How Memory Reordering Can Ruin Your Day

... and your lock-free algorithms.

Multicore CPUs are **everywhere**.

- In your server,
- in your laptop,
- in your pocket,
- even in your hard drive itself.

Cool, let's share some data!

- Sometimes, it's simple atomic operations,
- often, it's not.
- Concurrent modifications may interfere.

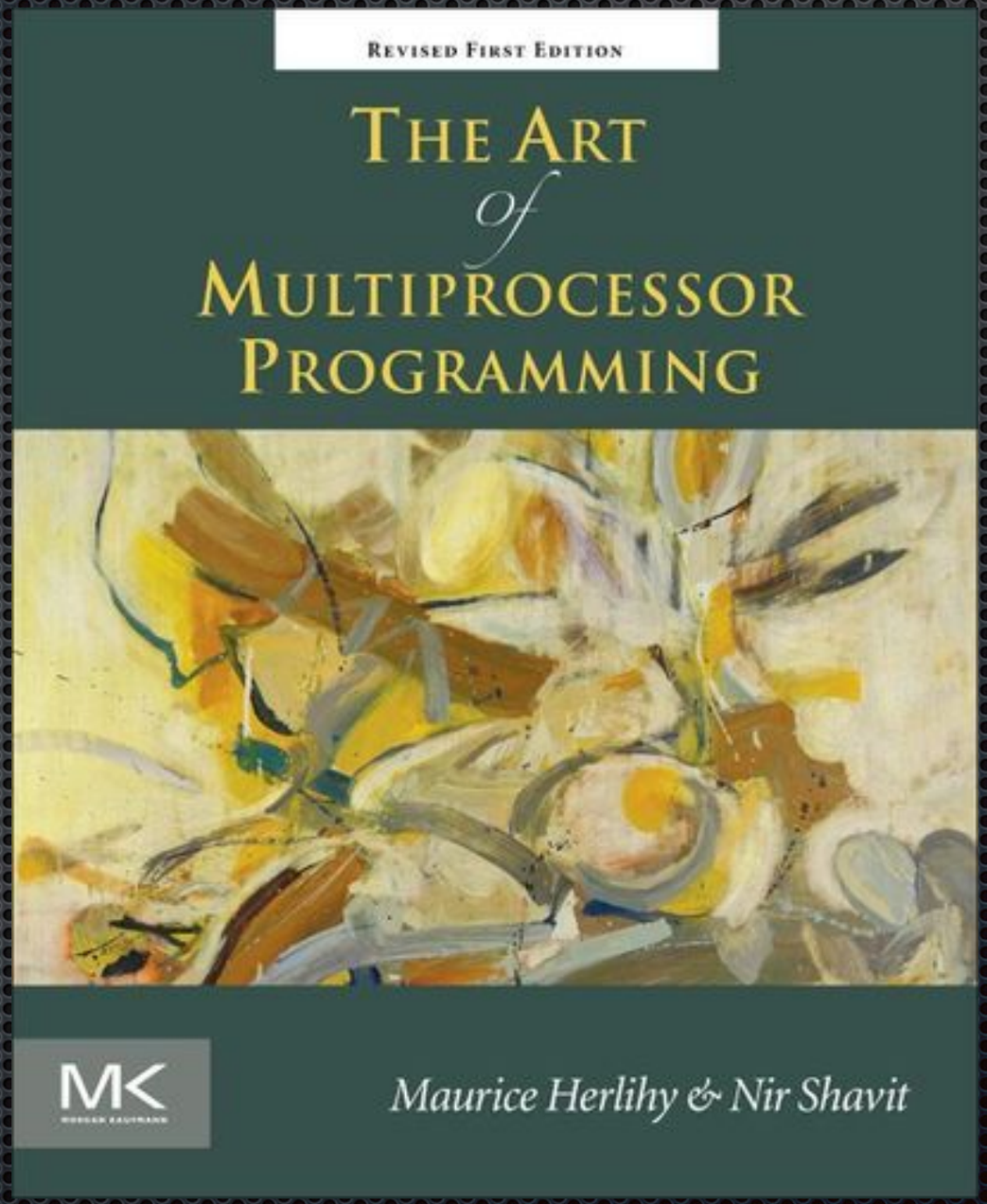
That's tricky. Let's use locks.

