X86 memory coherency

ISAIAH PERUMALLA

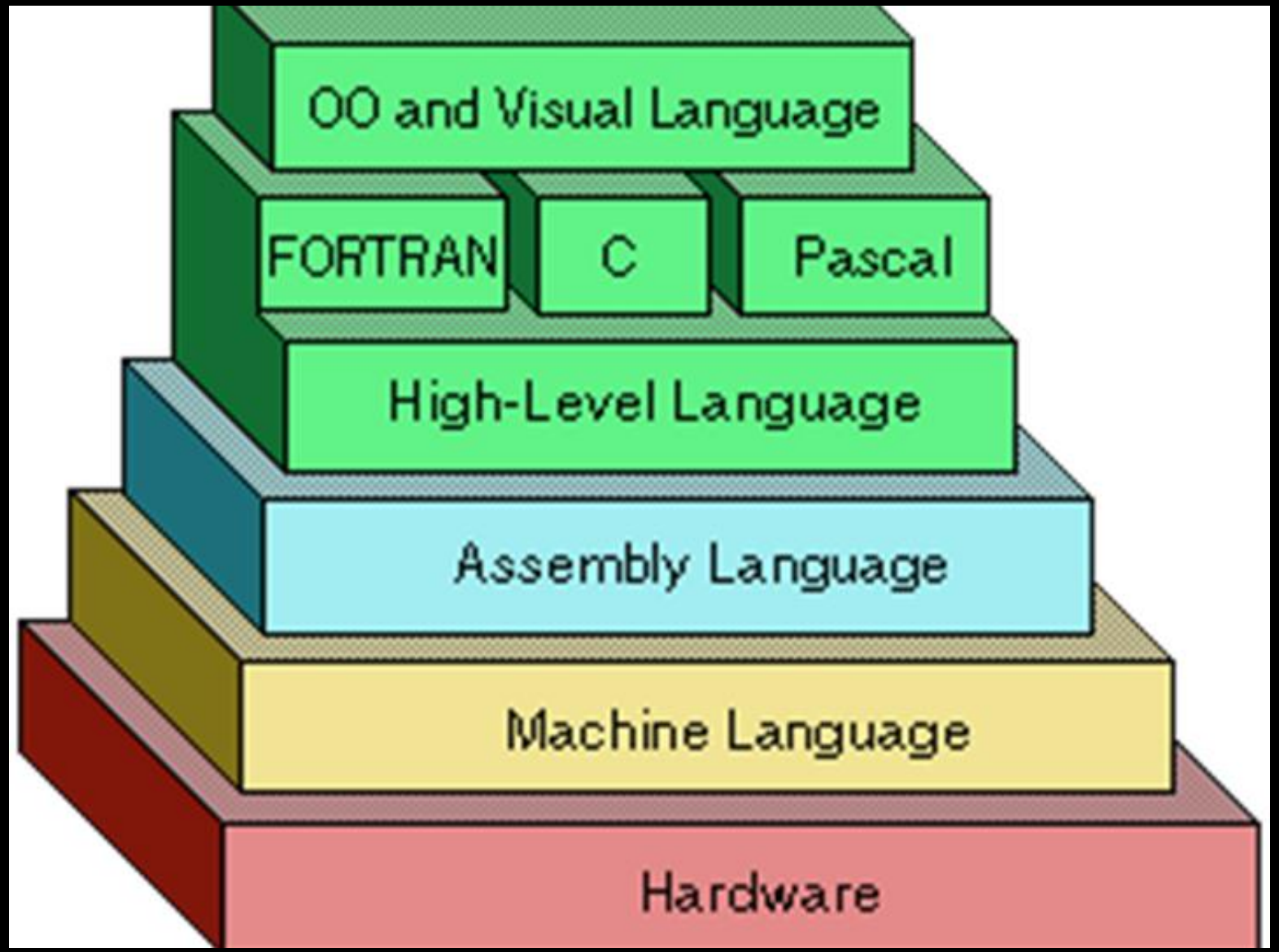
Background/Motivation

Virtual machines are just C/C++ programs running on real Machines like X86

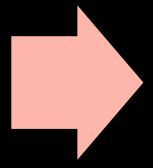
Understand from bottom up perspective (Why)

