Ordering Read and Write Operations on a Multicore CPU

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR
@JosePaumard http://blog.paumard.org/

Agenda

We saw what synchronization is

This module is about visibility

Bound to the way multicore CPUs work

What is the happens before link

What does volatile mean

The impact of false sharing on code

Synchronization and Visibility

Synchronization

Synchronization protects a block of code

Guarantees this code is executed by one thread at a time

Prevents race condition

Race Condition

```java
public void consume() {
    synchronized(lock) {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}
```

```java
public void produce() {
    synchronized(lock) {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}
```

Race Condition

count

```
public void consume() {
    synchronized(lock) {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}
```

```
public void produce() {
    synchronized(lock) {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}
```

Race Condition

count

```
public void consume() {
    synchronized(lock) {
        while (isEmpty(buffer)) {}
        buffer[--count] = 0;
    }
}
```

1) **reads** count **from memory**
2) **decrement it**
3) **writes** count **back to memory**

Race Condition

count

1) **reads** count **from memory**
2) **increment it**
3) **writes** count **back to memory**

```
public void produce() {
    synchronized(lock) {
        while (isFull(buffer)) {}
        buffer[count++] = 1;
    }
}
```

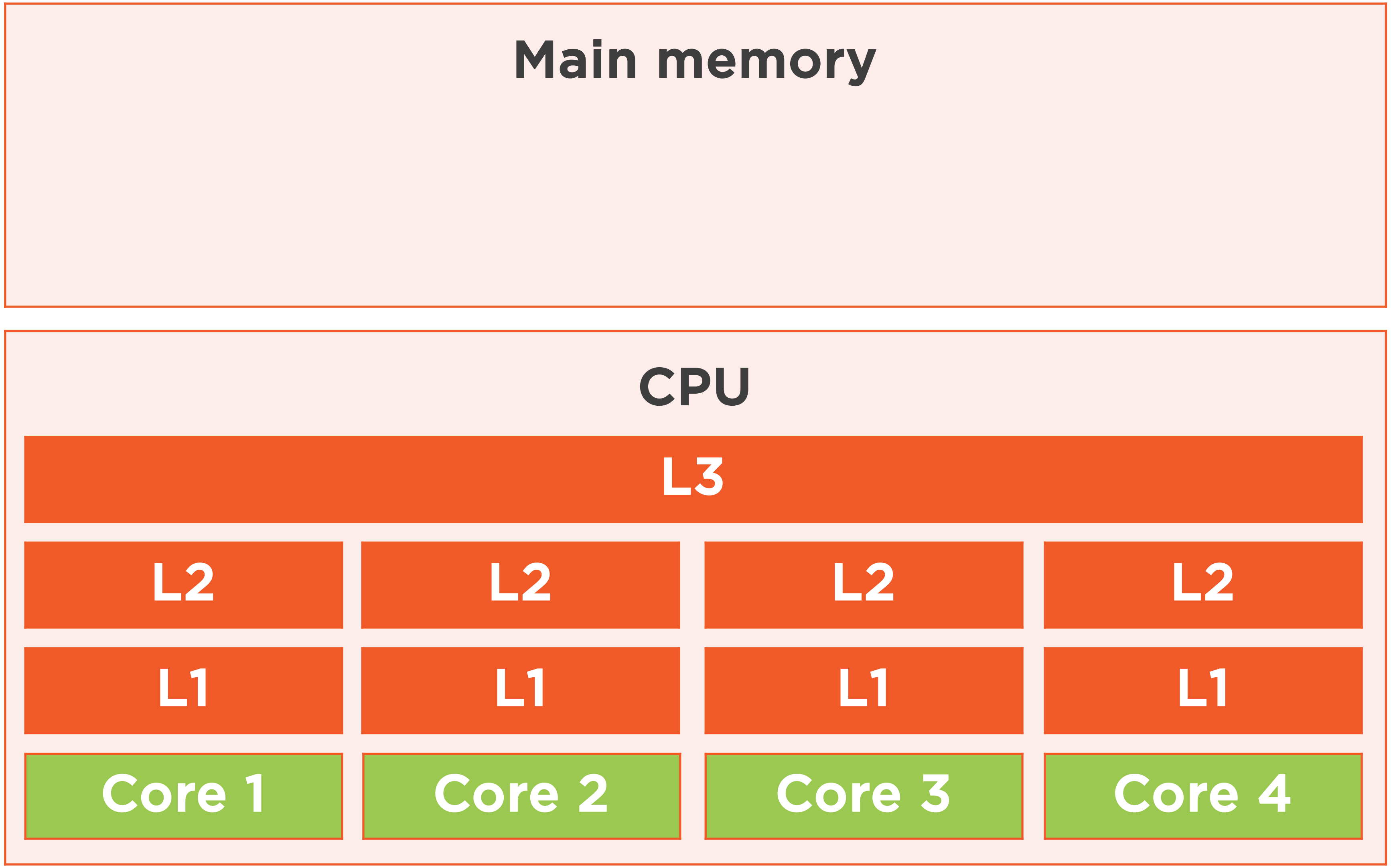
Memory Access

20 years ago, when CPU had no cache, this code was working fine

But nowadays, things do no work like that any more!

A CPU does not read a variable from the main memory, but from a cache

CPU Architecture



Why Has It Been Made Like That?

