# Operating System Design and Implementation

## Lecture 18: Multi-core locks

Tsung Tai Yeh

Tuesday: 3:30 – 5:20 pm
Classroom: ED-302

# Acknowledgements and Disclaimer

- Slides was developed in the reference with
  MIT 6.828 Operating system engineering class, 2018
  MIT 6.004 Operating system, 2018
  Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC
  Onur Mutlu, Computer architecture, ece 447, Carnegie Mellon University
- CSE 506, operating system, 2016,
  https://www.cs.unc.edu/~porter/courses/cse506/s16/slides/sync.pdf

# Outline

- Locks in Multi-core
- Cache coherence protocol (MSI model)
- MCS lock
- Scalable lock – read-copy-update (RCU)

# Motivation

- Modern CPUs are multicore
- Applications rely heavily on kernel for networking, filesystem, etc.
- If kernel can’t scale across many cores, applications that rely on it won’t scale either

# Problem is sharing

- OS maintains many data structures
  - Process table, file descriptor table, buffer cache, scheduler queues, etc.
  - **They depend on locks to maintain invariants**
  - Applications may **contend** on locks, **limiting scalability**
- OS evolution
  - Early kernels depended on a single “**big lock**” to protect kernel data
  - Later, kernels transitioned to **fine-grained locking**
  - Now, many **lock-free approaches** are used too

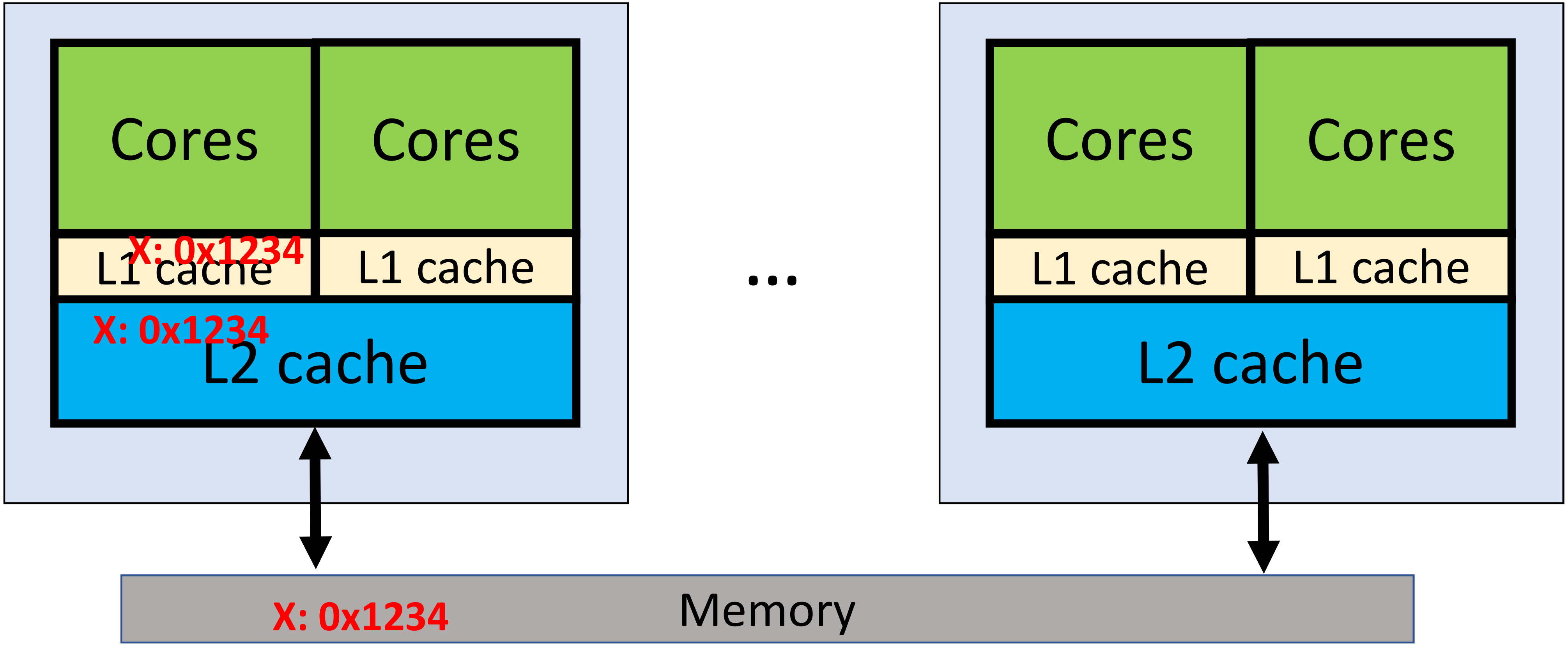
# Lock problems in Multi-core Processors

- Locks prevent us from harnessing multi-core to improve performance (why ?)
  - Non-scalable lock (what ?)
  - Locking bottleneck caused by interaction with multi-core caching (why ?)
- **Cache consistency**
  - Order of reads and writes among **MANY** memory locations
- **Cache coherence**
  - Data movement caused by reads and writes for a **SINGLE** memory location

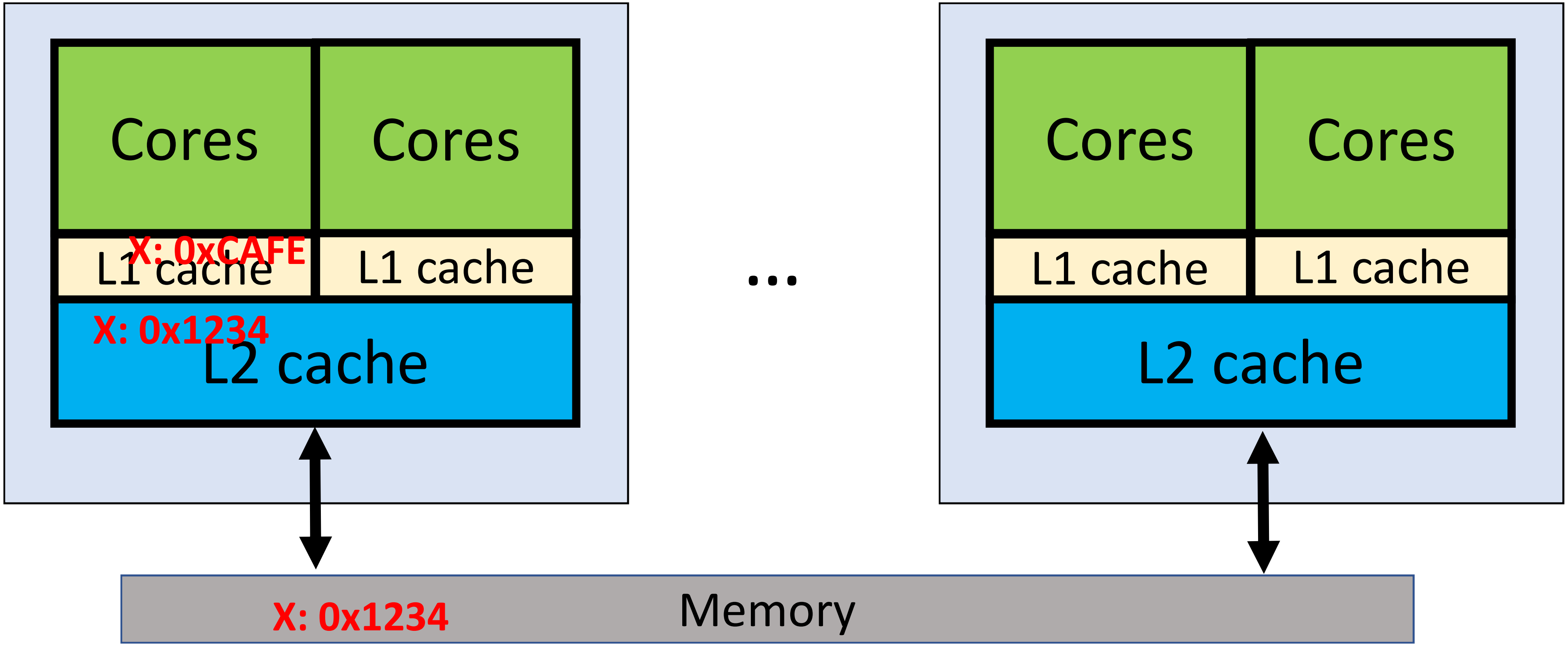
# Write-back cache

- Problems of caches in the presence of multiple processors (or cores) ?
  - The cache is divided into fixed-sized chunk called “**cache-lines**”
  - **Two cores might access the same data**
  - Therefore, we might have two copies of the same cache line present in two different caches
  - Core 1 might wish to access data that is dirty in core2’s cache
  - → **cores/processors can see the incorrect data if hardware does not compromise the needs from cores**