→ Initially out of interest and curiosity

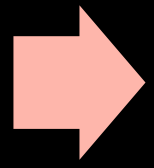
→ But turned out to be useful in day job too



Multiprocessors reorder memory operations in unintuitive ways



This behaviour is necessary for performance but affects correctness too!



We only need to care about what we can observe in software

Learning through experiments

Initially A = B = 0

Thread 1

```
A = 1
if (B == 0)
    print “Hello”;
```

Thread 2

```
B = 1
if (A == 0)
    print “World”;
```

What can be printed?

Initially A = B = 0

Thread 1	Thread 2
`A = 1`	`B = 1`
`if (B == 0)`	`if (A == 0)`
`  print “Hello”;`	`  print “World”;`

What can be printed?

Single writer** for each location in **memory
Single reader for different location

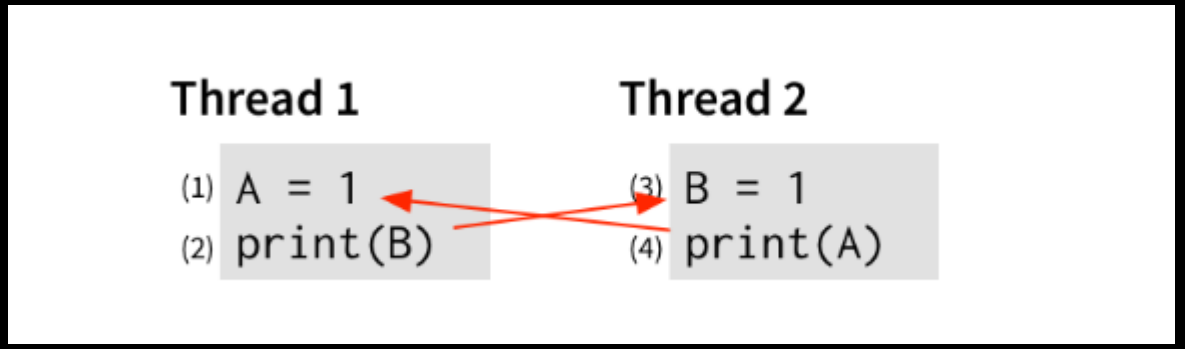
can it print hello world ?

Initially A = B = 0

Thread 1	Thread 2
A = 1	B = 1
if (B == 0)	if (A == 0)
print “Hello”;	print “World”;

What can be printed?

For line 2 to print 0
Print B must happen before B = 1 on Thread-2



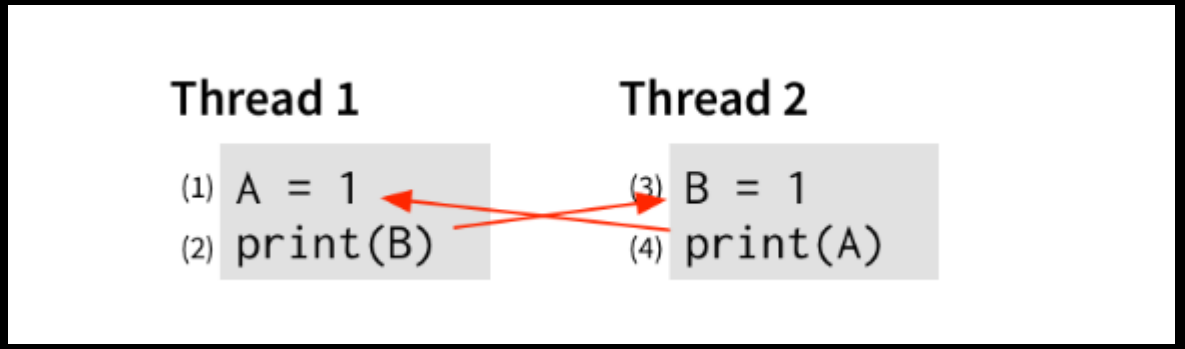
For line 4 to print 0
Print A must happen before A = 1 on Threa-1