Because access to caches is much faster than access to main memory

Access to the main memory: ~100ns

Access to the L2 cache: 7ns

Access to the L1 cache: 0.5ns

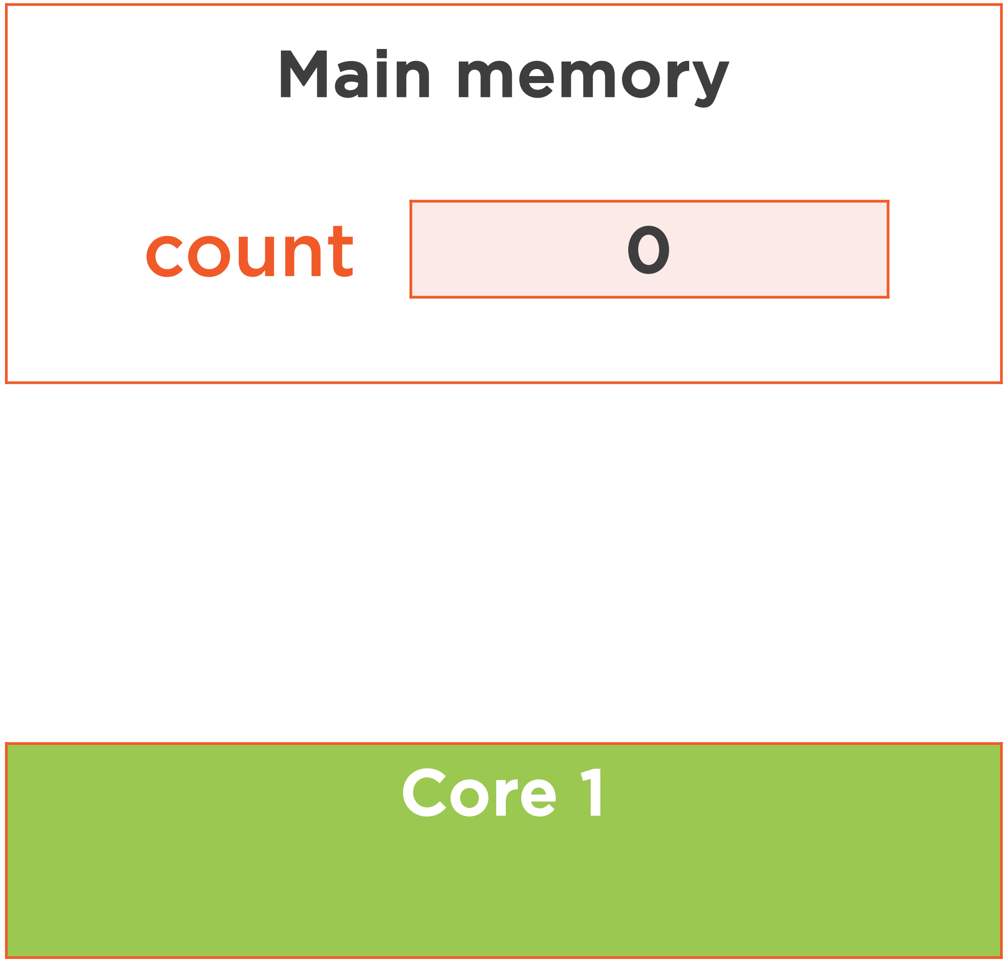
There Are Tradeoffs

Size of the main memory: several GB

Size of the L2 cache: 256kB

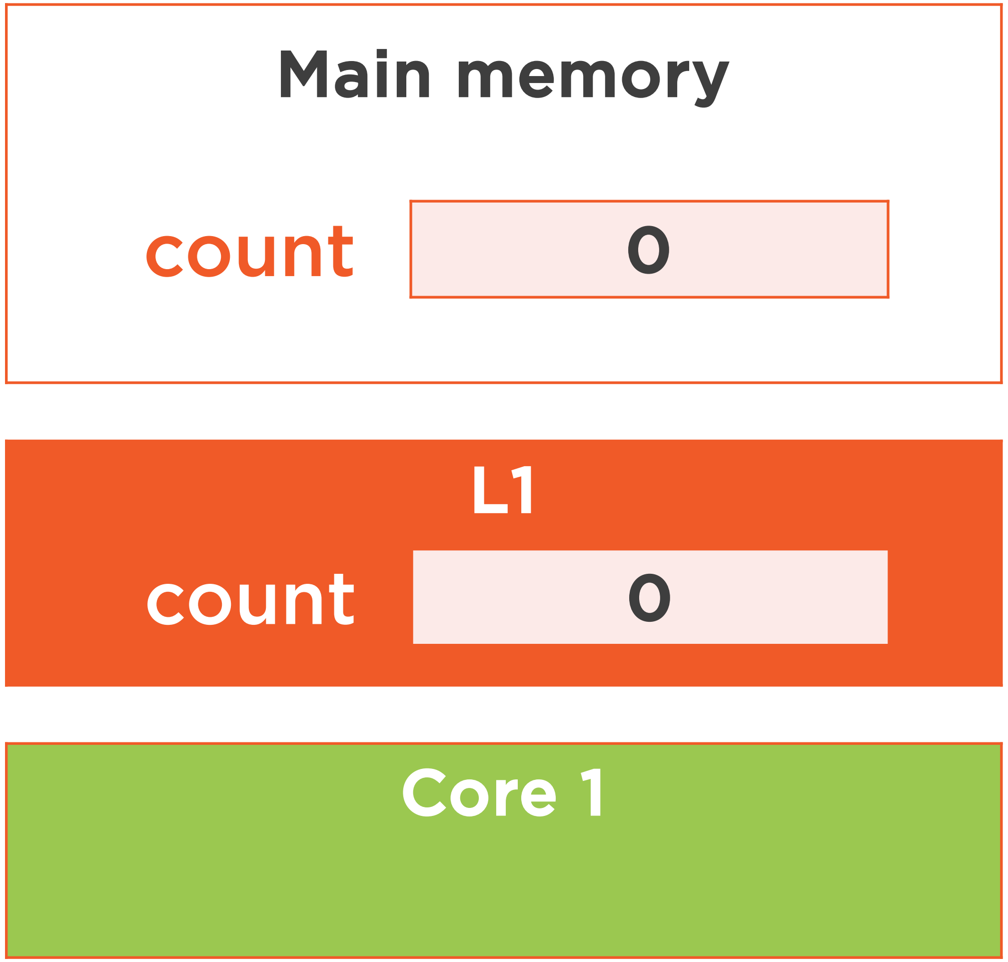
Size of the L1 cache: 32kB

CPU Architecture



Core 1 needs count

