Basic concurrency

Emmett Witchel

CS380L

Concurrency Quiz

If two threads execute this program concurrently, how many different final values of X are there?

Thread 1

Initially, X == 0

Thread 2

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

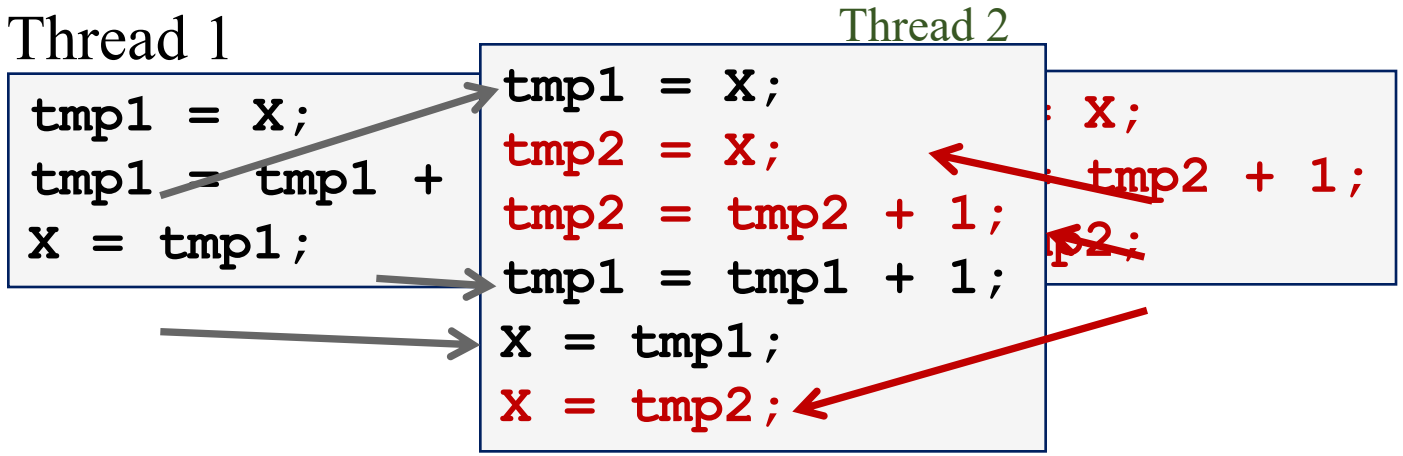
```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Answer:

A. 0
B. 1
C. 2
D. More than 2

Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, some synchronization is needed



If X==0 initially, X == 1 at the end. WRONG result!

Locks fix this with Mutual Exclusion

```
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

- Mutual exclusion ensures only safe interleavings
  - *But it limits concurrency, and hence scalability/performance*

