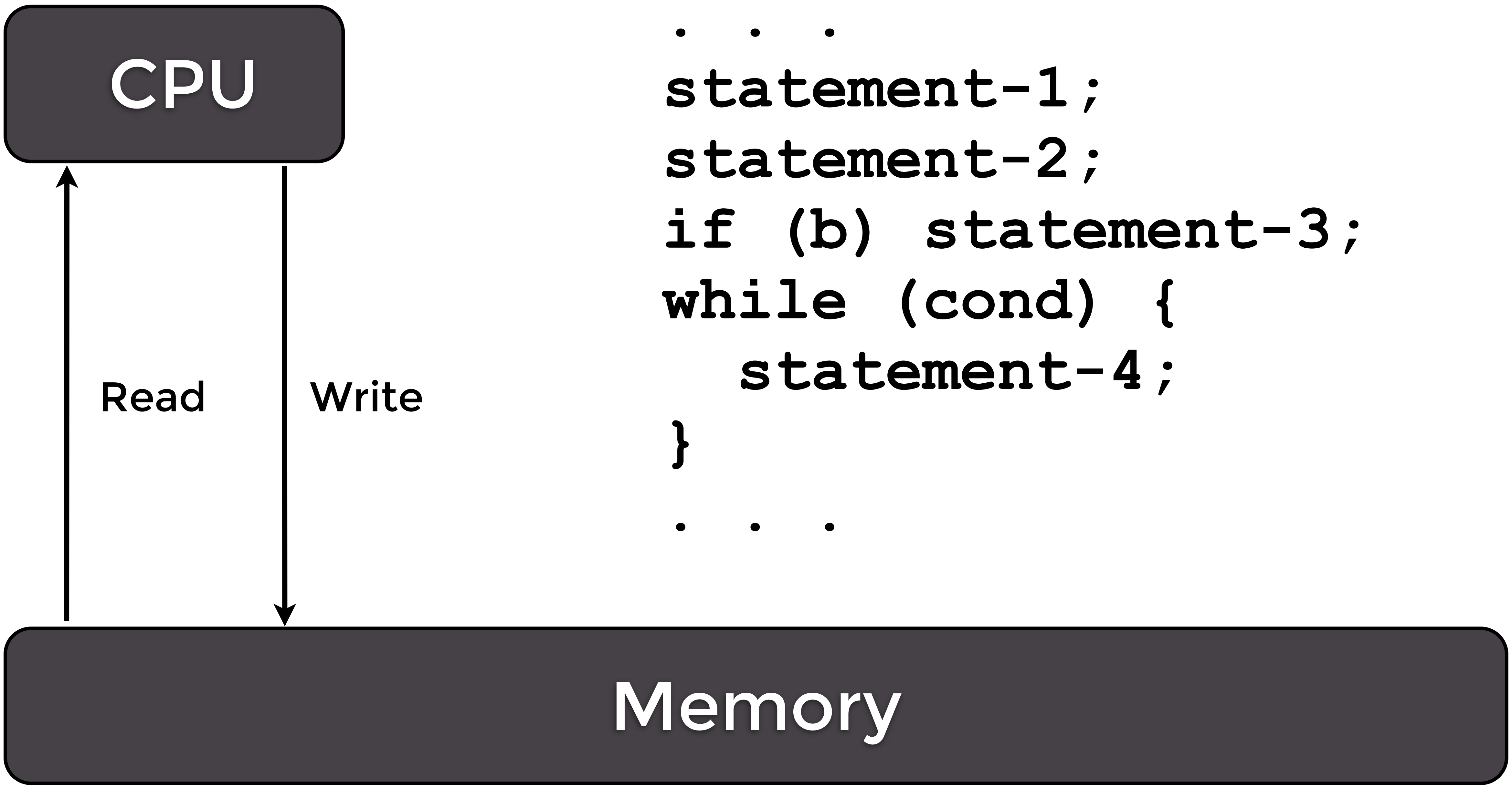
Stalking the Lost Write: Memory Visibility in Concurrent Java

Jeff Berkowitz, New Relic
December 2013

The Computer We Imagine




```
. . .
statement-1;
statement-2;
if (b) statement-3;
while (cond) {
  statement-4;
}
. . .
```

Monday, December 30, 13

When we first learned to program, we all learned it like this. All work done by statements {1..N} that have already executed is visible to statments {N+1, ...} when they later execute. Call it the intuitive rule of program order.

The Compiler We Imagine

Java	Assembly Language*

`x++;`	`mov mem.x, reg1`
	`incr reg1`
	`mov reg1, mem.x`

`y++;`	`mov mem.y, reg1`
	`incr reg1`
	`mov reg1, mem.y`

* Typical assembly language - no particular CPU

Later we learned that computers didn’t really execute what we wrote; rather, a program called a “compiler” translated our program into “machine language”. But we could still imagine the intuitive rule of program order for machine language.

The Compiler We Get

Java	Assembly Language
	`mov mem.x, reg1`
`x++;`	`mov mem.y, reg2`
	`incr reg1`
	`mov reg1, mem.y`
`y++;`	`incr reg2`
	`mov reg2, mem.x`

Eventually, perhaps by debugging an optimized build, we learn that it's not so simple. Example: reads take time, so compilers learned to issue them in advance.

The End Result

Java	Assembly Language	Hardware Level*
`x++;`	`mov mem.x, reg1` `mov mem.y, reg2`	`rd.issue(x)` `rd.issue(y)`

	`incr reg1`
	`mov reg1, mem.x`
		`resp.mov(r1)`
		`incr r1`
		`wr.async(r1, x)`

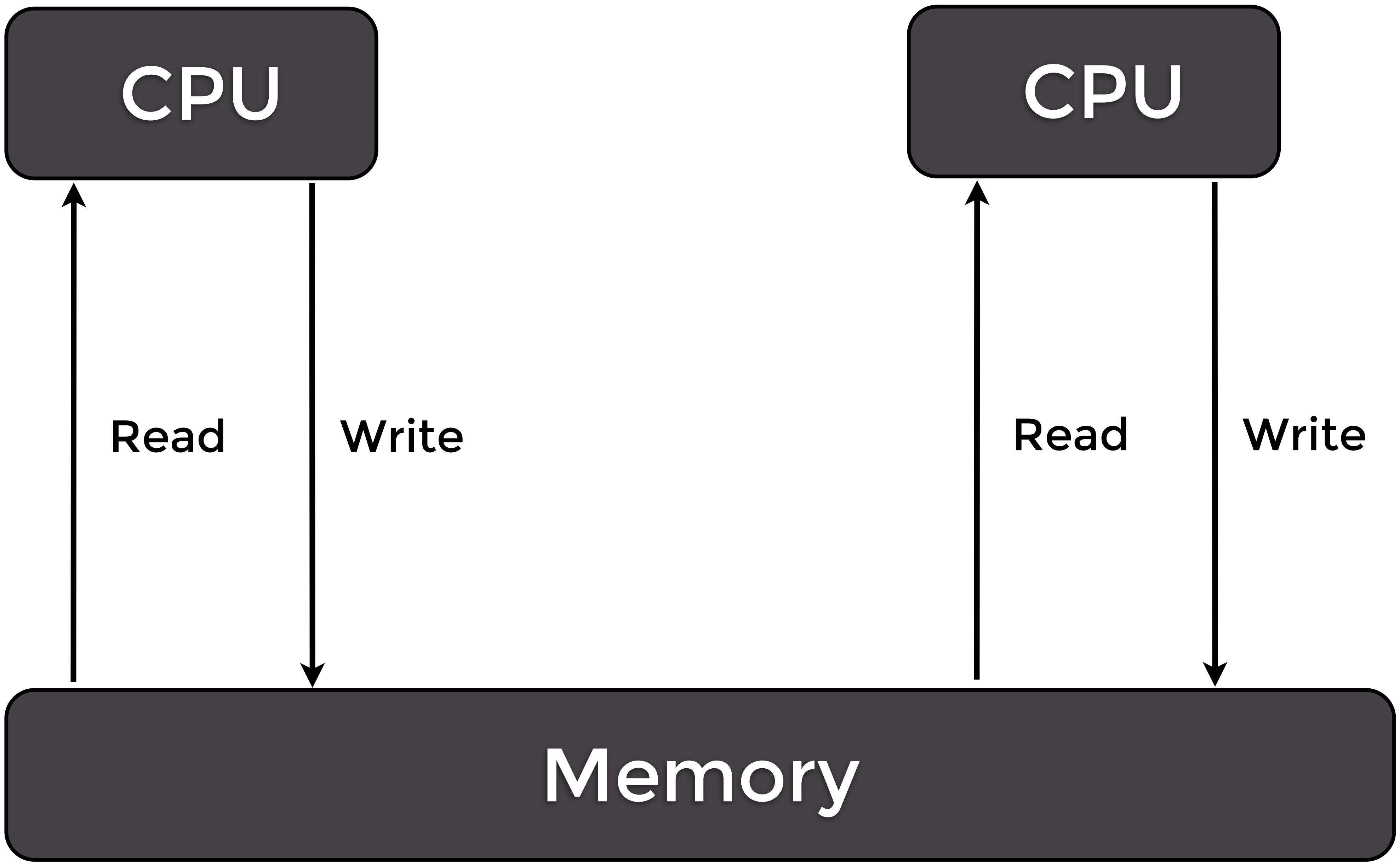
`y++;`	`incr reg2`
	`mov reg2, mem.y`
		`resp.mov(r2)`
		`incr r2`
		`wr.async(r2, y)`

* Typical micro operations - no particular CPU

Monday, December 30, 13

The hardware splits reads into two parts, issue and collect-response. The compiler writer recognized that issuing the read for “y” before the increment of “x” would allow the hardware to overlap issuing the read of “y” with the memory cycle for “x”. Also, the writes are asynchronous where possible.