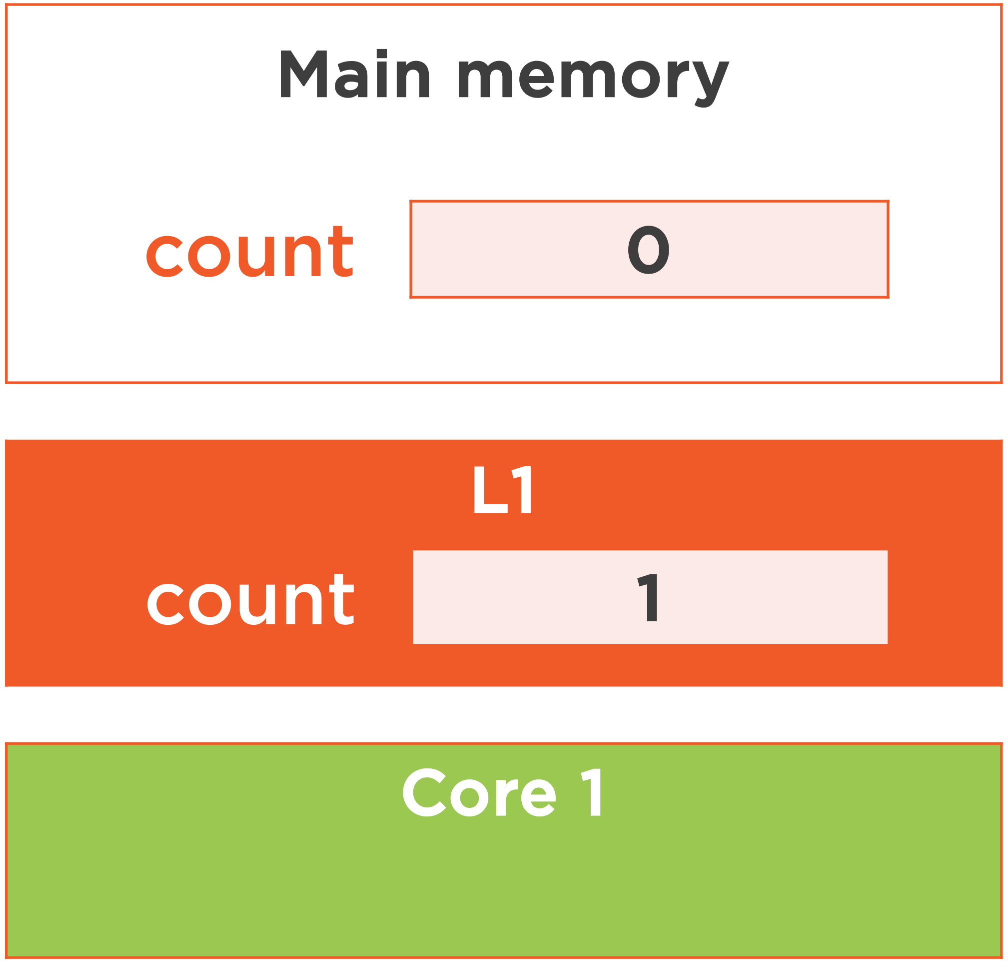
CPU Architecture



Core 1 needs count

1) The variable is copied in L1

CPU Architecture

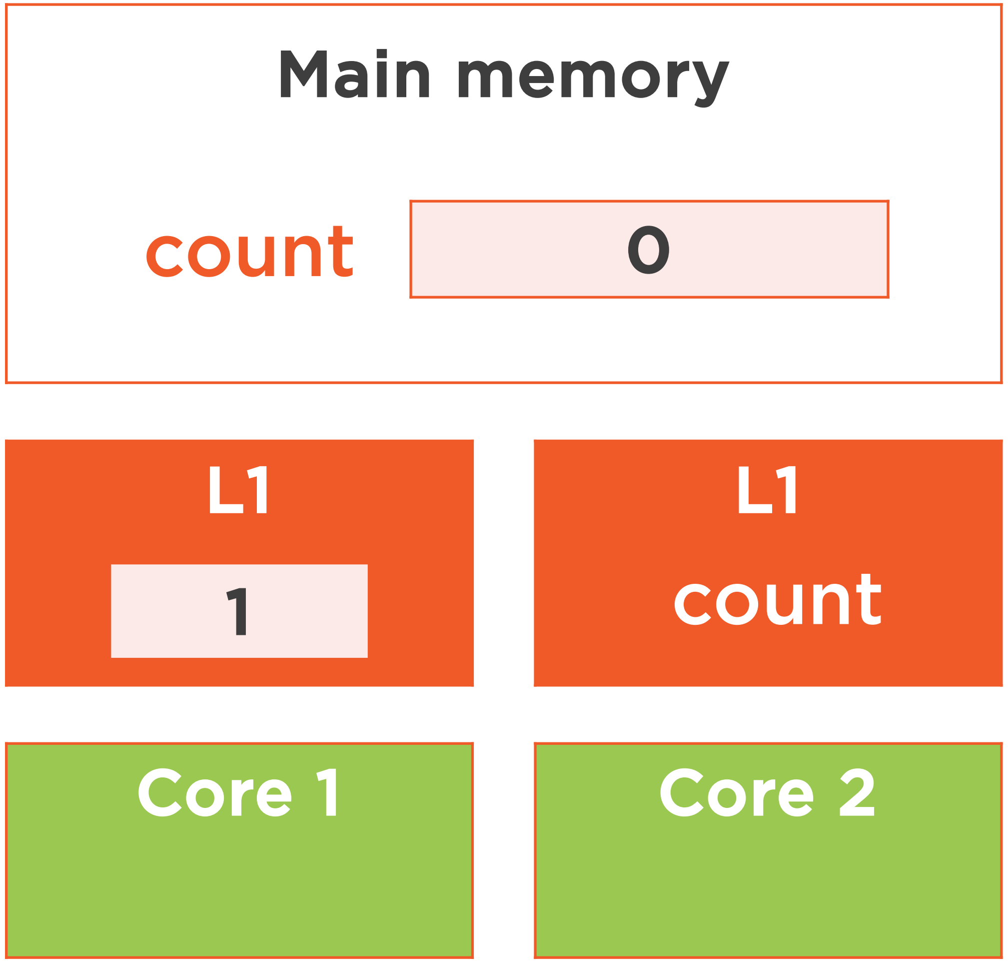


Core 1 needs count

1) The variable is copied in L1

2) Core 1 can modify it

CPU Architecture

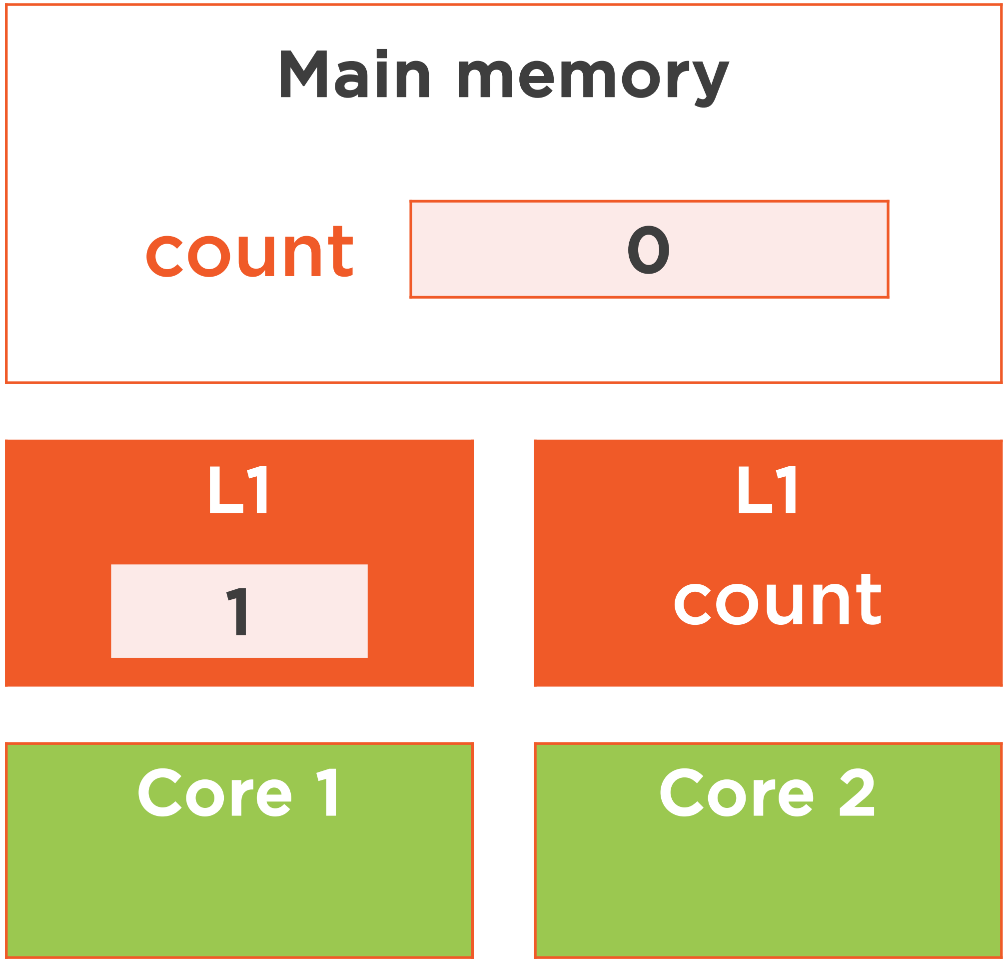


Core 1 needs count

1) The variable is copied in L1
2) Core 1 can modify it

