/Example

```
int x=0; boolean done=false;

void executeOnCPU1() {
    x = 1;
    done=true;
}

void executeOnCPU2() {
    while(!done) {
      assert x=1;
     }
}
```

Hardware Memory Model
Types of Memory Barrier/Example

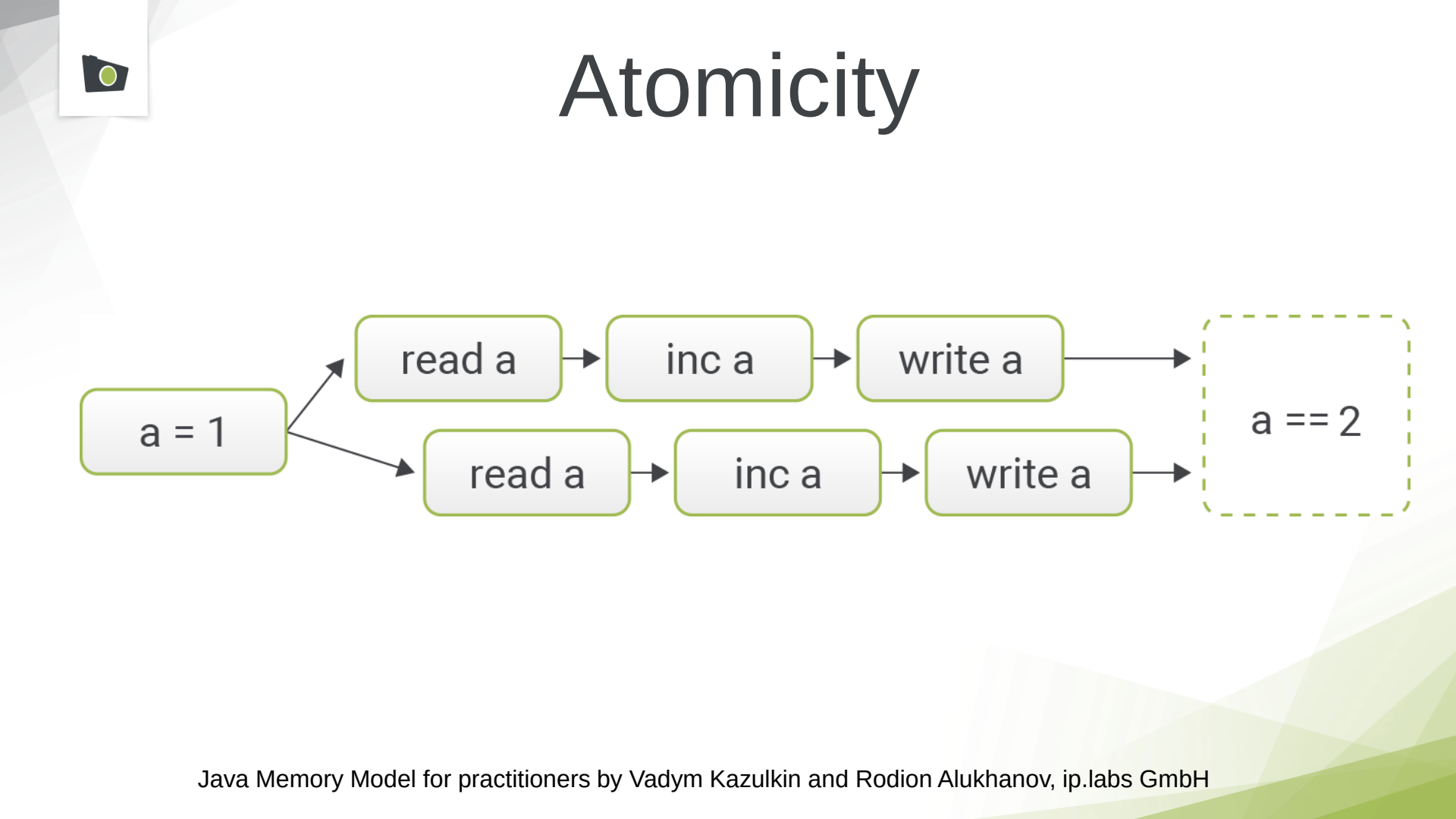
```
int x=0; boolean done=false;

void executeOnCPU1() {
    x = 1;
    InsertStoreStoreMembar();
    done=true;
}

void executeOnCPU2() {
    while(!done) {
      assert x=1;
     }
}
```