Why Locks are Hard

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

```
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

Thread 0	Thread 1
`move(a, b, key1);`	
	`move(b, a, key2);`

DEADLOCK!

The correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
- Failure atomicity
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

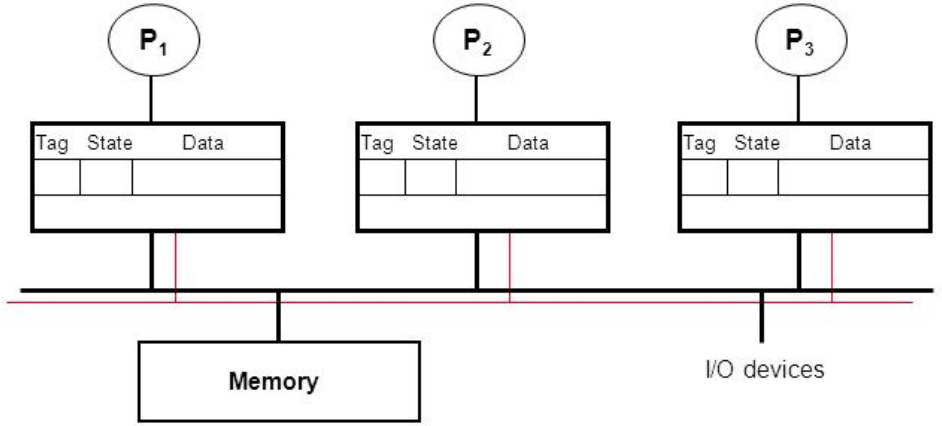
Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    - read a memory location into a register, **AND**
    - write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - LOCK prefix makes an instruction (e.g., add) atomic with respect to all cores by forcing exclusive ownership of the cache line and providing strong ordering
  - Compare and Exchange (atomic by nature, does not require lock prefix)
    - Compares the value in EAX with the first operand (destination operand)
    - If the two values are equal, the second operand (source operand) is loaded into the destination operand
    - Otherwise, the destination operand is loaded into EAX
  - Load linked/store conditional (PowerPC,Alpha, MIPS)

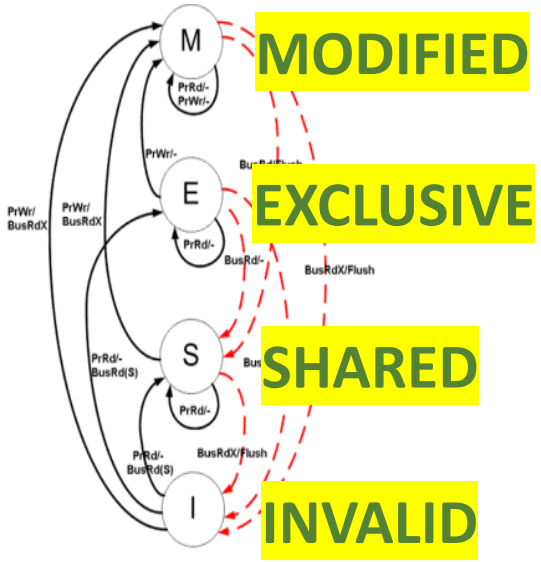
Background: Read-Modify-Write (RMW)

- Implementing locks requires read-modify-write operations
- Required effect is:
  - An atomic and isolated action
    1. read memory location **AND**
    2. write a new value to the location
  - RMW is *very tricky* in multi-processors
  - Cache coherence alone doesn't solve it




```
// (straw-person lock impl)
// Initially, lock == 0 (unheld)
lock() {
try:  load lock, R0
      test R0
      bnz try
      store lock, 1
}
```

Make this into a single (atomic hardware instruction)
OR
A set of instructions with well-defined protocol



Background: HW Support for RMW

Test & Set	CAS	Exchange, locked increment/decrement,	LLSC: load-linked store-conditional
Most architectures	Many architectures	x86	PPC, Alpha, MIPS

```
int TST(addr) {
  atomic {
    ret = *addr;
    if(!*addr)
      *addr = 1;
    return ret;
  }
}
```

```
bool cas(addr, old, new) {
  atomic {
    if(*addr == old) {
      *addr = new;
      return true;
    }
    return false;
  }
}
```

```
int XCHG(addr, val) {
  atomic {
    ret = *addr;
    *addr = val;
    return ret;
  }
}
```

```
bool LLSC(addr, val) {
 ret = *addr;
 atomic {
   if(*addr == ret) {
     *addr = val;
     return true;
  }
  return false;
}
```

```
void lock(lock) {
  while(CAS(&lock, 0, 1) != true);
}
```

Implementing Locks with Test&set