The Multiprocessor We Imagine



There are no caches or memory buffering here

Now let's consider the intuitive rule of program order on an idealized multiprocessor.

Code Example 1

Possible outcomes for x and y?

```
int x, y, a, b; // all zero
```

First CPU ("thread")

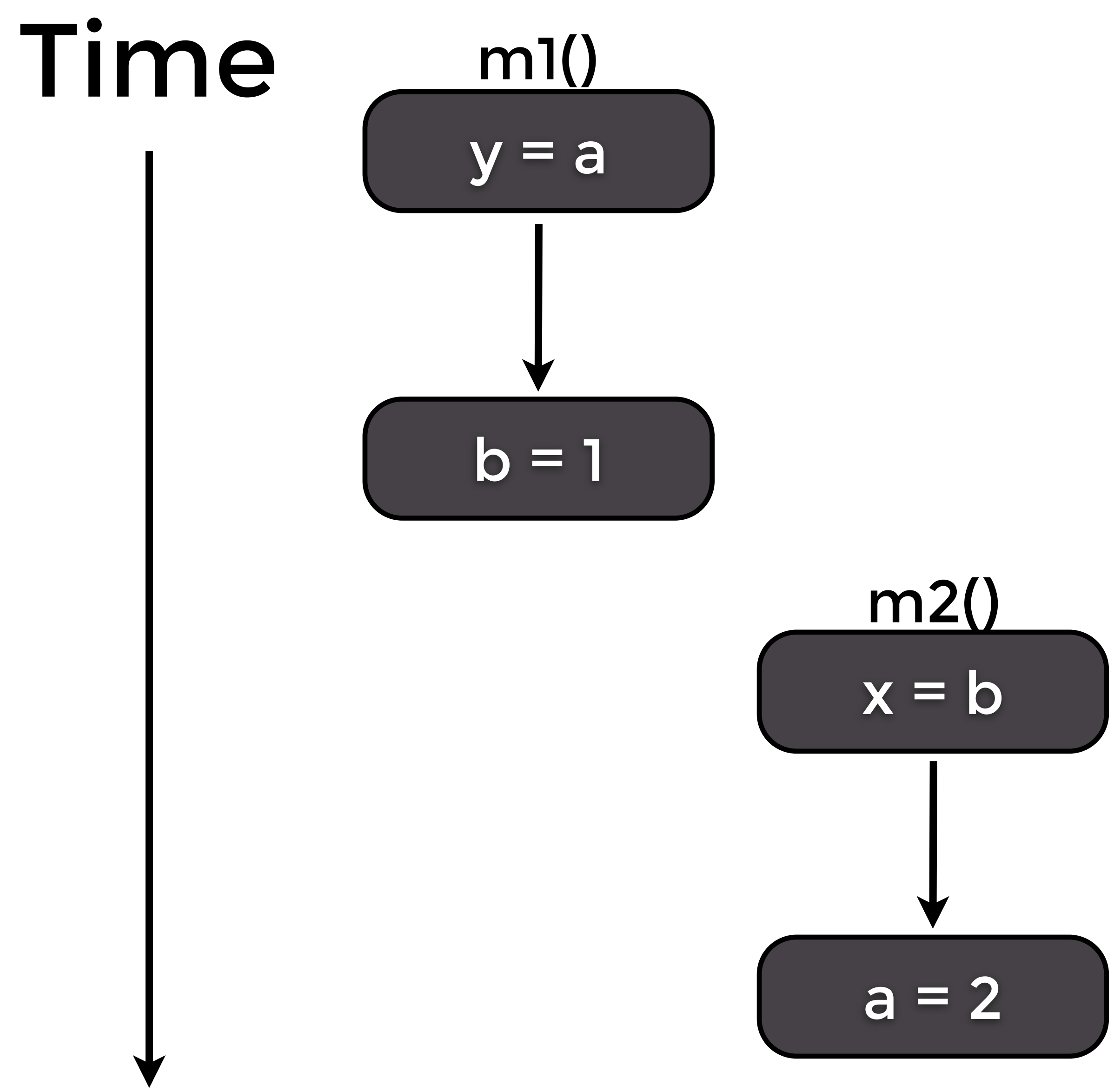
```
void m1() {
  y = a;
  b = 1;
}
```

Second CPU

```
void m2() {
  x = b;
  a = 2;
}
```

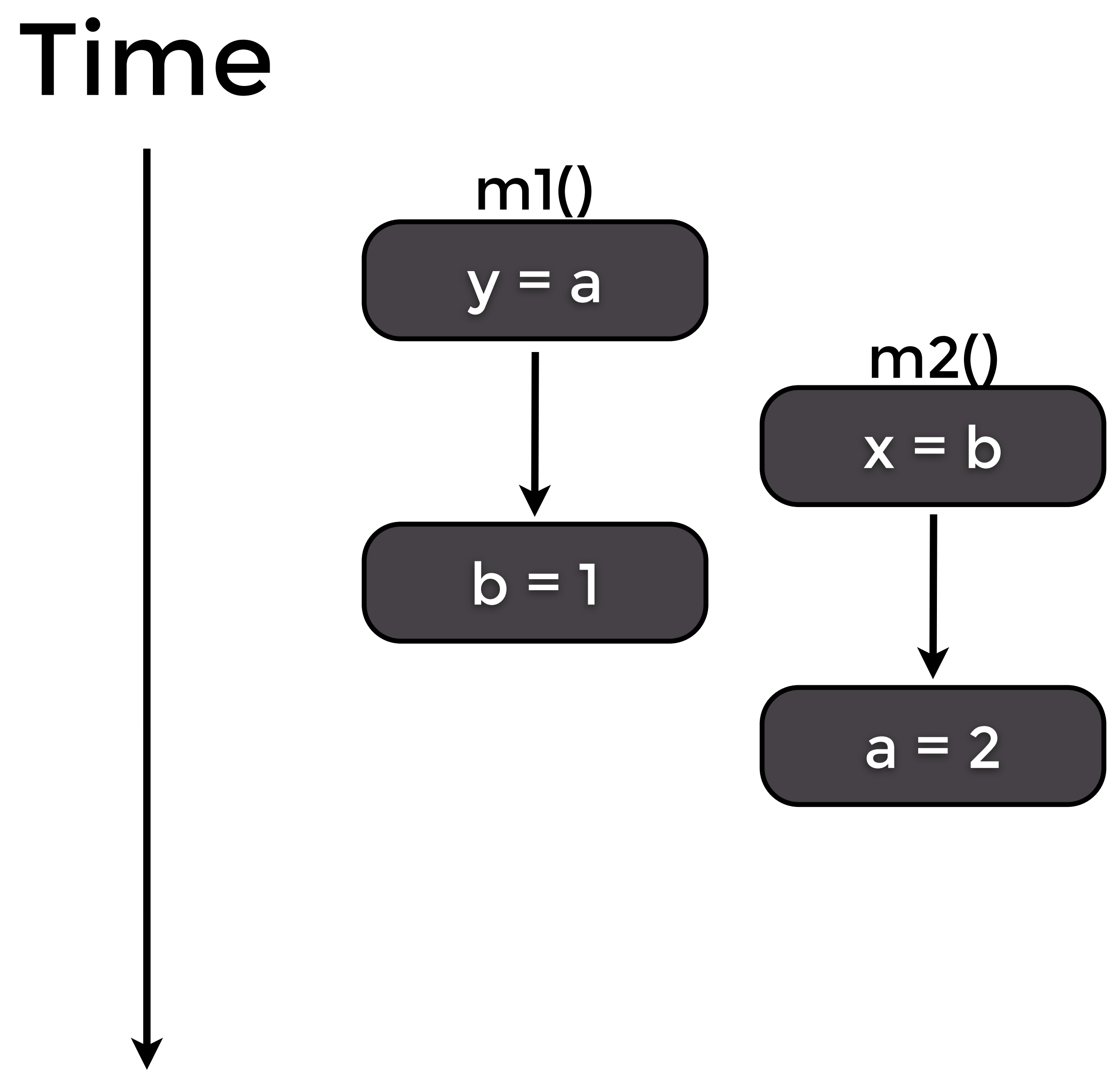
Do not emulate this example

Possible Execution Trace