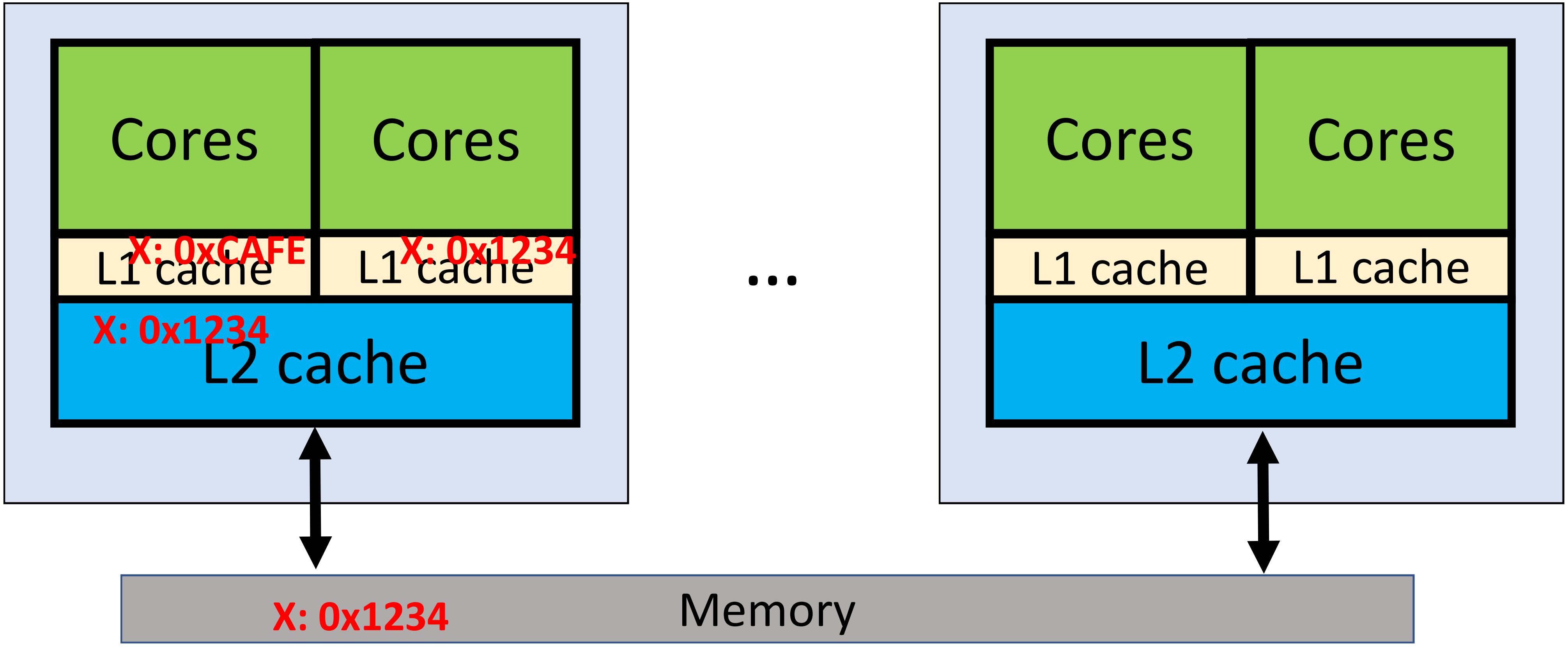
# Core 1 reads X



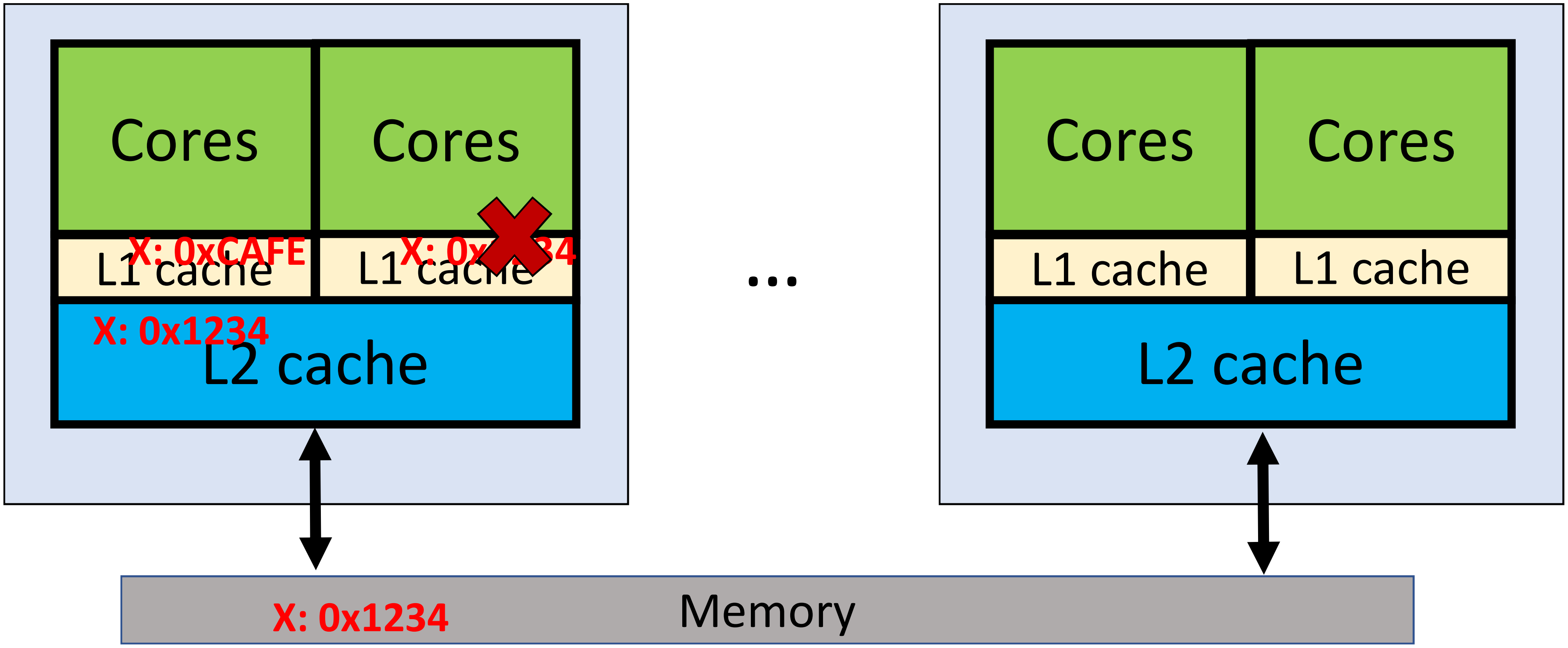
# Core 1 reads X, Core 1 writes X



# Core 2 reads X



# Core 2 reads X



Cores

Cores

X: 0xCAFE

L1 cache

~~X: 0x 34~~

L1 cache

X: 0x1234

L2 cache

…

Cores

Cores

L1 cache

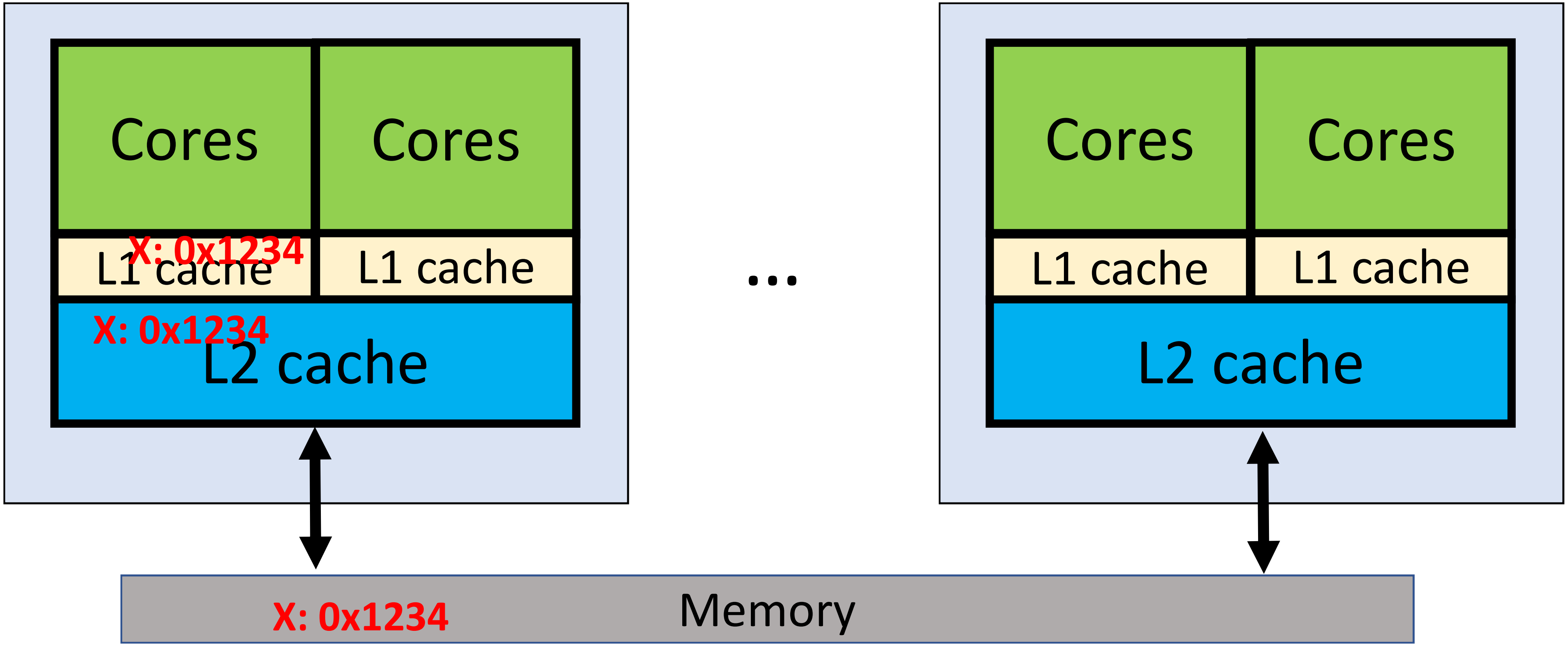
L1 cache

L2 cache

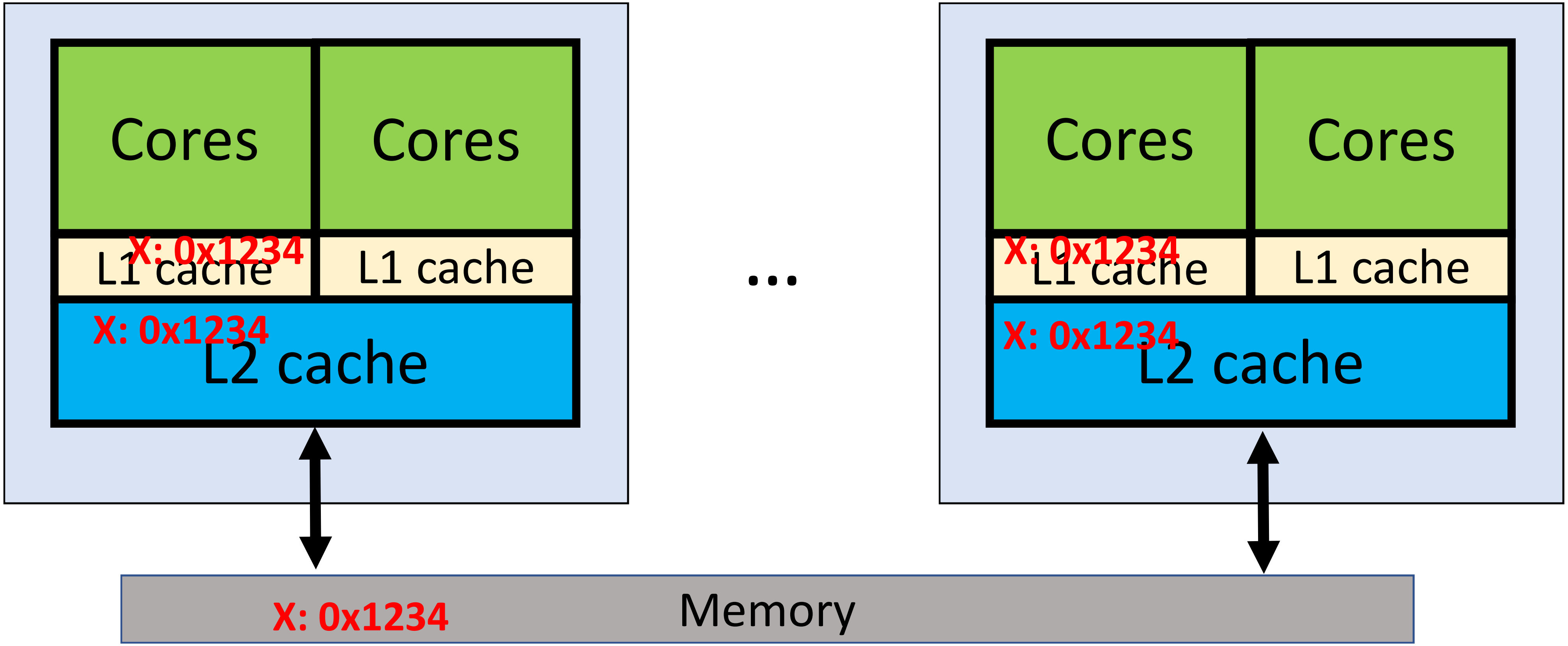
X: 0x1234

Memory

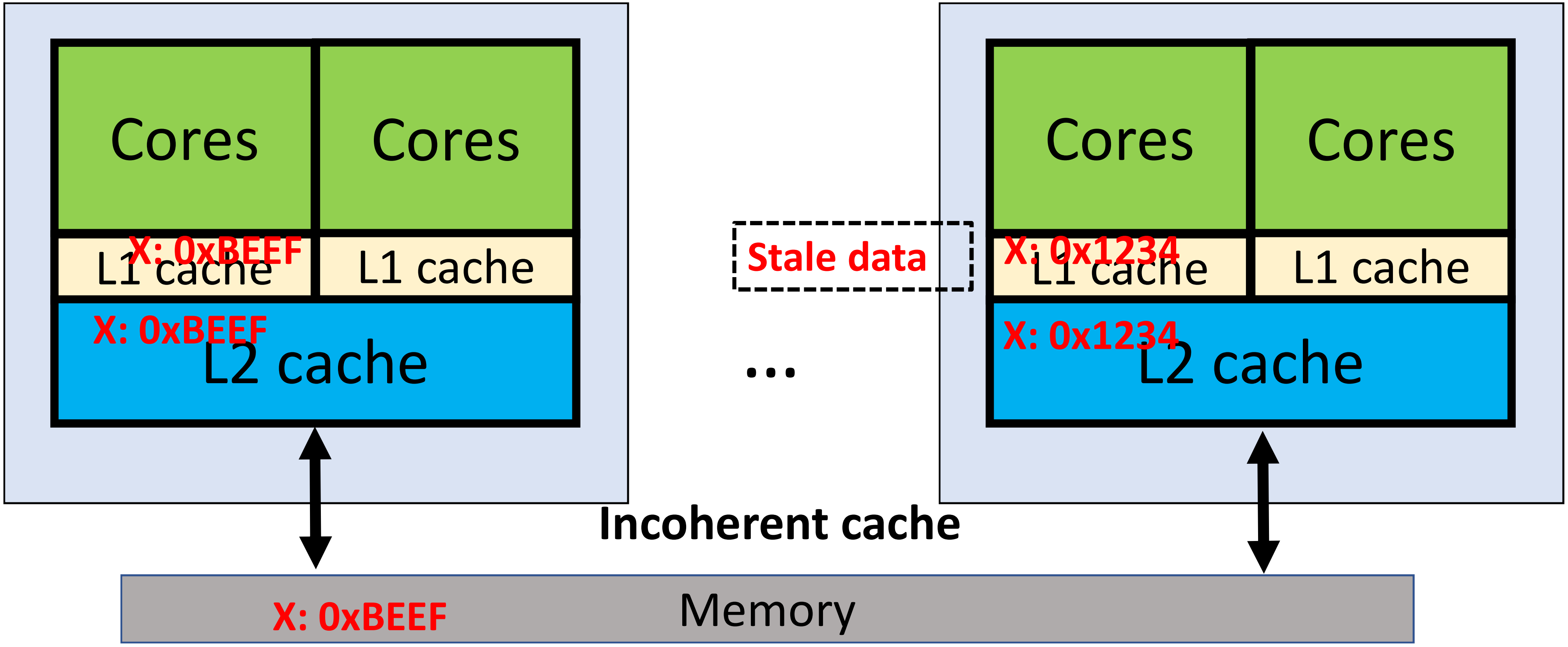