Hardware Memory Model

Allowed Memory reordering in some architectures

Type	ARMv7	x86	AMD64
Loads reordered after Loads	Yes	No	No
Loads reordered after Stores	Yes	No	No
Stores reordered after Stores	Yes	No	No
Stores reordered after Loads	Yes	Yes	Yes

Source : https://en.wikipedia.org/wiki/Memory_ordering

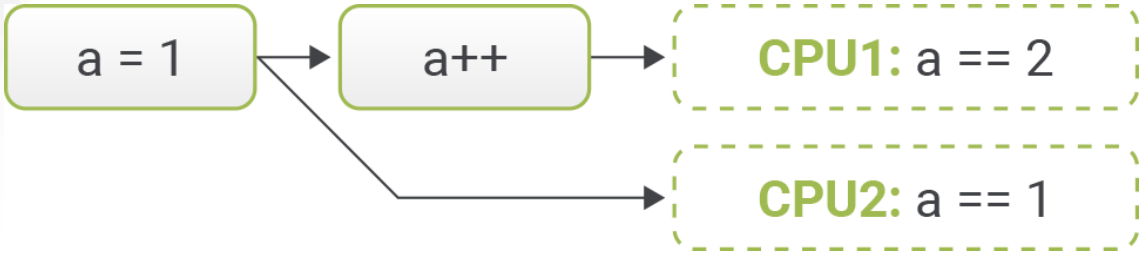


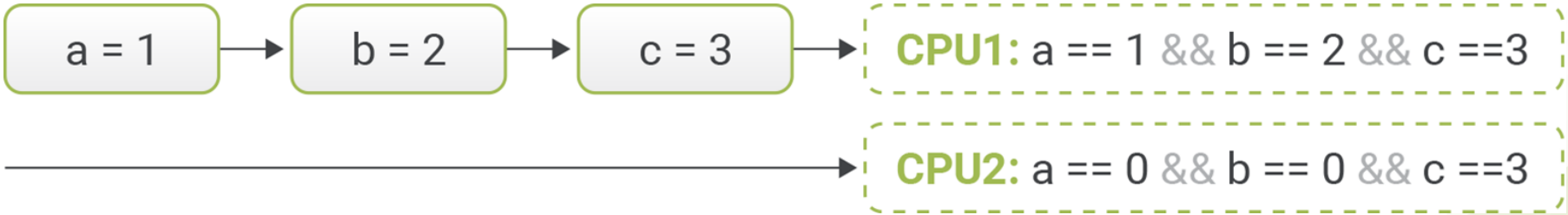
Atomicity

Counter

```
class Counter {
        private volatile long counter;

        public synchronized long inc() {
                counter++;
                return counter;
        }
        public void set(long v) {
                counter = v;
        }
        public long getCounter() {
                return counter.get();
        }
}
```

