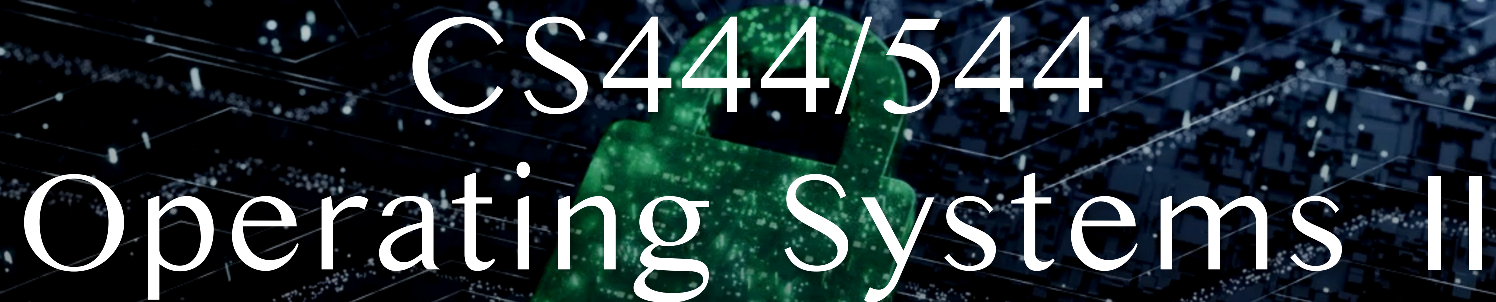


CS444/544 Operating Systems II

Prof. Sibin Mohan

Spring 2022 | Lec. 13: Locks 2

Adapted from content originally created by: Prof. Yeongjin Jang

2nd Candidate: **xchg_lock** Result

- **Consistent!**

```
[jangye@os2 (master) ~/test/lock-example$] ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:     946.416 ms
```

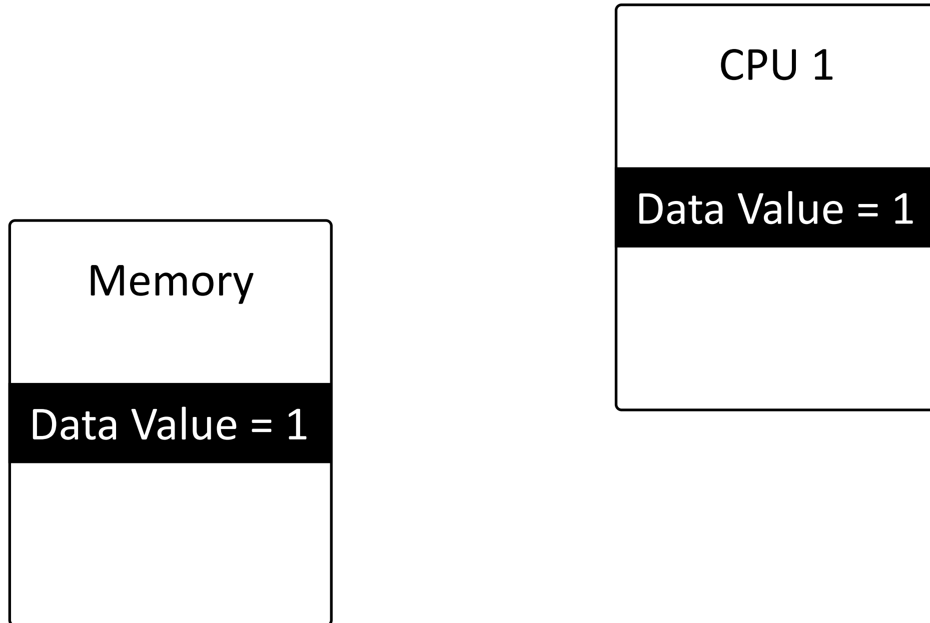
High overheads!

cache coherency!

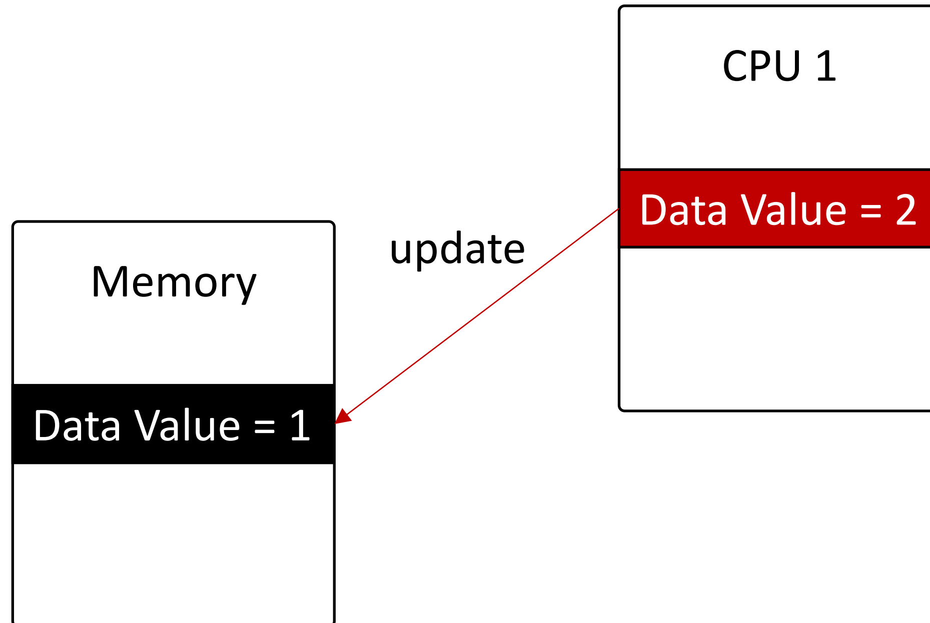
https://gitlab.unexploitable.systems/root/lock-example

[You can run this by cloning the repo!]