# Processor 1/ Core 1 reads X



# Processor 2/ Core 1 reads X



# Processor 1/ Core 1 writes X

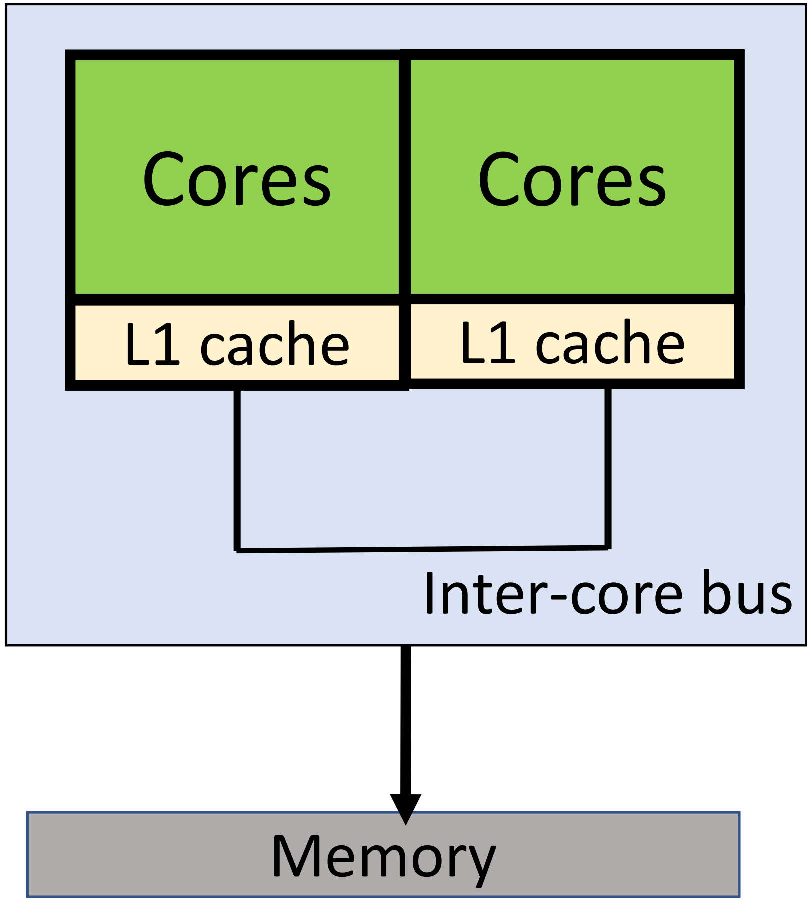


# Cache coherence

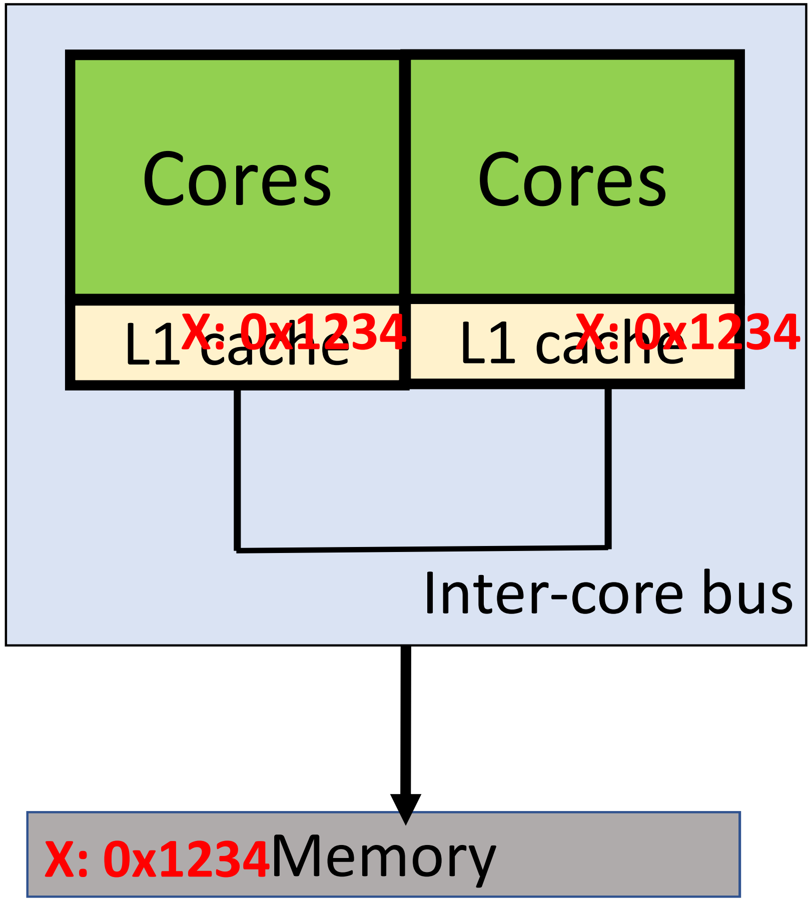
- **Cache coherence protocol**
  - Ensure that loads from all cores will return the value of the latest store to that memory location
  - Use cache metadata to track the state of cache data
- Two major approaches
  - **Snoopy caches**
  - **Directory based coherence**