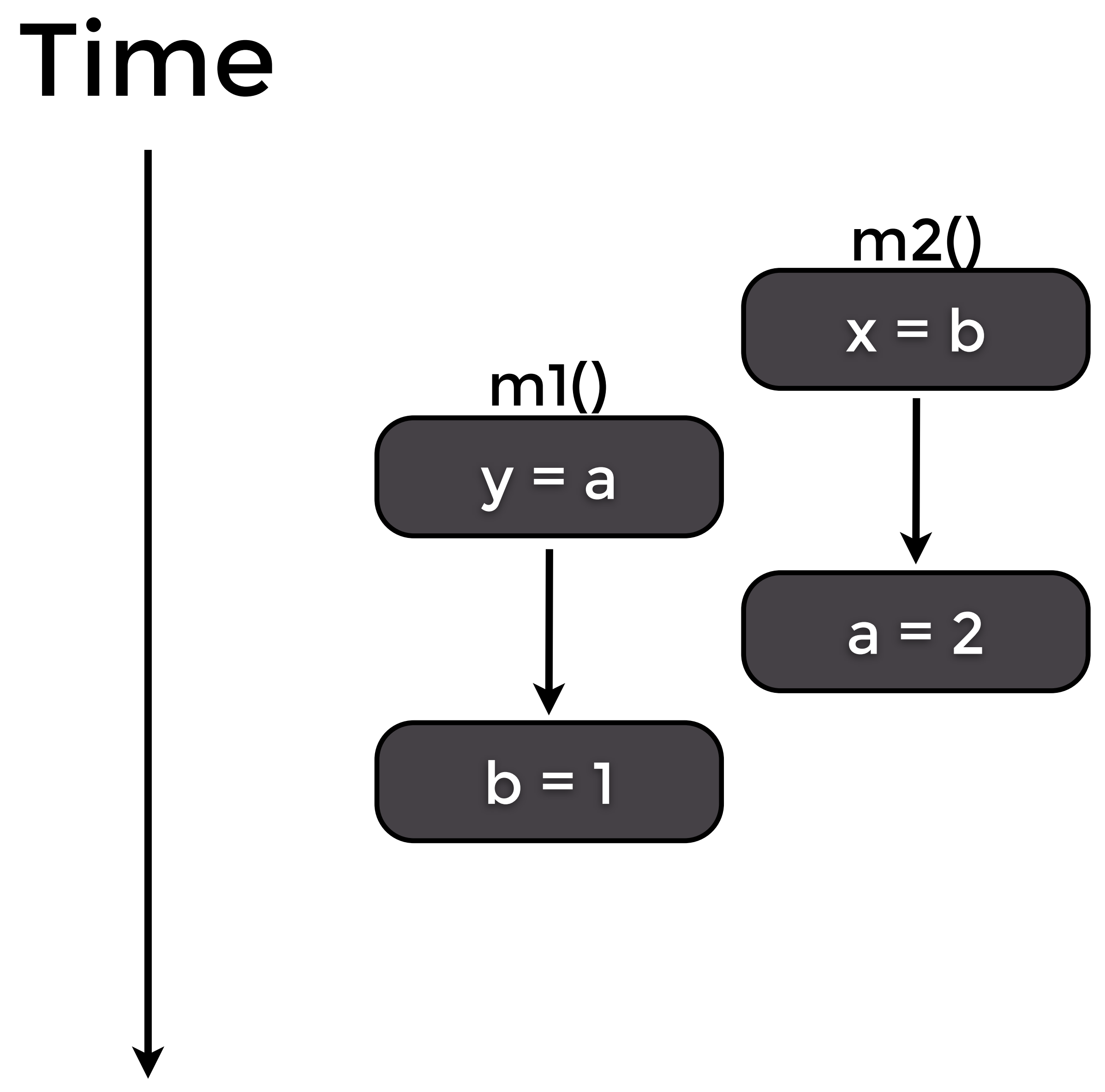
Outcome: x == 1, y == 0

Possible Trace #2



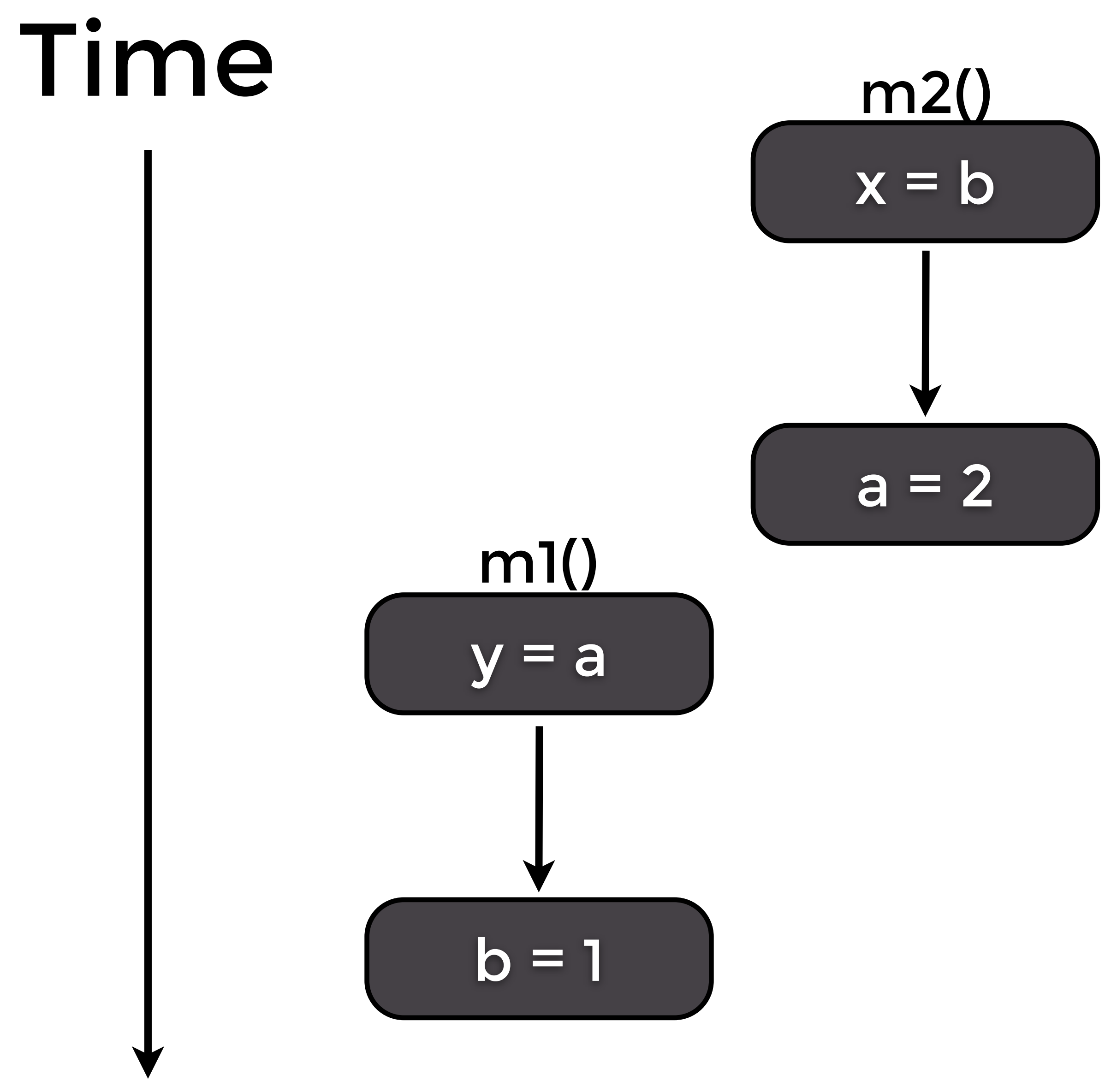
Outcome: x == 0, y == 0

Possible Trace #3



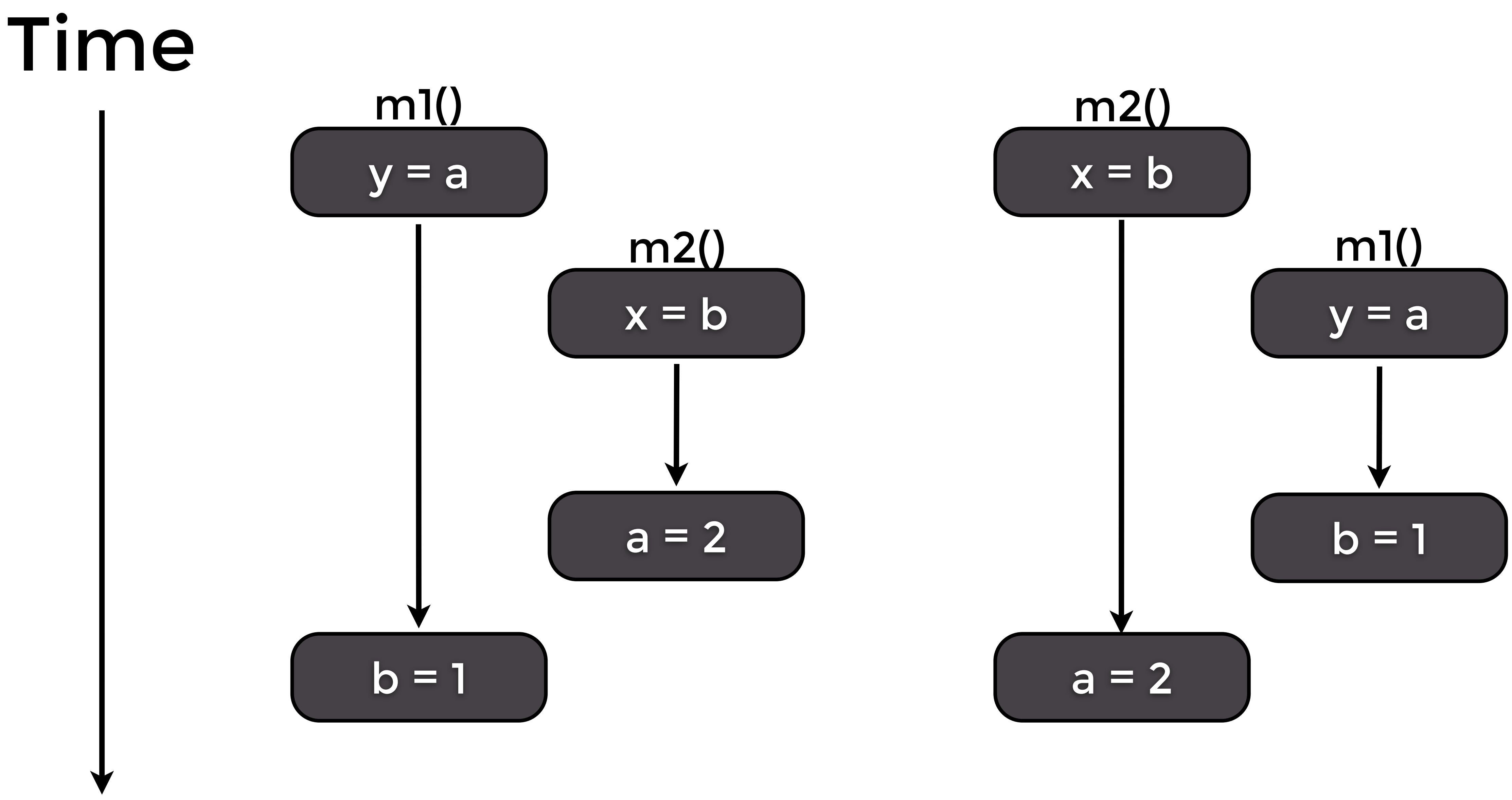
Outcome: x == 0, y == 0

Possible Trace #4



Outcome: x == 0, y == 2

Oh, And #5 and #6



Outcome: x == 0, y == 0 x == 0, y == 0

Is That It?

- It looks like x or y must be 0 in the result
  - Makes sense: the first statement of m1() grabs a 0, and so does the first statement of m2()
- Is our reasoning correct?