Initially A = B = 0

Thread 1	Thread 2
A = 1	B = 1
if (B == 0)	if (A == 0)
print “Hello”;	print “World”;

What can be printed?

For line 2 to print 0
Print B must happen before B = 1



For line 4 to print 0
Print A must happen before A = 1

```
// x and y are initialised to 0

CORE-0   CORE-1
-------------------
x = 1    y = 1
r0 = y   r1 = x
```

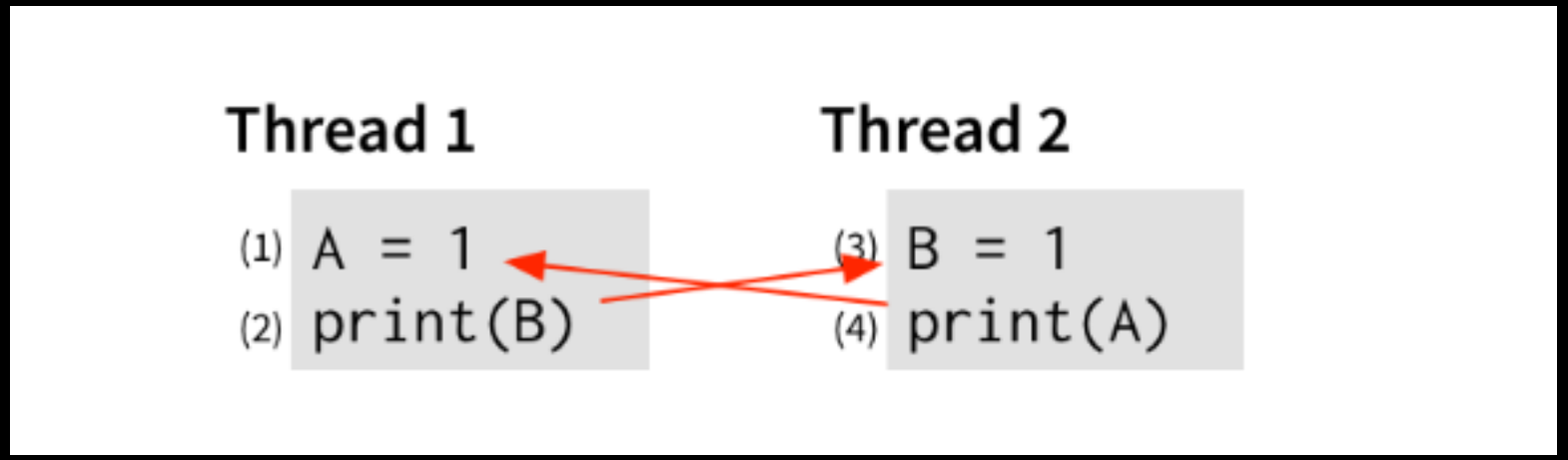
1. Thread0 runs first all way through r0=0, r1=1
2. Thread1 runs first all way through r0=1, r1=0
3. any other interleaving opertion can only result in r0=1 , r1=1

Interleaving possiblites

STEP-1	STEP-2	STEP-3	STEP-4	(R0, R1)
core-1: y=1	core-0: x=1	core-1: r1=x	core-0: r0=y	(1,1)
core-1: y=1	core-0: x=1	core-0: r1=y	core-1: r0=x	(1,1)
core-0: x=1	core-1: y=1	core-0: r0=y	core-1: r1=x	(1,1)
core-0: x=1	core-1: y=1	core-1: r1=x	core-0: r0=y	(1,1)

- How can r0 and r1 == 0
- A re-order happen

Who is Re-Ordering



Is it compiler ?