- They provide simple **mutual exclusion**.
- Acquire the lock: Get in... or wait.

Let’s not use locks?

- Bottlenecks
- Overhead

Lock-free algorithms



So, let's implement...

... a lock.

... a lock.

- More specifically, a simple spinlock...
- ... which is itself a lock-free algorithm.
- Easy to use, easy to reason about.

Peterson's Algorithm

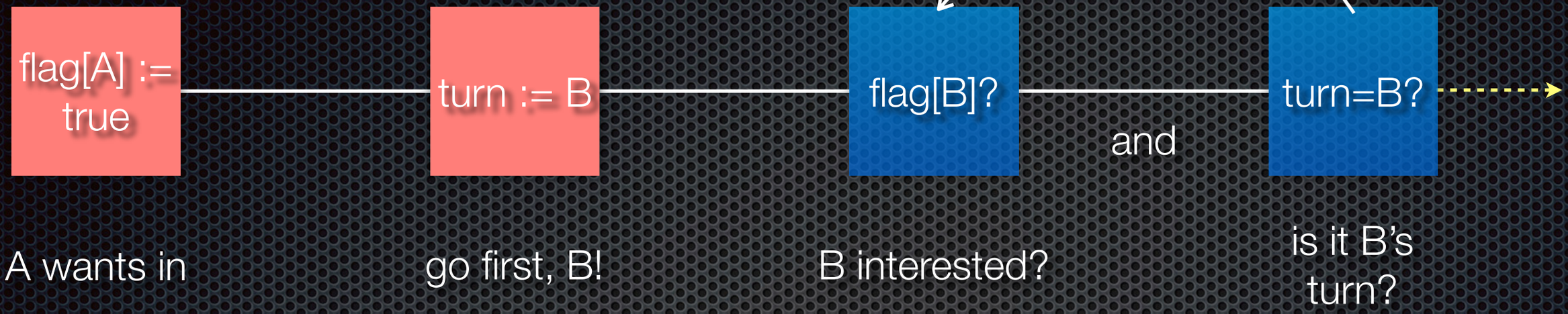
Simple spin lock, suitable for **two threads**.

Uses three shared variables:

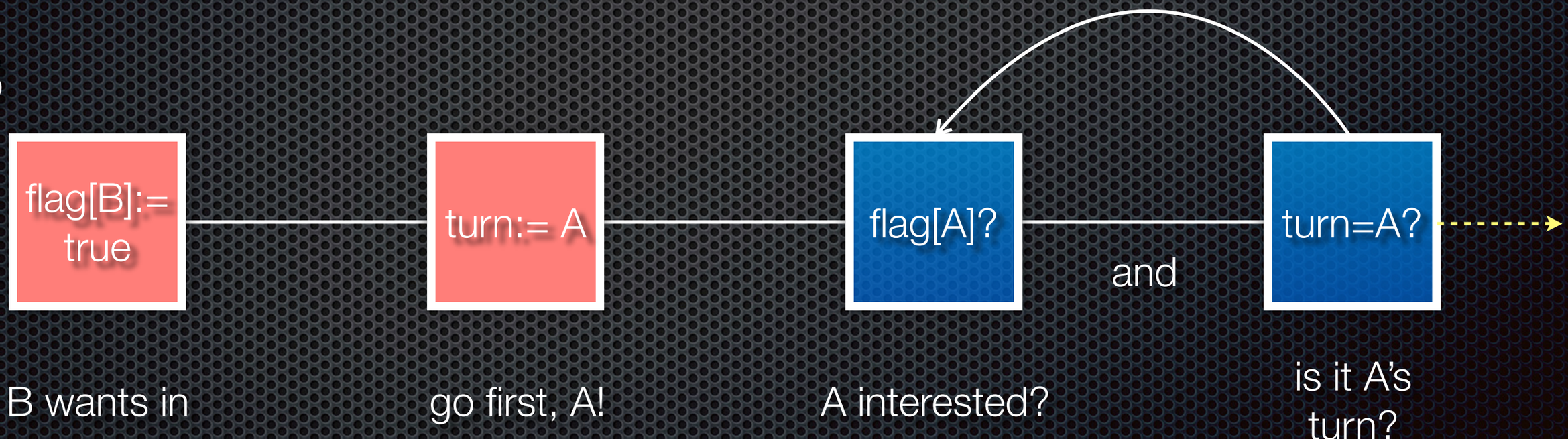
One flag per-thread indicating that it "wants in",

