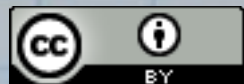


Introducing

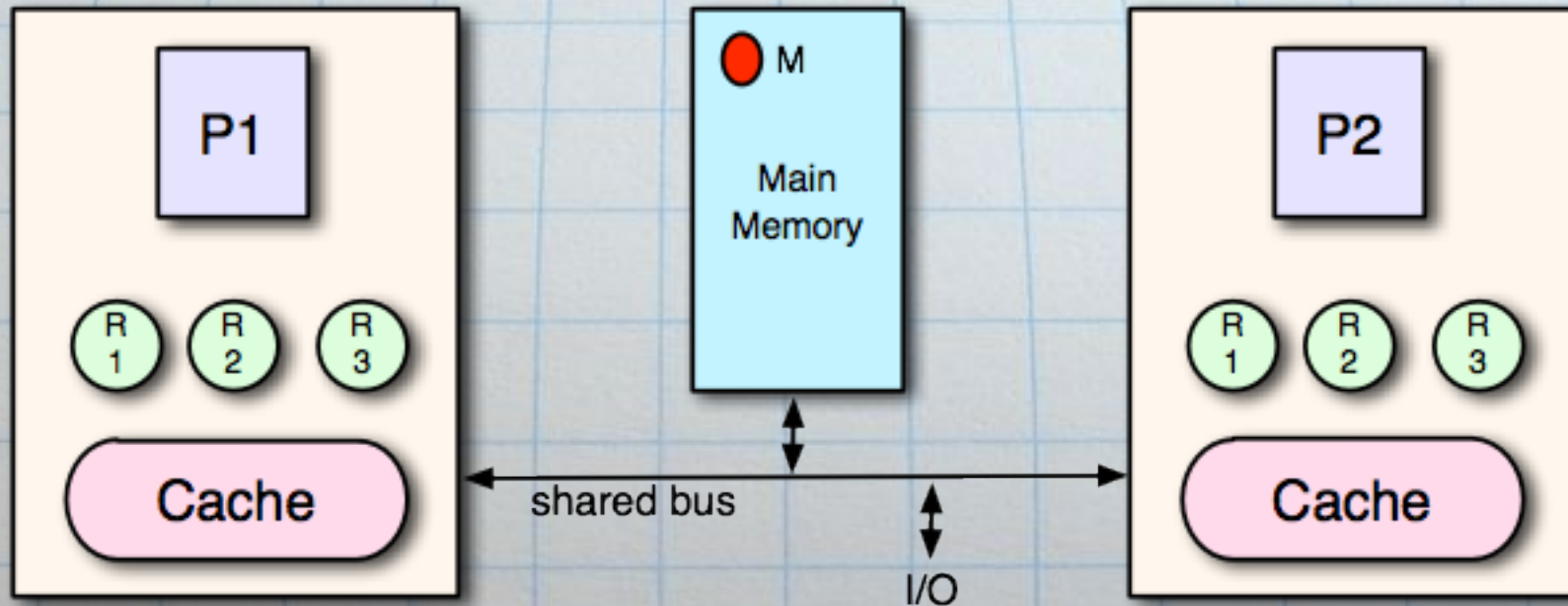
The Java Memory Model

Rubén Barroso - Jun 2010
ruben.bm@gmail.com

Licensed under



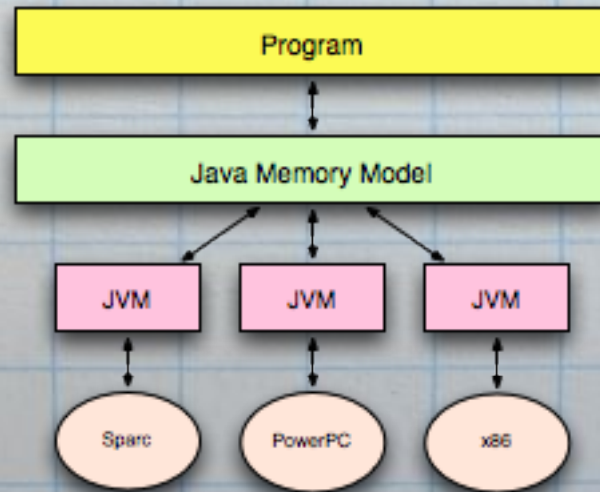
What is a Memory Model?



Under what conditions does a thread running on P2 see a value written to M by a thread running on P1?