# Snoopy cache coherence

- Bus-based "**snooping**"
  - **All cores continuously snoop (monitor) the bus connecting their cores**
  - If a cache see some messages across the bus
    - A cache can update the current data
    - Or send the message across the bus then other processor can pay attention to
- **Invalidation**
  - If a core writes to a data item, all copies of this data item in other caches are invalidated

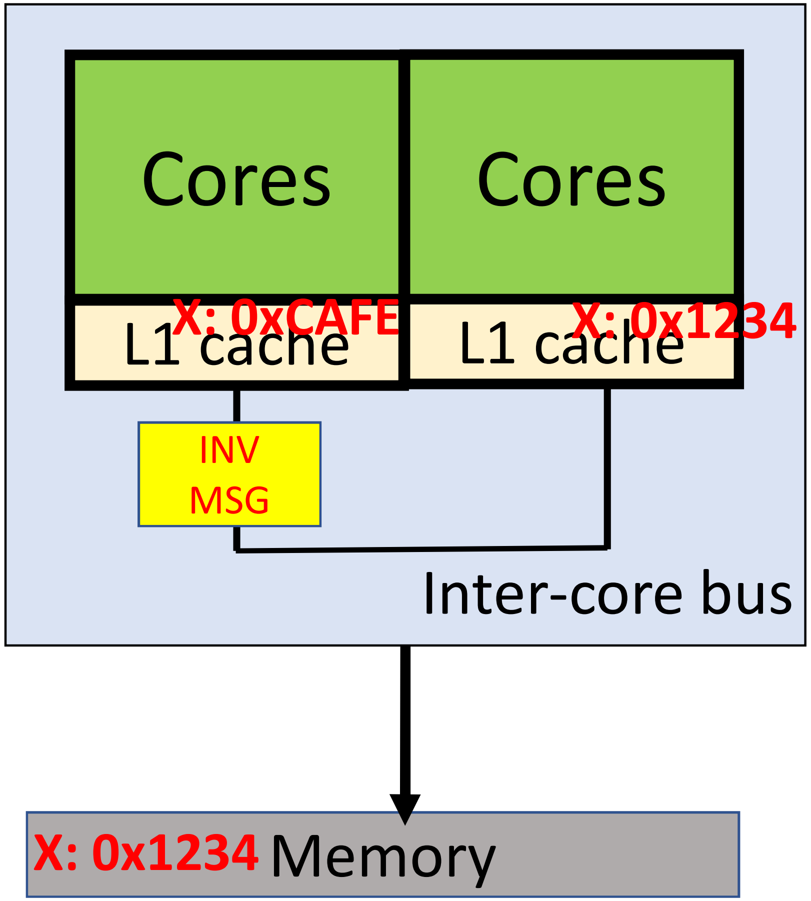


# Core 1 read X, core 2 reads X



# What happens this time if core 1 writes X ?

- Core 1 cannot directly update the its value
- Core 1 has to send out invalidation message to core 2
- Core 2 sees the invalidation message
  - Invalidate its cache line
  - Evict that invalided cache line
- What metadata should we need to support this ?



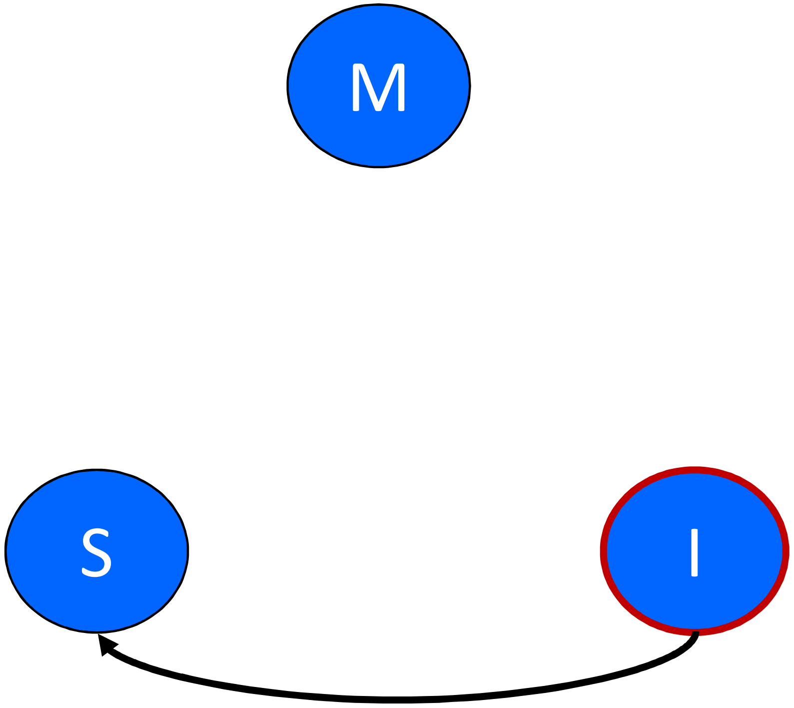
# MSI cache coherence model

- **Cache operations**
  - Change state
  - Send invalidate requests
  - Request cache lines in a particular state (fetch)
- **A minimal set of states (MSI model)**
  - Assume a writeback cache
  - **M**: cache line is modified (i.e., dirty)
  - **S**: cache line is shared; appears in multiple caches -> allows every core to keep a copy
  - **I**: cache line invalid (i.e., contains invalid data)

# MSI protocol (1)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

- **Local read**
  - A processor wants to read
  - Get the data from the data source
  - The data is not dirty -> don't go to M state
  - Go to the S state
  - The data is shared with other processors/cores

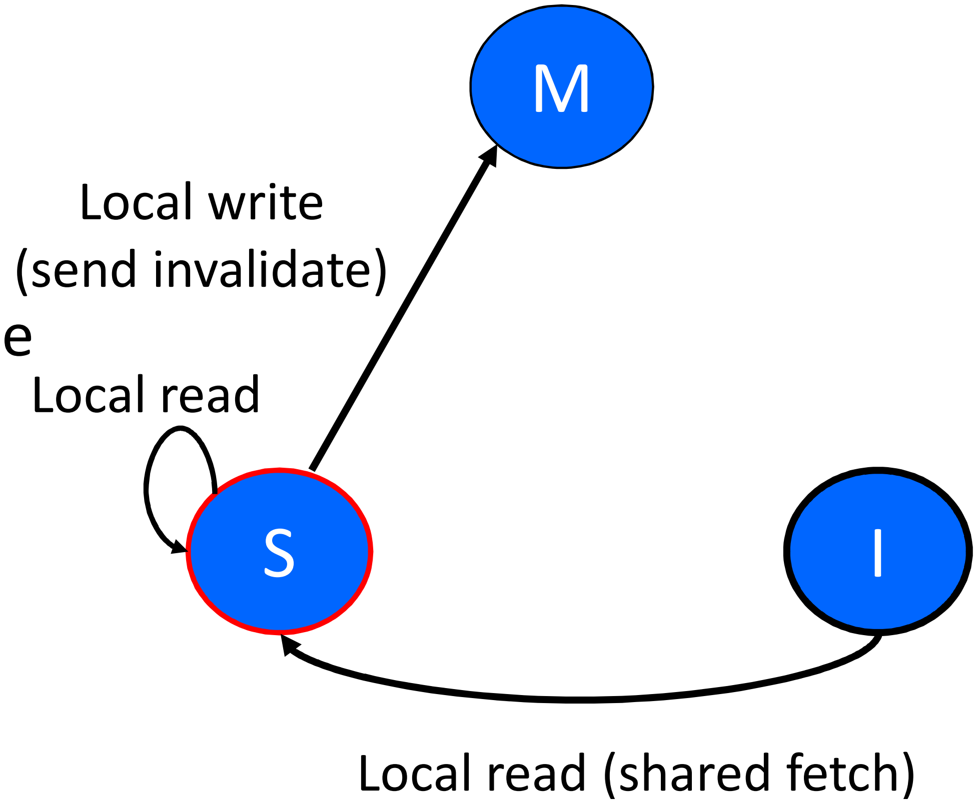


Local read (shared fetch)

# MSI protocol (2)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

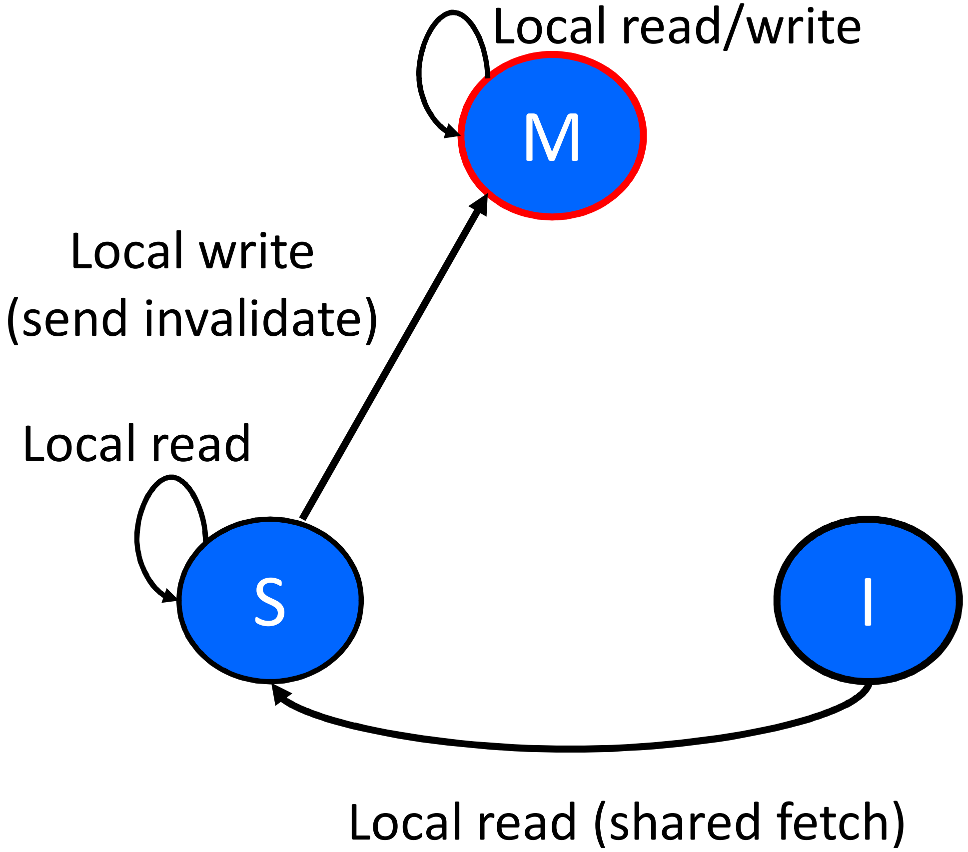
- **Local write**
  - Since the data in S state
  - Get the write hit
  - Go to the M state
  - Send an invalidated message to notify other processors the data is stale
  - If other processor has this data -> invalidate it