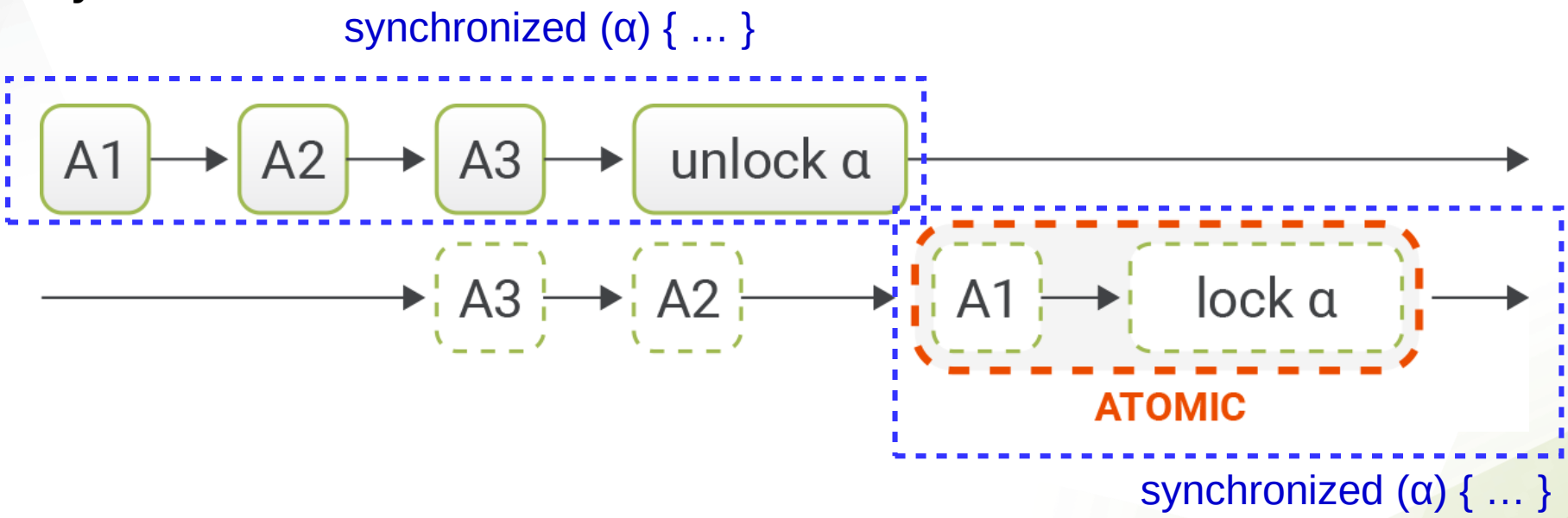
Visibility / Ordering





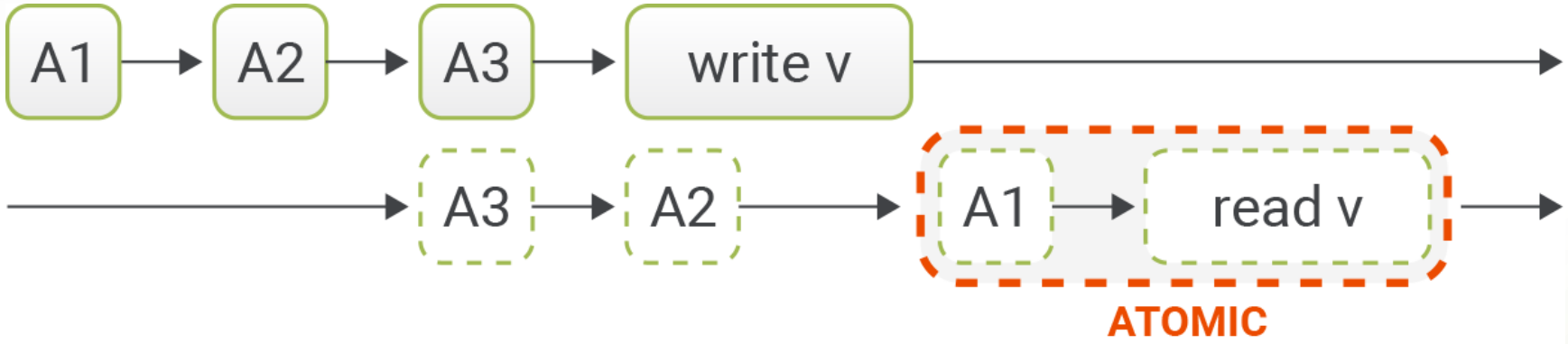
Happens-Before

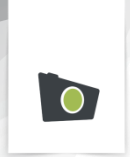
Synchronized



Happens-Before

Volatile





Visibility

```
class Counter {
        private volatile int counter;

        public synchronize int inc() {
                counter++;
                return counter;
        }
        public void set(int v) {
                counter = v;
        }
        public int getCounter() {
                return counter.get();
        }
}
```