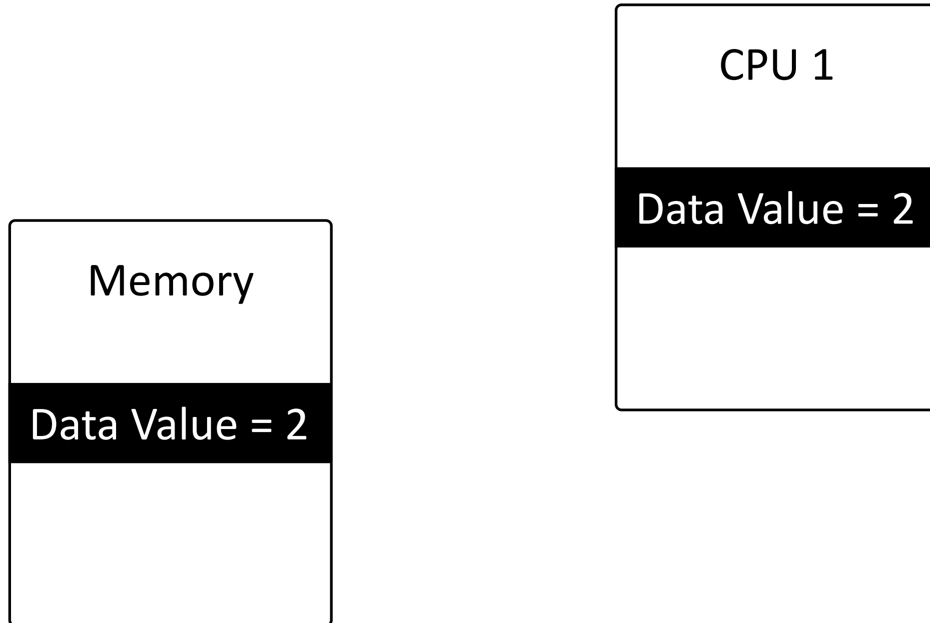
Detour | Cache Coherency



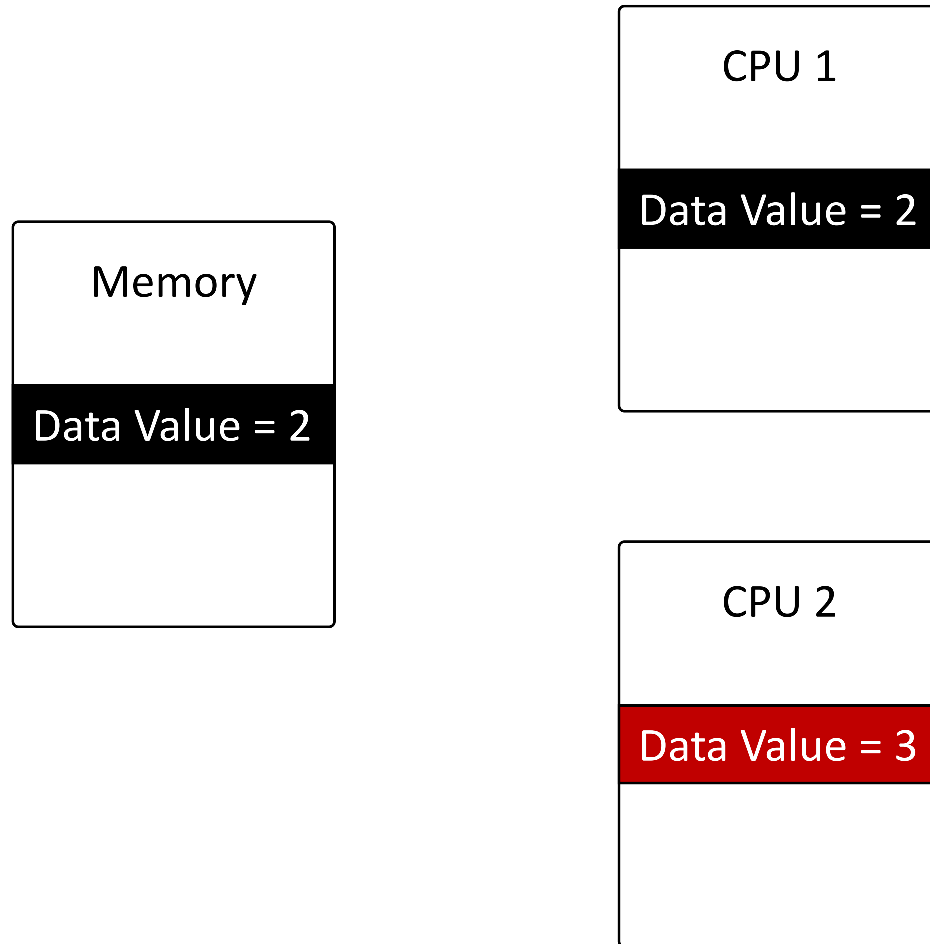
Detour | Cache Coherency



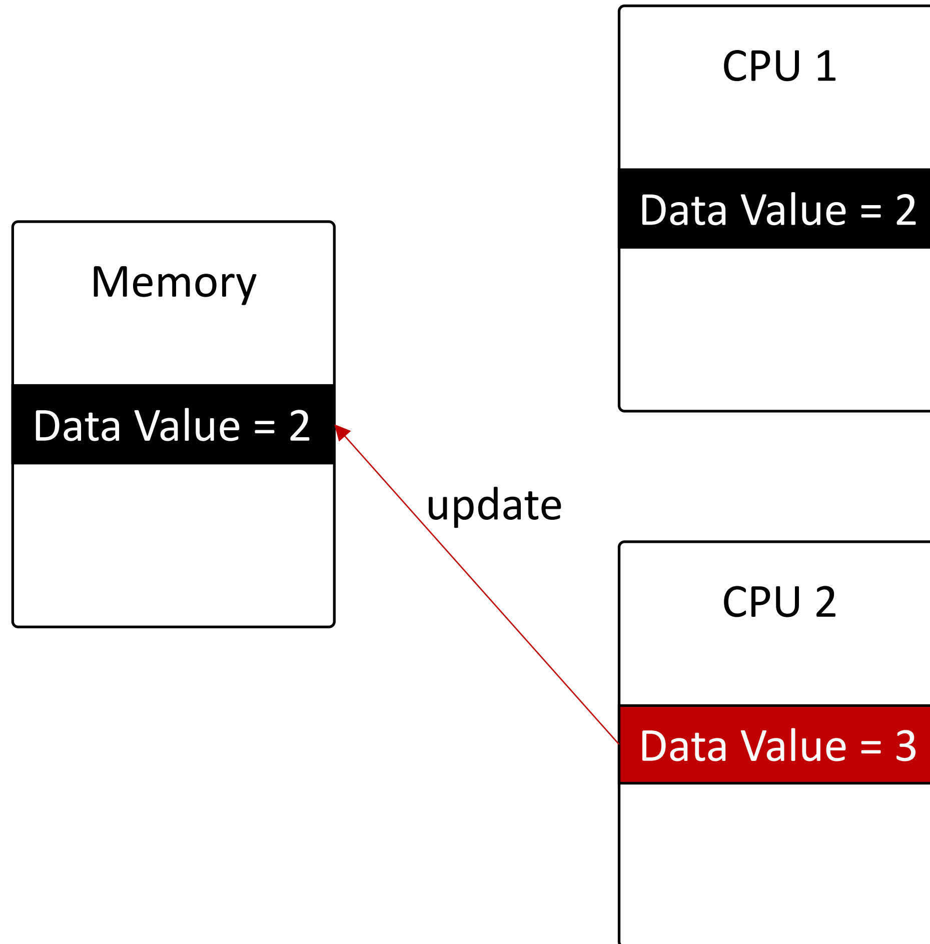
Detour | Cache Coherency



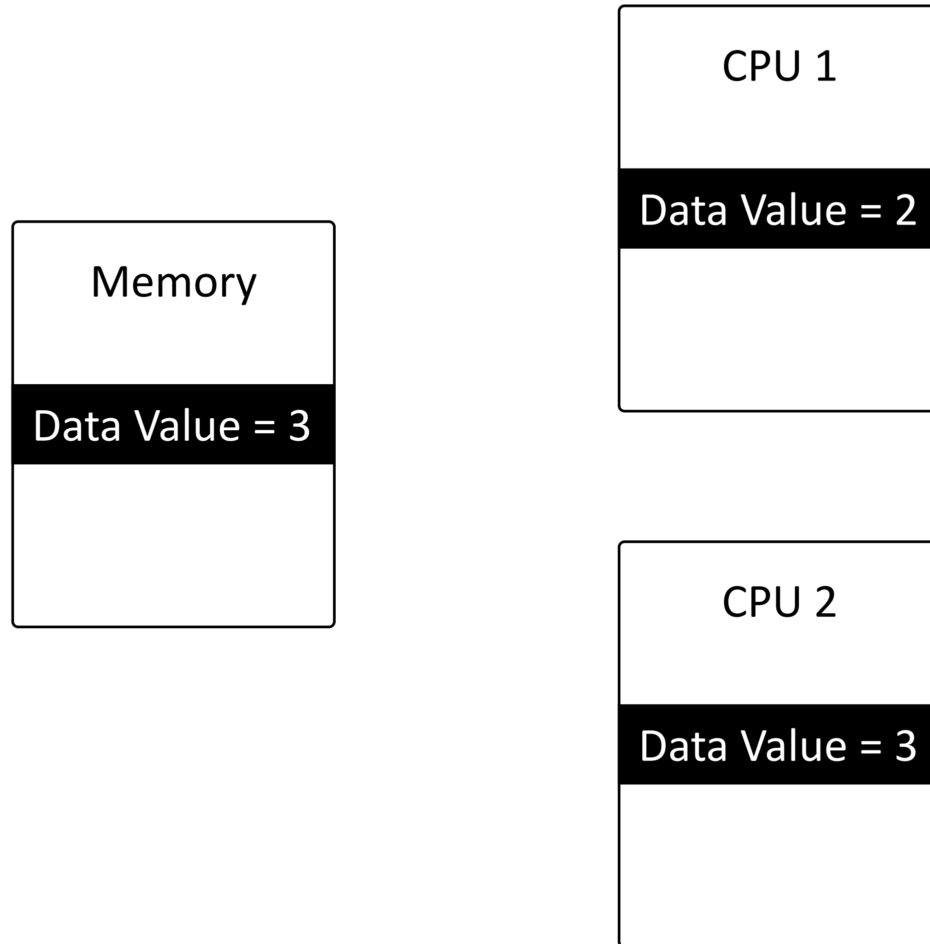
Detour | Cache Coherency



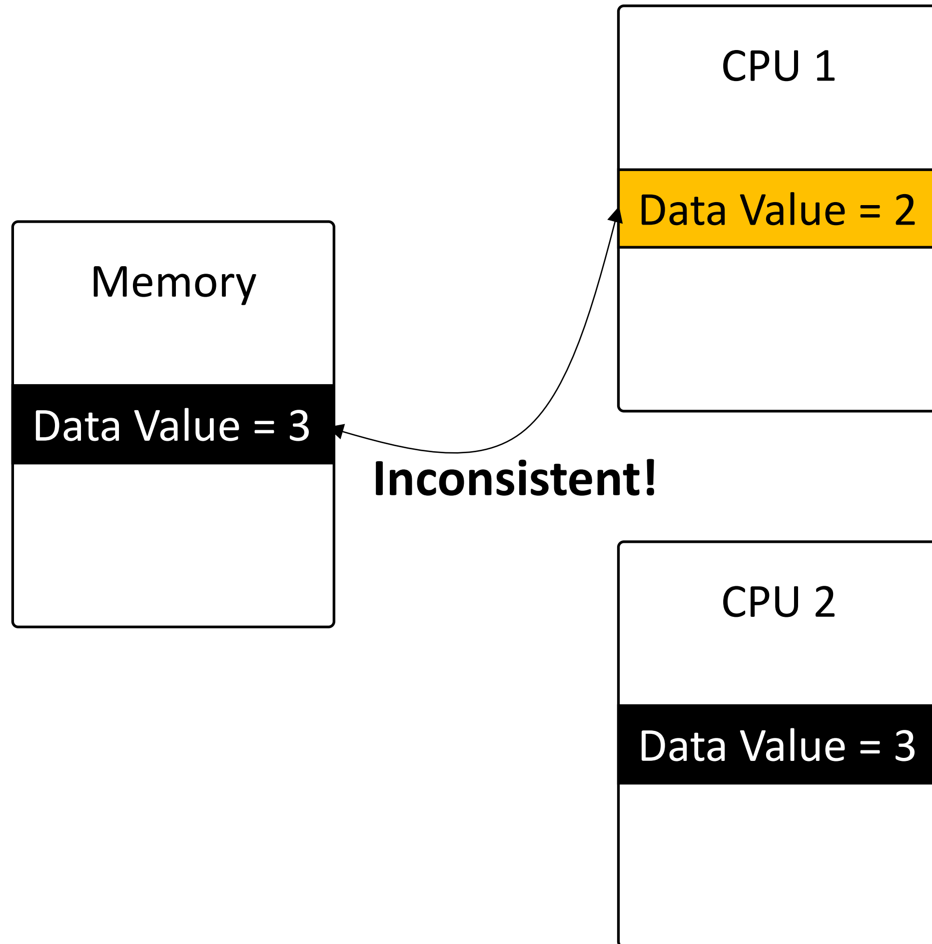
Detour | Cache Coherency



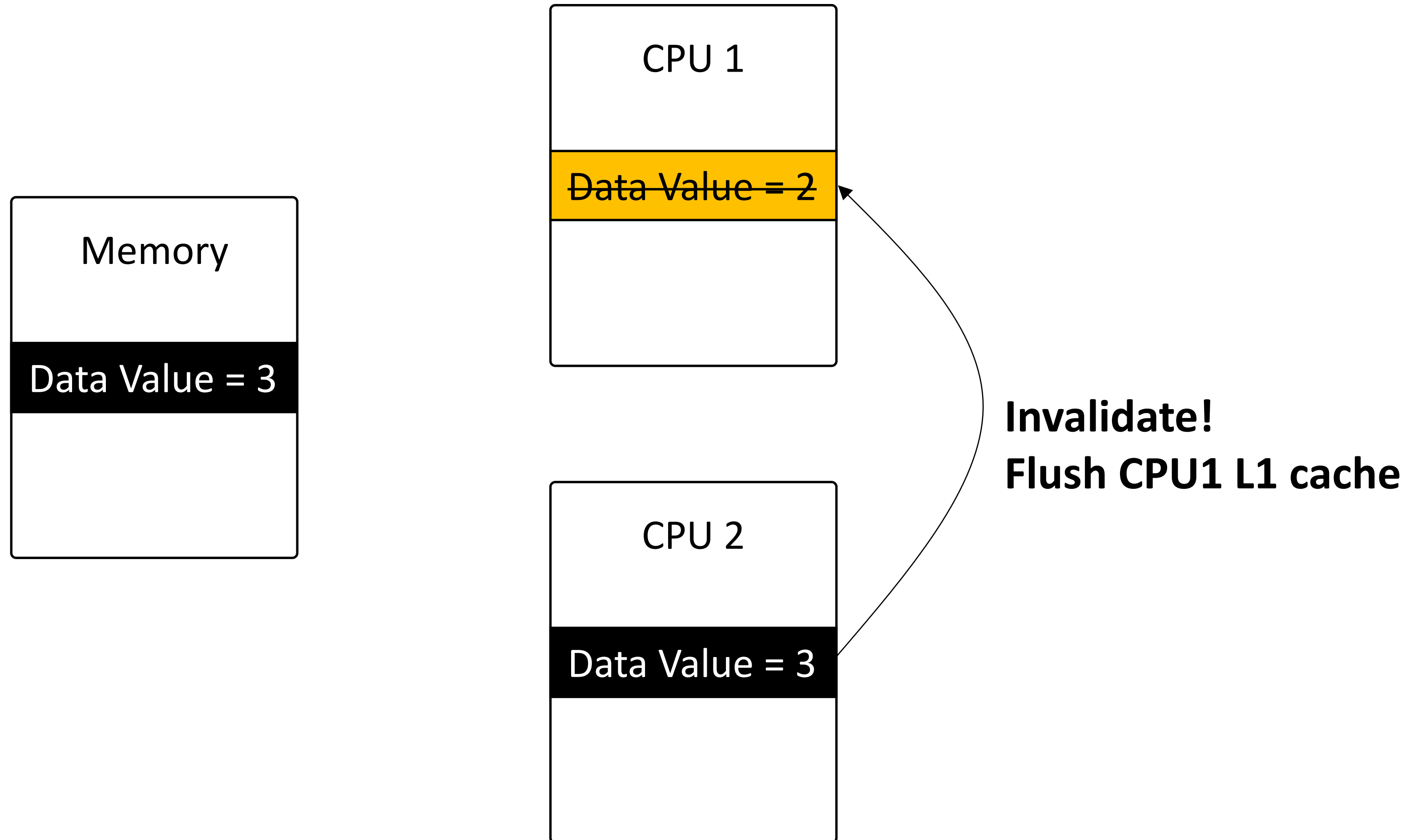
Detour | Cache Coherency



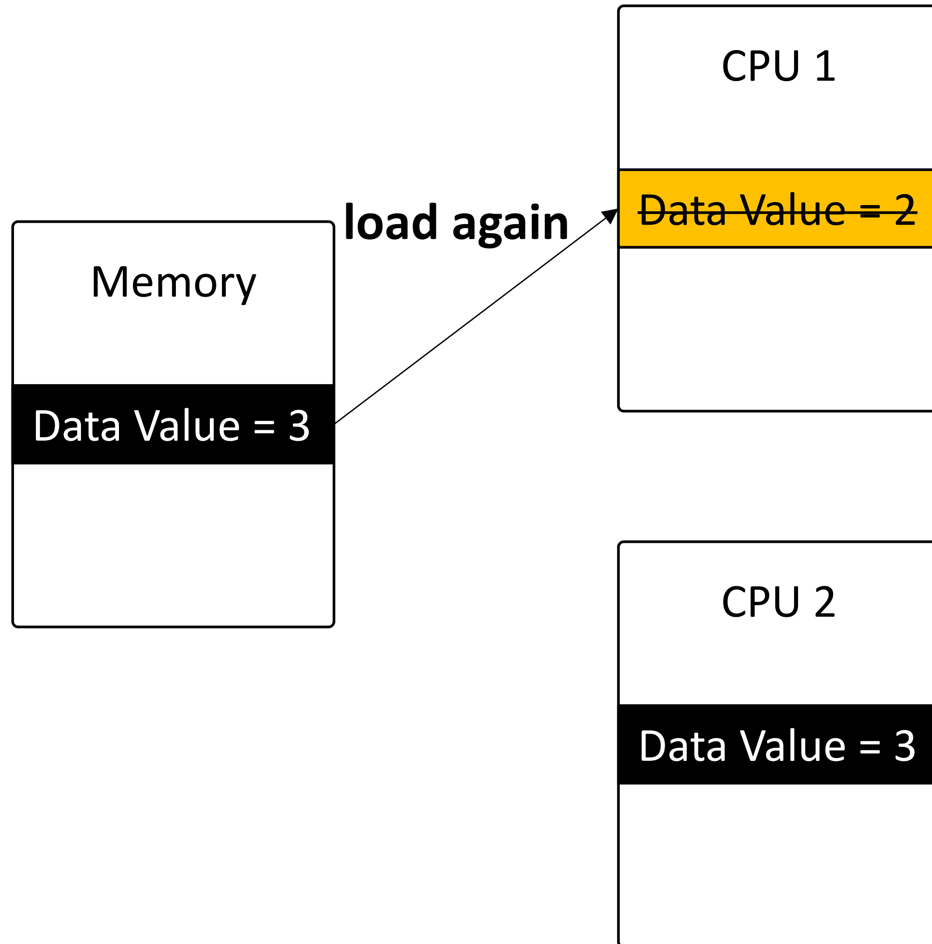
Detour | Cache Coherency



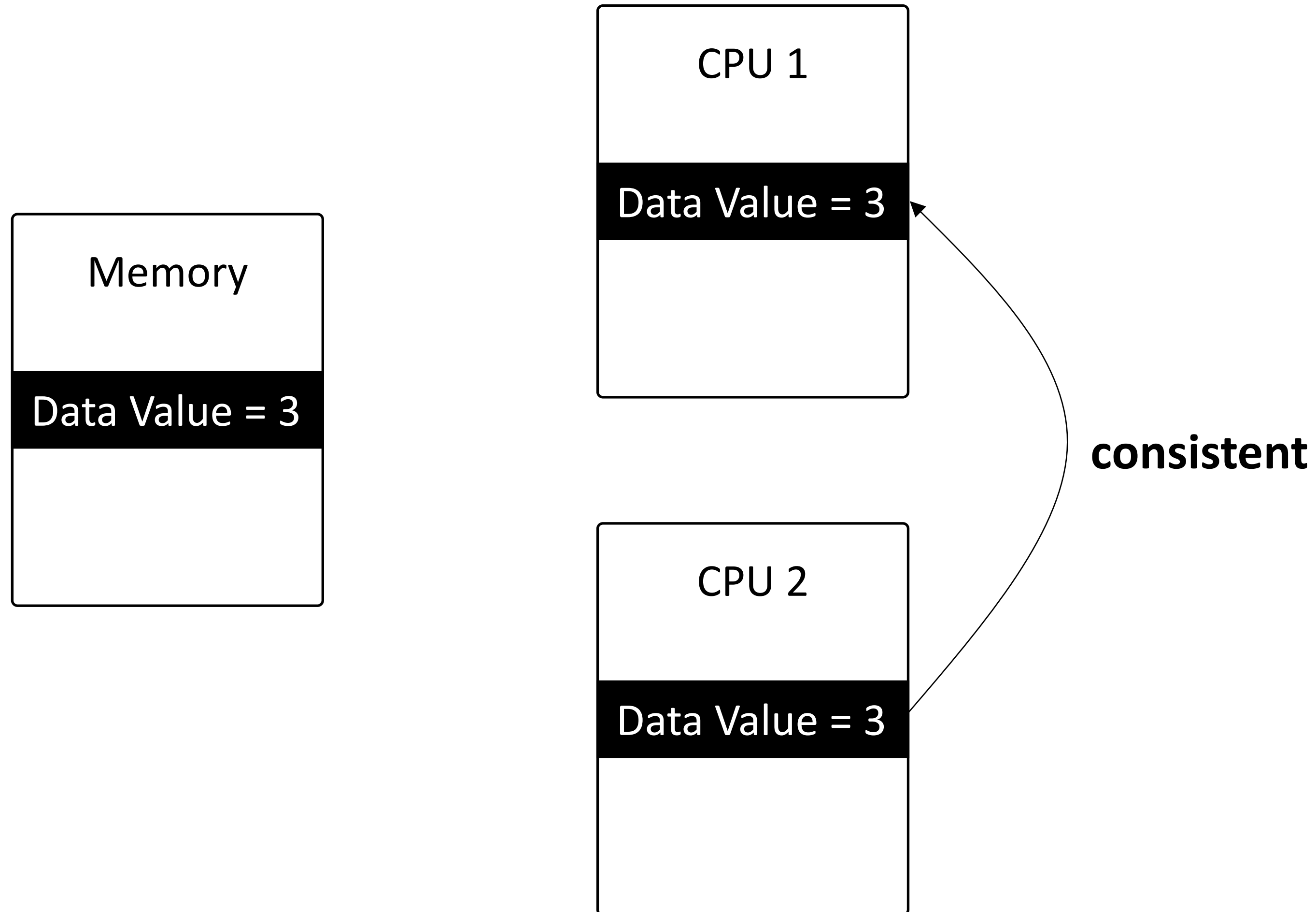
Detour | Cache Coherency



Detour | Cache Coherency



Detour | Cache Coherency



Back to `xchg`

- Atomic xchg instruction loads/stores data at the same time
  - There is no gap for race condition
- But it could **cause cache contention!**
  - Many threads update the same '`lock`' variable
  - Multiple CPUs cache '`lock`' variable
  - **Update to `lock` invalidates cache!**

```
[jangye@os2 (master) ~/test/lock-example$] ./lock xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:     946.416 ms
```