Surprisingly, No

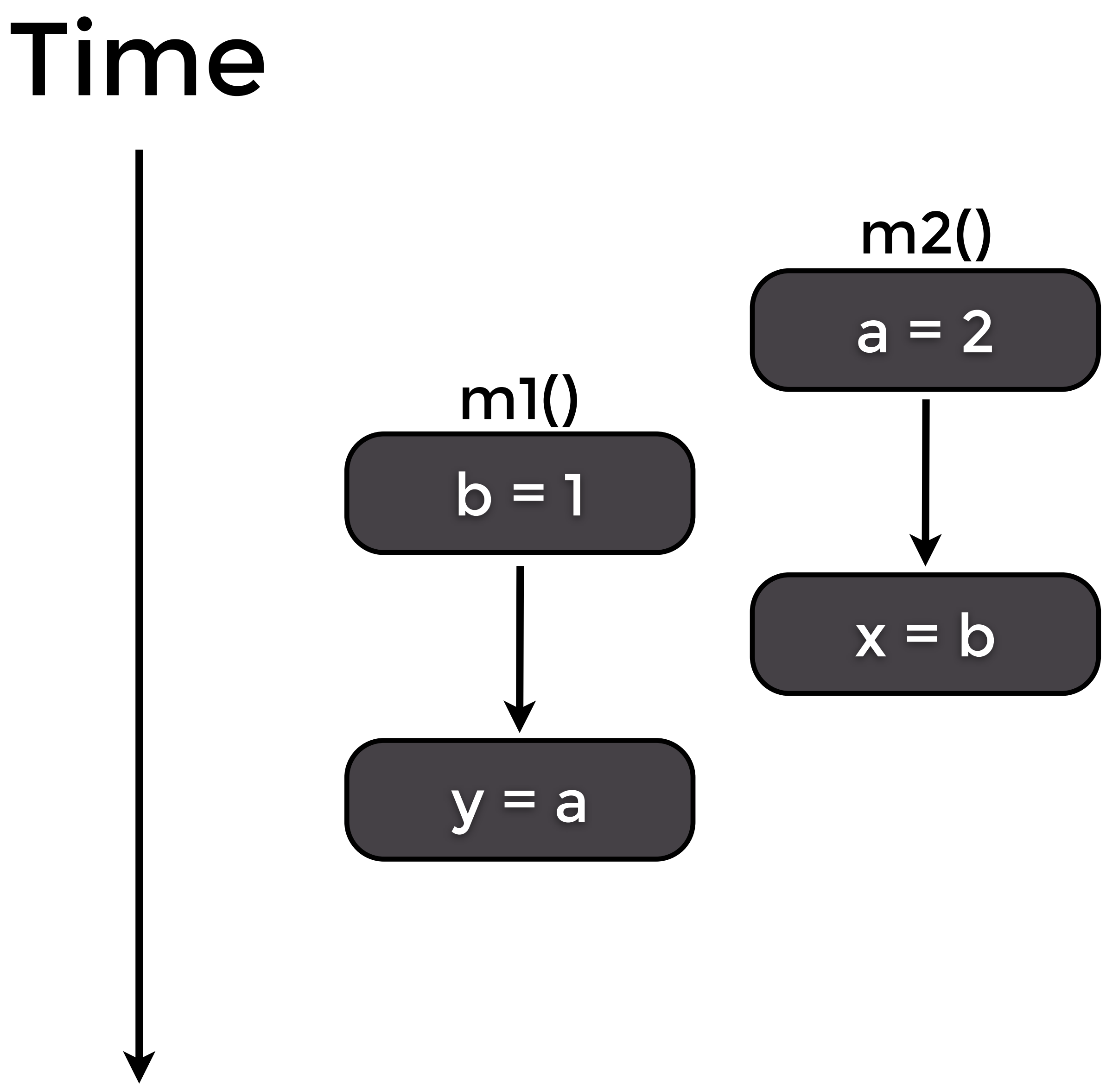
```
void m1() {
   y = a;        mov #1, mem.b
   b = 1;        mov mem.a, mem.y
}

void m2() {
   x = b;        mov #2, mem.a
   a = 2;        mov mem.b, mem.x
}
```

Counterintuitively, the compiler can reverse the order

When the order of the two statements is reversed in either m1 or m2, the intuitive rule of progam order is not violated because neither method tries to observe the values of these variables. Compilers do not understand cross-thread visibility!