3) Core 2 also needs count

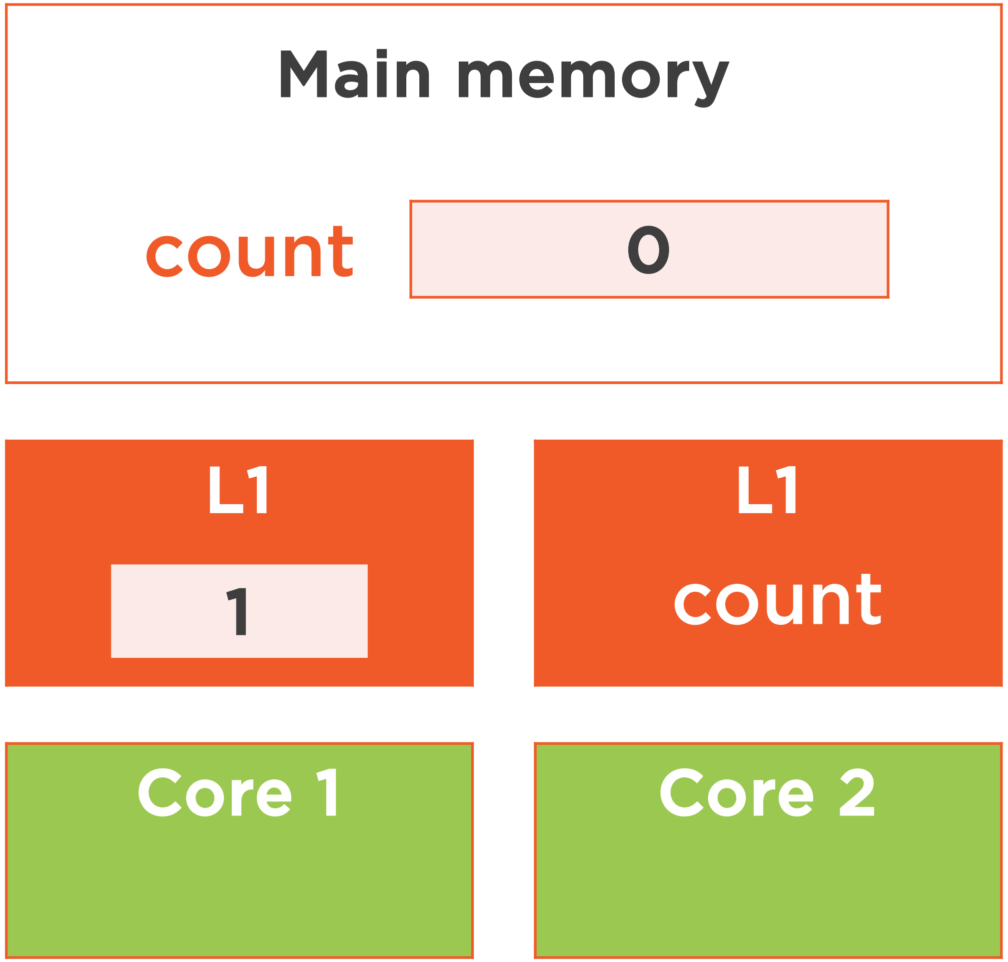
CPU Architecture



Core 1 needs count

1) The variable is copied in L1
2) Core 1 can modify it
3) Core 2 also needs count
4) It should get the value 1, not 0

CPU Architecture



Core 1 needs count

1) The variable is copied in L1
2) Core 1 can modify it
3) Core 2 also needs count
4) It should get the value 1, not 0

This is visibility!