# MSI protocol (3)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

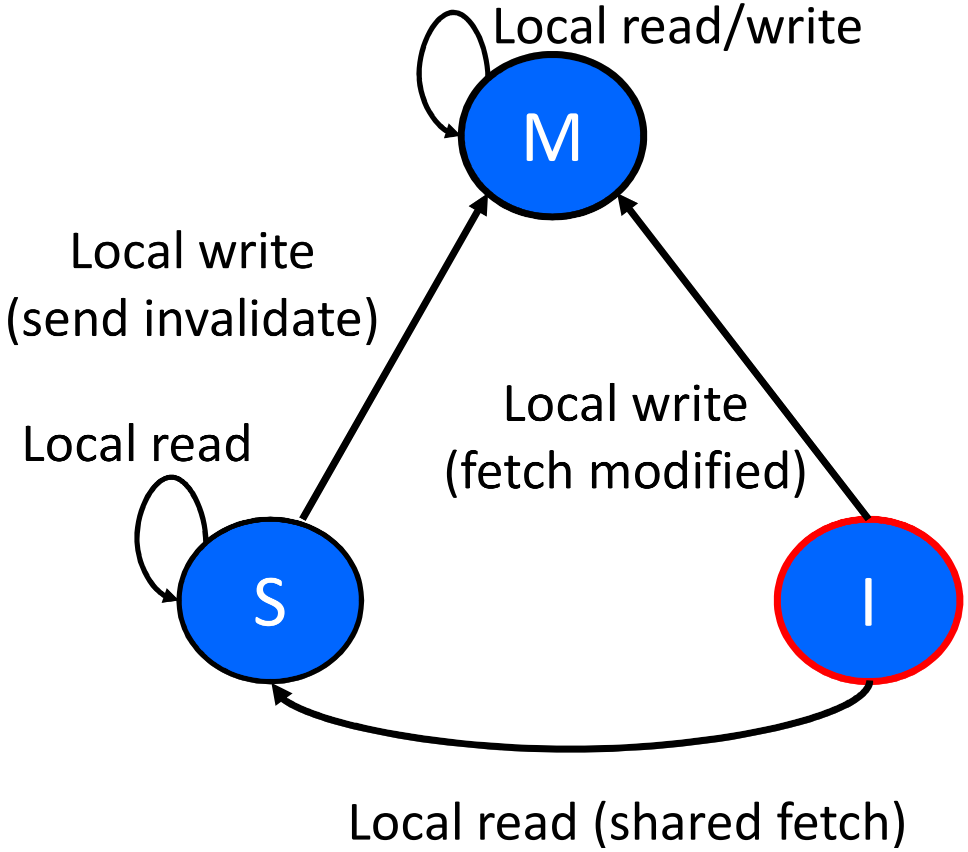
- **Local read/write**
  - Current we are in the M state
  - Local read/write happens
  - Keep sitting in the M state



# MSI protocol (4)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

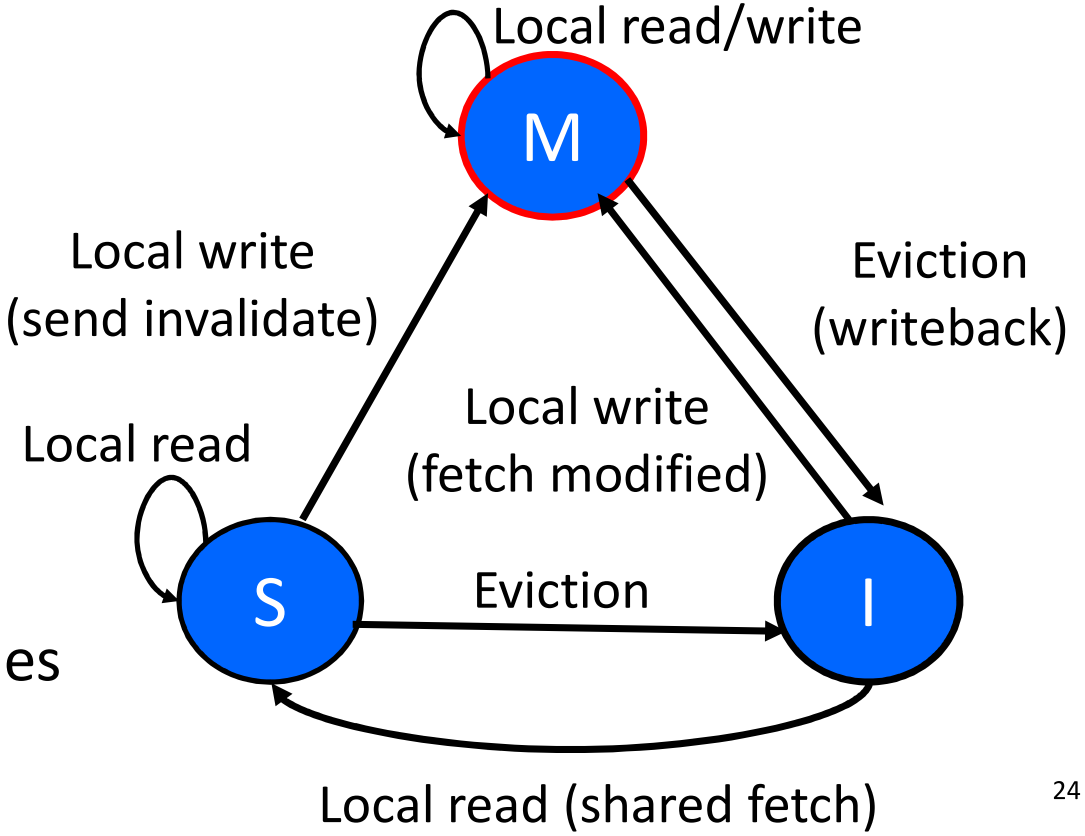
- **Local write miss**
  - Current we are in the I state
  - We get the write happens
  - It is write, so it is better to go to M state
  - Go to get the data from the data source
  - Notify other processors



# MSI protocol (5)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

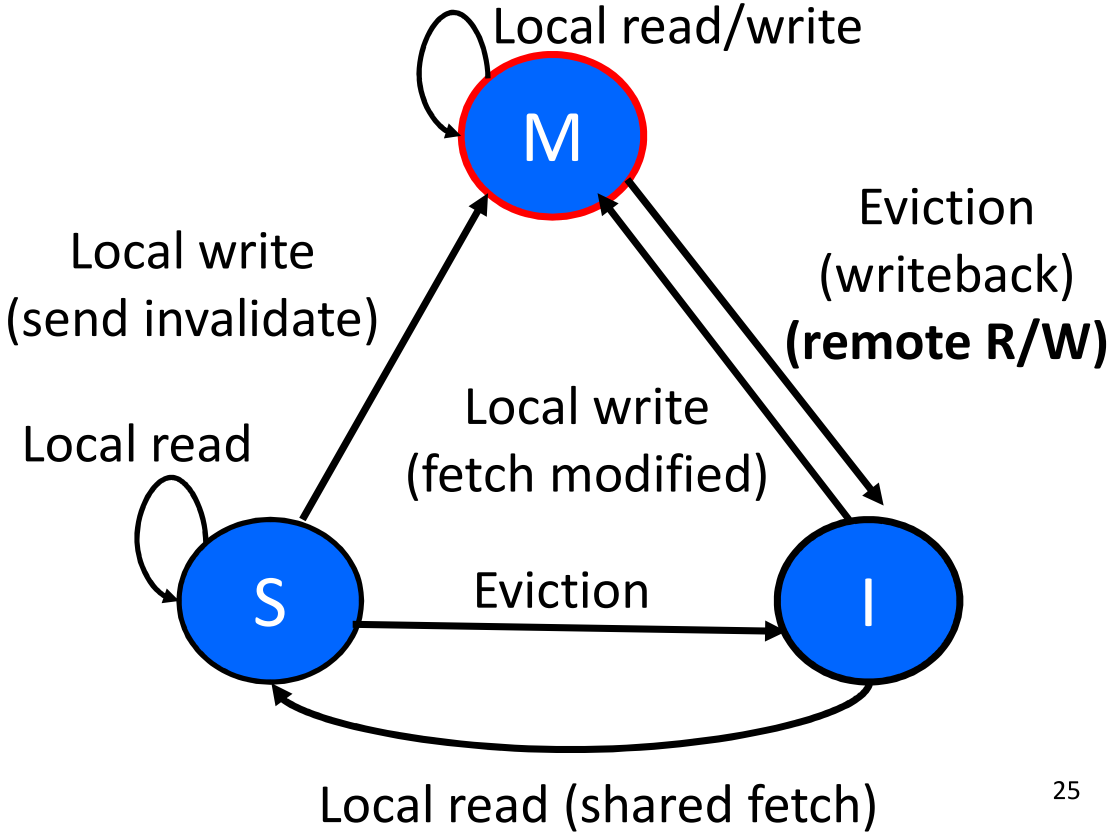
- **Eviction**
  - Current we are in the M state
  - Have dirty data in the cache that is needed to be evicted
  - Write the dirty back before evicting
  - Eviction also happens in S state -> no notify other ones
  - Switch to I state



