Sharing Memory Across Threads

David Flynn
SOFTWARE ENGINEER, FLYNN IT LTD

Inter-Thread Communication

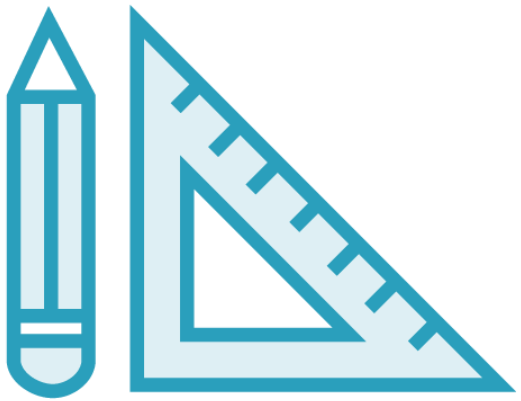
Threads need to share their data

More involved than making values visible

Value must also be correct:

- Issues with caching [reading stale values]
- Problems with compiler 'trickery'

Good Design Is Key



Important to be aware of this material

- Need to consider how to share data at the design stage
- Writing code without a plan will lead to difficulties

Bugs can be:

- Subtle
- Difficult to track down
- Only turn up occasionally

You'll Gain an Appreciation of ...

Problems with incorrectly synchronized data over four examples

How to safely publish immutable objects to threads

Synchronization guarantees in Java Memory Model

- Java Language Specification (chapter 17)

Thread-safe

Code executes correctly in a multithreaded environment

What Is Shared State?

Program data

- Has been stored
- Can be updated

Potentially shareable:

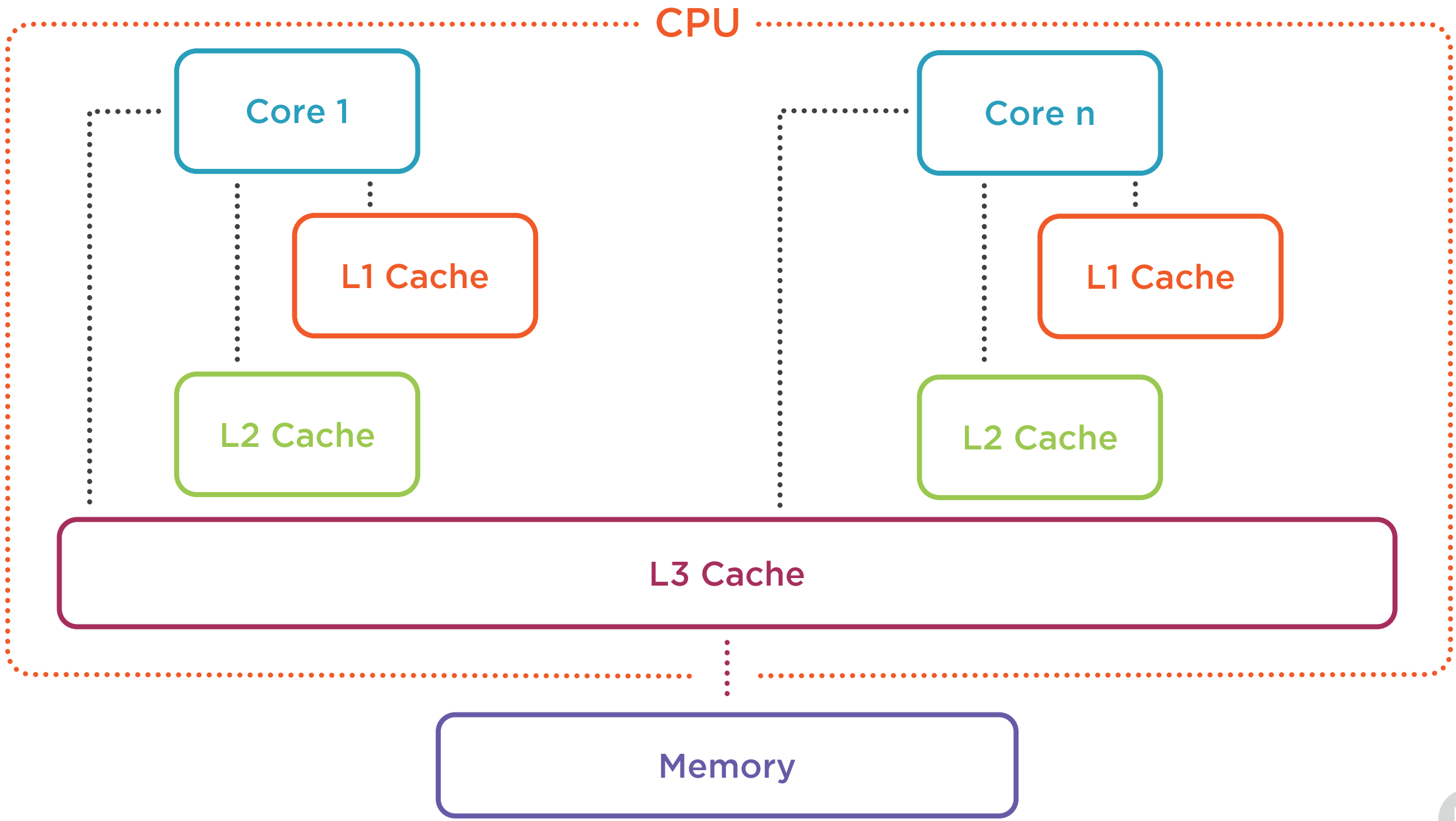
- Visible to other threads
- Static variables, class data, composed class data, array items, collection items
- Not local variables