Visibility

A variable is said visible

If the writes made on it are visible

All the synchronized writes are visible

“Happens Before” Link

Happens Before

There are multiple references to *happens-before* in the Javadoc

Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a `BlockingQueue` *happen-before* actions subsequent to the access or removal of that element from the `BlockingQueue` in another thread.

This interface is a member of the Java Collections Framework.

Since:

1.5

Memory consistency effects: Actions in a thread prior to calling `await()` *happen-before* actions that are part of the barrier action, which in turn *happen-before* actions following a successful return from the corresponding `await()` in other threads.

Since:

1.5

Memory consistency effects: Until the count reaches zero, actions in a thread prior to calling `countDown()` *happen-before* actions following a successful return from a corresponding `await()` in another thread.

Since:

1.5

The Java Memory Model

Multicore CPU brings new problems

Read and writes can really happen at the same time

A given variable can be stored in more than one place

Visibility means “a read should return the value set by the last write”

What does last mean in a multicore world?

The Java Memory Model

We need a timeline to put read and write operations on

The Java Memory Model

Thread T_1

x = 1

1) T_1 writes 1 to x

The Java Memory Model

Thread T_1

x = 1

Thread T_2

r = x

1) T_1 writes 1 to x

2) T_2 reads x and copy it to r

The Java Memory Model

Thread T_1

```
x = 1
```

Thread T_2

```
r = x
```

1) T_1 writes 1 to x
2) T_2 reads x and copy it to r

3) What is the value of r?

The Java Memory Model

Thread T_1

```
x = 1
```

Thread T_2

```
r = x
```

1) T_1 writes 1 to x
2) T_2 reads x and copy it to r
3) What is the value of r?

The Java Memory Model fixes the answer to this question

The Java Memory Model

Thread T_1

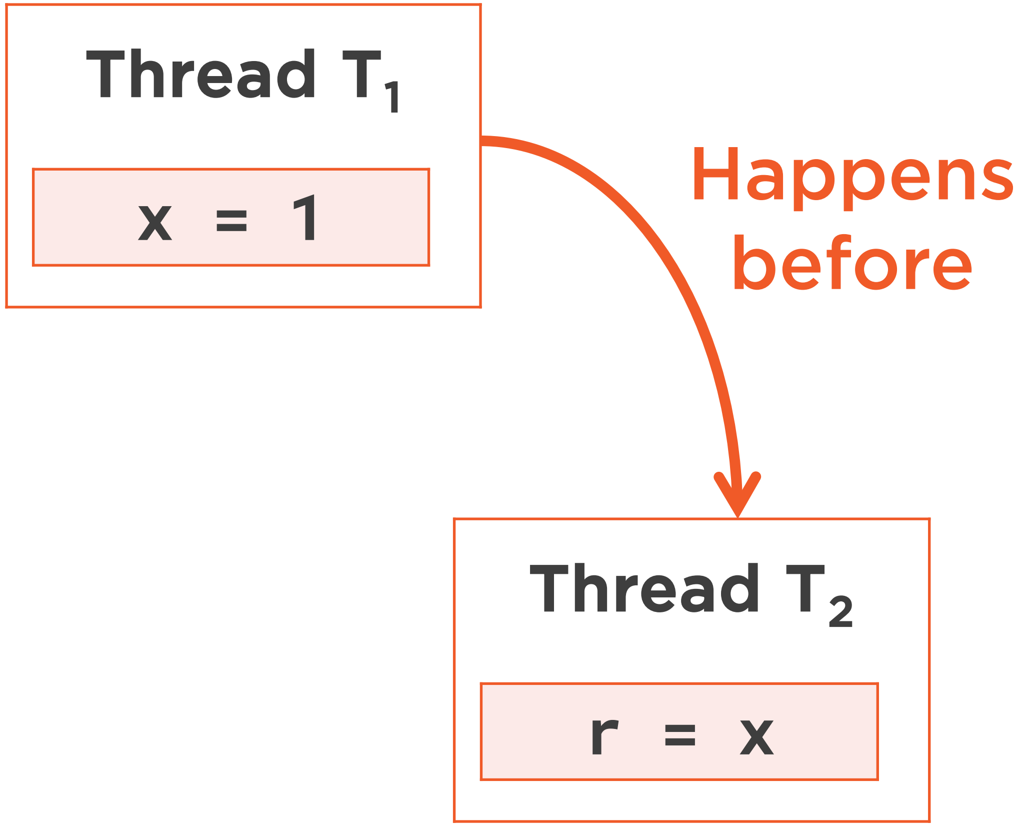
```
x = 1
```

Thread T_2

```
r = x
```

If there is no “happens before” link between the two operations, the value of `r` **is unknown**

The Java Memory Model



If there is no “happens before” link between the two operations, the value of r is unknown

If there is a “happens before” link between the two operations, the value of r is 1

How Can We Set up a “Happens Before” Link?

There is no such keyword in the Java language...

Happens before link

A “happens before” link exists between all synchronized or volatile write operations and all synchronized or volatile read operations that follow

Happens Before Examples

```java
int index;

void increment() {
    index++;
}

void print() {
    System.out.println(index);
}
```

Two operations:

1) a write operation
2) a read operation

Happens Before Examples

```java
int index;

void increment() {
     index++;
}

void print() {
     System.out.println(index);
}
```

Two operations:

1) a write operation
2) a read operation

What does this code print in multithread?

Happens Before Examples

```java
int index;

void increment() {
    index++;
}

void print() {
    System.out.println(index);
}
```

Two operations:

1) a write operation

2) a read operation

What does this code print in multithread?

No synchronization, no volatility → impossible to say

Happens Before Examples

```java
int index;

void synchronized increment() {
    index++;
}

void synchronized print() {
    System.out.println(index);
}
```

Two operations:

1) **a write operation**
2) **a read operation**

What does this code print in multithread?

Synchronization → the correct value is always printed

Happens Before Examples

```java
volatile int index;

void increment() {
    index++;
}

void print() {
    System.out.println(index);
}
```

Two operations:

1) a write operation

2) a read operation

What does this code print in multithread?