So This is Also Possible



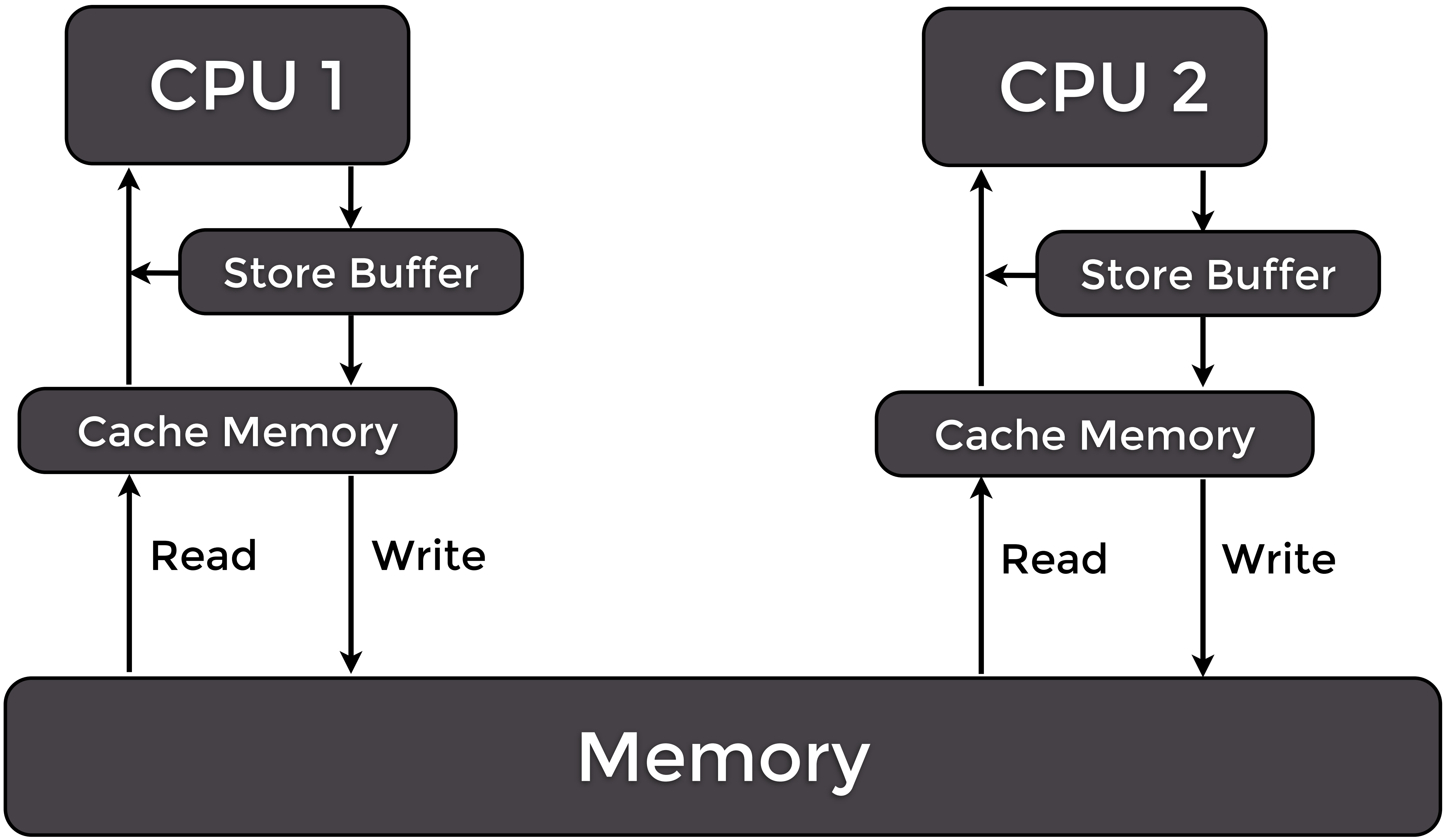
Outcome: x == 1, y == 2

*

Monday, December 30, 13

Again, neither m1() nor m2() individually attempts to observe the result of its own writes, and “compilers don’t do threads.”

And It Gets Worse...



This is just an example. It's way worse than this.

Monday, December 30, 13

Hardware designers invented caches and store buffers and other performance enhancing hardware tricks. In general, all these optimizations are considered acceptable so long as the intuitive rule of program order isn't violated. We've just seen that the intuitive rule of program order can allow for confusing compiler behavior. Now let's look at the hardware.

Code Example 2

```
int a, b; // both zero
```

<u>First CPU ("thread")</u>

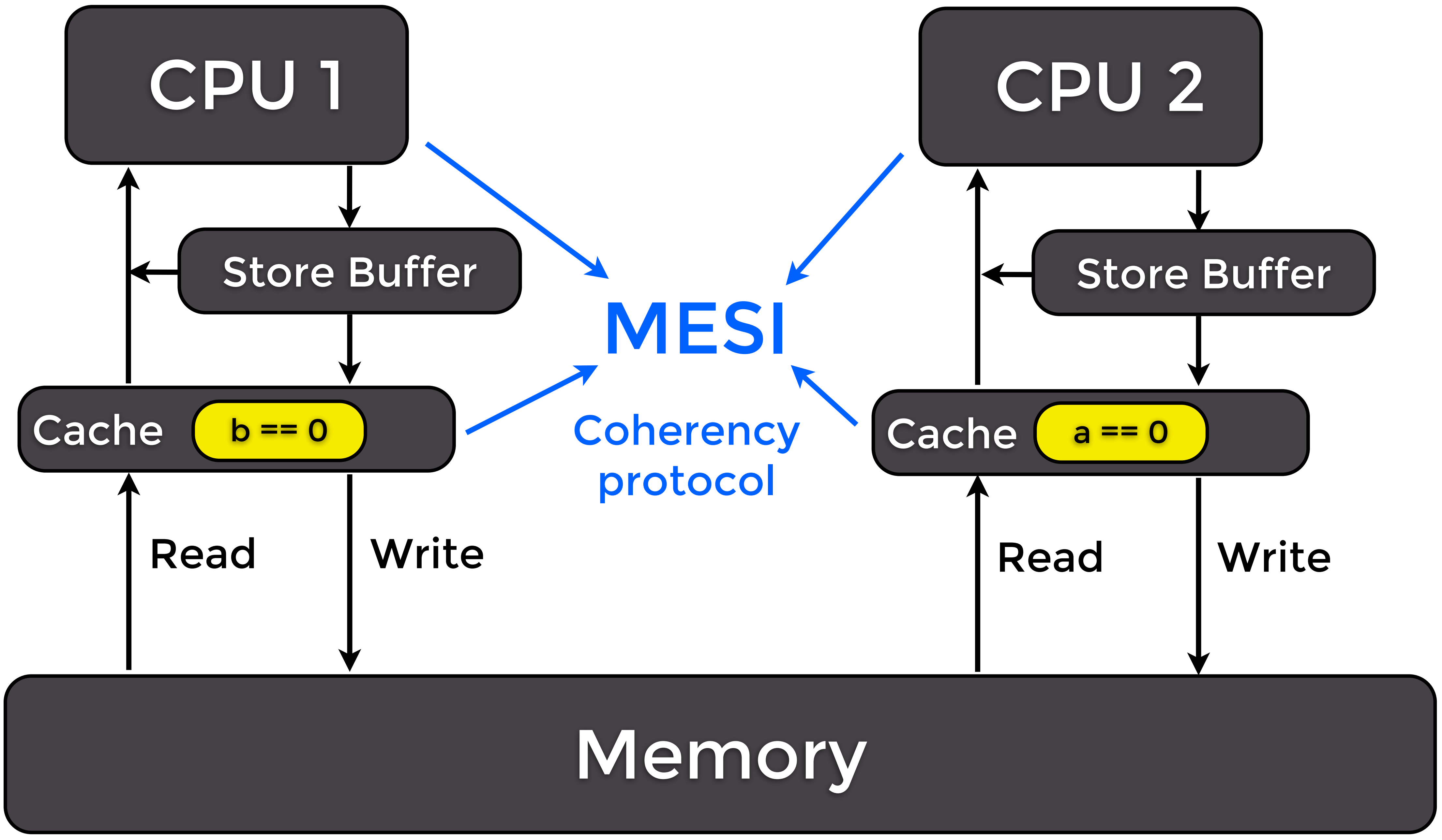
```
void m1() {
  a = 1;
  b = 1;
}
```

<u>Second CPU</u>

```
void m2() {
  while (b == 0)
        ;
  assert(a == 1);
}
```

Credit: http://bit.ly/pjug2013-mckenney-parallel, Appendix C, p. 231 et seq.

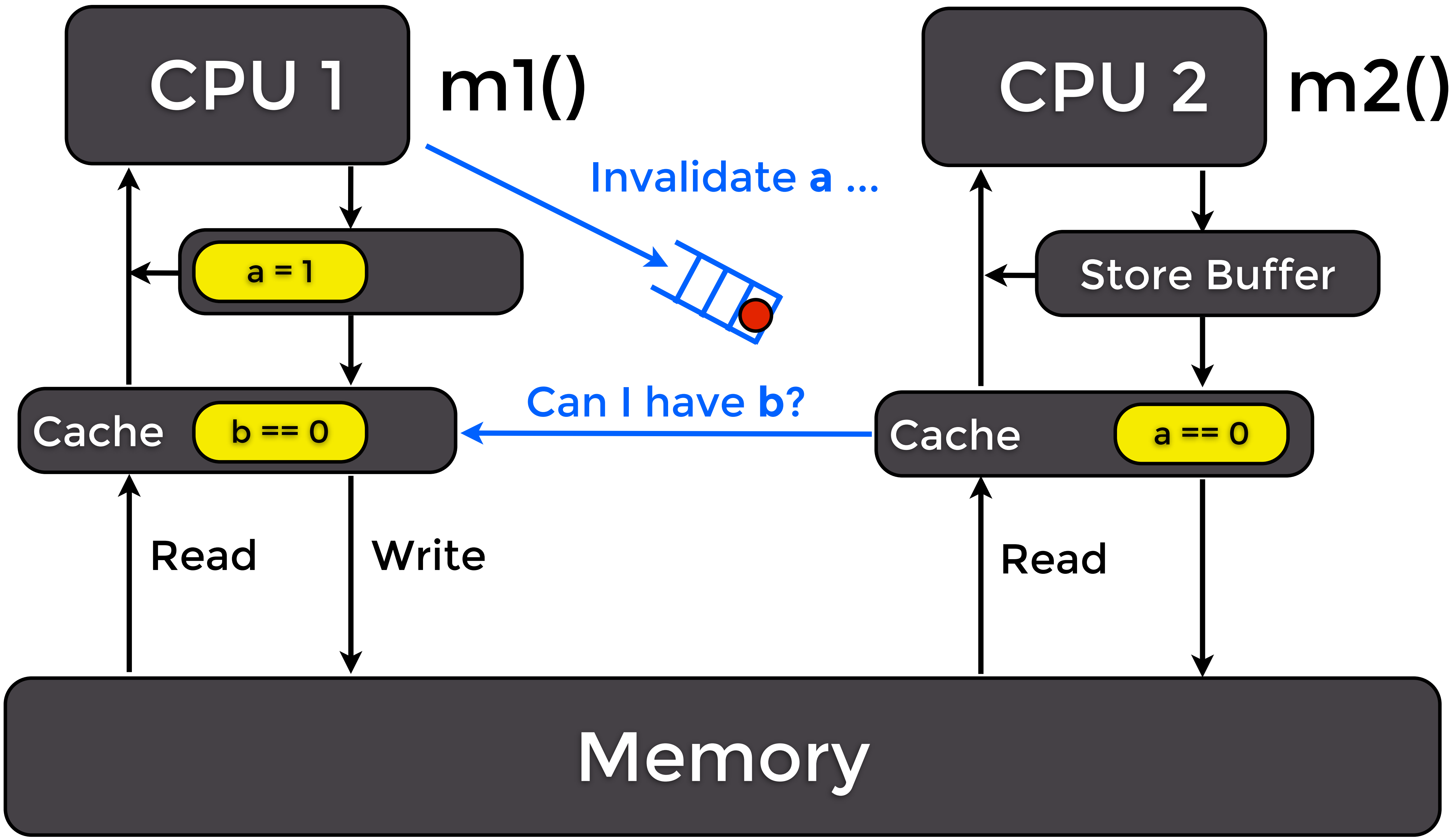
Initial State



Monday, December 30, 13

Caches communicate using MESI protocol. Time doesn't permit going into the protocol in detail, so we'll just cover one representative example. See the reference above.