# MSI protocol (6)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

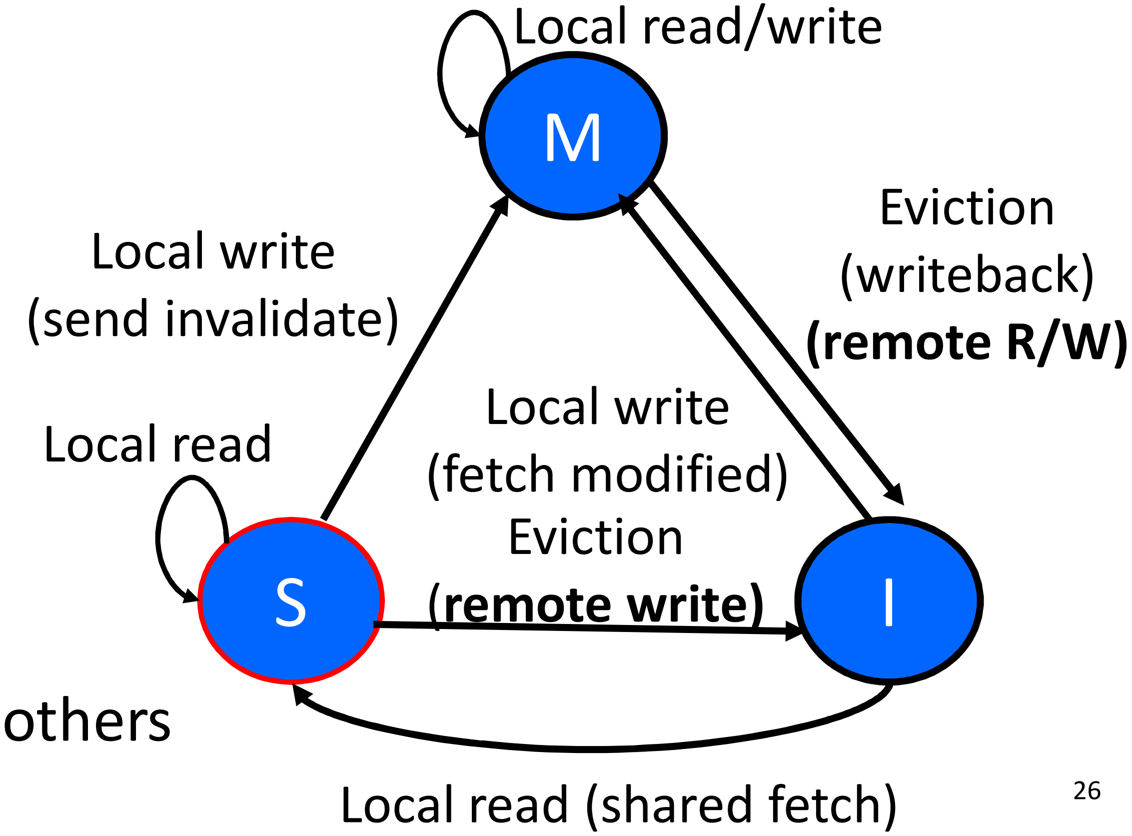
- **Invalidate (from remote)**
  - Current we are in the M state
  - We see someone on the remote core is either trying to read or write items
  - I have the dirty cache line and see other is writing
  - Evict the cache line and go to I state



# MSI protocol (7)

**M**: cache line is modified (i.e., dirty)
**S**: cache line is shared; appears in multiple caches
**I**: cache line invalid (i.e., contains invalid data)

- **Invalidate (from remote)**
  - Current we are in the S state
  - We see someone on the remote core is either trying to read or write items
  - It is ok to see others are reading data
  - However, we need to evict the staled cache line if we see others are writing -> go to I state



# Why locks if we have cache coherence ?

- **Cache coherence**
  - Ensures that cores read fresh data
- **Locks**
  - Avoid lost updates in read-modify-write cycles
  - Prevent anyone from seeing partially updated data structures
- **How does hardware implement locks ?**
  - Get the line in **M** state
  - Defer coherence messages
  - Do all the steps (read and write)
  - Resume handling messages

# Locking performance criteria

- Assume N cores are waiting for a lock
- How long does it take to hand off from previous to next holder ?
- Bottleneck is usually the interconnect
  - The measure cost is in terms of # of messages
- What can we hope for ?
  - If N cores waiting, get through them all in O(N) time
  - Each handoff takes O(1) time; doesn’t increase with N

# Test & set spinlocks

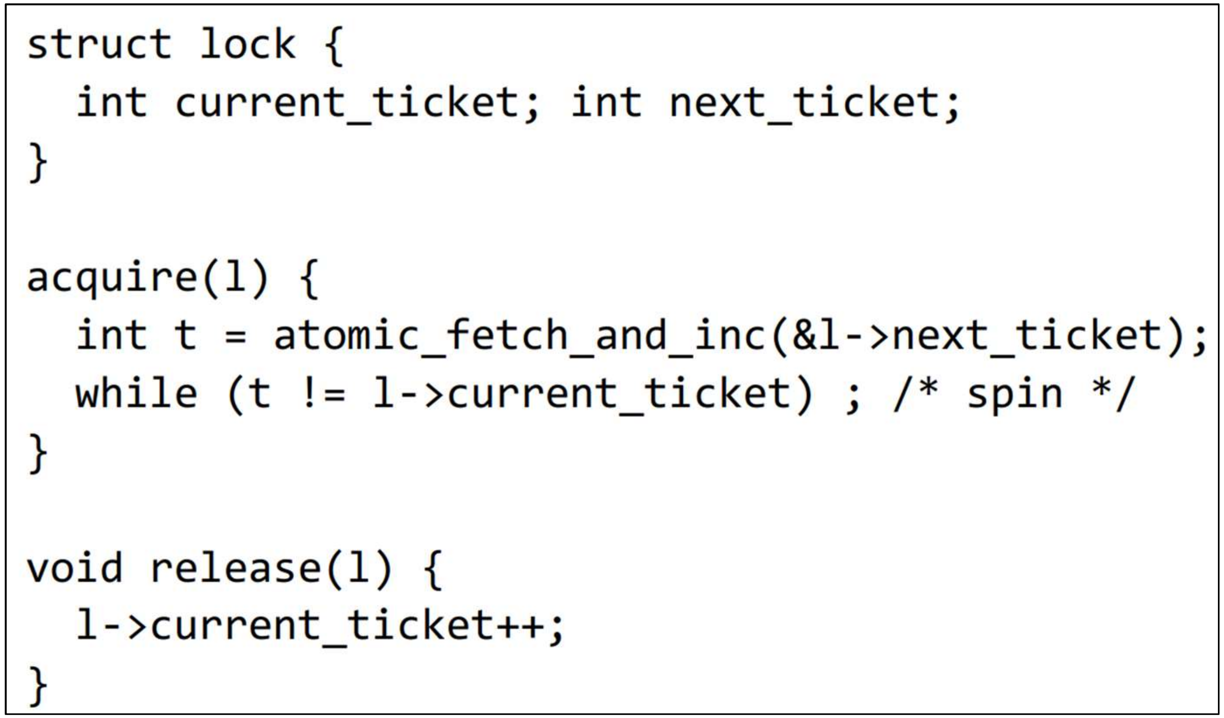
```
struct lock { int locked; };

acquire(l){
  while(1){
    if(!xchg(&l->locked, 1))
      break;
  }
}

Release(l){
  l->locked = 0;
}
```

# Test & set spinlocks

- Spinning cores repeatedly execute atomic exchange
- Is this a problem ?
  - Yes !
  - It’s okay if waiting cores waste their own time
  - Bad if waiting cores slow lock holder
- Time for critical section and release
  - Holder must wait in line for access to bus
  - Halder’s handoff takes O(N) time
- O(N) handoff means all N cores take O($N^2$)

# Ticket locks (Linux)

- Goal of ticket locks
  - Read-only spinning rather than repeated atomic instructions
  - Fairness -> waiter order preserved
- **Key idea**
  - **Assign numbers, wake up one waiter at a time**



```
struct lock {
  int current_ticket; int next_ticket;
}

acquire(l) {
  int t = atomic_fetch_and_inc(&l->next_ticket);
  while (t != l->current_ticket) ; /* spin */
}

void release(l) {
  l->current_ticket++;
}
```

# Ticket lock time analysis

- **Atomic increment**
  - O(1) broadcast message
  - Just once, not repeated
- Then **read-only spin, no cost until next release**
- **What about release ?**
  - Invalidate message sent to all cores
  - Then O(N) find messages, as they re-read
  - Still O(N) handoff work
  - But fairness and less bus traffic while spinning

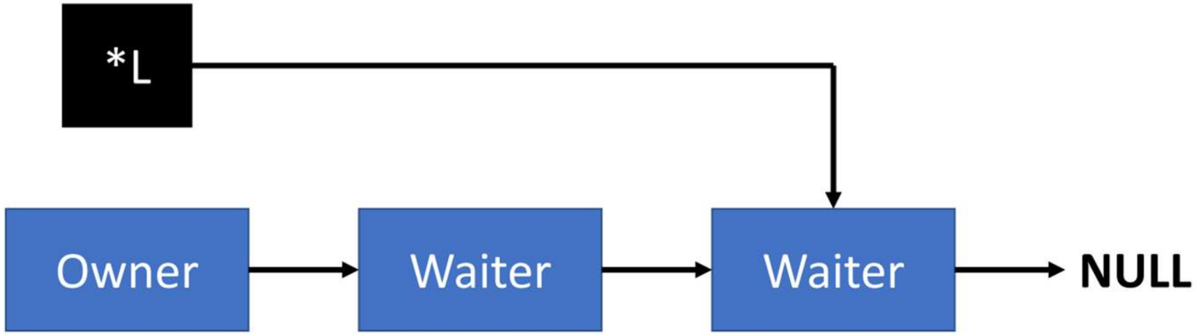
# How to make locks be scale ?