The Java Memory Model

- Minimal guarantees for making visible writes of variables from one thread to other threads
- Abstraction layer provided by the JVM:



Working Memory

- Every thread is assigned a *working memory* in which to store values
- Working Memory = Caches + Registers
- The memory model defines rules in terms of when values must be transferred between main and working memories
- Issues

Ordering

Visibility

Atomicity

Sequential Consistency

- Intuitive extension to the von Neumann model applied to multiprocessors:

$\text{read}_p(a) \leftarrow \text{write}_q(a)$

Read of a by processor p sees the last write by any processor q

- Operations of a single processor in program order
- Intuitive and simple programming model
- PROBLEM: strict ordering among shared memory operations - lotsa optimizations disallowed

Within-thread as-if-serial semantics

- As if executed in a strictly sequential environment
- BUT (here's the catch): all kinds of optimizations are allowed
- For example, instruction reordering (examples looming...)

In a Multithreaded Environment ...

- Illusion of sequentiality cannot be maintained - very expensive! (*inter-thread coordination*)
- Coordination only required when data is shared

Processors

- Out-of-order execution
- *Cache Coherence* may affect the order of the write commits to main memory
- Single-threaded environments are not affected - Only they are way much faster

Optimizations at HW Level

- Reordering of instructions
- Higher clock rates
- Instruction pipelining
- Speculative execution
- Multilevel memory caches
- Many others

Reordering - Example

Initially $x = y = a = b = 0$

P1 P2

$a = 1$ $b = 1$

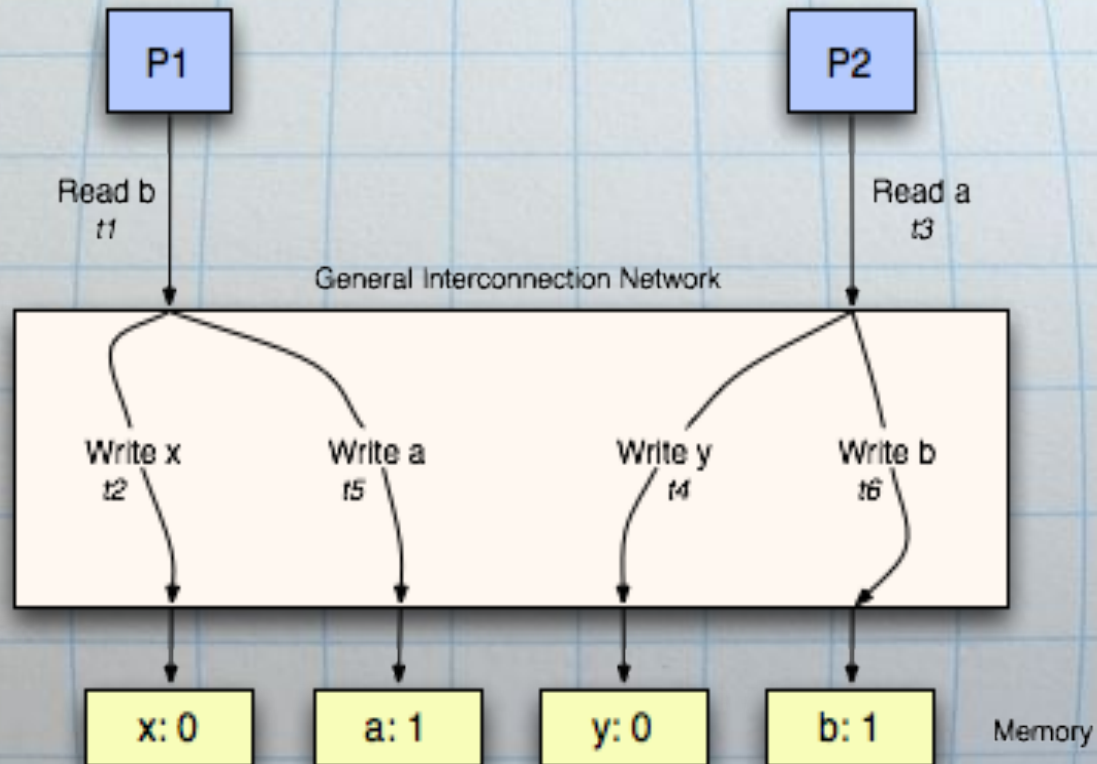
$x = b$ $y = a$

Plausible values for (x,y) are $(1,0)$, $(0,1)$ and $(1,1)$.