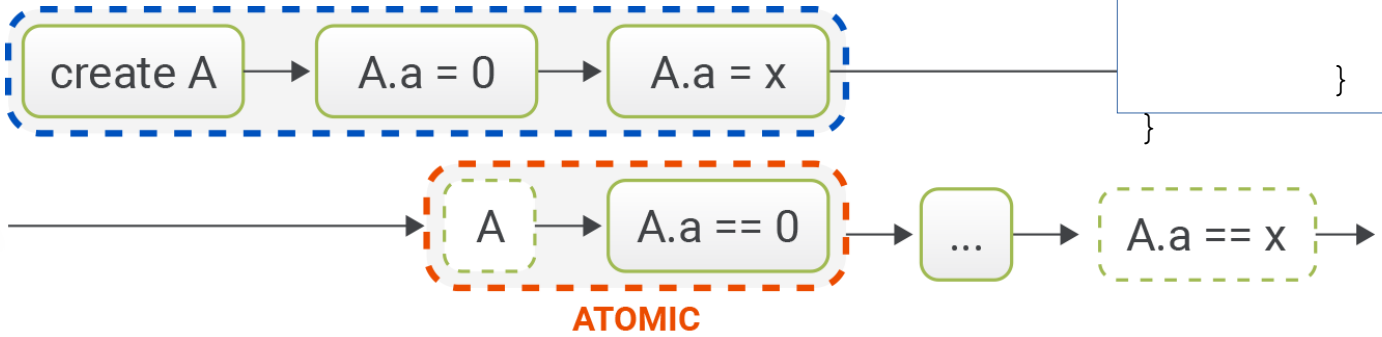
Counter

```
class Counter {
	private final AtomicLong counter = new AtomicLong();

	public long inc(long v) {
		return counter.incrementAndGet(v)
	}
	public void set(long v) {
		counter.set(v);
	}
	public long getCounter() {
		return counter.get();
	}
}
```

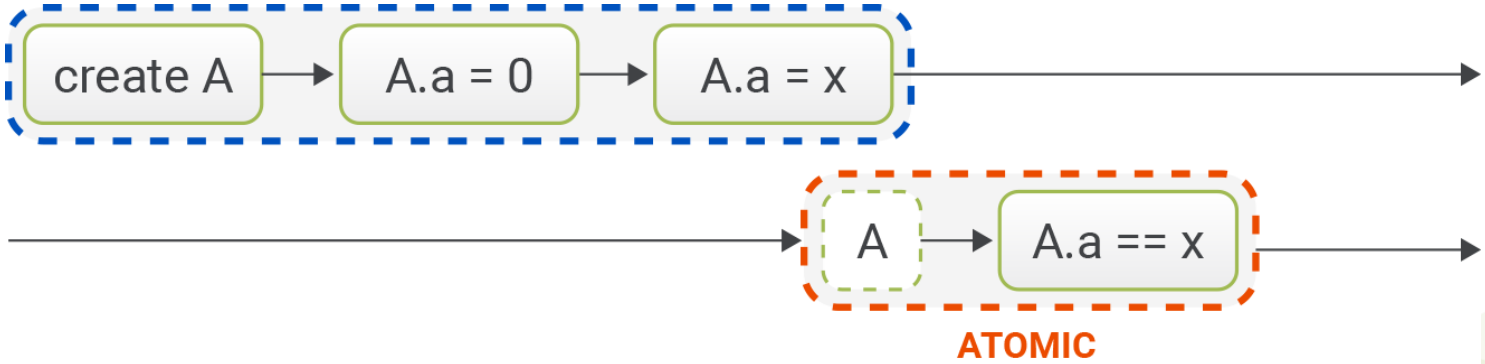
Safe Publication

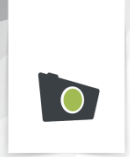
```
class A {
        int a;
        public A(int
x) {
                a = x;
        }
}
```