- TAS and Ticket lock are “non-scalable” locks
  - Cost of handoff scales with the number of waiters
- **Goal**
  - O(1) message release time
  - Wake just one core at a time
- **Idea**
  - Have each core spin on a different cache-line

# MCS locks

- Each CPU has a qnode structure in its local memory (queue spin lock)
- A lock is a qnode pointer to the tail of the list
- Each CPU only spin its own "locked" value

```
typedef struct qnode {
        struct qnode *next;
        bool locked;
} qnode;
```



*L

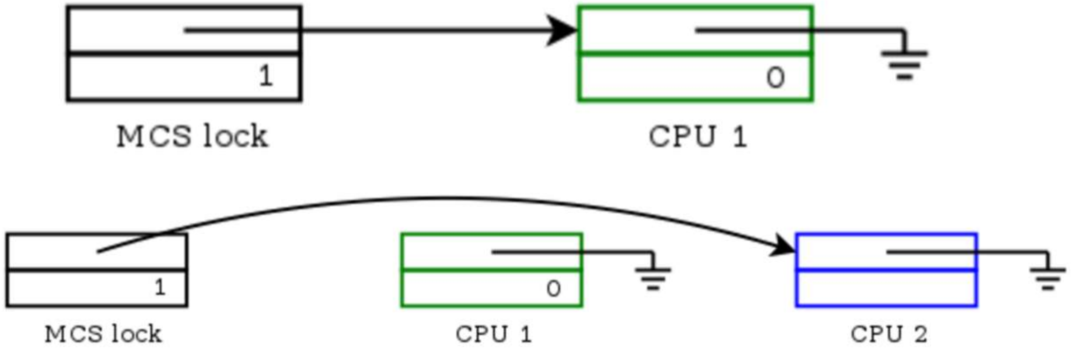
Owner

Waiter

Waiter

**NULL**

# Acquiring MCS locks

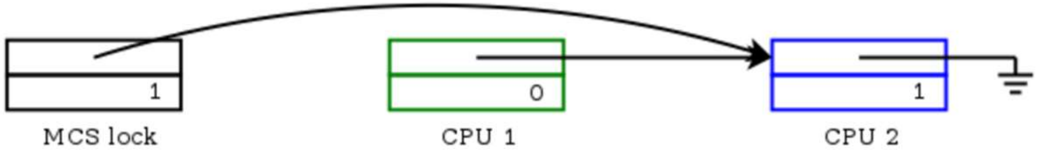


- CPU 1 creates a qnode struct
    - Main lock executes atomic exchange -> store the object locked value of next pointer (CPU 1) to the lock of its own struct
    - When CPU2 is trying to get lock
        - CPU2 found main lock points to CPU1
        - The locked value of CPU 2 becomes 1
        - CPU2 is locked
    - Every CPU is spinning with locked value of its own struct

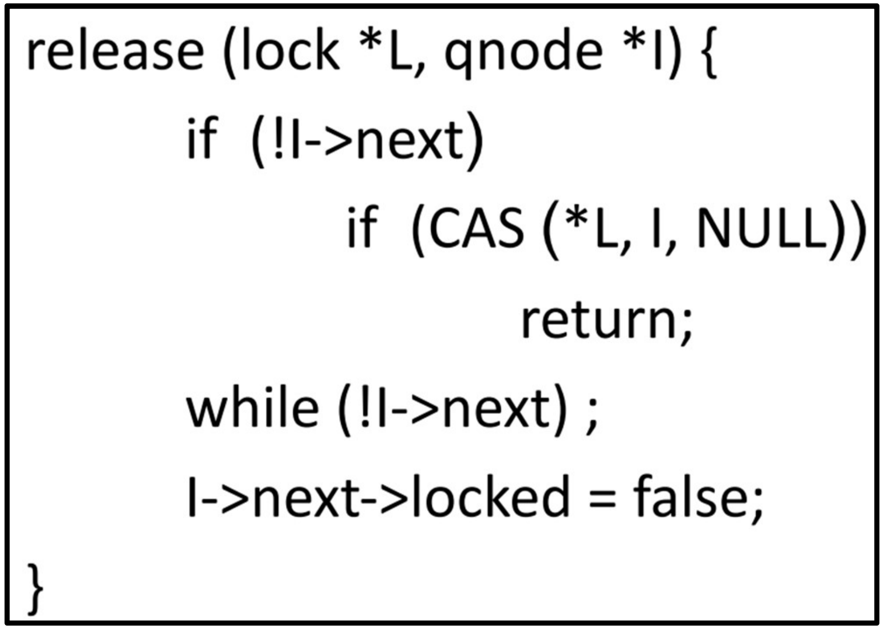
```
acquire (qnode *L, qnode *I) {
        I->next = NULL;
        qnode *predecessor = I;
        XCHG (*L, predecessor);
        if (predecessor != NULL) {
                I->locked = true;
                predecessor->next = I;
                while (I->locked) ;
        }
}
```

https://lwn.net/Articles/590243/

# Releasing MCS locks



- If next of main lock points to NULL
  - No one uses lock – lock released
- CPU1’s next points to CPU2
  - Main lock always points to the last item of the entire queue
  - Thus, we can know who is the next one to get the lock
- After CPU1 completes its work
  - Atomic exchange with main lock
  - Point itself to NULL



```
release (lock *L, qnode *I) {
        if  (!I->next)
                if  (CAS (*L, I, NULL))
                        return;
        while (!I->next) ;
        I->next->locked = false;
}
```

https://lwn.net/Articles/590243/

# Read-heavy data structures

- The data read is much more often than modified in kernels
  - Network tables: routing, ARP
  - File descriptor arrays, most types of system call state
  - **Read-copy-update (RCU)** optimizes for these use cases
  - Over 10, 000 RCU API uses in the Linux kernel
- Goal
  - Concurrent reads even during updates
  - Low space overhead
  - Low execution overhead

# Plan #1: spin locks

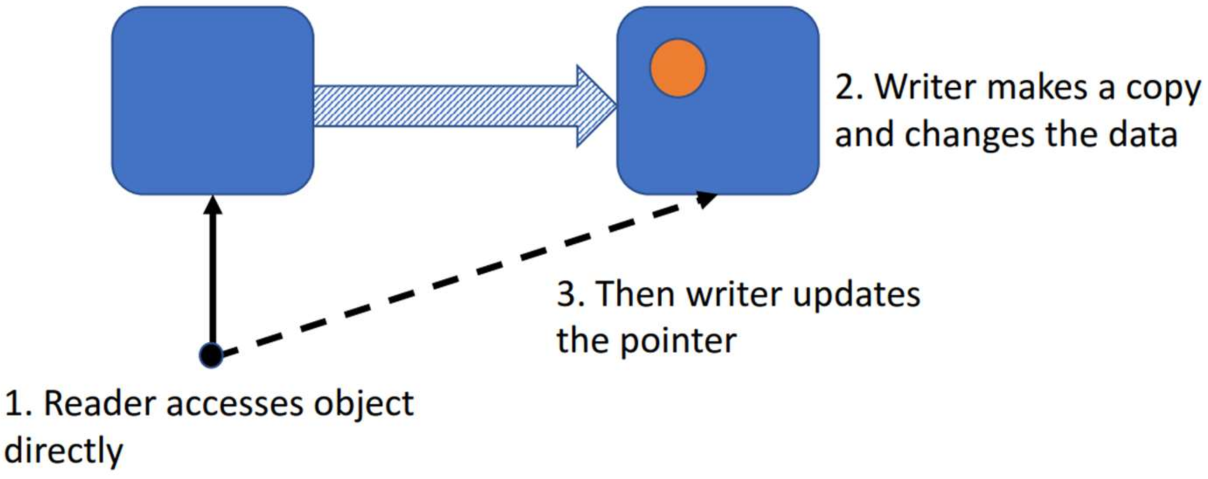
- **Problem**
  - Serializes all critical sections
  - Read-only critical sections would have to wait for other read-only sections to finish
- **Idea**
  - Allow parallel readers but still serialize writers

# Plan #2: Read-write locks

- A modification to spin locks that allow parallel reads
- Every reader uses CMPXCHG instruction
  - S->M cache coherence state transition
  - Find + invalidate messages to contend read_lock() and read_unlock ()
- If writer holds lock, readers must spin and wait
  - Violates goal of concurrent read, even during updates

# Plan #3 Read-copy-update (RCU)

- Data is accessible through “root pointer”
  - Could be index into an array
  - Must be atomic
- **Reader**
  - Acquire “root pointer” atomically, access data
- **Writer**
  - Read current data, copy to new data, update new data, and publish it



2. Writer makes a copy and changes the data

3. Then writer updates the pointer

1. Reader accesses object directly

https://pdos.csail.mit.edu/6.828/2018/lec/l-rcu.pdf

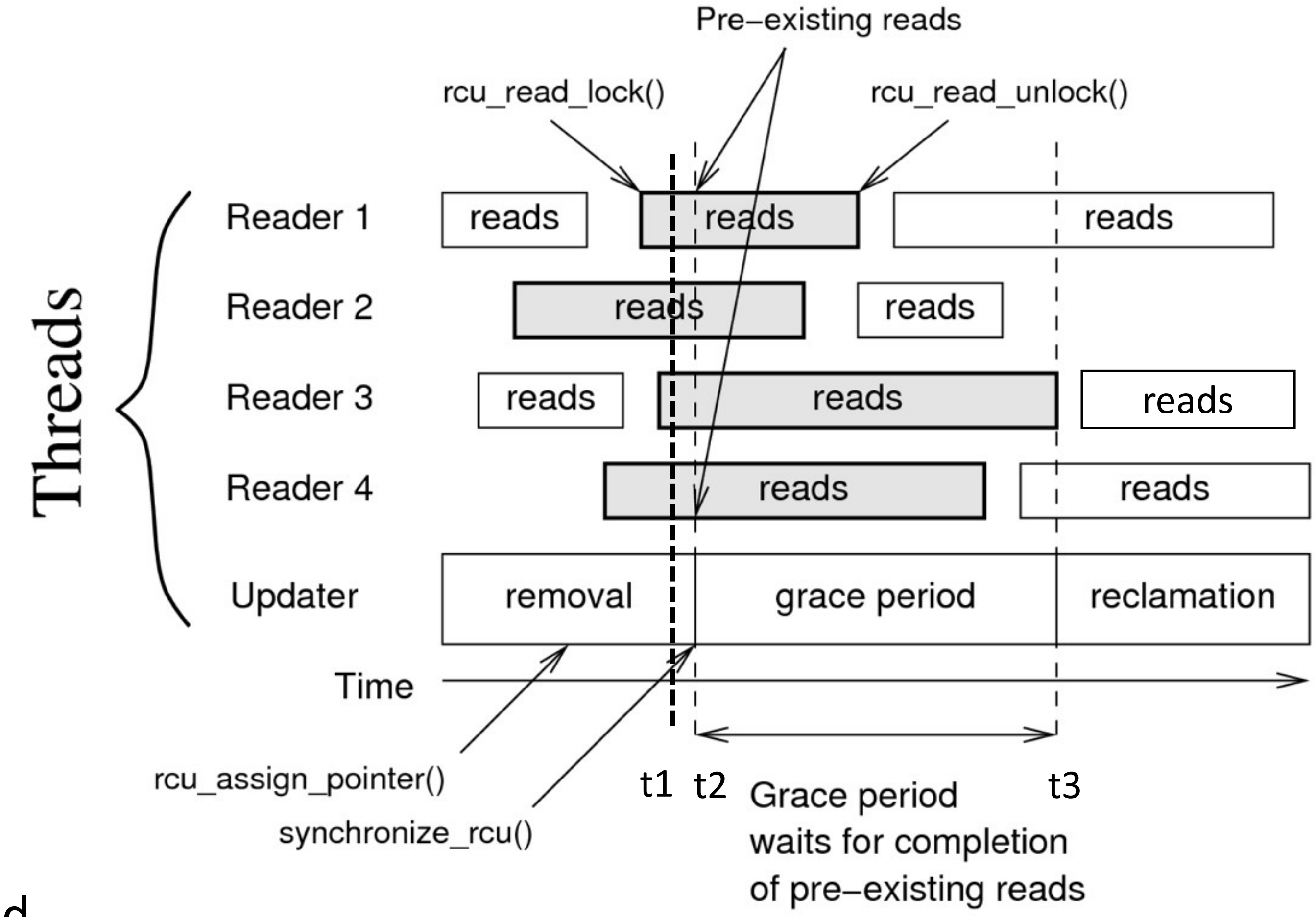
- Some readers see old data, other readers see new data