What about $(0,0)$?

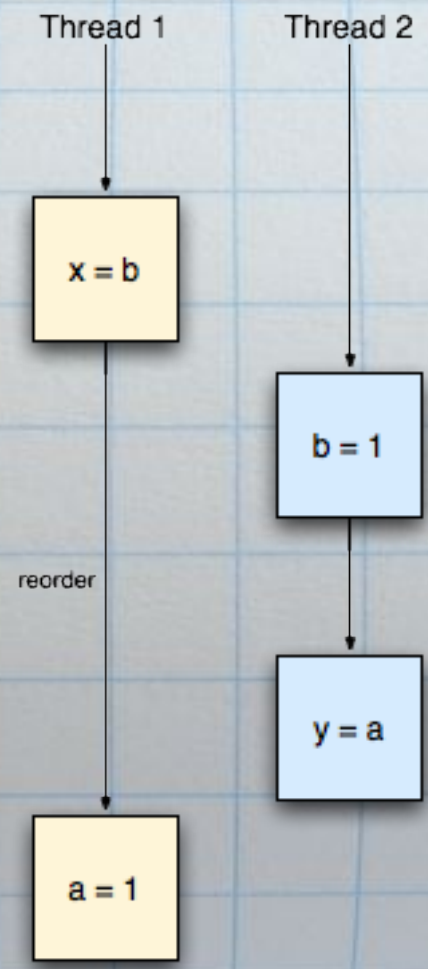
Reordering at HW Level

Violation of *sequential consistency*:



Reordering at Compiler Level

The compiler can reorder the instructions also:



Compilers

- Reordering of instructions
- Register-allocation algorithms
- Inlining of methods
- Others

Reordering at Compiler Level (Another Example*)

Three registers needed if no *spilling* is required:

$x = 1$

$y = 2$

$z[0] = x$

$z[1] = y$

What if the CPU contains only two registers?

If x and y are not referenced anywhere else, then:

$x = 1$

$z[0] = x$

$y = 2$

$z[1] = y$

No spilling!

*example from <http://useless-factor.blogspot.com/2010/02/instruction-scheduling-for-register.html>

Visibility

- Actions: read, write, lock, unlock, thread start and thread join
- Action A *happens-before* action B if a thread executing B sees the results of action A
- Data races

1 writer, n readers not ordered by *happens-before*

Visibility Rules - Program Order

Thread A

a = 2

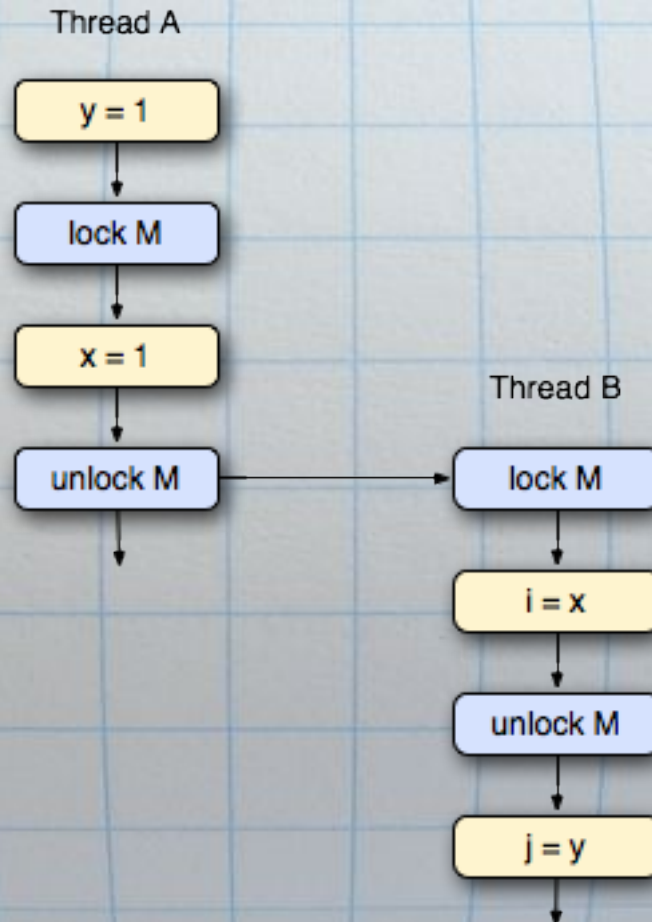


b[3] = a

Visibility Rules - Monitor Lock I

Lock Release: flush working memory (*everything*)

Lock Acquire: forced reload (*everything*)



Visibility Rules - Monitor Lock II

Java provides constructs to identify specific regions of code to be *synchronization*:

Thread A

Thread B

```
y = 1;
synchronized (myLock)
{
  x = 1;
}
synchronized (myLock)
{
  i = x;
  j = y;
}
```

(Answer to "How do we convey this information at Programming Language Level?")