one integer saying which thread's turn it is

A


flag[A] := true
turn := B
flag[B]?
and
turn=B?
A wants in
go first, B!
B interested?
is it B's turn?

B


flag[B]:= true
turn:= A
flag[A]?
and
turn=A?
B wants in
go first, A!
A interested?
is it A's turn?

```
volatile bool flag[2] = {false, false};
volatile int turn = 0;


void lock(int id) {
    int other_id = 1 - id;

    flag[id] = true;     // we want in
    turn = other_id;     // ... but let the other in first

    while (flag[other_id] && turn == other_id) /* spin */;
}


void unlock(int id) {
    flag[id] = false;    // we don't want in anymore
}
```

A simple shared counter:

```
volatile int counter = 0;

void counting_thread(int my_id) {
    for (int i = 0; i < thread_cycles; i++){
        lock(my_id);
        counter++;
        unlock(my_id);
    }
}

int main(void) {
while(true) {
... create threads ...
... wait for threads to exit ...

    printf("counter: %i (%i)\n", counter,
        2*thread_cycles - counter);
}

    return 0;
}
```

not atomic!

Let’s try it out on our desktop!

- Each thread increments the counter **10,000,000** times.
- We expect the final counter to be **20,000,000**.

Let's try it out on our desktop!

```
% ./count-lock-no-barrier-02
counter: 19999865 (135)
counter: 19999775 (225)
counter: 19999839 (161)
counter: 19999881 (119)
counter: 19999802 (198)
counter: 19999832 (168)
counter: 19999844 (156)
```

Why?

Broken algorithm...?

- No, pretty sure it's right.
- Besides, it does work on Uni-Processor.

So this means that....

- Our assumptions are wrong.
- Our CPU's memory model is not as **strong**.
- The strongest memory model is when every core sees every memory access in **program order**.
  - This is called **sequentially consistent**.

Let's take a look at the manual.

Intel® 64 and IA-32 Architectures Software Developer's Manual,
Volume 3A: System Programming Guide, Part 1,
Section 8.2.3.4:

"Loads May Be Reordered with Earlier Stores to Different Locations

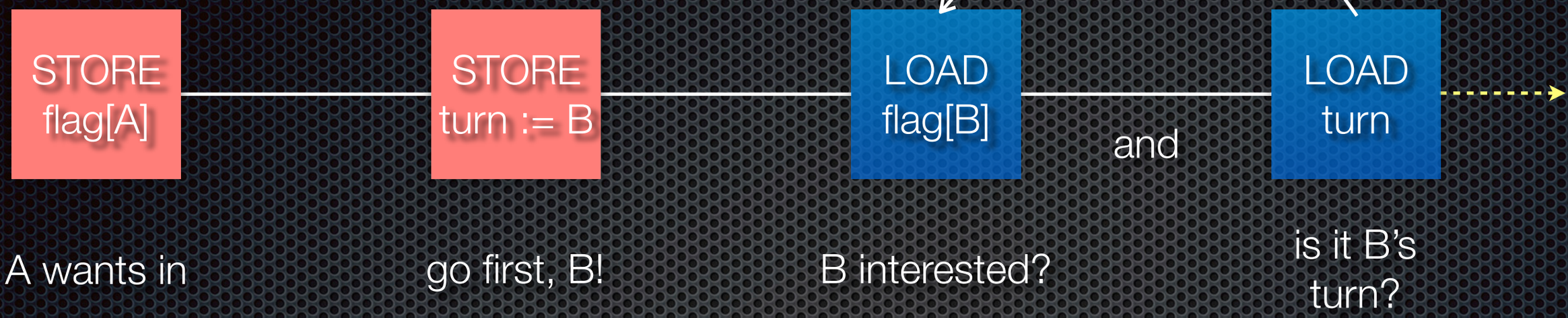
The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location."