The variable is volatile → the correct value is always printed

A More Complex Example

```java
int x, y, r1, r2;
Object lock = new Object();
```

```java
void firstMethod() {
    x = 1;
    synchronized(lock) {
        y = 1;
    }
}
```

```java
void secondMethod() {
    synchronized(lock) {
        r1 = y;
    }
    r2 = x;
}
```

`firstMethod()` **is writing** `x` **and** `y`

`secondMethod()` **is reading them**

They are executed in threads T_1 and T_2

Question: what is the value of `r2`**?**

A More Complex Example

```java
int x, y, r1, r2;
Object lock = new Object();
```

```java
void firstMethod() {
    x = 1;
    synchronized(lock) {
        y = 1;
    }
}
```

```java
void secondMethod() {
    synchronized(lock) {
        r1 = y;
    }
    r2 = x;
}
```

Due to the way the code is written, there is a *happens-before* link between:

- `x = 1` **and** `y = 1`
- `r1 = y` **and** `r2 = x`

A More Complex Example

```
int x, y, r1, r2;
Object lock = new Object();
```

```
void firstMethod() {
    x = 1;
    synchronized(lock) {
        y = 1;
    }
}
```

```
void secondMethod() {
    synchronized(lock) {
        r1 = y;
    }
    r2 = x;
}
```

If T_1 is the first to enter the synchronized block, then the execution is in this order:

- x = 1
- y = 1
- r1 = y
- r2 = x

Happens-before link between a synchronized write and a synchronized read

The value of r2 is 1

A More Complex Example

```
int x, y, r1, r2;
Object lock = new Object();
```

```
void firstMethod() {
    x = 1;
    synchronized(lock) {
        y = 1;
    }
}
```

```
void secondMethod() {
    synchronized(lock) {
        r1 = y;
    }
    r2 = x;
}
```

If T_2 is the first to enter the synchronized block, then the execution is in this order:

- `r1 = y`
- `r2 = x` **or** `x = 1`**?**
- `y = 1`

No happens-before link between r2 = x and x = 1

The value of `r2` **may be 0 or 1**

Synchronization

Guarantees the exclusive execution of a block of code

Synchronization

Guarantees the exclusive execution of a block of code

Visibility

Guarantees the consistency of the variables

All shared variables should be accessed in a synchronized or a volatile way

False Sharing

What Is “False Sharing”?

False sharing happens because of the way the CPU caches work

It is a side effect, that can have a tremendous effect on performance

What Is “False Sharing”?

The cache is organized in lines of data

Each line can hold 8 longs (64 bytes)

When a visible variable is modified in an L1 cache, all the line is marked “dirty” for the other caches

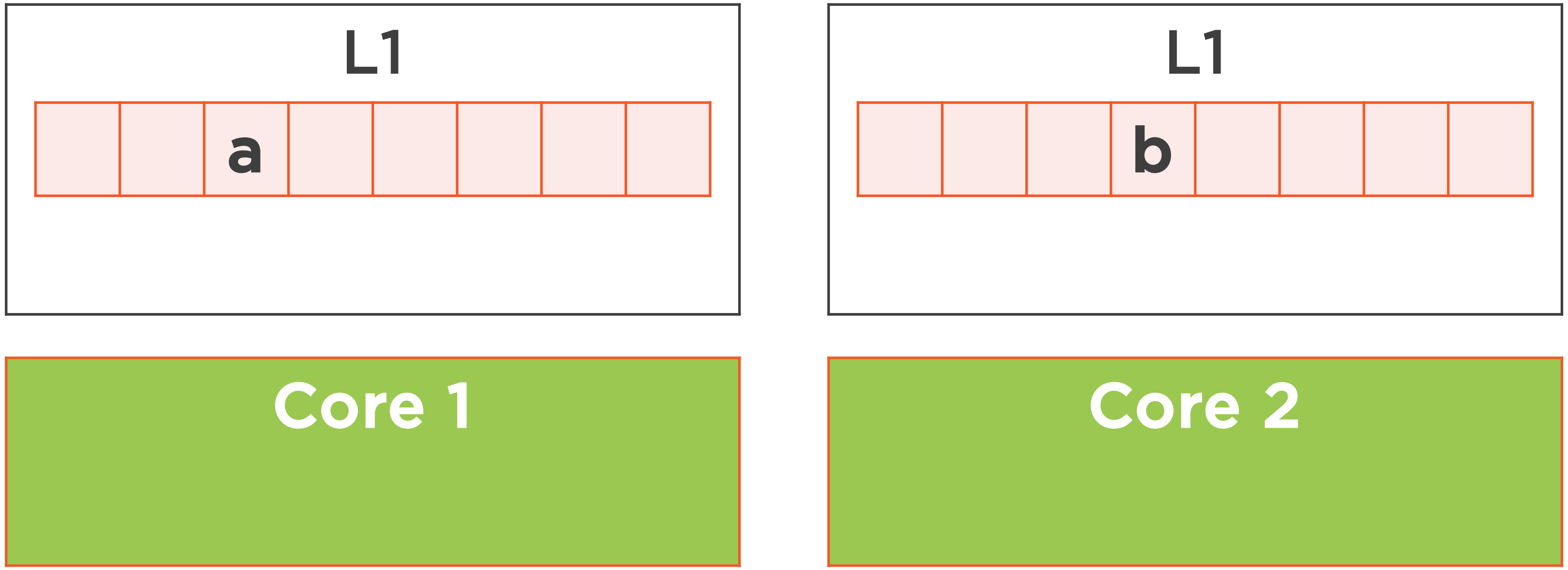
A read on a dirty line triggers a refresh on this line