The Problem with xchg

xchg and Cache Coherence

- `xchg` **always updates** the value
- **Every** `xchg` instruction swaps in "1" into the memory location, **`lock`**

```
void
xchg_lock(volatile uint32_t *lock) {
    while(xchg(lock, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

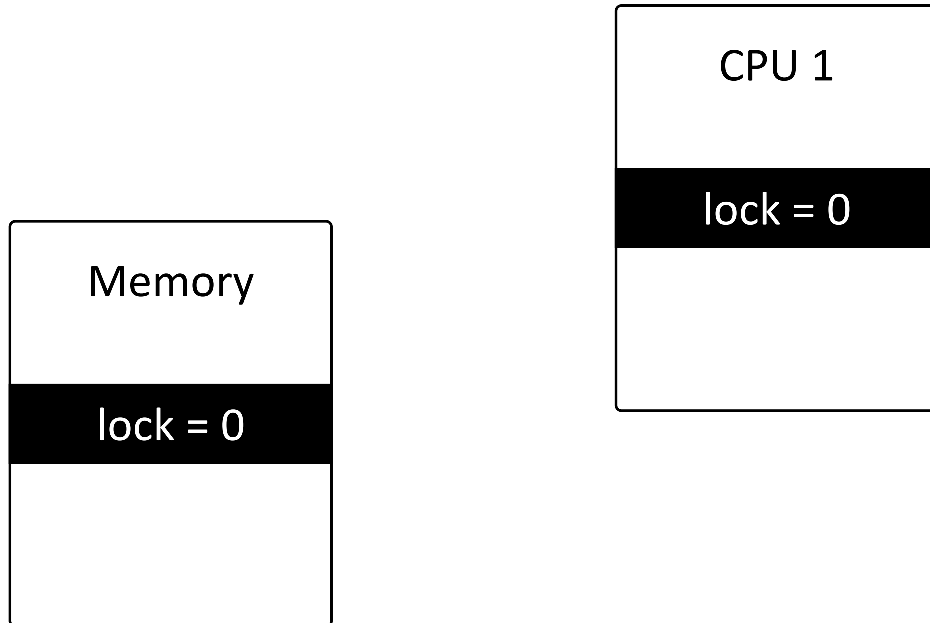
start: `lock` was "**0**"
result after xchg:
`lock` → 1
`eax` → 0

start: `lock` was "**1**"
result after xchg:
`lock` → 1
`eax` → 1

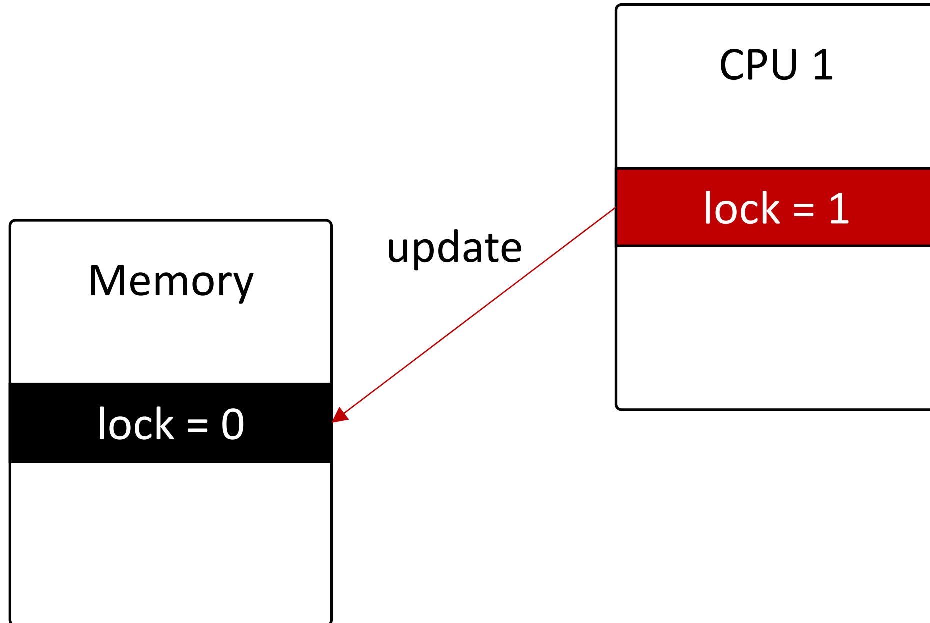
Loaded into **cache** (while loop)

Cache invalidations for all other threads!

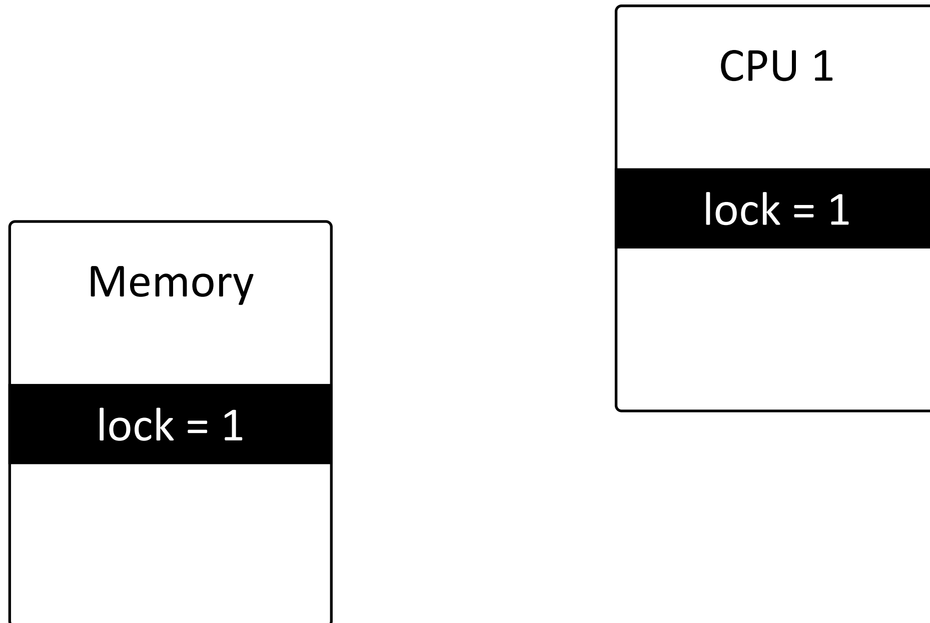
xchg & Cache Coherency Example