Intuitively (and entirely speculative):

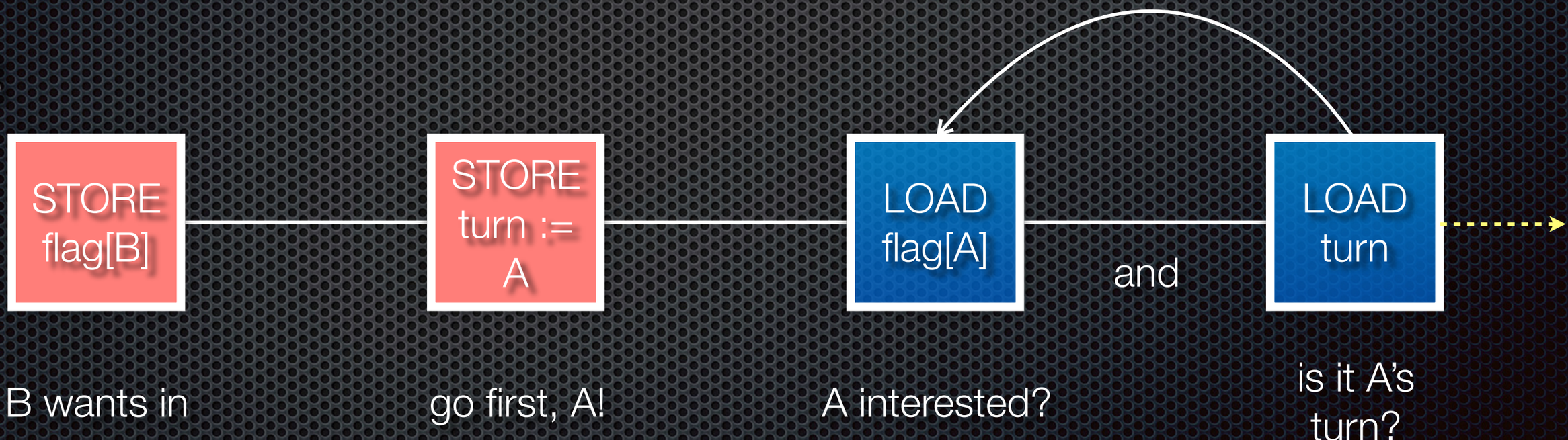
- A core modifies a value, writes it out in its *own* cache...
- ... but it may be more convenient to commit those stores to memory “later”.
- In the meantime: some loads from other locations.

```
void lock(int id) {
    int other_id = 1 - id;

    flag[id] = true;              // STORE flag[id]
    turn = other_id;              // STORE turn

    while (flag[other_id] &&      // LOAD flag[other_id]
           turn == other_id)      // LOAD turn
        /* spin */;
}
```

A


STORE
flag[A]
STORE
turn := B
LOAD
flag[B]
and
LOAD
turn
A wants in
go first, B!
B interested?
is it B's
turn?

B


STORE
flag[B]
STORE
turn :=
A
LOAD
flag[A]
and
LOAD
turn
B wants in
go first, A!
A interested?
is it A's
turn?