Default



Final field



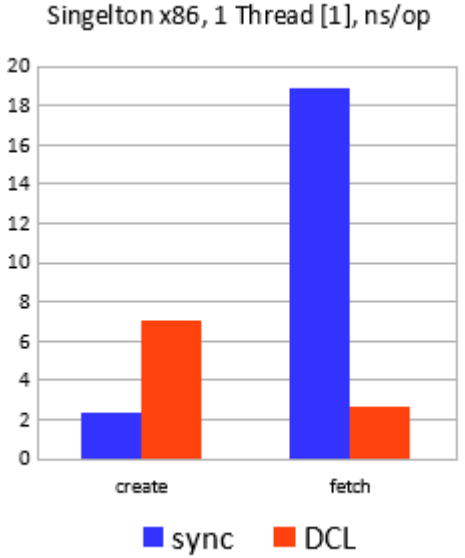


Safe Publication

```
public class Pair() {
        int final lower, upper;

        public Pair(int i1, int i2) {
                if (i1 > i2) throw new SomeException();
                …
        }

        public int getLower() {…}
        public int getUpper() {…}
}
```

Double-Checked Locking

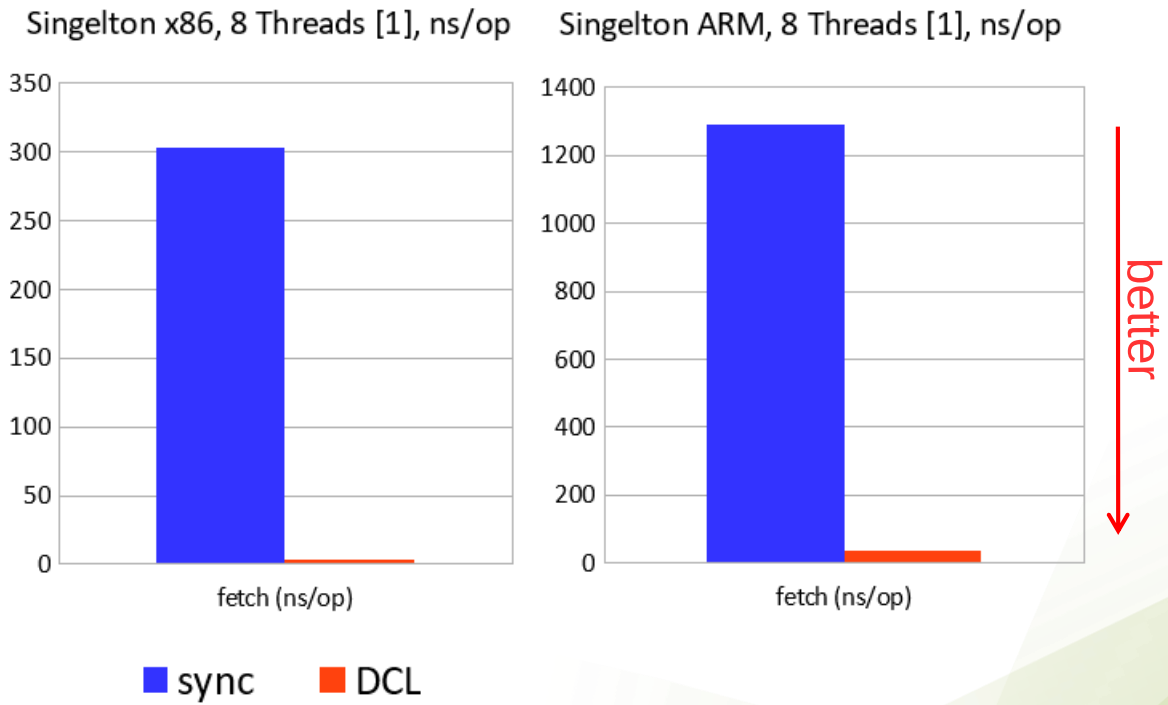
```
class DoubleCheckLocking {

private volatile Resource resource;
public Resource getResource() {
    if (resource == null) {
      synchronized (this) {
        if (resource == null) {
          resource = new Resource();
        }
      }
    }
    return resource;
  }
}
```



[1] https://shipilev.net/blog/2014/safe-public-construction

Double-Checked Locking

```
class DoubleCheckLocking {
  private volatile Resource resource;

  public Resource getResource() {
    if (resource == null) {
      synchronized (this) {
        if (resource == null) {
          resource = new Resource();
        }
      }
    }
    return resource;
  }
}
```