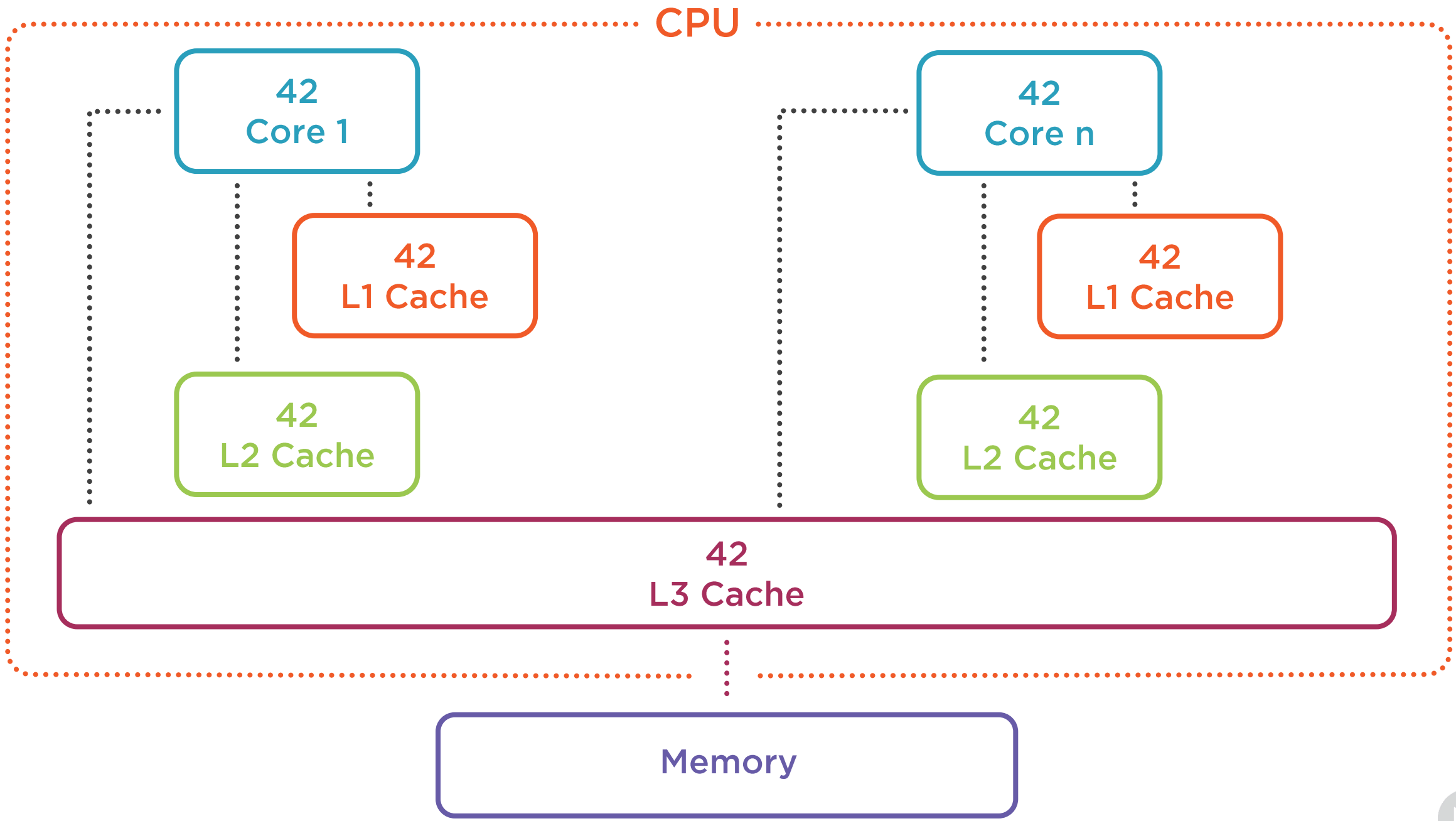
Published Data

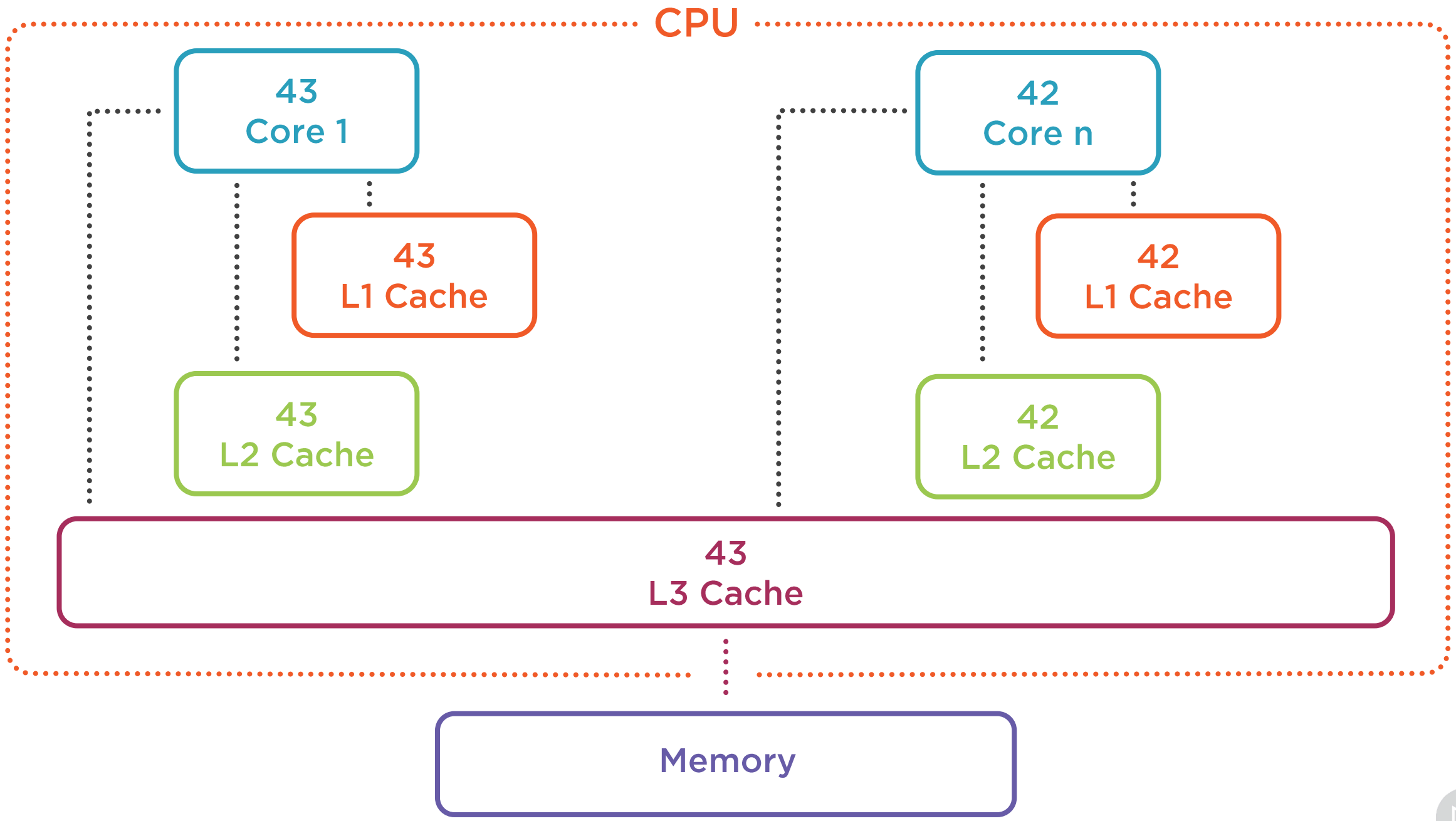
Data exposed to other threads

Must follow guidelines in the Java Memory Model

- Or suffer the consequences [subtle bugs]


CPU
Core 1
L1 Cache
L2 Cache
Core n
L1 Cache
L2 Cache
L3 Cache
Memory


CPU
42
Core 1
42
L1 Cache
42
L2 Cache
42
Core n
42
L1 Cache
42
L2 Cache
42
L3 Cache
Memory


CPU
43
Core 1
42
Core n
43
L1 Cache
42
L1 Cache
43
L2 Cache
42
L2 Cache
43
L3 Cache
Memory

Is Data Correctly Published?

Final fields safe?

- Yes, but see caveat later

Knowing data is published correctly

- Need to inspect the program
- All contained data from references must be correctly published

Java Memory Model

Listen to the definition:

- Does it leave you a little confused?

Java Memory Model

‘The Java memory model describes how threads in the Java programming language interact through memory.’