```
int lock_value = 0;
int* lock = &lock_value;
```

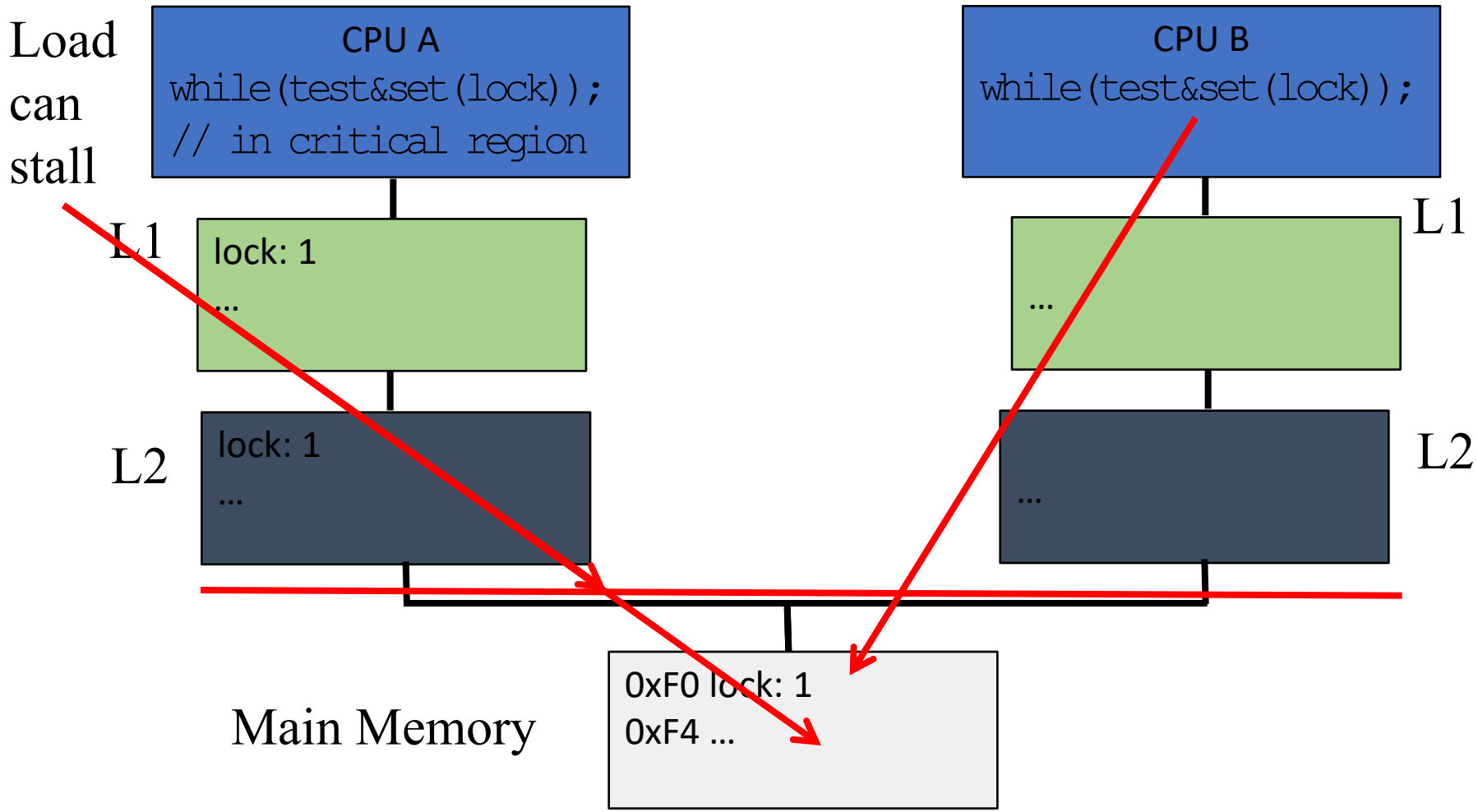
```
Lock::Acquire() {
while (test&set(lock) == 1)
   ; //spin
}
```

```
Lock::Release() {
    *lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



Cheaper Locks with Cheap busy waiting

Using Test&Test&Set

```
Lock::Acquire() {
while (test&set(lock) == 1);
}
```

Busy-wait on in-memory copy

```
Lock::Acquire() {
while(1) {
  while (*lock == 1) ; // spin just reading
  if (test&set(lock) == 0)  break;
}
```