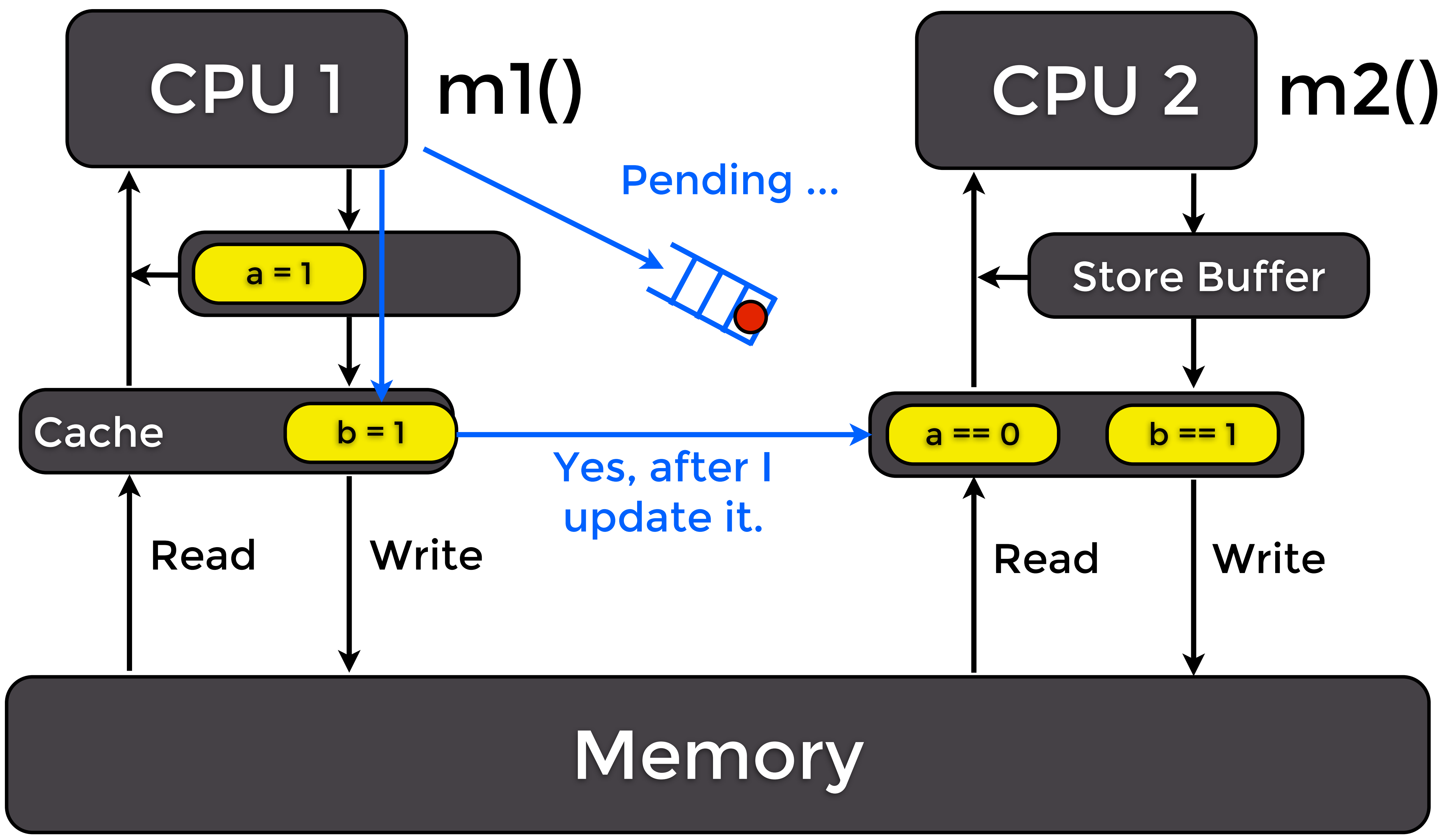
Step 1



Monday, December 30, 13

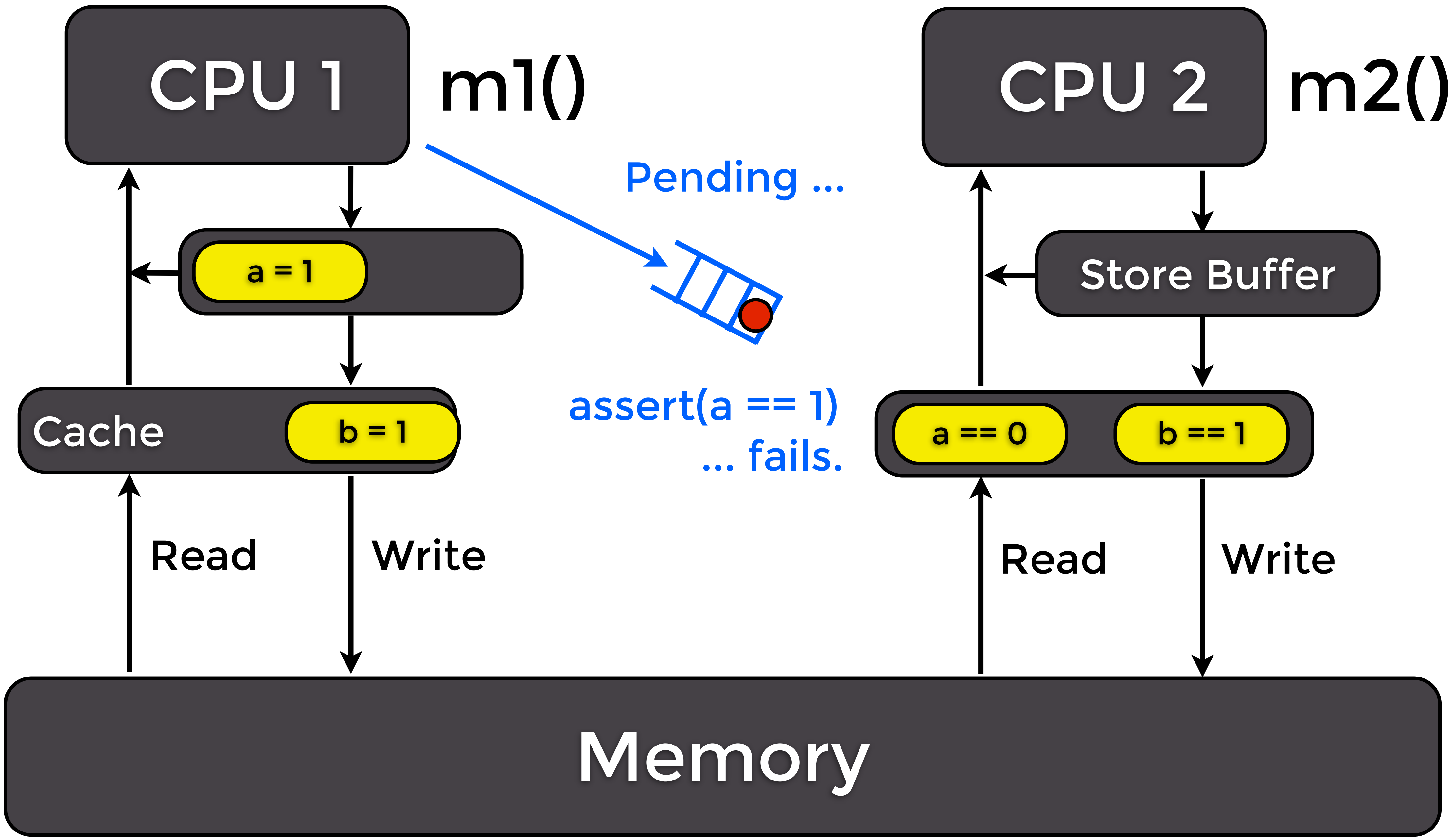
CPU 1 writes “a” and this write is held in the store buffer. CPU 1 also send an “invalidate” message to all other caches, but invalidate messages can be queued.

Step 2



CPU 1 then writes 1 to “b” and CPU 2 tries to test it. The timing happens to work out so that the updated value of “b” is provided to CPU 2. The MESI messages cross like ships in the night – another kind of “reordering”, this time by the hardware.