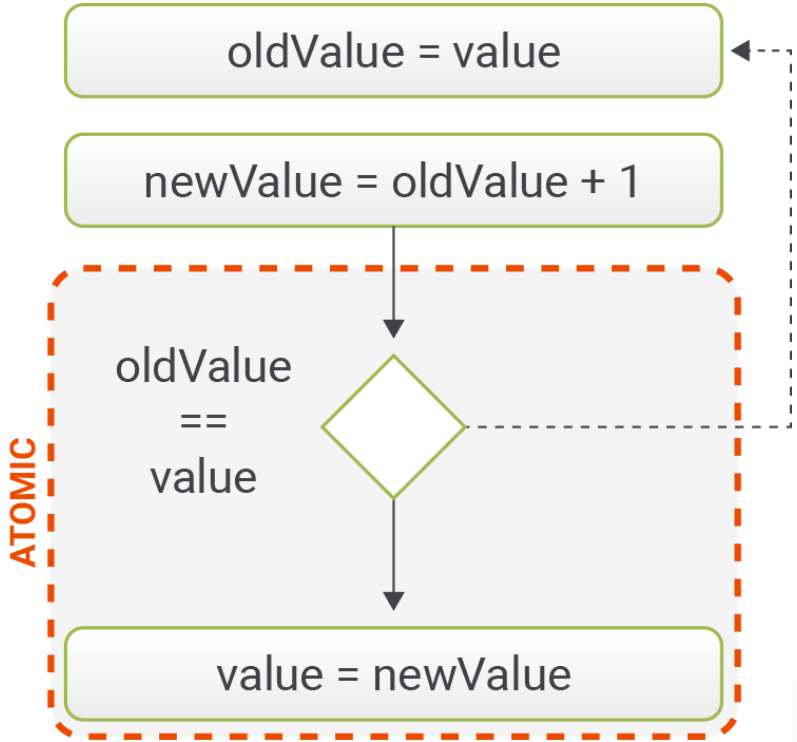
[1] https://shipilev.net/blog/2014/safe-public-construction

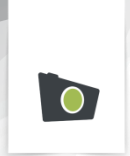
Java Memory Model for practitioners by Vadym Kazulkin and Rodion Alukhanov, ip.labs GmbH

Compare And Swap

```
class Fibonacci { // 1, 1, 2, 3, 5, 8, 13, 21...
        private Pair v = new Pair(0, 1);

        static class Pair() {
                int lower, upper;
                public Pair(int i1, int i2) {...}
        }

        public synchronize int next() {
                v = new Pair(v.upper, v.lower + v.upper);
                return v.lower;
        }
}
```

Compare And Swap



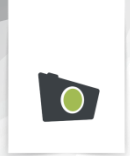


Compare And Swap

```
class Fibonacci { // 0, 1, 1, 2, 3, 5, 8, 13, 21...
  private final AtomicReference v = new AtomicReference();

  public int next() {
    Pair newValue;
    do {
      Pair old = v.get();
      newValue = new Pair(old.upper, old.lower + old.upper);
    } while(v.compareAndSet(old, newValue));
      return newValue.lower;
  }
}
```

JcStress Tool

The Open JDK Tool „Java Concurrency Stress tests (jcstress)“ is an experimental harness and a suite of tests to aid the research in the correctness of concurrency support in the JVM, class libraries, and hardware.



JcStress Tool

- requires at least 2 online CPUs
- few threads executing the test concurrently, sometimes rendez-vous'ing over the shared state
- multiple state objects generated per each run. Threads then either mutate or observe that state object
- test harness is collecting statistics on the observed states

Quiz for Reordering

What values of x can we see, if y=1 is printed out?

```
public class MyClass{
    int x, y=0;
    public void executeOnCPU1() {
        x = 1;
        x = 2;
        y = 1;
        x = 3;
    }
    public void executeOnCPU2() {
        System.out.println("x: "+x+ " y: "+y);
    }
}
```