Is it hardware ?

Or Both

- C is simple and easier to reason
- So good starting point

```
// x and y are initialised to 0

 CORE-0   CORE-1
-------------------------
 x = 1    y = 1
 r0 = y   r1 = x
```

- Should only terminate if r0 and r1 equal to zero
- A re-order happen


```c
#include <pthread.h>
#include <stdio.h>

int x, y = 0;
int r0, r1;

void *core0 (void *arg)
{
  x = 1;
  r1 = y;
  return 0;
}

void *core1 (void *arg)
{
  y = 1;
  r0 = x;
  return 0;
}


int main (void)
{
  pthread_t thread0, thread1;
  while (1) {
    x = y = 0;
    //Start threads
    pthread_create (&thread0, NULL, core0, NULL);
    pthread_create (&thread1, NULL, core1, NULL);

    //wait for threads to complete
    pthread_join (thread0, NULL);
    pthread_join (thread1, NULL);

    if (r0 == 0 && r1 ==0) {
      printf ("(r0=%d, r1=%d)\n", r0, r1);
      break;
    }

  }
  return 0;
}
```

- Lets rule out compiler Re-ordering
- Small hack to stop GCC compiler from re-ordering

- Program still terminates
- Meaning a Re-order happen !


```c
#include <pthread.h>
#include <stdio.h>

int x, y = 0;
int r0, r1;

void *core0 (void *arg)
{
  x = 1;
  asm volatile ("" ::: "memory"); // ensure GCC compiler will not reorder
  r1 = y;
  return 0;
}

void *core1 (void *arg)
{
  y = 1;
  asm volatile ("" ::: "memory"); // ensure GCC compiler will not reorder
  r0 = x;
  return 0;
}


int main (void)
{
  pthread_t thread0, thread1;
  while (1) {
    x = y = 0;
    //Start threads
    pthread_create (&thread0, NULL, core0, NULL);
    pthread_create (&thread1, NULL, core1, NULL);

    //wait for threads to complete
    pthread_join (thread0, NULL);
    pthread_join (thread1, NULL);

    if (r0 == 0 && r1 ==0) {
      printf ("(r0=%d, r1=%d)\n", r0, r1);
      break;
    }

  }
  return 0;
}
```

Looking at one level lower just to be sure

```
8   void *core0 (void *arg)
9   {
10    x = 1;
11    asm volatile ("" ::: "memory"); /
12    r1 = y;
13    return 0;
14  }
15
16  void *core1 (void *arg)
17  {
18    y = 1;
19    asm volatile ("" ::: "memory"); /
20    r0 = x;
21    return 0;
22  }
23
```