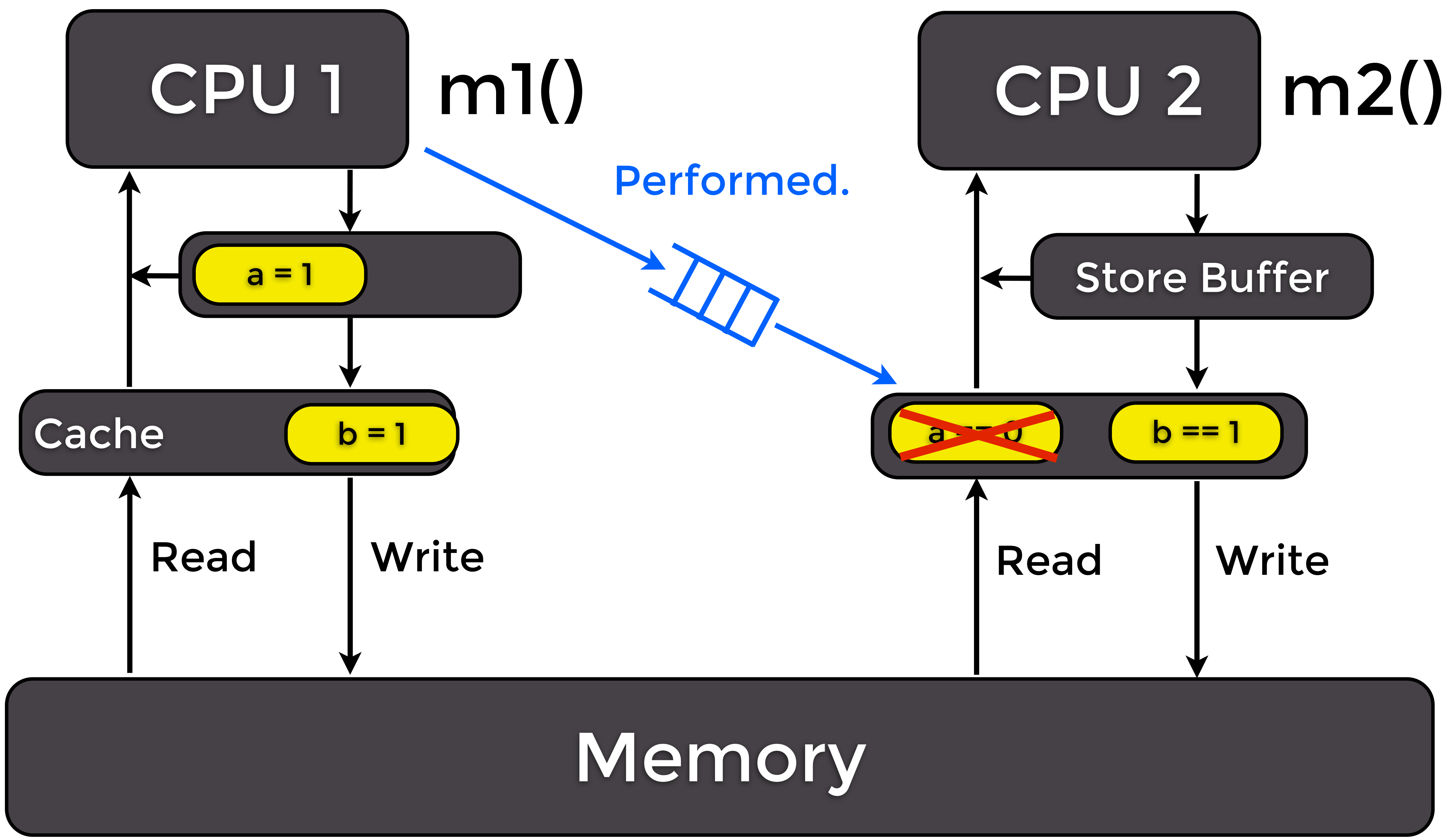
Sigh



Monday, December 30, 13

With b == 1 in hand, CPU 2 tests the variable a and fails.

Too Late!



If the compiler doesn't get you, the hardware still can *

Eventually the invalidate is processed, but the damage has been done.

Where Are We?

- The “intuitive rule of program order” is sufficient for single-threaded programs
- Applying the rule to each of multiple threads leads to surprising results
- For multiprocessors, we need better a better rule.

Happily, some of the best and brightest have thought about this.

This presentation is about memory “visibility” because writes that occur first on a given CPU do not necessarily become visible in the same order on another CPU. This makes it extremely difficult to reason about our programs.

“Good” and “Bad” Traces

- Why are some traces better than others?
- “Good” traces are equivalent to **some serial execution** of the parallel steps
- A parallel trace that is equivalent to **some serial execution** is said to be **Sequentially Consistent (SC).**

SC programs are programs we can reason about

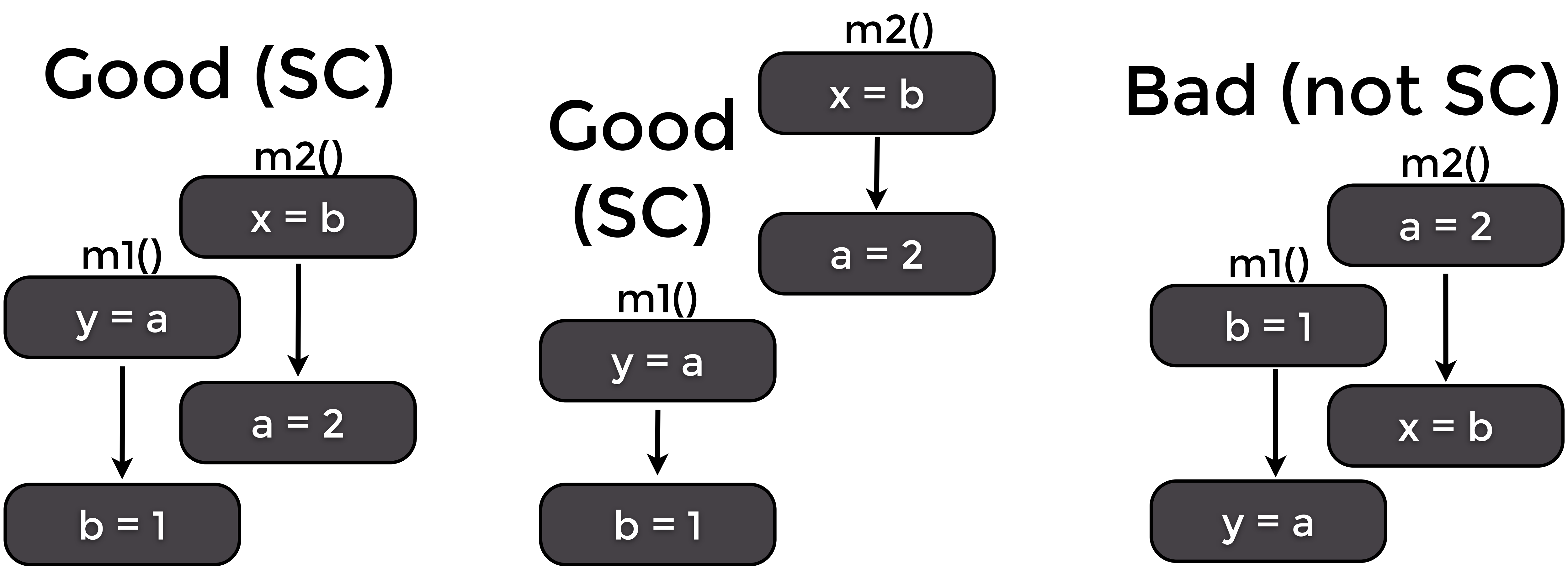
One way of explaining “some serial execution” is to imagine executing on one CPU with very rapid context switching. A trace that is possible on a single CPU with very rapid context switching is SC.

Traces (Example 1)

Given:

```
void m1() {
  y = a;
  b = 1;
}
```

```
void m2() {
  x = b;
  a = 2;
}
```



Intuitively, again, the left two traces are “good”, the third one not. To us, it’s common sense. The point to the memory model is to give the compiler, runtime and hardware some common sense. ;-)

Language Designer's Choices

- Could force **everything** to be SC ... but
  - Most actions by a thread are irrelevant to other threads
  - Significant performance penalty
  - Resulting language uncompetitive (?)
- Therefore ... as designed, Java requires help from the programmer to ensure SC.

Once the problem was understood, the designers of the Java language were faced with a choice: make everything SC (at huge cost), or require the programmer to define the synchronization points. They chose the latter.

Java Memory Model

- Introduced in Java 1.5 (2004)
- Section 17.4 and 17.5 of JLS
- Based on the concept of a **partial order**
  - Most memory operations are unordered
- **Happens-before** establishes ordering in specific memory operations

http://bit.ly/PJUG2013-JLS-17

The memory model itself reads like a math paper. It was not introduced until Java had been around for 8 years. Early Java virtual machines (through 1.4) could exhibit the kinds of confusing behaviors described in my example and the language spec did not make guarantees about how programmers could avoid them.