A


STORE
flag[A]
STORE
turn := B
LOAD
flag[B]
LOAD
turn

B


LOAD
flag[A]
STORE
flag[B]
STORE
turn :=
A
LOAD
turn

B A A B B A A B

LOAD flag[A] — STORE flag[A] — STORE turn := B — STORE flag[B] — STORE turn := A — LOAD flag[B] — LOAD turn — LOAD turn

A interested?

No! My Turn!

A wants in

go first, B!

B wants in

go first, A!

B interested?

Yes, but...

is it B's turn?

No! My Turn!

is it A's turn?

B A A B B A A B

LOAD flag[A]

STORE flag[A]

STORE turn := B

STORE flag[B]

STORE turn := A

LOAD flag[B]

LOAD turn

LOAD turn

A interested?

A wants in

go first, B!

B wants in

go first, A!

B interested?

Yes, but...

is it B's turn?

is it A's turn?

No! My Turn!

No! My Turn!

both enter

Memory Barriers: Saving us from weakness!

- Dedicated CPU instructions.
  - e.g. **`MFENCE`** on x86
- “Do not reorder memory across this barrier!”

```
#define MFENCE() { __asm__("mfence" ::: "memory"); }

void lock(int id) {
    int other_id = 1 - id;

    flag[id] = true;              // STORE flag[id]
    turn = other_id;              // STORE turn

    MFENCE();

    while (flag[other_id] &&      // LOAD flag[other_id]
           turn == other_id)      // LOAD turn
        /* spin */;
}
```

=> `flag[other_id]` not reordered with the stores anymore

```
% ./count-lock-mem-barrier-O2
counter: 20000000 (0)
counter: 20000000 (0)
counter: 20000000 (0)
counter: 20000000 (0)
counter: 20000000 (0)
counter: 20000000 (0)
counter: 20000000 (0)
...
```

Cool!

Let’s try it out on our mobile!

mp-talk — adb — 120×40

```
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 59999999 (1)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
```

mp-talk — adb — 120×40

```
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 59999999 (1)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
counter: 60000000 (0)
```

What!
But I used a barrier!

Well...

	AMD64	x86	
incoherent instruction pipeline		X	
stores reordered after loads	X	X	

Well...

	AMD64	x86	ARMv7
incoherent instruction pipeline		X	X
stores reordered after loads	X	X	X
stores reordered after stores			X
loads reordered after loads			X
loads reordered after stores			X
atomic reordered with loads			X
atomic reordered with stores			X

Many additional hazards...

For example:

```
void counting_thread(int my_id) {
    for (int i = 0; i < thread_cycles; i++){
        lock(my_id);
        counter++;
        unlock(my_id); // flag[id] = false;
    }
}
```

Reorder those two, and you’re gonna have a bad time.

We need more barriers.

Funny story with Intel, by the way.

- Intel's memory model specification came late.
- Besides **`MFENCE`**, there are also **`LFENCE`** and **`SFENCE`**.
- Still useful, because *some* parts of the CPU *aren't* so strong... (SSE)

- Use locks.
- Do not implement your own locks.

- Use the atomic primitives of your platform.
- Do not implement your own atomic primitives.

- **If you use C++11, use `std::atomic`.**
  - It's wonderful!
  - `volatile` is not enough.
- **If you use Java, use `volatile`, `synchronized`.**
  - They strengthened it to make it less confusing.
- **If you use C...**

- **... then it really depends on what you have.**
  - **gcc** has atomic builtins.
  - **BSDs** have the `atomic_*` library calls.
  - **iOS and OS X** have the `atomic(3)` functions:

```
OSAtomicIncrement32Barrier,
OSAtomicCompareAndSwap64Barrier,
OSAtomicTestAndSetBarrier...
```

  - **Windows** has... essentially the same stuff, but I don't know a lot about Windows.

- **If you write an operating system, then you know what you're doing.**
  - Implement your own locks.
  - Implement your own atomic primitives.
  - Hand-code your barriers.
  - **Read your CPU's manual first.**

- Your CPU can, and will reorder your memory accesses.

- Identify where this is a problem,

- and use your platform's synchronization primitives appropriately.