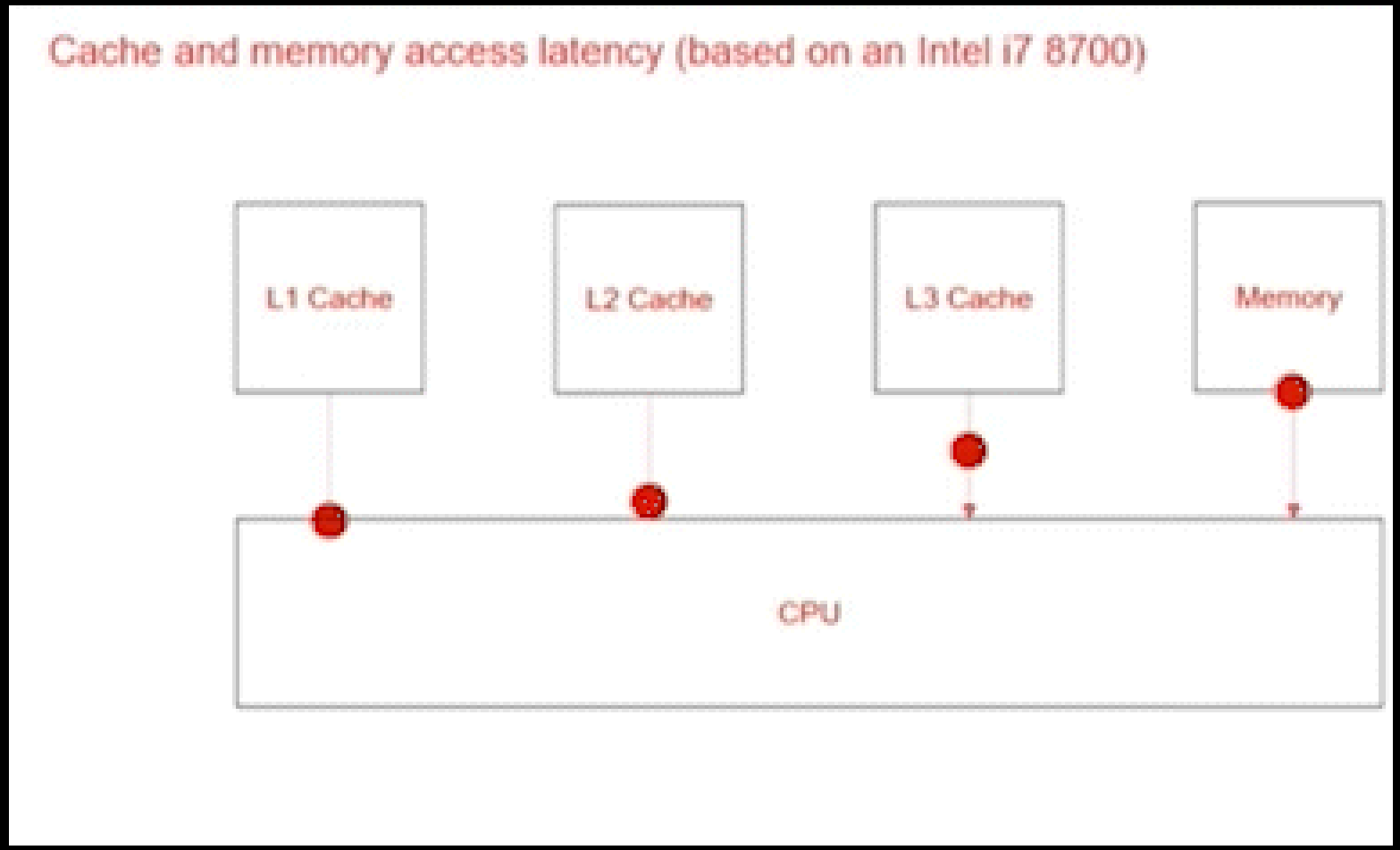
```
1  core0:
2          push    rbp
3          mov     rbp, rsp
4          mov     QWORD PTR [rbp-8], rdi
5          mov     DWORD PTR x[rip], 1   ## write value 1 to x
6          mov     eax, DWORD PTR y[rip] ## read contents of y into  register
7          mov     DWORD PTR r1[rip], eax
8          mov     eax, 0
9          pop     rbp
10         ret
11 core1:
12         push    rbp
13         mov     rbp, rsp
14         mov     QWORD PTR [rbp-8], rdi
15         mov     DWORD PTR y[rip], 1 ## write value 1 to y
16         mov     eax, DWORD PTR x[rip] ## read contents of x into  register
17         mov     DWORD PTR r0[rip], eax
18         mov     eax, 0
19         pop     rbp
20         ret
```

Has to be at hardware level

1. Writes to memory even slower than reads as more book keeping required to keep caches coherent

Has to be at hardware level

1. Remember this from last week memory is super slow
2. Writes to memory even slower as more book keeping required to keep caches coherent