UnfencedAcquireReleaseTest

```
@JCStressTest @State
@Outcome(id = "1, 0", expect = Expect.ACCEPTABLE_INTERESTING)
@Outcome(id = "1, 1", expect = Expect.ACCEPTABLE_INTERESTING)
@Outcome(id = "1, 2", expect = Expect.ACCEPTABLE)
@Outcome(id = "1, 3", expect = Expect.ACCEPTABLE)
public class UnfencedAcquireReleaseTest {
    int x,int y=0;
    @Actor
    public void actor1() {
        x = 1;
        x = 2;
        y = 1;
        x = 3;
    }
    @Actor
    public void actor2(IntResult2 r) {
        r.r1 = y;
        r.r2 = x;
    }
}
```

Answer to Quiz : UnfencedAcquireReleaseTest

JVM options: [-Xint] Iterations: 5 Time: 1000

Observed state	Occurrence	Expectation	Interpretation
1, 0	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 1	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 2	2191	ACCEPTABLE	Can see a released value of $x if $y is observed.
1, 3	3830389	ACCEPTABLE	Can see a released value of $x if $y is observed.

JVM options: [-client] Iterations: 5 Time: 1000

Observed state	Occurrence	Expectation	Interpretation
1, 0	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 1	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 2	7	ACCEPTABLE	Can see a released value of $x if $y is observed.
1, 3	22165994	ACCEPTABLE	Can see a released value of $x if $y is observed.

JVM options: [-server] Iterations: 5 Time: 1000

Observed state	Occurrence	Expectation	Interpretation
1, 0	1	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 1	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 2	4	ACCEPTABLE	Can see a released value of $x if $y is observed.
1, 3	24198455	ACCEPTABLE	Can see a released value of $x if $y is observed.

JVM options: [-server, -XX:-TieredCompilation] Iterations: 5 Time: 1000

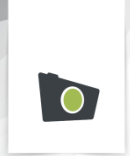
Observed state	Occurrence	Expectation	Interpretation
1, 0	6	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 1	0	ACCEPTABLE_INTERESTING	Without fence or volatile can read the default or old value for $x after $y is observed.
1, 2	63	ACCEPTABLE	Can see a released value of $x if $y is observed.
1, 3	22030031	ACCEPTABLE	Can see a released value of $x if $y is observed.

Answer to Quiz : UnfencedAcquireReleaseTest

Even on strong hardware memory models such as x86 reordering occurs because the Java Compiler und Just-In-Time Compiler also reorder in absence of happens-before.

So when y=1 , x =0 and x=1 are also possible.

Fix : declare x and y as *volatile*



Future of Java Memory Model

Old spec JSR 133 is outdated.

New spec JEP 188: Java Memory Model Update :

- C11/C++11 compatibility
- Volatile for automicy for long/double on 64bit
- Final field initialization problem
- Testing/Tool support. JcStress Tool is still experimental

Key take-aways

- Understanding of Java Memory Model is crucial in order to develop the correct code running concurrently
- Don't rely on hardware memory model (re)ordering-policy, compiler and JIT-Compiler also make optimizations and therefore re-order the instructions
- Use JcStress to test your assumptions
- Use immutable objects to avoid concurrency issues at all

Useful links

- JSR 133 (Java Memory Model) FAQ
https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html
- Memory Model
https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4
- Aleksey Shipilev „Java Memory Model Pragmatics“
http://shipilev.net/blog/2014/jmm-pragmatics/
- Aleksey Shipilev „Close Encounters of The Java Memory Model Kind“
http://shipilev.net/blog/2016/close-encounters-of-jmm-kind/
- JEP 188: Java Memory Model Update
http://openjdk.java.net/jeps/188
- OpenJDK Java Concurrency Stress tests (jcstress) Tool Webseite
https://wiki.openjdk.java.net/display/CodeTools/jcstress

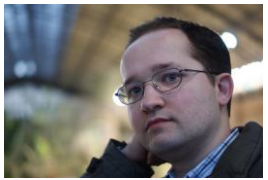
Questions?

Contact

Vadym Kazulkin :

Email : v.kazulkin@iplabs.de
Xing : https://www.xing.com/profile/Vadym_Kazulkin

Rodion Alukhanov :



Email : r.alukhanov@iplabs.de
Xing : https://www.xing.com/profile/Rodion_Alukhanov

Thank You!