- Wikipedia

‘The Java Memory Model defines a set of guarantees which, when applied to a program, ensure memory interactions between threads occur in a specified deterministic fashion.’

Data Synchronization

When we write a value on one thread

- We want to know the value will be correctly visible when read by another

Data correctly synchronized

- Implies correct visibility

Not to be confused with synchronized keyword

Pseudocode Convention

Examples:

- L1 – local variable L1
- S2 – shared variable S2
- S2.X – field X of S2
- 1.2 – thread 1, statement 2

Variables start with default values

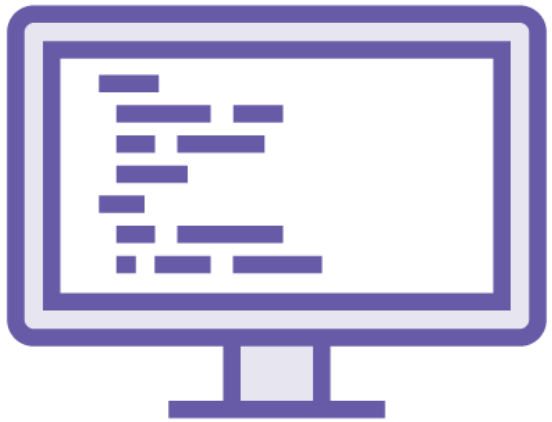
Thread One

```
1.1 WHILE (!S1) {}
1.2 PRINT “HELLO!”
```

Thread Two

```
2.1 S1=TRUE
2.2 PRINT “HI!”
```

Execution Order



In single threaded code, just one

In multithreaded code, execution order depends on:

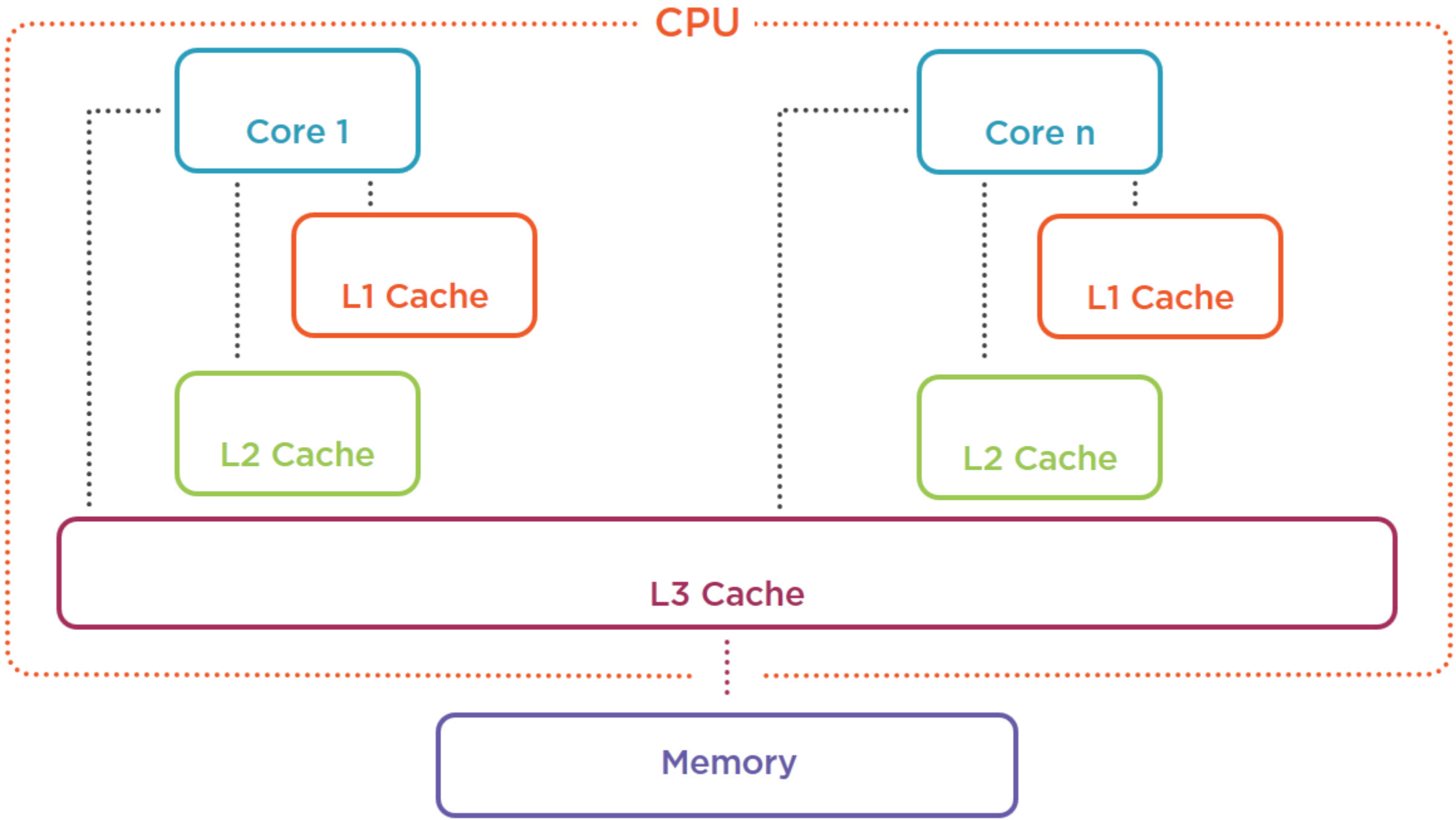
- Scheduler
- Processor
- Interactions between threads