Busy-wait on cached copy

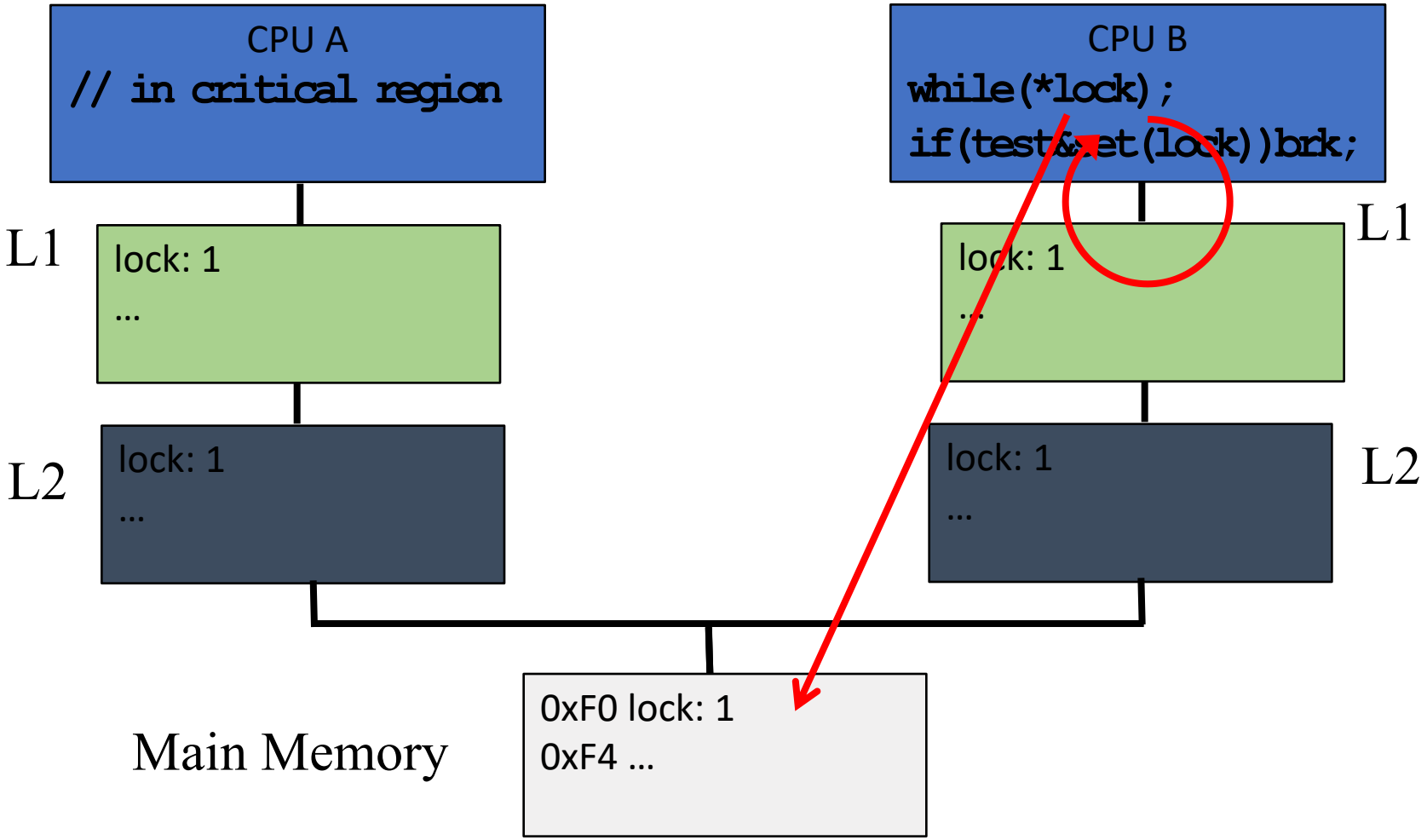
```
Lock::Release() {
   *lock = 0;
}
```

```
Lock::Release() {
*lock = 0;
}
```

- What is the problem with this?
  - A. CPU usage B. Memory usage C. Lock::Acquire() latency
  - D. Memory bus usage E. Does not work