From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```

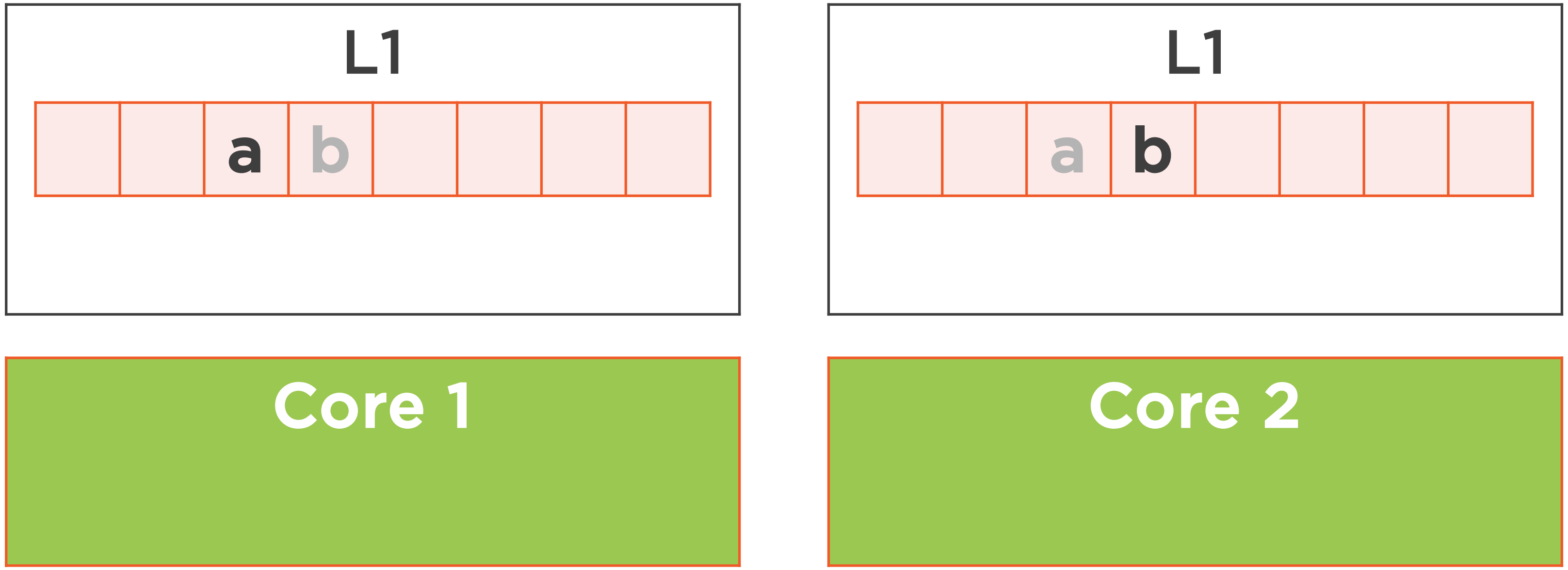


From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```

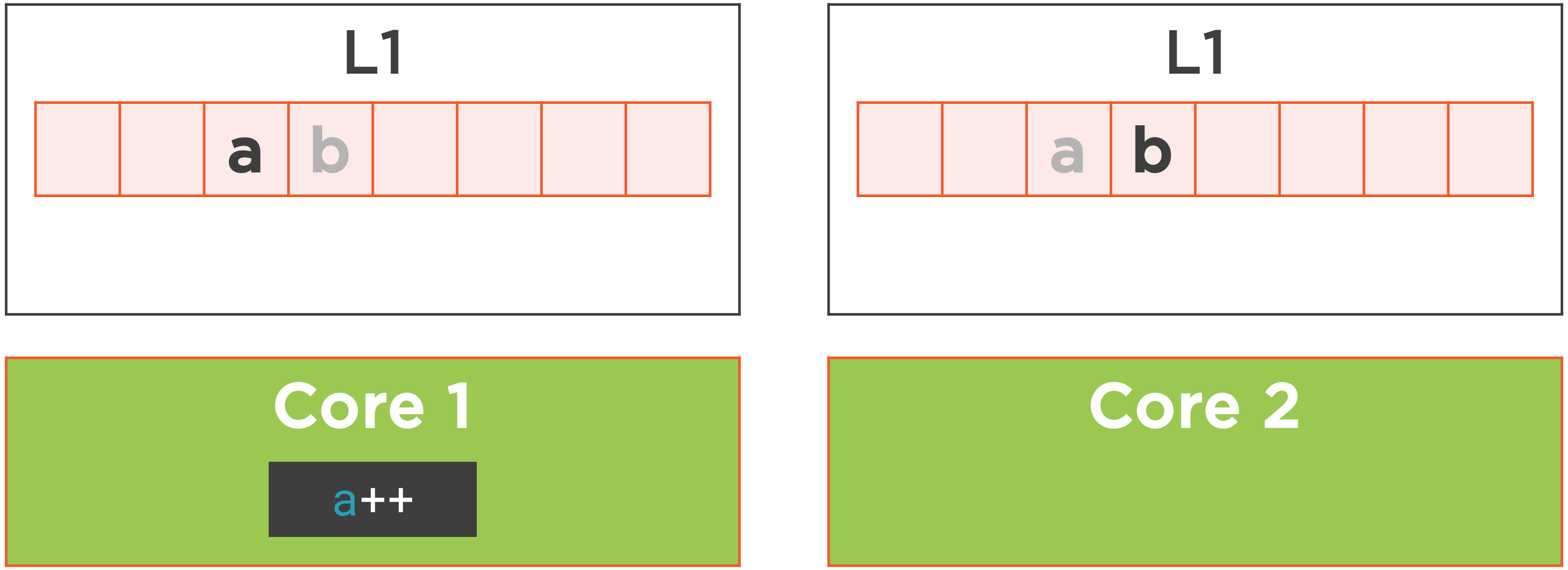


From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```

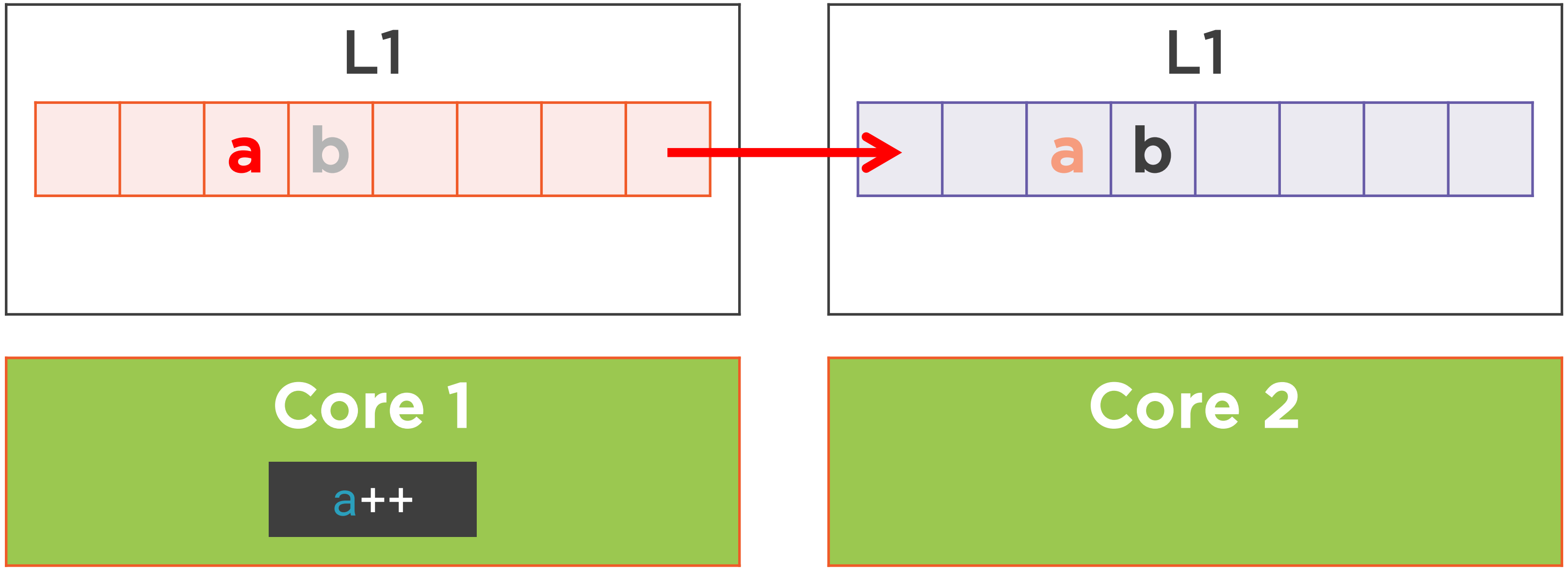


From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
    a++;
}
```

```
void secondMethod() {
    b++;
}
```



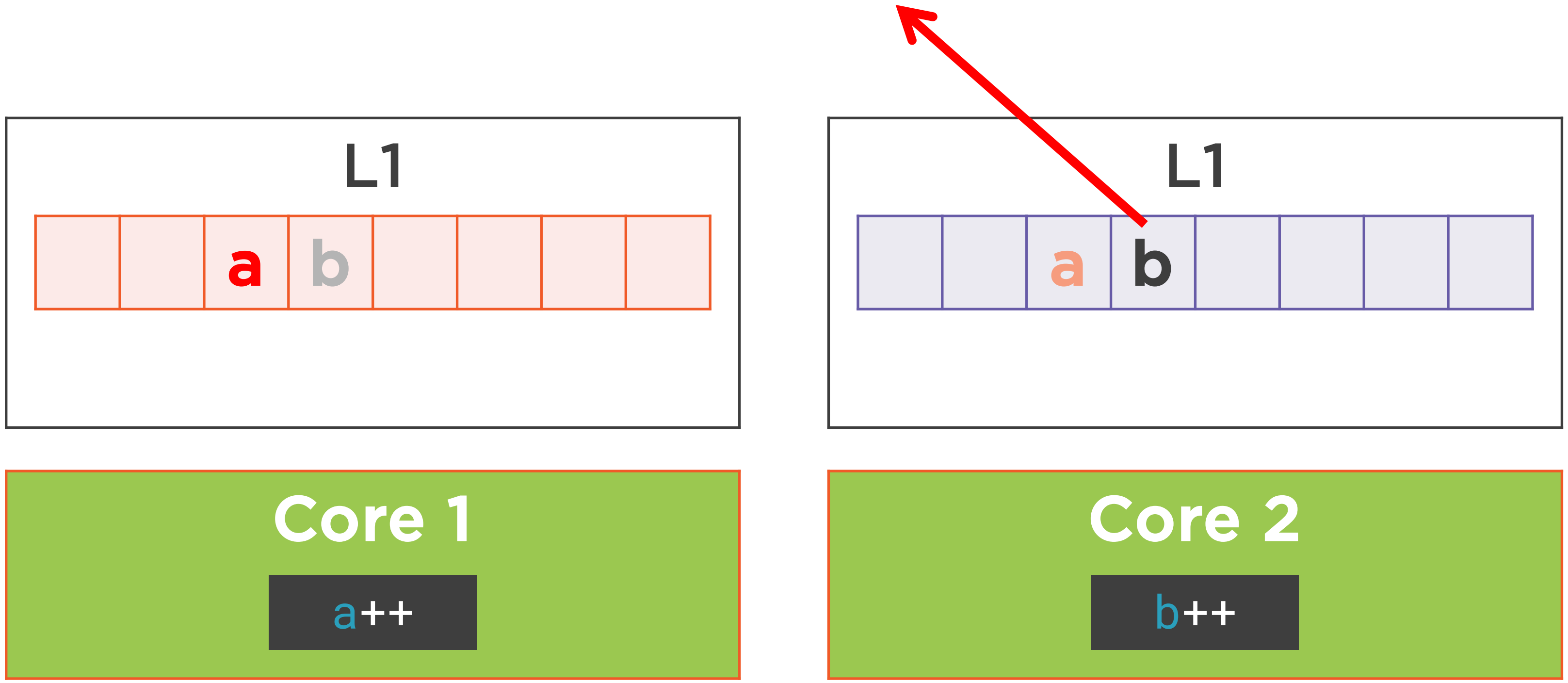
From Main Memory to L1 Cache

```
volatile long a, b;
```

```
void firstMethod() {
     a++;
}
```

```
void secondMethod() {
     b++;
}
```

Cache miss,
go to main memory



False Sharing

False sharing happens in an invisible way

Hard to predict, hits your performance

There are workarounds though…

Demo

Let us see some code!

An example of false sharing and the influence of variable padding

Wrapup

What did we learn?

The importance of visibility in multicore CPU

Two fundamental notions in concurrent applications: synchronization and volatility

How the way caching works in CPU can generate false sharing