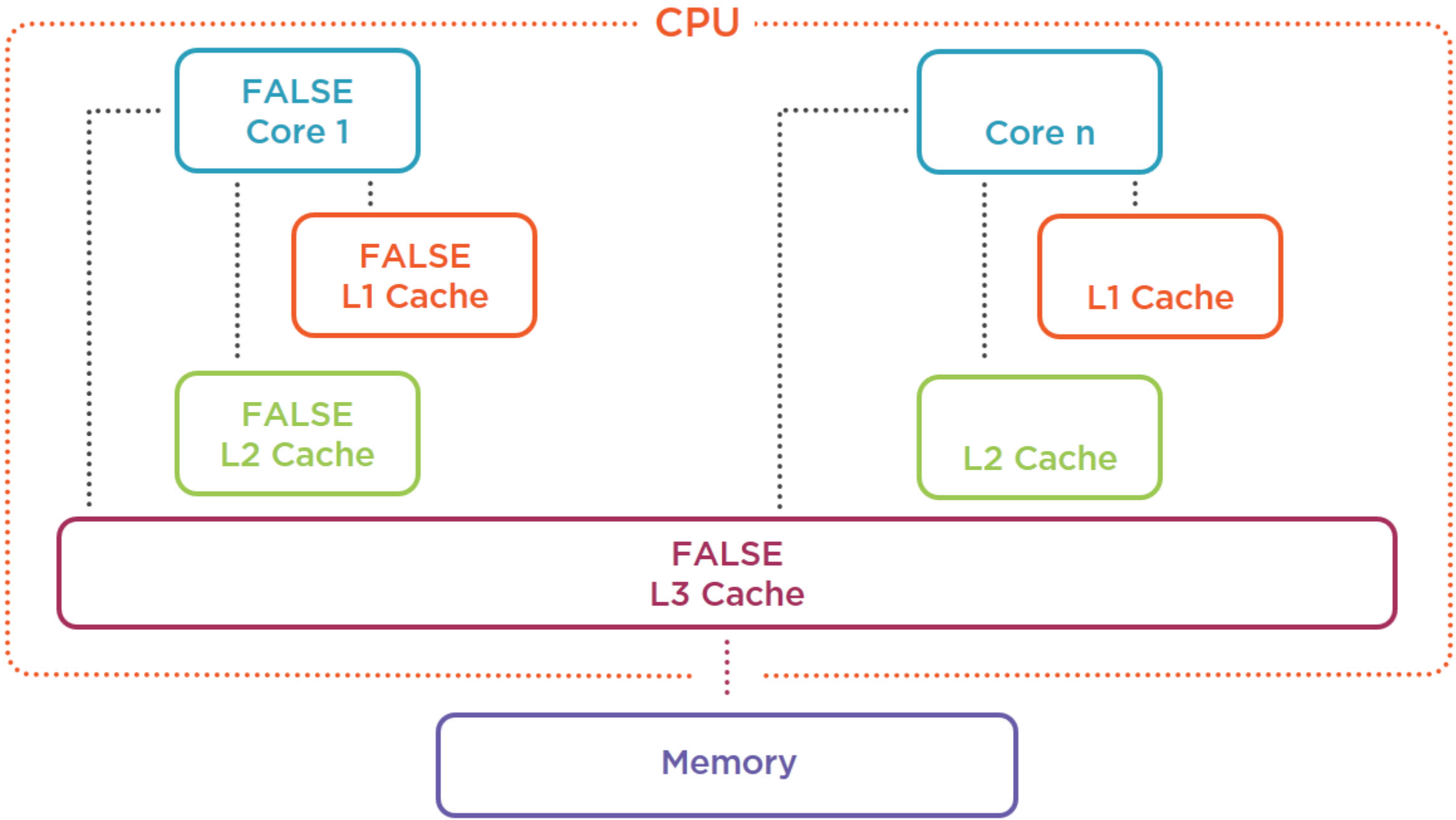
Thread One

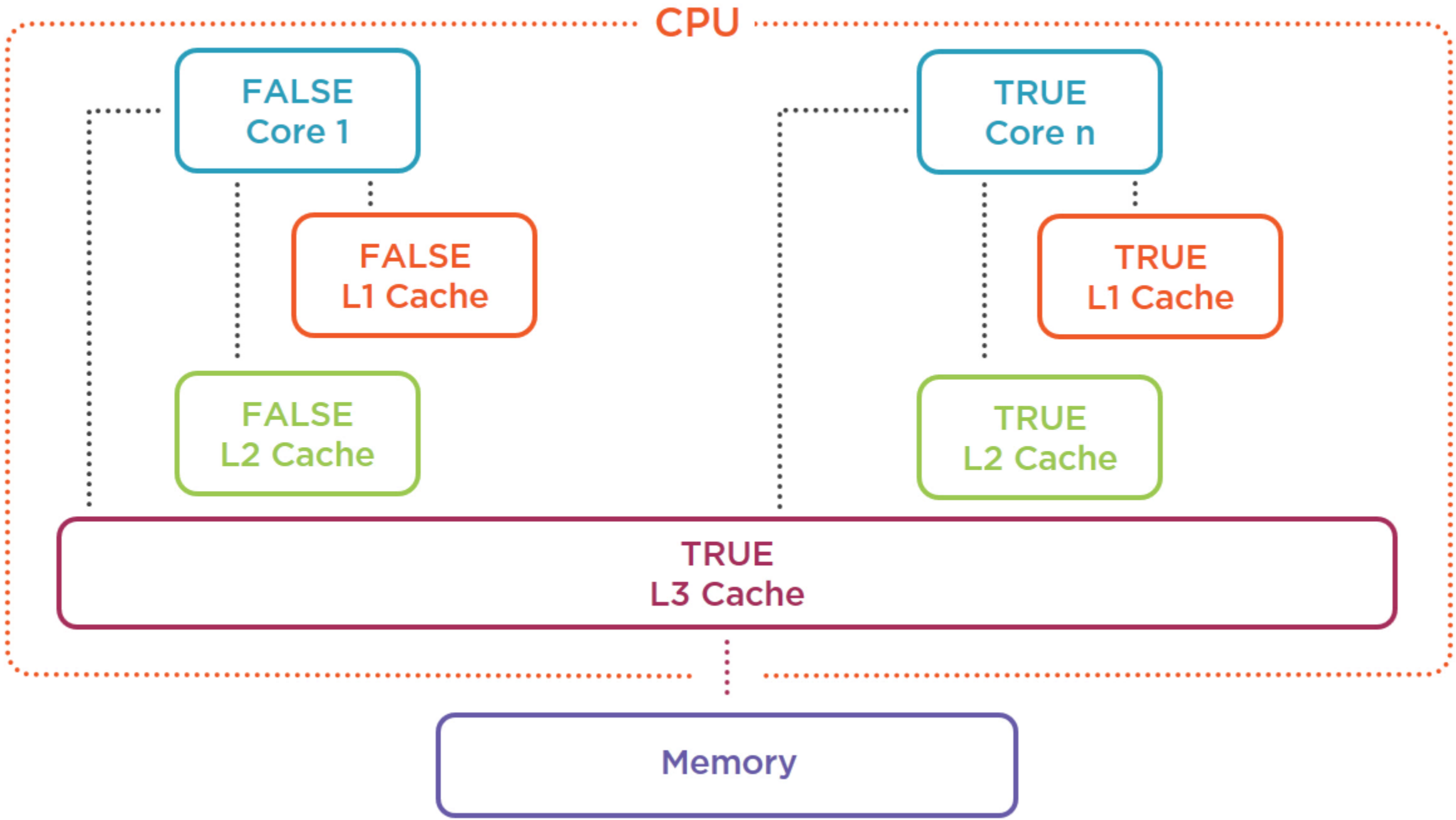
```
1.1 WHILE (!S1) {}
1.2 PRINT “HELLO!”
```