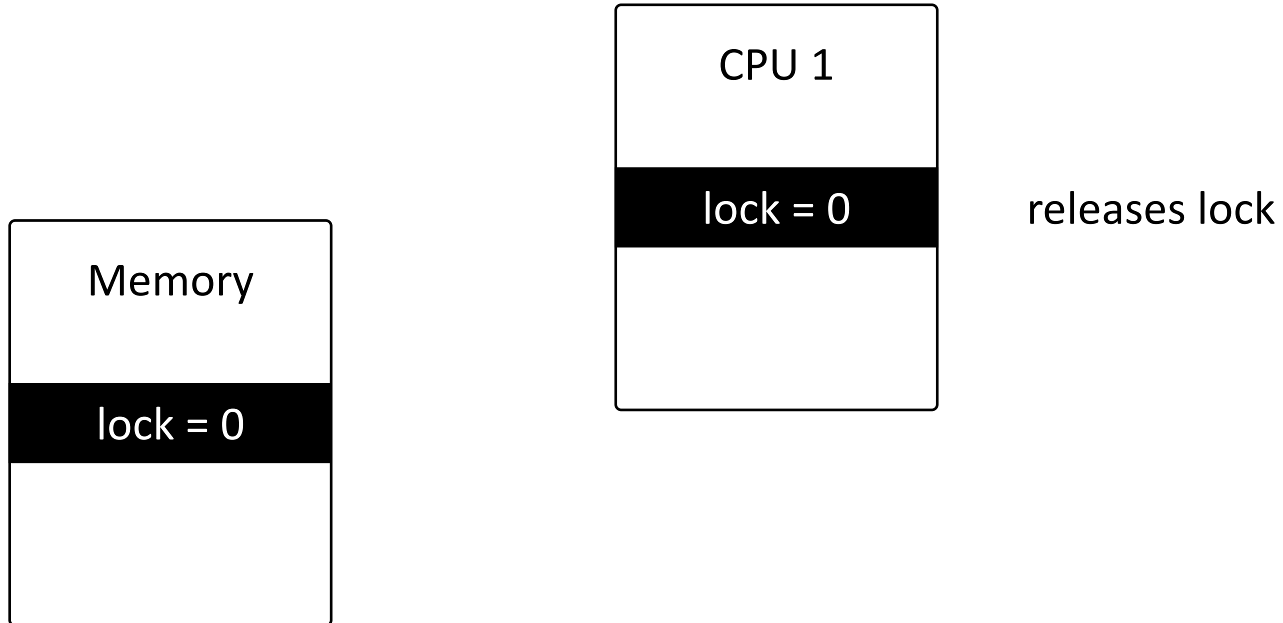
xchg & Cache Coherency Example



xchg & Cache Coherency Example

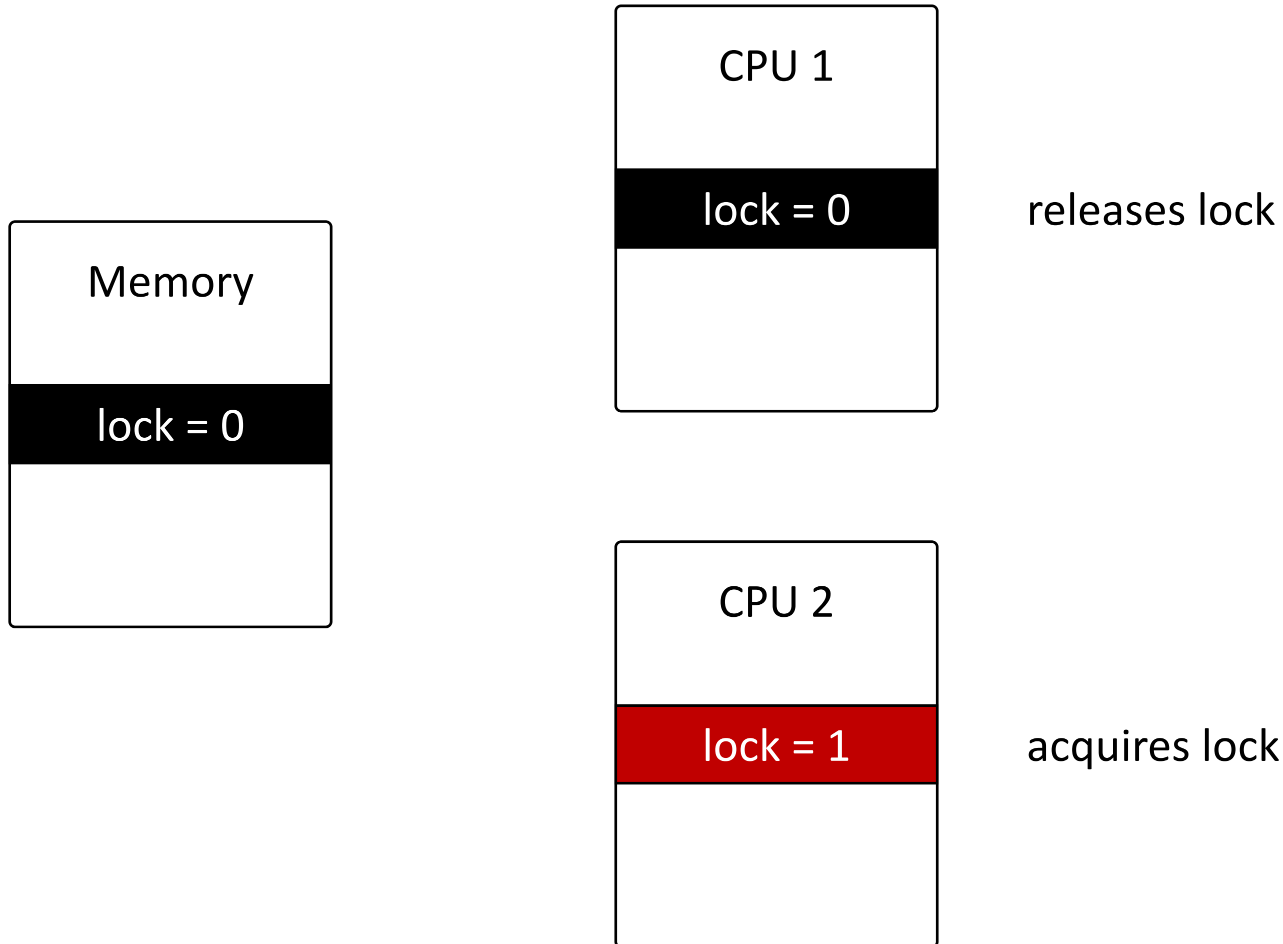


xchg & Cache Coherency Example

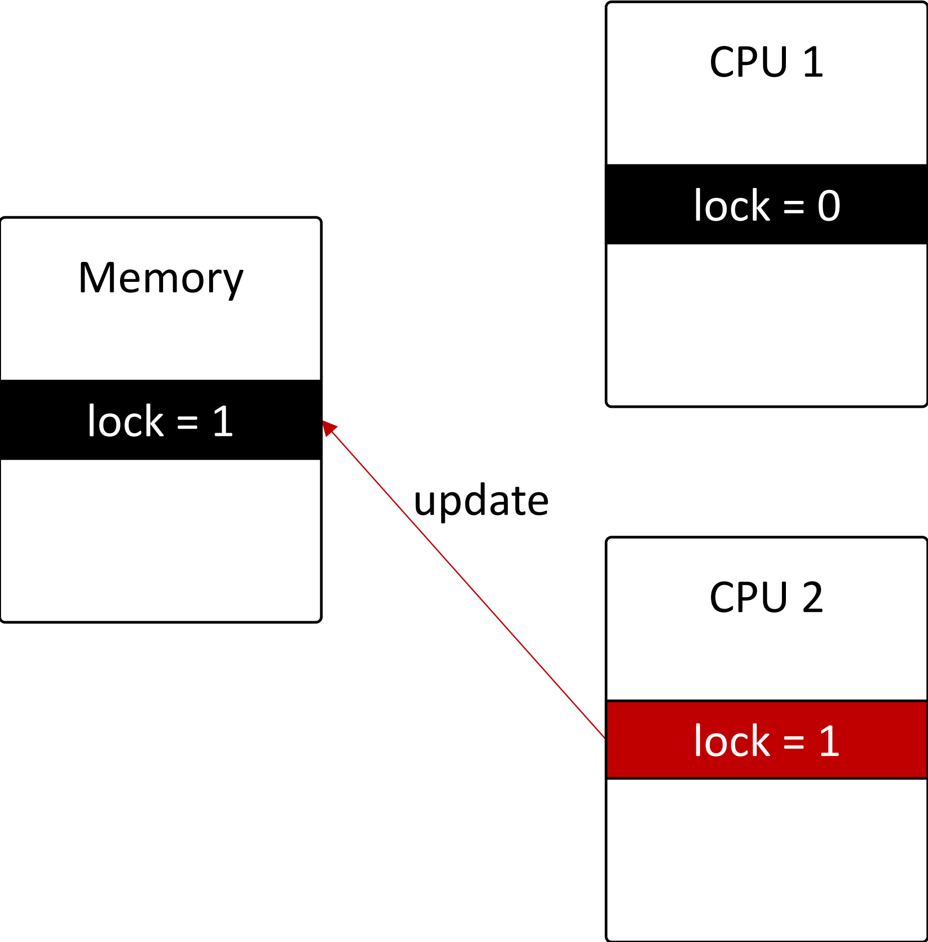


releases lock

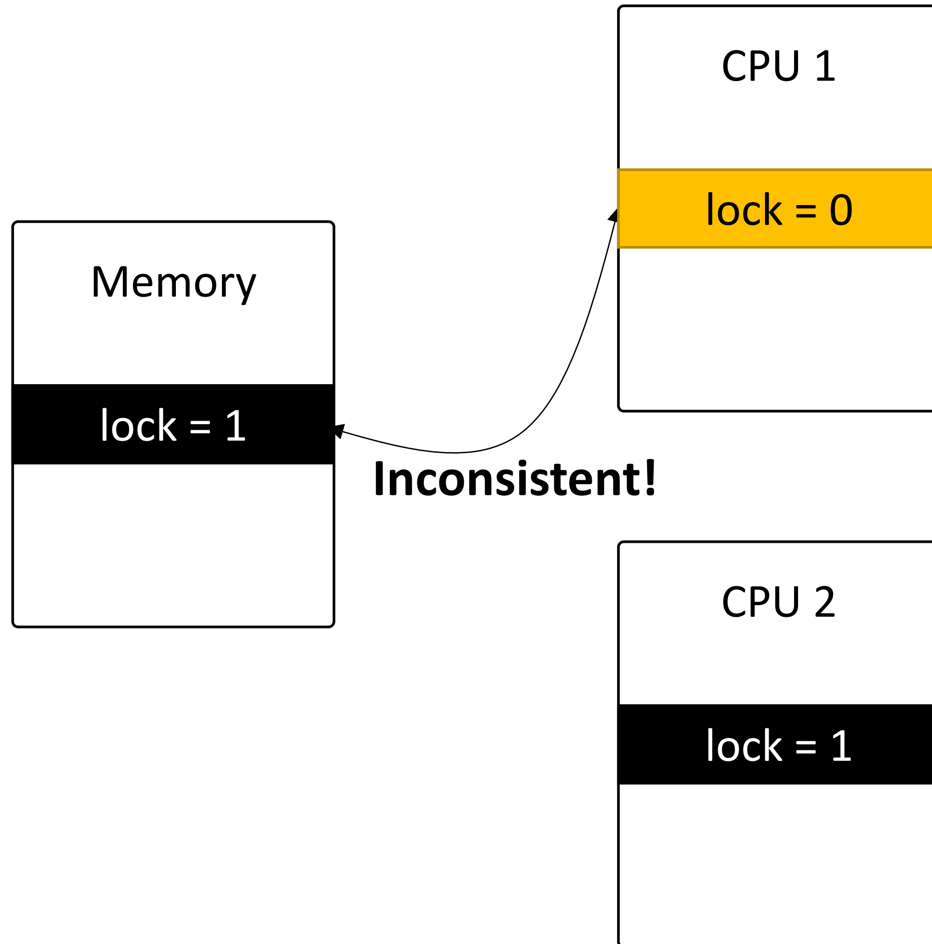
xchg & Cache Coherency Example



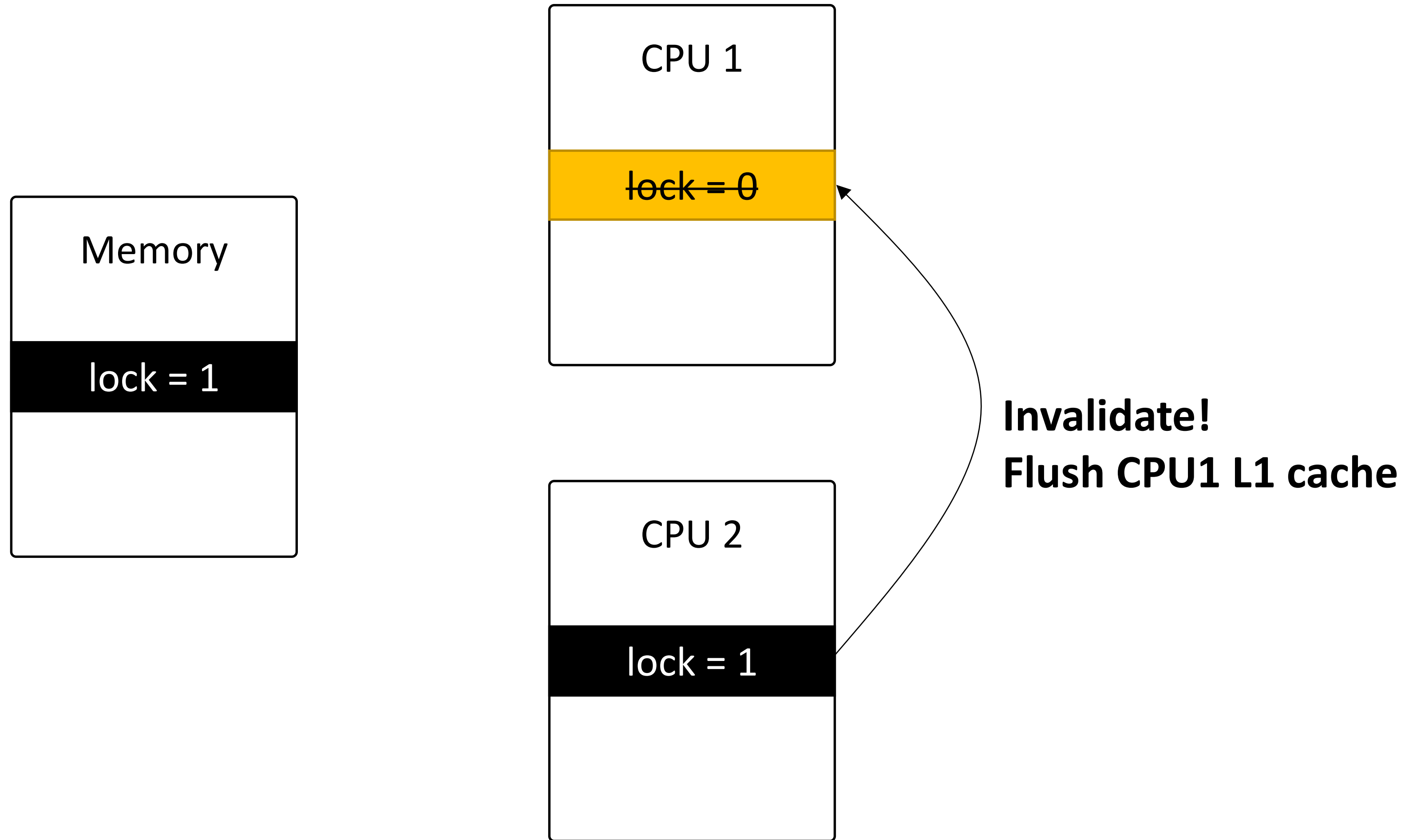
xchg & Cache Coherency Example



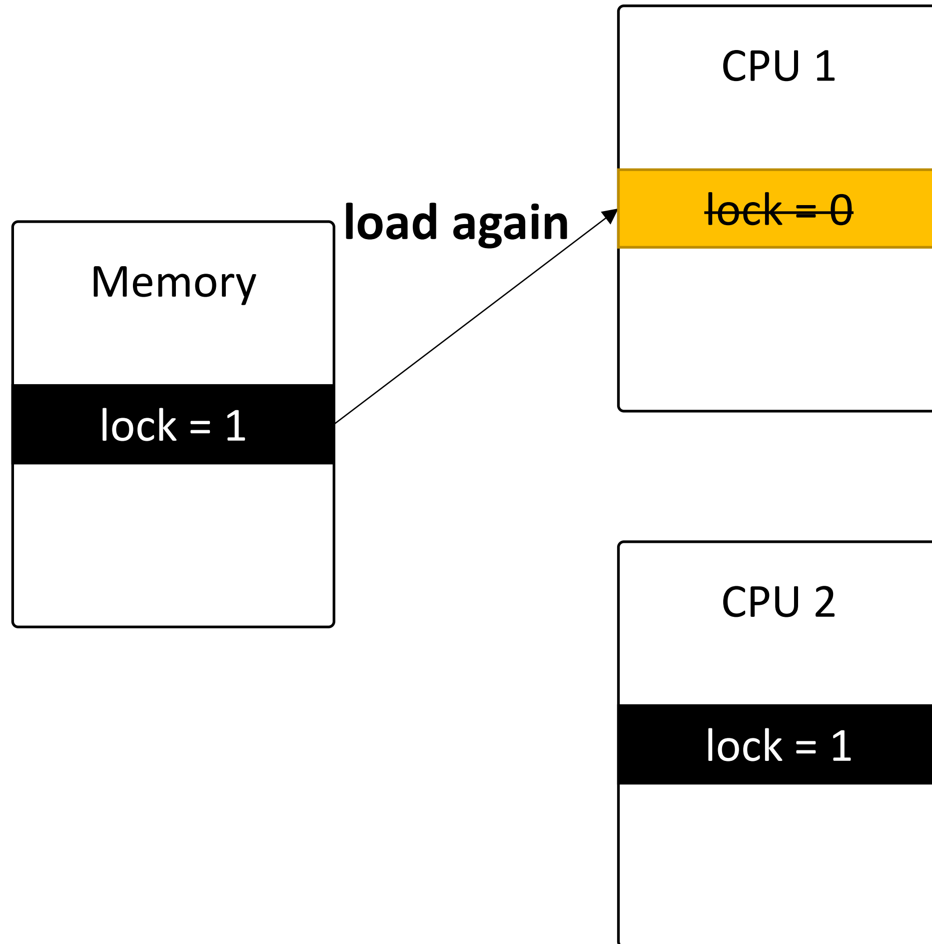
xchg & Cache Coherency Example



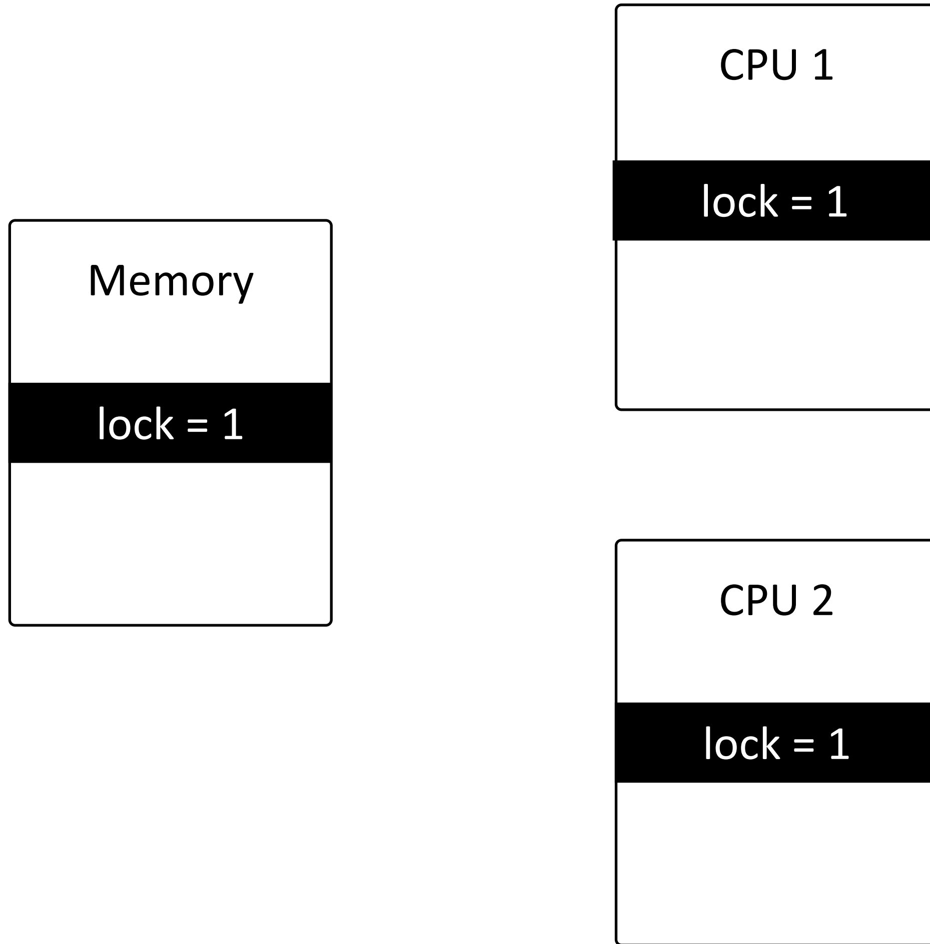
xchg & Cache Coherency Example



xchg & Cache Coherency Example

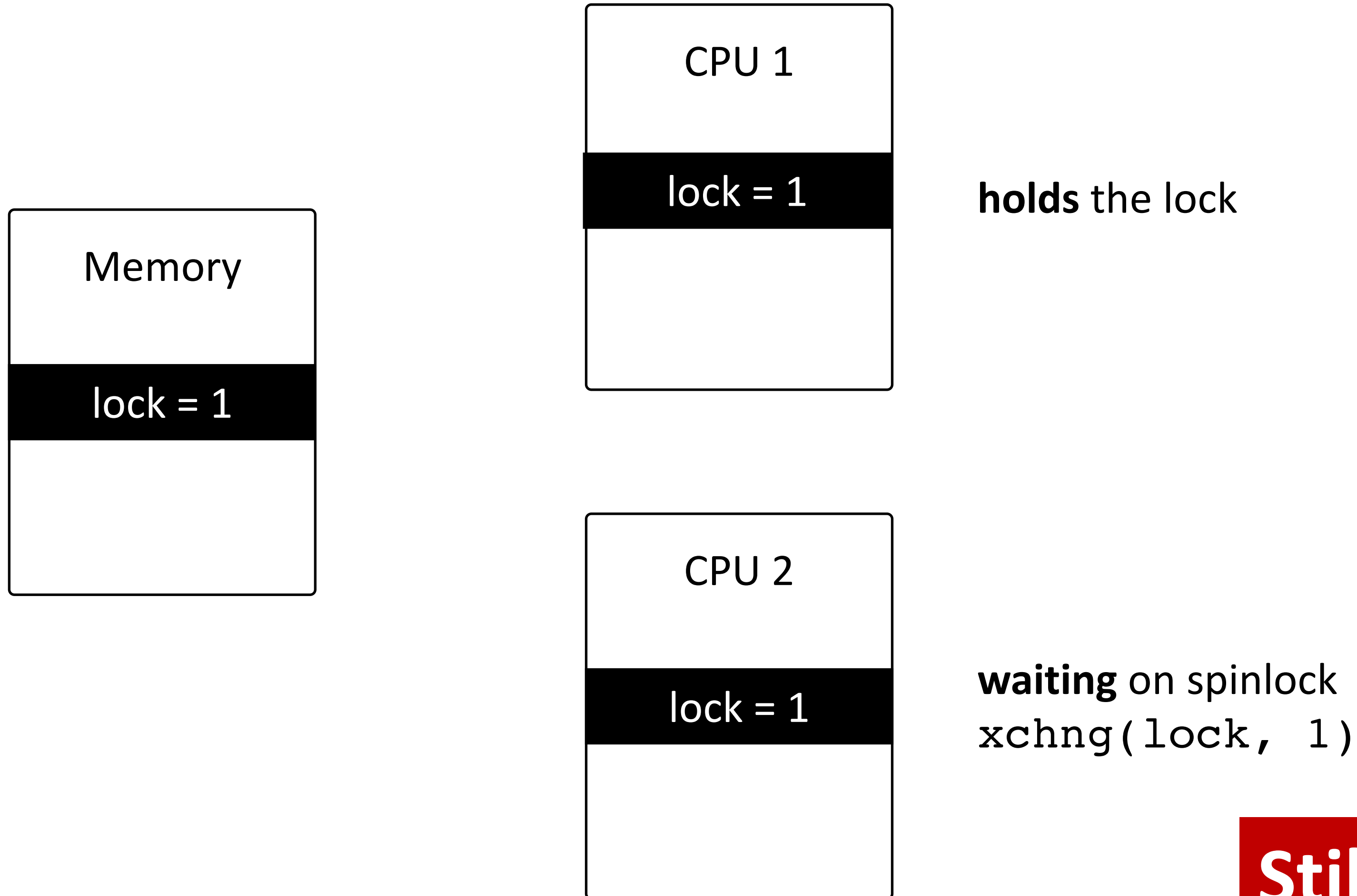