Coherent Memory

Coherent memory (L1/L2/L3 caches and RAM)

→ **Caches in CPU cores are coherent so if one CPU writes to its L1 cache, x86 hardware guarantees other CPUs will observe the change too. As software devs we don’t need to worry about coherency here**

Non Coherent Memory

Non coherent memory (Store buffers, Load buffers, TLB, maybe more)

→ **Store buffers (visible effect and we need to handle this)**

→ Load Buffer (no visible effect on software on x86)

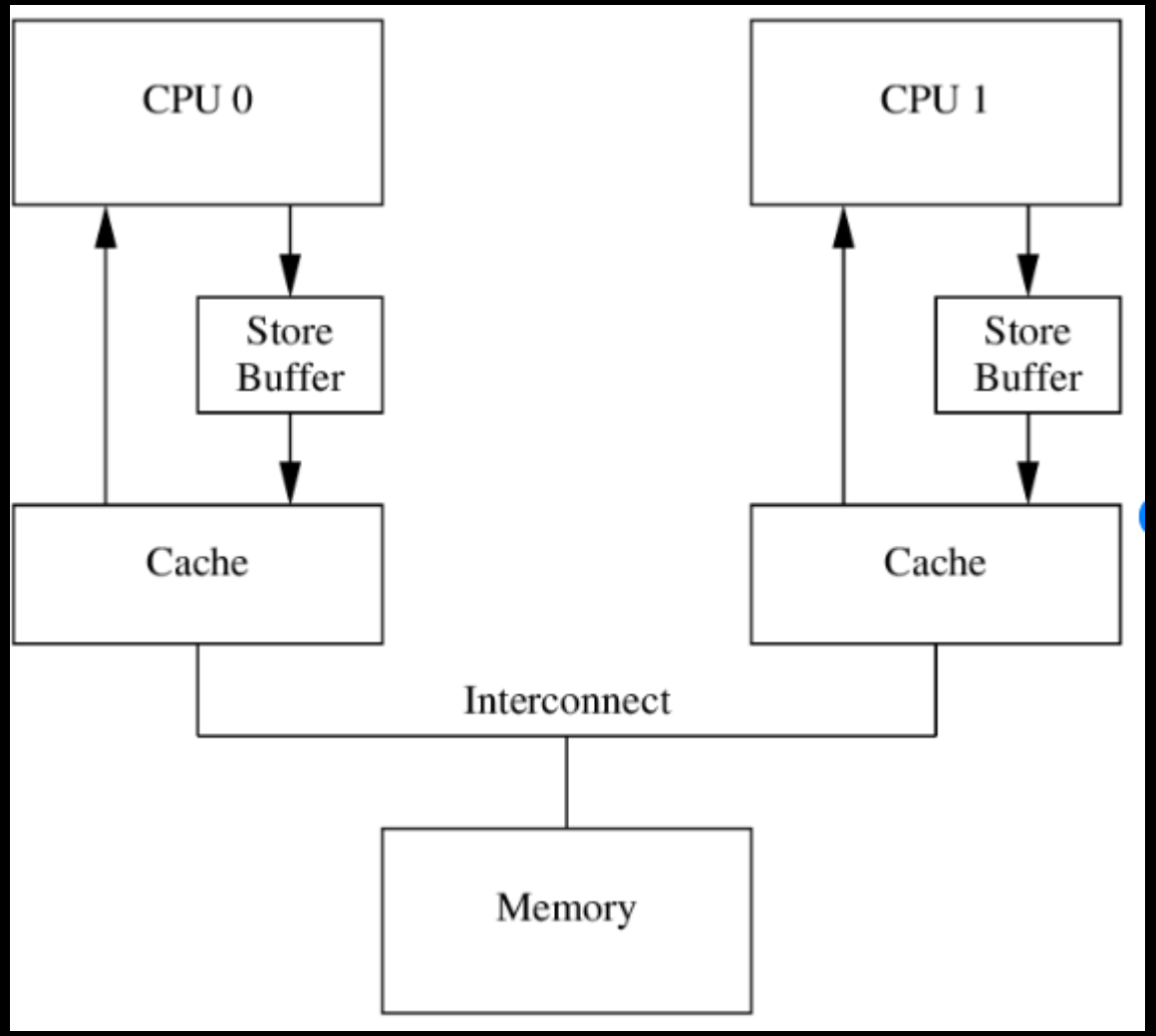
→ TLB (visible effect especially in mem mapped files (another presentation maybe)