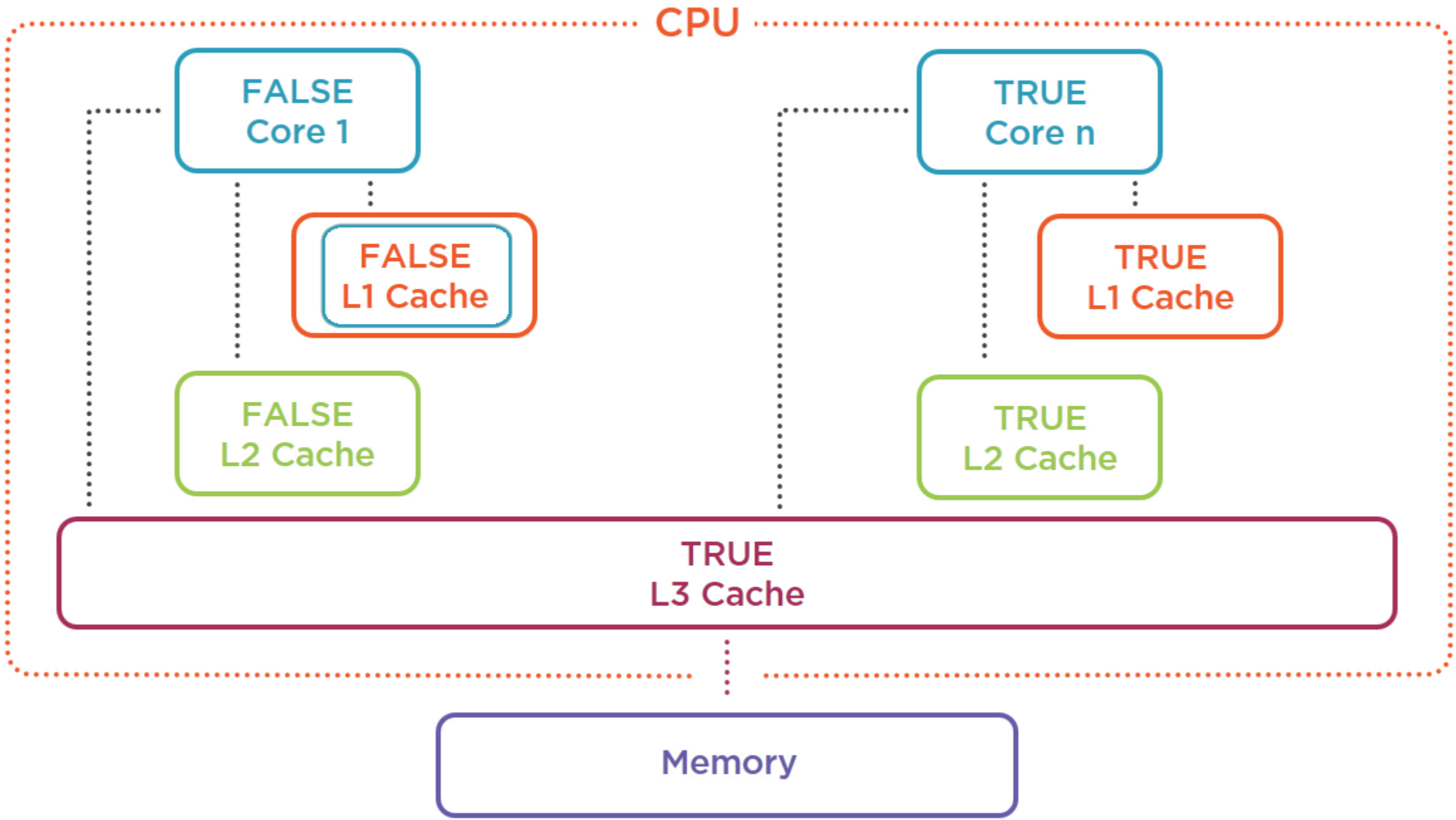
Thread Two

```
2.1 S1=TRUE
2.2 PRINT “HI!”
```


CPU
Core 1
Core n
L1 Cache
L1 Cache
L2 Cache
L2 Cache
L3 Cache
Memory


CPU
FALSE
Core 1
Core n
FALSE
L1 Cache
L1 Cache
FALSE
L2 Cache
L2 Cache
FALSE
L3 Cache
Memory


CPU
FALSE
Core 1
TRUE
Core n
FALSE
L1 Cache
TRUE
L1 Cache
FALSE
L2 Cache
TRUE
L2 Cache
TRUE
L3 Cache
Memory