# When to free old objects ?

- At any given moment, **readers could be accessing the latest copy or older copies of an object**
  - Safely free objects when they are no longer “reachable”
- Usually only **one pointer to an RCU object**
  - Can’t be copied, stored on the stack, or in registers (except inside critical sections)
- Need a “**quiescent period**”, after which it’s safe to free
  - Wait until all cores have passed through a context switch
  - Pointer can only be dereferenced inside a critical section
  - **Read critical sections disable preemption (why?)**

# Quiescent (grace) period

- Reader (1-4) reads the pointer fp before t1
- At t2, updater calls synchronize_rcu (), but reader (1-4) are in CS
- **Grace period**
  - Wait for the complete of all readers that are in the CS
  - Readers refers the new version fp and old data of all readers can be freed



https://hackmd.io/@sysprog/linux-rcu

# Example program using RCU

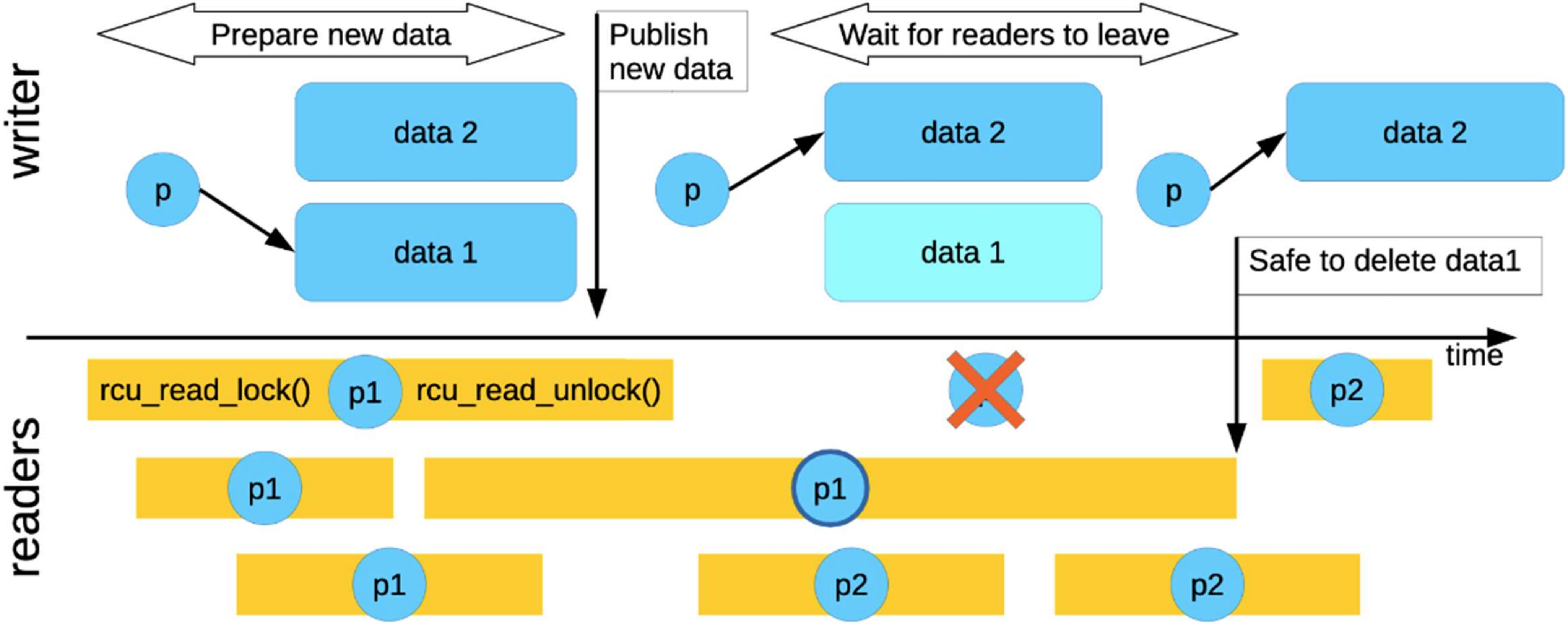
- **rcu_read_lock () and rcu_read_unlock ()**
  - Used to indicate the start and the end of grace period

```
void foo_read () {
    rcu_read_lock ();
    foo *fp = global_foo;
    if (fp)
            do_something (fp->a, fp->b, fp-> c);
    rcu_read_unlock ();
}
```

```
void foo_update () {
    spin_lock (&foo_mutex);
    foo *old_fp = global_foo;
    global_foo = new_fp;
    spin_unlock (&foo_mutex);
    synchronize_rcu ();
    kfree (old_fp);
}
```

https://github.com/CppCon/CppCon2017/blob/master/Presentations/Read%2C%20Copy%2C%20Update...%20Then%20What/Read%2C%20Copy%2C%20Update...%20Then%20What%20-%20Fedor%20Pikus%20-%20CppCon%202017.pdf

# Publish-subscribe mechanism



**Publish:** The writer update the reference of pointer, publish new data

**Subscribe**: The reader safely deletes old data after the grace period

# RCU memory reclamation

- **RCU uses cooperative protocol**
  - Track when it is safe to reclaim memory (when no reader can access it)
- **Readers MUST** follow these steps to access shared data
  - 1) Call rcu_read_lock () to request access
  - 2) Get the root pointer
  - 3) Call rcu_read_unlock () to announce the end of access
- Reader may access shared data only between the calls
  - rcu_read_lock () and rcu_read_unlock ()

# RCU memory reclamation

- **Writer MUST** follow these steps to modify shared data
  - 1) Make old shared data inaccessible from the root
  - 2) Call synchronize_rcu () to wait for all readers who called rcu_read_lock () before step 1 to call rcu_read_unlock ()
  - 3) Delete old data and reclaim the memory
- We don’t need to
  - Wait for all readers to exit critical section
  - Only wait for those who acquire the old root pointer

# Disable preemption during RCU read critical sections

- If didn’t disable preemption during RCU read critical sections
  - Need to wait for all cores to context switch
  - Wouldn’t be an effective quiescent period
- A task could still hold a pointer to an RCU object while it is preempted
  - Hard to decide when its safe to free
  - Unless we wait until all tasks are killed
  - Need to define a read critical section such that references to RCU objects cannot persist outside the section

# How to synchronize writes ?

- **Against other writers**
  - Allow only one writer
  - Just use normal synchronization like locks
- **Against readers (memory order matters)**
  - Writers must fully finish writes to new object before updating pointer
  - Readers must not reorder reads such that contents of an object are read before its pointer
  - rcu_dereference() and rcu_assign_pointer() automatically insert the appropriate compiler and memory barriers

# RCU APIs

- **rcu_read_lock():** Begin an RCU critical section
- **rcu_read_unlock()**: End of an RCU critical section
- **synchronize_rcu():** wait for existing RCU critical sections to complete
- **call_rcu (callback, argument):** call the callback when existing RCU critical sections complete
- **rcu_dereference (pointer):** Signal the intent to dereference a pointer in an RCU critical section
- **rcu_dereference_protected(pointer, check):** signals the intent to dereference a pointer outside of an RCU critical section
- **rcu_assign_pointer(pointer_addr, pointer):** Assign a value to a pointer that is read in RCU critical sections

# Example RCU usage (reader)

```
float get_cost(void) {
  item_t *p;
  float cost;
  rcu_read_lock();
  p = rcu_dereference(item); // read
  cost = p->price – p->discount;
  rcu_read_unlock();
  return cost;
}
```

# Example RCU usage (writer)

```
void set_cost(float price, float discount) {
  item_t *oldp, *newp;
  spin_lock(&item_lock);
  oldp = rcu_dereference_protected(item, spin_locked(&item_lock));
  newp = kmalloc(sizeof(*newp));
  *newp = *oldp; // copy
  newp->price = price;
  newp->discount = discount;
  rcu_assign_pointer(item, newp); // update
  spin_unlock(&item_lock);
  rcu_synchronize();
  kfree(oldp); // free
}
```

# Does RCU achieve its goals ?

- Goal: concurrent reads even during updates ?
  - Yes ! Reads are never stalled by updates
- Goal: low space overhead ?
  - Yes ! A RCU pointer is the same size as an ordinary pointer
  - No extra synchronization data is required
  - However, objects can’t be freed until quiescent period has passed. Forcing this to happen incurs overhead

# Does RCU achieve its goals ?

- Goal: low execution overhead ?
  - For readers, RCU has practically no execution overhead
  - For writers, a slight overhead due to allocation, free, and copying
  - Fine-grained locking can help to make updates concurrent

# Summary

- The performance of locks should be scalable in multi-cores
- Multi-core caching exhibits the bottleneck of performance scalability of locks
- MCS lock – queue spin lock
- RCU enables zero-cost read-only access
  - Very useful for read-mostly data (extremely common in kernels)