→ Maybe others I don't know yet but they don't affect our software

Store Buffer

→ Unlike Cache this affects correctness

→ Non-Coherent part of x86 memory

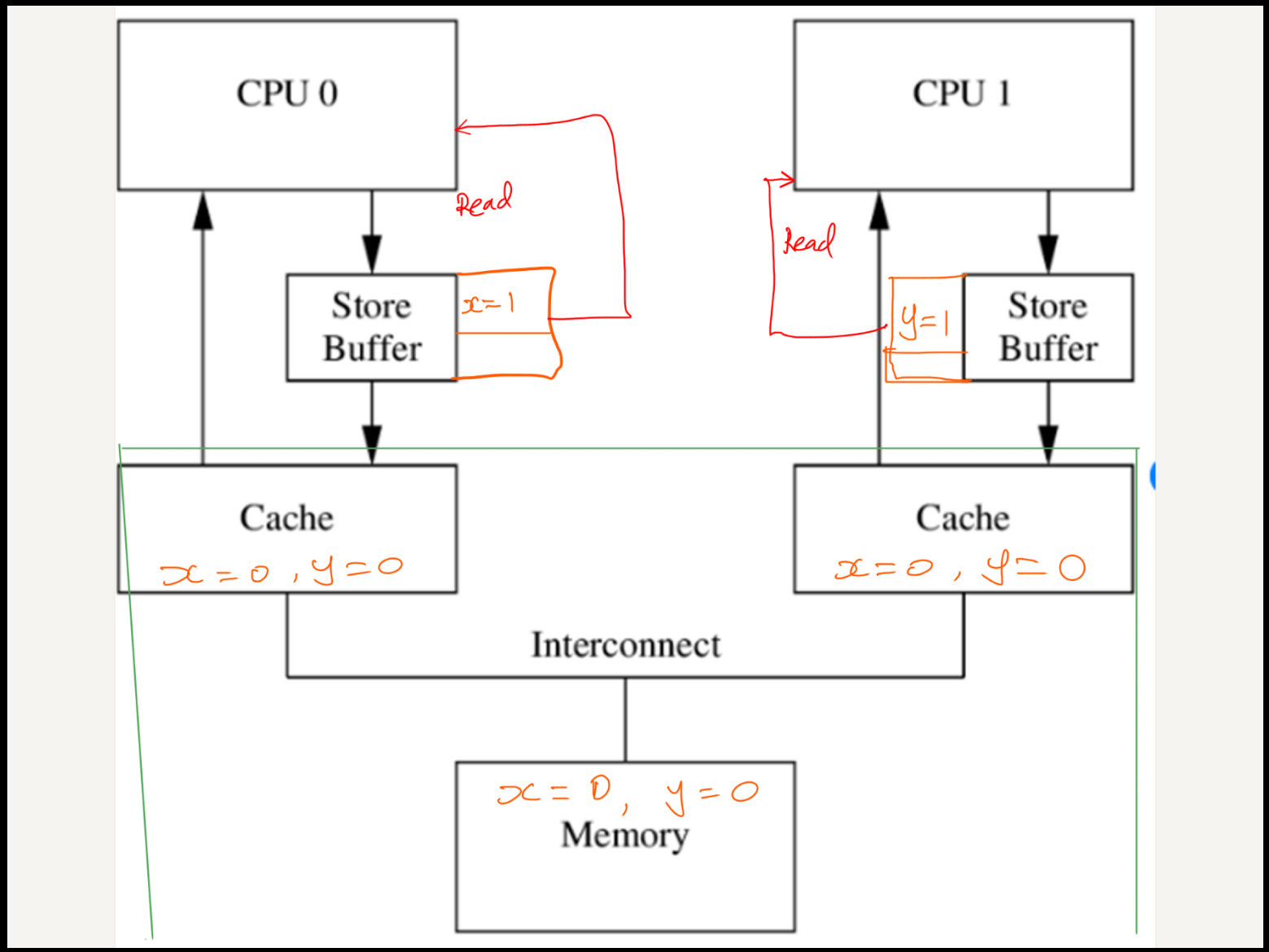


Store Buffer

→ Write is in Store buffer

→ Non-Coherent part of x86 memory

→ Other core cannot observer the writes



Memory Barriers

→ Flush Store buffer to coherent memory

→ CPU will stall until this is done

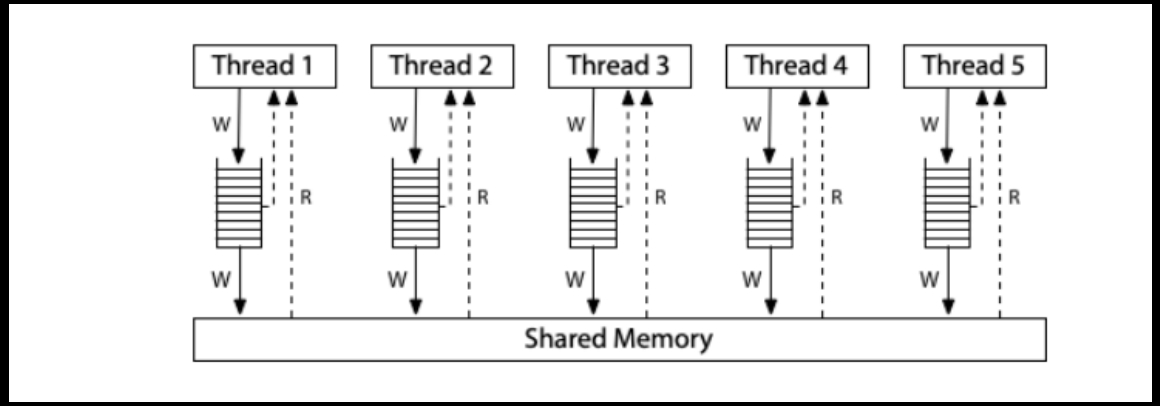
→ X86 MFENCE instruction

→ X86 LOCK prefix has same effect

```
 8 void *core0 (void *arg)
 9 {
10   x = 1;
11   asm volatile ("mfence" ::: "memory"); /
12   r1 = y;
13   return 0;
14 }
15
16 void *core1 (void *arg)
17 {
18   y = 1;
19   asm volatile ("mfence" ::: "memory"); /
20   r0 = x;
21   return 0;
22 }
```

Memory Barriers

- → X86 is one of the easiest architectures to understand
- → ***Only Write and Read can be re-ordered***
- → Write and Write don't get re-ordered or we cannot observe from software
- → Read followed by Write is not Reordered
- → Read followed by Read is also not reordered
- → Simple model on right is enough



Moving up Stack to JAVA

- → Flush Store buffer to coherent memory
- → CPU will stall until this is done

- → X86 MFENCE instruction
- → X86 LOCK prefix has same effect

```
30      static void thread0()
31      {
32          thread0Val = 1;
33          UNSAFE.fullFence();
34          r1 = thread1Val;
35
36      }
37
```

→ X86 is one of the easiest architectures to understand

→ ARM and other are bit more complex, more possible re-ordering possible