Two Audiences

- Compiler, JVM, and class library authors
  - **Java Memory Model**
  - "Concurrency Assembly Language"
- All the rest of us
  - Class Library
  - Idioms, patterns, and new languages

*

I.e. the language designers have two audiences: those who actually rely on the language spec, and those who *use* the language.

Example Rules from JMM

"Stmt-1 **happens-before** stmt-2 if stmt-1 precedes stmt-2 in program order."   [Note: this is "the intuitive rule of program order"]

"All memory operations prior to writing a volatile variable on one thread **happen-before** a read of the same volatile from another thread."

It's a Tufte Nightmare! (Too many words.) Sorry!

This is for the low level audience, e.g. compiler authors.

Modified Example 1

```
volatile int a, b, x, y; // all 0
```

<u>First CPU ("thread")</u>

```
void m1() {
  y = a;
  b = 1;
}
```

<u>Second CPU</u>

```
void m2() {
  x = b;
  a = 2;
}
```


This code is just example 1 but modified with a volatile declaration.

Another View

```
void m1() {
  y = a;
  b = 1;
}
```

Happens-Before

Happens-Before

```
void m2() {
  x = b;
  a = 2;
}
```

Happens-before is transitive, so if (y = a) **hb** (b = 1) and (b = 1) **hb** (x = b), then (y = a) **hb** (x = b).

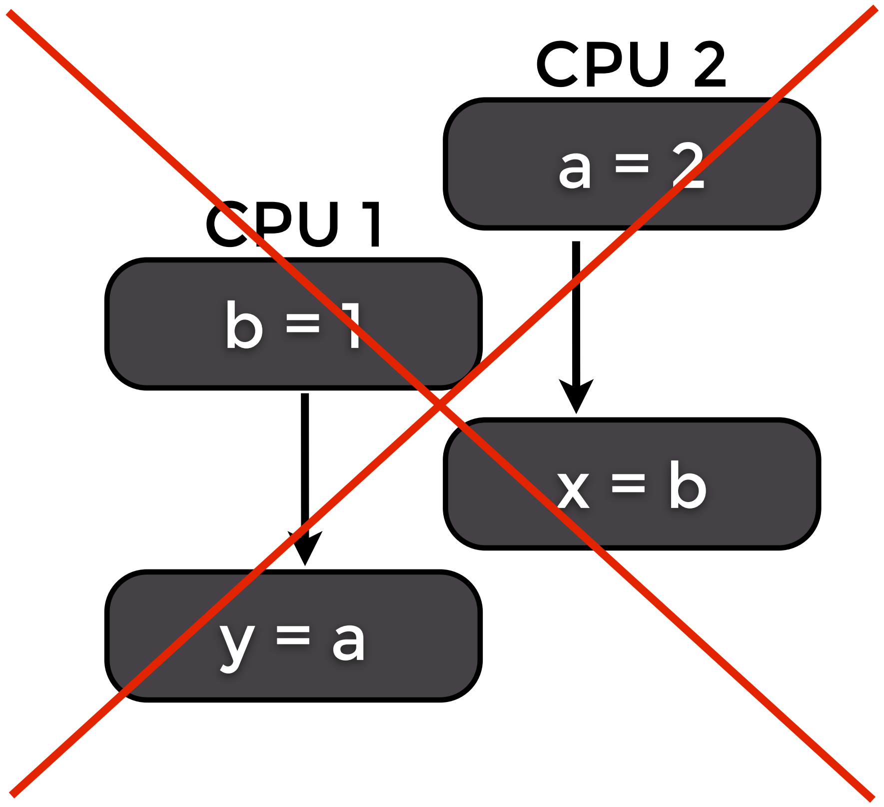
There are two relevant happens-before operations. Nothing stops m2() from executing first and capturing either y = a or nothing at all from m1(). But if m2() sees b == 1, it *must* also see y = a, and hence the counterintuitive non-SC execution ordering is prevented.

Result

The two **happens-before** operations mean that if CPU 2 observes **b = 1**, it must also observe **y = a**.

This rules out the non-SC trace.

Cannot Occur



*

The compiler and runtime must cooperate to ensure the hardware does not execute a non-SC trace. More properly, the hardware must not make the non-SC behavior visible to any software.

JMM Guarantee

- JMM rule: when one thread writes data another will read, a **happens-before** must separate the write and read.
- Java programs that follow the rule are sequentially consistent.
- Otherwise, the program is said to have a **data race**.
- Programmer's task is to avoid data races.

*Examples 1 and 2 contain **data races**.*

We can say: "A sequentially consistent program contains no data races."

What Does Volatile Do?

- Cause javac to avoid reordering optimizations, so preventing Example 1.
- Generated bytecode does not change
- JIT sees volatile annotation on variables and generates machine-specific barrier instructions to prevent Example 2.
- Also, mutex implementation must cause execution of machine-specific barriers.

http://bit.ly/PJUG2013-Memory-Barriers

The focus on “volatile” here is, again, just an example. Acquiring and releasing a lock has the same effect on memory visibility, so long as the “other” thread synchronizes on the same lock.

For the Rest of Us

- Idioms
  - E.g. Initialization on demand holder (http://bit.ly/PJUG2013-Holder)
- Patterns
  - Proper construction, publication, etc.
- New technology
  - Languages, Frameworks, Java 8, etc.

Key Patterns

- Proper Construction
- Immutability
- Safe Publication
- Concurrent Collections
- Documentation
- Doing it yourself: `volatile`, `synchronized`, etc.

Java Concurrency in Practice: http://jcip.net

Proper Construction

- The closing curly brace of a constructor is a special point in program execution.
- `this` object must never be published before its constructor is complete
  - Traditional fail: event registration
  - Common remedy: static factory method

When a class instance creates and starts a thread, the thread often has some ref to the class that created it. Executor framework may solve the problem for threads, but then it's likely the **task** will have a reference to its creating class instead.

Immutability

- Immutable object = all fields final and no way to modify state of contained objects.
- Properly constructed immutable objects are thread safe
  - May be passed between threads “willy nilly”

Safe Publication

- Create object in static initializer
- Hold ref in `volatile` or `AtomicReference`
- Hold ref in `final` field of some other properly constructed object
- Pass ref through a field guarded by a lock (e.g. a synchronized accessor)

There Are Only Four Ways To Do It.

Important note: the last bullet (field guarded by a lock) includes publishing the object by placing it in any kind of properly-synchronized collection or holder.

Use the Class Library

- Learn what's in java.util.concurrent!
- Use concurrent collections for safe publication.
- Do not roll your own operations similar to the ones offered by atomics

Documentation

- Some widely-used libraries and frameworks handle this badly
- Describe thread safety of each class and/or method in Javadoc
- Use JSR-305 or similar annotations if they are available to you.

Doing It Yourself

- Use `volatile` (or `atomic`*`Type`*) for single items of state
  - Same memory visibility guarantees
  - Use `atomic` if methods are helpful
- Intrinsic locking (`synchronized`) when multiple items must be kept consistent
- Threads must refer to the **same** volatile or lock.

http://bit.ly/PJUG2013-FAQ *

New JVM Languages

- Scala
  - Functional language on JVM
    - Potentially huge advantages
  - Scala dev team must deal with JMM
- Java 8 (not a new language exactly)
  - Closures, streams, Spliterators, etc

http://bit.ly/PJUG2013-Scala-Issue

New JVM Frameworks

- Example: Akka
  - Actor (event) framework on JVM
  - Beautiful docs about memory model: http://bit.ly/PJUG2013-Akka-Jmm
- Dalvik (Android "Java" virtual machine)
  - History of issues - maybe better now
  - Details: http://bit.ly/PJUG2013-Dalvik

Mention of Akka is an example. There are many others.

Non-JVM Languages

- C
  - You’re on your own. Distinct compile and runtime tools. Example follows.
- C++
  - Developing (have?) a memory model
- C#
  - Similar to Java, but docs don’t allow for a precise comparison

“You’re on your own” is not intended as a criticism or slam (some of my best friends are C programmers). ;-) Correctness in C requires more knowledge of low level details than Java. But the robustness of the Linux kernel shows that high quality implementations are possible.

Explicit Control in C

- Compiler directives/annotations to prevent aggressive compiler reordering
- Linux kernel: macros expand to explicit memory barrier instructions

```
void m1(void) {
  stmt-1;
  stmt-2;
  smp_mb();
}
```

http://bit.ly/PJUG2013-C-Linux-Example

And More Languages

- Go
  - Memory model uses terminology and concepts from JMM
  - http://bit.ly/PJUG2013-Go-MM
- Rust? Objective C? GPU code?
  - Left as exercise for the reader. ;-)

Summary

- These issues affect all languages that support programming with threads
- Java community was ahead of the curve in addressing them
- Awareness wins - you may not program against the JMM, but understanding it is powerful.
- Keep learning - avoid “DIY” and use the highest level tools you can.

References

http://bitly.com/bundles/pdxjjb/2

Contains all the “bit.ly” links
from this presentation

There are some links in the bit.ly bundle that didn’t make it into any slide.

THANK YOU

- Java Agent team and so many others at New Relic for attending my practice talks and providing feedback

Q&A

Followed By

Implementation of the Asynchronous Hopped Products Pattern