Visibility Rules - Monitor Lock III

How is this information conveyed to the hardware?

- *Memory barrier (aka Memory Fence)*

Processor instructions that force the serialization of pending memory operations.

Example Itanium 2 running JDK 6

```
0x2000000001de81d0:    st1.rel [r36]=r0
0x2000000001de81d6:    mf
```

- The JVM inserts no-ops on a uni-processor

Visibility Rules - Volatile Variables I

Java provides the *volatile* type declaration modifier to identify variables that are used for synchronization purposes.

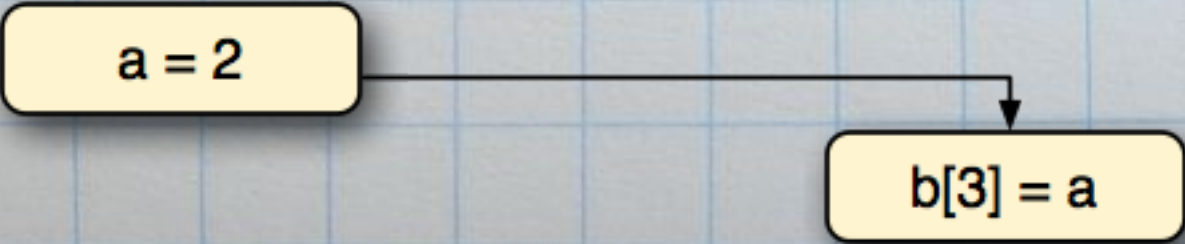
Initially `volatile int a = 0`

Thread A

`a = 2`

Thread B

`b[3] = a`

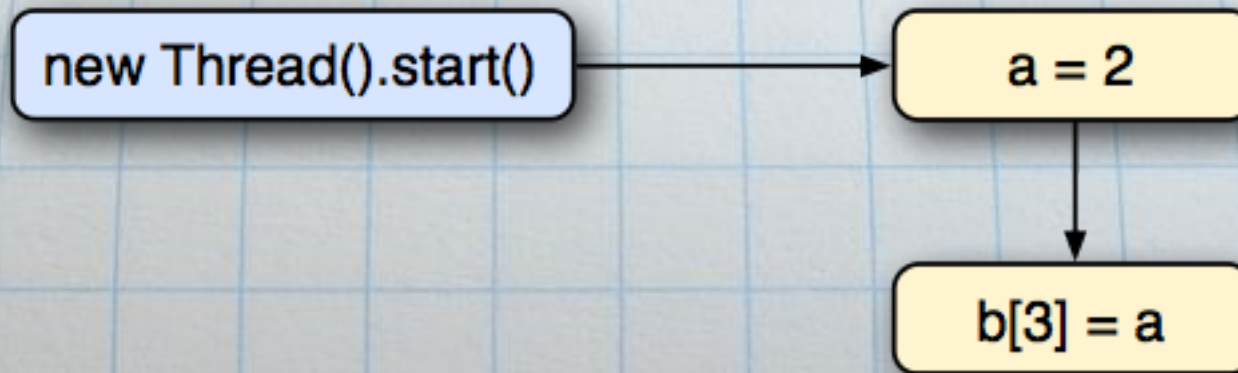


Visibility Rules - Volatile Variables II

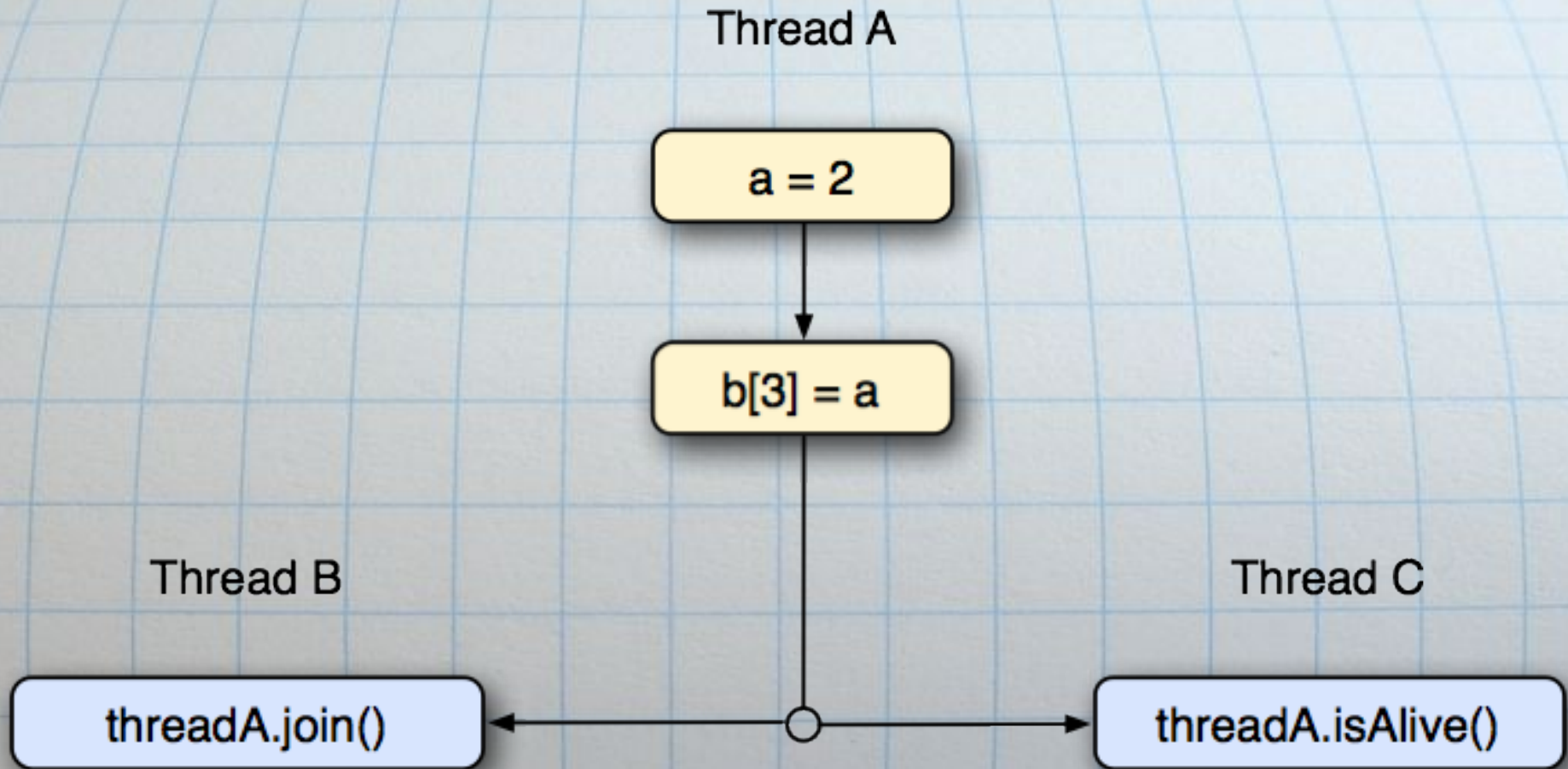
- *Writes* to volatile variables are flushed and made visible by the writer thread before proceeding with further memory operations
- *Reads* of volatile variables require the reader thread to reload the values of all volatile fields upon each access