xchg & Cache Coherency Example



Hang on a minute…

What If…



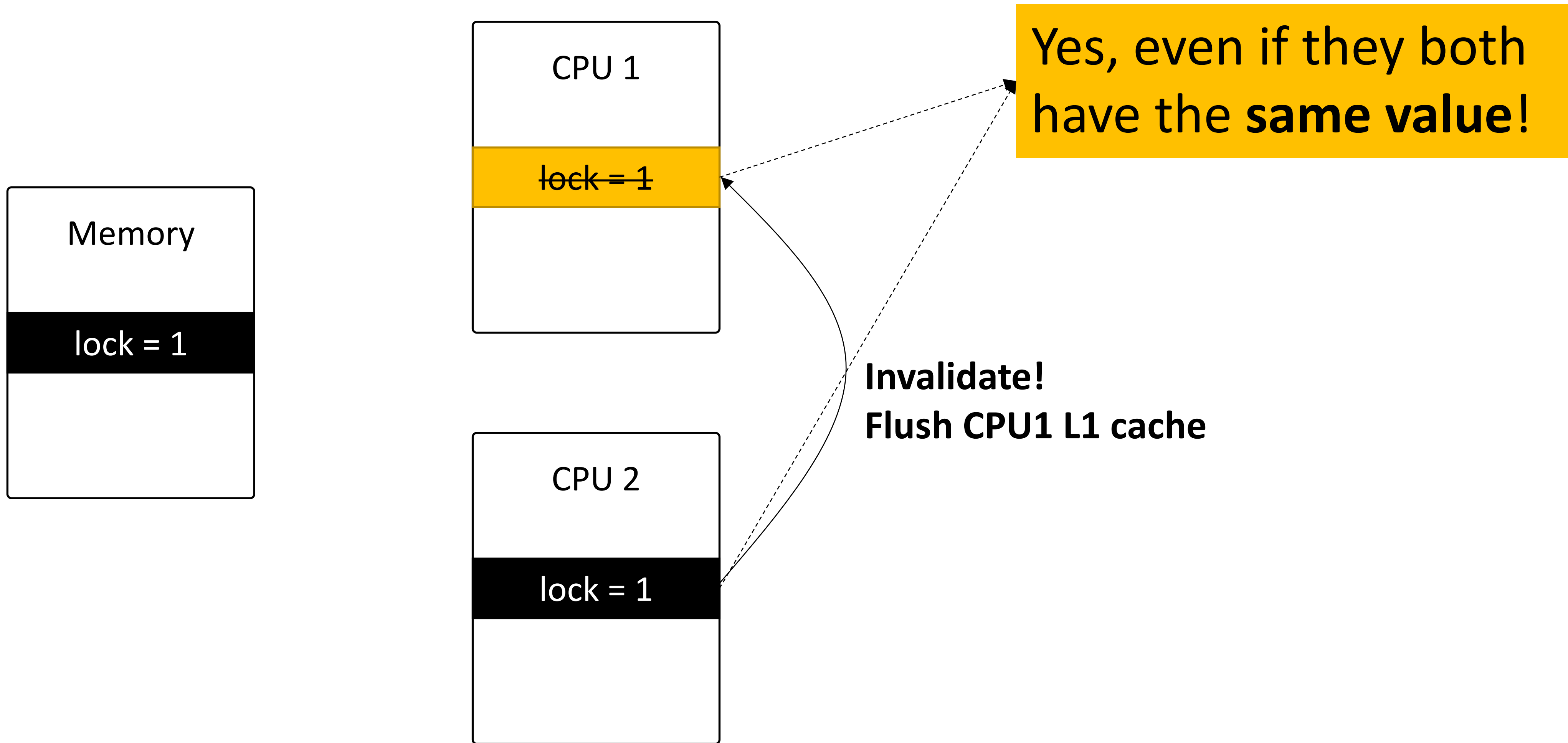
holds the lock

What now?

waiting on spinlock
`xchng(lock, 1)`

Still suffers from Cache invalidations!

What If…



Multiple Threads → Multiple Cache Invalidations!

- Previous example was for two threads
- In our `lock` implementation, we have **30** threads!

- Only one thread can be in the critical section
- Remaining 29 threads → **causing cache invalidations!!!**
- `xchg` implementation can be **really slow**!

- How slow?

Let's Measure the Cache Misses

- `perf` → built in Linux command to monitor hardware events
  - e.g., cache misses