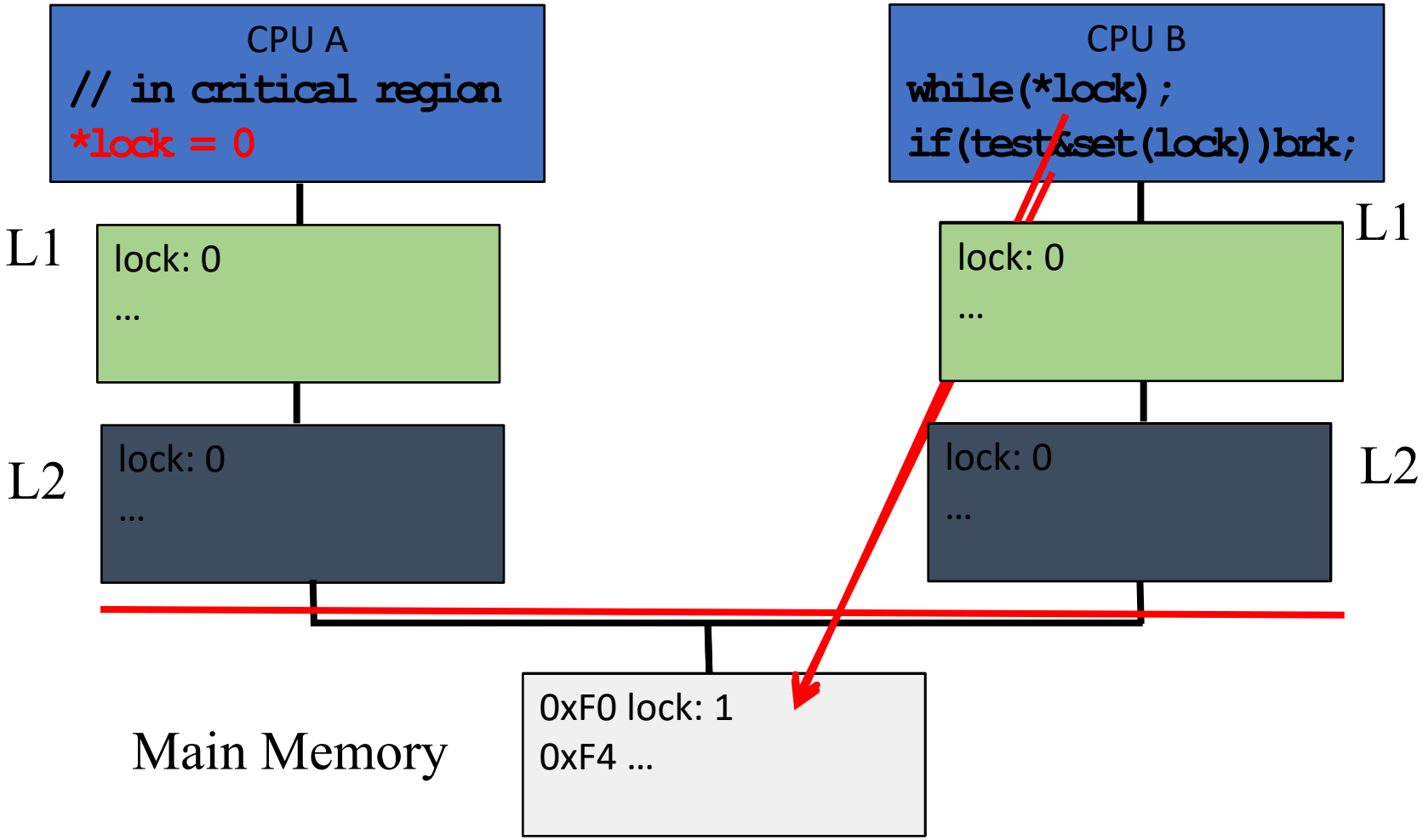
Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different cpu's contend for the same lock?



```
typedef struct { _Atomic int v; } spinlock_t;   // 0 = unlocked, 1 = locked

static inline void lock(spinlock_t *l) {
    // Fast path: test first to avoid locked bus cycles when already taken
    for (;;) {
        // atomic_exchange implemented as an x86 xchg
        if (!atomic_load_explicit(&l->v, memory_order_relaxed) &&
            !atomic_exchange_explicit(&l->v, 1, memory_order_acquire)) {
            return; // acquired
        }
        // Polite spin while the lock looks held, reducing contention
        while (atomic_load_explicit(&l->v, memory_order_relaxed)) _mm_pause();
    }
}
static inline void unlock(spinlock_t *l) {
    // On x86, a plain store has release semantics; use a release
    // store for the compiler too
    atomic_store_explicit(&l->v, 0, memory_order_release);
}
```

```
; int lock(spinlock_t *l)  ; returns with lock held
; l->v is a 32-bit int: 0 = unlocked, 1 = locked

lock:
.spin_check:
    mov     eax, dword ptr [rdi]      ; read l->v (relaxed)
    test    eax, eax
    jne     .spin_wait                ; if held, go spin politely

.try_xchg:
    mov     eax, 1
    xchg    eax, dword ptr [rdi]      ; atomic RMW: eax <- old, [l->v] <- 1
    test    eax, eax
    jne     .spin_wait                ; someone else had it; keep spinning
    ret                               ; acquired

.spin_wait:
    pause                             ; polite spin (reduces contention)
.wait_loop:
    mov     eax, dword ptr [rdi]
    test    eax, eax
    jne     .wait_loop                ; wait while it still looks held
    jmp     .try_xchg                 ; try to acquire again
```

```
; void unlock(spinlock_t *l)

unlock:
    mov     dword ptr [rdi], 0        ; x86 TSO gives release on stores
    ret
```