Visibility Rules - Thread Start

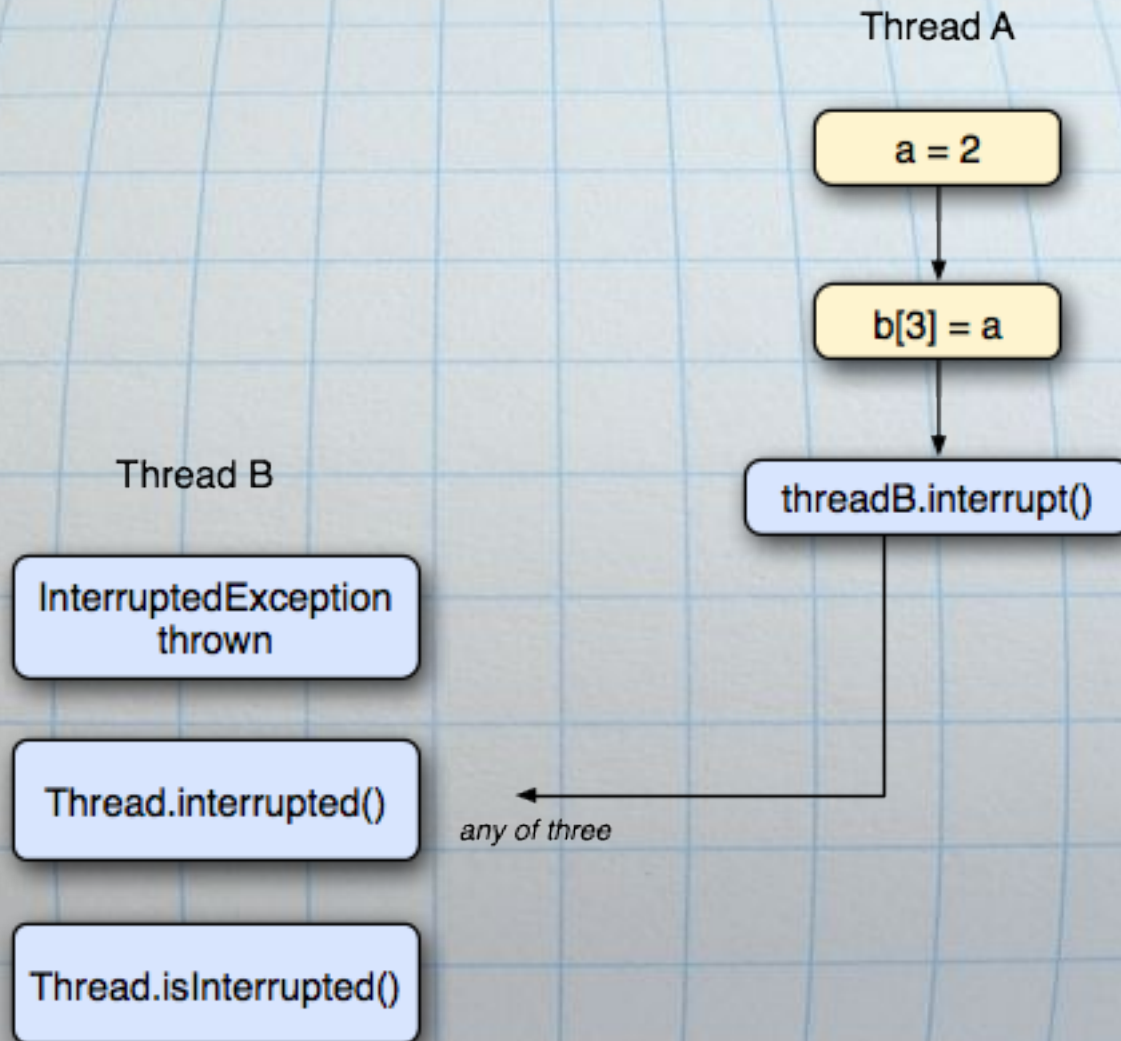
Thread A



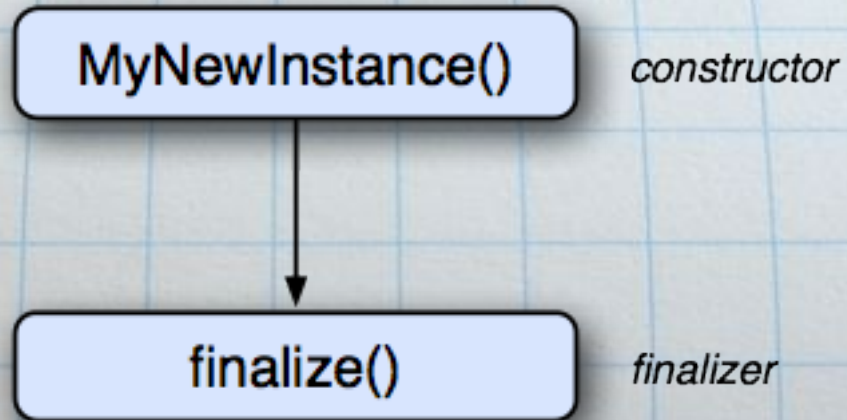
Visibility Rules - Thread Termination



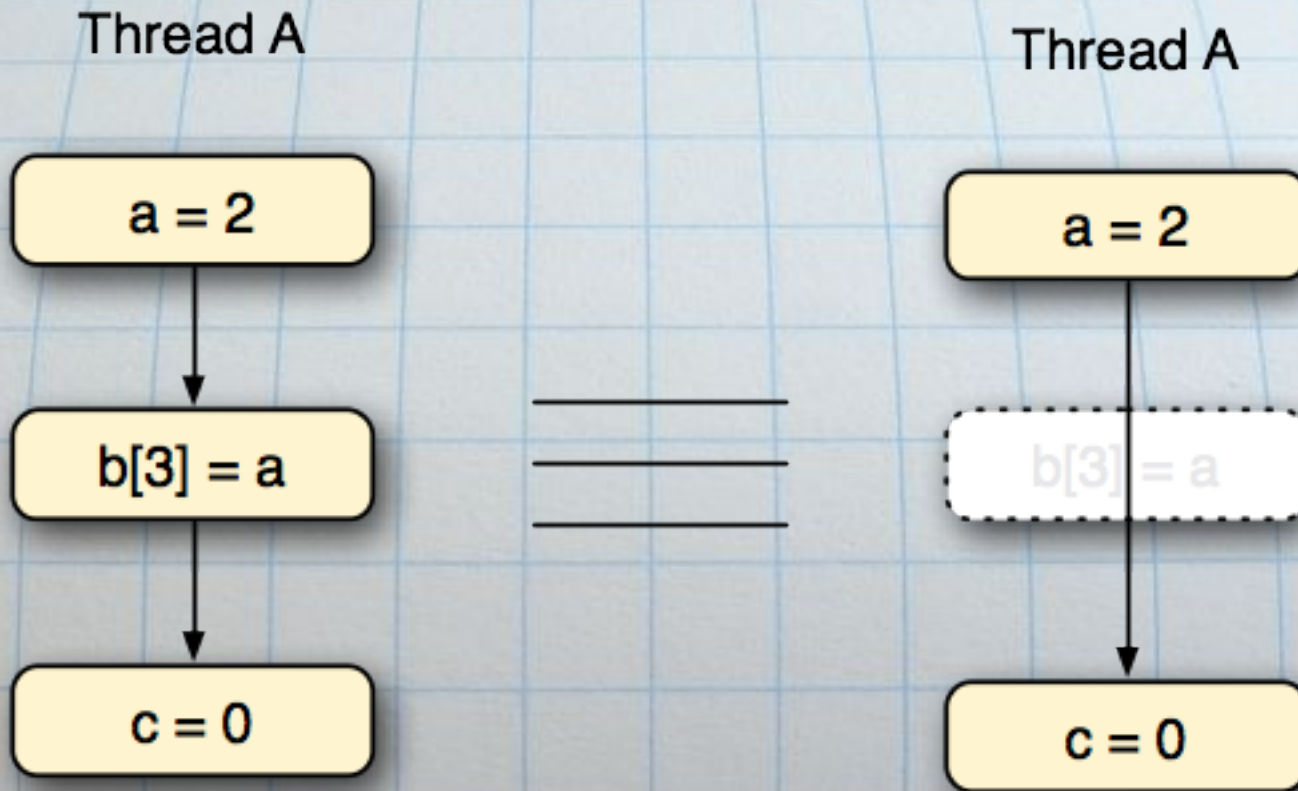
Visibility Rules - Interruption



Visibility Rules - Finalizer



Visibility Rules - Transitivity



Visibility - Out-of-thin-air Safety

- Variable values are not created "out of thin air"
- If a thread sees the value a for a variable, either some other thread must have written that value a in the past
- Or a is the initial value of the field

Atomicity

- long / double variables

Further Readings

- Lea, Douglas. *Concurrent Programming in Java: Design Principles and Patterns*. Reading, Mass.: Addison-Wesley, 2000. Print.
- Goetz, Brian. *Java Concurrency in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2006. Print.
- Gharachorloo K., Adve S. *Shared Memory Consistency Models: A Tutorial*. <http://research.compaq.com/wrl/techreports/abstracts/95.7.html>
- Byrne, Dennis. *Memory Barriers and JVM Concurrency*. http://www.infoq.com/articles/memory_barriers_jvm_concurrency
- *The Java Memory Model*. <http://www.cs.umd.edu/~pugh/java/memoryModel/>
- *Useless Factor Blog*. <http://useless-factor.blogspot.com/>