```
[jangye@os2 (master) ~/test/lock-example$] taskset -c 1 ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:   3612.080 ms

 Performance counter stats for './lock xchg':

            84,130      L1-dcache-load-misses:u

       3.613420345 seconds time elapsed

       3.571214000 seconds user
       0.032928000 seconds sys
```

Single CPU
no cache coherence invalidations
84,130 L1 cache misses

```
[jangye@os2 (master) ~/test/lock-example$] ./perf-lock.sh xchg
Counting 10000 with 30 threads using XCHG_LOCK...
Count: 300000, elapsed Time:    943.568 ms

 Performance counter stats for './lock xchg':

        16,825,378      L1-dcache-load-misses:u

       0.946774344 seconds time elapsed

      23.707364000 seconds user
       0.097770000 seconds sys
```

30 CPUs
many cache coherence invalidations
16,825,378 L1 cache misses

200x worse!

Test-and-Set (xchg)

Pros

- Synchronizes threads well!

Cons

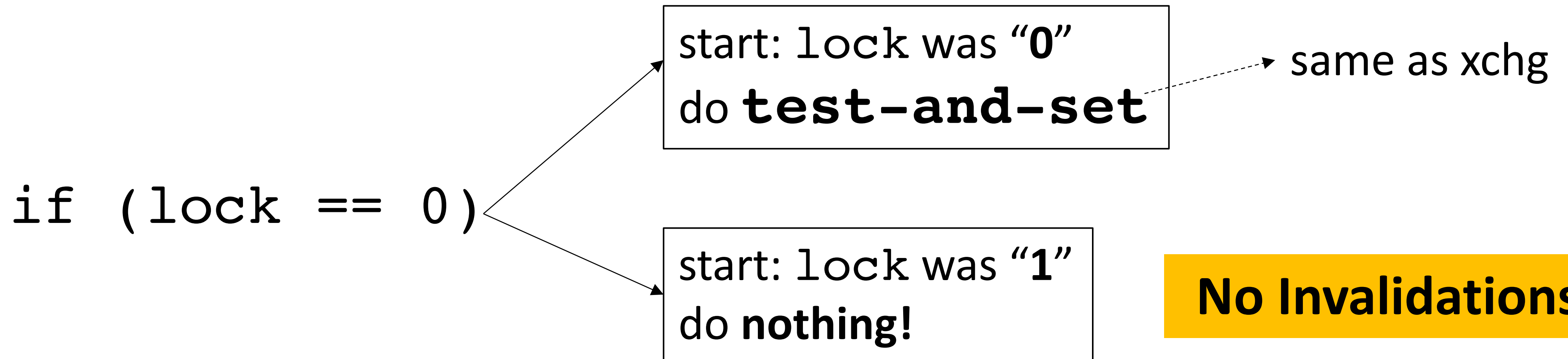
- SLOW
- Lots of cache miss

How do we solve this? **Can we solve it?**

Solution | test and test-and-set

- **Why update** the `lock` if its value is already '1'?

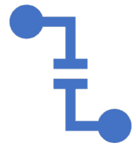
- `` `test and test-and-set' ``
- **check value first!**



Test and Test-and-set in x86

- `lock cmpxchg [update-value], [memory]`
  - Compare the value in `[memory]` with `%eax` **test**
  - If matched, exchange value in `[memory]` with `[update-value]` **test-and-set**
- Otherwise, **do not perform exchange**
- Must use with '`lock-prefix`' for thread synchronization

- `xchg(lock, 1)`
  - Lock = 1
  - Returns old value of the `lock`
- **`cmpxchg(lock, 0, 1)`**
  - Arguments: Lock, test value, update value
  - Returns old value of `lock`



CAVEAT

- `xchg` is an atomic operation in x86
- **`cmpxchg`** is **not** an atomic operation in x86
  - Must be used with lock prefix to guarantee atomicity
- **`lock`** `cmpxchg`

3rd Candidate: cmpxchg_lock

- **cmpxchg_lock**
  - Use `cmpxchg` to set lock = 1
  - Do not update if lock == 1
  - Only write 1 to lock if lock == 0
- **xchg_unlock**
  - Use `xchg_unlock` to set lock = 0
  - Because we have 1 writer and
  - This always succeeds

```
void *
count_cmpxchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        cmpxchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

critical section

```
void
cmpxchg_lock(volatile uint32_t *lock) {
    while(cmpxchg(lock, 0, 1));
}
```

3rd Candidate: `cmpxchg_lock` Cache Results

- **Consistent!**

```
[jangye@os2 (master) ~/test/lock-example$] ./perf-lock.sh cmpxchg
Counting 10000 with 30 threads using CMPXCHG_LOCK...
Count: 300000, elapsed Time:  1123.668 ms

 Performance counter stats for './lock cmpxchg':

       18,955,490      L1-dcache-load-misses:u

      1.125629906 seconds time elapsed

     28.905853000 seconds user
      0.111732000 seconds sys
```

But still showing lots of cache misses → **more than xchg!**
Why????

Intel CPU is TOO COMPLEX

This *[cmpxchg]*instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processors bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

`cmpxchg` → **designed** to be test and test-and-set instruction

Intel CPU complexity → so **always update value regardless the result of comparison**

Lame! 🙄

Let's implement test and test-and-set in software instead

4th Candidate: Test and Test & Set [Software?]