CPU
FALSE
Core 1
TRUE
Core n
FALSE
L1 Cache
TRUE
L1 Cache
FALSE
L2 Cache
TRUE
L2 Cache
TRUE
L3 Cache
Memory

Thread One

```
1.1 L1 = S1
1.2 S2 = 2
1.3 PRINT “Thread1: ” + L1
```

Thread Two

```
2.1 L2 = S2
2.2 S1 = 1
2.3 PRINT “Thread2: “ + L2
```

Code Reordering

Compiler, JVM or processor can reorder code

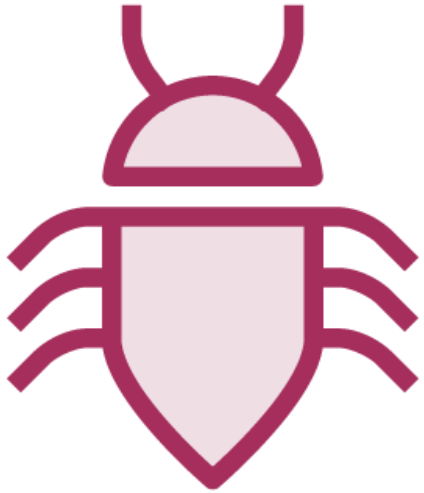
- To make it execute faster

Don't notice it when single threaded

- Except maybe when using a debugger

Can be harmful when multithreaded

Optimizations Are Platform Specific



Bugs might only show up occasionally on the production system

Despite extensive dev testing

Catching Bugs in the Act

Try logging or stepping with the debugger?

But this affects:

- Execution order, timings, cache contents, optimizations

Printing changes observations made by threads

- Bugs might disappear

Heisenbugs

Hard to detect

Cause weird sounding bug reports

Disappear when you try to observe them

Thread One

```
1.1 L1 = S1

1.2 L2 = L1.X

1.3 L3 = S2

1.4 L4 = L3.X

1.5 L5 = L1.X

1.6 PRINT “Thread1: “ +
L2, L4, L5
```

Thread Two

```
2.1 L6 = S1

2.2 L6.X = 3

2.3 PRINT “Thread2: “ +
L6.X
```

Thread One

```
S1[0] = S2.X
S1[1] = S2.X
…
S1[999] = S2.X
```

Thread Two

```
S2.X=1
S2.X=2
…
S2.X=9
```

Data Race

Different execution orders are possible

- Cannot say which will happen

When reading unsynchronized shared mutable data results in unexpected or incorrect values, we have a data race.

Synchronizing Data

Add the volatile keyword to shared variable definitions

Volatile has several effects and is asked about in Java interviews

Volatile on C/C++ (But Not Java!)

Means do not cache

No guarantees under multithreaded conditions

Volatile in Java

Volatile function one:

Any thread reading will see latest value

- Mechanism not specified

Volatile Variables

Will use v in pseudocode to indicate volatile

- e.g. vS1 – shared variable S1 is volatile

Finals cannot be volatile

Arrays and object references can be marked volatile

- Doesn't affect the contents

Thread One

```
1.1 WHILE (!vS1) {}
1.2 PRINT “HELLO!”
```

Thread Two

```
2.1 vS1=TRUE
2.2 PRINT “HI!”
```

Volatile Prevents Optimizations

Volatile function 2:

Indicates value may be shared between threads

Prevents optimizations based on program order

Volatile and Memory Fences

Volatile function 3:

Synchronizes data between threads

- By installing memory fences

Memory Fence Definition

‘A type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction’ - Wikipedia

Effect of Memory Fence

The memory state which was visible a writer of a volatile variable

Must at least be visible to any reader of the same variable

'At least' means at the point in time of the write or later

- Tricky/impossible to infer exactly what state

Volatile in Java

Volatile function four:

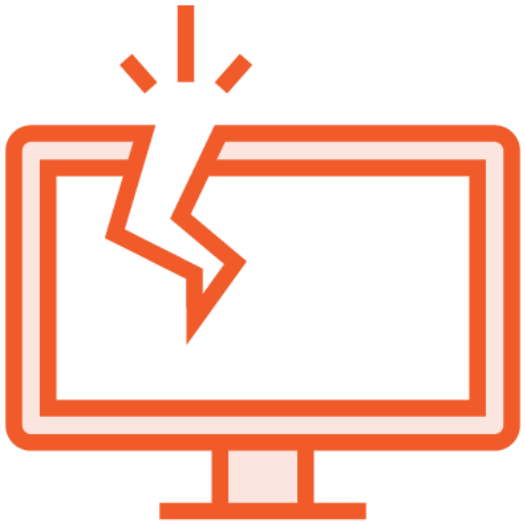
Prevents certain reorderings (see the table on the next slide)

Code Reordering Possibilities

	2nd Operation		
1st Operation	Normal Load Normal Store	Volatile Load	Volatile Store
Normal Load Normal Store	Yes (1)	Yes (2)	No (3)
Volatile Load	No (4)	No (5)	No (6)
Volatile Store	Yes (7)	No (8)	No (9)

Adapted from *The JSR-133 Cookbook for Compiler Writers* by Doug Lea

Inconsistencies and Broken Invariants



State we can see on a volatile read:

- At the time of the previous volatile write _OR_ later

May mean only some fields of an object updated

- Inconsistent / broken invariants

Solutions:

- Correctly publishing the object
- Mutexes

Thread One

```
1.1 L1 = vS1
1.2 vS2 = 2
1.3 PRINT “Thread1: ” + L1
```

Thread Two

```
2.1 L2 = vS2
2.2 vS1 = 1
2.3 PRINT “Thread2: “ + L2
```

Thread One

```
1.1 L1 = vS1

1.2 L2 = L1.vX

1.3 L3 = vS2

1.4 L4 = L3.vX

1.5 L5 = L1.vX

1.6 PRINT “Thread1: “ +
L2, L4, L5
```

Thread Two

```
2.1 L6 = vS1

2.2 L6.vX = 3

2.3 PRINT “Thread2: “ +
L6.vX
```

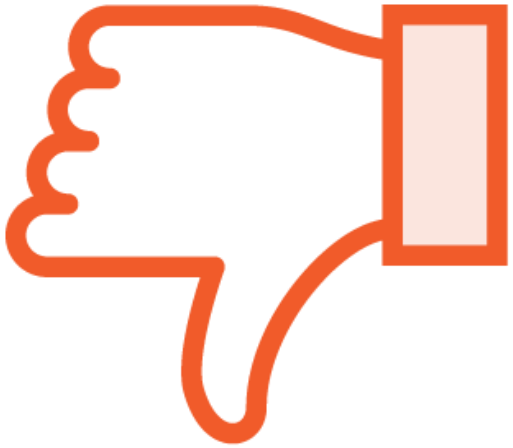
Thread One

```
vS1[0] = S2.vX
vS1[1] = S2.vX
…
vS1[999] = S2.vX
```

Thread Two

```
S2.vX=1
S2.vX=2
…
S2.vX=9
```

Drawbacks of Volatile



Impacts performance

- Latest value has to be made visible to other cores

Doesn't guarantee data consistency

Publishing Objects

One thread does the update

- Publishes changes to other threads
- Other threads use the updated data

Thread One

```
volatile boolean publish = false;


obj.field1 = 1;
obj.field2 = “hi”;
…
publish = true;
```

Thread Two

```
if (publish) {
  field1 = obj.field1;
  field2 = obj.field2;
}
```

Final Variables

If the object is published correctly

- Reference to 'this' not allowed to escape during construction

Other threads will only see initialised final values

- Don't need to worry about data races

Publishing immutable objects is better than agreeing not to change them

Publishing and Libraries

Issues:

- Documentation of multithreaded behaviour?
- Source code available?
- May mutate under the hood on a read operation
- Implementation changing with upgrades

Suggest storing results instead of publishing library objects

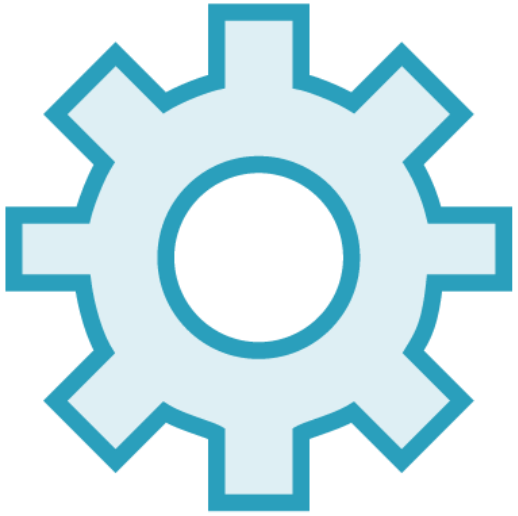
Publishing in Practice

It works, but...

- Needs good design
- Clear/simple[/strict] working practices
- Developers knowledgeable
- Care taken

Better than having to check for synchronization all the time

Thread Creation/Death Guarantee



When thread is created

- It can see at least the state creator could see at time of creation

When thread is dead and another thread calls isAlive or join

- It can see at least the state the thread saw at exit

Word Tearing

Where modification of an array affects adjacent elements

E.g.

- Given an array of bytes
- If the processor has to write whole words
- Adjacent elements might be affected by synchronization issues

Not an issue in Java

64 Bit Primitives

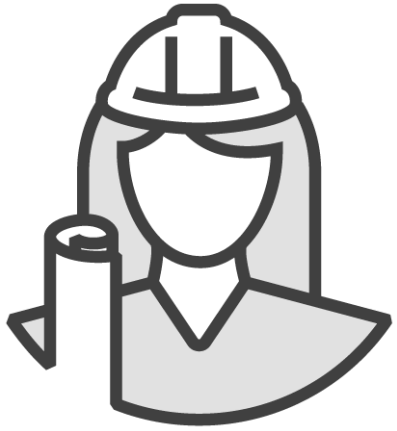
Long /double not thread safe (unless volatile)

- May be split in two 32 bit operations to read/write
- So may read a value never assigned (only one operation carried out)

Or use AtomicLong, Long or Double wrappers

64 bit references always safe

Thread Safety, Synchronization and Data Races



Due to caching, compiler and processor optimizations

- Code which works in single threaded mode doesn't work under multithreaded conditions
- Where sharing mutable state
- Need to be properly synchronized to avoid data races

Thread Safety, Synchronization and Data Races

What marking a variable volatile does:

- Always see latest version
- Prevent harmful optimizations
- Installs memory fences limiting reordering
- Synchronizes threads 'view of the world'

How to Safely Publish Objects

JMM guarantees:

- Thread creation
- Thread death
- No word tearing

64 bit primitive issues

- References are safe