- **`tts_xchg_lock`**
- **Wait** until **lock becomes 0**
- **After** lock == 0
  - `xchg (lock, 1)`
  - This **only updates lock = 1 if lock was 0**

- Why `xchg`, why not ***lock = 1** directly?
  - while and xchg are not atomic
  - Load/Store must happen at same time!

```
void *
count_tts_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        tts_xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

critical section

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

4th Candidate TTS Result

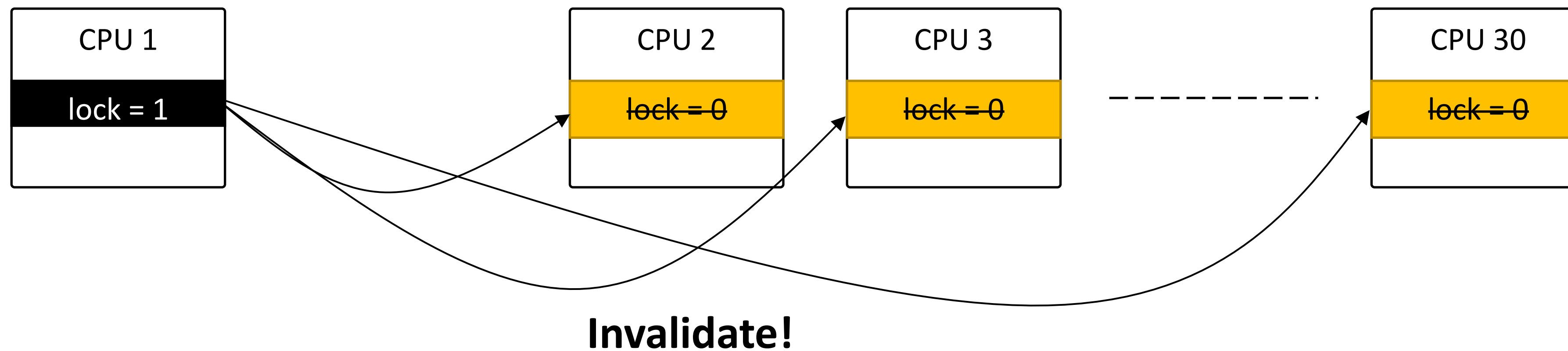
- **Consistent!**

```
[jangye@os2 (master) ~/test/lock-example$] ./perf-lock.sh tts
Counting 10000 with 30 threads using TTS_LOCK...
Count: 300000, elapsed Time:    498.578 ms

 Performance counter stats for './lock tts':

        14,426,153      L1-dcache-load-misses:u

       0.501079419 seconds time elapsed

      14.039150000 seconds user
       0.105730000 seconds sys
```

- **Fewer cache misses** (by a bit)
- **Faster** (~500ms vs. 900 ~ 1200 ms)

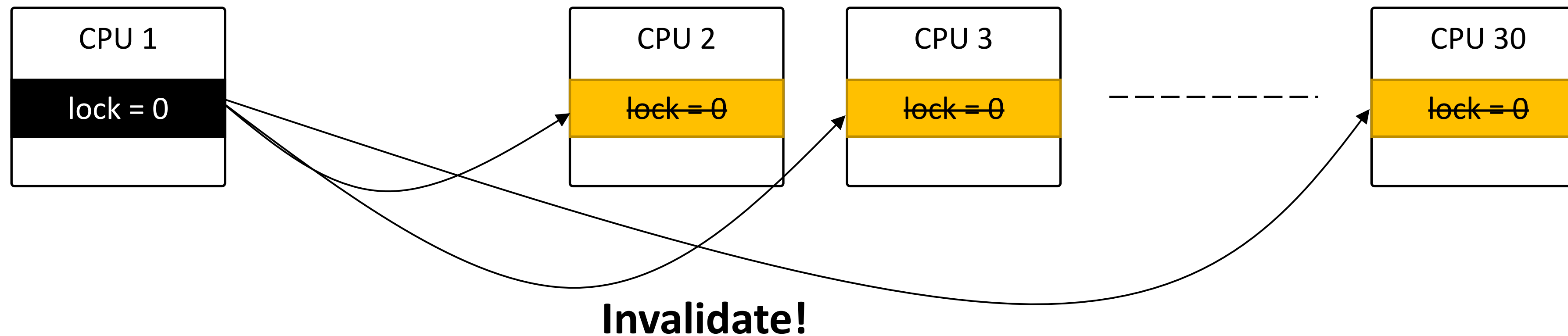
Still Slow and Many Cache Misses..

- Why do we still have so many misses?
- A thread **acquires** the lock [update 0 → 1]
  - **Invalidate caches in 29 other cores**



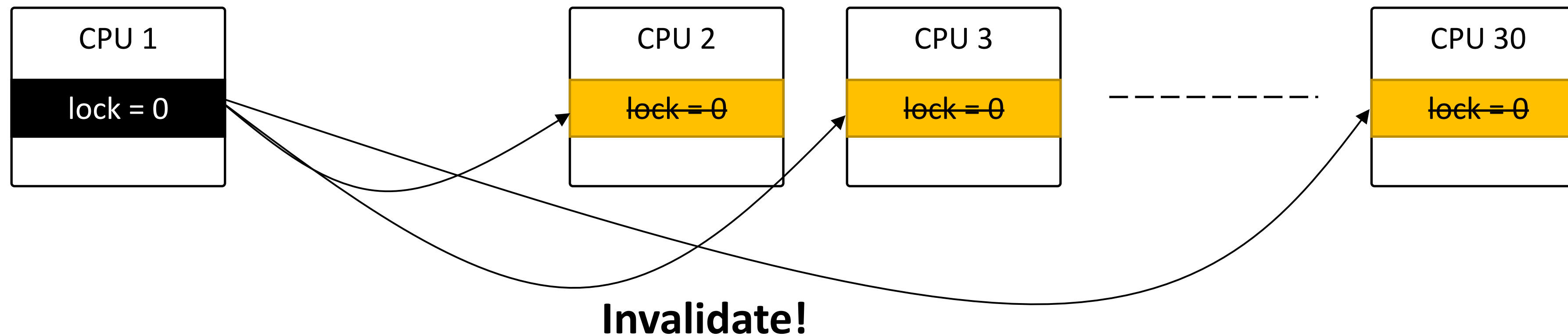
Still Slow and Many Cache Misses..

- Why do we still have so many misses?
- A thread **acquires** the lock [update 0 → 1]
  - **Invalidate caches in 29 other cores**
- A thread **releases** the lock [update 1 → 0]
  - **Invalidate caches in 29 other cores**



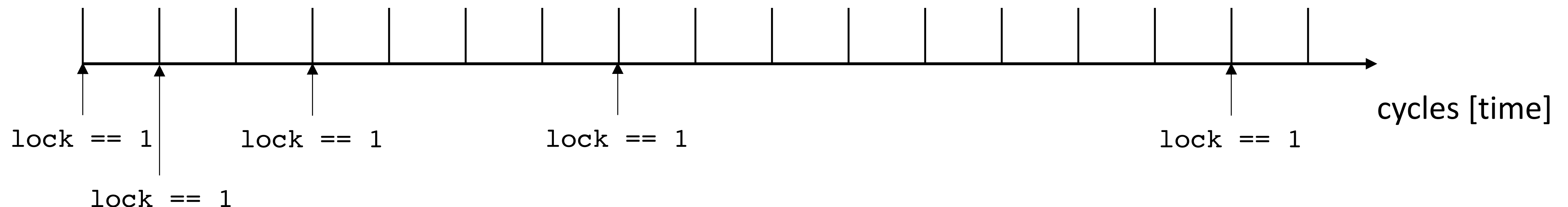
Still Slow and Many Cache Misses..

- **29 other cores** are all reading the variable lock
  - Immediately after invalidate, they load data to cache
  - Then invalidated again by either lock/release
  - This **happens every 3~4 cycles**



5th Candidate: Backoff Lock

- **Too much contention** on reading locks while only 1 thread runs critical sec
  - All other 29 cores running → `while (*lock == 1);`
  - This is the slow down factor
- Idea: **can we slow down that check?**
- Let's set a **wait time** once CPU checks whether the value of the lock is '1'
- Say, **exponential backoff**



5th Candidate: Backoff Lock

- `backoff_cmpxchg_lock(lock)`
- Try `cmpxchg`
  - If **success, acquire the lock**
  - If **fail**
    - Wait 1 cycle (pause) for 1st trial
    - Wait 2 cycles for 2nd trial
    - Wait 4 cycles for 3rd trial
    - …
    - Wait 65536 cycles for 17th trial
    - Wait 65536 cycles for 18th trial
- https://en.wikipedia.org/wiki/Exponential_backoff

```
void
backoff_cmpxchg_lock(volatile uint32_t *lock) {
    uint32_t backoff = 1;
    while(cmpxchg(lock, 0, 1)) {
        for (int i=0; i<backoff; ++i) {
            __asm volatile("pause");
        }
        if (backoff < 0x10000) {
            backoff <<= 1;
        }
    }
}
```

5th Candidate: Backoff Result

- **Consistent!**

faster than `pthread_mutex()`!

```
Counting 10000 with 30 threads using BACKOFF_LOCK...
Count: 300000, elapsed Time:    196.576 ms

 Performance counter stats for './lock backoff':

           232,980      L1-dcache-load-misses:u

       0.198420582 seconds time elapsed

       4.143351000 seconds user
       0.128103000 seconds sys
```

```
Counting 10000 with 30 threads using MUTEX_LOCK...
Count: 300000, elapsed Time:    457.688 ms

 Performance counter stats for './lock mutex':

         1,616,255      L1-dcache-load-misses:u

       0.459790351 seconds time elapsed

       0.467834000 seconds user
      12.165732000 seconds sys
```

Lock	Cache Misses [approx.]	Time [ms]
xchg	17 million	944
cmpxchng	19 million	1124
tts	14 million	500

- **Much fewer cache misses**
- **Faster! [less than 200ms!]**

Summary

- Mutex is implemented with **Spinlock**
  - Waits until lock == 0 with a while loop (why it's called spinlock)
- Naïve code implementation never works
  - **Load/Store must be atomic**
- **`xchg`** is a "test and set" atomic instruction
  - Consistent, however, many cache misses, **slow**! (950ms)
- Lock **`cmpxchg`** is a "**test and test&set**" **atomic** instruction
  - But Intel implemented this as xchg… **slow**! (1150ms)
- We can implement test-and-test-and-set (tts) with while + xchg
  - Faster! (500ms)
- We can also implement **exponential backoff** to reduce contention
  - **Much faster**! (**200ms**)